COMP20007 Design of Algorithms

Introduction and Welcome

Lars Kulik

Lecture 1

Semester 1, 2020

Welcome to COMP20007

- Data structures, including stacks, queues, trees, priority queues and graphs.
- Algorithms for various problems, including sorting, searching, string manipulation, graph manipulation, and more.
- Algorithmic techniques, including including brute force, decrease-and-conquer, divide-and-conquer, dynamic programming and greedy approaches.
- Analytical and empirical assessment of algorithms.
- Complexity classes.

Anany Levitin. *Introduction to the Design and Analysis of Algorithms* Pearson, 2012.
Steven S Skiena. *The Algorithm Design Manual* Springer, 2008.

Staff; Learning Management System

Lecturer and subject coordinators: Daniel Beck and Lars Kulik

Tutors: Tobias Edwards (head tutor), Lianglu Pan, Anh Vo, Alex Ligthart-Smith, Luca Kennedy, Santa Maiti, William Price

Tobias will have a weekly time for consultation in the Doug McDonell building. Exact time and venue to be announced.

Other support is provided by your classmates, for example via the LMS Discussion Board.

The LMS is our notice board, repository, and discussion forum.

The Timetable

We use lecture capture which is useful for revisiting points from a lecture; and it can, sort of, be a substitute for the lecture proper, even if it is a poor one. In other words: come to the classes whenever you can :).

Workshops are unfortunately not a single venue! You will need to walk from one venue to another one for the workshops.

We will also offer online workshops via Zoom. For details please visit the LMS.

They start in Week 2.

Time Commitment

For the 12 weeks of semester, expect

- 22 hours of classtime,
- 48 hours of reading and tute preparation
- 48 hours on assignments

That is roughly an average of 10 hours per week.

The commitment is well worth it: Knowledge of algorithms is essential for any computing professional, it expands your mind, improves complexion, and contains all the minerals and vitamins essential for developing boundless wisdom.

Assessment

- Assignment 1, due around Week 5-6, worth 10%.
- Mid-semester take-home test in Week 5-6, worth 0%.
- Assignment 2, due around Week 11-12, worth 20%.
- A 3-hour exam, worth 70%.

To pass the subject you must obtain at least

- 50% in assignments (total $\geq 15/30$); and
- 50% in the exam (total $\geq 35/70$).

Expectations

You need to catch up on any “assumed background knowledge” that you may not have:

- An understanding of sets and relations.
- A grasp of recursion and recurrence relations; a short tutorial on the latter is in Levitin’s book, Appendix B.
- Knowledge of basic data structures, such as arrays, records, linked lists, sets and dictionaries.
- Knowledge of some programming language that has a concept of “pointer”.

How to Succeed

Understand the material, don't just memorize it (apart, perhaps, from the formulas in Levitin's Appendix A).

If you fall behind, try to catch up as fast as possible.

Don't procrastinate. Start assignments before you are ready. Put in the necessary time.

Attempt the tutorial questions every week, before you attend the tutorial, if at all possible.

How to Succeed

Support the learning of your fellow students and expect their support, in class and through the LMS discussion board.

Remember that we are all on the same “learning journey” and have the same goal.

Participate in the discussions on the subject’s LMS site and check regularly for announcements.

Over to You—Introductions

Please introduce yourself to your neighbours.

Tell them where you are from, what degree program you are enrolled in, whatever.

Over to You—A Maze Problem

A maze (or labyrinth) is contained in a 10×10 rectangle; rows and columns are numbered from 1 to 10.

It can be traversed along rows and columns: up, down, left, right.

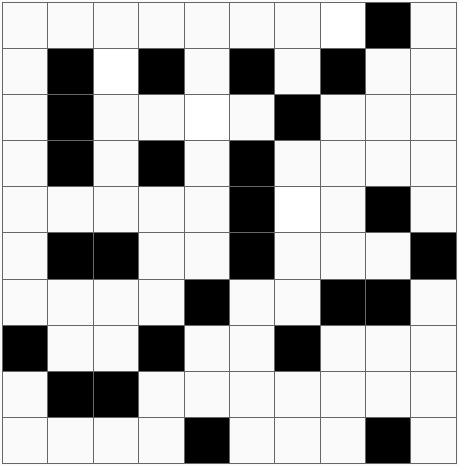
The starting point is (1,1), the goal point is (10,10).

These points are obstacles that you cannot travel through:

(3,2) (6,6) (2,8) (5,9) (8,4) (2,4) (6,3) (9,3)
(1,9) (3,7) (4,2) (7,8) (2,2) (4,5) (5,6) (10,5)
(6,2) (6,10) (7,5) (7,9) (8,1) (5,7) (4,4) (8,7)
(9,2) (10,9) (2,6)

Find a path through the maze.

The Maze Problem ...



What Is a Problem?

My ODE says: "doubtful or difficult question or task."

In computer science we use the term like that too, but there is a more technical concept of algorithmic problem.

We usually want to find a single generic solution to a bunch of similar questions.

For example, the "maze problem" is to come up with a mechanical solution to any particular maze.

So to us, a "problem" usually has many instances, sometimes infinitely many.

Algorithmic Problems

So a problem in computer science typically means a family of instances of a general problem.

An algorithm for the problem has to work for all possible instances (input).

Example: The sorting problem—an instance is a sequence of items.

Example: The graph colouring problem—an instance is a graph.

Example: Equation solving problems—an instance is a set of, say, linear equations.

What Is an **Algorithm**?

My ODE says: "process or rules for (esp. machine) calculation etc."

A finite sequence of instructions

- No ambiguity, and each step precisely defined
- Should work for all (well-formed) input
- Should finish in a finite (reasonable) amount of time

The (single) description of a process that will transform arbitrary input to the correct output—even when there are infinitely many possible inputs.

What Is an Algorithm?

Not long ago, "algorithm" was synonymous with "numeric algorithm".

Mathematicians had found many clever algorithms for all sorts of numeric problems.

The following algorithm for calculating the greatest common divisor of positive integers m and n is known as "Euclid's Algorithm".

To find $gcd(m, n)$:

Step 1: If $n = 0$, return the value of m as the answer and stop.
Step 2: Divide m by n and assign the value of the remainder to r.
Step 3: Assign the value of n to m, and the value of r to n; go to Step 1.

Non-Numeric Algorithms

350 years ago, Thomas Hobbes, in discussing the possibility of automated reasoning, wrote:

> *"We must not think that computations, that is, ratiocination, has place only in numbers."*

Today, numeric algorithms are just a small part of the syllabus in an algorithms course.

The kind of computation that Hobbes was really after was mechanised reasoning, that is, algorithms for logical formalisms, for example, to decide "does this formula follow from that?"

Computability

In 2012 we celebrated Alan M. Turing's 100th birthday.

At the time of Turing's birth, a "computer" was a human employed to do tedious numerical calculations.

Legacy: "Turing machine", the "Church-Turing thesis", "Turing reduction", the "Turing test", the "Turing award"

One of Turing's great accomplishments was to put the concept of an algorithm on a firm foundation and to establish that certain important problems do not have algorithmic solutions.

Abstract Complexity

In a course like this, we are only interested in problems that do have algorithmic solutions.

However, amongst those, there are many that provably do not have efficient solutions.

Towards the end of this subject we discuss complexity theory briefly—this theory is concerned with the inherent “hardness” of problems.

Why Study Algorithms?

Computer science is increasingly an enabler for other disciplines, providing useful tools for these.

Algorithmic thinking is relevant in the life sciences, in engineering, in linguistics, in chemistry, etc.

Today computers allow us to solve problems whose size and complexity is vastly greater than what could be done a century ago.

The use of computers has changed the focus of algorithmic study completely, because algorithms that work well for a human (small scale) usually do not work well for a computer (big scale).

Why Study Algorithms and Their Complexity?

To collect a number of useful problem solving tools.

To learn, from examples, strategies for solving computational problems.

To be able to write robust programs whose behaviour we can reason about.

To develop analytical skills.

To learn about the inherent difficulty of some types of problems.

Problem Solving Steps

- Understand the problem
- Decide on the computational means (sequential/parallel, exact/approximate)
- Decide on method to use (algorithm design technique or strategy, use of randomization)
- Design the necessary data structures and algorithm
- Check for correctness, trace example input
- Evaluate analytically (time, space, worst case, average case)
- Code it
- Evaluate empirically

What we will study

Algorithm analysis

Important algorithms for various problems, primarily

- Sorting
- Searching
- String processing
- Graph algorithms

Approaches to algorithm design

- Brute force
- Decrease and conquer
- Divide and conquer
- Transform and conquer

Study Tips

Before the lecture, as a minimum make sure you have read the introductory section of the relevant chapter.

Always read (and work) with paper and pencil ready; run algorithms by hand.

Always have a go at the tutorial exercises; this subject is very much about learning-by-doing.

After the lecture, reread and consolidate your notes.

Identify areas not understood and use the LMS Discussion Forum.

Rewrite your notes if that helps.

Things to Do in the First Two Weeks

Read the text, read Chapter 1, and skim Chapter 2.

Make sure you have a unimelb account.

Visit the COMP20007 LMS pages and check any announcements.

Use the LMS Discussion Board; for example, if you are interested in forming a study group with like-minded people, the Discussion Board is a useful place to say so.

COMP20007 Design of Algorithms

Design of Algorithms

Lars Kulik

Lecture 2

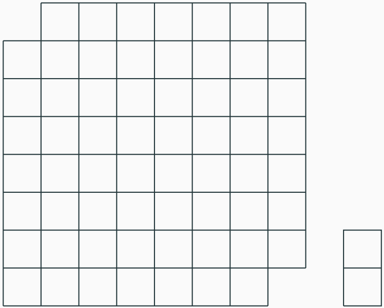
Semester 1, 2020

Approaching a Problem

Can we cover this board with 31 tiles of the form shown?

This is the mutilated checkerboard problem.

There are only finitely many ways we can arrange the 31 tiles, so there is a brute-force (and very inefficient) way of solving the problem.

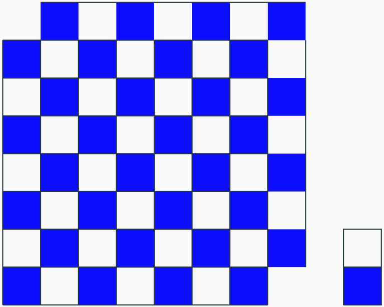


Transform and Conquer? Use Abstraction?

Can we cover this board with 31 tiles of the form shown?

Why can we quickly determine that the answer is no?

Hint: Using the way the squares are coloured helps.



✎

Algorithms: for solving problems, transforming data.

Data structures: for storing data; arranging data in a way that suits an algorithm.

- Linear data structures: Stacks and queues
- Trees and graphs
- Dictionaries

Which data structures are you familiar with?

Exercise: Data Structures

Pick a data structure and describe:

- How to insert an item into the data structure
- How to find an item
- How to handle duplicate items

Primitive Data Structures: The Array

An array consists of a sequence of consecutive cells in memory.

Depending on programming language: `A[0]` up to `A[n-1]`, or `A[1]` up to `A[n]`.

Locating a cell, and storing or retrieving data at that cell is very fast.

The downside of an array is that maintaining a contiguous bank of cells with information can be difficult and time-consuming.

Primitive Data Structures: The Linked List

A collection of objects with links to one another, possibly in different parts of the computer's memory.

Often we use a dummy head node that points to the first object, or to a special `null` object that represents an empty list.

Inserting and deleting elements is very fast: just move a few links around.

Finding the *i*th element can be time-consuming.

Iterative Processing

Walk through the array or linked list. For example, to locate an item.

```
j := 0
while j < last
  if A[j] == x
    return j
  j := j+1
return null
```

```
p := head
while p != null
  if p.val == x
    return p
  p := p.next
return null
```

Recursive Processing

Solve the problem on a smaller collection and use that solution to solve on the full collection.

```
function find(A,x,lo,hi)
  if lo > hi
    return null
  else if A[lo] == x
    return lo
  else
    return find(A,x,lo+1,hi)
```

Initial call: `find(A,x,0,last)`

```
function find(p,x):
  if p == null
    return p
  else if p.val == x
    return p
  else
    return find(p.next,x)
```

Initial call: `find(head,x)`

We return to recursion in more depth later.

Abstract Data Types

A collection of data items, and a family of operations that operate on that data.

Think of an ADT as a set of promises, or contracts.

We must still implement these promises, but it is an advantage to separate the implementation of the ADT from the “concept”.

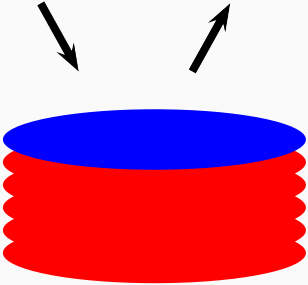
Good programming practice is to support this separation: Nothing outside of the definitions of the ADT should refer to anything inside, except through function calls for the basic operations.

Fundamental Data Structures: The Stack

Last-in-first-out (LIFO).

Operations:

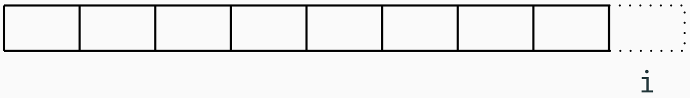
- CreateStack
- Push
- Pop
- Top
- EmptyStack?
- ...



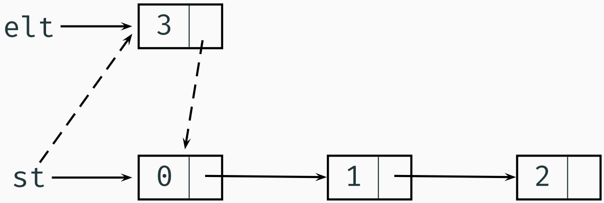
Usually implemented as an ADT.

Stack Implementation

By array:



By linked list (push):

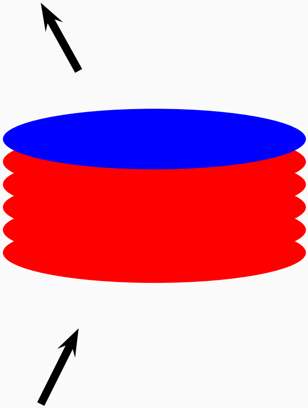


Fundamental Data Structures: The Queue

First-in-first-out (FIFO).

Operations:

- CreateQueue
- Enqueue
- Dequeue
- Head
- EmptyQueue?
- ...



Other Data Structures

We shall meet many other (abstract) data structures, such as

- The priority queue
- Various types of “tree”
- Various types of “graph”

Next Up

Algorithm analysis - how to reason about an algorithm's resource consumption.

COMP20007 Design of Algorithms

Growth Rate and Algorithm Efficiency

Lars Kulik

Lecture 3

Semester 1, 2020

Assessing Algorithm "Efficiency"

Resources consumed: time and space.

We want to assess efficiency as a function of input size:

- Mathematical vs empirical assessment
- Average case vs worst case

Knowledge about input peculiarities may affect the choice of algorithm.

The right choice of algorithm may also depend on the programming language used for implementation.

Running Time Dependencies

There are many things that a program's running time depends on:

1. The complexity of the algorithms used
2. Input to the program
3. Underlying machine, including memory architecture
4. Language/compiler/operating system

Since we want to compare algorithms, we ignore (3) and (4); just consider units of time.

Use a natural number n as measure of (2)—size of input.

Express (1) as a function of n.

The RAM Word Model

Assumptions

- Data is represented in words of fixed length in bits (e.g., 64-bit)
- Fundamental operations on words take one unit of time

Basic arithmetic:	$+$, $-$, $\times$, $/$
Memory access:	load, store
Comparisons:	$<$, $>$, $=$, $\neq$, $\geq$, $\leq$
Logical operators:	&&, ||
Bitwise operators:	&, |

The goal of analysis is to count the operations based on parameters such as input size or problem size.

Estimating Time Consumption

If c is the cost of a basic operation and $g(n)$ is the number of times the operation is performed for input of size n,

then running time $t(n) \approx c \cdot g(n)$.

Examples: Input Size and Basic Operation

Problem	Size measure	Basic operation
Search in list of n items	n	Key comparison
Multiply two matrices of floats	Matrix size (rows times columns)	Float multiplication
Graph problem	Number of nodes and edges	Visiting a node

Best, Average, or Worst Case?

The running time $t(n)$ may well depend on more than just n.

Worst-case analysis makes the most adverse assumptions about input. Are they the worst n things your algorithm could see?

Best-case analysis makes optimistic assumptions. Are they the best n things the algorithm could see?

Average-case analysis aims to find the expected running time across all possible input of size n. (Note: This is not an average of the worst and best cases but assumes that your input is drawn randomly from all possible inputs of size n.)

Amortised analysis takes the context of running an algorithm into account and calculates cost spread over many runs.

Average-case Analysis: Sequential Search

function $\textsc{SequentialSearch}(A[0..n-1], K)$
    $i \leftarrow 0$
    while $i < n$ and $A[i] \neq K$ **do**
        $i \leftarrow i + 1$
    if $i < n$ **then**
        return i
    else
        return -1

If the probability of a successful search is equal to p $(0 \leq p \leq 1)$, then the average number of average number of key comparisons $C_{avg}(n)$ is

$$p \times (n+1)/2 + n \times (1-p).$$

Large Input Is What Matters

Small input does not provide a stress test for an algorithm.

As an alternative to Euclid's algorithm (Lecture 1) we can find the greatest common divisor of m and n by testing each k no greater than the smaller of m and n, to see if it divides both.

For small input (m, n), both these versions of *gcd* are fast.

Only as we let m and n grow large do we witness (big) differences in performance.

The Tyranny of Growth Rate

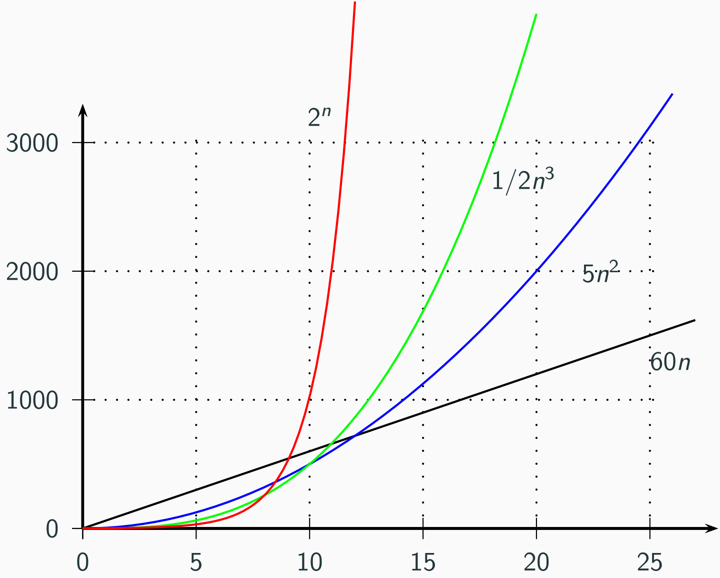
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10^1	3	10^1	$3 \cdot 10^1$	10^2	10^3	10^3	$4 \cdot 10^6$
10^2	7	10^2	$7 \cdot 10^2$	10^4	10^6	10^{30}	$9 \cdot 10^{157}$
10^3	10	10^3	$1 \cdot 10^4$	10^6	10^9	–	–

10^{30} is one thousand times the number of nano-seconds since the Big Bang.

At a rate of a trillion (10^{12}) operations per second, executing 2^{100} operations would take a computer in the order of 10^{10} years.

That is more than the estimated age of the Earth.

The Tyranny of Growth Rate



Functions Often Met in Algorithm Classification

1: Running time independent of input.

$\log n$: Typical for “divide and conquer” solutions, for example, lookup in a balanced search tree.

n: Linear. When each input element must be processed once.

$n \log n$: Each input element processed once and processing involves other elements too, for example, sorting.

n^2, n^3: Quadratic, cubic. Processing all pairs (triples) of elements.

2^n: Exponential. Processing all subsets of elements.

Asymptotic Analysis

We are interested in the growth rate of functions:

- Ignore constant factors
- Ignore small input sizes

Asymptotics

$$f(n) \prec g(n) \text{ iff } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

That is: g approaches infinity faster than f. For example,

$$1 \prec \log n \prec n^{\epsilon} \prec n^{c} \prec n^{\log n} \prec c^{n} \prec n^{n}$$

where $0 < \epsilon < 1 < c$.

In asymptotic analysis, think big!

For example, $\log n \prec n^{0.0001}$, even though for $n = 10^{100}, 100 > 1.023$.

Big-Oh Notation

$O(g(n))$ denotes the set of functions that grow no faster than g, asymptotically.

We write

$$t(n) \in O(g(n))$$

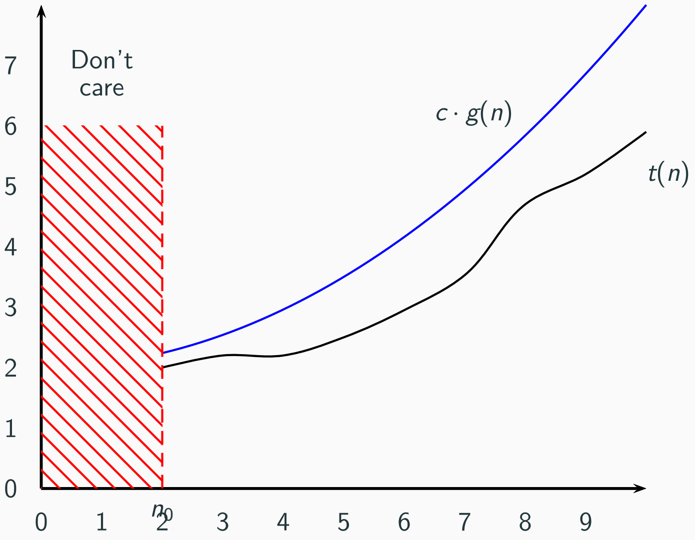
when, for some c and n_0,

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

For example,

$$1 + 2 + \cdots + n \in O(n^2)$$

Big-Oh: What $t(n) \in O(g(n))$ Means



Big-Oh Pitfalls

Levitin's notation $t(n) \in O(g(n))$ is meaningful, but not standard.

Other authors use $t(n) = O(g(n))$ for the same thing.

As O provides an upper bound, it is correct to say both $3n \in O(n^2)$ and $3n \in O(n)$ (so you can see why using '$=$' is confusing); the latter, $3n \in O(n)$, is of course more precise and useful.

Note that c and n_0 may be large.

Big-Omega and Big-Theta

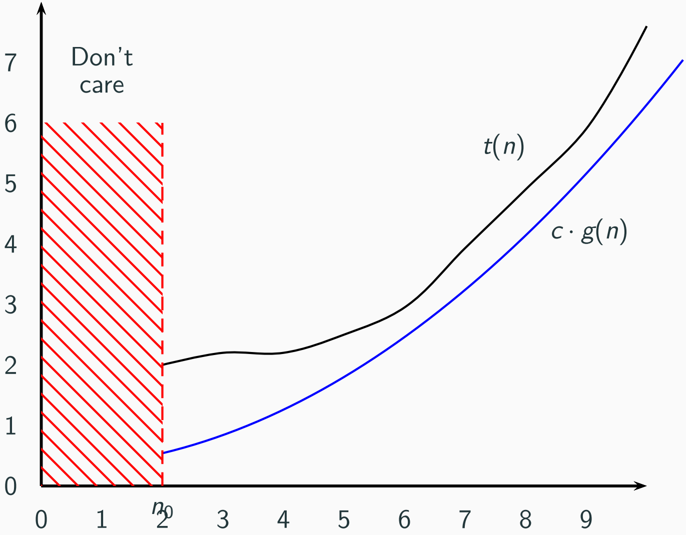
$\Omega(g(n))$ denotes the set of functions that grow no slower than g, asymptotically, so Ω is for lower bounds.

$t(n) \in \Omega(g(n))$ iff $n > n_0 \Rightarrow t(n) > c \cdot g(n)$, for some n_0 and c.

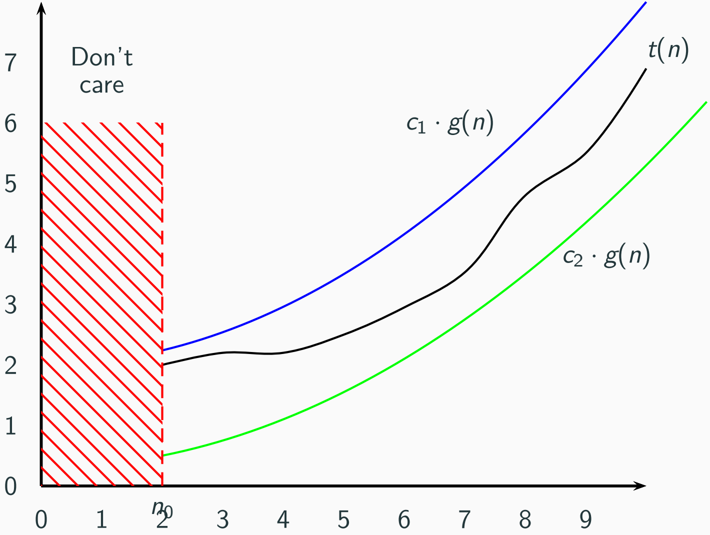
Θ is for exact order of growth.

$t(n) \in \Theta(g(n))$ iff $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$.

Big-Omega: What $t(n) \in \Omega(g(n))$ Means



Big-Theta: What $t(n) \in \Theta(g(n))$ Means



Establishing Growth Rate

We can use the definition of O directly.

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

Exercise: Use this to show that

$$1 + 2 + \cdots + n \in O(n^2)$$

Also show that

$$17n^2 + 85n + 1024 \in O(n^2)$$

✎

Next Up

We go through some examples of time complexity analysis for specific algorithms.

COMP20007 Design of Algorithms

Analysis of Algorithms

Lars Kulik

Lecture 4

Semester 1, 2020

Establishing Growth Rate

In the last lecture we proved $t(n) \in O(g(n))$ for some cases of t and g, using the definition of O directly:

$$n > n_0 \Rightarrow t(n) < c \cdot g(n)$$

for some c and n_0. A more common approach uses

$$\lim_{n \to \infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & \text{implies } t \text{ grows asymptotically slower than } g \\ c & \text{implies } t \text{ and } g \text{ have same order of growth} \\ \infty & \text{implies } t \text{ grows asymptotically faster than } g \end{cases}$$

Use this to show that $1000n \in O(n^2)$. ✎

L'Hôpital's Rule

Often it is helpful to use L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

where t' and g' are the derivatives of t and g.

For example, we can show that $\log_2 n$ grows slower than $\sqrt{n}$:

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

Induction Trap (Polya)

- $A(n)$: All horses are the same colour
- Base case: $A(1)$ is trivially true (only one horse)
- Assume in a set of n horses, all are the same colour
    - For a set of $n+1$ horses, take the subsets $\{1, \ldots, n\}$ and $\{2, \ldots, n+1\}$.
    - Both subsets are of size n, so all horses are the same colour in each subset (by inductive hypotheses).
    - Since $n-1$ of the horses are the same in both sets, the horses in both sets must be all the same colour, hence all $n+1$ horses are the same colour.
- What went wrong?

