THE UNIVERSITY OF AUCKLAND

COURSEBOOK FOR

COMPSCI 220
Algorithms and Data Structures

2020

Jonathan Klawitter David Welch Mark C. Wilson

This is the official coursebook for COMPSCI 220. The lecture schedule will follow this coursebook closely. However small variations are always possible, because lecturers must react to events and student needs. It is strongly recommended to bring a paper copy of this book to each lecture. Many of the examples in the book will be worked through in lectures.

Thanks to Michael Dinneen and Georgy Gimel'farb for their past help with materials for this course. In particular, the textbook written by Dinneen, Gimel'farb and Wilson covers topics in this coursebook in more detail and is strongly recommended.

Thanks also to Simone Linz for help with proofreading.

Contents

Lecture 1

Basic mathematical background

We list here some important mathematical facts and techniques that will be used often in this course. It is very important to understand them well - ask for help if your background has substantial holes.

1.1 Sets

A ***set*** is an unordered collection of distinct objects (repeated elements are not allowed), called ***elements*** of the set. We write $a \in X$ to mean that a is an element of X. Some important sets are

- $\mathbb{N} = \{0, 1, 2, \dots\}$, the set of ***natural numbers***,
- $\mathbb{R}$, the set of ***real numbers***, and
- $\varnothing = \{\}$, the ***empty set*** having no elements.

Notation: List elements, e.g. $X = \{2, 3, 5, 7, 11\}$ or use set-builder notation $X = \{x \in \mathbb{N} : x \text{ is prime and } x < 12\}$. Note that $\{a, a, b\} = \{a, b\}$ because elements cannot be repeated in a set.

Operations on sets A and B:

- $A \cap B = \{x : x \in A \text{ and } x \in B\}$, the ***intersection*** of A and B.
- $A \cup B = \{x : x \in A \text{ or } x \in B\}$, the ***union*** of A and B.
- $|A|$ is the number of elements of A, the ***cardinality*** of A.

Example 1.1. What is the cardinality of $A = \{1, 1, 5\}$? Of $B = \{x^2 : x \in \{-1, 1, 2\}\}$?

What is $A \cap B$? What is $A \cup B$?

1.2 Functions

A ***function*** is a mapping f from a set X (the ***domain***) to a set Y (the ***codomain***) such that every $x \in X$ maps to a unique $y \in Y$. We write $f : X \to Y$. A function $f : X \to Y$ is ***one-to-one*** if whenever $x \neq x'$, we have $f(x) \neq f(x')$. It is ***onto*** if every element of Y has the form $f(x)$ for some $x \in X$. A function that is both one-to-one and onto has an ***inverse***: $f(x) = y$ if and only if $g(y) = x$.

Important functions from $\mathbb{R}$ to $\mathbb{R}$:

- Power functions $f(x) = x$, $f(x) = x^2$, $f(x) = x^3$, etc.
- Exponential functions $f(x) = 2^x$, $f(x) = (1.5)^x$, etc.
- Logarithm (inverse of exponential) is defined by $\log_a(y) = x$ if and only if $y = a^x$. Its domain is $\{x \in \mathbb{R} : x > 0\}$. We usually just write $\log_a y$, omitting parentheses, if there is no confusion about what is meant.

Other useful functions:

- ***Ceiling*** rounds up to nearest integer, e.g. $\lceil 3.7 \rceil = 4 = \lceil 4 \rceil$.
- ***Floor*** rounds down, e.g. $\lfloor 3.7 \rfloor = 3 = \lfloor 3 \rfloor$.

1.3 Basic properties of important functions

- For fixed $a > 1$ the exponential $f(x) = a^x$ is **strictly increasing** and positive, and satisfies
$$a^{x+y} = a^x a^y$$
for all $x, y \in \mathbb{R}$.
- For fixed $a > 1$ the logarithm $\log_a$ is strictly increasing and satisfies
$$\log_a(xy) = \log_a x + \log_a y$$
for all $x, y > 0$.
- We write $\ln = \log_e$ and $\lg = \log_2$. Note that $\log_a x = \log_a b \log_b x$ and $a^x = e^{x \ln a}$. If the base of the logarithm does not matter (for example when we do asymptotic analysis) we often just write $\log$.

Example 1.2. Prove that for every $n \in \mathbb{N}$ such that $n \geq 1$,

$$\lceil \lg(n+1) \rceil = 1 + \lfloor \lg n \rfloor.$$

1.4 Sums

A ***sequence*** is a function $f : \mathbb{N} \to \mathbb{R}$.

Notation: $f_0, f_1, f_2, \cdots$ where $f_i = f(i)$.

Sum notation: $f_m + f_{m+1} + \cdots + f_n = \sum_{i=m}^{n} f_i$.

Some important sums:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=m}^{n} a^i = \frac{a^{n+1} - a^m}{a-1}.$$

Example 1.3. Simplify

$$\sum_{i=1}^{2n} 2^i.$$

1.5 Proof by induction

Mathematical induction is a standard technique for proving a collection of results, one for each natural number. If we can prove for $n = 0$, and also show that

whenever the result we want is true for n, it is true for $n+1$, then we know it must be true for all $n \in \mathbb{N}$.

Example 1.4. Prove that if $a \neq 1$ then for all $n \geq 0$

$$\sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}.$$

Proof. It is true for $n = 0$: $1 = 1$. Assume it is true for a given $n \geq 0$. Then

$$\begin{aligned}\sum_{i=0}^{n+1} a^i &= a^{n+1} + \sum_{i=0}^{n} a^i = a^{n+1} + \frac{a^{n+1}-1}{a-1} \\ &= \frac{a^{n+1}(a-1) + a^{n+1} - 1}{a-1} \\ &= \frac{a^{n+2}-1}{a-1}.\end{aligned}$$

Thus the result holds for **all** $n \geq 0$, by **mathematical induction**.

Example 1.5. Consider the sequence defined by $a_0 = 1$ and $a_n = 3a_{n-1} - 4$ for $n > 0$. Prove that a_n is odd for all $n \in \mathbb{N}$.

1.6 Binary trees

A ***binary tree*** is an object that is either empty or consists of a ***root*** node connected to an ordered pair of binary trees.

We can also define them using explicit external nodes (null pointers) to represent empty trees. Then a binary tree is an external node or an internal node connected to an ordered pair of binary trees.

There is a unique path from the root to each node. The length of this path is the ***depth*** of the node. The maximum depth of all nodes is the ***height*** of the tree.

Binary trees can be traversed in several ways, including ***preorder***, ***postorder***, ***inorder***. The latter recursively visits the left child, node, and right child.

1.7 Limits

We only need to deal with functions that take nonnegative values in this course. Write $\lim_{x\to\infty} f(x) = \infty$ if for **all** $N > 0$ there is **some** point past which $f(x) > N$ for **all** x, and write $\lim_{x\to\infty} f(x) = 0$ if for **all** $\varepsilon > 0$ there is **some** point past which $f(x) < \varepsilon$ for **all** x.

If $0 < L < \infty$ we say $\lim_{x\to\infty} f(x) = L$ if for all $\varepsilon > 0$ there is **some** point beyond which $L - \varepsilon < f(x) < L + \varepsilon$ for **all** x.

Example 1.6. $\lim_{x\to\infty} x^2 = \infty$, $\lim_{x\to\infty} 1/x = 0$, $\lim_{x\to\infty} \frac{x-1}{x+1} = 1$.

An important use of limits is to compute the ***derivative*** of a function. This gives the instantaneous rate of change, and is defined by

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

when this limit exists.

Example 1.7. If $f(x)$ is respectively e^x, $\ln x$, x^a for some constant a, then $f'(x)$ is respectively e^x, $1/x$, ax^{a-1}.

There are some basic rules for derivatives, for example $(f+g)' = f' + g', (fg)' = fg' + f'g$.

An important technique is ***L'Hôpital's rule***:

$$\lim_{x\to\infty} f(x)/g(x) = \lim_{x\to\infty} f'(x)/g'(x)$$

as long as the latter exists.

Example 1.8. $\lim_{x\to\infty} \frac{e^x}{x^2} = \lim_{x\to\infty} \frac{e^x}{2x} = \lim_{x\to\infty} \frac{e^x}{2} = \infty$.

Example 1.9. What is $\lim_{x\to\infty} \frac{\ln x}{x}$?

Another use of limits is to define the definite ***integral*** of a function, which gives the area under the curve $y = f(x)$. We compute them usually using the Fundamental Theorem of Calculus: if we can find F such that $F' = f$, then

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

Part I

Algorithm analysis

Lecture 2

What is an algorithm and why analyse it?

An ***algorithm*** is a sequence of clearly stated rules that specify a step-by-step method for solving a given problem. The rules should be unambiguous and sufficiently detailed that they can be carried out without creativity. Some examples of algorithms are a (sufficiently detailed) cake recipe, the usual primary school method for multiplication of decimal integers, quicksort.

Algorithms predate electronic computers by thousands of years – for example Euclid's greatest common divisor algorithm (seen in COMPSCI 225).

Algorithms and programs are different – a *program* is a sequence of computer instructions implementing the algorithm. A program may implement more than one algorithm.

Experience in computing over many decades shows that more performance gains can be achieved by optimizing algorithms than by optimizing other factors such as processors, languages, compilers, or human programmers.

Algorithms that have not been analysed for correctness often lead to major bugs in programs. The analysis process often results in us discovering simpler algorithms. Many algorithms have parameters that must be set before implementation, and analysis allows us to set the optimal values.

Example 2.1. The Fibonacci sequence is recursively defined by

$$F(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1; \\ F(n-1) + F(n-2) & \text{if } n \geq 2. \end{cases}$$

This immediately suggests a recursive algorithm.

Algorithm 1 Slow method for computing Fibonacci numbers.

```
1: function SLOWFIB(integer n)
2:     if n < 0 then return 0
3:     else if n = 0 then return 0
4:     else if n = 1 then return 1
5:     else return SLOWFIB(n − 1) + SLOWFIB(n − 2)
```

The algorithm `slowfib` is obviously correct, but does a lot of repeated computation. With a small (fixed) amount of extra space, we can do better, by working from the bottom up instead of from the top down.

Algorithm 2 Fast method for computing Fibonacci numbers.

```
1:  function FASTFIB(integer n)
2:      if n < 0 then return 0
3:      else if n = 0 then return 0
4:      else if n = 1 then return 1
5:      else
6:          a ← 1                          ▷ stores F(i) at bottom of loop
7:          b ← 0                          ▷ stores F(i − 1) at bottom of loop
8:          for i ← 2 to n do
9:              t ← a
10:             a ← a + b
11:             b ← t
12:     return a
```

Proving correctness of `fastfib` is done by induction on n. We omit the proof here.

Example 2.2. It is easy to see that the number of additions, function calls, etc needed by `fastfib` to compute $F(n)$ has the form $An + B$ for some constants A, B. What about `slowfib`?

Lecture 3

How to measure running time?

There are three main characteristics of an algorithm designed to solve a given problem.

Domain of definition: The set of legal inputs.

Correctness: The algorithm gives correct output for each legal input. This depends on the problem we are trying to solve, and can be tricky to prove.

Resource use: This is usually the computing time and memory space.

- This depends on the input, and on the implementation (hardware, programmer skill, compiler, language, ...).
- It usually grows as the input size grows.
- There is a trade-off between resources (for example, time vs space).
- Running time is usually more important than space use.

In this course we mainly consider how to estimate resource use of an algorithm, ignoring implementation as much as possible.

Given an algorithm `algo`, the actual running time on a given input `inp` depends on many implementation details. This is not desirable because it does not allow direct comparison of algorithms. The running time usually grows with the size of the input. Running time for very small inputs is not usually important; it is large inputs that cause problems if the algorithm is inefficient.

We use a clean mathematical model of the problem. For example, sorting distinct records is really the same problem as computing the inverse of a permutation.

Definition 3.1. We define a notion of ***input size*** on the data. This is a positive integer; for example, the number of records in a database to be sorted.

We use the concept of ***elementary operation*** as our basic measuring unit of running time. This is any operation whose execution time does not depend on the size of the input. The running time $T(\texttt{inp})$ of algorithm `algo` on input `inp` is the number of elementary operations used when `inp` is fed into `algo`.

3.1 How running time scales with problem size

The table below describes the growth rate of some commonly used functions as the problem size grows. Note that there are about 3×10^{18} nanoseconds in a century.

Running time		**Input size**			
Function	*Notation*	10	100	1000	10^7
Constant	1	1	1	1	1
Logarithmic	$\lg n$	1	2	3	7
Linear	n	1	10	100	10^6
"Linearithmic"	$n \lg n$	1	20	300	7×10^6
Quadratic	n^2	1	100	10000	10^{12}
Cubic	n^3	1	1000	10^6	10^{18}
Exponential	2^n	1	10^{27}	10^{298}	$10^{3010296}$

Example 3.2. A quadratic algorithm with running time $T(n) = cn^2$ uses 500 elementary operations for processing 10 data items. How many will it use for processing 1000 data items?

Example 3.3. Algorithm A takes n^2 elementary operations to sort a file of n lines, while Algorithm B takes $50n \lg n$. Which algorithm is better when $n = 10$? When $n = 10^6$? How do we decide which algorithm to use?

Example 3.4. Algorithms **A** and **B** use exactly $T_{\mathrm{A}}(n) = c_{\mathrm{A}} n \lg n$ and $T_{\mathrm{B}}(n) = c_{\mathrm{B}} n^2$ elementary operations, respectively, for a problem of size n. Find the faster algorithm for processing $n = 2^{20}$ data items if **A** and **B** spend 10 and 1 operations, respectively, to process $2^{10} \equiv 1024$ items.

Lecture 4

Techniques for estimating running time

Here are some easy basic rules.

- Running time of disjoint blocks adds.
- Running time of nested loops with non-interacting variables multiplies.
- For example, single, double, and triple loops with fixed number of elementary operations inside the inner loop yield linear, quadratic, and cubic running time.

Algorithm 3 Swapping two elements in an array.

Require: $0 \le i \le j \le n-1$

```
function SWAP(array a[0..n − 1], integer i, integer j)
    t ← a[i]
    a[i] ← a[j]
    a[j] ← t
    return a
```

Algorithm 3 is a constant time algorithm.

Algorithm 4 Finding the maximum in an array.

```
function FINDMAX(array a[0..n − 1])
    k ← 0                                   ▷ location of maximum so far
    for j ← 1 to n − 1 do
        if a[k] < a[j] then
            k = j
    return k
```

Algorithm 4 is a linear time algorithm, since it makes one pass through the array and does a constant amount of work each time.

What happens if the loop variable changes in a more complicated way?

Algorithm 5 Example: exponential change of variable in loop.

```
i ← 1
while i < n do
    i ← 2i
    print i
```

Example 4.1. What is the running time for Algorithm 5 and why?

Algorithm 6 Snippet: Nested loops.

```
for i ← 1 to n do
    for j ← i to n do
        print i + j
```

Example 4.2. What is the running time for Algorithm 6 and why?

What do we do if the control flow of the algorithm is more complicated? For example, how do we handle **if** statements?

Algorithm 7 Snippet: If statements.

```
for i = 1; i < n; i ← 2i do
    for j = 1; j < n; j ← 2j do
        if j = 2i then
            for k = 0; k < n; k ← k + 1 do
                { constant number of operations }
        else
            for k = 1; k < n; k ← 3k do
                { constant number of operations }
```

Example 4.3. What is the running time for Algorithm 7 and why?

Algorithm 8 Snippet: Nested loops 2.

```
m ← 2
for j ← 1 to n do
    if j = m then
        m ← 2m
        for i ← 1 to n do
            ...                          ▷ constant number of elementary operations
```

Example 4.4. Let us roughly estimate the running time of Algorithm 8.
The inner loop is executed k times for $j = 2, 4, \ldots, 2^k$ where $k \leq \lg n < k + 1$. The total time for the elementary operations is proportional to kn, that is, $T(n) = n\lfloor \lg n \rfloor \approx n \lg n$.

Algorithm 9 Snippet: Nested loops 3.

```
m ← 1
for j ← 1 step j ← j + 1 to n do
    if j = m then
        m ← m(n − 1)
        for i ← 0 step i ← i + 1 to n − 1 do
            ...                          ▷ constant number of elementary operations
```

Example 4.5. Is the running time of Algorithm 9 quadratic or linear?

Lecture 5

Asymptotic notation

5.1 Asymptotic comparison of functions

In order to compare running times of algorithms we want a way of comparing the growth rates of functions. We want to see what happens for large values of n – small ones are not relevant, because almost any algorithm is good enough in practice for very small input. We are not usually interested in constant factors and only want to consider the dominant terms in the running time.

The standard mathematical approach to this is to use ***asymptotic notation*** O, Ω, Θ which we will now describe.

5.2 Big-O notation

Definition 5.1. Suppose that f and g are functions from $\mathbb{N}$ to $\mathbb{R}$, which take on non-negative values.

- Say f ***is*** $O(g)$ ("f is Big-Oh of g") if there is some $C > 0$ and some $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$, $f(n) \leq Cg(n)$.

  Informally, f/g is eventually bounded away from infinity, and f grows at most as fast as g.

- Say f ***is*** $\Omega(g)$ ("f is big-Omega of g") if g is $O(f)$.

  Informally, f/g is eventually bounded away from zero, and f grows at least as fast as g.

- Say f ***is*** $\Theta(g)$ ("f is big-Theta of g") if f is $O(g)$ and g is $O(f)$.

  Informally, f/g is bounded away from zero and infinity, and f grows at the same rate as g.

Note that $O(g)$ is really a class of functions and strictly speaking we should write $f \in O(g)$, but we use the less formal terminology to reduce the overload of symbols.

5.3 Examples

Example 5.2. Every linear function $f(n) = an + b$, $a > 0$, is $O(n)$.
Proof. $an + b \leq an + |b| \leq (a + |b|)n$ for $n \geq 1$.

Example 5.3. If $f(n) = n, g(n) = n^2/2$, then f is $O(g)$ and g is not $O(f)$, so g grows asymptotically faster than f.
Proof. First note that $f(n) \leq 2g(n)$ for $n \geq 0$ (because $n \leq n^2$). Conversely, suppose that eventually $n^2 \leq Cn$. Then $n \leq C$ for all sufficiently large n, a contradiction.

Example 5.4. Show that $n \lg n$ is $O(2^{-10}n^2)$.

Note that we could always reduce n_0 at the expense of a bigger C but it is often easier not to. For example, in Example 5.3, we could have used $n_0 = 2$ and $C = 1$, because $n \leq n^2/2$ for all $n \geq 2$.

We usually do not prove such results from the definition but you need to know how to, in case the following rules do not apply.

5.4 Rules for asymptotic notation

Irrelevance of constant factors: If $c > 0$ is constant then cf is $\Theta(f)$.

Transitivity: If f is $O(g)/\Omega(g)/\Theta(g)$ and g is in $O(h)/\Omega(h)/\Theta(h)$, then f is $O(h)/\Omega(h)/\Theta(h)$.

Sum rule: If f_1 is $O(g_1)$ and f_2 is $O(g_2)$ then $f_1 + f_2$ is in $O(\max\{g_1, g_2\})$.

Product rule: If f_1 is $O(g_1)$ and f_2 is $O(g_2)$ then $f_1 f_2$ is $O(g_1 g_2)$.

Limit rule: *Suppose that* $L := \lim_{n\to\infty} f(n)/g(n)$ *exists.* Then

- if $L = 0$ then f is $O(g)$ and f is not $\Omega(g)$;
- if $0 < L < \infty$ then f is $\Theta(g)$;
- if $L = \infty$ then f is $\Omega(g)$ and f is not $O(g)$.

L'Hôpital's rule is often useful for the application of the limit rule. Note that the limit may not exist at all.

5.5 More examples

Example 5.5. $\log_a(n)$ is $\Theta(\lg n)$ for each $a > 1$.

Example 5.6. $n \lg n$ is $O(n^2)$ and $n \lg n$ is not $\Omega(n^2)$, by the limit rule.

Example 5.7. 2^n is $\Omega(n^{100})$, by the limit rule.

Example 5.8. Is $10^{-100}n^2 + 10^{100}n$ in $O(n)$?

Example 5.9. $1 + (-1)^n$ is $O(1)$ but not $\Theta(1)$ since it takes on the value 0 infinitely often. The limit rule does not apply either.

5.6 Asymptotics via integral approximation

Example 5.10. We want the asymptotic behaviour of $\log(n!) = \sum_{k=1}^{n} \log k$. Clearly there is an upper bound $\log n! \leq n \log n$, because each term in the sum is at most $\log n$. How can we get a lower bound? We use approximation by an integral.

Example 5.11. By integral approximation we obtain H_n is $\Theta(\log n)$ where $H_n := 1 + 1/2 + 1/3 + \cdots + 1/n$ is the n**th harmonic number**.

5.7 Summary of results

Write $f \prec g$ if f is $O(g)$ but f is not $\Theta(g)$, so g grows at a faster rate than f.

Example 5.12. $\log n \prec (\log n)^2 \prec \sqrt{n} \prec n \prec n \log n \prec n(\log n)^2 \prec n^2 \prec n^3 \prec \cdots \prec (1.5)^n \prec 2^n \prec n! \prec n^n$.

Lecture 6

Q&A

Here we consider some more subtle questions that have been ignored so far.

6.1 Is addition of integers an elementary operation?

If the integers (and their sum) can fit into a machine word (typically 64 bits, so not bigger than $9\,223\,372\,036\,854\,775\,807$) then adding two of them can be done in constant time. If the integers are much longer, as occurs for example in symbolic algebra systems, cryptography, etc, then to avoid overflow they must be represented another way ("big integers"). Typically these are strings. In this case the addition is done componentwise and the running time grows linearly with the size of the integers.

Example 6.1. Assuming that $F(n)$ has the order of n decimal digits (which is true), the amount of work done by `fastfib` is of order $1 + 2 + \cdots + n$ which is order n^2, not n. So `fastfib` is technically a quadratic time algorithm!

6.2 Have we been measuring input size correctly?

We used n to measure the size of the integer n. This seems wrong. In fact the number of bits needed to represent n seems a much better idea. If m is the size of the positive integer n, then $m = 1 + \lfloor \lg n \rfloor$, and $2^{m-1} \leq n < 2^m$. This turns `fastfib` from a polynomial time algorithm to an exponential time algorithm! The input size measure **must** be specified in algorithm analysis.

6.3 What happens if there are many inputs of a given size?

We usually do not want to have to consider the distribution of running time over all possible inputs of a given size. There may be (infinitely) many inputs of a given size, and running time may vary widely on these. For example, for sorting the integers $1, \ldots, n$, there are $n!$ possible inputs, and this is large even for $n = 10$.

We consider statistics of $T(\texttt{inp})$ such as ***worst-case*** or ***average-case*** running time for instances `inp` of size n.

6.4 What are the pros and cons of worst and average case analysis?

- Worst-case bounds are valid for all instances: this is important for mission-critical applications.
- Worst-case bounds are often easier to derive mathematically.

- Worst-case bounds often hugely exceed typical running time and have little predictive or comparative value.
- Average-case running time is often more realistic. Quicksort is a classic example.
- Average-case analysis requires a good understanding of the probability distribution of the inputs.

Conclusion: a good worst-case bound is always useful, but it is just a first step and we should aim to refine the analysis for important algorithms. Average-case analysis is often more practically useful, provided the algorithm will be run on "random" data and we have some tolerance for risk.

6.5 Why can constants often be ignored?

A linear time algorithm when implemented will take at most $An + B$ seconds to run on an instance of size n, for some specific constants A, B that depend on the implementation. For large n, this is well approximated by An. Small n are not usually of interest anyway, since almost any algorithm is good enough for tiny instances.

No matter what A is, we can easily work out how the running time scales with increasing problem size (linearly!). The difference between a linear and a quadratic time algorithm is usually huge, no matter what the constants are. For large enough n, a linear time algorithm will always beat a quadratic time one.

Conclusion: in practice we often need to make only crude distinctions. We only need to know whether the running time scales like $n, n^2, n^3, n \log n, 2^n, \ldots$. If we need finer distinctions, we can do more analysis.

6.6 Can we always ignore constants?

When we want to choose between two good algorithms for the same problem ("is my linear-time algorithm faster than your linear-time algorithm?"), we may need to know constants. These must be determined empirically.

For important algorithms that will be used many times, it is worth being more precise about the constants. Even small savings will be worth the trouble.

An algorithm with running time $10^{-10}n^2$ is probably better in practice than one with running time $10^{10}n$, since the latter will eventually beat the former, but only on instances of size at least 10^{20}, which is rarely met in practice.

Conclusion: we should have at least a rough feel for the constants of competing algorithms. However, in practice the constants are usually of moderate size.

6.7 Summary

- Our goal is to find an asymptotic approximation for the (worst or average case) running time of a given algorithm. Ideally we can find a simple function f and prove that the running time is $\Theta(f(n))$.
- The main $f(n)$ occurring in applications are $\log n$, n, $n \log n$, n^2, n^3, 2^n, and each grows considerably faster than the previous one. The gap between n and $n \log n$ is the smallest.

Part II

Analysis of sorting

Lecture 7

The problem of sorting, selection sort

Sorting (especially integers or words) is ubiquitous in computing. Sorting also makes many other problems easier, e.g., selection and finding duplicates. The problem of sorting is as follows.

Definition 7.1. Given n ***keys*** $a_1, \ldots, a_n$ from a totally ordered set, put them in increasing order. The keys may be just part of a larger data record.

Example 7.2. Sort the integers $\{42, 7, 911, -2\}$ with $\leq$ and the words {*banana, computer, Auckland, bamboo*} with alphabetical (lexicographic) order.

7.1 Properties of sorting algorithms

Definition 7.3. A sorting algorithm is ***comparison-based*** if it only uses the order relation to compare keys.

This is consistent with the ADT (Abstract Data Type), object-oriented approach. There are other algorithms like bucket sort, radix sort, etc. that do not do this. They use properties of the data representation such as the decimal expansion of keys. We only analyse comparison-based algorithms, since they work for general data. Note that the data structure used for sorting is important. Sorting an array is a different problem from sorting a linked list. However, these differences affect only efficiency, not correctness, and for correctness we can (and should) consider the algorithm as acting on a general list.

We consider only two elementary operations: a **comparison** of two items and a **move** of an item. For a comparison, we choose keys x and y and answer the question "is $x < y$?" (or maybe we ask which of $x < y, x = y, y < x$ is true). There are two main types of move-related operations.

- We can **swap** the position of elements (at positions i and j say). Each swap requires 3 updates of variables.
- Another way is to insert the element in the list at position i after position j and remove it from its current position.

If the list is implemented using an array, then insertion-deletion is expensive ($\Theta(n-j)$ data moves in worst case), since elements must be moved along, and hence this is not an elementary operation. Then swaps are better.

