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 $T(\frac{n}{4}) \le T(\frac{n}{2})$ (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:

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

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:

$a = 2$

$b = 2$

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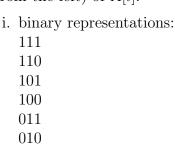
$\log_b a = \log_2 2 = 1 = k \longrightarrow$ case 2 of Master theorem $\longrightarrow O(n \log n)$

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) 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 $A[1, \frac{n}{2}]$, 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 $A[\frac{n}{2}]$; this gives us the element at the inflection point, where the most significant bit would have changed from 0 to 1. Since one element is missing however, we know that the number of integers with leading bit 0 and leading bit 1 are not equal, so one "half" has one more element than the other. This means the element in the middle must belong to the half that *isn't* missing the integer.

   ii. Thus, if the jth bit of the middle element is a 1, we know the missing integer's j^{th} bit is a 0 (bottom half of the interval), and vice versa. Once we know which half of the interval to search in, we can eliminate the other half as a possibility.

   iii. Once we determine the correct "half" of the interval, we can increment j (so now, we will ask John about the 2nd most significant bit), and this time ask for this j^{th} bit at $A[\frac{n}{4}]$ and $A[\frac{n}{4} + 1]$ or $A[\frac{3n}{4}]$ and $A[\frac{3n}{4} + 1]$, depending on what we found in step ii.

   iv. From here, we can repeat steps i., ii., and iii. in a manner similar to binary search, by repeatedly cutting the search space in half each time and asking John about the inflection points. Each time we limit the search space, we can "fill in" the jth bit of the missing number. This process will continue recursively until the final bit is reached and its value is filled in, giving us the complete binary representation of the missing number.

   v. since the search space is cut in half after each question, the runtime of this algorithm is $O(\log n)$

   Below is a toy example of this, with numbers in the range of 0 to 7, with 1 being the missing integer. Note that all arrays here are 0-indexed, and bits are indexed from the most- to least-significant bit for simplicity. For example, $A[i][j]$ signifies the jth bit (from the left) of $A[i]$.

   i. binary representations:
   111
   110
   101
   100
   011
   010
   000

   ii. ask John for $A[3][0]$; it's a 1, so the missing number should be in the lower half. Thus, our currently "built" representation of the missing number is "0". The sub-array to search in now is:

   011
   010
   000

   iii. Ask John for $A[1][1]$; it's 1, so again the missing number is in the lower half. So, we add another 0 to our missing number's representation, which is now "00". The subarray is now:

   000

   iv. We have only 1 number left, and we ask John for $A[0][2]$, which is 0, so we know the missing number's rightmost bit must be 1. Appending that to our current representation, we find that the missing integer is "001" = 1.

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)

$j, k, final_j, final_k \leftarrow 0$
$max_{total} \leftarrow -\infty$
$max_{curr} \leftarrow 0$
$N \leftarrow length(A)$
for $i \leftarrow 0$ to N **do**
    if $A[i] > max_{curr} + A[i]$ **then**
        $j \leftarrow i$
        $k \leftarrow i + 1$
        $max_{curr} \leftarrow A[i]$
    else
        $k \leftarrow i$
        $max_{curr} \leftarrow max_{curr} + A[i]$
        if $max_{curr} > max_{total}$ **then**
            $final_j \leftarrow j$
            $final_k \leftarrow k$
            $max_{total} \leftarrow max_{curr}$
        end if
    end if
end for

---

5. idea: If the two sums X and Y are unequal but can be made equal by swapping their addends, it should be the case that one sum has an added that "over-contributes" to it while the other sum has an added that "under-contributes", causing the difference between the sums. If so, their sums can only be made equal if the two are swapped, and the difference between the sums must be an even number (if it is odd, swapping any of the numbers would still result in an unequal sum). So, our goal is to find the pair of numbers that contribute "evenly" to the difference in the two sums; that is, their difference must be equal to $\frac{1}{2}$ the difference between Y and Y. Thus, when they're swapped, the lesser sum between X and Y increases by $\frac{d}{2}$ and the greater sum decreases by $\frac{d}{2}$ (where $d = |X - Y|$ and d is even), thus making them equal.

Analysis: My approach uses a hash map to quickly retrieve elements in X that satisfy $X_i + Y_j = \frac{d}{2}$ in constant rather than linear time, allowing us to find suitable pairs in amortized $O(max(n, m))$ time instead of $O(n \cdot m)$. The total runtime is amortized $O(max(n, m))$ since both X and Y are iterated over (either to populate the hashmap or to find a suitable Y_j). Of course, worst case insertion into a hashmap is $O(n)$, taking the total worst-case runtime to $O(n^2)$, which is only seen in the event of many hash collisions caused by a poor hashing function. The total space complexity is $O(n)$, since every addend in X is mapped to its index via the hashmap.

Algorithm 2 Hash map pair checking $O(max(n, m))$ time with $O(n)$ space

---

`diff` $\leftarrow X - Y$
if `diff` $\% 2 \neq 0$ **then**
    return `null`
end if
`diff` $\leftarrow \frac{\texttt{diff}}{2}$
`diffMap` $\leftarrow$ `HashMap`$\{x_i \rightarrow i | i \in 1 \ldots n\}$

for $j \leftarrow 0$ to m **do**
    if `diff` $+ y_j \in$ `diffMap` **then**
        $i \leftarrow$ `diffMap`$[\texttt{diff} + y_j]$
        return (i, j)
    end if
end for

---

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 $A[i][j]$ is greater than the target, we can eliminate a rectangular section of the grid; that is, all cells with $row > i$ and $col > j$, since all of those elements will also be greater than the target. We'd then increment the row i and try again. The inverse applies if the current element is less than the target; we'd instead eliminate all cells with $row < i$ and $col < j$, since the target is larger than them. We'd then decrement the column j and try again. If $A[i][j]$ is ever equal to the target, we've found it; else, if the loop ends, the target must not exist in the matrix.

This is demonstrated more explicitly in the algorithm below.

Algorithm 3 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

---

7. use something like quicksort's partitioning algorithm?