Example: Finding the Largest Element in a List

function $\textsc{MaxElement}(A[0..n-1])$
    $max \leftarrow A[0]$
    for $i \leftarrow 1$ to $n-1$ **do**
        if $A[i] > max$ **then**
            $max \leftarrow A[i]$
    return max

We count the number of comparisons executed for a list of size n:

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 = \Theta(n)$$

Example: Selection Sort

function $\textsc{SelSort}(A[0..n-1])$
  for $i \leftarrow 0$ to $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ to $n-1$ **do**
      if $A[j] < a[min]$ **then**
        $min \leftarrow j$
    swap $A[i]$ and $A[min]$

We count the number of comparisons executed for a list of size n:

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i) = (n-1)^2 - \sum_{i=0}^{n-2} i$$

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

Example: Matrix Multiplication

function
$\textsc{MatrixMult}(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
  for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
      $C[i,j] \leftarrow 0.0$
      for $k \leftarrow 0$ to $n-1$ **do**
        $C[i,j] \leftarrow C[i,j] + A[i,k] \cdot B[k,j]$
  return C

The number of multiplications executed for a list of size n is:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

Analysing Recursive Algorithms

Let us start with a simple example:

function $\mathrm{F}(n)$
    if $n = 0$ **then return** 1
    else return $\mathrm{F}(n-1) \cdot n$

The basic operation here is the multiplication.

We express the cost recursively as well:

$$\begin{array}{rcl} M(0) & = & 0 \\ M(n) & = & M(n-1) + 1 \quad \text{for } n > 0 \end{array}$$

To find a closed form, that is, one without recursion, we usually try "telescoping", or "backward substitutions" in the recursive part.

Telescoping

The recursive equation was:

$$\boxed{M(n) = M(n-1) + 1 \quad (\text{for } n > 0)}$$

Use the fact $M(n-1) = M(n-2) + 1$ to expand the right-hand side:

$$M(n) = [M(n-2) + 1] + 1 = M(n-2) + 2$$

and keep going:

$$\ldots = [M(n-3) + 1] + 2 = M(n-3) + 3 = \ldots = M(n-n) + n = n$$

where we used the base case $M(0) = 0$ to finish.

A Second Example: Binary Search in Sorted Array

function $\textsc{BinSearch}(A[], lo, hi, key)$
  if $lo > hi$ **then return** -1
  $mid \leftarrow lo + (hi - lo)/2$
  if $A[mid] = key$ **then return** mid
  else
    if $A[mid] > key$ **then**
      return $\textsc{BinSearch}(A, lo, mid - 1, key)$
    else return $\textsc{BinSearch}(A, mid + 1, hi, key)$

The basic operation is the key comparison. The cost, recursively, in the worst case:

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n/2) + 1 \quad \text{for } n > 0 \end{aligned}$$

Telescoping

A smoothness rule allows us to assume that n is a power of 2.

The recursive equation was:

$$\boxed{C(n) = C(n/2) + 1 \text{ (for } n > 0)}$$

Use the fact $C(n/2) = C(n/4) + 1$ to expand, and keep going:

$$\begin{aligned} C(n) &= C(n/2) + 1 \\ &= [C(n/4) + 1] + 1 \\ &= [[C(n/8) + 1] + 1] + 1 \\ &\vdots \\ &= \underbrace{[[\ldots[[C(0) + 1] + 1] + \cdots + 1] + 1]}_{1+\log_2 n \text{ times}} \end{aligned}$$

Hence $C(n) = \Theta(\log n)$.

Logarithmic Functions Have Same Rate of Growth

In O-expressions we can just write "log" for any logarithmic function, no matter what its base is.

Asymptotically, all logarithmic behaviour is the same, since

$$\log_a x = (\log_a b)(\log_b x)$$

So, for example, if ln is the natural logarithm then

$$\begin{aligned} \log_2 n &= O(\ln n) \\ \ln n &= O(\log_2 n) \end{aligned}$$

Also note that since $\log n^c = c \cdot \log n$, we have, for all constants c,

$$\log n^c = O(\log n)$$

Summarising Reasoning with Big-Oh

$$\begin{aligned} O(f(n)) + O(g(n)) &= O(\max\{f(n), g(n)\}) \\ c \cdot O(f(n)) &= O(f(n)) \\ O(f(n)) \cdot O(g(n)) &= O(f(n) \cdot g(n)). \end{aligned}$$

The first equation justifies throwing smaller summands away.

The second says that constants can be thrown away too.

The third may be used with some nested loops. Suppose we have a loop which is executed $O(f(n))$ times, and each execution takes time $O(g(n))$. Then the execution of the loop takes time $O(f(n) \cdot g(n))$.

Some Useful Formulas

From Stirling's formula:

$$n! = O(n^{n+\frac{1}{2}})$$

Some useful sums:

$$\begin{aligned}
\sum_{i=0}^{n} i^2 &= \frac{n}{3}(n+\tfrac{1}{2})(n+1) \\
\sum_{i=0}^{n}(2i+1) &= (n+1)^2 \\
\sum_{i=1}^{n} 1/i &= O(\log n)
\end{aligned}$$

See also Levitin's Appendix A.

Levitin's Appendix B is a tutorial on recurrence relations.

The Road Ahead

You will become much more familiar with asymptotic analysis as we use it on algorithms that we meet.

We shall begin the study of algorithms by looking at brute force approaches.

COMP20007 Design of Algorithms

Brute Force Methods

Lars Kulik

Lecture 5

Semester 1, 2020

Brute Force Algorithms

Straightforward problem solving approach, usually based directly on the problem's statement.

Exhaustive search for solutions is a prime example.

- Selection sort
- String matching
- Closest pair
- Exhaustive search for combinatorial solutions
- Graph traversal

Example: Selection Sort

We already saw this algorithm:

function $\textsc{SelSort}(A[0..n-1])$
  for $i \leftarrow 0$ to $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ to $n-1$ **do**
      if $A[j] < A[min]$ **then**
        $min \leftarrow j$
    swap $A[i]$ and $A[min]$

The complexity is $\Theta(n^2)$.

We shall soon meet better sorting algorithms.

Properties of Sorting Algorithms

A sorting algorithm is

- in-place if it does not require additional memory except, perhaps, for a few units of memory.
- stable if it preserves the relative order of elements that have identical keys.
- input-insensitive if its running time is fairly independent of input properties other than size.

Properties of Selection Sort

While running time is quadratic, selection sort makes only about n exchanges.

So: A good algorithm for sorting small collections of large records.

In-place?

Stable?

Input-insensitive?

✎

Brute Force String Matching

Pattern p: A string of m characters to search for.

Text t: A long string of n characters to search in.

for $i \leftarrow 0$ to $n - m$ **do**
  $j \leftarrow 0$
  while $j < m$ and $p[j] = t[i + j]$ **do**
    $j \leftarrow j + 1$
  if $j = m$ **then**
    return i
return -1

Analysing Brute Force String Matching

For each of $n - m + 1$ positions in t, we make up to m comparisons.

Assuming n is much larger than m, this means $O(mn)$ comparisons.

However, for random text over a reasonably large alphabet (as in English), the average running time is linear in n.

There are better algorithms, in particular for smaller alphabets such as binary strings or strings of DNA nucleobases.

But for many purposes, the brute-force algorithm is acceptable.

Later we shall see more sophisticated string search.

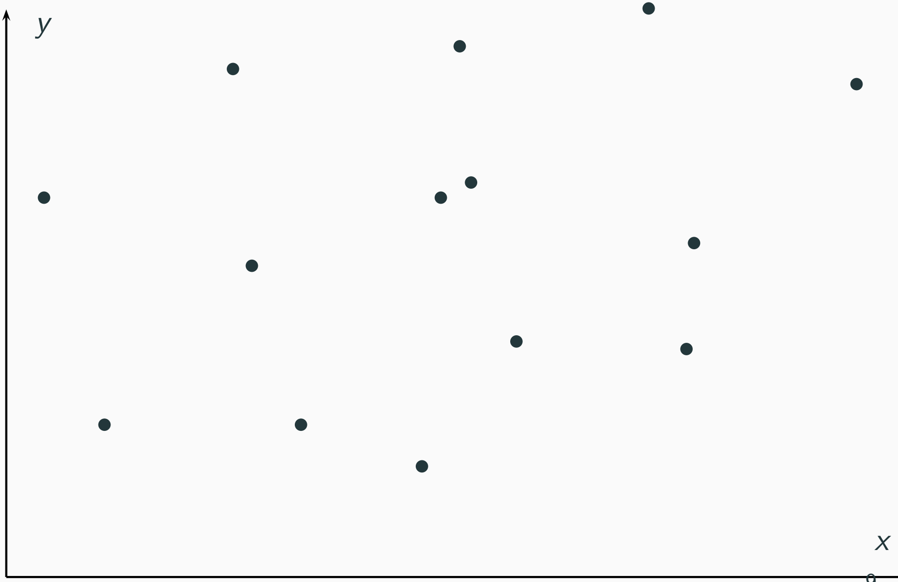
Brute Force Geometric Algorithms: Closest Pair

Problem: Given n points is k-dimensional space, find a pair of points with minimal separating Euclidean distance.

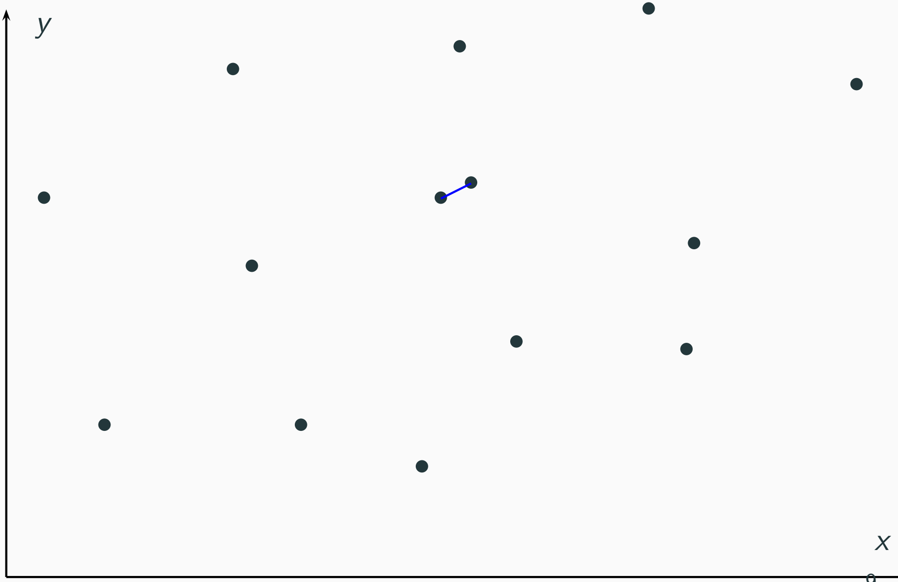
The brute force approach considers each pair in turn (except that once it has found the distance from x to y, it does not need to consider the distance from y to x).

For simplicity, we look at the 2-dimensional case, the points being $(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})$.

The Closest Pair Problem (Two-Dimensional Case)



The Closest Pair Problem (Two-Dimensional Case)



Brute Force Geometric Algorithms: Closest Pair

$min \leftarrow \infty$
for $i \leftarrow 0$ to $n-2$ **do**
  for $j \leftarrow i+1$ to $n-1$ **do**
    $d \leftarrow sqrt((x_i - x_j)^2 + (y_i - y_j)^2)$
    if $d < min$ **then**
      $min \leftarrow d$
      $p_1 \leftarrow i$
      $p_2 \leftarrow j$
return p_1, p_2

Analysing the Closest Pair Algorithm

It is not hard to see that the algorithm is $\Theta(n^2)$.

Note, however, that we can speed up the algorithm considerably, by utilising the monotonicity of the square root function.

How?

Does this contradict the $\Theta(n^2)$ claim?

Later we shall see how a clever divide-and-conquer approach leads to a $\Theta(n \log n)$ algorithm for this problem.

Brute Force Summary

Simple, easy to program, widely applicable.

Standard approach for small tasks.

Reasonable algorithms for some problems.

But: Generally inefficient—does not scale well.

Use brute force for prototyping, or when it is known that input remains small.

Exhaustive Search

Problem type:

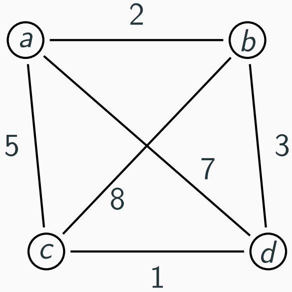
- Combinatorial decision or optimization problems
- Search for an element with a particular property
- Domain grows exponentially, for example all permutations

The brute-force approach—generate and test:

- Systematically construct all possible solutions
- Evaluate each, keeping track of the best so far
- When all potential solutions have been examined, return the best found

Example 1: Travelling Salesperson (TSP)

Find the shortest tour (visiting each node exactly once before returning to the start) in a weighted undirected graph.



$$
\begin{array}{lcl}
a-b-c-d-a & : & 18 \\
a-b-d-c-a & : & 11 \\
a-c-b-d-a & : & 23 \\
a-c-d-b-a & : & 11 \\
a-d-b-c-a & : & 23 \\
a-d-c-b-a & : & 18
\end{array}
$$

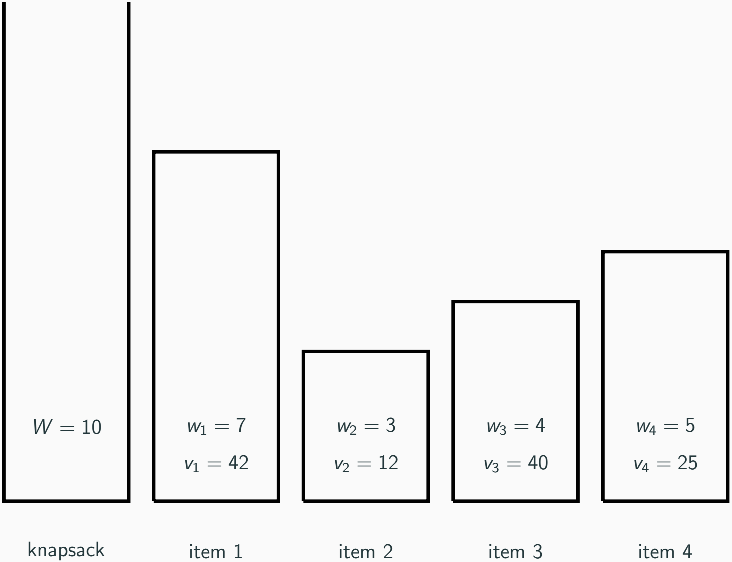
Example 2: Knapsack

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

Example 2: Knapsack



Example 2: Knapsack

Set	Weight	Value
$\emptyset$	0	0
$\{1\}$	7	42
$\{2\}$	3	12
$\{3\}$	4	40
$\{4\}$	5	25
$\{1,2\}$	10	54
$\{1,3\}$	11	NF
$\{1,4\}$	12	NF

Set	Weight	Value
$\{2,3\}$	7	52
$\{2,4\}$	8	37
$\{3,4\}$	9	65
$\{1,2,3\}$	14	NF
$\{1,2,4\}$	15	NF
$\{1,3,4\}$	16	NF
$\{2,3,4\}$	12	NF
$\{1,2,3,4\}$	19	NF

NF means "not feasible": exhausts the capacity of the knapsack.

Later we shall consider a better algorithm based on dynamic programming.

Comments on Exhaustive Search

Exhaustive search algorithms have acceptable running times only for very small instances.

In many cases there are better alternatives, for example, Eulerian tours, shortest paths, minimum spanning trees, ...

For some problems, it is known that there is essentially no better alternative.

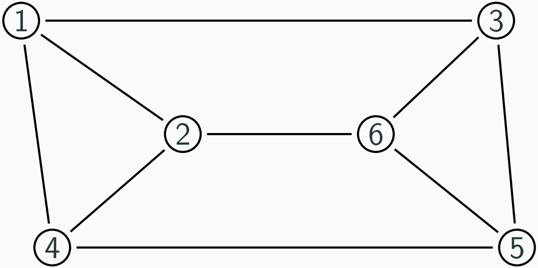
For a large class of important problems, it appears that there is no better alternative, but we have no proof either way.

Hamiltonian Tours

The Hamiltonian tour problem is this:

In a given undirected graph, is there a simple tour (a path that visits each node exactly once, except it returns to the starting node)?

Is there a Hamiltonian tour of this graph?

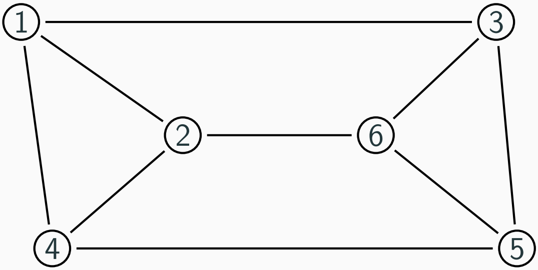


Eulerian Tours

The Eulerian tour problem is this:

In a given undirected graph, is there a path which visits each edge exactly once?

Is there a Eulerian tour of this graph?



Hard and Easy Problems

Recall that by a problem we usually mean a parametric problem: an infinite family of problem "instances".

The Hamiltonian Tour problem and the Eulerian Tour problem look very similar, but one is hard and the other is easy. We will see more examples of this phenomenon later.

For many optimization problems we do not know of solutions that are essentially better than exhaustive search (a whole raft of NP-complete problems, including TSP and knapsack).

In those cases we try to find approximation algorithms that are fast and close to the optimal solution.

We return to this idea later.

Next Up

Graphs.

COMP20007 Design of Algorithms

Graphs and Graph Concepts

Lars Kulik

Lecture 6

Semester 1, 2020

Graphs Again

One instance of the exhaustive search paradigm is graph traversal.

After this lecture we shall look at two ways of systematically visiting every node of a graph, namely depth-first and breadth-first search.

These two methods of graph traversal form the backbone of a surprisingly large number of useful graph algorithms.

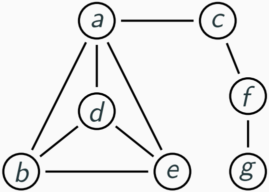
Graph algorithms are useful for a large number of practical problems: network design, flow design, planning, scheduling, route finding, and other logistics applications.

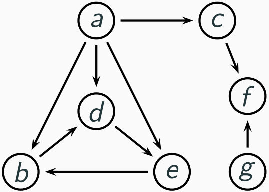
Graphs, Mathematically

$G = \langle V, E \rangle$

V: Set of nodes or vertices

E: Set of edges (a binary relation on V)



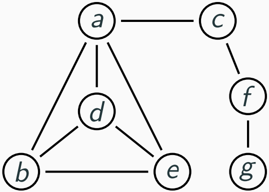


$$\begin{aligned} V &= \{a, b, c, d, e, f, g\} \\ E &= \text{symmetric closure of} \\ &\quad \{(a,b),(a,c),(a,d), \\ &\quad (a,e),(b,d),(b,e), \\ &\quad (c,f),(d,e),(f,g)\} \end{aligned}$$

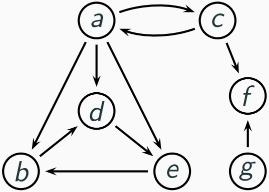
$$\begin{aligned} V &= \{a, b, c, d, e, f, g\} \\ E &= \{(a,b),(a,c),(a,d), \\ &\quad (a,e),(b,d),(c,f), \\ &\quad (d,e),(e,b),(g,f)\} \end{aligned}$$

Graph Concepts

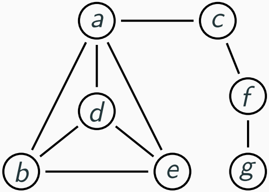
Undirected:



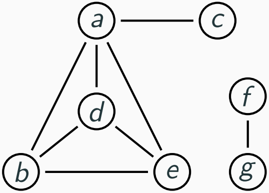
Directed:



Connected:



Not connected, two components:



More Graph Concepts: Degrees of Nodes

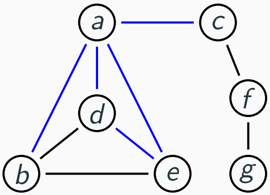
If $(u, v) \in E$ then u and v are adjacent, or neighbours.

(u, v) is connects u and v, and are u and v incident to (u, v).

The degree of node v is the number of edges incident to v.

For directed graphs, we talk about v's in-degree (number of edges going to v) and its out-degree (number of edges going from v).

More Graph Concepts: Paths and Cycles



Path b, a, d, e, a, c shown in blue

A path in $\langle V, E\rangle$ is a sequence of nodes $v_0, v_1, \ldots, v_k$ from V, so that $(v_i, v_{i+1}) \in E$.

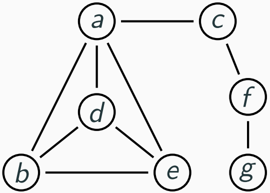
The path $v_0, v_1, \ldots, v_k$ has length k.

A simple path is one that has no repeated nodes.

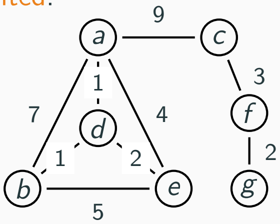
A cycle is a simple path, except that $v_0 = v_k$, that is, the last node is the same as the first node.

More Graph Concepts

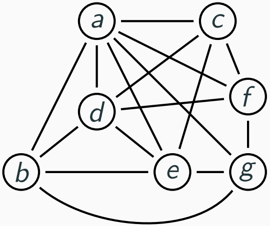
Unweighted:



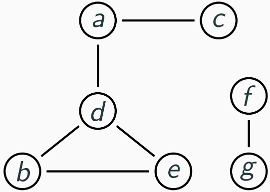
Weighted:



Dense:

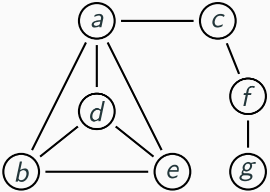


Sparse:

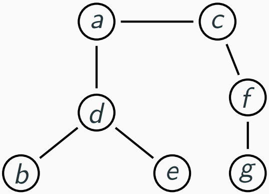


More Graph Concepts

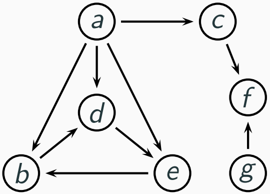
Cyclic:



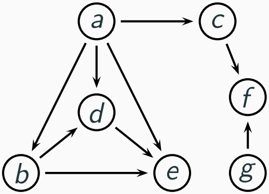
Acyclic (actually, a tree):



Directed cyclic:



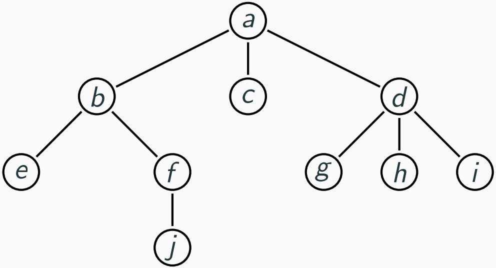
Directed acyclic (a dag):



Rooted Trees

A (free) tree is a connected acyclic graph.

A rooted tree is a tree with one node identified as special. Every other node is reachable from the root node.



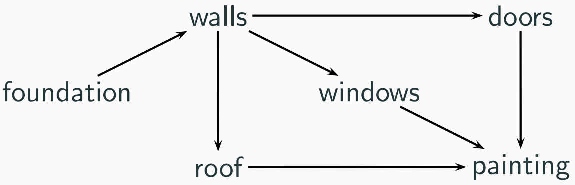
When the root is removed, a set of rooted (sub-)trees remain.

We should draw the rooted tree as a directed graph, but usually we instead rely on the layout: “parents” sit higher than “children”.

Modelling with Graphs

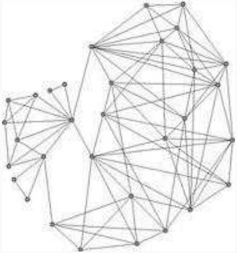
Graph algorithms are of great importance because so many different problem types can be abstracted to graph problems.

For example, directed graphs are central in scheduling problems:



Modelling with Graphs

Graphs find use in all sorts of modelling.



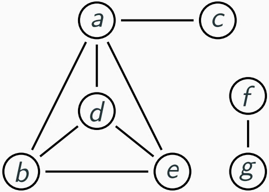
Assume you want to invite friends to dinner and you have k tables available.

Some guests dislike some of the others; thus we need a seating plan that avoids placing foes at the same table.

The natural model is an undirected graph, with a node for each guest, and an edge between any two guests that don't get along.

This reduces your problem to the "graph k-colouring problem": find, if possible, a colouring of nodes so that all connected nodes have a different colour.

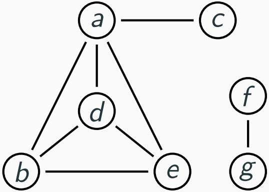
Graph Representations, Undirected Graphs



	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	1	0	0	1	1	0	0
c	1	0	0	0	0	0	0
d	1	1	0	0	1	0	0
e	1	1	0	1	0	0	0
f	0	0	0	0	0	0	1
g	0	0	0	0	0	1	0

The adjacency matrix for the graph.

Graph Representations, Undirected Graphs

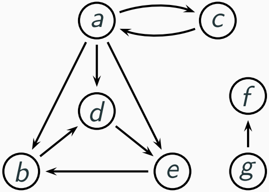


a	$\to b \to c \to d \to e$
b	$\to a \to d \to e$
c	$\to a$
d	$\to a \to b \to e$
e	$\to a \to b \to d$
f	$\to g$
g	$\to f$

The adjacency list representation.

(Assuming lists are kept in sorted order.)

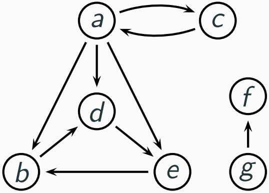
Graph Representations, Directed Graphs



	a	b	c	d	e	f	g
a	0	1	1	1	1	0	0
b	0	0	0	1	0	0	0
c	1	0	0	0	0	0	0
d	0	0	0	0	1	0	0
e	0	1	0	0	0	0	0
f	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0

The adjacency matrix for the graph.

Graph Representations, Directed Graphs



$a \to b \to c \to d \to e$

$b \to d$

$c \to a$

$d \to e$

$e \to b$

f

$g \to f$

The adjacency list representation.

Graph Representations

Each representation has advantages and disadvantages.

Think of some! ✍

Up Next

Graph traversal, in which we get down to the nitty-gritty details of graph algorithms.

COMP20007 Design of Algorithms

Graph Traversal

Lars Kulik

Lecture 7

Semester 1, 2020

Breadth-First and Depth-First Traversal

There are two natural approaches to the traversal of a graph.

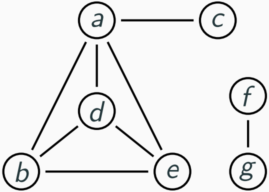
Suppose we have a graph and we want to explore all its nodes systematically. Suppose we start from node v and v has neighbouring nodes x, y and z.

In a breadth-first approach we, roughly, explore x, y and z before exploring any of their neighboring nodes.

In a depth-first approach, we may explore, say, x first, but then, before exploring y and z, we first explore one of x's neighbours, then one of its neighbours, and so on.

