workshop2 2
solutions2 4
workshop3 6
solutions3 10
workshop4 14
solutions4 17
workshop6 21
solutions6 23
workshop7 28
solutions7 30
workshop8 32
solutions8 35
workshop9 42
solutions9 46
workshop10 54
solutions10 57
workshop11 60
solutions11 62
solutions11-2 66
workshop12 67
solutions12 70

COMP20007 Design of Algorithms

Week 2 Workshop

Tutorial

Welcome to the first COMP20007 tutorial. Introduce yourself to your peers, and work through the following exercises together.

Reminder: Big O notation Recall from prerequisite subjects that big O notation allows us to easily describe and compare algorithm performance. Algorithms in the class $O(n)$ take time linear in the size of their input. $O(\log n)$ algorithms run in time proportional to the logarithm of their input, (increasing by the same amount whenever their input doubles in size). $O(1)$ algorithms run in 'constant time' (a fixed amount of time, independent of their input size). We'll have more to say about big O notation this semester, but these basics will help with today's tutorial exercises.

1. Arrays Describe how you could perform the following operations on **(i) sorted** and **(ii) unsorted arrays**, and decide if they are $O(1)$, $O(\log n)$, or $O(n)$, where n is the number of elements initially in the array. Assume that there is no need to change the size of the array to complete each operation.

- Inserting a new element
- Deleting the final element
- Searching for a specified element
- Deleting a specified element

2. Linked lists Describe how you could perform the following operations on **(i) singly-linked** and **(ii) doubly-linked lists**, and decide if they are $O(1)$, $O(\log n)$, or $O(n)$, where n is the number of elements initially in the linked list. Assume that the lists need to keep track of their final element.

- Inserting an element at the start of the list
- Deleting an element from the start of the list
- Inserting an element at the end of the list
- Deleting an element from the end of the list

3. Stacks A stack is a collection where elements are removed in the reverse of the order they were inserted; the first element added is the last to be removed (much like a stack of books or plates). A stack provides two basic operations: `push` (to add a new element) and `pop` (to remove and return the top element). Describe how to implement these operations using

(i) An unsorted array (ii) A singly-linked list

4. Queues A standard queue is a collection where elements are removed in the order they were inserted; the first element added is the first to be removed (just like lining up to use an ATM). A standard queue provides two basic operations: `enqueue` (to add an element to the end of the queue) and `dequeue` (to remove the element from the front of the queue). Describe how to implement these operations using

(i) An unsorted array (ii) A singly-linked list

Can we perform these operations in constant time?

5. Bonus problem (optional) Stacks and queues are examples of *abstract data types*. Their behaviour is defined independently of their implementation — whether they are built using arrays, linked lists, or something else entirely.

If you have access only to stacks and stack operations, can you faithfully implement a queue? How about the other way around? You may assume that your stacks and queues also come with a size operation, which returns the number of elements currently stored.

COMP20007 Design of Algorithms

Week 2 Workshop Solutions

Tutorial

1. Arrays

(i) Using a sorted array

- Inserting a new element: $O(n)$
  First, find the right place for the new element with linear or binary search. Then, make a free space here in the array by shifting all larger elements over by one space. Put the new element in this free space.
- Searching for a specified element: $O(\log n)$
  Use binary search.
- Deleting the final element: $O(1)$
  Just remove the element from the end of the array.
- Deleting a specified element: $O(n)$
  Find the element with linear or binary search, remove it, and then fill the free space by shifting all larger elements over by one space.

(ii) Using an unsorted array

- Inserting a new element: $O(1)$
  Since there's no need to maintain sorted order, just put the new element at the end of the array.
- Searching for a specified element: $O(n)$
  Since the array isn't sorted, binary search won't work. We might need to scan the entire array.
- Deleting the final element: $O(1)$
  Just remove the element from the end of the array.
- Deleting a specified element: $O(n)$
  Find the element with linear search (binary search won't work), remove it, and then fill the free space with the element from the end of the array.

2. Linked lists In all of these operations, ensure that the link(s) in the inserted/removed node, the link(s) in the next and previous nodes, and the links to the first and last elements of the list are all updated.

Don't forget the case where you are inserting an element into an empty list, or removing the only element in a list. In these cases both the link to the start of the list and the link to the end of the list need to be updated.

All of these operations on singly-linked and doubly-linked lists are $O(1)$ except for deleting the last element from a singly-linked list. In this case, to find the new last element and update the link to the end of the list, we need to follow the list all the way from the start to the second last element. (This isn't the case in a doubly-linked list, because the last element in a doubly-linked list has a direct link to the second last element.)

3. Stacks In an unsorted array, `push` by adding elements to the end of the array, and `pop` by removing them from the end of the array (both of these operations can be done in constant time).

In a singly-linked list, `push` by adding elements to the start of the list, and `pop` by removing them from the start of the list (it's best to use the start of the list because removing from the end of the list requires $O(n)$ time).

4. Queues In an unsorted array, `enqueue` by adding elements to the end of the array, and `dequeue` by removing them from the start of the array.

Removing from the start of the array would usually take linear time, because we'd need to move all of the remaining elements one space closer to the start to fill the free space we made. We can avoid doing this if we let the starting index of the queue drift as we repeatedly dequeue elements. We can then also let the end of the queue wrap back around to the start of the array, as if the array was a circle. We just need to carefully keep track of where the queue starts and ends as elements are enqueued and dequeued.

In a singly-linked list, `enqueue` by adding elements to the end of the list, and `dequeue` by removing them from the start of the list.

Adding at the end and removing from the start allows both operations to run in constant time, compared with adding at the start and removing from the end. Assuming our list needs to keep track of its last element, removing from the end takes linear time in a singly-linked list.

5. Bonus problem (optional) With access only to stack operations, one way to create queue operations is as follows:

- For `enqueue`, add the new element to the top of a stack using `push`.
- For `dequeue`, we need the item that was enqueued first, which will be at the bottom of the stack. `pop` items from the stack and `push` them into a second stack until `size` of the first stack is 1. `pop` this last element. Before returning it, `pop` the elements off the second stack and `push` them back onto the first stack.

With access only to queue operations, one way to create stack operations is as follows:

- For `push`, add the new elements to the end of a queue using `enqueue`.
- For `pop`, we need the item that was enqueued last, which will be at the end of the queue. `dequeue` the items from the queue and `enqueue` them into a second queue until the first queue has `size` 1. `dequeue` and return this last element. There's no need to put the elements in the second queue back into the first queue; they are already in the right order in the second queue, so we can `push` into the end of the second queue from this point onwards (until we swap again, next time we `pop`).

Note that for both of these approaches insertion (*i.e.,* `enqueue` and `push` respectively) is $O(1)$, while removal (*i.e.,* `dequeue` and `pop`) is an $O(n)$ operation.

COMP20007 Design of Algorithms

Week 3 Workshop

Tutorial

0. Sums Give closed form expressions for the following sums.

(a) $\sum_{i=1}^{n} 1$ (b) $\sum_{i=1}^{n} i$ (c) $\sum_{i=1}^{n}(2i+3)$

(d) $\sum_{i=0}^{n-1}\sum_{j=0}^{i} 1$ (e) $\sum_{i=1}^{n}\sum_{j=1}^{m} ij$ (f) $\sum_{k=0}^{n} x^k$

1. Complexity classes For each of the following pairs of functions, $f(n)$ and $g(n)$ determine whether $f(n) \in O(g(n))$, $f(n) \in \Omega(g(n))$ or both (*i.e.*, $f(n) \in \Theta(g(n))$).

(a) $f(n) = \frac{1}{2}n^2$ and $g(n) = 3n$

(b) $f(n) = n^2 + n$ and $g(n) = 3n^2 + \log n$

(c) $f(n) = n \log n$ and $g(n) = \frac{n}{4}\sqrt{n}$

(d) $f(n) = \log(10n)$ and $g(n) = \log(n^2)$

(e) $f(n) = (\log n)^2$ and $g(n) = \log(n^2)$

(f) $f(n) = \log_{10} n$ and $g(n) = \ln n$

(g) $f(n) = 2^n$ and $g(n) = 3^n$

(h) $f(n) = n!$ and $g(n) = n^n$

2. Sequential search adapted from *Levitin* [2nd Ed.] 2.2.1. Use O, Ω and/or Θ to make the strongest possible claim about the runtime complexity of sequential search in,

(a) general (*i.e.*, all possible inputs)

(b) the best case

(c) the worst case

(d) the average case

3. Solving recurrence relations Solve the following recurrence relations, assuming $T(1) = 1$.

(a) $T(n) = T(n-1) + 4$ (b) $T(n) = T(n-1) + n$ (c) $T(n) = 2T(n-1) + 1$

4. k-Merge adapted from *DPV* 2.19. Consider a modified sorting problem where the goal is to sort k lists of n sorted elements into one list of kn sorted elements.

One approach is to merge the first two lists, then merge the third with those, and so on until all k lists have been combined. What is the time complexity of this algorithm? Can you design a faster algorithm using a divide-and-conquer approach?

5. Mergesort complexity (optional) Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

Construct a recurrence relation to describe the runtime of mergesort sorting n elements. Explain where each term in the recurrence relation comes from.

This kind of recurrence is difficult to solve by expansion. We haven't seen the master theorem yet, but you can look it up and use it to solve this recurrence relation and find the runtime complexity of mergesort if you finish these questions early.

Computer Lab

Modules and Multi-file C Programs. Up until this point, we have primarily written C programs using a single file containing all of the structs and functions for our program to run, as well as the main function which serves as the entry point to our program.

As our programs become more complex we will want to separate our code into smaller components containing related functionality. This helps us organise our code, as well as making code re-use easier.

1. Modules A module is made up of two files, a **header file** and a **C file**. Download `racecar.h` and `racecar.c` from the LMS, which makes up the racecar module.

A header file declares the functions and types which the module implements. It doesn't contain the definitions of any functions, just the prototypes.

Take a look at `racecar.h` and list the types and functions which make up the racecar module.

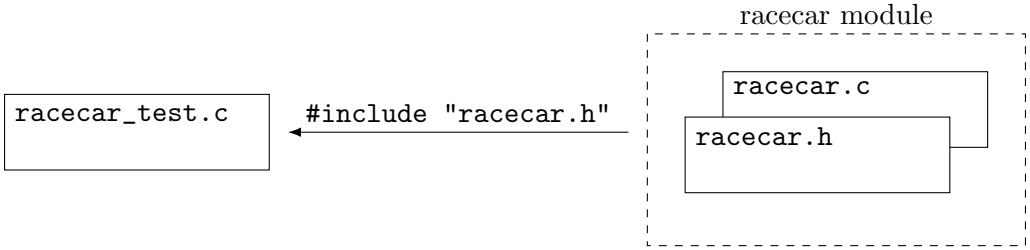
2. Using Modules Download the file `racecar_test.c` from the LMS.

The racecar module doesn't define a main function, and so on its own it is not a program that we can run. The `racecar_test.c` file defines a main function which creates some `Racecar` structs and prints out some information about them to test the racecar module.

The `racecar_test.c` file knows about the functions from the racecar module because it *includes the module*, which is achieved by the following line:

```
#include "racecar.h"
```

Note that we use double quotes (*e.g.,* `#include "..."`) for modules we have created, and angle brackets (*e.g.,* `#include <...>`) for C standard library modules.



By including the `.h` file, the C pre-processor essentially copy-pastes the content of `racecar.h` into `racecar_test.c` before compiling it, meaning it knows about any `typedef`s and function prototypes.

Note that multiple files can include the same module which can result in the same function prototype or type being declared more than once, which will be an error in C. To avoid erros when a module is included more than once we use **include guards**, which allows us to check whether the file has been included yet, and only declare the types/functions if it has not been. This is done using the following pre-processor directives:

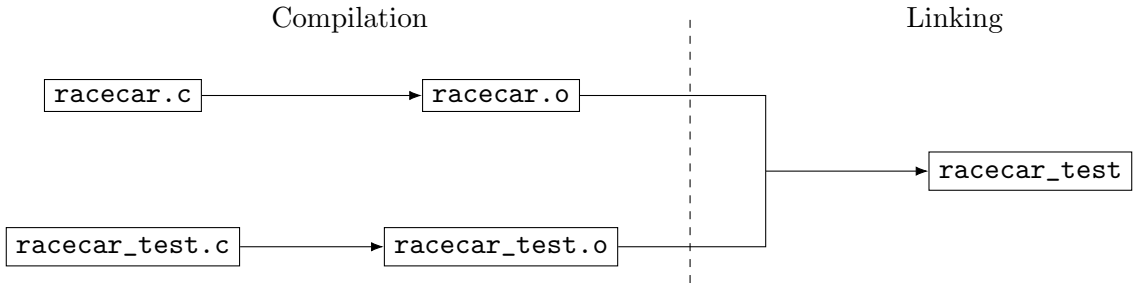
```
#ifndef RACECAR_H
#define RACECAR_H

... .h file contents go here ...

#endif
```

Compiling a multi-file C program involves two steps: compiling and linking.

First, each `.c` file has to be compiled into an object file (`.o`) and then the object files are linked to produce the program (*i.e.,* the executable file). In this example this looks like:



- To compile a file `file.c` into its object file `file.o` run:
  `$ gcc -Wall -c file.c`

- To link the object files (`file1.o`, ..., `fileN.o`) into an executable called `program` run:
  `$ gcc -Wall -o program file1.o ... fileN.o`

We can also do the compilation and linking in a single command, however this means re-compiling all `.c` files even if they haven't changed, so it's often faster to just re-compile the `.c` files which have changed and link everything again. To do this in one command you would run:
`$ gcc -Wall -o program file1.c ... fileN.c`

Compile and link the program to test the racecar module. Call this program `racecar_test`. Run this program to confirm that you've compiled it successfully. What does it output?

3. Compilation using a Makefile `make` is a tool for automatically compiling and linking multi-file C programs. It keeps track of which files have changed, so that only files that have changed (or had their dependencies changed) since the previous compilation will ever be recompiled.

To make use of `make` you must provide instructions for how your program should be compiled in a file called `Makefile` or `makefile`. Note that this file should not have a file extention, it won't work if your file is named `Makefile.txt` for example.

A `Makefile` is made up of *rules*. A rule has the following structure:

```
target: dependency1 .. dependencyN
  instruction1
  ...
  instructionM
```

Note: the indentation of the instuctions component of a rule must be a single tab, using spaces to indent will result in an error.

The `target` is the name of the file that should be created by a rule (*e.g.,* an ȯ file, or an executable file), the dependencies are the files that must already exist before the target can be created and the instructions are just command line instructions used to construct the target.

For example, the following rules compile and link a program named main which uses a module called foo.

```
main: main.o foo.o
  gcc -Wall -o main main.o foo.o

main.o: main.c foo.h
  gcc -Wall -c main.c

foo.o: foo.c foo.h
  gcc -Wall -c foo.c
```

Note that `main.c` and `foo.c` would both include `foo.h`, and so it must be a dependency for each of these rules.

To run this `Makefile` we would run `make <target>` (*e.g.,* `make main`) or just `make`, which will make the first target by default.

Now, create your own `Makefile` to compile the `racecar_test` program from *Question 2.*

4. Which parts of the module can we access? In `racecar.h` we have the following typedef:

```
typedef struct racecar Racecar;
```

In `racecar.c` we have the struct's definition:

```
struct racecar {
  char *driver;
  char *team;
  double *laps;
  size_t n_laps;
  size_t laps_capacity;
};
```

So any C file which includes `racecar.h` will know about a type called `Racecar` and know that it is a struct. But will it know *what information the struct contains*?

In the `racecar_test.c` file, try to print out the number of laps in the `renault` variable using `printf("n_laps: %d\n", renault->n_laps)`. Does this work? What error do you get?

It turns out that `racecar_test.c` doesn't know what information is in the struct, nor how to access it. This is potentially desirable, as we have abstracted the implementation details of the `Racecar` type away from those who use it. Rather, to interact with this struct you should use the functions which `racecar.h` provides.

Write a new function in `racecar.c` called `num_laps()` which returns the number of laps a particular `Racecar` has done. Use this function in `racecar_test.c` and re-compile your program.

Notice as well that the function `print_error` is defined within `racecar.c`, but the prototype is not present in `racecar.h`. Try calling this function from `racecar_test.c`. Does it work? What error do you get?

5. Linked List Module We'll use linked lists for many algorithms and data structures which are covered in this class, and it would be nice if we had some C code to use whenever we require a linked list in our programs.

In this question you should write a linked list module, comprised of `list.c` and `list.h`.

There will be some decisions you must make when you design your modules, for instance will your linked list be singly- or doubly-linked? Will you have a single node structure, or a linked list structure which contains more information about your list (*e.g.,* the head, tail and size of your list)?

Create functionality to insert and delete at the front and end of your list, find out how many elements are in the linked list and to free the linked list.

Once you've created your linked list module, create a main program to test your linked list.

COMP20007 Design of Algorithms

Week 3 Workshop Solutions

Tutorial

0. Sums

(a)

$$\sum_{i=1}^{n} 1 = \underbrace{1 + 1 + \ldots + 1}_{n \text{ times}} = n$$

(b)

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \quad \text{(Triangle numbers formula)}$$

(c)

$$\sum_{i=1}^{n}(2i+3) = 2\sum_{i=1}^{n} i + \sum_{i=1}^{n} 3 = 2 \times \frac{n(n+1)}{2} + 3 \times n = n^2 + 4n$$

(d)

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

(e)

$$\sum_{i=1}^{n}\sum_{j=1}^{m} ij = \left(\sum_{i=1}^{n} i\right)\left(\sum_{j=1}^{m} j\right) = \left(\frac{n(n+1)}{2}\right)\left(\frac{m(m+1)}{2}\right) = \frac{nm(n+1)(m+1)}{4}$$

(f)

$$\sum_{k=0}^{n} x^k = \frac{1 - x^{n+1}}{1 - x} \quad \text{(Geometric series)}$$

1. Complexity classes

(a) $\frac{1}{2}n^2 \in \Omega(3n)$ – ignore constants, n^2 grows faster than n

(b) $n^2 + n \in \Theta(3n^2 + \log n)$ – ignore constants, the fastest growing terms are both n^2

(c) $n \log n \in O\left(\frac{n}{4}\sqrt{n}\right)$ – ignoring constants the difference here is $\log n$ vs. $\sqrt{n}$

(d) $\log(10n) \in \Theta(\log(n^2))$ – using log laws $f(n) = \log 10 + \log n$ and $g(n) = 2 \log n$

(e) $(\log n)^2 \in \Omega(\log(n^2))$ – we can see that $(\log n)^2/(2 \log n) = \frac{1}{2} \log n$ so $f(n)$ grows faster

(f) $\log_{10} n \in \Theta(\ln n)$ – using change of base formula $\log_{10} n = \frac{1}{\ln 10} \ln n$

(g) $2^n \in O(3^n)$ – unlike logarithms, the base in the exponential changes the growth rate

(h) $n! \in O(n^n)$ – $1 \times 2 \times \cdots \times n \in O(n \times n \times \cdots \times n)$ but not vice versa

2. Sequential search

(a) general (*i.e.*, all possible inputs): the best case for sequential search is when the element we're searching for is at index 0. So $C_{\text{best}}(n) = 1$. Also $C_{\text{worst}}(n) = n$ occurs when the element is not in the array. So $C_{\text{best}}(n) \leq C(n) \leq C_{\text{worst}} \implies C(n) \in \Omega(1)$ and $C(n) \in O(n)$.

