1 Short answer (2 questions) - 20 points

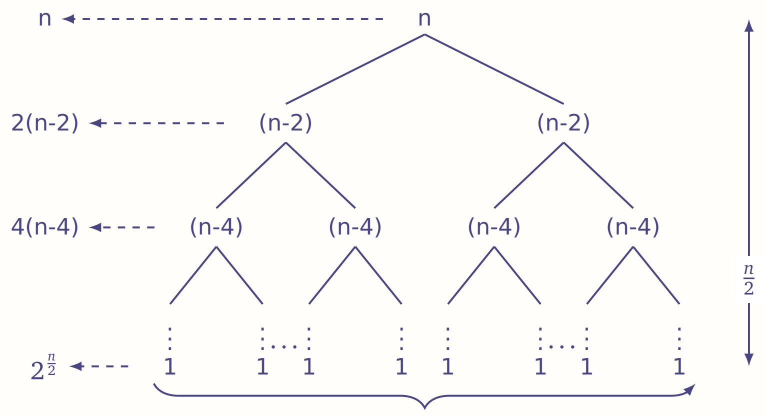
Answer the following questions. You may **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required.

(a) Give a *tight* asymptotic bound for the following recurrences :

(i)

$$A(n) = 2A(n-2) + n \qquad A(0) = A(1) = A(2) = 1$$

Solution: Let's visualize the recursion tree:

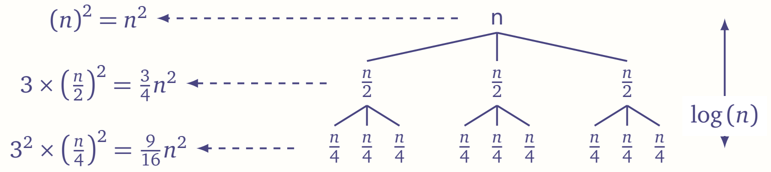


As we can see, the recurrence is leaf dominated. Hence, the asymptotic bound is $O\left(2^{\frac{n}{2}}\right)$. ■

(ii)

$$B(n) = 3B(n/2) + n^2 \qquad B(0) = B(1) = 1$$

Solution: Let's visualize the recursion tree:



$(n)^2 = n^2$
$3 \times \left(\frac{n}{2}\right)^2 = \frac{3}{4}n^2$
$3^2 \times \left(\frac{n}{4}\right)^2 = \frac{9}{16}n^2$
n
$\frac{n}{2}$
$\frac{n}{4}$
$\log(n)$

Recurrence is root dominated. Therefore we can safely say the asymptotic bound of the recurrence is $O\left(n^2\right)$. ■

(b) Give the recurrence that describes the following program. What is the running time?

```
foolishness(n)
sum=0
if n = 1,
return sum = 1
for i from 1 to n^3:
sum += 4*i
return foolishness (n-1)
```

Solution: The above code has two parts: a recursive call to itself to foolishness(n-1) and a for loop that makes n^3 calculations. Therefore, it becomes easy to see that the recurrence would be:

Recurrence: $T(n) = T(n-1) + n^3$

which would make the running time:

Running time: $O\left(n^4\right)$

■

2 Short answer II (5 questions) - 30 points

Answer the following questions. You *may* **briefly** (no more than 2 sentences) justify your answers, but a complete proof is not required. For the following graph problems, use the notation $G = (V, E)$, $n = |V|$ and $m = |E|$

(a) Suppose we implemented a vanilla version of QuickSort where the pivot was always chosen from the first element of the array. Under what input array will the QuickSort algorithm result in a running time of $O\left(n^2\right)$

Solution: If the input array is sorted in either ascending or descending order, then the pivot will always segment the array into one group that has all the elements (besides the pivot) and another group that has none of the elements. ■

(b) In the longest increasing subsequence algorithm (found in the cheat sheet), we defined a recurrence $LIS(i, j)$. **Give an English description (no more than 2 sentences) of what $LIS(i, j)$ represents.** Note: what's in the cheat sheet does not constitute a english description for the recurrence.

Solution: $LIS(i, j)$ is the longest increasing subsequence in the prefix of the input array, $A[1 \ldots i]$ assuming none of the elements in that subsequence are greater than $A[j]$. ■

(c) Given a graph with n vertices and no edges, how many topological sorts will this graph have?

Solution: Since there are no edges, the vertices can be arranged in any order and therefore, there are $n!$ arrangements. ■

(d) You are given a graph $(G = (V, E))$ and two vertices a and b. Write a algorithm that determines if both a and b are part of the same strongly connected component. What is the run-time of this algorithm?

