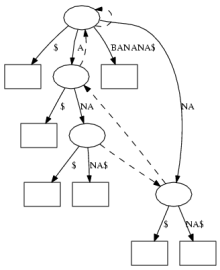
Algorithms and Analysis

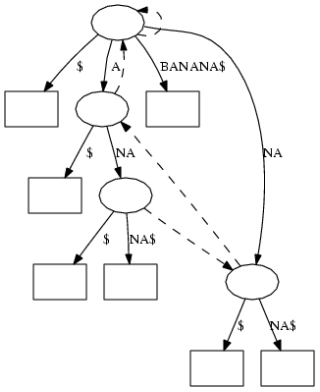
Lesson 1: *Use Data Structures and Algorithms!*



Course structure, examples of data structures and algorithms

Outline

1. **Course structure**
2. Example of Using DSA
3. Sophisticated Program
4. State-of-the-Art



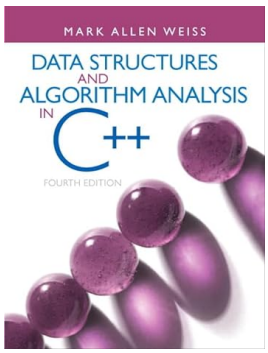
Welcome to Algorithms and Analysis

- Taught by Dr Daniela Mihai and me (Adam Prügel-Bennett)
- I'm teaching you algorithms and data structures in C++
- The analysis is an ability to reason about programming
- Learning C++ will be a joint effort involving *Low-level programming*, me and you
- My ambition is not only to teach you data structures and algorithms academically, but also to get to a new level of coding

Course Structure

- 30ish lectures
- 4 labs (worth 10%)
- 1 coursework (worth 30%)
- Week 6 is a reading week with an in-class exam (worth 10%?)
- Final exam (worth 50%?)

Recommended Course Text



- *Data Structures and Algorithm Analysis in C++* by M. A. Weiss
  - ⋆ Best introduction to Data Structures and Algorithms
  - ⋆ Not huge, but covers all the basics
- Available in the library

What is a Data Structure?

any of various methods of organising data items (as records) in a computer

- Container for data
- E.g. sets, stacks, lists, trees, graphs
- Clean interface, e.g. `push`, `pop`, `delete`
- Usually designed for fast or convenient access

What is an Algorithm?

a sequence of unambiguous instructions for solving a problem, i.e. for obtaining a required output for a legitimate input in a finite amount of time

- E.g. `sort`, `search`, `match`
- Well defined and generic
- Guarantees on performance

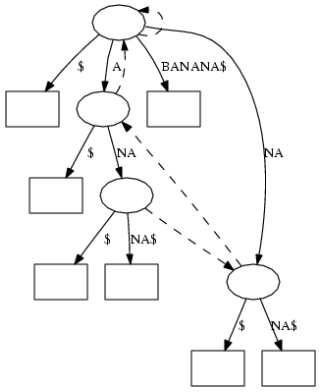
Exemplary OO-Software

- Abstraction from details of problem
- Declaration of intention
- Clean interfaces
- Hidden implementations
- Makes programs readable and maintainable
- Reuse code—don't even have to write it yourself

  Thou shall not re-implement common data structures

Outline

1. Course structure
2. **Example of Using DSA**
3. Sophisticated Program
4. State-of-the-Art



Example: Sort program

- Suppose we want to write a program to
  - read an input file of integers
  - sort the integers
  - write a list of integers to standard out
- In Unix there is a command called `sort` which does just this
- Note that you don't know the number of inputs

Code for sort

```
#include <iostream>
#include <fstream>

int main(int argc, char** argv) {
  std::ifstream myfile(argv[1]);

  int array_size = 10;
  int* array = new int[array_size];
  int cnt = 0;
  while(myfile.good()) {
    if (cnt==array_size) {
      int* new_array = new int[2*array_size];
      for(int i=0; i<array_size; ++i)
        new_array[i] = array[i];
      delete[] array;
      array = new_array;
      array_size *= 2;
    }
    myfile >> array[cnt++];
  }
```



```
  for(int i=0; i<cnt; ++i) {
    int index = 0;
    for(int j=1; j<cnt-i; ++j) {
      if (array[j]<array[index])
        index = j;
    }
    std::cout << array[index] << std::endl;
    array[index] = array[cnt-i-1];
  }
}
```


Notes on Code

- Details of code don't matter
- Simple program ($\sim$ 20 lines of code)
- Uses a simple array
- Difficult to see what is going on
- On 100 000 inputs it takes 10 seconds to run

Using Data Structures and algorithms

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[])
{
  ifstream in(argv[1]);
  vector<int> data;
  copy(istream_iterator<int>(in), istream_iterator<int>(),
       back_inserter(data));
  sort(data.begin(), data.end());
  copy(data.begin(), data.end(), ostream_iterator<int>(cout,"\n"));
}
```


Sorting Doubles

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>
using namespace std;

int main(int argc, char *argv[])
{
  ifstream in(argv[1]);
  vector<double> data;
  copy(istream_iterator<double>(in), istream_iterator<double>(),
       back_inserter(data));
  sort(data.begin(), data.end());
  copy(data.begin(), data.end(), ostream_iterator<double>(cout,"\n"));
}
```


Notes on C++

- `vector<int>` is the C++ standard resizable array
- input/output is treated as a copy
- Code is easy to read
  - Declare `vector<int>` or `vector<double>`
  - copy input file into vector
  - sort vector
  - copy sorted vector to standard output stream
- On 100 000 inputs takes 10ms to run

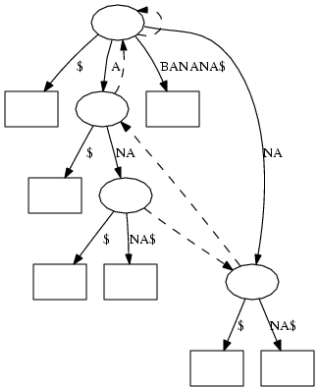
Summary: Why use Data Structures?

Data structure version is

- Easier/quicker to code
- More readable (less bugs)
- Easier to modify and change
- Easier to port to another language
- Better (in this case faster)

Outline

1. Course structure
2. Example of Using DSA
3. **Sophisticated Program**
4. State-of-the-Art



Sophisticated Programs

- Data structures and algorithms allow moderately competent programmers to write some very impressive programs
- E.g. consider a program to count all occurrences of words in a document
- We want to output the words in sorted order

countWords

```
#include <stuff>

int main(int argc, char** argv) {
  ifstream in(argv[1]);
  map<string, int> words;

  string s;
  while(in >> s) {
    ++words[s];
  }

  vector<pair<string,int> > pairs;
  copy(words.begin(), words.end(), back_inserter(pairs));
  sort(pairs.begin(), pairs.end(),
    [](auto& a, auto&b){return a.second>b.second;});

  for(auto w=pairs.begin(); w!=pairs.end(); ++w) {
    cout << w->first << "␣occurs␣" << w->second << "␣times\n";
  }
}
```


Using countWords

```
> countWords text.dat | more
the occurs 97 times
of occurs 96 times
to occurs 57 times
and occurs 42 times
a occurs 36 times
be occurs 31 times
will occurs 26 times
we occurs 23 times
that occurs 23 times
is occurs 21 times
have occurs 19 times
freedom occurs 18 times
```


Programming Challenge

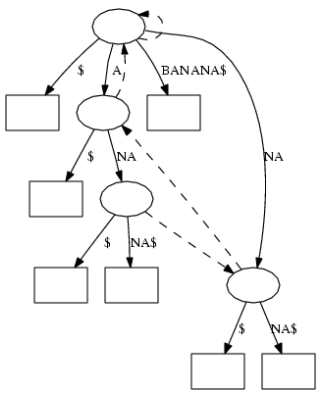
- Run on "I have a dream" speech with 1550 words in 0.02 seconds
- Challenge for good programmers

  Write a program without use data structures in less that 10 times as much code that runs in less than 10 times as long

- Probably possible, but certainly not easy—almost certainly take you 10 times longer to code

Outline

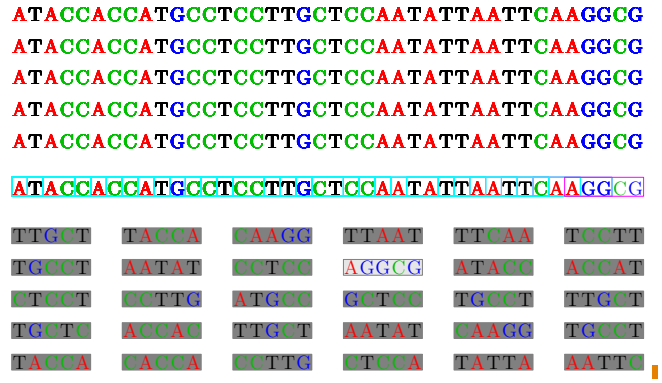
1. Course structure
2. Example of Using DSA
3. Sophisticated Program
4. **State-of-the-Art**



DNA Sequencing

- In modern whole shotgun genome sequencing the full genome is broken into small pieces
- The pieces are then read by a sequencing machine
- This reads short sections (around 1000) bases
- The reads are then assembled to construct the full genome

Sequencing and Assembly



New Generation Sequencers

- The estimated cost of sequencing the human genome in 2005 was $\$10\,000\,000$
- To reduce the cost there was and is a drive to produce new sequencing machines
- These tend to read much shorter sections of DNA (e.g. 20-100nt)
- Can these be assembled?

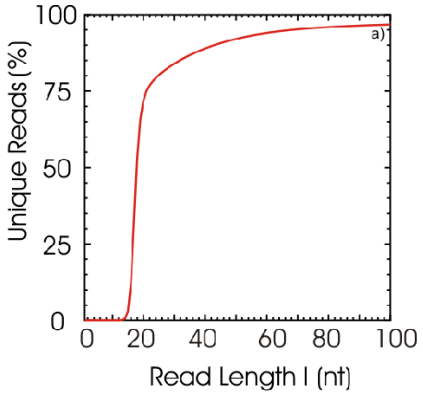
Repeats

- The difficulty of assembly is caused by repeats

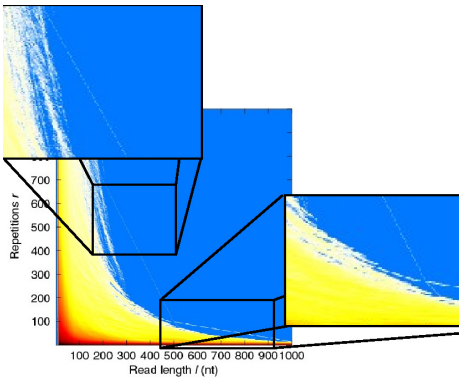
ATACCACCATGCCTCCTTGCTCCAATCCACCATCAAGGCG

- How many repeats are there in the human genome? (Incidentally the human genome is 3.2 billion base **pairs**)
- This is an important question for developing new sequencing technologies

Repeats in Human Genome



Repeats Structure



Computing Repeats

- A naive program would take n^2 operations where $n = 6.4 \times 10^9$
- If we used this we would still be waiting for the program to finish
- Could not answer this question a few years ago—not because computers weren't powerful, but because the algorithms had not been developed
- Used state-of-the-art suffix arrays
- Smart algorithms allow you to do things which you cannot do otherwise

To Use DSA You Need To

- Know what common data structures and algorithm do
- Understand the implementations enough to modify existing data structures to be fit for purpose
- Understand time/space complexity to select the right data structure or algorithm
- Understand software interfaces for DSA
- Be able to combine data structures
- The rest of this course teaches you these skills

Algorithms and Analysis

Lesson 2: *Know How Long A Program Takes*

TSP, Sorting, time complexity, Big-Theta, Big-O, Big-Omega

Outline

1. TSP
2. Sorting
3. Big O

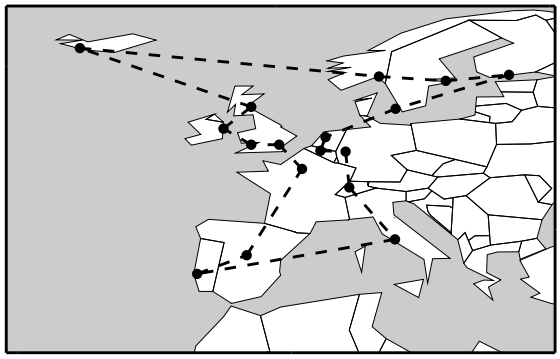
Travelling Salesperson Problem

- Given a set of cities
- A table of distances between cities
- Find the shortest tour which goes through each city and returns to the start

Example of Distance Table

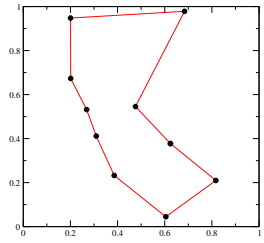
	Lon	Car	Dub	Edin	Reyk	Oslo	Sto	Hel	Cop	Amst	Bru	Bonn	Bern	Rome	Lisb	Madr	Par
London	0	223	470	538	1896	1151	1426	1816	950	349	312	503	743	1429	1587	1265	337
Cardiff	223	0	290	495	1777	1277	1589	1985	1139	564	533	725	927	1600	1492	1233	492
Dublin	470	290	0	350	1497	1267	1628	2026	1239	756	775	956	1207	1886	1638	1449	777
Edinburgh	538	495	350	0	1374	933	1314	1708	984	662	758	896	1243	1931	1964	1728	872
Reykjavik	1896	1777	1497	1374	0	1746	2134	2418	2104	2020	2130	2255	2617	3304	2949	2892	2232
Oslo	1151	1277	1267	933	1746	0	416	788	481	917	1088	1048	1459	2011	2739	2390	1343
Stockholm	1426	1589	1628	1314	2134	416	0	398	518	1126	1281	1181	1542	1978	2987	2593	1543
Helsinki	1816	1985	2026	1708	2418	788	398	0	881	1504	1650	1530	1856	2203	3360	2950	1910
Copenhagen	950	1139	1239	984	2104	481	518	881	0	625	769	662	1036	1538	2479	2076	1030
Amsterdam	349	564	756	662	2020	917	1126	1504	625	0	173	235	629	1296	1860	1480	428
Brussels	312	533	775	758	2130	1088	1281	1650	769	173	0	194	489	1174	1710	1315	262
Bonn	503	725	956	896	2255	1048	1181	1530	662	235	194	0	422	1067	1843	1420	400
Bern	743	927	1207	1243	2617	1459	1542	1856	1036	629	489	422	0	689	1630	1156	440
Rome	1429	1600	1886	1931	3304	2011	1978	2203	1538	1296	1174	1067	689	0	1862	1365	1109
Lisbon	1587	1492	1638	1964	2949	2739	2987	3360	2479	1860	1710	1843	1630	1862	0	500	1452
Madrid	1265	1233	1449	1728	2892	2390	2593	2950	2076	1480	1315	1420	1156	1365	500	0	1054
Paris	337	492	777	872	2232	1343	1543	1910	1030	428	262	400	440	1109	1452	1054	0

Example Tour

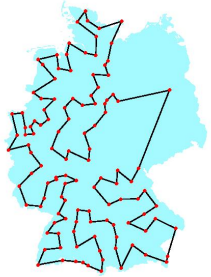


Brute Force

- I wrote a program to solve TSP by enumerating every path and finding the shortest
- I checked that it worked on some problems with 10 cities
- It takes just under half a second to solve this problem
- I set the program running on a 100 city problem—**How long will it take to finish?**

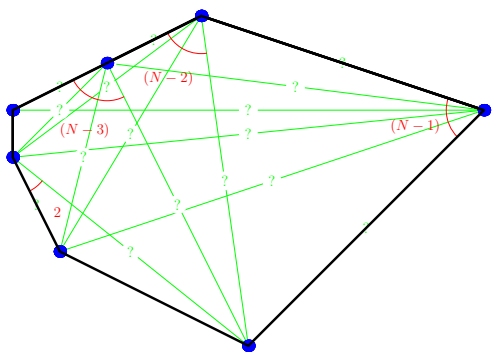


How Many Possible Tours Are There?



- For 100 cities how many possible tours are there?
- It doesn't matter where we start
- Starting from Berlin there are 99 cites we can try next

Counting Tours



Number of tours = $(N-1) \times (N-2) \times (N-3) \times \cdots 2 \times 1 = (N-1)!$

How Long Does It Take?

- The direction we go in is irrelevant
- Total number of tours is $99!/2$
- **Any more guesses how long it will take?**

How Big is 99 Factorial?

- $99! = 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 = ?$
- Upper bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 2 \cdot 1 \\ 99! &< 99 \cdot 99 \cdot 99 \cdots 99 \cdot 99 = 99^{99} \end{aligned}$$

- Lower bound

$$\begin{aligned} 99! &= 99 \cdot 98 \cdot 97 \cdots 50 \cdot 49 \cdots 2 \cdot 1 \\ 99! &> 50 \cdot 50 \cdot 50 \cdots 50 \cdot 1 \cdots 1 \cdot 1 = 50^{50} \end{aligned}$$

How Long Does It Take?

- For $N > 1$

$$\left(\frac{N}{2}\right)^{N/2} < N! < N^N$$

- $99!/2 = 4.666 \times 10^{155}$
- How long does it take to search all possible tours?
  - We computed about $200\,000$ tours in half a second
  - 3.15×10^7sec $= 1$ year
  - Age of Universe ≈ 15 billion years

Answer

- 2.72×10^{132} ages of the universe!
- Incidental

$$\begin{aligned} 99!/2 = {} & 46663107721972076340849619428133350 \\ & 24535798413219081073429648194760879 \\ & 99966149578044707319880782591431268 \\ & 48960413611879125592605458432000000 \\ & 0000000000000000 \end{aligned}$$

Record TSP Solved—$15\,112$ and $24\,978$ Cities

In Case You're Curious

- Number of tours: $15111!/2 = 7.3 \times 10^{56592}$
- Current record $24\,978$ cities with 1.9×10^{98992} tours
- The algorithm for finding the optimum path does not look at every possible path
- If your interested look for the TSP homepage on the web `http://www.math.uwaterloo.ca/tsp/`

Lessons

- Even relatively small problems can take you an astronomical time to solve using simple algorithms
- As a professional programmer you need to have an estimate for how long an algorithm takes—otherwise you can look silly
- For the 100 city problem, if
  - I had 10^{87} cores, one for every particle in the Universe
  - I could compute a tour distance in 3×10^{-24} seconds, the time it takes light to cross a proton
  - It would still take $10^{39}\times$ the age of the universe
- Smart algorithms can make a much larger difference than fast computers!

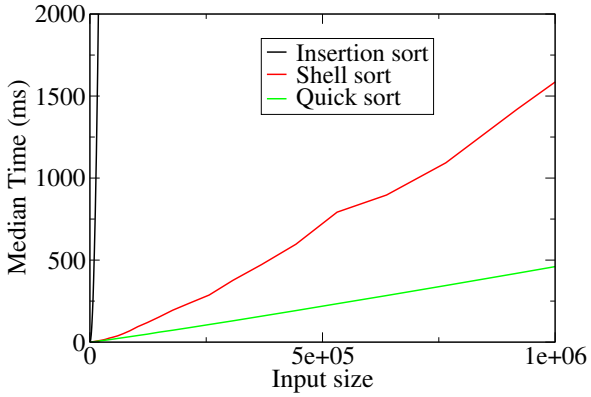
Outline

1. TSP
2. **Sorting**
3. Big O

Sort

- Comparison between common sort algorithms
  - Insertion sort—an easy algorithm to code
  - Shell sort—invented in 1959 by Donald Shell
  - Quick sort—invented in 1961 by Tony Hoare
- These take an array of numbers and returns a sorted array
- Sort is very commonly used algorithm so you care about how long it takes

Empirical Run Times



Lessons

- There is a right and wrong way to do easy problems
- You only really care when you are dealing with large inputs
- Good algorithms are difficult to come up with, but they exist
- We would like to quantify the performance of an algorithm—how much better is quick sort than insertion sort?

Outline

1. TSP
2. Sorting
3. **Big O**

Estimating Run Times

- We would like to estimate the run times of algorithms
- This depends on the hardware (how fast is your computer)
- We could count the number of elementary operations, but
  - different machines have different elementary operations
  - many algorithms use complex functions such as `sqrt` (matrix inversion using Cholesky decomposition) or `sin` and `cos` (FFT)
  - would need to count memory accesses which you shouldn't need to think about
  - code after compiling can be very different from code before compiling

Engineering Solution

- Compute the **asymptotic leading functional behaviour**
- Lets take that statement to pieces
- Suppose we have an algorithm that takes $4n^2 + 12n + 199$ operations (clock cycles)
  - **asymptotic**: what happens when n becomes very large
  - **leading**: ignore the $12n + 199$ part as it is dominated by $4n^2$ (i.e. for large enough n we have $4n^2 \gg 12n + 199$)
  - **functional behaviour**: ignore the constant 4
- We call this an order n^2, or quadratic time, algorithm
- We can write this in 'Big-Theta' notation as $\Theta(n^2)$
- This notion of 'run time' is known as **time complexity**

Advantages of Big-Theta Notation

- Doesn't depend on what computer we are running
- Don't need to know how many elementary operations are required for a non-elementary operation
- Can estimate run times by measuring run time on a small problem
  - If I have a $\Theta(n^2)$ algorithm
  - It takes x seconds on an input of 100
  - It will take about $\frac{x \times n^2}{100^2}$ seconds on a problem of size n
    ($T(100) \approx c\,100^2 = x$ therefore $c = x/100^2$
    thus $T(n) = c\,n^2 = x\,n^2/100^2$)

Counting Instructions

- Big-Theta run times are often easy to calculate
- a $\Theta(n)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
}
// clean up
```

- a $\Theta(n^2)$ algorithm

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  for (int j=0; j<n; j++) {
    // do other stuff
  }
}
// clean up
```

Disadvantage with Big-Theta notation

- Can't compare algorithms with the same Big-Theta time complexity
- For small inputs Big-Theta time complexity can be misleading. E.g.
  - algorithm A takes $n^3 + 2n^2 + 5$ operations
  - algorithm B takes $20n^2 + 100$ operations
  - algorithm A is $\Theta(n^3)$ and algorithm B is $\Theta(n^2)$
  - algorithm A is faster than algorithm B for $n < 18$

  but who cares?
- In some cases Big-Theta time complexity is hard to compute

Not So Sure

- Some algorithms are harder to compute

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- Time complexity now depends on the `if` statement
- If the condition is often satisfied we have a $\Theta(n^2)$ algorithm
- If the condition is true only rarely then we have a $\Theta(n)$ algorithm

Bounds

- To avoid having to think really hard we define upper and lower bounds
- The upper bound we write using **big-O** notation
  - The above algorithm is an $O(n^2)$ algorithm
  - I.e. it runs in no more than order n^2 operations
- The lower bound we write using **big-Omega** notation
  - The above algorithm is a $\Omega(n)$ algorithm
  - I.e. it runs in no less than order n operations

Precise Definitions of $O(n)$

- An algorithm that runs in $f(n)$ operations is $O(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g.. $f(n) = 3\,n^2 + 2\,n + 12$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n^2} = 3 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = O(n^2)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n^3} = 0 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = O(n^3)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3\,n^2 + 2\,n + 12}{n} = \infty \quad \Rightarrow 3\,n^2 + 2\,n + 12 \neq O(n)$$

Lower Bound Definition

- An algorithm that runs in $f(n)$ operations is $\Omega(g(n))$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a constant (could be zero)}$$

- E.g. $f(n) = 3\,n^2 + 2\,n + 12$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^2}{3\,n^2 + 2\,n + 12} = \frac{1}{3} \Rightarrow 3\,n^2 + 2\,n + 12 = \Omega(n^2)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n^3}{3\,n^2 + 2\,n + 12} = \infty \Rightarrow 3\,n^2 + 2\,n + 12 \neq \Omega(n^3)$$

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = \lim_{n\to\infty} \frac{n}{3\,n^2 + 2\,n + 12} = 0 \quad \Rightarrow 3\,n^2 + 2\,n + 12 = \Omega(n)$$

Big-Theta

- An algorithm that runs in $f(n)$ operations is $\Theta(g(n))$ if

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c \qquad \text{where } c \text{ is a non-zero constant}$$

- That is, $f(n) = \Theta(g(n))$ if

$$f(n) = O(g(n)) \quad \text{and} \quad f(n) = \Omega(g(n))$$

- I.e. the lower bound is identical to the upper bound
- Often the most straightforward way of obtaining big-Theta is to show the upper and lower bounds are the same

Use and Misuse

- Note: big-O notation is most commonly used
- often people say they have a $O(n^2)$ when in fact they mean they have a $\Theta(n^2)$ algorithm (a much stronger result)
- Note that an $O(n^2)$ algorithm is also a $O(n^3)$ algorithm
- Strictly a $O(n^2)$ algorithm **may not** be faster than a $O(n^3)$ algorithm when n becomes larger
- A $\Theta(n^2)$ algorithm **will** be faster than a $\Theta(n^3)$ algorithm when n becomes larger

Lessons to Learn

- Run times (computational time complexity) matters
- Choosing an algorithm with the best time complexity is important
- Understand the meaning of big-Theta, big-O and big-Omega
- Know how to estimate time complexity for simple algorithms (loop counting)

Algorithms and Analysis

Lesson 3: ***Declare your intentions (not your actions)***

ADTs, stacks, queues, priority queues, sets, maps

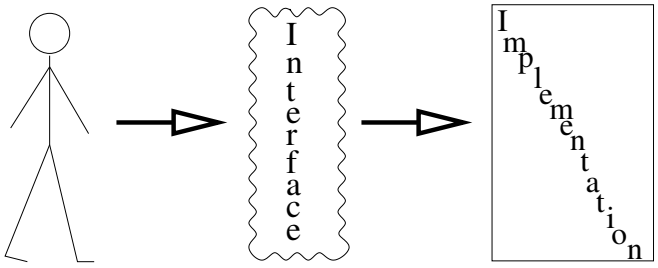
Outline

1. **Abstract Data Types (ADTs)**
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together

Object Oriented Programming

- OO-programming allows you to build large systems reliably
- In the OO-methodology you separate the interface from the implementation
- The **interface** is the public methods (functions) of a class
- The implementation is hidden (**encapsulated**) and may be changed without affecting how the class is used
- There exist other ways of programming, but C++ is designed to support the OO-methodology—for building systems it is brilliant

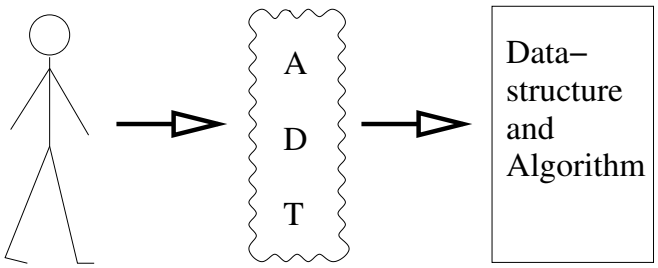
Object-Oriented Classes



Abstract Data Types

- With data structures there are some traditional interfaces called **Abstract Data Types** or ADTs
- These are implementation free data structures
- They are mathematical abstractions of the data structure
- Their purpose is to allow you to declare you intentions
- You are entering into an agreement that you only intend to use the underlying data structure in the way specified by the interface

ADTs



Say it with an ADT

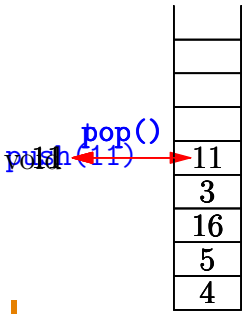
- Common ADTs include stacks, queues, priority queues, sets, multisets and maps
- There are many possible implementations of these ADTs (some far from obvious)
- Each ADT has a limited set of methods associated with it
- They are an abstraction away from the implementation
- By declaring your intentions you are making your code easier to understand and maintain

Outline

1. Abstract Data Types (ADTs)
2. **Stacks**
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. Putting it Together

Stacks

- Last In First Out (LIFO) memory
- Standard functions
  - `push(item)`
  - `T top()`
  - `T pop()` except in C++ `pop()` doesn't return the top of the stack
  - `boolean empty()`
- Implemented using an array (or a linked-list)



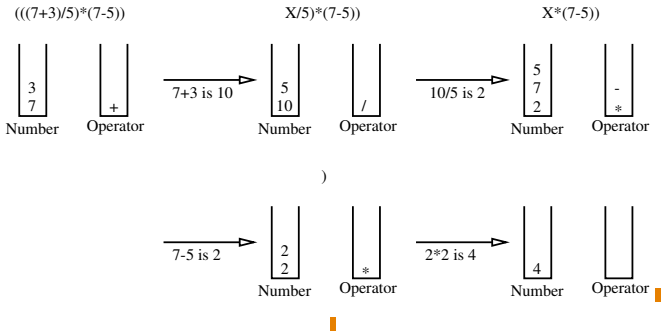
Why Use a Stack?

- Stacks reduces the access to memory—no longer random access
- Seems counter intuitive to reduce what you can do
- Gives you a very simple interface
- Prevents another programmer from using memory in a way that will break existing code
- Sufficient for large number of algorithms

Uses of Stacks

- Reversing an array
- Parsing expression for compilers
  - balancing parentheses
  - matching XML tags
  - evaluating arithmetic expression
- Clustering algorithm

Evaluating Arithmetic Expressions

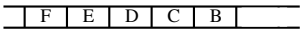


Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. **Queues and Priority Queues**
4. Lists, Sets and Maps
5. Putting it Together

Queues

- First-in-first-out (FIFO) memory model
- `enqueue(elem)`
- `peek()`
- `dequeue()`
- C++ has a double ended queue (`deque`) with `push_front()`, `push_back()`, etc.

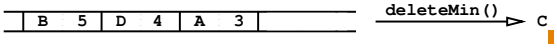


Uses of Queues

- Queues are heavily used in multi-threaded applications (e.g. operating systems)
- Multi-threaded applications need to minimise waiting and ensure the integrity of the data structure (for instance when an exception is thrown)
- Because of this they are more complicated than most data structures
- They can be implemented using linked-lists or circular arrays

Priority Queues

- Queue with priorities
- `insert(elem,priority)` (in C++ `push()`)
- `findMin()` (in C++ `top()`)
- `deleteMin()` (in C++ `pop()`)



Uses of Priority Queues

- Queues with priorities (e.g. which threads should run)
- Real time simulation
- Often used in "greedy algorithms"
  - Huffman encoding
  - Prim's minimum spanning tree algorithm

Implementation of Priority Queue

- Could be implemented using a binary tree or linked list
- Most efficient implementation uses a heap
- A heap is a binary tree implemented using an array

Outline

1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. **Lists, Sets and Maps**
5. Putting it Together

Lists

- In C++ the standard list is known as `vector<T>`
- That is, it is a collection where the order in which you put items into the list counts
- You can have repetitions of elements
- It has random access, e.g. `v[i]`
- You can `push_back(i)`, `insert`, `erase`, etc.
- C++ has a linked list class `list<T>`

Sets

- Models mathematical sets
- Container with no ordering or repetitions
- Methods include `insert`, `find`, `size`, `erase`
- Provides fast search (`find`)
- This is the class to use when you have to rapidly find whether an object is in the set or not—don't use a list like `vector<T>`!

Iterators

- Wish to act on all members of the set
- Performed using an `iterator`
- Iterators are used by many collections
- In C++ iterators follow the pointer convention

```
set<string> words;

words.insert("hello");
words.insert("world");

for(auto iter = words.begin(); iter != words.end(); ++iter) {
  cout << *iter << endl;
}
```


Implementation of Sets

- Sets are very important and there are many implementations depending on their usage
- Two common implementations of sets are
  - hash tables: `unordered_set<T>`
  - binary trees: `set<T>`
- Which is most efficient depends on the application
- Binary trees allow you to iterate in order (iterating over a hash table will give you outputs in random order)
- `multiset<T>` are sets with repetition

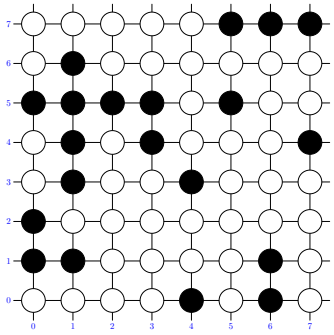
Maps

- A map provides a content addressable memory for pairs *key: value*
- It provides fast access to the *value* through the *key*
- Implement as tree or hash table
- Multimaps allows different data to be stored with the same keyword

Outline

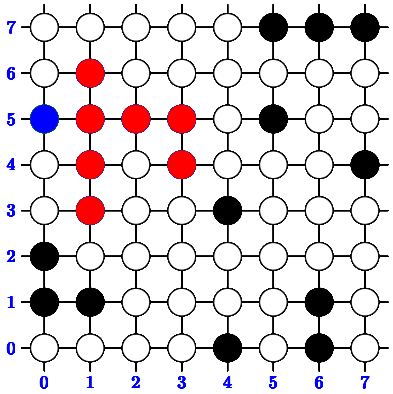
1. Abstract Data Types (ADTs)
2. Stacks
3. Queues and Priority Queues
4. Lists, Sets and Maps
5. **Putting it Together**

Connected Nodes



- A frequent problem is to find clusters of connected cells
- Applications in computer vision, computer go, graph connectedness, . . .

Connected Nodes



next = (2,5)

uncheckedNodes =

(1,6)
(3,4)
(0,5)

clusterNodes =
{(2,5),(1,5),(3,5),}
(3,4),}(0,5),(1,4),
(1,6),}(1,3)}

Connected Node Algorithm

```
set<Node> findCluster(Node startNode, Graph graph)
{
  stack<Node> uncheckedNodes;
  set<Node> clusterNodes;

  uncheckedNodes.push(startNode);
  clusterNodes.insert(startNode);

  while (!uncheckedNodes.empty()) {
    Node next = uncheckedNodes.top();   uncheckedNodes.pop();
    vector<Node> neighbours = graph.getNeighbours(next);
    for (Node neigh: neighbours) {
      if (graph.isOccupied(neigh) && !clusterNodes.contains(neigh) ) {
        uncheckedNodes.push(neigh);
        clusterNodes.insert(neigh);
      }
    }
  }

  return clusterNodes;
}
```


Lessons

- Abstract Data Types (ADT) are interfaces to data
- Their purpose is to allow the programmer to declare their intentions
- They often have different implementations with different properties
- The most efficient implementation is not always obvious—we will see many of these implementations as we go through this course
- You need to know the common ADTs (e.g. Stack, Queue, List, Set, Map) and how and when to use them

Algorithms and Analysis

Lesson 4: *C++ 101*

C with classes, new, overloading, templates

Outline

1. C with Classes
2. New
3. Overloading
4. Templates

C

- C was developed in the 1970s by Dennis Ritchie for writing UNIX tools
- It supported structural programming through functions
- It allowed run-time allocation of memory (through `malloc` and `free`)
- It allowed manipulation of memory through pointers
- This made it efficient, but not safe or easy to use

Keeping Things Together

- As soon as you start programming bigger systems you want to keep information together
- C facilitated this through C structures `struct`

```
struct MyStructure {    // Structure declaration
  int myNum;            // Member (int variable)
  char myLetter;        // Member (char variable)
}; // End the structure with a semicolon

int main() {
  struct myStructure s1;

  s1.myNum = 13;
  s1.myLetter = 'B';

  printf("My number: %d\n", s1.myNum);
  printf("My letter: %c\n", s1.myLetter);
  return 0;
}
```


Estimated Errors in the Mean

- When working with empirical data, $\{X_i, i = 1, 2, \ldots, n\}$, we want to compute the *mean* and *variance* (from which we can estimate the error in the mean)
- We can do this on the fly by storing

$$n, \qquad \hat{\mu}_n = \frac{1}{n}\sum_{i=1}^{n} X_i, \qquad Q_n = \sum_{i=1}^{n}(X_i - \hat{\mu}_n)^2$$

- Given X_{n+1} we can update our data using

$$\Delta = \frac{X_{n+1} - \hat{\mu}_n}{n+1}, \quad Q_{n+1} = Q_n + n\,\Delta\,(X_{n+1} - \hat{\mu}_n), \quad \hat{\mu}_{n+1} = \hat{\mu}_n + \Delta$$

this requires the back of an envelop to verify

Second Order Statistics in C

- In C we can use a `struct` to keep this data together

```
struct Sos {
  unsigned n;
  double mu;
  double Q;
};
```

- We can write functions that update thos

```
void add(struct Sos& sos, double x) {
  double delta = = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}
```


Classes

- C++ was developed by Bjarne Stroustrup and released in 1985 as "C with classes"
- It was syntactic sugar that compiled down to C (as such if was intended to be as fast as C)
- You are familiar with classes from python and they are very much the same thing, except C++ is a lot more elegant than python
- It has grown since 1985, adding templates and a lot of nice functionality

Classes by Example

- Define programme in header file `sos.h`

```
class Sos {
private:                // encapsulate
  int n;
  double mu;
  double Q;

public:                 // interface
  Sos();                // constructor
  void add(double x);   // add data
  double mean();        // return mean
  double var();         // unbiased estimate of variance
  double error();       // estimated error in mean
}
```

Implementation of sos.cc

```
Sos::Sos() {n=0; mu=0.0; Q=0.0;}

void Sos::add(double x) {
  double delta = = (x - mu)/(n+1.0);
  Q += n*delta*(x - mu);
  mu += delta;
  n++;
}

double Sos::mean() const {return mu;}

double Sos::var() const
{
  assert(n>1.0);
  return nvar/(n-1.0);
}

double error() const
{
  sqrt(var()/n);
}
```


Using Classes

- Classes are easy to use

```
#include "sos.h"
#include <iostream>
using namespace std;

void main() {
  Sos mean;
  for(int i=0; i<n; ++i) {
    // compute X
    mean.add(X);
  }
  cout << mean.mean() << ' ' << mean.error() << endl;
}
```

- `Sos` is the class that I use most (both in C++ and python)

Libraries

- C++ comes with a lot of in built libraries
- I include libraries using include statements

```
#include <iostream>
#include <vector>
```

- This is the same as C, but the C++ libraries don't have ".h
- These are known as the standard library or the standard template library

Namespaces

- When you are writing very large programmes (possibly involving other peoples code) you might accidentally use the same name for a class, function or variable used elsewhere
- If you are luck this won't compile, or crash. If you are unlucky you will have a weird bug that will be very difficult to find
- To prevent this, C++ invented a new scope called **namespaces**
- By default all the standard library classes and functions are in namespace `std`
- To call the library we write `std::vector<double>`
- We can be lazy and write `using namespace std;`

Print

- Rather than pesky `printf` statements C++ allows us to use the opeartor <<
- When you get used to it, you will love if

```
#include <iostream>      // header file the defines library
using namespace std;

void main() {
  int i = 5;
  double x = 3.3;

  cout << "hello there" << i << ' ' << x << endl;
}
```


Outline

1. C with Classes
2. New
3. Overloading
4. Templates

Pointers

- In C and C++ we can access an object through its memory address

```
int a = 5;    // creates an object a with value 5
int* b = &a;  // b is the memory address of object a
*b = 6        // *b is now a pseudonym for a
```

- `b` is called a pointer
- The *dereferencing* operator * turns the pointer back into the object

New Object

- The operator **new** will create an object and return a reference

```
Widget w(arg);                 // w is an instance of class Widget
Widget* wpt = new Widget(args); // pointer to instance of class Widget
```

- To call a member function of `wp` use either

```
(*wpt).func();  // dereference object and call member function
wpt->func();    // easy to type
```

Inheritence

- C++ allows classes to inherit from other classes
- Suppose `Square` and `Circle` inherits from `Shape`
- If `Shape` has a (virtual) member function `area` then `Square` and `Circle` can redefine this

```
class Square: public Shape {
  private:
    double l;

  public:
    Square(double len) {l=len;}  // constructor
    double area() {return l*l;}  // define area
}
```


Polymorphism

- Polymorphism is a way of using inheritance where we instantiate a parent pointer with a child class

```
Shape* shape = new Square(2.5);

cout << shape->area() << endl;
```

- This provides a clean way of choosing a behaviour depending on the object type
- It is used in *iterator*s which we will come to later in the course

Arrays

- C++ also uses **new** to return arrays (in place of malloc)

```
int* pt = new int[20];
```

creates a pointer to memory location where we can store 20 integers

- We can dereference the i^{th} element using `pt[i]` (which is equivalent to `*(pt+i)`)—this is the same as C
- We can free this up with

```
delete[] pt;
```


References

- C and C++ also provides references

```
int a = 5;          // create a memory location called a
int& b = a;         // b is a pseudonym for a
b = 6               // both b and a are now 6
```

- References are like dereferenced pointers
- There are many uses of references, one is so we can make functions change their value

```
void f(int x) {x += 6;}   // define function f

void g(int& x) {x += 2;}  // define function g

int a = 5;

f(a);                      // does nothing a=5
g(a);                      // now a=7
```


Saving Copying

- When we declare a function `f(Widget w)` then widget `w` is copied to the function (this is known as passed by value)
- If widget is big, even if we don't want to change it we might not want to copy it

```
void f(const Widget& w);
void g(Widget w);
```

- In both cases `w` is a Widget, but function `f` avoids copying its input

Outline

1. C with Classes
2. New
3. **Overloading**
4. Templates

Overloading

- C and C++ allow you to define different functions with the same name but different arguments

```
void func(int a);      // called if argument is an int
void func(double a);   // called if argument is a double
```

- Needs to be used sensibly, but provides flexibility

Example

- In the second order statistics class we could define a member function

```
void add(const Sos& rhs);
```

- With an implementation

```
void Sos::add(const Sos& rhs)
{
  double total = n + rhs.n;
  double diff = rhs.mu-mu;
  mu += rhs.n*diff/total;
  Q += rhs.Q + n*rhs.n*diff*diff/total;
  n = total;

  return rhs;
}
```

Overloading Continued

- This allows us to add second order statistics

```
Sos total;
for(int i=0; i<10; ++i) {
  Sos local;
  for(int j=0; j<100; ++j) {
    // compute X
    cout << local.mean() << ',' << local.error() << endl;
    local.add()
  }
  total.add(local)
  cout << total.mean() << ',' << total.error() << endl;
}
```


Opeartor Overloading

- C++ like python allows us to overload operators
- Rather than using add I might prefer to use

```
class Sos {
  ...
  double operator+=(double x) { add(x); return(x); }
}
```

- Then we can write

```
Sos sos;
sos += X;
```


Overloading <<

- To print an object of type `Sos` we define

```
ostream& operator<<(ostream& out, const Sos& d)
{
  out << d.mean() << "␣" << d.error();
  return(out);
}
```

- We can then print

```
Sos sos;
...

cout << sos << endl;
```

- I've made `sos.h` and `sos.cc` available on the web site—I use them a lot, you might want to keep them around

Outline

1. C with Classes
2. New
3. Overloading
4. **Templates**

Templates

- Many algorithms and data structures can be applied to a wide range of types

```
vector<double> double_vec; // resizable array of doubles
vector<int>    int_vec;    // resizable array of int
map<string, int>  mymap    // map with string keys and int value
```

- C++ allows us to define a template class

```
template <typename T>
class myclass {
  private T data;
}
```


Templates

- Templates work very simply
- They provide a template for same type (e.g. `T`)
- When you ask for an instance of that object

```
myclass<int> instance;
```

  the C++ compiler takes your template and substitutes the `T` with **`int`**

- This is both simple and powerful

Template Functions

- As well as classes I can create template functions

```
template <typename T>
T accumulate(const vector<T>& vec) {
  T sum = 0;
  for(int i=0; i<vec.size(); ++i) {
    sum += vec[i];
  }
  return sum
}
```

- This will work with `vector<int>`, `vector<double>`

Summary

- C++ is a rich language
- You should learn some C++ in low-level programming
- There are a lot of resources
- I'm afraid you will only get good at it by writing programs
- The lab session are to help you learn C++

Algorithms and Analysis

Lesson 5: *Use Arrays*

Variable length arrays, implementing stacks

Outline

1. **Why Arrays?**
2. Variable Length Arrays
3. Programming Language
4. Implementing Stacks

Use Arrays

- An array is a contiguous chunk of memory
- In C we can create arrays using
  `int *array = new int[20]`
- The array has an access time of $\Theta(1)$
- The constant factor is small (i.e. access time ≈ 1 time step)
- Arrays provide a very efficient use of memory
- 95% of the time using arrays is going to give you the best performance, although never use raw arrays!

