CS 3000: Algorithms & Data — Spring 2024

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

Name: Ryan Tietjen
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:

Using psuedocode from slide 15 of lecture 5:

$u = 47$
$v = 63$
$n = 2$

Karatsuba(u, v, n):
$m = \frac{n}{2} = 1$
$a = 4$
$b = 7$
$c = 6$
$d = 3$
$e = \text{Karatsuba}(a, c, m) = \text{Karatsuba}(4, 6, 1) = 24$
$f = \text{Karatsuba}(b, d, m) = \text{Karatsuba}(7, 3, 1) = 21$
$g = \text{Karatsuba}(b - a, c - d, m) = \text{Karatsuba}(3, 3, 1) = 9$
Return $10^{2m} \cdot e + 10^m \cdot (e + f + g) + f$
$= 10^2 \cdot 24 + 10^1 \cdot (24 + 21 + 9) + 21$
$= 100 \cdot 24 + 10 \cdot 54 + 21$
$= 2400 + 540 + 21$
$= \boxed{2961}$

Karatsuba(4, 6, 1):
Return $4 \cdot 6 = 24$

Karatsuba(7, 3, 1):
Return $7 \cdot 3 = 21$

Karatsuba(3, 3, 1):
Return $3 \cdot 3 = 9$

Thus, Karatsuba's Algorithm calculates $47 \cdot 63$ to be $\boxed{2961}$

Problem 2. *(13 points) Recursion Tree*

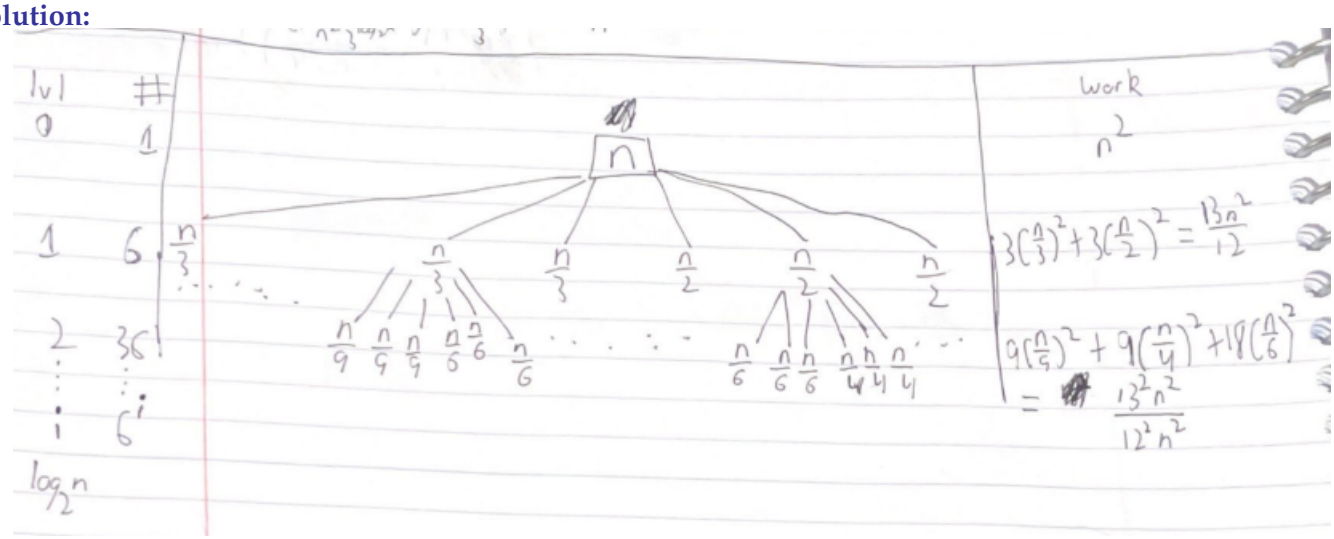
Consider the following recurrence:

$$T(n) = 3T(n/3) + 3T(n/2) + n^2$$
$$T(1) = C$$

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:



Hence, the work at level i can be represented by $\frac{13^i n^2}{12^i n^2}$.
Since $\frac{n}{3}$ is dominated by $\frac{n}{2}$, the final level will be $\log_2 n$
So we have:

$$n^2 \sum_{i=0}^{\log_2 n} \left(\frac{13}{12}\right)^i = n^2 \left(\frac{13}{12}\right)^{\log_2 n} = n^2 \frac{13^{\log_2 n}}{2^{\log_2 n} \cdot 2^{\log_2 n} \cdot 3^{\log_2 n}}$$

$$= n^2 \left(\frac{13^{\log_2 n}}{n^2 3^{\log_2 n}}\right) = \frac{13^{\log_2 n}}{3^{\log_2 n}} = \left(\frac{13}{3}\right)^{\log_2 n} = n^{\log_2(\frac{13}{3})}$$

Hence, $T(n) = O(n^{\log_2(\frac{13}{3})})$ as required.

[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:

The initial array is recursively split into two sub-arrays through the middle. The maximum sub-array (i.e. the sub-array that contains the max value of the sum of the differentials) of the left and right sub-arrays is found. When combining the sub-arrays, the maximum sub-array that is lower-bound in the left sub-array upper-bound in the right sub-array is found. The maximum of these three sub-arrays is returned.

Algorithm 1: Finds the range of indices (l, h) that output the greatest sum from $A[l]$ to $A[h]$ in A. Returns (lower bound, upper bound, sum of differentials between bounds)

Function $MaxDifferential(A[1...n], l, h)$**:**
- **If** $l == h$ **:**
  - **Return** $(l, l, A[l])$
- $mid = (l + h)/2$
- $(leftL, leftH, leftValue) = MaxDifferential(A, l, mid)$
- $(rightL, rightH, rightValue) = MaxDifferential(A, mid + 1, h)$
- $(combinedL, combinedH, combinedValue) = MaxCombined(A, l, mid, h)$
- **If** $leftValue \geq rightValue$ AND $leftValue \geq combinedValue$ **:**
  - **Return** $(LeftL, leftH, leftValue)$
- **If** $rightValue \geq leftValue$ AND $rightValue \geq combinedValue$ **:**
  - **Return** $(rightL, rightH, rightValue)$
- **Return** $(combinedL, combinedH, combinedValue)$

Function $MaxCombined(A[1...n], l, m, h)$**:**
- $LeftValue = A[m]$
- $sum = 0$
- $furthestLeftIndex = m$
- **For** int i from mid down to l:
  - $sum = sum + A[i]$
  - **If** $sum > leftValue$ **:**
    - $leftValue = sum$
    - $furthestLeftIndex = i$
- $rightValue = A[m + 1]$
- $sum = 0$
- $furthestRightIndex = m + 1$
- **For** int i from $mid + 1$ to h:
  - $sum = sum + A[i]$
  - **If** $sum > rightValue$ **:**
    - $rightValue = sum$
    - $furthestRightIndex = i$
- **Return** $(furthestLeftIndex, furthestRightIndex, leftValue + RightValue)$

(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:

Each function call splits the array into two sub-arrays of size n/2. This process will continue until each sub-array is of length 1. In the combine step, a linear search of time $O(n)$ is performed. Hence:

$T(n) = 2T(n/2) + cn$; $T(1) = 1$. Master Theorem: $T(n) = a \cdot T(\frac{n}{b}) + Cn^d$

$\frac{a}{b^d} = \frac{2}{2^1} = \frac{1}{1}$. Since $\frac{1}{1} = 1$, $T(n) = \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$

Thus, the running time of this algorithm is $\boxed{\text{T(n)} = \Theta(n \log n)}$

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:

"no index found"

Since A is sorted and contains distinct integers, every subsequent value $A[n]$ after 1 will be $> n$. So, there is no i such that $A[i] = i$

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

Solution:

"no index found"

Any value $A[k]$ where $k < n$ will be $< k$. So, there is no i such that $A[i] = i$

