Homework 2

CS 325

Daniel Kim

1. **Problem 1.**
   Give the asymptotic upper bounds for $T(n)$ in each of the following recurrences. Show your work and explain how you solve each case
   Problem 1.a. (2 points)

$T(n) = b \cdot T(n - 1) + 1$, where b is a fixed positive integer greater than 1.

Masters theorem:

T(n) = [coefficient]*T(n - b) + f(n)

for coefficient >0, b>0, f(n) = O(n^k), k >= 0

$T(n) = b \cdot T(n - 1) + 1$

For coefficient > 1,

O(n^k * coefficient ^{n})

- f(n) = 1 = $n^{k=0}$
- k=0
- b = coefficient >1

T(n) = $b^{(n)}$ *n^0

T(n) = O($b^{(n)}$)

Problem 1.b. (2 points)

• $T(n) = 3 \cdot T(n/9) + n \cdot \log n$

Master method for divide and conquer

T(n)= aT(n/b) + f(n)

a>=1 b>1 f(n) = $\theta(n^k * \log_n^p)$

$\log_9 3 = 1/2$, p = 1, k = 1

Since $\log_9 3 = 1/2 < k = 1$

$and\ p \geq 1,$ $\boldsymbol{T(n) = \Theta(n^{1/2} * log_n)}$

Type equation here.

Problem 2. (6 points)
Solve Exercise 4.1-5 from the 3rd Ed. of the textbook.

4.1-1

What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?

We assume that the array A contains all negative values. With each recursive call, FIND-MAXIMUM-SUBARRAY calls FIND-MAX-CROSSING SUBARRAY. FIND-MAX-CROSSING SUBARRAY returns the sum of two sub arrays ("cross sum" of which summing elements include both left and right subarrays)

With all the values in A negative, FIND-MAX-CROSSING SUBARRAY always returns a negative cross-sum that is less than the largest negative value in A.

```
FIND-MAXIMUM-SUBARRAY($A$, $low$, $high$)
 1  if $high == low$
 2      return ($low$, $high$, $A[low]$)          // base case: only one element
 3  else $mid = \lfloor (low + high)/2 \rfloor$
 4      ($left$-$low$, $left$-$high$, $left$-$sum$) =
            FIND-MAXIMUM-SUBARRAY($A$, $low$, $mid$)
 5      ($right$-$low$, $right$-$high$, $right$-$sum$) =
            FIND-MAXIMUM-SUBARRAY($A$, $mid + 1$, $high$)
 6      ($cross$-$low$, $cross$-$high$, $cross$-$sum$) =
            FIND-MAX-CROSSING-SUBARRAY($A$, $low$, $mid$, $high$)
 7      if $left$-$sum \geq right$-$sum$ and $left$-$sum \geq cross$-$sum$
 8          return ($left$-$low$, $left$-$high$, $left$-$sum$)
 9      elseif $right$-$sum \geq left$-$sum$ and $right$-$sum \geq cross$-$sum$
10          return ($right$-$low$, $right$-$high$, $right$-$sum$)
11      else return ($cross$-$low$, $cross$-$high$, $cross$-$sum$)
```

In lines 7-11, FIND-MAXIMUM-SUBARRAY compares the left-sum and right-sum with cross-sum and returns values according to the conditional statements.

Since the largest negative value of A either found on the left sub-array or right sub-array is always greater than the cross-sum (of both left and right sub-arrays), FIND- MAXIMUM-SUBARRAY returns the largest negative value of A and the respective index of that largest negative value.

Thus, if all the elements of A are negative, then FIND-MAXIMUM-SUBARRAY returns the largest negative value in A and respective index. Intuitively, any negative number added to a single negative number will only decrease the total value and the value is maximized by not summing additional negative numbers.

4.1-2

Write pseudocode for the brute-force method of solving the maximum-subarra
problem. Your procedure should run in $\Theta(n^2)$ time.