(This is really hard to express in English—we do need pseudo-code!)

Depth-First Search



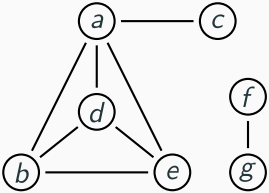
Both graph traversal methods rely on marking nodes as they are visited—so that we can avoid revisiting nodes.

Depth-first search is based on backtracking.

Neighbouring nodes are considered in, say, alphabetical order.

For the example graph, nodes are visited in the order a, b, d, e, c, f, g.

Depth-First Search: The Traversal Stack

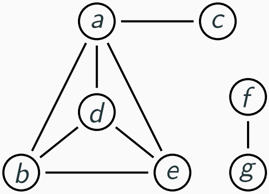


DFS corresponds to using a stack discipline for keeping track of where we are in the overall process.

Here is how the "where-we-came-from" stack develops for the example:

			e									
		d	d	d								
	b	b	b	b	b		c				g	
a	a	a	a	a	a	a	a	a		f	f	f
—	—	—	—	—	—	—	—	—	—	—	—	—

Depth-First Search: The Traversal Stack

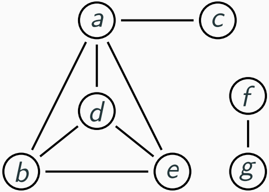


Levitin uses a more compact notation for the stack's history. Here is how the stack develops, in Levitin's notation:

$$\begin{array}{lll} e_{4,1} & & \\ d_{3,2} & & \\ b_{2,3} & c_{5,4} & g_{7,6} \\ a_{1,5} & & f_{6,7} \end{array}$$

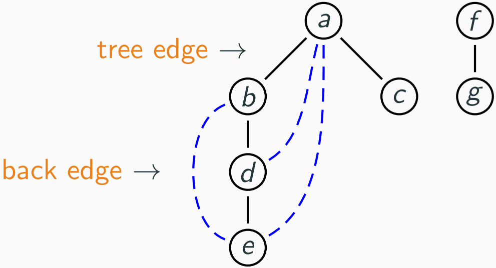
The first subscripts give the order in which nodes are pushed, the second the order in which they are popped off the stack.

Depth-First Search: The Depth-First Search Forest



Another useful tool for depicting a DF traversal is the DFS tree (for a connected graph).

More generally, we get a DFS forest:



Depth-First Search: The Algorithm

function $\mathrm{DFS}(\langle V, E \rangle)$
  mark each node in V with 0
  $count \leftarrow 0$
  for each v in V **do**
    if v is marked 0 **then**
      $\textsc{DfsExplore}(v)$

function $\textsc{DfsExplore}(v)$
  $count \leftarrow count + 1$
  mark v with *count*
  for each edge (v, w) **do** $\triangleright$ w is v's neighbour
    if w is marked with 0 **then**
      $\textsc{DfsExplore}(w)$

This works both for directed and undirected graphs.

Depth-First Search: The Algorithm

The "marking" of nodes is usually done by maintaining a separate array, `mark`, indexed by V.

For example, when we wrote "mark v with *count*", that would be implemented as "`mark[v] := count`".

How to find the nodes adjacent to v depends on the graph representation used.

Using an adjacency matrix `adj`, we need to consider `adj[v,w]` for each `w` in V. Here the complexity of graph traversal is $\Theta(|V|^2)$.

Using adjacency lists, for each v, we traverse the list `adj[v]`. In this case, the complexity of traversal is $\Theta(|V| + |E|)$. Why? ✎

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✍

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

How? ✎

It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

Applications of Depth-First Search

It is easy to adapt the DFS algorithm so that it can decide whether a graph is connected.

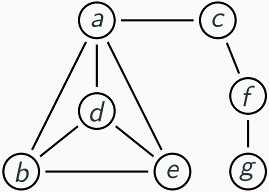
How? ✎

It is also easy to adapt it so that it can decide whether a graph has a cycle.

How? ✎

In terms of DFS forests, how can we tell if we have traversed a dag?

Breadth-First Search



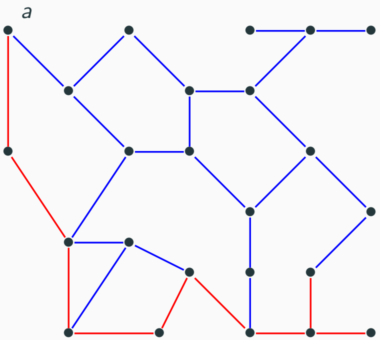
Breadth-first search proceeds in a concentric manner, visiting all nodes that are one step away from the start node, then all those that are two steps away (except those that were already visited), and so on.

Again, neighbouring nodes are considered in, say, alphabetical order.

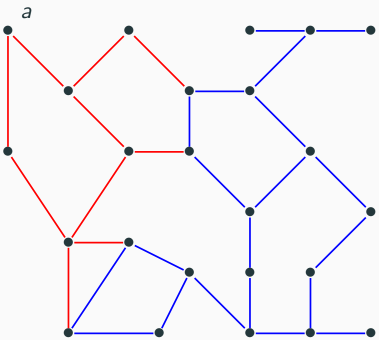
For the example graph, nodes are visited in the order a, b, c, d, e, f, g.

Depth-First Search vs Breadth-First

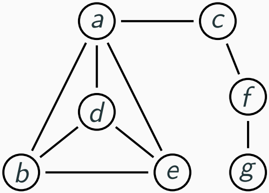
Typical depth-first search:



Typical breadth-first search:



Breadth-First Search: The Traversal Queue



BFS uses a queue discipline for keeping track of pending tasks.

How the queue develops for the example:

a_1

$b_2 \quad c_3 \quad d_4 \quad e_5$

$c_3 \quad d_4 \quad e_5$

$d_4 \quad e_5 \quad f_6$

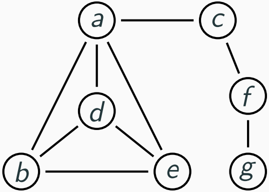
$e_5 \quad f_6$

f_6

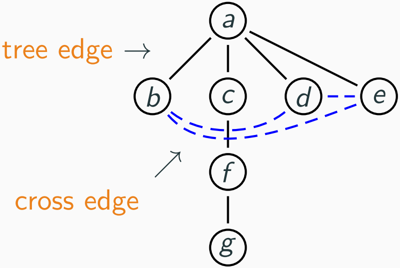
g_7

The subscript again is Levitin's; it gives the order in which nodes are processed.

The Breadth-First Search Forest



Here is the BFS tree for the example:



In general, we may get a BFS forest.

Breadth-First Search: The Algorithm

```
function BFS(⟨V, E⟩)
    mark each node in V with 0
    count ← 0, init(queue)                          ▷ create an empty queue
    for each v in V do
        if v is marked 0 then
            count ← count + 1
            mark v with count
            inject(queue, v)                        ▷ queue containing just v
            while queue is non-empty do
                u ← eject(queue)                    ▷ dequeues u
                for each edge (u, w) adjacent to u do
                    if w is marked with 0 then
                        count ← count + 1
                        mark w with count
                        inject(queue, w)            ▷ enqueues w
```

Breadth-First Search: The Algorithm

BFS has the same complexity as DFS.

Again, the same algorithm works for directed graphs as well.

Certain problems are most easily solved by adapting BFS.

For example, given a graph and two nodes, *a* and *b* in the graph, how would you find the fewest number of edges between two given vertices *a* and *b*?

✎

Topological Sorting

We mentioned scheduling problems and their representation by directed graphs.

Assume a directed edge from a to b means that task a must be completed before b can be started.

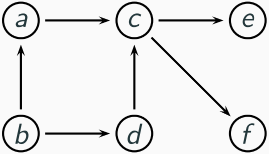
Then the graph has to be a dag.

Assume the tasks are carried out by a single person, unable to multi-task.

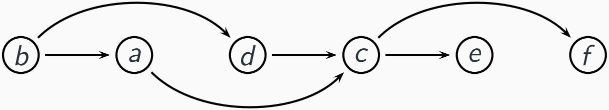
Then we should try to linearize the graph, that is, order the nodes in a sequence $v_1, v_2, \ldots, v_n$ such that for each edge $(v_i, v_j) \in E$, we have $i < j$.

Topological Sorting: Example

There are four different ways to linearize the following graph.



Here is one:



Topological Sorting Algorithm 1

We can solve the top-sort problem with depth-first search:

1. Perform DFS and note the order in which nodes are popped off the stack.
2. List the nodes in the reverse of that order.

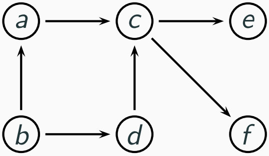
This works because of the stack discipline.

If (u, v) is an edge then it is possible (for some way of deciding ties) to arrive at a DFS stack with u sitting below v.

Taking the "reverse popping order" ensures that u is listed before v.

Topological Sorting Example Again

Using the DFS method and resolving ties by using alphabetical order, the graph gives rise to the traversal stack shown on the right (the popping order shown in red):



$e_{3,1}$	$f_{4,2}$	
$c_{2,3}$		$d_{6,5}$
$a_{1,4}$		$b_{5,6}$

Taking the nodes in reverse popping order yields b, d, a, c, f, e.

Topological Sorting Algorithm 2

An alternative method would be to repeatedly select a random source in the graph (that is, a node with no incoming edges), list it, and remove it from the graph.

This is a very natural approach, but it has the drawback that we repeatedly need to scan the graph for a source.

However, it exemplifies the general principle of decrease-and-conquer.

COMP20007 Design of Algorithms

Greedy Algorithms: Prim and Dijkstra

Lars Kulik

Lecture 8

Semester 1, 2020

Greedy Algorithms

A natural strategy to problem solving is to make decisions based on what is the locally best choice.

Suppose we have coin denominations 25, 10, 5, and 1, and we want to change 30 cents using the smallest number of coins.

In general we will want to use as many 25-cent pieces as we can, then do the same for 10-cent pieces, and so on, until we have reached 30 cents. (In this case we use 25+5 cents.)

This greedy strategy will work for the given denominations, but not for, say, 25, 10, 1.

Greedy Algorithms

In general we cannot expect locally best choices to yield globally best outcomes.

However, there are some well-known algorithms that rely on the greedy approach, being both correct and fast.

In other cases, for hard problems, a greedy algorithm can sometimes serve as an acceptable approximation algorithm.

Here we shall look at

- Prim's algorithm for finding minimum spanning trees
- Dijkstra's algorithm for single-source shortest paths

The Priority Queue

A priority queue is a set (or pool) of elements.

An element is injected into the priority queue together with a priority (often the key value itself) and elements are ejected according to priority.

As an abstract data type, the priority queue supports the following operations on a “pool” of elements (ordered by some linear order):

- find an item with maximal priority
- insert a new item with associated priority
- test whether a priority queue is empty
- eject the largest element

Stacks and Queues as Priority Queues

Special instances are obtained when we use time for priority:

- If "large" means "late" we obtain the stack.
- If "large" means "early" we obtain the queue.

Possible Implementations of the Priority Queue

Assume priority = key.

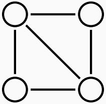
	INJECT(e)	EJECT()
Unsorted array or list		
Sorted array or list		

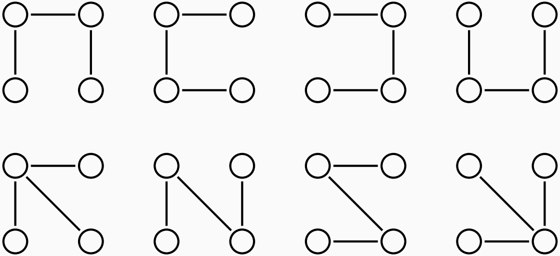
✎

Spanning Trees

Recall that a tree is a connected graph with no cycle.

A spanning tree of a graph $\langle V, E \rangle$ is a tree $\langle V, E' \rangle$ with $E' \subseteq E$.

The graph  has eight different spanning trees:

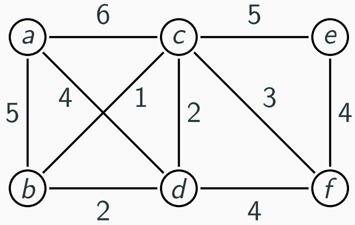


Minimum Spanning Trees of Weighted Graphs

In applications where the edges correspond to distances, or cost, some spanning trees will be more desirable than others.

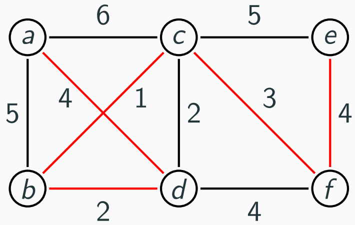
Suppose we have a set of 'stations' to connect in a network, and also some possible connections, together with the cost of each connection.

Then we have a weighted graph problem, of finding a spanning tree with the smallest possible cost.



Minimum Spanning Trees

Given a weighted graph, a sub-graph which is a tree with minimal weight is a minimum spanning tree for the graph.



Minimum Spanning Trees: Prim's Algorithm

Prim's algorithm is an example of a greedy algorithm.

It constructs a sequence of subtrees T, each adding a node together with an edge to a node in the previous subtree. In each step it picks a closest node from outside the tree and adds that. A sketch:

function $\textsc{Prim}(\langle V, E \rangle)$
    $V_T \leftarrow \{v_0\}$
    $E_T \leftarrow \emptyset$
    for $i \leftarrow 1$ to $|V| - 1$ **do**
        find a minimum-weight edge $(v, u) \in V_T \times (V \setminus V_T)$
        $V_T \leftarrow V_T \cup \{u\}$
        $E_T \leftarrow E_T \cup \{(v, u)\}$
    return E_T

Prim's Algorithm

Note that in each iteration, the tree grows by one edge.

Or, we can say that the tree grows to include the node from outside that has the smallest cost.

But how do we find the minimum-weight edge (v, u)?

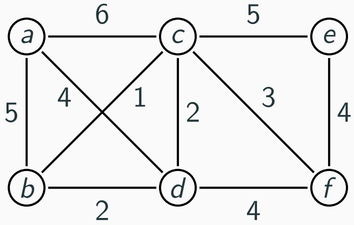
A standard way to do this is to organise the nodes that are not yet included in the spanning tree T as a priority queue organised by edge cost.

The information about which nodes are connected in T can be captured by an array *prev* of nodes, indexed by V. Namely, when (v, u) is included, this is captured by setting $prev[u] = v$.

Prim's Algorithm

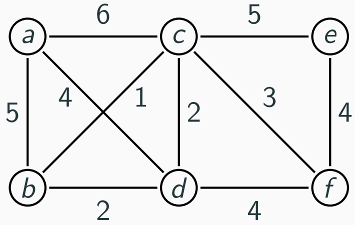
function $\textsc{Prim}(\langle V, E\rangle)$
  for each $v \in V$ **do**
    $cost[v] \leftarrow \infty$
    $prev[v] \leftarrow nil$
  pick initial node v_0
  $cost[v_0] \leftarrow 0$
  $Q \leftarrow \textsc{InitPriorityQueue}(V)$ ▷ priorities are cost values
  while Q is non-empty **do**
    $u \leftarrow \textsc{EjectMin}(Q)$
    for each $(u, w) \in E$ **do**
      if $w \in Q$ and $weight(u, w) < cost[w]$ **then**
        $cost[w] \leftarrow weight(u, w)$
        $prev[w] \leftarrow u$
        $\textsc{Update}(Q, w, cost[w])$ ▷ rearranges priority queue

Prim's Algorithm: Example



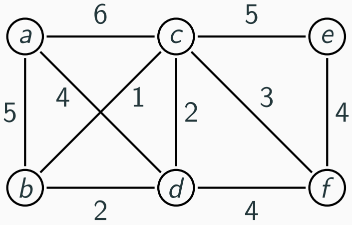
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil

Prim's Algorithm: Example



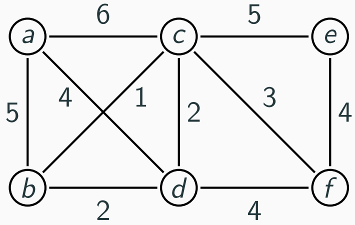
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil

Prim's Algorithm: Example



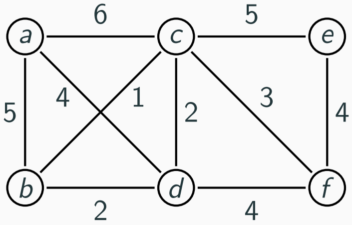
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$

Prim's Algorithm: Example



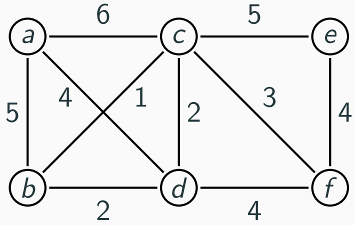
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$

Prim's Algorithm: Example



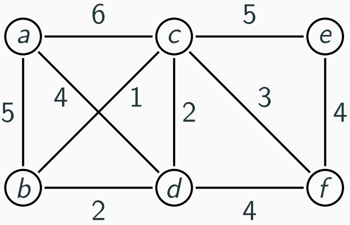
Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	

Prim's Algorithm: Example



Tree T	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		$5/a$	$6/a$	$4/a$	∞/nil	∞/nil
a, d		$2/d$	$2/d$		∞/nil	$4/d$
a, d, b			$1/b$		∞/nil	$4/d$
a, d, b, c					$5/c$	$3/c$
a, d, b, c, f					$4/f$	
a, d, b, c, f, e						

Analysis of Prim's Algorithm

First, a crude analysis: For each node, we look through the edges to find those incident to the node, and pick the one with smallest cost. Thus we get $O(|V| \cdot |E|)$. However, we are using cleverer data structures.

Using adjacency lists for the graph and a min-heap for the priority queue, we can do better! We will discuss this later.

Dijkstra's Algorithm

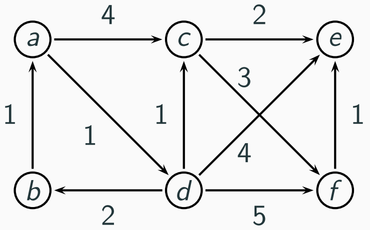
Another classical greedy weighted-graph algorithm is Dijkstra's algorithm, whose overall structure is the same as Prim's.

Dijkstra's algorithm is also a shortest-path algorithm for (directed or undirected) weighted graphs. It finds all shortest paths from a fixed start node. Its complexity is the same as that of Prim's algorithm.

Dijkstra's Algorithm

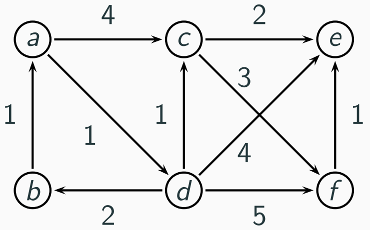
$$
\begin{array}{l}
\textbf{function } \textsc{Dijkstra}(\langle V, E\rangle, v_0) \\
\quad \textbf{for } \text{each } v \in V \textbf{ do} \\
\quad\quad \mathit{dist}[v] \leftarrow \infty \\
\quad\quad \mathit{prev}[v] \leftarrow \mathit{nil} \\
\quad \mathit{dist}[v_0] \leftarrow 0 \\
\quad Q \leftarrow \textsc{InitPriorityQueue}(V) \qquad\qquad \triangleright \text{ priorities are distances} \\
\quad \textbf{while } Q \text{ is non-empty } \textbf{do} \\
\quad\quad u \leftarrow \textsc{EjectMin}(Q) \\
\quad\quad \textbf{for } \text{each } (u, w) \in E \textbf{ do} \\
\quad\quad\quad \textbf{if } w \in Q \text{ and } \mathit{dist}[u] + \mathit{weight}(u, w) < \mathit{dist}[w] \textbf{ then} \\
\quad\quad\quad\quad \mathit{dist}[w] \leftarrow \mathit{dist}[u] + \mathit{weight}(u, w) \\
\quad\quad\quad\quad \mathit{prev}[w] \leftarrow u \\
\quad\quad\quad\quad \textsc{Update}(Q, w, \mathit{dist}[w]) \qquad \triangleright \text{ rearranges priority queue}
\end{array}
$$

Dijkstra's Algorithm: Example



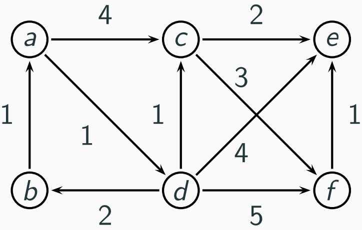
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil

Dijkstra's Algorithm: Example



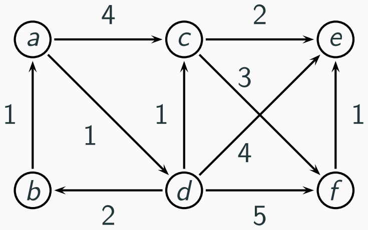
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil

Dijkstra's Algorithm: Example



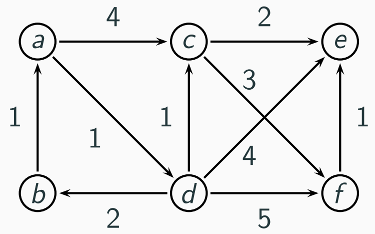
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$

Dijkstra's Algorithm: Example



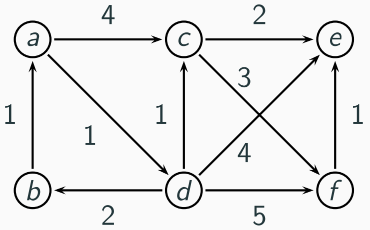
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$

Dijkstra's Algorithm: Example



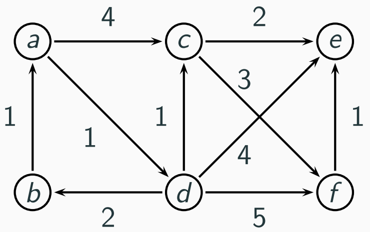
Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$

Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$

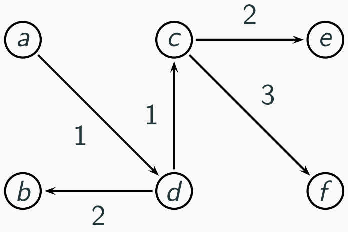
Dijkstra's Algorithm: Example



Covered	a	b	c	d	e	f
—	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
a		∞/nil	$4/a$	$1/a$	∞/nil	∞/nil
a, d		$3/d$	$2/d$		$5/d$	$6/d$
a, d, c		$3/d$			$4/c$	$5/c$
a, d, c, b					$4/c$	$5/c$
a, d, c, b, e						$5/c$
a, d, c, b, e, f						

Dijkstra's Algorithm: Tracing Paths

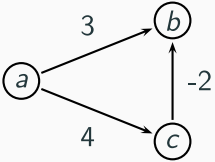
The array `prev` is not really needed, unless we want to retrace the shortest paths from node *a*:



Negative Weights

In our example, we used positive weights, and for a good reason: Dijkstra's algorithm may not work otherwise!

In this example, the greedy pick—choosing the edge from a to b—is clearly the wrong one.



COMP20007 Design of Algorithms

Divide-and-Conquer Algorithms

Lars Kulik

Lecture 9

Semester 1, 2020

Divide and Conquer

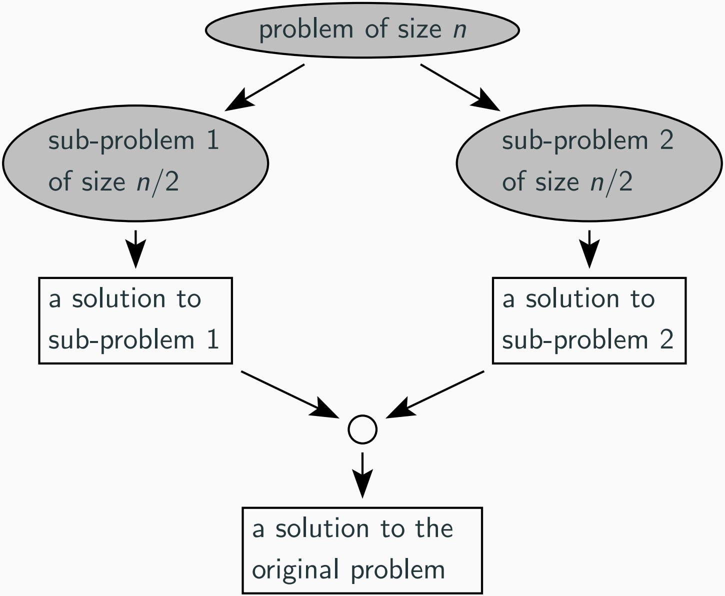
We earlier saw recursion as a powerful problem solving technique.

The divide-and-conquer strategy tries to make the most of this:

1. Divide the given problem instance into smaller instances.
2. Solve the smaller instances recursively.
3. Combine the smaller solutions to solve the original instance.

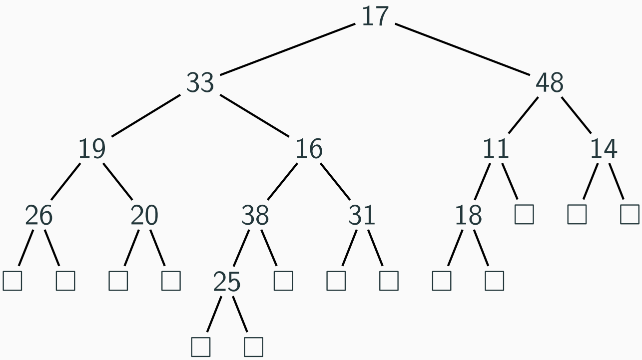
This works best when the smaller instances can be made to be of equal size.

Split-Solve-and-Join Approach



Binary Trees Again

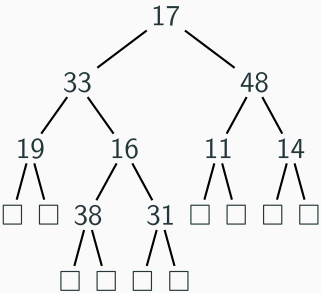
An example of a binary tree, with empty subtrees marked with □:



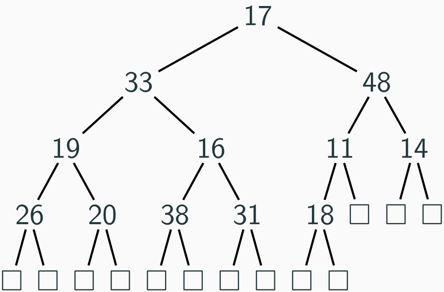
This tree has height 4, the empty tree having height -1.

Binary Tree Concepts

Special trees have their external nodes □ only at level h and $h + 1$ for some h:



A full binary tree: Each node has 0 or 2 children.



A complete tree: Each level filled left to right.

Binary Tree Concepts

A non-empty tree T has a root T_{root}, a left subtree T_{left}, and a right subtree T_{right}.

Recursion is the natural way of calculating the height:

function $\textsc{Height}(T)$
  if T is empty **then**
    return -1
  else
    return $max(\textsc{Height}(T_{left}), \textsc{Height}(T_{right})) + 1$

It is not hard to prove that the number x of external nodes $\square$ is always one greater than the number n of internal nodes.

