**CS375 Assignment 2 (Fall 2023)**

**(Due Sept 28, 2023 by 11:59pm)**

**Objective:**

1. Design and analyze an algorithm with D&C or recursion.
2. Establish recurrence equation and solve it.

There are two parts in this assignment: (A) Theory part and (B) programming part

**[Part A] Theory [78%]:**

1. (6%) We have a problem that can be solved by a direct (non-recursive) algorithm that operates in N2 times. We also have a recursive algorithm for this problem that takes NlgN operations to divide the input into two equal pieces and lgN operations to combine the two solutions together. Show whether the direct or the recursive version is more efficient. (Note: For the recursive algorithm, the base case is T(n) = 1 if n=1.)

**Answer:** The recursive version is more efficient because its time complexity is NlgN + lgN = O(NlgN + lgN) while the time complexity of the direct algorithm is O(N^2). This means that the recursive algorithm takes less time than the direct algorithm to run.

1. [10%]
   1. [3%]Write the recurrence equation for the code below. Use the number of comparisons as your barometer operation (The *min* operation requires 1 comparison and the *max* operation requires 1 comparison): *): (Note: you can count min and max as the comparison operation and skip the rt-lt <=1)*

MinMax(A,lt,rt)

// return a pair with the minimum and the maximum

if (rt - lt  1)

return (min(A[lt], A[rt]), max(A[lt],A[rt]));

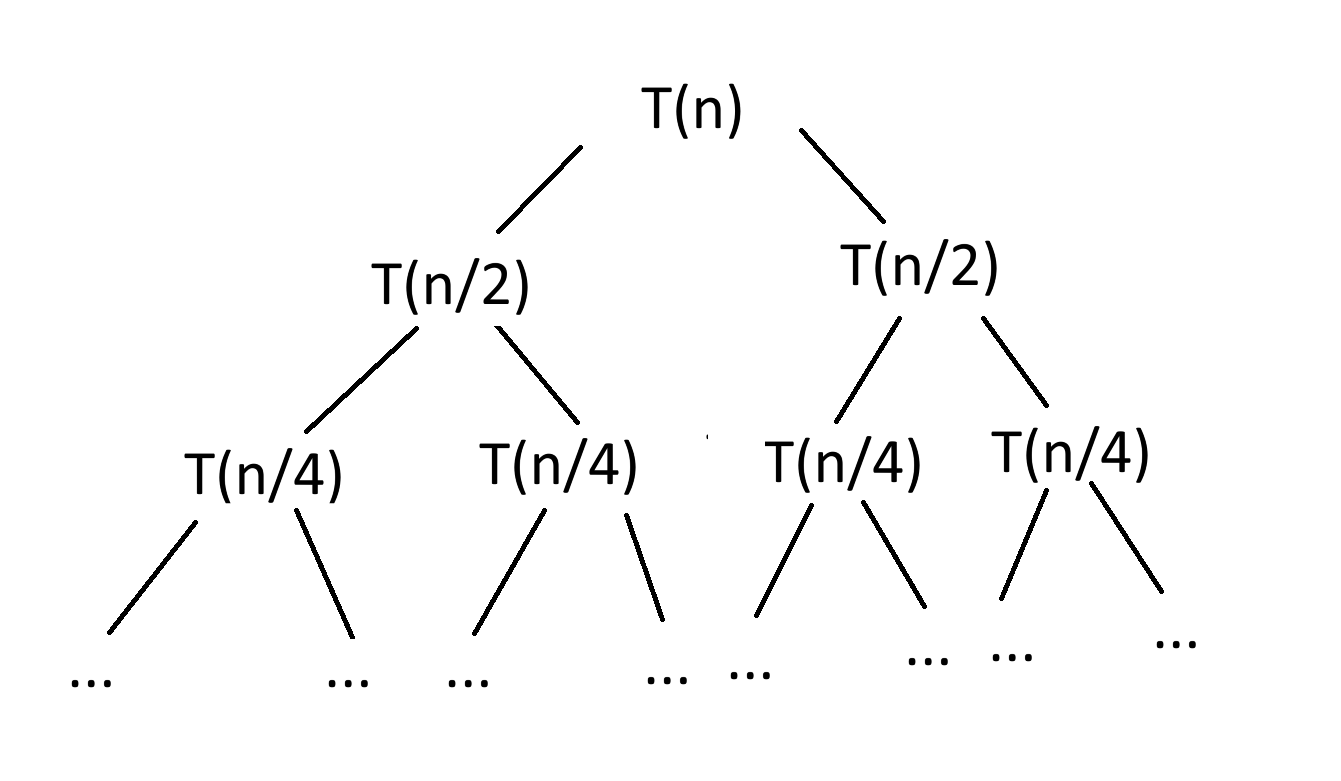
(min1, max1) = MinMax(A,lt, (lt+rt)/2 );

(min2, max2) = MinMax(A, c (lt+rt)/2 +1,rt);

return (min (min1, min2), max(max1, max2));

**Answer:** The recurrence equation is T(n) = T(n/2) + T(n/2) + 2, where T(n/2) represents the left side, T(n/2) represents the right side and +2 represents the calls for the initial minimum and maximum comparisons.

