CS 3000: Algorithms & Data — Spring 2024

Homework 2
Due Saturday, February 3 at 11:59pm via Gradescope

Name: Alexander Rosenthal
Collaborators:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- This homework is due Saturday, February 3 at 11:59pm via Gradescope. No late assignments will be accepted. Make sure to submit something before the deadline.
- Solutions must be typeset. If you need to draw any diagrams, you may draw them by hand as long as they are embedded in the PDF. We recommend that you use LaTeX, in which case it would be best to use the source file for this assignment to get started.
- We encourage you to work with your classmates on the homework problems, but also urge you to attempt all of the problems by yourself first. If you do collaborate, you must write all solutions by yourself, in your own words. Do not submit anything you cannot explain. Please list all your collaborators in your solution for each problem by filling in the `yourcollaborators` command.
- Finding solutions to homework problems on the web, or by asking students not enrolled in the class is strictly forbidden.

Problem 1. *(10 points) Karatsuba Example*

Carry out Karatsuba's Algorithm to compute $47 \cdot 63$. What are the inputs for each recursive call, what does that recursive call return, and how do we compute the final product?

Solution:
Karatsuba's Algorithm can be denoted by $K(n, m, k)$ for multiplying integers n and m with k digits. This gives us:
$K(47, 63, 2)$: $a \leftarrow 4$, $b \leftarrow 7$, $c \leftarrow 6$, $d \leftarrow 3$
$e \leftarrow K(a, c, \frac{k}{2}) = K(4, 6, 1)$, $k = 1$ so simply return $4 \cdot 6 = 24$
$f \leftarrow K(b, d, \frac{k}{2}) = K(7, 3, 1)$, $k = 1$ so return $7 \cdot 3 = 21$
$g \leftarrow K(b - a, c - d, \frac{k}{2}) = K(7 - 4, 6 - 3, 1) = K(3, 3, 1)$, $k = 1$ so return $3 \cdot 3 = 9$
The final result is: $10^k \cdot e + 10^{\frac{k}{2}} \cdot (g + e + f) + f$
$= 10^2 \cdot 24 + 10^1 \cdot (9 + 24 + 21) + 21$
$= 2400 + 540 + 21$
$= 2400$
$+ \quad 540$
$+ \quad 21$
$= 2961$

Problem 2. *(13 points) Recursion Tree*

Consider the following recurrence:

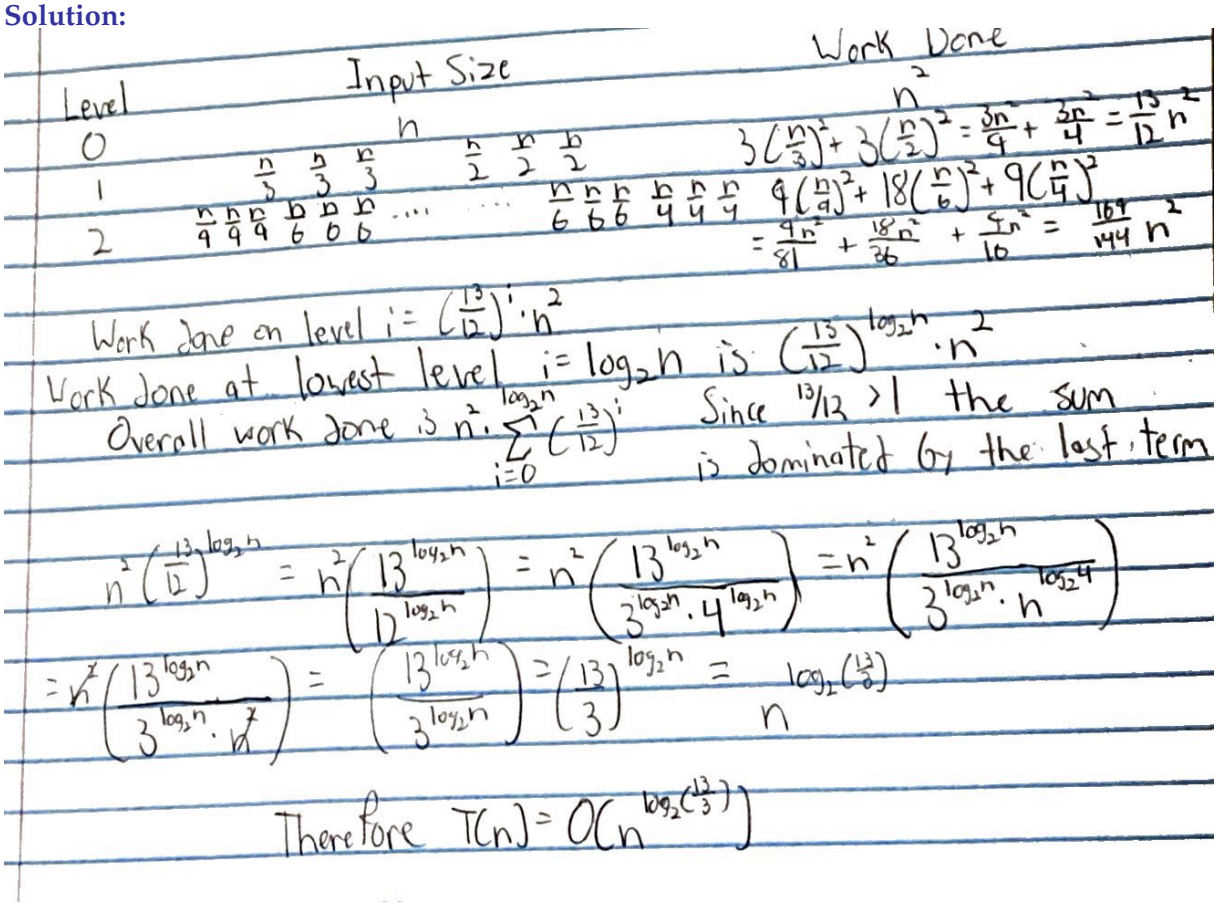
$$T(n) = 3T(n/3) + 3T(n/2) + n^2$$

$$T(1) = C$$

We will show that $T(n) = O(n^{\log_2(13/3)})$. To do this, start by examining the first three levels of the recursion tree, showing how to compute the amount of work at each level. From here, establish a formula for the amount of work on level i. Then, determine the last level of the recursion tree (note that it is sufficient to focus on the largest piece at level i, as we are only concerned with a Big-O bound). Finally, construct a summation which totals the amount of work over all levels and show why this summation is $T(n) = O(n^{\log_2(13/3)})$.

You are welcome to embed a photo of a hand draw image into your LaTeX file[1].

Solution:



[1] \includegraphics is useful for this

Problem 3. *(14 + 5 = 19 points) Help thy neighbor*

2023 was a tough year for the Red Sox. Looking to find some silver linings in the difficult season, manager Alex Cora has asked you to help identify the best stretch of the season for the team. To do this, the analytics department has shared with you the run differentials from n games this season. For each game, this value can be positive (if the Sox won) or negative (if they lost). For example, if they won 10-4, then lost 6-2, then won 3-2, the run differentials would be 6, -4, 1. Alex would like you to find the stretch of consecutive games such that the total run differential is maximized. The total run differential of a stretch of games is simply the sum of the run differentials of those games.

For example, consider the following:

$n = 7$, differentials: $-1, 5, -2, 1, 4, -3, 1$
Here, the optimal solution is from game 2 to game 5, and the total run differential is 8.

(a) Write (in pseudocode) a divide and conquer algorithm which solves this problem in $O(n \log n)$ time. Provide a few sentences describing your approach. Note that your algorithm should output the range of games that yields the maximum total run differential.

Solution:
This solution finds the maximum run differential by dividing the array into left, right, then middle arrays and finding the run differential of each part. The left and right half maximums are found using recursion, and the middle max is found using a helper function called MaxMiddleSubarray that returns the maximum subarray which MUST include the middle element.

Algorithm 1: Finding Maximum Run Differential

```
Function MaxSubarray(A, l, r):
    If l ≥ r :
        Return (l, r, A[l])
    mid ← ⌊(l+r)/2⌋
    leftMax ← MaxSubarray(A, l, mid−1)
    rightMax ← MaxSubarray(A, mid+1, r)
    middleMax ← MaxMiddleSubarray(A, l, mid, r)
    \\MaxSubarray returns an array of (left index, right index, differential) so [3] gets the
     differential
    If (leftMax[3] ≥ rightMax[3]) and (leftMax[3] ≥ middleMax[3]) :
        Return leftMax
    If (rightMax[3] ≥ leftMax[3]) and (rightMax[3] ≥ middleMax[3]) :
        Return rightMax
    Else
        Return middleMax
Function MaxMiddleSubarray(A, l, m, r):
    leftSum ← −∞
    tempSum ← 0
    leftIdx ← -1
    For i from m down to l
        tempSum ← tempSum + A[i]
        If tempSum > leftSum :
            leftSum ← tempSum
            leftIdx ← i
    rightSum ← −∞
    tempSum ← 0
    rightIdx ← -1
    For i from m up to r
        tempSum ← tempSum + A[i]
        If tempSum > rightSum :
            rightSum ← tempSum
            rightIdx ← i
    middleSum ← leftSum + rightSum - A[m]
    Return (leftIdx, rightIdx, middleSum)
```

(b) Write a recurrence for the runtime of your algorithm and solve it. You may use any method for solving the recurrence that we have discussed in class. Justify your recurrence.