Disadvantages of Arrays

- Arrays have a fixed length
- Very often we don't know how big an array we want
  - E.g. reading words from a file
- Adding or deleting elements from the middle of an array is costly
- Sorted arrays are expensive to maintain
- Arrays don't know how big they are—annoying

Outline

1. Why Arrays?
2. **Variable Length Arrays**
3. Programming Language
4. Implementing Stacks

Variable Length Arrays

- We want a variable length array
- Initially a variable length array would have length zero
- We should be able to
  - Add an element to an array
  - Access any element in the array
  - Change an element
  - Delete elements
  - Know how many elements we have

ADT for a List

- What do we want of a list of **int**s?
  - `void push_back(int value)`
  - random access `array[i]`
  - `int size()`
- It would be useful if it resized
- It would be great to have some algorithms (e.g. sort) that can be run on a list

Implementation

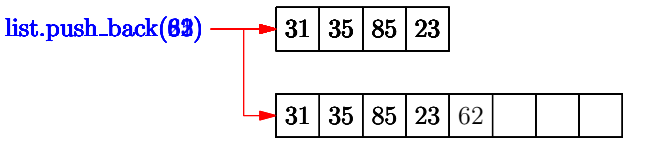
- How should we implement a list?
- Use an array, of course!
- We need to distinguish between
  - the number of elements in the list `size()`
  - the number of elements in the array `capacity()`
- If the number of elements grows larger than the capacity then we need to increase the capacity

Initial Capacity

- We could prevent resizing arrays by using a huge initial capacity
- However, how big is big enough?
- What happens when we have an array of arrays?
- Memory like time is resource we should care about
- In an analogy with **time complexity** we also care about **space complexity** (i.e. how much memory we need)
- If we want to store n elements it is reasonable to expect that we use cn bits of memory where we want to keep c small

Resizing Memory

- We start with some reasonable capacity
- We can add elements until we reach the capacity
- A simple method for resizing memory is
  - create a new array with double the capacity of the old array
  - copy the existing elements from the old array to the new array



Amortised Time Analysis

- How efficient is resizing?
- Most `push_back(elem)` operations are $\Theta(1)$
- When we are at full capacity we have to copy all elements
- Adding to a full array is slow but it is **amortised** by other quick adds

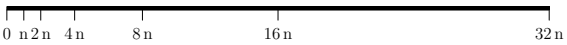
  amortised: effect of a single operations 'deadened' by other operations

Example

- If we have an initial capacity of 10 and add 100 elements then the number of operations needed is
  - adds: 100
  - copies: 10+20+40+80=150
  - **new int**[]: 4
- 250 adds and copies operations + 4 **new** operations

General Time Analysis

- If we perform N adds with an initial capacity of n
- We must perform m copies where



$$n \times 2^{m-1} < N \leq n \times 2^m \quad \text{i.e.} \quad m = \left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil$$

- The number of elements copied is

$$n + 2n + 4n + \cdots + 2^{m-1}n = n(1 + 2 + \cdots + 2^{m-1}) = n\,(2^m - 1)$$

- Total number of operations is (using $\lceil \log(a) \rceil < \log(a) + 1$)

$$N + n\,(2^m - 1) = N + n2^{\left\lceil \log_2 \left(\frac{N}{n} \right) \right\rceil} - n < N + 2N - n < 3N$$

Insertion and Deletion

- `vector<T>` is very useful and very fast for lots of things
- But if you try to insert or delete an element anywhere other than the end then you have to shove all the subsequent elements one space forward
- This is not the right data structure if you want to keep elements in order (binary trees will do that for you much more efficiently)
- Linked lists allow you to splice in a sublist into a list in constant time although linked lists have a lot of drawbacks

Outline

1. Why Arrays?
2. Variable Length Arrays
3. **Programming Language**
4. Implementing Stacks

Computer Languages

- Different computer languages are designed for different roles and have different advantages and disadvantages
- **C++** was designed to be fast (as fast as C), it pays the price of allowing bugs that hard to detect
- **Java** was designed to be vary safe (avoiding lots of bugs), but is not fast and a bit long winded
- **Python** was designed so you can rapidly write powerful programmes with a small amount of code, but it is not fast or safe

Problems with C++

- Amongst a number of issues that make C++ dangerous are
  - Memory management
  - Writing to parts of memory that you should not
  - Multiple inheritance, although you seldom need to do this
- However, by using existing data structures (STL) and following established programming patterns these don't have to be an issue

Memory Management

- Most programming languages have two types of memory

  The Stack: is the area of memory controlled by compiler for local variables, function calls, etc.

  The Heap: is area that the programmer (you) can request, which is nice
- In C++ you are given the **right** to ask for memory

```
int *storage = new int[n];
```

- You have **responsibility** to free the memory

```
delete[] storage;
```


Trouble with Memory Management

- If you don't release memory acquired with `new` using `delete` you cause a **memory leak**
- Often memory leaks are no concern, but in large programs memory leaks will rapid exhaust the computer's memory, slowing down the code and eventually leading to the programme crashing
- To release a block of memory we can use: **`delete`**`[] storage;`
- Now `storage` is a **dangling pointer** and must not be used as it is no longer valid
- If we accidentally delete the storage twice we get an *undefined behaviour*, but often the programme will crash

Resource Acquistion is Initialisation (RAII)

- Java and Python use garbage collectors which automatically checks whether memory can be accessed and if not it is removed
- In C++ this is your responsibility
- But there is a standard **programming pattern** to elevate the problem known as **Resource Acquistion is Initialisation (RAII)**

  Wrap all resources in classes. Request the resources in the constructor and release the resource in the destructor
- When the object goes out of scope (you leave a `for` loop, function call, etc.) the destructor is called and the resource is safely released

RAII

```
template <typename T>
class container {
  private:
    T* data;

  public:
    container(unsigned n) {data = new T[n];}
    ~container(unsigned n) {delete[] data;}
};


  main() {

    for (int i=0; i<1000; ++i) {
      container<int> my_container(10000);
      // do something
    }

  }
```


Writing over Memory

- In C++ the following will compile and run

```
int *array = new int[4];
int *a = new int[2];
double *darray = new double[4];
array[4] = 4;
```

- However `array[4]` has not been assigned (unlike `array[0]`, `array[1]`, `array[2]` and `array[3]`)
- The memory on the heap corresponding to the address of `array[4]` might have been assigned to `a[0]` in which case you may inadvertently have set `a[0]` to 4 leading to the program not doing what you want
- It might be that you have put an `int` into `darray[0]` which will then crash the system when you read `darray[0]`

Guarding Against Mistakes

- These are really hard problems to debug because where the program goes wrong or crashes can be very far from the assignment that caused the error
- Java takes the approach that it always tests whether you are writing in valid memory
- By default C++ doesn't even for data structures—making this check slows down random access
- Checks can also make pipeline optimisations harder to make
- The onus is on the user to use the memory correctly

Follow Programming Idioms

- Using common data structures and following common idioms will prevent most errors

```
int n = 5;
vector<int> array(n);

for(int i=0; i<array.size(); ++i) {
  array[i] = i;
}

for(auto pt=array.begin(); pt != array.end(); ++pt){
  *pt *= 2
}

for(int& element: array) {
  element += 2;
}
```

Outline

1. Why Arrays?
2. Variable Length Arrays
3. Programming Language
4. **Implementing Stacks**

Stacks

- Lets look at implementing a stack
- Remember a stack has methods
  - `push(Object )`
  - `pop()`
  - `top()`
  - `empty()`

Implementation of Stack

```
template <typename T>
class MyStack
{
private:
  std::vector<T> stack;

public:
  void push(const T& obj) {stack.push_back(obj);}

  T top() const {return stack.back();}

  T pop() {
    T tmp = stack.back();
    stack.pop_back();
    return tmp;
  }

  T empty() {return stack.size()==0;}
};
```


Notes on Implementation

- I don't need to write a constructor as C++ generates a default constructor that will initialise the stack correctly
- I don't need to write a desctuctor because by default the destructor for `vector<T>` will be called which releases memory
- I've written the `pop` command, that I like, but if I run

```
stack<Widget> widget_stack;
Widget w;
widget_stack.push(w);
Widget w1(widget.pop());
```

  if the last command throws an exception then the last term on the stack is lost for ever

Why not use a `vector`

- Surely it is mad to use `MyStack<T>` as I could just use the more powerful `vector<T>`
- I can make `MyStack<T>` as efficient as `vector<T>` by inlining function calls
- But why would I want to lose functional?
- By using `MyStack<T>` I am **declaring my intention** of using this data structure as a stack
- I'm not going to do something weird like modify an element inside the stack
- My code becomes self-explanatory—I don't need to write comments as it is clear what I am doing

Using `MyStack`

- Implementing a stack using a dynamically re-sizable array is trivial
- Stacks have many applications
- E.g. suppose we want to write a program to reverse the order of strings in a file

you can't swallow a cage can't you

you
can't
swallow
a
cage
can
you

Reversing Strings in File

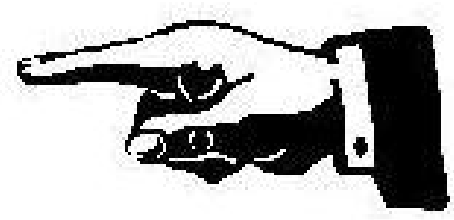
```
#include <stack>
#include <iostream>
#include <fstream>
using namespace std;

int main(int argc, char *argv[]) {
  ifstream in(argv[1]);

  stack<string> stack;

  string word;
  while (in >> word)
    stack.push(word);

  while(!stack.empty()) {
      cout << stack.top() << '_';
      stack.pop();
  }
}
```


Lessons

- Arrays are very efficient both in space (memory) and access time
- Resizing an array is not that costly
- insertion and deletion are expensive, $O(n)$
- Arrays are often the simplest way to implement many other data structures, e.g. stacks
- Use (dynamically re-sizable) arrays (`vector<T>`) frequently!
- Stop using raw arrays

Algorithms and Analysis

Lesson 5: *Point to where you are going: links*



Linked lists

Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. Skip Lists

Non-Contiguous Data

- So far we have considered arrays where the data is stored in a contiguous chunk of memory
- This has the great advantage of allowing random access
- It has the disadvantage that it is expensive to add or remove data from the middle of the list or to rearrange the data
- A different approach is to use units of data that point to other units

Non-Contiguous Data Structures

- There are a lot of important data structures using non-contiguous memory
  - Binary trees
  - Graphs
- In this lecture we consider **linked-lists**
- This is a classic data structure which is almost entirely useless
- However, it serves as a good introduction to much more useful data structures

Self-Referential Classes

- The building block for a linked list is a node class

```
struct Node<T>
{
  Node(U value, Node<U> *node): value(value), next(node) {}
  T element;
  Node<T> *next;
}
```

- We create new nodes
  `Node<int> *node = new Node<int>(10, pt_to_next)`
- Note that `node` is the address of this node
- I make it a `struct` as this is a class where I want public access to the element and next
- I can make this class a private class of my linked list

Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. Skip Lists

Singly Linked List

- We can build a linked list by stringing nodes together

  We don't show the "pointer" to `element`
- A singly linked list has a single "pointer" to the next element
- A doubly linked list has "pointers" to the next and previous element—we will see this later
- We should be able to create a linked list, add elements, remove elements, see if an element exists, etc.

Implementation

- We consider a lightweight implementation
- The class will have a head, a size counter and have a `Node` as a nested class

```
class MyList {
private:
  template <typename U>
  struct Node{
    Node(U value, Node<U> *node): value(value), next(node) {}
    U value;
    Node<U> *next;
  };
  Node<T> *head;
  unsigned noElements;
```

Simple Methods

- The constructor is simple (and not strictly necessary)

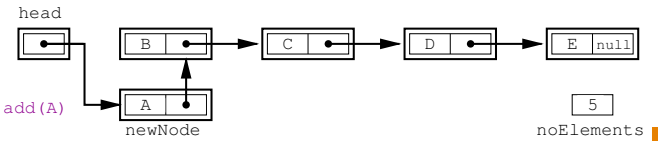
```
MyList(): n(0), head(0) {}
```

- Other simple methods are

```
unsigned size() const {return noElements;}

bool empty() const {
  return head == 0;
}
```

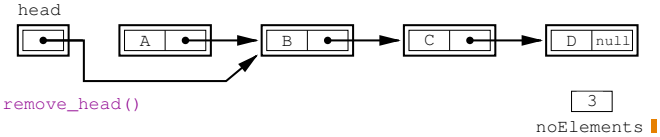

Adding elements

```
void add(T element)
{
    Node<T> *newNode = new Node<E>();
    newNode.element = element;
    newNode.next = head;
    head = newNode;
    noElements++;
    return true;
}
```



Remove Head of List

```
void remove_head()
{
    Node<T>* dead = head;
    head = head->next
    noElements--;
    delete dead;
}
```



Other Methods

- We can easily implement many other methods
  - `get(int i)`—return i^{th} item in list
  - `remove(T obj)`-remove `obj` from list
  - `insert(int position, element)`
- Note that `get(int i)` requires moving down the list so is $O(n)$ (i.e. not random access)

Outline

1. References
2. Singly Linked List
3. **Stacks and Queues**
4. Doubly Linked List
5. Using Linked Lists
6. Skip Lists

Stack

- It is easy to implement a stack using a linked list

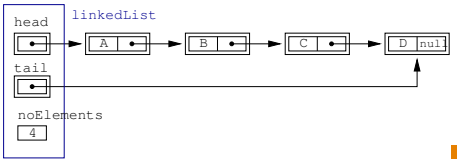
```
template <typename T>
class Stack<E>
{
  private Mylist<T> list = new mylist<T>();

  boolean push(E obj) {list.add(obj);}

  E top() {return list.get_head();} // throw exception

  E pop() {
    T tmp = list.get_head();
    list.remove_head();
    return tmp;
  }

  boolean empty() {return list.empty();}
}
```


Complexity of Stack

- All operations of the stack is constant time, i.e. $O(1)$
- This is the same time complexity as an array implementation
- Memory requirement is approximately $2 \times n$ reference and n objects—same as worst case for an array
- However, hidden cost of creating and destroying `Node` objects
- The array implementation is therefore slightly faster

Point to the Back

- To find the end of the queue takes n jumps
- Thus our linked list isn't the right data structure to implement a queue
- However, we could include a pointer to the end of the queue



Implementing a Queue

- We can then add elements to the tail in constant time
- We can the implement a queue in $O(1)$ time by
  - `enqueue`ing at the back
  - `dequeue`ing at the head
- I leave the implementation details as an exercise for you
- Note that although adding an element to the tail is constant time, removing an element from the tail is $O(n)$ as we have to find the new tail

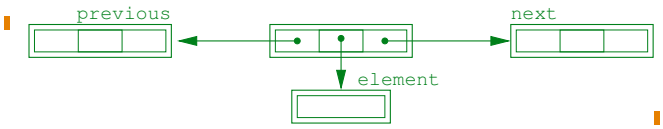
Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. **Doubly Linked List**
5. Using Linked Lists
6. Skip Lists

Doubly linked list

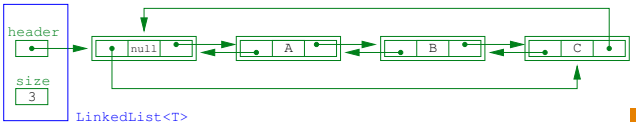
- In a more powerful linked list we would like to navigate the list in either direction
- To achieve this it uses a doubly-linked lists with elements to next and previous

```
class Node<T>
{
  T element;
  Node<T> *next;
  Node<T> *previous;
}
```



Dummy Node

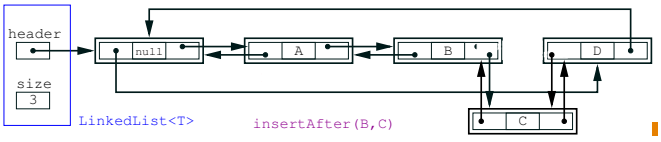
- List includes a dummy node—this make the implementations slicker



- Symmetric data structure so processing head and tail is equally efficient

Time Complexity

- `add` and `remove` from head and tail $O(1)$
- `find` $O(n)$ and slow
- `insert` and `delete` $O(1)$ (faster than an array list) once position is found



Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. **Using Linked Lists**
6. Skip Lists

When To Use Linked Lists

- It is difficult to think of applications where linked lists are the best data structure
- lists—variable length arrays are usually better
- queues—linked list OK, but circular arrays are probably better
- sorted lists—binary trees much better
- linked lists have efficient insertion and deletion but it is difficult to think of an application where this matters

Line Editor

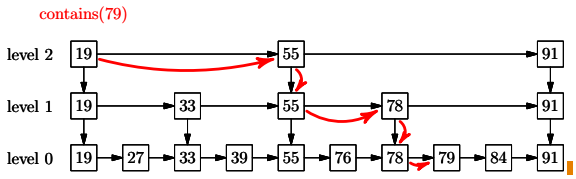
- One application where efficient insertion and deletion matters is a line editor
- We are usually working at a particular location in the text
- We often want to add or delete whole lines
- Storing the lines as strings in a linked list would allow a fairly efficient implementation

Outline

1. References
2. Singly Linked List
3. Stacks and Queues
4. Doubly Linked List
5. Using Linked Lists
6. **Skip Lists**

Skip Lists

- Linked lists have the disadvantage that to get to anywhere in the list takes on average $\Theta(n)$ steps
- Even if you kept an ordered list you still need to traverse it
- Skip lists are hierarchies of linked lists which allow binary search



Efficiency of Skip Lists

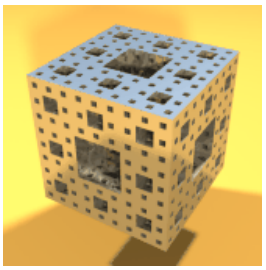
- Skip lists provide $\Theta(\log_2(n))$ search as opposed to $\Theta(n)$
- They have the similar time complexity to binary trees, although binary trees are slightly faster
- They have one advantage over binary trees—they allow efficient concurrent access
- The standard template library provides a doubly linked list, `list<T>` as well as a slingly linked-list `slist<T>`

Lessons

- Node structures that point to other Node structures are used in many important data structures
- Linked lists are the simplest examples of this kind of structure and consequently has a dominant position in most DSA books
- In practice linked lists are seldom the data structure of choice—before choosing to use a linked list consider the alternatives
- There are some important uses for linked lists, e.g. skip lists and hash tables (see lecture on hashing)

Algorithms and Analysis

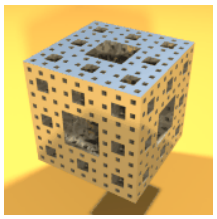
Lesson 6: *Recurse!*



Induction, integer power, towers of hanoi, analysis

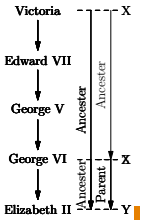
Outline

1. Simple Recursion
2. Programming Recursively
   - Simple Examples
   - Thinking about Recursion
3. Analysis of Recursion
   - Integer Powers
   - Towers of Hanoi



Recursion

- Recursion is a strategy whereby we reduce a problem to a smaller problem of the same type
- We repeat this until we reach a trivial case we can solve by some other means
- Recursion can also be used to describe situations in a succinct manner using references to itself. E.g.
  - ⋆ Definition of factorial: $n! = n \times (n-1)!$ with $0! = 1$
  - ⋆ Definition of ancestor: X is the ancestor of Y if X is the parent of Y or Z is the parent of Y and X is the ancestor of Z



Structure of Recursion

- Notice that these are *self-referential* definitions
- A recursive definition consists of two elements
  - ⋆ **The Base Case:** or boundary cases where the problem is trivial
  - ⋆ **The Recursive Clause:** which is a self-referential part driving the problem towards the base case
- This should be reminiscent of proofs by induction—indeed the two are very closely related (many mathematical functions are defined recursively and there properties are proved by induction)

Recap on Proof by Induction

- Let us prove

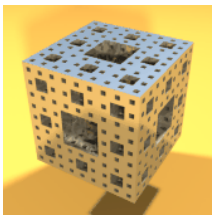
$$S_n = \sum_{i=0}^{n} 2^i = 1 + 2 + 4 + \cdots + 2^n = f(n) = 2^{n+1} - 1$$

- We use induction
  - ⋆ **Base Case:** $S_0 = \sum_{i=0}^{0} 2^i = 1$ and $f(0) = 2^{0+1} - 1 = 1$ so $S_0 = f(0)$ ✓
  - ⋆ **Recursive Case:** We assume $S_n = f(n) = 2^{n+1} - 1$, we want to prove that $S_{n+1} = f(n+1) = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$ now

$$S_{n+1} = \sum_{i=0}^{n+1} 2^i = \sum_{i=0}^{n} 2^i + 2^{n+1} = S_n + 2^{n+1} = f(n) + 2^{n+1}$$
$$= 2^{n+1} - 1 + 2^{n+1} = 2 \times 2^{n+1} - 1 = 2^{n+2} - 1 \quad ✓$$

Outline

1. Simple Recursion
2. Programming Recursively
   - Simple Examples
   - Thinking about Recursion
3. Analysis of Recursion
   - Integer Powers
   - Towers of Hanoi



Programming Recursively

- Most modern programming languages, including C++, allow you to program recursively
- That is they allow functions/methods to be defined in terms of themselves

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    else if (n==0)
        return 1;
    else
        return n*factorial(n-1);
}
```

- This will work, is very intuitive, but is certainly not the best way to code a factorial function

Integer Powers

- How do we compute 0.95^{25}?
- One way is to multiply together 0.95 twenty five times
- A more efficient way is to observe

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- We can repeat this until we reach $x^1 = x$

$$0.95^{25} = 0.95 \times (0.95)^{24} = 0.95 \times \left((0.95)^{12}\right)^2 = 0.95 \times \left(\left((0.95)^6\right)^2\right)^2$$
$$= 0.95 \times \left(\left(\left((0.95)^3\right)^2\right)^2\right)^2 = 0.95 \times \left(\left(\left((0.95 \times (0.95)^2\right)^2\right)^2\right)^2$$

six multiplications rather than 24!

Implementing Integer Power

- Integer power looks rather intimidating to code
- However, the recursive definition is easy
- We can easily code this function recursively

```
double power(double x, long n)   // (Overflow is possible)
{
    return  n <  0  ? 1 / power(x,-n)          // Negative power
        :  n == 0   ? 1                        // Special case
        :  n == 1   ? x                        // Base case
        :  n%2 == 0 ? (x = power(x, n/2)) * x  // Even power
        :             x * power(x, n-1);       // Odd power
}
```


Helper Functions

- This is a slick implementation from the web, but not terribly efficient
- We only need to do the first two checks once
- A more efficient implementation would use a helper function

```
double power(double x, long n) { // (Overflow is possible)
    return  n <  0  ? power_recurse(1.0/x,-n)  // Negative power
        :  n == 0   ? 1                        // Special case
        : power_recurse(x,n);

}

double power_recurse(double x, long n) {
   return  n == 1  ? x                                 // Base case
       : n%2 == 0 ? (x = power_recurse(x, n/2)) * x    // Even powe
       :            x * power_recurse(x, n-1);         // Odd power
}
```


Writing Recursive Programs

- You need to make sure that you catch the base case **before** you recurse
- The recursive case can call itself, possibly many times, provided the inductive argument is closer to the base case
- That is,
  - Ensure that you use a 'smaller problem'
  - Assume that you can solve the 'smaller problem'

The Cost of Recursion

- Recursion acts just like any other function call
- The values of all local variables in scope are put on a stack
- The function is called and
  - returns a value or
  - change some variable or object
- When the function returns, the values stored on the stack are popped and the local variables restored to their original state
- Although this operation is well optimised it is time consuming

Unrolling Recursion

- Recursion can frequently be replaced
- E.g. we can easily write a factorial function

```
long factorial(long n)
{
    if (n<0)
        throw new IllegalArgumentException();
    long res = 1;
    for (int i=2; i<=n; i++)
        res *= i;
    return res;
}
```

with no function calls this will run much faster than the recursive version

The Greatest Common Denominator

- One of the most famous algorithms is Euclid's algorithm for calculating the greatest common denominator
- The greatest common denominator of A and B is the largest integer, C, which exactly divides A and B
- E.g. the greatest common denominator of 70 and 25 is 5
- Euclid's algorithms uses the fact that
  - $\gcd(A, B) = \gcd(B, A \mod B)$
  - $\gcd(A, 0) = A$
- Thus $\gcd(70, 25) = \gcd(25, 20) = \gcd(20, 5) = \gcd(5, 0) = 5$

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

    if (b==0) {
       return a;
    }
    return gcd(b, a%b);
}
```


Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{

    if (b==0) {
       return a;
    }
    long c = a%b;
    a = b;
    b = c;
    return gcd(a, b);
}
```

Implementation of GCD

- The implementation of gcd is trivial using recursion

```
long gcd(long a, long b)
{
  while(true) {
     if (b==0) {
        return a;
     }
     long c = a%b;
     a = b;
     b = c;
  }
}
```

- Example of tail recursion

When Definitely not to Recurse

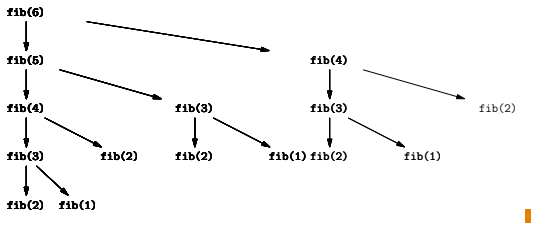
- A classic recursively defined sequence is the Fibonacci series
  - $f_n = f_{n-1} + f_{n-2}$
  - $f_1 = f_2 = 1$
- We might be tempted to write a recursive function to define the series

```
long fibonacci(long n)
{
  if (n<=2)
     return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Why shouldn't you want to do this?

Fibonacci

```
long fib(long n)
{
  if (n<=2)
    return 1;
  return fib(n-1) + fib(n-2);
}
```

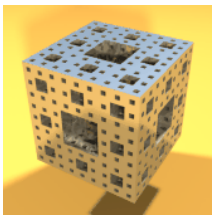


Why Use Recursion At All?

- Both `factorial` and `gcd` could be written without using recursion
- The programs would probably run faster
- The `gcd` program would be less clear
- The cost of the additional function calls is often insignificant
- It would considerably harder to write many programs such as `power` non-recursively
- Later we will see algorithms like `quick sort` which rely on recursion

Outline

1. Simple Recursion
2. Programming Recursively
   - Simple Examples
   - Thinking about Recursion
3. **Analysis of Recursion**
   - Integer Powers
   - Towers of Hanoi



Analysis

- We can use recursion to compute the time complexity of a recursive program!
- To do this we denote the time taken to solve a problem of size n by $T(n)$
- To compute the time complexity of factorial, we note that to compute $n!$ we have to multiply n by $(n-1)!$
- That is, the number of multiplications we need to compute is

$$T(n) = T(n-1) + 1$$

- Now $T(0) = 0$ so

$$T(n) = T(n-1) + 1 = T(n-2) + 2 = \cdots = T(0) + n = n$$

Time to Compute Power

- How long does it take to compute x^n?
- Remember

$$x^{2n} = (x^n)^2$$
$$x^{2n+1} = x \times x^{2n}$$

- Thus

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n \text{ is even} \\ T((n-1)/2) + 2 & \text{if } n \text{ is odd} \end{cases}$$
$$\leq T(\lfloor n/2 \rfloor) + 2$$

- Where $T(1) = 0$

How many times?

- We want to solve $T(n) \leq T(\lfloor n/2 \rfloor) + 2$ with $T(1) = 0$
- How many times do we divide n by two until we reach 1?
- Denoting n by a binary number $n = b_m b_{m-1} \cdots b_2 b_1$
  - $b_i \in \{0, 1\}$
  - $b_m = 1$
  - m is the number of digits in the binary representation of n
  - $\lfloor n/2 \rfloor = b_m b_{m-1} \cdots b_2$
  - After $m-1$ 'divides' we reach 1
- Thus $T(n) \leq 2(m-1)$

How Big is m

- How many binary digits do you need to represent an integer n
- Note that an m digit number can represent a number from 2^m to $2^{m+1} - 1$
- Thus

$$2^m \leq n < 2^{m+1}$$
$$m \leq \log_2(n) < m + 1$$

- But $T(n) \leq 2(m-1) \leq 2(\log_2(n) - 1) = \Theta(\log_2(n))$

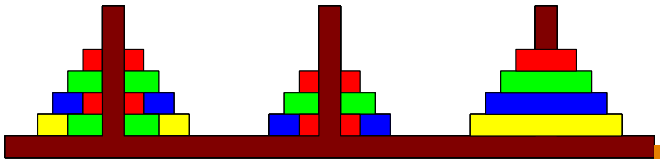
Towers of Hanoi

In an ancient city, so the legend goes, monks in a temple had to move a pile of 64 sacred disks from one location to another. The disks were fragile; only one could be carried at a time. A disk could not be placed on top of a smaller, less valuable disk. In addition, there was only one other location in the temple (besides the original and destination locations) sacred enough for a pile of disks to be placed there.

Using the intermediate location, the monks began to move disks back and forth from the original pile to the pile at the new location, always keeping the piles in order (largest on the bottom, smallest on the top). According to the legend, before the monks could make the final move to complete the new pile in the new location, the temple would turn to dust and the world would end.

A Smaller Tower of Hanoi

- Here is a smaller problem of just four disks

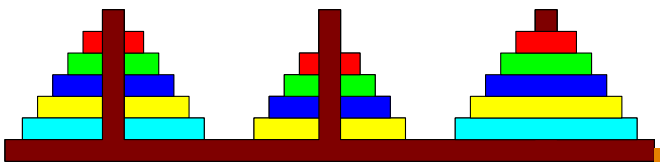


Algorithms in the Real World

- We require an algorithm to solve the towers of Hanoi
- Algorithms don't just apply to computers!
- If you try to solve the problem by hand you will discover that its quite fiddly
- There is a simple recursive solution which turns out to be optimal
- Let `move(X,Y)` denote the procedure of moving the top disk from peg `X` to peg `Y`
- Let `hanoi(n,X,Y,Z)` denote the procedure of moving the top `n` disks from peg `X` to peg `Z` using peg `Y`

Solving Towers of Hanoi

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```



Optimality of Solution

- This is optimal because
  - ⋆ You have to move the largest disk from peg A to peg C
  - ⋆ We do this only once
  - ⋆ To make this move all the other disks must be on peg B
  - ⋆ Assuming that we solve the $n-1$ disk problem optimally then we solve the n disk problem optimally
  - ⋆ We solve the one disk problem optimally (i.e. we move it to where it should go)
  - ⋆ This completes a proof by induction!

How Long Does It Take?

- How many moves does it take to transfer n disks?
- We can use the same procedure as before

```
hanoi(n, A, B, C)
{
  if (n>0) {
     hanoi(n-1, A, C, B);
     move(A, C);
     hanoi(n-1, B, A, C);
  }
}
```

  - ⋆ $T(n) = 2\,T(n-1) + 1$
  - ⋆ $T(0) = 0$

Lets Enumerate

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- $T(1) = 2 \times 0 + 1 = 1$
- $T(2) = 2 \times 1 + 1 = 2 + 1 = 3$
- $T(3) = 2 \times 3 + 1 = 6 + 1 = 7$
- $T(4) = 2 \times 7 + 1 = 14 + 1 = 15$
- Looks like $T(n) = 2^n - 1$

Proof by Induction

- $T(n) = 2\,T(n-1) + 1$
- $T(0) = 0$
- We want to prove $T(n) = 2^n - 1$
- Base case: $T(0) = 2^0 - 1 = 1 - 1 = 0$ ✓
- Recursive case: Assume $T(n-1) = 2^{n-1} - 1$ then

$$T(n) = 2\,T(n-1) + 1$$
$$T(n) = 2 \times (2^{n-1} - 1) + 1$$
$$T(n) = 2^n - 2 + 1 = 2^n - 1$$ ✓

Time Complexity of Recursion

- The time complexity for recursion can be tricky to calculate
- The procedure is to calculate the time, $T(n)$, taken for a problem of size n in terms of the time taken for a smaller problem
- The difficulty is to solve the recursion
- Recursive programs can be very quick (e.g. $O(\log n)$ for computing integer powers)
- Recursive programs can also be very slow, (e.g. $O(2^n)$ for towers of Hanoi)
- In case your interested, if it takes 1 second to move a disk it will take almost $585\,000\,000\,000$ years to move 64 disks

Lessons

- Recursion is a powerful tool for writing algorithms
- It often provides simple algorithms to otherwise complex problems
- Recursion comes at a cost (extra function calls)
- There are times when you should avoid recursion (computing Fibonacci numbers)
- You need to be able to analyse the time complexity of recursion
- Used appropriately, recursion is fantastic!

Lesson 7: *Make Friends with Trees*

Binary trees, binary search trees, sets, tree iterators

Outline

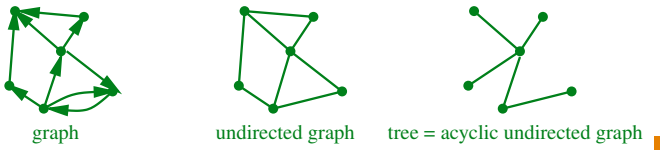
1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. Tree Iterators

Trees

- Trees are one of the major ways of structuring data
- They are used in a vast number of data structures
  - Binary search trees
  - B-trees
  - splay trees
  - heaps
  - tries
  - suffix trees
- We shall cover most of these

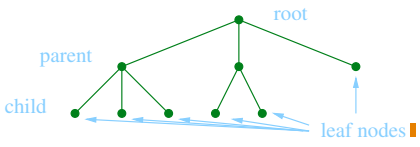
Defining Trees

- Mathematically a tree is an **acyclic undirected graph**
  - **graph**: a structure consisting of **nodes** or **vertices** joined by **edges**
  - **undirected**: the edges goes both ways
  - **acyclic**: there are no cycles in the graph



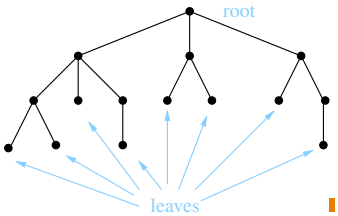
Borrowing from Nature

- We often impose an ordering on the nodes (or a direction on the edges)—known as a rooted tree
- Borrowing from nature, we recognise one node as the **root** node
- Nodes have **children** nodes living beneath them
- Each child has a **parent** node above them except the root
- Nodes with no children are **leaf** nodes



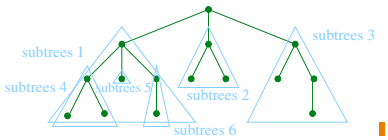
Spot the Error

- One small biological inconsistency
- Yep!, computer scientists draw there trees upside down
  - root at the top
  - leaves at the bottom



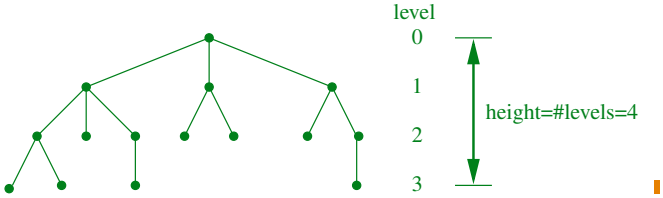
Subtrees

- We can think of the tree made up of **subtrees**



Level of Nodes

- It is useful to label different levels of the tree
- We take the **level** of a node in a tree as its distance from the root
- We take the **height** of a tree to be the number of levels



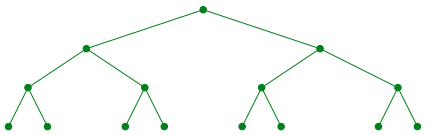
Outline

1. Trees
2. **Binary Trees**
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Trees

- A **binary tree** is a tree where each node can have zero, one or two children
- The total number of possible nodes at level l is 2^l
- The total number of possible nodes of a tree of height h is

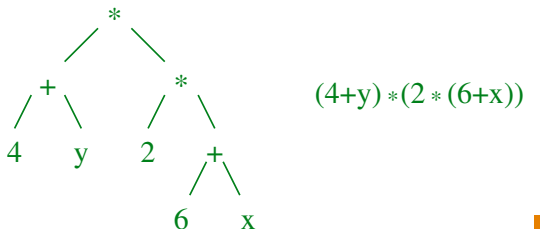
$$1 + 2 + \cdots + 2^{h-1} = 2^h - 1$$



Level	# Nodes
0	1
1	2
2	4
3	8
	15

Uses of Binary Trees

- Binary trees have a huge number of applications
- For example, they are used as **expression trees** to represent formulae



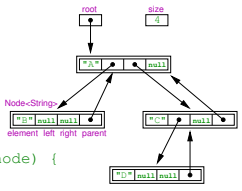
$(4+y)*(2*(6+x))$

Implementation

- We wish to build a generic binary tree class with each node housing an element
- Again we use a `Node<T>` class as the building block for our data structure—in this case a node of the tree
- The `Node<T>` class will contain a pointer to left and right children
- To help navigate the tree each node will contain a pointer to its parent

C++ Code

```
template <typename T>
class binary_tree {
private:
  class Node {
  public:
    T element;
    Node* parent;
    Node* left = 0;
    Node* right = 0;

    Node(const T& value, Node* parent_node) {
      element = value;
      parent = parent_node;
    }
  };

  unsigned no_elements = 0;
  Node* root = 0;

private:
  ...
```



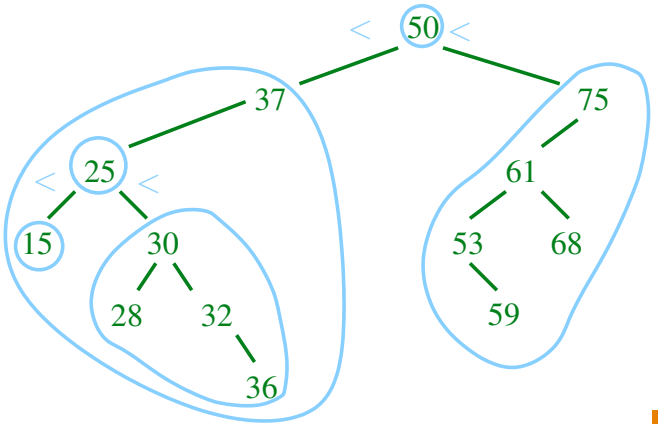
Outline

1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. **Binary Search Trees**
   - Definition
   - Implementing a Set
4. Tree Iterators

Binary Search Trees

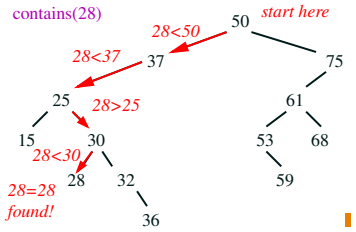
- We will concentrate on one of the most important binary trees, namely the **binary search tree**
- The binary search tree keeps the elements ordered
- We can define a binary search tree recursively
  1. Each element in the left subtree is less than the root element
  2. Each element in the right subtree is greater than the root element
  3. Both left and right subtrees are binary search trees

Example Binary Search Tree



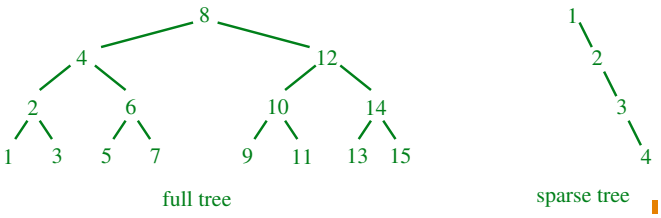
Searching A Binary Search Tree

- Searching a binary search tree is easy
- Start at the root
- Compare with element
  - If less than element go left
  - If greater than element go right
  - If equal to element found



