CSE 551 Homework 4 Solutions

October 2021

1 Question 1

Define $T(0) = T(1) = 1$ and

$$T(n) = \sum_{i=1}^{n-1} T(i) \cdot T(i-1)$$

1. What is the runtime of the recurrence relation above if intermediate results are not stored?
2. If the value of $T(i)$ is memoized, what happens to the runtime?
3. Can you devise an evaluation technique with linear runtime?

Solution:
(1) The run time is $\Omega(2^n)$.

Proof. Base case: Observe that $T(0) = T(1) = 1$ takes constant time to compute and both can be bound below by a sufficiently small constant for $n = 0, 1$.
Now assume it holds true $T(k)$ $k \in \{1 \dots n\}$ (IH). So, $T(n+1)$ makes calls to $T(1), T(0), T(2), T(1) \dots T(n), T(n-1)$. Each of these calls is bounded below by $c_1 2^1, c_1 2^2, \dots c_1 2^n, c_1 2^{n-1}$. Thus, the total time is the sum of these individual components is bounded below by[1]

$$\begin{aligned}
\sum_{i=1}^{n} c_1 2^i + c_1 2^{i-1} &= \sum_{i=1}^{n} c_1 (2^i + 2^{i-1}) && (1)\\
&= c_1 \sum_{i=1}^{n} (2^i + 2^{i-1}) && (2)\\
&= c_1 \left(\sum_{i=1}^{n} 2^i + \sum_{i=1}^{n} 2^{i-1} \right) && (3)\\
&= c_1 \left(2^{n+1} - 2 + 2^n - 1 \right) && (4)\\
&\geq c_1 2^{n+1} && (5)
\end{aligned}$$

Hence, by inductive hypothesis, the claim is true. □

(2) Algorithm 1 provides a polynomial algorithm. Observe the outer and inner loops are require a polynomial number of iterations and the functions called in between the loops do not exceed polynomial time either.

[1] This analysis treats multiplication as a constant time operation, however, it does not actually matter as not treating multiplication as a constant time operation only makes things asymptotically slower.

Algorithm 1 Polynomial time $T(n)$

```
1: procedure POLY-T(n)
2:     M[n] = {0}                          ▷ Initialize all n entries to 0.
3:     M[0] = 1
4:     M[1] = 1
5:     for j = 2, j < n, j = j + 1 do
6:         for k = 1, k < j, k = k + 1 do
7:             M[j] = M[j] + M[k] · M[k − 1].
8:     return M[n]
```

(3) This can be reduced by observing that

$$\begin{aligned}\sum_{i=1}^{n-1} T(i) \cdot T(i-1) &= T(n-1) \cdot T(n-2) + \sum_{i=1}^{n-2} T(i) \cdot T(i-1) \\ &= T(n-1) \cdot T(n-2) + T(n-1)\end{aligned}$$

giving a linear time algorithm described in Algorithm 2.

Algorithm 2 Linear time computation of $T(n)$

```
1: procedure LINEAR-T(n)
2:     T_table[n]
3:     for i = 0, i ≤ n − 1, i = i + 1 do
4:         T_table[i] = 0
5:     T_table[0] = 1
6:     T_table[1] = 1
7:     for i = 2, i ≤ n − 1, i = i + 1 do
8:         T_table[i] = T_table[i − 1] + T_table[i − 1] · T_table[i − 2]
```

2 Question 2

A fast food joint sells chicken nuggets in packs of varying sizes. We want to know, given packs with sizes $[n_1, n_2, \ldots n_k]$, if we can order N chicken nuggets.

1. Devise a brute force solution and a dynamic programming solution.
2. Derive and compare the complexities of the brute force solution and the dynamic programming (DP) solution. What aspect of DP changes the complexity?

Solution:

(1) For brute force: Find the smallest number in the pack sizes call it n_i. Then assign $b = \lceil \frac{N}{n_i} \rceil$. Then for any possible solution to this problem, the number of purchases possible for any sized pack is $0 \ldots b$ purchases. Thus, if there are k packs, each configuration corresponds to a length k base b number. Enumerate from 0 to $b^k - 1$ and check if the corresponding configuration is a valid solution.

