Course – Analysis and Design of Algorithms

Unit 3:Divide-and-Conquer

Unit 2:Divide-and-Conquer

- Master’s Theorem
- Merge Sort
- Quick Sort
- Multiplication of Large Integers and Strassen’s Matrix Multiplication

Divide-and-Conquer Strategy Inspiration with example of finding Defective

- Brute-Force algorithms were giving us the most straight forward approach to solve problems but they were not efficient.

Identifying Defective Coin Exam

- We are give a set of coins
- Exactly one coin is defective: it has lesser weight
- Which one is it?

- A brute force algorithm
  - Compare weights of coins 1 and 2
  - If one weights less, return it
  - Else compare coins 3 and 4
  - .....

Defective Coin Example n=16

- Compare first two coins, if weights are same then eliminate them

After one comparison

- Compare weights of next two coins

After two comparisons

After three comparisons

After four comparisons

After five comparisons

- But at the sixth comparisons will give us a mismatch

Cost of Brute Force Defective coin

- How many comparisons do we make?
- Worst case if the defective coin was in the last pair so that we make (n/2) comparisons. Hence it is O(n)

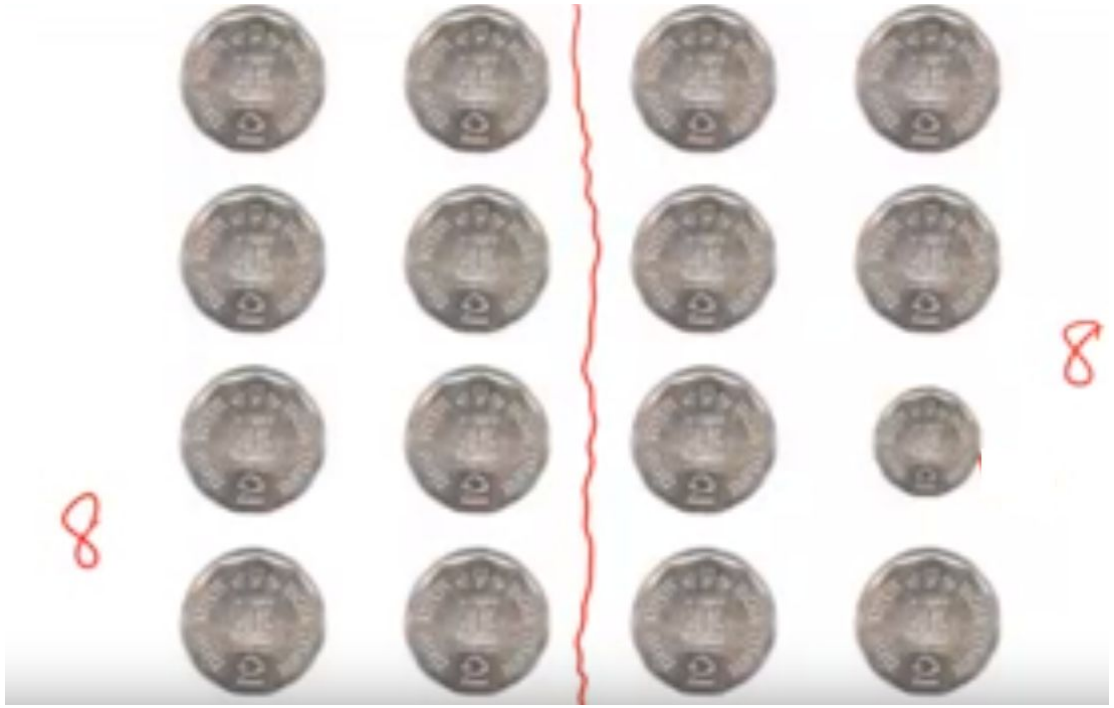
- Can we do better?

Motivation to Divide and Conquer Startegy

- Defective Coin Example n=16

Defective Coin Example n=16

- Divide eight coins on the left and eight coins on right. And then compare the weights.
- IN the first comparison we will come to know that weights of the first subset of eight coins is greater than remaining second subset. Hence we can eliminate first subset.



After one comparison

- After one comparison we can throw away eight coins

After one comparison

- Now apply the strategy recursively i.e., divide coins into two subsets of four coins each.
- Now after second comparison we can throw away first subset of four coins because weight of first top four coins are greater than the weights of bottom four coins.

After two comparisons

After three comparisons

- After three comparisons we are left with only two coins, now we can easily identify the defective coin.

After four comparisons

- We are left with only one defective coin

- Using new startegy i.e., Divide-and-Conquer number of comparisons are
- If $n=2^k$, then it is maxium of k comparisons i.e., O(k) and not O(n) which was for Brute-force strategy
- $O(k) = O(\log_2 n)$ because $k=\log_2 n$

Question

If we are not required to identify the defective coin, but only to say whether all the coins have the same weight or there is exactly one defective coin, which of the following is true (assuming there is never more than one defective coin and number of coins $n=2^k$) ?

A. We still need $log_2 n$ comparisons in the worst case

B. We need only *one* comparison

C. We need n comparisons in the worst case

D. We need $log_2 n$ comparisons in the best case

Answer

If we are not required to identify the defective coin, but only to say whether all the coins have the same weight or there is exactly one defective coin, which of the following is true (assuming there is never more than one defective coin and number of coins $n=2^k$) ?

A. We still need $log_2 n$ comparisons in the worst case

B. We need only *one* comparison

C. We need n comparisons in the worst case

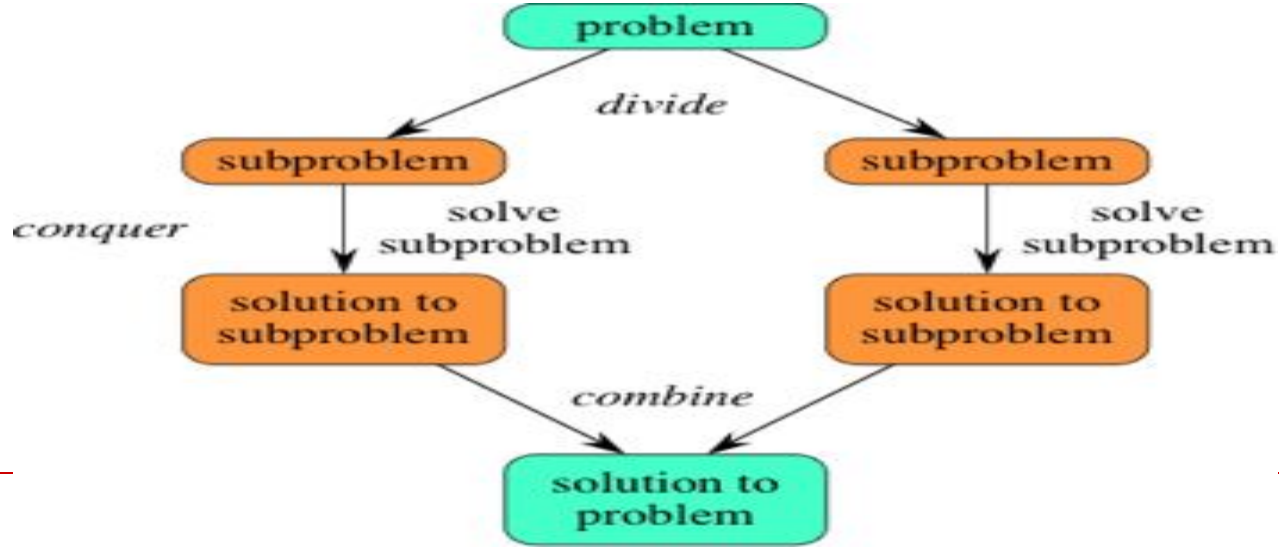
D. We need $log_2 n$ comparisons in the best case

Divide-and-Conquer

It is a strategy that suggests splitting the inputs into k distinct subsets 1<k<=n, yielding k sub-problems. These sub-problems must be solved and then a method must be found to combine sub-solutions into a solution of the whole.

Divide-and-Conquer

- **Divide-and-Conquer**, **breaks a problem into subproblems** that are similar to the original problem, **recursively solves the subproblems**, and finally **combines the solutions to the subproblems** to solve the original problem.
- Divide-and-conquer algorithms have three parts:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
  - **Combine** the solutions to the subproblems into the solution for the original problem.



The Idea of Divide-and-Conquer

- Given a problem P of size $\mathbf{n = 2^k}$,
- Algorithm DAndC (P)
  - if **n** is small, solve it (e.g., using brute-force);
  - else, divide P into two sub-problems P_1 and P_2

    // of size $\mathbf{n/2 = 2^{k-1}}$
  - DAndC(P_1); // Solve each sub-problem recursively
  - DAndC(P_2);
  - Combine solutions to sub-problems P_1 and P_2

Example

- A Divide-and-Conquer algorithm to find sum of n numbers stored in an array.

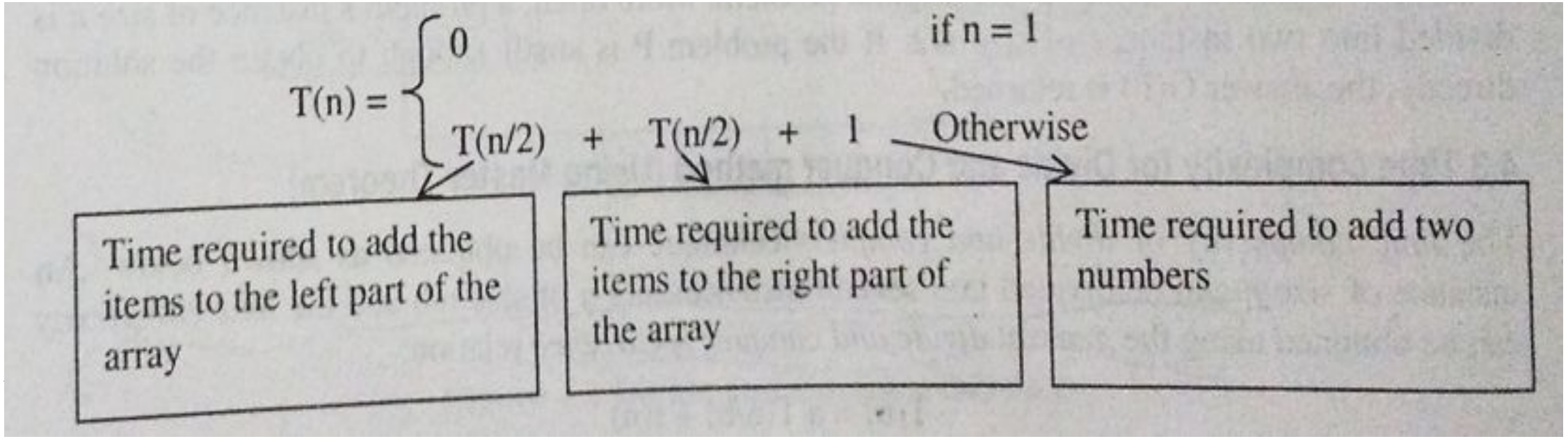
Algorithm

```
Algorithm Sum(low,high,a)
if (low==high)
    return a[low]
mid=(low+high)/2
return Sum(low,mid,a)+Sum(mid+1,high,a)
```