The function $\textsc{Height}$ makes a tree comparison (empty or non-empty?) per node (internal and external), so altogether $2n+1$ comparisons.

Binary Tree Traversal

Preorder traversal visits the root, then the left subtree, and finally the right subtree.

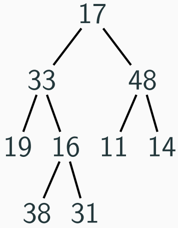
Inorder traversal visits the left subtree, then the root, and finally the right subtree.

Postorder traversal visits the left subtree, the right subtree, and finally the root.

Level-order traversal visits the nodes, level by level, starting from the root.

Binary Tree Traversal: Preorder

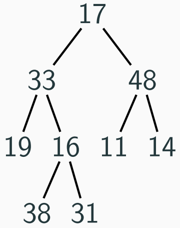
function $\textsc{PreorderTraverse}(T)$
  if T is non-empty **then**
    visit T_{root}
    $\textsc{PreorderTraverse}(T_{left})$
    $\textsc{PreorderTraverse}(T_{right})$



Visit order for the example: 17, 33, 19, 16, 38, 31, 48, 11, 14.

Binary Tree Traversal: Inorder

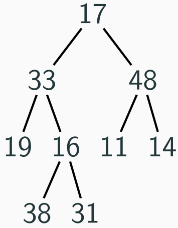
function INORDERTRAVERSE(T)
    if T is non-empty **then**
        INORDERTRAVERSE(T_{left})
        visit T_{root}
        INORDERTRAVERSE(T_{right})



Visit order for the example: 19, 33, 38, 16, 31, 17, 11, 48, 14.

Binary Tree Traversal: Postorder

function $\textsc{PostorderTraverse}(T)$
  if T is non-empty **then**
    $\textsc{PostorderTraverse}(T_{left})$
    $\textsc{PostorderTraverse}(T_{right})$
    visit T_{root}



Visit order for the example: 19, 38, 31, 16, 33, 11, 14, 48, 17.

Preorder Traversal Using a Stack

We could also implement preorder traversal of T by maintaining a stack explicitly.

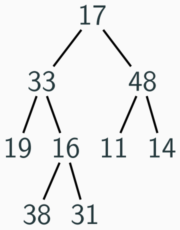
$push(T)$
while the stack is non-empty **do**
    $T \leftarrow pop$
    visit T_{root}
    if T_{right} is non-empty **then**
        $push(T_{right})$
    if T_{left} is non-empty **then**
        $push(T_{left})$

In an implementation, the elements placed on the stack would not be whole trees, but pointers to the corresponding internal nodes.

Tree Traversal Using a Queue: Level-Order

Level-order traversal results if we replace the stack with a queue.

inject(T)
while the queue is non-empty **do**
  $T \leftarrow$ *eject*
  visit T_{root}
  if T_{left} is non-empty **then**
    inject(T_{left})
  if T_{right} is non-empty **then**
    inject(T_{right})



Visit order for the example: 17, 33, 48, 19, 16, 11, 14, 38, 31.

The Closest Pair Problem Revisited

In Lecture 5 we gave a brute-force algorithm for the closest pair problem: Given n points in the Cartesian plane, find a pair with minimal distance.

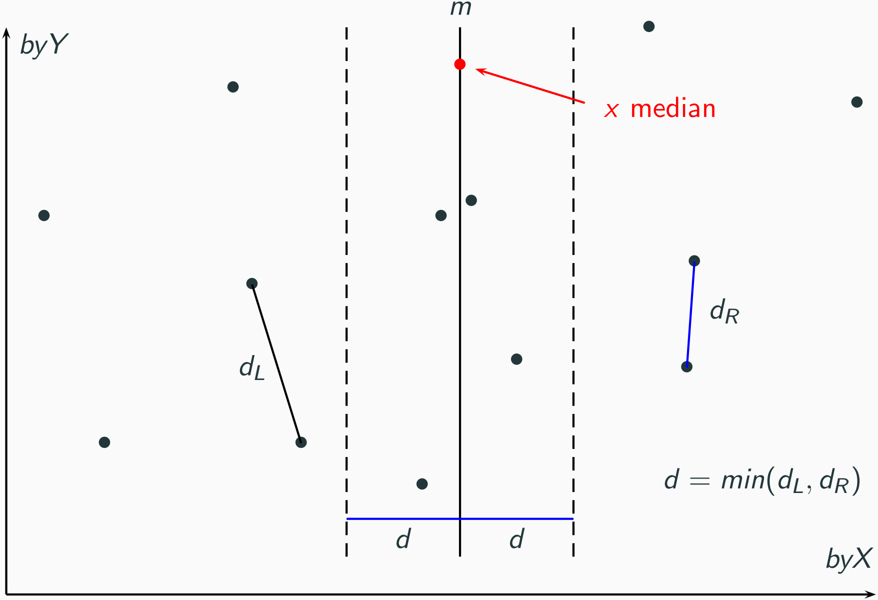
The brute-force method had complexity $\Theta(n^2)$. We can use divide-and-conquer to do better, namely $\Theta(n \log n)$.

First, sort the points by x value and store the result in array byX.

Also sort the points by y value and store the result in array byY.

Now we can identify the x median, and recursively process the set P_L of points with lower x values, as well as the set P_R with higher x values.

The Closest Pair Problem Revisited



The Closest Pair Problem Revisited

The recursive calls will identify d_L, the shortest distance for pairs in P_L, and d_R, the shortest distance for pairs in P_R.

Let m be the x median and let $d = min(d_L, d_R)$. This d is a candidate for the smallest distance.

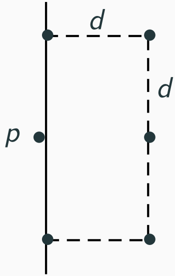
But d may not be the global minimum—there could be some close pair whose points are on opposite sides of the median line $x = m$.

For candidates that may improve on d we only need to look at those in the band $m - d \leq x \leq m + d$.

So pick out, from array byY, each point p with x-coordinate between $m - d$ and $m + d$, and keep these in array S.

For each point in S, consider just its "close" neighbours.

The following calculates the smallest distance and leaves the (square of the) result in *minsq*.



It can be shown that the while loop can execute at most 5 times for each i value—see diagram.

$minsq \leftarrow d^2$
copy all points of byY with $|x - m| < d$ to array S
$k \leftarrow |S|$
for $i \leftarrow 0$ to $k - 2$ **do**
    $j \leftarrow i + 1$
    while $j \leq k - 1$ and $(S[j].y - S[i].y)^2 < minsq$ **do**
        $minsq \leftarrow min(minsq, (S[j].x - S[i].x)^2 + (S[j].y - S[i].y)^2)$
        $j \leftarrow j + 1$

COMP20007 Design of Algorithms

Master Theorem

Lars Kulik

Lecture 10

Semester 1, 2020

Divide and Conquer

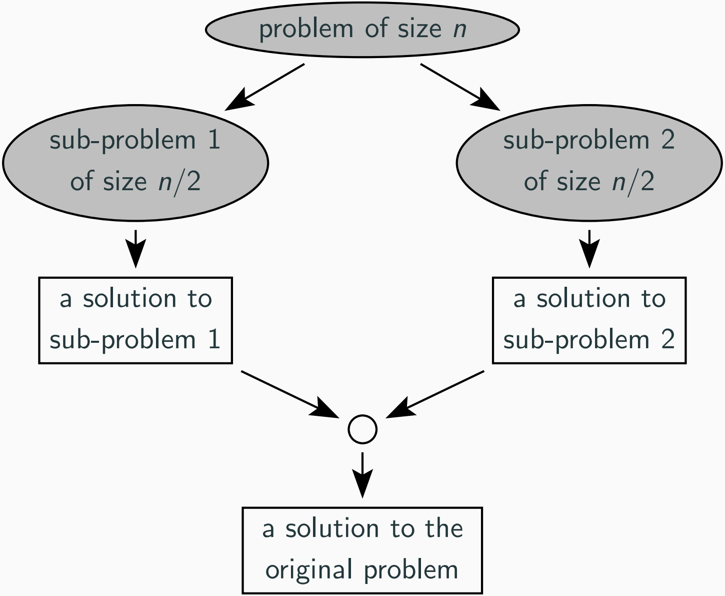
We earlier studied recursion as a powerful problem solving technique.

The divide-and-conquer strategy tries to make the most of this:

1. Divide the given problem instance into smaller instances.
2. Solve the smaller instances recursively.
3. Combine the smaller solutions to solve the original instance.

This works best when the smaller instances can be made to be of equal size.

Split-Solve-and-Join Approach



Divide-and-Conquer Algorithms

You have seen:

- Tree traversal
- Closest pair

You will learn later:

- Mergesort
- Quicksort

Divide-and-Conquer Recurrences

What is the time required to solve a problem of size n by divide-and-conquer?

For the general case, assume we split the problem into b instances (each of size n/b), of which a need to be solved:

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

where $f(n)$ expresses the time spent on dividing a problem into b sub-problems and combining the a results.

(A very common case is $T(n) = 2T(n/2) + n$.)

How do we find closed forms for these recurrences?

The Master Theorem

(A proof is in Levitin's Appendix B.)

For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d), d \geq 0$, the recurrence

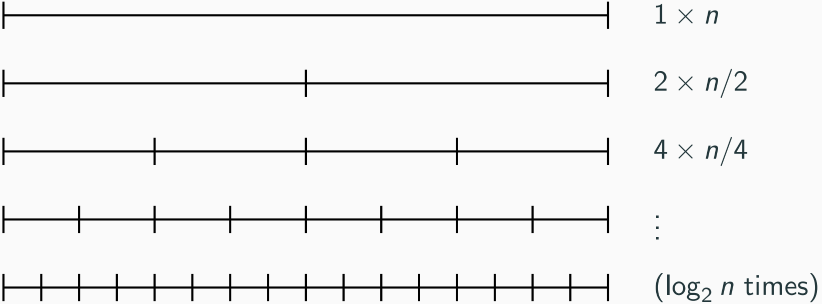
$$T(n) = aT(n/b) + f(n)$$

(with $T(1) = c$) has solutions, and

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

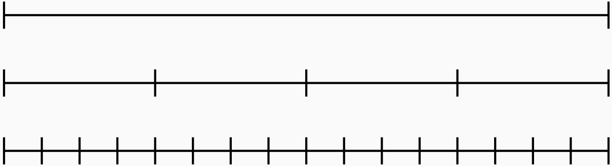
Note that we also allow a to be greater than b.

$$T(n) = 2T(n/2) + n \qquad a = 2, b = 2, d = 1$$



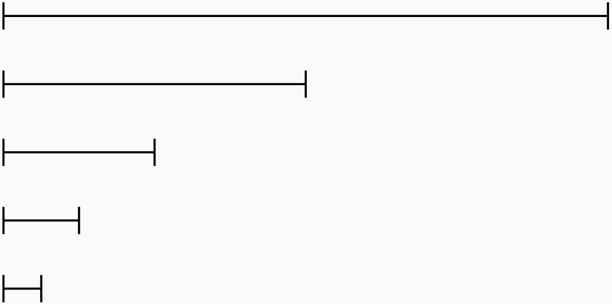
Master Theorem: Example 2

$$T(n) = 4T(n/4) + n \qquad\qquad a = 4, b = 4, d = 1$$



Master Theorem: Example 3

$$T(n) = T(n/2) + n \qquad a = 1, b = 2, d = 1$$



n

$n/2$

$n/4$

$n/8$

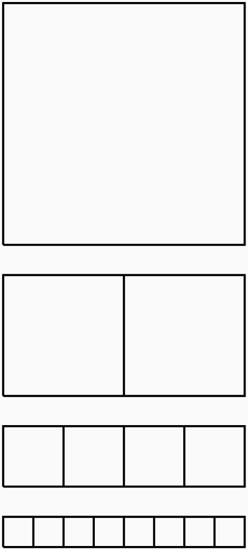
$\vdots$

Master Theorem: Example 4

$$T(n) = 2T(n/2) + n^2$$

$$a = 2, b = 2, d = 2$$

Here $a < b^d$ and we simply get n^d.



COMP20007 Design of Algorithms

Sorting - Part 1

Daniel Beck

Lecture 11

Semester 1, 2020

Insertion Sort

function $\textsc{InsertionSort}(A[0..n-1])$

  for $i \leftarrow 1$ **to** $n - 1$ **do**

    $j \leftarrow i - 1$

    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**

      $\text{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

Insertion Sort

$$
\begin{array}{l}
\textbf{function } \textsc{InsertionSort}(A[0..n-1]) \\
\quad \textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do} \\
\quad\quad j \leftarrow i-1 \\
\quad\quad \textbf{while } j \geq 0 \textbf{ and } A[j+1] < A[j] \textbf{ do} \\
\quad\quad\quad \text{SWAP}(A[j+1], A[j]) \\
\quad\quad\quad j \leftarrow j-1
\end{array}
$$

- Decrease-And-Conquer algorithm.

Insertion Sort

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

- Decrease-And-Conquer algorithm.
- The idea behind Insertion Sort is recursive, but the code given here, using iteration, is preferable to the recursive version.

Insertion Sort - Properties

Questions!

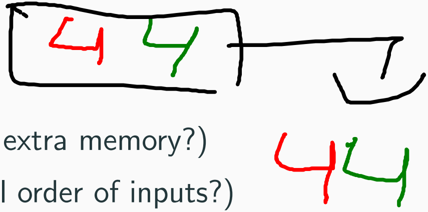
Insertion Sort - Properties

Questions!

- In-place? (does it require extra memory?)

Insertion Sort - Properties

Questions!



- In-place? (does it require extra memory?)
- Stable? (preserves original order of inputs?)

Insertion Sort - Properties

Questions!

- In-place? (does it require extra memory?)
- Stable? (preserves original order of inputs?)

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$

  for $i \leftarrow 1$ **to** $n-1$ **do**

    $j \leftarrow i - 1$

    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**

      $\mathsf{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$
for $i \leftarrow 1$ **to** $n-1$ **do**
$j \leftarrow i - 1$
while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
SWAP$(A[j+1], A[j])$
$j \leftarrow j - 1$

- In-place?

Insertion Sort - Properties

function InsertionSort($A[0..n-1]$)
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      SWAP($A[j+1], A[j]$) $\rightarrow$ $v \leftarrow A[j]$
      $j \leftarrow j - 1$

$A[j] \leftarrow A[j+1]$
$A[j+1] \leftarrow v$

- In-place? Yes! (may need additional O(1) memory)

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$
 for $i \leftarrow 1$ **to** $n-1$ **do**
  $j \leftarrow i-1$
  while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
   $\text{SWAP}(A[j+1], A[j])$
   $j \leftarrow j-1$

- In-place? Yes! (may need additional O(1) memory)
- Stable?

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$

  for $i \leftarrow 1$ **to** $n-1$ **do**

    $j \leftarrow i - 1$

    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**

      $\text{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

- In-place? Yes! (may need additional O(1) memory)
- Stable? Yes! (**local**, adjacent swaps ensure stability)

Insertion Sort - Properties

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\mathrm{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

- In-place? Yes! (may need additional O(1) memory)
- Stable? Yes! (**local**, adjacent swaps ensure stability)

Compare with Selection Sort:

- Also in-place.
- **Not stable.** (swaps are not **local**)

Insertion Sort - Complexity

function InsertionSort($A[0..n-1]$)
    for $i \leftarrow 1$ **to** $n-1$ **do**
        $j \leftarrow i-1$
        while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
            SWAP($A[j+1], A[j]$)
            $j \leftarrow j-1$

Insertion Sort - Complexity

function $\textsc{InsertionSort}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      $\text{SWAP}(A[j+1], A[j])$
      $j \leftarrow j-1$

- Worst case?
- Best case?

Insertion Sort - Worst case

function $\textsc{InsertionSort}(A[0..n-1])$

  $\boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}}$

    $j \leftarrow i - 1$

    while $j \geq 0$ **and** $\boxed{A[j+1] < A[j]}$ **do**

      $\text{SWAP}(A[j+1], A[j])$

      $j \leftarrow j - 1$

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2)$$

Insertion Sort - Best case

$$
\begin{array}{l}
\textbf{function } \textsc{InsertionSort}(A[0..n-1]) \\
\quad \boxed{\textbf{for } i \leftarrow 1 \textbf{ to } n-1 \textbf{ do}} \\
\qquad j \leftarrow i - 1 \\
\qquad \textbf{while } j \geq 0 \textbf{ and } \boxed{A[j+1] < A[j]} \textbf{ do} \\
\qquad\quad \text{SWAP}(A[j+1], A[j]) \\
\qquad\quad j \leftarrow j - 1
\end{array}
$$

$$\sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

Insertion Sort - Average case

$\Theta(n+v)$

v: # INV. PAIRS

function InsertionSort($A[0..n-1]$)
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $A[j+1] < A[j]$ **do**
      SWAP($A[j+1], A[j]$) → INV. PAIRS
      $j \leftarrow j - 1$

- BEST: 0 INV. PAIRS $= \boxed{\Theta(n)}$
- WORST: $\frac{n(n-1)}{2}$ INV. PAIRS $= \Theta\left(n + \frac{n(n-1)}{2}\right) = \Theta(n^2)$
- AVG: $\frac{n(n-1)}{4} = \Theta\left(n + \frac{n(n-1)}{4}\right) = \Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Compare with Selection Sort, which is input-insensitve: best, average and worst case complexity is $\Theta(n^2)$

Insertion Sort - Complexity

- Worst case: $\Theta(n^2)$
- Best case: $\Theta(n)$
- Average case: $\Theta(n^2)$

Compare with Selection Sort, which is input-insensitve: best, average and worst case complexity is $\Theta(n^2)$

Insight
In many cases, real-world data is **already partially sorted.** This makes Insertion Sort a powerful sorting algorithm in practice, particularly useful for small arrays (up to hundreds of elements).

Insertion Sort - A faster version

function $\text{INSERTIONSORT}(A[0..n-1])$
  for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ **and** $v < A[j]$ **do**
      $A[j+1] \leftarrow A[j]$
      $j \leftarrow j - 1$
    $A[j+1] \leftarrow v$

Insertion Sort - A faster version

function $\textsc{InsertionSort}(A[0..n-1])$

  for $i \leftarrow 1$ **to** $n-1$ **do**

    $\boxed{v \leftarrow A[i]}$

    $j \leftarrow i - 1$

    while $\boxed{j \geq 0}$ **and** $v < A[j]$ **do**

      $\boxed{A[j+1] \leftarrow A[j]}$

      $j \leftarrow j - 1$

    $\boxed{A[j+1] \leftarrow v}$

This is the version presented in the Levitin book.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort can be made faster by using a *min* **sentinel** in that cell ($A[-1]$) and change the test from

$$\cancel{j \geq 0} \textbf{ and } v < A[j]$$

to just

$$\boxed{v < A[j]}$$

Insertion Sort - Sentinels

- Assume the domain is bounded from below.
- There is a **minimal** element *min*.
- Assume a **free cell** to the left of $A[0]$

Insertion Sort can be made faster by using a *min* **sentinel** in that cell ($A[-1]$) and change the test from

$$j \geq 0 \textbf{ and } v < A[j]$$

to just

$$v < A[j]$$

For this reason, extreme array cells (such as $A[0]$ in C, and/or $A[n+1]$) are sometimes left free deliberately, so that they can be used to hold sentinels; only $A[1]$ to $A[n]$ hold proper data.

Sorting - Practical Implementations

- C - Quicksort (fastest)
- C++ - Introsort (a variant of Quicksort)
- Javascript/Mozilla: Mergesort (stable)
- Python: Timsort (very roughly, a mix of Mergesort and Insertion Sort, stable)
- Linux Kernel: Heapsort (low memory consumption, guaranteed $\Theta(nlogn)$ worst case performance: important for security reasons)

COMP20007 Design of Algorithms

Sorting - Part 2

Daniel Beck

Lecture 12

Semester 1, 2020

Mergesort

function $\textsc{Mergesort}(A[0..n-1])$

  if $n > 1$ **then**

SAME ARRAY

    $B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1]$

    $C[0..\lfloor n/2 \rfloor - 1] \leftarrow A[\lfloor n/2 \rfloor..n - 1]$

    $\textsc{Mergesort}(B[0..\lfloor n/2 \rfloor - 1])$

    $\textsc{Mergesort}(C[0..\lfloor n/2 \rfloor - 1])$

    $\textsc{Merge}(B, C, A)$

Mergesort - Merge function

function $\textsc{Merge}(B[0..p-1], C[0..q-1], A[0..p+q-1])$

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

  if $\boxed{B[i] \leq C[j]}$ **then**

    $A[k] \leftarrow B[i]; i \leftarrow i + 1$

  else

    $A[k] \leftarrow C[j]; j \leftarrow j + 1$

  $k \leftarrow k + 1$

if $i == p$ **then**

  $A[k..p+q-1] \leftarrow C[j..q-1]$

else

  $A[k..p+q-1] \leftarrow B[i..p-1]$

Mergesort - Properties

Divide-and-Conquer algorithm

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Questions!

Mergesort - Properties

Divide-and-Conquer algorithm

- In contrast with decrease-and-conquer algorithms, such as Insertion Sort.

Questions!

- In-place? (or does it require extra memory?)
- Stable?

Mergesort - Properties

- In-place?

Mergesort - Properties

ITERATIVE

- In-place? **No.** (requires $O(n)$ auxiliary array + $\cancel{O(\log n)}$ stack space for recursion)
- Stable?

Mergesort - Properties

344

35

→ SORTED

- In-place? **No.** (requires $O(n)$ auxiliary array + $O(\log n)$ stack space for recursion)
- Stable? **Yes!** (Merge keeps relative order with additional bookkeeping)

$B[i] \boxed{\leq} C[j]$

~~34~~ 45

3445

Mergesort - Complexity

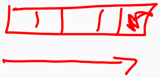
- Worst case?
- Best case?
- Average case?

Mergesort

$$
\begin{array}{l}
\textbf{function } \textsc{Mergesort}(A[0..n-1]) \quad = C(n) \\
\quad \textbf{if } n > 1 \textbf{ then} \\
\quad\quad B[0..\lfloor n/2 \rfloor - 1] \leftarrow A[0..\lfloor n/2 \rfloor - 1] \\
\quad\quad C[0..\lfloor n/2 \rfloor - 1] \leftarrow A[\lfloor n/2 \rfloor..n-1] \\
\quad\quad \textsc{Mergesort}(B[0..\lfloor n/2 \rfloor - 1]) \\
\quad\quad \textsc{Mergesort}(C[0..\lfloor n/2 \rfloor - 1]) \\
\quad\quad \textsc{Merge}(B, C, A)
\end{array}
$$

$$C(n) = 2C(n/2) + \underline{C_{MERGE}(n)}$$

Mergesort - Complexity

Merge: B  $n/2$ C  $n/2$ $= n-1$

$C^{merge}_{worst} = n - 1$

$$C(n) = 2C(n/2) + \overset{\Theta(n)}{\boxed{n-1}}$$

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

$a = 2$ $d = 1$

$b = 2$

$$2 = 2^1 \Rightarrow a = b^d \Rightarrow \Theta(n^d \log n)$$

$$\boxed{\Theta(n \log n)}$$

$$C^{merge}_{best} = \frac{n}{2} - 1 = \Theta(n)$$

$$\hookrightarrow \boxed{\Theta(n \log n)}$$

Mergesort - In Practice

DATA CENTER

HARD DRIVE

- Guaranteed $\boxed{\Theta(n \log n)}$ complexity
- Highly parallelisable
- Multiway Mergesort: excellent for secondary memory
- Used in JavaScript (Mozilla)
- Basis for hybrid algorithms (TimSort: Python, Android)

Take-home message: Mergesort is an excellent choice if **stability** is required and the extra memory cost is low.

$O(n)$

Quicksort

function $\textsc{Quicksort}\boxed{(A[l..r])}$ $\triangleright$ Starts with $A[0..n-1]$
    if $l < r$ **then**
PIVOT → $s \leftarrow \textsc{Partition}(A[l..r])$
        $\textsc{Quicksort}(A[l..s-1])$
        $\textsc{Quicksort}(A[s+1..r])$

Quicksort - Lomuto partitioning

function LomutoPartition($A[l..r]$)
    $p \leftarrow A[l]$
    $s \leftarrow l$
    for $i \leftarrow l + 1$ **to** r **do**
        if $A[i] < p$ **then**
            $s \leftarrow s + 1$
            Swap($A[s], A[i]$)
    Swap($A[l], A[s]$)
    return s

Quicksort - Properties

Divide-and-Conquer algorithm

Quicksort - Properties

Divide-and-Conquer algorithm

Questions!

Quicksort - Properties

Divide-and-Conquer algorithm

Questions!

- In-place?
- Stable?

Quicksort - Properties

- In-place?

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.
- Stable?

Quicksort - Properties

- In-place? **Yes**, buy still requires $O(\log n)$ memory for the stack.
- Stable? **No** (non-local swaps)

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.
- Instead, practical implementations use Hoare partitioning (proposed by the inventor of Quicksort).

Quicksort - Partitioning

- Lomuto partitioning can be used but not the best in pratice.
- Instead, practical implementations use Hoare partitioning (proposed by the inventor of Quicksort).
- How does it work? Let's go back to my games first...

Quicksort - Hoare partitioning

function $\textsc{HoarePartition}(A[l..r])$

- $p \leftarrow A[l]$
- $i \leftarrow l; j \leftarrow r+1$
- **repeat**
  - **repeat** $i \leftarrow i+1$ **until** $A[i] \geq p$
  - **repeat** $j \leftarrow j-1$ **until** $A[j] \leq p$
  - $\textsc{Swap}(A[i], A[j])$
- **until** $i \geq j$ → CROSS
- $\textsc{Swap}(A[i], A[j])$ → UNDO SWAP
- $\textsc{Swap}(\boxed{A[l]}, A[j])$ → PIVOT INTO FINAL POSITION
- **return** j

Quicksort - Complexity

- Worst case?
- Best case?
- Average case?

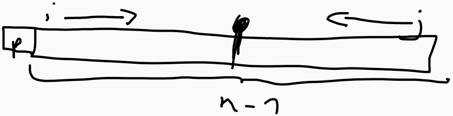
Quicksort - Complexity

- Worst case?
- Best case?
- Average case?

Warning: not trivial, but give it a go. ;)

Quicksort - Complexity

PARTITION



$$n-1+1+1 = n+1$$

$$\boxed{\Theta(n)}$$

QUICKSORT

$$C_{BEST} = 2\,C_{BEST}(n/2) + \Theta n = \Theta(n \log n$$

Quicksort - Complexity

QUICKSORT – WORST

$\emptyset$ p — LARGER

0 — $n-1$ — p, $n-1$

PARTITION $= n+1$

$$C_{worst} = (n+1) + n + (n-1) + \dots + 3 = \frac{(n+1)(n+2)}{2} - 3 \in \Theta(n^2)$$

$$C_{best} \in \Theta(n \log n)$$

C_{avg}? $s \to$ ~~PIVOT~~ SIZE

$$\boxed{\frac{1}{n} \sum_{s=0}^{n-1} n+1 + C_{avg}(s) + C_{avg}(n-1-s)} \to \Theta(n \log n)$$

Quicksort - In practice

- Used in C (qsort)
- Basis for C++ sort (Introsort)
- **Fastest** sorting algorithm in most cases

Quicksort - In practice

- Used in C (qsort)
- Basis for C++ sort (Introsort)
- **Fastest** sorting algorithm in most cases

Take-home message: Quicksort is the algorithm of choice when **speed** matters and stability is not required.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Quicksort: Usually the best choice for large arrays, with excellent **empirical** performance and only $O(\log n)$ memory cost.

