FIT3155 S1/2021: Assignment 2
(Due midnight 11:59pm on Fri 30th April 2021)

[Weight: $20 = 3 + 8 + 3 + 6$ marks.]

Your assignment will be marked on the *performance/efficiency* of your program. You must write all the code yourself, and should not use any external library routines, except those that are considered standard. The usual input/output and other unavoidable routines are exempted.

Follow these procedures while submitting this assignment:

The assignment should be submitted online via moodle strictly as follows:

- All your scripts MUST contain your name and student ID.
- Use `gzip` or `Winzip` to bundle your work into an archive which uses your student ID as the file name. (STRICTLY AVOID UPLOADING `.rar` ARCHIVES!)
  - Your archive should extract to a directory which is your student ID.
  - This directory should contain a subdirectory for each of the four questions, named as: `q1/, q2/, q3/ and q4/`.
  - Your corresponding scripts and work should be tucked within those subdirectories. **Note:** If you have scripts that are common to multiple questions, such as your suffix tree construction, you may include the script in the parent directory, or copy it into the relevant subdirectories.
- Submit your zipped file electronically via Moodle.

Academic integrity, plagiarism and collusion

Monash University is committed to upholding high standards of honesty and academic integrity. As a Monash student your responsibilities include developing the knowledge and skills to avoid plagiarism and collusion. Read carefully the material available at https://www.monash.edu/students/academic/policies/academic-integrity to understand your responsibilities. As per FIT policy, all submissions will be scanned via MOSS.

Assignment Questions

1. (3 marks) Recall that a spanning tree of a graph $G(V, E)$ is a subgraph T consisting of all the vertices in V and some subset of edges $E' \subset E$ such that T is a tree. That is, a spanning tree is a collection of edges that form a tree and connect all the vertices in V. A spanning tree in a weighted graph is called a weighted spanning tree and a weighted spanning tree of minimum total possible weight is called a minimum spanning tree.

   Given a weighted undirected graph $G(V, E)$, your task is to write a program that uses Kruskal's algorithm (see notes from FIT2004) to find a minimum weight spanning tree in G. You may assume that G is connected.

   Strictly follow the following specification to address this task:

   Program name: `kruskals.py`

   Arguments to your program: The total number of vertices $|V|$ (the size of G's vertex set) as a positive integer, and a plain text file. Every line in the text file contains single edge with weight w connecting vertex u to vertex v in G represented by 3 integers in the following format:

   u v w

   The integers u and v are integers in the range $[0 \ldots |V| - 1]$. Each of the integers in the line are separated by a single space.

   Command line usage of your script:

   `kruskals.py <|V|> <edge file for G>`

   Do not hard-code the filename in your program.

   Output file name: `output_kruskals.txt`

   Output format: The output file should contain:

   - The total weight (as an integer) of a minimum spanning tree in G on the first line.
   - The edges of the minimum weight spanning found by your program. There should be one edge per line, (starting from the second line) in the same format as the input edge file.

   Example: In the example below the minimum weight spanning tree (shown in red) has weight $1 + 4 + 6 + 2 + 1 + 2 = 16$. In this case the usage of your program would be:

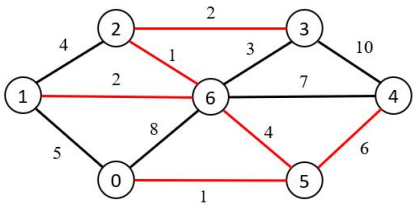
   ```
   python kruskals.py 7 G.txt
   ```

Where G.txt would contain:

```
0   1   5
1   2   4
2   3   2
3   4   10
4   5   6
5   0   1
0   6   8
1   6   2
2   6   1
3   6   3
4   6   7
5   6   4
```

Finally, the output of your program in this case would be a file `output_kruskals.txt` containing the weight of the minimum spanning tree (16) on the first line, followed by the edges contained in the tree:

```
16
0   5   1
2   6   1
2   3   2
1   6   2
5   6   4
4   5   6
```



Questions 2 and 3 must be addressed **using a suffix tree**. There are 6 marks assigned to suffix tree construction and these are included in the 8 marks available for question 2. That is for question 2, 6/8 marks are dedicated to suffix tree construction while the remaining 2 are allocated to your use of the suffix tree and the output of your program. As always, marking will be based on the efficiency of your suffix tree construction, and those of the tasks below.

