2019 Semester 1
COMP20007 Design of Algorithms
School of Computing and Information Systems
The University of Melbourne

Assignment 2

Due: 11:59 PM Wednesday May 29th

Introduction

This assignment consists of 3 programming problems and 5 analysis problems. You will submit your solutions to the programming tasks via `dimefox submit` and the analysis component via *Turnitin*. This is *not* the same submission process as in Assignment 1.

This assignment has a total of 20 marks and will contribute 20% to your final grade for this subject.

The submission for this assignment will have two components. The analysis will again be submitted via *Turnitin*, however the programming component will now be submitted using `dimefox submit`. Please note that this is *different from Assignment 1*. We will **not** be lenient with submissions for which the submission fails, so please ensure you **receive confirmation** that your assignment has been submitted for *both components*.

Programming Problems

For the following programming problems you must write a C program which reads input from `stdin` and outputs the required output to `stdout`.

Your program must solve each test case in 3 (three) seconds on `dimefox` to achieve full marks. Any algorithm with an appropriate runtime complexity should easily meet these test time bounds.

Note that the comments starting with `//` are for illustrative purposes and are not included in the actual input/output.

Problem 1.a. (Max-Heaps) Write a C program which takes as input a sequence of n (possibly negative, not necessarily distinct) integers and constructs a max-heap containing these elements.

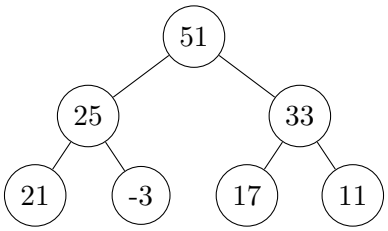
Your program will be provided with the value n (the total number of integers in the input) on the first line. The next n lines will contain the integers, one integer per line. Your program must output the level-order traversal of your heap (*i.e.*, the array representation starting at index 1).

The input heap is a perfect binary tree, *i.e.*, $n = 2^k - 1$.

For example if `input.txt` contains:

```
7
25
21
-3
33
17
11
51
```

One example of a max-heap containing these elements is:



For this heap the output is:

```
$ p1 a < input.txt
51
25
33
21
-3
17
11
```

Note: the solution is not unique, any max-heap will do!

The input may contain duplicate integers, and you must insert all occurrences of each element into the heap.

For full marks your algorithm for this problem must have a time-complexity of $O(n)$.

Problem 1.b. (Right-Handed Max-Heaps) Create a modified version of your program in Problem 1.a. which constructs a *right-handed max-heap* out of the provided input.

A *right-handed max-heap* is a max-heap which satisfies the additional property that for each non-leaf node the right child has a value greater than or equal to the value of the left child.

Your output must be in the same format as in Problem 1.a. and for full marks must have a time complexity of $O(n)$.

Hint: consider starting from a heap.

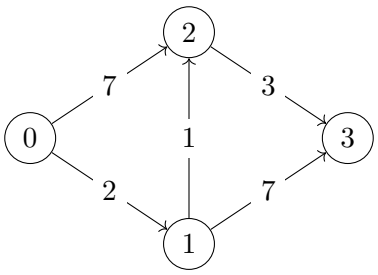
Problem 2.a. (Shortest DAG Paths) In this task you will write a *dynamic programming* solution which computes the shortest path from node v_0 (a source) to node v_{n-1} (a sink) in a weighted directed acyclic graph (DAG).

The nodes in the input graph have labels $\{v_0, \ldots, v_{n-1}\}$. This graph is already topologically ordered, *i.e.*, if the graph contains an edge (v_i, v_j) then j is strictly greater than i.

The first line will contain the number of nodes in the graph n. The n adjacency lists will follow, *i.e.*, for each $i \in \{0, \ldots, n-1\}$ there will be a line giving out-degree(v_i), and then out-degree(v_i) many

lines containing pairs (v_j, w_{ij}) which indicate that the DAG contains an edge (v_i, v_j) with weight w_{ij}. Note that all edge weights w_{ij} are positive.

