CSE373, MAT373: Analysis of Algorithms, Fall 2022 **Date:** 10/23/22

Homework #2

(Due: 10/23/22 **)**

Group Number: 6

Group Members		
Name	SBU ID	% Contribution
Michael Lee	112424954	50
Christopher Lobato	114661869	50

CSE 373 HW2

1. a) If C is a n x n matrix where $n = 2^k$, for $k \geq 0$ ints,

$$C = A \times B \rightarrow \begin{array}{c} C \\ \begin{bmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{bmatrix} \end{array} = \begin{array}{c} A \\ \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \end{array} \times \begin{array}{c} B \\ \begin{bmatrix} c & c \\ c & c \end{bmatrix} \end{array}$$

$$= \begin{bmatrix} x_{11}c + x_{12}c & x_{11}c + x_{12}c \\ x_{21}c + x_{22}c & x_{21}c + x_{22}c \end{bmatrix}$$

We can divide each of the 8 matrix multiplication into another recursive call of $n/2$ matrix as $n = 2^k$. However, since ~~the~~ B is a matrix of constants we can see that only 4 matrix multiplications need to be made as z_{i1} and z_{i2} are duplicates.

```
mm(X, C) {
    if n = 1 {
        z = X · C
    }
    else {
        z_11 = mm(X_11, C) + mm(X_12, C)
        z_21 = mm(X_21, C) + mm(X_22, C)
        z_12 = z_11
        z_22 = z_21
    }
}
```

b)

$$\text{So, } T(n) = \begin{cases} \Theta(1) & n = 1 \\ \underbrace{4T(n/2)}_{\text{4 mm calls made.}} + \underbrace{\Theta(n)}_{\text{as multiplying by constant is } \Theta(n)} & \text{else} \end{cases}$$

# of leaves $= n^{\log_2 4} = n^2$

so $T(n) = \Theta(n^2)$
(case 1 applied)

2

a) $n > 0$,

$$T(n) = \begin{cases} 1 & n=1, \\ T(n-1) + n^{2022} & \text{else} \end{cases}$$

as $a = 1$, $b = 1$, # of leaves $= n^1 = n$.

# of leaves is asymptotically smaller than $f(n) = n^{2022}$.
Case 3 is applied and $\boxed{T(n) = \Theta(n^{2022})}$

b) $n = 2^{2^k}$,

$$T(n) = \begin{cases} 1 & n=2, \\ T(\sqrt{n}) + 1 & \text{else} \end{cases}$$

$T(2) = 1$

$k=1$, $T(4) = T(2) + 1 = 2$

$k=2$, $T(16) = T(4) + 1 = 3$

$k=3$, $T(256) = T(16) + 1 = 4$ or, $T(n) = \log_2(\log_2(n)) + 1$

Therefore, $\boxed{T(n) = \Theta(\log(\log(n)))}$

c) $n = 2^k$,

$$T(n) = \begin{cases} 1 & n=1, \\ 3T(n/2) + n & \text{else} \end{cases}$$ $a = 3$, $b = 2$, $f(n) = n$

using master theorem, # of leaves $= n^{\log_2 3} \approx n^{1.58}$

as # of leaves is asymptotically larger than $f(n)$, case 1 is applied and $\boxed{T(n) = \Theta(n^{\log_2 3})}$

d) $n = 2^k$

$$T(n) = \begin{cases} 1 & n = 1, \\ 4T(n/2) + n^2 & \text{else} \end{cases} \qquad a = 4,\ b = 2,\ f(n) = n^2$$

by master theorem, # of leaves = $n^{\log_2 4} = n^2$

neither # of leaves or f(n) dominates so case 2 is applied.

$$\boxed{T(n) = \Theta(n^2 \log n)}$$

3a

As we know the algorithm for the merging step of Merge Sort is as follows

Merge (A, p, q, r)

```
1.  n1 = q – p + 1
2.  n2 = r – q +
3.  Let L[1:n1 + 1] and R[1:n2 + 1] be the new arrays
4.  for i = 1 to n1
5.      L[i] = A[P + i – 1]
6.  for j = 1 to n2
7.      R[j] = A[q + j]
8.  L[n1 + 1] = inf
9.  L[n2 + 1] = inf
10. i = 1
11. j = 1
12. for k = p to r
13.     if L[i] <= R[j]
14.         A[k] = L[i]
15.         i = i +1
16.     else A[k] = R[j]
17.         j = j + 1
```