2. (8 marks) Given a string `str`$[0 \ldots n-1]$, write a program that constructs its **suffix tree**, and using that suffix tree, outputs the suffix array of the string. Note suffix arrays were covered in FIT2004.
Strictly follow the following specification to address this task:

   Program name: `suffix_array.py`

   Argument to your program: An input file containing `str`$[0 \ldots n-1]$. It is safe to assume that:

   - `str` consists of lower case English characters, i.e. ASCII characters with values in the range $[97 \ldots 122]$.
   - There are no line breaks in the input file, and that all characters `str`$[0 \ldots n-1]$ are lexicographically larger than the terminal character, \$, which you will need to append to `str`$[0 \ldots n-1]$ after reading it from the file.

   Command line usage of your script:

   ```
   suffix_array.py    <file containing str[0...n-1]>
   ```

   Do not hard-code the filename in your program.

   Output file name: `output_suffix_array.txt`

   - Output format: The output file should contain the start indices of all the suffixes in `str`$[0 \ldots n-1]$\$ in the order that they appear in the corresponding suffix array. Each line of the file should contain a single suffix index as shown in the example below.

   Example: If `str`$[0 \ldots 10]$\$ = `mississippi`\$ then the output would be:

   11
   10
   7
   4
   1
   0
   9
   8
   6
   3
   5
   2

3. (3 marks) For strings `S1`$[0 \ldots n-1]$ and $S2[0 \ldots m-1]$, define the values $L(i,j)$ for any $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$, as the longest prefix common to the suffixes starting at position i in `S1` and position j in `S2`.
For a given pair of input strings `S1`$[0 \ldots n-1]$ and $S2[0 \ldots m-1]$ and a list of (i,j) pairs your task is **to use a suffix tree** to compute the $L(i,j)$ value for each (i,j) pair in the list.

Strictly follow the following specification to address this task:

Program name: `lcp.py`

Arguments to your program:

(a) An input file containing `S1`$[0 \ldots n-1]$.
(b) An input file containing `S2`$[0 \ldots m-1]$.
(c) An input file containing a list of (i,j) pairs, one per line in the format:
i j
where $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.

It is safe to assume that:

- `S1` and `S2` consist of lower case English characters, i.e. ASCII characters with values in the range $[97 \ldots 122]$.
- There are no line breaks in the files containing `S1` and `S2`.
- Each line of the (i,j) pairs file contains a single pair where each index is separated by a single whitespace (see example below).

Command line usage of your script:

```
lcp.py <S1[0...n-1] file> <S2[0...m-1] file> <file containing (i,j) pairs>
```

Do not hard-code the filename in your program.

Output file name: `output_lcp.txt`

- Each line should contain of a single pair of i and j indices followed by the corresponding $L(i,j)$ value, with each value being separated by a single whitespace e.g. in the following format:
i j L(i,j)

Example: If `S1`$[0 \ldots 9]$ = `abcdacbdab`, `S2`$[0 \ldots 7]$ = `dacbdabc` and the input (i,j) pairs file contained:

3 0
4 2
0 5

then the output should be:

3 0 7
4 2 0
0 5 3

4. (6 marks)
To give your fingers a break from coding this question will focus on the amortised analysis of a Fibonacci heap.