(b) the best case: in the best case $C_{\text{best}}(n) = 1$ so $C_{\text{best}} \in \Theta(1)$.

(c) the worst case: in the words case $C_{\text{worst}}(n) = n$ so $C_{\text{worst}} \in \Theta(n)$.

(d) the average case: let the probability of the element being in the array be p. If the element is not in the array the cost is n. If the element is in the array we assume it's equally likely to be in any of the n indices. Taking the average of the number of comparisons when the element is in each index:

$$\frac{1}{n}\left(1 + 2 + \cdots + n\right) = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

So the cost is $\frac{n+1}{2}$ with probability p, or n with probability $(1-p)$, *i.e.*,

$$C_{\text{average}}(n) = p \times \frac{n+1}{2} + (1-p) \times n.$$

Since p is a constant here, $C_{\text{average}} \in \Theta(n)$.

3. Solving recurrence relations Solve the following recurrence relations, assuming $T(1) = 1$.

(a) $T(n) = 4n - 3$
$T(n) = T(n-1)+4$ means $T(n-1) = T((n-1)-1)+4 = T(n-2)+4$, and $T(n-2) = T(n-3)+4$, and so on. Repeatedly substitute into the first equation, until the pattern becomes clear. Then, produce the base case to eliminate T.

$$\begin{aligned}
T(n) &= T(n-1) + 4 \\
&= T(n-2) + 4 + 4 \\
&= T(n-3) + 4 + 4 + 4 \\
&\vdots \qquad \text{(starting to see the pattern?)} \\
&= T(n-k) + k \times 4 \\
&\vdots \qquad \text{(bring in the base case)} \\
&= T(n-(n-1)) + (n-1) \times 4 \\
&= T(1) + (n-1) \times 4 \\
&= 1 + 4n - 4 \\
&= 4n - 3
\end{aligned}$$

(b) $T(n) = \dfrac{n(n+1)}{2}$

$$\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + (n-1) + n \\
&= T(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= T(n-k) + (n-(k-1)) + \cdots + (n-2) + (n-1) + n \\
&\vdots \\
&= T(n-(n-1)) + (n-(n-1-1)) + \cdots + (n-2) + (n-1) + n \\
&= T(1) + (n-n+1+1) + \cdots + (n-2) + (n-1) + n \\
&= 1 + 2 + \cdots + (n-2) + (n-1) + n \\
&= \frac{n(n+1)}{2} && \text{(triangle numbers)}
\end{aligned}$$

(c) $T(n) = 2^n - 1$

$$\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2(2T(n-2)+1) + 1 = 2^2 T(n-2) + 2 + 1 \\
&= 2^2(2T(n-3)+1) + 2 + 1 = 2^3 T(n-3) + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^k T(n-k) + 2^{k-1} + \cdots + 2^2 + 2 + 1 \\
&\vdots \\
&= 2^{(n-1)} T(n-(n-1)) + 2^{(n-1)-1} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} T(1) + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^{n-1} + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^n - 1 && \text{(sum of powers of 2 is the next power of 2, minus 1)}
\end{aligned}$$

4. k-Merge Merging two lists of sizes a and b takes about $a + b$ steps. Merging the first two lists (sizes n and n) will take $2n$ steps. Merging this list with the next list (sizes $2n$ and n) will take $3n$ steps. The next will be $4n$ steps, and so on, until the last list of n is merged with the rest of the $(k-1)n$ items, taking kn steps. In total,

$$2n + 3n + 4n + \cdots + kn = n(2 + 3 + 4 + \cdots + k)$$

Simplifying by recognising the triangle numbers, we're looking at $\Theta(k^2 n)$:

$$n(2 + 3 + 4 + \cdots + k) = n(1 + 2 + 3 + 4 + \cdots + k) - n = n\frac{k(k+1)}{2} - n \in \Theta(k^2 n)$$

A faster algorithm would use the merging strategy from mergesort, merging the lists in pairs. First, merge $k/2$ pairs of length n lists, resulting in $k/2$ lists of length $2n$ (and taking $k/2 \times 2n = kn$ steps). Next, merge $k/4$ pairs of length $2n$ lists, resulting in $k/4$ lists of length $4n$ (and taking $k/4 \times 4n = kn$ steps). Continue, a total of $\log k$ times, and you will have one list of length kn in $\Theta(kn \log k)$ time.

5. Mergesort complexity (optional) Recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

where $T(n)$ is the runtime of mergesort sorting n elements. The first $T(\frac{n}{2})$ is the time it takes to sort the left half of the input using mergesort. The other $T(\frac{n}{2})$ is the time it takes to sort the right half. $\Theta(n)$ is a bound on the time it takes to merge the two halves together.

We haven't seen the master theorem in class yet, and for this question we aren't required to solve the recurrence relation. However if we did want to solve this we can recognise that this recurrence relation fits the master theorem, with $a = 2$, $b = 2$, and $d = 1$.

$$\log_b(a) = \log_2(2) = 1 = d$$

so, by the master theorem, $T(n) \in \Theta(n \log n)$.

COMP20007 Design of Algorithms

Week 4 Workshop

Tutorial

1. Subset-sum problem Design an exhaustive-search algorithm to solve the following problem:

Given a set S of n positive integers and a positive integer t, does there exist a subset $S' \subseteq S$ such that the sum of the elements in S' equals t, *i.e.*,

$$\sum_{i \in S'} i = t.$$

If so, output the elements of this subset S'.

Assume that the set is provided as an array of length n. An example input may be $S =$ `[1, 8, 4, 2, 9]` and $t = 7$, in which case the answer is Yes and the subset would be $S' =$ `[1, 4, 2]`.

What is the time complexity of your algorithm?

2. Partition problem Design an exhaustive-search algorithm to solve the following problem:

Given a set S can we find a partition (*i.e.*, two disjoint subsets A, B such that all elements are in either A or B) such that the sum of the elements in each set are equal, *i.e.*,

$$\sum_{i \in A} i = \sum_{j \in B} j.$$

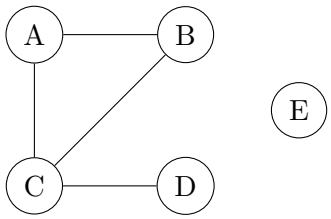
If so, which elements are in A and which are in B?

Again, assume S is given as an array. For example for $S =$ `[1, 8, 4, 2, 9]` one valid solution is $A =$ `[1, 2, 9]` and $B =$ `[8, 4]`.

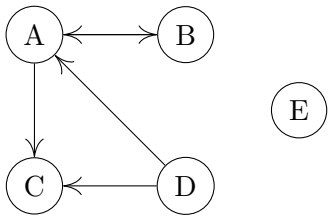
Can we make use of the algorithm from *Question 1.*? What's the time complexity of this algorithm?

3. Graph representations Consider the following graphs.

(a) An undirected graph:



(b) A directed graph:



Give their representations as:

(i) adjacency lists (ii) adjacency matrices (iii) sets of vertices and edges

Which node from (a) has the highest degree? Which node from (b) has the highest in-degree?

4. Graph representations continued Different graph representations are favourable for different applications. What is the time complexity of the following operations if we use (i) adjacency lists (ii) adjacency matrices or (iii) sets of vertices and edges?

(a) determining whether the graph is *complete*

(b) determining whether the graph has an *isolated node*

Assume that the graphs are undirected. A *complete* graph is one in which there is an edge between every pair of nodes. An *isolated node* is a node which is not adjacent to any other node.

Assume that the graph has n vertices and m edges.

5. Sparse and dense graphs (optional) We consider a graph to be *sparse* if the number of edges, m, is order $\Theta(n)$. On the other hand we say a graph is *dense* if $m \in \Theta(n^2)$.

Give examples of types of graphs which are sparse and types which are dense.

What is the space complexity (in terms of n) of a sparse graph if we store it using:

(i) adjacency lists (ii) adjacency matrix (iii) sets of vertices and edges

Computer Lab

In assignment 1 you will need to create and provide a Makefile, and you'll be required to test and submit your program using the School of Engineering `dimefox` servers.

1. Make and Makefiles If you haven't already, take a look back to last week's lab and ensure that you know how to create a `Makefile` and use `make`.

2. Dimefox Copy the code and Makefile from last weeks lab onto the `dimefox` server. Compile and run your code on the server manually, and then also using the Makefile you created.

The *Dimefox Instructions* document on the LMS will help with this task.

3. Linked List Module

This question is the same as the final question from last week. We'll spend some time this week making sure we can all write and create our own linked list module.

We'll use linked lists for many algorithms and data structures which are covered in this class, and it would be nice if we had some C code to use whenever we require a linked list in our programs.

In this question you should write a linked list module, comprised of `list.c` and `list.h`.

There will be some decisions you must make when you design your modules, for instance will your linked list be singly- or doubly-linked? Will you have a single node structure, or a linked list structure which contains more information about your list (*e.g.*, the head, tail and size of your list)?

Create functionality to insert and delete at the front and end of your list, find out how many elements are in the linked list and to free the linked list.

Once you've created your linked list module, create a main program to test your linked list.

COMP20007 Design of Algorithms

Week 4 Workshop Solutions

Tutorial

1. Subset-sum problem First, we'll cover some terminology. The *power set* of a set S – often denoted $\mathcal{P}(S)$ – is the set of all subsets of S, *e.g.,* if $S = \{1, 2, 3\}$ then,

$$\mathcal{P}(S) = \big\{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}.\big\}$$

If a set has $|S|$ elements then $\mathcal{P}(S)$ has $2^{|S|}$ elements. An exhaustive-search approach to this subset-sum problem will include iterating over all subsets of S, and checking whether their sum is t:

function SubsetSum(S, t)
  for each S' in Powerset(S) **do**
    if $\sum_{i \in S'} i = t$ **then**
      return Yes
  return No

Since $|S| = n$, we know that there are 2^n power sets to iterate through. For each power set we must compute a sum, which will be an $O(n)$ operation. So the runtime of this subset sum problem is $O(n2^n)$.

How can we systematically generate all subsets of the given set S? For each element $x \in S$, we need to generate all the subset of S that happen to contain x, as well as those that do not. This gives us a natural recursive algorithm:

function Powerset(S)
  if $S = \emptyset$ **then**
    return $\{\emptyset\}$
  else
    $x \leftarrow$ some element of S
    $S' \leftarrow S \setminus \{x\}$
    $P \leftarrow$ Powerset(S')
    return $P \cup \{s \cup \{x\} \mid s \in P\}$

2. Partition problem At first it might seem we have to do something more sophisticated than in the subset-sum problem, however this problem reduces nicely to this problem.

First we'll notice that if there are sets A and B which partition S and have equal sum they'll have to satisfy the following two equations:

$$\sum A + \sum B = \sum S \tag{1}$$

$$\sum A = \sum B \tag{2}$$

So we can see that $\sum A = \sum B = \frac{1}{2} \sum S$ would have to hold. First, if $\sum S$ is odd we know the answer is No (given our elements are integers). Second, if such sets A and B exist, any set with sum $\frac{1}{2} \sum S$ will do. So:

function HasPartition(S)
  $sum \leftarrow \sum_{i \in S} i$
  if sum is odd **then**
    return No
  return SubsetSum($S, sum/2$)

3. Graph representations

(a) Adjacency lists:

A → B, C
B → A, C
C → A, B, D
D → C
E →

Adjacency matrix:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Sets of vertices and edges:

$G = (V, E)$, where $V = \{A, B, C, D, E\}$, and $E = \{\{A, B\}, \{A, C\}, \{B, C\}, \{C, D\}\}$

Degree is the number of edges connected to a vertex. Node C has the highest degree (3).

(b) Adjacency lists:

A → B, C
B → A
C →
D → A, C
E →

Adjacency matrix:

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Sets of vertices and edges:

$G = (V, E)$, where $V = \{A, B, C, D, E\}$, and $E = \{(A, B), (A, C), (B, A), (D, A), (D, C)\}$

For a directed graph, degree is separated into *in*-degree and *out*-degree: the number of edges going into or out of each vertex, respectively. Nodes A and C have the highest in-degree (2).

4. Graph representations continued Note that in this question we assume that self-loops are not allowed.

Additionally, we could always check whether a graph is constant time if we know (in constant time) the number of edges. If the number of edges is exactly $\binom{n}{2} = n(n-1)/2$ then the graph is complete, otherwise it is not.

(a) Determining whether a graph is *complete*.

(i) In the adjacency list representation we just want to check that for each node u, every other node is in its adjacency list.

If we can check the size of the list in $O(1)$ time then we just need to go through each vertex and check that the size of the list is $n - 1$, and thus this operation will be linear in the number of nodes: $O(n)$.

If we have to do a scan through the adjacency list then this would take $O(m)$, which for a complete graph will be asymptotically equivalent to $O(n^2)$.

(ii) In the adjacency matrix representation, a complete graph would contain 1's in all position except for the diagonal (since we can not have self loops). So there will be $n^2 - n$ positions in the matrix to lookup, and thus this operation will take $O(n^2)$ time.

(iii) For the graph to be complete the set of edges must contain each pair of vertices. There will be $\binom{n}{2}$ edges if the graph is indeed complete, so we could just check the size of the edge set E. This would be a constant time operation: $O(1)$.

However if we can't just check the size of the set we'll have to look through all edges which will take $O(m)$ time.

(b) Determining whether the graph has an *isolated node*.

We'll have a think about how long it takes to determine if a single node is isolated, and then apply that to each vertex (*i.e.*, n times)

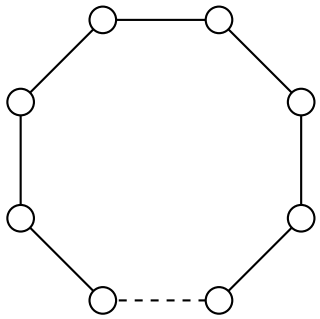
(i) In the adjacency list representation, a node is isolated if its adjacency list is empty. This is $O(1)$ to check per node, so $O(n)$ for the whole graph.

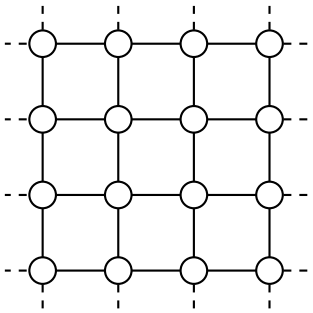
(ii) In the adjacency matrix representation to check if a node is isolated we must look at each entry in that node's row or column and confirm that all entries are 0. This is $O(n)$ per node so $O(n^2)$ in total.

(iii) In the sets of vertices and edges representation we can loop through the set of edges and confirm that a vertex does not appear. This will take $O(m)$ for a single node.

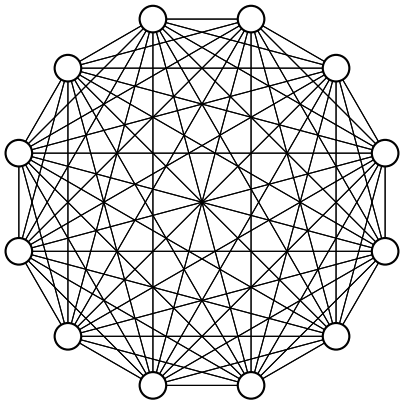
However we can also check a whole graph in a single pass by keeping track of all the nodes at once (in an array or a hash table for instance) and ticking them off as we see them, at the end we can iterate through this array/hash table to check if there were any isolated nodes. So this will take $O(n + m)$ time.

5. Sparse and dense graphs (optional) Graphs with a constant number of edges per vertex (*i.e.*, the degree of the vertices doesn't grow with n) are sparse. Some examples of these are cycles and grid graphs:





Examples of dense graphs are complete graphs:



Real world examples of sparse graphs might arise from a graph representing the internet, and for dense graphs we could consider a network of cities, connected by all the possible aeroplane routes.

Storing sparse graphs using the various graph representations give rise to the following space complexities:

(i) To store an adjacency list we need to store one piece of data per edge. So the space complexity is $O(m)$. Thus in a sparse graph the space complexity is $O(n)$.

(ii) To store an adjacency matrix we need to store n^2 pieces of information, regardless of m. So a sparse graph is still $O(n^2)$ space.

(iii) Sets of vertices and edges just require n items for the vertices and m items for the edges, so $O(n+m)$. In a sparse graph this becomes $O(n+n) = O(n)$.

COMP20007 Design of Algorithms

Week 6 Workshop

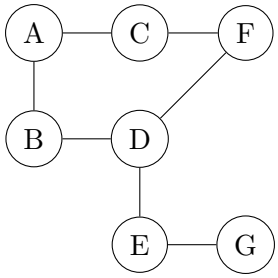
Tutorial

1. Depth First Search and Breadth First Search List the order of the nodes visited in the following graph when the following search algorithms are run:

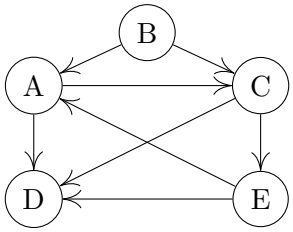
(i) depth first search

(ii) breadth first search

Start at A and break ties by choosing nodes in alphabetic order.



2. Tree, Back, Forward and Cross Edges A depth-first search of a directed graph can be represented as a tree (or a collection of trees, *i.e.*, a forest). Each edge of the graph can then be classified as a *tree edge*, a *back edge*, a *forward edge*, or a *cross edge*. A tree edge is an edge to a previously un-visited node, a back edge is an edge from a node to an ancestor, a forward edge is an edge to a non-child descendent and a cross edge is an edge to a node in a different sub-tree (*i.e.*, neither a descendent nor an ancestor). Draw a depth-first search tree based on the following graph, and classify its edges into these categories. Begin the depth-first search at A.

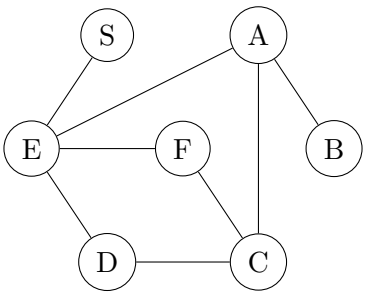


In an undirected graph, you wont find any forward edges or cross edges. Why is this true? You might like to consider the graph above, with each of its edges replaced by undirected edges.