Consider the following graph:



The following input represents this graph:

```
4 // number of nodes
2 // out-degree of v_0
2 7 // indicates that the edge (v_0, v_2) is present with weight 7
1 2
2
2 1
3 7
1
3 3
0
```

Your task is to compute the cost of the shortest path from 0 to $n-1$, and output what that path is. For this example, the shortest path is v_0, v_1, v_2, v_3 with a cost of 6.

Your program must output the cost of the path, and the number of edges in the path, followed by these vertices (starting with 0 and ending with $n-1$). The output is then:

```
$ p2 a < input.txt
6 // path cost
3 // number of edges in the path
0
1
2
3
```

If there is no path from v_0 to v_{n-1} in the DAG then the only output your program must produce is `"No Path"`.

It is possible to solve this problem in $O(n+m)$ time (for a DAG with n vertices and m edges), although any implementation which solves the test cases within the time limit will be accepted.

Problem 2.b. (Shortest DAG k-Paths) Write a program which finds the shortest path which *uses no more than k edges* from v_0 to v_{n-1} in an input DAG given in topological order.

The input will be in the same format as for Problem 2.a., except that k is provided on the first line as well. So for the example DAG from the previous part, with $k = 2$ the input will be:

```
4 2 // n and k respectively
2
2 7
1 2
2
2 1
3 7
1
3 3
0
```

Your program must output the solution in the same format as before. For this example the shortest path using no more than $k = 2$ edges is v_0, v_1, v_3 which uses 2 edges for a cost of 9. The output is:

```
$ p2 b < input.txt
9 // path cost
2 // number of edges in the path
0
1
3
```

If there does not exist a path from v_0 to v_{n-1} in the DAG using k or fewer edges then the only output your program must produce is `"No Path"`.

It is possible to solve this problem in $O(k(n + m))$ time, although any implementation which solves the test cases within the time limit will be accepted.

Problem 3. (Binary Search Trees) First, some definitions:

Definition 3.1. (Height) The *height* of a node is the number of levels below it (including the level it is on). More precisely, the *height* of a node x in a tree is one more than the number of edges between it and the deepest leaf in the subtree rooted at x. The *height* of a tree is the *height* of its root.

Definition 3.2. (Balanced Binary Tree) A binary tree is *balanced* if every node x, with (possibly null) children ℓ and r, satisfies:

$$-1 \leq \text{height}(\ell) - \text{height}(r) \leq 1$$

Note that we consider height(null) to be 0.

Write a C program which does the following:

- Reads in an array of n positive integers.
- Inserts these n integers into a binary search tree, in the order in which they are provided.
- Output the number of nodes in the binary search tree.
- Output the height of the binary search tree (see *Definition 3.1.*).
- Balance the binary search tree (see *Definition 3.2.*). Output the elements of this balanced binary search tree.

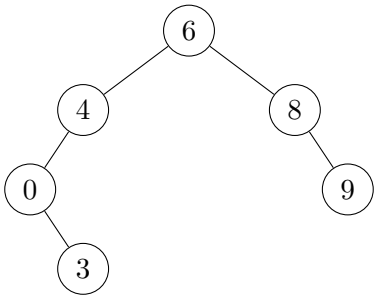
Note that elements in the binary search tree must be distinct, however the input array may contain duplicates. For this reason the number of nodes in the tree and the size of the input may not be the same.

Your program will be provided with n on the first line. The next n lines contain the n integers which make up the input array.

For example, given the following input:

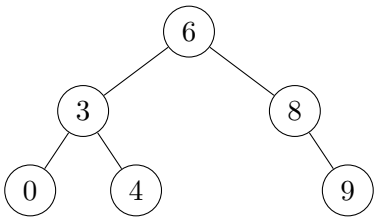
```
10 // this is the size of the input array, n
6
8
9
4
4
6
0
3
6
9
```

The resulting binary search tree is:



The number of nodes in the binary search tree is 6, and the height is 4.