Strictly follow the following specification to address this task:

File name: `fibonacci.pdf`

Submission instructions:

Your pdf file should contain your solutions to parts a-j below. Clearly indicate which question each section in your solution refers to and be careful to include all the necessary working. You may type your solutions or write them by hand and scan them into a pdf file. Please note that if you choose to submit hand written solutions it is your responsibility to ensure that your hand writing is legible. If your marker cannot read your work you will not receive marks for it!

Once you have collected your solutions into a pdf file (named appropriately) simply place it inside of your q4 subdirectory.

Preliminaries

Recall that the goal of any amortised analysis is to provide an upper bound on the cost of **any** sequence of operations that might be performed on a given data structure. That is, if we denote the true, or actual cost of some sequence of operations $\{op_1, op_2, \ldots, op_n\}$ as $TC(\{op_1, op_2, \ldots, op_n\})$ then we desire some costing scheme that always ensures that the amortised cost of this sequence ($AC(\{op_1, op_2, \ldots, op_n\})$) is such that,

$$AC(\{op_1, op_2, \ldots, op_n\}) \geq TC(\{op_1, op_2, \ldots, op_n\}). \tag{1}$$

It is important to stress that this must be true for **any** sequence of operations not just one specific sequence!

At this point you are likely already familiar with one costing scheme known as the aggregate method where we simply assign $AC(\{op_1, op_2, \ldots, op_n\})$ to be the cost (or some upper bound on it) of the worst case sequence. We then distribute this work evenly amongst the constituent operations to define the amortised cost of each operation o_i - denoted ac_i - as,

$$ac_i = \frac{AC(\{op_1, op_2, \ldots, op_n\})}{n}. \tag{2}$$

However, the amortised cost of the sequence need not be distributed evenly amongst the operations in the sequence.

In the *potential method* we instead define the amortised cost of an operation o_i to be,

$$ac_i = tc_i + \Phi(D_i) - \Phi(D_{i-1}), \tag{3}$$

where tc_i is the true, or the actual cost of operation o_i, D_i is the state of the data structure after operation o_i has been completed, and Φ is known as a *potential function*. The potential function simply maps the state of the data structure D_i to a real number $\Phi(D_i)$, which is referred to as the potential associated with D_i.

Using our newly defined notation and noting that the amortised cost of a sequence of operations is simply the sum of the amortised cost of each individual operation in the sequence we obtain,

$$AC(\{op_1, op_2, \ldots, op_n\}) = \sum_{i=1}^{n} ac_i = \sum_{i=1}^{n} tc_i + \Phi(D_n) - \Phi(D_0) \tag{4}$$

(a) (0.5 marks) Use equation 3 to derive the right-hand-side of equation 4.

Since $TC(\{op_1, op_2, \ldots, op_n\}) = \sum_{i=1}^{n} tc_i$ we require $\Phi(D_n) \geq \Phi(D_0)$ in equation 4 so that the amortised cost of the sequence always upper bounds its true cost. Moreover, it is common to define a potential function such that $\Phi(D_0) = 0$ and since in practice we don't always know how many operations will be in our sequence we also require $\Phi(D_i) \geq \Phi(D_0)$ for all i. Thus, if we can show that $\Phi(D_i) \geq 0$ for all i, we guarantee that the amortised cost will always upper bound the true cost of any sequence.

Fibonacci heaps

To this point everything has been a little abstract, so let's try to intuit what we have so far and begin applying it to a Fibonacci heap. Examining equation 3,

$$ac_i = tc_i + \Phi(D_i) - \Phi(D_{i-1}),$$

we can see that the amortised cost for operation o_i can be more or less than the true cost. This means that the potential function Φ acts as a kind of credit and debit account for our data structure. If the change in potential $\Delta\Phi = \Phi(D_i) - \Phi(D_{i-1})$ is positive this amounts to overestimating the cost of the operation, which increases the potential and corresponds to making a credit to our account. Conversely if $\Delta\Phi < 0$ then our potential decreases as we have underestimated the cost of the operation, which corresponds to making a debit to our account. Since initially $\Phi(D_0) = 0$ and we want $\Phi(D_i) \geq 0$ for all i, we essentially start with an empty account that we want to keep out of debit by ensuring that we overestimate at least as much as we underestimate. This ensures that our amortised cost always upper bounds the true cost of the sequence. With this you should now have the necessary background knowledge required to analyse a Fibonacci heap which will be the focus of the remainder of the assignment.
Let's begin by considering a restricted version of a Fibonacci heap that only supports the operations:

- Merge[1]
- Insert
- Find_min
- Extract_min

The reason for the omission of decrease key and delete from this list will become clear later. To analyse the amortised complexity of these operations let's define the potential of a restricted Fibonacci heap H to be,

$$\Phi(H) = T(H), \tag{5}$$

where $T(H)$ is the number of nodes in the root list of H. If we were to use this potential to derive the amortised cost of *find_min* we'd proceed as follows.
First we need to find the true cost of the operation, which is clearly $O(1)$ since the minimum element in the heap can be accessed directly via $H_{\min}$. In addition, since this operation does not alter the number of nodes in the root list of H we have,

$$\Phi(D_i) - \Phi(D_{i-1}) = T(H) - T(H) = 0,$$

and so by equation 3 the amortised cost of *find_min* is given by,

$$\begin{aligned} ac_i &= tc_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= O(1) + 0 \\ &= O(1). \end{aligned}$$

In this case we see that the amortised cost for *find_min* is actually equal to the true cost of the operation.

[1]To analyse merge using our potential method we'd need to consider the potential of two separate heaps H_1 and H_2 initially. To do so we simply define the potential of a collection of heaps to be the sum of their individual potentials.

(b) (0.5 marks) Use the potential function given in equation 5 to derive the amortised cost of insert.

Now that we've seen some examples of the potential method let's quickly revisit equation 3,

$$ac_i = tc_i + \Phi(D_i) - \Phi(D_{i-1}).$$

If we were to overestimate the cost of operation o_i by 3 units of potential then our current interpretation would imply that those 3 units of potential are worth 3 units of work. We can then use this potential to compensate for 3 units of work the next time we underestimate the cost of an operation. However, we are free to scale the units of the potential so that a single unit corresponds to any constant amount of work. In other words, equation 3 should really be written as,

$$ac_i = tc_i + m\left[\Phi(D_i) - \Phi(D_{i-1})\right], \tag{6}$$

where m is a positive real constant that we've implicitly assumed to be equal to 1 up until this point.

(c) (1 mark) Show that the amortised cost of *extract_min* is $O(D_{\max}(N))$, where N is the number of nodes in H and $D_{\max}(N)$ denotes the maximum degree of a node in H.
Hint: Start by showing the true cost is $O(T(H) + D_{max}(N))$ and then show that you can scale the units of potential to leave $O(D_{max}(N))$ for the amortised cost.

(d) (0.5 marks) Argue that for our restricted heap $D_{\max}(N) \leq \lfloor \log_2(N) \rfloor$.

Now that we've analysed the operations of our restricted heap (you can convince yourself that the amortised cost of merge is $O(1)$) let's reintroduce *decrease_key*.
The reason we initially omitted this operation from our list was that it complicates the analysis slightly. For instance, the bound derived in part (f) is no longer valid, but we can still show that the amortised cost of *extract_min* is $O(\log(N))$ even when *decrease_keys* are allowed.
To do so we need to show that $D_{\max}(N)$ is still $O(\log(N))$. This amounts to showing that in the worst case the size of a given tree in H is still given by a function that is exponential in the degree of the tree, so let's determine how small our trees can get. Let's define a *maximally decreased* tree of degree k to be a tree - of degree k - in a Fibonacci heap that has lost the maximum number of nodes from its subtrees due to *decrease_key* operations. Put another way, a maximally decreased tree is one from which we can't remove any more nodes via *decrease_keys* without also decreasing the degree of the root. For instance, the maximally decreased trees of degree 0, 1, 2 and 3 contain 1, 2, 3 and 5 nodes respectively, as shown in figure 1.