Algorithm

```
Algorithm Sum(low,high,a)
if (low==high)
      return a[low]
mid=(low+high)/2
return Sum(low,mid,a)+Sum(mid+1,high,a)
```

Time Analysis: It is clear from the algorithm that the problem instance is divided into two equal parts. Let us assume the number of elements in the array a are integral multiple of 2 i.e. $n=2^k$. The recurrence relation for this algorithm is shown below:



$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(n/2) + T(n/2) + 1 & \text{Otherwise} \end{cases}$$

- $T(n/2)$: Time required to add the items to the left part of the array
- $T(n/2)$: Time required to add the items to the right part of the array
- 1: Time required to add two numbers

Time complexity using backward substitution: Consider the relation

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + 1$$

$$= 2T\left(\frac{n}{2}\right) + 1$$

$$= 2\left[2T\left(\frac{n}{4}\right) + 1\right] + 1$$

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + 1\right] + 2 + 1 = 2^3 T\left(\frac{n}{2^3}\right) + 2^2 + 2 + 1$$

.....

$$= 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} + 2^{k-2} + \ldots\ldots 2^3 + 2^2 + 2 + 1$$

$$= 2^k T\left(\frac{n}{n}\right) + 2^{k-1} + 2^{k-2} + \ldots\ldots 2^3 + 2^2 + 2 + 1$$

$$= 2^k T(1) + 2^{k-1} + 2^{k-2} + \ldots\ldots 2^3 + 2^2 + 2 + 1$$

$$= n * 0 + \frac{a(r^n - 1)}{r - 1} = \frac{1(2^k - 1)}{2 - 1} = 2^k = n$$

$\therefore$, Time complexity to find sum of n numbers is given by $T(n) = \theta(n)$

Note: $T(n) = 2T\left(\frac{n}{2}\right) + 1$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 1$ by replacing n by n/2

$T\left(\frac{n}{2^2}\right) = 2T\left(\frac{n}{2^3}\right) + 1$

OR for $n=2^k$

$=2^kT(n/2^k)+2^{k-1}+2^{k-2}+....2+1$
$=2^kT(n/n)+2^{k-1}+2^{k-2}+....2+1$
$=2^kT(1)+2^{k-1}+2^{k-2}+....2+1$
$=2^{k-1}+2^{k-2}+....2+1$
$=2^k-1 = n-1$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Quiz

Consider the polynomial $p(x) = a_0 + a_1 * x + a_2 * x^2 + a_3 * x^3$, where $a_i != 0$, for all i.
The minimum number of multiplications needed to evaluate p on an input x is:

(A) 3
(B) 4
(C) 6
(D) 9

Quiz

Consider the polynomial p(x) = a_0 + a_1 * x + a_2 * x^2 + a_3*x^3, where a_i != 0, for all i. The minimum number of multiplications needed to evaluate p on an input x is:

(A) 3
(B) 4
(C) 6
(D) 9

Answer: (A)

Explanation: Multiplications can be minimized using following order for evaluation of the given expression.

p(x) = a_0 + x(a_1 + x(a_2 + a_3x))

Try

Using Divide-and-Conquer strategy, write your own pow(x, n) to calculate x^n

- Given two integers x and n, write a function to compute x^n.
- Input : x = 2, n = 3 Output : 8
- Input : x = 7, n = 2 Output : 49

Example

Write your own pow(x, n) to calculate x^n

- Given two integers x and n, write a function to compute x^n. We may assume that x and n are small and overflow doesn't happen.
- Input : x = 2, n = 3 Output : 8
- Input : x = 7, n = 2 Output : 49

Below solution divides the problem into subproblems of size n/2 and call the subproblems recursively.

```
#include<stdio.h>
 /* Function to calculate x raised to the power y */
int power(int x, int n) {
    if (n == 0)
        return 1;
    else if (n%2 == 0)
        return power(x, n/2)*power(x, n/2);
    else
        return x*power(x, n/2)*power(x, n/2);
}
int main() { int x = 2;  unsigned int n = 3; printf("%d", power(x, n));
    return 0; }
```

Example

Above function can be optimized by calculating power(x, y/2) only once and storing it.

```
/* Function to calculate x raised to the power y in O(logn)*/
int power(int x, unsigned int y)
{
    int temp;
    if( y == 0)
        return 1;
    temp = power(x, y/2);
    if (y%2 == 0)
        return temp*temp;
    else
        return x*temp*temp;
}
```

Homework Problem

a. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.

b. What will be your algorithm's output for arrays with several elements of the largest value?

c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

d. How does this algorithm compare with the brute-force algorithm for this problem?

Homework Problem

a. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.

b. What will be your algorithm's output for arrays with several elements of the largest value?

c. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

d. How does this algorithm compare with the brute-force algorithm for this problem?

a. Call **Algorithm** $MaxIndex(A[0..n-1])$ where

Algorithm $MaxIndex(A[l..r])$
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The index of the largest element in $A[l..r]$
if $l = r$ **return** l
else $temp1 \leftarrow MaxIndex(A[l..\lfloor (l+r)/2 \rfloor])$
$temp2 \leftarrow MaxIndex(A[\lfloor (l+r)/2 \rfloor + 1..r])$
if $A[temp1] \ge A[temp2]$
return $temp1$
else return $temp2$

a. Call **Algorithm** $MaxIndex(A[0..n-1])$ where

Algorithm $MaxIndex(A[l..r])$
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The index of the largest element in $A[l..r]$
if $l = r$ **return** l
else $temp1 \leftarrow MaxIndex(A[l..\lfloor (l+r)/2 \rfloor])$
$temp2 \leftarrow MaxIndex(A[\lfloor (l+r)/2 \rfloor + 1..r])$
if $A[temp1] \ge A[temp2]$
return $temp1$
else return $temp2$

b. This algorithm returns the index of the leftmost largest element.

c. The recurrence for the number of element comparisons is

$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \qquad C(1) = 0.$$

Solving it by backward substitutions for $n = 2^k$ yields the following:

$$\begin{aligned} C(2^k) &= 2C(2^{k-1}) + 1 \\ &= 2[2C(2^{k-2}) + 1] + 1 = 2^2 C(2^{k-2}) + 2 + 1 \\ &= 2^2[2C(2^{k-3}) + 1] + 2 + 1 = 2^3 C(2^{k-3}) + 2^2 + 2 + 1 \\ &= \dots \\ &= 2^i C(2^{k-i}) + 2^{i-1} + 2^{i-2} + \dots + 1 \\ &= \dots \\ &= 2^k C(2^{k-k}) + 2^{k-1} + 2^{k-2} + \dots + 1 = 2^k - 1 = n - 1. \end{aligned}$$

Homework Problem

a. Write a pseudocode for a divide-and-conquer algorithm for finding values of both the largest and smallest elements in an array of n numbers.

b. Set up and solve (for $n = 2^k$) a recurrence relation for the number of key comparisons made by your algorithm.

c. How does this algorithm compare with the brute-force algorithm for this problem?

a. Call **Algorithm** $MinMax(A[0..n-1], minval, maxval)$ where

Algorithm $MinMax(A[l..r], minval, maxval)$
//Finds the values of the smallest and largest elements in a given subarray
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \le r)$
//Output: The values of the smallest and largest elements in $A[l..r]$
//assigned to $minval$ and $maxval$, respectively
if $r = l$
    $minval \leftarrow A[l]$; $maxval \leftarrow A[l]$
else if $r - l = 1$
    if $A[l] \le A[r]$
        $minval \leftarrow A[l]$; $maxval \leftarrow A[r]$
    else $minval \leftarrow A[r]$; $maxval \leftarrow A[l]$
else //$r - l > 1$
    $MinMax(A[l..\lfloor (l+r)/2 \rfloor], minval, maxval)$
    $MinMax(A[\lfloor (l+r)/2 \rfloor + 1..r], minval2, maxval2)$
    if $minval2 < minval$
        $minval \leftarrow minval2$
    if $maxval2 > maxval$
        $maxval \leftarrow maxval2$

b. Assuming for simplicity that $n = 2^k$, we obtain the following recurrence for the number of element comparisons $C(n)$:

$$C(n) = 2C(n/2) + 2 \quad \text{for } n > 2, \quad C(2) = 1, \quad C(1) = 0.$$

Solving it by backward substitutions for $n = 2^k$, $k \ge 1$, yields the following:

$$\begin{aligned} C(2^k) &= 2C(2^{k-1}) + 2 \\ &= 2[2C(2^{k-2}) + 2] + 2 = 2^2 C(2^{k-2}) + 2^2 + 2 \\ &= 2^2[2C(2^{k-3}) + 2] + 2^2 + 2 = 2^3 C(2^{k-3}) + 2^3 + 2^2 + 2 \\ &= \dots \\ &= 2^i C(2^{k-i}) + 2^i + 2^{i-1} + \dots + 2 \\ &= \dots \\ &= 2^{k-1} C(2) + 2^{k-1} + \dots + 2 = 2^{k-1} + 2^k - 2 = \frac{3}{2}n - 2. \end{aligned}$$

c. This algorithm makes about 25% fewer comparisons,
1.5n compared to 2n, than the brute-force algorithm.
(Note that if we didn't stop recursive calls when n = 2, we would've lost this gain.)
In fact, the algorithm is optimal in terms of the number of comparisons made.
As a practical matter, however, it might not be faster than the brute-force algorithm because of the recursion-related overhead.

General Recurrence Relation

- For Divide-and-Conquer algorithms

General divide-and-conquer recurrence relation

- In the most typical case of divide-and-conquer a problem's instance of size *n* is divided into two instances of size *n*/2.
- More generally, an instance of size *n* can be divided into ***b*** instances of size *n/b*, with ***a*** of them needing to be solved. (Here, *a* and *b* are constants; $a \geq 1$ and $b > 1$.)
- Assuming that size *n* is a power of *b* to simplify our analysis, we get the following recurrence for the running time *T (n)*:

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

where *T(n/b)* is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n/b* and f(n) is the time required in combining their solutions. (For the sum example in previous slides, $a = b = 2$ and $f(n) = 1$)

Master Theorem

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by using Master's theorem.

