Homework 1 – COT5405

Kobee Raveendran

August 2021

1. note: all logs are implicitly base 2, but other bases may be specified explicitly. Also, I use the Master theorem template for some questions, so it is listed below.

   $T(n) = aT(\frac{n}{b}) + f(n)$, where $f(n) \in O(n^k \log^p n)$

   (a) $T(n) = 9T(\frac{n}{2}) + n^3 \longrightarrow O(n^{\log 9}) \longrightarrow O(n^{2 \log 3})$

   Steps:
   using the Master theorem, we have:
   $a = 9$
   $b = 2$
   $f(n) = n^3 \longrightarrow k = 3, p = 0$
   Since $\log_2 9 > 3$, we follow case 1 of the Master theorem ($\log_b a > k$), which means the bound is in $O(n^{\log_b a})$. After substitution, that gives us:
   $O(n^{\log 9}) = O(n^{2 \log 3})$

   (b) $T(n) = 7T(\frac{n}{2}) + n^3 \longrightarrow O(n^3)$

   Steps:
   using the Master theorem, we have:
   $a = 7$
   $b = 2$
   $f(n) = n^3 \longrightarrow k = 3, p = 0$
   We find that this recurrence falls under case 3 of the theorem, in which $\log_b a < k$ (since $(\log_2 7 < 3)$), so we arrive at a complexity of the form $O(n^k) \longrightarrow O(n^3)$.

   (c) $T(n) = T(\sqrt{n}) + \log n$

   (d) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

   (e) $T(n) = 3T(\frac{n}{3}) + \frac{n}{3}$

   Steps:
   using the Master theorem, we have:
   $a = 3$
   $b = 3$
   $f(n) = \frac{n}{3} \longrightarrow k = 1, p = 0$
   This recurrence falls under case 2 of the theorem, in which $\log_b a = k$ (since $\log_3 3 = 1$), so our complexity is of the form $O(n^k \log n) \longrightarrow O(n \log_3 n)$ (base changes since we divide the work done each recurrent step by 3 rather than the usual 2).

   (f) $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n \log n$

2. blah

3. dynamic programming ($O(n)$): overall idea is to store maximum partial sums that we've encountered as we traverse through the array. For each index, we'd store the max partial sum if the ending index, k, were to be at that index, and also track the starting index, j, of each max partial sum's subarray. At each index, we will decide whether to "continue" the

subarray if the current element's addition will increase the sum, or start a new subarray if the current element is already greater than whatever sum we have (i.e. the current sum was negative). If the current max partial sum is ever greater than the overall max partial sum, we update it and also update the start and end points of the subarray, $final_j$ and $final_k$.

Algorithm 1 Dynamic programming approach ($O(n)$ time with constant space)

```
j, k, final_j, final_k ← 0
max_total ← −∞
max_curr ← 0
N ← length(A)
for i ← 0 to N do
    if A[i] > max_curr + A[i] then
        j ← i
        k ← i + 1
        max_curr ← A[i]
    else
        k ← i
        max_curr ← max_curr + A[i]
        if max_curr > max_total then
            final_j ← j
            final_k ← k
            max_total ← max_curr
```

4. blah

5. idea: since the rows and columns are sorted, we can eliminate entire sections of the grid if we an element is greater than or lesser than the target to be found. In the case where an element of the matrix is greater than the target, we can eliminate all rows that come after that element, and all columns that come after that element, since all of those elements will also be greater than the target. The inverse applies if the current element is less than the target; we'd instead eliminate all rows and columns before it, since the target can't be there. If the loop ends without finding the target, it does not exist in the matrix.

   View algorithm **??**

Algorithm 2 Iterative elimination approach ($O(n)$ time with constant space)

```
1:  N ← length(A)
2:  i ← 0
3:  j ← N − 1
4:  while i < N and j ≤ 0 do
5:      if target = A[i][j] then
6:          return (i, j)
7:      else
8:          if target > A[i][j] then
9:              j ← j − 1
10:         else
11:             i ← i + 1
```

6. use something like quicksort's partitioning algorithm?