For dynamic programming, consider the following algorithm:

(2) The complexity for the dynamic DP solution is pseudo-polynomial. The recurrence relation for the complexity is described by:

$$T(n, k) = kT(N - b, k) + O(k)$$

Algorithm 3 Determine if you can achieve this many chicken nuggets.

```
1: procedure CHICKEN-NUGGETS(N, packs, k)                ▷ k the number of packs.
2:     if N = 0 then
3:         return True
4:     if packs is empty then
5:         return False
6:     res = False
7:     for i = 0; i < k; i = i + 1 do
8:         res = res or Chicken-Nuggets(N−pack[i], packs, k)
       return res
```

Observe the depth of the tree generated by T is b (as defined above) and the number of descendants for each tree is k, this means there are a total of

$$\frac{1}{k-1}\left(k^{b+1}-1\right)$$

nodes, each of which requires $O(k)$ amount of time, so:

$$\frac{1}{k-1}\left(k^{b+1}-1\right)O(k)$$

which is polynomial when N is fixed. However, we do not have that luxury, and as N is logarithmic with the input[2], we can choose sufficiently large N to make the problem exponential. The brute force, of course requires checking b^k all the time which, unless $b = N$ is always exponential in time complexity.

3 Question 3

A palindrome is a string that is the same whether read left-to-right or right-to-left. Devise a dynamic programming algorithm to find the *minimum* number of characters to delete from an input string to make it a palindrome.

Solution:

In a palindrome, for every character there is a corresponding character that is the same as the character, and when the string is reversed, the index of those characters swap. Keep two variables i and j to indicate what those two variables are. For a given string s that we want to convert to a palindrome, observe the following:

1. If $s[i] \neq s[j]$, something needs to be done as the string is not a palindrome. There are really only two options here: Remove $s[i]$ or remove $s[j]$. Removing a string that a character at index $a < i$ or $a > j$ does not help, nor does removing any character $i < a < j$. Thus, you can choose to remove i or j.[3]

2. If $s[i] = s[j]$, the optimal solution should not require you to delete either of these strings. Both of them can be skipped.

Thus the algorithm is described in Algorithm 4.

4 Question 4

There are N stairs and Alice can climb one or two stairs at a time. How many different ways can Alice climb the stairs such that she starts at the bottom and ends at the top? The algorithm does not need

[2]Unless, of course, you choose base 1 for what ever reason.

[3]Removing both is also an option too, but it is implicitly handled by removing, say, i first then j second.

Algorithm 4 Minimum number of characters that need to be removed for the string to be a palindrome.

```
1: procedure MIN-TO-PAL(s, i, j)                    ▷ Start with i = 0 and j = len(s) − 1
2:     if (s, i, j) have already been solved then
3:         Lookup the optimal number o.              ▷ For instance, use @lru_cache in Python. return o
4:     if i = j then
5:         return 0.
6:     if s[i] = s[j] then
7:         return MIN-TO-PAL(s, i + 1, j + 1).
8:     if s[i] ≠ s[j] then
9:         return 1 + min{MIN-TO-PAL(s, i, j − 1), MIN-TO-PAL(s, i + 1, j)}
```

to explicitly list the ways. First devise a top-down (recursive) approach to solve this problem. What is its runtime complexity if intermediate results are not stored and not reused? How does memoizing intermediate results change the complexity? Next devise a bottom-up DP approach to solve this problem. What is its runtime complexity?

Solution:

If $N = 0$ then Alice is done and the only (still 1!) way she managed to climb up the stairs was by doing nothing. If $N = 1$ Alice must climb up the stair. For any $N \geq 2$ Alice can always choose to ascend two stairs in one go, or only one stair and the options she has is the sum of those two choices. A top-down approach is formulated in Algorithm 5.