Speed of Search

- The number of comparisons necessary to find an element in a binary tree depends on the level of the node in the tree
- The worst case number of comparisons is therefore the height of the tree
- This depends on the density of the tree



Implementing a Set

- A set is a fundamental **abstract data type**
- It is a collection of things with no repetition and no order
- Ironically because order doesn't matter we can order the elements

$$\{1,3,5,5,3,4\} = \{5,3,4,1\} = \{1,3,4,5\}$$

- This allows rapid search—a feature we care about
- Binary trees are one of the efficient ways of implementing a set

Fitting In

- The standard template library provides a class `std:set<T>`
- This contains many functions like
  - Constructors
  - `size()`
  - `insert(T o)`
  - `find(T o)`
  - `erase(T o)`
  - `begin()` and `end()`

Comparable

- To sort any objects they must be comparable
- In the STL the set implementation has a second template parameter: `std::set<T, Compare = less<T> >`
- by default this is defined to be `less<T>` (which is a function already defined for most common types) which you can define
- If you have a set of complex objects you will have to define `Compare`

```
bool MyCompare(MyObject left, MyObject right) {
  return something
}

mySet = set<MyObject, MyCompare>;
```


Find an Element

- One of the core operations of a binary tree is to find a node

```
iterator find(const T& element) {
 Node* current = root;
 while (current!=0) {
   if (current->element == element) {
     return iterator(current);
   }
   if (element < current->element) {
     current = current->left;
   } else {
     current = current->right;
   }
 }
 return iterator(0);
}
```

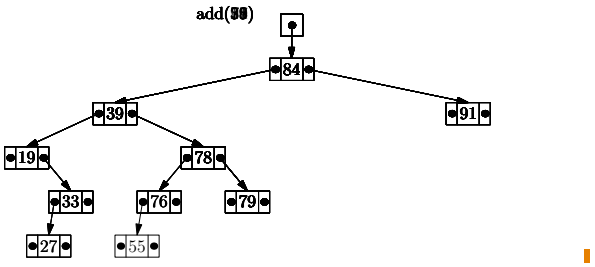

Add an Element

```
pair<iterator, bool> insert(const T& element) {
  if (no_elements==0) {
    root = new Node(element, 0);
    ++no_elements;
    return pair<iterator, bool>(iterator(root), true);
  }
  Node* parent = 0;
  Node* current = root;
  while(current != 0) {
    if (current->element == element) {
      return pair<iterator, bool>(iterator(0), false);
    }
    parent = current;
    if (element < current->element) {
      current = current->left;
    } else {
      current = current->right;
    }
  }
```



```
  current = new Node(element, parent);
  if (element < parent->element) {
    parent->left = current;
  } else {
    parent->right = current;
  }
  ++no_elements;
  return pair<iterator, bool>(iterator(current), true);
}
```

Tree in Action



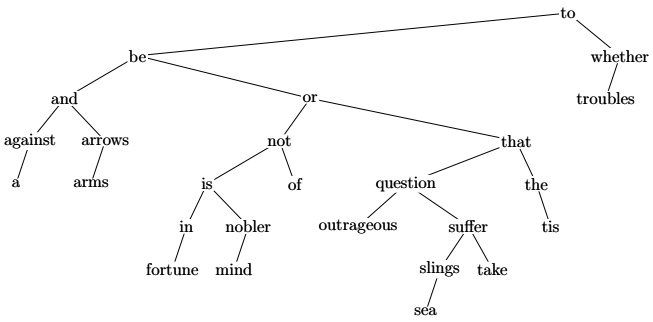
Shape of Tree

- The structure of the tree depends on the order in which we add elements to it
- Suppose we add

  To be, or not to be: that is the question:
  Whether 'tis nobler in the mind to suffer
  The slings and arrows of outrageous fortune,
  Or to take arms against a sea of troubles,

- Ignoring punctuation we get the following tree

Hamlet



Outline

1. Trees
2. Binary Trees
   - Implementing Binary Trees
3. Binary Search Trees
   - Definition
   - Implementing a Set
4. **Tree Iterators**

Tree Iterators

- As with most container classes it is very useful to define iterators
- `begin()` should return a "pointer" to the start of the tree
- `end()` provides a "pointer" past the end
- `operator*()` returns the element
- `opeator++()` increments the "pointer"
- `operator!=(lhs, rhs)` is used to compare iterators

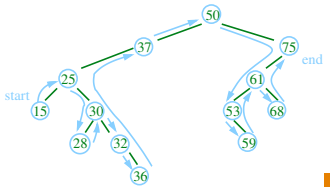
```
set<int> mySet;
...
for(auto pt=mySet.begin(), pt!=mySet.end(), ++pt) {
  cout << *pt;
}
```


C++ Code

```
class binary_tree {

public:
  class iterator {
  private:
    Node* current;

  public:
    iterator(Node* node) {current=node;}
    T operator*() const {return current->element;}
    iterator operator++() {
      current = successor(current);
      return *this;
    }
    bool operator!=(const iterator& other) {
      return current!=other.current;
    }
  };

  iterator begin() {...}
  iterator end() {return iterator(0)}

};
```


Successor

- To find the successor we first start in the left most branch
- We follow two rules
  1. **If** right child exist **then** move right once and then move as far left as possible
  2. **else** go *up* to the left as far as possible and then move up right



$\{15, 25, 28, 30, 32, 36, 37, 50, 53, 59, 61, 68, 75\}$

Lessons

- Trees and particularly binary trees are one of the most important tools of a computer scientist
- Conceptually they are quite simple
- However, there are a lot of details that need to be understood
- Coding even simple trees needs great care
- As we will see things get more complicated

Algorithms and Analysis

Lesson 8: *Keep Trees Balanced*

AVL trees, red-black trees, TreeSet, TreeMap

Outline

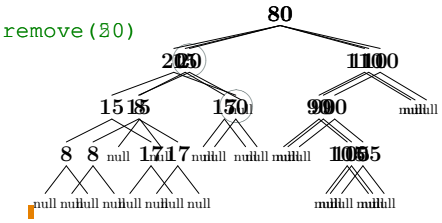
1. Deletion
2. Balancing Trees
   - Rotations
3. AVL
4. Red-Black Trees
   - TreeSet
   - TreeMap

Recap

- Binary search trees are commonly used to store data because we need to only look down one branch to find any element
- We saw how to implement many methods of the binary search tree
  - ⋆ `find`
  - ⋆ `insert`
  - ⋆ `successor` (in outline)
- One method we missed was `remove`

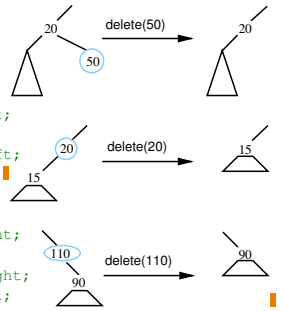
Deletion

- Suppose we want to delete some elements from a tree
- It is relatively easy if the element is a leaf node (e.g. 50)
- It is not so hard if the node has one child (e.g. 20)



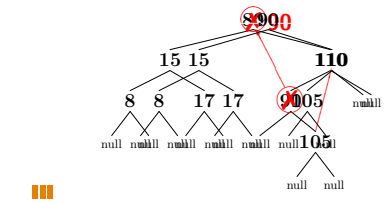
Code to remove Node n

```
if (n->left==0 && n->right==0) {
   if (n == n->parent->left)
      n->parent->left = 0;
   else
      n->parent->right = 0;
} else if (n->right==0) {
   if (n == n->parent->left)
      n->parent->left = n->left;
   else
      n->parent->right = n->left;
   n->left->parent = n->parent;
} else if (n->left==0) {
   if (n == n->parent->left)
      n->parent->left = n->right;
   else
      n->parent->right = n->right;
   n->right->parent = n->parent;
}
delete n;
```



Removing Element with Two Children

- If an element has two children then
  - ⋆ replace that element by its successor
  - ⋆ and then remove the successor using the above procedure

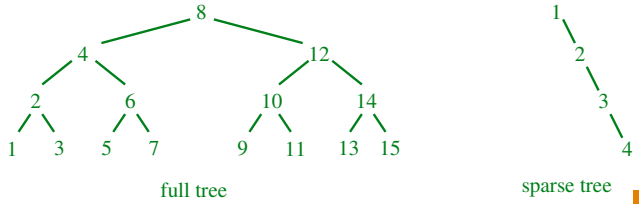


Outline

1. Deletion
2. Balancing Trees
   - Rotations
3. AVL
4. Red-Black Trees
   - TreeSet
   - TreeMap

Why Balance Trees

- The number of comparisons to access an element depends on the depth of the node
- The average depth of the node depends on the shape of the tree



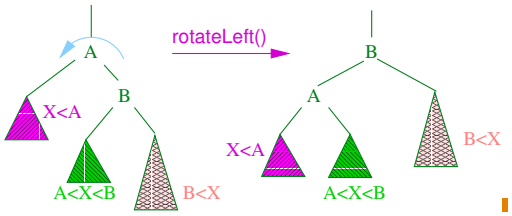
- The shape of the tree depends on the order the elements are added

Time Complexity

- In the best situation (a full tree) the number of elements in a tree is $n = \Theta(2^l)$ the depth is l so that the maximum depth is $\log_2(n)$
- It turns out for random sequences the average depth is $\Theta(\log(n))$
- In the worst case (when the tree is effectively a linked list), the average depth is $\Theta(n)$
- Unfortunately, the worst case happens when the elements are added *in order* (not a rare event)

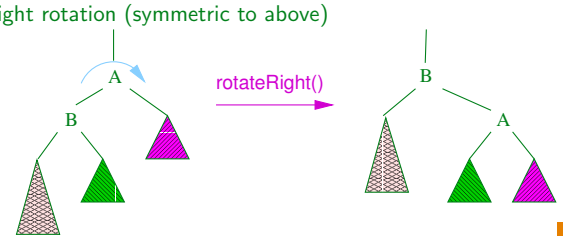
Rotations

- To avoid unbalanced trees we would like to modify the shape
- This is possible as the shape of the tree is not uniquely defined (e.g. we could make any node the root)
- We can change the shape of a tree using **rotations**
- E.g. left rotation



Types of Rotations

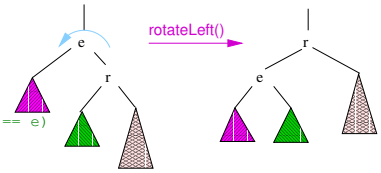
- We can get by with 4 types of rotations
  - ⋆ Left rotation (as above)
  - ⋆ Right rotation (symmetric to above)



  - ⋆ Left-right double rotation
  - ⋆ Right-left double rotation

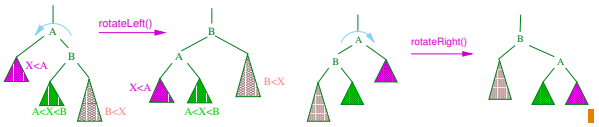
Coding Rotations

```
void rotateLeft(Node* e)
{
  Node* r = e->right;
  e->right = r->left;
  if (r->left != 0)
     r->left->parent = e;
  r->parent = e->parent;
  if (e->parent == 0)
     root = r;
  else if (e->parent->left == e)
     e->parent->left = r;
  else
     e->parent->right = r;
  r->left = e;
  e->parent = r;
}
```



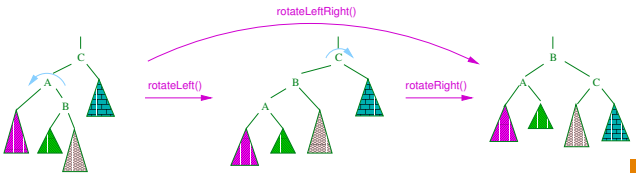
When Single Rotations Work

- Single rotations balance the tree when the unbalanced subtree is on the outside



Double Rotations

- If the unbalanced subtree is on the inside we need a double rotation




```
leftRotation(C.left);
rightRotation(C);
```


Outline

1. Deletion
2. Balancing Trees
   - Rotations
3. **AVL**
4. Red-Black Trees
   - TreeSet
   - TreeMap

Balancing Trees

- There are different strategies for using rotations for balancing trees
- The three most popular are
  - ⋆ AVL-trees
  - ⋆ Red-black trees
  - ⋆ Splay trees
- They differ in the criteria they use for doing rotations

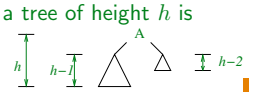
AVL Trees

- AVL-trees were invented in 1962 by two Russian mathematicians Adelson-Velski and Landis
- In AVL trees
  1. The heights of the left and right subtree differ by at most 1
  2. The left and right subtrees are AVL trees
- This guarantees that the worst case AVL tree has logarithmic depth

Minimum Number of Nodes

- Let $m(h)$ be the minimum number of nodes in a tree of height h
- This has to be made up of two subtrees: one of height $h-1$; and, in the worst case, one of height $h-2$
- Thus, the least number of nodes in a tree of height h is

$$m(h) = m(h-1) + m(h-2) + 1$$



- with $m(1) = 1$, $m(2) = 2$

Proof of Exponential Number of Nodes

- We have $m(h) = m(h-1) + m(h-2) + 1$ with $m(1) = 1$, $m(2) = 2$
- This gives us a sequence $1, 2, 4, 7, 12, \cdots$
- Compare this with Fibonacci $f(h) = f(h-1) + f(h-2)$, with $f(1) = f(2) = 1$
- This gives us a sequence $1, 1, 2, 3, 5, 8, 13, \cdots$
- It looks like $m(h) = f(h+2) - 1$
- Proof by substitution

Proof of Logarithmic Depth

- $m(h) = m(h-1) + m(h-2) + 1$ with $m(1) = 1$, $m(2) = 2$
- We can prove by inductions, $m(h) \geq (3/2)^{h-1}$
- $m(1) = 1 \geq (3/2)^0 = 1$, $m(2) = 2 \geq (3/2)^1 = 3/2$ ✓

$$m(h) \geq \left(\tfrac{3}{2}\right)^{h-3}\left(\tfrac{3}{2}+1+\left(\tfrac{3}{2}\right)^{3-h}\right) \geq \left(\tfrac{3}{2}\right)^{h-3}\tfrac{5}{2} = \left(\tfrac{3}{2}\right)^{h-3}\tfrac{10}{4} \geq \left(\tfrac{3}{2}\right)^{h-3}\tfrac{9}{4} = \left(\tfrac{3}{2}\right)^{h-1}$$ ✓

- Taking logs: $\log(m(h)) \geq (h-1)\log(3/2)$ or

$$h \leq \frac{\log(m(h))}{\log(3/2)} + 1 = O\left(\log(m(h))\right)$$

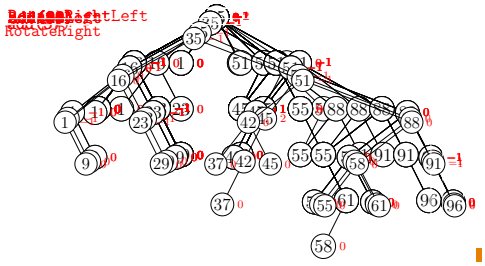
- The number of elements, n, we can store in an AVL tree is $n \geq m(h)$ thus

$$h \leq O(\log(n))$$

Implementing AVL Trees

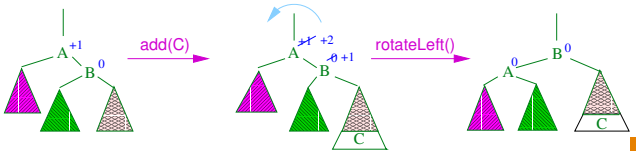
- In practice to implement an AVL tree we include additional information at each node indicating the balance of the subtrees

$$\texttt{balanceFactor} = \begin{cases} -1 & \text{right subtree deeper than left subtree} \\ 0 & \text{left and right subtrees equal} \\ +1 & \text{left subtree deeper than right subtree} \end{cases}$$



Balancing AVL Trees

- When `adding` an element to an AVL tree
  - ⋆ Find the location where it is to be inserted
  - ⋆ Iterate up through the parents re-adjusting the `balanceFactor`
  - ⋆ **If** the balance factor exceeds ± 1 `then` re-balance the tree and stop
  - ⋆ `else if` the balance factor goes to zero **then** stop



AVL Deletions

- AVL deletions are similar to AVL insertions
- One difference is that after performing a rotation the tree may still not satisfy the AVL criteria so higher levels need to be examined
- In the worst case $\Theta(\log(n))$ rotations may be necessary
- This may be relatively slow—but in many applications deletions are rare

AVL Tree Performance

- Insertion, deletion and search in AVL trees are, at worst, $\Theta(\log(n))$
- The height of an average AVL tree is $1.44\log_2(n)$
- The height of an average binary search tree is $2.1\log_2(n)$
- Despite being more compact insertion is slightly slower in AVL trees than binary search trees without balancing (for random input sequences)
- Search is, of course, quicker

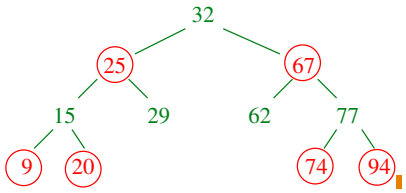
Outline

1. Deletion
2. Balancing Trees
   - Rotations
3. AVL
4. **Red-Black Trees**
   - TreeSet
   - TreeMap

Red-Black Trees

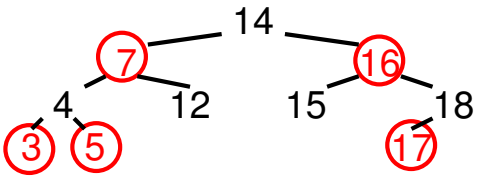
- Red-black trees are another strategy for balancing trees
- Nodes are either *red* or *black*
- Two rules are imposed

  Red Rule: the children of a red node must be black
  Black Rule: the number of black elements must be the same in all paths from the root to elements with no children or with one child



Restructuring

- When inserting a new element we first find its position
- If it is the root we colour it black otherwise we colour it red
- If its parent is red we must either relabel or restructure the tree



Performance of Red-Black Trees

- Red-black trees are slightly more complicated to code than AVL trees
- Red-black trees tend to be slightly less compact than AVL trees
- However, insertion and deletion run slightly quicker
- Both Java Collection classes and C++ STL use red-black trees

Set

- The standard template library (STL) has a class `std::set<T>`
- It also has a `std::underordered_set<T>` class (which uses a hash table covered later)
- As well as `std::multiset<T>` that implements a multiset (i.e. a set, but with repetitions)
- Using sets you can also implement **maps**

Maps

- One major abstract data type (ADT) we have not encountered is the map class
- The map class `std:map<Key,V>` contain key-value pairs `pair<Key,V>`
  - ⋆ The first element of type `Key` is the **key**
  - ⋆ The second element of type `V` is the **value**
- Maps work as content addressable arrays

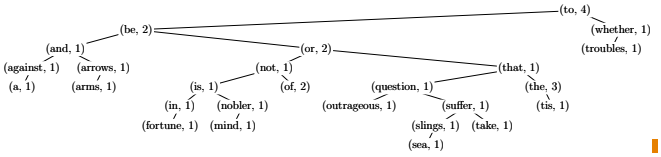
```
map<string, int> students;
student["John Smith"] = 89;
student["Terry Jones"] = 98;
cout << students["John Smith"];
```


Implementing a Map

- Maps can be implemented using a set by making each node hold a `pair<K,V>` objects

```
class pair<K,V>
{
  public:
  K first;
  V second;
}
```

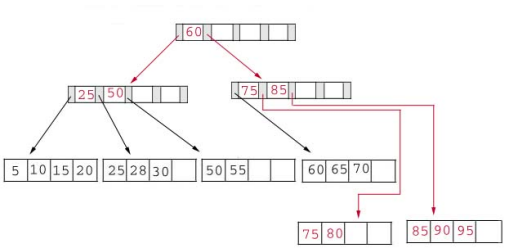
- We can count words using the key for words and value to count



Lessons

- Binary search trees are very efficient (order $\log(n)$ insertion, deletion and search) provided they are balanced
- Balanced trees are achieved by performing rotations
- There are different strategies for deciding when to rotate including
  - ⋆ AVL trees
  - ⋆ Red-black trees
- Binary trees are used for implementing **sets** and **maps**

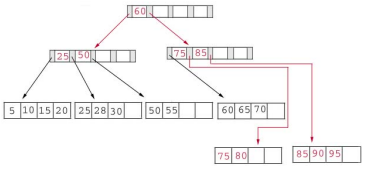
Lesson 9: *Sometimes It Pays Not to Be Binary*



B-Trees, Tries, Suffix Trees

Outline

1. **B-Trees**
2. Tries
3. Suffix Tree



B-Trees

- **B-trees** are balanced trees for fast search, finding successors and predecessors, insert, delete, maximum, minimum, etc.
- Not to be confused with binary trees
- They are designed to keep related data close to each other in (disk) memory to minimise retrieval time
- Important when working with large amount of data that is stored on secondary storage (e.g. disks)
- Used extensively in databases

When Big-O Doesn't Work

- An underlying assumption of Big-O is that all elementary operations take roughly the same amount of time
- This just isn't true of disk look-up
- The typical time of an elementary operation on a modern processor is 10^{-9} seconds
- But a typical hard disk might do 7 200 revolutions per minute or 120 revolutions per second
- The typical time it takes to locate a record is around 10ms or 10^7 times slower than an elementary operation

Accessing Data from Disk

- When accessing data from disk minimising the number of disk accesses is critical for good performance
- In database applications we want to store data as large sets
- Storing data in binary trees is disastrous as we typically need around $\log_2(n)$ disk accesses before we locate our data
- It is not unusual in databases for $n = 10\,000\,000$ so that $\log_2(n) \approx 24$
- Using binary trees it would often take several seconds to find a record

Multiway-Trees

- To remedy this we can use M-way trees so that the access time is

$$\log_M(n) = \frac{\log_2(n)}{\log_2(M)}$$

- In practice we might use $M \approx 200 \approx 2^8$ so we can reduce the depth of the tree by around a factor of 8
- The basic data structures for doing this is the B-tree
- There are many variants of B-tree, all trying to squeeze a bit more performances from the basic structure

B^+ Tree

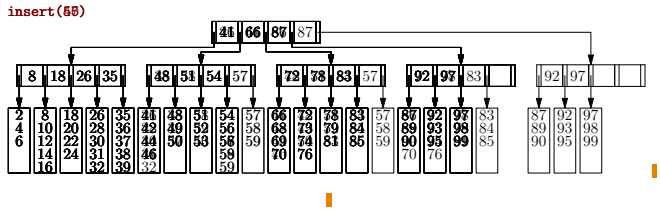
- A pretty basic implementation would obey the following rules
  1. The data items are stored at leaves
  2. The non-leaf nodes store up to M-1 keys to guide the search: key i represents the smallest key in subtree $i + 1$
  3. The root is either a leaf or has between 2 and M children
  4. All non-leaf nodes except the root have between $\lceil M/2 \rceil$ and M children
  5. All leaves are at the same depth and have between $\lceil L/2 \rceil$ and L data entries

Choosing M and L

- The choice of M and L depends on the block size (the information read in one go from disk)
- It also depends on the type of data that is being stored (integer, reals, strings, etc.)
- M and L might be in the hundreds or thousands
- In the examples below we consider tiny $M = L = 5$ which is unrealistic, but drawable

B-Tree Example

- $M = 5$, $L = 5$



Other Changes

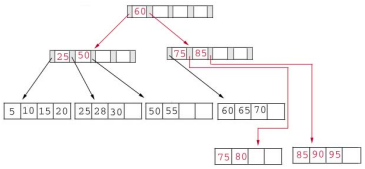
- If the root is full then it can be split into two and a new root created
- B-trees also have to allow the removal of records without losing its structure
- There are a number of variant strategies (e.g. neighbouring nodes can adopt a child if the current node cannot expand any more)
- The actual implementation of B-trees is tricky because there are many special cases

B-Tree Summary

- B-trees are an important data structure for databases where reducing the number of disk searches is vital
- They tend to be much more complex than the other data structures we have seen
- The problem of disk access can be improved by replacing disk memory with solid-state drives (still slow compared to memory)
- For massive databases new data structures have been developed to allow faster (although less flexible) information access (e.g. NOSQL, MongoDB, Neo4j)

Outline

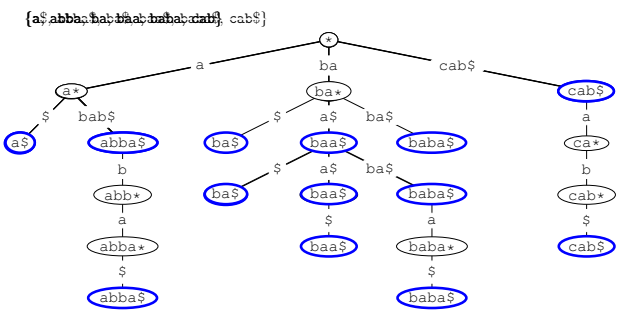
1. B-Trees
2. **Tries**
3. Suffix Tree



Tries

- A **Trie** (pron. 'try') or **digital tree** is a multiway tree often used for storing large sets of words
- They are trees with a possible branch for every letter of an alphabet
- Their names comes from the word *retrieval*
- Tries usually compactify the edges in the tree
- All words end with a special letter "$"

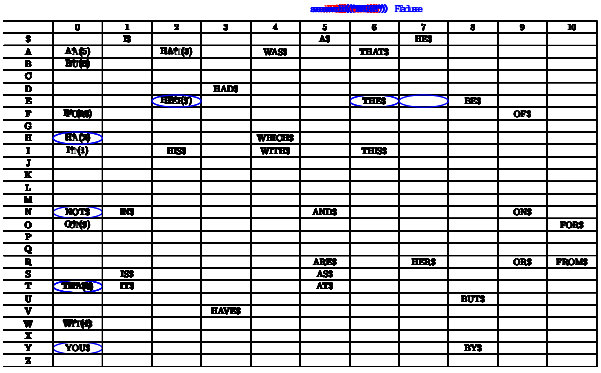
Trie



Uses of Tries

- Tries are yet another way of implementing sets
- They provide quick insertion, deletion and find
- Typically considerably quicker than binary trees and hash tables
- They are particularly good for spell checkers, completion algorithms, longest-prefix matching, hyphenation
- Each search finds the longest match between the words in the set and the query

Trie for 31 Most Common English Words



Disadvantage of Tries

- Table-based tries typically waste large amounts of memory
- Often table-based tries are used for the first few layers, while lower levels use a less memory intensive data structure
- These days memory is less of a problem so table-based tries are acceptable for some applications
- There are many implementations of tries each suited to a particular task

Binary Tries

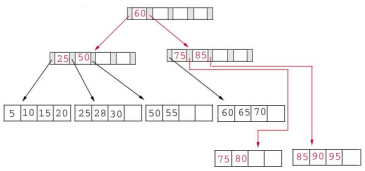
- One extreme (though not uncommon) solution to address memory issues is to build a bit-level trie so the data-structure is a binary tree
- It differs from a binary tree in that the decisions to go left or right depends on the current bit
- Although you lose the advantage of a multiway tree (of reducing the depth) it does find the longest match and it speeds up finds which fail

Why Tries?

- Tries are a classic example of a trade-off between memory and computational complexity
- Tries are slightly specialist and tend to get used in very particular applications
  - ⋆ Finding longest matches
  - ⋆ Completion, spell checking, etc.
- A basic trie is not too complicated, however, . . .
- There are many implementation which try to overcome the difficulty of wasting too much memory

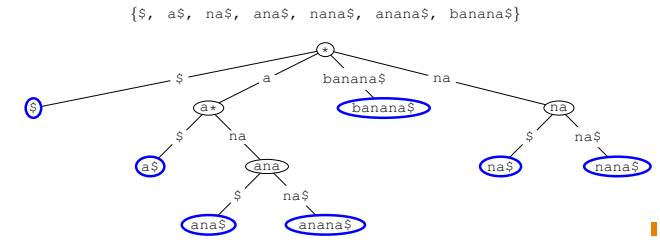
Outline

1. B-Trees
2. Tries
3. Suffix Tree



Suffix Tree

- Suffix tree is a trie of all suffixes of a string
- E.g. banana

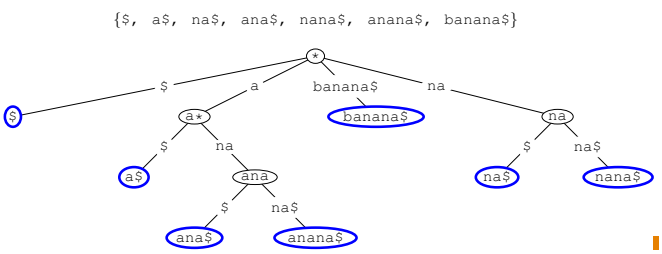


Importance of Suffix Tree

- The first linear-time algorithm for computing suffix trees was proposed by Peter Weiner in 1973, a more space efficient algorithm was proposed by Edward M. McCreight in 1976
- Esko Ukkonen in 1995 proposed a variant of McCreight's algorithm, but in a way that was much easier to understand
- It really only got implemented after this
- They are very important for string-based algorithms
- The classic application is in finding a match for a query string, Q, in a text, T

String Matching

- To find a match of a query string, Q, in a text, T, we can first construct the suffix tree of the string T we then simple look up the query, Q, using the trie



Complexity of Suffix Trees

- Using a regular trie for a suffix tree would typically use far too much memory to be useful
- However, by using pointers to the original text it is possible to build a suffix tree using $O(n)$ memory where n is the length of the text
- Furthermore (and rather incredibly) there is a linear time ($O(n)$) algorithm to construct the trie
- The algorithm is not however trivial to understand

Uses of Suffix Trees

- Suffix trees are efficient whenever it is likely that you will do multiple searches
- Exact word matching is in itself a very important application
- Suffix trees in combination with dynamic programming (which we will eventually get to) can be used to do inexact matching (finding the match with the smallest edit distance)
- Suffix trees get used in bioinformatics, advanced machine learning algorithms, . . .

Lessons

- Multiway trees can considerable speed up search over binary trees
- They are very important in some specialised applications (e.g. databases, spell-checking, completion, suffix trees)
- They are not as general purpose as binary search trees and are more complicated to implement
- But they can give the best performance—sometimes performance matters enough to make it worthwhile implementing multiway trees

Algorithms and Analysis

Lesson 13: *Make a hash of it*

Hash tables, separate chaining, open addressing, linear/quadratic probing, double hashing

Outline

1. **Why Hash?**
2. Separate Chaining
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. Hash Set and Map

Content Addressable Memory

- Suppose we have a list of objects which we want to look up according to its contents
- This is often referred to as **associative memory** structures
- A classical example would be a telephone directory
  - We look up a name
  - We want to know the number
- What data structure should we use?

Lists and Trees

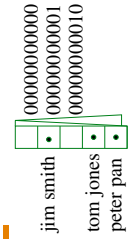
- To find an entry in a normal list takes $\Theta(n)$ operations
- If we had a sorted list we could use "binary search" to reduce this to $\Theta(\log(n))$
  - We will study binary search later
  - Maintaining an ordered list is costly ($\Theta(n)$ insertions)
- We could use a binary search tree
  - Search is $\Theta(\log(n))$
  - Insertion/deletion is $\Theta(\log(n))$

Thinking Outside the Box

- As with many data structures thinking about the problem differently can lead to much better solutions
- Let us consider the content we want to search on as a **key**
- For telephone numbers the key would be the name of the person we want to phone
- We could get $O(1)$ search, insertion and deletion if we used the key as an index into a big array
- That is the key is a string of, say, 100 characters so can be represented by an 800 digit binary number
- We could look up the key in a table of 2^{800} items

Hashing

- This approach is slightly wasteful of memory
- Almost all memory locations would be empty
- We can save on memory by folding up the table up onto itself



Hashing Codes

- A **hashing function** `hashCode(x)` takes an object, `x`, and returns a positive integer, the **hash code**
- To turn the hash code into an address take the modulus of the table size

```
int index = abs(hashCode(x) % tableSize);
```

- If $\texttt{tableSize} = 2^n$ we can compute this more efficiently using a mask

```
int index = abs(hashCode(x) & (tableSize -1));
```


Hashing Functions

- Hashing functions take an object and return an integer
- Hashing functions aren't magic
  - They tend to add up integers representing the parts of the object
- We want the integers to be close to random so that similar objects are mapped to different integers
- Sometimes two objects will be mapped to the same address—this is known as a **collision**
- Collision resolution is an important part of hashing

Hashing Strings

- A strings might be hashed using a function

```
unsigned long long hash(string const& s) {
  unsigned long long results = 12345;

  for (auto ch = s.begin(); ch != s.end(); ++ch) {
    results = 127*results + static_cast<unsigned char>(*ch);
  }
  return results;
}
```

- The numbers 12345 and 127 is to try to prevent clashes—there are lots of alternatives
- What we want is that strings that might be similar receive very different hash codes

DIY

- The `unordered_set<T, Hash<T> >` allows you to define your own hash function
- By default this is set to `std::hash<T>(T)`
- Not all classes have hash function defined so you will need to do this
- Care is needed to make you hash function produce near random hash codes

Outline

1. Why Hash?
2. **Separate Chaining**
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. Hash Set and Map

Collision Resolution

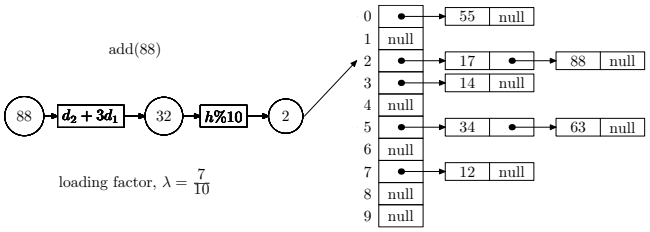
- Collisions are inevitable and must be dealt with
- There are two commonly used strategies
  - ⋆ Separate chaining—make a hash table of lists
  - ⋆ Open addressing—find a new position in the hash table
- Collisions add computational cost
- They occur when the hash table becomes full
- If the hash table becomes too full then it may need to be resized

Resizing a Hash Table

- Resizing a hash table is easy
  - ⋆ Create a new hash table of, say, twice the size
  - ⋆ Iterate through the old hash table adding each element to the new hash table
- Note that you have to recompute all the hash codes
- Resizing a hash table has a modest amortised cost, but can give you a very hiccupy performance
- The size of a hash table is a classic example of a memory-space versus execution time trade off—using bigger (sparser) hash tables speeds up performance

Separate Chaining

- In separate chaining we build a singly-linked list at each table entry

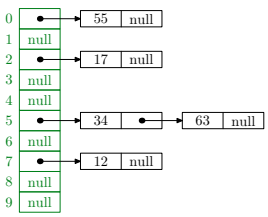


Search

- To find an entry in a hash table we again use the hash function on a key to find the table entry and then we search the list
- The time complexity depends on where objects are hashed
- If the objects are evenly dispersed in the table, search (and insertion) is $\Omega(1)$
- If the objects are hashed to the same entry in the hash table then search is $O(n)$
- Provided you have a good hashing function and the hash table isn't too full you can expect $\Theta(1)$ average case performance

Iterating Over a Hash Table

- To iterate over a hash table we
  - ⋆ Iterate through the array
  - ⋆ At each element we iterate through the linked list
- The order of the elements appears random
- This becomes more efficient as the table becomes fuller



55, 17, 34, 63, 12

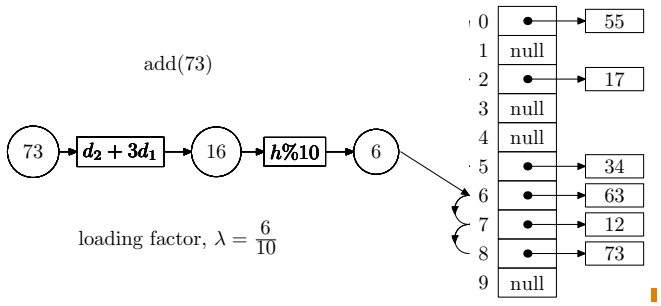
Outline

1. Why Hash?
2. Separate Chaining
3. **Open Addressing**
   - Quadratic Probing
   - Double Hashing
4. Hash Set and Map

Open Addressing

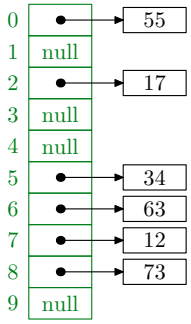
- In open addressing we have a single table of objects (without a linked-list)
- In the case of a collision a new location in the table is found
- The simplest mechanism is known as **linear probing** where we move the entry to the next available location

Linear Probing

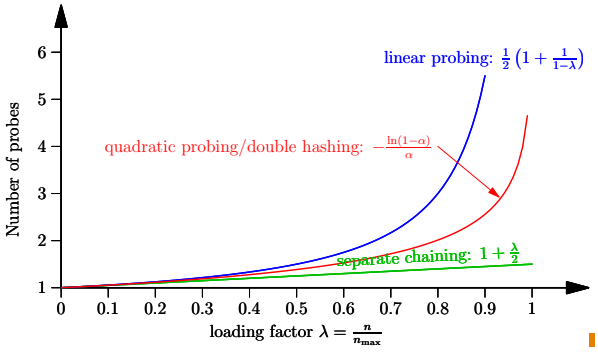


Linear Probing Pile Up

- The entries will tend to pile up or cluster—this is sometimes referred to as **primary clustering**
- Clusters become worse as the number of entries grow
- Clusters will increase the number of probes needed to find an insert location
- The proportion of full entries in the table is known as the **loading factor**



Reducing Number of Probes



- To avoid clustering we can use **quadratic probing** or **double hashing**

Quadratic Probing

- In quadratic probing we try the locations $h(x) + d_i$ where $h(x)$ is the original hash code and $d_i = i^2$
- That is we takes steps 1, 4, 9, 16, . . .
- Quadratic probing prevents primary clustering so dramatically decreases the number of probes needed to find a free location when the table is reasonably full
- One problem is that if we are unlucky we might not be able to add an element to the hash table even if the table isn't full
- However, if the size of the table is prime then quadratic probing will always find a free position provided it is not more than half full

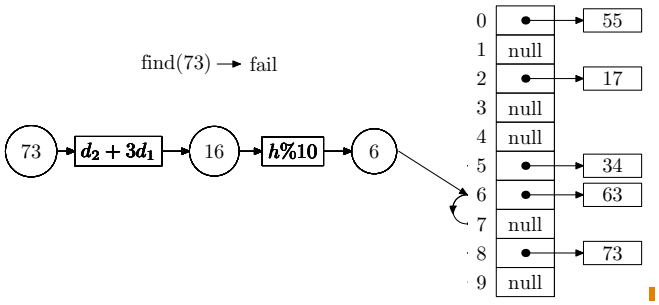
Double Hashing

- An alternative strategy is to known as double hashing where the locations tried are $h(x) + d_i$ where $d_i = i \times h_2(x)$
- $h_2(x)$ is a second hash function that depends on the key
- A good choice is $h_2(x) = R - (x \mod R)$ where R is a prime smaller than the table size
- It is important that $h_2(x)$ is not a divisor of the table size
  - ⋆ Either make sure the table size is prime or
  - ⋆ Set the step size to 1 if $h_2(x)$ is a divisor of the table size

Problems with Remove

- For all open addressing hash systems removing an entry is a problem
- Remember our strategy to find an input x is
  1. Compute the array index based on the hash code of x
  2. If the array location is empty then the search fails
  3. If the array location contains the key the search succeeds
  4. otherwise find a new location using an open addressing strategy and go to 2
- If we remove an entry then find might reach an empty location which was previously full
- This can prevent us finding a true entry

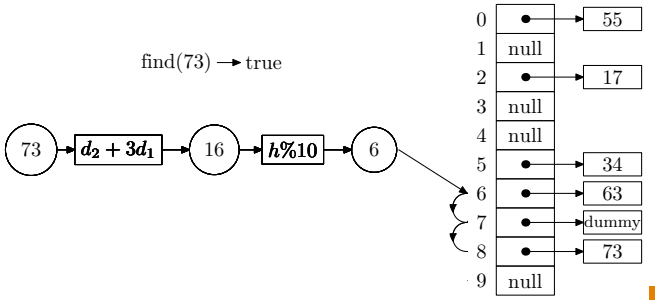
Linear Probing Example



Lazy Remove

- One easy fix is to mark the deleted table with a special entry
- A find method would consider this entry as full
- An iterator would ignore this entry
- An insert operator could insert a new entry in these special locations

Lazy Remove in Action



Outline

1. Why Hash?
2. Separate Chaining
3. Open Addressing
   - Quadratic Probing
   - Double Hashing
4. **Hash Set and Map**

What Strategy to Use?