Additionally we know lines 4-5 and 6-7 each take *O*(n1)and *O*(n2) respectively as at most we will need to visit all the indices of t he sub arrays. 8-12 are constant time *O*(1). Usually for lines 12-17 we would consider the step to Worst Case to be *O*(n) as we loop through both subarrays and compare the elements at each. However, if time needed to compare the two elements at each sub array is *O*(n) then this would change the worst-case time complexity at line 13. We originally assume that the comparison takes constant time O(1) and since we do this at most n times worst case is *O*(n*1) = *O*(n). Then we can say that instead when comparing takes *O*(n) then lines 12-17 would take *O*(n*n) = O(n^2) as we are now doing n operations for each of the n elements in the worst case. Lines 14,15,17 are assignments, and we assume that it is still constant time. In all merge step is *O*(n)+ *O*(n)+O(n^2) + *O*(1) = *O*(n^2). We know that for Merge-Sort we are creating two subproblems at each split where each subproblem is half the size of the previous. We can use Master Theorem to solve for the following recurrence **T(n) = 2T(n/2) + C(n)** where C(n) is O(n^2). This falls under case 3 since we can represent f(n) = n^2 as $f(n) = n^{log_b a+1}$ so we know that the first level dominates, and worst-case time complexity is ***O*(n^2).**

3b.

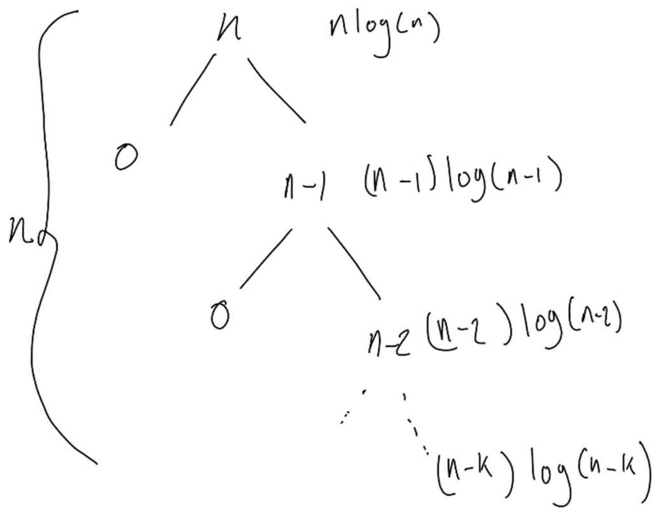
Merge-Sort (A, p, r)

```
1. if p < r then
2. q = floor(p+r/2)
3. Merge-Sort(A, p, q)
4. Merge-Sort(A, q+1, r)
5. Merge(A, p, q, r)
```

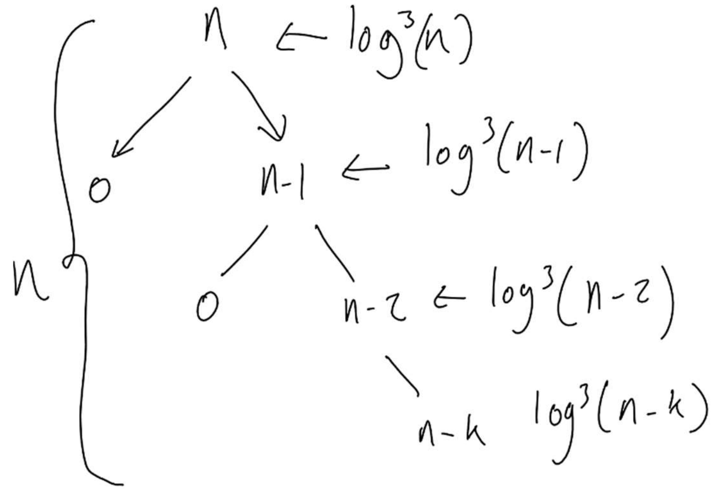
Like the previous problem we can look at the algorithm for Merge Sort and we know that it splits the problem into 2 subproblems of half size each time. In this case however we are considering that the time it takes to merge is $O(log^{2022}(n))$ therefore like in the previous problem the recurrence relation will be the following **T(n) = 2T(n/2) + C(n)** but f(n) is $log^{2022}(n)$. In this case we know that f(n) is of the form $n^k\ log^p(n)$ where k = 0 and p = 2022. Therefore since $log_b\ a$ is greater than k this falls under case 1 so worst case time complexity for this new algorithm is, $O(n^{log_2 2})$ = ***O*(n)**

3c.

