Name ____

CSCI 332, Fall 2024

Exam 2—Practice 1

Note that this exam has four sections. The first section covers algorithm analysis (20 points), the second section covers greedy algorithms (20 points), the third section covers divide and conquer algorithms (20 points), and the fourth section covers dynamic programming algorithms (20 points). If you need more space, develop your solution on scratch paper before copying your final answer to the exam paper.

Good luck!

Section 1 (Algorithm Analysis)

1. (5 points) Take the following list of functions and arrange them in ascending order of growth rate. That is, if function $g(n)$ follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

   - $f_1(n) = 4^{n+1}$ (4 to the power of $n+1$)
   - $f_2(n) = \sqrt{5n}$ (square root of $5n$)
   - $f_3(n) = n!$ (n factorial)
   - $f_4(n) = n \log_2 n$ (n times log base two of n)
   - $f_5(n) = \log_2 2^n$ (log base two of 2 to the n)
   - $f_6(n) = n^2$ (n squared)

2. (5 points) Suppose you know that an algorithm has a best-case runtime that is $\Theta(n^2)$. For each of the following, decide whether it is definitely true, definitely false, or could be true or false and circle your choice.

   - The algorithm's worst-case runtime is $\Omega(n \log n)$. T, F, T or F
   - The algorithm's worst-case runtime is $\Omega(n^2)$. T, F, T or F
   - The algorithm's worst-case runtime is $O(n^2)$. T, F, T or F
   - The algorithm's worst-case runtime is $O(n \log n)$. T, F, T or F
   - The algorithm's best-case runtime is $O(n^3)$. T, F, T or F

Here is a the algorithm we saw in class to compute heaviest weight of a weighted interval scheduling problem using dynamic programming. Let $p(j)$ be the latest-finishing interval compatible with interval j that also ends before j.

```
Schedule($s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, w_1, w_2, \ldots, w_n$):
  Sort $f_1, f_2, \ldots f_n$ and reorder $s$, $f$ and $w$ values accordingly
  Compute $p(1), p(2), \ldots p(n)$
  M[0] = 0
  Compute_OPT($n$)
  Return M[$n$]
```

```
Compute_OPT($j$):
  If M[$j$] is uninitialized:
      M[$j$] = max{OPT($j - 1$), $w_j$ + OPT($p(j)$))
  Else:
      Return M[$j$]
```

You know that Compute_OPT(n) takes $O(n)$ time on any input of size n and that it takes $O(n \log n)$ to sort any set of n jobs by finish time. But you're unsure of how quickly you can compute the $p(j)$ values.

Your friend proposes the following algorithm to compute $p(j)$ values, assuming that the intervals are already sorted by finish time.

```
Compute_Ps($s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n$):
For $j = 1, 2, \ldots, n$:
  Let $i = 1$
  While $f_i \leq s_j$:
      $i = i + 1$
$p(j) = i$
```

3. (3 points) We would like to analyze the runtime of Compute_Ps. Describe an input of size n that would have worst-case runtime.

4. (7 points) Give a function $f(n)$ such that the worst-case runtime of Compute_Ps is $\Theta(f(n))$.

Section 2 (Greedy Algorithms)

Suppose that you will drive your car for a long trip between New York City and San Francisco along a pre-specified path. In preparation for your trip, you have downloaded a map that contains the distances in miles between all the gas stations in your route. Assume that your car's gas tank, when full, holds enough gas to travel n miles. Assume that the value n is given, and that you want to make the minimum number of stops possible along the way, without running out of gas at any point. Your friend proposes a greedy algorithm for selecting which gas stations to stop at:

- Start your trip with a full tank.
- Check your map to determine the farthest away gas station in your route within n miles.
- at that gas station, fill up your tank and check your map again to determine the farthest away gas station in your route within n miles from this stop.
- Repeat the process until you get to San Francisco.

Let $s_g(j)$ denote the station where we make the j^{th} stop for the greedy algorithm. For example, if $s_g(2) = 7$ it means that we make the 2nd stop at the 7th gas station. Let $s_O(j)$ be the index of the gas station for the j^{th} stop for an optimal solution to the problem.

We know that if we can prove that the greedy algorithm "stays ahead" of the optimal, then we can prove that the greedy algorithm is, in fact, optimal.

5. (2 points) In words, what would it mean for the greedy algorithm to stay ahead of the optimal solution for this problem? (Your answer should probably use the phrase "gas station" at least once.)

6. (18 points) Fill in the blanks.

We prove that that the greedy algorithm stays ahead using induction. Suppose that the greedy algorithm uses k stops and the optimal solution uses m stops. By definition $k \leq m$.

Claim 1. *For all $r \leq k$, we have* _____________.

Proof: Let $r \leq k$.

Assume that ____.
(This is the inductive hypothesis, IH.)

There are two cases:

If $r = 1$, we know that ______________________________ because the greedy algorithm chooses the farthest possible gas station for the first stop, so the optimal solution cannot use a closer gas station.

If $r > 1$, we know by the IH that ______________________________. We want to show that the next gas station chosen by the greedy algorithm must be farther along than the next gas station in the optimal solution. By definition, the greedy algorithm picks the farthest gas station within n miles of $s_g(r-1)$. Since $s_O(r-1) \leq s_g(r-1)$, it must be that ______________________________.

Because the claim is true in all cases, it holds for all $r \leq k$. $\square$

Section 3 (Divide and Conquer)

7. Your friend says that they know how to solve the significant inversions problem in $O(n \log n)$ time. Just as in the mergesort-based algorithm for counting regular inversions, they count inversions across A and B during the merge step of mergesort. But when merging an element b_j from B, they only count the inversions from the remaining elements in A if $2b_j < a_i$. So in the below example, merging the $b_1 = 4$ would count for 3 inversions since $2b_j = 2 \cdot 4 = 8 < a_2 = 9$

	a_2		
1	9	15	25

Left half, A

b_1			
4	8	11	12

Right half, B

1							

However, when $b_2 = 8$ is merged into the sorted array, since $2b_2 = 2 \cdot 8 = 16 \not< a_2 = 9$, 0 inversions would be counted.

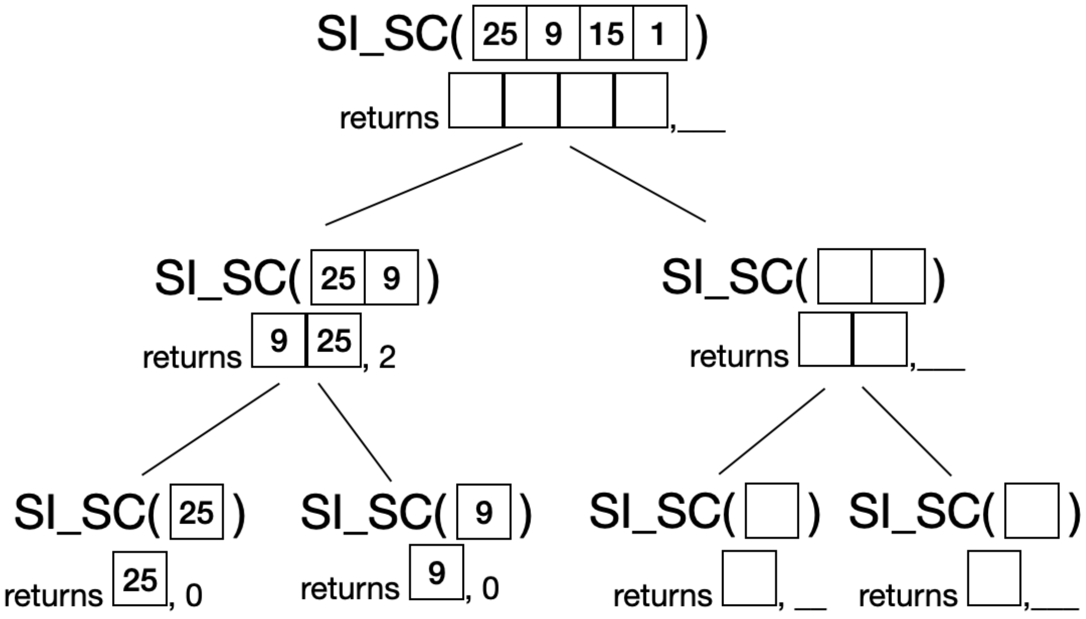
8. (2 points) How many significant inversions does the above algorithm count between A and B?

9. (3 points) What is the true number of significant inversions between A and B? (Or equivalently, in the full list 1, 2, 4, 25, 4, 8, 11, 12?)

In the next problem, you will show that you understand the execution of a recursive algorithm. Here is the pseudocode for a correct algorithm to count significant inversions, called SI_SC (for significant inversion sort and count). Assume that the third to last and second to last lines correctly merge and count inversions between the two lists. (The focus here is on showing that you understand recursive algorithms, not on the merging specifically.)

```
SI_SC(unsorted list $L$):
  If the length of $L$ is 1:
      Return $L$ and 0
  Else:
      Divide $L$ into $L_1$ (first half) and $L_2$ (second half)
      $L_1'$ and count1 = SI_SC($L_1$)
      $L_2'$ and count2 = SI_SC($L_2$)
      Let $L'$ be the sorted combination of $L_1'$ and $L_2'$
      Let count3 be the number of significant inversions between $L_1'$ and $L_2'$
      Return $L'$, count1 + count2 + count3
```

10. (7 points) In the diagram below, fill in both the inputs to the three recursive calls to SI_SC on the bottom right and the four missing return values. Make sure you fill in both the returned list and the returned inversion count.



Consider the following algorithm.

```
Alg1(list $L$):
  If length of $L$ is 1:
      done
  Else:
      $L'$ = new list consisting of every third element of $L$
      Alg1($L'$)
```

11. (4 points) Draw the recursion tree for this algorithm.

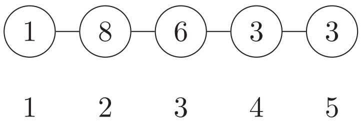
12. (4 points) What is the recurrence relation for the runtime of this algorithm on a list of length n?

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \underline{\hspace{6cm}} & \text{if } n > 1. \end{cases}$$

Section 4 (Dynamic Programming)

In class, we studied the *heaviest independent set on a path* problem, where we were given a path $v_1, v_2, \ldots, v_n$ where v_i is the weight of the node, and we wanted to find the heaviest independent set in the path.

For example, we could have the five-node path below. The weights are the numbers drawn in the nodes. The heaviest independent set below has weight 11 and uses nodes 2 and 4 (or 5).



In this problem, we will consider a variation of the problem where we want the heaviest *squared* weight of an independent set. For example, the heaviest squared independent set above is still 2 and 4 for a total squared weight of $8^2 + 3^2 = 64 + 9 = 73$.

13. (4 points) Give an input where the independent set with the heaviest squared weight is different from the independent set with the regular heaviest weight.

14. (6 points) Give a recurrence relation for the weight of the heaviest *squared* independent set.

$$OPT(j) = \begin{cases} \underline{\hspace{6cm}} & \text{if } j = 0 \\ \underline{\hspace{6cm}} & \text{if } j = j \\ \underline{\hspace{6cm}} & \text{if } j > 1. \end{cases}$$

15. (7 points) Give a polynomial time, recursive algorithm to compute the weight of the heaviest squared independent set using your recurrence relation above. You should fill in the following two functions so that Heaviest_Set_SqWeight returns the heaviest possible squared weight of any independent set in the input.

Heaviest_Set_SqWeight($v_1, v_2, \ldots, v_n$):

Compute_OPT(j):

16. (3 points) Fill in the rest of a recursive algorithm to compute the optimal set of nodes to choose, given a full set of OPT values for $j = 0$ through $j = n$.

Nodes(j):