3. Finding Cycles Explain how one can also use breadth-first search to see whether an undirected graph is cyclic. Which of the two traversals, depth-first and breadth-first, will be able to find cycles faster? (If there is no clear winner, give an example where one is better, and another example where the other is better.)

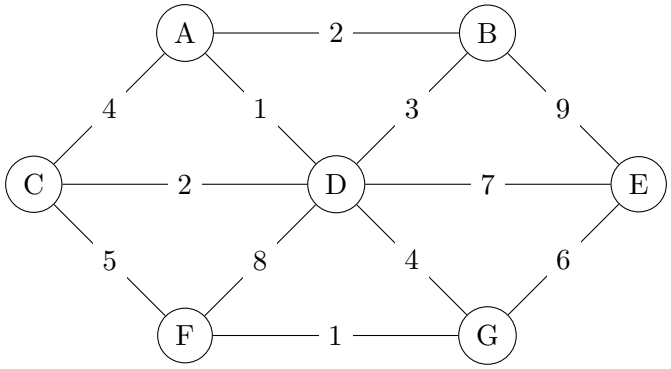
4. 2-Colourability Design an algorithm to check whether an undirected graph is 2-colourable, that is, whether its nodes can be coloured with just 2 colours in such a way that no edge connects two nodes of the same colour.

To get a feel for the problem, try to 2-colour the following graph.



Do you expect we could extend such an algorithm to check if a graph is 3-Colourable, or in general: k-Colourable?

5. Single Source Shortest Path with Dijkstra's Algorithm Dijkstra's algorithm computes the shortest path to each node in a graph from a single starting node (the 'source'). Trace Dijkstra's algorithm on the following graph, with node E as the source. Repeat the algorithm with node A as the source. How long is the shortest path from E to A? How about A to F?



6. Minimum Spanning Tree with Prim's Algorithm Prim's algorithm finds a minimum spanning tree for a weighted graph. Discuss what is meant by the terms 'tree', 'spanning tree', and 'minimum spanning tree'.

Run Prim's algorithm on the graph from Question 5, using A as the starting node. What is the resulting minimum spanning tree for this graph? What is the cost of this minimum spanning tree?

Computer Lab

Use today's lab time to work on Assignment 1. Make sure you can log in to `dimefox`, copy your code over, compile and run your program on the server.

If you've completed the assignment already feel free to use this time to complete previous lab exercises.

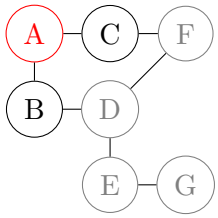
COMP20007 Design of Algorithms

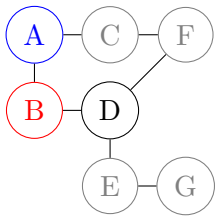
Week 5 Workshop Solutions

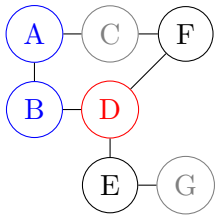
Tutorial

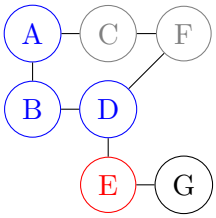
1. Depth First Search and Breadth First Search

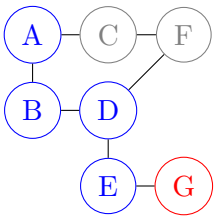
(i) Depth First: ABDEGFC The nodes visited are as follows. The node currently being visited is in red, the previously visited nodes are in blue and the nodes currently being considered are in solid black. Others are in gray.

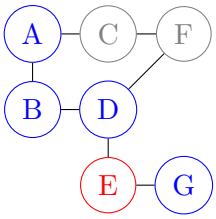
1. 

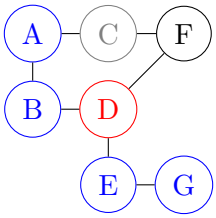
2. 

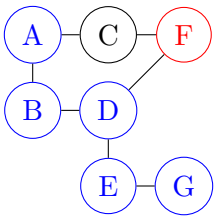
3. 

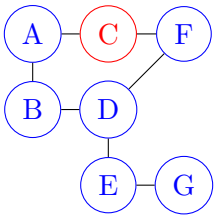
4. 

5. 

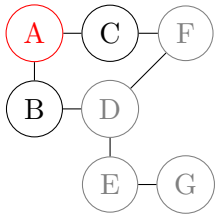
6. 

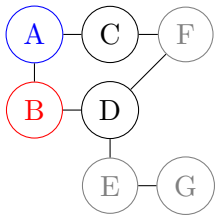
7. 

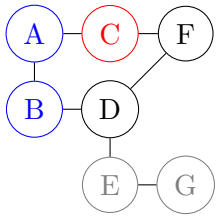
8. 

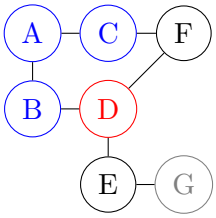
9. 

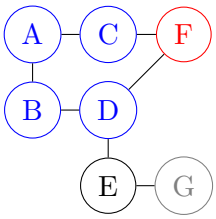
(ii) Breadth First: ABCDFEG

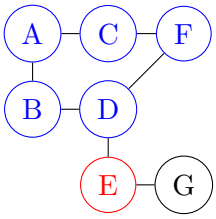
1. 

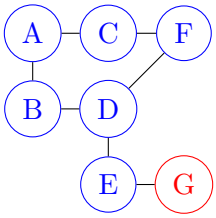
2. 

3. 

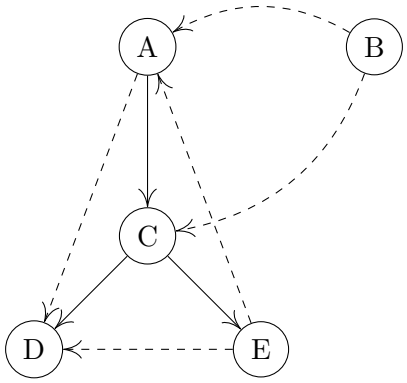
4. 

5. 

6. 

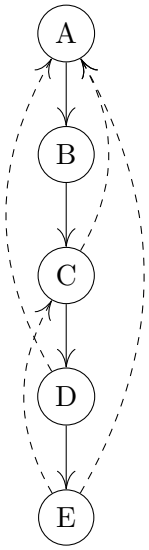
7. 

2. Tree, Back, Forward and Cross Edges For the directed version of this graph the DFS tree looks like this:



The solid edges are *tree* edges. The dashed edge (A, D) is a forward edge as it is from A to one of its non-children descendants. The edge (E, A) is a back edge as it connects E to a non-parent ancestor. The remaining edges (*i.e.*, (E, D), (B, A) and (B, C)) are all cross edges, as they connect vertices which are neither descendants nor ancestors of each other.

In the undirected version of this graph we get:[1]



We can see that only tree edges and back edges appear in the DFS forest for the undirected graph.

In fact undirected graphs only have tree and back edges:

- Suppose we had a forward edge (x, y), *i.e.*, y is a descendent of x. Since the graph is undirected, x is connected to y and visa versa. However we would have either visited y from x (making (x, y) a tree edge) or seen x while we're visiting y before y is popped from the stack (making (y, x) a back edge). So (x, y) can not be a forward edge in an undirected graph.
- Suppose we have a cross edge (x, y), *i.e.*, x is visited during some other part of the tree (*i.e.*, y has already been visited and popped). This cannot arise in an undirected graph though, since we would have visited x while we were visiting y since y connects to x and visa versa. Thus we can't have a cross edge.

3. Finding Cycles First, looking at DFS – it turns out that an undirected graph is cyclic if and only if it contains a back edge. We change the exploration strategy to find back edges:

function CYCLIC($\langle V, E \rangle$)
    mark each node in V with 0

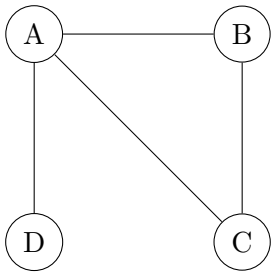
[1] *Update:* this has been updated to add a previously missing back edge from E to A

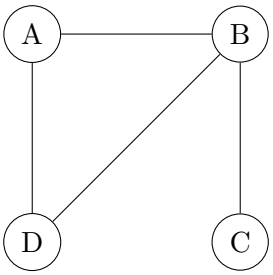
```
    for each v in V do
        if v is marked 0 then
            if DfsExplore(v) = True then                    ▷ a back edge was found
                return True
    return False

function DfsExplore(v)
    mark v with 1
    for each edge (v, w) do                                 ▷ w is v's neighbour
        if w is marked with 0 then
            if DfsExplore(w) then
                return True
        else
            if (v, w) is a back edge then
                return True
    return False
```

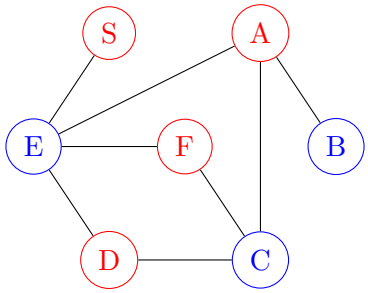
For breadth first search however we get cycles when there exist cross edges. Similar alterations to the breadth first search algorithm could be made to check for cross edges.

Sometimes depth-first search finds a cycle faster, sometimes not. Below, on the left, is a case where depth-first search finds the cycle faster, before all the nodes have been visited. On the right is an example where breadth-first search finds the cycle faster.





4. 2-Colourability First, the (only possible, up to swapping colours) 2-colouring of this graph is:



An undirected graph can be checked for two-colourability by performing a DFS traversal.

This begins by first assigning a colour of 0 (that is, no colour) to each vertex. Assume the two possible "colours" are 1 and 2.

Then traverse each vertex in the graph, colouring the vertex and then recursively colouring (via DFS) each neighbour with the opposite colour. If we encounter a vertex with the same colour as its sibling then a two-colouring is not possible.

In simpler terms, we're just doing a DFS and assigning layers alternating colours: *i.e.*, the first layer gets colour 1, then the second layer colour 2 *etc.* We know that we are not 2-colourable if we ever find a node which is adjacent to a node which has already been coloured the same colour.

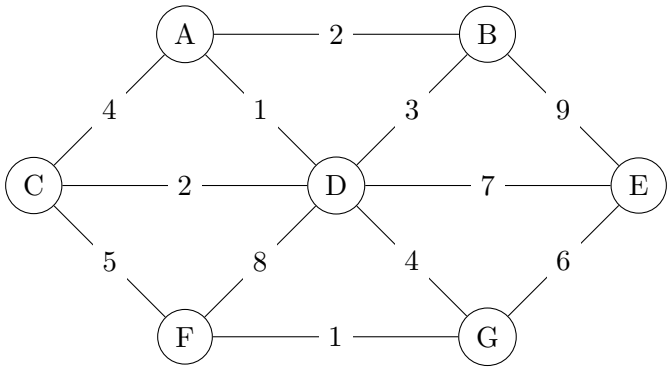
```
function IsTwoColourable(G)
    let ⟨V, E⟩ = G
    for each v in V do
        colour[v] ← 0
    for each v in V do
        if colour[v] = 0 then
            Dfs(v, 1)
    output True

function Dfs(v, currentColour)
    colour[v] ← currentColour
    for each node u in V adjacent to v do
        if u is marked with currentColour then
            output False and exit
        if u is marked with 0 then
            Dfs(u, 3 − currentColour)
```

As for 3-colourability and onwards, it turns out this is an NP-Complete problem, that is, it's the hardest class of problem we know. In practice this means that we only have exponential time algorithms to compile such a property of a graph.

To understand why we can't apply the same strategy for more than 2 colours we just need to think about the "choices" the algorithm needs to make at each step. For 2-colourability there is no choice, as a node has to be different to the node we're coming from. In 3-colourability and onwards the algorithm would have to start trying mutliple different combinations – this gives rise to the need for exponential time algorithms.

5. Single Source Shortest Path with Dijkstra's Algorithm



The following table provides the values of the priority queue at each time step (each column corresponds to each time step). The vertices which have already been removed from the queue do not have entries in that column. The subscript for each distance in the table indicates the vertex from which the vertex in question was added to the priority queue from. This is useful for tracing back shortest paths.

First with E as the source:

Node							
A	∞	∞	∞	8_D	8_D		
B	∞	9_E	9_E	9_E	9_E	9_E	
C	∞	∞	∞	9_D	9_D	9_D	9_D
D	∞	7_E	7_E				
E	0						
F	∞	∞	7_G	7_G			
G	∞	6_E					

Now with A as the source

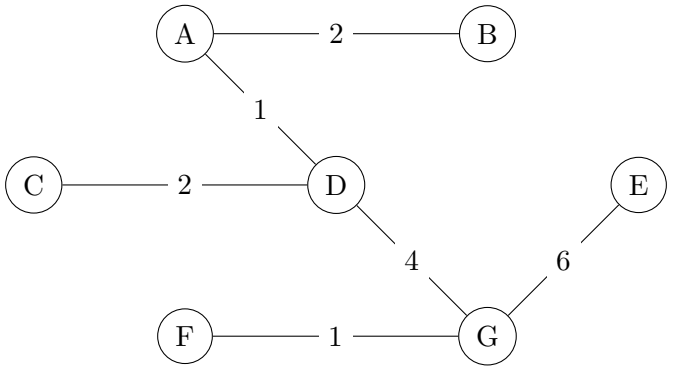
Node							
A	0						
B	∞	2_A	2_A				
C	∞	4_A	3_D	3_D			
D	∞	1_A					
E	∞	∞	8_D	8_D	8_D	8_D	8_D
F	∞	∞	9_D	9_D	8_C	6_G	
G	∞	∞	5_D	5_D	5_D		

So the shortest path from E to A is cost 8 and goes $E \to D \to A$. The shortest path from A to F is cost 6 and goes $A \to D \to G \to F$.

6. Minimum Spanning Tree with Prim's Algorithm Running Prim's is almost the same as Dijkstra's, making a greedy (locally optimal) decision at each time step and taking the lowest cost vertex from the priority queue. The main difference is we take the cost of the single edge which connects each new vertex to the tree, rather than a cumulative cost. Again, keeping track of the vertex we come from gives us an easy way to read off the minimum spanning tree from the table.

Node							
A	0						
B	∞	2_A	2_A				
C	∞	4_A	2_D	2_D			
D	∞	1_A					
E	∞	∞	7_D	7_D	7_D	6_G	6_G
F	∞	∞	8_D	8_D	5_C	1_G	
G	∞	∞	4_D	4_D	4_D		

Summing up the final entry in each row gives us the total cost of the minimum spanning tree: 16. To find the edges in the minimum spanning tree we reach the vertex from the end of each row (except for A), e.g., (B, A), (C, D), (D, A), (E, G), (F, G), (G, D).



COMP20007 Design of Algorithms

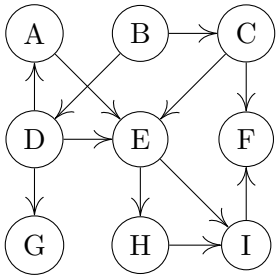
Week 7 Workshop

Tutorial

1. Topological sorting A *topological ordering* of a directed acyclic graph (DAG) is a way of sorting the nodes of the graph such that all edges point in one direction: to nodes later in the ordering.

One of the algorithms discussed in lectures involves running a DFS on the DAG and keeping track of the order in which the vertices are popped from the stack. The topological ordering will be the reverse of this order.

Find a topological ordering for the following graph.

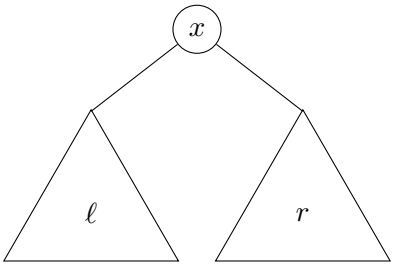


2. Negative edge weights Dijkstra's algorithm, unmodified, can't handle some graphs with negative edge weights. Your friend has come up with a modified algorithm for finding shortest paths in a graph with negative edge weights:

1. Find the largest negative edge weight, call this weight $-w$.
2. Add w to the weight of all edges in the graph. Now, all edges have non-negative weights.
3. Run Dijkstra's algorithm on the resulting non-negative-edge-weighted graph.
4. For each path found by Dijkstra's algorithm, compute its true cost by subtracting w from the weight of each of its edges.

Will your friend's algorithm work?

3. Binary tree traversals When traversing a binary tree we have a decision to make about order. If we represent a general binary tree as follows, where x is the root, and ℓ and r are the (possibly empty) left and right sub-trees respectively, we can discuss different traversal orders more concretely.



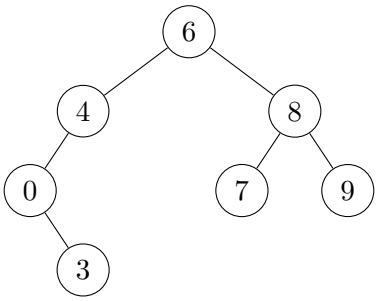
Now, we can define the traversal orders like so:

Inorder: traverse ℓ, visit x, traverse r

Preorder: visit x, traverse ℓ, traverse r

Postorder: traverse ℓ, traverse r, visit x

Write the *inorder*, *preorder* and *postorder* traversals of the following binary tree:



4. Level-order traversal of a binary tree A level-order of a binary tree first visits the root, then the left child of the root, then the right child of the root, then the left child of the left child (the leftmost grandchild of the root) *etc.*, going left to right at each level.

In which order are the nodes of the binary tree from Question 3 visited in a level-order traversal.

Which graph traversal algorithm could we modify to perform a level-order traversal of a binary tree? Write the pseudo-code.

5. Binary tree sum Write a recursive algorithm to calculate the sum of a binary tree where each node contains a number.

Confirm that your algorithm works on the tree from Question 3.

Which order does your algorithm process nodes in? *Inorder, preorder* or *postorder*?

Computer Lab

There is no Computer Lab this week.