Master Theorem

If $f(n) \in \ominus(n^d)$ or $f(n)=c*n^d$ where $d \geq 0$ in the recurrence $T(n) = aT(n/b) + f(n)$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Master Theorem

If $f(n) \in \ominus(n^d)$ or $f(n)=c*n^k$ where $d \geq 0$ in recurrence $T(n) = aT(n/b) + f(n)$ then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

For example, the recurrence for the number of additions $A(n)$ made by the divide-and-conquer sum-computation algorithm on inputs of size $n = 2^k$ is

$$A(n) = 2A(n/2) + 1.$$

Thus, for this example, $a = 2$, $b = 2$, and $d = 0$; hence, since $a > b^d$,

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n).$$

Question

Find the order of growth for solutions of the following recurrences using Master Theorem.

a. $T(n)=4T(n/2) + n, T(1) = 1$ a=4 b=2 d=1

b. $T(n)=4T(n/2) + n^2, T(1) = 1$

c. $T(n)=4T(n/2) + n^3, T(1) = 1$

Masters Theorem:
If $f(n) \in \ominus(n^d)$ *or* $f(n)=c*n^k$ where $d \geq 0$ in recurrence $T(n) = aT(n/b) + f(n)$ *then*

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Question

Find the order of growth for solutions of the following recurrences using Master tHeorem.

a. T(n)=4T(n/2) + n, T(1) = 1

b. T(n)=4T(n/2) + n^2, T(1) = 1

c. T(n)=4T(n/2) + n^3, T(1) = 1

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

a. $T(n) = 4T(n/2) + n$. Here, $a = 4$, $b = 2$, and $d = 1$. Since $a > b^d$, $T(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$.

b. $T(n) = 4T(n/2) + n^2$. Here, $a = 4$, $b = 2$, and $d = 2$. Since $a = b^d$, $T(n) \in \Theta(n^2 \log n)$.

c. $T(n) = 4T(n/2) + n^3$. Here, $a = 4$, $b = 2$, and $d = 3$. Since $a < b^d$, $T(n) \in \Theta(n^3)$.

Question

	Recurrence Relation	Solution
1	$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$	
2	$T(n) = 2T\left(\frac{n}{2}\right) + 10n$	
3	$T(n) = 2T\left(\frac{n}{2}\right) + n^2$	

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Question

	Recurrence Relation	Solution
1	$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$	T(n)= $\Theta\left(n^3\right)$
2	$T(n) = 2T\left(\frac{n}{2}\right) + 10n$	T(n)= $\Theta\left(n \log n\right)$
3	$T(n) = 2T\left(\frac{n}{2}\right) + n^2$	T(n)= $\Theta\left(n^2\right)$.

Merge Sort

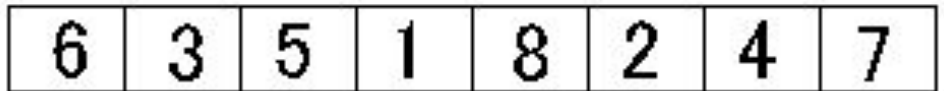
Logic:

Merge sort uses the concept of '**divide and conquer**' to sort elements.

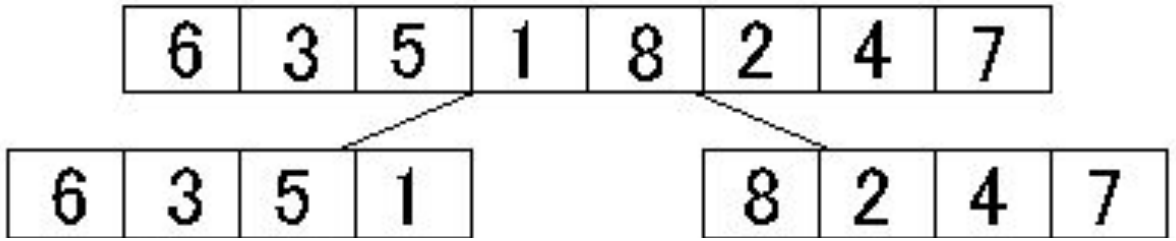
a. **Divide** given elements in half, continue dividing elements unless there is only one element present to sort. As one element is always a sorted element, return and start next step.

b. **Merge** two elements together in a sorted order. Continue merging two sorted sub arrays till all input elements are in sorted order.