Summary so far

Selection Sort: Slow, but only $O(n)$ key exchanges.

Insertion Sort: Very good for small arrays and when data is expected to be "almost sorted".

Mergesort: Better for mid-size arrays and when stability is required.

Quicksort: Usually the best choice for large arrays, with excellent **empirical** performance and only $O(\log n)$ memory cost.

Next lecture: Heapsort

COMP20007 Design of Algorithms

Sorting - Part 3

Daniel Beck

Lecture 13

Semester 1, 2020

Priority Queues

- A *set* of elements, each one with a *priority key*.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.
- Used as part of an algorithm (ex: Dijkstra)...

Priority Queues

- A *set* of elements, each one with a *priority key*.
- **Inject:** put a new element in the queue.
- **Eject:** find the element with the *highest* priority and remove it from the queue.
- Used as part of an algorithm (ex: Dijkstra)...
- ...or on its own (ex: OS job scheduling).

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.
- If using an *unsorted array/list*, we obtain Selection Sort.

Sorting using a Priority Queue

- Different implementations result in different sorting algorithms.
- If using an *unsorted array/list*, we obtain Selection Sort.
- If using an *heap*, we obtain Heapsort.

The Heap

It's a tree with a set properties:

The Heap

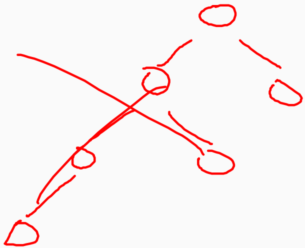
It's a tree with a set properties:

- Binary (at most two children allowed per node)

The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)



The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)
- Complete (all levels are full except for the last, where only rightmost leaves can be missing) (implies Balanced)

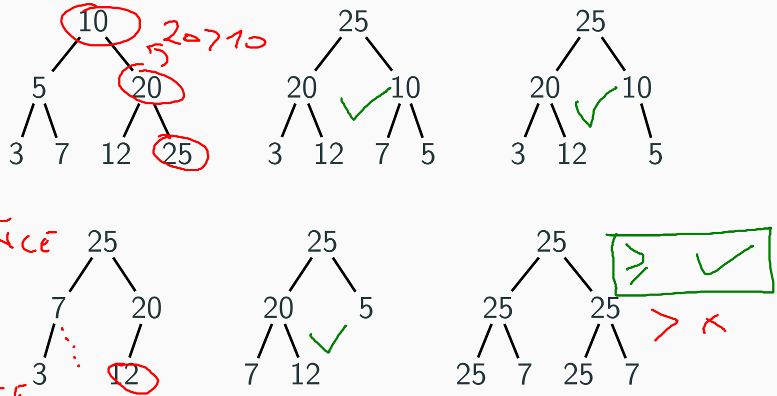
The Heap

It's a tree with a set properties:

- Binary (at most two children allowed per node)
- Balanced (the difference in height between two leaf nodes is never higher than 1)
- Complete (all levels are full except for the last, where only rightmost leaves can be missing) (implies Balanced)
- Parental dominance (the key of a parent node is always higher than the key of its children)

Heaps and Non-Heaps

Which of these are heaps?



✗ DOESN'T HAVE PARENTAL DOMINANCE

✗ NOT COMPLETE

Heaps as Arrays

We can utilise the completeness of the tree and place its elements in level-order in an array A.

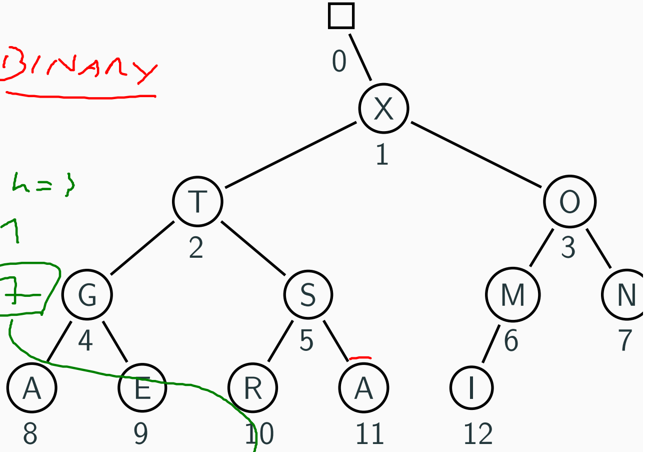
BINARY

h = 3

$2^h - 1$

$2^3 - 1 = \boxed{7}$

Note that the children of node i will be nodes $2i$ and $2i + 1$.



ROOT

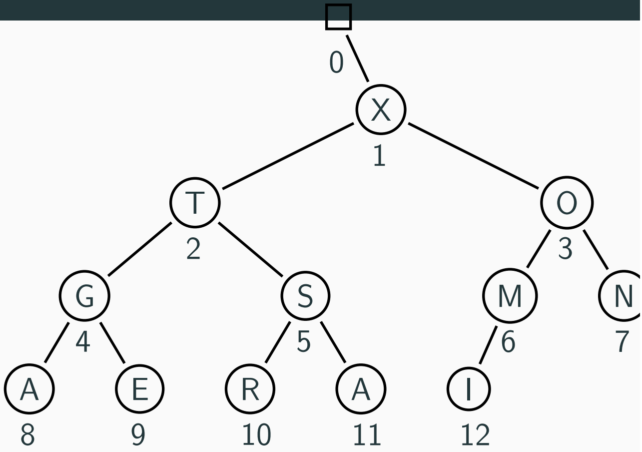
LEAVES

A:													
		X	T	O	G	S	M	N	A	E	R	A	I
	0	1	2	3	4	5	6	$\boxed{7}$	8	9	10	11	12

Heaps as Arrays

This way, the heap condition is simple:

$\forall i \in \{0, 1, \ldots, n\}$, we must have $A[i] \leq A[\lfloor i/2 \rfloor]$.



A:		X	T	O	G	S	M	N	A	E	R	A	I
	0	1	2	3	4	5	6	7	8	9	10	11	12

Heapsort

function $\textsc{Heapsort}(A[1..n])$ $\triangleright$ Assume $\boxed{A[0]}$ as a sentinel

  $\textsc{Heapify}(A[1..n])$ → MAKES $\boxed{A}$

  for $i \leftarrow n$ **to** $\cancel{0}$ **do** → A PRIORITY QUEUE

    $\textsc{Eject}(A[1..i])$

Heapify

```
function BottomUpHeapify(A[1..n])
    for i ← ⌊n/2⌋ downto 1 do
        k ← i
        v ← A[k]
BOOLEAN heap ← False
        while not heap and 2 × k ≤ n do
            j ← 2 × k
            if j < n then                          ▷ two children
                if A[j] < A[j + 1] then j ← j + 1
            if v ≥ A[j] then heap ← True
            else A[k] ← A[j]; k ← j
        A[k] ← v
```

NON-LEAVES

Analysis of Bottom-Up Heapify

$h = 2$ $\quad n = 2^{2+1} - 1 = 7$

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

Analysis of Bottom-Up Heapify

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

TOTAL KEY COMPARISONS

Here is an upper bound on the number of "down-sifts" needed

2 KEY COMPARISONS

$$\sum_{i=0}^{h-1} \sum_{\text{nodes at level } i} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1))$$

The last equation can be proved by mathematical induction.

Analysis of Bottom-Up Heapify

Assume the heap is a full binary tree: $n = 2^{h+1} - 1$.

Here is an upper bound on the number of "down-sifts" needed

$$\sum_{i=0}^{h-1} \sum_{\text{nodes at level } i} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n-\log_2(n+1))$$

The last equation can be proved by mathematical induction.

$\boxed{2n} - 2\log(n+1)$

Note that $2(n - \log_2(n+1)) \in \Theta(n)$, hence we have a linear-time algorithm for heap creation.

Eject

function $\textsc{Eject}(A[1..i])$

$\textsc{Swap}(A[i], A[1])$

ROOT

$k \leftarrow 1$

$v \leftarrow A[k]$

$heap \leftarrow False$

while not $heap$ **and** $2 \times k \leq i - 1$ **do**

$j \leftarrow 2 \times k$

if $j < i - 1$ **then** ▷ two children

if $A[j] < A[j+1]$ **then** $j \leftarrow j + 1$

if $v \geq A[j]$ **then** $heap \leftarrow True$

else $A[k] \leftarrow A[j]$; $k \leftarrow j$

$A[k] \leftarrow v$

Heapsort - Properties

Transform-and-Conquer paradigm

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

- In-place?

Heapsort - Properties

Transform-and-Conquer paradigm

- Exactly as Selection Sort, but with a *preprocessing* step.

Questions!

- In-place?
- Stable?

Heapsort - Properties

- In-place?

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.
- Stable?

Heapsort - Properties

- In-place? **Yes!** Only additional $O(1)$ memory for auxiliary variables.
- Stable? **No.** Non-local swaps break stability.

Heapsort - Complexity

- We already saw in the videos that the complexity of Heapsort in the worst case is $\Theta(n \log n)$.

Heapsort - Complexity

- We already saw in the videos that the complexity of Heapsort in the worst case is $\Theta(n \log n)$.
- What's the complexity in the best case?

Heapsort - Complexity

- Heapify is $\Theta(n)$ in the worst case.

Heapsort - Complexity

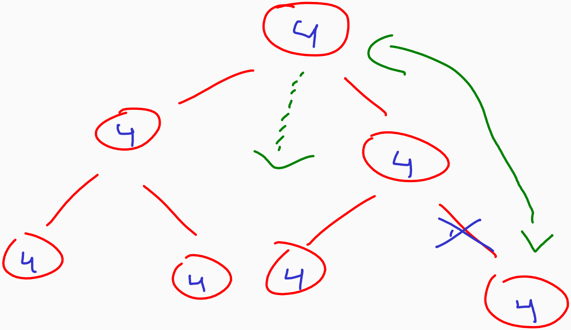
- Heapify is $\Theta(n)$ in the worst case.
- Eject is $\Theta(\log n)$ in the worst case.

Heapsort - Complexity

- ~~Heapify~~ is $\Theta(n)$ in the worst case.
- Eject is $\Theta(\log n)$ in the worst case.
- In the worst case, heapsort is $\Theta(n) + \boxed{n \times \Theta(\log n)} \in \Theta(n \log n)$.

$\Theta(n) \Rightarrow n \times \boxed{\Theta(1)}$

Heapsort - Complexity (best case)



EJECT = $\Theta(1)$

BEST CASE = $\Theta(n)$

DISTINCT ELEMENTS = $\Theta(n \log n)$

Heapsort - In practice

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.
- Used in the Linux kernel.

Heapsort - In practice

- Vs. Mergesort: fully in-place but not stable
- Vs. Quicksort: guaranteed $\Theta(n \log n)$ performance but empirically slower.
- Used in the Linux kernel.

Take-home message: Heapsort is the best choice when low-memory footprint is required and guaranteed $\Theta(n \log n)$ performance is needed (for example, security reasons).

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.
- Quicksort: best for more general cases and large amounts of data. Not stable.

Sorting - Practical Summary

- Selection Sort: slow, but $O(n)$ swaps.
- Insertion Sort: low-memory, stable, excellent for small and almost sorted data.
- Mergesort: good for mid-size data and when stability is required. Also good with secondary memory devices. Extra $O(n)$ memory cost can be a barrier.
- Quicksort: best for more general cases and large amounts of data. Not stable.
- Heapsort: slower in practice but low-memory and guaranteed $\Theta(n \log n)$ performance.

Sorting - Paradigm Summary

- Selection Sort: *brute force*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

There are all *comparison-based* sorting algorithms.

Sorting - Paradigm Summary

- Selection Sort: *brute force*
- Insertion Sort: *decrease-and-conquer*
- Mergesort, Quicksort: *divide-and-conquer*
- Heapsort: *transform-and-conquer*

There are all *comparison-based* sorting algorithms.

Next lecture: Ok, my data is sorted. Now how do I keep it sorted?

COMP20007 Design of Algorithms

Binary Search Trees and their Extensions

Daniel Beck

Lecture 14

Semester 1, 2020

Dictionaries

Dictionaries

- Abstract Data Structure

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
    - Values are usually *records*, such as my videogames or the Unimelb students.

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
  - Values are usually *records*, such as my videogames or the Unimelb students.
  - Keys are (unique) identifiers, such as the name of a game or the student ID.

Dictionaries

- Abstract Data Structure
- Collection of *(key, value)* pairs
  - Values are usually *records*, such as my videogames or the Unimelb students.
  - Keys are (unique) identifiers, such as the name of a game or the student ID.
- Required operations:
  - *Search* for a value (given a key)
  - *Insert* a new pair
  - *Delete* an existent pair (given a key)

Dictionaries - Implementations

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\boxed{\Theta(n)}$ comparisons;

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

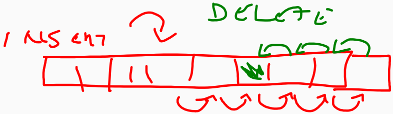
- Search: $\boxed{\Theta(\log n)}$ comparisons;

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array



- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\boxed{\Theta(n) \text{ record swaps;}}$

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(n)$ record swaps;

This lecture: a better data structure

Dictionaries - Implementations

Unsorted array / Linked list

- Search: $\Theta(n)$ comparisons;

Sorted array

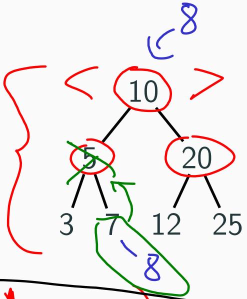
- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(n)$ record swaps;

This lecture: a better data structure

- Search: $\Theta(\log n)$ comparisons;
- Insert/Delete: $\Theta(\log n)$ record swaps;

Binary Search Tree

$h = \log n$



AVERAGE CASE

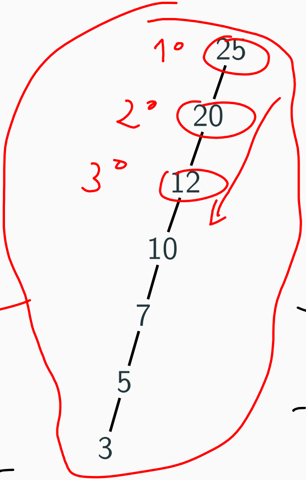
SEARCH: $\Theta(\log n)$

INSERT: $\Theta(\log n) + \Theta(1) \in \Theta(\log n)$

DELETE: $\Theta(\log n) + \Theta(\log n) \in \Theta(\log n)$

Binary Search Tree - Worst Case

LINKED LIST



$\Theta(n)$

WORST CASE

→ HIGHLY UNBALANCED

⇓

BALANCED

BST - How to avoid degeneracy?

Two options:

LINKED
LIST

BST - How to avoid degeneracy?

Two options:

- Self-balancing
  - **AVL trees**
  - Red-black trees
  - Splay trees

BST - How to avoid degeneracy?

Two options:

- Self-balancing
    - **AVL trees**
    - Red-black trees
    - Splay trees
- Change the representation → NODES TO HAVE > 1 ELEMENT
    - **2-3 trees**
    - 2-3-4 trees
    - B-trees

AVL trees

AVL trees

- Named after Adelson-Velsky and Landis.

AVL trees

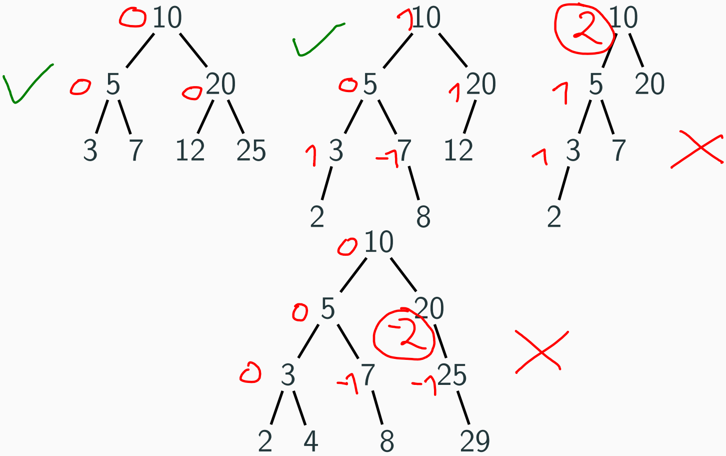
- Named after Adelson-Velsky and Landis.
- A BST where each node has a *balance factor*: the difference in height between the left and right subtrees.

AVL trees

- Named after Adelson-Velsky and Landis.
- A BST where each node has a *balance factor*: the difference in height between the left and right subtrees.
- When the balance factor becomes 2 or -2, *rotate* the tree to adjust them.

AVL Trees: Examples and Counter-Examples

Which of these are AVL trees?



AVL Trees - Rotations

- Search is done as in BSTs.

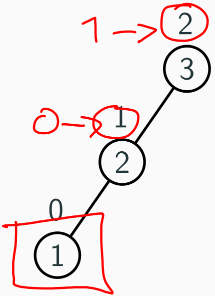
AVL Trees - Rotations

- Search is done as in BSTs.
- Insertion and Deletion also done as in BSTs, with additional steps at the end.

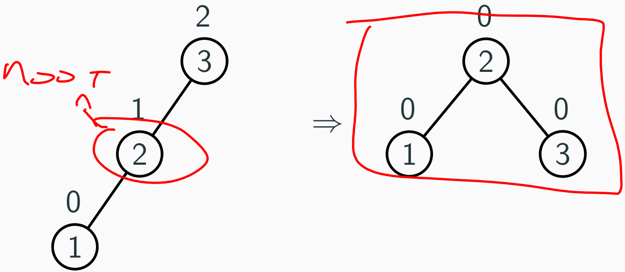
AVL Trees - Rotations

- Search is done as in BSTs.
- Insertion and Deletion also done as in BSTs, with additional steps at the end.
    - Update balance factors.
    - If the tree becomes unbalanced, perform *rotations* to rebalance it.

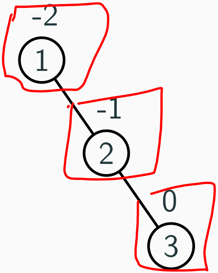
AVL Trees: R-Rotation



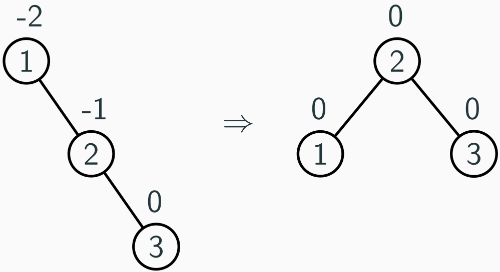
AVL Trees: R-Rotation



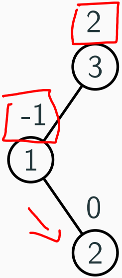
AVL Trees: L-Rotation



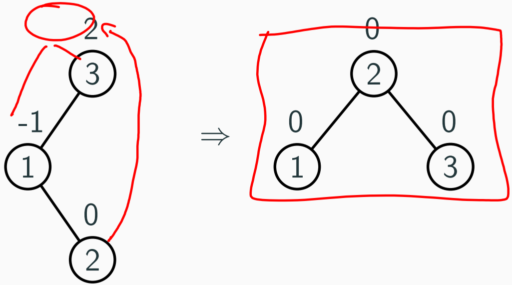
AVL Trees: L-Rotation



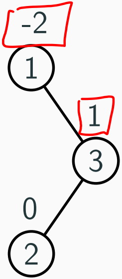
AVL Trees: LR-Rotation



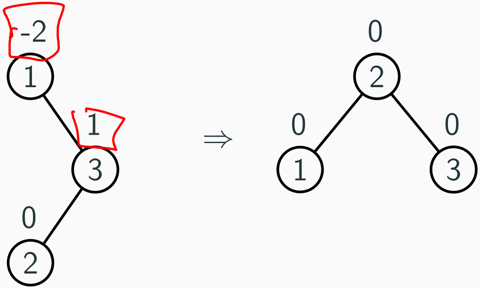
AVL Trees: LR-Rotation



AVL Trees: RL-Rotation



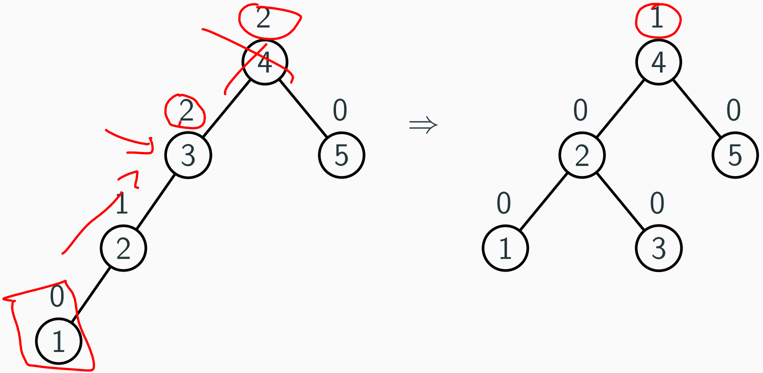
AVL Trees: RL-Rotation



AVL Trees: Where to Perform the Rotation

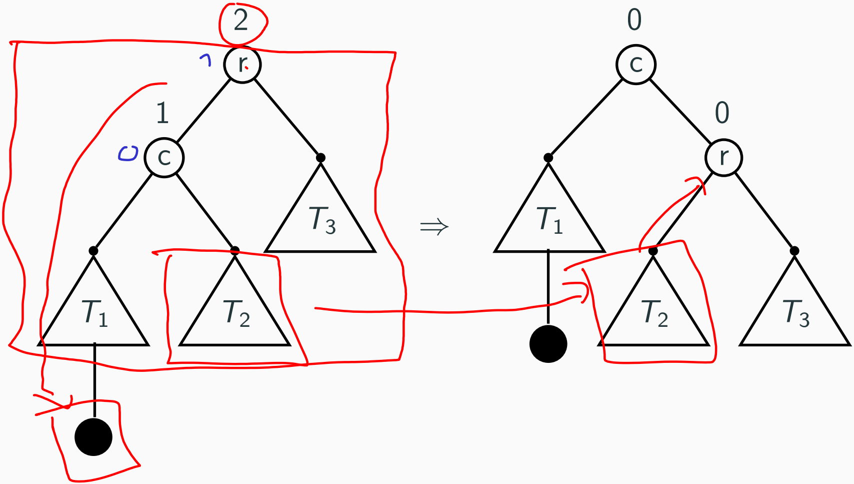
Along an unbalanced path, we may have several

nodes with balance factor 2 (or -2):



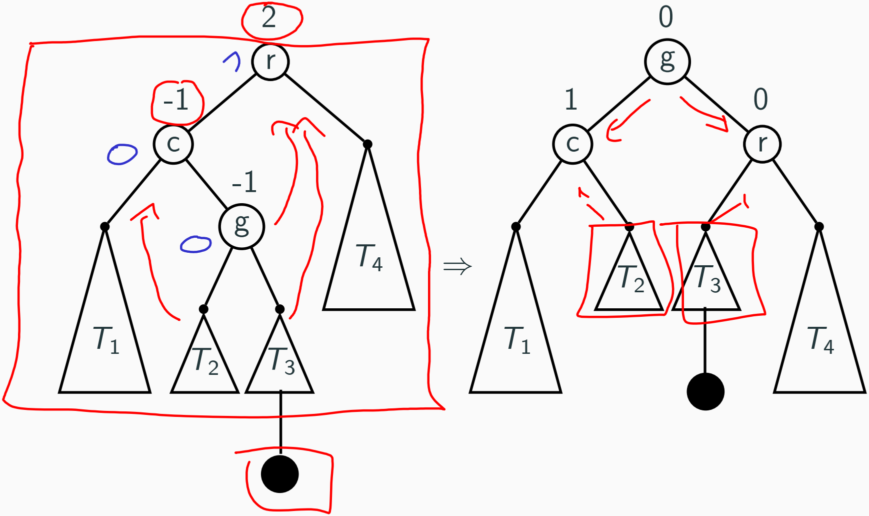
It is always the lowest unbalanced subtree that is re-balanced.

AVL Trees: The Single Rotation, Generally



This shows an R-rotation; an L-rotation is similar.

AVL Trees: The Double Rotation, Generally



This shows an LR-rotation; an RL-rotation is similar.

Properties of AVL Trees

- Rotations ensure that an AVL tree is always balanced.

Properties of AVL Trees

- Rotations ensure that an AVL tree is always balanced.
- An AVL tree with n nodes has depth $\boxed{\Theta(\log n)}$.

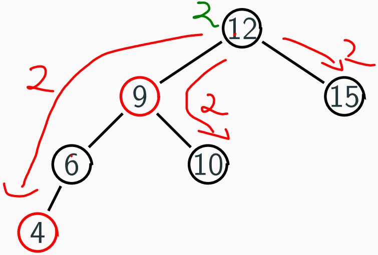
Properties of AVL Trees

- Rotations ensure that an AVL tree is always balanced.
- An AVL tree with n nodes has depth $\Theta(\log n)$.
- This ensures all three operations are $\Theta(\log n)$.

Red-black Trees

- A red-black tree is another self-balancing BST.
- Its nodes are coloured red or black, so that:

1. No red node has a red child.
2. Every path from the root to the leaves has the same number of black nodes.



A worst-case red-black tree (the longest path is twice as long as the shortest path).

Self-balancing trees - In practice

AVL trees vs. red-black trees

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are “more balanced” but require more frequent rotations.
  - Better if searches are more frequent than insertions/deletions.

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are “more balanced” but require more frequent rotations.
    - Better if searches are more frequent than insertions/deletions.
- Red-black trees are “less balanced” but require less rotations.
    - Better if insertions/deletions are more frequent than searches.

Self-balancing trees - In practice

AVL trees vs. red-black trees

- AVL trees are "more balanced" but require more frequent rotations.
  - Better if searches are more frequent than insertions/deletions.
- Red-black trees are "less balanced" but require less rotations.
  - Better if insertions/deletions are more frequent than searches.

Key property: rotations keep trees in a shape that guarantees $\Theta(\log n)$ operations.

Representational Changes

Representational Changes

- Goal is the same: keep the tree balanced.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.

Representational Changes

- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
    - A 2-node contains one element and at most two children (as in BSTs)
    - A 3-node contains two elements and at most three children.

Representational Changes

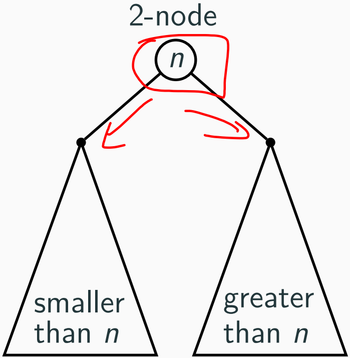
- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
    - A 2-node contains one element and at most two children (as in BSTs)
    - A 3-node contains two elements and at most three children.
- Easy way to keep the tree balanced.