COMP20007 Design of Algorithms

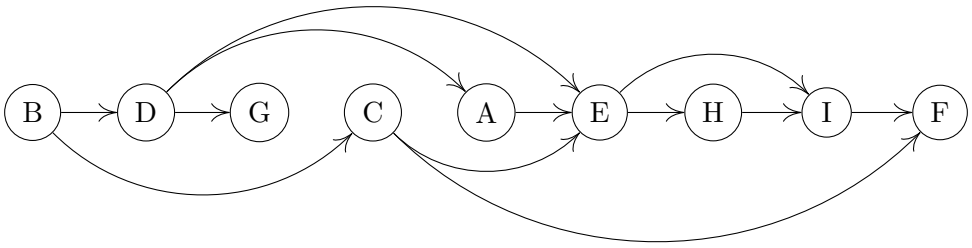
7 Workshop Solutions

Tutorial

1. Topological sorting Running depth-first search from A results in the following sequence of operations (and resulting stacks):

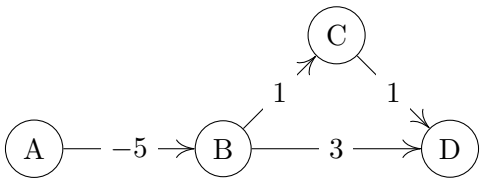
push A	$S = \{A\}$
push E	$S = \{A, E\}$
push H	$S = \{A, E, H\}$
push I	$S = \{A, E, H, I\}$
push F	$S = \{A, E, H, I, F\}$
pop F	$S = \{A, E, H, I\}$
pop I	$S = \{A, E, H\}$
pop H	$S = \{A, E\}$
pop E	$S = \{A\}$
pop A	$S = \{\}$
push B	$S = \{B\}$
push C	$S = \{B, C\}$
pop C	$S = \{B\}$
push D	$S = \{B, D\}$
push G	$S = \{B, D, G\}$
pop G	$S = \{B, D\}$
pop D	$S = \{B\}$
pop B	$S = \{\}$

Taking the order of nodes popped and reversing it we get a topological ordering. So if we use a depth-first search starting from A, we get the topological ordering B, D, G, C, A, E, H, I, F. Rearranged into this order, the graph's edges all point from left to right:

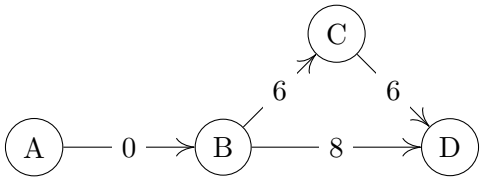


2. Negative edge weights Your friend's algorithm might sound like a good idea, but it sadly won't cure Dijkstra's algorithm of its inability to handle negative edge weights properly. Simply adding a constant value to the weight of each edge distorts the length of paths differently depending on how many edges they contain. Therefore the shortest paths found by Dijkstra's algorithm in the modified graph might not correspond to true shortest paths in the original graph.

As an example, consider the graph below.



The shortest path from A to D is A, B, C, D. However, when you add 5 to every edge, the shortest path becomes A, B, D:



(Interestingly, however, a similar idea forms the basis of a fast *all pairs, shortest path* algorithm called Johnson's algorithm. See `https://en.wikipedia.org/wiki/Johnson%27s_algorithm` for details, it's an interesting read!)

3. Binary tree traversals The traversal orders of the example binary tree provided are:

Inorder: $0, 3, 4, 6, 7, 8, 9$

Preorder: $6, 4, 0, 3, 8, 7, 9$

Postorder: $3, 0, 4, 7, 9, 8, 6$

4. Level-order traversal of a binary tree The level-order traversal will visit the nodes in the tree in left to right order starting at the root, then doing the first level, the second level *etc.* For the binary tree in Question 3 the level-order traversal will be: 6, 4, 8, 0, 7, 9, 3.

We can use breadth-first search to traverse a binary tree in level-order, as long as we break ties by selecting left children first.

```
function LevelOrder(root)
    init(queue)
    enqueue(queue, root)
    while queue is not empty do
        node ← dequeue(queue)
        visit node
        if leftChild(node) is not NULL then
            enqueue(queue, leftChild(node))
        if rightChild(node) is not NULL then
            enqueue(queue, rightChild(node))
```

5. Binary tree sum Our recursive Sum algorithm will return the sum of the subtree T. We'll process the nodes in pre-order traversal order.

```
function Sum(T)
    if T is non-empty then
        sum ← value(T_root)
        sum ← sum + Sum(T_left)
        sum ← sum + Sum(T_right)
        return sum
    else
        return 0
```

COMP20007 Design of Algorithms

Week 8 Workshop

Tutorial

1. Master Theorem The Master Theorem states that if we have a recurrence relation $T(n)$ such that

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d),$$
$$T(1) = c,$$

then,

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

Note: a similar formula holds for O and Ω as well.

Apply the Master Theorem to find the time complexities of the following recurrence relations (in Big-Theta terms).

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

(d) $T(n) = 2T\left(\frac{n}{2}\right),\ T(1) = 1$

2. Simple Sorting Algorithms In lectures before the break we saw the following sorting algorithms,

(a) Selection Sort

(b) Insertion Sort

(c) Quicksort (with Lomuto partitioning)

Answer the following questions about each algorithm:

(i) Run the algorithm on the following input array:

$$\boxed{\text{A N A L Y S I S}}$$

(ii) What is the time complexity of the algorithm?

(iii) Is the sorting algorithm stable[1]?

(iv) Does the algorithm sort in-place[2]?

(v) Is the algorithm input sensitive[3]?

If you get time, try to answer these questions for (d) Quicksort (with Hoare partitioning), and (e) Merge Sort.

[1] a sorting algorithm is *stable* if the relative order of elements with the same value is preserved.

[2] Sorting *in-place* is when only $O(1)$ additional space is required.

[3] An algorithm is *input sensitive* if the runtime depends on properties of the input other than its size, for instance whether or not the input is already sorted.

3. Mergesort Time Complexity Mergesort is a divide-and-conquer sorting algorithm made up of three steps (in the recursive case):

1. Sort the left half of the input (using mergesort)
2. Sort the right half of the input (using mergesort)
3. Merge the two halves together (using a merge operation)

Construct a recurrence relation to describe the runtime of mergesort sorting n elements. Explain where each term in the recurrence relation comes from.

Use the Master Theorem to find the time complexity of Mergesort in Big-Theta terms.

4. Lower bound for the Closest Pairs problem The closest pairs problem takes n points in the plane and computes the Euclidean distance between the closest pair of points.

The algorithm provided in lectures to solve the closest pairs problem applies the divide and conquer strategy and has a time complexity of $O(n \log n)$.

The *element distinction problem* takes as input a collection of n elements and determines whether or not all elements are distinct. It has been proved that if we disallow the usage of a hash table[4] then this problem cannot be solved in less than $n \log n$ time (*i.e.*, this class of problems is $\Omega(n \log n)$).

Describe how we could use the closest pair algorithm from class to solve the element distinction problem (where the input is a collection of floating point numbers), and hence explain why this proves that the closest pair problem must not be able to be solved in less that $n \log n$ time (and is thus $\Omega(n \log n)$).

[4]Excluding the use of a hash table is done to conform to the *algebraic decision tree* model of computation, where we cannot use the values of the elements to index into memory, and only allows us to compute and compare the elements themselves (and simple functions of these elements).

Computer Lab

For today's computer lab we've provided a graph module, a list module and a naive priority queue module (later in the semester we'll use a min-heap to re-implement this module).

Your task will be to implement a number of graph algorithms using the modules provided.

We have already implemented a DFS in the `graphalgs` module for you as an example.

You should read through the provided code and understand the example provided. Then you should implement the following graph algorithms in the graph algorithms package.

We've provided a number of input graphs for you to test your algorithm with.

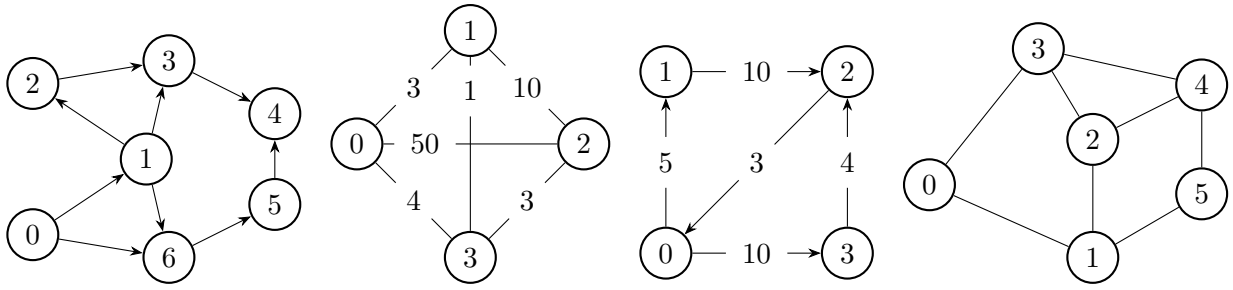


Figure: The graphs represented in `graph-01.txt`, `graph-02.txt`, `graph-03.txt` and `graph-04.txt`.

1. BFS We've written a `dfs` function which returns an array with the order in which nodes are explored using a depth-first search.

Write your own algorithm with the same behaviour, except this time for a **breadth-first search**. You may find using an iterative aproach easier for this task.

Your algorithm should have the signature `int *bfs(Graph *graph);`

2. Prim's Algorithm Write a function which performs Prim's algorithm on the input graph. Remember that Prim's algorithm computes the *minimum spanning tree* for a connected undirected graph.

Your function should take in pointers to two arrays `from` and `to` which have length $n-1$ and fill them so that the edges that make up the minimum spanning tree are:

(`from[0]`, `to[0]`), ..., (`from[n-1]`, `to[n-1]`)

Your function should return whether or not the algorithm was succesful.

The signature should be `bool prims(Graph *graph, int *from, int *to);`

3. Dijkstra's Algorithm Write a function which performs Dijkstra's algorithm on the input graph. Remember that Dijkstra's algorithm is a *single source shortest paths* (SSSP) algorithm which finds the shortest paths from a given start node to every other node.

Your function should take a pointer to an array `dist` which has length n and store the distances from the start node to each other node, or -1 if there is no path from the start node to that node.

Your function signature should be `void dijkstras(Graph *graph, int start, int *dist);`

(*Challenge*) Modify your function to store the paths from the start node to each other node in an array of linked lists or an array of arrays. Which data struture will be easier to work with?

COMP20007 Design of Algorithms

Week 8 Workshop Solutions

Tutorial

1. Master Theorem Note for this question that comparing a and b^d is the same as comparing $\log_b a$ and d.

(a) $T(n) = 9T\left(\frac{n}{3}\right) + n^3,\ T(1) = 1$

We have $a = 9, b = 3, d = 3$ and $c = 1$. So $\log_b(a) = \log_3(9) = 2$.

Also $2 < 3 = c$, so the $\Theta(n^3)$ is the dominating term. So $T(n) \in \Theta(n^3)$.

(b) $T(n) = 64T\left(\frac{n}{4}\right) + n + \log n,\ T(1) = 1$

We have $a = 64$ and $b = 4$, so $\log_b(a) = \log_4(64) = 3$.

Also, since $n + \log n \in \Theta(n^1)$, $d = 1 < 3$, so $T(n) \in \Theta(n^3)$.

(c) $T(n) = 2T\left(\frac{n}{2}\right) + n,\ T(1) = 1$

We have $a = b = 2$ so $\log_b(a) = \log_2(2) = 1$. Also $n \in \Theta(n^1)$ so $d = 1$ as well.

So $T(n) \in \Theta(n^d \log n) = \Theta(n \log n)$.

(d) $T(n) = 2T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{2}\right) + \Theta(1),\ T(1) = 1$

Here $a = b = 2$ and $d = 0$ so $b^d = 1 < 2 = a$, thus we get $\Theta(n^{\log_2 2}) = \Theta(n)$.

2. Simple Sorting Algorithms

(a) *Selection Sort.* When running selection sort the key idea is that for each i from 0 to $n - 2$ we find the smallest (or largest if we're sorting in descending order) element in $A[i \ldots n - 1]$ and swap it with $A[i]$.

(i) We'll show the elements at the start of the array which we have already sorted (i.e., the

elements $A[0 \ldots i-1]$) in bold, and the minimum element in $A[i \ldots n-1]$ in red.

$$
\begin{bmatrix} \text{A N A L Y S I S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A}\ \text{N A L Y S I S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A}\ \text{N A L Y S I S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A}\ \text{N L Y S I S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A}\ \text{N L Y S I S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I}\ \text{L Y S N S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I}\ \text{L Y S N S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L}\ \text{Y S N S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L}\ \text{Y S N S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L\ N}\ \text{S Y S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L\ N}\ \text{S Y S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L\ N\ S}\ \text{Y S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L\ N\ S}\ \text{Y S} \end{bmatrix}
$$
$$
\begin{bmatrix} \mathbf{A\ A\ I\ L\ N\ S\ S}\ \text{Y} \end{bmatrix}
$$

(ii) At each $i \in \{0, \ldots, n-2\}$ we must take the minimum element from the array $A[i \ldots n-1]$, which requires $n-1-i$ comparisons. So the total number of comparisons, $C(n)$, will be:

$$
\begin{aligned}
C(n) &= \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i \\
&= (n-1)(n-1) - \frac{(n-2)(n-1)}{2} \\
&= \frac{1}{2}\Big(2n^2 - 4n + 2 - n^2 + 3n - 2\Big) \\
&= \frac{1}{2}\Big(n^2 - n\Big) \\
&\in \Theta(n^2)
\end{aligned}
$$

(iii) Selection sort is **not** stable. Consider the following counter example (where 1_a and 1_b are

used to differentiate between the two elements with the same value).

$$\begin{bmatrix} 1_a & 1_b & 0 \end{bmatrix}$$
$$\begin{bmatrix} 1_a & 1_b & 0 \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & 1_b & 1_a \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & 1_b & 1_a \end{bmatrix}$$
$$\begin{bmatrix} \mathbf{0} & \mathbf{1_b} & 1_a \end{bmatrix}$$

Since 1_a and 1_b end up out of order relative to one another, this sorting algorithm is *not* stable.

(iv) Yes, selection sort sorts *in-place*, as it requires only $O(1)$ additional memory.

(v) No selection sort is not input sensitive, the number of comparisons is a function of n only and the order of the elements has no effect.

(b) *Insertion Sort.* The key idea of insertion sort is that we have some section at the start of the array which is sorted (initially only $A[0]$) and we repeatedly *insert* the next element into the correct position in this sorted segment, until the whole array is sorted.

More precisely, with i going from 1 to $n-1$ we have $A[0 \ldots i-1]$ already sorted, and we swap $A[i]$ backwards until it is in the correct position in the sorted section $A[0 \ldots i]$ (*i.e.*, it's greater then the element immediately preceding it).

(i) For each i we'll show the sorted section in bold, and then when we're performing the insertion we'll show the element we're inserting in red, and the element we're compar-

ing/swapping

$$[\mathbf{A}\ \mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{N}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{N}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{N}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{N}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{L}\ \mathbf{N}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{L}\ \mathbf{N}\ \mathbf{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{L}\ \mathbf{N}\ \mathbf{S}\ \mathbf{Y}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{I}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{I}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{I}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{I}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}\ \mathrm{I}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{I}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{L}\ \mathrm{I}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{I}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{I}\ \mathbf{L}\ \mathbf{N}\ \mathbf{S}\ \mathbf{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{I}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}]$$
$$[\mathrm{A}\ \mathrm{A}\ \mathrm{I}\ \mathrm{L}\ \mathrm{N}\ \mathrm{S}\ \mathrm{S}\ \mathrm{Y}]$$
$$[\mathbf{A}\ \mathbf{A}\ \mathbf{I}\ \mathbf{L}\ \mathbf{N}\ \mathbf{S}\ \mathbf{S}\ \mathbf{Y}]$$

(ii) The best case for insertion sort is when the array is already sorted and there are no swaps to be made. There are exactly $n-1$ comparisons made in this case, thus insertion sort is $\Omega(n)$.

On the other hard, the worst case input is when the input is sorted in reverse order, and for each $i \in \{1, \ldots, n-1\}$ there are i swaps to be made (and hence i comparisons), thus the number of comparisons is:

$$C(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in O(n^2).$$

So the worst case time complexity of insertion sort is $O(n^2)$.

(iii) Insertion sort *is stable*. The only way two elements relative order will be reversed is if they are swapped with each other directly. Since we do not swap elements of equal value it must be the case that relative orderings of elements with equal value must be maintained.

(iv) Insertion sort does sort in place, there is only a constant amount of additional memory required.

(v) Insertion sort is input sensitive, as we can see from (iii) where we show that the time complexity is different depending on how the input is arranged.

(c) *Quicksort (with Lomuto partitioning).* Quicksort is a recursive algorithm which repeatedly partitions the input array and recursively quicksorts the left and right portions of the array which $< p$ and $\geq p$ for the pivot p, respectively.

The *Quicksort* algorithm is:

```
function QUICKSORT(A[l ... r])
    if l < r then
        s ← PARTITION(A[l ... r])
        QUICKSORT(A[l ... s − 1])
        QUICKSORT(A[s + 1 ... r])
```

The *Lomuto* partitioning algorithm is:

```
function LOMUTOPARTITION(A[l ... r])
    p ← A[l]
    s ← l
    for i ← l + 1 to r do
        if A[i] < p then
            s ← s + 1
            SWAP(A[s], A[i])
    SWAP(A[l], A[s])
    return s
```

(i) Performing the algorithm on [A N A L Y S I S]:

Partition [A N A L Y S I S]

$$\left[\mathrm{A}_s\ \mathrm{N}_i\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\right] \to \left[\mathrm{A}_s\ \mathrm{N}\ \mathrm{A}_i\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\right] \to \ldots \to \left[\mathrm{A}_s\ \mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}_i\right]$$

So $s = 1$ and we're left with (where bold characters are already fixed in place):

$$\left[\mathbf{A}\ \mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\right]$$

So partitioning [N A L Y S I S]:

$$\Big[\mathrm{N}_s\ \mathrm{A}_i\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}_{s,i}\ \mathrm{L}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}_s\ \mathrm{L}_i\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\Big]$$
$$\to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}_{s,i}\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}_s\ \mathrm{Y}_i\ \mathrm{S}\ \mathrm{I}\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}_s\ \mathrm{Y}\ \mathrm{S}_i\ \mathrm{I}\ \mathrm{S}\Big]$$
$$\to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}_s\ \mathrm{Y}\ \mathrm{S}\ \mathrm{I}_i\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{Y}_s\ \mathrm{S}\ \mathrm{I}_i\ \mathrm{S}\Big]$$
$$\to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{I}_s\ \mathrm{S}\ \mathrm{Y}_i\ \mathrm{S}\Big] \to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{I}_s\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}_i\Big] \to \Big[\mathrm{N}\ \mathrm{A}\ \mathrm{L}\ \mathrm{I}_s\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}\Big] \to \Big[\mathrm{I}\ \mathrm{A}\ \mathrm{L}\ \mathrm{N}_s\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}\Big]$$

So we have,

$$\Big[\mathbf{A}\ \mathrm{I}\ \mathrm{A}\ \mathrm{L}\ \mathbf{N}\ \mathrm{S}\ \mathrm{Y}\ \mathrm{S}\Big]$$

So we need to recursively Quicksort [I A L] and [S Y S].