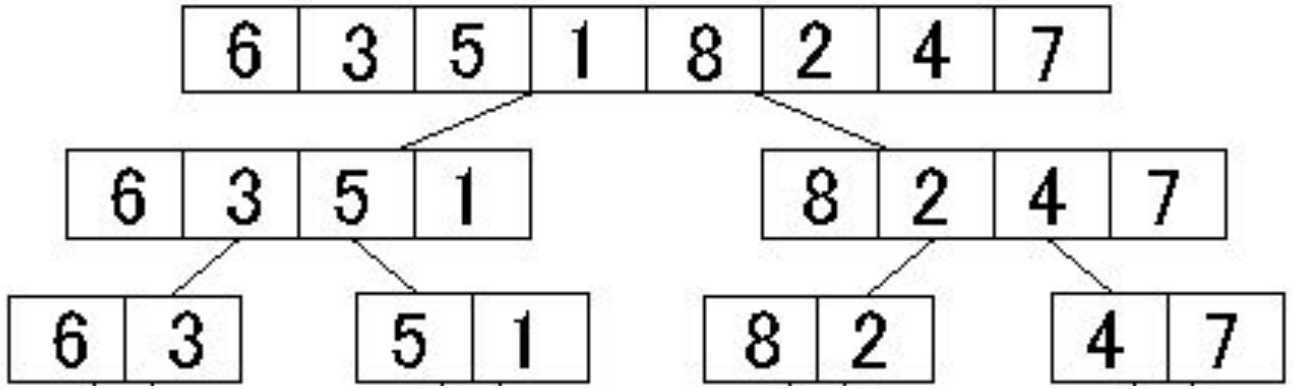
Merge Sort: Example



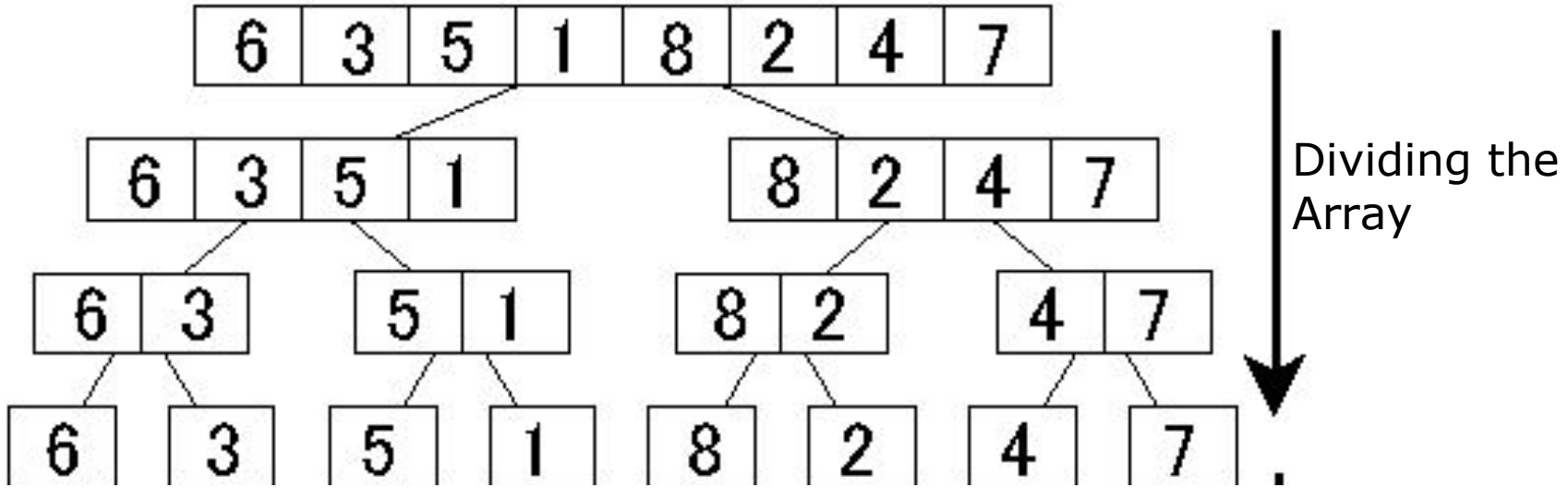
Merge Sort: Example



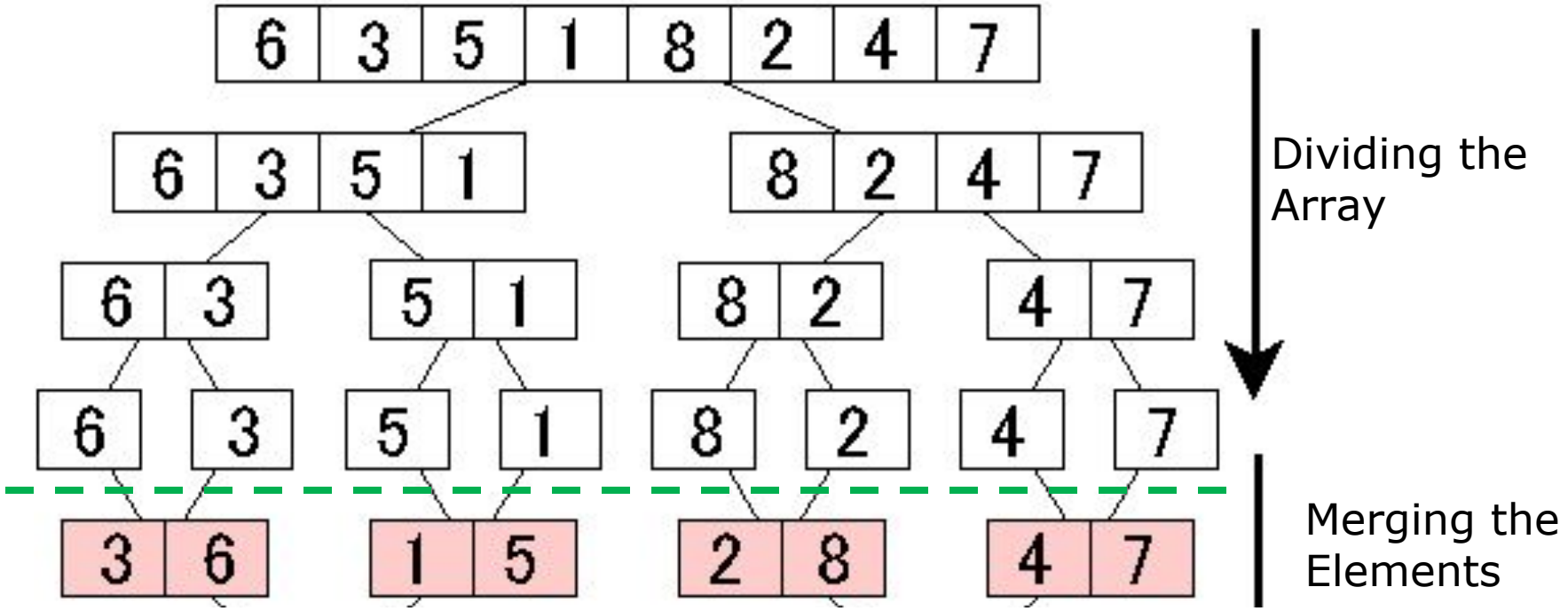
Merge Sort: Example



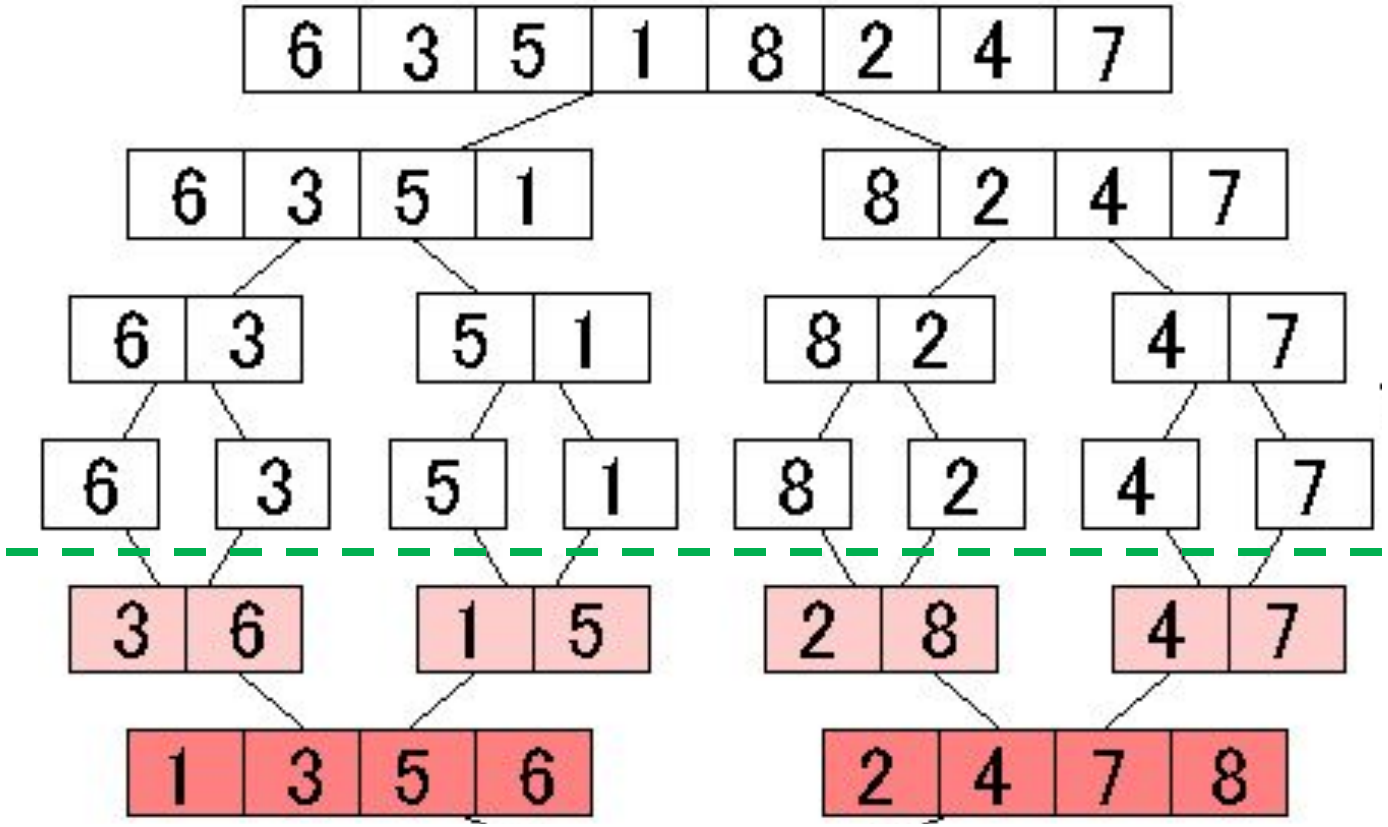
Merge Sort: Example



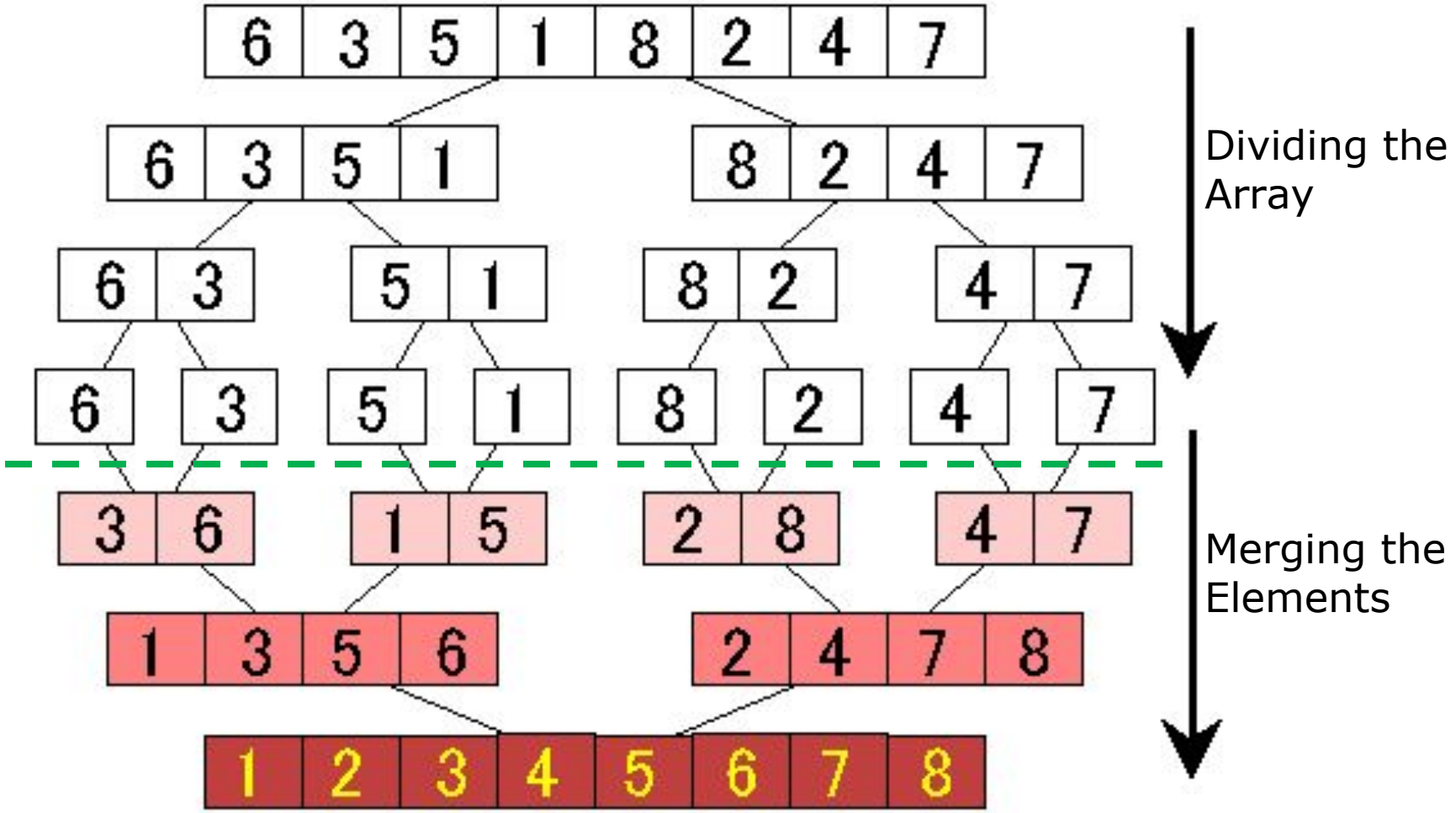
Merge Sort: Example



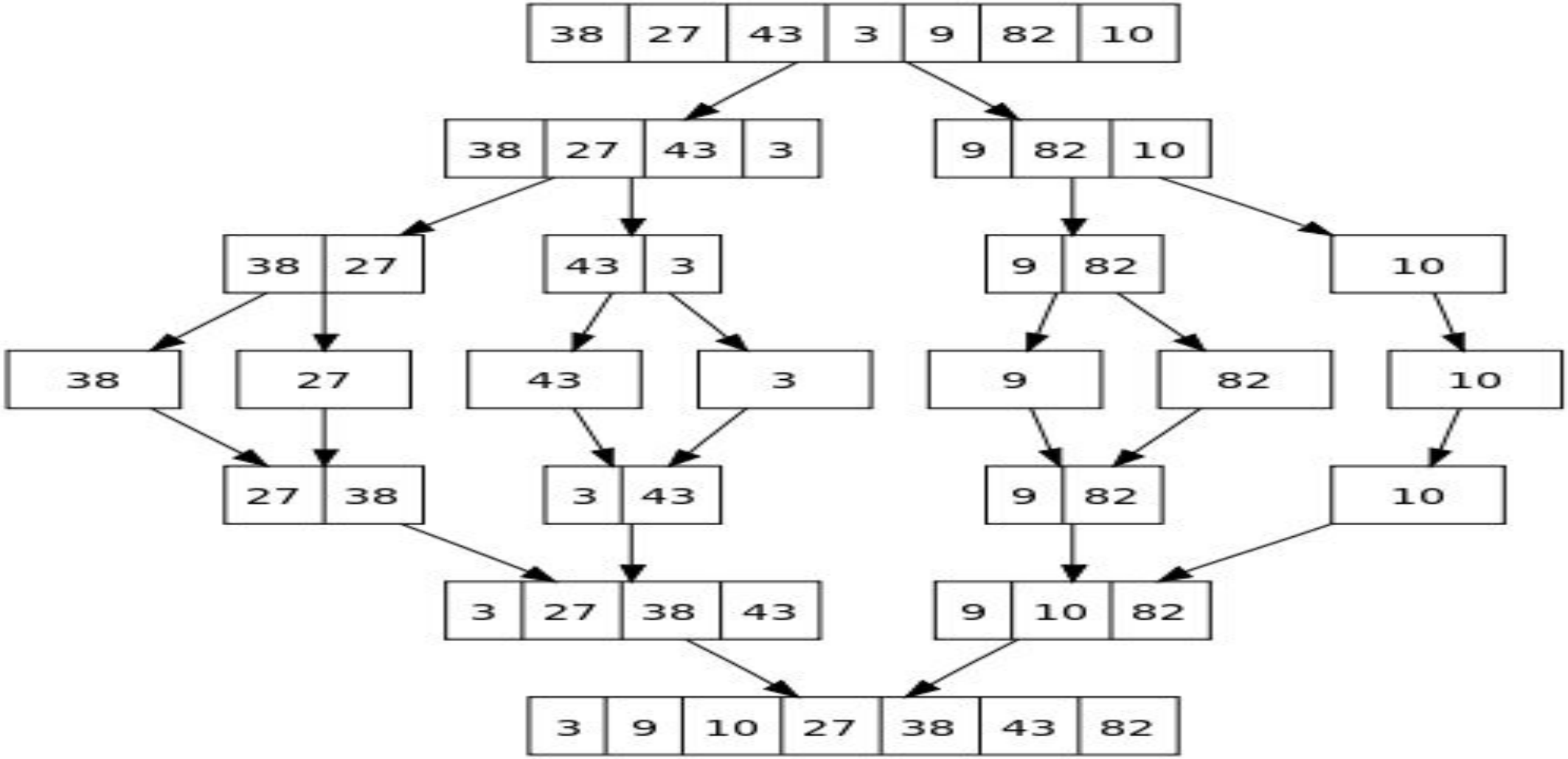
Merge Sort: Example



Merge Sort: Example



Merge Sort: Example



Question

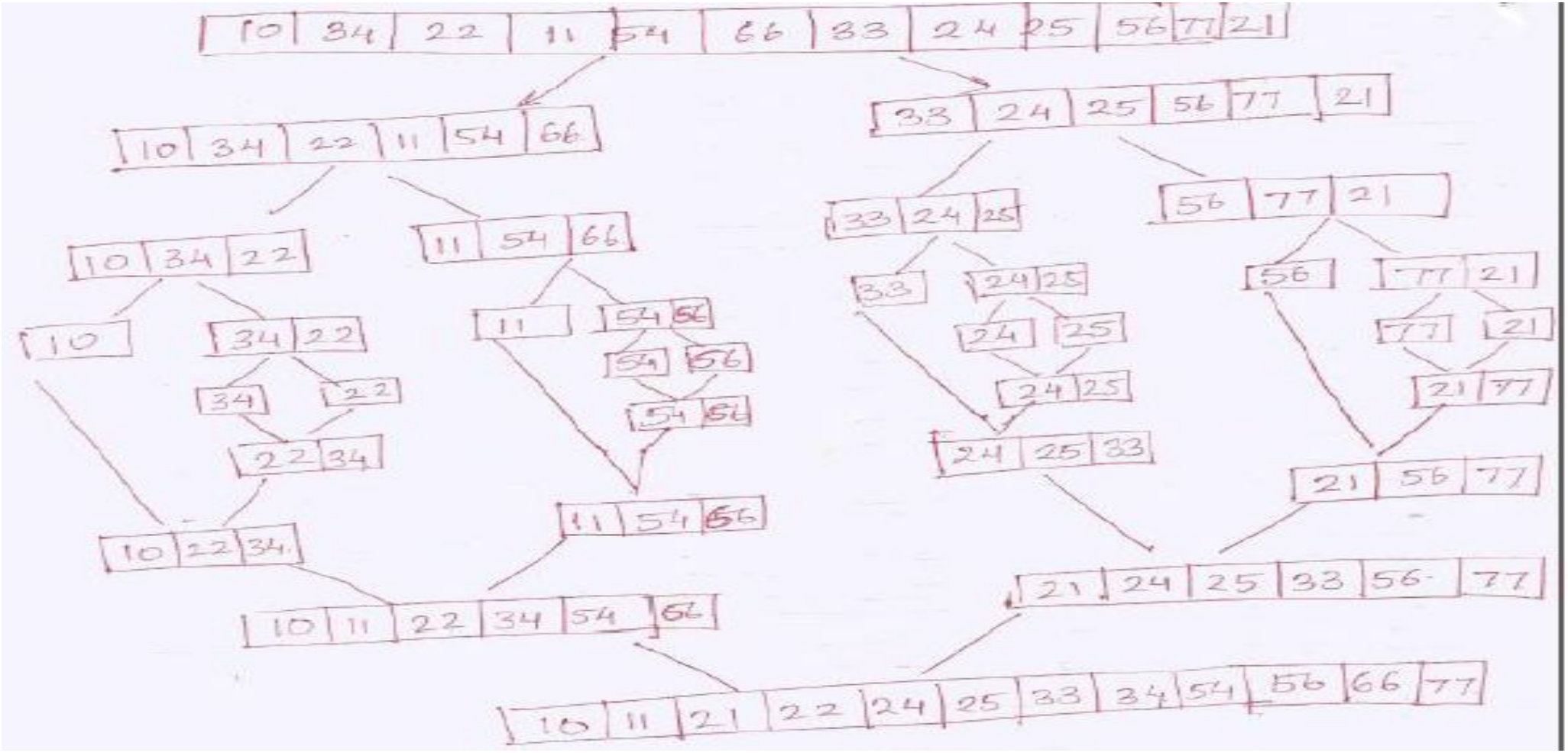
- Given the numbers 5, 4, 1, 3, 7, 2, 9, 8. Write mergesort tree to sort these numbers.

Question

- Given the numbers 10,34,22,11,54,66,33,24,25,56,77,21. Write mergesort tree to sort these numbers.

Question

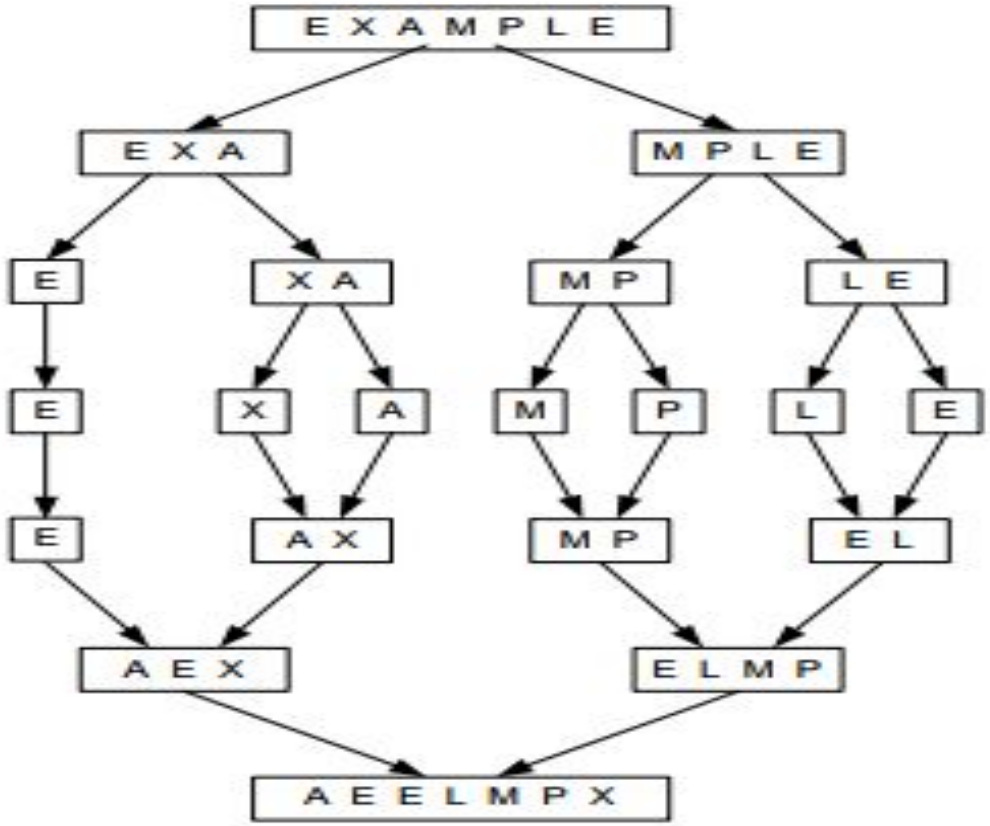
- Given the numbers 10,34,22,11,54,66,33,24,25,56,77,21. Illustrate how mergesort would sort these numbers.



Homework Problem

- Apply mergesort to sort the list E, X, A, M, P, L, E in alphabetical order.

Here is a trace of mergesort applied to the input given:



MergeSort Algorithm

ALGORITHM: split(a[0….n-1],low,high)
//Sorts array a[0….n-1] by recursive mergesort
//Input :An array a[0….n-1] of orderable elements
//Output : Array a[0….n-1] sorted in nondecreasing order
if low<high
    mid□(low+high)/2
    split(a,low,mid)
    split(a,mid+1,high)
    combine(a,low,mid,high)
end if

- 4 7 2 8 1

MergeSort Algorithm (Contd….)

ALGORITHM : combine(a[0….n-1],low,mid,high)

//merge two sorted arrays where first array starts from *low* to *mid* and second starts from *mid*+1 to *high*

//Input : ***a*** is a sorted array from index position low to mid

