Homework 5

Professor Rachit Garg
Name: Francis Olakangil

Due October 13 (11:59 p.m.)

Instructions

- **Answers should be written in the boxes by modifying the provided Latex template or some other method answers are clearly marked and legible. Submit via Gradescope.**
- **Honors questions are optional. They will not count towards your grade in the course. However you are encouraged to submit your solutions to these problems to receive feedback on your attempts.**
- **You must enter the names of your collaborators or other sources (including use of LLMs) as a response to Question 0. Do NOT leave this blank; if you worked on the homework entirely on your own, please write "None" here. Even though collaborations in groups of up to 3 people are encouraged, you are required to write your own solution.**
- **For the late submission policy, see the website.**
- Acknowledgment - Thank you to Amir, Daji, Oded, Peter and Rotem for help in setting the problem set.

1-0 List all your collaborators and sources: ($-\infty$ points if left blank)

tutor, textbook

Problem 5-1 – Counting Sort (20 points)

Recall that in Counting (or Small-Key Sort) sort, we append the current element $A[i]$ to the end of the list L_{k_i} where $k_i \in \{1, \ldots, k\}$ is the key associated with $A[i]$. Consider instead adding $A[i]$ to the beginning of the list (the rest of the algorithm is unchanged and outputs the concatenated final lists $L_1, \ldots, L_k$). What property of Counting sort is broken?

Solution: Stability in Counting Sort is broken. If we were to append $A[i]$ to the beginning of the list and each bucket, the relative order of both those elements in the original list and their respective buckets would switch up (assuming that there are multiple elements of the same key). For example, take array $A = 1, 2, 3, 4, 5, 6, 6, 7, 7, 8, 8$ with matching bucket-indices of $A, A, A, A, A, B, A, B, A, B$. The As in this array represent a unique key and the Bs represent a second copy of the element at that key. When running the specialized counting sort algorithm, the expectation is that there will be 8 buckets for A with buckets 6, 7, and 8 having both $\mathcal{A}[A]$ and $\mathcal{A}[B]$ in them. The problem is that if we append these elements to the beginning of the list, that would mean that for these buckets (at least), $\mathcal{A}[B]$ would come before $\mathcal{A}[A]$, disrupting the relative order that the array should be in when re-concatenating.

Problem 5-2 – Sorting Efficiently (30 Points)

1. (15 points) Suppose you have an array A of length 2^{16}, where each entry is a 128-bit integer. For each of *Radix Sort*, *Counting Sort*, *Merge Sort*, state the optimal runtime for each algorithm running on A (Note that for radix sort, this means choosing the optimal base – please state this concretely.) Write the runtimes as an approximate number of steps — you may assume Merge Sort runs in time $n \log n$ (i.e., no constants), and Counting Sort runs in $2n + 2k$ steps. Finally, state which of these algorithms is most efficient for sorting A.

Solution:

$\rightarrow k = \log_2(2^{128}) = 128$
$\rightarrow n = 2^{16}, k = 128$ keys = 128 b/c num of possible unique keys = num of digits
$\rightarrow 2(2^{16}) + 2(128)$ Counting sort
$\rightarrow 131,072 + 156 = 131,228$ Final Counting sort runtime
$\rightarrow (2^{16}) \log(2^{16})$ Merge sort
$\rightarrow 65,536((16) \log_2(2))$
$\rightarrow 65,536 \cdot 16 = 1,048,576$ Final Merge sort runtime
$\rightarrow (2^{16} + 2)(\log_2(2^{128}) = 8,388,864$ Radix Sort with base=2 but let's optimize the base
$\rightarrow (2^{16} + b)(\log_b(2^{128})$
$\rightarrow$ C Code Here Click the link here to see code that minimizes the possible bases
$\rightarrow b = 8182, 725937.17$ Final Radix sort runtime

Radix Sort is the most efficient for sorting A.

2. (15 points) Assume you are given an array of n integers with many duplications, so that you know that there are at most $\log n$ distinct elements in the array. Show how to sort this array in time $O(n \operatorname{loglog} n)$.

Solution: We can store the number of counts in a comparison-balanced search tree that is keyed by each given value. Processing each of those elements should only take $n \log \log n$ time. Outputting the sorted result and writing each repeated key results in linear time.

So we would sort this by using an empty, balanced binary search tree. For each elem of the array, increment the count of that element if it's alrdy in the tree, otherwise insert it like in a frequency dictionary. Each lookup/insertion will only cost $\log n$. Afterwards, do an ino-order traversal, returning keys in sorted order where k is outputted the number of times that it was counted.