If the list is implemented as a linked list, then insertion is cheap ($\Theta(1)$ data moves in worst case). So we usually do that instead of swapping.

Definition 7.4. A sorting algorithm is ***in-place*** if it only uses fixed additional space, independent of n. It is ***stable*** if records with equal keys have their order unchanged by the algorithm.

7.2 Selection sort

The sorting algorithm ***selection sort*** works as follows.

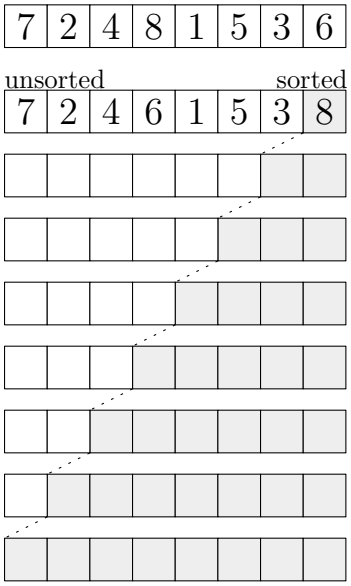
- Find the maximum element of the unsorted part of the list by sequential scan, and move it to the end of the sorted part. Iterate.

This is perhaps the most obvious sorting algorithm. It makes the smallest possible number of swaps of any comparison-based sorting algorithm, so may be useful if data moves are VERY expensive. Is it clearly inefficient, since useful comparison information gathered in each pass is forgotten.

Algorithm 10 Selection sort.

```
1: function SELSORT(list a[0..n − 1])
2:     for i ← n − 1 to 0 do
3:         k ← findmax(a[0..i])
4:         if k ≠ i then
5:             swap(a, i, k)
6:     return a
```

Example 7.5. Complete the following selection sort process.



We now analyse the properties of selection sort.

7.3 What is the running time?

Finding the maximum of $a[i..n-1]$ by sequential search takes $(n-1)-i$ comparisons, so the total number of key comparisons is $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$. The number of index comparisons is n and the number of swaps is at most $n-1$. Hence, the running time is in $\Theta(n^2)$ if we take comparisons and swaps as elementary operations.

7.4 Which inputs give the best and worst case?

The algorithm is very insensitive to the input. Its best and worst case running time are very similar. The best case is when the list is already sorted; the worst is when every swap is needed, and this occurs when the input permutation has no fixed point. However the number of comparisons, which usually dominates the running time, is the same for every input.

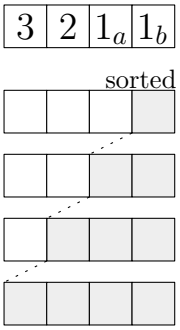
7.5 Is selection sort in-place?

Yes, selection sort is an in-place sorting algorithm. Convince yourself why.

7.6 Is selection sort stable?

No, selection sort is not a stable sorting algorithm.

Example 7.6. Execute selection sort on the following input to convince yourself that the algorithm is not stable.



7.7 What happens if we run selection sort on a linked list?

Running time on a linked list is not much different from performance on an array. The differences involve swap versus insertion, but comparisons dominate the running time anyway.

Lastly, note that there is no better way to find the maximum using a list than what we have done above. But if we change the data structure ... (see Lecture 13 on heapsort).

Lecture 8

Insertion sort

The sorting algorithm ***insertion sort*** works as follows.

- For each element x in the unsorted part of the input list, scan backward while the preceding element exceeds it. Move x to its correct position. Iterate.

This is the sorting method used by card players to arrange cards in a hand. It works well on small lists, or lists that are nearly sorted.

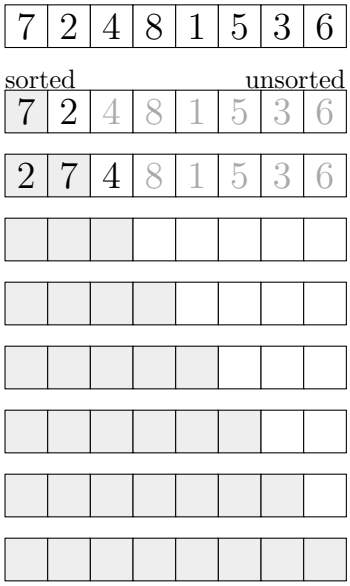
Algorithm 11 Insertion sort.

```
1:  function INSORT(list a[0..n − 1])
2:      for i ← 1 to n − 1 do
3:          k ← a[i]                          ▷ save value so we do not overwrite
4:          j ← i − 1
5:          while a[j] > k and j ≥ 0 do
6:                                  ▷ scan backward to find correct insertion position
7:              a[j + 1] ← a[j]
8:              j ← j − 1
9:          a[j + 1] ← k                      ▷ insert a[i] in correct position
10:     return a
```

Algorithm 12 Insertion sort swap version.

```
1:  function INSORT2(list a[0..n − 1])
2:      for i ← 1 to n − 1 do
3:          j ← i − 1
4:          while a[j] > a[j + 1] and j ≥ 0 do
5:                                  ▷ swap backward to move to right place
6:              swap(a, j, j + 1)
7:              j ← j − 1
8:      return a
```

Example 8.1. Complete the following insertion sort process.



8.1 Insertion sort analysis

The number of key comparisons for fixed i is one more than the number of j such that $j < i$ and $a[i] < a[j]$. Thus the total number of comparisons is $n - 1$ plus the number of ***inversions***, namely pairs (i, j) of indices with $j < i$ and $a[j] > a[i]$ ("out of order" pairs of elements). The number of swaps in the second version is the number of inversions. The number of element assignments in the first version is $n - 1$ plus the number of inversions. In the best case insertion sort runs in linear time. In the worst case the number of inversions is $n(n-1)/2$ (where input is in reverse sorted order) so it runs in quadratic time $\Theta(n^2)$.

Example 8.2. How many inversions are in the permutation 72481536?

8.2 What is the average-case performance of insertion sort on random data?

Example 8.3. What is the average number of inversions in a random permutation of $\{1, \ldots, n\}$?

8.3 Which inputs give the best and worst case?

Unlike selection sort, insertion sort has running time that is very sensitive to the input. As mentioned above, the best case occurs when the input is already sorted, and the worst when the input is in reverse sorted order.

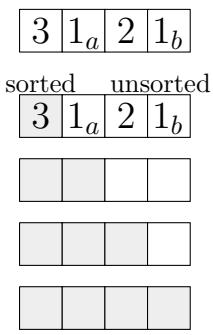
8.4 Is insertion sort in-place?

Yes, insertion sort is an in-place sorting algorithm. Convince yourself why.

8.5 Is insertion sort stable?

Yes.

Example 8.4. Execute insertion sort on the following input to convince yourself that the algorithm is stable.



8.6 Analysis when implemented on a linked list

We can reduce the number of data moves if we use a linked list to implement the list ADT. However, searching to find the right insertion point still takes time in $\Theta(n^2)$ in the worst case.

8.7 How can we improve insertion sort?

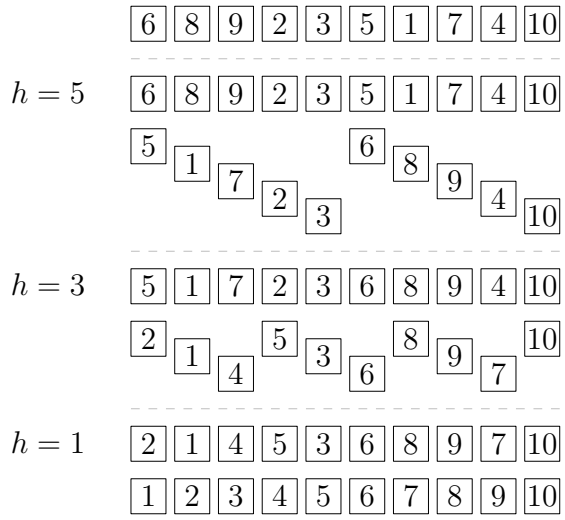
On an array, we can reduce the number of comparisons by using binary search (see future lecture) to find the insertion point. However, the swaps still take time in $\Theta(n^2)$ as above, so this is of little use.

We can improve by changing the algorithm somewhat. We can move elements to the left using big steps, gradually reducing the step size to 1. This is called ***Shellsort*** (D. Shell, 1959).

- Fix $h = n/2$. Run insertion sort on the sublists formed by taking every hth element. Iterate, reducing h to 1 systematically.

The key property is that each run removes inversions and none are restored by later runs. The last run is just insertion sort, but on a file that is almost sorted.

Example 8.5. An example of Shellsort with $h = 5, 3, 1$.



This is a simple algorithm whose analysis is very hard. Values of h chosen from all relevant numbers of the form $2^i 3^j$ have been proved to give $O(n(\log n)^2)$ running time, but better ones are used in practice. Almost nothing is known about average-case running time of Shellsort. Empirically, behaviour that looks like $O(n^{7/6})$ has been obtained.

Shellsort is easy to program and useful in practice for fairly large arrays (say 10^5) especially for partially sorted data. It is substantially better than insertion sort.

Lecture 9

Mergesort

Here we discuss two "industrial strength" sorting algorithms, ***Mergesort*** and ***Quicksort***, and look at the first one in more detail.

- Each algorithm splits the input list into two sublists, recursively sorts each sublist, and then combines the sorted sublists to sort the original list.
- Mergesort (J. von Neumann, 1945): splitting is very easy, most of the work is in combining. We divide the list into left and right half, and merge the recursively sorted lists with a procedure `merge`.
- Quicksort (C. A. R. Hoare, 1962): most of the work is in the splitting, combining is very easy. We choose a ***pivot*** element, and use a procedure `partition` to alter the list so that the pivot is in its correct position relative to the left and right sublists on each side. Then the left and right sublists are sorted recursively.

Each is used as the basis for built-in sorting algorithms in common programming languages.

Algorithm 13 Mergesort.

```
function MERGESORT(list a[0..n − 1])
    if n > 1 then
        m ← ⌊(n − 1)/2⌋                          ▷ median index of list
        l ← MERGESORT(a[0..m])                    ▷ sort left half
        r ← MERGESORT(a[m + 1..n − 1])            ▷ sort right half
        a ← MERGE(l, r)                           ▷ merge both halves
    return a
```

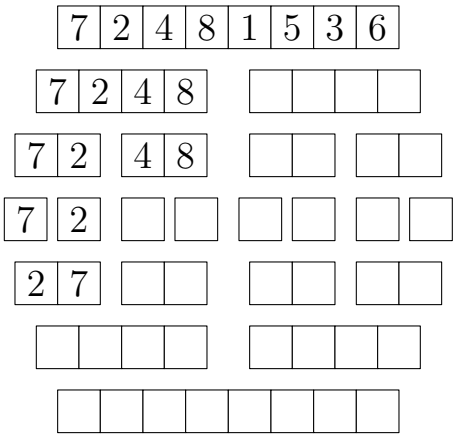
9.1 Linear time merging for Mergesort

The key is an efficient merge algorithm. If L_1 and L_2 are sorted lists, they can be merged into one sorted list in linear time.

- Start pointers at the beginning of each list.
- Compare the elements being pointed to and choose the lesser one to start the sorted list. Increment that pointer.
- Iterate until one pointer reaches the end of its list, then copy remainder of other list to the end of the sorted list.

If the list is implemented as an array, this merging takes linear extra space (cannot be done in place). But if linked lists are used, it can be done in place. In any case the number of comparisons is in $\Theta(n)$.

Example 9.1. Execute mergesort on the following input.



9.2 Mergesort analysis

The number of comparisons C_n satisfies (approximately)

$$C_n = \begin{cases} C_{\lceil n/2 \rceil} + C_{\lfloor n/2 \rfloor} + n & \text{if } n > 1; \\ 0 & \text{if } n = 1. \end{cases}$$

Thus if n is a power of 2 we have $C_n = 2C_{n/2} + n$. We need to develop tools to solve recurrences like this — see the next lecture.

9.3 Which inputs give the best and worst case?

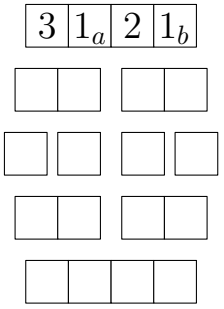
Mergesort is not very sensitive to the input order. If the input is already sorted, the merge operation does the fewest possible comparisons. The worst case occurs when the input looks like 5,1,7,3,6,2,8,4 (every possible comparison is made at every level of the merge). However the input only affects lower order terms – it turns out (next lecture) that mergesort does about $n \lg n$ comparisons on every input.

9.4 Is mergesort in-place?

Example 9.2. Show by example that it is not, when implemented on an array.

9.5 Is mergesort stable?

Example 9.3. Execute mergesort on the following input to convince yourself that the algorithm is stable.



9.6 Analysis when implemented on a linked list

Since finding the middle element of the list is time-consuming, in this case it makes more sense to "unroll" the recursion and process all the instances at the bottom of the recursion tree (merging two size 1 lists) first, then merge all the resulting size 2 lists two-by-two, etc. This "bottom-up mergesort" is not only equivalent to mergesort, it is in place! Note that "bottom-up mergesort" is strictly speaking a different algorithm from mergesort.

Lecture 10

Recurrences

Definition 10.1. A ***recurrence relation*** or ***difference equation*** is an equation that defines an unknown function F recursively. The value $F(n)$ is determined by the values $F(0), F(1), \ldots, F(n-1)$ for sufficiently large n. The smaller values of n are called the ***initial conditions***.

10.1 Important special cases

Tower of Hanoi: $T(n) = 2T(n-1) + 1, T(1) = 0$.

Searching a linked list: $T(n) = T(n-1) + 1, T(0) = 0$.

Insertion sort: $T(n) = T(n-1) + n, T(0) = 0$.

Binary search: $T(n) = T(n/2) + 1, T(1) = 0$ makes sense when n is a power of 2.

Mergesort: $T(n) = 2T(n/2) + n, T(1) = 0$ makes sense when n is a power of 2.

Quicksort: $T(n) = (2/n)\sum_{i<n} T(i) + n - 1, T(1) = 0$.

Important examples in this course are ***constant coefficient linear recurrences*** which have the form $F(n) = \sum_{k=1}^{K} a_k F(n-k)$ for some fixed K and some constants a_k. Examples are the Fibonacci numbers, $F(n) = F(n-1) + F(n-2)$ and the power of 2, $F(n) = 2F(n-1)$.

We deal with only two easy techniques here: "bottom-up" and "top-down" guessing. In each case we guess a pattern and (should) prove it by mathematical induction. Sometimes a simple change of variable can convert a more difficult-looking recurrence into an easy one.

10.2 Top-down solution method example

Example 10.2. We are given $T(n) = 2T(n-1) + 1, T(1) = 0$. We can iterate this to obtain

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2[2T(n-2) + 1] + 1 = 2^2 T(n-2) + (2+1) \\
&= 2^2[2T(n-3) + 1] + (2+1) = 2^3 T(n-3) + (2^2 + 2 + 1) \\
&= \cdots = 2^{n-1}T(1) + (2^{n-2} + \cdots + 2 + 1) \text{ if } n \geq 1 \\
&= \sum_{0 \leq i < n-1} 2^i = 2^{n-1} - 1.
\end{aligned}$$

10.3 Bottom-up solution method example

Example 10.3. We are given $T(n) = 2T(n-1)+1$, $T(1) = 0$. We compute values from the bottom up using the recurrence.

We have $T(1) = 0$, $T(2) = 2T(1)+1 = 1$, $T(3) = 2T(2)+1 = 3$, $T(4) = 7$, $T(5) = 15$. We guess $T(n) = 2^{n-1} - 1$. The guess is correct when $n = 1$. Now assume that $n > 1$ and the formula holds for all smaller values than n. From the recurrence we then have $T(n) = 2T(n-1)+1 = 2(2^{n-2}-1)+1 = 2^{n-1}-1$. Thus the formula holds for all $n \geq 1$, by **mathematical induction**.

10.4 Change of variable

Example 10.4. Consider the divide-and-conquer recurrence $T(n) = T(n/2)+1$, $T(1) = 0$. This makes sense for n a power of 2.

Changing variable by $n = 2^i$, $T(n) = U(i)$ gives $U(i) = U(i-1)+1$, $U(0) = 0$. By top-down or bottom-up guessing as above, we get

$$T(n) = U(i) = i = \lg n$$

when n is a power of 2.
Similarly the mergesort recurrence $T(n) = 2T(n/2) + n, T(1) = 0$ has solution $T(n) = n \lg n$ for n a power of 2.

Example 10.5. How does the solution to Example 10.4 look if we do not do the change of variable?

10.5 Mathematical extra: dealing with floors and ceilings, etc

The mergesort recurrence involves floor and ceiling functions. How do we deal with these?

We can solve the mergesort recurrence $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ asymptotically. "Exact" solutions are hard. See COMPSCI 720 and beyond. Then we consider the recurrence $\hat{T}(n) = 2\hat{T}(\lceil (n/2) \rceil) + n$. It is easily shown that $T \leq \hat{T}$, and as above $\hat{T}(n) = n \lg n$ for n a power of 2. We can then show that $T(n)$ is $O(n \lg n)$. A similar argument gives $T(n)$ is $\Omega(n \lg n)$.

In real life we usually do not have exactly an additive term n but just some $f(n)$ where $f(n)$ is $\Theta(n)$. Similar techniques show that we get the same answer, asymptotically.

10.6 How do we solve the Fibonacci recurrence?

Recall $F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$, $(n \geq 2)$. Such linear homogeneous recurrences can be solved exactly with more theory than we have time for in this course.

Example 10.6. Try to solve the Fibonacci recurrence using the top-down or bottom-up method. Why does it fail?

To find an asymptotic solution for $F(n)$ (which is all we need here), we conjecture, based on examples, that $F(n)$ grows exponentially in n. In other words, there are $C > 0$ and $r > 1$ such that $F(n) \geq Cr^n$ for all sufficiently large n.

Try to prove this by induction. The obvious method gives $F(n+1) = F(n) + F(n-1) \geq C(r^n + r^{n-1}) = Cr^n(1 + r)$, so that we easily obtain the conclusion we want if and only if $1 + r \geq r^2$, or $r \leq \phi := (1 + \sqrt{5})/2 \approx 1.618$.

Now suppose $F(n) \leq Dr^n$ for all sufficiently large n. Trying to prove this by induction, we require $1 + r \leq r^2$ or $r \geq \phi$.

Thus $F(n)$ is $\Theta(\phi^n)$.

Lecture 11

Quicksort

Quicksort was described briefly in Lecture 9. We give the details now. Note that a purely recursive version would do a lot of copying to create new versions of the left and right sublists, as happens with mergesort. In terms of programming languages, we are talking about "passing by value". We can get an in-place implementation if we use functions that can modify the value of their argument without explicitly returning them (they have "side effects", usually implemented in programming languages via "passing by mutable reference", which is commonly done for arrays by default).

Algorithm 14 Quicksort - basic.

Require: $0 \le i \le j \le n-1$ ▷ Side effect: changes a
  function QUICKSORT(list $a[0..n-1]$, integer i, integer j)
    if $i < j$ **then**
      $q \leftarrow$ PARTITION(a, i, j) ▷ put pivot in correct position
      QUICKSORT$(a, i, q-1)$ ▷ sort left half
      QUICKSORT$(a, q+1, j)$ ▷ sort right half

The exact way in which quicksort works depends on the details of the way we choose the pivot element and how the partition algorithm works. Each choice for these gives a different quicksort algorithm. We use the first entry as pivot element for a basic presentation, but there are better options in practice.

Example 11.1. Illustration of quicksort. First the array is partitioned and the pivot element 4 moved into the middle. Then the left side ("< 4") and right side ("> 4") are recursively sorted.

4	2	3	5	1	6
$<$			$=$	$>$	
2	3	1	4	5	6
$<$	$=$	$>$		$=$	$>$
1	2	3	4	5	6

11.1 Linear time partitioning

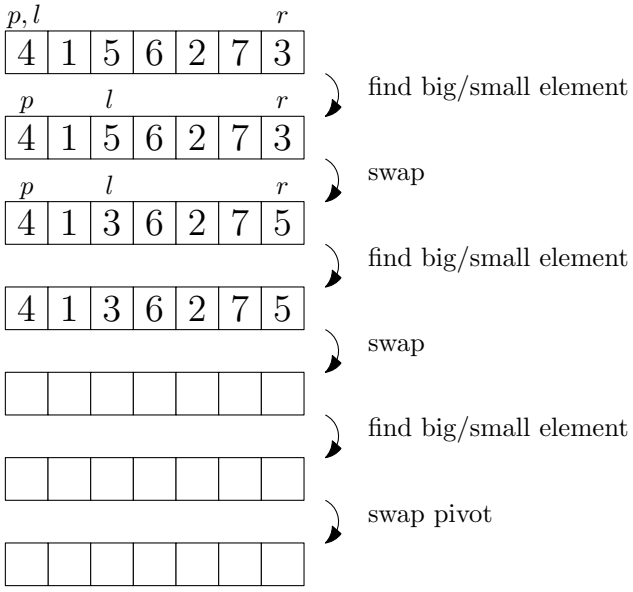
Given a list L and an element p of L called the pivot, we can partition so that all elements to the left of L are $\le p$ and all to the right are $\ge p$. One method is Algorithm 15 (Hoare, 1960) and it works as follows.

- Start with pointers at opposite ends of the list. Stop when pointers cross.
- At each step, increment the left pointer until we reach an element $\geq p$, and decrement the right one until we reach an element $\leq p$.
- Swap these elements and continue. When pointers cross, swap right pointer with pivot.

Algorithm 15 Partition - Hoare's method.

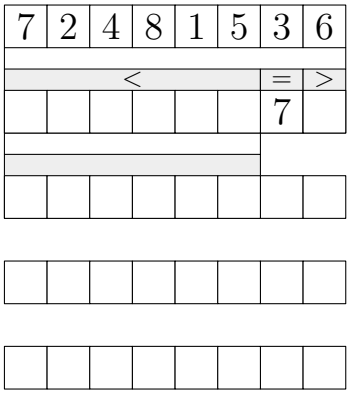
```
Require: 0 ≤ i ≤ j ≤ n − 1                          ▷ Side effect: changes a
  function PARTITION(list a[0..n − 1], integer i, integer j)
    p ← a[i]                                          ▷ pivot element
    l ← i                                             ▷ left pointer
    r ← j + 1                                         ▷ right pointer
    while True do
      repeat
        l ← l + 1                                     ▷ find big element
      until a[l] ≥ p
      repeat
        r ← r − 1                                     ▷ find small element
      until a[r] ≤ p
      if l < r then
        swap(a, l, r)                                 ▷ swap big and small elements
      else
        swap(a, i, r)                                 ▷ put pivot in correct place
        return r
```

Example 11.2. Complete the following example for the partition algorithm Algorithm 15. Begin with setting the pointers p, l and r after finding the big and small element in the second iteration.



Example 11.3. Does the quicksort algorithm in Example 11.1 use Hoare's method, Algorithm 15, for partitioning?

Example 11.4. Execute the quicksort algorithm Algorithm 14 using Hoare's partition algorithm.



11.2 Quicksort analysis

The number of comparisons satisfies a recurrence like $C_n = C_{p-1} + C_{n-p} + n$. If we are unlucky (like when the list is already sorted), this degenerates to $C_n = C_{n-1} + n$ and we get quadratic running time.

However, quicksort seems to behave well in practice. We can prove that if elements are distinct and all input permutations are equally likely, then quicksort has average running time in $O(n \log n)$.

The recurrence for this average running time is basically

$$a_n = \frac{2}{n} \sum_{i<n} a_i + n - 1.$$

This requires a different solution method to the easier recurrences.

11.3 Quicksort recurrence solution

Let a_n be the average of C_n so that $a_n = n - 1 + \frac{1}{n} \sum_{1 \le p \le n} (a_{p-1} + a_{n-p})$. Collect common terms in the sums to obtain

$$a_n = n - 1 + \frac{2}{n} \sum_{j=0}^{n-1} a_j.$$

Note that to compute each term using this recurrence, we require knowledge of all previous terms. We first *eliminate the history*, (a general form of *telescoping*).

We have for $n > 1$

$$\begin{aligned} na_n - (n-1)a_{n-1} &= n(n-1) + 2 \sum_{0 \le j < n} a_j \\ &\quad - (n-1)(n-2) - 2 \sum_{0 \le j < n-1} a_j \\ &= 2(n-1) + 2a_{n-1}. \end{aligned}$$

Thus $na_n = 2(n-1) + (n+1)a_{n-1}$, so that

$$\frac{a_n}{n+1} = \frac{2(n-1)}{n(n+1)} + \frac{a_{n-1}}{n} = \frac{a_{n-1}}{n} + \frac{4}{n+1} - \frac{2}{n}.$$

We can iterate to obtain

$$\frac{a_n}{n+1} = \frac{a_1}{2} + \sum_{j=2}^{n} \left(\frac{4}{j+1} - \frac{2}{j} \right) = 4(H_{n+1} - (1 + 1/2)) - 2(H_n - 1) \cdots = 2H_n + \frac{4}{n+1} - 4.$$

11.4 Which inputs give the best and worst-case?

The worst case number of comparisons occurs when the pivot is always at the end of the sublist. For example, if we always choose the first element as the pivot and the input is in sorted order, this will happen. The running time is then $\Theta(n^2)$. The best case occurs when the pivot turns out to be the median element, so that the left and right subarrays are balanced at each level of the recursion, and this gives running time in $\Theta(n \log n)$ as with mergesort.

11.5 Is quicksort in-place?

Not technically, but almost. It requires $\Theta(\lg n)$ space for the recursion stack. This is very little extra space in practice, but it is not a constant amount.

11.6 Is quicksort stable?

No.

Example 11.5. Create an example that shows that quicksort is not a stable sorting algorithm.

11.7 What happens if we use quicksort on a linked list?

Although it could be done, quicksort is almost never used on a linked list. It is too difficult to quickly find a good pivot element, for example.

11.8 How can we improve quicksort?

If we always choose the first element of the list as the pivot, it is easy to implement but the worst-case runtime is $\Omega(n^2)$, and it is easy for a malicious adversary to submit input which will trigger this behaviour. This naive choice (the first one used by Hoare) is very rarely used any more.

For a better way of choosing the pivot, we can take the median of a sample of (say 3) elements. This reduces the likelihood of worst-case behaviour and also makes performance on sorted arrays and arrays with equal keys better.

Choosing a pivot element uniformly at random (or randomly shuffling the list and choosing the first element) gives a **randomized** algorithm whose expected running time on every input can be proved to be in $O(n \log n)$.

No matter how we choose the pivot as above, there is always some chance of quicksort running in quadratic time. Choosing the exact median turns out to be too time-consuming: see Lecture 14.

It is best to use insertion sort once the sublists become very small, to avoid excessive overhead owing to recursion. Choosing the cutoff point requires more detailed analysis; somewhere around 10 is often used. This idea can be used with any recursive sorting algorithm.