// ***a*** is a sorted array from index position mid+1 to high

//Output: Array a[0….n-1] sorted in nondecreasing order

i□low ; j□mid+1; k□low;

while i<=mid and j<=high **{**

  if a[i]<a[j] //compares the elements of the two sub-arrays and merges them

    c[k]□a[i]; k□k+1; i□i+1

  else

    c[k]□a[j] ; k□k+1 ; j□j+1

  end if }

if i>mid

  while j<=high { c[k]□a[j]; k□k+1; j□j+1 } //copies the remaining elements if any

end if

if j>high

  while i<=mid { c[k]□a[i]; k□k+1; i□i+1 }

end if

for i□low to high { a[i]□c[i] }

end for

Efficiency of Merge Sort

- Assuming for simplicity that *n* is a power of 2, the recurrence relation for the number of key comparisons *C(n)* is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

- Let us analyze $C_{merge}(n)$ *(or* $C_{combine}(n)$*),* the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. **In the worst case, neither of the two arrays becomes empty before the other one contains just one element** (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

For large values of n,
(n-1)≈ n

Hence, according to the Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Lab Program 8

- Sort a given set of N integer elements using Merge Sort technique and compute its time taken. Run the program for different values of N and record the time taken to sort.

Quick Sort (or Partition Exchange Sort)

QuickSort

- This sorting algorithm uses the idea of divide and conquer
- It finds the element called **pivot** which divides the array into two parts in such a way that elements in the left half are smaller than pivot and elements in the right half are greater than pivot.