Problem 5-3 – Pumpkin Pickin' (50 points)

You work at a pumpkin patch and you are tasked with harvesting the most pounds of pumpkin without ruining the aesthetic of the pumpkin vine. In particular, you are not to harvest two *adjacent* pumpkins. The vine of n pumpkins is represented by an array A where $A[i]$ is the weight of pumpkin i (in pounds). Use dynamic programming to give an efficient algorithm determining the maximum total weight of pumpkin you can harvest while adhering to the rule of not harvesting two in a row.

1. (8 points) Define the subproblems for the dynamic programming algorithm. How many subproblems are there?

Solution: Defining the subproblem: Pumpkin-vine[i] = max total weight using first i pumpkins without picking two adjacent ones. There are $O(n)$ subproblems.

2. (8 points) Write out the base cases (there should be two of them!).

Solution: $P[1] = A[1]$
$P[2] = \max(A[1], A[2])$

3. (10 points) Give and justify the recurrence your subproblems should satisfy.

Solution: $P[i] = \max(P[i-1], P[i-1] + A[i])$
Justification: There are only two possibilities for an optimal solution on the first i pumpkin. You either don't pick the current pumpkin because it's adjacent (in which case you take the most optimal path for $i-1$ pumpkins) or you do pick the current pumpkin to harvest. But that means that you can't pick pumpkin $i-1$, thus the recurrence. The best choice for the current $P[i]$ is the better of these two options, which the max() function accounts for.

4. (10 points) Using this recurrence, give your algorithm in pseudocode. The return value of the algorithm should be the maximum pounds of pumpkin.

Solution:

```
$\rightarrow n = len(A)$
$\rightarrow ifn == 0 : return0$
$\rightarrow ifn == 1 : returnA[1]$
$\rightarrow A[1] \rightarrow P[1]$
$\rightarrow \max(A[1], A[2]) \rightarrow P[2]$    Base cases
$\rightarrow$ for i $= 3$ to $n$ : $P[i] \rightarrow max(P[i-1], P[i-2] + A[i])$, return $P[n]$
```

5. (8 points) State and justify the runtime of your algorithm.

Solution: $O(n)$. The algo computes one of the subproblems $P[i]$ for each of the n pumpkins. Each computation of $P[i]$ takes constant because it's the same operations, just the max function and addition. So, the total time spent will be proportional to the number of pumpkins, which will be $O(n)$.

6. (6 points) So far, the algorithm that you developed only computes the *pounds* of the pumpkin that can be harvested, but it does not tell you *which pumpkins* to pick.

   Explain how to modify the algorithm so that at the end, we can also produce the list of pumpkins to harvest. You may want to store additional information in the DP table as it is filled out.

   (**Hint:** think about what we did in the LCS problem to extract the string after filling out the table.)

   Solution: So we're assuming that we already ahve the array $P[i]$ that stores the maximum total weight from the pumpkins. I'd modify the algo to produce the list of pumpkins to harvest by tracing back which pumpkins were chosen to add to the final maximum that is returned. In the algo, there are two choices, $P[i] = P[i-1]$, where the pumpkin isn't taken, and $P[i] = P[i-2] + A[i]$, where the pumpkin is harveested. Basically, we can either record this choice during the computation or reconstruct it by walking backward from $i = n$. For the purposes of the explanation, I would store the index of the pumpkin chosen during the computation itself. It should still take $O(n)$ time and you would essentially be storing the indices of the cases where you calculate $P[i] = P[i-2] + A[i]$ to record where in the vine you harvested.

Honors Problem – Optimal Substructure for Rod Cutting (0 Points, *)

Suppose a profit-maximizing way to cut a rod of length n is given by a sequence of piece lengths

$$x_1, x_2, \ldots, x_k,$$

where each x_i is a positive integer and $x_1 + x_2 + \cdots + x_k = n$.

1. (5 points) Fill in the blank: Then $x_2, \ldots, x_k$ is __________________________.
   (**Hint:** Think about the remaining rod of length $n - x_1$.)

   Solution:

2. (20 points) Prove your claim from part (1). (**Hint:** Try proof by contradiction.)

Solution:

Honors Problem – little-o (0 Points, *)

We present another asymptotic notation that is used in the analysis of algorithms: For functions f, g, we say $f = o(g)$ ("f is little-o of g") if and only if the ratio $\frac{f(n)}{g(n)}$ goes to zero as n tends to infinity. For example, $\log n = o(n)$ but $n \neq o(n)$.

1. Is $\sqrt{n} = o(n)$? Prove that it is or not.

Solution:

Homework 5

2. Is it possible that $f = o(g)$ and $g = O(f)$? If yes, give an example. Otherwise prove that it is not possible.

Solution:

3. Show that the worst case number of comparisons needed to merge two sorted lists, each of size n, is at least $2n - o(n)$.
(Hint: How many ways can we write down $2n$ distinct numbers as two sorted lists of n numbers each? It may also be useful to look up the **central binomial coefficient (click here)** for its asymptotic growth.)

Solution: