NATIONAL UNIVERSITY OF SINGAPORE
SCHOOL OF COMPUTING

FINAL TERM TEST FOR
Semester 2, AY2019/2020

CS3230 – DESIGN AND ANALYSIS OF ALGORITHMS

5 May 2020 Time Allowed: 2 hours

Instructions to Candidates:

1. This paper consists of **FOUR** questions (each consisting of **TWO** sub-questions) and comprises **NINE (9)** printed pages, including this page.
2. Answer **ALL** questions.
3. This is an **OPEN BOOK** examination.
4. Write down **Metric No.** on top of every page of your answer sheet

5. **SUBMISSION:** Page limit for each one of **FOUR** questions is **three.** After writing down your answers scan them to convert to **.pdf** file. Create **exactly one file for each of FOUR questions** and rename your file as **<Metric No.>_T<tutorial group no.>_Q<question no.>.pdf** (for example, A324516H_T01_Q3.pdf). Then submit each file separately in the corresponding LumiNUS folder (you can see four different folders for each of FOUR questions).

QUESTION	POSSIBLE	SCORE
Q1	20	
Q2	20	
Q3	20	
Q4	20	
TOTAL	**80**	

IMPORTANT NOTE:

- *You can* ***freely quote*** *standard algorithms and data structures covered in the lectures, tutorials and assignments. Explain* ***any modifications*** *you make to them.*
- *Unless otherwise specified, you are expected to* ***prove (justify)*** *your results.*

Q1. (20 points)

(a) Consider a line with n+1 stations S_0, S_1, …, S_n, where S_0 is the initial station and S_n is the final station. We want to transfer an object from S_0 to S_n. At every station, we have a catapult. We can use the catapult at station S_i to throw the object to any station S_j where j = i+1, …, i+f(i). Note that f(i)≥1 for all i. We want to transfer the object from S_0 to S_n using the minimum number of shoots. You believe that this problem can be solved using dynamic programming. Given f(i) for i=0, 1, …, n-1 (as an array F[0…n-1]), can you give an $O(n^2)$-time dynamic programming algorithm to solve this problem? Please detail your algorithm and argue that it is correct. Please also state the time complexity of your algorithm.

Soln: Let V(i) = the minimum number of rounds of shoots to transfer the object from S_i to S_n.

We have V(n) = 0.
For 0 ≤ i < n, $V(i) = \min_{i+1 \le j \le \min\{n, i+f(i)\}} V(j) + 1$.

The dynamic programming algorithm is as follows.

```
V(n)=0;
For i = n-1 down to 0
  V(i) = V(i+1)+1
  For j = i+2 to min{$n, i + f(i)$}
          V(i) = min { V(i), V(j)+1 };
Report V(0);
```

We need to fill in n entries. Each entry can be computed in n steps. The running time of the algorithm is $O(n^2)$.

Q1. (Continued …)

(b) Suppose we have $i+f(i) \geq j+f(j)$ for all $i>j$. Under this extra assumption can you give an O(n)-time algorithm to solve this problem? Please detail your algorithm and argue that it is correct. Please also state the time complexity of your algorithm.

Soln: We claim that $V(i) = \min_{i+1 \leq j \leq \min\{n, i+f(i)\}} V(j) + 1 = V(\min\{n, i+f(i)\}) + 1$.

Based on the claim, the following algorithm returns the answer.

```
i=0; step=0;
While (i<n) {
  i = i+f(i);
  step++;
}
return step;
```

The running time of the algorithm is O(n).

Now, we show that the claim is correct by exchange argument.

Suppose the optimal solution passes the object in the following sequence of stations: 0, i_1, i_2, …, i_k, n.

Assume the algorithm passes the object in the following sequence of stations: 0, j_1, j_2, …, j_h, n.

By contrary, assume the two sequences are different.
Let p be the smallest index such that $i_p \neq j_p$. Note that there exists an optimal solution that pass the object in the following sequence 0, i_1, …, i_{p-1}, j_p, i_{p+1}, …, i_k, n.

The note is correct as our greedy choice always choose the further station to go. So, $j_p = i_{p-1} + f(i_{p-1}) > i_p$. This implies $j_p+f(j_p) \geq i_p+f(i_p)$. Hence, from jp, we can go to i_{p+1}. The claim is correct.

