Due 23:59 Sept 25 (Sunday). There are 100 points in this assignment.
Submit your answers (**must be typed**) in pdf file to CourSys
`https://coursys.sfu.ca/2022fa-cmpt-705-x1/`. Submissions received after 23:59 will get penalty of reducing points: 20 and 50 points deductions for submissions received at $[00:00, 00:10]$ and $(00:10, 00:30]$ of Sept 26, respectively; no points will be given to submissions after 00 : 30 of Sept 26.

1. (Chapter 1 Problem 4 of the text book) 20 points

   There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. Assume that there were more students graduating than there were slots available in the m hospitals. Consider a problem of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. An assignment of students to hospitals is stable if neither of the following situations arises.

   First type of instability: There are students s and s', and a hospital h, so that

   - s is assigned to h, and
   - s' is assigned to no hospital, and
   - h prefers s' to s.

   Second type of instability: There are students s and s', and hospitals h and h', so that

   - s is assigned to h, and
   - s' is assigned to h', and
   - h prefers s' to s, and
   - s' prefers h to h'.

   So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students. Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

Answer

As stated in the question this problem is a Stable Matching Problem, except that Hospitals want more than one students and number of graduates (n) are more than the number of hospitals (m) $n > m$. For this problem the hospitals are the "proposers", in a sense they accept students into the hospital as residents. So when a student is accepted in a hospital, he/she has the option to accept or reject the hospital, based on

their preferences. The goal is to fill all of the h in H, where h is an individual hospital and H is the set of the hospitals. Similarly, s is an individual student and S is the set of students. Before the algorithm runs all of the slots are free. The algorithm runs as follows:

```
// Initializing
for h in H:
        h = Free
for s in S:
        s = Free

while(h in H such that h.free_slots>0 & h.offered.length!=S.length)
        h offers admission to s
        if s.free // has not received any other offers
                s accepts the offer
                h.free_slots −= 1
        else  // has received an offer from h'
                if s.preferences(h') > s.preferences(h)
                        s rejects the offer from h
                        s accepts the offer from h'
                        h.free_slots += 1
                        h'.free_slots −= 1
                else if s.preferences(h) > s.preferences(h')
                        s rejects h'
```

The Algorithm will find hospitals with free slots which haven't offered admission to every student, then the hospital h according to their preference offer admission to a student s. If s hasn't been accepted, they will accept this offer (temporarily) and the number of free slots will decrease by 1. If s has not received another offer from h', they accept the offer. If they have accepted a previous offer from h', they will see if the new hospital lists higher in their preferences in which case they will accept the offer and reject the previous hospital h. The $free\_slots$ in h' decrease by 1 and the $free\_slots$ in h increase by 1. If the previous hospital h ranks higher than h' they will reject the offer. This algorithm will keep on running until all the positions are filled.

---

2. (Chapter 2 Problems 1 and 2 of the text book. $\log n = \log_2 n$) 20 points

   Suppose you have algorithms with the five running times listed below. (Assume these are the exact running times.) How much slower do each of these algorithms get when you (a) double the input size, or (b) increase the input size by one?

   (1) n^2, (2) n^3, (3) $100n^2$, (4) $n \log n$, (5) 2^n.

Answer

(a) For the part (a) we just need to replace n by 2n and put it into the running time bound:

   (1). n^2: $(2n)^2 = 4n^2$. Difference $(4n^2 - n^2)$ in running time is $3n^2$.

   (2). n^3: $(2n)^3 = 8n^3$. Difference $(8n^3 - n^3)$ in running time is $7n^3$.

   (3). $100n^2$: $100(2n)^2 = 400n^2$. Difference $(400n^2 - 100n^2)$ in running time is $300n^2$.

   (4). $nlogn$: $(2n)log(2n) = 2n(log2 + logn) = 2nlog2 + 2nlogn$. Difference $(2nlog2 + 2nlogn - nlogn)$ in running time is $2nlog2 + nlogn$.

   (5). 2^n: $2^{(2n)} = 2^{2n}$. Difference $(2^{2n} - 2^n = 2^{2n-n} = 2^n)$ in running time is 2^n.

(b) For the second part, we will just replace n by n+1:

   (1). n^2: $(n+1)^2 = n^2 + 2n + 1$. Difference $(n^2 + 2n + 1 - n^2)$ in running time is $2n + 1$.

   (2). n^3: $(n+1)^3 = n^3 + 3n^2 + 3n + 1$. Difference $(n^3 + 3n^2 + 3n + 1 - n^3)$ in running time is $3n^2 + 3n + 1$.