The following is a balanced binary search tree containing these elements:



The first line of your output must be the number of nodes in the tree. The second line must be the height of the binary search tree (before re-balancing).

You must output the balanced binary search tree in a format similar to how we store a heap, *i.e.*, the first line contains the root (index 1) and the node on line i has children at lines $2i$ and $2i + 1$. Note that your tree might have nodes which do not have a left and/or right child, for these lines in the output (where the child is null) output -1.

Before printing your balanced binary search tree you must print the number of lines used to represent the balanced binary search tree. There must not include any trailing -1s.

The output for the example given is:

```
$ p3 < input.txt
6 // the number of nodes
4 // the height of the original binary search tree
7 // the number of lines required to represent the balanced binary search tree
6
3
8
0
4
-1
9
```

Note: implementing a self-balancing tree such as an AVL tree is **not required** to solve this problem.

Analysis Problems

Problem 4. (Average-case complexity of Quicksort) In the lectures, we have seen proof for the average-case complexity of Quicksort (see the lecture capture video for Lecture 14, time stamp 7:35). Your task is to provide proof using a different argument, following the logic below.

There are two components in the partition algorithm of Quicksort: 1) select the pivot and 2) partition the array based on the pivot. Consider an oracle algorithm[1] that can, in constant time, choose a pivot that is guaranteed to lie in between the top 25% and top 75% of the elements in the array.

(a) Prove that using the oracle as a pivot selection strategy would ensure that Quicksort does not perform worse than $O(n \cdot log(n))$.

(b) What is the probability that a randomly selected element of the array falls in between the top 25% and top 75%?

(c) Prove that the average case complexity of Quicksort is $O(n \cdot log(n))$ by combining the reasoning from the previous two points.

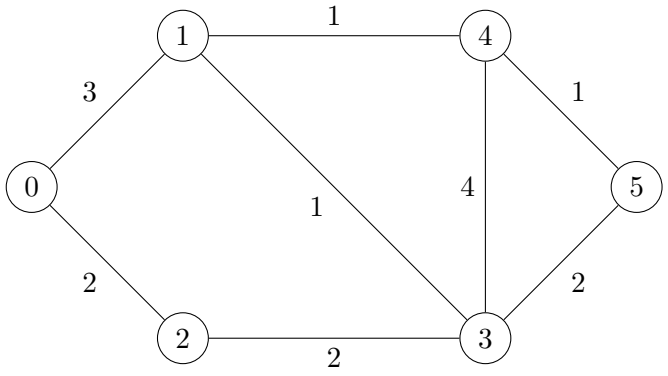
You may find the following statement helpful in your proof: if a trial's probability of success is p, then on average it requires n/p trials to achieve n successes.

Problem 5. (Lexicographical Optimisation with Paths) Provide pseudocode and an explanation for an algorithm that computes a path between two nodes in an undirected graph such that:

- The maximum weight in the path is minimised, *i.e.*, there does not exist another path with a smaller maximum weight.
- Amongst all such paths, it finds the path with minimum cost.
- The time complexity is no worse than $O((|E|^2 + |E| \cdot |V|) \cdot log(|V|))$, where E is the set of edges and V is the set of nodes.

[1] In mathematical arguments an *oracle* refers to some black box mechanism which provides some value. In this case the value is the pivot. It usually corresponds to a convenient assumption for the sake of the proof, in this case the assumption is that we can always select a pivot in the middle half of values in the array.

For example, consider computing the optimum path between nodes v_0 and v_5 for the following graph:



The optimum solution according to the above criteria is the path $p* = (v_0, v_2, v_3, v_5)$. The path $p*$ is considered more favourable than the path $p' = (v_0, v_1, v_4, v_5)$ as the minimum maximum weight of $p*$ is 2 compared to 3 for p', even though $cost(p*) = 6 > 5 = cost(p')$.

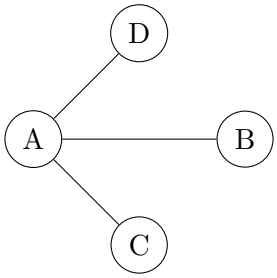
Your algorithm should take the following input/produce the following output:

MinMaxWeightPath **Input:** An weighted undirected graph G, a start node v_s, an end node v_e
Output: The path between v_s and v_e which satisfies the above conditions, and the total cost.

Problem 6. (Weighted Graph Reduction) Your friend has written an algorithm which solves the all pairs shortest path problem for *unweighted undirected graphs*. The *cost* of a path in this setting is the number of edges in the path. The algorithm UnweightedAPSP takes the following input and output:

UnweightedAPSP **Input:** An unweighted undirected graph G
Output: The costs of the shortest paths between each pair of vertices $\{u, v\}$

For example, consider the following graph G.



The output of UnweightedAPSP would be:

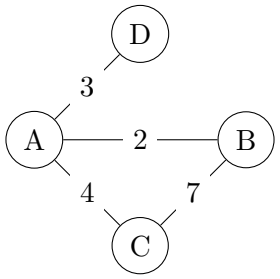
$$\{A, B\} : 1, \{A, C\} : 1, \{A, D\} : 1, \{B, C\} : 2, \{B, D\} : 2, \{C, D\} : 2$$

Your task is two use your friend's algorithm UnweightedAPSP to solve the all pairs shortest paths problem for weighted undirected graphs.

Write an algorithm WeightedAPSP in pseudocode which makes use of the algorithm UnweightedAPSP to solve the problem. Your algorithm must transform its input into an unweighted graph, call UnweightedAPSP, and then interpret the output to correctly compute the costs of the shortest paths for the original weighted graph. The input and output specification of your algorithm must be:

WeightedAPSP **Input:** A weighted undirected graph G
Output: The costs of the shortest paths between each pair of vertices $\{u, v\}$

For example, consider the following graph G.



The output of your algorithm UnweightedAPSP must be:

$$\{A, B\} : 2, \{A, C\} : 4, \{A, D\} : 3, \{B, C\} : 6, \{B, D\} : 5, \{C, D\} : 7$$

This type of problem is called a *reduction*: we have *reduced* the problem WeightedAPSP to the problem UnweightedAPSP. Reductions show up frequently in computer science, and is of particular importance to complexity theory which is concerned with the "hardness" of different problems. Usually we are concerned with what we refer to as a "polynomial-time reduction", which this problem is not, more on this in Week 12!

Problem 7.a. (Heap Algorithm Analysis) In *Problem 1.a.* you are asked to construct a max-heap from an input array of n integers in $O(n)$ time.

Write pseudocode for the algorithm you implemented to construct the max-heap. The input/output specification is:

ConstructHeap **Input:** An array A of n integers
Output: An array H representing a max-heap

Prove that your algorithm runs in $O(n)$ time.

The algorithm you discuss must be the same as the algorithm you used in *Problem 1.a.*

Problem 7.b. (Right-Handed Heap Algorithm Analysis) In *Problem 1.b.* you are asked to construct a right-handed max-heap from an input array of n integers in $O(n)$ time.

Write pseudocode for the algorithm you implemented to construct the right-handed max heap. The input/output specification is:

ConstructRightHandedHeap **Input:** An array A of n integers
Output: An array H representing a right-handed max-heap

Prove that your algorithm runs in $O(n)$ time.

The algorithm you discuss must be the same as the algorithm you used in *Problem 1.b.*

Problem 8. (Heap Top-k) Prof Dubious has made the following claim, and has provided a proof:

Claim. Let n and k be positive integers such that $2^k - 1 \leq n$. In a max-heap H of n elements, the top $2^k - 1$ elements are in the first k layers of the heap.

Proof. Since H is a max-heap, each node in H must satisfy the heap property, *i.e.*, if H_i is an element of H with at least one child then $H_i \geq \max\{\text{children}(H_i)\}$.

We know that every subtree of the heap H is a heap, as subtrees of complete binary trees are complete binary trees, and the heap property holds.

Therefore since each subtree is a heap, the maximum element in each subtree must be at the root of that subtree.

Clearly, the largest element in H is at the root. Since the left and right subtrees are also heaps the next two largest elements must be at the root of each of these subtrees, *i.e.*, the left and right child of the root.

Hence, the largest $2^2 - 1 = 3$ elements of the heap lie in the first 2 layers. We repeatedly apply this argument to each subtree until we have considered the first k layers. Thus the largest $2^k - 1$ elements in the heap must be in the first k layers. □

Briefly describe what is wrong with Professor Dubious's argument and provide a counter-example to their claim.

Completing The Programming Problems

Your program for each problem will receive input via `stdin` (*i.e.*, command line input) which can be read using `scanf(...)` and must produce output to `stdout`, using `printf(...)` for instance.

You must create three separate executable files for this project, `p1`, `p2` and `p3`. The programs `p1` and `p2` must take in the "part" as a command line option, for example:

```
$ ./p1 a < input.txt
$ ./p1 b < input.txt
$ ./p2 a < input.txt
$ ./p3 < input.txt
```

The files provided have the following directory structure:

```
provided_files/
  Makefile
  p1.c
  p2.c
  p3.c
  tests/
    p1a-in-0.txt
    p1a-out-0.txt
    ...
    p3-in-9.txt
    p3-out-9.txt
  verifier.py
  test_all.sh
```

You must complete `Makefile`, `p1.c`, `p2.c` and `p3.c`, and you are encouraged to use multi-file C modules to better structure your programs.

The `Makefile` contains targets `p1`, `p2` and `p3` which will be compiled with `make <target>`, *e.g.,* `make p1`. You must keep the `Makefile` updated if you use different compilation flags or include additional `.c` and `.h` files.

Testing Your Programs

Each problem has 10 test cases which we will test your submission against (*i.e.*, 50 test cases in total). All of these test cases (and examples of expected outputs for each) are provided to you for you to do your own local testing. There will be no hidden test cases. Note that these problems do not have

unique solutions so your output doesn't need to be exactly the same as the expected output files we have provided.

Your program will be given a time limit of 3 seconds to solve each test case.

We have provided the *verifier* we will use to verify your submissions on `dimefox`. The python program, `verifier.py`, expects the following input:

```
usage: python3 test.py <problem> <input> <expected_output> <actual_output>

  This program takes as input the path to the test case input file, the expected
  output for that test case and the actual output (i.e., your program's output).

  You must specify the problem which is being tested as well.
  The possible problems are: p1a, p1b, p2a, p2b, p3
```

For example, to run our program against the `p1a-0` test case, we might perform the following sequence of commands:

```
$ make p1

$ ./p1 < tests/p1a-in-0.txt > tests/p1a-my-out-0.txt

# Note: the following line is only wrapped so that it fits here
$ python verifier.py p1a tests/p1a-in-0.txt tests/p1a-out-0.txt \
      tests/p1a-my-out.txt
p1a-0: succeeded
```

The expected output here is in a file called `tests/p1a-out-0.txt`, be careful not to override these files when you create your own output.

We have also provided a bash script which will run and verify all of the tests for you, for example:

```
$ make p1
$ make p2
$ make p3
$ sh test_all.sh
p1a-0: succeeded
p1a-1: failed (...)
...
```

This script expects your `Makefile` and all required files to be in the same directory as the `test_all.sh` script, the `verifier.py` script and the `tests/` directory.

Programming Problem Submission

You will submit your programs via `dimefox submit`. Instructions for how to connect to `dimefox` can be found under the Workshops section of the LMS.

You should copy all files required to run and compile your code to `dimefox`, this includes the `Makefile` and all `.c` and `.h` files.