Lecture 12

Lower complexity bound for sorting

We have seen that various algorithms do different amounts of work on different inputs, and their best and worst case inputs are different in general.

Example 12.1. How many comparisons do selection sort, insertion sort, mergesort, quicksort do on the following input?

3	4	2	1

selection sort

insertion sort

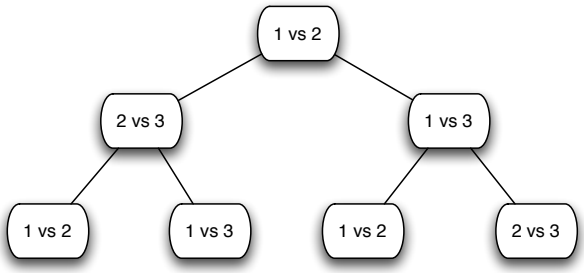
mergesort

quicksort

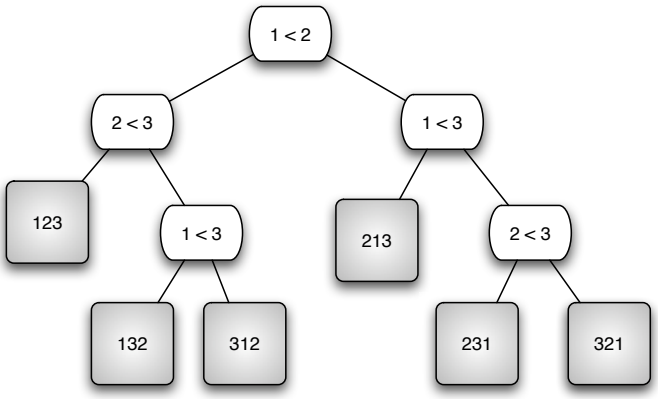
Is it possible to sort n items in $O(n)$ time in the worst case? Is there a general sorting algorithm A and constant $c > 0$ such that for all inputs of size n, A uses at most cn comparisons to sort? Interestingly, the answer is no, as we shall now see.

Let `algo` be a comparison-based sorting algorithm. The execution of `algo` on an input of n distinct keys can be modelled by a ***decision tree***. This is a binary tree whose leaves are the $n!$ possible permutations of the keys. At each step we can only ask a Yes/No question of the form "$(a_i < a_j)$?", and this determines the branching: left for Yes, right for No. The number of comparisons required to obtain the output at a leaf is the length of the path from the root to that leaf (the depth of the leaf node). Thus the worst case number of comparisons is the maximum depth (the height) of the tree.

Example 12.2. Here is the decision tree for selection sort when run on an input list of size 3. The label "i vs j" means that the ith element of the input list is compared to the jth element, by asking "is $a[i] < a[j]$?"



The next figure shows the corresponding decision tree for insertion sort. We add in the grey background nodes to give information about which input yields which sequence of comparisons.



12.1 Worst case height of binary decision tree

So, what can be said about the height of a binary tree with n nodes?

Suppose that T is a binary tree with n nodes. Going from the root, the number of nodes at each level at most doubles. So if the maximum depth (height) is h then the number of leaves is at most 2^h.

In our sorting decision tree we have $n!$ leaves. Thus we must have $2^h \geq n!$ and therefore $h \geq \lg n!$.

12.2 Worst case running time of comparison-based sorting algorithm

We have already shown that $\log n!$ is $\Omega(n \lg n)$. Thus the worst-case number of comparisons of any comparison-based sorting method must grow at least at the asymptotic rate $n \log n$ (any input that had duplicates could only make the worst case worse!)

Example 12.3. We showed that the worst case for any sorting algorithm is in $\Omega(n \lg n)$. What about the average case?

Example 12.4. Radix sort can sort n strings in time in $\Theta(n)$. What can we conclude about radix sort? Look it up and check your guess.

Lecture 13

Priority queues and heapsort

In selection sort, finding the minimum of $a[i..n-1]$ by sequential search is slow, and it dominates the running time. Can we do this operation faster? Not with the current data structure. We want a data structure that allows us to find and extract the minimum quickly. This will have applications to problems other than sorting.

13.1 Priority queues

Definition 13.1. A ***priority queue*** is a container ADT where each element has a key (from a totally ordered set, as with sorting) called its priority. There are operations allowing us to insert an element, and to find and delete the element of highest priority.

Priority queues are important in many areas: discrete event simulation, graph algorithms (later in this course), sorting, ... Priority queues can be implemented in many ways: unsorted list, sorted list, binary heap, binomial heap, ...

Example 13.2. Show how (a) a queue and (b) a stack can be interpreted as special cases of a priority queue.

13.2 Priority queue sort

Suppose we are given an input list. Start with an empty priority queue Q and

- successively insert all elements into Q, using the sorting keys as the priority;
- successively remove the highest priority element until Q is empty.

This gives a sorting algorithm that is obviously correct.

13.3 Priority queue sort analysis

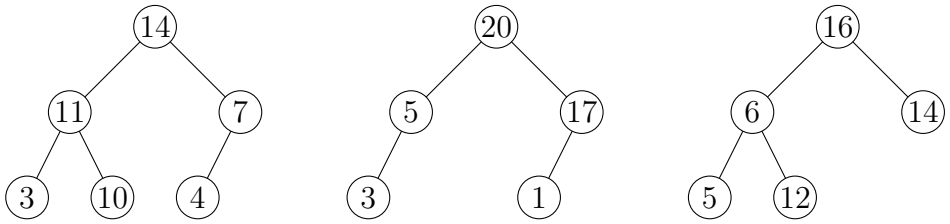
- If we implement Q using an unsorted list, we obtain selection sort. Insertion takes $\Theta(1)$ time but deletion time is $\Theta(n)$. The total is quadratic.
- If we use a sorted list to implement Q, we obtain insertion sort. Insertion takes $\Theta(n)$ time but deletion is $\Theta(1)$. The total is quadratic.
- We can do better with an implementation in which insertion and deletion each take time $O(\log n)$. The simplest is the binary heap.

13.4 Priority queue: binary heap implementation

Definition 13.3. A ***binary heap*** is a binary tree that

- is ***left-complete*** (every level except perhaps the last is full and the last level is left-filled);
- has the ***partial order property*** (on every path from the root, the keys decrease).

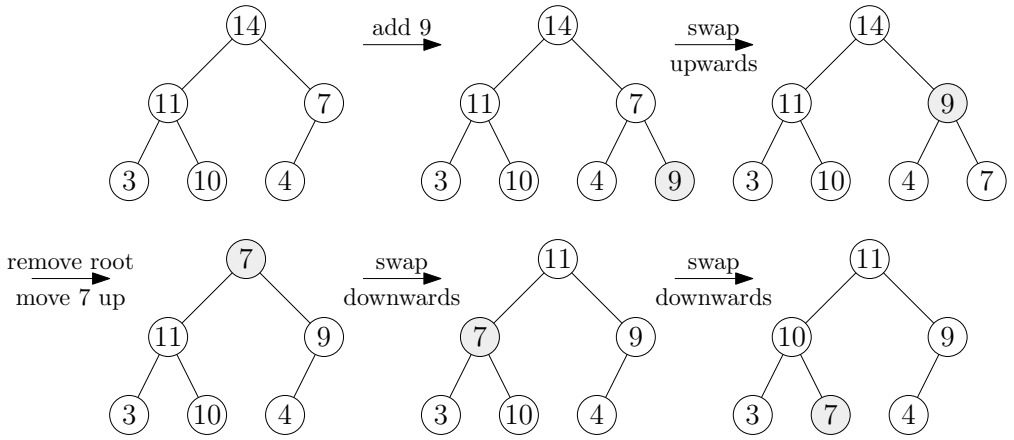
Example 13.4. The tree on the left is a heap. The other two trees are not heaps; the one in the middle is not left-complete and the one on the right does not have the partial order property.



A heap can implement a priority queue as follows.

- Keys are stored in nodes.
- To insert a node, create a new leaf at the bottom level as far left as possible. Swap it upward until no swap is required.
- To delete the maximum, remove the root. Put the rightmost leaf in the root position. Swap it downward (choosing the larger child each time) until no swap is required.

Example 13.5. Example of heap insertion and deletion.



13.5 Heapsort analysis

- Building a heap using n successive insertions takes time in $O(n \log n)$ since the tree has height in $O(\log n)$.
- Deleting the root n times, restoring the heap property each time, takes time in $O(n \log n)$.
- Thus heapsort is a worst-case $\Theta(n \log n)$ sorting algorithm, like mergesort.

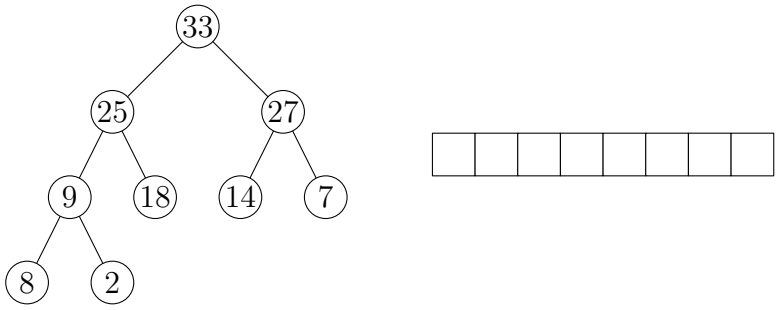
13.6 How can we improve the algorithm?

It turns out we can make heapsort work in-place without building a separate heap, because the tree structure is so restricted.

13.7 Array representation of binary heap

- A binary heap can be represented **implicitly** (without pointers) in an array.
  - Each node corresponds to an array index (starting with 1).
  - The children of the node corresponding to index k are in positions $2k$ (left) and $2k+1$ (right). The keys are stored directly in the array.
- To insert a key x, put it in position $n+1$ and swap as above.
- To delete the root, swap $a[1]$ with $a[n]$, and swap as above. Note that deleted elements end up at the end of the array.

Example 13.6. Give the array representation of the binary heap.

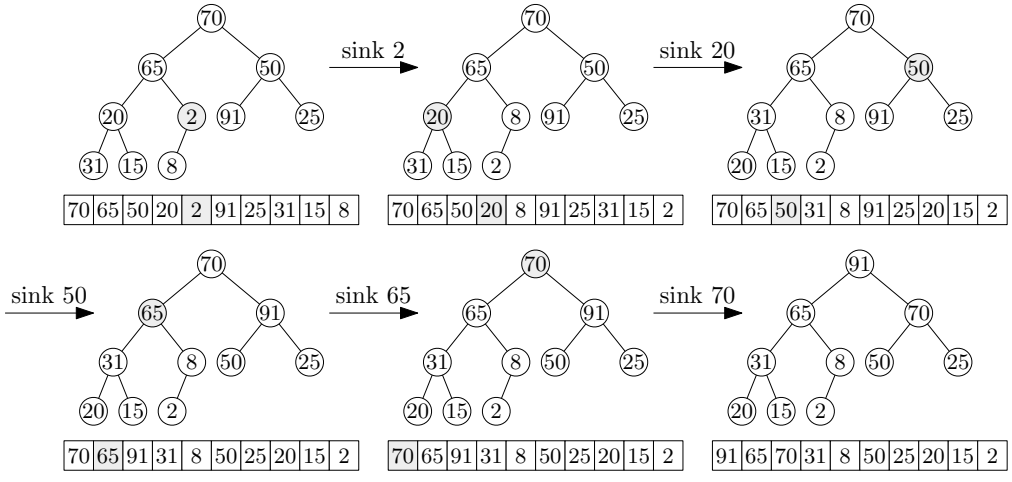


Note that for heapsort we only need the heap property at the end, but our method keeps it after inserting every element. Can we be more efficient? Yes - one way is to build a complete binary tree without the heap property, then recursively heapify the left and right subtrees, and then let the root swap down to the right position. The recursion $T(n) = 2T(n/2)+\lg n$ describes the running time of the latter method: solution is $\Theta(n)$ (exact formula is tricky to guess).

This method can be rewritten to avoid recursion ("bottom-up") and to work in place in the array implementation.

There is still serious research being done on better priority queue implementations.

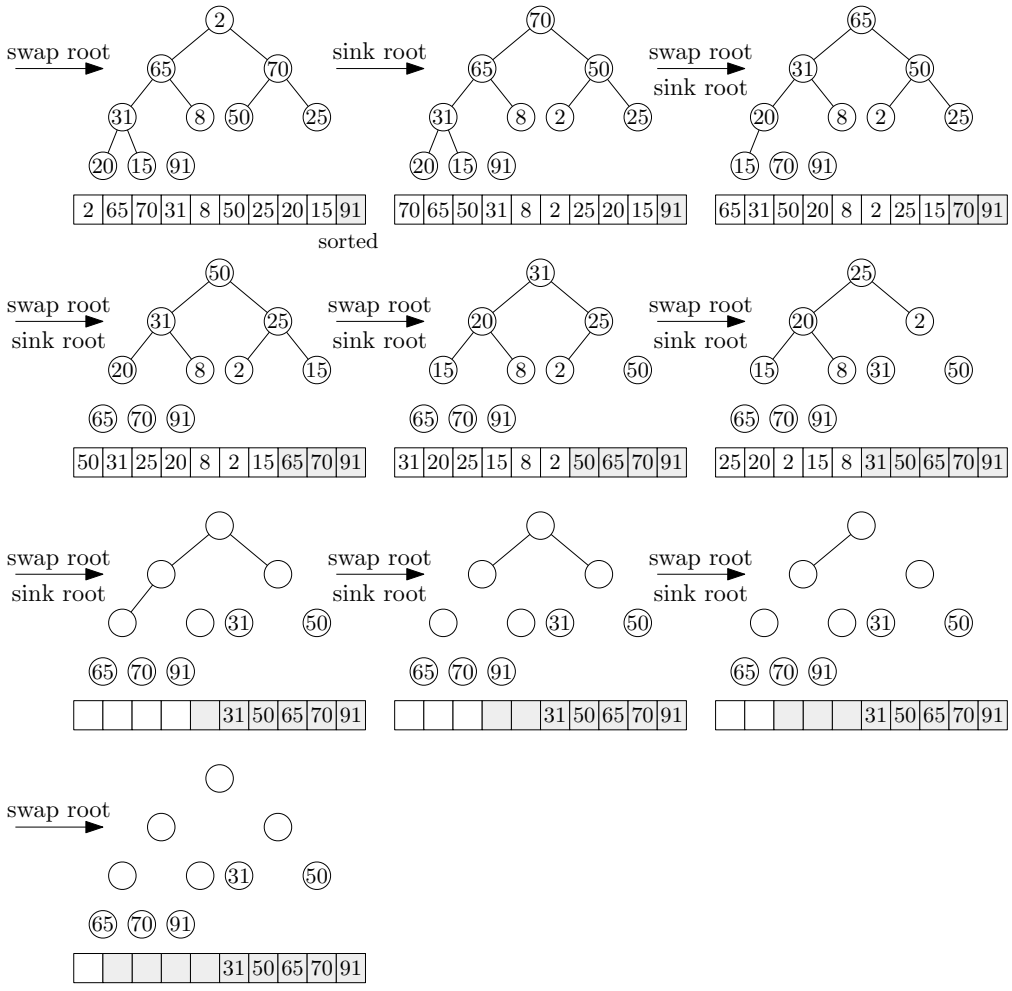
Example 13.7. Heap-building algorithm as in the first while-loop of Algorithm 16.



Algorithm 16 Heapsort.

```
1: function HEAPSORT(array a[0..n − 1])
2:     k ← n − 1                                   ▷ index of last leaf in heap
3:     i ← ⌊(n−1)/2⌋
4:     while i ≥ 0 do
5:         sink(a, i, k)                           ▷ let element sink to its correct place
6:         i ← i − 1
                                                   ▷ heap now built
7:     while k > 0 do
8:         swap(a[k], a[0])
9:         k ← k − 1
10:        sink(a, 0, k)                           ▷ let element at root sink to its correct place
11:    return a                                    ▷ array is now sorted
```

Example 13.8. Complete the last steps of Heapsort.



Lecture 14

Data selection and quickselect

The table below gives a summary of results so far on sorting algorithms. Running times give asymptotic order only. Shellsort analysis depends on the increments used, and is difficult. Quicksort needs a stack of size $\Theta(\log n)$ for the recursion. Selection sort is stable for linked lists. Mergesort is in-place for linked lists.

Method	Worst	Average	Best	Stable?	In-place?
Insertion	n^2	n^2	n	Yes	Yes
Selection	n^2	n^2	n^2	No	Yes
Shellsort	??	??	n	No	Yes
Mergesort	$n \log n$	$n \log n$	$n \log n$	Yes	No
Quicksort	n^2	$n \log n$	$n \log n$	No	Almost
Heapsort	$n \log n$	$n \log n$	$n \log n$	No	Yes

14.1 Selection

We want to consider a related but different topic. Fix r with $1 \leq r \leq n$. We want to find the element with the rth key from an input list (the rth ***order statistic***). This should be easier than sorting!

Example 14.1. Suppose that we build a priority queue and repeatedly extract elements until we get the one we want. What is the asymptotic running time in terms of n and r?

14.2 Quickselect

One good approach is to use the quicksort idea. At each stage we only need to make a recursive call on one half of the array because we know where the pivot is relative to the desired element. This algorithm is called ***Quickselect***.

Algorithm 17 Quickselect.

Require: $0 \le i \le j \le n-1, 1 \le r \le j-i+1$

```
function QUICKSELECT(list a[0..n − 1], int i, int j, int r)
    if i ≤ j then
        p ← a[i]                                   ▷ pivot element
        q ← PARTITION(a, i, j, p)                  ▷ put p in correct position
        if q − i = r − 1 then
            return a[r − 1]
        else if q − i > r − 1 then
            QUICKSELECT(a, i, q − 1, r)            ▷ look in left half
        else
            QUICKSELECT(a, q + 1, j, r − q + i − 1)   ▷ right half
```

Example 14.2. Execute quickselect on the array below. Hint: You have to make one recursive call to the left and one to the right.

8	9	0	14	3	4	11	6

$r = 4 \qquad i = 0 \qquad j = 7$

$q =$

$q - i = \qquad r - 1 =$

$r = \qquad i = \qquad j =$

$q =$

$q - i = \qquad r - 1 =$

$r = \qquad i = \qquad j =$

$q =$

$q - i = \qquad r - 1 =$

14.3 Quickselect analysis

The recurrence for the average number of comparisons has the form

$$E(n) = n + \frac{1}{n}\sum_{i=0}^{n-1} E(p).$$

This has a solution that is $\Theta(n)$, via the same method used for the quicksort recurrence.

Example 14.3. Solve the quickselect recurrence above.

The worst case running time is still quadratic, for the same reason as for quicksort.

14.4 How can we improve quickselect?

There is another divide-and-conquer algorithm (see Wikipedia, "Median of medians") that is worst-case linear, but much more complicated and never used in practice. Note that if it did work well in practice, we would use it to find an optimal (i.e. median) pivot, and thereby achieve $O(n \log n)$ worst case running time for quicksort.

Part III

Analysis of searching

Lecture 15

Searching, binary search trees

15.1 Table ADT

Definition 15.1. A ***table*** is an ADT that supports operations to insert, retrieve and delete an element with given search key. Another name is ***dictionary***.

Tables are used for databases. There are many ways in which a table could be implemented; with an unsorted list, with a sorted list, with a (balanced) binary search tree, with a hash table.

15.2 Sequential search

In a list with no other structure, the only way to find an element is to check each element. This takes time in $\Theta(n)$ for any reasonable implementation (such as array or linked list).

We only need to be able to iterate through the elements in linear time, so even more general structures than lists also allow for this type of search.

15.3 Binary search

In a sorted list where constant time access is possible (such as an array implementation), we can find a key K as follows.

- Start at the middle element y, and recursively go left (right) if $y > K$ (if $y < K$);
- stop if we hit K or search subinterval is empty.

Note that binary search is conceptually easy but surprisingly hard to program correctly even for professionals (see J. Bentley, *Programming Pearls*).

The (worst-case) recurrence for the running time is $T(n) = 1 + T(\lfloor n/2 \rfloor)$, with the solution in $\Theta(\log n)$. The best case is when we find the element on the first try (it is in the middle position).

The execution of this algorithm (looking for all possible keys) can be described by a decision tree called a (static) ***binary search tree***. The number of comparisons required to find the key is the depth of the leaf containing that key.

Algorithm 18 Binary search.

```
1: function BINSEARCH(sorted list a[0..n − 1], key K)
2:     L ← 0
3:     R ← n − 1
4:     while L ≤ R do
5:         m ← ⌊(L + R)/2⌋                          ▷ median index
6:         if A[m] < K then
7:             L ← m + 1
8:         else if A[m] > K then
9:             R ← m − 1
10:        else
11:            return m                              ▷ key present at index m
       return −1                                     ▷ key not present in list
```

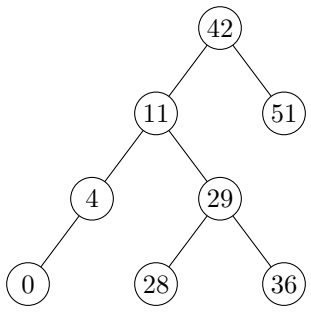
15.4 Binary search trees

Definition 15.2. A binary search tree (BST) is a binary tree with keys stored in nodes, such that the key of each node is ≥ the key of every node in the left subtree and ≤ the key of every node in the right subtree.

A BST implements the Table ADT. To find/insert a key, use binary search as described above. To remove a node, we need more work:

- A node with no children: simply delete.
- A node with only one child: delete the node, connect the child to the parent.
- A node n with two children: find the minimum key K in the right subtree, delete that node, and replace the key of n by K.

Example 15.3. Follow the above steps to delete node 11 in the BST shown.



15.5 Analysis of BST operations

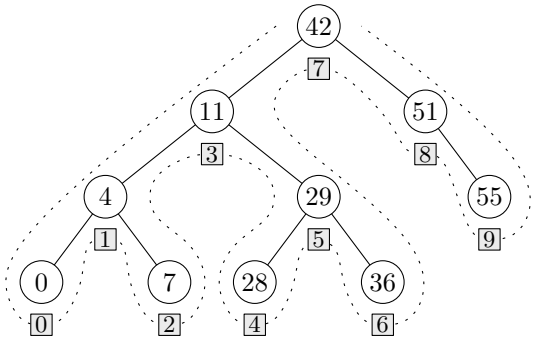
The running time of all basic operations is proportional to the number of nodes visited. In the worst case, finding/removing/inserting take time in $\Theta(h)$ where h is the height of the tree. Unfortunately the height can be as large as $n-1$ in the worst case. Hence we need another idea to guarantee good worst-case performance – we need to **rebalance** BSTs.

Example 15.4. When deleting a node in a BST, what if we use the maximum key in the left subtree instead? Carry this out for node 11 in the BST shown in Example 15.3.

The above example shows that if for deletion we instead choose the maximum key in the left subtree, everything proceeds as expected. However, if we always use the same side for deletion, this can lead to unbalance. Alternating sides, or choosing a side randomly, will more likely result in a more balanced tree.

15.6 Treesort

As we did with heapsort, we can build a binary search tree from our initial list, by inserting elements one at a time. We can recover the sorted list by reading the keys according to an inorder traversal (as shown below). This takes time in $\Theta(n \log n)$.

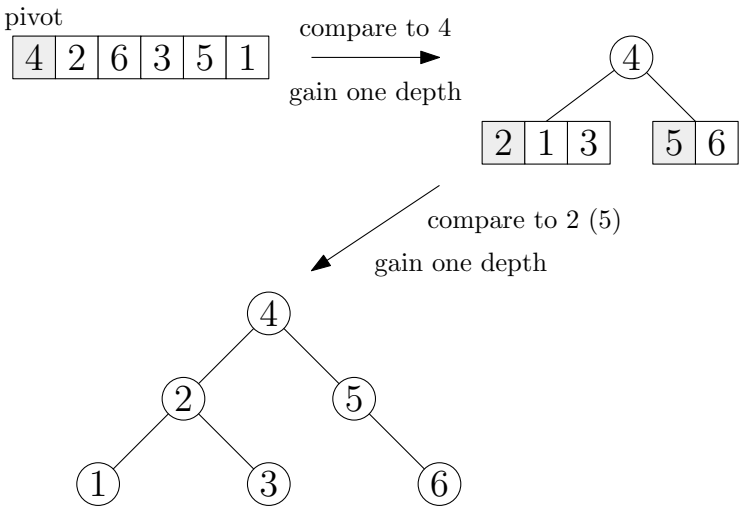


As with heapsort, we can do this in place in an array. This is basically quicksort!

15.7 Relation between Quicksort and BSTs

After choosing the pivot and partitioning, we can represent the array as a binary tree: pivot at the root, left subfile on the left, right subfile on the right. It has the BST property with respect to the sort keys.

Given the above BST describing the execution of quicksort on the array, note that the cost of constructing the tree (measured by key comparisons) is the same as the number of comparisons used by quicksort in sorting the file. This is equal to the internal path length, the sum of all depths of nodes.



Thus the average search cost in a BST built by random insertions is $\Theta(\log n)$.

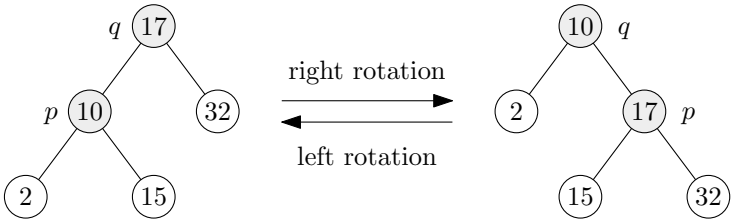
Lecture 16

Self-balancing binary search trees

There are several classes of BSTs (such as AVL, red-black, AA) that perform rebalancing operations at crucial times in order to keep the height close to $\lg n$.

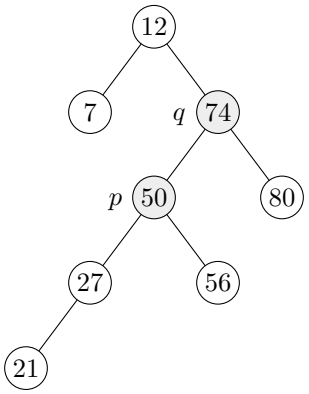
Example 16.1. How many types of self-balancing search trees can you find (e.g. Wikipedia)?

These operations are based on local ***rotations*** of the tree. A right and left rotation of a BST is shown below.



An analysis of the performance is fairly difficult. The average-case height is not really known.