We follow three steps recursively

1. Bring the pivot to its appropriate position such that left of the pivot is smaller and right is greater
2. Quick Sort the left half
3. Quick sort the right half

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

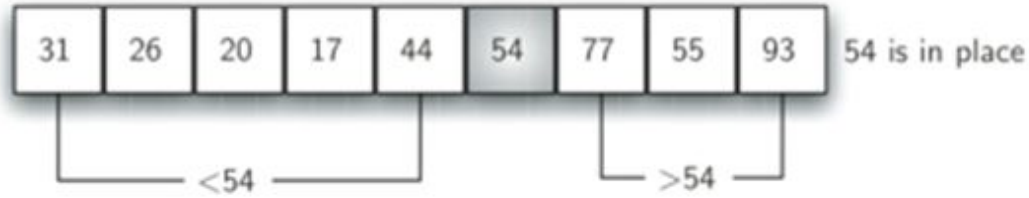
QuickSort: Logic

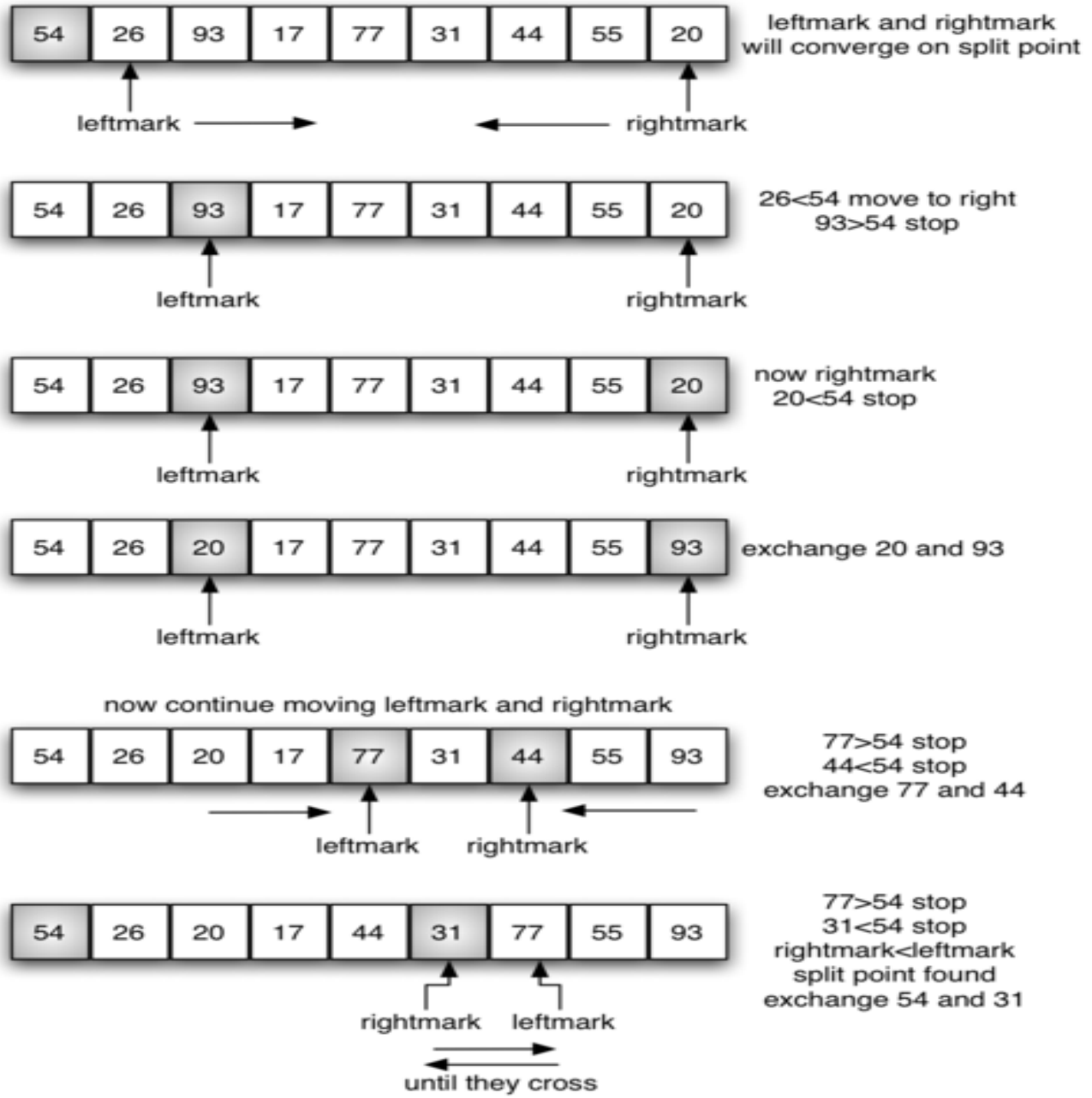
Quicksort uses divide-and-conquer. As with merge sort, think of sorting a subarray array[p..r], where initially the subarray is array[0..n-1].

- **Divide** by choosing any element in the subarray array[p..r]. Call this element the **pivot** (Usually choose first element of the array as pivot element). Rearrange the elements in array[p..r] so that all other elements in array[p..r] that are less than or equal to the pivot are to its left and all elements in array[p..r] are to the pivot's right. We call this procedure **partitioning**. At this point, it doesn't matter what order the elements to the left of the pivot are in relative to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.
- **Conquer** by recursively sorting the subarrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+1..r] (all elements to the right of the pivot, which must be greater than the pivot).
- **Combine** by doing nothing. Once the conquer step recursively sorts, we are done. Why? All elements to the left of the pivot, in array[p..q-1], are less than or equal to the pivot and are sorted, and all elements to the right of the pivot, in array[q+1..r], are greater than the pivot and are sorted. The elements in array[p..r] can't help but be sorted!

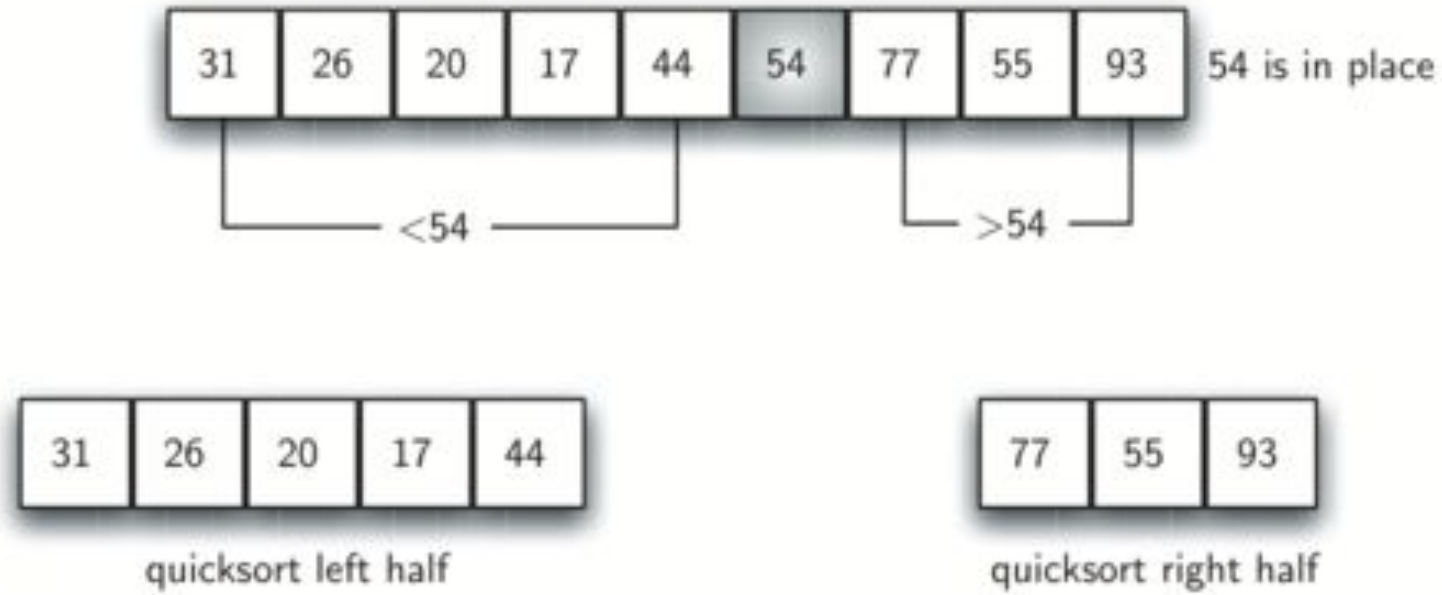
QuickSort: Example

Remember this rule:
- All elements to the **right** of pivot must be greater than **pivot**
- All elements to the **left** of pivot must be smaller than **pivot**





QuickSort: Example



Example of quicksort operation

0	1	2	3	4	5	6	7
5	3	1	9	8	2	4	7
5	3	1	9	8	2	4	7
5	3	1	4	8	2	9	7
5	3	1	4	8	2	9	7
5	3	1	4	2	8	9	7
5	3	1	4	2	8	9	7
2	3	1	4	5	8	9	7
2	3	1	4				
2	3	1	4				
2	1	3	4				
2	1	3	4				
1	2	3	4				
1							
		3	4				
		3	4				
			4				
					8	9	7
					8	7	9
					8	7	9
					7	8	9
					7		
							9

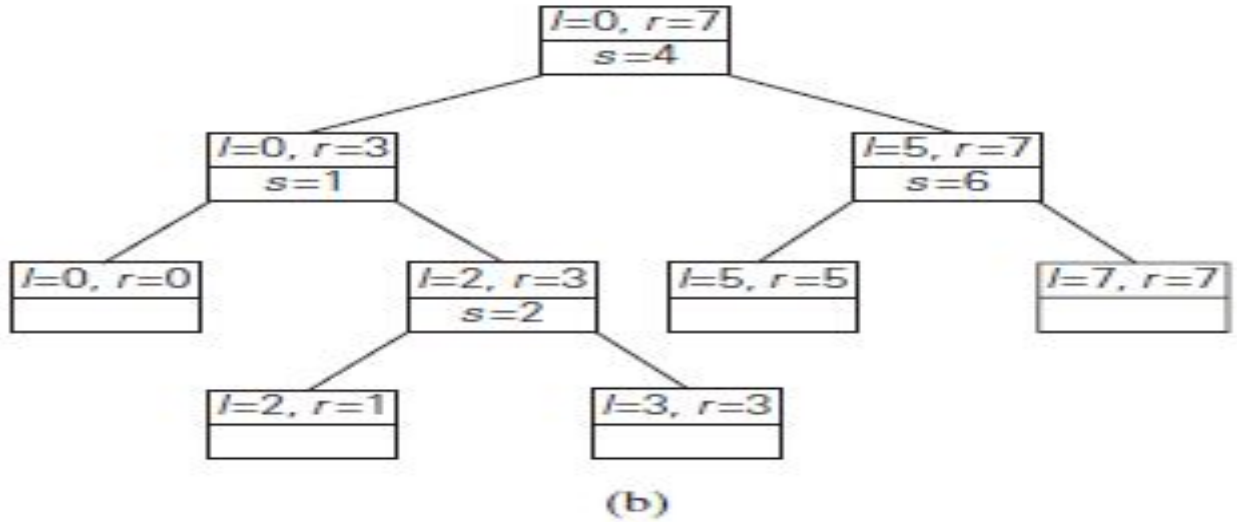
I.