   (3). $100n^2$: $100(n+1)^2 = 100(n^2 + 2n + 1) = 100n^2 + 200n + 100$. Difference $(100n^2 + 200n + 100 - 100n^2)$ in running time is $200n + 100$.

   (4). $nlogn$: $(n+1)log(n+1) = nlog(n+1) + log(n+1)$. Difference $[nlog(n+1) + log(n+1) - nlogn = n(log(n+1) - logn) + log(n+1) = nlog(\frac{n+1}{n}) + log(n+1)]$ in running time is $nlog(1 + \frac{1}{n}) + log(n+1)$.

   (5). 2^n: $2^{(n+1)} = 2^n * 2$. Difference $(2^n * 2 - 2^n)$ in running time is 2 times.

Take the following list of functions and arrange them in ascending order of growth rate, that is, if function $g(n)$ immediately follows function $f(n)$ in your list, then $f(n)$ is $O(g(n))$.

$f_1(n) = n^{2.5}$, $f_2(n) = \sqrt{2n}$, $f_3(n) = n+10$, $f_4(n) = 10^n$, $f_5(n) = 100^n$, $f_6(n) = n^2 \log n$.

Answer

Growth Rate (ascending order):

$f_2(n) = \sqrt{2n} < f_3(n) = n + 10 < f_6(n) = n^2 \log n < f_1(n) = n^{2.5} < f_4(n) = 10^n < f_5(n) = 100^n$

which means:

- $f_1(n) = n^{2.5}$ is $O(f_4(n)) = O(10^n)$
- $f_2(n) = \sqrt{2n}$ is $O(f_3(n)) = O(n + 10)$
- $f_3(n) = n + 10$ is $O(f_6(n)) = O(n^2 \log n)$
- $f_4(n) = 10^n$ is $O(f_5(n)) = O(100^n)$
- $f_5(n) = 100^n$ increases the most
- $f_6(n) = n^2 \log n$ is $O(f_1(n)) = O(n^{2.5})$

---

3. (Chapter 2 Problem 6 of the text book) 20 points

Consider the following problem: given an array A consisting of n integers $A[1], A[2], ..., A[n]$, output a two-dimensional n-by-n array B in which $B[i, j]$ (for $i < j$) contains the sum of array entries $A[i]$ through $A[j]$, that is, the sum $A[i] + A[i + 1] + ... + A[j]$. (The value of array entry $B[i, j]$ is unspecified for $i \geq j$, so it does not matter what is output for these values.) Here is a simple algorithm to solve this problem.

```
For i = 1, 2, .., n
    For j = i + 1, i + 2, .., n
        Add up array entries A[i] through A[j]
        Store the result in B[i, j]
    Endfor
Endfor
```

(a) For some function f that you should choose, give a bound of the form $O(f(n))$ on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).

(b) For this same function f, show that the running time of the algorithm on an input of size n is also $\Omega(f(n))$. (This shows an asymptotically tight bound of $\Theta(f(n))$ on the running time.)

(c) Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time $O(g(n))$, where $\lim_{n\to\infty} g(n)/f(n) = 0$.

When $\lim_{n\to\infty} g(n)/f(n) = 0$, $g(n)$ is little-O of $f(n)$, denoted as $g(n) = o(f(n))$.

Answer

(a) Adding the entries of the array $A[i]$ through $A[j]$ would have a running time of $O(n)$ because it can loop through the whole array for $i = n$. The outer loop has a running time of $O(n)$ because it has to loop through the whole of the array and for the worst case the inner loop would have a running time of $O(n)$. This means that the whole algorithm would have a running time complexity of $O(n^3)$.

(b) The **outer loop** in this would take n time steps, in any case.

For $i = 1$, the **inner loop** would execute for $n-1$ times for $i = 2$ it would be $n-2$, this keeps on going until we reach $i = n$ which does not execute at all. This would mean that the inner loop would be executed like this: $(n-1) + (n-2)... + 2 + 1$. This means that all of the +1s have their counterpart -1s which cancel each other, which means that there are $(n-1)/2$ pairs. This means that n is added $(n-1)/2$ times. Which makes the expression be $\frac{n(n-1)}{2}$, these are called the **Triangular Numbers**.

For the **additions** inside the inner loop, any value of i would mean that the array would sum itself from the value of i to n. So, for $i = 1$, using the previous formula $\frac{n(n-1)}{2}$ we get $n^2/2 - n/2$.

for $i = 2$, we get $1 + 2 + ... + n - 2 = (n-1)(n-2)/2$

for $i = 3$, we get $1 + 2 + ... + n - 3 = (n-2)(n-3)/2$

for any $i = t$, we get $1 + 2 + ... + n - t = (n-t)(n-t+1)/2$

for the avg iterations $i = n/2$, we get $1+2+...+n-n/2 = (n-n/2)(n-n/2+1)/2 = (n/2)(n/2+1)/2 >= (n/2)(n/2)/2 = n^2/8$

There are at least $n/2$ iterations for which we have $n^2/8$ iterations each. Which gives us $n^3/16$ bound. Hence the Algorithm is tightly bounded by $O(n^3)$.

(c) It would be better if the algorithm would, for each of the iteration of the outer loop, put the value of $A[i]$ in a variable, *sum*. The inner loop should execute from $i + 1$ as usual and go to n. The sum of the previous values and the current value of $A[j]$ are summed up and put into the variable *sum*, which can be stored into $B[i, j]$. The inner loops contents would take constant time steps. Whereas, the iterations performed by the inner loop would be the same as the previous part, i.e., $\frac{n(n-1)}{2}$. So the running time complexity of this algorithm would be $O(n^2)$. The algorithm is as follows:

```
sum = 0
for i in [1, 2, ..., n]
        sum = A[i]
        For j in [i+1, i+2, ..., n]
                sum = sum + A[j]
                B[i, j] = sum
```

4. (Chapter 3 Problem 2 of the text book)

   Give an algorithm to detect whether a given undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

Answer A cycle in a graph is a special path whose starting and ending vertex are the same. An un-directed graph is a graph in which there is no restriction as to which way we can travel. To detect such paths we can use Depth First Traversal Algorithm which maintains an array of already visited nodes and this list also contains the ancestors of the visited node. A cycle is detected if we traverse the nodes of the graph and we encounter the same node which we just visited, or its ancestors. This edge is called **back edge**. A simple pseudo-code of Depth First Search (DFS) is given below:

Initialize the nodes to unvisited

```
def function detect_cycle(source_node):
        source_node = visited

        for loop over unvisited_nodes
                if current_node != visited
                        parents[current_node] = source_node
                        detect_cycle(current_node)
                else if parents[source_node] != current_node
                        return True
        return
```

The list of unvisited nodes is denoted by *unvisited_nodes*, *current_node* is the node which is being visited at the moment and *parents* is a list containing the parents of the visited nodes. *visited* property can be 1 in the sense of programming languages and 0 can be the initial state of the unvisited nodes.

DFS has a running time complexity of $O(m + n)$ where m are the edges and n is the nodes.

---

5. (Chapter 3 Problem 10 of the text book) 20 points

   In a social network, a shortest path between two nodes is a path connecting the two nodes with the minimum number of edges in the network. An interesting problem is to find the number of shortest paths between two nodes of a social network. Given an undirected graph $G(V, E)$, and two nodes v and w in G, design an algorithm that computes the number of shortest paths between v and w in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

Answer

For this problem we will consider Breadth First Search (BFS) traversal algorithm because each following layer is one edge away, which means that this new node which is one edge away contributes to the shortest path from one node to the other. Given that the graph is an undirected graph, we will use this algorithm. We need to just output a number telling the number of paths which are the shortest to w from v, so we maintain an attribute in every node namely *shortest_path* and *distance*, which maintains the distance covered. We will initialize all of the nodes to have an arbitrarily large number for *distance* attribute, except for the source node v, which will have value 0 because we are already at the source node while starting the traversal. The *shortest_path* values of all nodes hold the value 0, except for source node v which has the value 1 because we are already at the source node. The *neighbour* node is the node which is the child of a node, *pointer* holds the node which the algorithm is at. Then we have a conditional statement which makes sure of the layer condition described above, i.e., $distance[neighbour] = distance[pointer] + 1$. If the other condition is true, i.e., $distance[neighbour] > distance[pointer]+1$ we assign the value of $distance[pointer]+1$ to $distance[neighbour]$, which takes us closer every step. The algorithm runs like this:

```
for dist in distance  // Initializing
        dist = inf
for sp in shortest_path // Initializing
        sp = 0
distance[v] = 0
shortest_path[v] = 1
while(pointer in G)
        for neighbour in G[pointer]
                if distance[neighbour] > distance[pointer]+1
                        distance[neighbour] = distance[pointer]+1
                        shortest_path[neighbour] = shortest_path[pointer]
                else if distance[neighbour] = distance[pointer]+1
                        shortest_path[neighbour] += shortest_path[pointer]
for sp in shortest_path
        print(sp)   // number of shortest paths
```

This algorithm has a running complexity of $O(m+n)$, where m is the number of edges and n is the number of nodes. This is the reason because each node is traversed once, which makes the running time $O(n)$, and we iterate over their neighbor only once, which represents the number of edges, the running time complexity of this action is $O(m)$. Which makes the total time complexity $O(m+n)$.