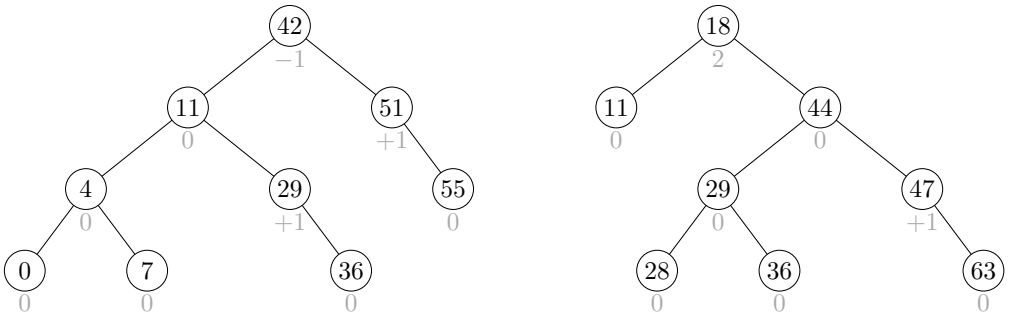
Example 16.2. Perform a right rotation on the BST around the vertices $p = 50$ and $q = 74$ as shown above. Then perform a left rotation on the resulting BST around the vertices 12 and 50.



16.1 AVL trees

AVL trees were the first self-balancing BSTs, introduced in 1962 by Adelson-Velskii and Landis. The **balance of a node** is defined to be the difference between heights of its right subtree and left subtree. AVL trees adjust themselves after each insertion or deletion (via tricky rotations not discussed here – see Wikipedia for gory details) so that each node has balance $0, 1$ or -1.

Example 16.3. An AVL tree on the left and a non-AVL tree on the right with node balances shown below the nodes.



16.2 Height of an AVL tree

Example 16.4. An AVL tree with n nodes has height at most $1.44 \lg n$, about 44% more than optimal.
Proof. Let A_h be the minimum number of nodes in a tree of height h. Clearly $A_0 = 1$, $A_1 = 2$. The subtrees at the root are also AVL trees of height $h-1$ or $h-2$, because of the balance property. Thus $A_h \geq A_{h-1} + A_{h-2} + 1$ (in fact we have equality here (can you see why?)). This is similar to the Fibonacci recurrence, and we know that $A_h \sim C\phi^h$ for some $C > 0$, where $\phi = (1 + \sqrt{5})/2$. Taking logarithms gives $h \sim \lg n / \lg \phi \approx 1.44 \lg n$.

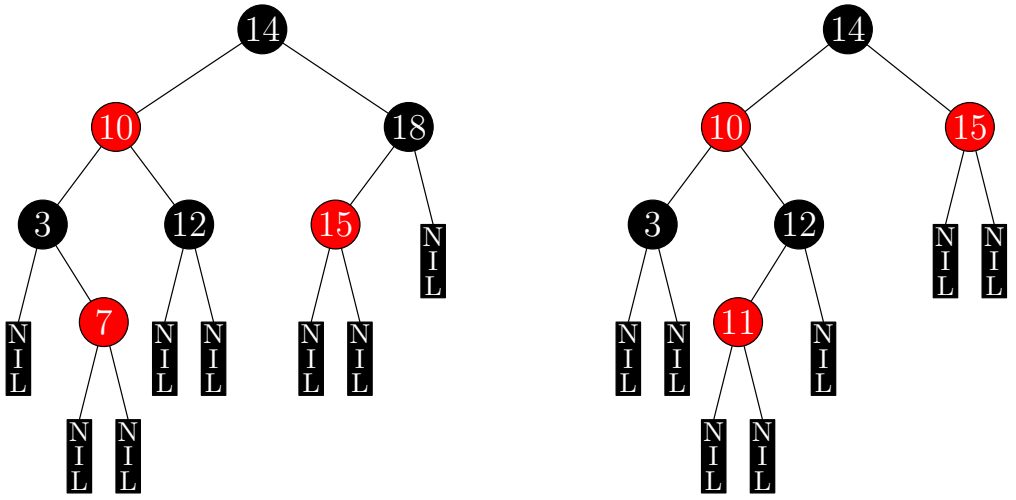
16.3 Red-black trees

Introduced in 1978, these have a balance condition that ensures (via tricky rotations not discussed here – see Wikipedia for gory details) that the maximum depth of a leaf is no more than twice the minimum depth. They do this by colouring each node red or black (note that we are explicitly using external nodes), with the rules:

- the root is black;
- all leaves (external nodes) are black, even when the parent is black;
- both children of every red node are black;
- every path from a node to a leaf contains the same number of black nodes.

We call the number of black nodes the ***black-height*** of a node, and denote it $b(v)$.

Example 16.5. A red-black tree on the left. The BST on the right is not a red-black tree because not every path from 14 to a leaf contains the same number of black nodes.



16.4 Height of a red-black tree

Example 16.6. Prove by induction that the subtree rooted at node v has at least $2^{b(v)-1}$ nodes. Then, using the fact that the height of the tree is at most twice the black-height of the root, show that the worst case height of a red-black tree on n nodes is at most $2\lg(n+1)$.

16.5 Balanced search trees in practice (Dec 2018)

- Java Collections Framework uses red-black trees to implement `TreeMap`.
- C++ uses red-black trees to implement `map`.
- C# uses self-balancing trees to implement `Sorted Dictionary`.
- Python uses doubly-linked lists to implement `OrderedDict`.

Lecture 17

Hashing

Definition 17.1. A ***hash function*** is a function h that outputs an integer value for each key. A ***hash table*** is an array implementation of the table ADT, where each key is mapped via a hash function to an array index.

A hash function should:

- be computable quickly (constant time);
- produce hashed values that are uniformly distributed if keys are drawn uniformly at random;
- produce hash values that are "spread out" for keys that are "close".

A hash function should be deterministic, but appear "random" – in other words it should pass some statistical tests (similar to pseudorandom number generators).

Example 17.2. Suppose that we hash dictionary words by letting $h(w)$ be the address of the last letter of w, converted to lower case (so there are 26 slots in the hash table, numbered $0, \ldots, 25$ to correspond to $a, b, \ldots, z$).
List two words that are very "different" yet have the same hash value. Which of the criteria listed above are satisfied by this hash function?

17.1 Collisions

The number of possible keys is usually much larger than the actual number of keys so we do not want to allocate an array large enough to fit all possible keys. But this means that hash functions are not 1-to-1. When two keys may be mapped to the same index we call it a ***collision***.

We need a ***collision resolution policy*** to prescribe what to do when collisions occur. We assume the first-come-first served model for resolving collisions.

We consider two collision resolution policies:

1. ***Chaining*** uses an "overflow" list for each element in the hash table.
2. ***Open addressing*** uses no extra space. Every element is stored in the hash table. If it gets overfull, we can reallocate space and rehash.

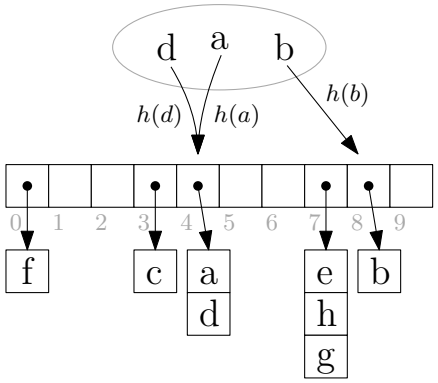
When a collision occurs in open addressing, we can

- ***probe*** nearby for a free position (linear probing, quadratic probing, ...);
- go to a "random" position by using a second-level hash function (***double hashing***).
- We **employ the convention that we always probe to the left**.

17.2 Collision resolution via chaining

- Elements that hash to the same slot are placed in a list. The slot contains a pointer to the head of this list.
- Insertion can then be done in constant time.
- Deletion can be done in constant time with a doubly linked list, for example.
- A drawback is the additional space overhead. Also, the distribution of sizes of lists turns out to be very uneven.

Example 17.3. A hash table with chaining, showing empty chains and chains of length 1, 2 and 3.

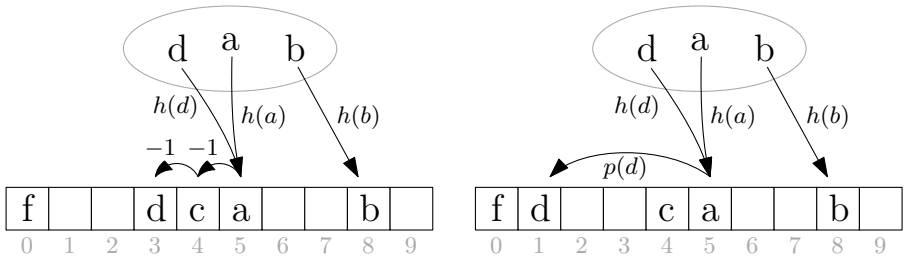


Example 17.4. Using the hash function from Example 17.2 to hash the following 14 words using chaining. What is the length of the longest chain formed? The words are:
abacus, abaci, acrid, grampus, radii, celery, homely, comely, xylitol, animal, seminar, seminal, radar, box

17.3 Collision resolution via open addressing

- Every key is stored somewhere in the array; no extra space is required.
- If a key k hashes to a value $h(k)$ that is already occupied, we probe (look for an empty space).
  - The most common probing method is ***linear probing***, which moves left one index at a time, wrapping around if necessary, until it finds an empty address. This is easy to implement but leads to ***clustering***.
  - Another method is ***double hashing***. Move to the left by a fixed step size t, wrapping around if necessary, until we find an empty address. The difference is that t is not fixed in advance, but is given by a second hashing function $p(k)$.

Example 17.5. The hash table on the left uses linear probing, so d is inserted two left of $h(d) = h(a)$, because $h(d) - 1$ is already occupied. The hash table on the right uses double hashing, so d is moved $p(d)$ to the left of $h(d) = h(a)$.



Example 17.6. If we use the hash function for dictionary words, open addressing with linear probing, and the list of 14 words as above, where does the word "comely" hash to in the table? If instead of linear probing we use double hashing with $p(w)$ defined to be the number of vowels in w, where does "comely" end up?

Linear probing is simple but is prone to *clustering* – large stretches filled entries form making average search times longer.

17.4 Analysis of hashing

Cost is measured by the number of key comparisons or probes.

Insertion has the same cost as an unsuccessful search, provided the table is not full. Thus the average cost of a successful search is the average of all insertions needed to construct the table from empty.

We often use the ***simple uniform hashing*** model. That is, each of the n keys is equally likely to hash into any of the m slots. So we are considering a "balls in bins" model.

If n is much smaller than m, collisions will be few and most slots will be empty. If n is much larger than m, collisions will be many and no slots will be empty. The most interesting behaviour is when m and n are of comparable size. We define the ***load factor*** to be $\lambda := n/m$.

Example 17.7. Compute the load factor for the hash tables shown in Examples 17.3 and 17.5.

Lecture 18

Analysis of hashing

This lecture presents theoretical results, some without proof. The analysis of hashing leads to some interesting and tricky mathematics.

18.1 Analysis of balls in bins

The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m,n)$ where

$$Q(m,n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m}\cdots\frac{m-n+1}{m}.$$

Note that $Q(m,0) = 1, Q(m,n) = 0$ for $n > m$.

For example, $Q(366,180) \approx 0.4487 \times 10^{-23}$ (negligible chance of no collision when 180 balls thrown into 366 boxes) while $Q(366,24) \approx 0.4627$ (more likely than not to have a collision when throwing 24 balls in 366 boxes).

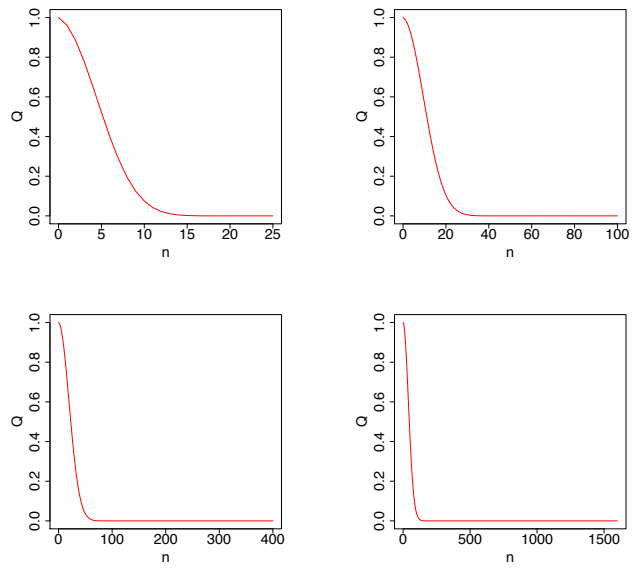


Figure 18.1: Plots of $Q(m,n)$ against n for $m = 25, 100, 400, 1600$

18.2 Analysis of chaining

With chaining, the worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys. The average cost for an unsuccessful search is the average list length, namely λ.

The average cost for a successful search is then

$$\frac{1}{n}\sum_{k=1}^{n}(1+\frac{k-1}{m}) = 1+\frac{n-1}{2m} \approx 1+\frac{n}{2m} = 1+\lambda/2.$$

Thus provided the load factor is kept bounded, basic operations all run in constant time on average.

18.3 Analysis of open addressing

We assume ***uniform hashing***: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a uniformly random permutation of the keys.

Uniform hashing is a theoretical model and cannot be implemented practically but is a good approximation when double hashing is used.

We show in Section 18.4 that the average cost for insertion (unsuccessful search) under uniform hashing is $\Theta(1/(1-\lambda))$ as $m \to \infty$.

Linear probing is not well described by the uniform hashing model (because of the clustering). The average insertion cost is $\Theta(1+1/(1-\lambda)^2)$ (proof omitted).

If the load factor is bounded away from 1, basic operations run in constant time; otherwise performance will be very bad. We should resize as λ grows!

Example 18.1. Sketch the functions $f(\lambda) = 1+1/(1-\lambda)^2$ and $g(\lambda) = 1/(1-\lambda)$

18.4 Unsuccessful search under uniform hashing hypothesis

- Let X be the number of probes taken for an unsuccessful search. Let p_i be the probability that we need to make $i+1$ probes (exactly i probes hit an occupied cell) and let q_i be the probability that we make at least $i+1$ probes.
- Then $E[X] = \sum_i (i+1)p_i = \sum_i q_i$ (why?).
- For $i \geq 1$, we have

$$q_i = \binom{n}{i} / \binom{m}{i} = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+1}{m-i+1} \leq \lambda^i.$$

  The exact value is then

$$E[X] = \frac{1}{\binom{m}{n}} \sum_{j=0}^{n} \binom{(m-n)+j}{m-n} = \cdots = \frac{1+m}{1+(1-\lambda)m}$$

  which is $\Theta(1/(1-\lambda))$ as $\lambda \to 1$. This agrees with intuition.

18.5 Statistics for balls in bins: some facts without proof

- When do we expect the first collision? This is the ***birthday problem***. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables.
- When do we expect all boxes to be nonempty? This is the ***coupon collector*** problem. Answer: After about $m \log m$ balls. It takes a long time to use all lists when chaining.
- What proportion of boxes are expected to be empty when n is $\Theta(m)$? Answer: $e^{-\lambda}$. Many of the lists are just wasted space even for pretty full tables.
- When m is $\Theta(n)$, what is the expected maximum number of balls in a box? Answer: About $(\log n)/(\log \log n)$. Some of the lists may be fairly long.

18.6 Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement `unordered_map`, resizing when $\lambda > 1$, and prime table size.
- C# uses chaining, resizing when $\lambda > 1$, and prime table size.
- Python uses open addressing, resizing when $\lambda > 0.66$, and table size a power of 2.

Lecture 19

Universal hashing

Hash functions must be deterministic, but we do not want to let users know what they are. However, they can often guess. As with quicksort, a malicious user can submit input data that makes the algorithm run an unreasonably long time, which can cause security and performance problems.

Example 19.1. Java String `hashcode` computes the address of a string, say $s = s[0] \cdots s[n-1]$ using integer arithmetic via the formula $s[0] \cdot 31^{n-1} + s[1] \cdot 31^{n-2} + \ldots + s[n-1]$. Can you find two 2-character strings with the same hashcode? Why might this cause a problem in practice?

A nice way to use randomization is to choose the hash function at runtime randomly from a family of hash functions. Ideally, the probability of a collision between two elements is only $1/m$, which is what a random guess would yield, so malicious users cannot force collisions. A family of hash functions that satisfies this is called ***universal***.

Formally, if F is a universal family of hash functions then for any keys $k \neq k'$,

$$\frac{1}{|F|} |\{h \in F : h(k) = h(k')\}| \leq \frac{1}{m}.$$

Example 19.2. Construct the family of hash function $F = \{h_{ab}\}$ by taking a prime $p \geq m$ and for $0 < a < p, 0 \leq b < p$ define

$$h_{ab}(x) = ax + b \mod p \mod m.$$

Prove that this is a universal family.

19.1 Summary of Table implementations

We summarise the table implementations that we have looked at so far in terms of worst and average case.

Table 19.1: Worst case running time (asymptotic order)

Data structure	Insert	Find	Delete	Traverse
Unsorted list (array)	1	n	n	n
Unsorted list (pointer)	1	n	n	n
Sorted list (array)	n	$\log n$	n	n
Sorted list (pointer)	n	n	n	n
Binary search tree	n	n	n	n
Balanced BST	$\log n$	$\log n$	$\log n$	n
Hash table (chaining)	n	n	n	$m+n$

Table 19.2: Average case running time (asymptotic order)

Data structure	Insert	Find	Delete	Traverse
Unsorted list (array)	1	n	n	n
Unsorted list (pointer)	1	n	n	n
Sorted list (array)	n	$\log n$	n	n
Sorted list (pointer)	n	n	n	n
Binary search tree	$\log n$	$\log n$	$\log n$	n
Balanced BST	$\log n$	$\log n$	$\log n$	n
Hash table (chaining)	λ	λ	λ	$m+n$

Example 19.3. If we want to print out all items of a table in sorted order, what is the best data structure? If we do not mind an occasional long wait and mostly do lookups, which is best? If we do a lot of insertions and deletions and few lookups, and are risk-averse, which is best?

Part IV

The graph abstract data type

Lecture 20

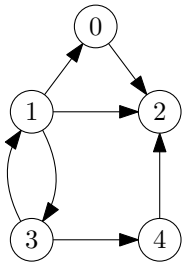
Graph definitions

Graphs are important and general mathematical objects that are widely used in theory and practice. They distill the basic idea of a relationship among a set of objects. Informally we can think of a graph as a collection of dots (the set of objects) with lines connecting them (describing the relationship). The lines can be either directed (arrows) or undirected.

We are interested in the algorithmic aspects of graph theory ("how can we do it efficiently and systematically?"). To talk about this precisely, we must start with precise definitions.

Definition 20.1. A ***digraph*** $G = (V, E)$ is a finite nonempty set V of ***nodes*** together with a (possibly empty) set E of ordered pairs of nodes of G called ***arcs***. Digraph stands for **di**rected **graph**.

Example 20.2. For the digraph shown, write down the sets V and E.

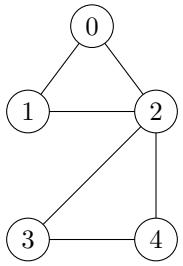


$V =$

$E =$

Definition 20.3. A ***graph*** $G = (V, E)$ is a finite nonempty set V of ***vertices*** together with a (possibly empty) set E of unordered pairs of vertices of G called ***edges***. Note that the singular of vertices is ***vertex***.

Example 20.4. For the graph shown, write down the sets V and E.



$V =$

$E =$

Some notes on graphs vs digraphs.

- In order to save writing "(di)graph" too many times, we treat the digraph as the fundamental concept.
- When we say something about digraphs, nodes and arcs, it is understood to also hold for graphs, vertices and edges unless explicitly stated otherwise.
- However, if we talk about graphs, vertices and edges, our statement is not necessarily true for digraphs.
- Some authors use "undirected graph" to mean graph and use the term "graph" to mean what we call a directed graph. We always use digraph and graph.
- E is a set so there are no multiple arcs between a pair of nodes.
- An arc that begins and ends at the same node is called a ***loop***. We make the convention that **loops are not allowed in our digraphs**.
- For a digraph G we may denote the node set $V(G)$ and arc set $E(G)$ for clarity.
- A graph can be viewed as a digraph where every unordered edge $\{u, v\}$ is replaced by two directed arcs (u, v) and (v, u). This works in most instances and has the advantage of allowing us to consider only digraphs.

Definition 20.5. If $(u, v) \in E$ (that is, if there is an arc going from u to v) we say that v is ***adjacent*** to u, that v is an ***out-neighbour*** of u, and that u is an ***in-neighbour*** of v. In an (undirected) graph G, if $\{u, v\} \in E$, then u is a ***neighbour*** of v and v is a neighbour of u.

Example 20.6. In the digraph in Example 20.2, find all in-neighbours of node 2 and all out-neighbours of node 0.

Definition 20.7. The ***order*** of a digraph $G = (V, E)$ is $|V|$, the number of nodes. The ***size*** of G is $|E|$, the number of arcs. We usually use n to denote $|V|$ and m to denote $|E|$.

For a given order n, the size m can be as low as 0 (a digraph consisting of n nodes and no arcs) and as high as $n(n-1)$ (each node can point to each other node; recall that we do not allow loops).

Definition 20.8. If m is toward the low end, the digraph is called ***sparse***, and if m is toward the high end, then the digraph is called ***dense***. These terms are obviously very informal. For our purposes we will call a class of digraphs sparse if m is $O(n)$ and dense if m is $\Omega(n^2)$.

Definition 20.9. A ***walk*** in a digraph G is a sequence of nodes $v_0\, v_1\, \ldots\, v_l$ such that, for each i with $0 \leq i < l$, (v_i, v_{i+1}) is an arc in G.

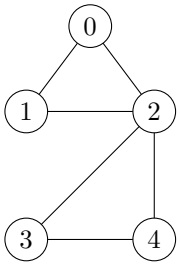
The ***length*** of the walk $v_0\, v_1\, \ldots\, v_l$ is the number l (that is, the number of arcs involved).

A ***path*** is a walk in which no node is repeated.

A ***cycle*** is a walk in which $v_0 = v_l$ and no other nodes are repeated.

In a graph, a walk of the form $u\, v\, u$ – going back and forth along the same edge – is not considered a cycle. A cycle in a graph must be of length at least 3. Note that a walk and a path can have length 0.

Example 20.10. For the graph on the left the following sequences of vertices are classified as being walks, paths, or cycles. Complete the table.



vertex sequence	walk?	path?	cycle?
0 3 2	no	no	
0 1 2 3 4		yes	no
0 1 2 0	yes		yes
0 1 0	yes	no	
1 2 3 4 2 0			
3	yes		

Example 20.11. Show that if there is a walk from u to v, then we can find a path from u to v.

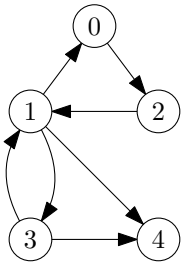
Definition 20.12. In a graph, the ***degree*** of a vertex v is the number of edges meeting v. In a digraph, the ***outdegree*** of a node v is the number of out-neighbours of v, and the ***indegree*** of v is the number of in-neighbours of v.

A node of indegree 0 is called a ***source*** and a node of outdegree 0 is called a ***sink***.

Definition 20.13. The ***distance*** from u to v in G, denoted by $d(u, v)$, is the number of arcs on a shortest path from u to v. If no path from u to v exists, the distance is undefined (or $+\infty$).

For graphs, we have $d(u, v) = d(v, u)$ for all vertices u, v.

Example 20.14. Give the following distances for the digraph on the left.



$d(0,1) =$ $d(0,4) =$

$d(2,3) =$ $d(3,2) =$

$d(2,2) =$ $d(4,0) =$

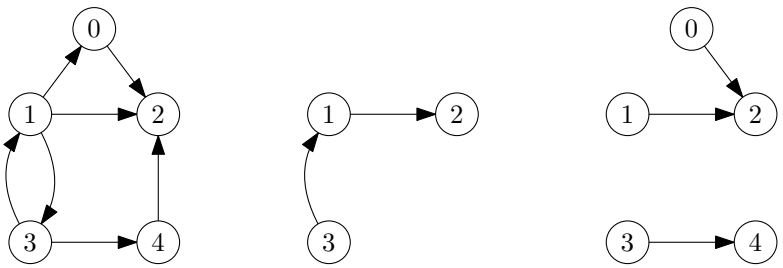
Why are the values $d(4, v)$ not defined unless $v = 4$?

20.1 Creating new digraphs from old ones

There are several ways to create new digraphs from old ones. One way is to delete nodes and arcs in such a way that the resulting object is still a digraph (no arcs missing endpoints).

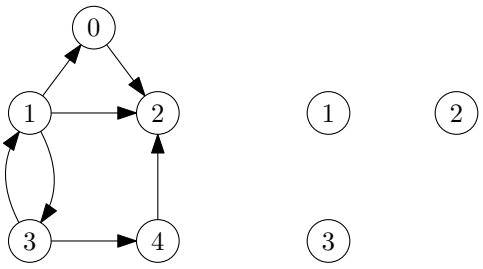
Definition 20.15. A ***subdigraph*** of a digraph $G = (V, E)$ is a digraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$. A ***spanning*** subdigraph is one with $V' = V$; that is, it contains all nodes.

Example 20.16. A digraph (left) with a subdigraph (middle) and a spanning subdigraph (right).



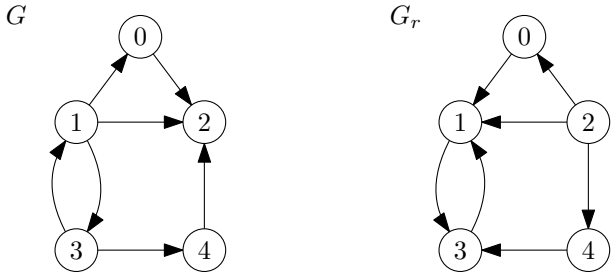
Definition 20.17. The subdigraph ***induced*** by a subset V' of V is the digraph $G' = (V', E')$ where $E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$.

Example 20.18. For the digraph shown on the left, draw the subdigraph induced by $\{1, 2, 3\}$ on the right.



Definition 20.19. The ***reverse digraph*** of the digraph $G = (V, E)$, is the digraph $G_r = (V, E')$ where $(u, v) \in E'$ if and only if $(v, u) \in E$.

Example 20.20. Digraph G and its reverse G_r. We simply reverse all the arrows.

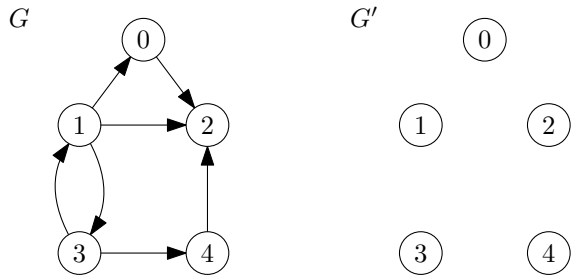


It is sometimes useful to ignore the direction of arcs in a digraph to find the associated 'underlying graph'.