- Most libraries including the STL (and the Java Collection class) use separate chaining
- This has the advantage that its performance does not degrade badly as the number of entries increase
- This reduces the need to resize the hash table
- The C++ standard did not include a hash table until C++11 ☹—although very good hash tables existed in C++

Hash Sets and Maps

- C++ also provides an `unordered_map<Key, V>` class
- It's performance is asymptotically superior to `map`, $O(1)$ rather than $O(\log(n))$
- Hash functions can take time to compute (it is often $O(\log(n))$) so `unordered_sets` might not be faster than `sets`
- One major difference is that the iterator for `sets` return the elements in order, `undordered_set`'s iterator doesn't

Applications

- Hash tables are used everywhere
- E.g. most databases use hash tables to speed up search
- In many document applications hash tables will be being generated in the background
- Content addressability is ubiquitous to many application where hash tables are used as standard

Lessons

- Hash tables are one of the most useful tools you have available
- They aren't particularly difficult to understand, but you need to know about
  - ⋆ hashing functions
  - ⋆ collision strategies
  - ⋆ performance (i.e. when they work)

Algorithms and Analysis

Lesson 11: *Use Heaps!*

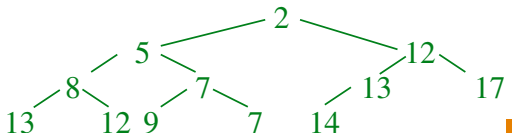
Heaps, Priority queues, Heap Sort, Other heaps

Outline

1. **Heaps**
2. Priority Queues
   - Array Implementation
3. Heap Sort
4. Other Heaps

Heaps

- A (min-)heap is (from one perspective) a binary tree
- It is a binary tree satisfying two constraints
  - It is a **complete** tree
  - Each child has a value 'greater than or equal to' its parent



- **complete** means that every level is fully occupied above the lowest level and the nodes on the lowest level are all to the left

Outline

1. Heaps
2. **Priority Queues**
   - Array Implementation
3. Heap Sort
4. Other Heaps

Priority Queues

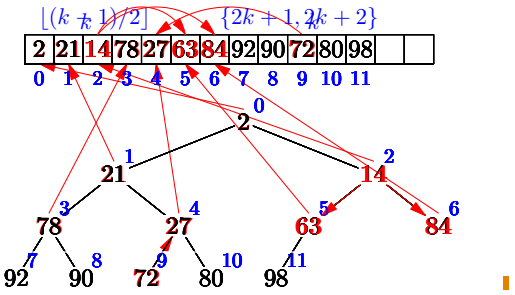
- One of the prime uses of heaps is to implement a priority queue
- A Priority Queue is a queue with priorities
- That is, we assign a priority to each element we add
- The head of the queue is the element with highest priority (smallest number)
- Used, for example, in simulating real time events
- Used to implement "greedy algorithms"

Priority Queue

- A simple Priority Queue might include
  - `unsigned size()` returning the the number of elements
  - `bool empty()` returns true if empty
  - `void push(T element, int priority)` adds an element
  - `T top()` returns head of queue
  - `void pop()` dequeues head of queue

Array Implementation of Heaps

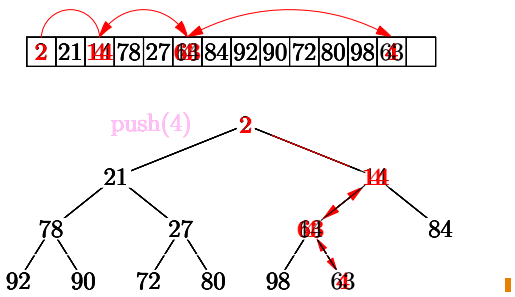
- Because the tree is complete we can implemented the heap efficiently using an array



Code for a Priority Queue

```
#include <vector>
using namespace std;

template <typename T, typename P>
class heapPQ {
private:
  vector<pair<T, P> > array;

public:

  heapPQ(unsigned capacity=11) {
    array.reserve(capacity);
  }

  unsigned size() {return array.size();}

  bool empty() {return array.empty();}

  const T& top() {return array[0].first;}
```

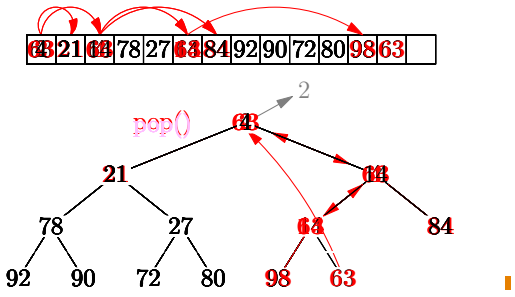
Adding an Element



Adding an Element

```
void push(T value, P priority) {
  pair<T,P> tmp(value, priority);
  array.push_back(tmp);
  unsigned child = size() - 1;

  /* Percolate Up */
  while(child!=0) {
    unsigned parent = (child-1)>>1;  // floor((child-1)/2)
    if (array[parent].second < array[child].second)
      return;
    array[child] = array[parent];
    array[parent] = tmp;
    child = parent;
  }
}
```

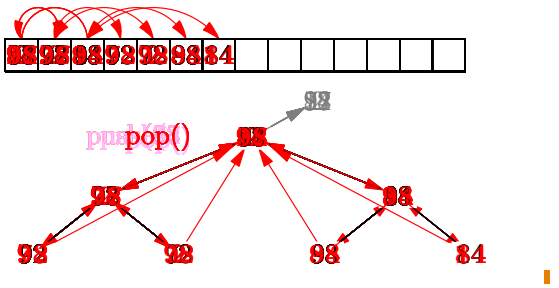

Popping the Top



Popping the Top

```
  void pop() {
  unsigned parent = 0;
  pair<T, P> tmp = array.back();
  array[0] = tmp;
  array.pop_back();
  unsigned child = 1;

  /* Percolate down */
  while(child<size()) {
    if (child+1<=size() && array[child+1].second < array[child].second)
      ++child;
    if (array[child].second > array[parent].second)
      return;
    array[parent] = array[child];
    array[child] = tmp;
    parent = child;
    child = 2*parent + 1;
  }
}
```


Heaps in Action

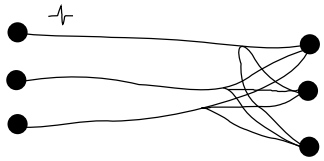


Time Complexity of Heaps

- The two important operation are `add` and `removeMin`
- These both either percolating an element up the tree or percolating an element down the tree
- The number of operations depends on the depth of the tree which is $\Theta(\log(n))$
- Thus `add` and `removeMin` are $O(\log(n))$
- Except `add` could also require resizing the array, but the amortised cost of this is low

Real Time Simulation

- A nice application of priority queues is to perform real time simulations
- I was once modelling a neural network where neuron fired an impulse which would then be received by other neurons



Synchronised Firing

- We wanted to show that if a group of neurons fired together they could make another group of neurons fire in synchrony despite the fact that it would take different times for the receiving neurons to feel the pulse (due to the different lengths of the axons)
- A famous Israeli group had "proved" this couldn't happen
- Using a priority queue we modelled the neurons
  - ⋆ When a neuron fired the receiving neurons would be put on a priority queue according to when they received the pulse
  - ⋆ If the receiving neurons received enough pulse in a short enough time they would then fire

Efficient Modelling

- Using a priority queue meant we knew when the next event would happen
- We did not have to run a clock where most of the time nothing happened
- This allowed us to perform a very large simulation efficiently
- The simulation showed that the pulse of neurons synchronised despite the "proof" that this wouldn't happen

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. **Heap Sort**
4. Other Heaps

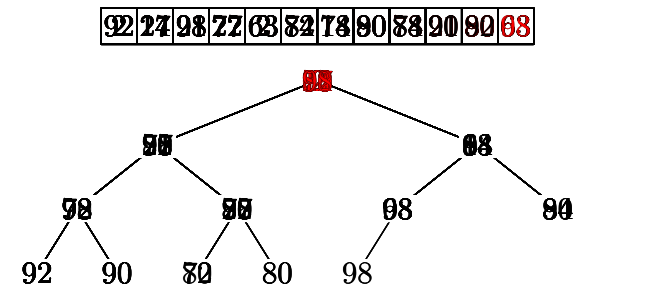
Heap Sort

- A priority queue suggests a very simple way of performing sort
- We simply add elements to a heap and then take them off again

```
template <typename T>
void sort(vector<T> aList)
{
    HeapPQ<T> aHeap = new HeapPQ<T>(aList.size());
    for (T element: aList)
      aHeap.push(element, element);

    aList.clear();
    while(aHeap.size() > 0) {
      aList.push_back(aHeap.top());
      aHeap.pop();
    }
}
```

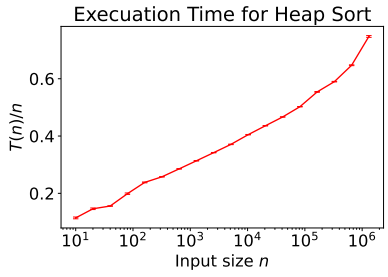
- Note that this is not an in-place sort algorithm (i.e. it uses lots of memory)

Example of Heap Sort



Complexity of Heap Sort

- As we have to add n elements and then remove n elements the time complexity is log-linear, i.e. $O(n \log(n))$



- This is actually a very efficient algorithm

Outline

1. Heaps
2. Priority Queues
   - Array Implementation
3. Heap Sort
4. **Other Heaps**

Other Heaps

- Binary Heaps are so useful that other types of heaps have been developed
- The simplest enhancement is to combine a binary heap with a map which maintains a pointer to each element
- The map has to be updated every time elements are moved in the heap (fortunately only $O(\log(n))$ elements are move each time the heap is updated)
- The advantage of this heap is that the priorities of elements can be changed (involving percolating elements up or down the tree)

Merging Heaps

- One common demand on a heaps is to merge two heaps
- Unfortunately binary heaps are not efficiently merged
- There are a variety of different heaps (leftist heaps, skew heaps, binomial queues, . . .) designed to be merged
- All these heaps are real binary trees (i.e. represented by pointers rather than put in an array)
- They are slower than a binary heap because indexing is slightly slower as is creating node objects
- However, they allow merging

Other Operations

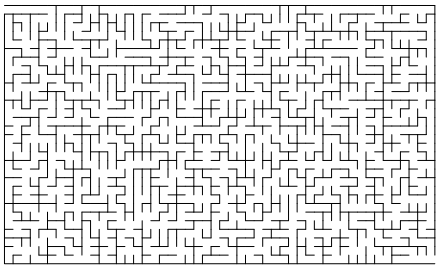
- All other operations are achieved by merging
  - Adding an element is achieved by merging the current heap with a heap of one element
  - Removing the minimum element is achieved by removing the root and merging the left and right tree
- For details see the course text (you won't be examined on the details of these heaps)

Lessons

- Heaps are a powerful data structure—they are particularly useful for implementing priority queues
- Heaps are binary trees that can be implemented as arrays
- Priority queue have many (often surprising uses)
  - They are used when you need a queue with priorities, e.g. in operating systems
  - They can be used to perform pretty efficient sort
  - They are often used for implementing greedy type algorithms
  - One important application is in real time simulations
- There exists many extensions of heaps

Algorithms and Analysis

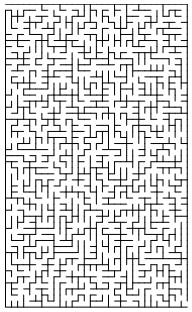
Lesson 12: *Use Arrays for Fast Set Algorithms*



Equivalent classes, Disjoint Set, Fast Sets

Outline

1. **Equivalent Classes**
2. Disjoint Sets
3. Fast Sets

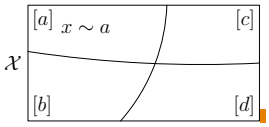


Equivalence Relations

- Given a set of elements $\mathcal{X} = \{x_1, x_2, \ldots\}$ and a binary relationship $\sim$ with the following properties

  (Reflexivity) For every element $x \in \mathcal{X}$, $x \sim x$
  (Symmetry) For every two elements $x, y \in \mathcal{X}$ if $x \sim y$ then $y \sim x$
  (Transitivity) For every three elements $x, y, z \in \mathcal{X}$ if $x \sim y$ and $y \sim z$ then $x \sim z$

- Then $\sim$ defines a partitioning of the set into **equivalence classes**



Example of Equivalence Classes

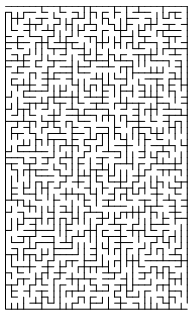
- Although, equivalent classes sound very mathematical they often provide a useful formalisation of the real world
- E.g. Pairs of web pages with a link in each direction between them
- Consider web pages in the same equivalence class if you can get from one to the other by clicking links
- Partitions the web into linked domains
- Friendship relations in social media

Dynamic Equivalence Classes

- Finding equivalence classes is rather easy using graph traversal algorithms
- However, as our web example suggests, there are applications where equivalence classes change over time
- Adding a link could join two domains which were separate
- We will see this is a useful idea both for building mazes and (in a later lecture) for finding minimum spanning trees
- Building a data structure which finds equivalence classes where the equivalence relation changes over time is challenging but fortunately there is an elegant solution to this

Outline

1. Equivalent Classes
2. **Disjoint Sets**
3. Fast Sets



Union-Find

- In the union-find algorithm we have a set of objects $x \in \mathcal{S}$ which are to be grouped into subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots$
- Initially each object is in its individual subset (no relationships)
- We want to make the **union** of two subsets (add relationship between elements)
- We also want to **find** the subset given an element
- This is a common problem for which we will write a class `DisjointSets` to perform fast `unions` and `finds`

DisjointSets

- We want to create a class

```
class DisjointSets
{
    DisjointSets(int numElements) {/* Constructor */}
    int find(int x) {/* Find root */}
    void union_(int root1, int root2) {/* Union */}

  private:
    int* s;
}
```

- Where `find(x)` returns a unique identifier for the subset which element `x` belongs to
- The array `s` contains labelling information to implement `find(x)`

The Union-Find Dilemma

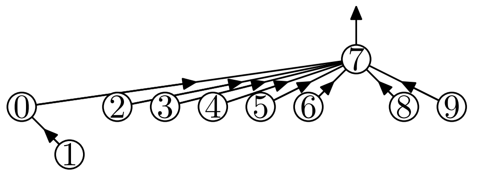
- A natural algorithm to perform finds is to maintain an array returning a subset label for each element—this makes `find` fast
- However, every time we combine two subset we have to change all the labels in this array (taking $O(n)$ operations)
- If we are unlucky the cost of performing n unions is $\Theta(n^2)$
- If we ensure that we relabel the smaller subset then the time complexity is $\Theta(n \log(n))$
- Fast *finds* seems to give slow(ish) *unions*
- What about the other way around?

Fast Union

- To achieve fast unions we can represent our disjoint sets as a forest (many disjoint trees)
- Every time we perform a union we make one of the trees point to the head of the other tree
- The cost of `find` depends on the depth of the tree
- To make unions efficient we make the shallow tree a subtree of the deeper tree

Putting it Together

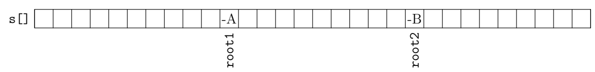
`find(6)=7`



7	0	7	7	7	7	7	-3	7	7
0	1	2	3	4	5	6	7	8	9

Smart Union

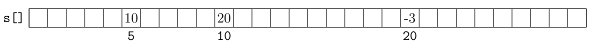
```
DisjointSets::DisjointSets(int numElements)
{
    s = new int[numElements];
    for(int i=0; i<numElements; i++)
        s[i] = -1;                  // roots are negative number
}

void DisjointSets::union_(int root1, int root2)
{
    if (s[root2]<s[root1]) {        // root2 is deeper
        s[root1] = root2;           // make root2 the root
    } else {
        if (s[root1]==s[root2])
            s[root1]--;             // update height if same
        s[root2] = root1;           // make root1 new root
    }
}
```



Path Compression

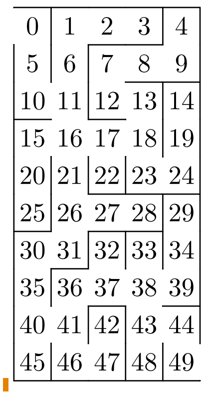
- To speed up `find` we relabel all nodes we visit during `find` by the root label

```
int DisjointSets::find(int index)
{
    if (s[index]<0)
        return index;
    else
        return s[index] = find(s[index]);
}
```



Mazes

- Union-Find is a data structure which can occur in very different applications
- One application is building a maze
- Start from a complete lattice
- Remove a randomly chosen edge if it connects two unconnected regions
- Stop when the start and end cell are connected
- Or better after all cells are connected



Time Complexity of Union-Find

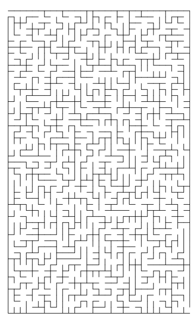
- If we perform M finds and N unions then the time complexity is $O(M \log_2^*(N))$
- Where $\log_2^*(N)$ is the number of times you need to apply the logarithm function before you get a number less than 1
- In practice $\log_2^*(N) \leq 5$ for all conceivable N

$\log_2($[illegible]

- The proof of this time complexity is rather involved

Outline

1. Equivalent Classes
2. Disjoint Sets
3. **Fast Sets**



Comparison of Sets

- Binary Search Trees: $O(\log_2(n))$, general purpose
- Hash tables: $O(1)$, but need to compute hash, slow iterator when sparse, general purpose
- B-trees: $O((k-1)\ \log_k(n))$ very complicated, used for large amounts of data
- Tries: $O(\log_k(n))$ for large k expensive in memory, complicated to code efficiently

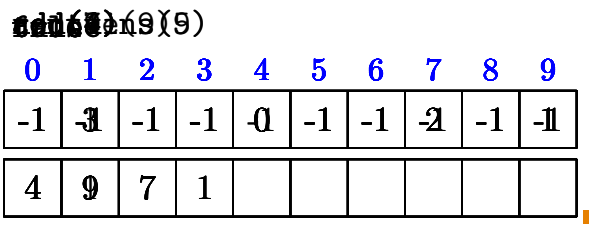
What Set to Use?

- A PhD student and I were working on writing a fast solver for a combinatorial optimisation problem
- We had to choose one variable to change out of a small number of possible variables
- Each time we changed a variable then we had to update the list of possible variables (remove some variables add others)
- We wanted a data structure which had quick add and remove and where we could choose a variable at random—what should we use?

Bounded Set

- One special feature is that we knew we only wanted the set to contain integers between 0 and n (where n might be 100 000)
- This allowed us to use an array to represent whether an integer belong to that set
- But how do we find a random element of the set quickly?
- Use another array of course!

FastSet



Implementation

```
class FastSet {
  private:
    int* indexArray;
    int* memberArray;
    int noMembers;

  public:
    FastSet(int n) {
      indexArray = new int[n];
      memberArray = new int[n];
      for(int i=0; i<n; i++) {
           indexArray [i] = -1;
      }
      noMembers = 0;
  }

  int size() {
     return noMembers;
  }
```


Add and Remove

```
bool add(int i) {
   if (indexArray[i]>-1)
      return false;
   memberArray[noMembers] = i;
   indexArray[i] = noMembers;
   ++noMembers;
   return true;
}

bool remove(int i) {
   if (indexArray[i]==-1)
         return false;
   --noMembers;
   memberArray[indexArray[i]] = memberArray[noMembers];
   indexArray[memberArray[noMembers]] = indexArray[i];
   indexArray[i] = -1;
   return true;
}
```


Collection Methods

```
void clear() {
   for(int i=0; i<noMembers; i++) {
      indexArray[memberArray[i]] = -1;
   }
   noMembers = 0;
}

bool isEmpty() {
   return noMembers==0;
}

int* begin() {return &memberArray[0];}
int* end() {return &memberArray[noMembers];}

}
```


And Random?

- We can add additional methods taking advantage of the classes strength

```
private:
  random_device rd;  // Seed for the random number engine
  mt19937 gen(rd()); // Mersenne Twister RNG

public:
  int getRandomElement() {
    return memberArray[uniform_int_distribution<int>(0, noMembers)];
  }
```

- Need to use FastSet signature to use this

```
FastSet fastSet(n);
:
int r = fastSet.getRandomElement();
```

Speed Up

- We compared our algorithm to a very highly regarded "state-of-the-art" algorithm
- For large problems we were over 10 times faster because of this data structure
- The competitor algorithm used a complex tree structure instead of the simple array
- Why? The array solution isn't in the books

Lessons

- If you have a bounded set then using an array is usually going to be very fast $O(1)$ (or $O(\log^*(n))$)
- These data structures are not general purpose for solving every day problems (c.f. `vector<T>`, `set<T>` and `map<T>`)
- They are "back pocket" data structures that solve problems that come up often enough that they are worth knowing about
- Sometimes good algorithms are not documented, but it doesn't mean they don't exist

Lesson 16: *Analyse!*

Pseudo code, binary search, insertion sort, selection sort, lower bound complexity

Outline

1. **Algorithm Analysis**
2. Search
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. Lower Bound

Algorithm Analysis

- We've covered most of the basic data structures
- The rest of the course is going to focus more on algorithms
- We will look predominantly at
  - Searching
  - Sorting
  - Graph Algorithms
- Emphasise general solution strategies

Code and Pseudo Code

- C++ code is often difficult to read—there are often programming details we don't care abour
- It contains details such as throwing exception which are repetitive and often depends on who you are writing the code for
- Algorithms are not language dependent (data structures are a bit more language dependent)
- To focus on what is important we will use a stylised programming language called **pseudo code**

Pseudo Code

- There is no standard for pseudo code
- The commands are not too dissimilar to C++
- The one strange convention is that assignments use an arrow ←
- Arrays are written in bold $\boldsymbol{a}$ with elements a_i
- In pseudo-code you are free to invent any operations that can be easily interpreted

Outline

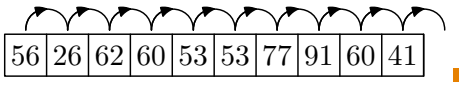
1. Algorithm Analysis
2. **Search**
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. Lower Bound

Dumb Search

```
DumbSearch(a, x)
{
  /* search array a = (a_1, ... a_n)  */
  /* for x return true */
  /* if successful else false */
  for i←1 to n
    if (a_i = x)
      return true
    endif
  endfor

  return false
}
```

```
bool search(T a[], T x)
{
  for (int i=0; i<n; i++) {
    if (a[i] == x)
      return true;
  }

  return false;
}
```

find(12) ⟶ false



Time Complexity

- Worst case:
  - The worst case for a successful search is when the element is in the last location in the array
  - This takes n comparisons: worst case is $\Theta(n)$
- Best case:
  - The best case is when the element is in the first location
  - This takes 1 comparison: best case is $\Theta(1)$
- Average case:
  - Assume every location is equally likely to hold the key

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

- For an unsuccessful search n comparison are necessary

Binary Search

- If the array is ordered we can do better
- At each step we bisect the array

```
BinarySearch(a, x)
{
  low ←1
  high ←n
  while (low ≤ high)
    mid ←⌊(low + high)/2⌋
    if x > a_mid
      low ←mid + 1
    elseif x < a_mid
      high ←mid −1
    else
      return true
    endif
  endwhile
  return false
}
```

- ⋆ Based on a **divide-and-conquer** strategy
- ⋆ We check the middle of the array

$$\underbrace{a_1, a_2, \cdots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, \cdots a_n}_{x>a_m}$$

- ⋆ Based on a recursive idea

Binary Search in Action

BinarySearch(a, 88) not found

14	19	27	33	36	39	47	51	55	60	62	63	71	76	78	79	84	91	91	95
low	high	mid	high	mid				high	mid	low				mid			mid		low

Analysis

- We count the number of comparisons (counting each `if/else if` statement as a single comparison)
- Let $C(n)$ be the number of comparisons needed to search in an array of size n
- After one comparison we are left (in the worst case) with having to search an array not larger than $\lfloor n/2 \rfloor$, thus

$$C(n) < C(\lfloor n/2 \rfloor) + 1$$

- We've seen this relation before (lesson on Recursion)
- Easy to show $C(n) < \lfloor \log_2(n) \rfloor + 1 = O(\log(n))$

Outline

1. Algorithm Analysis
2. Search
3. **Simple Sort**
   - Insertion Sort
   - Selection Sort
4. Lower Bound

Sort Characteristics

- Sort is one of the best studied algorithms. We care about stability, space and time complexity
- A sort algorithm is said to be **stable** if it does not change the order of elements that have the same value
- Space Complexity. Sort is said to be
  - ⋆ **In-place** if the memory used is $O(1)$
- Time Complexity. In particular we are interested in
  - ⋆ Worst case
  - ⋆ Average case
  - ⋆ Best case

Insertion Sort

- In insertion sort we keep a subsequence of elements on the left in sorted order
- This subsequence is increased by *inserting* the next element into its correct position

```
InsertionSort(a)
{
  for i←2 to n
    v←a_i
    j←i-1
    while j ≥ 1 and a_j>v
      a_{j+1}←a_j
      j←j-1
    endwhile
    a_{j+1}←v
  endfor
}
```

23	37	39	50	66	69	69	74	84	90

sorted unsorted

Properties of Insertion Sort

- Insertion sort is **stable**. We only swap the ordering of two elements if one is strictly less than the other
- It is **in-place**
- Worst time complexity
  - ⋆ Occurs when the array is in inverse order
  - ⋆ Every element has to be moved to front of the array
  - ⋆ Number of comparisons for an array of size $C_w(n)$

$$C_w(n) = \sum_{i=2}^{n}(i-1) = 1 + 2 + \cdots n - 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Time Complexity

- Average Time Complexity
  - ⋆ On average we can expect that each new element being sorted moves half the way down sorted list
  - ⋆ This gives us an average time complexity, $C_a(n)$ of half the worst time

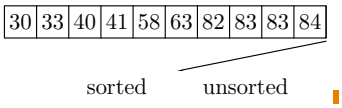
$$C_a(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$$

- Best Time Complexity
  - ⋆ This occurs if the array is already sorted
  - ⋆ In this case we only need $C_b(n) = n - 1 \in \Theta(n)$ comparisons
- Insertion sort is a good sort for small arrays because it is stable, in-place and is efficient when the arrays are almost sorted

Selection Sort

- A more direct **brute force** method is to find the least element iteratively
- We can make this an in-place method by swapping the least element with the first element, the second least element with the second element, etc.

```
SELECTIONSORT(a)
{
 for i←1 to n-1
  min ←i
  for j←i+1 to n
   if a_j<a_min
    min←j
   end if
  end for
  swap a_i and a_min
 end for
}
```

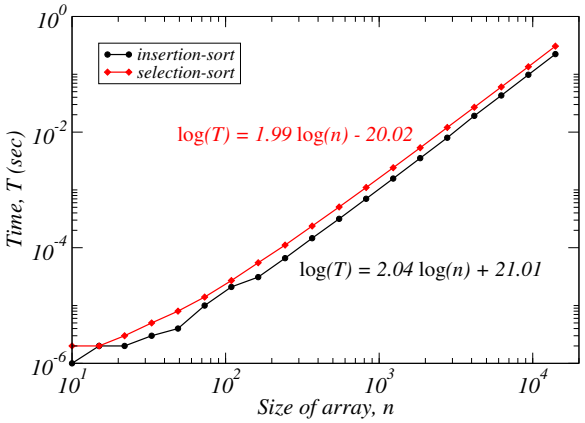


Analysis of Selection Sort

- Selection sort is in-place
- It isn't stable

- Selection sort always requires $n(n-1)/2$ comparisons so has the same worst case, but worse average case and best case complexity as insertion sort
- It only performs $n-1$ swaps—this makes it attractive (insertion sort moved more elements)

Insertion versus Selection Sort



Bubble Sort

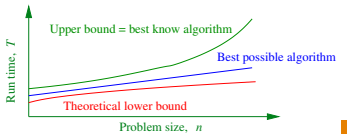
- There are many other simple sort strategies
- One popular one is bubble sort—keep on swapping neighbours until the array is sorted
- It is stable and in-place
- This again has $O(n^2)$ complexity
- This isn't bad for a simple sort, but it does do more work than insertion sort and selection sort
- Apart from its name it just doesn't have anything going for it

Outline

1. Algorithm Analysis
2. Search
3. Simple Sort
   - Insertion Sort
   - Selection Sort
4. **Lower Bound**

How Well Can You Do?

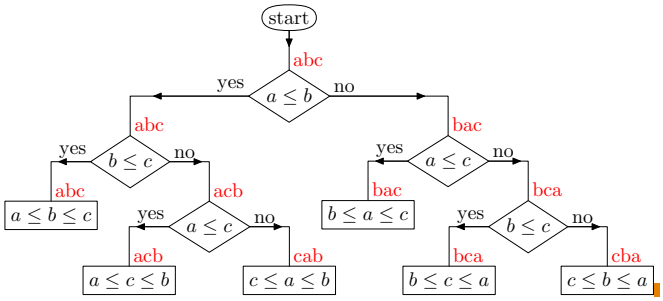
- Given a problem we would like to know what is the time complexity of the best possible program
- Usually there is no way of knowing this
- We can get an upper bound—if we know the time complexity of any algorithm that solves the problem we have an upper bound
- Lower bounds are far trickier
- A lower bound of $f(n)$ is a guarantee that we spend at least $f(n)$ operations to solve the problem



Decision Trees

- Decision trees are a way to visualise (at least, in principle) many algorithms
- They will eventually give us a lower bound on the time complexity of sort using binary decisions
- A decision tree shows the series of decisions made during an algorithm
- For sort based on binary comparisons the decision tree shows what the algorithm does after every comparison

Decision Tree for Insertion Sort



- Note there is one leaf for every possible way of sorting the list

Decision Trees and Time Complexity

- The time taken to complete the task is the depth of the tree at which we finish (i.e. the leaf nodes)
- We can thus read of the time complexity
  - ⋆ worst case time: depth of the deepest of leaf
  - ⋆ best case time: depth of the shallowest of leaf
  - ⋆ average case time: average depth of leaves
- Different sort strategies will have different decision trees
- Decision trees are usually far too large to write out ☹

Requirements of Correct Sort

- Any sort based on binary comparisons must have a leaf of the tree for every possible way of sorting the list
- The array $[a, b, c]$ must be arranged differently for all combinations

$$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$$

- That is they must go through a different path of the decision tree
- If not sort won't work

Minimum Number of Leaves

- There must be, at least, one leaf node of the decision tree for each possible permutation of the list
- How many permutations are there of a list of size n?
- Start with a sequence $(a_1, a_2, \ldots, a_n)$
- To create a new permutation we can choose any member of the list as the first element
- We can choose any of the remaining $n - 1$ elements of the list as the second element of the permutation
- The total number of permutation is $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = n!$

Lower Bound Time Complexity for Sorting

- Any sort algorithm using binary comparisons must have a decision tree with at least $n!$ leaf nodes
- This will be a binary tree with some depth d
- The number of leaves at depth d is 2^d
- Thus the smallest depth tree must have a depth d such that $2^d \geq n!$
- That is, the depth of the decision tree satisfies $d \geq \log_2(n!)$
- But this is the number of comparisons needed in our sort
- We are left with a lower bound on the time complexity of $\log_2(n!)$

How Big is $\log_2(n!)$

- We showed in the second lecture that

$$\left(\frac{n}{2}\right)^{n/2} < n! < n^n$$

- It is not too difficult to show that asymptotically (i.e. as $n \to \infty$) that $n!$ approaches $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$—this is known as **Stirling's approximation**
- Thus

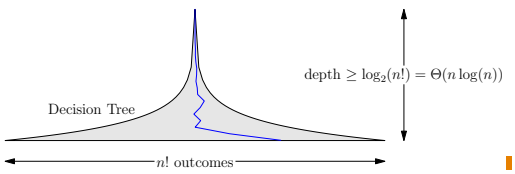
$$\begin{aligned}\log_2(n!) &\approx n\log_2(n) - n\log_2(e) + \frac{\log_2(n)}{2} + \frac{\log_2(2\pi)}{2} \\ &= \Theta(n\log_2(n))\end{aligned}$$

Complexity of Sorting

- We therefore have a lower bound on the time complexity of $\Omega(n\log(n))$
- This is true for any sort using binary comparisons
- We will see in the next lecture there exists algorithms with time complexity $O(n\log(n))$
- This means our lower bound is tight—i.e. it is the true cost of the best algorithm
- Having a lower bound we know we are not going to obtain a substantially faster algorithm

Lessons

- Analysis of algorithms is hard
- Analysis is important: without it we don't know if we have a good algorithm or whether we should try to find a more efficient one
- Lower bounds are particularly important



Algorithms and Analysis

Lesson 17: *Sort Wisely*

Merge sort, quick sort and radix sort

Outline

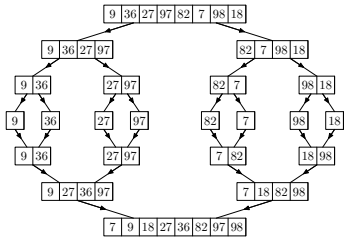
1. Merge Sort
2. Quick Sort
3. Radix Sort

Merge Sort

- Merge sort is an example of sort performed in log-linear (i.e. $O(n \log(n))$) time complexity
- It was invented in 1945 by John von Neumann
- It is an example of a divide-and-conquer strategy
  - ⋆ That is, the problem is divided into a number of parts recursively
  - ⋆ The full solution is obtained by recombining the parts

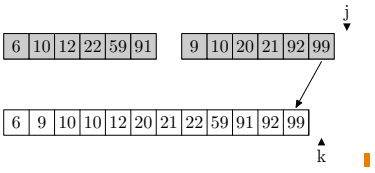
Algorithm

```
MergeSort(a)
{
 if n > 1
   copy a[1 : ⌊n/2⌋] to b
   copy a[⌊n/2⌋ + 1 : n] to c
   MergeSort(b)
   MergeSort(c)
   Merge(b, c, a)
 endif
}
```



Merge

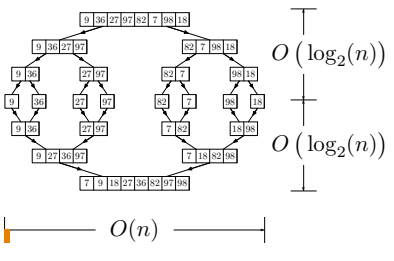
```
Merge(b[1 : p], c[1 : q], a[1 : p + q])
{
 i←1
 j←1
 k←1
 while i ≤ p and j ≤ q do
  if b_i ≤ c_j
   a_k ←b_i
   i ←i+1
  else
   a_k ←c_j
   j ←j+1
  endif
  k ←k+1
 end
 if i=p
  copy c[j : q] to a[k : p + q]
 else
  copy c[i : q] to a[k : p + q]
}
```



Properties of Merge Sort

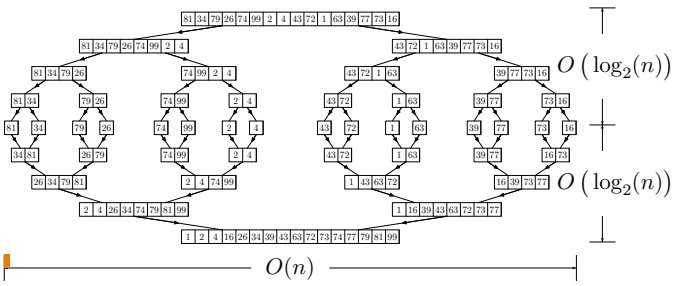
- Merge sort is stable provided we merge carefully (i.e. it preserves the order of two entries with the same value)
- Merge sort isn't in-place—we need an array of at most size n to do the merging
- Merging is quick. Given two arrays of size n the most number of comparisons we need to perform is $n - 1$

Time Complexity of Merge Sort



$n = 8$

Time Complexity of Merge Sort



$n = 16$

Time Complexity

- We again measure the complexity in the number of comparisons
- From the above argument $C(n) = O(n \times \log_2(n))$
- We can be a bit more formal

$$C(n) = 2C(\lfloor n/2 \rfloor) + C_{\text{merge}}(n) \qquad \text{for } n > 1$$
$$C(0) = 1$$

- But in the worst case $C_{\text{merge}}(n) = n - 1$
- Leads to $C_{\text{worst}}(n) = n \log_2(n) - n + 1$

General Time Complexity

- In general if we have a recursion formula

$$T(n) = aT(n/b) + f(n)$$

with $a \geq 1$, $b > 1$

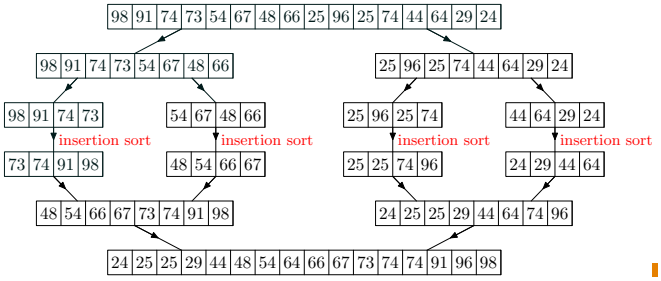
- If $f(n) \in \Theta(n^d)$ where $d \geq 0$ then

$$T(n) \in \begin{cases} \Theta\left(n^d\right) & \text{if } a < b^d \\ \Theta\left(n^d \log(n)\right) & \text{if } a = b^d \\ \Theta\left(n^{\log_d(a)}\right) & \text{if } a > b^d \end{cases}$$

- Analogous results hold for the family O and Ω

Mixing Sort

- For very short sequences it is faster to use insertion sort than to pay the overhead of function calls



Outline

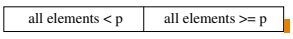
1. Merge Sort
2. **Quick Sort**
3. Radix Sort

Quicksort

- The most commonly used fast sorting algorithm is **quicksort**
- It was invented by the British computer scientist by C. A. R. Hoare in 1962
- It again uses the divide-and-conquer strategy
- It can be performed in-place, but it is **not** stable
- It works by splitting an array into two depending on whether the elements are less than or greater than a **pivot** value
- This is done recursively until the full array is sorted

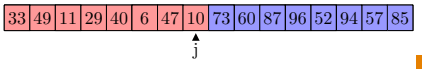
Partition

- We need to partition the array around the pivot p such that



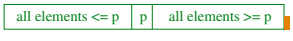

```
PARTITION(a, p, left, right)
{
  i ←left
  j ←right
  repeat {
    while a_i < p
      i++
    while a_j ≥ p
      j--
    if i ≥ j
      break
    SWAP(a_i, a_j)
  }
}
```

pivot = 52



Optimising Partitioning

- There are different ways of performing the partitioning
- We want to minimise the time taken on the inner loop
- This means we want to perform as few checks as possible
- One method of doing this is to place *sentinels* at the ends of the array
- We can also reduce work by placing the partition in its correct position



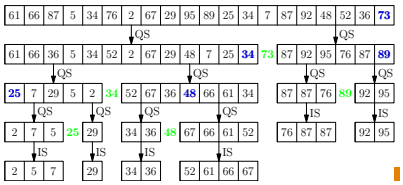
Choosing the Pivot

- There are different strategies to choosing the pivot
- Choose the first element in the array
- Choose the median of the first, middle and last element of the array
- This increases the likelihood of the pivot being close to the median of the whole array
- For large arrays (above 40) the median of 3 medians is often used

Quicksort

We recursively partition the array until each partition is small enough to sort using insertion sort

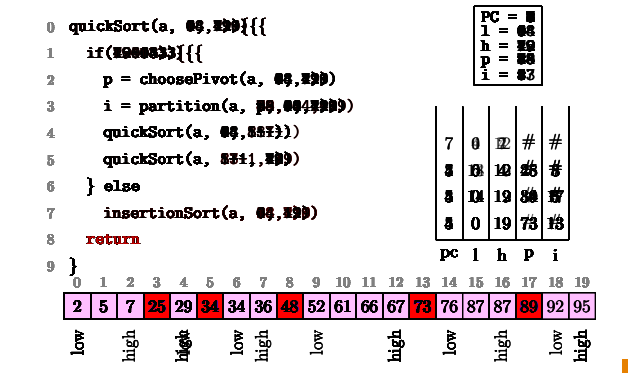
```
QuickSort(a, left, right) {
  if (right-left < threshold)
    InsertionSort(a, left, right)
  else
    pivot = ChoosePivot(a, left, right)
    part = Partition(a, pivot, left, right)
    QuickSort(a, left, part-1)
    QuickSort(a, part+1, right)
  endif
}
```



Time Complexity

- Partitioning an array of size n takes $\Theta(n)$ operations
- If we split the array in half then number of partitions we need to do is $\lceil \log_2(n) \rceil$
- This is the best case thus quicksort is $\Omega\left(n \log(n)\right)$
- If the pivot is the minimum element of the array then we have to partition $n-1$ times
- This is the worst case so quicksort is $O\left(n^2\right)$
- This worst case will happen if the array is already sorted and we choose the pivot to be the first element in the array!

