COMP3821 Assignment 2

March 2021

1.1 **Solution:** Let the algorithm take the a positive integer input n. If $n = 1$ or $n = 2$, return n. Have a function called $closestFib(n)$, that initializes three variables x, y, z to 1, 2, 3 respectively (i.e., the first 3 terms in the given Fibonacci sequence). While $z <=$ n, assign y to x and z to y, and assign the sum of the updated x and y to z.

The returned value of y will be the greatest Fibonacci number <= n. In another function $fibNums(n)$, while $n > 0$, decrement n by the returned value of $closestFib(n)$, after every call to $closestfib(n)$. The sum of these returned values by $closestFib$ will be a non-consecutive sum of Fibonacci numbers representing n.

With $fibNums$ taking $O(log*n)$ and the $closestFib$ function being called inside the loop of $fibNums$ taking $O(n)$ time. The resulting time complexity of the above algorithm will be $O(n * logn)$. .

1.2 **Proof:** Let the greatest Fibonacci number <= the positive input n, returned by $closestFib$, be f(k) (k^{th} Fibonacci number). The algorithm is called $fibNum$ as described in 1.1

Base case: Let n=1.
Since 1 is F_1 in the given Fibonacci sequence, 1 is returned. Thus, $fibNum(1)$ returns the correct output.

Inductive Hypothesis: Assume that $fibNum(n - f(k))$ gives a representation of n-f(k) as a sum of non-consecutive Fibonacci numbers.

Inductive Step: We will show that $fibNum(n)$ also gives a representation of n as a sum of non-consecutive Fibonacci numbers.
$n - f(k)$ can also be represented as a sum of non-consecutive Fibonacci numbers. By induction, $n - f(k)$ does have consecutive Fibonacci numbers in its representation. If $n - f(k)$ has $f(k - 1)$ in its Fibonacci representation, then n will have consecutive Fibonacci numbers in its representation.

If $n - f(k) = n - f(k-1) + n - f(k-y) + ...$, then $f(k)$ cannot be the closest Fibonacci number <= n, since $f(k+1) = f(k) + f(k-1)$. Then, $f(k+1)$ would be the closest Fibonacci number to n if $n - f(k)$ has $f(k-1)$ in its Fibonacci representation, which is a contradiction to our assumption that $f(k)$ is the greatest Fibonacci number <= n.

Hence by induction, our algorithm $fibNum(n)$ is correct.

1.3 **Proof:** To prove the solution obtained by the algorithm is unique for each n, we will use the following theorem:
The sum of any (non-empty) set of non-consecutive Fibonacci numbers with it's largest element being F_i, is strictly less than F_{i+1} (the next Fibonacci number in the sequence).

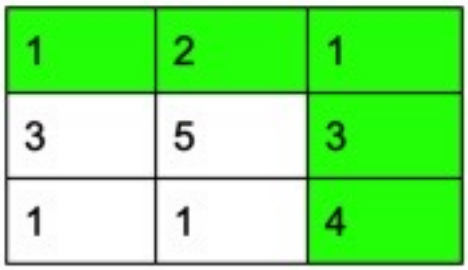
Let X and Y be sets of two distinct non-consecutive Fibonacci sum representations of some positive input n.
Let $X' = X/Y$ and $Y' = Y/X$, which will remove the common elements between X and Y. Since X and Y have the same sum, it follows that $\sum X' = \sum Y'$.

Assume X' is an empty set. Then Y' must also be empty to have the same sum, since it only must consist of positive Fibonacci numbers otherwise.
But neither X' or Y' can be empty since $X \neq Y$. Let F_X and F_Y be the largest elements of X and Y respectively, where $F_X \neq F_Y$ as $X' \cap Y' = \phi$.

Assume $F_X < F_Y$. Using the above theorem, $\sum X' < F_{X+1}$.
Therefore from our assumption, $\sum x' < F_Y$.

But $\sum X' = \sum Y'$, thus from this contradiction we can conclude that X' and Y' are empty sets. Thus X=Y, and thus each sum representation produced by the algorithm is unique.

2.1 **Solution:**



This solution using the greedy approach yields a solution of $1+2+1+3+4 = 11$.

This counterexample doesn't use the greedy approach and yields the minimum solution of $1 + 3 + 1 + 1 + 4 = 10$.

From the counter example, we can see that that the method to get minimum sum along the path from top-left to bottom right doesn't take the locally optimal choice. Instead of going 1→2 at the start, it goes 1→3 which a greedy algorithm wouldn't do, but nonetheless leads to an overall minimum sum.

2.2 **Solution:** To solve this problem, we can use a modified version Dijkstra's shortest path algorithm to find the minimum sum from the top left of the grid to the bottom right.

We can imagine the n*n grid as a directed graph where the values on the grid are vertices, and the arrows from these vertices can only go down and/or right for any vertex on the grid, except for the bottom-right vertex. The only thing the modified Dijkstra algorithm does differently is it uses the value of the vertex as

the edge weights when relaxing the "distances" between two vertices on the grid.

The running time of a standard Dijkstra's algorithm is $O(ElogV)$, where E is the number of edges and V is the number of vertices.

Since the number of edges in n*n grid is $2 * n^2 - 2n$, this modified algorithm has a running time of $O((2 * n^2 - 2)logn^2)$, which can be simplified to $O(n^2logn)$.

3.1 **Solution:** To design an algorithm that runs in $O(m^2logm)$ we do the following:

- Have an array called critEdges[] that stores all the edges that appear in every MST for the graph G.
- Get the MST (minimum spanning tree) using Kruskal's algorithm on G, and caluclate the cost of the MST. Call this cost minCost.
- Have a loop that takes one edge E (that hasn't been considered previously) at a time. In this loop, get the MST using a Kruskal's, but don't consider E while getting the MST.
- Get the cost of the MST for updated graph G'.
- If the cost of the MST of G' greater than the minCost, then E must appear in every MST of G as the MST can't exist without it. Add this E to the critEdges[] array.
- Loop until the above has been done for every edge in G.

Calculating the cost of the MST takes $O(m)$. Kruskal's algorithm takes $O(mlogn)$. Since our algorithm checks if every edge is critical to the MST one at a time, and there are m edges, the loop will run in $O(m^2logn)$ and thus overall complexity is $O(m^2logn)$.

3.2 **Solution:** To design an algorithm that runs in $O(nmlogn)$ we do the following:

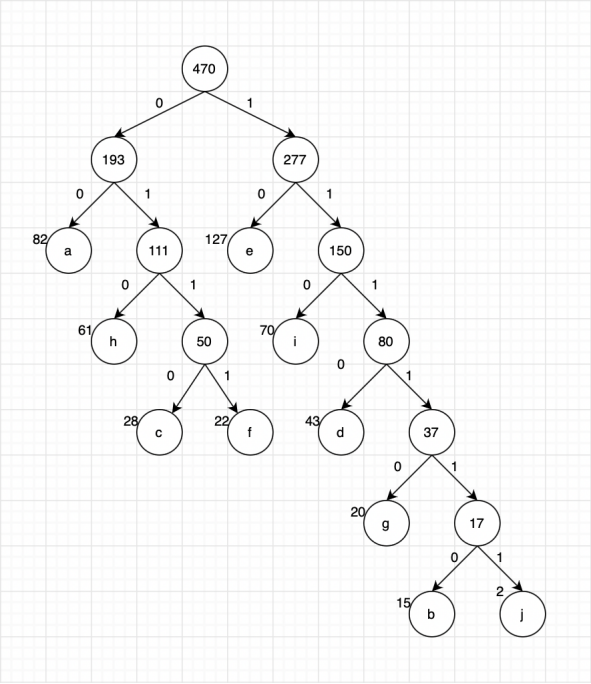
- Have an array that called critEdges[] that stores all the edges that appear in every MST for the graph G.
- Get the MST (minimum spanning tree) using Prim's algorithm on G, and calculate the cost of the MST. Call this cost minCost.
- Then, use a modifed version of Prim's algorithm where you store vertices of the graph by the order of their edge weights. The edge to be put in a

Fibonacci heap is chosen by the adjacent vertex to the current vertex with the smallest edge weight between the two vertices. The group of vertices represents the vertices already selected in the MST. Calculate the cost of the MST

- You run the above algorithm n times to go over each vertex in the graph, each time with an edge E of the graph "removed" (not considered by Prim's algorithm). If the cost of the MST of the graph G' is greater than minCost, add E to critEdges[].

Using a Fibonacci Heap, choosing the suitable minimum edge to add to the heap for the MST takes $O(logn)$, thus for m edges it will take $O(mlogn)$. We run the modified Prim's algorithm n times, this the overall time complexity is $O(nmlogn)$.

4. **Solution:** Huffman Tree with 0 for a left arrow and 1 for a right arrow.



The path from the root node to a leaf node is the optimal prefix code for the letter corresponding to that leaf node.

a: 00
b: 111110
c: 0110
d: 1110

e: 10
f: 0111
g: 11110
h: 010
i: 110
j: 111111

5.1 **Solution:** To have an allocation algorithm that maximizes Abdallah's overall donation to CSESoc we can adopt a greedy approach to design the algorithm. We start with n in the array A[1..n] of awesomeness values for n students where n is odd and $n >= 3$.

- We can first sort the array in non-increasing order using MergeSort. We then have a loop that iterates while the pointer to the (i+1)th index is less than the array.length()-1 .
- If i is the index of the feature of the highest awesomeness value, we select the corresponding student to be the "leader".
- The (i+1)th and (i+2)th indexes will correspond to the second and third highest features by value. The corresponding students to these 2 indexes are chosen as "submitters". The rest of the students become "reviewers".
- Let the value at the i^{th} index be returned as leader, (i+1)th and (i+2)th values be returned as submitter1 and submitter2 respectively for that week.
- The pointers to (i+1)th and (i+2)th index are incremented by 2.
- The student chosen to be the leader at the start never changes his/her role as leader.
- Each iteration of the loop represents a week worked on the project.

Shifting pointers happens in $O(1)$ time. MergeSort is $O(nlogn)$, so the overall running time of this algorithm is $O(nlogn)$.

Proof: (Using the Greedy Stays Ahead proving technique) Based on the algorithm as described above, we sort the initial array of awesomeness values. Let the set of donation values of each week, for x weeks be $\{i_1, ...i_x\}$, where the donation value of the week represents the sum of the leader and the 2 submitter values for that week. We will prove that the sum of all the elements in the set represents the optimal solution for leader allocation.

Let A = $\{i_1, ...i_x\}$ be the weekly sums produced by the greedy algorithm, in the order of increasing week number. Let O = $\{j_1, ...j_x\}$ be the optimal solution with the same order.

Let f(i_p) be the sum of the elements up-to and including i_p, where i_p belongs to $\{i_1, ...i_l\}$. We must show that for all $l <= x$, $f(i_l) >= f(j_l)$, using induction.

Base Case: Let l=1. Since this corresponds to week 1, every student demonstrates their branch and therefore, $f(i_l) >= f(j_l)$.

Induction hypothesis: For some $h > 1$, assume the statement is true for h-1

Induction step: Prove true for h.
From the induction hypothesis, $f(i_{h-1}) >= f(j_{h-1})$, so any sums produced by the optimal solution up-to a particular week h-1 is certainly valid if produced by the greedy algorithm for the same number of weeks. Therefore, it must be that $f(i_h) >= f(j_h)$.

So for all $l <= x$, $f(i_l) >= f(j_l)$. If A is not optimal, then the sum of all the week values in A is strictly less than than the sum for all the week values in O. This contradicts the fact that $f(i_x) >= f(j_x)$, hence A must be the optimal solution and produce the maximum sum (i.e. maximum contribution by Abdallah for CSESoc). Since it produces the maximum sum each week, our method of leader allocation is correct.

5.2 **Solution:** We can adopt a greedy approach to design the algorithm. We start with n in the array A[1..n] of awesomeness values for n students where n is odd and $n >= 3$.

- We can first sort the array in non-increasing order using MergeSort. We then have a loop that iterates while the pointer to the (i+2)th index is less than the array length.
- If i is the index of the feature of the highest awesomeness value, we select the corresponding student to be the "leader".
- The (i+1)th and (i+2)th indexes will correspond to the second and third highest features by value. The corresponding students to these 2 indexes are chosen as "submitters". The rest of the students become "reviewers".
- Sum the above 3 values and add it to the total donation.
- The pointers to (i+1)th and (i+2)th index are incremented by 2.
- The student chosen to be the leader at the start never changes his/her role as leader.

- Each iteration of the loop represents a week worked on the project.

The overall running time of this algorithm is $O(nlogn)$.

Pseudo-Code:

```
int allocation (int a[]) {
    //in non-increasing order
    mergeSort(A[])

    //initialize indices and donation value
    donation=0
    leader=0
    submitter1_index=1
    submitter2_index=2

    while(submitter2_index <= A.length()-1){
        week_donation = A[leader] + A[submitter1_index] + A[submitter2_index]
        donation += week_donation
        submitter1_index += 2
        submitter2_index += 2
    }

    return donation
}
```

5.3 **Solution:** Summing elements and shifting pointers in the above algorithm both happen in $O(1)$ constant time in a loop that takes $O(n)$. MergeSort that happens before hand is $O(nlogn)$, so the overall running time of this algorithm is $O(nlogn)$.

5.4 **Solution:** To calculate the maximum possible amount CSESoc can keep, we assume that CSEsoc can also potentially run a loss if the sum of the number of bugs is greater than the donation amount X.
To design an algorithm we can build a min-heap from the input array in O(n log n).

- Have a total sum called bugSum that is the sum of all the number of bugs demonstrated each week.
- 1. Store and then delete the three smallest values in the min-heap.
- 2. Add the sum of these 3 stored values (called weekSum) to the bugSum.
- 3. Insert the weekSum into the min-heap.

- 4. Repeat from step 1. until no elements are left in the min-heap.
- Return X-bugSum. This returned value is the maximum amount CSESoc can keep.

Inserting an element into a min-heap takes $O(logn)$ time. Since there are n elements in the array, building the min-heap will take $O(nlogn)$. Getting the minimum value from the heap takes $O(1)$, and deleting and inserting a node into the heap takes $O(logn)$ time. Thus the overall time complexity is $O(nlogn)$.

End.