Representational Changes

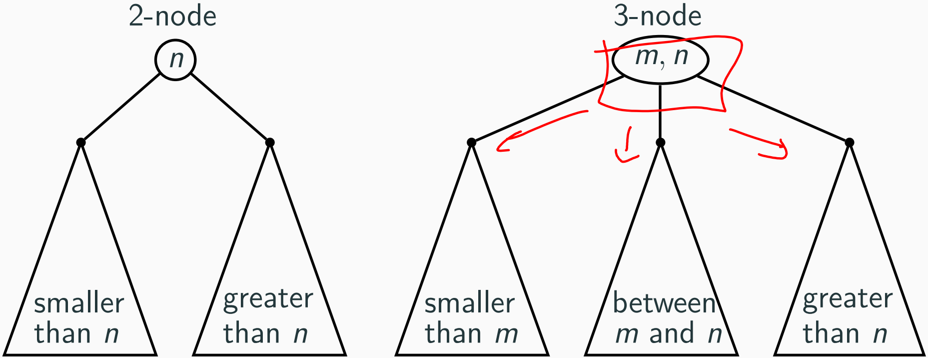
- Goal is the same: keep the tree balanced.
- But instead of relying on rotations, we will allow *multiple elements per node* and *multiple children per node*.
- 2–3 trees: contains only 2-nodes and 3-nodes.
  - A 2-node contains one element and at most two children (as in BSTs)
  - A 3-node contains two elements and at most three children.
- Easy way to keep the tree balanced.
- Can be extended in many ways: 2–3–4 trees, B-trees, etc.

2-Nodes and 3-Nodes

2-Nodes and 3-Nodes



2-Nodes and 3-Nodes



Insertion in a 2–3 Tree

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.

Insertion in a 2–3 Tree

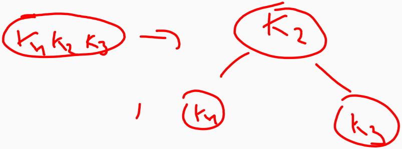
- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.
  - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.



Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
  - In sorted order, call them k_1, k_2, and k_3.
  - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.
  - If the parent node was a 3-node, repeat.

Insertion in a 2–3 Tree

- As in a BST, pretend that we are searching for k.
- If the leaf node is a 2-node, insert k, becoming a 3-node.
- Otherwise, momentarily form a node with three elements:
    - In sorted order, call them k_1, k_2, and k_3.
    - Split the node, so that k_1 and k_3 form their own individual 2-nodes, and k_2 is promoted to the parent node.
    - If the parent node was a 3-node, repeat.

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

2-node　　　3-node

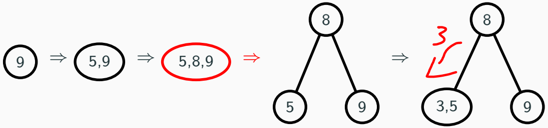
(9) $\Rightarrow$ (5,9)

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

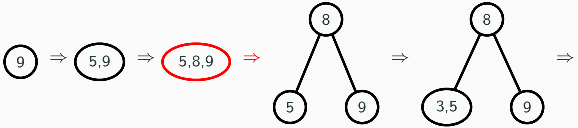
9 $\Rightarrow$ 5,9 $\Rightarrow$ 5,8,9

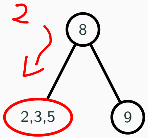
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

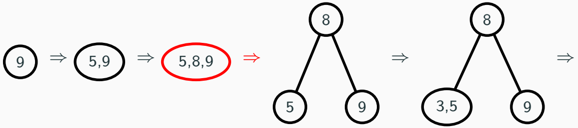


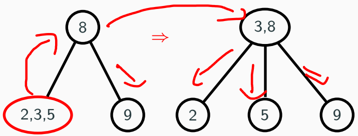
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



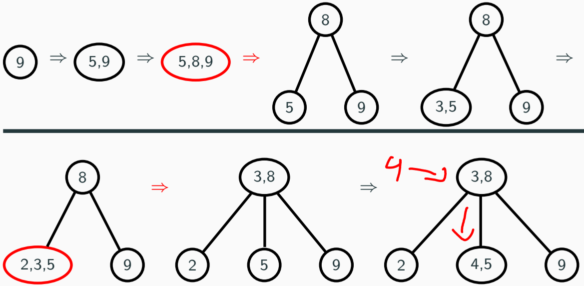


Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

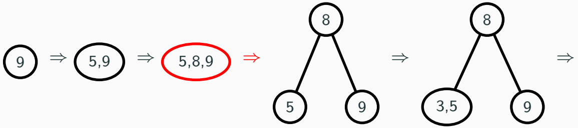


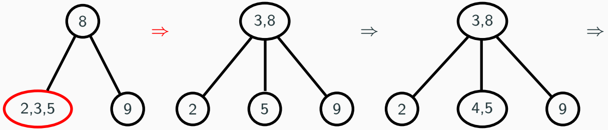


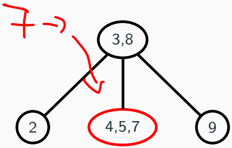
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



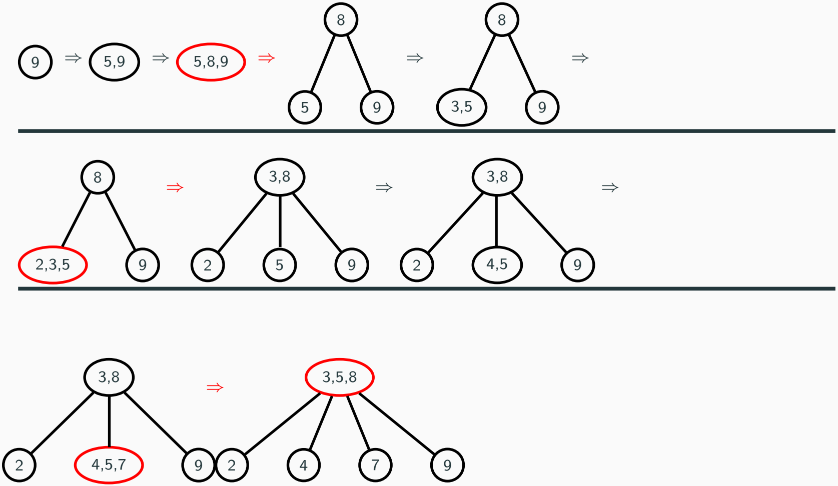
Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



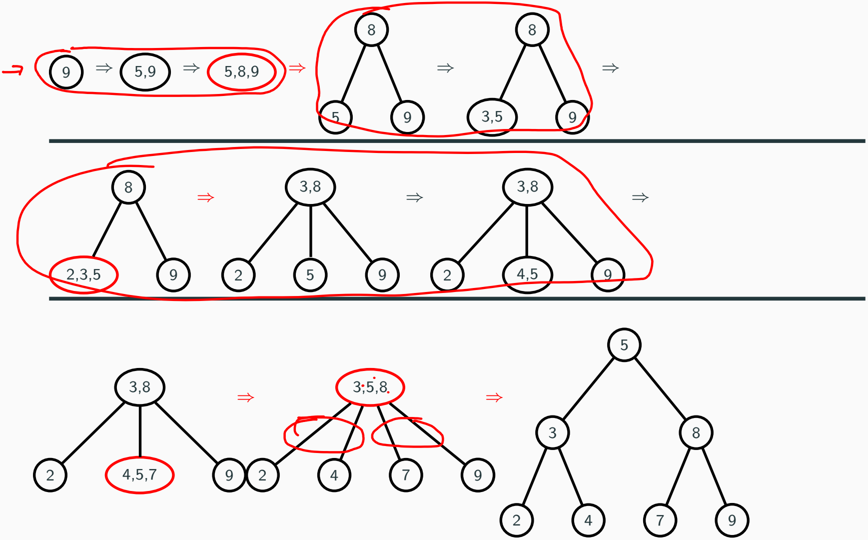




Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7

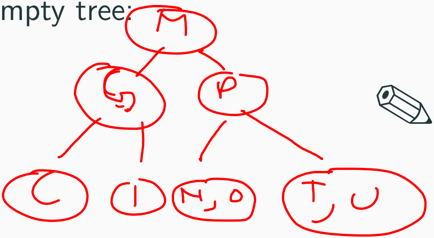


Example: Build a 2–3 Tree from 9, 5, 8, 3, 2, 4, 7



Exercise: 2–3 Tree Construction

Build the 2–3 tree that results from inserting these keys, in the given order, into an initially empty tree:



C, O, M, P, U, T, I, N, G

Extensions

- 2–3–4 trees: includes 4-nodes.

Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)

Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)
- B$^+$-trees: internal nodes *only contain keys*, values are all in the leaves (plus a bunch of optimisations)

Extensions

- 2–3–4 trees: includes 4-nodes.
- B-trees: a generalisation. (2–3 trees are B-trees of order 3)
- B^+-trees: internal nodes *only contain keys*, values are all in the leaves (plus a bunch of optimisations)

Key property: balance is achieved by allowing multiple elements per node.

BSTs - Summary

ELEMENT
RECORD

- Dictionaries store *(key, value)* pairs.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.
- Standard BSTs can degrade to linked lists and $\Theta(n)$ worst case performance.

BSTs - Summary

- Dictionaries store *(key, value)* pairs.
- BSTs provide $\Theta(\log n)$ search, insert and delete operations.
- Standard BSTs can degrade to linked lists and $\Theta(n)$ worst case performance.
    - Self-balancing: AVL trees
    - Change of representation: 2–3 trees

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved. → HARD DRIVE

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
    - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved.
    - B-trees (and its ~~in~~variants) are widely used in SQL databases (PostgreSQL, SQLite) and filesystems (Ext4, Reiser4).

BSTs - In practice

- Self-balancing trees tend to be better when the dictionary is fully in memory.
  - C++ *maps*: implemented via Red-black trees.
- Multiple elements per node are a better choice when secondary memory is involved.
  - B-trees (and its invariants) are widely used in SQL databases (PostgreSQL, SQLite) and filesystems (Ext4, Reiser4).

Next week: C++ maps use BSTs. What about Python *dicts*, do they also use BSTs? (spoiler: no)

COMP20007 Design of Algorithms

Hashing

Daniel Beck

Lecture 15

Semester 1, 2020

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete
- Last lecture: Binary Search Trees (and extensions)

$\Theta(n)$

$\Theta(\log n)$

Dictionaries - Recap

- Abstract Data Structure: collection of *(key, value)* pairs.
- Required operations: Search, Insert, Delete
- Last lecture: Binary Search Trees (and extensions)
- This lecture: Hash Tables.

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.

ARRAY

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$
- Requires a *hash function*: $h(K) \rightarrow i \in [0, m-1]$.

Hash Tables

- A hash table is a continuous data structure with m preallocated entries.
- Average case performance for Search, Insert and Delete: $\Theta(1)$
- Requires a *hash function*: $h(K) \rightarrow i \in [0, m-1]$.
- A hash function should:
  - Be efficient $\boxed{(\Theta(1)).}$
  - Distribute keys evenly (uniformly) along the table.

Identity Hash Function

STUDENT ID

737687 → POSITION

Question: if keys are integers, why do I need a hash function?

I could just use the key as the index, no?

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $\boxed{h(K) = K.}$

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m-1]$. In other words *we need to know the maximum number of keys in advance*.

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m-1]$. In other words *we need to know the maximum number of keys in advance*.
- Sometimes this is possible: postcodes, for example.

4 DIGITS

0 – 9999

Identity Hash Function

Question: if keys are integers, why do I need a hash function? I could just use the key as the index, no?

- This is the *identity* hash function: $h(K) = K$.
- Note that $K \in [0, m-1]$. In other words *we need to know the maximum number of keys in advance*.
- Sometimes this is possible: postcodes, for example.
- Many times it is not:
  - m is too large (need to preallocate)
  - Unbounded integers (student IDs)
  - Non-integer keys (games)

↳ 10000 PREALLOCATE

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = \underline{K \mod m}$

$m = 9$

$2 \bmod 9 = 2$

$17 \bmod 9 = 8$

$20 \bmod 9 = 2$

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = K \mod m$
- Allow us to set the size m.

Hashing Integers

- For large/unbounded integers, an alternative function is $h(K) = K \mod m$
- Allow us to set the size m.
- Small m results in lots of collisions, large m takes excessive memory. Best m will vary.

Hashing Strings

- Assume A $\mapsto 0$, B $\mapsto 1$, etc. 0 - 25
- Assume 26 characters and $m = 101$.
- Each character can be mapped to a *binary* string of length 5 ($2^5 = 32$). > 26

Hashing Strings

- Assume A $\mapsto 0$, B $\mapsto 1$, etc.
- Assume 26 characters and $m = 101$.
- Each character can be mapped to a *binary* string of length 5 ($2^5 = 32$).

We can think of a string as a long binary number:

M Y K E Y $\mapsto$ 0110011000010100010011000 (= 13379736)

4

$$13379736 \bmod 101 = 64$$

So 64 is the position of string M Y K E Y in the hash table.

Hashing Strings

We deliberately chose m to be prime.

$$13379736 = 12 \times 32^4 + 24 \times 32^3 + 10 \times 32^2 + 4 \times 32 + 24$$

With $m = 32$, the hash value of any key is the last character's value!

$m = 2^5$

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathrm{c}) = 2$.

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathrm{c}) = 2$.

Then we have

$h(s) = (\sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}) \bmod m$,

where m is a prime number. For example,

$h(\mathrm{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y}) = (21 \times 32^{10} + 4 \times 32^{9} + \cdots)\ mod\ 101$

Hashing Long Strings

Assume *chr* be the function that gives a character's number, so for example, $chr(\mathrm{c}) = 2$.

Then we have

$h(s) = (\sum_{i=0}^{|s|-1} chr(s_i) \times 32^{|s|-i-1}) \bmod m$,

where m is a prime number. For example,

$h(\mathrm{V\ E\ R\ Y\ L\ O\ N\ G\ K\ E\ Y}) = (21\times32^{10}+4\times32^{9}+\cdots)\ mod\ 101$

The term between parenthesis can become quite large and result in overflow.

Horner's Rule

Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} \cdots$$

factor out repeatedly:

$$(\cdots ((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

$21 \times \boxed{32^9} \times 32 + 4 \times \boxed{32^9} =$

$= (21 \times 32 + 4) \times 32^9 \cdots\cdots$

Horner's Rule

Instead of

$$21 \times 32^{10} + 4 \times 32^{9} + 17 \times 32^{8} + 24 \times 32^{7} \cdots$$

factor out repeatedly:

$$(\cdots ((21 \times 32 + 4) \times 32 + 17) \times 32 + \cdots) + 24$$

mod m

Now utilize these properties of modular arithmetic:

$$\begin{aligned}(x + y) \bmod m &= \boxed{((x \bmod m) + (y \bmod m)) \bmod m} \\ (x \times y) \bmod m &= \boxed{((x \bmod m) \times (y \bmod m)) \bmod m}\end{aligned}$$

So for each sub-expression it suffices to take values modulo m.

Collisions

Happens when the hash function give identical results to two different keys.

Collisions

Happens when the hash function give identical results to two different keys.

We saw two solutions:

- Separate Chaining
- Linear Probing

Collisions

Happens when the hash function give identical results to two different keys.

We saw two solutions:

- Separate Chaining
- Linear Probing

Practical efficiency will depend on the table **load factor**:
$\alpha = n/m$ $n = \#$ TOTAL OF RECORDS

Separate Chaining

Assign multiple records per cell (usually through a linked list)

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.

Separate Chaining

$0 < \alpha < 1$ $\alpha \geq 1$

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $\boxed{1 + \alpha/2}$ operations on average.

↳ ~ 1 operation

↳ $\sim \alpha = 2$

↳ 2 operations

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the *n* keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.
- Worst case $\boxed{\Theta(n)}$ only with a bad hash function (load factor is more of an issue).

Separate Chaining

Assign multiple records per cell (usually through a linked list)

- Assuming even distribution of the n keys.
- A sucessful search requires $1 + \alpha/2$ operations on average.
- An unsucessful search requires α operations on average.
- Almost same numbers for Insert and Delete.
- Worst case $\Theta(n)$ only with a bad hash function (load factor is more of an issue).
- Requires extra memory.

Linear Probing

Populate successive empty cells.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $\boxed{(1/2) \times (1 + 1/(1 - \alpha))}$ operations on average.
- An unsucessful search requires $\boxed{(1/2) \times (1 + 1/(1 - \alpha)^2)}$ operations on average.

ONLY MAKES SENSE FOR

$0 < \alpha < 1$

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1 - \alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1 - \alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1 - \alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1 - \alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.
- Does not require extra memory.

Linear Probing

Populate successive empty cells.

- Much harder analysis, simplified results show:
- A sucessful search requires $(1/2) \times (1 + 1/(1 - \alpha))$ operations on average.
- An unsucessful search requires $(1/2) \times (1 + 1/(1 - \alpha)^2)$ operations on average.
- Similar numbers for Insert. Delete virtually impossible.
- Does not require extra memory.
- Worst case $\Theta(n)$ with a bad hash function *and/or* clusters.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $\boxed{h(K)}$
- Second try: $(\underline{h(K) + s(K)}) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- ...

$$\boxed{s(K) = 1}$$

$$(h(K) + 0 \times \cancel{s}(K)) \mod m = h(K)$$

$$(h(K) + 1 \times s(K)) \mod m =$$

$$= (h(K) + 1 \times 1) \mod m = h(K) + 1$$

UNLESS
$h(K)+1 = m$
$= 0$

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $h(K)$
- Second try: $(h(K) + s(K)) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- ...

Another reason to use prime m in $h(K)$: will guarantee to find a free cell if there is one.

Double Hashing

A generalisation of Linear Probing.

Apply a second hash function in case of collision.

- First try: $h(K)$
- Second try: $(h(K) + s(K)) \mod m$
- Third try: $(h(K) + 2s(K)) \mod m$
- $\ldots$

Another reason to use prime m in $h(K)$: will guarantee to find a free cell if there is one.

Both Linear Probing and Double Hashing are sometimes referred as *Open Addressing* methods.

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).
- *Rehashing* allocates a new table (usually around double the size) and move every item from the previous table to the new one.

Rehashing

- High load factors deteriorate the performance of a hash table (for linear probing, ideally we should have $\alpha < 0.9$).
- *Rehashing* allocates a new table (usually around double the size) and move every item from the previous table to the new one.
- *Very expensive operation*, but happens infrequently.

Summary

Hash Tables:

Summary

Hash Tables:

- Implement dictionaries.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).
- Requires good *hash functions*.

Summary

Hash Tables:

- Implement dictionaries.
- Allow $\Theta(1)$ Search, Insert and Delete in the average case.
- Preallocates memory (size m).
- Requires good *hash functions*.
- Requires good collision handling.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like “give me all records with keys between 100 and 200” are easy within a BST but much less efficient in a hash table.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.
- Also: memory requirements of a hash table are much higher.

Pros and Cons

If Hash Tables are so good, why bother with BSTs?

- Hash Tables ignore *key ordering*, unlike BSTs.
- Queries like "give me all records with keys between 100 and 200" are easy within a BST but much less efficient in a hash table.
- Also: memory requirements of a hash table are much higher.

That being said, if hashing is applicable, a well-tuned hash table will typically outperform BSTs.

In Practice

Python dictionaries (*dict* type)

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

maps → COMPLEX QUERIES

- Uses chaining.
- Rehashing happens when $\alpha = 1$

In Practice

Python dictionaries (*dict* type)

- Open addressing using *pseudo-random probing*
- Rehashing happens when $\alpha = 2/3$

C++ *unordered_maps*

- Uses chaining.
- Rehashing happens when $\alpha = 1$

Next lecture: what happens if records/data is too large?

COMP20007 Design of Algorithms

Data Compression

Daniel Beck

Lecture 16

Semester 1, 2020

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.
- We assumed that records could fit in memory. (although we did mention secondary memory in Mergesort and B-trees)

→ HARD DRIVE

Introduction

- So far, we talked about speed and space performance from an algorithm point of view.
- We assumed that records could fit in memory. (although we did mention secondary memory in Mergesort and B-trees)
- What to do when records are *too large?* (videos, for instance)

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

01000010 01000001 01000111 01000111 01000101 01000100

8-bit

Fixed-length encoding

For text files, suppose each character has an fixed-size binary code.

B A G G E D

01000010 01000001 01000111 01000111 01000101 01000100

This is exactly what ASCII does.

Key insight: this coding has *redundant* information.

Run-length encoding

010000100100000101000111010001110100010101000100

Run-length encoding

010000100100000101000111010001110100010101000100

0140120150101303101303101301010130120

Run-length encoding

0100001001000000101000111010001110100010101000100

0140120150101303101303101301010130120

Character-level:

6 6

B A G G E D $\rightarrow$ B A 2 G E D

AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD

GENE SEQUENCES

4A3BAA5B8CDABCB3A4B3CD

Run-length encoding

- While not very useful for text data, it can work for some kinds of binary data.

Run-length encoding

- While not very useful for text data, it can work for some kinds of binary data.
- For text, the best algorithms move away from using fixed-length codes (ASCII).

Variable-length Encoding

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.
- Instead of a fixed number of bits per symbol, use a *variable* number:
  - More frequent symbols use less bits. E, A, n
  - Less frequent symbols use more bits. X, !

Variable-length Encoding

- **Key idea:** some symbols appear more *frequently* than others.
- Instead of a fixed number of bits per symbol, use a *variable* number:
  - More frequent symbols use less bits.
  - Less frequent symbols use more bits.
- For this scheme to work, no symbol code can be a prefix of another symbol's code.

E → 1

A → ~~10~~ 00

n → ~~01~~ ~~10~~ 01 ...

Variable-Length Encoding

Suppose we count symbols and find these numbers of occurrences:

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

Variable-Length Encoding

Suppose we count symbols and find these numbers of occurrences:

Symbol	Weight
B	4
D	5
G	10
F	12
C	14
E	27
A	28

Here are some sensible codes that we may use for symbols:

Symbol	Code
A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

Encoding a string

- Codes can be stored in a dictionary

Encoding a string

- Codes can be stored in a dictionary
- Once we have the codes, encoding is straightforward.

Encoding a string

- Codes can be stored in a dictionary
- Once we have the codes, encoding is straightforward.
- For example, to encode 'BAGGED', simply concatenate the codes for B, A, G, G, E and D:

A	11
B	0000
C	011
D	0001
E	10
F	010
G	001

B A G G E D

000011001001100001

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

0 : x
00 : x
000 : x
1 : x

11	A
0000	B
011	C
0001	D
10	E
010	F
001	G

000011001001100001

B A G G E D

Decoding a string

- To decode we can use another dictionary where keys are codes and values are symbols.
- Starting from the first digit, look in the dictionary. If not present, concatenate the next digit and repeat until code is valid.

11	A
0000	B
011	C
0001	D
10	E
010	F
001	G

000011001001100001

Seems like it requires lots of misses, is there a better way?

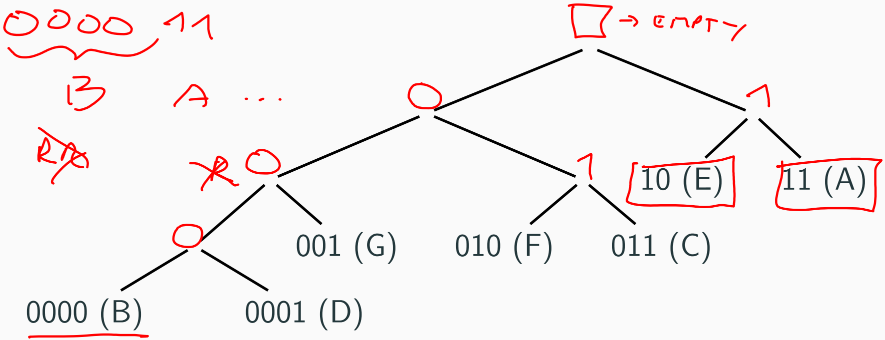
Tries

- Another implementation of a dictionary.

Tries

- Another implementation of a dictionary.
- Works when keys can be decomposed

- Another implementation of a dictionary.
- Works when keys can be decomposed

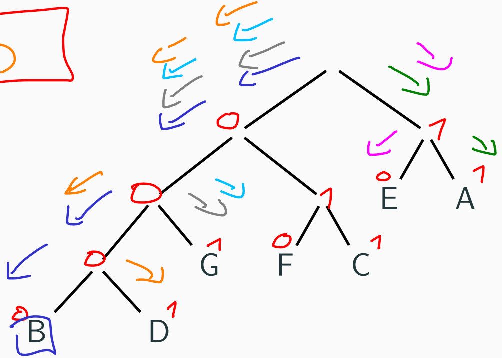


This specific trie stores values only in the leaves → keeps prefix property.

Tries

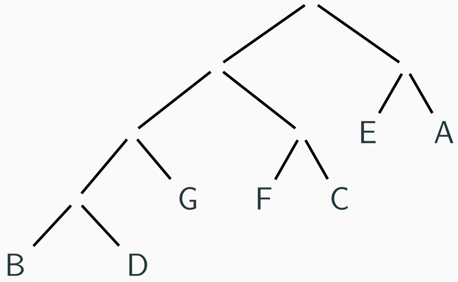
To decode 000011001001100001, use the trie, repeatedly starting from the root, and printing each symbol found as a leaf.

$\boxed{\text{BAGGED}}$



Tries

To decode 00001100100110001, use the trie, repeatedly starting from the root, and printing each symbol found as a leaf.



How to choose the codes?

Huffman Encoding

Huffman Encoding

SHORTEST
COMPRESSION IN BITS

- Goal: obtain the *optimal* encoding given symbol frequencies.

Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.

Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.
- Two nodes are *fused* if they have the *smallest* frequency.

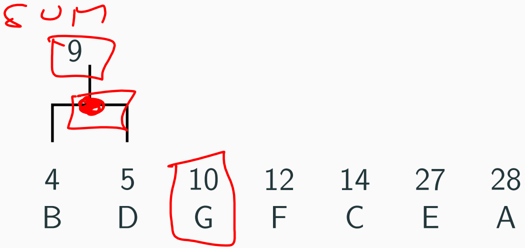
Huffman Encoding

- Goal: obtain the *optimal* encoding given symbol frequencies.
- Treat each symbol as a *leaf* and build a binary tree bottom-up.
- Two nodes are *fused* if they have the *smallest* frequency.
- The resulting tree is a Huffman tree.