For QuickSort to go through the Worst Case scenario the input array would need to be already sorted and therefore we would not have sub problems of equal size. For example:



In this case the partition step is *O*(nlogn), for simplicity we can drop the constant k and consider that each step we do nlogn partitions for n steps since we choose the pivot at each of the n elements because array is already sorted. Therefore the summation of this would O(n*nlogn), so worst case time complexity would be ***O*((n^2)log(n))**. To find average time complexity we can solve the recurrence relation T(n) = 2T(n/2) + *O*(nlogn) using master theorem since now the partitions are more even and we are creating two subproblems at each step. This falls under case 2 and since p is equal to 1 and is greater than -1 the average case time complexity is $\boldsymbol{\Theta\ (n\ log^2(n))}$**.**
Similar two the first part of problem 3c the worst case for quicksort happens when the array is already sorted and the sub problems are broken up as follows,



In this case however the partition step takes $O(log^3(n))$ time. Similarly to the previous problem we do this partition n times and the summation of this is $O(n*log^3(n))$ meaning worst case time complexity is

$\boldsymbol{O(nlog^3(n))}$. For average time complexity we consider the recurrence relation since partitions are now more evenly split as it is not worst case scenario, T(n) = 2T(n/2) + O ($log^3(n)$) and using master theorem we can solve this recurrence relation which falls under the following case $log_a b = k, log_2 2 > 0$,so average case time complexity is **Θ (n)**.

4a.

FindParetoPoints (A, p, q, r)

```
1. if p < r then
2. q = floor(p+r/2)
3. FindParetoPoints (A, p, q)
4. FindParetoPoints (A, q+1, r)
5. ParetoMege (A, p, q, r)
```

ParetoMerge (A, p, q, r)

```
1.  n1 = q – p + 1
2.  n2 = r – q
3.  Let L[1: n1+1] and R[1: n2+1] be new arrays
4.  for i = 1 to n1
5.      L[i] = A[pj + i – 1]
6.  for j = 1 to n2
7.      R[j] = A[q + i – 1]
8.  L[n1 + 1] = inf
9.  R[n2 + 1] = inf
10. i = 1
11. j = 1
12. Aux[1: r-p]
13. for k = 1 to (r – p)
14.     if L[i][0] >= R[j][0] //goes through this if left element is to the right of the right element
15.        if L[i][0] == R[j][0]
16.            Aux[k] = (L[i][0], max(L[i][1], R[j][0])) // choose one point if points have the same x
17.         else
18.            Aux[k] = L[i]
19.             i = i + 1
20.     else Aux[k] = R[j]
21.         j = j + 1
22. i = 2
23. A[p] = Aux[0]
24. for k = p to r
25.     if Aux[i][1] > Aux[i-1][1] // checks if the following point has a greater y
26.         A[k] = Aux[i]//
27.         i = i +1
28.     else
29.         A.remove(Aux[i])
```

This algorithm follows a similar way of thinking as the Merge-Sort Algorithm. In our Find Pareto Points function we split our input array into halves until we get a trivial case in which we only have one point which is trivially a pareto point. At each step/split we call the Pareto Merge function. Lines 4-5 and 6-7 are O(n1) and O(n2) respectively where n1 and n2 are the sizes of the sub arrays. For lines 13-21 we have to choose our points putting them in descending order into an auxiliary and make sure not to add points with the same x. These operations are done at each of the n elements so 12-17 takes O(n) time. Additionally we do another O(n) traversal in which we add the points from the auxiliary array into our input array and remove points that are not a part of the Pareto Set. So our Pareto Merge step takes O(n) + O(n) + O(n1) + O(n2) = O(n). However we have to split the array and using master theorem we can solve for the Following recurrence **T(n) = 2T(n/2) + C(n)** where C(n) is O(n). In this case it follows the same case as merge-sort(case 2) and we get that our worst case time complexity is O(nlog(n)).