Solution: Two vertices are part of the same strongly connected component if they can reach eachother. Therefore we an simply:

- Use DFS/BFS to see if $b \cap \text{rch}(a) \neq \emptyset$
- Use DFS/BFS to see if $a \cap \text{rch}(b) \neq \emptyset$
- if both statements are true, return true. Otherwise false.

■

(e) In the Bellman-Ford algorithm (found in the cheat sheet), we defined a recurrence $d(v, k)$. **Give an English description (no more than 2 sentences) of what $d(v, k)$ represents.** Note: what's in the cheat sheet does not constitute a english description for the recurrence.

Solution: $d(v, k)$ is the distance from vertex v to all other vertices using at most k edges. ■

3 Recursion - 10 points

Suppose we are given an array $A[1..n]$ of n *distinct integers*, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$. Describe a fast algorithm that either computes an index i such that $A[i] = n - i + 1$ or correctly reports that no such index exists.

(You *cannot* simply reference a prior lab/homework. You must give the actual solution.)

Solution: This is very similar to one of the first lab problems in the algorithms portion of the course. The solution has been adapted from that lab below:

Suppose we define a second array $B[1..n]$ by setting $B[i] = n - A[i] - i + 1$ for all i. For every index i we have

$$B[i] = n - A[i] - i + 1 \geq (n - A[i+1] - 1 + 1) - i = n - A[i+1] - (i+1) + 1 = B[i+1],$$

so this new array is sorted in decreasing order. Clearly, $A[i] = n - i + 1$ if and only if $B[i] = 0$. So we can find an index i such that $A[i] = n - i + 1$ by performing a binary search in B. We don't actually need to compute B in advance; instead, whenever the binary search needs to access some value $B[i]$, we can just compute $A[i] = n - i + 1$ on the fly instead!

Here are two formulations of the resulting algorithm, first recursive (keeping the array A as a global variable), and second iterative.

```
⟨⟨Return any index $i$ such that $\ell \le i \le r$ and $A[i] = n - i + 1$⟩⟩
FindMatch($\ell, r, n$):
  if $\ell > r$
      return None
  $mid \leftarrow (\ell + r)/2$
  if $A[mid] = n - mid + 1$            ⟨⟨$B[mid] = 0$⟩⟩
      return $mid$
  else if $A[mid] < n - mid + 1$       ⟨⟨$B[mid] < 0$⟩⟩
      return FindMatch($mid + 1, r, n$)
  else                                 ⟨⟨$B[mid] > 0$⟩⟩
      return FindMatch($\ell, mid - 1, n$)
```

```
FindMatch($A[1..n]$):
  $hi \leftarrow n$
  $lo \leftarrow 1$
  while $lo \le hi$
      $mid \leftarrow (lo + hi)/2$
      if $A[mid] = n - mid + 1$            ⟨⟨$B[mid] = 0$⟩⟩
          return $mid$
      else if $A[mid] < n - mid + 1$       ⟨⟨$B[mid] < 0$⟩⟩
          $lo \leftarrow mid + 1$
      else                                 ⟨⟨$B[mid] > 0$⟩⟩
          $hi \leftarrow mid - 1$
  return None
```

■

4 Finding outliers - 10 points

The ECE374 staff has just completed grading midterm 2. They have a list of S[1...n] of unsorted exam scores and are searching for outliers on either end. The interquartile range is defined by Q3 – Q1, that is the 75th percentile score – 25th percentile score. Specifically, an outlier is someone who scores $> Q3 + 1.5 * IQR$ (interquartile range) or someone who scores $< Q1 - 1.5 * IQR$. Give an efficient O(n) program to (1) compute the interquartile range, and (2) output all the outlier scores. A solution that uses hashing or sorting will be awarded 0 points.

Solution: This problem requires a understanding of the linear-time-selection (quick-select + median-of-medians) algorithm ($MoM(A, r)$). Once you understand this, the calculation becomes straightforward:

- **Interquartile range** - $IQR = MoM(A, 0.75) - MoM(A, 0.25)$. Running time: $O(n)$
- **Output outliers scores**:

```
$Q3 = MoM(A, 0.75)$
$Q1 = MoM(A, 0.25)$
$IQR = Q3 - Q1$

for $i \in$ scores
  if $i > Q3 + 1.5 * IQR$
      output $i$
  if $i < Q1 - 1.5 * IQR$
      output $i$
```

Running time is $O(n)$

■

5 Dynamic programming (1 questions) - 15

You are given an integer value x and an array A where each element of the array represents a coin denomination. Describe a dynamic programming algorithm that returns the number of ways to make change for x.