Starting with [I A L]:

$$\Big[\mathrm{I}_s\ \mathrm{A}_i\ \mathrm{L}\Big] \to \Big[\mathrm{I}\ \mathrm{A}_{s,i}\ \mathrm{L}\Big] \to \Big[\mathrm{I}\ \mathrm{A}_s\ \mathrm{L}_i\Big]$$
$$\to \Big[\mathrm{I}\ \mathrm{A}_s\ \mathrm{L}\Big] \to \Big[\mathrm{A}\ \mathrm{I}_s\ \mathrm{L}\Big]$$

So we have [A I L], and we recursively quicksort [A] and [L] (the base case, so we do nothing).

Now we have [**A A I L N** S Y S] and we need to quicksort [S Y S]:

$$\Big[\mathrm{S}_s\ \mathrm{Y}_i\ \mathrm{S}\Big] \to \Big[\mathrm{S}_s\ \mathrm{Y}\ \mathrm{S}_i\Big] \to \Big[\mathrm{S}_s\ \mathrm{Y}\ \mathrm{S}\Big]$$

Nothing was moved, but we now have [**A A I L N S** Y S] and we need to recursively quicksort [Y S]:

$$\Big[\mathrm{Y}_s\ \mathrm{S}_i\Big] \to \Big[\mathrm{Y}\ \mathrm{S}_{s,i}\Big] \to \Big[\mathrm{Y}\ \mathrm{S}_s\Big] \to \Big[\mathrm{S}\ \mathrm{Y}_s\Big]$$

So we have [**A A I L N S** S **Y**] and the final recursive call for [Y] hits the base case, and we're finished! So the final sorted array is [A A I L N S S Y]

(ii) From the lectures Quicksort is $\Theta(n \log n)$ in the best case and $\Theta(n^2)$ in the worst case (for example when the array is in reverse sorted order we do n partitions of cost $\Theta(n)$).

(iii) Quicksort with Lomuto partitioning is not stable. Consider the counter example:

$$\Big[2\ 1^a\ 1^b\Big] \to \Big[2_s\ 1^a_i\ 1^b\Big] \to \Big[2\ 1^a_{s,i}\ 1^b\Big] \to \Big[2\ 1^a_s\ 1^b_i\Big] \to \Big[2\ 1^a\ 1^b_{s,i}\Big] \to \Big[2\ 1^a\ 1^b_s\Big] \to \Big[1^b\ 1^a\ 2_s\Big]$$

After the first partition we're left with $[1^b\ 1^a]$ to recursively quicksort. Partitioning this with Lomuto partitioning leaves it as is (check this!) and so our input $[2\ 1^a\ 1^b]$ becomes $[1^b\ 1^a\ 2]$ after being quicksorted. The relative order of the 1s was not preserved so this algorithm must not be stable.

(iv) The algorithm is in place. Here we're allowing the $O(\log n)$ space required for the function stack when we do the recursive calls.

(v) This algorithm is input sensitive as demonstrated by the different best and worst case time complexities depending on the input order.

3. Mergesort Time Complexity Recurrence relation:

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

where $T(n)$ is the runtime of mergesort sorting n elements. The first $T(\frac{n}{2})$ is the time it takes to sort the left half of the input using mergesort. The other $T(\frac{n}{2})$ is the time it takes to sort the right half. $\Theta(n)$ is a bound on the time it takes to merge the two halves together.

Recall that the Master Theorem states that if we have a recurrence relation $T(n)$ such that

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d),$$
$$T(1) = c,$$

then,

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

We can recognise that the mergesort recurrence relation fits the form required by the Master Theorem, with constants $a = 2$, $b = 2$, and $d = 1$.

$$b^d = 2 = a$$

so, by the master theorem, $T(n) \in \Theta(n \log n)$.

4. Lower bound for the Closest Pairs problem We can use the closest pair algorithm from class to solve the element distinction problem like so, where the output is a collection of elements $C = \{c_1, \ldots, c_n\}$ and the output is DISTINCT or NOTDISTINCT:

```
function ElementDistinction(C = {c_1, ..., c_n})
    Points ← {(c_1, 0), ..., (c_n, 0)}
    Distance ← ClosestPair(Points)
    if Distnace is 0 then
        return NotDistinct
    else
        return Distinct
```

So, we can see that we can solve the element distinct problem using the closest pair algorithm. This is called a *reduction* from element distinction to closest pair.

We know that ELEMENTDISTINCTION is $\Omega(n \log n)$, and want to prove that CLOSESTPAIR is also $\Omega(n \log n)$.

We assume for the sake of contradiction that CLOSESTPAIR can be solved in a time complexity smaller (*i.e.*, asymptotically faster) that $n \log n$. As we have exhibited (provided) a reduction from ELEMENTDISTINCTION to CLOSESTPAIR then this must also give us an algorithm for ELEMENTDISTINCTION which is asymptotically faster than $n \log n$.

This contradicts the statement that ELEMENTDISTINCTION is $\Omega(n \log n)$, and as a result our assumption that CLOSESTPAIR can be solved in a time complexity smaller (*i.e.*, asymptotically faster) that $n \log n$ must be false.

Hence CLOSESTPAIR can not be solved in faster than $n \log n$ time, and is therefore $\Omega(n \log n)$.

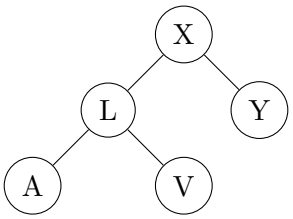
COMP20007 Design of Algorithms

Week 9 Workshop

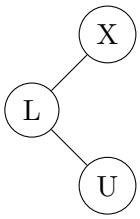
Tutorial

1. Rotations In the following binary trees, rotate the 'X' node to the right (that is, rotate it and its left child). Do these rotations make the tree more balanced, or less balanced?

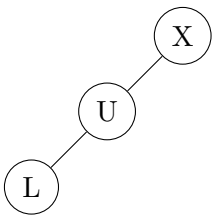
(a)



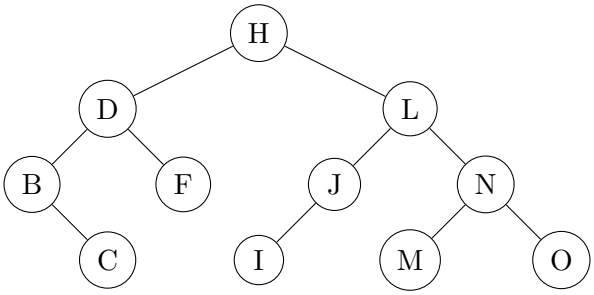
(b)



(c)



2. Balance factor A node's 'balance factor' is defined as the height of its right subtree minus the height of its left subtree. Calculate the balance factor of each node in the following binary search tree.



3. AVL Tree Insertion Insert the following letters into an initially-empty AVL Tree.

A V L T R E X M P

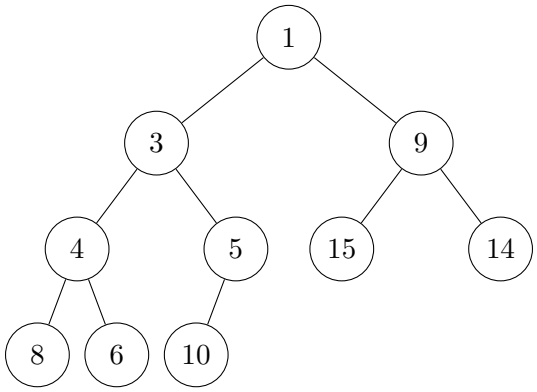
4. 2-3 Tree Insertion Insert the following letters into an initially-empty 2-3 Tree.

A L G O R I T H M

5. Heaps A *Heap* (for this question we will discuss a min-heap) is a complete binary tree which satisfies the heap property:

Heap Property: each node in the tree is no larger than its children.

Suppose we start out with the following heap:



(a) Show how this heap would be stored in an array as discussed in lectures, *i.e.*, the root is at index 1 and a node at index i has children in indices $2i$ and $2i + 1$.

(b) Run the RemoveRootFromHeap algorithm from lectures on this heap by hand (*i.e.*, swap the root and the "last"element and remove it. To maintain the heap property we then SiftDown from the root).

(c) Run the InsertIntoHeap algorithm and insert the value 2 into the heap (*i.e.*, add the new value to the end of the heap, and compare it with its parent, swapping the new value and its parent until its parent is smaller than it, or it is the root).

6. (Optional) Using a min-heap for kth-smallest element The *kth-smallest element problem* takes as input an array A of length n and an index $k \in \{0, \ldots, n-1\}$ and returns as output the kth-smallest element in A (with the 0th-smallest being the smallest, the 1th-smallest being the second smallest and so on).

How can we use a the min-heap data structure to solve the kth-smallest element problem? What is the time-complexity of this algorithm?

7. (Optional) Quickselect Quicksort uses a Partition$(A, pivot)$ function which partitions the array A to satisfy the constraint that all elements smaller than the *pivot* occur before it in the array, and those greater than the *pivot* occur after in. In quicksort we call Partition, keep track of the final index of the *pivot*, p and call Partition recursively on $A[0 \ldots p-1]$ and $A[p+1 \ldots n-1]$.

(a) Design an algorithm based on Quicksort which uses the Partition algorithm to find the kth-smallest element in an array A.

Hint: We know that once we partition the array, the index of the *pivot*, p, must be the correct position of *pivot* in the final sorted array. Can we use this fact to deduce where the kth smallest element must be?

(b) Show how you can run your algorithm to find the kth-smallest element where $k = 4$ and $A = \Big[9, 3, 2, 15, 10, 29, 7\Big]$.

(c) What is the best-case time-complexity of your algorithm? What type of input will give this time-complexity?

(d) What is the worst-case time-complexity of your algorithm? What type of input will give this time-complexity?

(e) What is the expected-case (*i.e.*, average) time-complexity of your algorithm?

(f) When would we use this algorithm instead of the heap based algorithm from Question 5?

Computer Lab

In this weeks lab we're going to experiment with different sorting functions, and different implementations of sorting functions.

In the lab files for this week we have provided,

- A function for timing a sorting algorithm: `time_sorting_function()`
- A function for generating random arrays of different sizes: `generate_array()`. This function can generate random arrays, sorted arrays, reversed arrays and almost sorted arrays.
- An implementation of `insertion_sort()` and `quicksort()`, however the `quicksort()` algorithm uses a very naive partitioning and pivoting strategy.

In this lab you'll be using the timing function to compare the performance of different sorting algorithms.

1. Lomuto and Hoare Partitioning In `sorting.c` we have implemented a function `partition_first_pivot(` which uses our naive partitioning function to partition the array based on the pivot being the first element in the array.

We can do better than the partitioning function we've created! Our function first allocates additional memory (an expensive operation) and does three passes over the array (to copy over elements smaller than, equal to and greater than the pivot respectively) to construct this intermediate array. It then copies this over to the original array.

Your task is to write a partitioning function which uses the following partition strategies: (a) **Lomuto** partitioning, and (b) **Hoare** partitioning.

Have a look at the last function in `main.c`, `quicksort_first()` to see how we can time these different pivot selection strategies.

1. Quicksort–Insertion Sort Hybrid Although the asymptotic time-complexity of quicksort is better than that of insertion sort, there is a high overhead associated with quicksort, so it may not always be the best choice of sorting algorithm in practice.

Time both quicksort and insertion sort on arrays of varying types and sizes to determine when each algorithm works better.

Use your results to write a hybrid sorting function `hybrid_sort()` which chooses between insertion sort and quicksort depending on the size of the input.

2. Pivot Selection Strategies Experiment with different pivot selection strategies by writing different partition functions in `sort.c` to pivot based on the last element, and a random element.

Use the functions discussed above to time your different pivot selection strategies. Which one works best for which types of arrays?

3. Better Partitioning Read about the *median-of-three* partitioning strategy which can lead to large performance improvements. Implement this median of three partitioning strategy and see if you see any performance improvements.

4. (Optional) Bottom-Up Mergesort In lectures we discussed both the *top-down* and the *bottom-up* strategies for implementing mergesort.

The *bottom-up* strategy involves starting with each pair of successive elements, sorting them with respect to each other, then moving to merge successive chunks of 2 sorted elements, continuing until we're merging the left and right half of the array. We can implement this iteratively.

Have a read of the Wikipedia's section[1] about bottom-up mergesort and implement your own function `mergesort()` which performs this algorithm. How does it compare to the other algorithms?

[1] `https://en.wikipedia.org/wiki/Merge_sort#Bottom-up_implementation`

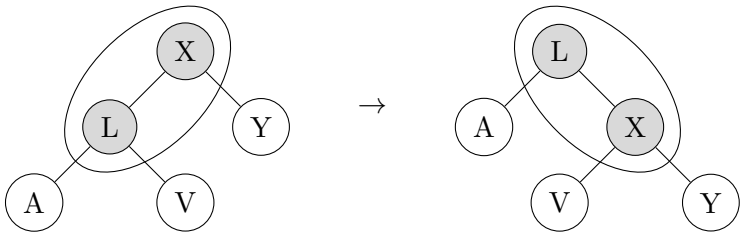
COMP20007 Design of Algorithms

Week 10 Workshop Solutions

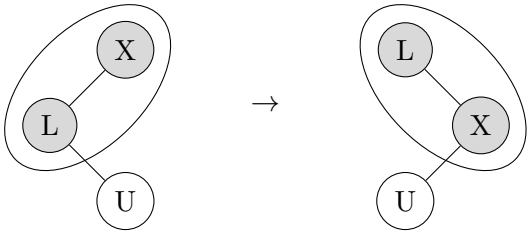
Tutorial

1. Rotations

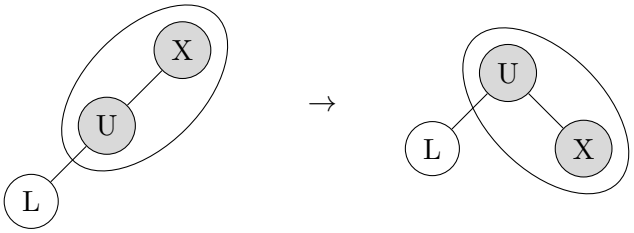
(a) doesn't improve overall balance:



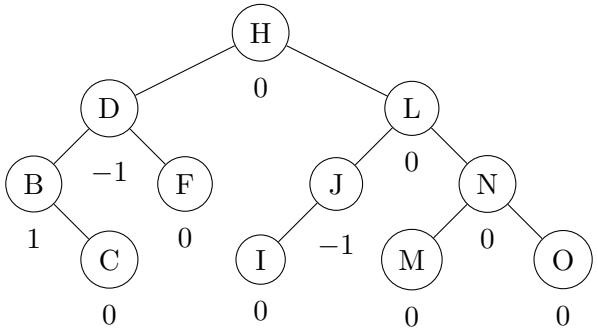
(b) doesn't improve overall balance:



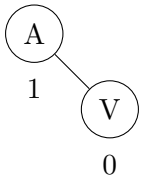
(c) *does* improve overall balance:



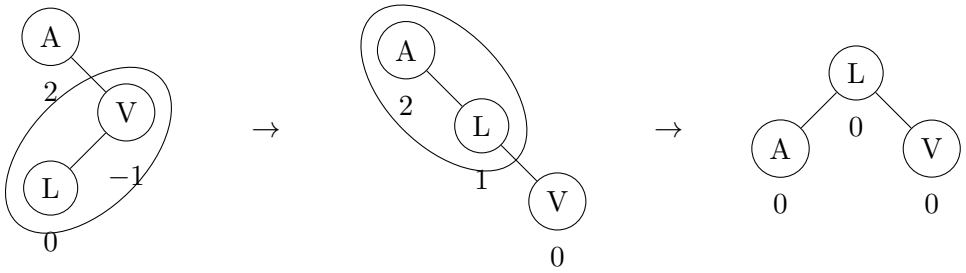
2. Balance factor Balance factor listed below each node. Calculated by subtracting the height of the node's left subtree from the height of its right subtree.



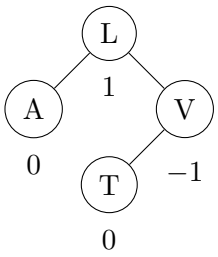
3. AVL Tree Insertion Insertion of A and V is fine, all balanced (balance factor below node):



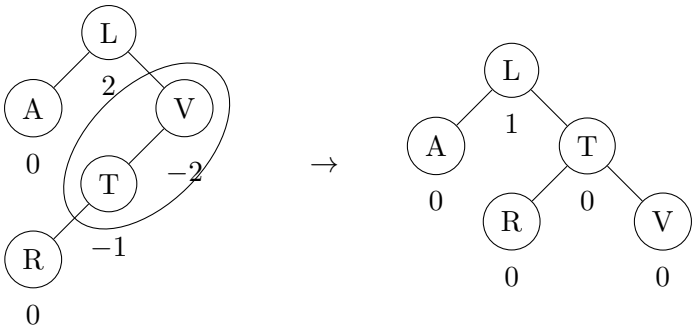
Inserting L causes an imbalance at A, and it's a zig-zag case so we need two rotations to fix it:



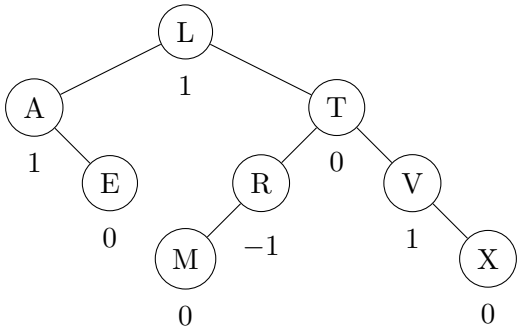
Then, inserting T is fine:



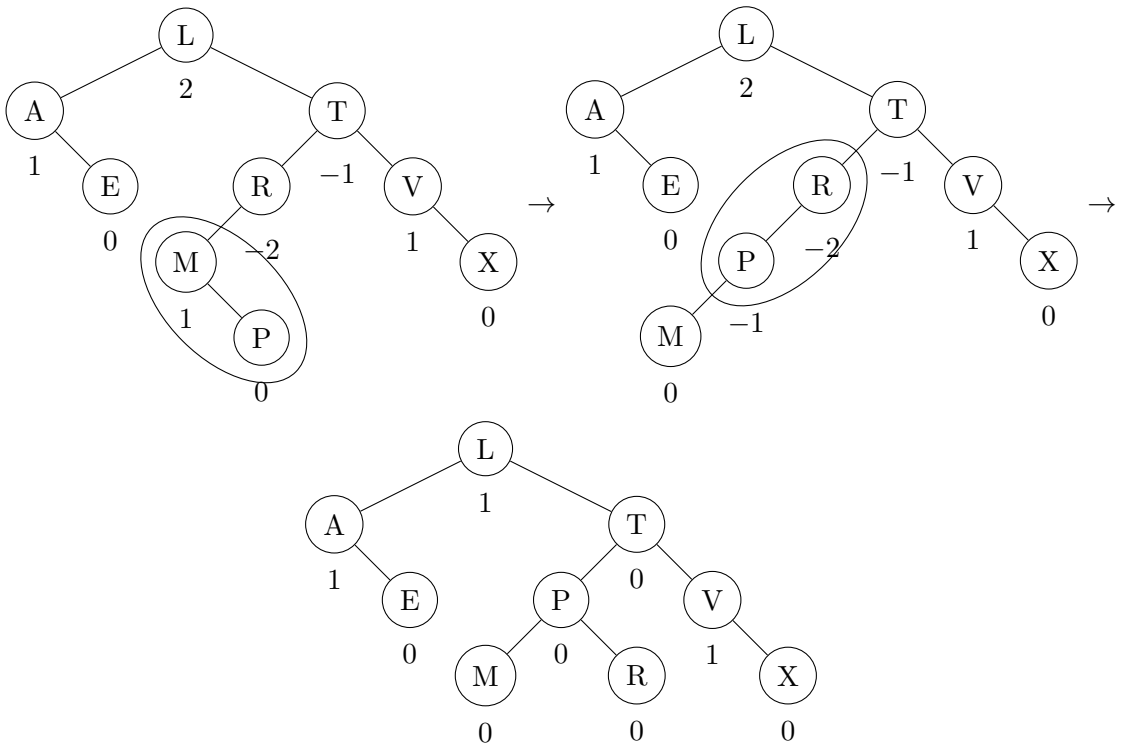
But inserting R gets us into trouble! It causes an imbalance at V. This time it's a zag-zag case, so we only need one rotation:



Next, insertion of E, X, and M cause no imbalances:



The final insertion, P, causes a zag-zig problem at R, once again fixed with two rotations:

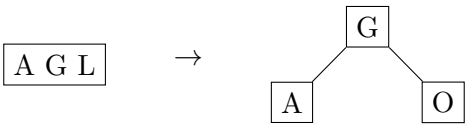


Notes:

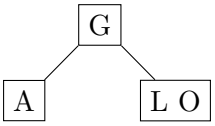
- The terms ‘zig’ and ‘zag’ are used to describe imbalances. A ‘zig-zag’ case, for example, refers to a node with an imbalance because of its right (‘zig’) subtree, and the imbalance in that subtree is coming from the left side (‘zag’).
- To determine which type of imbalance we are dealing with, we can look at the sign of the balance factor. If it’s a +2, that means the problem is in the right child. If the right child has a -1 (different sign), then the problem is with it’s left child, and it’s a zig-zag case. Two rotations are needed. If the right child has a +1 (same sign), then the problem is with its right child, and it’s a zig-zig case. One rotation is needed. Likewise, if the sign at the unbalanced node is -2, then the problem is with the left child. If the left child has a balance factor of -1 (same sign), then the problem is with its left child, i.e. it’s a zag-zag case. One rotation is needed. If the left child has a balance factor of +1 (different sign), then the problem is with its right child, i.e. it’s a zag-zig case. Two rotations are needed.
- Always deal with an imbalance at the deepest available point. For example, when inserting P, both R and L became unbalanced, but we dealt with the problem at R. This automatically fixed the problem at L. In reality, we only calculate balance factors on our way back up the tree from the insertion, so we can simply deal with the first imbalance (+2 or -2 balance factor) we encounter. All balance factors have been shown at all stages in the above diagrams, but in reality only the heights would be stored with each node — balance factors would only be calculated on the way back up the tree after an insertion, and only for the nodes on this direct path.

4. 2-3 Tree Insertion Inserting the first two elemens results in a leaf node:

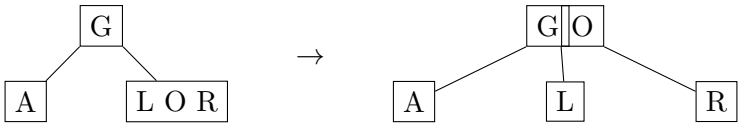
With the insertion of G, our leaf ends up with 3 elements, so it must be split up – we promote the middle element like so:



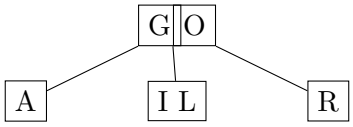
Inserting O adds to the right most leaf node:



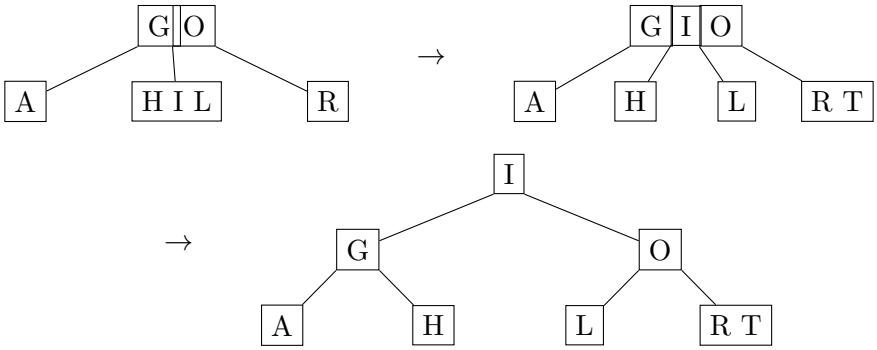
When we add R we get a node with 3 elements and must promote the middle one.



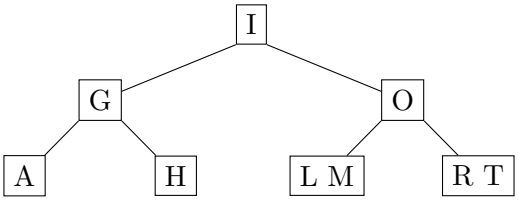
Adding I just adds to a leaf node.



Adding H leaves a node with 3 elements, we will fix this my promoting I twice:



Finally, we can insert M into a leaf node without having to do any promotions, so the final 2-3 tree looks like:

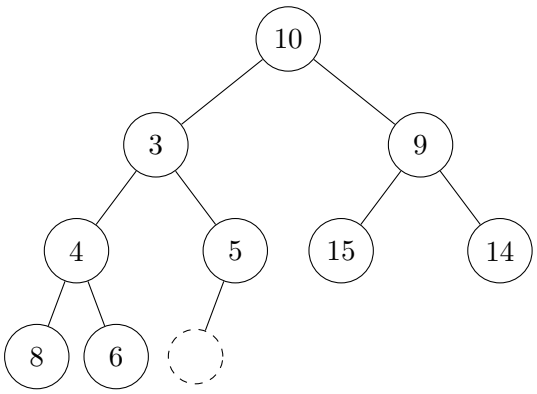


5. Heaps

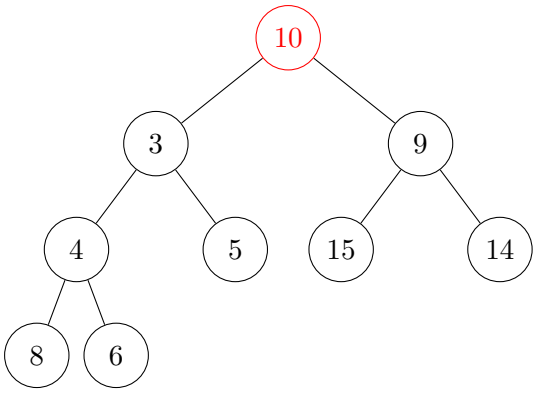
(a) As in the lectures we leave the first index empty, and then populate the array with the elements of the heap in level-order:

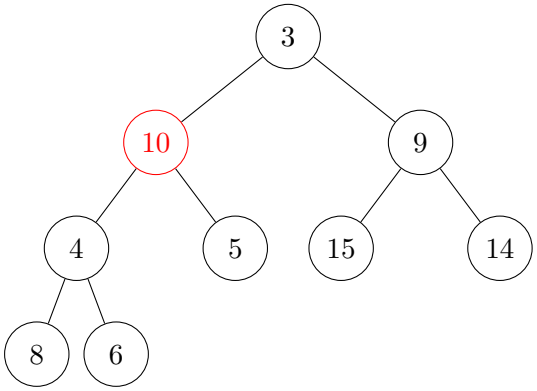
$$\Big[-, 1, 3, 9, 4, 5, 15, 14, 8, 6, 10\Big]$$

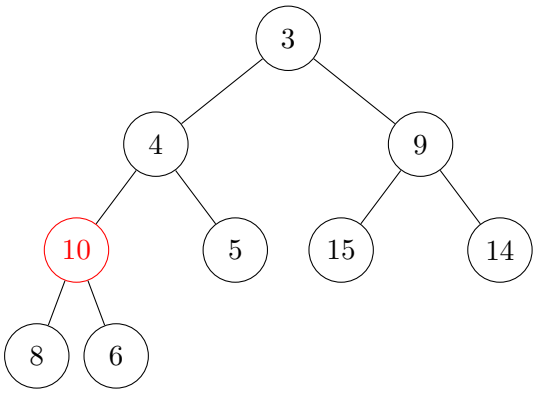
(b) First we store the value of the root of the heap, in this case it is 1, then we swap the last element in the heap to the root:

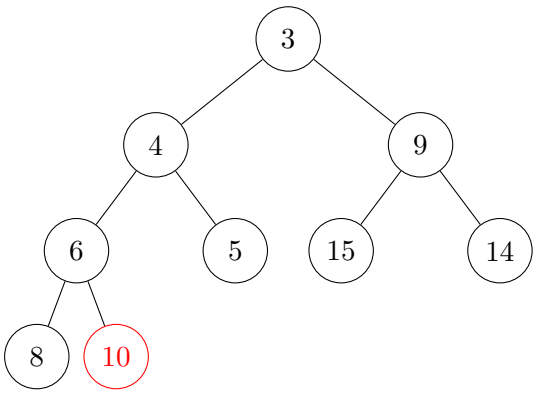


We then SiftDown the root, by comparing it to each of its children, and if it's larger than either swap it with the smaller child and repeat, if it's smaller than both children then stop.

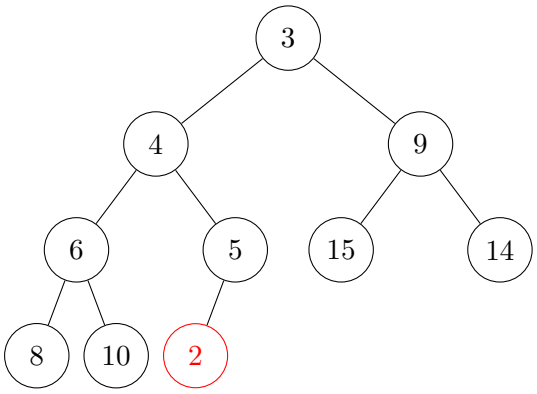


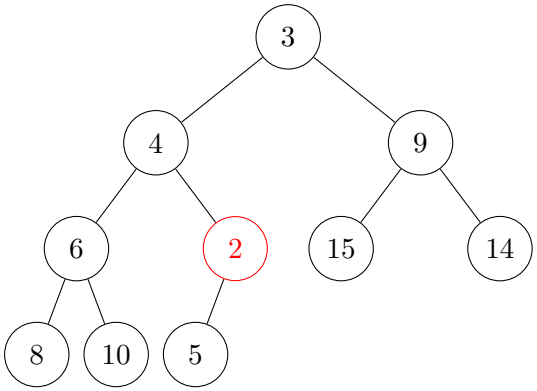


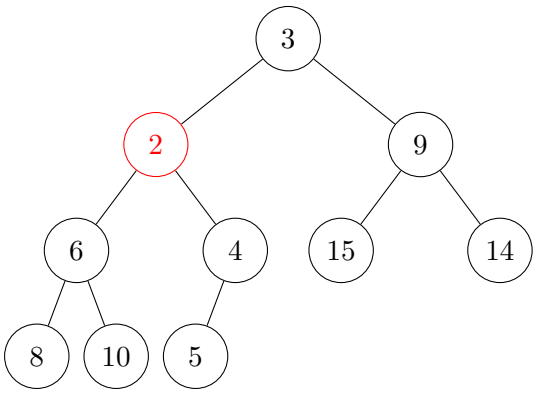


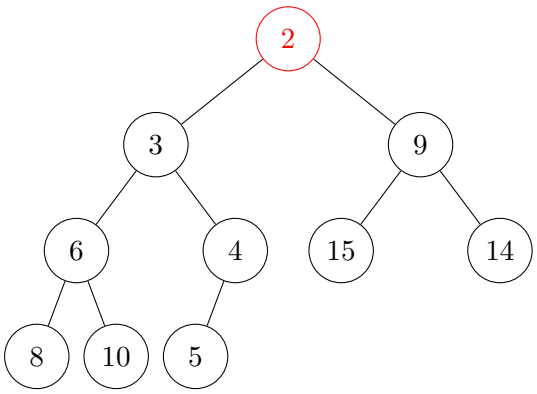


(c) We'll add the new value in a new node at the next spot in the tree, and then SiftUp. Sifting up consists of swapping the node with its parent as long its parent is larger than it.









6. (Optional) Using a min-heap for kth-smallest element Consider the following algorithm which finds the kth-smallest element using a min-heap.

function HEAPkTHSMALLEST($A[0 \ldots n-1], k$)
  $Heap \leftarrow$ HEAPIFY($A[0 \ldots n-1], k$)
  `// remove the smallest` $k-1$ `elements`
  for $i \leftarrow 0 \ldots k-2$ **do**
    REMOVEMIN($Heap$)
  return REMOVEMIN($Heap$)

The time complexity of HEAPIFY is $\Theta(n)$ and each REMOVEMIN is $\Theta(\log n)$, so the total time complexity of this algorithm is $\Theta(n + k \log n)$.

Notice that this algorithm is not input-sensitive on the order of the array, it is only input sensitive with respect to k. This may be a desirable property for a variety of reasons.

7. (Optional) Quickselect *Quickselect* is an algorithm for solving the kth-smallest element problem using the PARTITION algorithm from quicksort.

(a) We know that the *pivot* must occupy index p in the sorted array. We can make use of this fact and only search the section of the array which must contain the element which has the kth index in the sorted array. The pseudocode for quickselect is as follows.

function QUICKSELECT($A[0 \ldots n-1], k$)
  $pivot \leftarrow$ SELECTPIVOT($A[0 \ldots n-1]$)
  `// p is the index of the pivot in the partitioned array`
  $p \leftarrow$ PARTITION($A[0 \ldots n-1], pivot$)
  if $p == k$ **then**
    return $A[k]$
  else if $p > k$ **then**
    return QUICKSELECT($A[0 \ldots p-1], k$)
  else
    return QUICKSELECT($A[p+1 \ldots n-1], k-(p+1)$)

We will assume for this question that we are selecting the first element to be the pivot, but as we have seen in lectures this is not necessarily the best strategy.

(b)

$$\begin{aligned}
\Big[9,3,2,15,10,29,7\Big] &\xrightarrow{\text{PARTITION}} \Big[3,2,7,9,15,10,29\Big],\ p=3<k\\
k &\leftarrow k-(p+1)=0\\
\Big[15,10,29\Big] &\xrightarrow{\text{PARTITION}} \Big[10,15,29\Big],\ p=1>k\\
k &\leftarrow k=0\\
\Big[10\Big] &\xrightarrow{\text{PARTITION}} \Big[10\Big],\ p=0==k\\
&\implies \textbf{return } 10
\end{aligned}$$

(c) The best-case for QUICKSELECT is when $p = k$ after the first partition. With our pivot strategy of selecting $pivot = A[0]$ this corresponds to having the kth-smallest element at the start of the array.

The single PARTITION call takes $\Theta(n)$ time, so the best-case time complexity for QUICKSELECT is $\Theta(n)$.

(d) The worst-case for QUICKSELECT is when each call to PARTITION only rules out one element, and thus we have to PARTITION on n elements, $n-1$ elements, $n-2$ elements and so on.

An example of this is finding $k = 0$ when the input is reverse sorted order.

The time taken for each partition is linear, so the total time for QUICKSELECT in the worst case is,

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

(e) To compute the expected time-complexity of this algorithm we will make the assumption that the order of the array is uniformly random, and that after each PARTITION, the pivot ends up in the middle of the array, *i.e.*, $p = \frac{n}{2}$.

In this case, the recurrence relation for QUICKSELECT will be:

$$T(n) = \Theta(n) + T\left(\frac{n}{2}\right),\ T(1) = 1$$

We see that this first the format required by the Master theorem with $a = 1, b = 2, c = 1$. Since $log_b(1) = 0 < 1 = c$ we have that $T(n) \in \Theta(n^c) = \Theta(n)$.

Thus QUICKSELECT has an expected time-complexity of $\Theta(n)$.

It turns out that even if the array is split into very unevenly sized components after partitioning (*e.g.,* $0.99n$ and $0.01n$ length sub-arrays) and the kth smallest element *always ends up in the larger array* then the runtime of QUICKSELECT is still $\Theta(n)$. Can you use the Master Theorem to show why this is the case?

(f) Well the worst case for QUICKSELECT is much worse than the worst case for the heap based algorithm ($\Theta(n^2)$ vs. $\Theta(n \log n)$).

Also, if we make the assumption that $k \ll n$ (*i.e.*, k is very small compared to n) then we can treat $\Theta(n + k \log n)$ as $\Theta(n)$ meaning the heap-based approach is comparable to the best case for QUICKSELECT.

In general for unknown k we can expect QUICKSORT to be faster in the expected case.

COMP20007 Design of Algorithms

Week 10 Workshop

Tutorial

1. Separate chaining Consider a hash table in which the elements inserted into each slot are stored in a linked list. The table has a fixed number of slots $L = 2$. The hash function to be used is $h(k) = k \mod L$.

Show the hash table after insertion of records with the keys

17 6 11 21 12 33 5 23 1 8 9

Can you think of a better data structure to use for storing the records in each slot?

2. Open addressing Consider a hash table in which each slot can hold one record and additional records are stored elsewhere in the table using linear probing with steps of size $i = 1$. The table has a fixed number of slots $L = 8$ The hash function to be used is $h(k) = k \mod L$.

Show the hash table after insertion of records with the keys

17 7 11 33 12 18 9

Repeat using linear probing with steps of size $i = 2$. What problem arises, and what constraints can we place on i and L to prevent it?

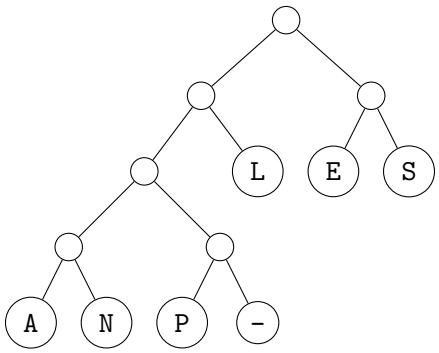
Can you think of a better way to find somewhere else in the table to store overflows?

3. Huffman code generation Huffman's Algorithm generates prefix-free code trees for a given set of symbol frequencies. Using these algorithms generate two code trees based on the frequencies in the following message:

`losslesscodes`

What is the total length of the compressed message using the Huffman code?

4. Canonical Huffman decoding The following code tree was generated using Huffman's algorithm, and converted into a Canonical Huffman code tree. Note: `_` denotes space.



Assign codewords to the symbols in the tree, such that left branches are denoted `0` and right branches are denoted `1`.

Use the resulting code to decompress the following message:

`00100110000011100011011011110011110110100010011011110001101111`

5. Asymptotic Complexity Classes (Revision) For each pair of the following functions, indicate whether $f(n) \in \Omega(g(n))$, $f(n) \in O(g(n))$ or both (in which case $f(n) \in \Theta(g(n))$).

(a) $f(n) = (n^3 + 1)^6$ and $g(n) = (n^6 + 1)^3$,

(b) $f(n) = 3^{3n}$ and $g(n) = 3^{2n}$,

(c) $f(n) = \sqrt{n}$ and $g(n) = 10n^{0.4}$,

(d) $f(n) = 2\log_2\{(n + 50)^5\}$ and $g(n) = (\log_e(n))^3$,

(e) $f(n) = (n^2 + 3)!$ and $g(n) = (2n + 3)!$,

(f) $f(n) = \sqrt{n^5}$ and $g(n) = n^3 + 20n^2$.

The following result may be useful,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & \text{implies that } f(n) \text{ has a smaller order of growth than } g(n), \\ c & \text{implies that } f(n) \text{ has the same rder of growth than } g(n), \\ \infty & \text{implies that } f(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

Computer Lab

These exercises center around building hash functions for strings. Download `main.c`, `strhash.c`, `strhash.h`, and `words.txt` (2.5MB) from the LMS. Compile these files using the command `gcc -o hash strhash.c main.c` (or create a Makefile).

The resulting program `hash` reads strings from standard input and counts how many strings hash to each bucket in a hash table. The size of the table, and the hash function to use, are specified through the command line. The task for this lab is to use this program to explore the distributions produced by different hash functions.

1. Horrible hash functions At the moment, `strhash.c` provides only two hash functions: 'hash everything to bucket 0' and 'hash everything to a random bucket'. The first is terrible because it only uses one bucket. The second evenly distributes keys among all buckets, but it isn't even really a hash function (why?).

Run `./hash 14 0` and `./hash 14 r`. In each case, enter your favourite 10-20 words, one per line, then press control+D (MinGW: control+Z, enter). Run the programs again, this time taking input from `words.txt` (using `<`).

2. Bad hash functions Implement two new hash functions in `strhash.c`. For the first, hash a string to the ASCII value of its first character, mod table size `size`. For the second, hash a string to its length, mod table size `size`.

Modify the `hash()` and `name()` functions at the bottom of `strhash.c` to allow these functions to be used by `main.c`. Recompile the program and experiment with these hash functions using different table sizes.

Note: for large table sizes, you might want to pipe the output into `less` (e.g by running a command like `./hash 100 r < words.txt | less`) to make it easier to view (exit `less` by pressing `q`).

(Optional) implement a third bad hash function which treats the bits in the first few characters of the input string as as an unsigned integer, and returns that integer modulo `size`. Be careful of the case where the string does not have enough characters to form an integer.

3. Universal hash function Implement the following universal hash function for strings:

$$h(s, m) = \left(\sum_{i=0}^{len(s)-1} r[i] \times s[i] \right) \mod m$$

where r is a fixed array of random integers (hint: use a `static` flag to initialise this array only the first time you call the function). You may assume that 128 is the maximum possible value for $len(s)$.

Add your universal hash function to `hash()` and `name()`, and compare the bucket distributions it achieves with those of the other hash functions.

COMP20007 Design of Algorithms

Week 10 Workshop Solutions

Tutorial

1. Separate chaining Here's the hash table after inserting the keys according to the hash function $h(k) = k \mod L$ with $L = 2$:

0	$6 \to 12$
1	$17 \to 11 \to 21 \to 33 \to 5 \to 23 \to 1$

In terms of better data structures over a standard linked list, there are plenty of options to try.

- A move-to-front (MTF) list could adapt to access patterns to provide better average performance
- An array (possibly also with MTF) could save on space for all those `next` pointers, and could and also yield better cache performance*.
- A balanced search tree could ensure $O(\log n)$ lookups in the worse case even if the hash function distributes keys unevenly.
  However, before reaching for more complicated data structures as a cure for poor hashing performance, it might be better to try increasing the table size and/or improving the hash function.

*Cache performance is related to how nicely your algorithm behaves in its memory access patterns. It's a matter of practical concern as it has a real impact on algorithm performance. With a MTF linked list, memory accesses might be distant from one another as we follow pointers to wildly different parts of memory. In contrast, the elements of an array are *contiguous*, so subsequent accesses are likely to be in nearby areas of memory.
Cache performance is not really a concern in this subject, but it's definitely something to be aware of.

2. Open addressing Here's the hash table after inserting the keys according to the hash function $h(k) = k \mod L$ with $L = 8$:

0	1	2	3	4	5	6	7
	17	33	11	12	18	9	7

If we repeat using $i = 2$, we can insert the first 6 keys without much trouble, but then we can't find a place for 9:

0	1	2	3	4	5	6	7
	17	18	11	12	33		7
	9?		9?		9?		9?

The problem is that $i = 2$ and $L = 8$ have a common factor other than 1 (it's 2), in maths terms, they are not *coprime*. So, it's possible for the key 9 to fall into a loop of steps that doesn't actually encounter every cell. As a result, we can't insert 9, even though there are still empty buckets in the hash table.

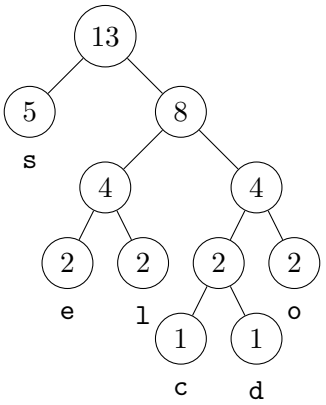
We can avoid this problem if we make sure i and L have no common factors other than 1. Appropriate choices include: L a power of 2 and i odd, or L prime and $i < L$.

In terms of better open addressing strategies, one major improvement comes from the idea of double hashing (choosing a different i for each element). Double hashing can help to eliminate clustering. However, based on the discussion above, we must place certain constraints on the output of the second hash function to ensure that it's coprime with L. Also, it should always be greater than 0 (why?).

3. Huffman code generation Frequency counts:

s	l	o	e	c	d
5	2	2	2	1	1

A possible Huffman tree:



Using 0 for left branches and 1 for right branches we get the following codewords:

s	0
l	101
o	111
e	100
c	1100
d	1101

Hence, the encoded version of `losslesscodes` is:

101 111 0 0 101 100 0 0 1100 111 1101 100 0

Other trees are also possible depending on how you break ties while constructing the tree. All trees give a total message length of 31 bits (sum of codeword lengths multiplied by frequencies).

4. Canonical Huffman decoding The code is:

symbol	codeword	length
A	0000	4
N	0001	4
P	0010	4
_	0011	4
L	01	2
E	10	2
S	11	2

The message is:

`PLEASE LESS SLEEPLESSNESS`

5. Asymptotic Complexity Classes (Revision) For each pair of the following functions, indicate whether $f(n) \in \Omega(g(n))$, $f(n) \in O(g(n))$ or both (in which case $f(n) \in \Theta(g(n))$).

(a) Using the binomial theorem or otherwise we get,

$$f(n) = n^{3\times 6} + \binom{6}{1} n^{3\times 5} + \binom{6}{2} n^{3\times 4} + \binom{6}{3} n^{3\times 3} + \binom{6}{4} n^{3\times 2} + \binom{6}{5} n^{3\times 1} + 1,$$

and

$$g(n) = n^{6\times 3} + 3n^{6\times 2} + 3n^{6\times 1} + 1$$

so $f(n) \in \Theta(n)$ as the heighest growing terms are n^{18}.

(b)

$$f(n) = 3^{3n} = 27^n \quad \text{and} \quad g(n) = 3^{2n} = 9^n$$

So,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{27^n}{9^n} = \lim_{n\to\infty} \left(\frac{27}{9}\right)^n = \infty$$

As such $f(n) = \Omega(g(n))$.

(c) $f(n) \in \Theta(n^{0.5})$ and $g(n) = 10n^{0.4} \in \Theta(n^{0.4})$. $f(n)$ grows faster. So $f(n) \in \Omega(g(n))$.

(d)

$$f(n) = 2\log_2\{(n+50)^5\} = 10\log_2(n+50)$$

and

$$g(n) = (\log_e(n))^3 = \frac{(\log_2(n))^3}{\text{const}}$$

Now,

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{10\log_2(n+50)}{\frac{(\log_2(n))^3}{\text{const}}} \\
&= \lim_{n\to\infty} \left(\text{const} \times \frac{\log_2(n+50)}{\log_2(n)} \times \frac{1}{(\log_2(n))^2}\right) \\
&= \text{const} \times \left(\lim_{n\to\infty} \frac{\log_2(n+50)}{\log_2(n)}\right) \times \left(\lim_{n\to\infty} \frac{1}{(\log_2(n))^2}\right) \\
&= \text{const} \times 1 \times 0 = 0
\end{aligned}$$

So $f(n) \in O(g(n))$.

To see why $\frac{\log_2(n+50)}{\log_2(n)} \to 1$ we apply l'Hopital's rule. Since $\log_2(n+50) \to \infty$ and $\log_2(n) \to \infty$ as $n \to \infty$ we can claim that,

$$\begin{aligned}
\lim_{n\to\infty} \frac{\log_2(n+50)}{\log_2(n)} &= \lim_{n\to\infty} \frac{\frac{1}{\ln 2}}{\frac{1}{\ln 2}} \times \frac{\ln(n+50)}{\ln(n)} = \lim_{n\to\infty} \frac{\ln(n+50)}{\ln(n)} = \lim_{n\to\infty} \frac{\frac{d}{dn}(\ln(n+50))}{\frac{d}{dn}(\ln(n))} \\
&= \lim_{n\to\infty} \frac{\frac{1}{n+50}}{\frac{1}{n}} = \lim_{n\to\infty} \frac{n}{n+50} = \lim_{n\to\infty} \frac{1}{1+\frac{50}{n}} = 1.
\end{aligned}$$

(e)

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{(n^2+3)!}{(2n+3)!} \\
&= \lim_{n\to\infty} \frac{(n^2+3)(n^3+2)\cdots(2n+4)(2n+3)(2n+2)\cdots 1}{(2n+3)(2n+2)\cdots 1} \\
&= \lim_{n\to\infty} (n^2+3)(n^3+2)\cdots(2n+4) \\
&= \infty
\end{aligned}$$

So $f(n) \in \Omega(g(n))$.

(f) $f(n) = n^{2.5}$ and $g(n) = n^3 + 20n^2$. Since n^3 grows faster than $n^{2.5}$ we have $f(n) \in O(g(n))$.

COMP20007 Design of Algorithms

Week 11 Workshop

Tutorial

1. Counting Sort Use counting sort to sort the following array of characters:

`[a, b, a, a, c, d, a, a, f, c, b]`

How much space is required if the array has n characters and our alphabet has k possible letters?

2. Radix Sort Use radix sort to sort the following strings:

`abc bab cba ccc bbb aac abb bac bcc cab aba`

As a reminder radix sort works on strings of length k by doing k passes of some other (stable) sorting algorithm, each pass sorting by the next most significant element in the string. For example in this case you would first sort by the 3rd character, then the 2nd character and then the 1st character.

3. Stable Counting Sort Which property is required to use counting sort to sort an array of tuples by only the first element, leaving the original order for tuples with the same first element. For example the input may be:

$(8, \texttt{campbell}),\ (6, \texttt{tal}),\ (3, \texttt{keir}),\ \ldots (6, \texttt{gus}),\ (0, \texttt{nick}),\ (8, \texttt{tom})$

Discuss how you would ensure that counting sort satisfies this property. Can you achieve this using only arrays? How about using auxiliarry linked data structures?

4. Horspool's Algorithm Use Horspool's algorithm to search for the pattern GORE in the string ALGORITHM.

5. Horspool's Algorithm Continued How many character comparisons will be made by Horspool's algorithm in searching for each of the following patterns it the binary text of one million zeros?

(a) 01001

(b) 00010

(c) 01111

6. Horspool's Worst-Case Time Complextity Using Horspool's method to search in a text of length n for a pattern of length m, what does a worst-case example look like?

7. (Revision) Recurrence Relations Solve the following recurrence relations. Give both a closed form expression in terms of n and a Big-Theta bound.

(a) $T(1) = 1$, $T(n) = T(n/2) + 1$.

(b) $T(0) = 0$, $T(n) = T(n-1) + n/5$.

8. (Optional) Karp-Rabin Hashing In this question we will use Karp-Rabin hashing to solve a string search problem.

For an alphabet with a characters, and some positive number m (we usually select m to be prime, why?) the Karp-Rabin hash function for a string of n characters $S =$"$s_0 s_1 \dots s_{n-1}$" is given by:

$$h(S) = \sum_{i=0}^{n-1} a^{n-1-i} \cdot \text{chr}(s_i) \mod m.$$

Here $\text{chr}(s)$ gives the index of the character s in the alphabet, *i.e.*, $\text{chr}(s) \in \{0, 1, \dots, a-1\}$.

Consider the following example. We have the alphabet { A, B, C, D }, and as such $a = 4$. Let $m = 11$.

We are going to search for the string $P =$"CAB" in the string $T =$"CADACAB".

Since $|P| = 3$ you'll have to compute the hash for each substring of 3 characters in T. To start you off, consider the first subtrings $T[0 \dots 2] =$"CAD",

$$\begin{aligned} h(\text{“CAD”}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(D)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 3 \mod 11 \\ &= 35 \mod 11 \\ &= 2 \end{aligned}$$

Now we can, in constant time, compute the successive three characters, by using the following formula:

$$h(s_1 s_2 \dots s_k) = a\left(h(s_0 s_2 \dots s_{k-1}) - a^{k-1} \cdot \text{chr}(s_0)\right) + \text{chr}(s_k) \mod m$$

So, we can compute $h($"ADA"$)$ like so:

$$\begin{aligned} h(\text{“ADA”}) &= 4\left(h(\text{“CAD”}) - 4^2 \cdot 2\right) + 0 \mod m \\ &= 4(2 - 32) + 0 \mod 11 \\ &= 4(-30) + 0 \mod 11 \\ &= 4(-30 \mod 11) \mod 11 \\ &= 4(3) \mod 11 \\ &= 12 \mod 11 \\ &= 1 \end{aligned}$$

Complete the string search by computing all of the required hashes (including $h(P)$) and determining whether or not $h(P)$ matches any of the hash values for the substrings of T.

How does this enable us to search for P in T in $O(|T| + |P|)$ time? What might go wrong?

COMP20007 Design of Algorithms

Week 11 Workshop Solutions

Tutorial

1. Counting Sort To perform counting sort we note that the range of possible input characters are a, b, c, d, e, and f. We then loop through the array and tally the frequencies of each character:

```
  a b c d e f
[ 5 2 2 1 0 1 ]
```

We then read these frequencies in order to reconstruct a sorted array. That is, we take 5 `a`s, 2 `b`s *etc.*

```
[a a a a a b b c c d f]
```

2. Radix Sort Start by sorting by the final letter, keeping strings with the same final letter in the original relative order:

```
  abc bab cba ccc bbb aac abb bac bcc cab aba
cba aba | bab bbb abb cab | abc ccc aac bac bcc
```

We then join all of these together and sort by the middle letter:

```
  cba aba bab bbb abb cab abc ccc aac bac bcc
bab cab aac bac | cba aba bbb abb abc | ccc bcc
```

Finally by the first letter:

```
  bab cab aac bac cba aba bbb abb abc ccc bcc
aac aba abb abc | bab bac bbb bcc | cab cba ccc
```

So the final sorted array is

```
aac aba abb abc bab bac bbb bcc cab cba ccc
```

3. Stable Counting Sort To sort tuples by the first element and maintian relative order between tuples with equal first values we must ensure that *counting sort is stable.*

We can use the cumulative coints array based approach introduced in lectures to do this.

Alternatively, rather than storing frequencies in a frequency array, we have an array of linked lists with each object appended to the linked list corresponding to the value we're sorting by. The original array can be reconstructed by sequentially removing the head of the linked list and inserting into the new reconstructed array.

Both methods will require $\Theta(n + k)$ additional space, where k is the number of distinct values we're counting and n is the size of the input array.

4. Horspool's Algorithm For that pattern we calculate the shifts: $S[G] = 3, S[O] = 2, S[R] = 1, S[x] = 4$ for all other letters x. So the first shift (when the string's O fails to match E) is 2 positions, bringing E under the string's I. The next shift is 4, which will take us beyond the end of the string, so the algorithm halts (after just two comparisons), reporting failure.

5. Horspool's Algorithm Continued

(a) The pattern's last 1 will be compared against every single 0 in the text (except of course the first four), since the skip will be 1. So 999,996 comparisons.

(b) Here we will will make two comparisons between shifts, and each shift is of length 2. So the answer is again 999,996 comparisons.

(c) For the last pattern, the skip is 4. So we will make 249,999 comparisons.

6. Horspool's Worst-Case Time Complexity Consider the case where the text contains n zeros, and the pattern contains a 1 in the first position, followed by $m-1$ zeros. In this case each skip will be just a single position, and between skips, m comparisons will be made.

Skips will occur until the pattern is starting at index $n-m$. Since we start at index 0 and end at index $n-m$ we do m comparisons $n-m+1$ times, giving $m(n-m+1) \in O(nm)$ comparisons.

We can see that if our pattern contains a 1 followed by $n/2$ zeros this time complexity becomes $O(n^2)$.

7. (Revision) Recurrence Relations

(a) $T(1) = 1$
$T(n) = T(n/2) + 1$

$$\begin{aligned} T(n) &= T(n/2) + 1 \\ &= (T(n/4) + 1) + 1 \\ &= (T(n/8) + 1) + 1 + 1 \\ &\vdots \\ &= T(n/2^k) + k \\ &\vdots && \text{let } k = \log_2(n) \\ &= T(n/2^{\log_2(n)}) + \log_2(n) \\ &= T(n/n) + \log_2(n) \\ &= T(1) + \log_2(n) \\ &= 1 + \log_2(n) \end{aligned}$$

So $T(n) = 1 = \log_2(n) \in \Theta(\log n)$.

(b) $T(0) = 0$
$T(n) = T(n-1) + \frac{n}{5}$

$$\begin{aligned} T(n) &= T(n-1) + \frac{n}{5} \\ &= T(n-2) + \frac{n-1}{5} + \frac{n}{5} \\ &= T(n-3) + \frac{n-2}{5} + \frac{n-1}{5} + \frac{n}{5} \\ &\vdots \\ &= T(n-k) + \frac{n-(k-1)}{5} + \cdots + \frac{n-1}{5} + \frac{n}{5} \\ &\vdots && \text{let } k = n \\ &= T(n-n) + \frac{n-(n-1)}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\ &= 0 + \frac{1}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\ &= \frac{1}{5}\left(1 + \cdots + (n-1) + n\right) \\ &= \frac{n(n+1)}{10} \end{aligned}$$

So $T(n) = n(n+1)/10 \in \Theta(n^2)$.

8. (Optional) Karp-Rabin Hashing First we must compute the hash of the pattern $P =$"CAB":

$$\begin{aligned} h(\text{“CAB”}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(B)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 1 \mod 11 \\ &= 33 \mod 11 \\ &= 0 \end{aligned}$$

As discussed in the question we can compute $h(\text{“CAD”})$ like so:

$$\begin{aligned} h(\text{“CAD”}) &= a^2 \cdot \text{chr(C)} + a \cdot \text{chr(A)} + \text{chr(D)} \mod m \\ &= 4^2 \cdot 2 + 4 \cdot 0 + 3 \mod 11 \\ &= 35 \mod 11 \\ &= 2 \end{aligned}$$

Then we can compute each successive substring of 3 characters like so:

$$\begin{aligned} h(\text{“ADA”}) &= 4\left(h(\text{“CAD”}) - 4^2 \cdot 2\right) + 0 \mod m \\ &= 4(2 - 32) + 0 \mod 11 \\ &= 4(-30) + 0 \mod 11 \\ &= 4(-30 \mod 11) \mod 11 \\ &= 4(3) \mod 11 \\ &= 12 \mod 11 \\ &= 1 \end{aligned}$$

From $h(\text{“ADA”})$ we can compute $h(\text{“DAC”})$ like so:

$$\begin{aligned} h(\text{“DAC”}) &= 4\left(h(\text{“ADA”}) - 4^2 \cdot 0\right) + 2 \mod m \\ &= 4(1) + 2 \mod 11 \\ &= 6 \mod 11 \\ &= 6 \end{aligned}$$

Completing this for the rest of the substrings with 3 characters in T we get:

$$\begin{aligned} h(T[0 \ldots 2]) &= h(\text{“CAD”}) = 2 \\ h(T[1 \ldots 3]) &= h(\text{“ADA”}) = 1 \\ h(T[2 \ldots 4]) &= h(\text{“DAC”}) = 6 \\ h(T[3 \ldots 5]) &= h(\text{“ACA”}) = 8 \\ h(T[4 \ldots 6]) &= h(\text{“CAB”}) = 0 \end{aligned}$$

Notice that $h(T[4 \ldots 6]) = 0$ and $h(P) = 0$. Therefore, $T[0 \ldots 2]$ *may* be equal to P.

Why can we not be sure that $T[0 \ldots 2] = P$, as we may get a **collision** with our hash function. The hash function may give the same result for two different substrings, for instance $h(\text{“DBCD”})$ also equals 0. As a result we must check manually that the strings match. If we choose a large m then these collisions become increasingly less likely, and thus the additional computation complexity required by checking “false positives” (*i.e.*, when the hash function values are the same but the strings differ) becomes negligible.

How about the time complexity of this algorithm? Initially computing $h(S)$ for a string S of n characters takes $O(n)$ time. So computing $h(P)$ takes $O(|P|)$ time.

Note that there are $|T| + 1 - |P|$ substrings of length $|P|$ in T. The first of which takes $O(|P|)$ time to hash. However, due to the ability to compute hashes incrementally in $O(1)$ time the remaining $|T| - |P|$ substrings only take $O(1)$ time each. Thus the total time complexity of computing the hashes for all substrings becomes:

$$|P| + (|T| - |P|) = O(|T|).$$

Taking into account the time complexity of computing the hash of P we get a total time complexity of the Karp-Rabin string search algorithm of $O(|T| + |P|)$.

7. (Revision) Recurrence Relations

(a) $T(1) = 1$
$T(n) = T(n/2) + 1$

$$\begin{aligned}
T(n) &= T(n/2) + 1 \\
&= (T(n/4) + 1) + 1 \\
&= (T(n/8) + 1) + 1 + 1 \\
&\vdots \\
&= T(n/2^k) + k \\
&\vdots && \text{let } k = \log_2(n) \\
&= T(n/2^{\log_2(n)}) + \log_2(n) \\
&= T(n/n) + \log_2(n) \\
&= T(1) + \log_2(n) \\
&= 1 + \log_2(n)
\end{aligned}$$

So $T(n) = 1 = \log_2(n) \in \Theta(\log n)$.

(b) $T(0) = 0$
$T(n) = T(n-1) + \frac{n}{5}$

$$\begin{aligned}
T(n) &= T(n-1) + \frac{n}{5} \\
&= T(n-2) + \frac{n-1}{5} + \frac{n}{5} \\
&= T(n-3) + \frac{n-2}{5} + \frac{n-1}{5} + \frac{n}{5} \\
&\vdots \\
&= T(n-k) + \frac{n-(k-1)}{5} + \cdots + \frac{n-1}{5} + \frac{n}{5} \\
&\vdots && \text{let } k = n \\
&= T(n-n) + \frac{n-(n-1)}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\
&= 0 + \frac{1}{5} + \cdots \frac{n-1}{5} + \frac{n}{5} \\
&= \frac{1}{5}(1 + \cdots + (n-1) + n) \\
&= \frac{n(n+1)}{10}
\end{aligned}$$

So $T(n) = n(n+1)/10 \in \Theta(n^2)$.

COMP20007 Design of Algorithms

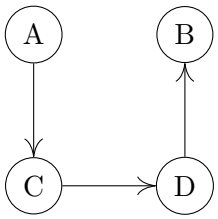
Week 12 Workshop

Tutorial

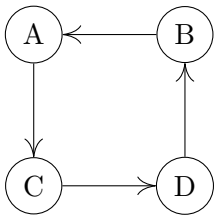
1. Transitive Closure The *transitive closure* of a directed graph G is a new graph H which contains an edge between vertices u and v if and only if there is a path from u to v using one or more edges in the original graph G.

Draw the transitive closure of the following two graphs:

(a)



(b)



2. Warshall's Algorithm Warshall's algorithm is a *dynamic programming* algorithm which computes the transitive closure of a graph defined by an adjacency matrix.

First, we'll assume that the n nodes are labelled $1, 2, \ldots, n$.

We take the problem of *is there a path in G from i to j* (for each pair of nodes $\{i, j\} \subseteq \{1, \ldots, n\}$) and break divide into the sub-problems (for each $k \in \{1, \ldots, n\}$):

> Is there a path from i to j in G using only nodes in $\{1, \ldots, k\}$ as intermediate nodes?

Once we have computed this for $k = n$ we have the answer to the original question for each $\{i, j\}$ and can hence compute the transitive closure.

For each of these sub-problems we will let R^k_{ij} be defined like so:

$$R^k_{ij} := \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \text{ in } G \text{ using only nodes in } \{1, \ldots, k\} \text{ as intermediate nodes} \\ 0 & \text{otherwise} \end{cases}$$

If we have A, the graph G's adjacency matrix we have the following base case and recursive definition of R^k:

$$R^0_{ij} := A_{ij}, \qquad R^k_{ij} := R^{k-1}_{ij} \text{ or } \left(R^{k-1}_{ik} \text{ and } R^{k-1}_{kj}\right)$$

The adjacency matrix of the transitive closure, B, is then defined by $B_{ij} := R^n_{ij}$.

To run Warshall's algorithm, we should compute each of the matrices $R^0, R^1, \ldots, R^n$.

Use Warshall's algorithm to compute the transitive closure of a graph G defined by the following adjacency matrix,

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

3. Floyd's Algorithm So we've seen Warshall's algorithm which computes whether or not there is a path between a pair of vertices $\{i, j\}$ by computing the adjacency matrix of the transitive closure, given the original graph's adjacency matrix.

Floyd's algorithm builds on Warshall's algorithm to solve the *all pairs shortest path* problem. That is, Floyd's algorithm computes the length of the shortest path from each pair of vertices in a graph.

Rather than an adjacency matrix, we will require a weights matrix W, where W_{ij} indicates the weight of the edge from i to j (if there is no edge from i to j then then $W_{ij} = \infty$). We will ultimately find a distance matrix D in which D_{ij} indicates the cost of the shortest path from i to j.

The sub-problems in this case will be answering the following question:

> What's the shortest path from i to j using only nodes in $\{1, \ldots, k\}$ as intermediate nodes?

To perform the algorithm we find D^k for each $k \in \{0, \ldots, n\}$ and set $D := D^n$. The update rule becomes the following:

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

Perform Floyd's algorithm on the graph given by the following weights matrix:

$$W = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}.$$

4. Baked Beans Bundles We have bought n cans of baked beans wholesale and are planning to sell bundles of cans at the University of Melbourne's farmers' market.

Our business-savvy friends have done some market research and found out how much students are willing to pay for a bundle of k cans of baked beans, for each $k \in \{1, \ldots, n\}$.

We are tasked with writing a *dynamic programming algorithm* to determine how we should split up our n cans into bundles to maximise the total price we will receive.

(a) Write the pseudocode for such an algorithm.

(b) Using your algorithm determine how to best split up 8 cans of baked beans, if the prices you can sell each bundle for are as follows:

Bundle Size k	1	2	3	4	5	6	7	8
Price	1	5	8	9	10	17	17	20

(c) What's the runtime of your algorithm? What are the space requirements?

5. (Revision) Quicksort & Mergesort

(a) Perform a single *Hoare Partition* on the following array, taking the first element as the pivot.

```
[3, 8, 5, 2, 1, 3, 5, 4, 8]
```

(b) Perform Quicksort on the array from (a). You may use whatever partitioning strategy you like (*i.e.,* you don't need to follow a particular algorithm).

(c) Perform Mergesort on the array from (a).

Computer Lab (Optional)

1. Baked Beans Bundles (Implementation) Write a C program which solves the baked beans problem from the tutorial component of this workshop.

Your program will be given the output as follows:

```
n
p_1
...
p_n
```

Here `p_k` is the price a student is willing to pay for a bundle of k cans of baked beans.

The example from the tutorial would be provided like so:

```
8
1
5
8
9
10
17
17
20
```

Your program should print the maximum price you can sell this collection of cans for, and the sizes of the bundles (in any order). For instance with the above input the output of your program might be:

```
$ ./baked-beans < input.txt
22
2
6
```

This is because the maximum price is 22, which is achieved by selling a bundle of 2, and a bundle of 6.

This problem is often referred to as the *Rod-Cutting Problem*, where you're given an n meter rod and prices for k meters of rod (with $k \in \{1, \ldots, n\}$) and must output the optimal way of splitting the rod to maximise profit.

If you have finished this question you may use the remaining lab time to work on Assignment 2.

COMP20007 Design of Algorithms

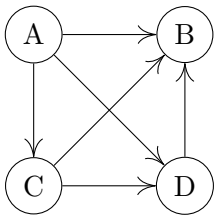
Week 12 Workshop Solutions

Tutorial

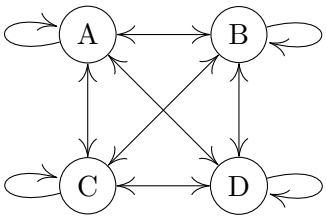
1. Transitive Closure The transitive closures of the graphs have edges between each pair of nodes which have some path connecting them.

We draw a *self-loop* (i.e., an edge from a vertex back to itself) if there is a cycle containing that vertex in the original graph (in other words if there's a path which starts and ends at that vertex).

(a)



(b)



2. Warshall's Algorithm Recall the update rule:

$$R^0_{ij} := A_{ij}, \qquad R^k_{ij} := R^{k-1}_{ij} \text{ or } \left(R^{k-1}_{ik} \text{ and } R^{k-1}_{kj}\right)$$

We'll start by setting R^0 to A, and then follow the update rule for each k from 1 to n (4 in this case).

In practice we run down the columns from left to right, stopping when we meet a 1. This first happens when we are in row 3, column 1. At that point, 'or' row 1 onto row 3 (and so on):

$$R^0 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^1 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^2 = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$R^3 = \begin{bmatrix} \mathbf{1} & 0 & 1 & 1 \\ \mathbf{1} & 0 & 1 & \mathbf{1} \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$B := R^4 = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3. Floyd's Algorithm Again, applying the following update rule for $k \in \{0, \ldots, 4\}$.

$$D^0_{ij} := W_{ij}, \qquad D^k_{ij} := \min\left\{D^{k-1}_{ij}, D^{k-1}_{ik} + D^{k-1}_{kj}\right\}$$

We can think about this as "selecting" the kth row and column at each step, and for all other elements of the matrix D^k we check whether the current distance can be improved by taking the sum of the corresponding elements in the "selected" row and column.

The "selected" rows and columns are shown in red. Updates are shown in bold.

$$D^0 = \begin{bmatrix} 0 & 3 & \infty & 4 \\ \infty & 0 & 5 & \infty \\ 2 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

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

$$D^2 = \begin{bmatrix} 0 & 3 & \mathbf{8} & 4 \\ \infty & 0 & 5 & \infty \\ 2 & 5 & 0 & 6 \\ \infty & \infty & 1 & 0 \end{bmatrix}$$

$$D^3 = \begin{bmatrix} 0 & 3 & 8 & 4 \\ \mathbf{7} & 0 & 5 & \mathbf{11} \\ 2 & 5 & 0 & 6 \\ \mathbf{3} & \mathbf{6} & 1 & 0 \end{bmatrix}$$

$$D := D^4 = \begin{bmatrix} 0 & 3 & \mathbf{5} & 4 \\ 7 & 0 & 5 & 11 \\ 2 & 5 & 0 & 6 \\ 3 & 6 & 1 & 0 \end{bmatrix}$$

4. Baked Beans Bundles

(a) Write the pseudocode for such an algorithm.

The subproblems we will try to solve will be $P[i]$, which will indicate the best price we can get if we have i cans.

Note that if we have 0 cans then we don't get anything, so $P[0] = 0$.

We'll let price_k indicate the price of a bundle with k cans.

The update rule then becomes:

$$P[i] = \max_{k \in \{1,\ldots,i\}} \left\{ \text{price}_k + P[i-k] \right\}$$

To perform our algorithm we compute these subproblems for $i = 1 \ldots n$, and the solution will be the answer to $P[n]$.

If we also want to keep track of the sizes of the bundles we select we should also keep track of which k gave the maximum value for each i. We can denote this mathematically using the argmax function:

$$B[i] = \text{argmax}_{k \in \{1,\ldots,i\}} \left\{ \text{price}_k + P[i-k] \right\}$$

The pseudocode for this algorithm is as follows:

function BAKEDBEANS($prices[1 \ldots n]$)
  $P \leftarrow$ new array of 0s with indices 0 through n
  $B \leftarrow$ new array of 0s with indices 0 through n
  for $i = 1 \ldots n$ **do**
    $max\ price \leftarrow 0$

```
    max k ← 0
    for k = 1 . . . i do
        if prices[k] + P[i − k] > max price then
            max price ← prices[k] + P[i − k]
            max k ← k
    P[i] ← max price
    B[i] ← max k
// print the maximum price for all n
output P[n]
// print each k used
j ← n
while B[j] > 0 do
    output B[j]
    j ← j − B[j]
```

(b) Running the algorithm above on the example given yields the following arrays:

$$P = \Big[0, 1, 5, 8, 10, 13, 17, 18, 22\Big]$$
$$B = \Big[0, 1, 2, 3, 2, 2, 6, 1, 2\Big]$$

So the maximum price we can get is 22 and we use a bundle of 2 ($B[8]$) and then 6 ($B[8-2] = B[6]$).

(c) The runtime of this algorithm is $O(n^2)$. We can reason about this like so: we're updating n subproblems, and each update takes up to n iterations, so the runtime complexity is $O(n^2)$.

Each subproblem requires $O(1)$ space and we have $n+1$ subproblems, so the space complexity of this algorithm is $O(n)$.

5. (Revision) Quicksort & Mergesort

(a)

```
[3, 8, 5, 2, 1, 3, 5, 4, 8]
 ^
 p

[3, 8, 5, 2, 1, 3, 5, 4, 8]
 i                        j

[3, 8, 5, 2, 1, 3, 5, 4, 8]
    i           j

Swap:
[3, 3, 5, 2, 1, 8, 5, 4, 8]
    i           j

[3, 3, 5, 2, 1, 8, 5, 4, 8]
       i     j

Swap:
[3, 3, 1, 2, 5, 8, 5, 4, 8]
       i     j

[3, 3, 1, 2, 5, 8, 5, 4, 8]
          j  i

Crossed Over => Stop
Swap A[l] with A[j]:
[2, 3, 1, 3, 5, 8, 5, 4, 8]

Done!
```

(b)

```
[3, 8, 5, 2, 1, 3, 5, 4, 8]
 p
[2, 1, 3, 3, 8, 5, 5, 4, 8]
          p
 [2, 1, 3]              [8, 5, 5, 4, 8]
  p                      p
 [1, 2, 3]              [5, 5, 4, 8, 8]
     p                           p

[1]     [3]           [5, 5, 4]      [8]
                       p
                      [5, 4, 5]
                             p

                     [5, 4]
                      p
                        [4]
=> [1, 2, 3, 3, 5, 3, 5, 8, 8]
```

(c)

```
[3,   8,   5,     2,   1,    3,   5,    4,   8]
[3,   8,   5,     2,   1]   [3,   5,    4,   8]
[3,   8,   5]    [2,   1]   [3,   5]   [4,   8]
[3,   8]  [5]    [2,   1]   [3,   5]   [4,   8]
[3]  [8]  [5] |  [2]  [1] | [3]  [5] | [4]  [8]
[3,   8]  [5] |  [2]  [1] | [3]  [5] | [4]  [8]
[3,   5,   8] |  [1,   2] | [3,   5] | [4,   8]
[1,   2,   3,     5,   8] | [3,   4,    5,   8]
[1,   2,   3,     3,   4,    5,   5,    8,   8]
```