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) $T(n) = T(\sqrt{n}) + \log n$
   Steps:
   using iterative substitution:
   $k = 0$: $T(n) = T(n^{\frac{1}{2}}) + \log n$
   $k = 1$: $T(n^{\frac{1}{2}}) = T(n^{\frac{1}{4}}) + \log n^{\frac{1}{2}}$

   Substituting back in, we have:
   $T(n) = T(n^{\frac{1}{4}}) + \log n^{\frac{1}{2}} + \log n = T(n^{\frac{1}{4}}) + \frac{1}{2} \log n + \log n$

   Which can be generalized to:
   $T(n) = T(n^{\frac{1}{2^{k+1}}}) + (1 + \frac{1}{2} + \frac{1}{4} + .. + \frac{1}{2^k}) \log n \approx T(n^{\frac{1}{2^{k+1}}}) + 2 \log n$

   Using a base case of $T(2) = c$, we can solve for k:
   $n^{\frac{1}{2^{k+1}}} = 2 \longrightarrow \log n^{\frac{1}{2^{k+1}}} = \log 2 = 1 \longrightarrow \frac{1}{2^{k+1}} \log n = 1 \longrightarrow \log n = 2^{k+1} \longrightarrow$
   $\log \log n = \log 2^{k+1} \longrightarrow \log \log n = k + 1 \longrightarrow k = \log \log n - 1$

   Substituting k back in, as well as $T(2) = c$, we have:
   $T(n^{\frac{1}{2 \log \log n}}) + 2 \log n \longrightarrow c + 2 \log n$

   Giving us a time complexity of $O(\log n)$, which can be confirmed using the Master theorem by substituting $n = 2^m$ to get a compatible recurrence format.

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

Steps:

using iterative substitution, we have:

$T(n) = n^{\frac{1}{2}}T(n^{\frac{1}{2}}) + n$

$k = 1$: $T(n^{\frac{1}{2}}) = n^{\frac{1}{4}}T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}$

Substituting: $T(n) = n^{\frac{1}{2}}[n^{\frac{1}{4}}T(n^{\frac{1}{4}}) + n^{\frac{1}{2}}] + n \longrightarrow T(n) = n^{\frac{1}{2}+\frac{1}{4}}T(n^{\frac{1}{4}}) + n + n$

$k = 2$: $T(n^{\frac{1}{4}}) = n^{\frac{1}{8}}T(n^{\frac{1}{8}}) + n^{\frac{1}{4}}$

Substituting: $T(n) = n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}}T(n^{\frac{1}{8}}) + n^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}} + n + n$

Simplifiying the fractional sums (which converge to 1), we have the general form:

$T(n) = nT(n^{\frac{1}{2^{k+1}}}) + (k+1)n$

Using base case $T(2) = c$ and solving for k, we have:

$n^{\frac{1}{2^{k+1}}} = 2 \longrightarrow \frac{1}{2^{k+1}} \log n = 1 \longrightarrow \log\log n = \log 2^{k+1} \longrightarrow \log\log n - 1 = k$

Substituting k back in and using the base case, we have:

$cn + (\log\log n)n \longrightarrow O(n \log\log n)$

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

Steps:

can't directly apply the Master theorem, but we can split the RHS since we can assume that $T(\frac{n}{4}) \leq T(\frac{n}{2})$ (since less of the input is processed in the recursive call where the input is quartered rather than halved), leaving us with the following true inequality:

$$2T(\tfrac{n}{4}) + n \log n \leq T(n) \leq 2T(\tfrac{n}{2}) + n \log n$$

Finding the asymptotic bounds of either side lets us narrow down the bounds for $T(n)$:

(using the Master theorem for both)

LHS:

$a = 2$

$b = 4$

$f(n) = n \log n \longrightarrow k = 1, p = 1$

$\log_b a = \log_4 2 = \frac{1}{2} \longrightarrow \frac{1}{2} < 1 \longrightarrow$ case 3 of Master theorem $\longrightarrow O(n \log n)$

RHS:

$a = 2$

$b = 2$

$f(n) = n \log n \longrightarrow k = 1, p = 1$

$\log_b a = \log_2 2 = 1 = k \longrightarrow$ case 2 of Master theorem $\longrightarrow O(n \log n)$

Since $T(n)$ is thus bounded on both sides by $O(n \log n)$, it is safe to claim that $T(n) \in O(n \log n)$.

2. blah

3. (a) strat:

   for a set of integers in an interval, roughly $\frac{1}{2}$ of them will have a most-signifcant bit of 1, while the other half will have a most significant bit of 0. Since John's array is sorted, we know that the most significant bit of the numbers in $A[1, \frac{n}{2}]$, where n is the length of the array, is 0, while the most significant bit of the numbers in $A[\frac{n}{2}, n]$ is 1. So, the process for finding the missing integer could be as follows:

   i. ask John for the most significant bit of the $A[\frac{n}{2}]$ and $A[\frac{n}{2} + 1]$ (in other words, the two elements at the inflection point, where the most significant bit should change from 0 to 1).

   ii. If both the leftmost bits of $A[\frac{n}{2}]$ and $A[\frac{n}{2} + 1]$, we know that the missing number must be in the bottom half of the interval. If both are 0, we know the missing integer must be in the top half of the interval.

   iii. From there, we can repeat steps i. and ii. in a manner similar to binary search, by repeatedly cutting search space in half each time.

   iv. since the search space is cut in half for every pair of questions, the runtime of this algorithm is $O(\log n)$

4. 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
        end if
    end if
end for
```

5. blah

6. idea: since the rows and columns are sorted, we can eliminate entire sections of the grid if we find 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, the target does not exist in the matrix.

   View algorithm 2 below

7. use something like quicksort's partitioning algorithm?

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

$N \leftarrow length(A)$
$i \leftarrow 0$
$j \leftarrow N - 1$
while $i < N$ and $j \leq 0$ **do**
    if `target` $= A[i][j]$ **then**
        `return` (i, j)
    else
        if `target` $> A[i][j]$ **then**
            $j \leftarrow j - 1$
        else
            $i \leftarrow i + 1$
        end if
    end if
end while