Definition 20.21. The ***underlying graph*** of a digraph $G = (V, E)$ is the graph $G' = (V, E')$ where $E' = \{\{u, v\} \mid (u, v) \in E\}$.

Note that the underlying graph does not have multiple edges even when there are arcs (u, v) and (v, u). In that case, only one edge joins u and v in the underlying graph G'.

Example 20.22. Draw the underlying graph G' of the digraph G.



Definition 20.23. We can combine two or more digraphs $G_1, G_2, \ldots G_k$ into a single graph where the vertices of each G_i are completely disjoint from each other and no arc goes between the different G_i. The constructed graph G is called the ***graph union***, where $V(G) = V(G_1) \cup V(G_2) \cup \ldots \cup V(G_k)$ and $E(G) = E(G_1) \cup E(G_2) \cup \ldots \cup E(G_k)$.

Lecture 21

Graph data structures

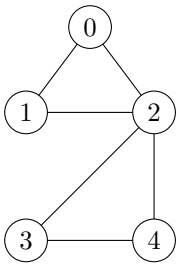
When representing a digraph in a computer, we assume that it has nodes given in a fixed order with the **convention that the nodes are labelled** $0, 1, \ldots, n-1$.

Definition 21.1. Let G be a digraph of order n. The ***adjacency matrix*** of G is the $n \times n$ Boolean matrix (often encoded with 0's and 1's) such that entry (i, j) is true if and only if there is an arc from the node i to node j.

Definition 21.2. For a digraph G of order n, an ***adjacency lists*** representation is a sequence of n sequences, $L_0, \ldots, L_{n-1}$. Sequence L_i contains all nodes of G that are out-neighbours of node i.

In the adjacency lists representation, L_i may or may not be sorted in order of increasing node number. Our **convention is to sort them whenever convenient**. Many implementations do **not** enforce this convention.

Example 21.3. A graph and its adjacency matrix and adjacency list.

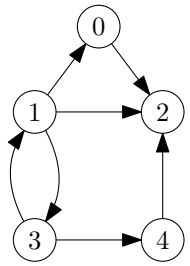


$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

0	1	2		
1	0	2		
2	0	1	3	4
3	2	4		
4	2	3		

Notice that the number of 1's in a row in the adjacency matrix is the outdegree of the corresponding node, while the number of 1's in a column is the indegree.

Example 21.4. A digraph and its adjacency matrix and adjacency list.



$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

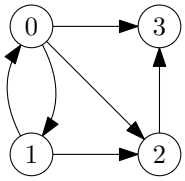
0	2
1	0 2 3
2	
3	1 4
4	2

An empty sequence occurs in the adjacency list where a node has no out-neighbours (for example, sequence 2).

Sometimes the node labels in adjacency lists are omitted, so the digraph in Example 21.4 would be written as:

2
0 2 3
1 4
2

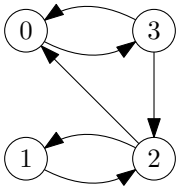
Example 21.5. Give the adjacency matrix of the digraph below.



Draw the digraph corresponding to the given adjacency matrix.

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Example 21.6. Give the adjacency lists of the digraph below.



Draw the digraph corresponding to the given adjacency lists.

0	1 2
1	3
2	
3	0 1 2

21.1 Representing multiple graphs in a single file

We can store several digraphs one after the other in a single file as follows:

- We have a single line giving the order at the beginning of each digraph.
- If the order is n then the next n lines give the adjacency matrix or adjacency lists representation of the digraph. Node labels are omitted.
- The end of the file is marked with a line denoting a digraph of order 0.

Example 21.7. The two digraphs on the left could be put in a single file:

0	2
1	0
2	1

0	1 2
1	3
2	
3	0 1 2

```
3
2
0
1
4
1 2
3

0 1 2
0
```

21.2 Using other structures to represent graphs

Other specialized digraph representations may be used to take advantage of special structure in a family of digraphs for improved storage or access time. For such specialized purposes they may be better than adjacency matrices or lists.

For example, trees can be stored more efficiently. We saw in Lecture 13 how to store a complete binary tree in an array. Here is way of storing a general tree in an array.

- A general rooted tree of n nodes can be stored in array `pred` of size n.
- `pred`$[i]$ is the parent of node i.
- The root has no parent, so assign it `null` or -1 if we number nodes from 0 to $n-1$ in the usual way.
- This is a form of adjacency lists, using in-neighbours instead of out-neighbours.

Example 21.8. Draw the tree represented by the array

$$\texttt{pred} = [-1, 0, 0, 1, 2, 2, 2, 3].$$

21.3 Implementation of digraph ADT

An adjacency matrix is simply a matrix which is an array of arrays.

Adjacency lists are a list of lists. There are several ways in which a list can be implemented, for example by an array, or singly- or doubly-linked lists using pointers. These have different properties, for example, accessing the middle element is $\Theta(1)$ for an array but $\Theta(n)$ for a linked list. Searching for a value that may or may not be in the list requires sequential search and takes $\Theta(n)$ time in the worst case. We do not consider other data structures (e.g. heaps) that can be used to represent lists.

21.4 Complexity of basic digraph operations

The basic operations we consider are checking for the existence of an arc between two nodes, finding the outdegree of a node, finding the indegree of a node, adding

an arc between two nodes, deleting an arc between two nodes, adding a node, and deleting a node.

For the two data structures, consider the steps we need to carry out various basic operations and the cost of all steps.

Example 21.9. Compare the matrix and lists data structures for checking whether arc (i, j) exists.
Adjacency matrix representation: We need to check whether element (i, j) is 1. This requires accessing an array element twice, to first find the ith array then its jth element. Each array access is in $\Theta(1)$ so overall it is in $\Theta(1)$.
Adjacency lists representation: We need to search for j in list i. The complexity then depends on the length of list i. List i is length d where d is the outdegree of node i so searching for j is in $\Theta(d)$. But how large is d? Even when the graph is sparse, it could still be the case that d is $O(n)$, though typically in a sparse graph d is $O(1)$. In a dense graph d is $O(n)$.

Example 21.10. Compare the matrix/lists data structures for deleting a node.
Adjacency matrix representation:

Adjacency lists representation:

Table 21.1 shows the steps required and Table 21.2 the time required for basic graph operations when using adjacency matrix or lists representations. Performance for the adjacency list representation is based on using doubly linked lists.

Table 21.1: Steps required to perform basic digraph operations by data structure.

Operation	**Adjacency matrix**	**Adjacency lists**
arc (i, j) exists?	is entry (i, j) 0 or 1	find j in list i
outdegree of i	scan row, count 1's	size of list i
indegree of i	scan column, count 1's	for $j \neq i$, find i in list j
add arc (i, j)	change entry (i, j)	insert j in list i
delete arc (i, j)	change entry (i, j)	delete j from list i
add node	create new row and column	add new list at end
delete node i	delete row/column i shuffle other entries	delete list i for $j \neq i$, delete i from list j

Table 21.2: Comparative worst-case performance of adjacency matrices and lists.

Operation	**Adjacency matrix**	**Adjacency lists**
arc (i, j) exists?	$\Theta(1)$	$\Theta(d)$
outdegree of i	$\Theta(n)$	$\Theta(1)$
indegree of i	$\Theta(n)$	$\Theta(n + m)$
add arc (i, j)	$\Theta(1)$	$\Theta(1)$
delete arc (i, j)	$\Theta(1)$	$\Theta(d)$
add node	$\Theta(n)$	$\Theta(1)$
delete node i	$\Theta(n^2)$	$\Theta(n + m)$

Using lists, apparently similar problems like finding the outdegree ($\Theta(1)$) and indegree ($\Theta(n + m)$) have very different time complexity.

Clearly finding all indegrees would be slow if we just used multiple calls to a method designed for getting the indegree of a single node.

Example 21.11. Show that the sorted adjacency lists representation of the reverse digraph of G can be found in time $\Theta(n+m)$ given the sorted adjacency lists of G. Also show how this can be used to find indegrees for all nodes of G in time $\Theta(n+m)$.

21.5 Space requirements

The adjacency matrix representation requires $\Theta(n^2)$ storage as we simply need a matrix of n^2 bits.

At first guess we might say adjacency lists require $\Theta(n+m)$ storage, since we need n lists and the combined length of all the lists is m. But node numbers require more than one bit of storage each; the number k uses about $\Theta(\log k)$ bits. The average entry in a list is $\frac{n}{2}$, so the total space requirement is more like $\Theta(n + m\log(n))$

Example 21.12. What is the storage requirement for a complete digraph on n nodes, that is, a digraph where every possible arc occurs?

For sparse digraphs, lists use less space than a matrix, whereas for dense digraphs the space requirements are comparable.

The representation that is best will depend on the application and we cannot make general rules. We will mostly use adjacency lists, which are clearly superior for many common tasks and generally better for sparse digraphs.

Part V

Graph traversals and applications

Lecture 22

Graph traversal algorithms

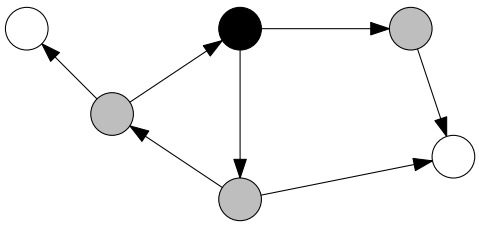
Traversals involve visiting each node of a digraph in a systematic way following only the arcs in the digraph. This is a very common task when dealing with digraphs and we will cover several applications in later lectures.

22.1 The general graph traversal algorithm

All graph traversal algorithms have the same basic structure that relies on keeping track of which nodes have been visited and whether they may be adjacent to nodes that have not yet been visited. We use a system of three colours:

- ***White nodes*** have not yet been visited;
- ***Grey nodes*** (or **frontier nodes**) have been visited but may have out-neighbours that are white;
- ***Black nodes*** have been visited and all their out-neighbours have been visited too (so are not white).

Example 22.1. Node states during a digraph traversal.



The traversal algorithm which visits all nodes in a digraph is loosely described as follows:

- All nodes are white to begin with.
- A starting white node is chosen and turned grey.
- A grey node is chosen and its out-neighbours explored.
- If any out-neighbour is white, it is visited and turned grey. If no out-neighbours are white, the grey node is turned black.
- The process of choosing grey nodes and exploring neighbours is continued until all nodes reachable from the initial node are black.

- If any white nodes remain in the digraph, a new starting node is chosen and the process continues until all nodes are black.

We keep track of the order in which nodes are visited by recording which arc was followed when a node first turned from white to grey. As we will see, this creates a sub-digraph of the original digraph which has a tree structure.

The traversal algorithm is formalised in the pseudocode of Algorithms 19 and 20. Algorithm 19 (`traverse`) initialises the process and is tasked with finding white nodes to start from. Most of the work is done in Algorithm 20 (`visit`) which takes a starting node and explores the portion of the subgraph reachable from that node.

Algorithm 19 Basic graph traversal main routine: `traverse`.

```
1:  function TRAVERSE(digraph G)
2:      array colour[0..n − 1]                          ▷ records node colour
3:      array pred[0..n − 1]                  ▷ records path of traversal as a tree
4:      for u ∈ V(G) do                                  ▷ make all nodes white
5:          colour[u] ← WHITE
6:      for s ∈ V(G) do                                    ▷ find a white node
7:          if colour[s] == WHITE then
8:              visit(s)                                  ▷ start traversal from s
        return pred
```

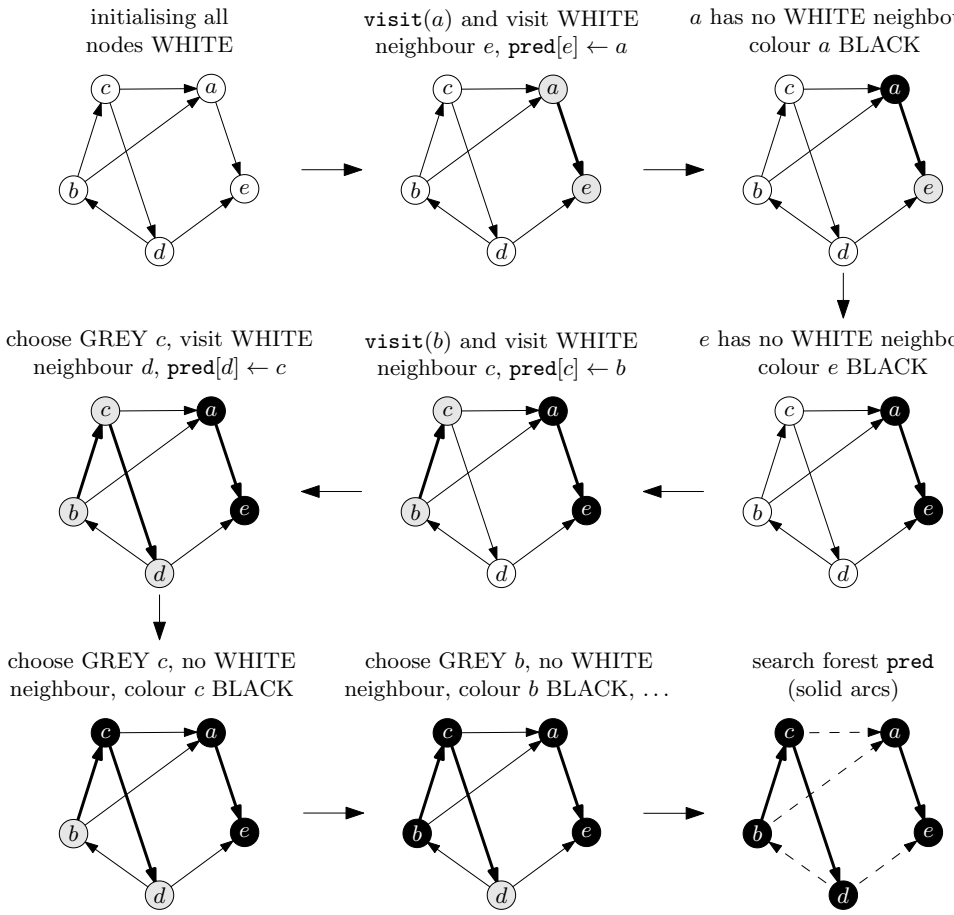
Algorithm 20 Basic graph traversal subroutine: `visit`.

```
1:  function VISIT(node s of digraph G)
2:      colour[s] ← GREY
3:      pred[s] ← null                          ▷ make start node the root of the tree
4:      while there is a GREY node do          ▷ continues until all nodes are black
5:          choose a GREY node u
6:          if u has a WHITE neighbour then
7:              choose a WHITE neighbour, v
8:              colour[v] ← GREY          ▷ v is visited for first time, so turns grey
9:              pred[v] ← u          ▷ make u the parent of v in the traversal tree
10:         else
11:             colour[u] ← BLACK      ▷ u has no white neighbours so done with u
```

A call to `visit` creates a subdigraph of G that is a tree: the nodes are precisely the black nodes (all nodes reachable from the initial node), and the arcs are those arcs followed when we found a white neighbour of a grey node.

Each white node chosen in `traverse` is the root of a tree in the call to `visit`. Eventually, we obtain a set of disjoint trees spanning the digraph (that is, it includes all the nodes of the digraph), which we call the ***search forest***. The search forest is returned by `traverse` in the array `pred`.

Example 22.2. Traversal of a digraph. Visiting a white vertex turns it grey.



22.2 Classifying arcs after a traversal

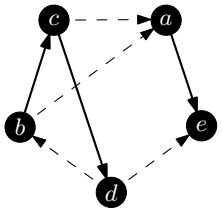
It helps in analysing the traversal algorithm to classify the arcs of G based on their relationships in the search forest.

Definition 22.3. Suppose we have performed a traversal of a digraph G, resulting in a search forest F. Let $(u, v) \in E(G)$ be an arc. Then (u, v) is a

- ***tree arc*** if it belongs to one of the trees of F;
- or, if it is not a tree arc, it is a
  - ***forward arc*** if u is an ancestor of v in F;
  - ***back arc*** if u is a descendant of v in F;
  - ***cross arc*** if neither u nor v is an ancestor of the other in F.

A cross arc may join two nodes in the same tree or point from one tree to another in the search forest. Tree, forward and back arcs require that u and v be in the same tree.

Example 22.4. Classify the arcs according to the definitions above.



(a, e) is a

(d, b) is a

(c, a) is a

The following theorem collects all the basic facts we need for proofs in later sections. The proofs are simple and can be done as exercises.

Theorem 22.5. Suppose that we have carried out `traverse` on G, resulting in a search forest F. Let $v, w \in V(G)$.

- Let T_1 and T_2 be different trees in F and suppose that T_1 was explored before T_2. Then there are no arcs from T_1 to T_2.
- Suppose that G is a graph. Then there can be no edges joining different trees of F.
- Suppose that v is visited before w and w is reachable from v in G. Then v and w belong to the same tree of F.
- Suppose that v and w belong to the same tree T in F. Then any path from v to w in G must have all nodes in T.

Example 22.6. Assuming the first statement of Theorem 22.5, prove the third statement.

22.3 Running time for the general traversal algorithm

The generality of our traversal algorithm makes its exact running time impossible to determine. It depends on how one chooses the next grey node u and its white neighbour v. Some schemes for choosing u and v can be complex and depend on n. The schemes we consider here are simple and constant time.

- The initialization of the array `colour` takes time $\Theta(n)$ so `traverse` is in $\Theta(n+t)$, where t is the total time taken by all the calls to `visit`.
- We execute the while-loop of `visit` in total $\Theta(n)$ times since every node must eventually move from white through grey to black.
- The time taken in choosing grey nodes is $\Theta(n)$.
- The time taken to find a white neighbour involves examining each neighbour of u and checking whether it is white, then applying a selection rule.
- The total time in choosing white neighbours is in $\Theta(m)$ if adjacency lists are used and is in $\Omega(n^2)$ if an adjacency matrix is used.
- So the running time of `traverse` is $\Theta(n + (n + m)) = \Theta(n + m)$ if adjacency lists are used, and $\Theta(n + n^2) = \Theta(n^2)$ if the adjacency matrix format is used.

So for simple selection rules and assuming a sparse input digraph, the adjacency list format seems preferable. But if more complex selection rules are used, for example, choosing a single grey node is of order n while choosing a single white node is still constant time, then the running time is asymptotically $\Theta(n^2)$ regardless of the data structure, so using the adjacency matrix is not clearly ruled out.

Lecture 23

Depth-first search (DFS)

Traversal algorithms differ in the rules for choosing the next grey and next white node, with different rules leading to very different results.

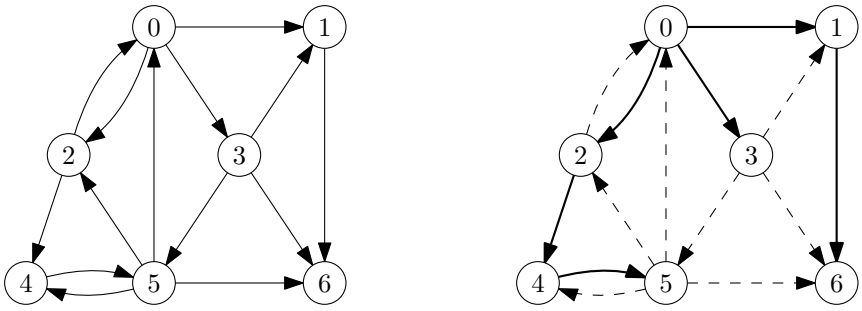
Definition 23.1. In ***depth-first search*** (DFS) the new grey node chosen is the one that has been grey for the **shortest** time.

DFS takes us away from the root node as quickly as possible. If the first visited neighbour of the root has a neighbour, we immediately visit that neighbour. Then, if that has a neighbour, we visit that and so on, thus "deeply" searching as far away from the root as possible. We backtrack as little as possible before continuing away from the root again. The root is the last node to turn black.

There still remains the choice of which white neighbour of the chosen grey node to visit. It does not matter what choice is made but our **convention is to choose the one with lowest index** (recall that nodes have indices $0, \ldots, n-1$).

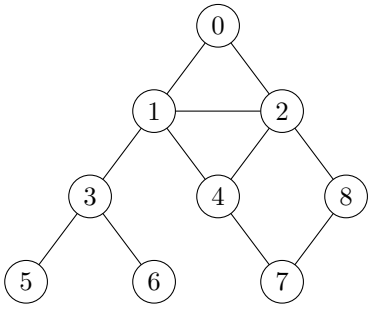
These choices can be made in constant time so that the running time is $\Theta(n+m)$ (assuming adjacency lists) and DFS is linear in the size + order of a digraph.

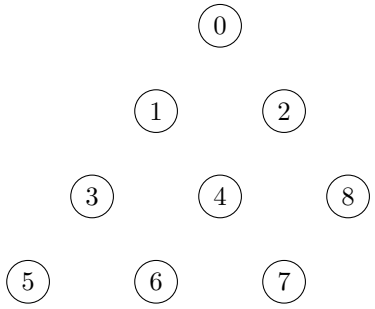
Example 23.2. A digraph and its DFS search tree, rooted at node 0. The dashed arcs indicate the original arcs that are not part of the DFS search tree.



Find a tree arc, cross arc, forward arc and back arc (or say if that type of arc does not exist for that traversal).

Example 23.3. Use the nodes on the right to draw the search tree you obtain by running DFS on the graph on the left, starting at vertex 0. Use dashed edges to indicate edges that are not arcs in the search tree.





Find a tree arc, cross arc, forward arc and back arc (or say if that type of arc does not exist for that traversal).

For a general digraph, and unlike the examples given here, all nodes may not be reachable from the starting node. In this case, we get a search forest where the number of trees in the forest is the number of calls made to `DFSvisit`.

23.1 Pseudocode for depth-first search

The "last in, first out" order of choosing grey nodes in DFS is mimicked by a **stack** data structure. We thus store grey nodes in a stack as they are discovered.

The pseudocode for `DFS` and `DFSvisit` is in Algorithms 21 and 22.

- We loop through the nodes adjacent to the chosen grey node u, and as soon as we find a white one, we add it to the stack.
- We also introduce a variable `time` and keep track of the time a node changes colour.
- The array `seen` records the time a node turns grey.
- The array `done` records the time a node turns black.

Algorithm 21 Depth-first search algorithm.

```
1:  function DFS(digraph G)
2:      stack S
3:      array colour[0..n − 1], pred[0..n − 1], seen[0..n − 1], done[0..n − 1]
4:      for u ∈ V(G) do
5:          colour[u] ← WHITE; pred[u] ← null                    ▷ initialise arrays
6:      time ← 0                    ▷ time will increment at every node colour change
7:      for s ∈ V(G) do
8:          if colour[s] = WHITE then                             ▷ find a WHITE node
9:              DFSvisit(s)                                       ▷ start traversal from s
10:     return pred, seen, done
```

Algorithm 22 Depth-first visit algorithm.

