Homework 5

CS 4104 (Spring 2021)

Assigned on March 29, 2021.
Submit PDF solutions on Canvas by the
11:59pm on April 7, 2021.

My team-mate is: Joseph McAlister

Problem 1 (20 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 $n \geq 1$.

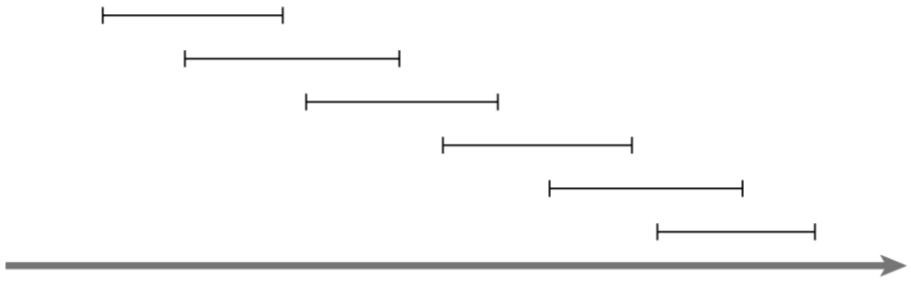

Figure 6.4 An instance of weighted interval scheduling on which the simple `Compute-Opt` recursion will take exponential time. The values of all intervals in this instance are 1.

Solution: We begin by "unrolling" the recurrence into an assumed exponential lower bound:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq c^{n-1} + c^{n-2} \\ &\leq c^n \end{aligned}$$

From here we can substitute 1.5 for c. We must show that $\frac{T(n)}{1.5^{n-2}} > 1$ for all $n \geq 1$.

The base case of $n = 1$ is trivial:

$$\frac{T(1)}{1.5^{1-2}} = \frac{1}{1.5^{-1}} \geq 1$$
$$= 1.5 \geq 1$$

For the inductive hypothesis and step, we return to the $\geq c^n$ term.

Inductive Hypothesis: the lower bound is satisfied if $T(n) \geq c^n$.

Inductive Step: This statement is true if $c^n \geq c^{n-1}+c^{n-2}$. Algebraically, we can rephrase this statement as $c^2 - c - 1 \geq 0$ which is true for all values greater than or equal to the golden ratio: $g = \frac{1+\sqrt{5}}{2}$ (the smallest value of c which satisfies the statement is g).

That is, for $c = 1.5, c \leq g$, 1.5^{n-2} is a lower bound for $T(n)$.

Problem 2 (35 points) Solve exercise 1 in Chapter 6 (pages 312–313) of your textbook.

Let $G = (V, E)$ be an undirected graph with n nodes. Recall that a subset of nodes is called an *independent set* if no two of them are joined by an edge. Finding large independent subsets is difficult in general; but here we'll see that it can be done efficiently if the graph is "simple" enough.

Call a graph $G = (V, E)$ a *path* if its nodes can be written as $v_1, v_2, ..., v_n$, with an edge between v_i and v_j if and only if then numbers i, j differ by exactly 1. With each node v_i, we associate a positive integer *weight* w_i.

Consider, for example, the five-node path drawn in Figure 6.28. The *weights* are the numbers drawn inside the nodes. The goal in question is to solve the following problem:

Find an independent set in a path G whose weight is as large as possible

1. Give an example to show that the following algorithm ("heaviest first") *does not* always find an independent set of maximum total weight.

   Solution: An input of nodes with weights $[3, 5, 3]$ would make this algorithm fail as the heaviest independent set would have a weight of 6 (taken from the first and third nodes), whereas the algorithm would select 5 and terminate.

2. Give an example to show that the following algorithm ("even odd") also *does not* always find an independent set of maximum total weight.

   Solution: An input of nodes with weights $[5, 1, 3, 5]$ would make this algorithm fail as the heaviest independent set would have a weight of 10 (taken from the first and last nodes), whereas the algorithm would select the "odd" nodes of weights $5 + 3 = 8$ and terminate.

3. Give an algorithm that takes an n-node path G with weights and returns an independent set of maximum total weight. The running time should be polynomial in n independent of the values of the weights.