* 1. [4%] Show the recursion tree and solve the recurrence equation for this code. For simplicity assume n = 2*k*.

**Answer:** sum = (2 \* (1-2^(log2(n)+1))) / (1-2)

= 2 \* (1-2^n) / (-1)

= 2^n - 2

[3%] Prove the solution using induction

**Answer:**

Base Case: n = 1

When n = 1, T(1) = 2^1 -2 = 0, so sum = 2^n – 2 for n = 1.

Inductive Hypothesis: Assume that the equation is true for T(k) = 2k – 2 for all positive integers k <= n.

Inductive Step: n = 2k + 1

When n = 2k + 1, T(2k + 1) = T(2^k \* 2) = T(2^k) + T(2^k) + 2

= (2(2^k) - 2) + (2(2^k) - 2) + 2 (inductive hypothesis)

= 2(2^k + 2^k) - 2 - 2 + 2

= 2(2 \* 2^k) - 2

= 2(2^(k + 1)) – 2

Therefore, T(n) = 2n – 2 is the solution to the recurrence equation.

1. (8%) Design an algorithm for visible lines detection: Given *n* non-vertical lines in a plane, labeled *L1…., Ln*, with line equation *y=ai\*x +bi (i=1, …n).* We can make an assumption that no three of the lines all meet at a single point.

Definition: line *Li* is uppermost at a given x-coordinate *x0* if its y-coordinate at *x0* is greater than the y-coordinates of all the other lines at *x0*. Line *Li* is visible if there is some x-coordinate at which it is uppermost – intuitively, some portion of it can be seen if you look from infinity of y (*y = ∝)*; For example: following figure is an instance of visible lines (labeled 1-5). All the lines except for 2 are visible.



Give an algorithm that takes n lines as input and in no more than O(n^3) time returns all of the ones that are visible. Show the time complexity.

**Answer:** We can use a brute-force algorithm to solve this. The algorithm iterates over each line and checks if it is visible by comparing it with all the other lines at different x-coordinates. One function checks if a line is uppermost at a given x-coordinate, and another function finds and prints the visible lines.

The time complexity is O(n^3) because we iterate over a range of x-coordinates for each pair of lines, which is O(n^2), and we compare the lines for each x-coordinate, so together the overall time complexity is O(n^3).

1. (9%) Given a sorted array of distinct integers A[1, …, n], you want to find out whether there is an index i for which A[i]=i. Give a divide-and-conquer algorithm to solve this problem. Derive the time complexity. (Note: the running time much be less than O(n)).

**Answer:** The algorithm first initializes two pointers for the start and end of the array, called *left* and *right*. While *left* is less than or equal to *right*, the algorithm calculates the middle index, called *mid*. If *A[mid] = mid*, then there is an index *i* for which A[i] = i, so *mid* is returned. If *A[mid] > mid*, then *i* is on the left side of *mid*, so *right* is set to *mid – 1*. If *A[mid] < mid*, then *i* is on the right side of *mid*, so *left* is set to *mid + 1*. Otherwise, there is no *i* such that *A[i] = i*.

The time complexity is O(log n) since the range is halved at each iteration.

1. [10%] Write a piece of pseudo-code to plot the following graph. Assuming that the plotting function has been provided as DrawSquare(x, y, r), which draw a square of size 2r with center (x, y).
2. Pseudo-Code to plot following graph. (4%)

**Answer:**

DrawSquare(x, y, r) {

if r > 1 then

DrawSquare(x – r/2, y – r/2, r/2) // bottom-left

DrawSquare(x – r/2, y + r/2, r/2) // top-left

DrawSquare(x + r/2, y + r/2, r/2) // top-right

DrawSquare(x + r/2, y – r/2, r/2) // bottom-right

(2) Write the recurrence equation for your algorithm. (3%)

**Answer:** T(r) = 1 + 4T(r/2).

(3) Plot the recursion tree to derive the solution of T(r). (3%)

(Note: r is the input size, which is the power of 2. The time complexity for drawing one square is O(1)). (Hint: The input of function can be: int x, int y, int r)

**Answer:**

A black lines with white text

Description automatically generated

Total Nodes = 4^0 + 4^1 + 4^2 + 4^3 + … + 4^(log2(r)).

