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. idea: we can compute the k^{th} smallest number in $A[1, n]$ in $O(n)$ via the median of medians selection algorithm, but cannot do so for all $k = \log_2 n$ elements as that would result in a total runtime of $O(n \log n)$. This naive approach, however, is only needed if we had to make use of the entire array after finding each k^{th}-smallest element. In this case though, after we find a k^{th}-smallest element, everything that was less than it is not useful in our search for the $(k+1)^{st}$-smallest element. So instead, we can try reducing the median of medians search space after each k^{th}-smallest element is found, by removing any invalid candidates once the k^{th}-smallest is found. Since earlier powers of 2 are closer together, I instead opt to start from the last power of 2 and work backwards, pruning elements *larger* than the k^{th}-smallest, to prune more of the search space earlier. The pruning process may either create a new array of valid elements or use a linked list to support fast removal of all invalid elements (to avoid more than $O(\frac{n}{i})$ pruning-related work each i^{th} iteration).

   The steps would go as follows:

   (a) initialize i such that $k = 2^i$ is the greatest power of 2 that is $\leq n$

   (b) run median of medians to find the 2^i-th smallest element

   (c) prune the array by removing all elements greater than the 2^i-th smallest element. In almost all iterations, this will remove half of the current iteration's search space.

   (d) decrement i and repeat steps (b) and (c) for all $i \geq 0$.

   In general, this approach will find all such k^{th} smallest elements in $O(n)$ time, because during each step, we are processing approximately half the array of the previous iteration (since the other half is discarded), giving us $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + \frac{n}{2^i} \approx cn$ steps of total work, yielding an $O(n)$ total time complexity. Depending on whether n is a power of 2, and the subsequent amount of pruning work, the constant c may change, but the runtime will remain linear.

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 j^{th} 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 2^{nd} 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 j^{th} 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, and only one question is asked each time the search space is limited (i.e. only one element is "processed" in $O(1)$), 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 j^{th} 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 subarray 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.

(b) Since the array is no longer sorted, we can't *automatically* eliminate half the array after each question like we did previously. This prevents us from "knowing" how many integers have a j^{th} bit of 1 or 0 unless we count each one. So, that's what we'll do: we can take the same approach as in part (a), but we would be responsible for pruning out the invalid integers manually. We can do this by first narrowing down the j^{th} bit of the missing integer with n questions, then eliminating integers that do not have the same j^{th} bit, which should be half of them. This would require $2n$ operations per iteration: n questions to find the missing number's j^{th} bit, and another n questions to find and keep any integers that don't have the same j^{th} bit. In the next iteration, we'd repeat the process on $\frac{n}{2}$ integers, then the next on $\frac{n}{4}$ integers, etc. until we are left with 1 integer.

The steps could go like this:

i. initialize an (unordered) set with all numbers from 1 to n; this will serve to track which indices i in A will be valid, since we don't actually know the values of the integers themselves.

ii. ask John for the leftmost (0^{th}) bit of all numbers in the set, counting the number of 1's and 0's. If the number of 0's is $\leq$ the number of 1's, the missing number's leftmost (0^{th}) bit is 0; else it's a 1.

iii. once again ask John for the j^{th} bit (where j is now 1) of all available numbers in the set, but remove numbers whose j^{th} bit does not match the missing number's j^{th} bit. This leaves us with only the halved search space for the next iteration.

iv. repeat steps ii. and iii. until the set contains 1 element. It's rightmost bit must be the inverse of the missing number's, so fill it in and end the loop.

This approach involves doing $2k$ operations per iteration, where k is the current cardinality of the valid set. Expanded out, the algorithm does $2n + 2 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + \ldots 2\frac{n}{2^i} \approx 4n$ total work, which is $O(n)$.

4. (a) divide and conquer ($O(n \log n)$): idea is to find max partial sums of all subarrays in the overall array, but rather than exhaustively test all subarrays, we can notice that the

max partial sum of two subarrays is either the max of each individual subarray's sum, or their combination (which crosses the midpoint). So, a simple solution is to recursively call this function on the left and right halves of the array, and the max partial sum will be the $max(f(left), f(right), f(left + right))$. The left and right half sums are computed recursively, while the midpoint sum requires iterating from the midpoint to its left and right and computing the max sum manually. See the below example using the array given in the homework problem:

$$[3, -5, 6, 7, 8, -10, 7]$$
$$[3, -5, 6, 7] \text{ — } [8, -10, 7]$$
$$[3, -5] \text{ — } [6, 7] \text{ — } [8, -10] \text{ — } [7]$$
$$[3] \text{ — } [-5] \text{ — } [6] \text{ — } [7] \text{ — } [8] \text{ — } [-10] \text{ — } [7]$$

On the way up from the base cases (single elements), the max sums (between the left, right, and across midpoint) are computed:

$$[3, -5, 6, 7, 8, -10, 7](21- \text{ from crossing midpoint})$$
$$[3, -5, 6, 7](13) \text{ — } [8, -10, 7](7)$$
$$[3, -5](-2) \text{ — } [6, 7](13) \text{ — } [8, -10](-2) \text{ — } [7](7)$$
$$[3] \text{ — } [-5] \text{ — } [6] \text{ — } [7] \text{ — } [8] \text{ — } [-10] \text{ — } [7]$$

(b) 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. 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 $X_i + Y_j = \frac{d}{2}$ for some i, j 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 the addends of 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 hash 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 ← X − Y
if diff %2 ≠ 0 then
    return null
end if
diff ← diff/2
diffMap ← HashMap{x_i → i | i ∈ 1 . . . n}

for j ← 0 to m do
    if diff + y_j ∈ diffMap then
        i ← diffMap[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 algorithm 3.

Algorithm 3 Iterative elimination approach ($O(n)$ time with constant space)

```
N ← length(A)
i ← 0
j ← N − 1
while i < N and j ≤ 0 do
    if target = A[i][j] then
        return (i, j)
    else
        if target > A[i][j] then
            j ← j − 1
        else
            i ← i + 1
        end if
    end if
end while
```

7. use something like quicksort's partitioning algorithm?

8. As mentioned, the naïve implementation will always run in $O(n^2)$ time, since every bolt is exhaustively compared with every nut. However, that approach doesn't take advantage of the fact that after we do a series of n comparisons for a bolt, we have a relative idea of which nuts are smaller than that bolt's nut, and which nuts are larger. We can then group the nuts into a larger set and smaller set, using the bolt's matching nut as a pivot, like in quicksort's partitioning algorithm. We can then reference the pivot bolt(s) to narrow down which group the next nut-bolt combo belongs to, placing them into their appropriate groups until we've "sorted" all nut-bolt pairs.

   This quicksort-like approach can give us $O(n \log n)$ average case run-time if we use a randomized partition at each step. Of course, there's still a chance that we happen to choose

the largest, then second largest, then third largest, etc. bolt as a pivot in each iteration, which can cause the runtime to degenerate to $O(n^2)$, but in almost all cases, we can expect a $O(n \log n) \in o(n^2)$ runtime.