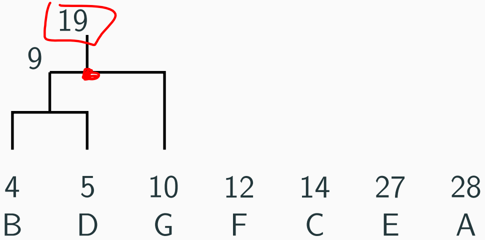
Huffman Trees

4	5	10	12	14	27	28
B	D	G	F	C	E	A

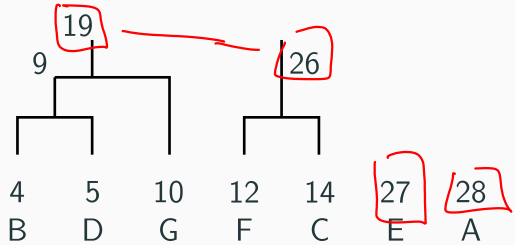
Huffman Trees



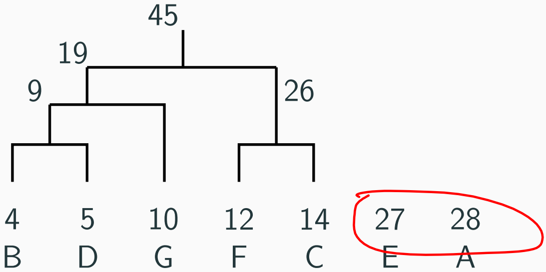
Huffman Trees



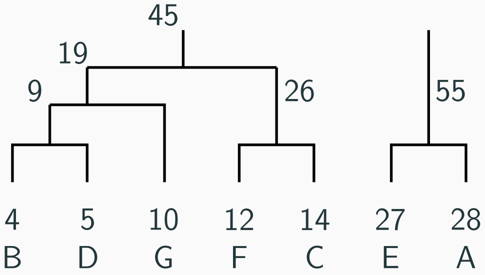
Huffman Trees



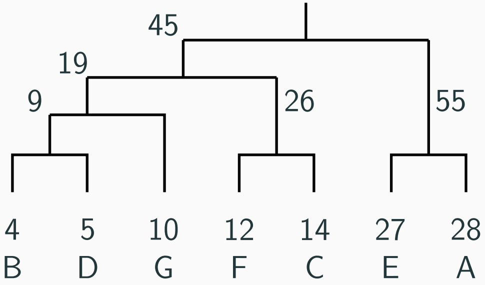
Huffman Trees



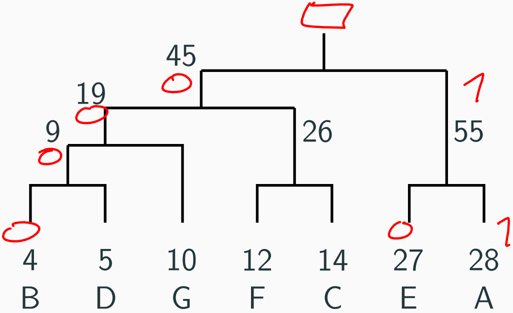
Huffman Trees



Huffman Trees



Huffman Trees



E - 10

A - 11

B - 0000

We end up with the trie from before!

The Greedy Method

Huffman is an example of the *greedy* method.

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.

The Greedy Method

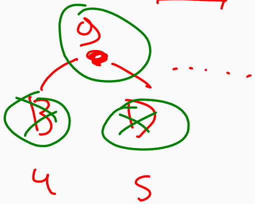
Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are “acceptable” but we want to find the *best* one.
- The problem can be divided into *local* subproblems.

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.



The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.
- For Huffman, the greedy methods finds the optimal. Dijkstra and Prim are other examples.

The Greedy Method

Huffman is an example of the *greedy* method.

- The goal is *optimisation*: many solutions are "acceptable" but we want to find the *best* one.
- The problem can be divided into *local* subproblems.
- The greedy method builds a *global* solution through *local* and *irrevocable* solutions.
- For Huffman, the greedy methods finds the optimal. Dijkstra and Prim are other examples.
- But this is not always the case.

Summary

Summary

- Most data we store in our computer has *redundancy*.

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.
- Huffman is based on variable-length encoding.

Summary

- Most data we store in our computer has *redundancy*.
- Compression uses redundancy to reduce space.
- Huffman is based on variable-length encoding.
- Tries to store codes.

↳ DECOMPOSE KEYS

↳ ANY KIND OF SYMBOLS

Data Compression - In Practice

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC. → LOSSLESS
- **Lossy compression:** JPEG, MP3, MPEG and others. Also employ Huffman (among other techniques).

Data Compression - In Practice

- Huffman encoding provides the basis for many advanced compression techniques.
- Lempel-Ziv compression assigns codes to *sequences of symbols*: used in GIF, PNG and ZIP.
- For sequential data (audio/video), an alternative is linear prediction: *predict* the next frame given the previous ones. Used in FLAC.
- **Lossy compression:** JPEG, MP3, MPEG and others. Also employ Huffman (among other techniques).

HASHING

Next lecture: how to trade memory for speed and get an $\Theta(n)$ worst case sorting algorithm...

COMP20007 Design of Algorithms

Input Enhancement Part 1: Distribution Sorting

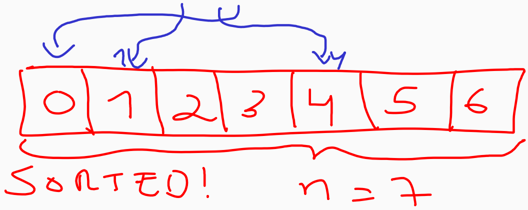
Daniel Beck

Lecture 17

Semester 1, 2020

Simple Distribution Sort

1 4 0 6 5 3 2



Simple Distribution Sort

1 4 0 6 5 3 2

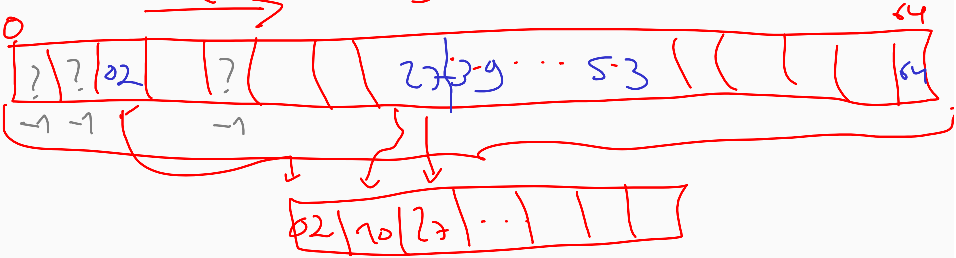
Looks $\Theta(n)$ even in worst case! Is it really?

Simple Distribution Sort

$n \rightarrow$ 10 41 02 $\boxed{64}$ 53 39 27

-1

$k = 64 + 1 = 65$



Simple Distribution Sort

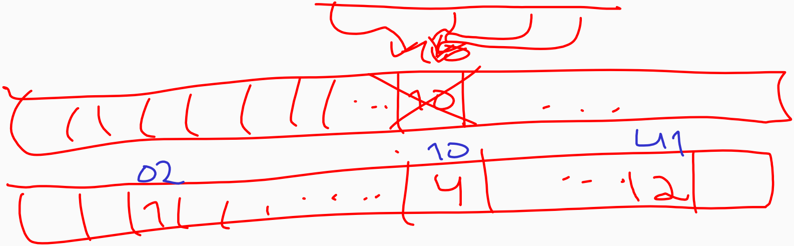
10 41 02 64 53 39 27

$k >> n$ $\Theta(k)$

$\boxed{\Theta(n+k)}$ worst case.

Simple Distribution Sort

10 41 02 10 41 10 10



Simple Distribution Sort

10 41 02 10 41 10 10

Use the auxiliary array to store counts.

Counting Sort

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

Counting Sort

$k = 9 + 1 = 10$ 6 3 3 8 1 0 8 7 9 2 5 3 5 3 1 8 7 6 5 1 2 1 5 3

WE KNOW

Key	0	1	2	3	4	5	6	7	8	9
Counts	1	4	2	5	0	4	2	2	3	1

$\Theta(k)$

Counting Sort

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

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts(shift)	0	1	4	2	5	0	4	2	2	3	1

Counting Sort

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

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts(shift)	0	1	4	2	5	0	4	2	2	3	1

Key	0	1	2	3	4	5	6	7	8	9	
Counts(acc)	0	1	5	7	12	12	16	18	20	23	24

Counting Sort

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

$\Theta(n) \longrightarrow$

Key	0	1	2	3	4	5	6	7	8	9	
Counts	1	4	2	5	0	4	2	2	3	1	
Counts(shift)	0	1	4	2	5	0	4	2	2	3	1

$\boxed{\Theta(k)} \longrightarrow$

$k \ll n \Rightarrow \Theta(n)$

$\Theta(k) \longrightarrow$

Key	0	1	2	3	4	5	6	7	8	9	
Counts(acc)	0	1 +1+4	$\boxed{5}$	7	12	12	16	18	20	23	24

$\Theta(n) \longrightarrow$ 0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7 8 8 8 9

0 1 2 3 4 5 6 7 8 9 10 11 12 . . .

Counting Sort

function Counting Sort($A[0..n-1]$)

  for $j \leftarrow 0$ **to** k **do**

    $\boxed{C[j] \leftarrow 0}$

  for $i \leftarrow 0$ **to** $n-1$ **do**

    $\boxed{C[A[i]+1] \leftarrow C[A[i]+1] + 1}$ ▷ (shift)

  for $j \leftarrow 1$ **to** k **do**

    $\boxed{C[j] = C[j] + C[j-1]}$ CUMULATIVE COUNTS

  for $i \leftarrow 0$ **to** $n-1$ **do**

    $B[C[A[i]]] \leftarrow A[i]$

    $C[A[i]] \leftarrow C[A[i]] + 1$

  return $B[\cancel{1..n}]$ $B[0..n-1]$

Counting Sort

Questions!

Counting Sort

Questions!

- Stable?
- In-place?

Counting Sort

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.
- In-place: **No**, requires $\boxed{\Theta(n + k)}$ memory.

Counting Sort

- Stable: **Yes**, but depends on how it is implemented.
- In-place: **No**, requires $\Theta(n + k)$ memory.

Take-home message: Counting Sort only works for integer keys and it works best when the key range is small.

Bucket Sort

Bucket Sort

- Generalisation of Counting Sort.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- Sort each bucket separately.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- Sort each bucket separately.
- Concatenate results.

Bucket Sort

- Generalisation of Counting Sort.
- Split data into k buckets.
- ~~Sort each bucket separately.~~ $k = K$
- Concatenate results.

If K is the maximum key value and $k = K$, then Bucket Sort becomes Counting Sort.

Bucket Sort

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

Bucket Sort

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

$k = 3$

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1 $\rightarrow \Theta(n)$

Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3 $\rightarrow \Theta(n)$

Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6 $\rightarrow \Theta(n)$

Bucket Sort

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

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1

Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3

Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6

Sorted Bucket 1 ($A[i] < 3$): 0 1 1 1 1 2 2

Sorted Bucket 2 ($3 <= A[i] < 6$): 3 3 3 3 3 5 5 5 5

Sorted Bucket 3 ($A[i] >= 6$): 6 6 7 7 8 8 8 9

Bucket Sort

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

Bucket 1 ($A[i] < 3$): 1 0 2 1 1 2 1
Bucket 2 ($3 <= A[i] < 6$): 3 3 5 3 5 3 5 5 3
Bucket 3 ($A[i] >= 6$): 6 8 8 7 9 8 7 6

Sorted Bucket 1 ($A[i] < 3$): 0 1 1 1 1 2 2
Sorted Bucket 2 ($3 <= A[i] < 6$): 3 3 3 3 3 5 5 5 5
Sorted Bucket 3 ($A[i] >= 6$): 6 6 7 7 8 8 8 9

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

Bucket Sort

function $\text{BUCKET SORT}(A[0..n-1], k)$

$K \leftarrow$ max key value

for $j \leftarrow 0$ **to** $k-1$ **do**

$\text{INITIALISE}(B[j])$

for $i \leftarrow 0$ **to** $n-1$ **do**

$\text{INSERT}(B[\lfloor k \times A[i]/K \rfloor], A[i])$

for $j \leftarrow 0$ **to** $k-1$ **do**

$\text{AUXSORT}(B[j])$

return $\text{CONCATENATE}(B[0..k-1])$

$k=3 \quad K=9$

$\frac{3 \times 2}{9} = \left\lfloor \frac{6}{9} \right\rfloor = 0$

$\frac{3 \times 8}{9} = \left\lfloor \frac{24}{9} \right\rfloor = 2$

Bucket Sort

function $\textsc{Bucket Sort}(A[0..n-1], k)$
$K \leftarrow$ max key value
for $j \leftarrow 0$ **to** $k - 1$ **do**
$\textsc{Initialise}(B[j])$
for $i \leftarrow 0$ **to** $n - 1$ **do**
$\textsc{Insert}(B[\lfloor k \times A[i]/K \rfloor], A[i])$
for $j \leftarrow 0$ **to** $k - 1$ **do**
$\textsc{AuxSort}(B[j])$
return $\textsc{Concatenate}(B[0..k-1])$

Stable?

Bucket Sort

Bucket Sort

Some properties depend on AuxSort:

Bucket Sort

Some properties depend on AUXSORT:

- Stability.

Bucket Sort

Some properties depend on $\boxed{\textsc{AuxSort}:}$

- Stability. $\Theta(n^2)$
- Worst case complexity. (assuming $k = \Theta(n)$)

Bucket Sort

Some properties depend on AUXSORT:

- Stability.
- Worst case complexity. (assuming $k = \Theta(n)$)

Average complexity: $\boxed{\Theta(n + \frac{n^2}{k} + k)}$. Linear if $k = \Theta(n)$.

$\hookrightarrow \Theta(n)$

Bucket Sort

Some properties depend on $\textsc{AuxSort}$:

- Stability.
- Worst case complexity. (assuming $k = \Theta(n)$)

Average complexity: $\Theta(n + \frac{n^2}{k} + k)$. Linear if $k = \Theta(n)$.

Take-home message: Compared to Counting Sort, Bucket Sort provides more control over how much memory to use, but it is slower in the worst case.

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011

Bucket 1: 110 000 010 110 010

Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011

Bucket 1: 000 001 101 101 001 101 001 001 101

Bucket 2: 110 010 110 010 011 011 111 011 011 111 011

000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111
000 001 001 001 001 010 010 011 011 011 011 011 101 101 101 101 110 110 111 111

Radix Sort

6 3 3 1 0 7 2 5 3 5 3 1 7 6 5 1 2 1 5 3

110 011 011 001 000 111 010 101 011 101 011 001 111 110 101 001 010 001 101 011
Bucket 1: 110 000 010 110 010
Bucket 2: 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
110 000 010 110 010 011 011 001 111 101 011 101 011 001 111 101 001 001 101 011
Bucket 1: 000 001 101 101 001 101 001 001 101
Bucket 2: 110 010 110 010 011 011 111 011 011 111 011
000 001 101 101 001 101 001 001 101 110 010 110 010 011 011 111 011 011 111 011
Bucket 1: 000 001 001 001 001 010 010 011 011 011 011 011
Bucket 2: 101 101 101 101 110 110 111 111
000 001 001 001 001 010 010 011 011 011 011 011 101 101 101 101 110 110 111 111

0 1 1 1 1 2 2 3 3 3 3 3 5 5 5 5 6 6 7 7

Radix Sort

Assumptions:

Radix Sort

Assumptions: 60 -> 6 bits

- Maximum key length is known in advance.

  -> 3 bits

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from <u>least</u> to the <u>most</u> significant digit. (also possible to do in reverse)

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit. (also possible to do in reverse)
By using Bucket ~~Sort~~ with max buckets we have guaranteed $\Theta(n)$ performance per pass.

Radix Sort

Assumptions:

- Maximum key length is known in advance.
- Keys can be sorted in **lexicographical** order (strings).

Start sorting from least to the most significant digit. (also possible to do in reverse)
By using Bucket Sort with max buckets we have guaranteed $\Theta(n)$ performance per pass.
Total worst case performance is $\Theta(n \times len(k))$

Radix Sort

function $\textsc{Radix Sort}(A[0..n-1], k)$

  for $j \leftarrow 0$ **to** $len(k)$ **do**

    $A \leftarrow \boxed{\textsc{AuxSort}(A, k[j])}$

$\Theta(n \cdot len(k))$

↳ BUCKET SORT WITH MAX BUCKET

↳ $\Theta(n)$

Radix Sort

function $\text{Radix Sort}(A[0..n-1], k)$
  for $j \leftarrow 0$ **to** $len(k)$ **do**
    $A \leftarrow \text{AuxSort}(A, k[j])$

- Typically, AuxSort is Bucket Sort with max buckets but can be any sorting algorithm as long as it is stable (why?)

Radix Sort

function $\textsc{Radix Sort}(A[0..n-1], k)$
    for $j \leftarrow 0$ **to** $len(k)$ **do**
        $A \leftarrow \textsc{AuxSort}(A, k[j])$

- Typically, $\textsc{AuxSort}$ is Bucket Sort with max buckets but can be any sorting algorithm as long as it is stable (why?)

Take-home message: Radix Sort can be very fast (faster than comparison sorting) if keys are short (need to known in advance).

Summary

- Distribution Sorting is a sorting paradigm that trades memory for speed.
- Relies on more assumptions, unlike Comparison Sorting algorithms:
    - Counting Sort, Bucket Sort: positive integer keys, with max bound known.
    - Radix Sort: more general but max key length must be known and keys should have lexicographical order.

In Practice

In Practice

- Distribution Sort is not as widely used as Comparison Sort:

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).
- However, they can be very useful as part of a more complex algorithm.

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).

k

KNOWN

- However, they can be very useful as part of a more complex algorithm.

$\boxed{\Theta(n)}$

STRINGS

  - Radix Sort is used to construct suffix arrays.
  - Controlled environment with guaranteed short key size.

In Practice

- Distribution Sort is not as widely used as Comparison Sort:
  - Less general.
  - Require more memory.
  - In practice, good sorting algorithms can be very close to $\Theta(n)$ already (Timsort).
- However, they can be very useful as part of a more complex algorithm.
  - Radix Sort is used to construct suffix arrays.
  - Controlled environment with guaranteed short key size.

Next lecture: string matching revisited.

COMP20007 Design of Algorithms

Input Enhancement Part 2: String Searching

Daniel Beck

Lecture 18

Semester 1, 2020

String Search - Recap

String Search - Recap

- Goal: given text of size n, find a string (pattern) of size m.

String Search - Recap

- Goal: given text of size n, find a string (pattern) of size m.
- Brute force algorithm: $O(m \times n)$.

String Search - Brute Force

NOT EVEN IN THE PATTERN

TEXT

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

PATTERN

B A R B E R

BARBER

BARBER

...

BARBER

BARBER

Input Enhancement

- Longer shifts can be made if we know statistics about the pattern.

Input Enhancement

- Longer shifts can be made if we know statistics about the pattern.
- Same way we could sort arrays more efficiently by knowing statistics about the array (Counting Sort).

Input Enhancement

- Longer shifts can be made if we know statistics about the pattern.
- Same way we could sort arrays more efficiently by knowing statistics about the array (Counting Sort).
- The Horspool's algorithm use this idea to make string search faster.

Input Enhancement

- Longer shifts can be made if we know statistics about the pattern.
- Same way we could sort arrays more efficiently by knowing statistics about the array (Counting Sort).
- The Horspool's algorithm use this idea to make string search faster.
- Key idea: scan the text from left to right but scan the pattern from right to left.

Longer shifts - Case 1

The last character is not in the pattern.

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

  B A R B E R
```

BARBER

Shift the whole pattern.

Longer shifts - Case 2

W → 6 positions

A → 4 positions

The last character does not match but it is in the pattern.

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P

B A R B E R

BARBER

Shift the pattern until the last occurence of the character.

Longer shifts - Case 3

The last character matches but one of the $m - 1$ characters does not match and the last character is unique.

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
  S E E S A W
```

SEESAW

Shift the whole pattern.

Longer shifts - Case 4

The last character matches but one of the $m - 1$ characters does not match and the last character is not uniques

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
                                    x |
                  R E O R D E R
                        REORDER
```

Shift the pattern until the last occurence of the character.

Horspool - Preprocessing

W - 6 shifts
A - 4 shifts

- The number of allowed shifts depends on the character type only.

Horspool - Preprocessing

- The number of allowed shifts depends on the character type only.
- Horspool builds a dictionary with all characters in the alphabet and their corresponding allowed skips for a pattern.

Horspool - Preprocessing

SEESAW

6

- The number of allowed shifts depends on the character type only.
- Horspool builds a dictionary with all characters in the alphabet and their corresponding allowed skips for a pattern.

BARBER

2 4 3 2 1 3

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Horspool - FindShifts

m = 6

BARBER

ALPHABET SIZE 26+1 = 27 '_'

function FindShifts($P[0..m-1]$)

  for $i \leftarrow 0$ to a **do**

    $Shift[i] \leftarrow m$

  for $j \leftarrow 0$ to $m-2$ **do**

    $Shift[P[j]] \leftarrow m-(j+1)$

6 − 0+1 = 5

STESAW

SHIFT

A	~~6~~ 4
B	~~4~~ ~~5~~ 2
C	6
D	6
E	~~6~~ 1
F	6
⋮	
R	~~6~~ 3
⋮	
Z	6
'_'	6

Horspool - Algorithm

function HORSPOOL($P[0..m-1]$, $T[0..n-1]$)

SHIFT ← $\boxed{\text{FINDSHIFTS}(P)}$

$i \leftarrow m - 1$ → LAST

while $i < n$ **do**

$k \leftarrow 0$

END OF PATTERN MISMATCH

while $k < m$ **and** $P[m-1-k] = T[i-k]$ **do**

$\boxed{k \leftarrow k + 1}$

if $k = m$ **then** ▷ We have a match

$\boxed{\textbf{return } i - m + 1}$ ▷ Start of the match

else

$\boxed{i \leftarrow i + Shift[T[i]]}$ ▷ Slide the pattern along

return -1

Horspool - Properties

- Worst-case still $O(m \times n)$.
- For random strings, it's linear and faster in practice compared to the brute force version.

Other String Search algorithms

- Boyer-Moore: extends Horspool to allow shifts based on suffixes.

Other String Search algorithms

- Boyer-Moore: extends Horspool to allow shifts based on suffixes.
- Knuth-Morris-Pratt: also preprocess the patten but builds a finite-state automaton.

Other String Search algorithms

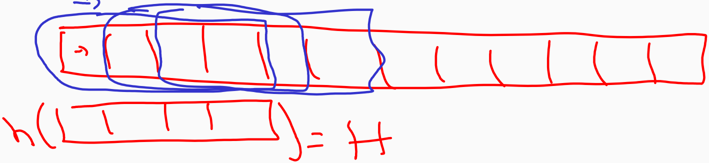
$\boxed{h \neq H}$ → MOST OF THE CASES

$h = H$ { COLLISION / TRUE MATCH

$h(T[i..j]) = f(h(T[i-1..j-1])) + f(j)$

- Boyer-Moore: extends Horspool to allow shifts based on suffixes.
- Knuth-Morris-Pratt: also preprocess the patten but builds a finite-state automaton.
- Rabin-Karp: uses hash functions to filter negative matches.
  - Needs a rolling hash function to be efficient.

$h(T[0..n-1])$



$\sim O(n)$

Summary and Practical Uses

- String search algorithms can be sped up by input enhancement.

Summary and Practical Uses

- String search algorithms can be sped up by input enhancement.
- Allocates extra memory to preprocess the pattern.

Summary and Practical Uses

- String search algorithms can be sped up by input enhancement.
- Allocates extra memory to preprocess the pattern.
- Horspool uses a dictionary of shifts.

Summary and Practical Uses

- String search algorithms can be sped up by input enhancement.
- Allocates extra memory to preprocess the pattern.
- Horspool uses a dictionary of shifts.
- The Boyer-Moore extension is one of the most used string searching algorithms, for instance in the “grep” Linx command line tool.

Summary and Practical Uses

- String search algorithms can be sped up by input enhancement.
- Allocates extra memory to preprocess the pattern.
- Horspool uses a dictionary of shifts.
- The Boyer-Moore extension is one of the most used string searching algorithms, for instance in the “grep” Linx command line tool.

Next week: trade memory for speed by storing intermediate solutions.

COMP20007 Design of Algorithms

Dynamic Programming Part 1: Warshall and Floyd algorithms

Daniel Beck

Lecture 19

Semester 1, 2020

Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

Fibonacci Numbers

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ \ldots$$

$$\boxed{\begin{gathered} F(n) = F(n-1) + F(n-2), \qquad n > 1, \\ F(0) = 1, \qquad F(1) = 1. \end{gathered}}$$

Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

$$F(n) = F(n-1) + F(n-2), \qquad n > 1,$$
$$F(0) = 1, \qquad F(1) = 1.$$

function $\textsc{Fibonacci}(n)$
    if $n == 0$ or $n == 1$ **then return** 1
    return $\textsc{Fibonacci}(n-1) + \textsc{Fibonacci}(n-2)$

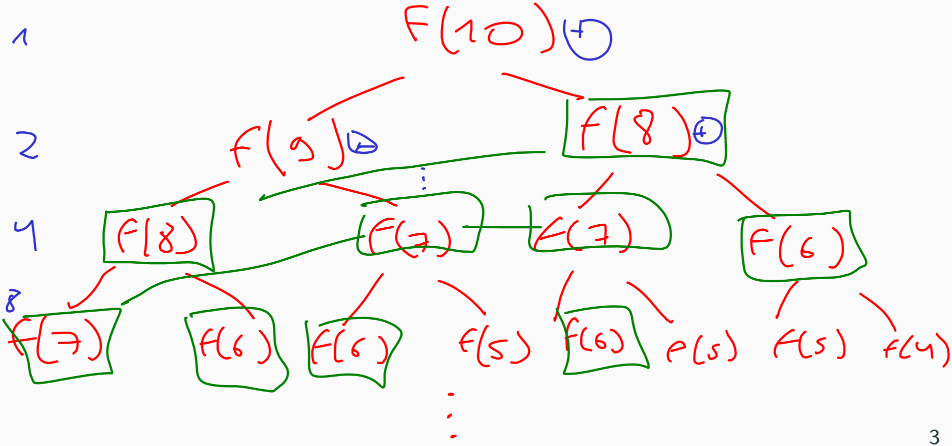
Fibonacci Numbers

function FIBONACCI(n)   10TH

2^n

if $n == 0$ or $n == 1$ **then return** 1

return FIBONACCI($n-1$) + FIBONACCI($n-2$)



Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

function $\textsc{FibonacciDP}(n)$

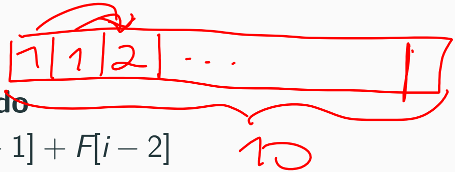
  $F[0] \leftarrow 1$

  $F[1] \leftarrow 1$

  for $i = 2$ to n **do**

    $F[i] = F[i-1] + F[i-2]$

  return $F[n]$



Storing Intermediate Solutions

- Allocate an array of size n to store previous solutions.

function $\textsc{FibonacciDP}(n)$
- $F[0] \leftarrow 1$
- $F[1] \leftarrow 1$
- **for** $i = 2$ to n **do**
  - $F[i] = F[i-1] + F[i-2]$
- **return** $F[n]$

- From exponential to linear complexity.

Dynamic Programming

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).
- Solutions to subproblems can **overlap** (calls to F for all values lesser than n).
    - Allocates extra memory to store solutions to subproblems.

Dynamic Programming

- The solution to a problem can be broken into solutions to subproblems (recurrence relations).
- Solutions to subproblems can **overlap** (calls to F for all values lesser than n).
  - Allocates extra memory to store solutions to subproblems.
- DP is mostly related to optimisation problems (but not always, see Fibonacci).
  - Optimal solution should be obtained through optimal solutions to subproblems (not always the case).

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems. $i \to \cdots k \to \cdots j$
    - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.

Transitive Closure using DP

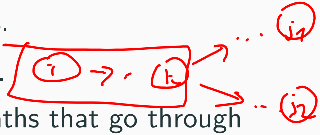
Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.
- Solutions to subproblems can overlap.

Transitive Closure using DP

Goal: find all node pairs that have a path between them.

- The solution to a problem can be broken into solutions to subproblems.
  - If there's a path between two nodes i and j which are not directly connected, that path has to go through at least another node k. Therefore, we only need to find if the pairs (i,k) and (k,j) have paths.
- Solutions to subproblems can overlap.



  - If the pairs (i,j_1) and (i,j_2) have paths that go through k, then finding if the pair (i,k) has a path is part of the solutions for <u>both</u> problems.

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

R^n

$$\begin{aligned}\boxed{R^0_{ij}} &= A[i,j]\\ \boxed{R^k_{ij}} &= \underset{\substack{1\\0}}{R^{k-1}_{ij}}\ \boxed{\text{or}}\ \underbrace{(\underset{1}{R^{k-1}_{ik}} \text{ and } \underset{1}{R^{k-1}_{kj}})}_{1}\end{aligned}$$

Warshall's Algorithm

- Assume nodes can be numbered from 1 to n, with A being the adjacency matrix.

$$
\begin{aligned}
R^0_{ij} &= A[i,j] \\
R^k_{ij} &= R^{k-1}_{ij} \text{ or } (R^{k-1}_{ik} \text{ and } R^{k-1}_{kj})
\end{aligned}
$$

function WARSHALL($A[1..n, 1..n]$)
    $R^0 \leftarrow A$
    for $k \leftarrow 1$ to n **do** R^k
        for $i \leftarrow 1$ to n **do**
            for $j \leftarrow 1$ to n **do**
                $R^k[i,j] \leftarrow R^{k-1}[i,j]$ **or** $(R^{k-1}[i,k]$ **and** $R^{k-1}[k,j])$
    return R^n

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.
- Namely, if $R^{k-1}[i,k]$ (that is, $A[i,k]$) is 0 then the assignment is doing nothing. And if it is 1, and if $A[k,j]$ is also 1, then $A[i,j]$ gets set to 1. $0 \to 1$ ✓

  ~~$1 \to 0$ ✗~~

Warshall's Algorithm

- We can allow input A to be used for the output, simplifying things.
- Namely, if $R^{k-1}[i, k]$ (that is, $A[i, k]$) is 0 then the assignment is doing nothing. And if it is 1, and if $A[k, j]$ is also 1, then $A[i, j]$ gets set to 1.

for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        for $j \leftarrow 1$ to n **do**
            if $\boxed{A[i, k]}$ **then**
                if $A[k, j]$ **then**
                    $A[i, j] \leftarrow 1$

Warshall's Algorithm

- Now we notice that $A[i, k]$ does not depend on j, so testing it can be moved outside the innermost loop.

Warshall's Algorithm

- Now we notice that $A[i,k]$ does not depend on j, so testing it can be moved outside the innermost loop.

for $k \leftarrow 1$ to n **do**
    for $i \leftarrow 1$ to n **do**
        if $A[i,k]$ **then**
            for $j \leftarrow 1$ to n **do**
                if $A[k,j]$ **then**
                    $A[i,j] \leftarrow 1$

Warshall's Algorithm

- Now we notice that $A[i,k]$ does not depend on j, so testing it can be moved outside the innermost loop.