= (4^(log2(r) + 1) – 1) / (4 – 1) = (4^(log2(r) + 1) – 1) / 3

T(r) = O(1) \* Total Nodes

T(r) = O((4^(log2(r) + 1) – 1) / 3

r

x,y

1. (9%) An Array A[1,…,n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. Show how to solve this problem in O(nlgn) time.(Hint: Split the array A into two arrays A1 and A2 of half the size. Does knowing the majority elements of A1 and A2 help you figure out the majority element of A?) Note that it is required to use a divide-and-conquer approach with O(nlgn) time complexity.

**Answer:** First, divide the array *A* into two halves, *A1* and *A2*. Recursively find the majority elements of *A1* and *A2*, where the majority element of *A1* is *M1* and the majority element of *A2* is *M2*. If *M1* and *M2* are the same element, then this element may be the majority element in *A*. This is the case if it appears more than half the time in *A*. Or, if *M1* and *M2* are different elements, then we count the number of times they appear in *A*. Then, if either *M1* or *M2* is a majority element in *A*, the algorithm returns it as the majority element. This is the case if both are the majority element as well. If neither of them are a majority element in *A*, then there is no majority element in *A*.

1. [6%] Find the asymptotic bound of the divide and conquer recurrence T(n) using master method:
   * + 1. T(n) = 9\*T(n/3) + n2+4
       2. T(n) = 6\*T(n/2) + n2-2
       3. T(n) = 4\*T(n/2) + n3 +7

**Answer:**

1. a = 9, b = 3, and f(n) = n^2 + 4.

According to the master method, f(n) is a polynomial of degree k = 2. Since log\_b(a) = log\_3(9) = 2, we look at case 2 of the master method, where if f(n) is O(n^k), where k > 0, and a = b^k, then T(n) = O(n^k \* log n). In this case, k = 2, so T(n) = O(n^2 \* log n).

1. a = 6, b = 2, and f(n) = n^2 – 2.

According to the master method, f(n) is a polynomial of degree k = 2. Since log\_b(a) = log\_2(6) > 2, we look at case 1 of the master method, where if f(n) is O(n^k), where k > 0, and a > b^k, then T(n) = O(n^log\_b(a)). So, T(n) = O(n^log\_2(6)).

1. a = 4, b = 2, and f(n) n^3 + 7.

According to the master method, f(n) is a polynomial of degree k = 3. Since log\_b(a) = log\_2(4) = 2 < 3, we look at case 3 of the master method, where if f(n) is O(n^k), where k > 0, and a < b^k, then T(n) = O(f(n)). So, T(n) = O(n^3).

1. [9%] Design an algorithm to rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements. Your algorithm should be both time- and space-efficient. (Hint: use the partition idea, which is similar to the quick-sorting algorithm).

**Answer:** To rearrange elements of a given array of n real numbers so that all its negative elements precede all its positive elements, first initialize two pointers, with one starting at the beginning of the array and the other starting at the end of the array. These two pointers will move towards each other such that when the algorithm finds a negative element on the left side and a positive element on the right side, it will swap them, partitioning the array into negative and positive segments. This is repeated until the pointers meet, having all the negative elements precede the positive elements.

1. [5%] Solve the following recurrence equation using methods of characteristic equation.

T(n)=8T(n-1)-21T(n-2)+18T(n-3) for n>2

T(0)=0

T(1)=1

T(2)=2

*Hint: if there are two same roots, the solution template is T(n) = C1\*r1^n + C2\*r2^n + C3\*n\*r3^n*

**Answer:** The characteristic equation is found by substituting T(n) into the recurrence: T(n) = 8T(n-1)21T(n-2)+18T(n-3), where we get 8(C1 \* r1^(n-1) + C2 \* r2^(n-1) + C3 \* (n-1) \* r3^(n-1)) - 21 \* (C1 \* r1^(n-2) + C2 \* r2^(n-2) + C3 \* (n-2) \* r3^(n-2)) + 18 \* (C1 \* r1^(n-3) + C2 \* r2^(n-3) + C3 \* (n-3) \* r3^(n-3)). Divide this by r3^n to get 8 \* (C1 \* (r1/r3)^(n-1) + C2 \* (r2/r3)^(n-1) + C3 \* (n-1)) - 21 \* (C1 \* (r1/r3)^(n-2) + C2 \* (r2/r3)^(n-2) + C3 \* (n-2)). Next, we find the roots r1, r2, and r3:

r^3 = 8r^2 – 21r + 18. Factor this to get (r -2)(r – 3)(r – 3) = 0, so r1 = 2, r2 = 3, and r3 = 3. r2 and r3 are the same, so we find C1, C2, and C3:

T(0) = 0: C1 + C2 + C3 \* 0 = 0 -> C1 + C2 = 0

T(1) = C1\*2 + C2\*3 + C3\*1\*3 = 1 -> 2C1 + 3C2 + 3C3 = 1

T(2) = 2: C1\*4 + C2\*9 + C3\*2\*9 = 2 -> 4C1 + 9C2 + 18C3 = 2

Lastly, we solve for the constants C1, C2, and C3 by using the simplified equations above. By the first equation, we find that C1 = -C2. Substitute this into the other equations to get:

2(-C2) + 3C2 + 3C3 = 1 -> C2 + 3C3 = 1

4(-C2) + 9C2 + 18C3 = 2 -> 5C2 + 18C3 = 2

Now, we solve for the second equation, where C2 = 1 – 3C3, and then substitute it for the third equation:

5(1 – 3C3) + 18C3 = 2 -> 5 – 15C3 + 18C3 = 2 -> 3C3 = -3

From this, we find that C3 = -1, and use this to solve for C2:

C2 = 1 – 3(-1) = 4

Then, we use this to solve for C1:

C1 = -(4) = -4

So, the solution to the recurrence equation is T(n) = -4\*2^n + 4\*3^n - 1\*n\*3^n.

10. [6%] Solve the following recurrence equations using recursion tree technique discussed in class. You may make any assumptions about *n*, such as to assume that *n* is an exact power of 2. (hint: Base case: T(0) = T(1) = Theta (1))

T(n)=T(n/2)+T(n/4)+T(n/8)+n

At each level of the recursion tree, the problem size is divided by 2. So, the first level starts with size n. At the second level, the problem size is n/2, at the third level, the problem size is n/4, and so on. Adding this together, we get T(n) = n + n/2 + n/4 + n/8 + …, which is a geometric series of common ratio r = ½. This can be calculated by S = a / (1 – r), where a is the first term and r is the common ratio. In this case, a = n and r = ½, so T(n) = n / (1 – ½) = 2n. So, the solution to the recurrence equation T(n) = T(n/2)+T(n/4)+T(n/8)+n is T(n) = Θ(n).

**[Part B]: Divide and Conquer Programming (22%)**

1. [12%] Implement the algorithm for finding the closest pair of points in two dimension plane using divide and conquer strategy.

A system for controlling air or sea traffic might need to know which are the two closest vehicles in order to detect potential collisions. This part solves the problem of finding the closest pair of points in a set of points. The set consists of points in R2 defined by both an x and a y coordinate. The "closest pair" refers to the pair of points in the set that has the smallest Euclidean distance, where Euclidean distance between points p1=(x1,y1) and p2=(x2,y2) is simply sqrt((x1-x2)2+(y1-y2)2). If there are two identical points in the set, then the closest pair distance in the set will obviously be zero.

Input data: n points with coordinates:

X coordinates: p[0].x, p[1].x, p[2].x,…., p[n].x

Y coordinates: p[0].y, p[1].y, p[2].y,…., p[n].y

Output: minimum distance between points p[i] and p[j] (index i and j should be identified)

Input data for test:

n = 10000;

for(i=0; i<n; i++)

{

p[i].x= n- i;

p[i].y= n- i;

}

Other Input data for test:

n = 10000;

for(i=0; i<n; i++)

{

p[i].x= i\*i;

p[i].y= i\*i;

}

Or:

Input:

If X>0

X=1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, ……,19995, 19997, 19999 (odd numbers)

Y=int( )

Else

X=0, -2, -4, -6, -8, -10, -12, ……, -19996, -19998, -20000 (even number)

Y= int( )

2. (10%) Use the brute-force approach to solve the above problem. Compare the two implemented algorithms in terms of time complexity. Print out the time cost during the execution of these two algorithms.

3. (Optional: Extra 10%) apply the close-pair algorithm to the three dimensional case.

Note 1:

Your report (.doc) of the programming part should follow the following format:

1. Algorithm description
2. Major codes
3. Running results
4. Report of any bugs

Note 2:

2.1 For Part B, your code package includes your code, makefile, executable file, and write-up (.doc).

2.2 Your program will read an input file and write an output file.

2.3 Your program should be invoked like this…

prompt>submission inputFile.txt outputFile.txt

where inputFile.txt is referring to an input file,

ouputFile.txt is referring to an output file.