Homework 4

CS 5114 (Fall)

Assigned on Monday, October 1, 2018.
Submit PDF solutions on Canvas
by the beginning of class on October 8, 2018.

Instructions:

- The Graduate Honor Code applies this to homework. In particular, you are not allowed to consult any sources other than your textbook, the slides on the course web page, your own class notes, and the instructor. Do not use a search engine.
- Do not forget to typeset your solutions. *Every mathematical expression must be typeset as a mathematical expression, e.g., the square of n must appear as n^2 and not as "n^2".* You can use the LaTeX version of the homework problems to start entering your solutions.
- Describe your algorithms as clearly as possible. The style used in the book is fine, as long as your description is not ambiguous. Explain your algorithm in words. A step-wise description is fine. *However, if you submit detailed code or pseudo-code without an explanation, we will not grade your solutions.*
- Do not make any assumptions not stated in the problem. If you do make any assumptions, state them clearly, and explain why the assumption does not decrease the generality of your solution.
- Do not describe your algorithms only for a specific example you may have worked out.
- You must also provide a clear proof that your solution is correct (or a counter-example, where applicable). Type out all the statements you need to complete your proof. *You must convince us that you can write out the complete proof. You will lose points if you work out some details of the proof in your head but do not type them out in your solution.*
- Describe an analysis of your algorithm and state and prove the running time. You will only get partial credit if your analysis is not tight, i.e., if the bound you prove for your algorithm is not the best upper bound possible.
- In general for a graph problem, you may assume that the graph is stored in an adjacency list and that the input size is $m + n$, where n is the number of nodes and m is the number of edges in the graph. Therefore, a linear time graph algorithm will run in $O(m + n)$ time.

Problem 1 (15 points) In this problem, you will analyse the worst-case running time of weighted interval scheduling without memoisation. Recall that we sorted the n jobs in increasing order of finish time and renumbered these jobs in this order, so that $f_i \leq f_{i+1}$, for all $1 \leq i < n$, where f_i is the finish time of job i. For every job j, we defined $p(j)$ to be the job with the largest index that finishes earlier than job j. Consider the input in Figure 6.4 on page 256 of your textbook. Here all jobs have weight 1 and $p(j) = j - 2$, for all $3 \leq j \leq n$ and $p(1) = p(2) = 0$. Let $T(n)$ be the running time of the dynamic programming algorithm *without memoisation* for this particular input. As we discussed in class, we can write down the following recurrence:

$$T(n) = T(n-1) + T(n-2), n > 2$$
$$T(2) = T(1) = 1$$

Prove an *exponential lower* bound on $T(n)$. Specifically, prove that $T(n) \geq 1.5^{n-2}$, for all $n \geq 1$.

Problem 2 (25 points) Given a sorted array of distinct integers $A[1, \dots n]$, you want to find out if there is an index i such that $A[i] = i$. Clearly, an algorithm with an $O(n)$ running time is simple: simply iterate over each index k in A and check if it satisfies $A[k] = k$. It seems that it should be possible to develop a divide and conquer algorithm modelled on binary search to solve this problem with a better running time than $O(n)$, specifically, $O(\log n)$. Here is a recursive algorithm `FindIndex`, which takes two arguments l and r where $l \leq r$. The goal of this algorithm is to check if there is an index i between l and r such that $A[i] = i$. To solve the problem, we will initially invoke `FindIndex` with the arguments 1 and n.

```
FindIndex($l, r$)

1. If $r < l$ return False.
2. Set $k$ to $\lfloor (l + r)/2 \rfloor$.
3. If $A[k] = k$, return $k$.
4. If $r = l$ return False.
5. If $A[k] < k$, return the result of FindIndex($k + 1, r$).
6. If $A[k] > k$, return the result of FindIndex($l, k - 1$).
```

It is quite clear that the running time of `FindIndex`$(1, n)$ is $O(\log n)$ time since the algorithm spends a constant amount of time before making a single recursive call to an input of half the size.

You must prove its correctness. The proof requires some very careful case analysis, especially to show that your algorithm is correct when it returns that there is no such index. Here are some points for you to consider:

1. If $A[k] < k$, then for each index i, $l \leq i \leq k$, prove that $A[i] < i$, i.e., it is not possible for the desired index to lie between the indices l and k (including both indices). If you prove this statement, you can conclude that the recursive call `FindIndex`$(k+1, r)$ will indeed find the correct index, if it exists.
2. If $A[k] > k$, then for each index index i, $k \leq i \leq r$, prove that $A[i] > i$, i.e., it is not possible for the desired index to lie between the indices k and r (including both indices). Hence, the recursive call `FindIndex`$(l, k - 1)$ will indeed find the correct index, if it exists.
3. There is one more important step. If the algorithm returns `False`, how can we convince ourselves that there is no index i such that $A[i] = i$? After all, the algorithm examined only $\log n$ indices and not all n indices. To answer this question, consider the invocation of `FindIndex` with arguments l and r. State a property that holds for all indices less than l and an analogous property that holds for all integers larger than r. Combine the results of the if statements in the algorithm with these properties to prove that they hold for each recursive call as well. Finally, complete the proof of correctness of the algorithm by applying these properties to the two steps when the algorithm returns `False`.

Problem 3 (25 points) Solve exercise 1 in Chapter 6 (pages 312–313) of your textbook.

Problem 4 (35 points) Many object-oriented programming language implement a class for manipulating strings. A primitive operation supported by such languages is to split a string into two pieces. This operation usually involves copying the original string. Hence, it takes n units of time to split a string of length n into two pieces, *regardless of the location of the split.* However, if we want to split a string into many pieces, the order in which we make the splits can affect the total running time of all the splits.

For example, suppose we want to split a 20-character string at positions 3 and 10. If we make the first cut at position 3, the cost of the first cut is the length of the string, which is 20. Now the cut at position 10 falls within the second string, whose length is 17, so the cost of the second cut is 17. Therefore, the total cost is 20 + 17 = 37. Instead, if we make the first cut at position 10, the cost of this cut is still 20. However, the second cut at position 3 falls within the first string, which has length 10. Therefore, the cost of the second cut is 10, implying a total cost of 20 + 10 = 30.

Design an algorithm that, given the locations of m cuts in a string of length n, finds the minimum total cost of breaking the string into $m + 1$ pieces at the given locations, minimised over all possible ways of breaking the string at the m locations.

Let us define some notation to help develop the solution. Sort the locations of the m cuts in increasing order along the length of the string. Let c_i be the location of the ith cut, $1 \leq i \leq m$. Set $c_0 = 1$, and $c_{m+1} = n$, locations at the beginning and the end of the string, respectively. Note that we can assume, without loss of generality, that no cuts have been specified at locations 1 and n, since making these cuts incurs no cost.

Hint: Suppose you make the first cut at some location c_j, where $1 \leq j \leq m$. This cut will split the string into two sub-strings. Consider where you may make the next cut in each sub-string. What types of sub-problems are you creating? In other words, each sub-problem is a sub-string: how do you denote or characterise this sub-string? Does it look like a sub-problem in weighted interval scheduling, segmented least squares, or RNA secondary structure? Figuring out the right set of sub-problems will go a long way towards helping you solve the problem. It will also help to define some notation for each sub-problem and for the cost of solving it.