From the directory containing these files you should run `submit` and **list all files required to compile and run your program**. For example, if we have a `heap` module as well as `p1.c`, `p2.c`, `p3.c` and the `Makefile` we would run:

```
submit comp20007 a2 Makefile p1.c p2.c p3.c heap.c heap.h
```

You should then `verify` your submission like so:

```
verify comp20007 a2 > a2-receipt.txt
```

This will create a submission receipt called `a2-receipt.txt` which will tell you **whether or not your code compiled, which test cases your code passed and failed and importantly, the mark your submission has received**.

To view the contents of this submission, you can use the command:

```
less a2-receipt.txt
```

You may submit multiple times.

Note: any attempt to manipulate the submission system and/or hard-code solutions to pass the specific test cases we have provided **will result in a mark of 0 for the whole assignment.**

Completing The Analysis Problems

You will submit your solutions to the analysis problems via *Turnitin*, as you did in Assignment 1.

For problems which ask for pseudocode we expect you to provide the same level of detail as the lectures and workshops do. Pseudocode which lacks sufficient detail or is too detailed (*e.g.*, looks like C code) will be subject to a mark deduction.

Your submission should be a `.pdf` file which contains your solutions to Problems 4–8.

The page limit for this assignment is 8 pages, however you should be able to answer these questions in much fewer than 8 pages. Any submission containing more than 8 pages will be subject to a mark deduction.

Make sure you receive a submission receipt from *Turnitin*. If you are having trouble submitting your analysis try using a different web browser such as Google Chrome.

Mark Allocation

The total number of marks in this assignment is 20. The maximum number of marks per problem are:

Problem 1.a. 2 marks.

Problem 1.b. 1 mark.

Problem 2.a. 3 marks.

Problem 2.b. 2 marks.

Problem 3. 3 marks.

Problem 4. 2 marks.

Problem 5. 2 marks.

Problem 6. 2 marks.

Problem 7.a. 1 mark.

Problem 7.a. 1 mark.

Problem 8. 1 mark.

For Problems 2.a. and 2.b., 50% of the marks are allocated for the correct path cost. To get 100% of these marks your program must produce the correct path cost as well as the correct path.

Similarly, 50% of the marks for Problem 3. are allocated for correctly outputting the height and number of nodes of the binary search tree, to get 100% of the marks your program must also produce the correct balanced binary search tree.

For each programming task there are 10 test cases (all of which are provided, *i.e.*, there are no hidden test cases). To get the full amount of marks for each task you must get all 10 of these test cases correct. You will lose 20% of the available marks for each failed test case, *i.e.*,

$$\text{Marks} = \max\Big\{\text{Maximum for Problem} \times (1 - 0.2 \times \text{Test Cases Failed}), 0\Big\}.$$

Late Policy

A late penalty of 20% per day will be applied to submissions made after the deadline. This applies *per component*, *i.e.*,

$$\begin{aligned}\text{Grade} = &\max\Big\{\text{Programming Grade} - 0.2 \times \text{Days Late} \times 11, 0\Big\} \\ &+ \max\Big\{\text{Analysis Grade} - 0.2 \times \text{Days Late} \times 9, 0\Big\}.\end{aligned}$$

For example, if you are 2 days late with the programming component but only 1 day late with the analysis component your grade for the programming component will be reduced by $0.4 \times 11 = 4.4$ and the grade for the analysis component will be reduced by $0.2 \times 9 = 1.8$.

Academic Honesty

You may make use of code provided as part of this subject's workshops or their solutions (with proper attribution), however you **may not** use code sourced from the Internet or elsewhere. Using code from the Internet is grounds for academic misconduct.

All work is to be done on an individual basis. All submissions will be subject to automated similarity detection. Where academic misconduct is detected, all parties involved will be referred to the School of Engineering for handling under the University Discipline procedures. Please see the Subject Guide and the "Academic Integrity" section of the LMS for more information.