1. Move the pointer,
i, if element <=pivot
j,if element >=pivot

2. swap(a[i],a[j])

II.

swap (a[**j**],pivot) when i and j cross over



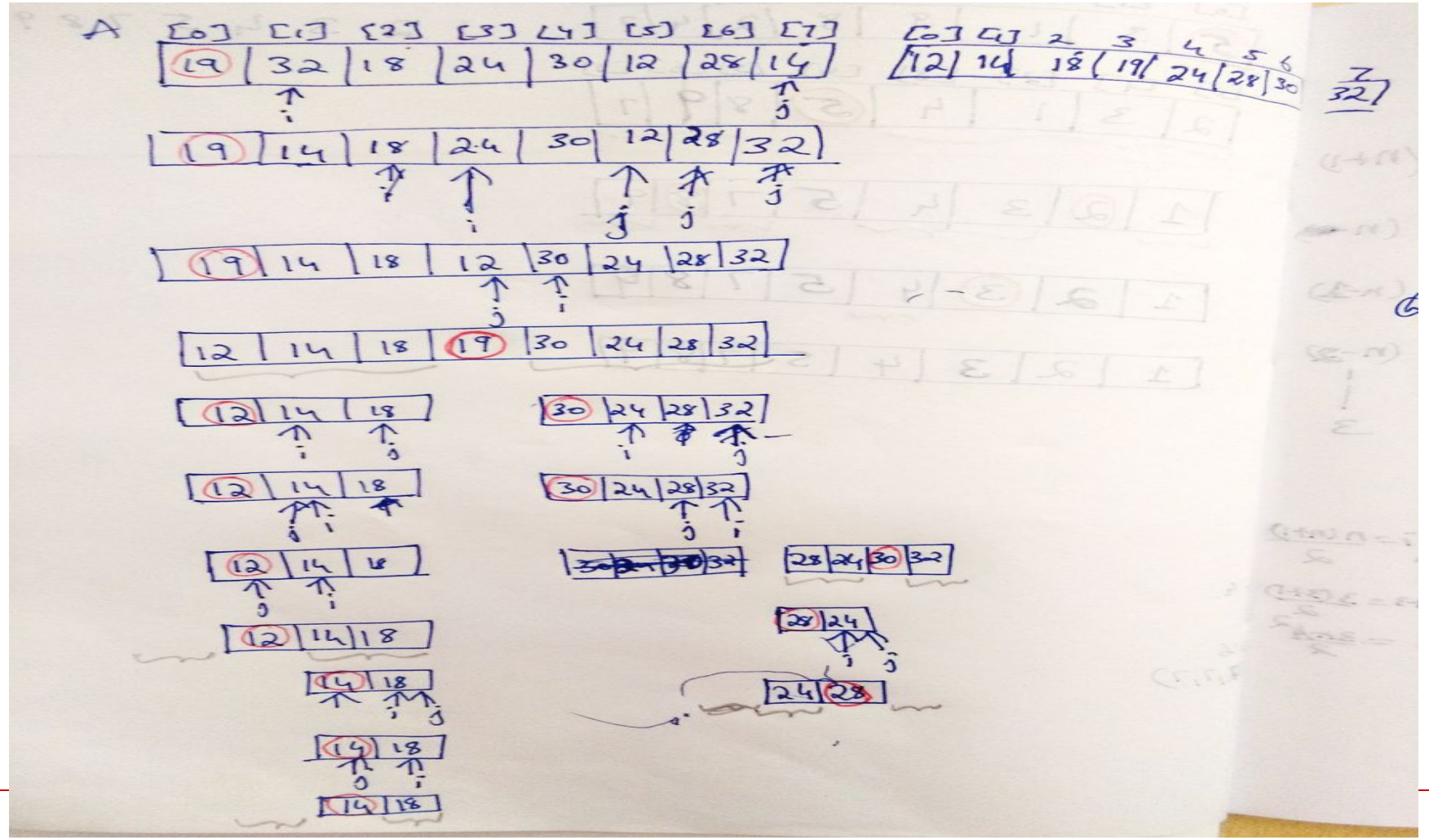
(b)

Question

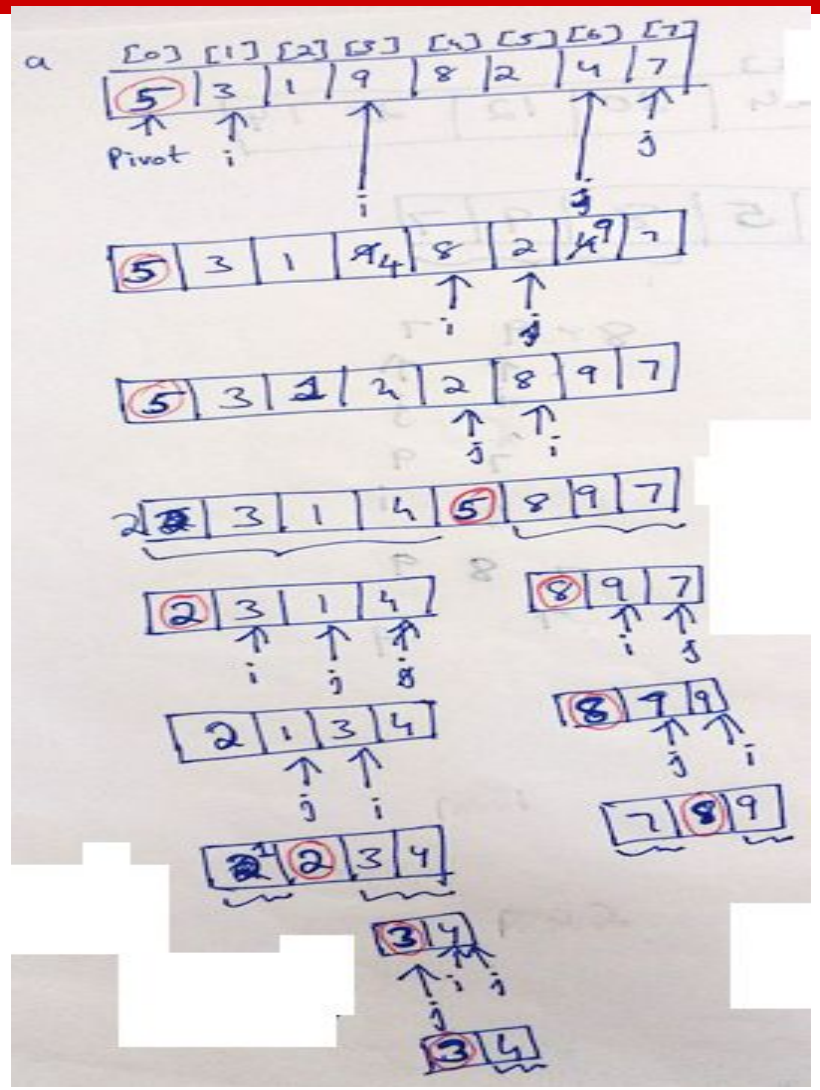
Sort the elements A = (19 32 18 24 30 12 28 14) using Quick Sort

Question

Sort the elements A = (19 32 18 24 30 12 28 14) using Quick Sort



QuickSort: Choosing Pivot as first element of the Array



I.

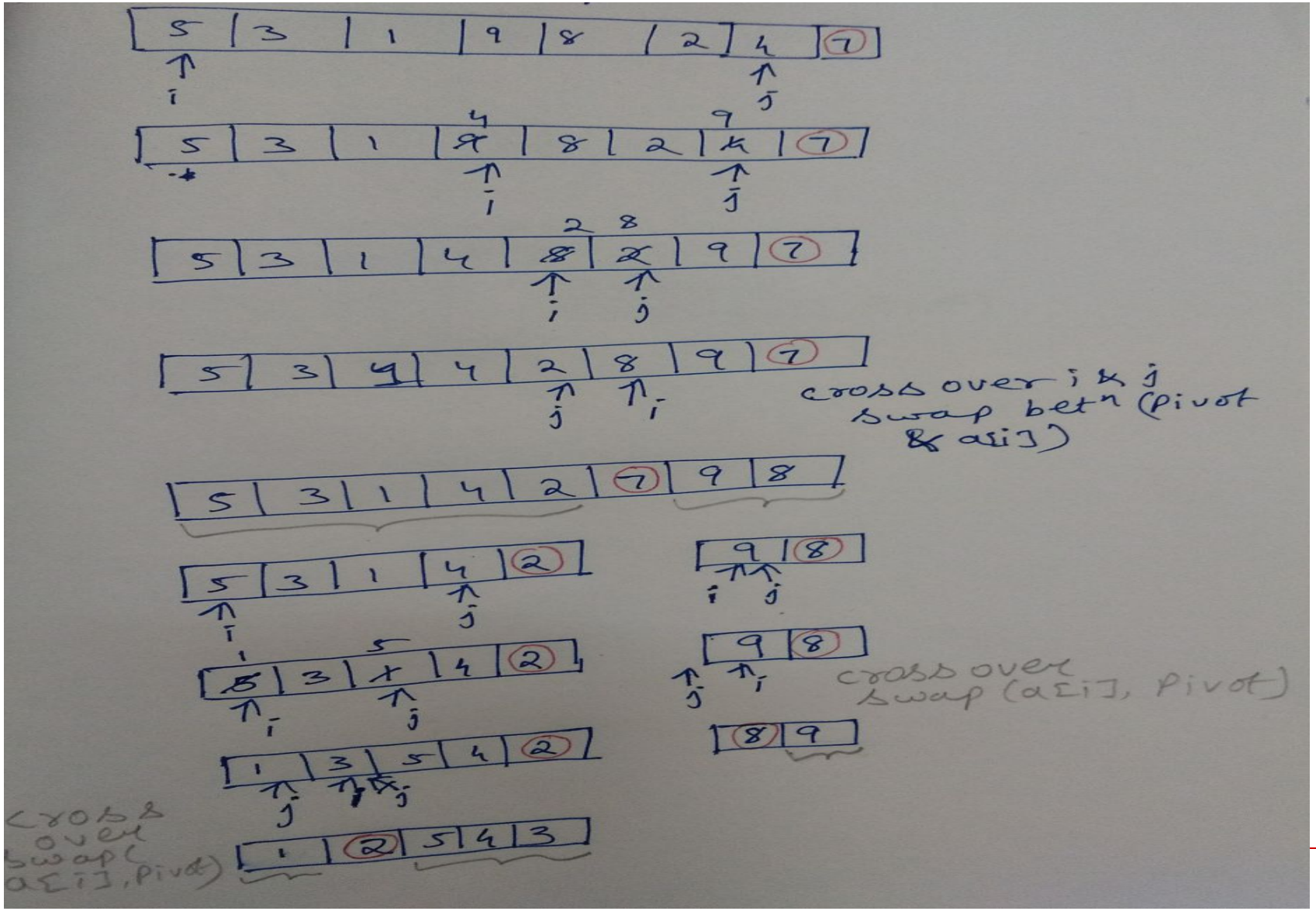
1. Move the pointer,
i, if element <=pivot
j,if element >=pivot

2. swap(a[i],a[j])

II.