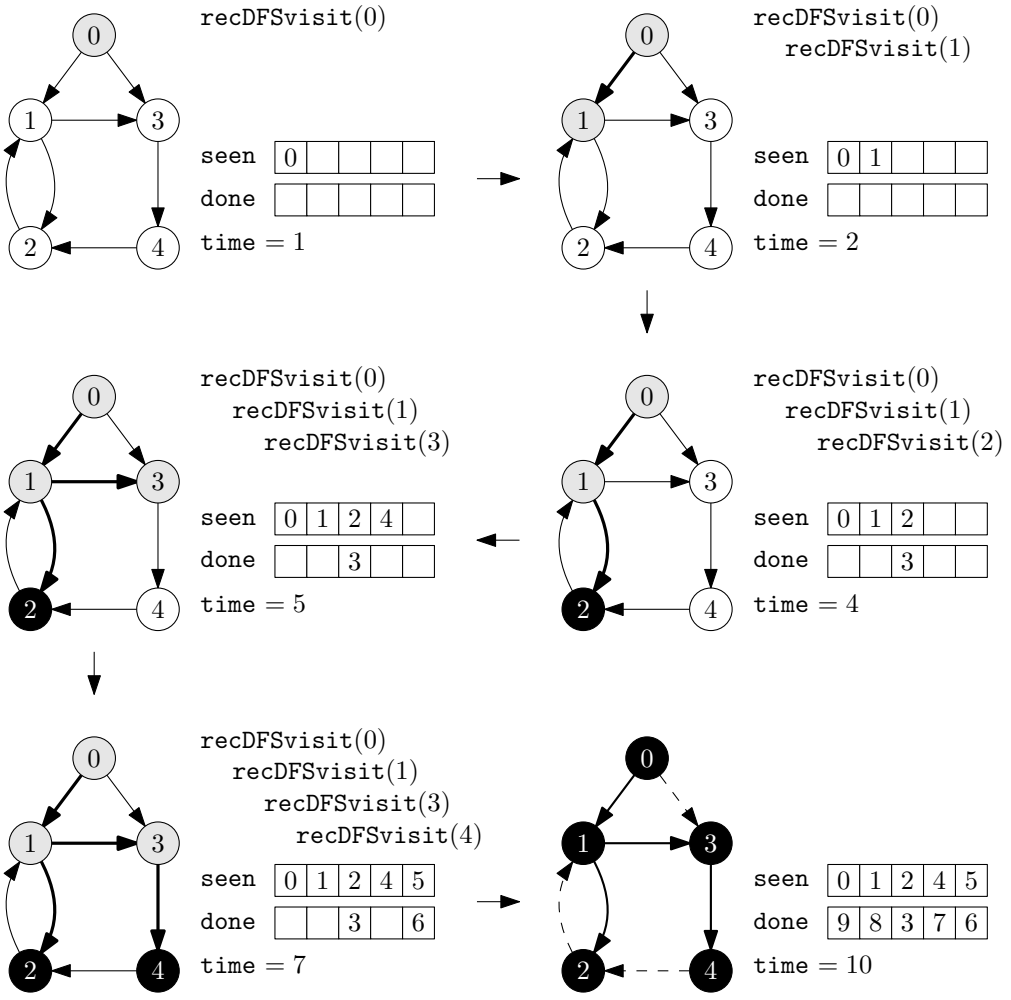
```
1:  function DFSVISIT(node s)
2:      colour[s] ← GREY
3:      seen[s] ← time; time ← time +1
4:      S.insert(s)                                               ▷ put s in stack
5:      while not S.isEmpty() do
6:          u ← S.peek()                                          ▷ get node at front of stack
7:          if there is a neighbour v with colour[v] = WHITE then
8:              colour[v] ← GREY; pred[v] ← u                     ▷ visit neighbour, update tree
9:              seen[v] ← time; time ← time +1                    ▷ record and increment time
10:             S.insert(v)                                       ▷ put neighbour in stack
11:         else
12:             S.delete()                                        ▷ delete node from stack
13:             colour[u] ← BLACK                                 ▷ colour it done
14:             done[u] ← time; time ← time +1                    ▷ record and increment time
```

Given the relationship between stacks and recursion, we can replace the `DFSvisit` procedure by a recursive version `recursiveDFSvisit`:

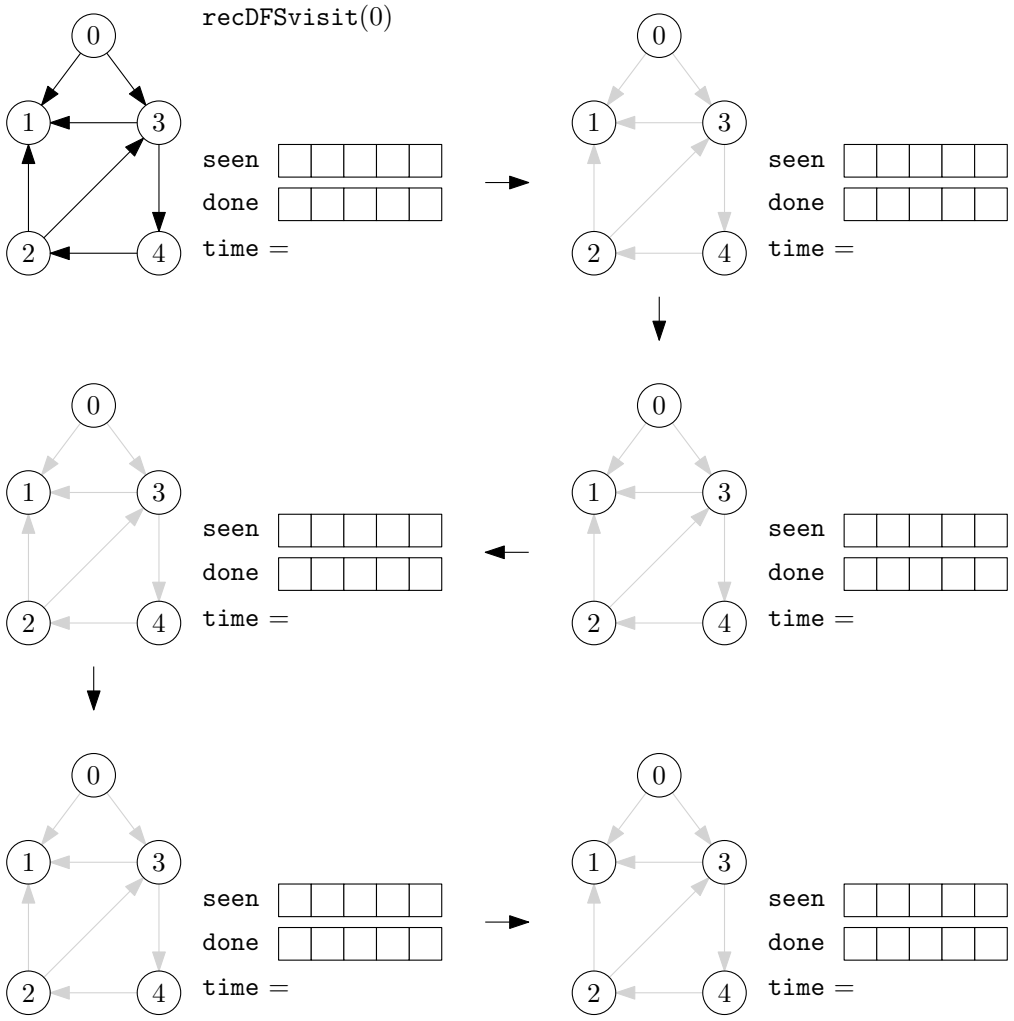
Algorithm 23 Recursive DFS visit algorithm.

```
1:  function RECURSIVEDFSVISIT(node s)
2:      colour[s] ← GREY
3:      seen[s] ← time; time ← time +1
4:      for each v adjacent to s do
5:          if colour[v] = WHITE then                             ▷ each unvisited neighbour
6:              pred[v] ← s                                       ▷ update tree
7:              recursiveDFSvisit(v)                              ▷ now visit neighbour
8:      colour[s] ← BLACK                   ▷ visted all white neighbours so done
9:      done[s] ← time; time ← time +1
```

Example 23.4. DFS with `recursiveDFSvisit`.



Example 23.5. Execute DFS with `recursiveDFSvisit` and following Example 23.4 fill out the values for `seen` and `done`, give the current `time`, and highlight the DFS search tree after each step.



23.2 DFS useful results and facts

Theorem 23.6. Suppose that we have performed DFS on a digraph G, resulting in a search forest F. Let $v, w \in V(G)$ and suppose that $\texttt{seen}[v] < \texttt{seen}[w]$.

- If v is an ancestor of w in F, then

$$\texttt{seen}[v] < \texttt{seen}[w] < \texttt{done}[w] < \texttt{done}[v].$$

- If v is not an ancestor of w in F, then

$$\texttt{seen}[v] < \texttt{done}[v] < \texttt{seen}[w] < \texttt{done}[w].$$

Note that this result rules out the timestamps $\texttt{seen}[v] < \texttt{seen}[w] < \texttt{done}[v] < \texttt{done}[w]$.

All four types of arcs in our search forest classification can arise with DFS. The different types of arcs can be easily distinguished while the algorithm is running or by looking at the timestamps `seen` and `done`.

Example 23.7. Explain how to determine, at the time when an arc is first explored by `DFS`, whether it is a tree-, back-, forward- or cross-arc.

Example 23.8. Suppose that we have performed `DFS` on a digraph G. Let $(v, w) \in E$. The following statements are true. Prove the first statement.

- (v, w) is a tree or forward arc if and only if

$$\texttt{seen}[v] < \texttt{seen}[w] < \texttt{done}[w] < \texttt{done}[v];$$

- (v, w) is a back arc if and only if

$$\texttt{seen}[w] < \texttt{seen}[v] < \texttt{done}[v] < \texttt{done}[w],$$

- (v, w) is a cross arc if and only if

$$\texttt{seen}[w] < \texttt{done}[w] < \texttt{seen}[v] < \texttt{done}[v].$$

Lecture 24

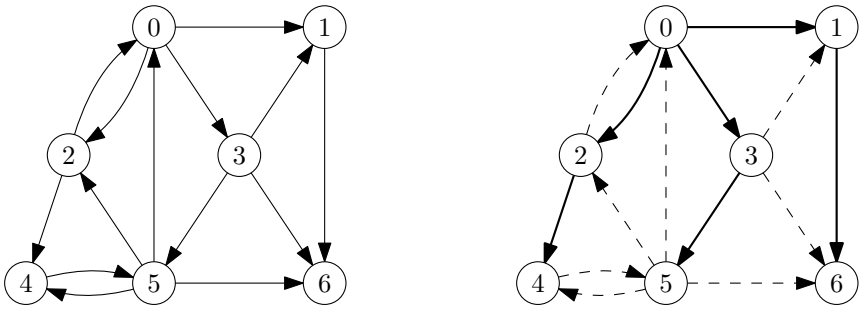
Breadth-first search (BFS) and priority-first search (PFS)

Definition 24.1. In ***breadth-first search*** (BFS) the new grey node chosen is the one that has been grey for the **longest** time.

BFS takes us away from the root node as slowly as possible. First we visit the root, then all its neighbours, then all neighbours of its neighbours and so on. The root is the first node to turn black.

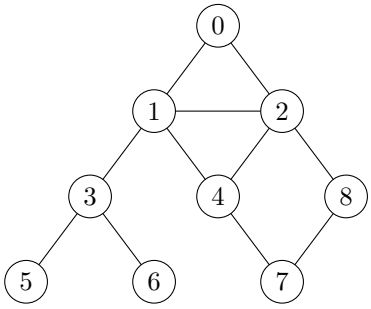
As with DFS, the running time of BFS is in $\Theta(n + m)$ when implemented using adjacency lists.

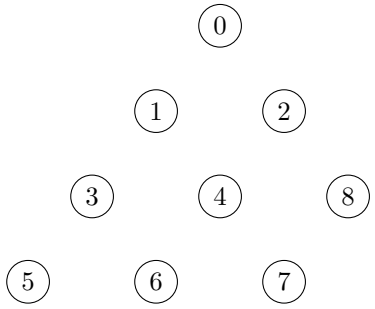
Example 24.2. A digraph and its BFS search tree, rooted at node 0. The dashed arcs indicate the original arcs that are not part of the BFS search tree.



Find a tree arc, cross arc, forward arc and back arc (or say if that type of arc does not exist for that traversal).

Example 24.3. Use the nodes on the right to draw the search tree you obtain by running BFS on the graph on the left, starting at vertex 0. Use dashed edges to indicate edges that are not arcs in the search tree.





Find a tree arc, cross arc, forward arc and back arc (or say if that type of arc does not exist for that traversal).

24.1 Pseudocode for breadth-first search

The first-in first-out processing of the grey nodes in BFS is ideally handled by a queue. The pseudocode for `BFS` and `BFSvisit` is in Algorithms 24 and 25. The timestamps `seen` and `done` of DFS are of less use here. It is more useful to record the number of steps from the root in the array d.

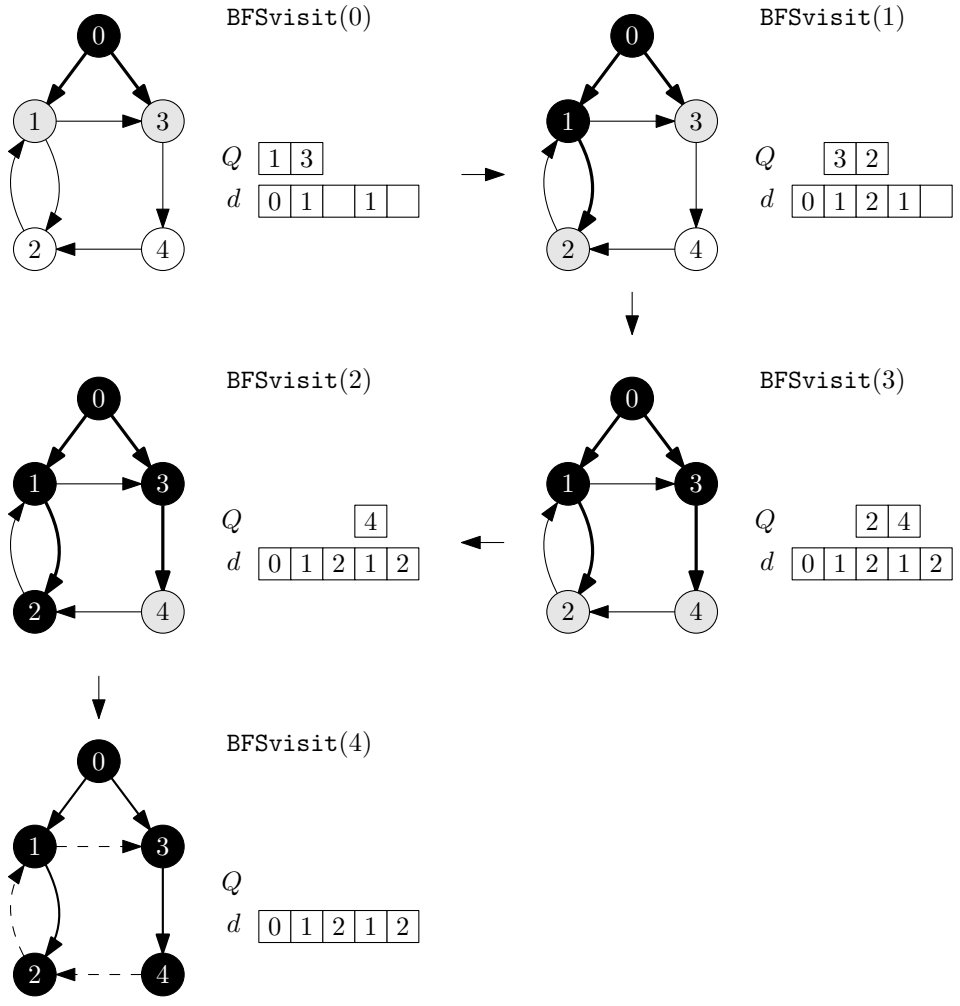
Algorithm 24 Breadth-first search algorithm.

```
1: function BFS(digraph G)
2:     queue Q
3:     array colour[0..n − 1], pred[0..n − 1], d[0..n − 1]
4:     for u ∈ V(G) do
5:         colour[u] ← WHITE; pred[u] ← null                ▷ initialise arrays
6:     for s ∈ V(G) do
7:         if colour[s] = WHITE then                        ▷ find a WHITE node
8:             BFSvisit(s)                                  ▷ start traversal from s
9:     return pred, d
```

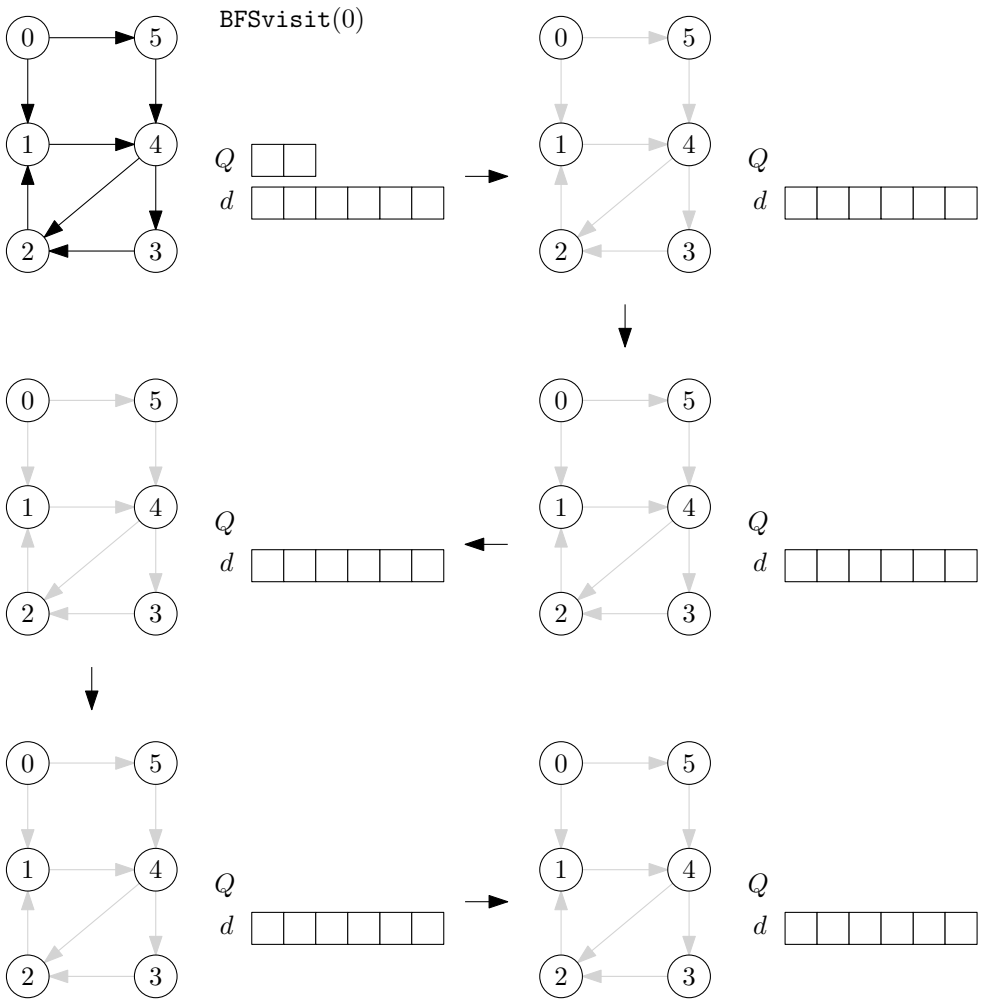
Algorithm 25 Breadth-first search visit algorithm.

```
1:  function BFSVISIT(node s)
2:      colour[s] ← GREY; d[s] ← 0
3:      Q.insert(s)                                   ▷ put s in queue
4:      while not Q.isEmpty() do
5:          u ← Q.peek()                              ▷ get node at front of queue
6:          for each v adjacent to u do
7:              if colour[v] = WHITE then             ▷ find white neighbours
8:                  colour[v] ← GREY; pred[v] ← u; d[v] ← d[u] + 1
9:                  Q.insert(v)      ▷ update colour, tree, and depth, put in queue
10:         Q.delete()                                ▷ done with this node
11:         colour[u] ← BLACK
```

Example 24.4. BFS on a digraph.



Example 24.5. Execute BFS starting at 0 and following Example 24.4 fill out the values for Q and d, and highlight the BFS search tree after each step.



24.2 BFS useful results and facts

It is rather obvious that BFS processes all nodes at distance 1, then all nodes at distance 2, etc, from the root. The formal theorem stating this is given below without proof (see book for a simple inductive proof).

Theorem 24.6. Suppose we run BFS on a digraph G. Let $v \in V(G)$, and let r be the root of the search tree containing v. Then $d[v] = d(r, v)$.

We can classify arcs, but the answer is not as nice as with DFS.

Theorem 24.7. Suppose that we are performing `BFS` on a digraph G. Let $(v, w) \in E(G)$ and suppose that we have just chosen the grey node v. Then

- if (v, w) is a tree arc then `colour`$[w]$ = WHITE, $d[w] = d[v] + 1$;
- if (v, w) is a back arc, then `colour`$[w]$ = BLACK, $d[w] \leq d[v] - 1$;
- there are no forward arcs; and
- if (v, w) is a cross arc then one of the following holds:
  - $d[w] < d[v] - 1$, and `colour`$[w]$ = BLACK;
  - $d[w] = d[v]$, and `colour`$[w]$ = GREY;
  - $d[w] = d[v]$, and `colour`$[w]$ = BLACK;
  - $d[w] = d[v] - 1$, and `colour`$[w]$ = GREY;
  - $d[w] = d[v] - 1$, and `colour`$[w]$ = BLACK.

Example 24.8. Explain why there are no forward arcs when performing BFS on a digraph.

In the special case of graphs we can say more.

Theorem 24.9. Suppose that we have performed `BFS` on a graph G. Let $\{v, w\} \in E(G)$. Then exactly one of the following conditions holds.

- $\{v, w\}$ is a tree edge, $|d[w] - d[v]| = 1$;
- $\{v, w\}$ is a cross edge, $d[w] = d[v]$;
- $\{v, w\}$ is a cross edge, $|d[w] - d[v]| = 1$.

24.3 Priority-first search

Priority-first search is a more general and sophisticated form of traversal that encompasses both BFS and DFS (and others).

- Each grey node has associated with it an integer ***key***.
- The interpretation of the key is of a priority: the smaller the key, the higher the priority.
- The rule for choosing a new grey node is to choose one with the smallest key.
- The key can either be assigned once when the node is first seen and then left unchanged, or could be updated at other times. We concentrate on unchanging keys here.
- To mimic BFS, set the key for the node v to be the time that v turns grey. It will always remain as the lowest key until it turns black.
- To mimic DFS, set the key for node v to be $-$ `seen`$[v]$, so that the most recently seen node has the lowest key.
- The running time of PFS depends on how long it takes to find the minimum key value.
- The rules described here implemented using an array take $\Omega(n)$ to find the lowest key, so the algorithm is $\Theta(n^2)$. Contrast with with $\Theta(m+n)$ for standard traversal.
- PFS is best described via the priority queue ADT, which has more efficient implementations than using a standard array.

Pseudocode `PFS` and `PFSvisit` demonstrating the PFS is presented in Algorithms 26 and 27. The subroutine `setKey` there is the rule for giving the key value when a node is inserted. We do not include any code for `setKey`.

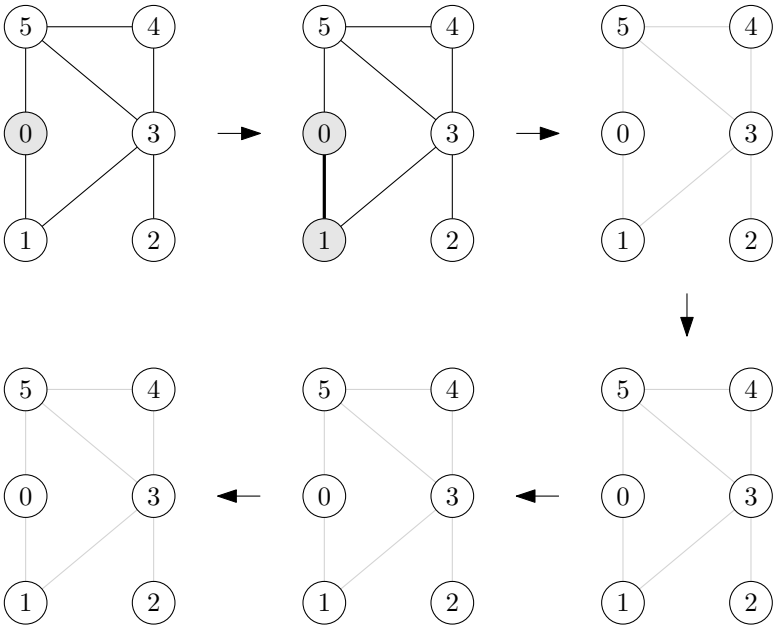
Algorithm 26 Priority-first search algorithm.

```
1: function PFS(digraph G)
2:     priority queue Q
3:     array colour[0..n − 1], pred[0..n − 1]
4:     for u ∈ V(G) do
5:         colour[u] ← WHITE; pred[u] ← null
6:     for s ∈ V(G) do
7:         if colour[s] = WHITE then
8:             PFSvisit(s)
9:     return pred
```

Algorithm 27 Priority-first visit algorithm.

```
1:  function PFSVISIT(node s)
2:      colour[s] ← GREY
3:      Q.insert(s, setKey(s))
4:      while not Q.isEmpty() do
5:          u ← Q.peek()
6:          if u has a neighbour v with colour[v] = WHITE then
7:              colour[v] ← GREY
8:              Q.insert(v, setKey(v))
9:          else
10:             Q.delete()
11:             colour[u] ← BLACK
```

Example 24.10. Execute PFS on the graph below so that nodes with higher degree have higher priority (recall that a lower key corresponds to a higher priority).
Start the traversal at vertex 0 and break ties by choosing the vertex with the lower index first.
At each step add one vertex and edge to the search tree.



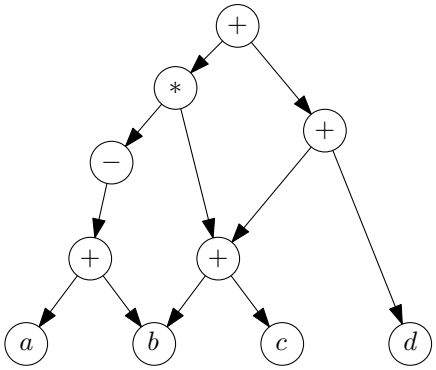
Lecture 25

Topological sort, acyclic graphs and girth

Many computer science applications require us to find precedence (or dependencies) among events. If we consider the events to be nodes and an arc (u, v) means that u precedes v, that is, that u must be calculated before v can be calculated, deciding the order in which to process events becomes a problem of sorting the nodes of a digraph.

Example 25.1. Consider a compiler evaluating sub-expressions of the expression $-(a+b)*(b+c)+((b+c)+d)$. The compiler must compute, for example, $(a+b)$ and $(b+c)$ before it could compute $-(a+b)*(b+c)$. This can be shown as a dependency digraph where the arc (u, v) means that u depends on v so that v must be calculated first (this is the opposite of a precedence digraph discussed above).

$-(a+b)*(b+c)+((b+c)+d)$



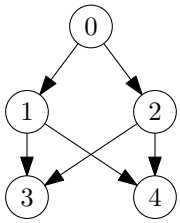
Respecting all precedences/dependencies is equivalent to drawing the digraph with all nodes in a straight line and the arcs all pointing the same direction. The order of calculation starts at one end of the line and proceeds to the other.

25.1 Topological sort

Definition 25.2. Let G be a digraph. A ***topological sort*** of G is a linear ordering of all its vertices such that if G contains an arc (u, v), then u appears before v in the ordering. It is also known as a ***topological order*** or ***linear order***.

For our arithmetic expression example above, a linear order of the sub-expression digraph gives us an order (actually the reverse of the order) where we can safely evaluate the expression.

Example 25.3. A digraph with all possible topological orders and drawn with topological order $0, 1, 2, 3, 4$. Draw the digraph for the topological order $0, 2, 1, 4, 3$.

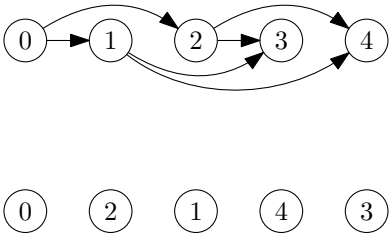


$0, 1, 2, 3, 4$

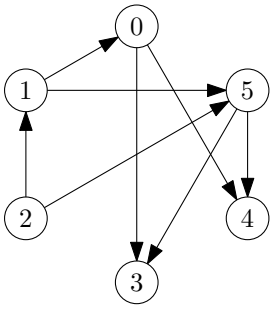
$0, 1, 2, 4, 3$

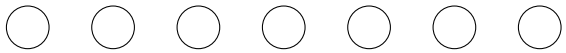
$0, 2, 1, 3, 4$

$0, 2, 1, 4, 3$

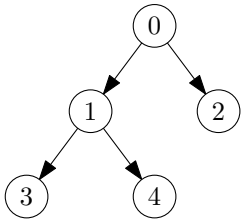


Find a topological order of the following digraph and draw it.





Topological orders are not unique. List all possible topological orders of the following digraph.



If the digraph contains a cycle, it is not possible to find such a linear ordering. This corresponds to inconsistencies in the precedences given (for example, a precedes b precedes c precedes a) and no scheduling of the tasks is possible.

Definition 25.4. A digraph without cycles is commonly called a ***DAG***, an abbreviation for **d**irected **a**cyclic **g**raph.

Theorem 25.5. A digraph has a topological order if and only if it is a DAG.

Example 25.6. It is clear that if a digraph has a topological order it has no cycles, so it a DAG. Show that a DAG always has a topological order by first showing that every DAG has a source and that by removing the source and any out-arcs from the source you still have a DAG. Show that the order in which nodes are removed gives a topological order.

This theorem and proof then gives an algorithm ***zero-indegree sorting*** for topologically sorting a DAG. If we apply zero-indegree sorting to a digraph that is not a DAG, eventually it will stop because no source node can be found at some point.

Example 25.7. What is the running time of a naive implementation of zero-indegree where a source is found and then removed at each step? How could this idea be made more efficient?

25.2 Finding cycles and topological sorting using DFS

DFS can be used to determine whether or not a digraph is a DAG and, if it is, find a topological sort.

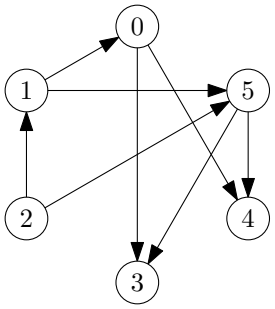
- Run DFS on G.
- If G contains a cycle, the traversal will eventually reach a node that points to one that has been seen before. In other words, we will detect a back arc.
- If the traversal finds no back arcs, G is a DAG and the listing nodes in reverse order of DFS `done` times is a topological sort of G.

Theorem 25.8. Suppose that DFS is run on a digraph G. Then G is acyclic if and only if there are no back arcs.

Theorem 25.9. Let G be a DAG. Then listing the nodes in reverse order of DFS finishing times yields a topological order of G.

Proof. Consider any arc $(u, v) \in E(G)$. Since G is a DAG, the arc is not a back arc by Theorem 25.8. In the other three cases, Example 23.8 shows that `done`$[u] >$ `done`$[v]$, which means u comes before v in the alleged topological order. □

Example 25.10. Find the topological order for the graph shown by running DFS starting at node 0. Is it the same order you produced for this graph in Example 25.3?



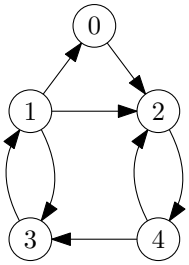
25.3 The girth of a graph

The length of the smallest cycle in a graph is an important quantity. For example, in a communication network, short cycles are often something to be avoided because they can slow down message propagation.

Definition 25.11. The ***girth*** of the graph is the length of the shortest cycle. If the graph has no cycles then the girth is undefined but may be viewed as $+\infty$.

For a digraph we use the term girth for its underlying graph and the (maybe non-standard) term ***directed girth*** for the length of the smallest directed cycle.

Example 25.12. What are girth and directed girth of the following digraph?



girth:

directed girth:

Lecture 26

Finding girth using BFS, connectivity, and components

26.1 Finding the girth of a graph with BFS

How to compute the girth of a graph? Here is an algorithm for finding the length of a shortest cycle containing a given vertex v in a graph G.

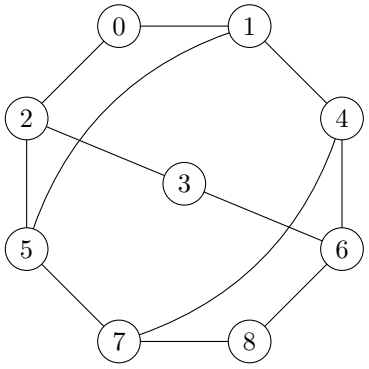
- Perform `BFSvisit` starting at v.
- If we meet a grey neighbour (that is, we are exploring edge $\{x, y\}$ from x and we find that y is already grey), continue only to the end of the current level and then stop.
- For each edge $\{x, y\}$ as above on this level, if v is the lowest common ancestor of x and y in the BFS tree, then there is a cycle containing x, y, v of length $l = d(x) + d(y) + 1$.
- Report the minimum value of l obtained along the current level.

Example 26.1. Prove above algorithm is correct by showing that:

1. meeting a grey neighbour means that you have indeed found a cycle;
2. any cycle that exists will be found in such a way; and
3. the cycle found in this way will be the shortest going through v.

To compute the girth of a graph, we can simply run the above algorithm from each vertex in turn, and take the minimum cycle length achieved.

Example 26.2. An easy-to-implement DFS idea may not work properly. Consider the DFS tree originating from vertex 0 of the graph below. Which is the smallest cycle it finds with the back edges? Which smaller cycle does it miss?



26.2 Finding the directed girth of a digraph with BFS

A similar idea can be applied to finding the directed girth of a digraph.

Example 26.3. Convince yourself that the following procedure finds the (length of the) shortest directed cycle through node v in a digraph.

1. Run `BFSvisit` starting at node v.
2. The first time a back-arc of the form (x, v) is found, we have found a cycle of length equal to (1 + the depth of x in the search tree).
3. Stop and Return the length found.

Why is there no need to continue to the end of the level before halting the traversal? How can this be used to find the directed girth of the digraph? What is the running time for doing so?

26.3 Connectivity in graphs

For many purposes it is useful to know whether a digraph is "all in one piece", and if not, to decompose it into pieces.

Definition 26.4. A graph is ***connected*** if for each pair of vertices $u, v \in V(G)$, there is a path between them.

Theorem 26.5. Let G be a graph. Then G can be uniquely written as a union of subgraphs G_i such that

- each G_i is connected, and
- if $i \neq j$, there are no edges from any vertices in G_i to any vertices in G_j.

The subgraphs G_i above are called the ***connected components*** of the graph G. Clearly, a graph is connected if and only if it has exactly one connected component.

Example 26.6. The graph obtained by deleting two edges from a triangle has 2 connected components. Draw a graph with 5 vertices, 5 edges and 2 connected components.

We can determine the connected components of a graph easily by using a traversal algorithm. The following obvious theorem explains why.

Theorem 26.7. Let G be a graph and suppose that DFS or BFS is run on G. Then the connected components of G are precisely the subgraphs spanned by the trees in the search forest.

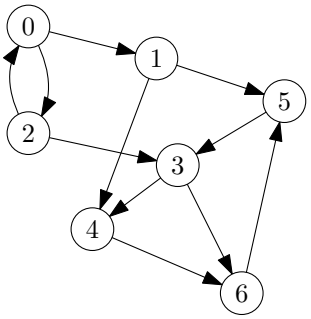
So to find the components of a graph G:

- Run BFS or DFS on G and count of the number of times we choose a root – this is the number of components.
- Store or print the vertices and edges in each component as we explore them.
- This is a linear time algorithm, $O(m+n)$.

26.4 Connectivity in digraphs

The intuition of a digraph being "all in one piece" is not as useful as it was for graphs.

Example 26.8. This digraph appears to be "all in one piece" as its underlying graph is connected. But can you get from any node to any other node following the direction of the arcs?



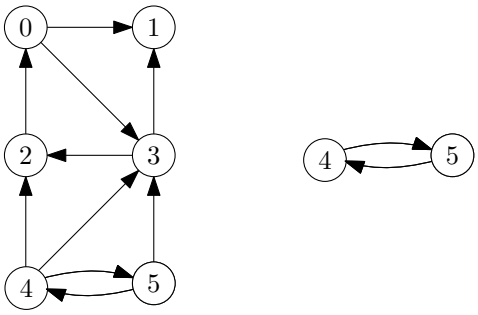
This example motivates the following definition.

Definition 26.9. A digraph G is ***strongly connected*** if for each pair of nodes u, v of G, there is a path in G from u to v and from v to u (that is, u and v are reachable from one another).

Example 26.10. Show that a strongly connected digraph of order at least two contains at least one cycle.

Definition 26.11. A ***strongly connected component***, G_i, of G is a maximal subdigraph of G such that G_i is strongly connected. All nodes of G are in exactly one such G_i, so the partition into strongly connected components is unique.

Example 26.12. This digraph has three strongly connected components. Find and draw the two missing ones.



Note that there are arcs of the digraph not included in the strongly connected components.

Example 26.13. Find a counter-example to show that the method for finding connected components of graphs in Theorem 26.7 fails at finding strongly connected components of digraphs.

Example 26.14. Using `BFSvisit` or `DFSvisit`, devise an algorithm that will determine whether or not G is strongly connected (i.e., has a single strongly connected component). What is the running time of the algorithm?

Lecture 27

Finding strong components, and bipartite graphs

27.1 Finding the strongly connected components

It turns out we can find all connected components of G in linear time ~~using~~ by a clever use of DFS, known as Tarjan's algorithm.

Suppose the underlying graph of G is connected but G is not strongly connected, then there are strong components C_1 and C_2 such that it is possible to get from C_1 to C_2 but not from C_2 to C_1. If C_1 and C_2 are different strong components, then any arcs between them must either all point from C_1 to C_2 or from C_2 to C_1.

This generalises to all strongly connected components (Figure 27.1). By reducing each strongly connected component to a single node, we derive a new graph, H, from G that is a DAG.

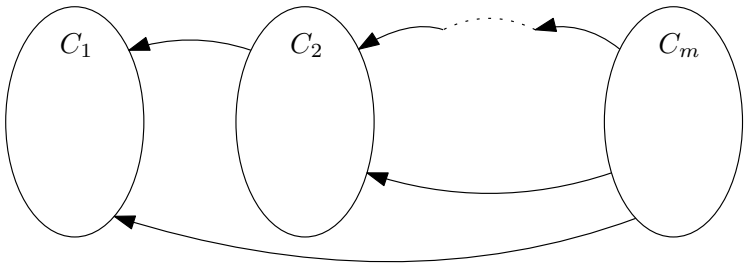


Figure 27.1: Decomposing a graph into its strongly connected components, $C_1, \ldots, C_m$. If each C_i is considered a single node, this decomposition is a DAG.

If we started a DFS in C_1, we'd reach only nodes in C_1 and no other nodes. If we then started a DFS in C_2, we'd see only those in $C2$ and so on. The question is, how do we know to start in C_1, the C_2 etc.?

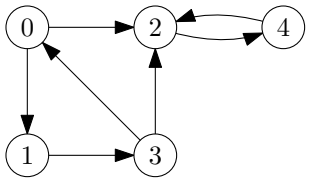
Since H is a DAG, we can find a topological order of it using the reverse order of finishing times of a DFS. We note that the strong components of G are the same as the strong components of its reverse, G_r. So, as before, starting a DFS in C_m of G_r means we would only visit nodes in C_m and so on. This leads to the following solution, due to Tarjan.

- Run a DFS on ~~G~~ G_r starting at an arbitrary node and record the finishing (`done`) times for each node.
- List nodes of ~~G~~ G_r in reverse order of finishing (from last finished to first).

- Run DFS on ~~G_r, the reverse of~~ G, always choosing a root (starting) node as the unvisited node that occurs first on the list in the previous step.
- This produces a forest F_r where each tree in the forest is a strongly connected component of G (and of G_r).

The above algorithm runs in linear time with adjacency lists, since it only requires two DFS traversals (which are each linear time), and the creation of the reverse digraph which also takes linear time.

Example 27.1. Run the algorithm above on the graph shown (starting in the initial DFS at node 0) to find all strongly connected components.

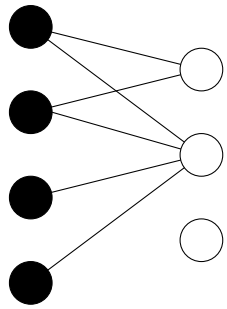


27.2 Bipartite graphs

Many graphs in applications have two different types of nodes, and no relations between nodes of the same type (this is a model of sexually reproducing organisms, for example).

Definition 27.2. A graph G is ***bipartite*** if $V(G)$ can be partitioned into two nonempty disjoint subsets V_0 and V_1 such that each edge of G has one endpoint in V_0 and one in V_1.

Example 27.3. This graph is bipartite. The isolated vertex could be placed on either side.



There are exponentially many partitions of V into two subsets so we do not want to have to test all possibilities to determine whether V is bipartite.

Definition 27.4. Let k be a positive integer. A graph G has a ***k*-colouring** if $V(G)$ can be partitioned into k nonempty disjoint subsets such that each edge of G joins two vertices in different subsets.

Example 27.5. The graph in Example 27.3 has a 2-colouring as indicated.

Theorem 27.6. The following conditions on a graph G are equivalent.

1. G is bipartite.
2. G has a 2-colouring.
3. G does not contain an odd length cycle.

Example 27.7. Prove Theorem 27.6 by showing that statement 1 implies 2 implies 3 which implies 1. To show statement 3 implies 1, consider a BFS on G where vertices at level i are given "colour" $i \bmod 2$.

Part VI

Weighted digraphs and optimization problems

Lecture 28

Weighted graphs, single-source shortest paths problem, Dijkstra

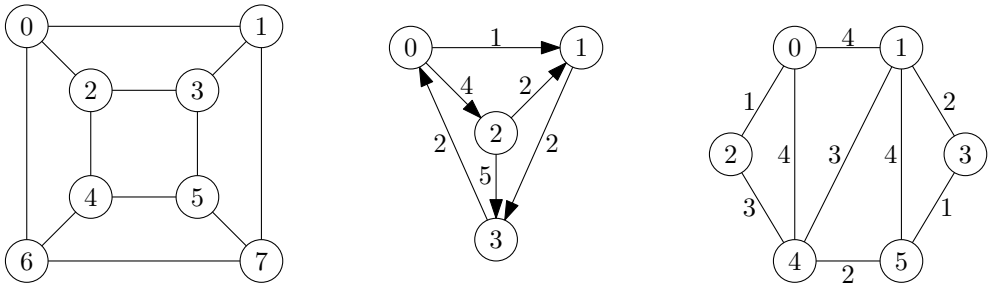
Weighted digraphs encode not only information about **whether** one can get from A to B, but **how much it will cost** to do so. The weight could represent the cost of using a link in a communication network, or distance between nodes in a transportation network. We use the terms of cost and distance interchangeably.

28.1 Weighted digraphs

Definition 28.1. A ***weighted digraph*** is a pair (G, c) where G is a digraph and c is a ***cost function*** associating a real number to each arc of G. For an arc (u, v), we interpret $c(u, v)$ as the ***cost*** of using (u, v).

An ordinary digraph can be thought of as a special type of weighted digraph where the cost of each arc is 1.

Example 28.2. A classic unweighted graph (called the 3-cube), a digraph with arc weights, and a graph with edge weights.



Weighted digraphs can be represented using adjacency matrices or lists:

- The adjacency matrix is modified so that each entry of 1 (signifying that an arc exists) is replaced by the cost of that arc.
- Care needs to be taken if using 0 to represent the absence of arcs, as 0 may be a legal edge weight. In these cases, `null`, ∞ or some suitable value may be used. We use **the convention that 0 or `null` represents the absence of an arc**.
- An adjacency list is modified so that the list associated with node v has each adjacency node followed by the cost of the arc to the adjacent node.

- For example, the list for node 6 with adjacent arcs $(6,2)$ with $c(6,2) = 4$ and $(6,7)$ with $c(6,7) = 5$ would be $2,4,7,5$.

Example 28.3. Draw the weighted graph given by the weighted matrix below.

$$\begin{bmatrix} 0 & 3 & 4 & 0 \\ 3 & 0 & 1 & 3 \\ 4 & 1 & 0 & 2 \\ 0 & 3 & 2 & 0 \end{bmatrix}$$

Draw the weighted digraph given by the weighted list representation below.

0	1	3	2	4
1	0	2	3	2
2	1	3		
3	2	1		

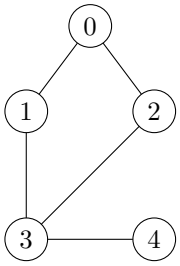
28.2 Distance and diameter

Recall that the distance $d(u,v)$ between nodes u and v in an (unweighted) digraph is simply the number of arcs in the shortest path between them.

In a weighted digraph, the distance to from node u to v, $d(u,v)$, is the cost of the minimum cost path from u to v where the cost of a path is just the sum of the costs on that path.

- It is often helpful to have the ***distance matrix*** for a digraph. The (i,j)-entry of this matrix contains the distance between node i and node j.
- For an unweighted digraph, the distance matrix can be generated by running `BFSvisit` from each node in turn, since the distance to a node is equal to its depth in the BFS tree (or infinite if the node is not reachable from the start node so not in the tree). This gives an algorithm with running time in $\Theta(n^2 + nm)$.

Example 28.4. A graph, its adjacency matrix and its distance matrix.



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 2 & 3 \\ 1 & 0 & 2 & 1 & 2 \\ 1 & 2 & 0 & 1 & 2 \\ 2 & 1 & 1 & 0 & 1 \\ 3 & 2 & 2 & 1 & 0 \end{bmatrix}$$

Example 28.5. Give an example of a weighted digraph in which the BFS approach does not find the shortest path from the root to a node.

Definition 28.6. The ***diameter*** of a strongly connected digraph G is the maximum of $d(u,v)$ over all nodes $u,v \in V(G)$. If the digraph is not strongly connected the diameter is undefined though can be set to ∞.

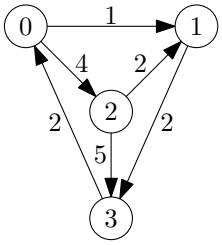
Clearly the diameter is just the maximum entry in the distance matrix.

Example 28.7. What is the diameter of the 3-cube in Example 28.2?

28.3 Single-source shortest path problem

Definition 28.8. In the ***single-source shortest path problem*** (SSSP) we are given a weighted digraph (G,c) and a source node s. For each node v of G, we must find the minimum weight of a path from s to v. By the weight of a path we mean the sum of the weights on the arcs. This is like finding row s in a weighted distance matrix.

Example 28.9. In the weighted digraph pictured, the unique shortest path from 0 to 3 is $0,1,3$ with weight 3. What is the path with minimum weight from 2 to 0, and what is its weight?



28.4 Dijkstra's algorithm

Dijkstra's algorithm solves the SSSP problem whenever **all weights are non-negative**. It may fail in the presence of negative weight arcs.

It is easiest to understand the algorithm in terms of a set of visited nodes S, which eventually includes all nodes in G. We'll consider only shortest paths through S. Once $S = V(G)$, all shortest path lengths are known.

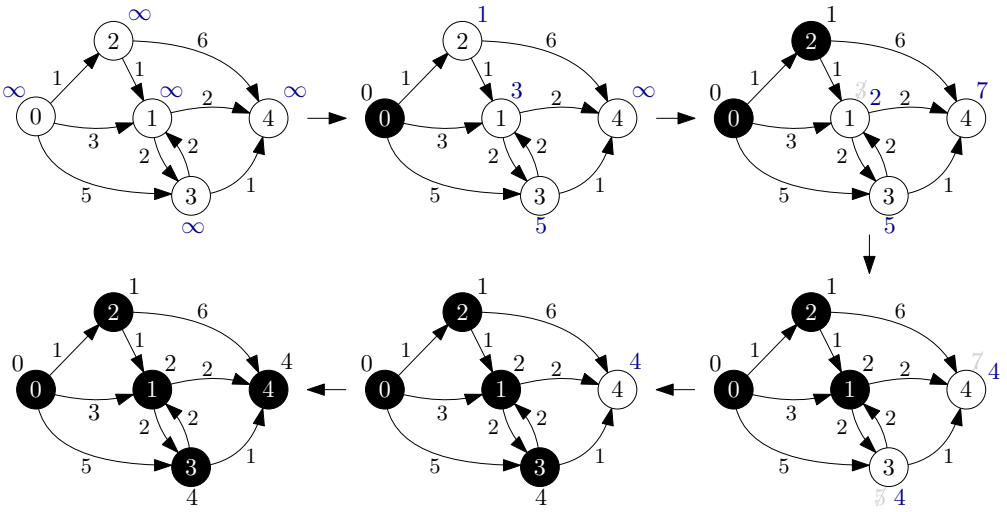
Initially S contains only the single node s. The only paths available are the one-arc paths from s to neighbours v, of weight $c(s,v)$. We choose the neighbour u with $c(s,u)$ minimal and add it to S.

Now the fringe nodes adjacent to s and u must be updated to reflect possible paths through u (it is possible that there exists a path from s to v, passing through u, that is shorter than the direct path from s). Now we choose the node whose current best distance to s is smallest, and update again. We continue in this way until all nodes belong to S. In Algorithm 28, the set S consists of the BLACK nodes.

Algorithm 28 Dijkstra's algorithm, first version.

```
 1: function DIJKSTRA(weighted digraph (G, c); node s ∈ V(G))
 2:     array colour[0..n − 1], dist[0..n − 1]
 3:     for u ∈ V(G) do
 4:         dist[u] ← c[s, u]; colour[u] ← WHITE
 5:     dist[s] ← 0; colour[s] ← BLACK
 6:     while there is a white node do
 7:         find a white node u so that dist[u] is minimum
 8:         colour[u] ← BLACK
 9:         for x ∈ V(G) do
10:             if colour[x] = WHITE then
11:                 dist[x] ← min{dist[x], dist[u] + c[u, x]}
12:     return dist
```

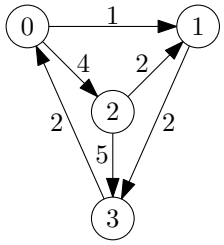
Example 28.10. An application of Dijkstra's algorithm with starting vertex 0. The distances to 0 at each step are displayed next to each node.



Dijkstra's algorithm is an example of a ***greedy algorithm***. At each step it makes the best choice involving only local information, and never regrets its past choices.

Example 28.11.

An application of Dijkstra's algorithm on the digraph below for each starting vertex s. Complete the table for the starting vertex 2.



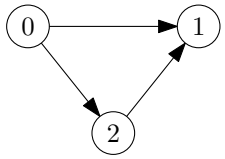
The table illustrates that the distance vector is updated at most $n-1$ times (only before a new vertex is selected and added to S). Thus we could have omitted the lines with $S = \{0,1,2,3\}$.

current $S \subseteq V$	**distance vector** `dist`
$\{0\}$	$0, 1, 4, \infty$
$\{0,1\}$	$0, 1, 4, 3$
$\{0,1,3\}$	$0, 1, 4, 3$
$\{0,1,2,3\}$	$0, 1, 4, 3$
$\{1\}$	$\infty, 0, \infty, 2$
$\{1,3\}$	$4, 0, \infty, 2$
$\{0,1,3\}$	$4, 0, 8, 2$
$\{0,1,2,3\}$	$4, 0, 8, 2$
$\{2\}$ $\{\quad\}$ $\{\quad\}$ $\{0,1,2,3\}$	
$\{3\}$	$2, \infty, \infty, 0$
$\{0,3\}$	$2, 3, 6, 0$
$\{0,1,3\}$	$2, 3, 6, 0$
$\{0,1,2,3\}$	$2, 3, 6, 0$

Lecture 29

Dijkstra proof and running time

Example 29.1. Give weights to the edges of the digraph below so that running Dijkstra's algorithm with starting vertex 0 fails.



Proving Dijkstra's algorithm works is trickier than for other algorithms we've seen.

Define an ***S*-path** from s to w as a path from s to w with s and all intermediate nodes belonging to S. In other words, w may not belong to S, but all other nodes in the path do.

Theorem 29.2. Suppose that all arc weights are non-negative. Then at the top of the **while** loop, we have the following properties:

P1: If $x \in V(G)$, then `dist`$[x]$ is the minimum cost of an S-path from s to x.

P2: If $w \in S$, then `dist`$[w]$ is the minimum cost of a path from s to w.

At every step, `dist`$[x]$ is the length of some path from s to x, or ∞. That path is an S-path if $x \in S$. Also note that the update formula ensures that `dist`$[x]$ never increases.

To prove P1 and P2, we use induction on the number of times k we have been through the while-loop. Let S_k denote the value of S at this stage.

Example 29.3. Show that P1 and P2 hold when $k = 0$.

Example 29.4. Show that P1 holds for any value of k.

Example 29.5. Show that P2 holds for any value of k and that this proves the correctness of Dijkstra's algorithm.

29.1 Running time of Dijkstra

The value of `dist`$[x]$ will change only if x is adjacent to u. Thus if we use a weighted adjacency list, the block inside the second for-loop need only be executed m times. However, if using the adjacency matrix representation, the block inside the for-loop must still be executed n^2 times.

The time complexity is of order $an + m$ if adjacency lists are used, and $an + n^2$ with an adjacency matrix, where a represents the time taken to find the node with minimum value of `dist`. The obvious method of finding the minimum is simply to scan through array `dist` sequentially, so that a is of order n, and the running time of Dijkstra is therefore $\Theta(n^2)$.

The next lecture looks at a more efficient implementation with running time $O((m+n)\log n)$.

Lecture 30

Dijkstra and PFS, Bellman-Ford algorithm

30.1 PFS implementation of Dijkstra

- We can implement Dijkstra's algorithm using priority-first search ideas.
- The key value associated to a node u is simply the value `dist`$[u]$, the current best distance to that node from the root s.

Algorithm 29 Dijkstra's algorithm, PFS version.

```
1:  function DIJKSTRA2(weighted digraph (G, c); node s ∈ V(G))
2:      priority queue Q
3:      array colour[0..n − 1], dist[0..n − 1]
4:      for u ∈ V(G) do
5:          colour[u] ← WHITE
6:      colour[s] ← GREY
7:      Q.insert(s, 0)
8:      while not Q.isEmpty() do
9:          t1 ← Q.getKeyQ.peek()
10:         u ← Q.pop()
11:         for each x adjacent to u do
12:             t2 ← t1 + c(u, x)
13:             if colour[x] = WHITE then
14:                 colour[x] ← GREY
15:                 Q.insert(x, t2)
16:             else if colour[x] = GREY and Q.getKey(x) > t2 then
17:                 Q.decreaseKey(x, t2)
18:         colour[u] ← BLACK
19:         dist[u] ← t1
20:     return dist
```

We can see that the dominant operations in terms of running time are

- n delete-min operations, and
- (at most) m decrease-key operations.

Hence using a binary heap, Dijkstra's algorithm runs in time $O((n+m)\log n)$. Thus if every node is reachable from the source, it runs in time $O(m \log n)$.

The best complexity bound for Dijkstra's algorithm, using a **Fibonacci heap**, is $O(m + n \log n)$.

30.2 Bellman–Ford algorithm

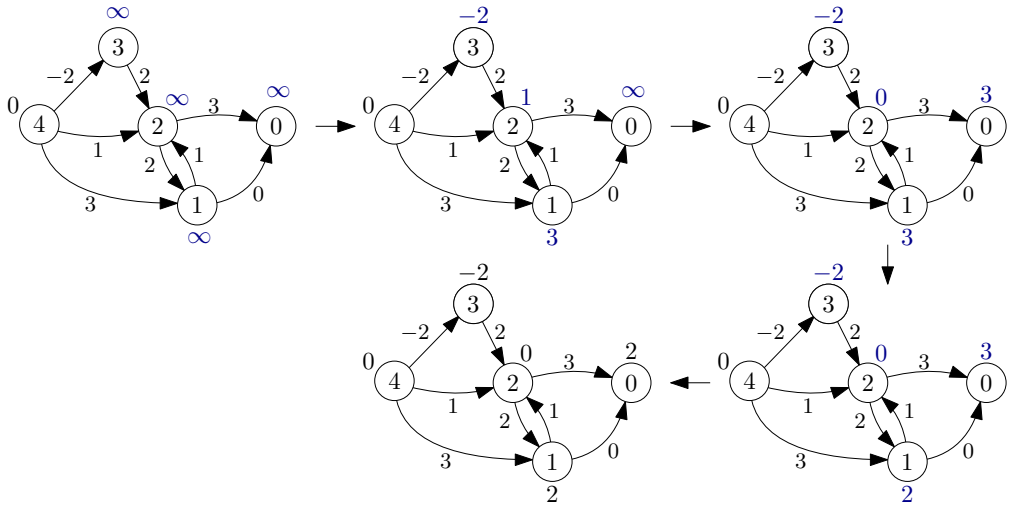
The ***Bellman–Ford algorithm*** solves the SSSP problem **even when there are negative weight arcs**. It is not surprising that the algorithm thus runs more slowly than Dijkstra's algorithm.