```
for k ← 1 to n do
    for i ← 1 to n do
        if A[i, k] then
            for j ← 1 to n do
                if A[k, j] then
                    A[i, j] ← 1
```

$$\begin{bmatrix}0\\0\\1\\1\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\\0\\1\\0\end{bmatrix} \to \begin{bmatrix}0\\1\\1\\1\\0\end{bmatrix}$$

- Can use bitstring operations.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\boxed{\Theta(n^3)}$ in all cases.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\Theta(n^3)$ in all cases.
- In practice:
    - Ideal for dense graphs.

Analysis of Warshall's Algorithm

- Straightforward analysis: $\Theta(n^3)$ in all cases.
- In practice:
  - Ideal for dense graphs.
  - Not the best for sparse graphs (#edges $\in O(n)$): DFS from each node tends to perform better.

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.
- Similar to Warshall's, but uses a weight matrix W instead of adjacency matrix A (with $\boxed{\infty}$ values for missing edges)

Floyd's Algorithm: All-Pairs Shortest-Paths

- Floyd's algorithm solves the all-pairs shortest-path problem for weighted graphs with positive weights.
- Similar to Warshall's, but uses a weight matrix W instead of adjacency matrix A (with ∞ values for missing edges)
- It works for directed as well as undirected graphs.

Floyd's Algorithm

- The recurrence follows Warshall's closely:

Floyd's Algorithm

- The recurrence follows Warshall's closely:

D^n

$$
\begin{aligned}
D_{ij}^{0} &= W[i,j] \\
D_{ij}^{k} &= min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})
\end{aligned}
$$

Floyd's Algorithm

- The recurrence follows Warshall's closely:

$$
\begin{aligned}
D_{ij}^{0} &= W[i,j] \\
D_{ij}^{k} &= min(D_{ij}^{k-1}, D_{ik}^{k-1} + D_{kj}^{k-1})
\end{aligned}
$$

function $\textsc{Floyd}(W[1..n, 1..n])$

  $D \leftarrow W$

  for $k \leftarrow 1$ to n **do** $\Theta(n^3)$

    for $i \leftarrow 1$ to n **do**

      for $j \leftarrow 1$ to n **do**

        $D[i,j] \leftarrow min(D[i,j], D[i,k] + D[k,j])$

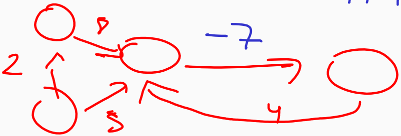
  return D

Negative weights

- Negative weights are not necessarily a problem, but negative cycles are.

Negative weights

$-7+4-7+4\ldots \Rightarrow -\infty$



- Negative weights are not necessarily a problem, but <u>negative cycles</u> are.
- These trigger arbitrarily low values for the paths involved.

Negative weights

- Negative weights are not necessarily a problem, but negative cycles are.
- These trigger arbitrarily low values for the paths involved.
- Floyd's algorithm can be adapted to detect negative cycles (by looking if diagonal values become negative).

Summary

- Dynamic programming is another technique that trades memory for speed.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.

Summary

- Dynamic programming is another technique that trades memory for speed.
    - Breaks into subproblems.
    - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.
- Floyd's algorithm: all-pairs shortest paths.

Summary

- Dynamic programming is another technique that trades memory for speed.
  - Breaks into subproblems.
  - Store overlapping solutions in memory.
- Warshall's algorithm: find the transitive closure of a graph.
- Floyd's algorithm: all-pairs shortest paths.

Next lecture: Dynamic Programming part 2.

COMP20007 Design of Algorithms

Dynamic Programming Part 2: Knapsack Problem

Daniel Beck

Lecture 20

Semester 1, 2020

The Knapsack Problem

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity $\boxed{W}$

find the most valuable selection of items that will fit in the knapsack.

The Knapsack Problem

Given n items with

- weights: $w_1, w_2, \ldots, w_n$
- values: $v_1, v_2, \ldots, v_n$
- knapsack of capacity W

find the most valuable selection of items that will fit in the knapsack.

We assume that all entities involved are positive integers.

Example 2: The Knapsack Problem

Express the solution recursively:

$$K(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Otherwise:

$$K(i,j) = \begin{cases} max(K(i-1,j), K(i-1,j-w_i) + v_i) & \text{if } j \geq w_i \\ K(i-1,j) & \text{if } j < w_i \end{cases}$$

Example 2: The Knapsack Problem

Express the solution recursively:

$$K(i,j) = 0 \text{ if } i = 0 \text{ or } j = 0$$

Otherwise:

$$K(i,j) = \begin{cases} max(K(i-1,j), K(i-1,j-w_i) + v_i) & \text{if } j \geq w_i \\ K(i-1,j) & \text{if } j < w_i \end{cases}$$

For a bottom-up solution we need to write the code that systematically fills a two-dimensional table.

The table will have $n+1$ rows and $W+1$ columns.

Example 2: The Knapsack Problem

function $\textsc{Knapsack}(v[1..n], w[1..n], W)$

    for $i \leftarrow 0$ to n **do** $K[i, 0] \leftarrow 0$

    for $j \leftarrow 1$ to W **do** $K[0, j] \leftarrow 0$

    for $i \leftarrow 1$ to n **do**

        for $j \leftarrow 1$ to W **do**

            if $j < w_i$ **then**

                $K[i, j] \leftarrow K[i-1, j]$

            else

                $K[i, j] \leftarrow max(K[i-1, j], K[i-1, j-w_i] + v_i)$

    return $K[n, W]$

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.
- In this situation, a top-down approach, with memoing, is preferable.

Solving the Knapsack Problem with Memoing

- To some extent the bottom-up (table-filling) solution is overkill: It finds the solution to every conceivable sub-instance.
- Most entries cannot actually contribute to a solution.
- In this situation, a top-down approach, with memoing, is preferable.
- To keep the memo table small, make it a hash table.

Solving the Knapsack Problem with Memoing

function $\textsc{Knap}$(i,j) $\triangleright$ Uses a global hashtable

  if $i = 0$ or $j = 0$ **then**

    return 0

  if key (i, j) is in hashtable **then**

    return the corresponding value (that is, $K(i, j)$)

  if $j < w_i$ **then**

    $k \leftarrow \textsc{Knap}(i - 1, j)$

  else

    $k \leftarrow max(\textsc{Knap}(i - 1, j), \textsc{Knap}(i - 1, j - w_i) + v_i)$

  insert k into hashtable, with key (i, j)

  return k

Knapsack - Complexity

- Time (and space) complexity of Knapsack is $\boxed{\Theta(nW)}$

Knapsack - Complexity

- Time (and space) complexity of Knapsack is $\Theta(n\boxed{W})$
- This is called pseudopolynomial time: the algorithm is polynomial in the value of the input, not its length.
  - Counting Sort is another example. $\Theta(n + W)$

Knapsack - Complexity

$\Theta(n^2)$ $\Theta((32b)^2) = \Theta(1024\, b^2) = \Theta(b^2)$

$\boxed{\text{32-bit}}$ → ALL INTS HAVE CONSTANT BIT LENGTH

integers

- Time (and space) complexity of Knapsack is $\Theta(nW)$
- This is called pseudopolynomial time: the algorithm is polynomial in the value of the input, not its length.
  - Counting Sort is another example.
- Pseudopolynomial is not in general polynomial because it is exponential in the number of bits.

$W =$

len = 5

11110 value = 30

$\boxed{1}$11110 len = 6 value $= \Theta(2^\ell)$

value = 62

Summary

- Dynamic Programming recipe:

Summary

- Dynamic Programming recipe:
    - Split into subproblems.
    - Solutions overlap.

STORE SOLUTIONS

Summary

- Dynamic Programming recipe:
    - Split into subproblems.
    - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two “variables” to split the problem.
  - Can use memoing to speed it up.

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two "variables" to split the problem.
  - Can use memoing to speed it up.

Next lecture: the last one! (before the revision)

Summary

- Dynamic Programming recipe:
  - Split into subproblems.
  - Solutions overlap.
- A DP solution for Knapsack results in a pseudopolynomial time algorithm.
  - Uses two "variables" to split the problem.
  - Can use memoing to speed it up.

Next lecture: the last one! (before the revision)

$$\boxed{\mathrm{P} = \mathrm{NP}?}$$

COMP20007 Design of Algorithms

Complexity Theory

Daniel Beck

Lecture 21

Semester 1, 2020

Complexity Theory

- So far, we have been concerned with the analysis of algorithms' running times.

Complexity Theory

- So far, we have been concerned with the analysis of algorithms' running times.
- Complexity theory asks a different question: "What is the inherent difficulty of the problem?"

Complexity Theory

- So far, we have been concerned with the analysis of algorithms' running times.
- Complexity theory asks a different question: "What is the inherent difficulty of the problem?"
- It does however also uses asymptotic notation, although usually more concerned with lower bounds (Ω notation).

Complexity Theory

- So far, we have been concerned with the analysis of algorithms' running times.
- Complexity theory asks a different question: "What is the inherent difficulty of the problem?"
- It does however also uses asymptotic notation, although usually more concerned with <u>lower bounds</u> (Ω notation).
- Tigther ("larger") lower bounds give us guarantees on best possible algorithms.

Complexity Theory - Examples

Comparison Sorting (worst case)

Complexity Theory - Examples

Comparison Sorting (worst case)

- A trivial lower bound: $\Omega(n)$.

Complexity Theory - Examples

Comparison Sorting (worst case)

n! POSSIBLE PERMUTATIONS

- A trivial lower bound: $\Omega(n)$.
- A less trivial lower bound: $\Omega(\log n!) \approx \boxed{\Omega(n \log n)}$
  - Can be found by using a technique called Decision Trees.

BINARY TREE

WITH n! LEAVES

$h = \log(\# \text{ LEAVES})$

$h = \log(n!) \simeq \Omega(n \log n)$

Complexity Theory - Examples

Comparison Sorting (worst case)

- A trivial lower bound: $\Omega(n)$.
- A less trivial lower bound: $\Omega(\log n!) \approx \Omega(n \log n)$
    - Can be found by using a technique called Decision Trees.
- Bound is tight: we know $\Theta(n \log n)$ algorithms.

Complexity Theory - Examples

Comparison Sorting (worst case)

- A trivial lower bound: $\Omega(n)$.
- A less trivial lower bound: $\Omega(\log n!) \approx \Omega(n \log n)$
  - Can be found by using a technique called Decision Trees.
- Bound is tight: we know $\Theta(n \log n)$ algorithms.

Matrix Multiplication SQUARE $n \times n$

- A trivial lower bound: $\Omega(n^2)$

Complexity Theory - Examples

Comparison Sorting (worst case)

- A trivial lower bound: $\Omega(n)$.
- A less trivial lower bound: $\Omega(\log n!) \approx \Omega(n \log n)$
  - Can be found by using a technique called Decision Trees.
- Bound is tight: we know $\Theta(n \log n)$ algorithms.

Matrix Multiplication

STRASSEN'S ALG

$O(n^{2.87})$

- A trivial lower bound: $\Omega(n^2)$
- Unknown if bound is tight: best algorithms are $\boxed{O(n^{2.37})}$

Complexity Theory - Examples

Comparison Sorting (worst case)

- A trivial lower bound: $\Omega(n)$.
- A less trivial lower bound: $\Omega(\log n!) \approx \Omega(n \log n)$
  - Can be found by using a technique called Decision Trees.
- Bound is tight: we know $\Theta(n \log n)$ algorithms.

Matrix Multiplication

- A trivial lower bound: $\Omega(n^2)$
- Unknown if bound is tight: best algorithms are $O(n^{2.37})$

This lecture: discussion about "hardness" of problems.

Decision Problems

- A decision problem takes an input and generates a **YES** or a **NO** answer as the output.

Decision Problems

- A decision problem takes an input and generates a **YES** or a **NO** answer as the output.
    - "Is the integer n a prime number?"

Decision Problems

- A decision problem takes an input and generates a **YES** or a **NO** answer as the output.
  - "Is the integer n a prime number?"
  - "Is there a circuit that visits all nodes in a graph exactly once?" (Hamiltonian Circuit Problem)

Decision Problems

- A decision problem takes an input and generates a **YES** or a **NO** answer as the output.
  - "Is the integer *n* a prime number?"
  - "Is there a circuit that visits all nodes in a graph exactly once?" (Hamiltonian Circuit Problem)
- Optimisation problems can be framed as a sequence of decision problems:

Decision Problems

- A decision problem takes an input and generates a **YES** or a **NO** answer as the output.
  - "Is the integer n a prime number?"
  - "Is there a circuit that visits all nodes in a graph exactly once?" (Hamiltonian Circuit Problem)
- Optimisation problems can be framed as a sequence of decision problems:
  - Knapsack: "Is there a set of items of values at least i and weight at most j?"

Verification Problems

- A verification problem takes an input, a proposed solution and verifies if the solution satisfy the input.

Verification Problems

- A verification problem takes an input, a proposed solution and verifies if the solution satisfy the input.
    - "Given n and $\{i_1, i_2, \ldots, i_k\}$, check if $\prod i = n$"

↳ TRUE → n is not prime

↳ FALSE → ? ?

Verification Problems

- A verification problem takes an input, a proposed solution and verifies if the solution satisfy the input.
  - "Given n and $\{i_1, i_2, \ldots, i_k\}$, check if $\prod i = n$"
  - "Given a graph G and a sequence of nodes $\{v_1, v_2, \ldots, v_n\}$, verify if there is a ~~path~~ that follows that sequence.

CIRCUIT

Verification Problems

- A verification problem takes an input, a proposed solution and verifies if the solution satisfy the input.
  - "Given n and $\{i_1, i_2, \ldots, i_k\}$, check if $\prod i = n$"
  - "Given a graph G and a sequence of nodes $\{v_1, v_2, \ldots, v_n\}$, verify if there is a path that follows that sequence.

Now we can define what is P and what is NP.

P and NP

- A problem is in P if its decision version has a solution which is polynomial in the input size.

P and NP

- A problem is in *P* if its decision version has a solution which is polynomial in the input size.
- A problem is in *NP* if its verification version has a solution which is polynomial in the input size.

P and NP

- A problem is in *P* if its decision version has a solution which is polynomial in the input size.
- A problem is in *NP* if its verification version has a solution which is polynomial in the input size.
- We can turn a verification problem into a decision problem:

P and NP

- A problem is in *P* if its decision version has a solution which is polynomial in the input size.
- A problem is in *NP* if its verification version has a solution which is polynomial in the input size.
- We can turn a verification problem into a decision problem:

  TURING

  - 1) A non-deterministic "machine" generates a candidate.
  - 2) The verification algorithm verifies the solution.
  - 3) Repeat until verified.

P and NP

- A problem is in *P* if its decision version has a solution which is polynomial in the input size.
- A problem is in *NP* if its verification version has a solution which is polynomial in the input size.
- We can turn a verification problem into a decision problem:
  - 1) A non-deterministic "machine" generates a candidate.
  - 2) The verification algorithm verifies the solution.
  - 3) Repeat until verified.
- In other words, a problem is in *NP* if its decision version has a solution which is non-deterministically polynomial in the input size.

P and NP

- A problem is in P if its decision version has a solution which is polynomial in the input size.
- A problem is in NP if its verification version has a solution which is polynomial in the input size.
- We can turn a verification problem into a decision problem:
  - 1) A non-deterministic "machine" generates a candidate.
  - 2) The verification algorithm verifies the solution.
  - 3) Repeat until verified.
- In other words, a problem is in NP if its decision version has a solution which is non-deterministically polynomial in the input size.

That's where the N in NP comes from. =)

P and NP

- Any problem in P is in NP ($P \subset NP$).

P and NP

- Any problem in P is in NP ($P \subset NP$).
    - One can verify a solution by solving the decision version and comparing the result.

P and NP

- Any problem in P is in NP ($P \subset NP$).
  - One can verify a solution by solving the decision version and comparing the result.
- The reverse is unknown... ($P \overset{?}{\supset} NP$)

P and NP

- Any problem in P is in NP ($P \subset NP$).
  - One can verify a solution by solving the decision version and comparing the result.
- The reverse is unknown... ($P \overset{?}{\supset} NP$)
  - The non-deterministic step does not guarantee efficiency.

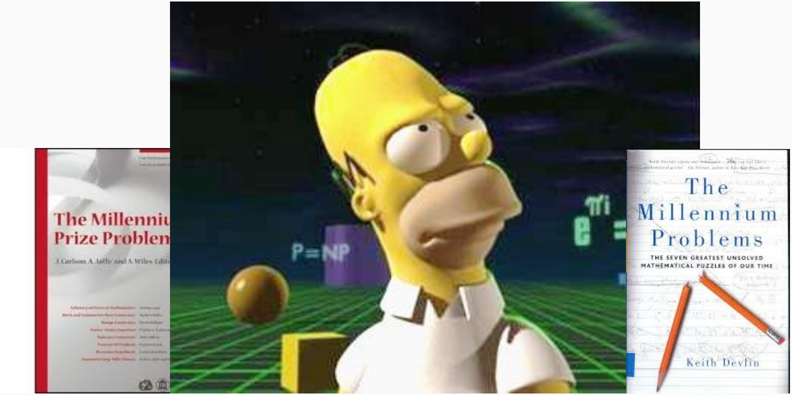
P and NP

- Any problem in P is in NP ($P \subset NP$).
  - One can verify a solution by solving the decision version and comparing the result.
- The reverse is unknown... ($P \stackrel{?}{\supset} NP$)
  - The non-deterministic step does not guarantee efficiency.

$$\boxed{\mathbf{P} \stackrel{?}{=} \mathbf{NP}}$$

A Million Dollar Question: Is P = NP?

This is one of the seven "millennium problems": The Clay Institute's seven most important unsolved mathematical problems.



Reductions

- It's a daunting task to find and prove bounds for every new problem.

Reductions

- It's a daunting task to find and prove bounds for every new problem.
- Reductions allow us to ease this by, roughly, framing a problem as equivalent to another one we know the class.

Reductions

- It's a daunting task to find and prove bounds for every new problem.
- Reductions allow us to ease this by, roughly, framing a problem as equivalent to another one we know the class.
- For instance, the Hamiltonian Circuit (HAM) problem can be reduced to the decision version of TSP.

Reductions

- It's a daunting task to find and prove bounds for every new problem.
- Reductions allow us to ease this by, roughly, framing a problem as equivalent to another one we know the class.
- For instance, the Hamiltonian Circuit (HAM) problem can be reduced to the decision version of TSP.
- The reduction function is polynomial. Therefore, since HAM is in *NP*, the decision version of TSP is also in *NP*.

From HAM to TSP

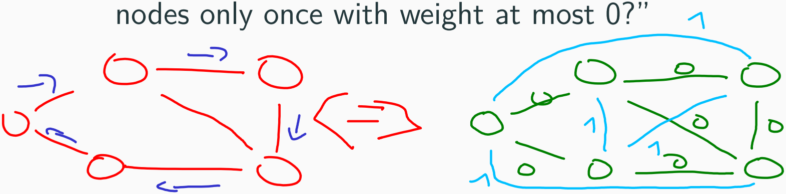
- Suppose we have a Hamiltonian Circuit in a graph G with n nodes.

From HAM to TSP

- Suppose we have a Hamiltonian Circuit in a graph G with n nodes.
- Build a new graph G' where connected nodes in G have an edge of weight 0 and non-connected nodes have weight 1.
    - This can be done in polynomial time.

From HAM to TSP

- Suppose we have a Hamiltonian Circuit in a graph G with n nodes.
- Build a new graph G' where connected nodes in G have an edge of weight 0 and non-connected nodes have weight 1.
  - This can be done in polynomial time.
- Frame Decision-TSP as "Is there a circuit that visit all nodes only once with weight at most 0?"



NP-Completeness

A decision problem D is said to be NP-Complete if:

NP-Completeness

A decision problem D is said to be *NP*-Complete if:

- $D \in NP$ **and**

NP-Completeness

A decision problem D is said to be *NP*-Complete if:

- $D \in NP$ **and**
  - Every problem in NP has a polynomial reduction to D **or**
  - A polynomial reduction from a known NP-complete problem to D exists.

NP-Completeness

A decision problem D is said to be *NP*-Complete if:

- $D \in NP$ **and**
  - Every problem in NP has a polynomial reduction to D **or**
  - A polynomial reduction from a known NP-complete problem to D exists.

Key property: if one finds a polynomial time algorithm to solve an NP-complete problem, then $\boxed{P = NP.}$

3-SAT

Proving that every problem in NP has a polynomial reduction to D is hard.

- This feat was accomplisehd in the 70's by Stephen Cook and Leonid Levin for the Boolean 3-satisfiability problem.

3-SAT

Proving that every problem in NP has a polynomial reduction to D is hard.

- This feat was accomplisehd in the 70's by Stephen Cook and Leonid Levin for the Boolean 3-satisfiability problem.
- "Given a boolean formula with a maximum of three literals, is there an assignment that results in TRUE?"

3-SAT

Proving that every problem in *NP* has a polynomial reduction to *D* is hard.

- This feat was accomplisehd in the 70's by Stephen Cook and Leonid Levin for the Boolean 3-satisfiability problem.
- "Given a boolean formula with a maximum of three literals, is there an assignment that results in TRUE?"
    - $(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$ = TRUE
    - $\{x_1 = true, x_2 = true, x_3 = false\}$

3-SAT

From 3-SAT, one can reduce it to many other problems, all being *NP*-complete as a consequence:

3-SAT

From 3-SAT, one can reduce it to many other problems, all being *NP*-complete as a consequence:

- SAT
- Clique
- Vertex Cover
- Hamiltonian Circuit
- Decision-TSP
- ...

3-SAT

From 3-SAT, one can reduce it to many other problems, all being *NP*-complete as a consequence:

- SAT
- Clique
- Vertex Cover
- Hamiltonian Circuit
- Decision-TSP
- ...

A polynomial time algorithm for any of these would imply that $P = NP$.

Summary

- Complexity Theory deals with bounds for problems.

Summary

- Complexity Theory deals with bounds for problems.
- Decision problems: P contains problems with polynomial time solutions.

Summary

- Complexity Theory deals with bounds for problems.
- Decision problems: P contains problems with polynomial time solutions.
- Verification problems: NP contains problems with polynomial time solutions.

Summary

- Complexity Theory deals with bounds for problems.
- Decision problems: P contains problems with polynomial time solutions.
- Verification problems: NP contains problems with polynomial time solutions.
- Reductions let us analyse new problems by framing them as existing ones.

Summary

- Complexity Theory deals with bounds for problems.
- Decision problems: P contains problems with polynomial time solutions.
- Verification problems: NP contains problems with polynomial time solutions.
- Reductions let us analyse new problems by framing them as existing ones.
- NP-completeness: solving one NP-complete problem implies in $\boxed{P = NP}$ due to reductions.

Last Words

- Some problems are undecidable: COMP30026

Last Words

- Some problems are undecidable: COMP30026
- Some scientists tried to prove that $P \stackrel{?}{=} NP$ is undecidable.

Last Words

- Some problems are undecidable: COMP30026
- Some scientists tried to prove that $P \stackrel{?}{=} NP$ is undecidable.
- Most scientists believe $P \neq NP$. NP-COMPLETE

Last Words

- Some problems are undecidable: COMP30026
- Some scientists tried to prove that $P \stackrel{?}{=} NP$ is undecidable.
- Most scientists believe $P \neq NP$.
- While the problem itself still eludes computer scientists, proposed solutions led to advancements in theory, even though they were wrong.