(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:

Algorithm 2: Finds i when A[i] = i, or "no index found" if there does not exist A[i] = i

```
Function BinarySearch(A[1...n], l, h):
    If h ≥ l :
        mid = l + (h − l)/2
        If A[mid] == mid :
            Return mid
        If A[mid] > mid :
            Return BinarySearch(A, l, mid - 1)
        Return BinarySearch(A, mid+1, h)
    Return "no index found"
```

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

Solution:

In each function call, two comparisons $A[mid] == mid$ and $A[mid] > mid$ with O(1) time complexity are made. So,

$T(n) = T(\frac{n}{2}) + 1 = T(\frac{n}{2}) + n^0$

$T(1) = 1$

Master Theorem: $T(n) = a \cdot T(\frac{n}{b}) + Cn^d$

$\frac{a}{b^d} = \frac{1}{2^0} = \frac{1}{1}$. Since $\frac{1}{1} = 1$, $T(n) = \Theta(n^d \log n) = \Theta(n^0 \log n) = \Theta(\log n)$

Thus, the running time of this algorithm is $\boxed{\text{T(n)} = \Theta(\log n)}$

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:

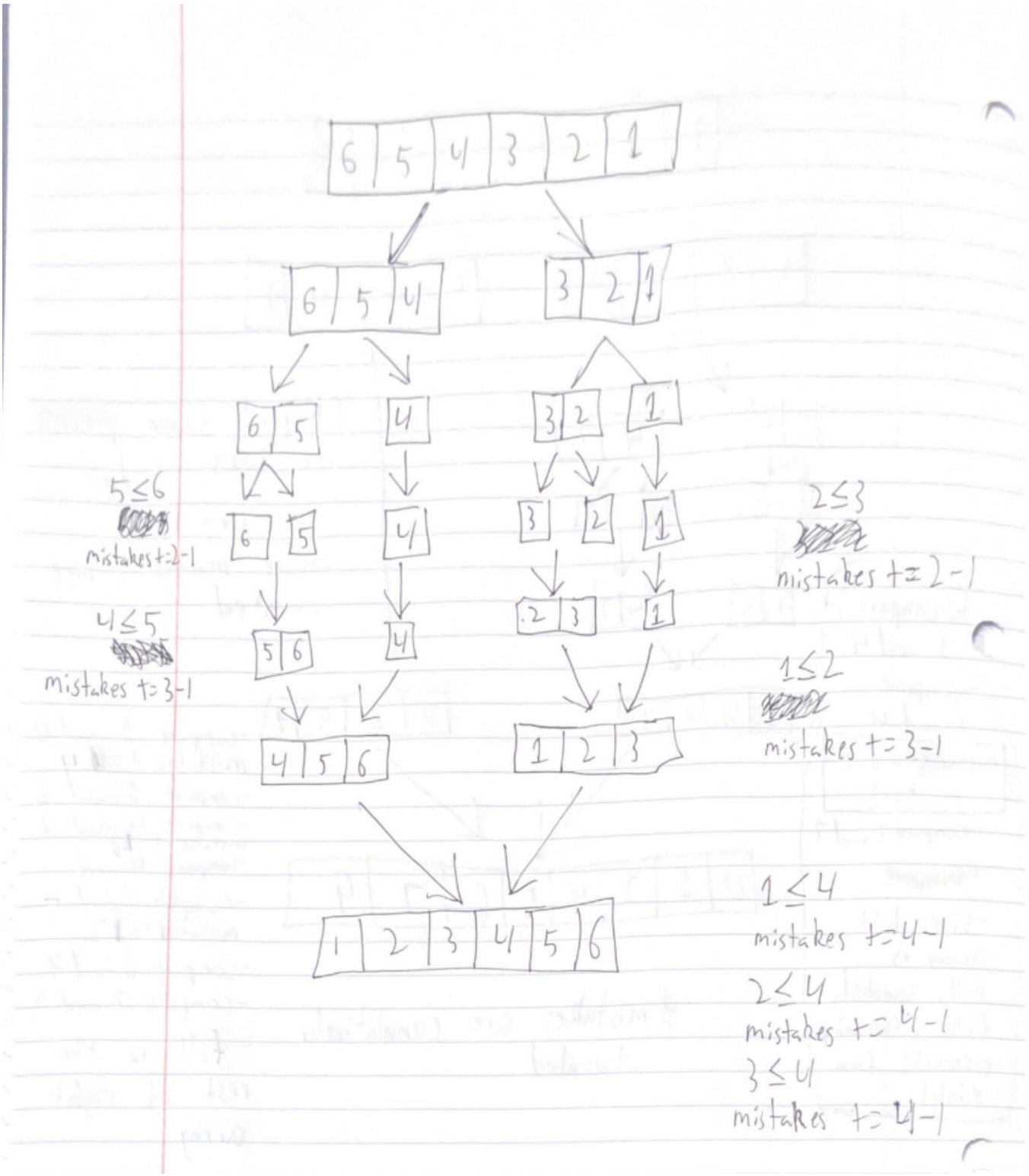
- Given an ordering of n potatoes, a mistake consists of two potatoes i and j such that potato i is before potato j in ther order, but i weighs more than j.