We can easily devise a brute-force solution to this problem: just try every possible
pair

INPUT: Array of A[1..n] of
OUTPUT: Subarray in an array of n values which make up the maximum sum

```
Brute-Force-Max-SubArray(A)
n = A.length
Max-sum = -infinity
for L = 1 to n
    sum = 0
    for H= L to n
    sum = sum + A[H]
    If sum > Max-sum
        low = L
        high = H
return (low, high, Max-sum)
```

First for loop goes from 1 to n => n iterations

2nd For loop goes from n, n-1, n-2, n -3, … , 2, 1 = n(n+2)/2 [arithmetic sum]

4.1-3

Implement both the brute-force and recursive algorithms for the maximum-subarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0. Does that change the crossover point?

N_0 Problem size of 86 is the point where dive and conquer outperforms Brute-force algorithm

	Array Size : 86	**Array Size : 87**
Brute-force	10,100	46,200
Divide-and-conquer	13,100	16,100
Hybrid	12,600	11,000

Yes, changing the base case of the recursive algorithm does change the crossover point. No problem size of 76 seems to be the defining crossover point. However, I am pretty sure other people will have a slight smaller No value because of the high amount of programs open might spike/skew the data.

	Array Size : 76	**Array Size :77**
Brute-force	7,600	12,800
Divide-and-conquer	17,700	11,600
Hybrid	50,300	12,400

4.1-4

Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

Add an if statement to check if all the elements of the array are negative or if the array is empty.

If the array is empty or all values are negative, return an empty array.

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward the right, keeping track of the maximum subarray seen so far. Knowing a maximum subarray of $A[1..j]$, extend the answer to find a maximum subarray ending at index $j+1$ by using the following observation: a maximum subarray of $A[1..j+1]$ is either a maximum subarray of $A[1..j]$ or a subarray $A[i..j+1]$, for some $1 \le i \le j+1$. Determine a maximum subarray of the form $A[i..j+1]$ in constant time based on knowing a maximum subarray ending at index j.

```
Iterative-Find-Max-SubArray(A)
    n = A.length
    max_sum = -infinity
    sum = -infinity
    for j = 1 to n
        current_high = j
        if sum > 0
            sum = sum + A[j]
        else
            current_low = j
            sum = A[j]
        if sum > max_sum
            max_sum = sum
            low = current_low
            high = current_high
    return (low, high, max_sum)
```

Problem 3.
Consider the recurrence $T(n) = 3 \cdot T(n/2) + n$.

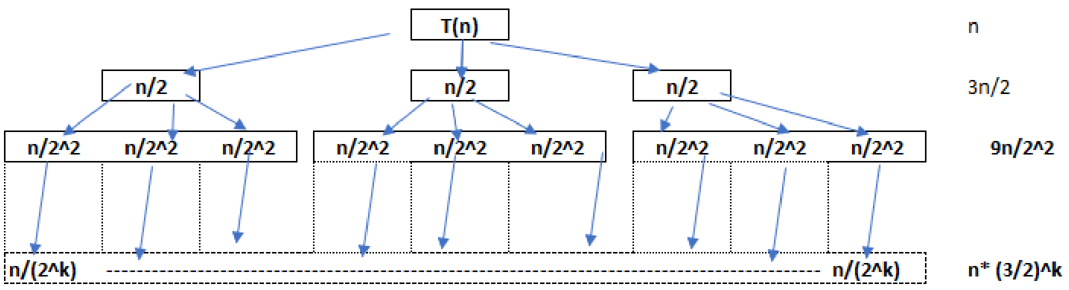
Problem 3.a. (3 points)

- Use the recursion tree method to guess an asymptotic upper bound for $T(n)$. Show your work.

Problem 3.b. (3 points)

- Prove the correctness of your guess by induction. Assume that values of n are powers of 2.

Consider the recurrence $T(n) = 3 \cdot T(n\text{-}2) + n$.



Subproblem for a node at depth i is n/2^i
levels = lg(n) + 1
leaves = lg(3)

T(n) =n * (1+ 3/2 + (3/2)^2 + ... + (3/2)^k)
T(n) =n * (1+ 3/2 + (3/2)^2 + ... + (3/2)^k)
Geometric progression
T(n) =n * ((3/2)^0 + (3/2)^1 + (3/2)^2 + ... + (3/2)^k) k = lg(n)

Solution continued in handwriting in next page.