Solution:

The run time of the algorithm for an input size of n is $T(n) = 2T(\frac{n}{2}) + n$. This is because the array is split into 2 groups of $\frac{n}{2}$, and the middle subarray is found by iterating over each half of the array, giving a total runtime of $O(n)$. The recurrence can be solved using the Master Theorem where $a = 2, b = 2$, and $d = 1$. Then $\frac{a}{b^d} = 1$ so $T(n) = O(n^d \log n) = O(n \log n)$

Problem 4. *(2 + 2 + 10 + 5 = 19 points) Lucky coincidence*

Given a sorted array of distinct integers $A[1,\ldots,n]$, you want to find out whether there is an index i such that $A[i] = i$. For example, in the array $A = \{3,5,6\}$, there is no i such that $A[i] = i$ and the algorithm should return "no index found". On the other hand, for the array $A = 0,2,7$ we have $A[2] = 2$ and the algorithm should return $i = 2$.

(a) What is the answer if $A[1] > 1$?

Solution:
If $A[1] > 1$ then all elements after the first will also be greater than their index because the elements must increase. Essentially, the index will never catch up to its element because the element can't slow down. The answer is "no index found".

(b) What is the answer if $A[n] < n$?

Solution:
if $A[n] < n$ then all elements before the last will also be less than their index because they must decrease. This is the same as the case for part (a) but in the backwards direction. The answer is "no index found".

(c) Give an efficient divide-and-conquer algorithm for this problem. Hint: check if $A[\lfloor n/2 \rfloor] = \lfloor n/2 \rfloor$. If not, try to recurse on a smaller problem.

Solution:

```
Function FindMatch(A, low, high):
    If (low == high) and (A[low] == low) :
        Return low
    If low ≥ high :
        Return "no index found."
    mid = ⌊ (low+high)/2 ⌋
    If A[mid] = mid :
        Return mid
    If A[mid] > mid :
        Return FindMatch(A, low, mid - 1)
    Else
        Return FindMatch(A, mid + 1, high)
```

(d) Write the recurrence of the running time and solve it using the master theorem.

Solution:
At each step of the algorithm the problem is divided into one subproblem of half the size.
This gives us $T(n) = T(\frac{n}{2})$
Using the Master Theorem, $a = 1, b = 2, d = 0$, so $\frac{a}{b^d} = \frac{1}{2^0} = \frac{1}{1} = 1$
Therefore $T(n) = O(n^d \log n) = O(n^0 \log n) = O(\log n)$

Problem 5. *(14 + 5 = 19 points) Deep in tot*

You have been put in charge of judging the potato sorting contest at this year's Eastern Idaho State Fair. Each contestant is tasked with sorting n potatoes by weight in ascending order, without the assistance of any equipment. The judging is as follows:

- Given an ordering of n potatoes, a mistake consists of two potatoes i and j such that potato i is before potato j in ther order, but i weighs more than j.

You need to be able to quickly calculate the number of mistakes in a given ordering. The runtime of your algorithm should be $O(n\log(n))$ for full credit. You have the ability to compare the weights of two potatoes in $O(1)$ time.

(a) Describe a divide-and-conquer algorithm (in pseudocode) which returns the number of mistakes. Provide a complete written description of your approach.

Solution:
The algorithm described below takes the given array and completes the steps for Merge-Sort while keeping track of the number of swaps made for elements that are out of order (AKA a mistake). It does this by counting the number of elements in the left subarray that are larger than an element in the right subarray while merging. The function returns the array itself and the number of mistakes found so far.

Algorithm 2: Counting the number of sorting mistakes

```
Function SortingMistakes(A):
    n ← length(A)
    If n == 1 :
        Return (A, 0)
    m ← ⌊n/2⌋
    left ← SortingMistakes(A[1 : m])
    right ← SortingMistakes(A[m : n])
    Return MergeMistakes(left[1], right[1], left[2] + right[2])
Function MergeMistakes(left, right, pastMistakes):
    n ← length(left) + length(right)
    merged ← [1 ... n]
    li ← 1, ri ← 1
    mistakes ← pastMistakes
    For i from 1 to n
        If li > length(left) :
            merged[i] ← right[ri]
            ri ← ri + 1
        If ri > length(right) :
            merged[i] ← left[li]
            li ← li + 1
        If left[li] ≤ right[ri] :
            merged[i] ← left[li]
            li ← li + 1
        Else
            merged[i] ← right[ri]
            ri ← ri + 1
            mistakes ← length(left[li : end of array])
    Return (merged, mistakes)
```

(b) State the worst-case asymptotic running time of your approach. Write a recurrence which captures the worst-case runtime of your algorithm and solve it using any method that we've seen in class.

Solution:

The algorithm has the same run time for all input sizes just like MergeSort. The algorithm divides the input into 2 subproblems of half the size and takes $O(n)$ time to combine the subproblems in the combining step (MergeMistakes).

This gives us a recurrence of $T(n) = 2T(\frac{n}{2}) + n$

By the Master Theorem, $a = 2, b = 2, d = 1$ so $\frac{a}{b^d} = \frac{2}{2^1} = \frac{2}{2} = 1$

Therefore $T(n) = O(n^d \log n) = O(n^1 \log n) = O(n \log n)$