QuickSort



Sort in Practice

- The STL in C++ offers three sorts
  - ⋆ `sort()` implemented using quicksort
  - ⋆ `stable_sort()` implemented using mergesort
  - ⋆ `partial_sort()` implemented using heapsort
- Java uses
  - ⋆ Quicksort to sort arrays of primitive types
  - ⋆ Mergesort to sort Collections of objects
- Quicksort is typically fastest but has worst case quadratic time complexity

Selection

- A related problem to sorting is selection
- That is we want to select the k^{th} largest element
- We could do this by first sorting the array
- A full sort is not however necessary—we can use a modified quicksort where we only continue to sort the part of the array we are interested in
- This leads to a $\Theta(n \log(n))$ algorithm which is considerably faster then sorting

Outline

1. Merge Sort
2. Quick Sort
3. **Radix Sort**

Radix Sort

- Can we get a sort algorithm to run faster than $O\left(n \log(n)\right)$?
- Our proof that this was optimal assumed we were performing binary decisions (is a_i less than a_j?)
- If we don't perform pairwise comparisons then the proof doesn't apply
- Radix sort is the classic example of a sort algorithm that doesn't use pairwise comparisons

Sorting Into Buckets

- The idea behind radix sort is to sort the elements of an array into some number of buckets
- This is done successively until the whole array is sorted
- Consider sorting integers in decimals (base 10 or radix 10)
- We can successively sort on the digits
- The sort finishes when we have got through all the digits

Radix Sort in Action

11	0	null
13	1	null
26	2	null
29	3	null
37	4	null
43	5	null
51	6	null
51	7	null
52	8	null
79	9	null

Time Complexity of Radix Sort

- We need not use base 10 we could use base r (the radix)
- If the maximum number to be sorted is N then the number of iterations of radix sort is $\log_r(N)$
- Each sort involves n operations
- Thus the total number of operations is $O\left(n\lceil\log_r(N)\rceil\right)$
- Since N does not depend on n we can write this as $O\left(n\right)$

Bucket Sort

- A closely related sort is bucket sort where we divide up the inputs into buckets based on the most significant figure
- We then sort the buckets on less significant figures
- Quicksort is a bucket sort with two buckets, but where we choose a pivot to determine which bucket to use

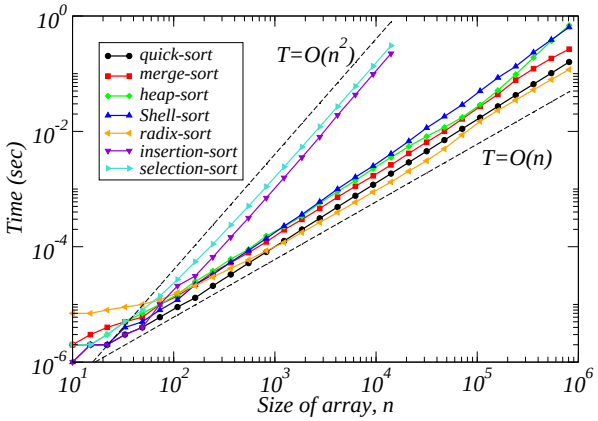
Minimum Time for Sort

- Can we do better?
- In any sort we need to examine all possible elements in the array
- If there is an element that isn't examined then we don't know where to put it
- Thus the lower bound on any sort algorithm is $\Omega(n)$

Practical Sort

- In practice, radix sort or bucket sort are rarely used
- The overhead of maintaining the buckets make them less efficient than they might appear
- Radix sort is harder to generalise to other data types than comparison based sorts
- In practice quick sort and merge sort are usually preferred
- Having said that there are some very neat implementations of radix sort

Comparison of Sort Algorithms

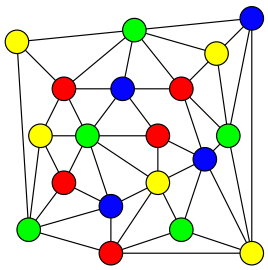


Lessons

- Sort is important—it is one of the commonest high level operations
- Merge sort and quick sort are the most commonly used sort
- There are sorts that have a better time complexity that quicksort
- In practice it is difficult to beat quicksort

Algorithms and Analysis

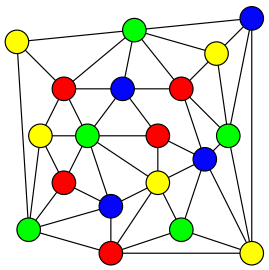
Lesson 19: *Think Graphically*



Graph theory, applications of graphs, graph problems

Outline

1. **Graph Theory**
2. Applications of Graphs
   - Geometric applications
   - Relational applications
3. Implementing Graphs
4. Graph Problems

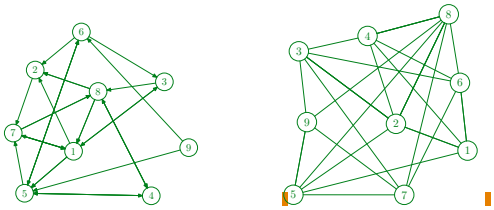


Motivation

- Many different problems can be described in terms of graphs
- This often reveals the true nature of the problem
- It unifies many apparently different problems
- As much is known about graph problems it often provides a pointer to the solution

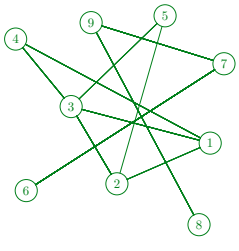
Definition of a Graph

- A graph, G, can be described by
  - A set of vertices or nodes $\mathcal{V} = \{1, 2, 3 \ldots n\}$
  - A set of edges $\mathcal{E} = \{(i, j) | \text{vertex } i \text{ is connected to vertex } j\}$
- The edges may be
  - **directed**—sometimes called a **digraph**
  - **undirected**



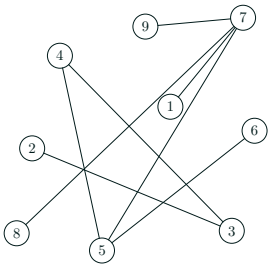
Connected and Unconnected Graphs

- A graph is **connected** if you can get from one node to any other along a series of edges
- Otherwise it is **disconnected**



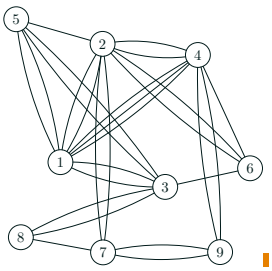
Trees

- A tree is a connected graphs with no cycles
- A tree will have $n - 1$ edges



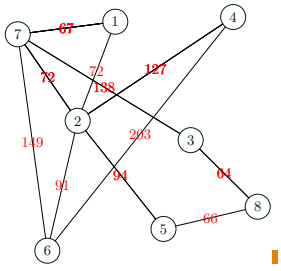
Multigraphs

- If the collection of edges is a *multiset* then we obtain a **multigraphs** where more than one edge is allowed between pairs of vertices



Weighted Graphs

- If we assign a number to an edge we obtain a **weighted graph**

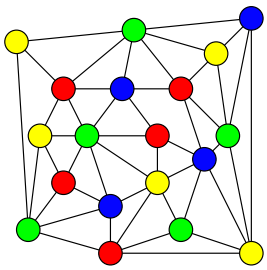


Networks

- Sometimes we add more information to the graph
- E.g. attributes to the nodes or edges
- Graphs with many attributes are often referred to as **networks**

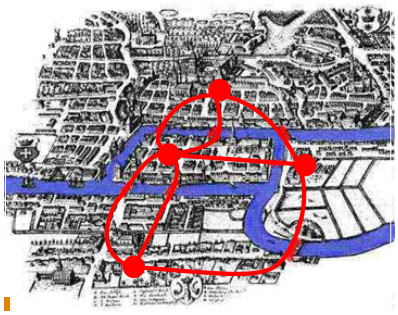
Outline

1. Graph Theory
2. **Applications of Graphs**
   - Geometric applications
   - Relational applications
3. Implementing Graphs
4. Graph Problems



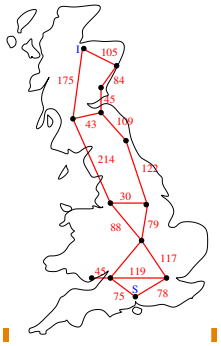
Bridges of Königsberg

Is there a tour around Königsberg going over every bridge once?



In 1736 Euler published a paper answering this question and founding graph theory

Representing Distances



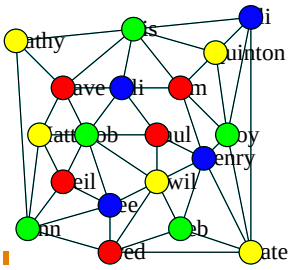
- Consider some graph
- With weights representing the distance between nodes
- What is the shortest distance between S and I?

Other Applications

- We could take the weights to represent the time taken to travel between nodes
- In a computer network the weights might represent the bandwidth
- In a representation of a transport system the weights might represent the carrying capacity of the traffic on a road
- Graphs can be used to represent other kinds of relationships
- E.g. We could create a digraph of links between web pages

Christmas Card Problem

- I have four types of Christmas cards
- Some of my friends know each other

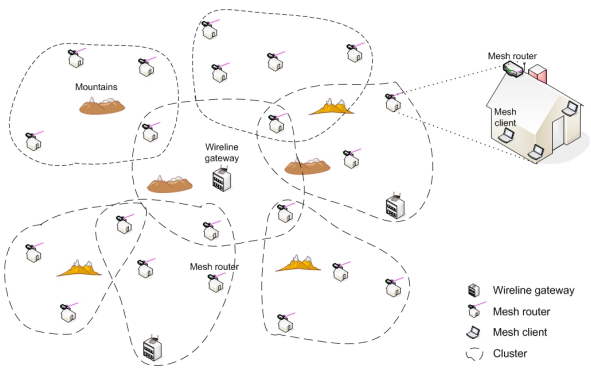


- I don't want to send friends that know each other the same card

A Real World Problem

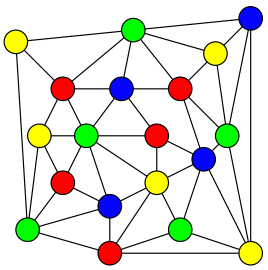
- A food company used different colour bags for each of it products
- To save money they reduced the stock of bags to 25
- They wanted to know what items to put in what bags so that as few customers as possible would have items with the same colour bags
- This can again be reduced to a graph colouring problem
  - Each node represents an item
  - The edges were weighted by the number of customers that took both items
  - The aim was to colour the nodes with 25 colours to minimise the weights where the edges shared the same colour

Frequency Assignment Problem



Outline

1. Graph Theory
2. Applications of Graphs
   - Geometric applications
   - Relational applications
3. **Implementing Graphs**
4. Graph Problems



Representations

- There is no single way to represent graphs
- The best representation depends on the graph
- Some books describe a $Graph\ ADT$—graphs are too varied for this to be very useful
- An important issue in representing a graph is how to store the edge information

Adjacency Matrices

- One representation of a graph $G = (\mathcal{V}, \mathcal{E})$ is in term of an $n \times n$ **adjacency matrix A** with elements

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{if } (i,j) \notin \mathcal{E} \end{cases}$$

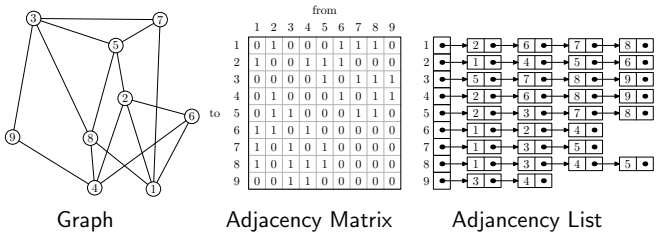
  where $n = |\mathcal{V}|$
- For undirected graphs **A** is a symmetric matrix, i.e. $\mathbf{A} = \mathbf{A}^{\mathsf{T}}$
- For weighted graphs we often store the **connectivity matrix** or **cost-adjacency matrix**, **C**, where

$$C_{ij} = \begin{cases} w_{ij} & \text{if } (i,j) \in \mathcal{E} \\ 0 & \text{if } (i,j) \notin \mathcal{E} \end{cases}$$

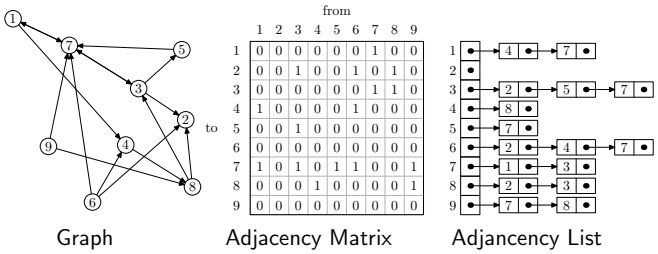
Adjacency Lists

- For **dense** graphs where the number of edges is $\Theta(n^2)$ the adjacency matrix is often a useful representation
- But in **sparse** graphs where the number of edges is $\Theta(n)$ the adjacency matrix has a very large number of zeros
- A more efficient representation is in terms of the adjacency list where the set of outgoing edges is stored for each node
- In some applications it is useful to store both the adjacency matrix and the adjacency list

Representing Undirected Graphs

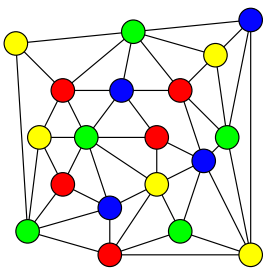


Representing Digraphs



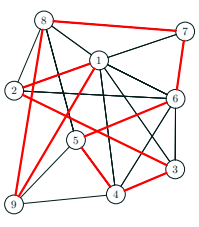
Outline

1. Graph Theory
2. Applications of Graphs
   - Geometric applications
   - Relational applications
3. Implementing Graphs
4. **Graph Problems**



Hamilton Cycle

- The Euler path problem is to find a path through a multigraph that passes through every edge once—easy to solve
- The Hamilton cycle problem is to find a cycle that goes through each vertex exactly once



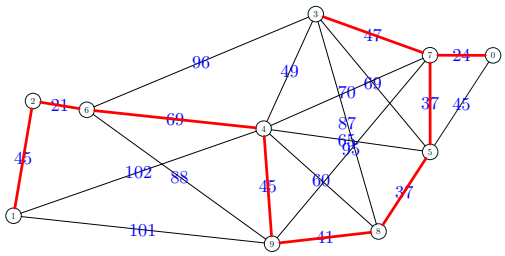
- There is no known efficient algorithm to solve this

Shortest Path and TSP

- The shortest path problem is to find a path between two nodes
- There is an efficient algorithm—see next lecture
- In the travelling salesperson problem the task is to find the shortest tour (Hamilton cycle)—we usually assume there is an edge between every pair of nodes
- There is no know efficient algorithm to solve all TSPs

Minimum Spanning Tree

- Suppose we want to construct pylons connecting a number of cities using the least amount of cable



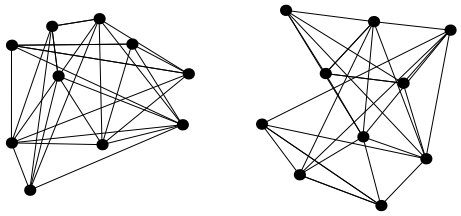
- We will study an efficient algorithm to solve this in the next but one lecture

Graph Partitioning

- The simplest version of this problem is to cut a graph into two equal halves so that you minimise the number of edges you cut
- If the edges are weighted then you want to minimise the sum of edges that are cut
- If the vertices are weighted you want to balance the sum of vertex weights in the two partitions
- An example of this problem is in dividing up a problem to run on a parallel computer
  - ⋆ Nodes are subtasks (weights on nodes are run times)
  - ⋆ Edge weights indicate communication cost
- There is no known efficient algorithm to solve this

Graph Isomorphism

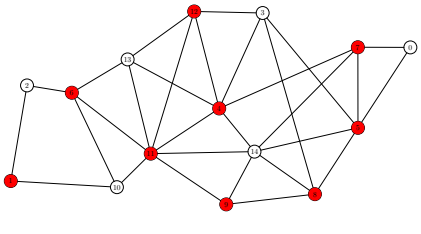
- Do two graphs have the same structure?



- There is no known efficient algorithm to solve this problem
- Theoretically it is interesting because it is not NP-complete

Vertex Cover

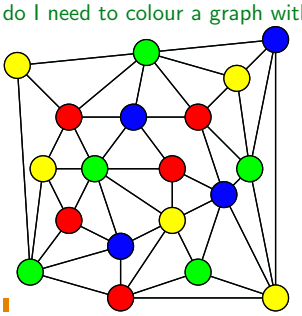
- How many guards do you need to cover all the corridors in a museum



- There is no known efficient algorithm to solve this

Graph Colouring

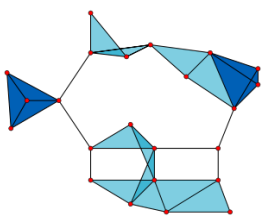
- How many colours do I need to colour a graph with no conflicts



- There is no known efficient algorithm to solve this

Other Graph Problems

- These are only a sample of the many famous graph problems
- Others include
  - ⋆ Max-clique (hard)
  - ⋆ Maximal independent set (hard)
  - ⋆ Maximal flow problem (easy)
  - ⋆ Max-cut (hard)

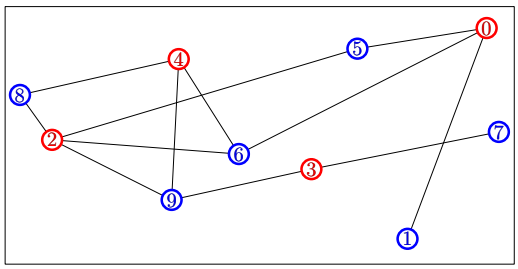


Lessons

- Graphs are an important method for abstracting problems
- They appear in a huge number of disparate fields
- There are many problems for which efficient algorithms are known
- There are many problems which are believed to be hard—i.e. there aren't any efficient algorithms
- Even for hard problems there are good algorithms for finding approximated solutions

Algorithms and Analysis

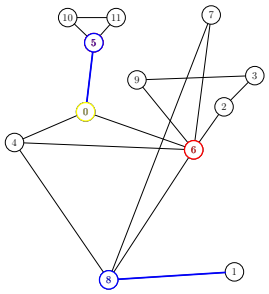
Lesson 20: *Learn to Traverse Graphs*



Breadth First Search, Depth First Search, Topological Sort

Outline

1. **Breadth First Search**
   - BFS applications
2. Depth First Search
   - DFS applications
3. Topological Sort



Basic Graph Algorithms

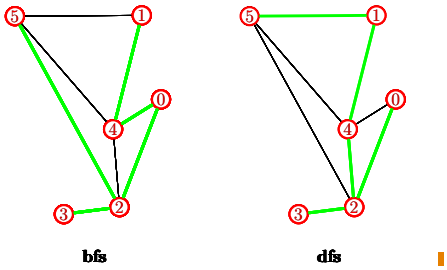
- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
  - ⋆ Breadth First Search
  - ⋆ Depth First Search

Advanced Generic

- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We then follow edges to create a tree

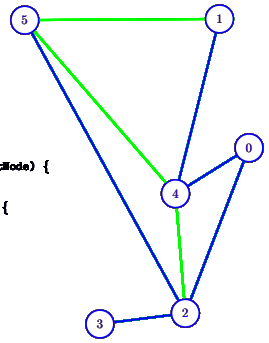


Breadth First Search

```
bfs(graph, node) {
  List state(graph.noNodes, "undiscovered")
  List parent(graph.noNodes)
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    foreach neighbour ∈ Neighbourhood(currentNode) {
      if (state[neighbour] ≠ "processed")
        processEdge(currentNode, neighbour)
      if (state[neighbour] = "undiscovered") {
        state[neighbour] ← "discovered"
        parent[neighbour] ← currentNode
        q.enqueue(neighbour)
      }
    }
    processVertexLate(currentNode)
  }
}
```

currentNode=8 neighbour=8

q= | 8 | 8 | 8 | | | |

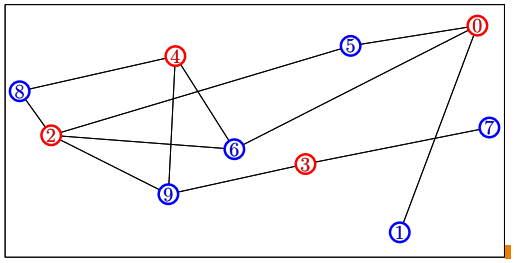


Applications of Breadth First Search

- Breadth first search can be used to find the shortest path from a source node to a destination node for an **unweighted** graph
  - ⋆ Run `bfs(graph,source)`
  - ⋆ Use parent information to find path from destination back to source
- BFS (as well as DFS) can be used to find connected components
  - ⋆ Use *processVertexEarly* to mark vertices connected to the current connected component
  - ⋆ Run *bfs* from all vertices that are not labelled

Bipartite Graphs

- Bipartite graphs are graphs where the vertices can be split into two sets so that there are no edges between vertices in the same graph



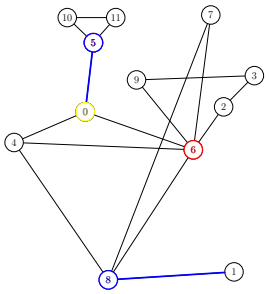
Checking Bipartiteness (Two-colourability)

- Each edge must connect nodes from different sets

```
isBipartite(graph) {
   colour = List(graph.noNodes(), "white");
   bipartite = true;

   foreach node in graph {
      if (colour[node] == "white") {
         colour[node] = "red";
         bfs(graph, node);
      }
   }
   return bipartite;
}

processEdge(node1, node2) {
   if (colour[node1] == colour[node2])
       bipartite = false;
   colour[node2] = (colour[node1]=="red")? "blue":"red";
}
```


Outline

1. Breadth First Search
   - BFS applications
2. **Depth First Search**
   - DFS applications
3. Topological Sort

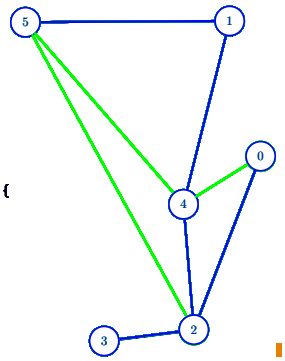


Depth First Search

- Depth first search is essentially like breadth first search except we replace the queue by a stack
- In practice it is often implemented using recursion rather than a stack
- It proves useful to keep a record of the traversal **time** for each vertex
  - ⋆ the clock ticks each time a vertex is entered or exited

Depth First Search

```
dfs(graph, node) {
  state ← Array[n, "undiscovered"]
  finished ← false
  dfs_recur(graph, node)
}

dfs_recur(graph, node) {
  if (finished) return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] = "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs_recur(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if (finished) return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}

    node=0    neighbour=Nil  time=0
```



Applications of DFS

- Depth first search has many applications
- Suppose we want to check if the graph is a tree (i.e. has no cycles)
- The only edges that are allowed are parent edges

```
processEdges(node1, node2) {
  if (parent[node1] ≠ node2) {
    isTree ←false
    finish ←true
  }
}
```

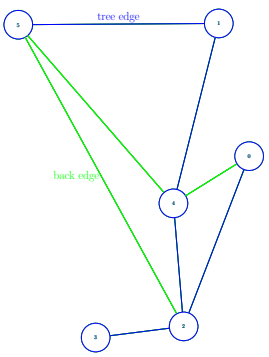
(note that we set `finish` to stop DFS prematurely)

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m+n))$ time. Can we do this any faster?

Single Pass Algorithm

- In DFS we divide the edges into tree edges that define the search tree (edges between nodes and parents) and back edges which take us back to vertices we have already seen
- Without the back edges we have a tree where all non-leaf nodes are articulation nodes
- The back edges secure the edges to the rest of the tree



Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a

  back edge we update the reachable ancestor (we check the $entryTime$ to see if the edge leads to a vertex which was discovered earlier)

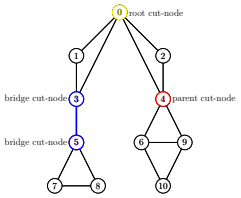
  tree edge we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

Types of Articulated Vertices

- Key is to recognise that articulated vertices only occur in three version

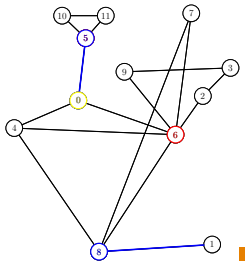
  Root cut-nodes Occur when the root has more than one child
  Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node
  Parent cut-nodes If the earliest reachable vertex is it parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



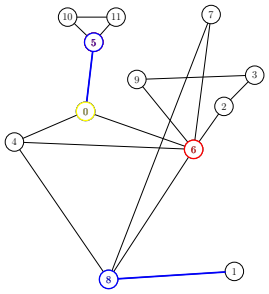
Biconnectivity Summary

- Algorithmic details are not too important
- One pass $(O(n+m))$ algorithm
- Uses `processVertexEarly`, `procesEdge` and `processVerexLate` methods
- Bridge cuts also shows articulation edges



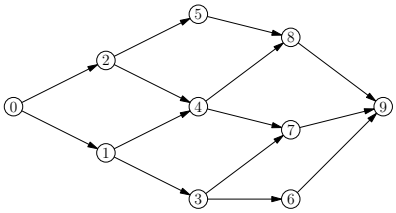
Outline

1. Breadth First Search
   - BFS applications
2. Depth First Search
   - DFS applications
3. **Topological Sort**



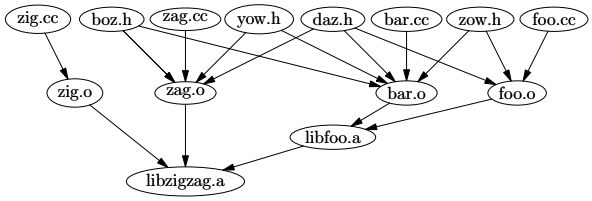
DAGs

- Directed acyclic graphs or DAGs are directed graphs without cycles
- They are often used to represent complex processes
  - Vertices are processes
  - Directed edge (i, j) indicates process i needs to occur before process j



Program Compilation

- One example of a DAG is in compiling programs
- Some programs depend on other programs so they need compiling first



Other Applications

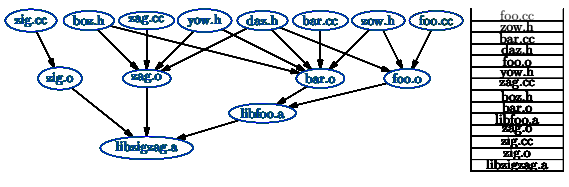
- The same problem occurs in compiling classes
  - The implementation of a class can depend on the implementation of other graphs
  - What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.—in what order should they precess?
- If the graphs were not acyclic it would be impossible process them!

Topological Sort

- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges
- That is, for each edge (i, j), vertex i will occur before vertex j
- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts
- Topological sort is not a "sort", but it is a useful algorithm for some applications

Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



DFS on Digraphs

DFS for undirected graphs

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] ≠ "discovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed")∨{ graph is directed) {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```


Implementing Topological Sort

- Given our DFS programme we define

```
topologicalSort(graph) {
  Stack stack
  for node ∈ graph.vertexSet()
    if (¬discovered[node])
       dfs(graph, node)

  List topSortList
  while (¬stack.isEmpty())
    topSortList.add(stack.pop())

  return topSortList
}
```


Enhance DFS

- Requires us to define a couple of helper function

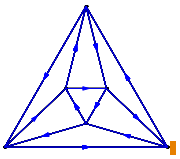
```
processVertexLast(node) {
  stack.push(node)
}

processEdge(currentNode, neighbour) {
  if (state[neighbour] == "processed") {
    print "error: graph not a DAG"
    finished = true
  }
}
```


Implementation Issues

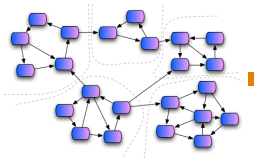
- Most awkward part of the implementation is that the `topologicalSort` algorithm needs access to dfs structures (`discovered[]`)
- `processVertexLast(node)` needs access to the stack
- Need to be able to redefine `processVertexFirst`, `processEdge` and `processVertexLast`
- Different languages and libraries cope with this differently
  - Java: JDSL, JGraphT
  - C++: Boost Graph Library, LEDA

Other Applications

- DFS is used for many other classic problems
- Euler Cycles



- Strongly Connected Components

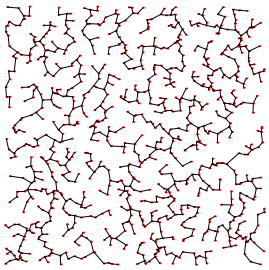


Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)

Algorithms and Analysis

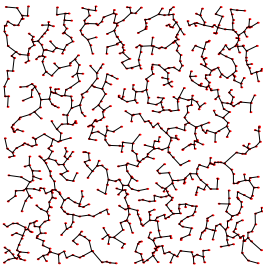
Lesson 21: *Know Your Graph Algorithms*



Weighted graph algorithms, Minimum spanning tree, Prim, Kruskal, shortest path, Dijkstra

Outline

1. **Minimum Spanning Tree**
2. Prim's Algorithm
3. Kruskal's Algorithm
4. Shortest Path

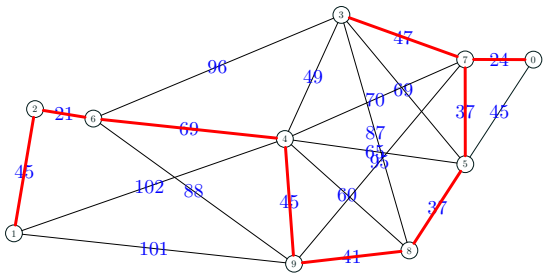


Graph Algorithms

- We consider a graph algorithm to be **efficient** if it can solve a graph problem in $O(n^a)$ time for some fixed a
- That is, an efficient algorithm runs in polynomial time
- A problem is **hard** if there is no known efficient algorithm
- This does **not** mean the best we can do is to look through all possible solutions—see later lectures
- In this lecture we are going to look at some efficient graph algorithms for weighted graphs

Minimum spanning tree

- A minimal spanning tree is the shortest tree which spans the entire graph

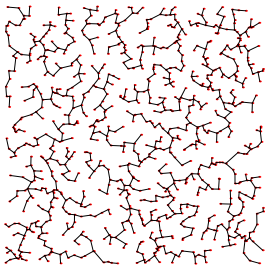


Greedy Strategy

- We consider two algorithms for solving the problem
  - Prim's algorithm (discovered 1957)
  - Kruskal's algorithm (discovered 1956)
- Both algorithms use a **greedy strategy**
- Generally greedy strategies are not guaranteed to give globally optimal solutions
- There exists a class of problems with a **matroid** structure where greedy algorithms lead to globally optimal solutions
- Minimum spanning trees, Huffman codes and shortest path problems are matroids

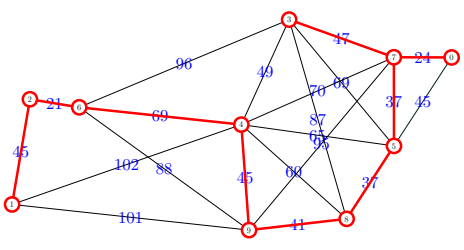
Outline

1. Minimum Spanning Tree
2. **Prim's Algorithm**
3. Kruskal's Algorithm
4. Shortest Path



Prim's Algorithm

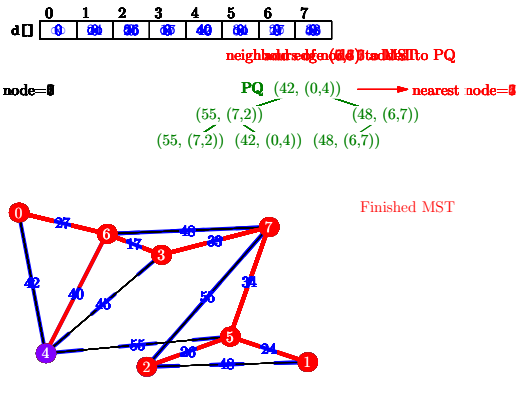
- Prim's algorithm grows a subtree greedily
- Start at an arbitrary node
- Add the shortest edge to a node not in the tree



Pseudo Code

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞            \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅              \\ Set of edges in subtree
  PQ.initialise()     \\ initialise an empty priority queue
  node ← v_1          \\ where v_1 ∈ V is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh < d_neigh )
        d_neigh ←w_node,neigh
        PQ.add( (d_neigh, (node,neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

Prim's Algorithm in Detail



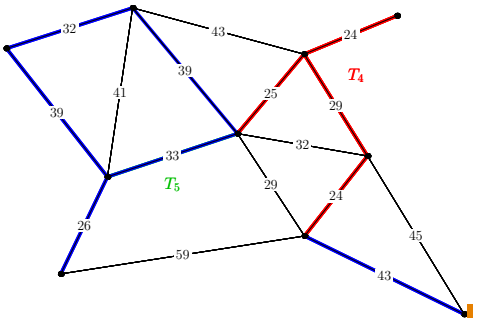
Why Does This Work?

- Clearly Prim's algorithm produces a spanning tree
  - ⋆ It is a tree because we always choose an edge to a node not in the tree
  - ⋆ It is a spanning tree because it has $|\mathcal{V}| - 1$ edges
- Why is this a minimum spanning tree?
- Once again we look for a proof by induction

Proof by induction

- We want to show that each subtree, T_i, for $i = 1, 2, \cdots, n$ is part of (a subgraph) of some minimum spanning tree
- In the base case, T_1 consists of a tree with no edges, but this has to be part of the minimum spanning tree
- To prove the inductive case we assume that T_i is part of the minimum spanning tree
- We want to prove that T_{i+1} formed by adding the shortest edge is also part of the minimum spanning tree
- We perform the proof by contradiction—we assume that this added edge isn't part of the minimum spanning tree

Contrariwise



Loop Counting

```
PRIM(G = (V, E, w)) {
  for i←0 to |V|
    d_i ←∞
  endfor
  E_T ←∅
  PQ.initialise()
  node ← v_1
  for i←1 to |V| − 1                    // loop 1  O(|V|)
    d_node ←0
    for k ∈ {v ∈ V|(node, v) ∈ E}  // inner loop O(|E|/|V|)
      if ( w_node,k < d_k )
        d_k ←w_node,k
        PQ.add( (d_k, (node,k)) )     // O(log(|E|))
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```


Run Time

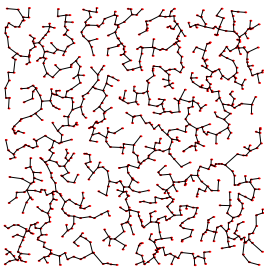
- The worst time is

$$O(|\mathcal{V}|) \times O\left(\frac{|\mathcal{E}|}{|\mathcal{V}|}\right) \times O(\log(|\mathcal{E}|)) = O\left(|\mathcal{E}|\log(|\mathcal{E}|)\right)$$

- Note that $|\mathcal{E}| < |\mathcal{V}|^2$
- Thus, $\log(|\mathcal{E}|) < 2\log(|\mathcal{V}|) = O\left(\log(|\mathcal{V}|)\right)$
- Thus the worst case time complexity is $|\mathcal{E}|\log(|\mathcal{V}|)$

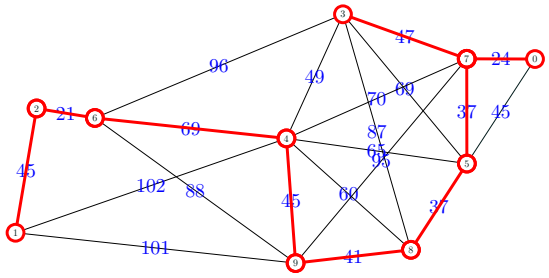
Outline

1. Minimum Spanning Tree
2. Prim's Algorithm
3. **Kruskal's Algorithm**
4. Shortest Path



Kruskal's Algorithm

- Kruskal's algorithm works by choosing the shortest edges which don't form a loop



Pseudo Code

```
KRUSKAL(G = (V, E, w))
{
  PQ.initialise()
  for edge ∈ |E|
    PQ.add( (w_edge, edge) )
  endfor

  E_T ←∅
  noEdgesAccepted ←0

  while (noEdgesAccepted < |V| − 1)
    edge ←PQ.getMin()
    if E_T ∪ {edge} is acyclic
      E_T ←E_T ∪ {edge}
      noEdgesAccepted ←noEdgesAccepted +1
    endif
  endwhile

  return E_T
}
```


Analysis

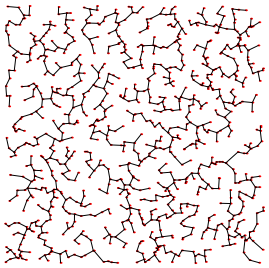
- Kruskal's algorithm looks much simpler than Prim's
- The sorting takes most of the time, thus Prim's algorithms is $O(|\mathcal{E}|\log(|\mathcal{E}|)) = O(|\mathcal{E}|\log(|\mathcal{V}|))$
- We can sort the edges however we want—we could use quick sort rather than heap sort using a priority queue
- But we haven't specified how we determine if the added edge would produce a cycle

Cycling

- For a path to be a cycle the edge has to join two nodes representing the same subtree
- To compute this we need to quickly **find** which subtree a node has been assigned to
- Initially all nodes are assigned to a separate subtree
- When two subtrees are combined by an edge we have to perform the **union** of the two subtrees
- But that is precisely the **union-find** algorithm we covered in lecture 13

Outline

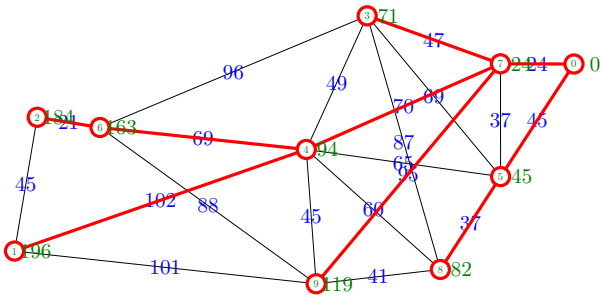
1. Minimum Spanning Tree
2. Prim's Algorithm
3. Kruskal's Algorithm
4. **Shortest Path**



Shortest path

- We can efficiently compute the shortest path from one vertex to any other vertex
- This defines a spanning tree, but where the optimisation criteria is that we choose the vertex that are closest to the *source*
- To find this spanning tree we use Dijkstra's algorithm where we successively add the nearest node to the source which is connected to the subtree built so far
- This is very close to Prim's algorithm and has the same complexity

Dijkstra's Algorithm



Pseudo Code

```
DIJKSTRA(G = (V, E, w), source) {
  for i←0 to |V|
    d_i ←∞          \\ Minimum 'distance' to source
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()  \\ initialise an empty priority queue
  node ←source
  d_node ←0
  for i←1 to |V| − 1
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh + d_node < d_neigh )
        d_neigh ←w_node,neigh + d_node
        PQ.add( (d_neigh, (node, neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    while next_node not in subtree
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```


Compare to Prim's Algorithm

```
PRIM(G = (V, E, w)) {
  for i←1 to |V|
    d_i ←∞          \\ Minimum 'distance' to subtree
  endfor
  E_T ←∅            \\ Set of edges in subtree
  PQ.initialise()  \\ initialise an empty priority queue
  node ← v_1        \\ where v_1 ∈ V  is arbitrary
  for i←1 to |V| − 1
    d_node ←0
    for neigh ∈ {v ∈ V|(node, v) ∈ E}
      if ( w_node,neigh < d_enigh )
        d_neigh ←w_node,neigh
        PQ.add( (d_neigh, (node,neigh)) )
      endif
    endfor
    do
      (a_node, next_node) ←PQ.getMin()
    until (d_next_node > 0)
    E_T ←E_T ∪ {(a_node, next_node)}
    node ←next_node
  endfor
  return E_T
}
```

Dijkstra Details

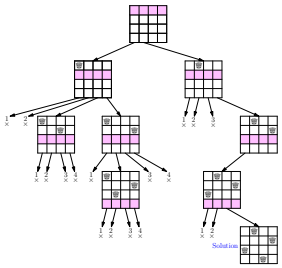
- Dijkstra is very similar to Prim's (it differs in the distances that are used)
- It has the same time complexity
- It can be viewed as using a greedy strategy
- It can also be viewed as using the dynamic programming strategy (see lecture 22)

Lessons

- There are many efficient (i.e. polynomial $O(n^a)$) graph algorithms
- Some of the most efficient ones are based on the Greedy strategy
- These are easily implemented using priority queues
- Minimum spanning trees are useful because they are easy to compute
- Dijkstra's algorithm is one of the classics

Algorithms and Analysis

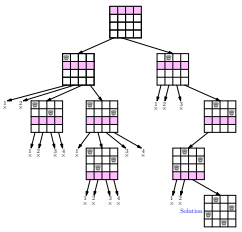
Lesson 22: *Know how to Search*



Backtracking, Branch and Bound

Outline

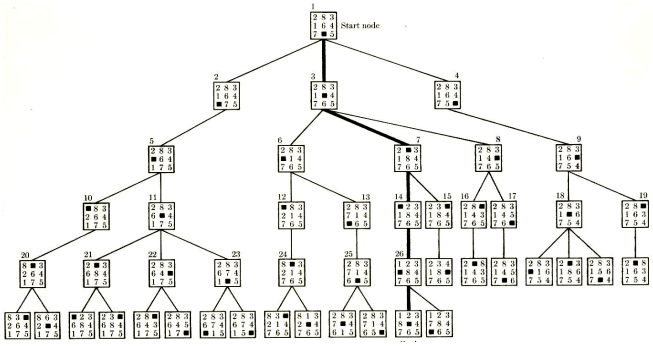
1. **Search Trees**
2. Backtracking
3. Branch and Bound
4. Search in AI



State Space Representation

- Many real world problems involve taking a series of actions to manipulate the state of the system
- This is the area of planning and search which sits within the domain of artificial intelligence
- One of the key props to help us develop algorithms is to think of the states as nodes of a graph which are linked if there exists an action taking us from one state to another
- This provides a **state space representation** of the problem (we saw this before when we derived a low bound on sorting)

8-Puzzle Example

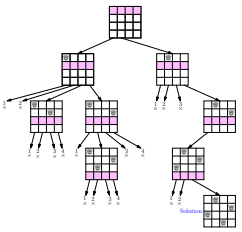


Large State Spaces

- The search space typically increase exponentially with the problem size
- We can find the quickest solution to the 8-puzzle (and the 15 puzzle) using breadth first search, but larger puzzles soon become intractable
- Nevertheless, a lot of important problems involve very large state spaces and we have to find algorithms to explore them

Outline

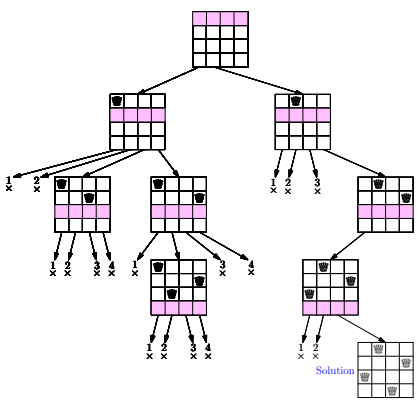
1. Search Trees
2. **Backtracking**
3. Branch and Bound
4. Search in AI



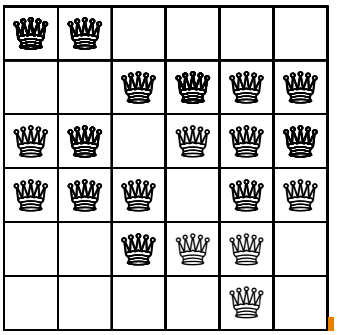
Backtracking

- Backtracking is used to find feasible solutions in large state spaces
- E.g. solving sudoku
- It works by growing partial solutions until either
  - a feasible solution is found when we can finish
  - no feasible solution is found when we backtrack
- We often search the state space using depth first search

4-Queens Problem



6-Queens Problem



Implementing n-Queens

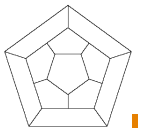
- Implementing backtracking is easily done using recursion
- Recall depth-first search is easily implemented using recursion
- We just need a recursive function `next(n, row, sol)` which for a n-Queens problem searches new solutions in `row` given queens in previous rows given in `sol`
- Run: `List sol = nextRow(6, 0, new List());`

Code

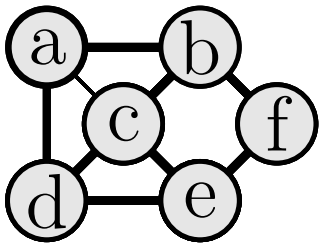
```
List nextRow(int noRows, int row, List queenPositions) {
  if (row==noRows) {return queenPositions;}
  for (int col=0; col<noRows; ++col) {
    if (legalQueen(col, row, queenPositions)) {
      queenPositions.add(col);
      List solution = nextRow(noRows, row+1, queenPositions);
      if (solution!=null)
        return solution;
    }
  }
  return null;
}

bool LegalQueen(int col, int row, List sol) {
  for(int r=0; r<row: ++r) {
    rf (sol[r] == col || sol[r]-row+r == col || sol[r]+row-r==col) {
      return false;
    }
  }
  return true;
}
```


Hamiltonian Circuit

- A Hamiltonian cycle is a tour through a graph which visits every vertex once only and returns to the start
- It is a hard problem in that there are no known algorithms that are guaranteed to find a Hamiltonian cycle in polynomial time
- For many graphs it is not too hard



Hamiltonian Circuit Example

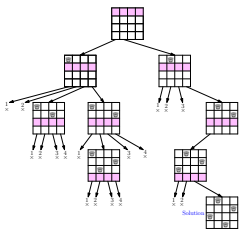


Backtracking

- Backtracking is a standard algorithm for solving constraint problems with large search spaces
- It can take exponential amount of time, however with many constraints it will often find solutions relatively quickly
- A backtracking algorithm does not solve, for example, sudoku in the same way as a human—it uses speed rather than brains
- We can often speed up backtracking by adding more constraints (although, this can make writing the program longer)

Outline

1. Search Trees
2. Backtracking
3. **Branch and Bound**
4. Search in AI



Optimisation Problems

- In many optimisation problems (TSP, Graph-colouring, etc.) we again have a huge search space ($n!$, k^n)
- However, we don't have hard constraints
- If we are interested in finding the optimal then we can use the cost as a constraint

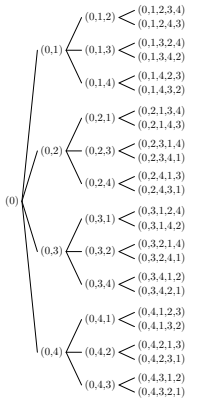
  any partial solution has to have a lower cost than the best solution we have found so far
- This allows us to develop a backtracking strategy known as branch and bound

Branch and Bound

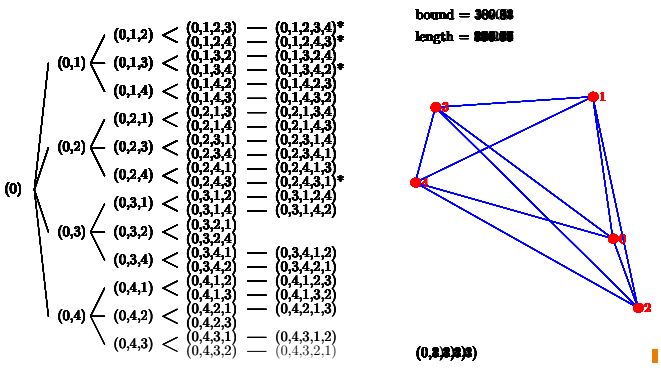
- Branch and bound is used on optimisation problems where efficient strategies just don't work
- It beats exhaustive enumeration by eliminate many possible solutions without having to enumerate them all
- Branch and bound can be slow as the constraints aren't necessarily very strong
- By working harder we can sometimes strengthen the constraints thus eliminating much of the search space
- This strategy works quite well on smallish problems, but usually fails on large problems

Cutting the Search Tree

- We can think of exact enumeration as exploring a giant search tree
- If we know a partial solution is worse than our bound we cut the search tree
- The earlier we cut the tree the more we can save

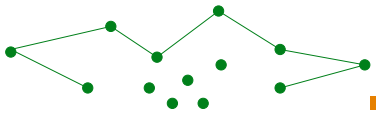


Branch and Bound in Action



Bound on Partial Solution

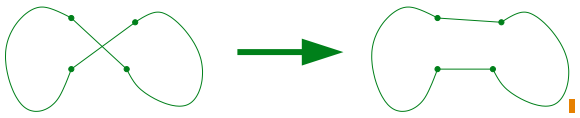
- We know that the partial solution has to include all the remaining cities



- We can use this to obtain a lower bound on the partial solution
- We know the remaining tour will go through each of the unvisited cities and the two edge cities
- In fact the remaining part of the tour is a spanning tree of these vertices (it connects all the vertices once and has no cycles)
- But we know a lower bound for this—**the minimum spanning tree**

Other Cuts

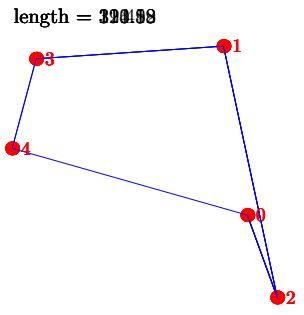
- For 2-D Euclidean TSPs edges should never cross



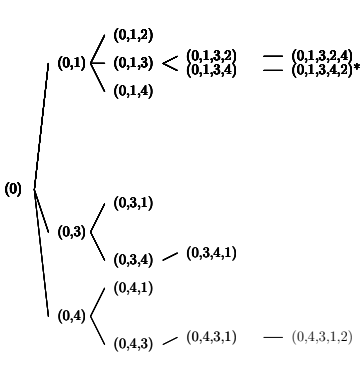
- In fact we can check that we cannot perform a 2-opt move
- We can also halve the search by considering only one direction—for example, by insisting we visit city 1 before city 2

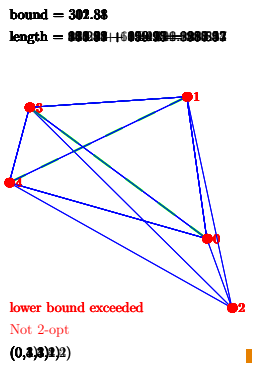
Good Starting Bound

- It helps to start with a good bound
- We can use an *incomplete heuristic algorithm* to find a good solution which will act as a starting bound
- One very simple heuristic is a greedy algorithm



Branch and Bound after Pruning



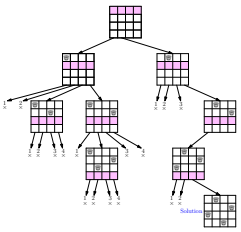


Applications of Branch and Bound

- Branch and bound works for many optimisation problems
- It's drawback is that you often end up still searching an exponentially large search space even though it might be massively faster than exhaustive enumeration
- To make it work well requires considerable work
- This is not an instantaneous algorithm, you may be waiting hours before you find a solution
- For really large problems branch and bound might be too slow

Outline

1. Search Trees
2. Backtracking
3. Branch and Bound
4. **Search in AI**



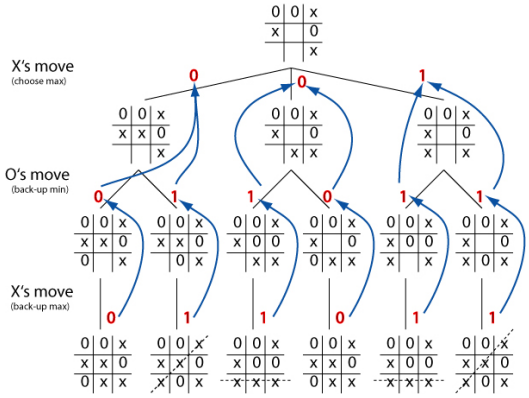
Other Search Strategies

- Search is a big topic in AI
- The algorithms used depends on the information available
- A classic search scenario is when there is "heuristic" information which provides a hint as to where an optimal solution lies
- Algorithms such as A^* exist which will finds the best route given an (admissible) heuristic as efficiently as possible
- You should learn about this next year in AI

Planning and Game Paying

- Search is also used to find the best action to take in planning problems and game playing (e.g. computer chess)
- Again it is useful to think in terms of a search tree
- Searching all paths on the search tree is usually infeasible
- Look for ways of pruning the search tree to focus on good moves
- Strategies include *minimax* and *alpha-beta pruning*

Minimax with Alpha-Beta Prunning

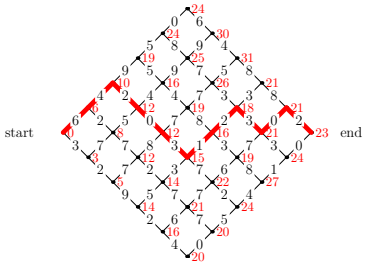


Lessons

- Search has many applications
- It is helpful to consider the search space as a tree whose branch corresponds to possible actions
- Backtracking is useful in search trees with constraints
- For optimisation problems branch and bound uses backtracking and costs of partial solutions as constraints
- Widely applicable, but can take too long

Algorithms and Analysis

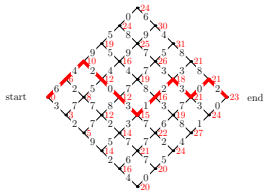
Lesson 23: *Dynamic Programming*



Dynamic programming, line breaking, edit distance, Dijkstra, TSP

Outline

1. Dynamic Programming
2. Applications
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. Limitation

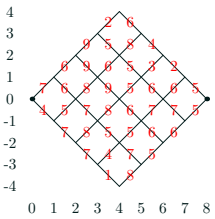


Dynamic Programming

- One of the most powerful strategies for solving optimisation problems is **dynamic programming**
  - ⋆ Build a set of optimal partial solutions
  - ⋆ Increase the size of the partial solutions until you have a full solution
  - ⋆ Each step uses the set of optimal partial solutions found in the previous step
- Developed by Richard Bellman in the early 1950's
- The name is unfortunate as it doesn't have much to do with programming

A Toy Problem

- Consider the problem of find a minimum cost path from point (0,0) to (8,0) on the lattice



- The costs of traversing each link is shown in red
- The cost of a path is the sum of weights on each link

Brute Force

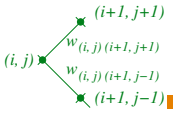
- The obvious brute force strategy is to try every path
- For a problem with n steps we require $n/2$ to be diagonally up and $n/2$ to be diagonally down
- The total number of paths is

$$\binom{n}{n/2} \approx \sqrt{\frac{2}{\pi\, n}}\, 2^n$$

- For the problem shown above with $n = 8$ there are 70 paths
- For a problem with $n = 100$ there are 1.01×10^{29} paths

Building a Solution

- We can solve this problem efficiently using dynamic programming by considering optimal paths of shorter length
- Let $c_{(i,j)}$ denote the cost of the optimal path to node (i, j)
- We denote the weights between two points on the lattice by $w_{(i,j)(i+1,j\pm1)}$



- Clearly $c_{(0,0)} = 0$

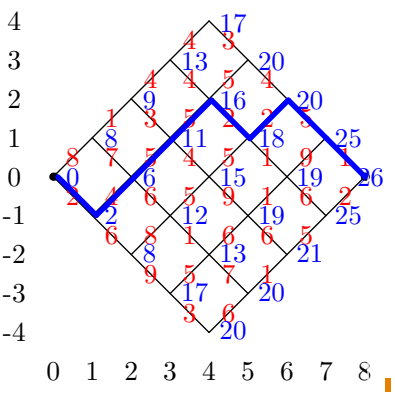
Forward Algorithm

- Suppose we know the optimal costs for all the edge in column i
- Our task is to find the optimal cost at column $i + 1$
- If we consider the sites in the lattice then the optimal cost will be

$$c_{(i+1,j)} = \min\big(c_{(i,j+1)} + w_{(i,j+1)(i+1,j)},\ c_{(i,j-1)} + w_{(i,j-1)(i+1,j)}\big)$$

- This is the defining equation in dynamic programming
- We have to treat the boundary sites specially, but this is just book-keeping

Example



Backward Algorithm

- Having found the optimal costs $c_{(i,j)}$ we can find the optimal path starting from $(n, 0)$
- At each step we have a choice of going up or down
- We choose the direction which satisfies the constraint

$$c_{(i,j)} = c_{(i-1,j\pm1)} + w_{(i-1,j\pm1)(i,j)}$$

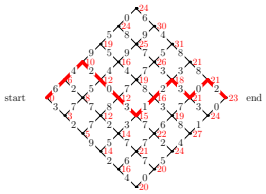
- If both directions satisfy the constraint we have more than one optimal path

Time Complexity

- In our dynamic programming solution we had to compute the cost $c_{(i,j)}$ at each lattice point
- There were $(\frac{n+1}{2})^2$ lattice point
- It took constant time to compute each cost so the total time to perform the forward algorithm was $\Theta(n^2)$
- The time complexity of the backward algorithm was $\Theta(n)$
- This compares with $\exp(\Theta(n))$ for the brute force algorithm

Outline

1. Dynamic Programming
2. **Applications**
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. Limitation



Applications of Dynamic Programming

- Dynamic programming is used in a vast number of applications
  - ⋆ String matching algorithms
  - ⋆ Shape matching in images
  - ⋆ Dynamical time-warping in speech
  - ⋆ Hidden Markov Models in machine learning
- Unlike greedy algorithms the idea is readily extended to many different applications

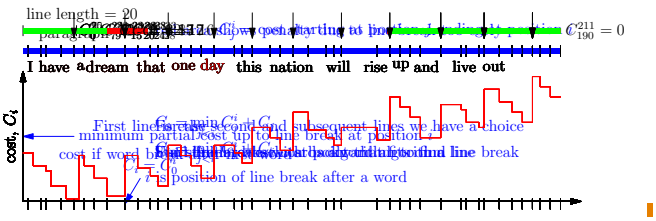
Using Dynamic Programming

- The challenge is recognising that you can use dynamic programming and representing the problem right
- Learn this from examples
- Consider writing a word processor that splits paragraphs up into lines
- You want to choose the line breaks so that the lines are all roughly the same length
- This is a global optimisation task

  minimise the total number of spaces left at the end of each line

Using Dynamic Programming

- I have a dream that one day this nation will rise up and live out the. . .



Real Word Breaking

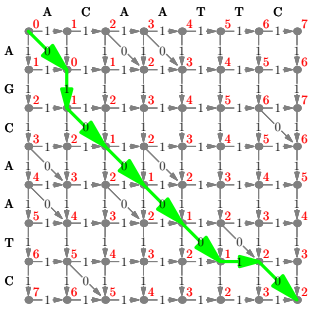
- In advanced word processing you care about hyphenation, large gaps at the end of lines, etc.
- These all affect the way you would assign costs
- Dynamic programming is used in LaTeX to produce nice line breaks
- A similar algorithm is used to produce nice page breaks

Inexact Matching

- A second example of dynamic programming is to find inexact matches
- The edit distance between two strings is the number of changes needed to move from one string to another
- The exact metric depends on the application, but might include number of substitutions, insertions and deletions
- This has many applications, e.g. in genomics to see what DNA strings (or proteins) are related

Edit Distance

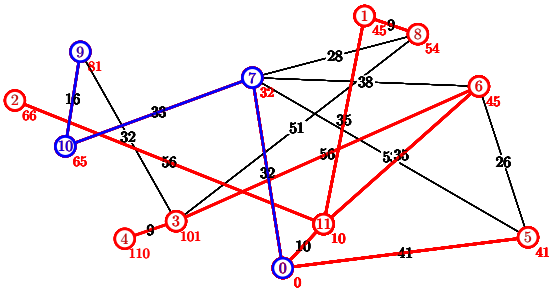
- What is the minimum edit distance between ACAATTC and AGCAATC



Dijkstra's Algorithm

- We saw early Dijkstra's algorithm for find the minimum distance between a source and destination node
- We grouped this with the greedy algorithms as we choose the next node to add to the minimum-distance spanning tree to be the closest node to the source we could access
- However, we should perhaps more rightly identify it as using a dynamic programming strategy as we are building up the cost of getting of the partial solution to reach a node
- We use the greedy strategy to ensure that we always find the shorter paths first

Going from Node 0 to Node 9



Uses of Dynamic Programming

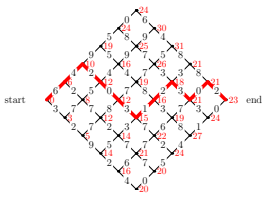
- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text

- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted

- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Outline

1. Dynamic Programming
2. Applications
   - Line Breaks
   - Edit Distance
   - Dijkstra's Algorithm
3. **Limitation**

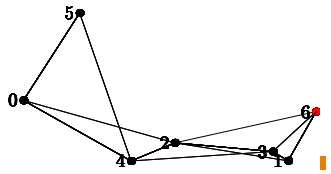


When You Can't Use It

- Not all problems can be split neatly to make dynamic programming possible
- Dynamic programming works on problems with some natural ordering
- We need this to build up a list of optimum cost of partial solutions—these have to depend on the cost of previous partial solutions
- Sometime no natural ordering exists

Travelling Salesman Problem

- For TSP we can find a dynamic programming solution by considering optimal sub-tours (paths) involving sets of k cities
- If we know the optimal sub-tour through all sets of cities of size k (starting and finishing at each possible pair in the set) then we can quickly compute the optimal sub-tour between set of size $k+1$



Travelling Salesman Problem

- The problem is there are $\binom{n}{k}$ subsets consisting of k cities out of a possible n
- The total number of subsets that need to be considered is

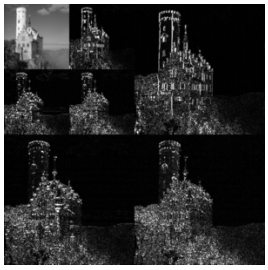
$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

- The time complexity of the DP solution is $n^2\,2^n$ which is better than $n!$ and is currently the fastest known exact algorithm for TSP, but it ain't very useful in practice!

Conclusions

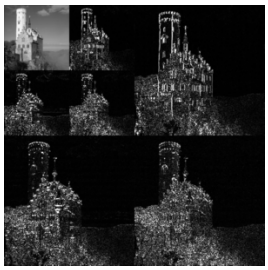
- Dynamic programming is one of the most powerful strategies for solving hard optimisation problems
- It works by iteratively building up costs for partial solutions using the costs of smaller partial solutions
- When it works it is great and there are hosts of practical algorithms which use DP
- However, it doesn't always work

Lesson 24: *Use Smart Encoding!*



File compression, Huffman codes, wavelets

1. **Huffman codes**
2. Wavelets



File Compression

- File compression comes in two varieties
  - Exact compression (e.g. zip used on text files)
  - Lossy compression (e.g. jpeg used on pictures—jpeg can also be loss-less or exact)
- Good exact compression (also known as entropy encodings) can give a compression ratio around 25%
- Lossy compression can give a compression ratio from 10-1%
- Important for saving space, but lossy compression can also be used for noise reduction
- Even used for plagiarism detection!

Entropy Encoding

- Exact encodings use the principle of using short words for frequently occurring sequences (symbols) and longer words for sequences that occur less often
- Claude Shannon showed that for an alphabet of n symbols where the probability of symbol i occurring is p_i no code exists which can transmit information in less than

$$-\sum_{i=1}^{n} p_i \log_2(p_i) \text{ bits}$$

  asymptotically this compression can be achieved
- Different encoding schemes differ in the way they identify symbols of the alphabet—this is rather specialist and we won't go into this

Huffman Coding

- Given a sequence of symbols and their probabilities of occurance, Huffman code provides a way of coding up the information
- It is an example of a **greedy** strategy that happens to be optimal
- Like many greedy strategies it is easily implemented using a priority queue
- It is used in the UNIX compress program and in the exact part of JPEG
- The idea is to assign short codes to commonly used symbols

Symbol Frequency

- We start from an alphabet describing the original document
  - This might be the set of characters
  - For an image it might be the set of pixel values
  - It might be pairs of pixel values
- We compute the number of occurrences of each symbol

Symbol	# Occurrences
a	145
b	67
⋮	⋮

Encoding

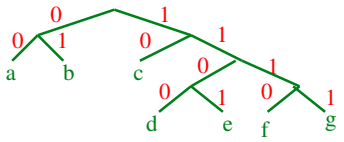
- We want to assign a code to each symbol
- To save space we want to assign short codes to frequently used symbols
- There is a problem: decoding
- If we assigned a code

$$e \to 0 \qquad a \to 1 \qquad r \to 01$$
$$o \to 10 \qquad i \to 11 \qquad t \to 000$$

  etc. we could compress a document very efficiently but we could never decode it uniquely

Huffman Trees

- Once again tree come to the rescue!
- We assign each symbol to a leaf of a binary tree
- We use the position of the branch as an encoding of the symbol



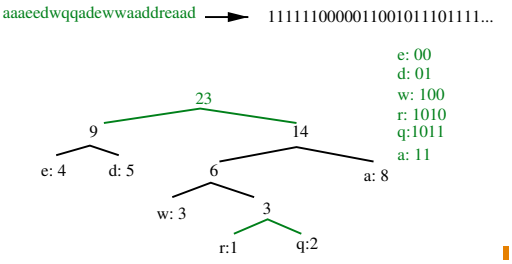
a ⟶ 00
b ⟶ 01
c ⟶ 10
d ⟶ 1100
e ⟶ 1101
f ⟶ 1110
g ⟶ 1111

```
01001111111111011100
LR LLRRRR RRRR RRLR RRLL
 b  a    g    g    e   d
```

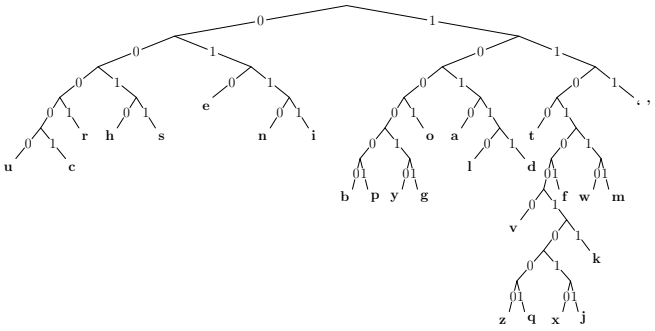
- The decoding is unique

Generating the Huffman Tree

- We are left with the problem of constructing the Huffman tree such that frequently occurring letters have short codes
- A greedy approach is to iteratively build a tree by
  1. combine the two most infrequent symbols into a subtree
  2. Add their scores and treat them as a single symbol



English Letters



the quick brown fox jumps over the lazy dog 344 bits

1100001001011111010010010000001110000111010011111100000000110011101100111111101

0110011101001010111110100101100000110111100001001111110011101000010000111111000011001011110110101011010010001000101111011110011001100011 211 bits

Implementing Huffman Encoding

- To implement Huffman encoding you need
  1. A class to build Huffman trees by combining subtrees
  2. A way to find the least frequently used symbols or symbol combinations
- Priority queues are ideal for this application
- They allow you to find the least frequently used symbols (`removeMin`) and to add new symbols (`add`)
- To decode you follow the Huffman tree

Greedy Strategy

- Huffman encoding is an example of a **Greedy solution pattern**
- That is we look for local optimality (i.e. we combine the two least frequently used symbols)
- In this case, we obtain global optimality (i.e. the Huffman tree obtained gives an optimal Huffman code)
- There are a number of important problems where greedy algorithms lead to global optimality (we saw this earlier)
- For these algorithms priority queues commonly are used for implementing the algorithm

Advanced Techniques

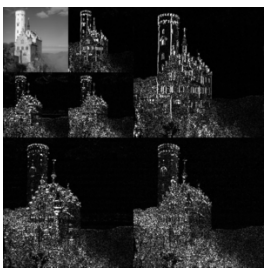
- Huffman code is optimal given the frequency of symbols
- However, there is considerable art in identifying which 'symbols' to use
- Advanced compression algorithms (LZ78, LZW Lempel-Ziv-Welch) build dictionaries of sequences seen in the files—they tend to be rather specialised
- Some recent algorithms (e.g. Burrows-Wheeler) transform the file in such a way that similar symbols are mapped to adjacent sites—depends on the generating mechanism of the language

File Compression and Plagiarism Detection

- One way of spotting plagiarism is to compare the compressed lengths of two files and the length of the compressed file when the two files are concatenated first
- If the files have the same structure the concatenated version can often be significantly reduced
- Also used in identifying closeness of species in constructing phylogenetic trees

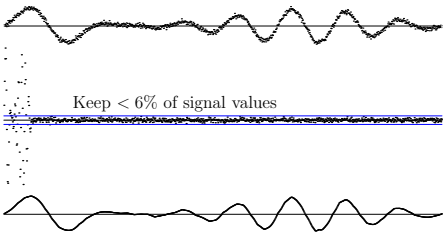
Outline

1. Huffman codes
2. **Wavelets**



Signals and Energies

- We consider compressing a signal $\boldsymbol{x} = (x_0, x_1, \ldots, x_{n-1})$



- We can define the "energy" as the squared deviations

$$E = \sum_{i=1}^{n} x_i^2$$

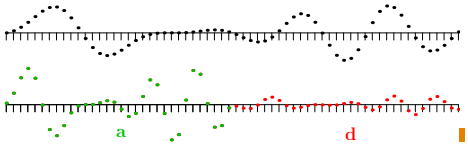
- Our strategy in lossy compression is to transmit as much "energy" in as few bits as possible
- There are different strategies to achieve good compress

Wavelets

- With wavelets we try to re-represent the signal so as to squeeze as much energy as possible into fewer bits
- The easiest way to do this is with Haar wavelets

$$a_i = \frac{x_{2i} + x_{2i+1}}{\sqrt{2}} \qquad d_i = \frac{x_{2i} - x_{2i+1}}{\sqrt{2}}$$

- Define new signal $(a_0, a_1, a_2, \ldots, a_{n/2-1}, d_0, d_1, \ldots, d_{n/2-1})$



Carrier and Difference Signals

- The terms $a_i = (x_{2i} + x_{2i+1})/\sqrt{2}$ takes the "average" of the signal, but compresses it in half the space
- The terms $d_i = (x_{2i} - x_{2i+1})/\sqrt{2}$ takes the difference and is small if the signal does not change much
- The energy is conserved since

$$\begin{aligned} a_i^2 + d_i^2 &= \left(\frac{x_{2i} + x_{2i+1}}{\sqrt{2}}\right)^2 + \left(\frac{x_{2i} - x_{2i+1}}{\sqrt{2}}\right)^2 \\ &= \frac{x_{2i}^2 + 2x_{2i}x_{2i+1} + x_{2i+1}^2 + x_{2i}^2 - 2x_{2i}x_{2i+1} + x_{2i+1}^2}{2} = x_{2i}^2 + x_{2i+1}^2 \end{aligned}$$

- Attempt to push all the energy into the carrier signal, a_i

Inverse Transform

- The wavelet transform can be easily reversed

$$a_i = \frac{x_{2i} + x_{2i+1}}{\sqrt{2}} \qquad d_i = \frac{x_{2i} - x_{2i+1}}{\sqrt{2}}$$
$$x_{2i} = \frac{a_i + d_i}{\sqrt{2}} \qquad x_{2i+1} = \frac{a_i - d_i}{\sqrt{2}}$$

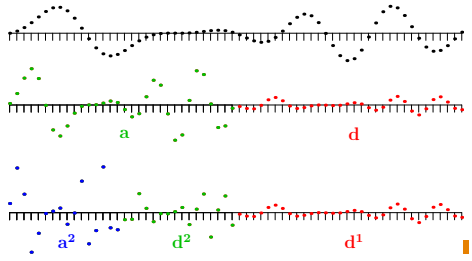
- Can compute transform using vectors (wavelets)

$$a_i = \boldsymbol{V_i} \cdot \boldsymbol{x} \qquad d_i = \boldsymbol{W_i} \cdot \boldsymbol{x}$$

- These vectors are orthogonal to each other ($\boldsymbol{V_i} \cdot \boldsymbol{V_j} = 0$, $\boldsymbol{V_i} \cdot \boldsymbol{W_j} = 0$, etc.)

And So On. . .

- We can repeat the process again to concentrate the energy further
- We apply the Haar transform just to the carry part $\boldsymbol{a} = (a_0, a_1, \ldots, a_{n/2-1})$



Daubechies Wavelets

- Ingrid Daubechies suggested a host of wavelets which do better than Haar for smooth signals
- The simplest is Daub4 defined by

$$a_i = c_0 x_{2i} + c_1 x_{2i+1} + c_2 x_{2i+2} + c_3 x_{2i+3}$$
$$d_i = c_3 x_{2i} - c_2 x_{2i+1} + c_1 x_{2i+2} - c_0 x_{2i+3}$$

$$c_0 = \frac{1+\sqrt{3}}{4\sqrt{2}} \qquad c_1 = \frac{3+\sqrt{3}}{4\sqrt{2}} \qquad c_2 = \frac{3-\sqrt{3}}{4\sqrt{2}} \qquad c_3 = \frac{1-\sqrt{3}}{4\sqrt{2}}$$

- Again conserves energy

$$\sum_{i=1}^{n/2} a_i^2 + b_i^2 = \sum_{i=1}^{n} x_i^2$$

Properties of Daub4

- Similar to the Haar transform

$$c_0 + c_1 + c_2 + c_3 = \sqrt{2}, \qquad c_3 - c_2 + c_1 - c_0 = 0$$

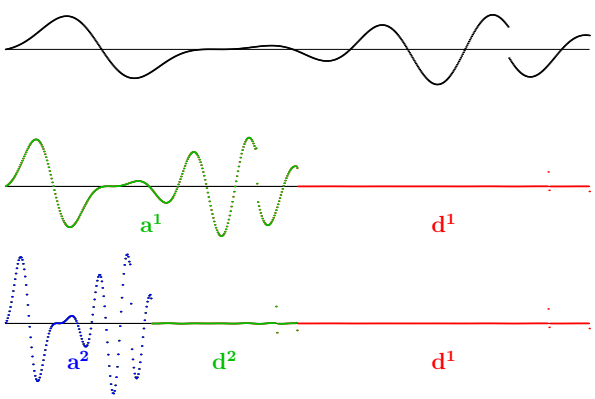
so the carrier signal (a_i) is approximately $\sqrt{2}$ times the original and the difference part (d_i) is equal to 0 for a flat signal, $\boldsymbol{x}$

- However in addition

$$0c_3 - 1c_2 + 2c_1 - 3c_0 = 0$$

so the difference part (d_i) is equal to 0 for any linear signal, $\boldsymbol{x}$

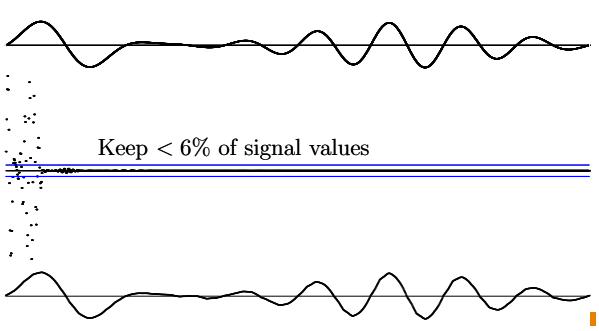
Daub4



Signal Compression

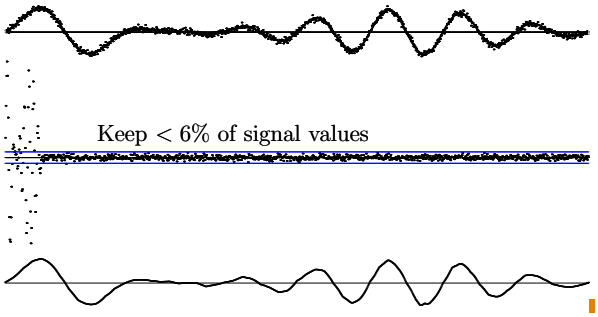
- To compress the signal we can set all components of the transformed signal whose magnitude lies below a threshold to 0
- We transmit the non-zero magnitude together with a binary mask showing the position of the non-zero magnitude
- We can reduce the accuracy (number of decimal places) of the non-zero magnitudes (quantisation)—this is repaired on inverting transform
- We can compress the binary mask using Huffman encoding or other scheme

Daub6



Noise Reduction

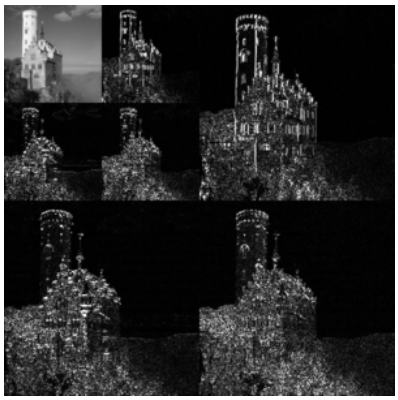
- Can also be used in noise reduction



Other Wavelets

- Can use high-order wavelets which captures more energy in the carrier signal, e.g. Daub10 or Daub20
- Many other wavelets capture other properties (e.g. Coiflets capture properties of a continuous signal sampled at discrete points)
- Efficiency of wavelets depend on how well the capture underlying properties of signals
- Can also construct 2-d wavelets for image compression (jpeg-2000)

2-D Wavelets

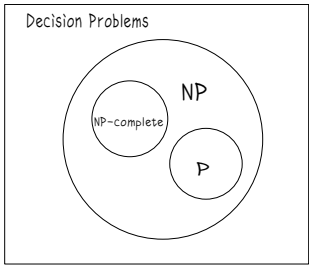


Summary

- File compression is an important task in its own right
- Files may either be compressed losslessly or lossily
- Lossy compression is typically much more efficient (e.g. an order of magnitude smaller)
- Huffman encoding often lies at the lowest level in many compression algorithms
- Wavelets illustrate a strategy of changing the representation to concentrate the energy of a signal

Algorithms and Analysis

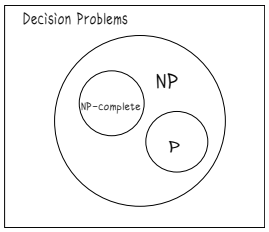
Lesson 24: *Know What's Possible*



Combinatorial optimisation, NP-completeness, polynomial reduction

Outline

1. **Motivation**
2. P, NP and NP-complete
3. Polynomial Reduction



Exponentially Large Search Spaces

- We have seen a large number of decision problems and optimisation problems involving an exponentially large search space
- For some of these we have found efficient algorithms (greedy algorithms, divide and conquer, dynamic programming, . . .)
- For other problems we have found good algorithms (backtracking, branch and bound), but they are not necessarily polynomial
- Can we say anything general about how easy they are to solve

Types of Problems

- We concentrate here on two types of problems
  - ⋆ Decision Problems
  - ⋆ Combinatorial Optimisation Problems
- Decision problems are problems with a true/false answer, e.g. is it possible to cross all the bridges of Königsberg once?
- We showed earlier that backtracking can be used to find a solution which answers the decision problems, e.g. Hamiltonian circuit problem
- There are many other decision problems, but the most famous is satisfiability or SAT

SAT

- Given n Boolean variables $X_i \in \{T, F\}$
- m disjunctive (or's) clauses, e.g.

$$c_1 = X_1 \vee \neg X_2 \vee X_3$$
$$c_2 = \neg X_2 \vee X_3 \vee X_5$$
$$\vdots \qquad \vdots$$
$$c_m = X_2 \vee \neg X_4 \vee \neg X_5$$

- Find an assignment, $\boldsymbol{X} \in \{T, F\}^n$ which satisfies all the clauses
- We can view this as finding an assignment that makes the formula $f(\boldsymbol{X})$ true where

$$f(\boldsymbol{X}) = c_1 \wedge c_2 \wedge \cdots \wedge c_m$$

Decisions and Optimisation Problems

- Often we can cast a decision problem as an optimisation problem
- E.g. the MAX-SAT problem is to find an assignment of variables that satisfies the most clauses
- If we can solve the MAX-SAT optimisation problem we can solve the decision problem
- We can also cast optimisation problems as decision problems

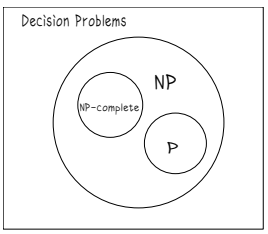
  Does there exist a TSP tour shorter than 200 miles?

Combinatorial Optimisation Problems

- In the set of discrete optimisation problems an important class are those that involve *combinatorial objects* such as permutations, binary string, etc.
- Optimisation problems involving such objects are termed **combinatorial optimisation problems**
- Classical examples of such problems include
  - ⋆ Travelling Salesperson Problem (TSP)
  - ⋆ Graph colouring
  - ⋆ Maximum Satisfiability (MAX-SAT)
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

Outline

1. Motivation
2. **P, NP and NP-complete**
3. Polynomial Reduction



Polynomial Problems

- Some optimisation problems are "easy"—there are known polynomial time algorithms to solve them
  - Minimum spanning tree
  - Shortest path
  - Linear assignment problem
  - Maximum flow between any two vertices of a directed graph
  - Linear programming
- Many apparently different problems can be mapped onto these problems

NP-Hard Problems

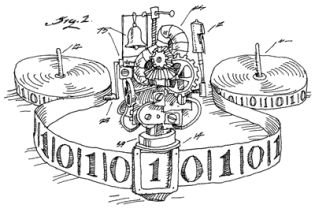
- Is it possible to solve the TSP in polynomial time?
- Answer: **maybe**
- However, no one has discovered such an algorithm and if they do it will have huge implications
- TSP is an example of a class of problems called NP-Hard
- If you can solve one of these problems then you can solve a whole class of problems
- To understand what NP-Hard means we must backtrack

Decision Problems

- Decision problems are problems with a true or false answer
- E.g. does a TSP have a tour less than $2\,000$ miles?
- Algorithmic complexity theory deals with classes of decision problems that have some characteristic size, n
- E.g. n is the number of cities in a TSP
- Any decision problem that can be answered with an algorithm that runs in polynomial time (n^a) on a normal computer (Turing machine) is said to be in class P
- E.g. The decision problem "Is there a path of length less that 450 miles between Southampton and Glasgow?" is in class P

Turing Machines

- A Turing machine can be viewed as a very dumb computer which computes by having a number of states and reading and writing to a tape



- Although dumb, it can do anything that any other computer can do (although it may take polynomially more time)

Non-Deterministic Turing Machines

- A non-deterministic Turing machine is a magic machine that can guess the answer and then use a normal Turing machine to verify the answer
- We assume that it guesses right in the first go
- No one knows how to simulate a non-deterministic Turing machine in polynomial time
- We can simulate it in exponential time by trying all possible guesses

Class NP

- Any decision problem that can be answered with an algorithm that runs in polynomial time on a non-deterministic Turing machine is said to be in class NP
- A whole lot of decision problems belong to class NP
- All decision problems with polynomial algorithms are also in class NP (P⊂NP)
- *"Is the game of chess winnable by white?"* is **not** in class NP

Belonging to NP

- For a problem to belong to class NP it must
  - be a decision problem (true of false)
  - describable by some polynomial sized string in the length of the input n
  - be verifiable if true in polynomial time on a normal Turing machine (e.g. a computer)
- To be verifiable it is sufficient for the decision problem to have a "**witness**" which is usually a solution that can be checked in polynomial time
- E.g. in TSP the witness would be a tour which satisfies the condition

Using Non-Deterministic Turing Machines

- A witness is sufficient for a problem to be in NP since a non-deterministic Turing machine can guess the witness in one step and then proceed to check the witness is true in polynomial time
- Thus TSP is also in NP
- As are graph-colouring, SAT, Max-Clique, Hamilton cycle and countless others

Class NP-complete

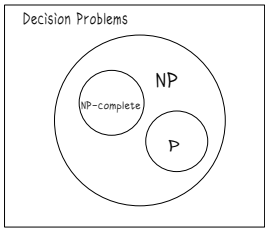
- In 1971 Cook showed that you could represent any NP problem on a non-deterministic Turing machine by a polynomially sized SAT (Satisfaction) problem
- Thus if you could solve SAT in polynomial time you can use that to simulate a non-deterministic Turing machine in polynomial time
- SAT is therefore an example of one of the hardest problems in NP since if you can solve SAT in polynomial time you can solve all problems in NP in polynomial time
- These hardest problems in NP are said to be in NP-complete
- If there existed a polynomial time algorithm for SAT then all problems in NP could be performed in polynomial time so that NP=P

Idea Behind Cook's Theorem

- Cook showed that a non-deterministic Turing machine could be encoded as a big SAT formula
- The evolution of the state and tape was represented by a big tableau ($n^k \times n^k$-table where n^k is the time it takes for the Turing machine to verify the answer)
- The structure of the clauses reflect the rules the Turing machine operates
- If the clauses are simultaneously satisfiable then there exists an input that satisfies the conditions

Outline

1. Motivation
2. P, NP and NP-complete
3. **Polynomial Reduction**



Polynomial Reductions

- Given two decision problems A and B we say there is a **polynomial reduction** from A to B if
  - Every instance of A can be mapped to an instance of B:
  - The truth of the instance A is the same as the corresponding instance B
- We can therefore use B to solve A
- So: $B \in P \rightarrow A \in P$
- The contrapositive of this statement is

  $A \notin P \rightarrow B \notin P$

SAT to 3-SAT

- We can reduce a clause with 4 variable to a clause with 3

  $$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \equiv (X_1 \vee \neg X_3 \vee Z) \wedge (\neg Z \vee X_6 \vee \neg X_{10})$$

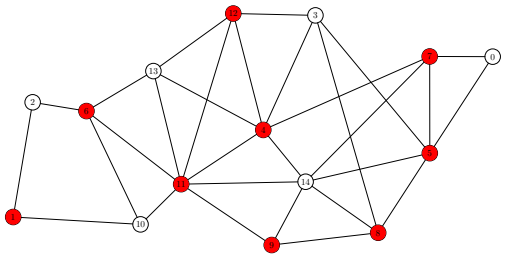
- In doing so we increase the number of variables and the number of clauses to satisfy
- We can similarly reduce a clause with more variables

  $$X_1 \vee \neg X_3 \vee X_6 \vee \neg X_{10} \vee \neg X_{11} \vee X_{15} \equiv$$
  $$(X_1 \vee \neg X_3 \vee Z_1) \wedge (\neg Z_1 \vee X_6 \vee Z_2) \wedge (\neg Z_2 \vee \neg X_{10} \vee Z_3) \wedge (\neg Z_3 \vee \neg X_{11} \vee X_{15})$$

- Because every instance of SAT can be written as a 3-SAT problem which is only polynomially larger than the SAT problem, 3-SAT is also NP-complete

Vertex Cover

- The vertex cover problem is: "can we choose K (9) vertices of a graph such that every edge is connected to a chosen vertex?"