$$T(n) = \sum_{i=0}^{(\lg n-1)} \left(\frac{3}{2}\right)^{i}\cdot n + \Theta(n^{\lg 3})$$

$$= \frac{(3/2)^{\lg n}-1}{(3/2)-1}\cdot n + \Theta(n^{\lg 3})$$

$$= 2\left[(3/2)^{\lg n}-1\right]\cdot n + \Theta(n^{\lg 3})$$

$$= 2\left[n^{\lg(3/2)} - 1\right]\cdot n + \Theta(n^{\lg 3})$$

$$= 2\left[n^{\lg 3-\lg 2} - 1\right]\cdot n + \Theta(n^{\lg 3})$$

$$= 2\cdot\left[n^{\lg 3-1+1} - n\right] + \Theta(n^{\lg 3})$$

$$= O(n^{\lg 3})$$

We guess ~~that~~ $T(n) \le c\cdot n^{\lg 3} - dn$

Then $T(n) = 3T([n/2]) + n$ $\quad \lg 3$

$$\le 3\cdot\left(c\cdot\frac{n}{2}^{\boxed{\lg 3}} - d\left(\frac{n}{2}\right)\right) + n$$

$$= \left(\frac{3}{2^{\lg 3}}\right)\cdot cn^{\lg 3} - \left(\frac{3d}{2}\right)\cdot n + n$$

where last step holds for $d \ge 2$ $\quad = cn^{\lg 3} + \left(1-\frac{3d}{2}\right)\cdot n.$

Problem 4.
Consider the following pseudocode for a sorting algorithm, for $0 < \alpha < 1$ and $n > 1$.

```
badSort($A[0 \cdots n-1]$)
          if ($n = 2$) and ($A[0] > A[1]$)
                    swap $A[0]$ and $A[1]$
          else if ($n > 2$)
                    $m = \lceil \alpha \cdot n \rceil$
                    badSort($A[0 \cdots m-1]$)
                    badSort($A[n-m \cdots n-1]$)
                    badSort($A[0 \cdots m-1]$)
```

Problem 4.a. (3 points)

- Show that the divide and conquer approach of badSort fails to sort the input array if $\alpha \leq 1/2$.

Problem 4.b. (2 points)

- Does badSort work correctly if $\alpha = 3/4$? If not, why? Explain how you fix it.

Problem 4.c. (2 points)

- State a recurrence (in terms of n and α) for the number of comparisons performed by badSort.

Problem 4.d. (2 points)

- Let $\alpha = 2/3$, and solve the recurrence to determine the asymptotic time complexity of badSort.

4a)

$0 < \alpha < 1$

With n elements, m will get the ceiling of α*n.

When α <=1/2, badSort fails to sort the array because n is being multiplied with a fractional number to obtain m. With α fractionally reducing the array recursively by a number less 1/2 the recursive calls can never catch up with the bigger reduction due to the α being less than 1/2.

4b)

4c)

4d)

Problem 5.

Problem 5.a. (3 points)

- **Implementation**: Implement badSort from Problem 4 to sort an array of integers. The value of α should be an input parameter to your program. Implement the algorithm in C/C++. Your program should be able to read inputs from a file called "data.txt", where the first value of each line is the number of integers that need to be sorted, followed by the integers. The output will be written to a file called "bad.out".

Problem 5.b. (3 points)

- **Modify code**: Modify the code to collect running time data. Call the new timing program badSortTime. Instead of reading arrays from the file data.txt and sorting, you will now generate arrays of size n containing random integer values from 0 to 10,000 to sort. Use the system clock to record the running times of each algorithm for $n = 5000, 10000, 15000, 20{,}000, \ldots$. for two values of $\alpha = 2/3$ and $\alpha = 3/4$. You may need to modify the values of n if an algorithm runs too fast or too slow to collect the running time data. Provide a table with the timing data.

Problem 5.c. (2 points)

- **Plot data and fit a curve**: Plot the running time data you collected for each value of $\alpha \in \{2/3, 3/4\}$ on an individual graph with n on the x-axis and time on the y-axis. You may use Excel, Matlab, R or any other software. How does your experimental running time compare to the theoretical running time of the algorithm?

Problem 5.d. (2 points)

- **Comparison**: Looking at the plots in the previous step for $\alpha \in \{2/3, 3/4\}$, which α provides better performance?

Submit a copy of all your code files and a README file that explains how to compile and run your code in a ZIP file to TEACH. We will only test execution with an input file named data.txt.