Example: $A = [1, 2, 3]$ and $x = 5$. Output is 5 ($\{1, 1, 1, 1, 1\}, \{1, 1, 1, 2\}, \{1, 1, 3\}, \{1, 2, 2\}, \{2, 3\}$).

Solution: This was also a lab (13) problems (first one). The solution below is taken from that lab:

Let $CC(i, j)$ denote the number of different ways to make change for j using the first i types of coins($A[1..i]$).

Observe that you can categorize the ways to make change for j using $A[1..i]$ into two mutually exclusive cases: either by including at least one of $A[i]$ or not including any $A[i]$ at all. The number of different ways to make change for j with $A[1..i]$ while having at least one $A[i]$ is equal to the number of different ways to make change for $j - A[i]$ with $A[1..i]$, since adding one $A[i]$ to $j - A[i]$ would give j while guaranteeing that there is at least one $A[i]$ in the change. The number of different ways to make change for j with $A[1..i]$ while not using any $A[i]$ is equal to the number of different ways to make change for j with $A[1..i-1]$, since you are not using $A[i]$ anyways.

Based on the observation, we obtain the following recurrence.

Recurrence and short English description(in terms of the parameters):

$$CC(i, j) = \begin{cases} 1 & \text{if } j = 0 \\ 0 & \text{if } j < 0 \text{ or } (j > 0 \text{ and } i = 0) \\ CC(i, j - A[i]) + CC(i - 1, j) & \text{otherwise} \end{cases}$$

The first base case is for $j = 0$ and it would be 1 since we always have exactly one way to make change for 0. The second base case corresponds to either making negative change, or making positive change while not using any coin, which would be 0. The recursive case comes from the observation above.

Memoization data structure and evaluation order:

To formulate a dynamic programming solution, we construct a 2D array $CC[i, j]$ of size $(n + 1) \times (x + 1)$, where n is the size of the array A, and x is the target amount of change.

Return value:

We evaluate both i and j in increasing order, and return $CC[n, x]$ at the end.

Time Complexity:

Since we are evaluating an array of size $O(nx)$ and the evaluation of a single subproblem takes $O(1)$, the runtime of the algorithm is $O(nx)$.

■

6 Graph algorithm - 15 points

You are given two weighted, directed acyclic graphs ($G = (V, E_G)$ and $H = (V, E_H)$) that have the same vertices but different edges. Every edge is assigned an integer value that could be positive or negative.

You are given two nodes s and t. The length of a path in each graph is defined as $\ell_G(s,t)$ for G and $\ell_H(s,t)$ for H. You need to find the path that exists in both G and H with the minimum combined length (in other words, find minimum value of $\ell_G(s,t) + \ell_H(s,t)$). (Hint: topological sort is your friend)

(You *cannot* simply reference a prior lab/homework. You must give the actual solution.)

Note: These details don't necessarily matter but might be useful to some. Both graphs are given as adjacency matrices where 0 indicates no edge and 1 indicates an edge. The weights are given by a weight matrix $w_G[i,j]$ or $w_H[i,j]$ for G and H respectively. Remember pseudo-code is one of the worst ways to describe your algorithm(s); be clear and succinct.

Solution: As discussed in lecture, we can compute the longest/shortest path of a DAG in $O(n)$ time by formulating this problem as a dynamic programming problem. All we need to do is have a extra condition that makes sure the path exists in both graphs and calculates the path lengths accordingly.

Let's get a topological sort of the vertices in G and label this ordering as TS_G. Let's also initialize the combined distance to all the other vertices as $D_{GH}[1 \dots n] \rightarrow \infty$ We can formulate a recurrence as follows:

$$\begin{aligned} D_{GH}(v) &= min(D_{GH}(u) + \ell_G(u,v) + \ell_H(u,v)) && v \in V \\ & && (*,v) \in E_G \\ & && (*,v) \in E_H \\ D_{GH}(s) &= 0 \end{aligned}$$

As long as we evaluate the recurrence in the order of the topological sort, we will get the minimum combined length path that appears in both G and H. The minimum to vertex t is $D_{GH}(t)$.

To evaluate the recurrence, we need to look at every edge and vertex exactly once. Hence, the algorithm takes $O(V+E)$ time. ■