- The basic idea, as with Dijkstra's algorithm, is to solve the SSSP under restrictions that become progressively more relaxed.
- Bellman–Ford solves the problem for all nodes at "level" $0, 1, \ldots, n-1$ in turn.
- By level we mean the minimum possible number of arcs in a minimum weight path to that node from the source.

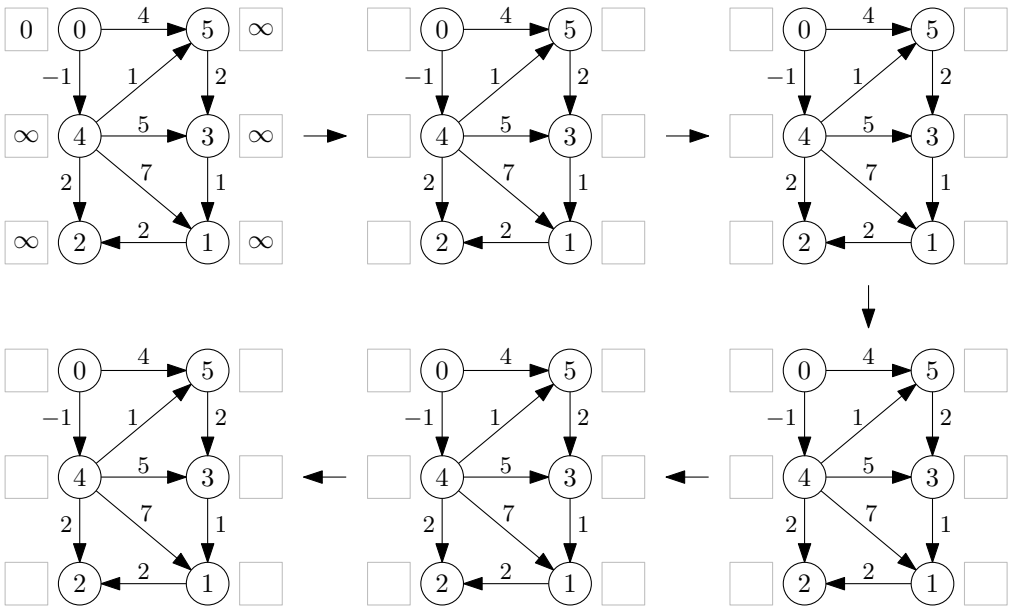
Algorithm 30 Bellman–Ford algorithm.

```
1:  function BellmanFord(weighted digraph (G, c); node s ∈ V(G))
2:      array dist[0..n − 1]
3:      for u ∈ V(G) do
4:          dist[u] ← ∞
5:      dist[s] ← 0
6:      for i from 0 to n − 1 do
7:          for x ∈ V(G) do
8:              for v ∈ V(G) do
9:                  dist[v] ← min(dist[v], dist[x] + c(x, v))
10:     return dist
```

Example 30.1. An application of Bellman–Ford algorithm with starting node 4 when the nodes are processed in the order from 0 to 4.



Example 30.2. Execute the Bellman–Ford algorithm on the graph below with starting vertex 0. Process nodes in the order from 0 to 5.



Theorem 30.3. Suppose that G contains no negative weight cycles. Then after the i-th iteration of the outer for-loop, `dist`$[v]$ contains the minimum weight of a path to v for all nodes v with level at most i.

Proof. Note that as for Dijkstra, the update formula is such that `dist` values never increase.

We use induction on i. When $i = 0$ the result is true because of our initialization. Suppose it is true for $i-1$. Let v be a node at level i, and let γ be a minimum weight path from s to v. Since there are no negative weight cycles, γ has i arcs. If y is the last node of γ before v, and γ_1 the subpath to y, then by the inductive hypothesis we have `dist`$[y] \le |\gamma_1|$. Thus by the update formula we have `dist`$[v] \le$ `dist`$[y] + c(y,v) \le |\gamma_1| + c(y,v) \le |\gamma|$ as required. □

The Bellman–Ford algorithm runs in time $\Theta(nm)$ using adjacency lists, since the statement in the inner for-loop need only be executed if v is adjacent to x, and the outer loop runs n times. Using an adjacency matrix it runs in time $\Theta(n^3)$.

Example 30.4. Explain why the SSSP problem makes no sense if we allow digraphs with cycles of negative total weight.

Example 30.5. Suppose the input to the Bellman–Ford algorithm is a digraph with a negative weight cycle. How could the algorithm detect this, so it can exit gracefully with an error message?

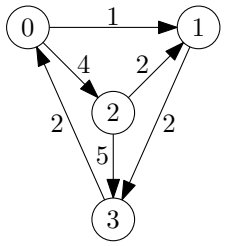
Lecture 31

All-pairs shortest path problem

Definition 31.1. In the ***all-pairs shortest path problem*** (APSP) we are given a weighted digraph (G, c), and must determine for each $u, v \in V(G)$ the weight of a minimum weight path from u to v.

The solution to the all-pairs shortest path problem can be presented as a distance matrix.

Example 31.2. For the digraph we have already calculated the all-pairs distance matrix in Example 28.11:



$$\begin{pmatrix} 0 & 1 & 4 & 3 \\ 4 & 0 & 8 & 2 \\ 6 & 2 & 0 & 4 \\ 2 & 3 & 6 & 0 \end{pmatrix}$$

The APSP problem can be solved by solving the SSSP problem from each node.

- If all weights were non-negative, we could run Dijsktra from each of the n nodes, to get a running time of $\Theta(n^3)$.
- To be robust to negative weights, using Bellman–Ford gives a $\Theta(n^2 m)$ solution to APSP.

Floyd's algorithm computes a distance matrix from a cost matrix (so solves APSP) in time $\Theta(n^3)$.

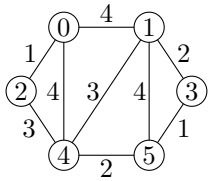
- The basic idea is that we find the length of the shortest path between nodes u and v that uses only a fixed set of intermediate nodes.
- This set of intermediate nodes starts at as the empty set (so only arcs from u to v are allowed) and grows, one node at a time, until it includes all nodes at which point the algorithm is complete.
- It is essentially just a triple for-loop so easy to programme.
- Floyd's algorithm is thus faster than Bellman–Ford for non-sparse digraphs.
- It is robust to negative weights.

We see how this works in Algorithm 31 where the outer-for loop iterates through all nodes, essentially adding them to the intermediate set, while the inner two loops cycle through all pairs seeing if a shorter path can be found through the new intermediate node.

Algorithm 31 Floyd's algorithm.

```
1: function FLOYD(weighted digraph (G, c))
2:     array d[0..n − 1, 0..n − 1]                    ▷ initialise distance matrix
3:     d ← c                          ▷ distance matrix starts as copy of weight matrix
4:     for x ∈ V(G) do                                  ▷ taking one vertex at a time
5:         for u ∈ V(G) do
6:             for v ∈ V(G) do
7:                 d[u, v] ← min(d[u, v], d[u, x] + d[x, v])
                          ▷ update distance by looking for a shorter path through x
8:     return d
```

Example 31.3. An application of Floyd's algorithm on the graph on the left is given below. The initial cost matrix is as follows.



$$\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \infty & 2 & 3 & 4 \\ 1 & \infty & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}$$

In the matrices below, we list the entries that change in bold after each iteration of the outer for-loop, that is, after x has been 0, 1, and so on.

$$\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \mathbf{5} & 2 & 3 & 4 \\ 1 & \mathbf{5} & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}$$

$x = 0$

$$\begin{bmatrix} 0 & 4 & 1 & \mathbf{6} & 4 & \mathbf{8} \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & \mathbf{7} & 3 & \mathbf{9} \\ \mathbf{6} & 2 & \mathbf{7} & 0 & \mathbf{5} & 1 \\ 4 & 3 & 3 & \mathbf{5} & 0 & 2 \\ \mathbf{8} & 4 & \mathbf{9} & 1 & 2 & 0 \end{bmatrix}$$

$x = 1$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 8 \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & 7 & 3 & 9 \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ 8 & 4 & 9 & 1 & 2 & 0 \end{bmatrix}$$

$x = 2$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{7} \\ 4 & 0 & 5 & 2 & 3 & \mathbf{3} \\ 1 & 5 & 0 & 7 & 3 & \mathbf{8} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{7} & \mathbf{3} & \mathbf{8} & 1 & 2 & 0 \end{bmatrix}$$

$x = 3$

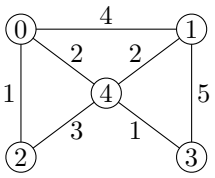
$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{6} \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & 7 & 3 & \mathbf{5} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{6} & 3 & \mathbf{5} & 1 & 2 & 0 \end{bmatrix}$$

$x = 4$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 6 \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & \mathbf{6} & 3 & 5 \\ 6 & 2 & \mathbf{6} & 0 & \mathbf{3} & 1 \\ 4 & 3 & 3 & \mathbf{3} & 0 & 2 \\ 6 & 3 & 5 & 1 & 2 & 0 \end{bmatrix}$$

$x = 5$

Example 31.4. Give the cost matrix for the graph on the left.



Execute Floyd's algorithm on the graph above by giving the matrices after each iteration of the outer for-loop.

31.1 Proof of Floyd's algorithm

Theorem 31.5. At the bottom of the outer **for** loop, for all nodes u and v, $d[u,v]$ contains the minimum length of all paths from u to v that are restricted to using only intermediate nodes that have been seen in the outer **for** loop.

Proof. To establish the above property, we use induction on the outer for-loop. Let S_k be the set of nodes seen after k times through the outer loop, and define an S_k-path to be one all of whose intermediate nodes belong to S_k. The corresponding value of d is denoted d_k. We need to show that for all k, after k times through the outer for-loop, $d_k[u,v]$ is the minimum length of an S_k-path from u to v.

When $k = 0$, $S_0 = \emptyset$ and the result holds. Suppose it is true after k times through the outer loop and consider what happens at the end of the $(k+1)$-st time through the outer loop. Suppose that x was the last node seen in the outer loop, so $S_{k+1} = S_k \cup \{x\}$. Fix $u, v \in V(G)$ and let L be the minimum length of an S_{k+1}-path from u to v. Obviously $L \leq d_{k+1}[u,v]$; we show that $d_{k+1}[u,v] \leq L$.

Choose an S_{k+1}-path γ from u to v of length L. If x is not involved then the result follows by inductive hypothesis. If x is involved, let γ_1, γ_2 be the subpaths from u to x and x to v respectively. Then γ_1 and γ_2 are S_k-paths and by the inductive hypothesis,

$$L \geq |\gamma_1| + |\gamma_2| = d_k[u,x] + d_k[x,v] \geq d_{k+1}[u,v].$$

□

Example 31.6. Draw a table to compare and summarise how BFS, Djikstra, Bellman-Ford and Floyd can be used to solve the SSSP and APSP problems for weighted and unweighted graphs and digraphs with or without negative arcs. Compare their running times and their ability to detect negative cycles.

Lecture 32

Minimum spanning tree problem

Definition 32.1. A ***spanning tree*** of a graph G is a subgraph of G that spans G (contains all nodes of G) and is a tree (a connected, acyclic graph).

Let G be a weighted graph. A ***minimum spanning tree*** (MST) is a spanning tree for G which has minimum total weight (sum of all edge weights).

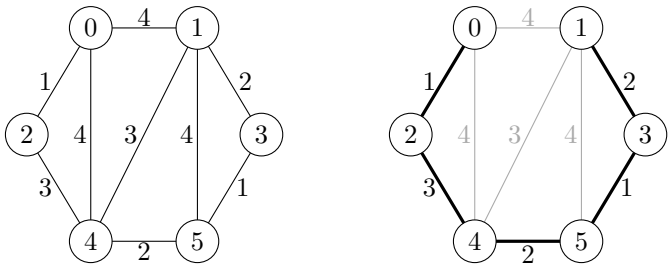
In the ***minimum spanning tree problem*** we have to find a weighted graph G is to find a MST for G.

We can assume throughout that G is connected.

Example 32.2. In the graph below, the tree determined by the edges

$$\{0,2\},\{1,3\},\{3,5\},\{4,5\},\{2,4\}$$

has total weight 9. It is a tree and has the 5 smallest weight edges, so must be a MST.



We look at two simple greedy algorithms that solve the MST problem.

32.1 Prim's algorithm

The basic idea of ***Prim's algorithm*** is simple:

- Start at any vertex.
- Choose at each step an edge of minimum weight from the remaining edges ensuring that
  1. adding the edge does not create a cycle in the subgraph built so far; and
  2. the subgraph built so far is connected.
- Stop when the tree is a spanning tree.

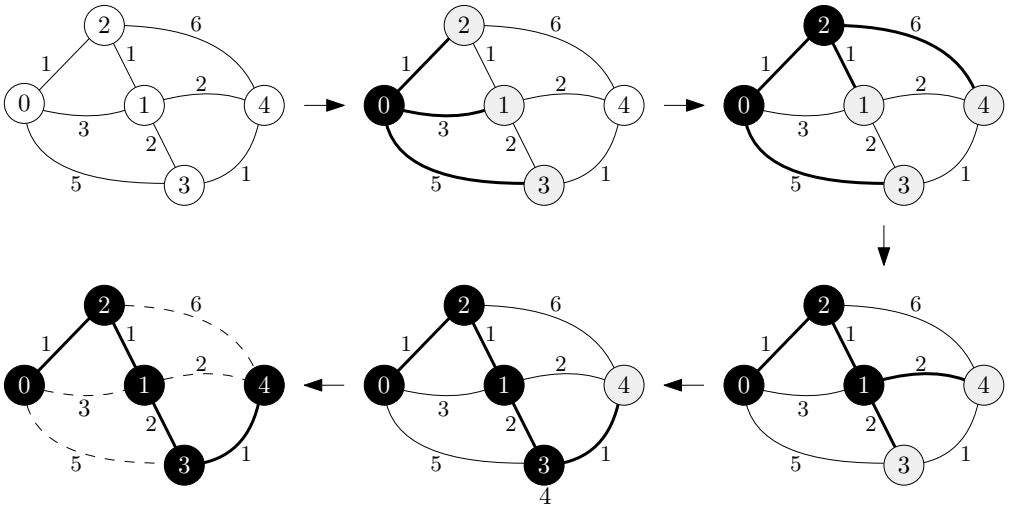
Since the subgraph built is acyclic and connected at each stage, it is a tree. It is also clear that the algorithm halts and as the tree grows by a node at each stage, it will eventually include all nodes of G, that is, be a spanning tree. So the only thing that needs to be proved for correctness is that the tree has the lowest possible weight (is minimum). We will prove this but first look at the code.

Pseudocode is given in Algorithm 32. Note how similar Prim's algorithm is to Dijkstra's algorithm. The main difference is in the update formula. We also store the PFS tree, which we did not do for Dijkstra.

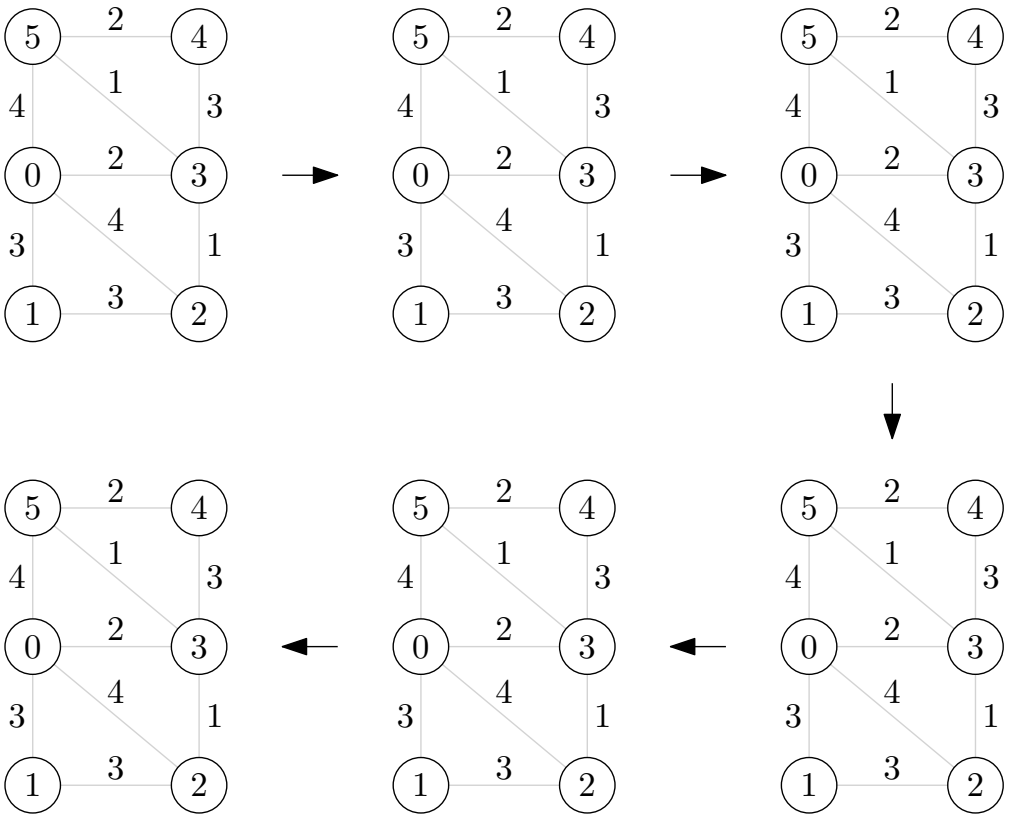
Algorithm 32 Prim's algorithm.

```
1:  function PRIM(weighted graph (G, c); vertex s ∈ V(G))
2:      priority queue Q
3:      array colour[0..n − 1], pred[0..n − 1]
4:      for u ∈ V(G) do
5:          colour[u] ← WHITE; pred[u] ← null
6:      colour[s] ← GREY
7:      Q.insert(s, 0)
8:      while not Q.isEmpty() do
9:          u ← Q.peek()
10:         for each x adjacent to u do
11:             t ← c(u, x)
12:             if colour[x] = WHITE then
13:                 colour[x] ← GREY; pred[x] ← u
14:                 Q.insert(x, t)
15:             else if colour[x] = GREY and Q.getKey(x) > t then
16:                 Q.decreaseKey(x, t); pred[x] ← u
17:         Q.delete()
18:         colour[u] ← BLACK
19:     return pred
```

Example 32.3. Application of Prim's algorithm on a graph, choosing node with lowest index where there is a choice.



Example 32.4. Execute Prim's algorithm.



Example 32.5. Prove that Prim's algorithm is correct.

Running time

The complexity of the algorithm depends to a great extent on the data structure used. The best known for Prim is the same as for Dijkstra and we get a running time in $O(m + n \log n)$.

32.2 Kruskal's algorithm

Kruskal's algorithm is even simpler than Prim's to state.

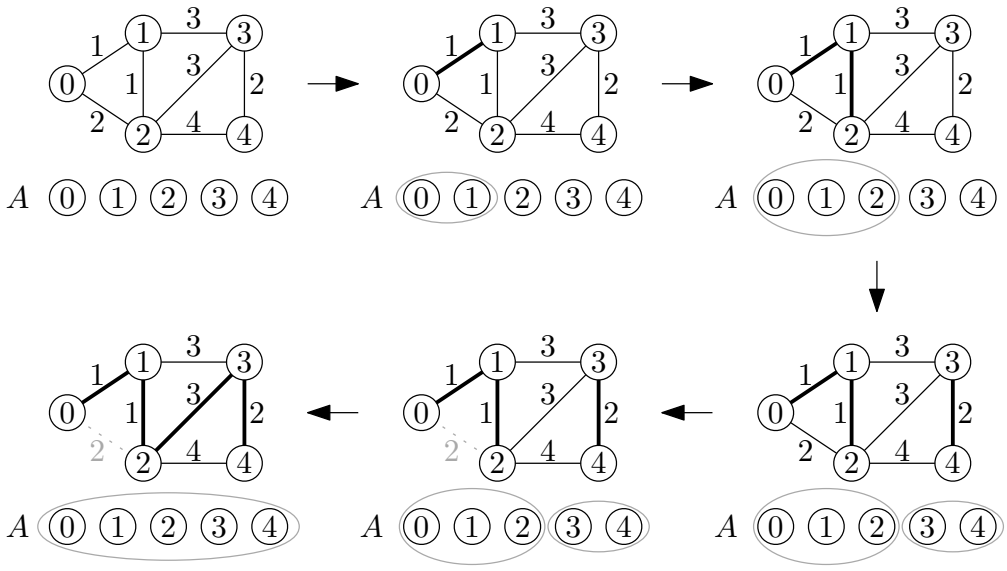
- Start with an empty set of edges.
- At each step choose an edge of minimum weight from the remaining edges ensuring that adding the edge does not create a cycle in the subgraph built so far.
- Stop when the subgraph is spanning tree.

Kruskal's algorithm maintains acyclicity, so it has a forest at each step, and the different trees merge as the algorithm progresses.

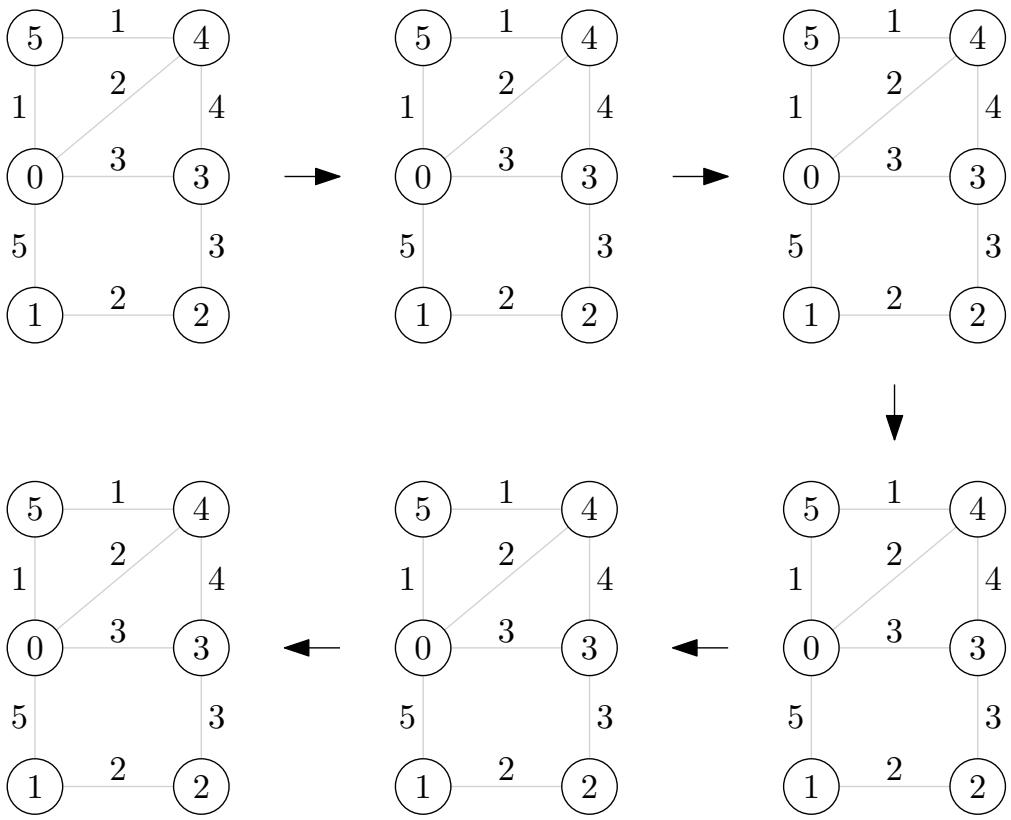
Algorithm 33 Kruskal's algorithm.

```
1: function KRUSKAL(weighted digraph (G, c))
2:     disjoint sets ADT A
3:     initialize A with each vertex in its own set
4:     sort the edges in increasing order of cost
5:     for each edge {u, v} in increasing cost order do
6:         if not A.set(u) = A.set(v) then
7:             add this edge
8:             A.union(A.set(u), A.set(v))
9:     return A
```

Example 32.6. Application of Kruskal's algorithm on a graph shown until an MST is found. Note that the edge $\{0,2\}$ with weight 2 is not added, because 0 and 2 are already in the same set in A.



Example 32.7. Execute Kruskal's algorithm by adding or crossing out the next edge. Stop when you reached an MST.



Observe that if we try to add an edge both of whose endpoints are in the same tree in the Kruskal forest, this will create a cycle, so the edge must be rejected. On the other hand, if the endpoints are in two different trees, a cycle definitely will not be created; rather, the two trees merge into a single one, and we should accept the edge.

We need a data structure that can handle this efficiently. All we need is to be able to find the tree containing an endpoint, and to merge two trees. The ***disjoint sets*** or ***union-find*** ADT is precisely what is needed. It allows us to perform the `find` and `union` operations efficiently. We do not look at the details of this ADT here.

Kruskal's is in $O(m \log n)$ using the disjoint sets ADT. The ADT can be implemented in such a way that the union and find operations in Kruskal's algorithm runs in **almost** linear time. So if the edge weights are presorted, or can be sorted in linear time (for example, if they are known to be integers in a fixed range), then Kruskal's algorithm runs for practical purposes in linear time.

Lecture 33

Hard graph problems

All of the problems we have considered so far have solutions whose running time is bounded above by a low degree polynomial in the size of the input. However, there are many essential problems that currently do not have known polynomial-time algorithms (so-called ***NP-hard*** problems). Some examples are:

- finding the longest path between two nodes of a digraph;
- finding a k-colouring of a graph, for fixed $k \geq 3$;
- finding a cycle that passes through all the vertices of a graph (a ***Hamiltonian cycle***);
- finding a minimum weight path that passes through all the vertices of a weighted digraph (the ***travelling salesperson problem*** or TSP);
- finding the largest ***independent set*** in a graph, that is, a subset of vertices no two of which are connected by an edge;
- finding the smallest ***vertex cover*** of a graph, that is, a special subset of vertices so that each ~~vertex~~ edge of the graph is adjacent to one in that subset.

Investigating these problems is an active research area in computer science. In many cases the only approach known to the general problem is to try all possibilities, with some rules that prevent the listing of obviously hopeless ones. In some special cases (for example, if a graph is bipartite, the vertex cover problem is solvable in polynomial time) or where the graph is in some sense "close to" a tree, much faster algorithms can be developed.