Q2. (20 points)

(a) You are given an integer array A[1..n] and an integer x. Can you develop an efficient algorithm Partition(A, x) that partitions A into two arrays X and Y such that all numbers in X are smaller than x and all numbers in Y are bigger than or equal to x? What is the running time of your algorithm?

Soln: It can be done in O(n) time.

```
Partition(A, x) {
  p=1; q=1;
  For i = 1 to n {
          If (A[i]<x) {
                  X[p]=A[i]; p++;
          } else {
                  Y[q]=A[i]; q++;
          }
  }
  return (X, Y);
}
```

Q2. (Continued …)

(b) You have an unsorted integer array A[1..n]. You also have a sorted integer array B[1..k] ($k<n$), where $B[1]<B[2]<\ldots<B[k]$. For any $1\le i<k$, let S_i be the set $\{ A[j] \mid B[i]\le A[j]<B[i+1] \}$. You need to compute the array C[1..k-1] where C[i] is the median of all integers in S_i if S_i is a non-empty set; otherwise set C[i]=0. Can you propose an $O(n \log k)$ time algorithm to compute C[1..k-1]?

Soln: We can compute C[1..k-1] by ComputeMedian(A, B[1..k]). The running time is O(n lg k).

```
ComputeMedian(A[1..n], B[i..j]) {
  If (i==j) return an empty list;
  If (j==i+1) {
          (X',X)=Partition(A, B[i]);
          (Y',Y)=Partition(X, B[i+1]);
          Compute and return the median of Y';
  }
  m=$\lfloor (i+j)/2 \rfloor$;
  (X, Y) = Partition(A, B[m]);
  C1=ComputeMedian(X, B[i..m]);
  C2=ComputeMedian(Y, B[m..j]);
  return C1•C2;
}
```

Q3. (10+10= 20 points)

(a) Suppose you are given two strings x and y of length n each, over ternary alphabet, i.e., $x, y \in \{0,1,2\}^n$. Now consider the problem of finding a Longest Common Subsequence of x and y. Design an $O(n)$ time algorithm with approximation ratio 1/3. Provide clear description of your algorithm and its proof of correctness. (Note, in the lecture we have seen an algorithm that finds Longest Common Subsequence of x and y in $O(n^2)$ time.)

Soln: Count the number of 0,1 and 2 in both the strings. Let c_0, c_1, c_2be the count of number of 0,1 and 2 respectively in the string x. Similarly let d_0, d_1, d_2be the count of number of 0,1 and 2 respectively in the string y. Next compute $m_i = \min\{c_i, d_i\}$ for all $i \in \{0,1,2\}$. Then find the i with the largest m_i value, and say it is j. Output the string j^{m_j}(string of length m_j that contains only the character j).

Consider any largest common subsequence, which is of length say l. Now consider the character that occurs maximum number of times in it and say count of that character is r. Clearly $l \leq 3r$ and $r \leq m_j$. This implies that $m_j \geq l/3$, and hence approximation ration of our algorithm is 1/3.

Q3. (Continued …)

(b) The Independent Set problem is defined as: Given an undirected graph $G = (V, E)$ find a maximum sized subset $X \subseteq V$such that for all $u, v \in X$, $(u, v) \notin E$.

(c) Now consider the following randomized algorithm. Among $n!$(where $|V| = n$) possible orderings of the vertices pick one uniformly at random. Then start processing each vertex based on this selected ordering. Initially take an empty set S. Then at any particular step, if $u \in V$ is processed and for all $(u, v) \in E$, $v \notin S$, add u to S. Otherwise continue with the next vertex based on the previously selected ordering. Stop when all the vertices have been processed, and output the set S.

Show that the set S output by the algorithm is an independent set of expected size at least $\frac{1}{d+1}OPT$, where OPT denotes the size of the maximum independent set and d is the maximum degree of any vertex.

Soln:

By the construction the set S output by the algorithm is an independent set, since we add a vertex in it only if its neighbors have not been added.

Note that a vertex u will be added to S if it is the first vertex in its neighborhood to be processed. Given that the process order is selected uniformly at random the probability that a vertex will be processed before all of its neighbors is at least $\frac{1}{d_u+1}$.