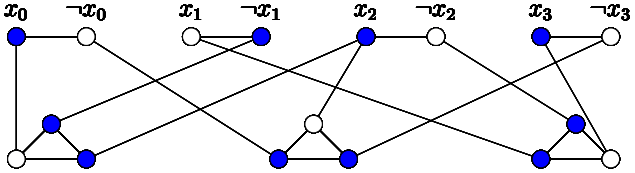


Vertex Cover is NP-complete

- Vertex cover is obviously in NP as a set of K vertices acts as a witness, i.e. it can be checked that it covers all edges
- To show vertex cover is NP-complete we show that every instance of 3-SAT is reducible to vertex cover
- The idea is to show that any 3-SAT problem with variables $\{X_1, X_2, \ldots, X_n\}$ and clauses $\{c_1, c_2, \ldots, c_m\}$ can be encoded as a vertex cover problem

3-SAT to Vertex Cover

$$(x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

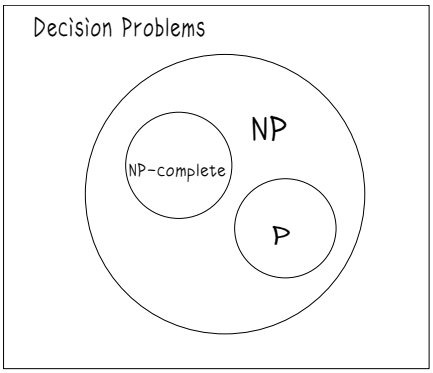


Cover all edges using $n + 2m = 4 + 2 \times 3 = 10$ vertices

Other Examples of NP-complete

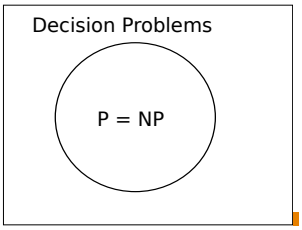
- As we can polynomial reduce any instance of SAT to vertex cover then vertex cover is also NP-complete
- Lots of problems have been shown to be in class NP-complete—some $10\,000$, or so, to date
- These include
  - ⋆ TSP
  - ⋆ Graph colouring
  - ⋆ Many scheduling problems
  - ⋆ Bin-packing
  - ⋆ Quadratic integer problems

Structure of Decision Problems



$P \neq NP$?

- No one has proved that any problem in NP is not solvable in polynomial time
- If any NP-complete problem was solved in polynomial time all problems in NP would be solve and NP=P



NP-Hard

- TSP is not a decision problem—although we can make it into one—Is there a tour shorter that L?
- However, if we can find the shortest tour in polynomial time we could solve the TSP decision problem
- Thus finding the shortest tour is at least as hard as solving the decision problems
- Problems that are at least as hard as NP-complete decision problems are said to be in **NP-hard**
- Graph colouring (finding a colouring with the least number of conflicts), job scheduling, etc. are all examples of NP-hard problems

Not All Hard Problems are NP-Hard

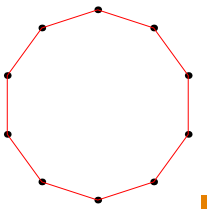
- Graph isomorphism, GI, (are two graphs identical up to a relabelling of the vertices?) has not been proved to be NP-complete—it is postulated that

$$GI \in \text{NP} \wedge GI \notin \text{P} \wedge GI \notin \text{NP-complete}$$

- Factoring is *not* believed to be NP-hard, but it is believed to be sufficiently hard that most banks use an encryption technique based on people not being able to factor large numbers easily
- For large problems polynomial algorithms can take too long

Not All NP-Hard Problem Instances are Hard

- NP-hardness is a worst case analysis
- It means there exist some instance of the problem that we don't know how to solve in polynomial time
- Many instance of the problem might be rather easy to solve
- What is the optimal TSP tour for the problem below?



Not All NP-Hard Problems are Hard

- For some problems almost all instances appear easy
- E.g. The subset-sum problem
  - ⋆ Given a set of numbers find a subset whose sums is as close as possible to some constant
  - ⋆ Subset-sum is in NP-hard but there exist a "pseudo-polynomial time" algorithm which solves almost every instance in polynomial time
- Many problems including subset-sum are known to be easy to approximate
- For other problems even finding a good approximation is known to be NP-hard

Lessons

- There exist efficient algorithms for many problems. . .
- . . . but probably not for all
- There are no known polynomial algorithm for any NP-complete problem
- These include many famous problems: TSP, graph-colouring, scheduling, . . .
- If you could find a polynomial algorithm for any of these problems then you could use it to solve all problems in NP in polynomial time

Algorithms and Analysis

Lesson 25: *Settle For Good Solutions*

neighbourhood search, heuristics, simulated annealing, GA

Outline

1. **Heuristic Search**
   - Constructive algorithms
   - Neighbourhood search
2. Simulated Annealing
3. Evolutionary Algorithms

Heuristic Algorithms

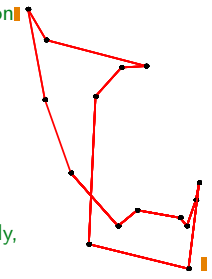
- Given that we know of no efficient algorithms for finding the optimal solution to NP-hard problems we must content ourselves with either
  - ⋆ Spending a very long time (e.g. using branch and bound)
  - ⋆ Accepting good solutions which aren't necessarily optimal
- Algorithms for finding good solutions are often called **approximation algorithms** or **heuristic algorithms**

Heuristics

- The idea behind heuristic algorithms is to use a rough guide or **heuristic** pointing you in reasonable direction
- If this heuristic is good you should find good solutions much faster than exhaustive search
- Two commonly used heuristics are
  - ⋆ A greedy heuristic (take the best move)
  - ⋆ Believe that good solutions are 'close' to each other

Constructive Algorithms

- Constructive algorithms build-up a solution
- They usually rely on a greedy heuristic
- They are very fast
- Once you have got a solution that's it
- They can give reasonable solutions quickly, but they are not usually very good



Neighbourhood search

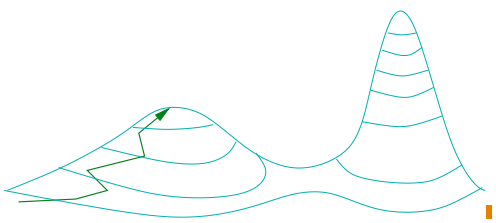
- An alternative to constructive algorithms are search algorithms relying on good solutions being close to each other
- In neighbourhood search we
  1. Start from some solution
  2. Examine the neighbouring solutions
  3. Move to a neighbour if it is better or, at least, not worse
  4. Repeat 2 until some stopping criteria
- If we are maximising this is often called a **hill-climber**
- If we are minimising it is often called **descent**

Iterative Improvement at its Best

- There are times when a neighbourhood algorithm will find the optimal solution
- The classic example of this is in **linear programming** where the simplex method leads to the optimal solution
- Other examples include
  - ⋆ Maximum Flow
  - ⋆ Maximum Matching in Bipartite Graphs
- Unfortunately, this doesn't always work

Local Optima

- Neighbourhood search is usually much slower than a constructive algorithm but tends to find better quality solutions
- However, it will often get stuck



Simple Fixes

- One simple fix is to restart from many different starting positions
- Or perturb the current solution and restart
- These give improvements over doing nothing, but aren't necessarily great strategies
- You can also increase the size of the neighbour to decrease the chance of getting stuck (e.g. in TSP swap more cites)

Outline

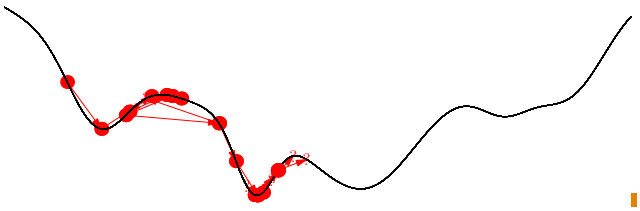
1. Heuristic Search
   - Constructive algorithms
   - Neighbourhood search
2. **Simulated Annealing**
3. Evolutionary Algorithms

Simulated Annealing

- Simulated annealing is an example of a stochastic hill-climber
- Sometimes you go in the wrong direction (down-hill)
- Historically it is an idea from physics—where you tend to minimise energy
- Idea is to obtain a (low energy) crystalline material you very slowly let the material cool from a liquid state (opposite of quenching)

Stochastic Descent

- It is easier to fall down hill than to go back up



Simulated Annealing

- Algorithm to minimise energy $E(\boldsymbol{X})$ where $\boldsymbol{X} = (X_1, X_2, \ldots, X_N)$
- Starting from a random configuration $\boldsymbol{X}$
- Choose a neighbour $\boldsymbol{X}'$
- If the neighbour is better (lower energy) move to it
- Otherwise move to the neighbour with some probability
- The parameter β controls the probability of moving to a neighbour
- We increase β to reduce the probability of going uphill over time

Cooling Schedule

- The parameter β is known as the inverse temperature because of an analogy with physics
- Over time we have to increase β (decrease the temperature) so that the system will remain in the low energy state
- The way you reduce the temperature (increase β) is known as the cooling schedule
- Choosing a good cooling schedule can be critical
- Choosing a good cooling schedule is something of a black art

Convergence Theorem

- There is a theorem that says if you choose a slow enough cooling schedule you will end up in the global minimum eventually
- Unfortunately eventually is a very long time
- It is quicker to search all possible states
- Still people get very excited about convergence proofs

Outline

1. Heuristic Search
   - Constructive algorithms
   - Neighbourhood search
2. Simulated Annealing
3. **Evolutionary Algorithms**

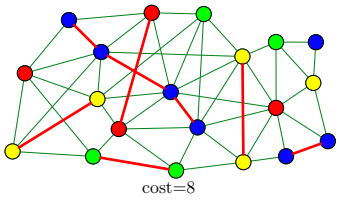
Genetic Algorithms

- Genetic algorithms are methods to evolve a population of potential candidates to find a good solution to an optimisation problem
- There are a whole set of related methods that go by the name of evolutionary algorithms, GAs are a subspecies of EAs
- They can be viewed as an engineering approach to solving hard problems
- I'm going to present my, highly prejudiced view of what's important in making a GA work

A Canonical GA

1. *Initialise population*
2. **for** $t = 1$ **to** T
   (a) *Evaluate fitness*
   (b) *Select* a new population based on fitness
   (c) *Mutate* members of the population
   (d) *Crossover* members of the population
3. Return best member of the population

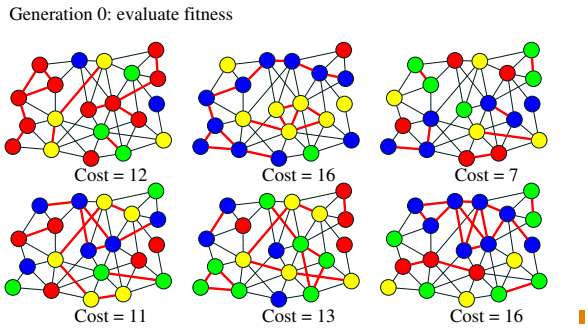
E.g. Graph Colouring



- Given a graph $G = (\mathcal{V}, \mathcal{E})$
- Assign colours, $c(v)$, to the vertices of the graph $v \in \mathcal{V}$
- Minimise the number of edges $e = (v, v') \in \mathcal{E}$ with the same coloured vertices $c(v) = c(v')$ (colour conflicts)

Initialise Population

Generate random colourings. E.g.



Selection

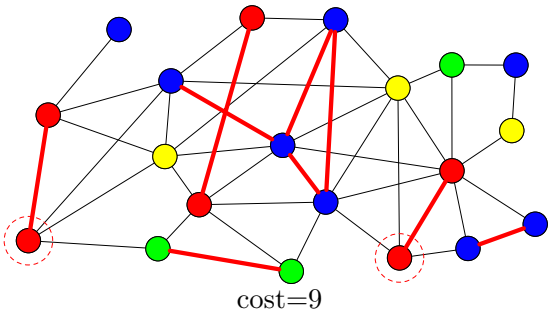
- Select a new population of P members preferentially choosing the fitter members
- Let w_α be a measure of the fitness of member α
- E.g. choose members α with a probability

$$p_\alpha = \frac{w_\alpha}{\sum_{\alpha'=1}^{P} w_{\alpha'}}$$

- Many different ways of doing this (some better than others)

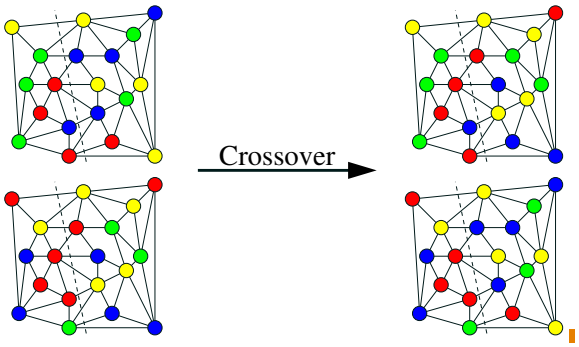
Mutation

Change the colour of one or more of the vertices. E.g.



Crossover

Take two solutions and combine them to form a new solution. E.g.



Crossover Operators

- Single-point crossover
  - Take two strings and cut them at some random site

$$\left.\begin{matrix}(GRBGBR|BGGBGBG)\\(RRBRGB|RGRBBGB)\end{matrix}\right\} \rightarrow \left\{\begin{matrix}(GRBGBR|RGRBBGB)\\(RRBRGB|BGGBGBG)\end{matrix}\right.$$

- Multi-point crossover
  - Take two strings and cut them at several sites

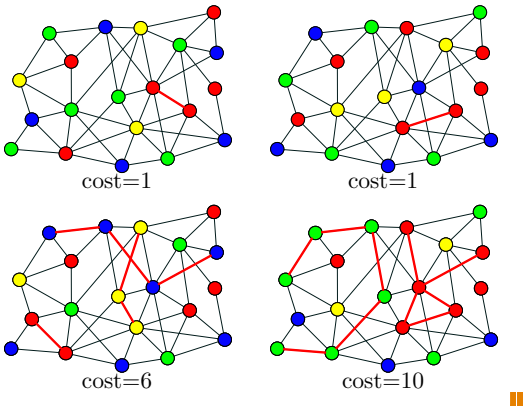
$$\left.\begin{matrix}(GRBGBR|BGGB|GBG)\\(RRBRGB|RGRB|BGB)\end{matrix}\right\} \rightarrow \left\{\begin{matrix}(GRBGBR|RGRB|GBG)\\(RRBRGB|BGGB|BGB)\end{matrix}\right.$$

- Uniform Crossover
  - Take two strings and create children by a random shuffle

$$\left.\begin{matrix}(GRBGBRBGGBGBG)\\(RRBRGBRGRBBGB)\end{matrix}\right\} \rightarrow \left\{\begin{matrix}(GRBRGBRGRGGGB)\\(RRBGBRBGGBBBG)\end{matrix}\right.$$

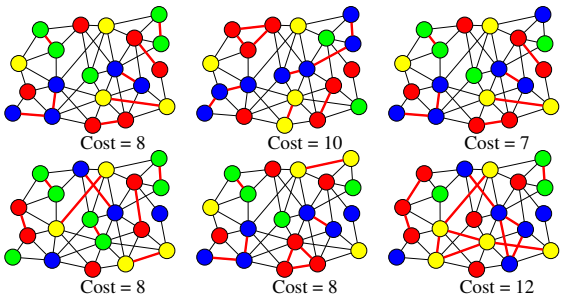
- Any of these crossover can be biased towards one parent

Cost of Crossover



GA

Final Population



Bit Simulated Crossover

- Choose each variable independently with the probability proportional to the frequency of the allele in the population

$$\begin{array}{lll}(\cdots,B,\cdots) & (\cdots,R,\cdots) & (\cdots,G,\cdots)\\ (\cdots,R,\cdots) & (\cdots,B,\cdots) & (\cdots,G,\cdots)\\ (\cdots,B,\cdots) & (\cdots,G,\cdots) & (\cdots,B,\cdots)\\ (\cdots,B,\cdots) & & \end{array}$$

$$p_i(B) = 0.5, \quad p_i(G) = 0.3, \quad p_i(R) = 0.2$$

- New algorithms built on this idea, "Estimation of Distribution Algorithms" EDAs

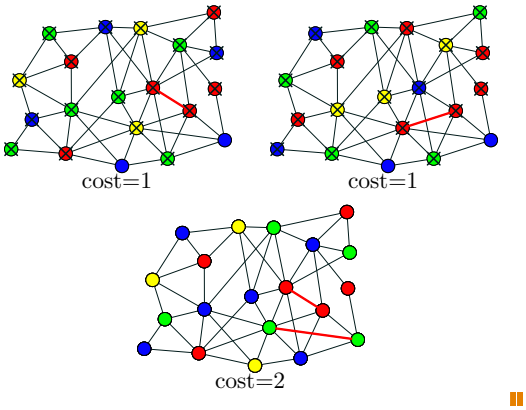
Galinier and Hao's Crossover Operator

- Choose two parents
- Sort nodes into colour-groups

	Parent 1	Parent 2
B	{1,3,4,8,. . . }	{3,5,7,10, . . . }
G	{2,6,7,10,. . . }	{1,11,12,13, . . . }
R	{5,9,11,12,. . . }	{2,3,6,8, . . . }
⋮	⋮	⋮

- Choose largest colour-group in parent 1
- Eliminate all nodes from that colour-group in parent 2
- Choose largest colour-group in parent 2
- etc.

Cost of Crossover



Other Heuristics

- There are many extensions to neighbourhood search, simulated annealing and genetic algorithms
- There are other techniques such as Tabu search
  - ⋆ Construct a list of place you cannot go to (usually the last few configurations)
  - ⋆ Make the best move you are allowed to make
  - ⋆ Rather a large number of $ad\ hoc$ rules to make it work
  - ⋆ Often very fast but runs out of steam
- Many other EAs including particle swarm optimisations (PSO), ant colony optimisation (ACO), evolutionary strategies, . . .

Which Heuristic is Best?

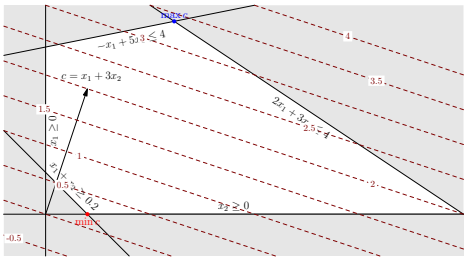
- The best heuristic depends on the application
- Descent is very fast, but only finds local optima—good starting place
- Tabu search is often very fast, but sometimes fails to find really good solutions
- Simulated annealing and Genetic Algorithms are slow, but often find good solutions
- The best algorithms tend to be special purpose algorithms designed for the problem

Lessons

- For many problems the best strategy is to find a good solution, but not the best
- Iterative search usually give good quality solutions
- There are many variants of heuristic search
- Heuristic search algorithms aren't fast (don't use these techniques in an interactive program if you want to keep customers)
- For large combinatorial optimisation problems this is often the only choice

Algorithms and Analysis

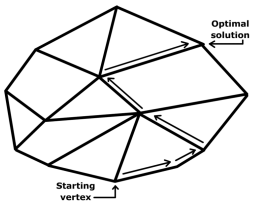
Lesson 27: *Use Linear Programmings*



linear programming, applications

Outline

1. **Examples**
2. Linear Programs
3. Properties of Solution
4. Normal Form



Going Shopping

- Suppose we have a number of food stuffs which we label with indices $f \in \mathcal{F}$
- The price of food stuff f per kilogram we denote p_f
- We are interested in buying a selection of foods $\boldsymbol{x} = (x_f | f \in \mathcal{F})$ where x_f is the quantity (in kg) of food f
- We want to minimise the total price $\sum_f p_f x_f = \boldsymbol{p} \cdot \boldsymbol{x}$
- However we want to ensure that the food has enough vitamins

Nutrition

- We consider the set of vitamins $\mathcal{V}$
- Let A_{vf} be the quantity of vitamin v in food stuff f
- Let b_v be the minimum daily requirement of vitamin v
- We therefore require

$$\forall v \in \mathcal{V} \qquad \sum_{f \in \mathcal{F}} A_{vf}\, x_f \geq b_v$$

Optimisation Problem

- We can write the food shopping problem as

$$\min_{\boldsymbol{x}} \boldsymbol{p} \cdot \boldsymbol{x} \quad \text{subject to} \quad \mathbf{A}\boldsymbol{x} \geq \boldsymbol{b} \quad \text{and} \quad \boldsymbol{x} \geq \mathbf{0}$$

- Note that the inequalities involving vectors means that each component must be satisfied, i.e.

$$\mathbf{A}\boldsymbol{x} \geq \boldsymbol{b} \quad \Rightarrow \quad \forall v \in \mathcal{V} \quad \sum_{f \in \mathcal{F}} A_{vf}\, x_f \geq b_v$$
$$\boldsymbol{x} \geq \mathbf{0} \quad \Rightarrow \quad \forall f \in \mathcal{F} \quad x_f \geq 0$$

- This is an example of a "**linear program**"

Transportation

- We consider a set of factories $\mathcal{F}$ producing a set of commodities $\mathcal{C}$
- The amount of commodity c produced by factory f we denote by x_{cf}
- The shipping cost of commodity c from factory f to the retailer of c we denote by p_{cf}
- We want to choose x_{cf} to minimise the transportation costs

$$\sum_{c \in \mathcal{C}, f \in \mathcal{F}} p_{cf}\, x_{cf}$$

- However, we have constraints. . .

Constraints

- Each factory can only produce a certain overall tonnage of commodities

$$\sum_{c \in \mathcal{C}} x_{cf} \leq b_f \qquad \forall f \in \mathcal{F}$$

where b_f is the maximum production capacity of factory f

- The total demand for each commodity is d_c so

$$\sum_{f \in \mathcal{F}} x_{cf} = d_c \qquad \forall c \in \mathcal{C}$$

- We can only produce positive amounts, i.e. $x_{cf} \geq 0$

Linear Program

- We can write the full problem as

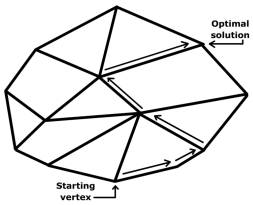
$$\min_{\boldsymbol{x}} \sum_{c \in \mathcal{C}, f \in \mathcal{F}} p_{cf}\, x_{cf}$$

subject to

$$\sum_{c \in \mathcal{C}} x_{cf} \leq b_f \qquad \forall f \in \mathcal{F}$$
$$\sum_{f \in \mathcal{F}} x_{cf} = d_c \qquad \forall c \in \mathcal{C}$$
$$x_{cf} \geq 0 \qquad \forall c \in \mathcal{C}, \quad \forall f \in \mathcal{F}$$

Outline

1. Examples
2. **Linear Programs**
3. Properties of Solution
4. Normal Form



General Linear Programs

- Linear programs are problems that can be formulated as follows

$$\min_{\boldsymbol{x}} \boldsymbol{c} \cdot \boldsymbol{x}$$

subject to

$$\mathbf{A}^{\leq}\boldsymbol{x} \leq \boldsymbol{b}^{\leq}, \quad \mathbf{A}^{\geq}\boldsymbol{x} \geq \boldsymbol{b}^{\geq}, \quad \mathbf{A}^{=}\boldsymbol{x} = \boldsymbol{b}^{=}, \quad \boldsymbol{x} \geq \mathbf{0}$$

- Note in the previous example it was convenient to use two indices c and f to denote the components x_{cf}, however, it still has this structure

Maximising

- We can also maximise rather than minimise
- Whether we want to maximise or minimise will depend on the application
- Note that

$$\max_{\boldsymbol{x}} \boldsymbol{c} \cdot \boldsymbol{x} \quad \equiv \quad \min_{\boldsymbol{x}} (-\boldsymbol{c}) \cdot \boldsymbol{x}$$

- We can thus always reformulate a maximisation problem as a minimisation problem and vice versa

Linear Program Applications

- A huge number of problems can be mapped to linear programming problems
- Or modelled as linear (even when they're not, e.g. oil extraction)
- Realistic problems might have many more constraints and large number of variables
- State of the art solvers can deal with problems with hundreds of thousands or even millions of variables

Key Features

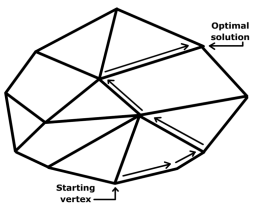
- There are three key features of linear programs
  1. The cost (objective function) is linear in x_i $(\boldsymbol{c} \cdot \boldsymbol{x})$
  2. The constraints are linear in x_i (e.g. $\mathbf{A}_1\boldsymbol{x} \leq b_1$)
  3. The component of $\boldsymbol{x}$ are non-negative (i.e. $x_i \geq 0$)
- These are very special features, very often they don't apply, but a surprising large number of problems can be formulated as linear programming problems

History

- Linear programming was "invented" by Leonid Kantorovich in 1939 to help Soviet Russia maximise its production
- It was kept secret during the war, but was finally made public in 1947 when George Dantzig published the **simplex method** which still today is a standard method for solving linear programs
- John von Neumann developed the idea of duality (you can turn a maximisation problem for a set of variables $\boldsymbol{x}$ into a minimisation problem for a dual set of variables $\boldsymbol{\lambda}$ associated with each constraint)
- von Neumann used this idea as the basis for "game theory"

Outline

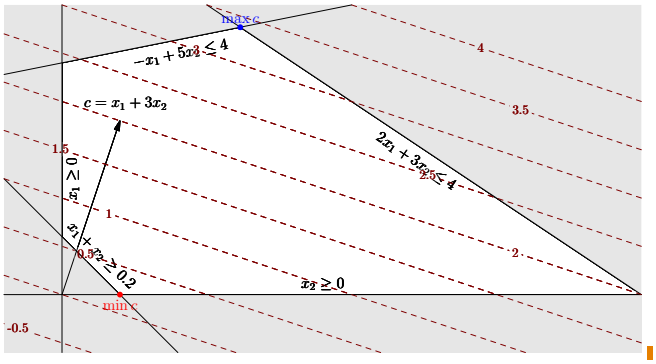
1. Examples
2. Linear Programs
3. **Properties of Solution**
4. Normal Form



Structure of Linear Programs

- Before we go into the details of solving linear programs its useful to consider the structure of the solutions
- The set of $\boldsymbol{x}$ that satisfy all the constraints is known as the set of **feasible solutions**
- The set of feasible solutions may be empty in which case it is impossible to satisfy all the constraints
- This is rather disappointing, but usually doesn't happen if we have formulated a sensible problem

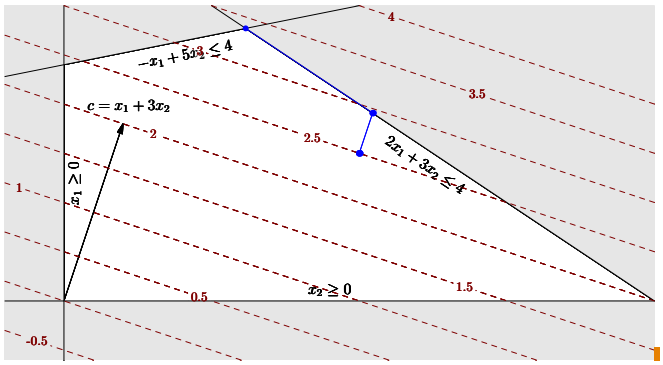
The Space of Feasible Solutions



Vertices of Polytope

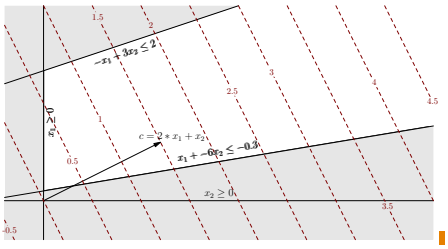
- The space of feasible solutions is a polyhedra or polytope
- The maximum or minimum solution will always lie at a vertex of the polytope
- Our solution policy will be to start at a vertex and move to a neighbouring vertex that gives the best improvement in cost
- When this isn't possible then we are finished
- However, there is still a lot of work to realise this solution strategy

Optimal Solution



Unbounded Solutions

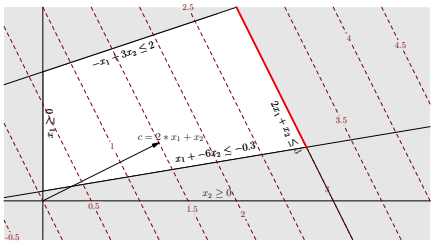
- If you are unlucky you might not have a bounded solution



- But usually this would not happen because of the problem definition

Multiple Solutions

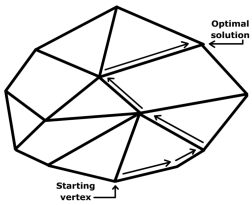
- You can also get multiple solutions if a constraint is orthogonal to the objective function



- Nevertheless the optimal will be at a vertex

Outline

1. Examples
2. Linear Programs
3. Properties of Solution
4. **Normal Form**



Converting Linear Programs

- Solving full linear programs is difficult
- However, it is much easier to solve linear programs in **normal form**
- This is basically a form where we get rid of all inequalities and rewriting the equalities
- Fortunately its rather easy to convert linear programs to normal form

Slack Variables

- We can change an inequality into an equality by introducing a new "**slack**" variable
- E.g.

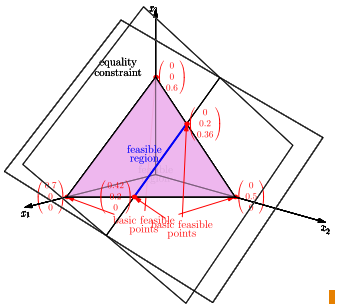
$$\begin{aligned} \boldsymbol{a}_1 \cdot \boldsymbol{x} \geq 0 \quad &\Rightarrow \quad \boldsymbol{a}_1 \cdot \boldsymbol{x} - z_1 = 0 \quad z_1 \geq 0 \\ \boldsymbol{a}_2 \cdot \boldsymbol{x} \leq 0 \quad &\Rightarrow \quad \boldsymbol{a}_2 \cdot \boldsymbol{x} + z_2 = 0 \quad z_2 \geq 0 \end{aligned}$$

  z_1 (the excess) and z_2 (the deficit) are known as slack variables
- We eliminate inequalities at the expense of increasing the number of variables
- We can treat the slack variables on an equivalent footing to the normal variables (they just provide a different way of describing the original problem)

Normal Form

- A linear program with only equality constraints is said to be in **normal form**
- We will find in the next lecture that this is a convenient form for solving linear programs
- An equality constraint restricts the solutions to a subspace (some lower dimensional space)

Solving Linear Programming

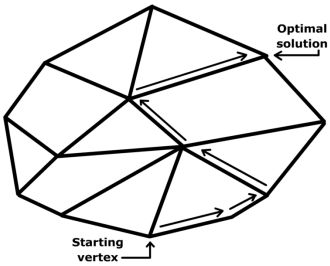


- The basic feasible points for LP problems with n variables and m constraints have at least $n - m$ zero variables
- Typical number of basic feasible solutions is $\binom{n}{m} \geq \left(\frac{n}{m}\right)^m$
- Simplex algorithm organises iterative search for global solutions

Lessons

- There are a huge number of problems that can be set up as linear programs
- They are particularly useful in resource allocation where the resources are all positive
- The solution to linear programming problems is at the vertex of the feasible space (intersection of constraints)
- We can search for solutions by moving from vertex to vertex
- We can transform inequality constraints to equality constraints using slack variables

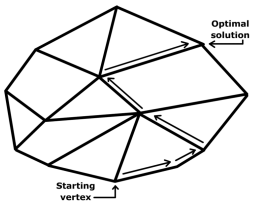
Lesson 28: *Solving Linear Programs*



linear programming, simplex methods, iterative search

Outline

1. Recap
2. Basic Feasible Solutions
3. Simplex Method
4. Classic LP Problems



Recap

- Linear programs are problems that can be formulated as follows

$$\min_{\boldsymbol{x}} \boldsymbol{c} \cdot \boldsymbol{x}$$

subject to

$$\mathbf{A}^{\leq}\boldsymbol{x} \leq \boldsymbol{b}^{\leq}, \quad \mathbf{A}^{\geq}\boldsymbol{x} \geq \boldsymbol{b}^{\geq}, \quad \mathbf{A}^{=}\boldsymbol{x} = \boldsymbol{b}^{=}, \quad \boldsymbol{x} \geq \mathbf{0}$$

- Where $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$
- $\mathbf{A}^*$ are matrices and we interpret the inequalities to mean

$$\forall k \qquad \sum_{j=1}^{n} A^{\leq}_{kj} x_j \leq b^{\leq}_k$$

Optima and Vertices

- Because the objective function is linear $(\boldsymbol{c} \cdot \boldsymbol{x})$ there is a direction where the objective is always improving
- Thus, the optima cannot lie in the interior of the search space
- When we meet a constraint that limits the direction we can move, but we can still move along the constraint
- We then meet another constraint which restricts the direction we can move by two degrees of freedom
- Eventually, we will reach n constraints which defines a vertex of the feasible region and is optimal

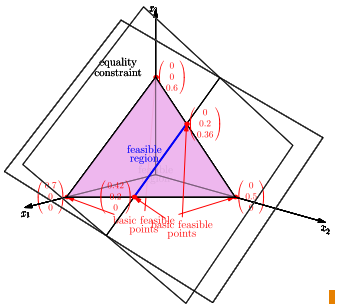
Transforming Linear Programs

- We can always transform an inequality constraint into an equality constraint by adding slack variables
- E.g.

$$\begin{aligned} \boldsymbol{a}_1 \cdot \boldsymbol{x} \geq 0 \quad &\Rightarrow \quad \boldsymbol{a}_1 \cdot \boldsymbol{x} - z_1 = 0 \quad z_1 \geq 0 \\ \boldsymbol{a}_2 \cdot \boldsymbol{x} \leq 0 \quad &\Rightarrow \quad \boldsymbol{a}_2 \cdot \boldsymbol{x} + z_2 = 0 \quad z_2 \geq 0 \end{aligned}$$

  z_1 (the excess) and z_2 (the deficit) are known as slack variables
- A linear program with just equality constraints and non-negativity constraints is said to be in normal form

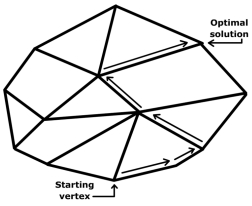
Solving Linear Programming



- The basic feasible points for LP problems with n variables and m constraints have at least $n - m$ zero variables
- Typical number of basic feasible solutions is $\binom{n}{m} \geq \left(\frac{n}{m}\right)^m$
- Simplex algorithm organises iterative search for global solutions

Outline

1. Recap
2. Basic Feasible Solutions
3. Simplex Method
4. Classic LP Problems



Basic Feasible Solution

- A *basic feasible solution* or *basic feasible point* is a solution that lies at a vertex of the feasible space
- To solve a linear program we will start at a basic feasible point and move to the neighbour which best improves the objective function
- When we cannot find a better solution we are at the optimal solution
- This is an example of an iterative improvement algorithm which gives an optimal solution

Constraints

- There are two types of constraints
  1. n non-negativity constraints $x_i \geq 0$
  2. m additional constraints, which we can take to be equalities $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$
- Note that some of the variables might be slack variables
- We consider the case when there are more variables than additional constraints, i.e. $n > m$
- This is usually be the case, but. . .
- If this isn't true it turns out you can consider an equivalent problem (dual problem) where you have a variable for each constraint and a constraint for each variable

Basic Variable

- In total we have n equality and m non-negativity constraints
- n constraints must be satisfied to be at a vertex of feasible region
- So at least $n - m$ of the non-negativity constraints are satisfied (i.e. $x_i = 0$)
- The $n - m$ variables that are zero are said to be **non-basic variables**
- The other m variables are said to be **basic variables**

Initial Basic Feasible Solution

- One of the tricky bits of tackling a linear program is to find an initial feasible solution
- We do this in **phase one** of the simplex program
- To do this for each additional constraint we add a new **auxiliary variable** ξ_k, e.g.

$$\forall\, k \in \{1, 2, \ldots, m\} \qquad \xi_k + \sum_i A_{ki} x_i = b_k \geq 0$$

- We then can find a basic feasible solution by setting $x_i = 0$ so

$$\xi_k = b_k \quad \forall\, k \in \{1, 2, \ldots, m\}$$

Eliminating Auxiliary Variables

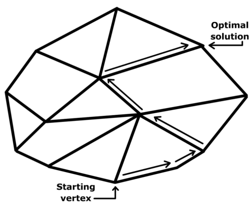
- In phase one we run a simplex algorithm with an auxiliary cost function

$$\min f_{\text{aux}}(\boldsymbol{x}, \boldsymbol{\xi}) = \sum_{k=1}^{m} \xi_k$$

- This should find a solution where all the $\xi_k = 0$
- If no solution exists it means there is no feasible solution and we're finished
- If there is a solution then we can eliminate the auxiliary variables and we have a feasible solution

Outline

1. Recap
2. Basic Feasible Solutions
3. **Simplex Method**
4. Classic LP Problems



Phase Two

- In phase two we now have an initial basic feasible solution (with $n - m$ zero variables)
- We then run the simplex algorithm on the original objective function $f(\boldsymbol{x}) = \boldsymbol{c} \cdot \boldsymbol{x}$
- That is we move to a neighbouring vertex which gives the best increase in the objective function
- To help organise this search we write the objective function and constraints in a **restricted normal form** and then build a **tableau** showing the *basic variables* and the *non-basic variables*

Restricted Normal Form

- To perform the moves between vertices it helps to represent the problem in a **restricted normal form**
- Starting from a basic feasible point we have a constraint for each basic (non-zero) variable
- We write the constraints as an equality between basic and non-basic (zero valued) variables
- Similarly we write the objective function in terms of non-basic variables
- This is always possible as we can use the constraints to eliminate the basic variables

Tableau

[illegible]

Awkward Problems

- If there are any column with all entries positive then this variable can be increase forever—this is a signal that the linear programming problem is unbounded
- You can also find that a basic variable becomes zero—this is known as a degenerate feasible vector
- It can by removed by exchanging variables on the left of the inequality with variables on the right
- This makes the algorithm a bit more complex to implement

High Performance Solvers

- Although the tableau method is the "classic solver" it doesn't cut the mustard for large scale problems
- The simplex update can also be viewed as solving a linear set of equations which is facilitated by performing an LU-decomposition
- However, the constraints are often very sparse so good solvers try to take advantage of the sparsity
- Top end simplex algorithms are rather complex
- There is a second approach known as the interior point method which is competitive on large problems

Time Complexity of Simplex

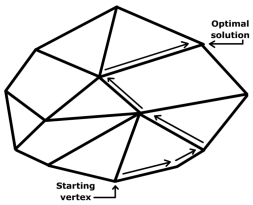
- The time complexity of the updates is $O(n^2)$
- The critical question is how many updates are necessary
- It turns out that typically this is $O(n)$ making the simplex algorithm $O(n^3)$
- However, it is possible to cook up problems where there is a "long path" from the initial solution to the optimum which is exponentially big
- Thus the worst case time is exponential, although this almost never happens in practice

Interior Point Method

- An alternative to the simplex method is the interior point method which always remains in the feasible region, away from the constraints
- These method iterate towards the constraints and are provably polynomial
- For small linear programming problems they are out-performed in practice by the simplex method
- On large and very large problems they seem to perform as well if not better than the simplex method
- The high-end solvers will have a variety of interior point methods tailored to the particular problem

Outline

1. Recap
2. Basic Feasible Solutions
3. Simplex Method
4. **Classic LP Problems**



LP Problems

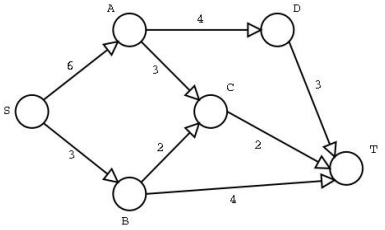
- Any problem that can be set up as a linear program can be solved in polynomial time
- One way is just to feed it to a LP-solver
- Sometimes the problems are important enough and have such a distinctive formulation that faster specialised algorithms have been developed
- We consider a couple of classic problems: *maximum flow* and *linear assignment*

Maximum Flow

- In maximum flow we consider a directed graph representing a network of pipes
- We choose one vertex as the source and a second vertex as a sink
- Each edge has a flow capacity that cannot be exceeded
- The problem is to maximise the flow between source of sink
- This can be used to model the flow of a fluid, parts in an assembly line, current in an electrical circuit or packets through a communication network

Example

- Consider a firm that has to ship haggis from Edinburgh to Southampton
- The shipping firm transports this in crates which it sends through intermediate cities
- The number of crates is limited by the size of the lorries it uses



Flow

- We are given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where each edge has a capacity $c(i, j)$
- We define the flow from i to j as $f(i, j)$ with $0 \leq f(i, j) \leq c(i, j)$
- For all vertices except the source (s) and sink (t) we assume

$$\forall i \in \mathcal{V}/\{s, t\} \qquad \sum_{j \in \mathcal{V} | (i,j) \in \mathcal{E}} f(i, j) = \sum_{j \in \mathcal{V} | (i,j) \in \mathcal{E}} f(j, i)$$

  (i.e. no flow is lost from source to sink)
- We want to maximise the flow from the source

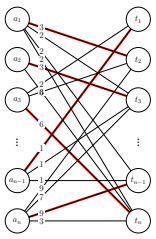
$$\sum_{i \in \mathcal{V} | (s,i) \in \mathcal{E}} f(s, i)$$

Solving Maximum Flow

- As set up we have a linear objective function with linear constraints
- We can therefore solve this problem with a LP-solver
- (Note the solution will typically involve a fraction flow)
- However, this is such a classic problem with a distinctive structure that we can solve it more quickly with other algorithms
- The classic algorithm is the Ford-Fulkerson method with run time $O(|\mathcal{E}| \times f_{\max})$ where $f_{\max}$ is the maximum flow, although we won't cover this in the course

Linear Assignment

- We are given a set of n agents, $\mathcal{A}$, and n tasks, $\mathcal{T}$
- Each agent has a cost associated with performing a task $c(a, t)$
- We want to assign an agent to one task so as to minimise the total cost
- Consider a taxi firm with taxi's at 5 different locations and 5 requests to fulfil. The cost is the distance to the clients. Which taxi should go to which client?