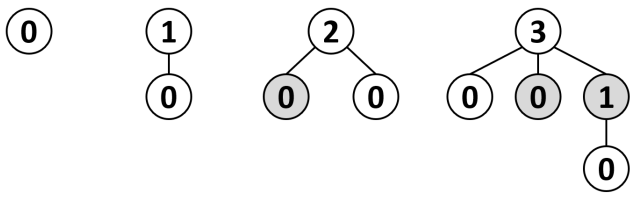


Figure 1: Maximally decreased trees of degree 0, 1, 2 and 3. The marked - denoted by the shading - nodes have lost one child and the numbers represent the degree of each node rather than the key of the node.

(e) (0.5 marks) How many nodes are in a maximally decreased tree of degree k?
Hint: Begin by simply drawing out maximally decreased trees of increasing degree (you might find the representation used in figure 1 helpful). Now count the number of nodes in each of these trees, can you spot a pattern? It may help to recall that the Fibonacci sequence is defined as,

$$F_0 = 0, F_1 = 1 \text{ and } F_n = F_{n-1} + F_{n-2} \text{ for } n > 1,$$

and the start of the sequence is shown below.

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
0	1	1	2	3	5	8	13	21	34	55

(f) (1.25 mark) Prove - using the principle of mathematical induction - that the expression you derived for the size of a maximally decreased tree in the previous part is correct.
Hint: Think about how you can construct a maximally decreased tree of degree k from maximally decreased trees of smaller degree.

At this point you should have an expression for the number of nodes in a maximally decreased tree of degree k. By definition we know that this tree contains the fewest nodes of any tree of degree k in our heap. Therefore, in general a tree of degree k rooted at r in our heap must contain at least this many nodes i.e.,

$$size(r) \geq g(k), \tag{7}$$

where $size(r)$ is the number of nodes in the tree and $g(k)$ is the number nodes in a maximally decreased tree of degree k that you found in part (e).

(g) (0.5 marks) Using equation 7 and the fact that,

$$F_{k+2} \geq \phi^k, \tag{8}$$

for all integers $k \geq 0$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, show that $D_{max}(N) = O(\log(N))$ even when *decrease_key* operations are allowed.

The result in part (g) proves that even when *decrease_key* operations are allowed the amortised complexity of *extract_min* is still $O(D_{\max}(N)) = O(\log(N))$.

Finally let's conclude by deriving the amortised cost of *decrease_key*.

(h) (0.5 marks) Using the potential given in equation 5, derive the amortised cost of a decrease key operation. Is this what you'd expect?
Hint: Assume that you perform $c \geq 0$ cascading cuts during the operation and consider the true and amortised cost in this case.

Currently our potential is simply storing credits during *decrease_key* that are used during the *extract_min* to offset the fact that we are adding more nodes to our root list. However, we aren't using any of our potential to compensate for the fact that marking nodes also produces extra work in the form of cascading cuts in future *decrease_keys*. To resolve this let's 'back charge' this work to the *decrease_keys* that marked the nodes in the first place.

That is, we want to modify our potential so that we account for this work (make a credit) every time we mark a node.
Let's consider the potential,

$$\Phi = T(H) + \alpha M(H), \tag{9}$$

where α is a positive constant and $M(H)$ denotes the number of marked nodes in H. Now every time we mark a node our potential increases by α units, but what value should we choose for α?

(i) (0.5 marks) Determine the smallest integral value of α for which the potential in equation 9 yields an $O(1)$ amortised cost for *decrease_key*.

(j) (0.25 marks) Explain why this value of α intuitively makes sense.

```
-=o0o=-
  END
-=o0o=-
```