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:

This algorithm first splits each array into two sub-arrays of size $n/2$ until each sub-array is of size 1. The algorithm then recombines the sub-arrays so that they are sorted. When recombining, if a number (a potato's weight) from the right array is sorted before a number from the left array, this indicates at least one mistake is present. Since each sub-array has since been sorted, the number of mistakes is equal to the number of elements in the left array that have not been recombined yet. The cumulative number of mistakes made when recombining is equal to the total number of mistakes made. The image below roughly depicts this process.


6 5 4 3 2 1
6 5 4
3 2 1
6 5
4
3 2
1
5≤6
mistakes += 2-1
6 5
4
3
2
1
2≤3
mistakes += 2-1
4≤5
mistakes += 3-1
5 6
4
2 3
1
1≤2
mistakes += 3-1
4 5 6
1 2 3
1 2 3 4 5 6
1≤4
mistakes += 4-1
2≤4
mistakes += 4-1
3≤4
mistakes += 4-1

Algorithm 3: Performs Merge Sort on an array A and returns the number of mistakes in the original array. Note: this psuedocode was partially adapted from the Merge Sort algorithm provided in Lecture 4

```
Function Merge(A[1...n], l, h)):
    If l ≥ h :
        mistakes = 0
    Else
        mid = (l + h)/2
        mistakes = Merge(A, l, mid)
        mistakes = mistakes + Merge(A, mid + 1, h)
        mistakes = mistakes + MergeSortCountMistakes(A, l, mid, h)
    Return mistakes
Function MergeSortCountMistakes(A[1...n], l, mid, h)):
    Let B be an array of size (h-l)
    mistakes = 0
    counter = 0
    i = l
    j = mid
    While i < mid AND j < high
        If A[i] ≥ A[j] :
            B[counter] = A[i]
            i = i + 1
        Else
            B[counter] = A[j]
            j = j + 1
            mistakes = mistakes + mid − i
        counter = counter + 1
    While i < mid
        B[counter] = A[i]
        i = i + 1
        counter = counter + 1
    While j < h
        B[counter] = A[j]
        j = j + 1
        counter = counter + 1
    Return 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 worst-case running time occurs when the initial array is reverse sorted.

Each function call splits the array into two sub-arrays of size n/2. This process will continue until each sub-array is of length 1. In the combine step, a linear search of time

$O(n)$ is performed on A. Hence:

$T(n) = 2T(n/2) + cn$

$T(1) = 1$.

Master Theorem: $T(n) = a \cdot T(\frac{n}{b}) + Cn^d$

$\frac{a}{b^d} = \frac{2}{2^1} = \frac{1}{1}$. Since $\frac{1}{1} = 1$, $T(n) = \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$

Thus, the running time of this algorithm is $\boxed{\text{T(n)} = \Theta(n \log n)}$