LA as LP

- The linear assignment problem can be set as a linear programming problem

$$\min_{\boldsymbol{x}} \sum_{a \in \mathcal{A}, t \in \mathcal{T}} c(a, t) x_{a,t}$$

subject to

$$\forall a \in \mathcal{A} \qquad \sum_{t \in \mathcal{T}} x_{a,t} = 1$$

$$\forall t \in \mathcal{T} \qquad \sum_{a \in \mathcal{A}} x_{a,t} = 1$$

$$\forall (a, t) \in (\mathcal{A}, \mathcal{T}) \qquad x_{a,t} \geq 0$$

Hungarian Algorithm

- Linear assignment is another classic problem that is commonly encountered
- Although it can be solved using a generic LP-solver this is not the most efficient algorithm
- The most efficient algorithm is the Hungarian algorithms
- This is rather complex (having once implemented it I can tell you from bitter experience it ain't easy)
- Its worst case time is $O(n^3)$ although it frequently takes $\Theta(n^2)$

Quadratic Programming

- If we have linear constraints and a quadratic objective function then we have a quadratic programming problem
- Again this can be solved in polynomial time
- Many of the ideas used are the same as for linear programming
- This also has important applications in science and engineering

Lessons

- Linear programming is a classic problem
- We know a huge number of problems are solvable in polynomial time because they can be formulated as linear programs
- Linear programs occur sufficiently often that they are hugely important
- They aren't easy to solve, although standard simplex is not massively complex
- For particular LP problems with distinctive structure there are sometimes better algorithms than generic LP-solvers

Algorithms and Analysis

Lesson 29: *Know What's Important*

Optimising code, strategies

Outline

1. **Time Complexity**
2. Strategies
   - Brute Force Methods
   - Divide and Conquer
   - Greedy Algorithms
   - Dynamic Programming
   - Linear Programming
   - Backtracking
   - Heuristic Search

Designing Algorithms from Scratch

- In writing code you will come across new problems—at least if your lucky
- Your first task is to see if you can map this onto a problem with a well known solution
- If not you are going to have to come up with a solution
- Two questions you need to ask yourself are
  - How efficient does my solution need to be?
  - How do I go about solving the problem?

Is Solving the Problem Time Critical?

- The majority of code is not time critical
- Even when programs are slow there is usually only one part of the code which takes almost all the time
- However there are times when solving the problem naïvely is going to take too long
- Advice on improving the performance of your code is not hard to come by. . .

Advice on Optimising Code

- "More computing sins are committed in the name of efficiency (without necessarily achieving it) than any other single reason—including blind stupidity" W. Wulf
- "We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." D. Knuth
- "We follow two rules in the matter of optimization:
  - Rule 1. Don't do it
  - Rule 2 (for experts only). Don't do it yet—that is, not until you have a perfectly clear and unoptimized solution"

  M. A. Jackson
- "Strive to write good programs rather than fast ones" J. Block

Time Complexity

- Ignoring this good advice, your next job is to decide what is the time complexity of your algorithm and whether you can tolerate that time complexity?
- Typically your algorithm will be
  - constant or (log)-linear time ☺
  - quadratic/cubic,. . .
  - exponential time ☹
- Lets deal with this in inverse order

Exponential Time

- You can often tolerate this if your problem is small
- E.g. you're trying to solve a small puzzle
- or planning a short sequence of actions
- However, in many cases exponential is just too long
  - Look for an efficient solution (e.g. dynamics programming)
  - Redefine the problem (e.g. use a linear approximation)
  - Settle for a sub-optimal solution (e.g. using heuristic search)

Quadratic and Cubic Time

- There are algorithms which are cubic (e.g. LP, inverting a matrix)
- For many applications this is acceptable because there is no hurry for the solution or the problem isn't that big
- For large data sets these algorithms might just be impractical
- Often taking advantage of the structure of the problem (e.g. sparsity) can speed things up (e.g. a good LP package)
- Sometimes quadratic algorithms can be made log-linear using a divide and conquer or a greedy strategy

Sub-Quadratic Algorithm

- If you have to run an algorithm on any data of arbitrary size you want a sub-quadratic algorithm
- We have seen this in practice with sorting
- The fast Fourier transform revolutionised digital signal processing when it was introduced (reducing a quadratic algorithm to a log-linear algorithm)
- An active area of research is **big data** where the only algorithms that can be used are sub-quadratic algorithms

Good Code is Fast Code

- Using appropriate data structures and algorithms is by far the best way of speeding up code
- Many coders wedded to arrays try to simulate sets and maps very inefficiently often using code that does not scale
- Changing the structure of a program can lead to huge speed ups
- If you require more speed then concentrate on the inner loop where almost all the work is done (usually gives less than a factor of two)—optimising code in outer loops is pointless
- Using the right strategy will give the biggest speed-up

Outline

1. Time Complexity
2. **Strategies**
   - Brute Force Methods
   - Divide and Conquer
   - Greedy Algorithms
   - Dynamic Programming
   - Linear Programming
   - Backtracking
   - Heuristic Search

Algorithmic Strategies

- Good algorithms are difficult to invent
- However, many algorithms follow particular patterns or strategies
- Understanding these strategies is important for deriving new algorithms
- We have seen the classic strategies throughout the course

Brute Force

- Many problems have obvious **brute force** solutions
- E.g. in sorting; selection sort, insertion sort and bubble sort are fairly simple algorithm that do the obvious
- Similarly searching an array using sequential search provides an obvious solution
- Sometimes, brute force methods are the best you can do—e.g. sequential search on an unordered array

Exhaustive Search

- For optimisation problems such as the travelling salesperson problem, the brute force method is to try all possible solutions
- This **exhaustive search** starts to hurt very quickly and becomes intractable for moderate size problems (e.g. tours of length 20)
- Even for sorting, brute force methods become unattractive when the inputs are long
- We really want to do better

Divide and Conquer

- Another strategy we have met is **divide and conquer**
  - ⋆ Divide the problem into two or more parts
  - ⋆ Solve the parts
  - ⋆ Combine the parts to obtain a full solution
- This needs to be quicker than solving the original problem by brute force
- We can do this division recursively until the problems are trivial to solve

Divide and Conquer Problems

- Algorithms based on this idea include
  - ⋆ Computing the integer power of a number
  - ⋆ Binary search
  - ⋆ Merge sort
  - ⋆ Quick sort
  - ⋆ Fast Fourier transform
- Often implemented using recursion
- It's nice when it works—but not all problems allow this

Fast Fourier Transform

- The Fourier Transform provides a different "view" of a sequence such as a signals, images, etc.

$$\tilde{f}(\boldsymbol{k}) = \sum_{x_1=0}^{n-1} \cdots \sum_{x_d=0}^{n-1} f(\boldsymbol{x})\, \mathrm{e}^{2\pi \mathrm{i} \boldsymbol{k}\cdot\boldsymbol{x}/n}$$

- The **Fast Fourier Transform** was an algorithm divised by John Tukey and James Cooley in 1965 to compute the Fourier Transform quickly
- Gauss had used exactly this idea in a paper in 1805 to save himself work computing a Fourier transform by hand
- It is based on a divide-and-conquer strategy and takes $O(n\log(n))$ operation compared to the $O(n^2)$ brute force method

Applications of FFT

- The application of FFT are enormous
- It lies at the heart of digital signal processing
- It is frequently used in image analysis
- It is a type of wavelet transform used in JPEG
- It is even used in fast multiplication of very large integers—with important applications in cryptography

Greedy Algorithms

- The **greedy strategy** is to build a solution to a problem by choosing the best available option
- If you are lucky this will give an optimal solution
- Example of optimal algorithms based on the greedy strategy include
  - Constructing Huffman trees
  - Prim's algorithms
  - Kruskal's algorithm
  - Dijkstra's algorithm
- Often uses priority queues

Non-optimal Greedy Algorithms

- Greedy algorithms can also be used to solve optimisation problems such as the travelling salesperson problem
- In the TSP we can start at some city and move to the nearest as-yet-unvisited city
- The algorithm is guaranteed to find a solution no longer than $0.5(\lfloor \log_2(n) \rfloor + 1)$ times the optimal tour length
- It usually does substantially better, but it is very unlikely to find the optimal for very long tours
- It is more the exception rather than the rule that Greedy algorithms find an optimal solutions

Dynamic Programming

- Dynamic programming can be used for solving many problems
- It requires some (partial) ordering so that you can assign costs to partial solutions from previous solutions
- Requires imagination to think how to do this
- Used in inexact matching, shortest paths, line breaks, etc.

Uses of Dynamic Programming

- Recurrent solutions to lattice models for protein-DNA binding
- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance)
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke–Younger–Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- Knuth's word wrapping algorithm that minimizes raggedness when word wrapping text

- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman–Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction
- Floyd's all-pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudo-polynomial time algorithms for the subset sum and knapsack and partition problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth–Lewis method for resolving the problem when games of cricket are interrupted

- The value iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in music information retrieval
- Adaptive-critic training strategy for artificial neural networks
- Stereo algorithms for solving the correspondence problem used in stereo vision
- Seam carving (content aware image resizing)
- The Bellman–Ford algorithm for finding the shortest distance in a graph
- Some approximate solution methods for the linear search problem
- Kadane's algorithm for the maximum subarray problem

Linear Programming

- Look out for problems with linear objectives or where a linearisation approximation is acceptable
- These can often be turned in linear programs which can be solved efficiently
- The constraints have to be linear and the variables take on continuous values
- Sometimes by being careful we can force integer solutions (see linear assignment in last lecture)
- Applications in planning, but also in many areas of optimisation

Backtracking

- Backtracking is in many ways a brute force method
- It is just a way of exploring a search space
- However, when we solving problems with constraints then we can vastly reduce the number of solutions that we visit
- Used in many game playing, planning, verification, puzzle solving situations
- For optimisation we can use branch and bound which is backtracking using the best solution as a constraint

Computer Chess

- Computer chess algorithms explore the search tree of possible moves using backtracking with pruning
- Although they cannot look at all possible moves, they can look deep enough to play good chess
- In 1968 International Master David Levy bet that he would not be beaten by a computer in the next decade, a bet he won
- He was beaten in 1989 by Deep Thought
- In 1997 Deep Blue beat Gary Kasparov the reigning world champion
- Modern chess engines such as Deep Rybka and Houdini have a chess rating of around 3200 (c.f. Magnus Carsen's 2872—the top human player)

Heuristic Search

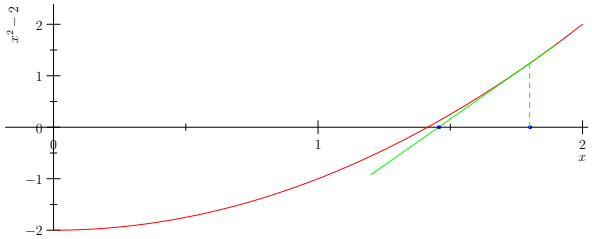
- When all else fails we have to settle for finding a good solution, not the best
- In fact, because so many problems that we are interested in turn out to be NP-hard this is quite common
- There are many strategies
  - Neighbourhood search (hill-climbing, descent)
  - Simulated annealing
  - Evolutionary algorithms, etc.

Lessons

- Many applications bring up interesting programming challenges
- Some are dealt with by using sensible data structures and common algorithms
- However, often you face a new challenge requiring thought
- Thinking in terms of strategies and having a feel for time complexity is part of armoury of a professional programmer

Algorithms and Analysis

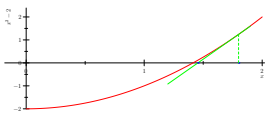
Lesson 30: *Understand Numerics*



Representing reals, rounding error, convergence, stability, conditioning

Outline

1. Numerical Approximations
2. Iterating to a Solution
3. Linear Algebra

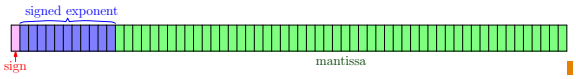


Numerical Analysis

- Numerical algorithms are usually taught separately from the "discrete algorithms" we have predominantly looked at
- The main difference stems from the fact that numerical algorithms model continuous variables
- Computers can only approximate continuous variables
- Numerical algorithms have to take into account this approximation

Representing Reals

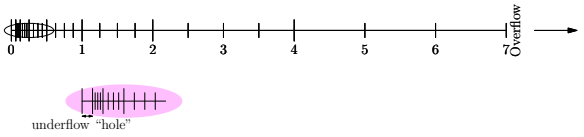
- All real numbers are approximated by a binary encoding



- $x = m \times 2^{e-t}$
- t is precision so that if $e = t$, then $0.5 \leq x < 1$
- For IEEE double $t = 1023$, $expon_{\min} = -1021$, $expon_{\max} = 1024$
- Typical rounding error is $u = 1 \times 10^{-16}$

The Number Line

- We approximate the continuous number line by a set of discrete values
- Imagine using a mantissa of 3 bits and an exponent of 2 bits (and a sign)

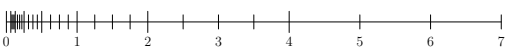


- The rounding error is half the gap between the discrete values

Overflow and Underflow

- An overflow will cause a program to fall over at run time
- An underflow is ignored
- This is usually innocuous, but can lead to trouble
- If you call `log(x)` or `1.0/x` but `x` has underflowed then your program will crash

Rounding Error



- The distance between two real numbers Δx grows with the number such that $\Delta x / x \leq u$ where $u \approx 10^{-16}$ for doubles
- Measure relative error

$$\text{Relative error} = \left| \frac{\text{Approx} - \text{Exact}}{\text{Exact}} \right|$$

- Thus almost every operation is only accurate up to this small (relative) rounding error
- Most operations are carefully designed that these rounding errors are unbiased so that the sum of random errors grows sub-linearly

Losing Precision

- There seems to be plenty of precision, so what's the problem?
- One issue is that its easy to lose precision
- Consider estimating derivatives by finite differencing

$$f'(x) \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\,\epsilon} = \frac{0.841470984861927 - 0.841470984753866}{2.0 \times 10^{-10}}$$

- The problem is $f(x+\epsilon)$ and $f(x-\epsilon)$ are very close so in taking their difference we lose precision
- $f(x) = \sin(x)$, $f'(x) = \cos(x)$ at $x = 1.0$

ϵ	10^{-6}	10^{-8}	10^{-10}	10^{-12}	10^{-14}
relative error	5×10^{-11}	5×10^{-9}	1×10^{-7}	2×10^{-5}	6×10^{-3}

Solving Quadratic Equations

- A classic example where you can lose precision is in solving a quadratic equation $a\,x^2 + b\,x + c = 0$

$$x = \frac{-b \pm \sqrt{b^2 - 4\,a\,c}}{2\,a}$$

- If $b^2 \gg |4\,a\,x|$ then for one solution we end up subtracting numbers very close
- We rather use this equation to compute one solution

$$x_1 = \frac{-b - \mathrm{sgn}(b)\sqrt{b^2 - 4\,a\,c}}{2\,a}$$

- Use the identity $x_1\,x_2 = c/a$ to find x_2 (i.e. $x_2 = c/(a\,x_1)$)

Accumulation of Rounding Error

- With many significant figures surely we can afford to lose some accuracy?
- This is sometimes true, but we often use "for loops" where we might be losing accuracy all the time

```
x = 1.6;
for (i=0; i<50; i++)
   x = sqrt(x);
for (i=0; i<50; i++)
   x = x*x;
```

```
1.6000000000000001
1.2649110640673518
1.1246826503869981
1.0605105611830079
1.0298109346783069
1.0147960064359274
1.0073708385872242
1.0036786530494828
1.0018376380679 07
1.0009183973071467
:
1.0000000000000002
```

- Gave the answer 1.2840 (if I run the for loop 60 times it gives the answer 1 for almost any input)

Coping With Truncation Errors

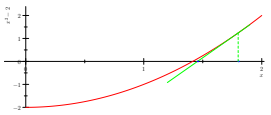
- Nothing is exact so to check that $x = y$ we use

```
Math.abs(x-y) < 1.0e-10  // a small constant
```

- Sometimes sums that add up to 1 don't quite so we have not to rely on anything being exact
- Avoid operations that are likely to lose accuracy (e.g. by taking the difference of similar numbers) where possible
- Sometimes it pays to do some operations using higher precision **`long double`**
- Make sure that errors are unbiased

Outline

1. Numerical Approximations
2. **Iterating to a Solution**
3. Linear Algebra



Iterative Algorithms

- We solve many numerical tasks by obtaining successively better solutions

$$x^{(0)},\ x^{(1)},\ x^{(2)},\ x^{(3)},\ x^{(4)},\ \ldots$$

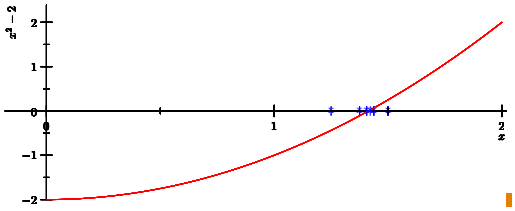
- We often stop when the change in solution is below some threshold, e.g. $|x^{(i+1)} - x^{(i)}| \leq \epsilon \approx u$
- The time complexity depends on the speed of convergence
- This can range from very fast to miserably slow

Bisection

- Suppose we want to compute $\sqrt{2}$ (without using `sqrt(2)`)

$$f(x) = x^2 - 2 = 0$$

- One of the classic methods of solving $f(x) = 0$ is **bisection**

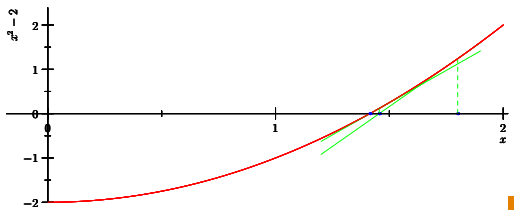


Newton Raphson

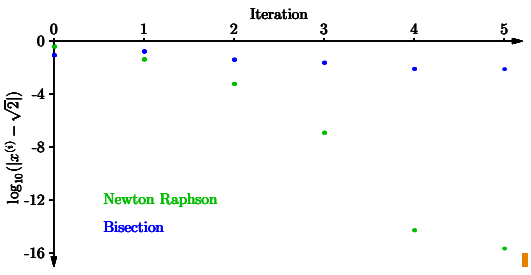
- A second classic method to solve $f(x) = 0$ is Newton-Raphson's method

$$x^{(i+1)} = x^{(i)} - \frac{f(x^{(i)})}{f'(x^{(i)})}$$

- For $f(x) = x^2 - 2$ so $x^{(i+1)} = ((x^{(i)})^2 - 1)/(2\,x^{(i)})$



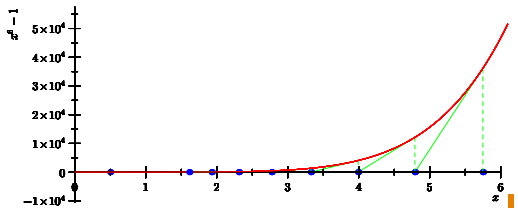
Convergence



- Bisection shows linear convergence (exponential increase in accuracy)
- Newton Raphson shows quadratic convergence

Beware of Asymptotic Convergence

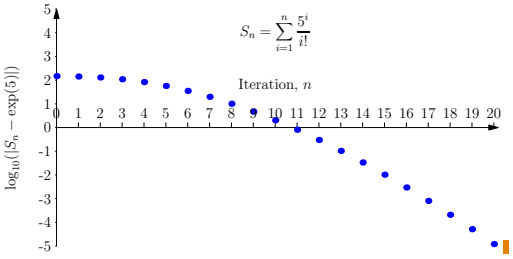
- Newton Raphson only converges quadratically if you start close enough to the solution
- Consider solving $x^6 - 1 = 0$ starting with $x^{(0)} = 0.5$



Evaluating Functions

- We can evaluate many functions using a series expansion

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$$



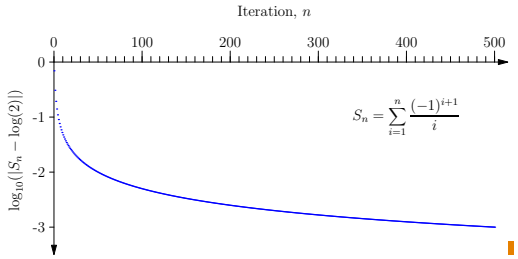
- For large i this converges since $i! \gg x^i$

Slow convergence

- Some expansions converge rather slowly (or even diverge)

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \cdots$$

- Converges for $-1 < x \leq 1$, but converges slowly for $x = 1$



Convergence

- Many functions can be approximated by a sum
- We get a truncation error by taking only a finite number of elements
- We want the truncation error to be around machine accuracy
- For quick evaluation we need a strongly convergent series
- This often depend on the value of the argument we give to the function
- Most special functions are approximated by different series depending on the input argument

Differential Equations

- Differential equations are used in many applications, for example in modelling the motion of object
- A typical equation of motion might be

$$\frac{\mathrm{d}^2x(t)}{\mathrm{d}t^2} = 2\frac{\mathrm{d}x(t)}{\mathrm{d}t} + 3\,x(t)$$

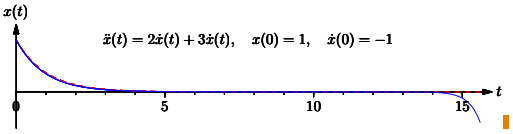
- Which has a general solution $x(t) = c_1 e^{-t} + c_2 e^{3\,t}$
- The constants are determined by initial conditions, for example, if $x(0) = 1$ and $\dot{x}(0) = -1$ then $x(t) = e^{-t}$

Euler's Method

- To solve a differential equation we use an approximate update equation

$$x(t + \epsilon) \approx x(t) + \epsilon\dot{x}(t)$$
$$\dot{x}(t + \epsilon) \approx \dot{x}(t) + \epsilon\ddot{x}(t)$$

- This becomes more exact as $\epsilon \to 0$

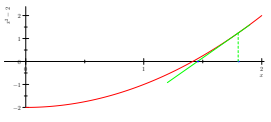


Stability

- Some iterative equations are unstable
- Round off errors can push a system of equations towards an unstable solution
- This can sometimes be overcome by cunning (e.g. running the equations backwards)
- Finding stable algorithms and avoiding unstable algorithms can be key to getting accurate predictions

Outline

1. Numerical Approximations
2. Iterating to a Solution
3. **Linear Algebra**



Solving Simultaneous Equations

- When problems involve many variables it is convenient to use matrices and vectors to store the numbers

$$\begin{aligned} 3x + 2y &= 5 \\ 7x - 8y &= -11 \end{aligned} \qquad \begin{pmatrix} 3 & 2 \\ 7 & -8 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 5 \\ -11 \end{pmatrix}$$

- Or $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$ with solution $\boldsymbol{x} = \mathbf{A}^{-1}\mathbf{b}$
- Linear algebra is an abstraction allowing mathematicians, scientists and engineers to write solutions at a higher level
- The job of the numerical analyst is to write the code that does this

Linear Algebra

- There are a large number of problems with matrices that people care about
- The solution often depends on the problem
- These include
  - Multiply matrices together
  - Solving linear equations $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$
  - Finding eigenvalues of symmetric and non-symmetric matrices
  - Performing singular valued decomposition
- These are important tasks that need to be done efficiently and reliably

Solving Linear Equations

- We consider the classic problem of solving $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$
- Although we can solve this by computing $\mathbf{A}^{-1}\boldsymbol{b}$, finding the inverse of a matrix is typically a $\Theta(n^3)$ operation
- It is preferable to decompose $\mathbf{A}$ into a product of a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ which takes $\Theta(n^2)$ operations

$$\mathbf{A} = \begin{pmatrix} 4 & 2 & 6 \\ 3 & 5 & 9 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0.75 & 1 & 0 \\ 0.25 & 0.428 & 1 \end{pmatrix} \begin{pmatrix} 4 & 2 & 6 \\ 0 & 3.5 & 4.5 \\ 0 & 0 & -4.28 \end{pmatrix} = \mathbf{LU}$$

- Solving $\boldsymbol{x} = \mathbf{U}^{-1}(\mathbf{L}^{-1}\boldsymbol{b})$ is also $\Theta(n^2)$ because of the structure of $\mathbf{L}$ and $\mathbf{U}$

LU-Decomposition

- LU-decomposition is achieved by Gaussian-elimination
- This is a straightforward procedure, but if done carelessly can lead to large rounding errors
- The standard solution is to permute the rows of the matrix (aka pivoting) to prevent loss of accuracy
- In addition we can "polish" solutions

$$\mathbf{A}(\boldsymbol{x} + \delta\boldsymbol{x}) - \boldsymbol{b} = \boldsymbol{\epsilon}$$

- Thus $\delta\boldsymbol{x} = \mathbf{A}^{-1}\boldsymbol{\epsilon}$ which we can use to get an improved estimate of $\boldsymbol{x}$

Norms

- With some work we can get a good approximation to $\boldsymbol{x}$ such that $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$
- But what if we have some error in $\boldsymbol{b}$, this induces an error $\delta\boldsymbol{x} = \mathbf{A}^{-1}\,\delta\boldsymbol{b}$
- How big is $\delta\boldsymbol{x}$?
- To measure the size of a vector we use a norm $\|\delta\boldsymbol{x}\|$, which is a number encoding the size of $\delta\boldsymbol{x}$
- There are a number of different norms, e.g.

$$\|\delta\boldsymbol{x}\|_2 = \sqrt{\delta x_1^2 + \cdots + \delta x_n^2}, \quad \|\delta\boldsymbol{x}\|_1 = |\delta x_1| + \cdots + |\delta x_n|$$

Conditioning

- The size of the error in $\boldsymbol{x}$: $\mathbf{A}\,\boldsymbol{x} = \boldsymbol{b}$ when $\boldsymbol{b}$ has error $\delta\boldsymbol{b}$ is

$$\|\delta\boldsymbol{x}\| = \|\mathbf{A}^{-1}\delta\boldsymbol{b}\| \leq \|\mathbf{A}^{-1}\|\,\|\delta\boldsymbol{b}\|$$

- Where $\|\mathbf{A}^{-1}\|$ provides a measure of the size of the error in the worst case
- For large matrices $\|\mathbf{A}^{-1}\|$ can be large meaning that any error in $\boldsymbol{b}$ is potentially magnified significantly
- Such matrices are said to be ill-conditions
- Ill-conditioning is not to due with rounding errors but the structure of the matrix

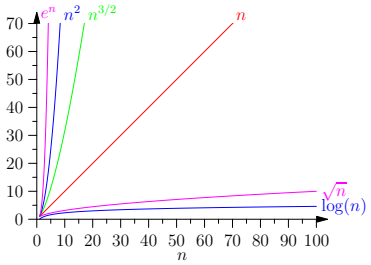
Linear Algebra

- Linear algebra packages provide an important set of tools used for solving linear equations
- Care has to be taken to ensure that needless operations (such as inverting a matrix) are not done
- Algorithms must ensure that as little accuracy as possible is lost (e.g. by permuting rows in LU-decomposition)
- Even when the algorithms are precise, small errors can get amplified in some operations, which requires care in formulating the problem
- The idea of poor conditioning (errors being amplified) is useful in understanding many numerical tasks

Lessons

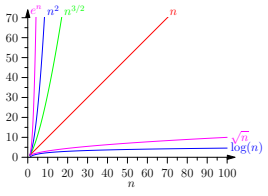
- Be wary of numerical algorithms, because computers approximate real numbers you don't always get what you expect
- Don't avoid numerical algorithms, they are hugely important with vast areas of applications
- This is a well studied area with large libraries of reliable algorithms that work most of the time
- There are some good books such as "Numerical Recipes" by Press, *et al.*, which describes the issues and provides code

Lesson 31: *Understand Time Complexity*



Theta, Big-O, little-o, Big-Omega, little-omega

1. **Time Complexity Classes**
   - Theta—Θ
   - Big O
   - Little o
   - Big Omega—Ω
   - Little omega—ω
2. Computing Time Complexity



Recap

- We have seen many algorithms taking times of order 1, $\log(n)$, n, $n\log(n)$, n^2, etc
- Sometimes these are worst time, average time or best time results
- We have lots of different notations, e.g. $O(1)$, $\Theta(\log(n))$, $\Omega(n^2)$, etc.
- What does it all mean?

Complexity Class Sets

- The correct way to think about complexity classes is in terms of sets
- Suppose we have an algorithm which takes an input of size n and computes an output in $f(n)$ operations
- E.g. $f(n) = 4n^2 + 2n + 3$
- We can partition all run times into sets by considering only the leading order term and ignoring the constant term
- We denote these sets by $\Theta(g(n))$
  - $4n^2 + 2n + 3 \in \Theta(n^2)$
  - $5n\log(n) + 3n + 2 \in \Theta(n\log(n))$

Defining $\Theta(g(n))$

- A function $f(n) \in \Theta(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad 0 < c < \infty$$

- E.g.

$$\lim_{n\to\infty} \frac{4n^2 + 2n + 3}{n^2} = 4$$
$$\lim_{n\to\infty} \frac{5n\log(n) + 3n + 2}{n\log(n)} = 5$$

Ignoring the Constant

- Does an algorithm that uses $4n^2$ operations run faster than one that uses $7n^2$ operations?
- Answer depends on the operations which might depend on the programming language or the machine architecture
- However asymptotically (i.e. for sufficiently large n) an order $n\log(n)$ algorithm will always run faster than an order n^2 algorithm even when they are run on different machines
- The constant is important in practice (if there are two algorithms A and B that are both $n\log(n)$, but algorithm A runs twice as fast as algorithm B, which one should you use?)
- Nevertheless, ignoring the constant is often essential to make analysis of algorithms doable

Ordering Complexity Classes

- We can define the relation $\Theta(f(n)) < \Theta(g(n))$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

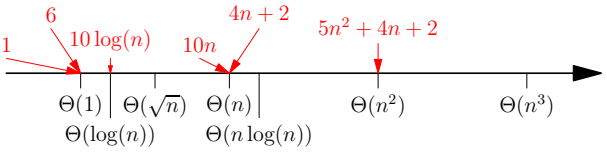
- Informally if algorithm A has time complexity $\Theta(f(n))$ and algorithm B has time complexity $\Theta(g(n))$ then if $\Theta(f(n)) < \Theta(g(n))$ algorithm A is faster for sufficiently large n
- The relation defines a **complete ordering**

The Complexity Line

- We can order all complexity classes. E.g.

$$\Theta(1) < \Theta(\log(n)) < \Theta(\sqrt{n}) < \Theta(n) < \Theta(n^2)$$

- We can depict this as a complexity line



- The line is dense (i.e. there are an uncountable infinity of complexity classes)

Complexity Dependent on Inputs

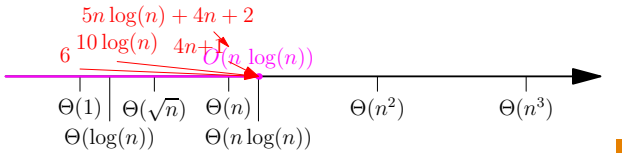
- The run time of many algorithms depends on the input
- In this case we can define different time complexities
  - Worst case time complexity (the longest time an algorithm will take)
  - Average complexity (the expected time averaged over all possible inputs)
  - Best case time complexity (the shortest time an algorithm will take)—usually not very interesting
- Every algorithm will have a Θ complexity class for the worst, average and best time complexity

Unknown Time Complexity

- Algorithms are often rather complicated and knowing the exact time complexity (for either worst, average or best cases) might not be known
- In reality it will have some run time (e.g. $f(n) = 3n^2\log(n) + 2n^2 - n + 3$) and will belong to a Θ time complexity set (e.g. $\Theta(n^2\log(n))$) but we might not be able to calculate it
- However, we can usually bound the run times of algorithms

Big-O

- Big-O is an upper bound on the time complexity
- If an algorithm is $O(g(n))$ then its time complexity is no more than $\Theta(g(n))$



- I.e. $\Theta(f(n)) \leq \Theta(g(n))$ implies $f(n) \in O(g(n))$

Upper Bounding Time Complexity

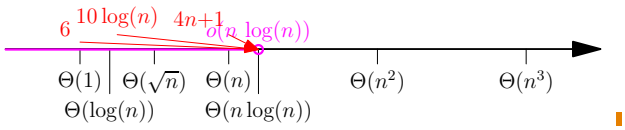
- Consider a program

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- If the `if` statements is never true this is a $\Theta(n)$ algorithm if it is always true it is a $\Theta(n^2)$ algorithm
- If we don't know then we can at least say that the run time is in $O(n^2)$—we assume the worst but the worst may never happen

Little-o

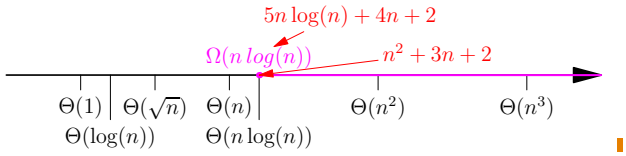
- Sometimes we want to say that the time complexity for an algorithm $\Theta(f(n))$ is **strictly less** than a known time complexity $\Theta(g(n))$



- I.e. $\Theta(f(n)) < \Theta(g(n))$ implies $f(n) \in o(g(n))$

Lower Bounds—Ω

- It is often easy to obtain a lower bound on a particular algorithm, although getting a tight general lower bound is often very difficult
- I.e. $\Theta(f(n)) \geq \Theta(g(n))$ implies $f(n) \in \Omega(g(n))$



Lower Bounding Time Complexity

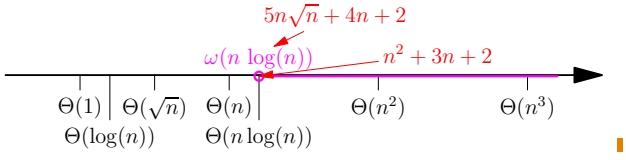
- Returning to the program

```
// define stuff
for(int i=0; i<n; i++)  {
  // do something
  if (/* some condition */) {
    for (int j=0; j<n; j++) {
      // do other stuff
    }
  }
}
// clean up
```

- We might not know how frequently the `if` statement is true, but we know in all cases the first `for` loop iterates over n
- Thus we know this algorithm is in $\Omega(n)$—we assume the best but the best may never happen

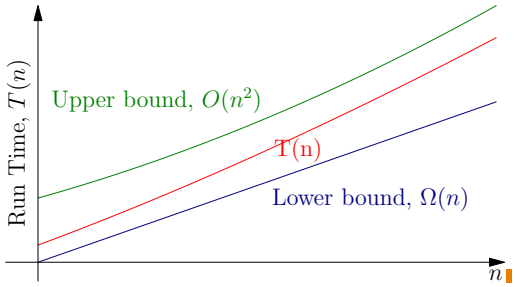
Little Omega—ω

- It is sometimes useful to talk about a strict lower bound
- I.e. $\Theta(f(n)) > \Theta(g(n))$ implies $f(n) \in \omega(g(n))$



Bounding Run Time Complexity

- When we are given an algorithm to analyse we want to compute $\Theta(n)$
- This may be difficult, however, it is often easy to find bounds



Proving Asymptotic Time Complexity

- If we know an algorithm is
  - $T(n) \in O(f(n))$
  - $T(n) \in \Omega(f(n))$
- Then $T(n) \in \Theta(f(n))$
- This is a common proof strategy

Meaning of Time Complexity

- Insertion sort has time complexity $\Theta(n^2)$
- Because it consists of two `for` loops
- It takes 2 seconds to sort $100\,000$ items
- How long does it take to sort $1\,000\,000$ items?
- n increases by 10, time complexity increases by $10^2 = 100$
- Time taken is approximately 200 seconds or around 3.5 minutes

Exponential Time Complexity

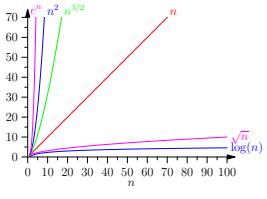
- When we talk about exponential time complexity we usually mean that

$$\log(T(n)) \in \Theta(n)$$

- This is true if
  - $T(n) = 2^n \quad \log(T(n)) = n\,\log(2)$
  - $T(n) = 6.1\,\mathrm{e}^{0.003n} \quad \log(T(n)) = 0.03\,n + \log(6.1)$
  - $T(n) = n\,10^n \quad \log(T(n)) = n\,\log(10) + \log(n)$
- Note that none of these are in complexity class $\Theta(\mathrm{e}^n)$

Outline

1. Time Complexity Classes
   - Theta—Θ
   - Big O
   - Little o
   - Big Omega—Ω
   - Little omega—ω
2. **Computing Time Complexity**



Counting For Loops

- How long does the following code take?

```
for(int i=0; i<n; i++)  {
  // prepare stuff
}
for(int i=0; i<n; i++)  {
  // do something
  for (int j=0; j<n; j++) {
    // do other stuff
  }
}
```

- The first `for` loop takes $\Theta(n)$ operations the second double `for` loop takes $\Theta(n^2)$
- Answer $\Theta(n^2)$

Recursion

- Determining time complexity is harder when we use recursion
- Consider Euclid's algorithm for determining the greatest common divisor

```
long gcd(long m, long n)
{
  while(n!=0) {
    long rem = m%n;
    m = n;
    n = rem;
  }
  return m;
}
```

```
long gcd(long m, long n)
{
  if (n==0)
    return m;
  else
    return gcd(n, m%n);
}
```

- This doesn't even look like a recursion

Example of gcd

- Example of Euclid's algorithm `gcd(1989,1590)`
- Sequence of remainders is 399, 393, 6, 3, 0
- The greatest common divisor is 3
- How long does it take compute `gcd(n,m)` with $n > m$
- This is subtle as could depend in a complex way on the pair n and m

Recursive Formulae

- An observation which makes the analysis relatively simple is that the remainder is reduced by at least 2 after two iterations
- To prove
  - Using the recursion (assuming $m, n < 0$)

$$\gcd(m,n) = \gcd(n, \mathrm{rem}(m,n)) = \gcd(\mathrm{rem}(m,n), \mathrm{rem}(n, \mathrm{rem}(m,n)))$$

  - The proof follows by showing that $\mathrm{rem}(n, \mathrm{rem}(m,n)) < n/2$
- Thus $T(n) < T(n/2) + 2$

Solving Recursions

- To show that $T(n) \in O(\log(n))$ we observe
  - Note that $T(1) = 1$

$$T(n) < T(2^{-1}n) + 2 < T(2^{-2}n) + 4 < \cdots < T(2^{-t}n) + 2t$$

  - Choose $t = \lceil \log_2(n) \rceil$
  - then $2^{-t}n = 2^{-\lceil \log_2(n) \rceil}n \leq 2^{-\log_2(n)}n = \frac{n}{n} = 1$
  - Thus $T(2^{-t}n) < T(1) = 1$
  - $T(n) < 1 + 2t = 1 + 2\lceil \log_2(n) \rceil \in O(\log(n))$
- A huge calculation shows the the average number of iterations is about $(12 \log(2) \log(n))/\pi^2 + 1.47$

Probability of Relative Primes

- Consider the following program to compute the probability of relative primes for all numbers up to n

```
public static double probRelPrime(n)
{
  int rel=0, tot=0;
  for(int i=1; i<=n; i++)
     for(int j=i+1; j<=n; j++) {
       tot++;
       if (gcd(i,j)==1)
          rel++;
     }
  return (double) rel / tot;
}
```

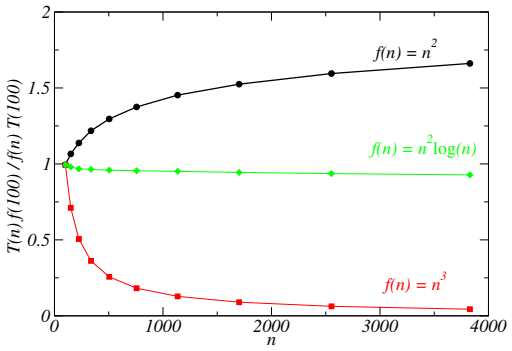
- What is the time complexity?

Time Complexity

- Program involves two nested loops of size $O(n)$
- Then we need to calculate `gcd(i, j)` at each iteration
- Time complexity is $n \times n \times \log(n) = n^2 \log(n)$
- How could we provide empirical support for this calculation?

Testing Hypothesis

- We can test our hypothesis by scaling the run time by the complexity



Conclusions

- You should understand the difference between Θ, O,o, Ω and ω
- You need to be able to compute time complexity by loop counting
- To compute time complexity for recursive functions you need to be able to obtain recurrence equations
- You should be able to solve simple recurrence equations and sum up simple series
- You should be able to prove more complicated results using proof by induction
- Thank you for attending the course