Algorithm 5 Count stair paths

```
1: procedure STAIR-COUNT-TD(N)
2:     if the particular N has been solved. then
3:         Lookup up the solution o.                 ▷ For instance, use @lru_cache in Python.
4:         return o.
5:     if N < 0 then
6:         return 0.
7:     if N = 0 or N = 1 then
8:         return 1
       return STAIR-COUNT-TD(N − 1) + STAIR-COUNT-TD(N − 2)
```

Observe that the algorithm is polynomial in time. Looking up does not require more than linear time in any step. A bottom up approach is described in Algorithm 6.

Algorithm 6 Count stair paths

```
1: procedure STAIR-COUNT-BU(N)
2:     store_count[N]
3:     for i = 0, i < n, i = i + 1 do
4:         store_count[i] = -1
5:     store_count[0] = 1
6:     store_count[1] = 1
7:     for k = 2, k ≤ n, k = k + 1 do
8:         store_count[k] = store_count[k − 1] + store_count[k − 2]
       return store_count[N]
   =0
```

5 Question 5

At a fair, there is a game with n balloons that are placed adjacent to each other in a line. In this game, you must shoot all the balloons. Each balloon $i \in 1, \ldots, n$ has an award $v(i)$. If you shoot ith balloon you will get an award of $v(i-1)v(i)v(i+1)$ unless the left or right balloon is missing (which is the case for the balloons on the ends of the line). In the case that left (or right) neighbor is missing, let $v(i-1) = 0$ (or $v(i+1) = 0$). When balloon i is shot, the balloons to the right of balloon i decrease their position by 1 and their awards remain the same. You have to shoot all balloons in some order so that you score a maximum total number of points.

1. Does the strategy "For the remaining balloons at each step, shoot the balloon with highest value" work?
2. Does the strategy "For the remaining balloons at each step, shoot the balloon with highest award at that point in time" work?
3. Devise a dynamic programming method to determine an optimal order in which to shoot the balloons. Make the algorithm as efficient as possible.

Solution:

1. No, consider balloons in order from left to right with values ②①③. If you shoot 3 first you get a score of 0 for doing so, and the remaining two balloons give a score of 0. Instead, if you shoot 1 first you will get $2 \cdot 1 \cdot 3 = 6 > 0$.
2. This will not work either, consider ⑨②①③⑨. Following highest award, you should first remove 3 adding 27 to the score, then 2 adding 18, and then 1 adding 81, giving a total of 126. However, you can easily increase this if you remove 2 and 1 before 3 so that way can get the award $9 \cdot 3 \cdot 9 = 243 > 126$.
3. You have to be careful on how you approach this. It seems at first that choosing a which balloon to remove will change the substructure of the problem because balloons shift based on which balloon is popped. To resolve this, select which balloon to pop *last*.

   Say we are looking at the range i through j in the array of balloons and select some balloon at index k to pop last. Then the award associated with popping the kth balloon is $\text{val}[i-1] \cdot \text{val}[k] \cdot \text{val}[j+1]$ as the award does not depend on any value in the range of i and j we look instead at the values just outside this range, namely $i-1$ and $j+1$. The algorithm is given in Algorithm 7.[4]

Algorithm 7 Balloon Popping Algorithm

procedure B-Pop(balloons, i, j, n) ▷ i and j are the range we are analyzing, n is the total length of the balloons array.
  if the given i, j has been solved **then**
    Look up the solution o in a table. ▷ For instance, use `@lru_cache` in Python.
    return o.
  $\text{val}[-1] = 0$
  $\text{val}[n+1] = 0$
  if $i > j$ or $j < i$ **then**
    return 0
  return $\max_{i \le k \le j}\{\text{val}[i-1] \cdot \text{val}[k] \cdot \text{val}[j+1] + \text{B-Pop}(\text{balloons}, i, k-1) + \text{B-Pop}(\text{balloons}, k+1, j)\}$

[4] If you max over an invalid range, e.g. $\max_{1 \le k \le 0}$ give a value of 0.