   Solution: The strategy is to select nodes by weight, recursively choosing the maximum between the previous node and the current node as well as the node before the previous node:

$$OPT(i) = \max(OPT(i-1), w_i + OPT(i-2))$$
$$OPT(1) = w_1, \quad OPT(2) = w_2$$

Algorithm 1: `max_independent_set`(L)

Input: G a list of node weights in order corresponding to the n-node path
Output: S the indices of the node weights in L that correspond to the maximum independent set

```
// M is the memoisation cache array of OPT values
M ← []
// S the set of boolean values corresponding to whether or not a node belongs in
   the independent set
S ← {}
for 1 ≤ i ≤ |L| do
    w_i ← L[i]
    // The base case for the OPT recurrence
    if i ≤ 2 then
        M[i] ← w_i
        S[i] ← true
    else
        // Implementation of the OPT recurrence
        if w_i + M[i − 2] > M[i − 1] then
            M[i] ← w_i + M[i − 2]
            S[i] ← true
        else
            M[i] ← M[i − 1]
            S[i] ← false
        end
    end
end
// Trace through S backwards to validate/prune the output
S' ← {}
n ← |L|
while n > 0 do
    if S[n] == true then
        Add n to S'
        n ← n − 2
    else
        n ← n − 1
    end
end
return S'
```

Proof of Correctness:

The strategy employed by the presented algorithm is to compare the weights of the independent sets formed by taking either the current node and the node two "connections" prior (in order to maintain independence). Then, we trace backwards through the resultant set S to prune the invalid node indices from being considered when evaluating the entire path. In this way we ensure that the resultant set of nodes is correct insofar as it does not violate the definition of an independent set on the input path.

Secondly, we know that the algorithm is correct in terms of maximum weight as the OPT values are calculated iteratively over the the nodes such that the implicit lookup table is populated from low node indices to high, and then selected in reverse order. In this way, the max operator is able to rapidly compute the optimal values for each sub-set of the input path, then take indices "marked" from highest (most correct) to lowest, meaning that $OPT(n)$ will be the optimal value for the complete n-node path.

Proof of Runtime Complexity:

The for loop iterates over all the nodes in $O(n)$ time, building the OPT lookup table.

Though the while loop necessarily skips some of the nodes in order to guarantee independence in the resultant set S', asymptotically this still takes $O(n)$ time.

Therefore the runtime complexity is $O(n)$.

Problem 3 (45 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.

Solution:

Proof of Correctness / Description

The strategy is to recursively iterate over every cut m_i in the input list which we label M, making that cut, computing the cost values as the sum of the cuts in the respective left and right halves of the sub strings until and performing the same process on the two resultant sub strings with the cuts present on their ranges until no more cuts can be made. We repeat this process for all m cuts, keeping track of which cut gives the minimum for each possibility.

Formally, we can present this recurrence as the following:

$$OPT(n, M) = \min_{c \in M} \Big(n + OPT(c, [c_j | c_j < c]) + OPT(n - c, [c_j | c_j > c]) \Big)$$

where n is the length of the string, $n + OPT(c, [c_j | c_j < c])$ is the cost of making the cuts remaining in the left sub string, and $OPT(n - c, [c_j | c_j > c])$ the cost of the remaining cuts in the right sub string.

We observe that each cut creates two sub problems: sub strings where a cut or list of cuts may exits. We desire the minimum cost from cutting each sub string similar to the segmented least squares problem.

Though not implemented in the presented algorithm, we can utilize memoisation to cache the costs of making different cuts to improve the runtime of this approach by saving the cost of executing a list of cuts to be fetched / referenced later on.

Algorithm 2: `min_cost_cuts`(n, M)

Input: n the length of the input string, or sub string, M the list of locations of cuts
Output: The minimum total cost of breaking the string into $m + 1$ pieces at the given locations

```
// If there are no cuts to be made, the cost is 0
if M = [] then
    return 0
end
min_cost ← ∞
foreach cut c ∈ M do
    // Split the remaining cuts into two lists
    left_cuts ← [c_j | c_j < c]
    right_cuts ← [c_j | c_j > c]
    // cost of making the all the cuts to left of c on the sub string formed by
       making cut c
    left_cost = min_cost_cuts(c, left_cuts)
    // cost of making the all the cuts to right of c on the sub string formed by
       making cut c
    right_cost = min_cost_cuts(n − c,right_cuts)
    total_cost = n + left_cost + right_cost
    // Keep track of the minimum cost
    if total_cost < min_cost then
        min_cost ← total_cost
    end
end
return min_cost
```

Proof of Runtime Complexity

Let $T(n)$ be the running time of this algorithm.

the main loop iterates m times, with each recursive call within the loop running fewer than m times according to the following relationship:

$T(n) = \sum_{1 \le j \le m} \sum_{1 \le i \le j} O(j - i) = O(n^3)$ time.