Let X_u be an indicator variable for whether or not $u \in S$. So by linearity of expectation expected size of S is

$\sum_{u \in V} E[X_u] = \sum_{u \in V} \Pr[u \in S] \geq \sum_{u \in V} \frac{1}{d_u+1} \geq \sum_{u \in V} \frac{1}{d+1} = \frac{n}{d+1}$.

Clearly $OPT \leq n$, and this completes the proof.

Q4. (10+10 = 20 points)

(a) In the Set Union problem we have n elements, that each are initially in n singleton sets, and we want to support the following operations:

- **Union(A,B):** Merge the two sets A and B into one new set C = A∪ B destroying the old sets.
- **SameSet(x, y):** Return true, if x and y are in the same set, and false otherwise.

We implement it in the following way. Initially, give each set a distinct color. When merging two sets, recolor the smaller (in size) one with the color of the larger one (break ties arbitrarily). Note, to recolor a set you have to recolor all the elements in that set.
To answer **SameSet** queries, check if the two elements have the same color. (Assume that you can check the color an element in O(1) time, and to recolor an element you also need O(1) time. Further assume that you can know the size of a set in O(1) time.)

Use Aggregate method to show that the amortized cost is O(log n) for **Union**. That means, show that any sequence of m union operations takes time $O(m \log n)$. (Note, we start with n singleton sets.)

Soln: Initially we have n singleton set. So any sequence of m union operations can involve at most $2m$ many elements. By our implementation, cost of a union operation is equal to the number of elements recolored during that operation. Hence total cost of m operations is at most twice the total number of recoloring happens. Now count the total number of recoloring by element wise. Observe that each element can be recolored at most $\log n$ times. This is because since we are recoloring smaller set, an element is recolored k times means it was part of smaller set k times, and each time the size doubles. More specifically, if we consider all these k unions where that element is part of the smaller set, and let $s_1, \ldots, s_k$ be the sizes of those smaller sets. Clearly $s_i \geq 2s_{i-1}$, and so $k \leq \log n$.Thus total number of recoloring is bounded by $O(m \log n)$.

Q4. (Continued …)

(b) Suppose Alice insists Bob to maintain a dynamic table (that supports both insertion and deletion) in such a way its size must always be a Fibonacci number. She insists on the following variant of the rebuilding strategy. Let F_k denote the k-th Fibonacci number. Suppose the current table size is F_k.
After an insertion, if the number of items in the table is F_{k-1}, allocate a new table of size F_{k+1}, move everything into the new table, and then free the old table.
After a deletion, if the number of items in the table is F_{k-3}, we allocate a new hash table of size F_{k-1}, move everything into the new table, and then free the old table.
Use either Potential method or Accounting method to show that for any sequence of insertions and deletions, the amortized cost per operation is still O(1). (If you use Potential method clearly state your potential function. If you use Accounting method clearly state your charging scheme.)

Soln: Suppose the current table size is F_k, then there must be at most F_{k-1}elements present in the table.
Consider the following charging scheme for insertion: Charge \$4 for each insertion, use \$1 for insertion and remaining \$3 put in the bank. Between the creation of table of size F_{k+1} and F_{k+2} there must be at least $F_k - F_{k-1} = F_{k-2}$ elements being inserted. So the bank balance at the time of creating table of size F_{k+2} must be $3F_{k-2} > F_{k-3} + 2F_{k-2} = F_{k-1} + F_{k-2} = F_k$. So we can move all the F_k elements to the new table using the bank balance (free of cost). Hence the amortization cost of insertion is O(1).
For the deletion consider the following charging scheme: Charge \$3 for each deletion, use \$1 for deletion and remaining \$2 put in the bank. Between the creation of table of size F_{k-1} and F_{k-2} there must be at least $F_{k-3} - F_{k-4} = F_{k-5}$ elements being deleted. So the bank balance at the time of creating table of size F_{k-2} must be $2F_{k-5} > F_{k-6} + F_{k-5} = F_{k-4}$. So we can move all the F_{k-4} elements to the new table using the bank balance (free of cost). Hence the amortization cost of deletion is O(1).
Note, at any point bank balance is some constant times the number of elements present in the table, and hence is non-negative.

-- End of Paper ---