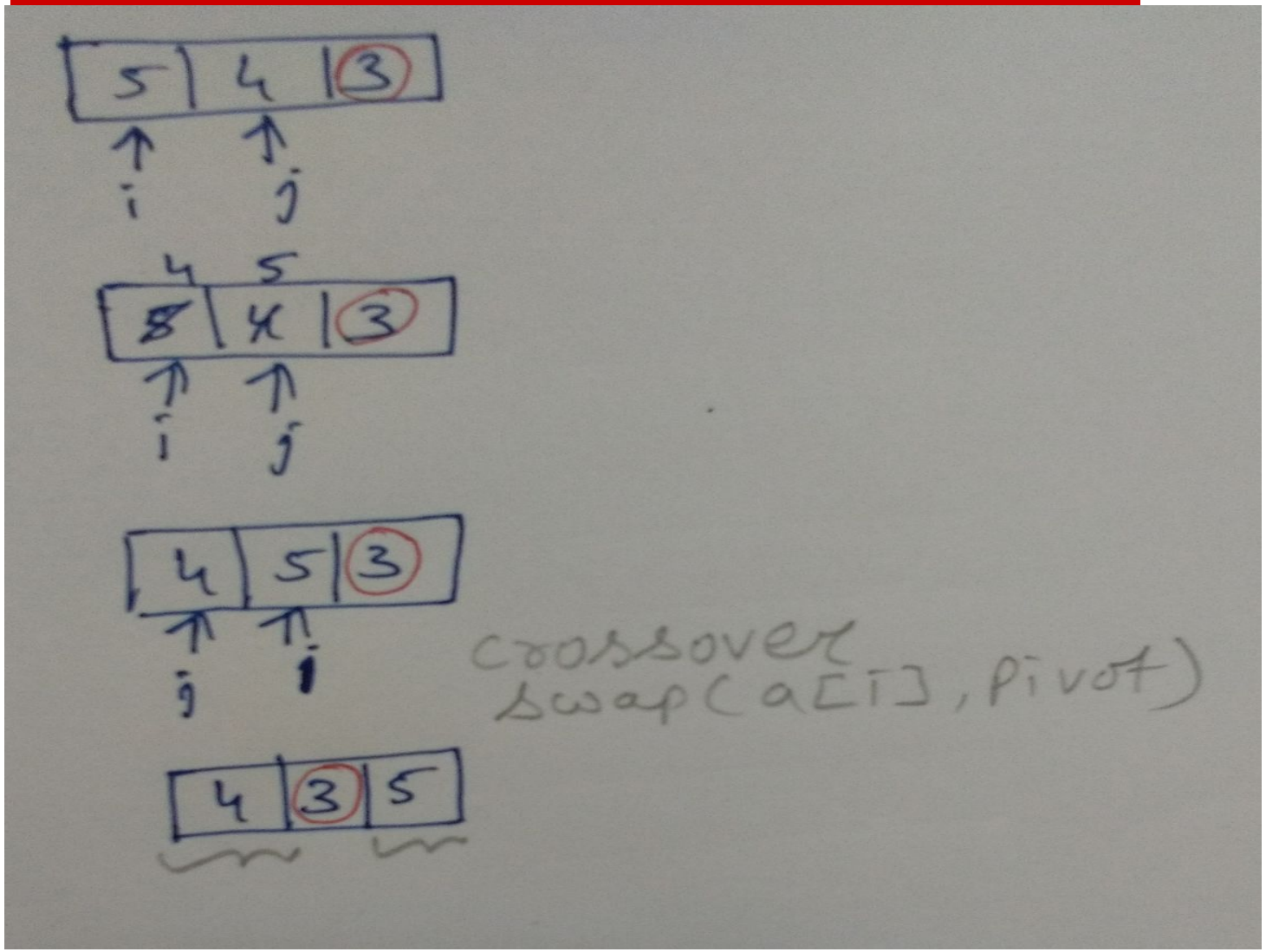
swap (a[**j**],pivot) when i and j cross over

QuickSort: Choosing Pivot as last element of the Array



I.
1. Move the pointer,
i, if element <=pivot
j,if element >=pivot
2. swap(a[i],a[j])
II.
swap (a[**i**],pivot) when
i and j cross over

QuickSort: Choosing Pivot as last element of the Array (Contd…)



Question

Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this:

2 5 1 7 9 12 11 10

Which statement is correct?

A. The pivot could be either the 7 or the 9.
B. The pivot could be the 7, but it is not the 9
C. The pivot is not the 7, but it could be the 9
D. Neither the 7 nor the 9 is the pivot

Question

Suppose we are sorting an array of eight integers using quicksort, and we have just finished the first partitioning with the array looking like this: 2 5 1 7 9 12 11 10 Which statement is correct?

A. The pivot could be either the 7 or the 9.

B. The pivot could be the 7, but it is not the 9

C. The pivot is not the 7, but it could be the 9

D. Neither the 7 nor the 9 is the pivot

Explanation:

- 7 and 9 both are at their correct positions (as in a sorted array). Also, all elements on left of 7 and 9 are smaller than 7 and 9 respectively and on right are greater than 7 and 9 respectively.

Question

Finding K'th Smallest Element in an unsorted Array

Example

Input	Output
Input: arr[] = {7, 10, 4, 3, 20, 15} **k = 3**	**7**
Input: arr[] = {7, 10, 4, 3, 20, 15} **k = 4**	**10**

Question

Finding K'th Smallest Element in an unsorted Array

Example

Input	Output
Input: arr[] = {7, 10, 4, 3, 20, 15} **k = 3**	**7**
Input: arr[] = {7, 10, 4, 3, 20, 15} **k = 4**	**10**

Answer:

We can solve this problem in two simple steps:

1. Sort the input set.
2. Scan the array till Kth element, it gives us the Kth smallest element in array.

Thing to ponder upon here is can we do better? Can we avoid the second step? Can we reduce the size of input to be sorted or not sorting the input at all? Answer to all above questions is Yes. Think Quick sort.

Algorithm to find Kth smallest element

Algorithm

1. Select a pivot and put that in its correct position
2. Now check if the pivot is at K-th position, If yes then return Pivot position
3. If pivot position is less than K, then desired element is right sub-array , repeat step 1 and 2 on right sub-array
4. If pivot position is greater than pivot position, then desired element is in left sub-array; repeat step 1 and 2 on left sub array.

Algorithm to find Kth smallest element

- Quick sort does not completely sorts two sub arrays when it gives us the correct position of pivot. By property of quick sort we know that all elements which are on left side of pivot are less than pivot.
- Let's say we have pivot swapped with jth element of the input set, so there are j elements which are less than pivot. Can we use this information to find Kth smallest element? Absolutely yes.

- If j is less than K, then we have smaller left subset and we need to look into right subset of the input to get the Kth smallest element. So we look (K-j)th smallest element in the right subset.
- Here we have not sorted left or right subset of the input and still we reduced the candidate by almost half.
- If j is more than K, then we need to look in left subset of input. Here we have discarded the right subset of the input.
- As we are doing partitions and calculating the position of pivot, we don't need to scan the array afterwards. As soon as position of pivot is equal to K, we got element.

Sort the elements A = (19 32 18 24 30 12 28 14) using Quick Sort: Recursive calls of QS algorithm

ALGORITHM: QuickSort(a[0….n-1],low,high)

if low<high

// sort only if there are more than two elements in the array

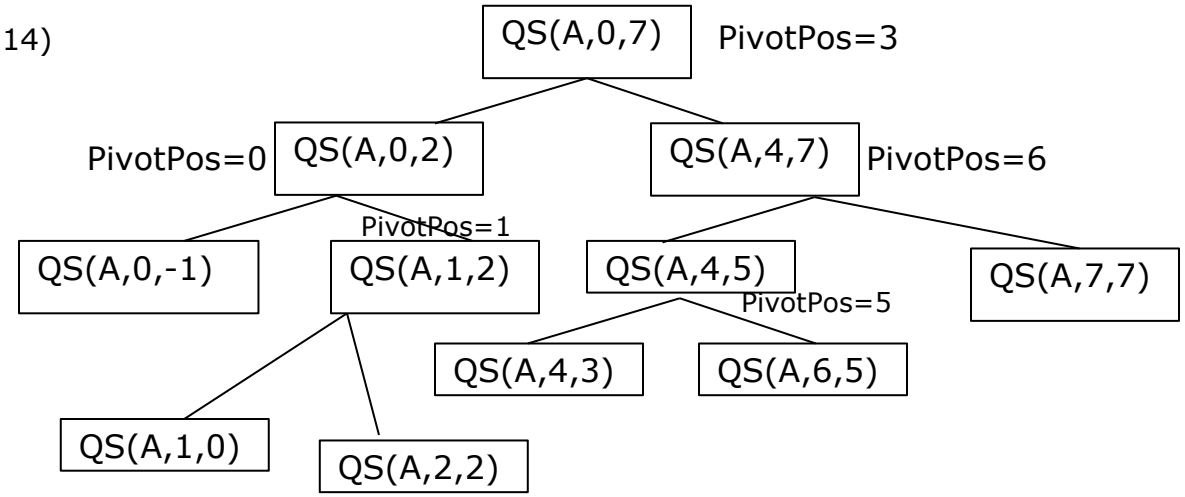
pivot_pos=partition(a,low,high) //*pivot_pos* is a Split position

QuickSort(a,low, *pivot_pos*-1)

QuickSort(a, *pivot_pos*+1,high)

end if

QuickSort Algorithm recursive calls
to sort the numbers (19,32,18,24,30,12,28,14)



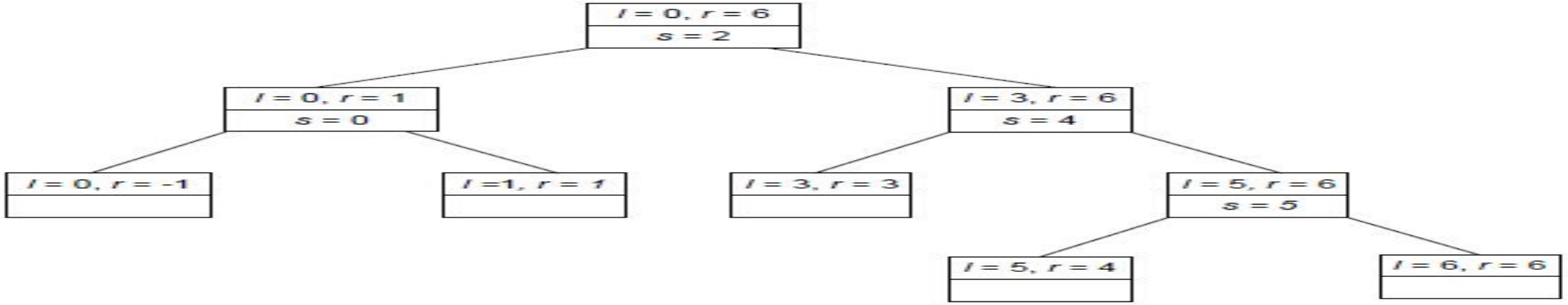
Homework Problem

Apply quicksort to sort the list E, X, A, M, P, L, E in alphabetical order. Draw the tree of the recursive calls made.

0	1	2	3	4	5	6
E	X^{i}	A	M	P	L	E^{j}
E	E	A^{j}	M^{i}	P	L	X
A	E	E	M	P	L	X
A	$E^{i,j}$					
A^{j}	E^{i}					
A	E					
	E					

0	1	2	3	4	5	6
			M	P^{i}	L	X^{j}
			M	P^{i}	L^{j}	X
			M	L^{i}	P^{j}	X
			M	L^{j}	P^{i}	X
			L	M	P	X
			L			

0	1	2	3	4	5	6
					P	$X^{i,j}$
					P^{j}	X^{i}
					P	X
					P	X



QuickSort Algorithm

ALGORITHM: QuickSort(a[0….n-1],low,high)

if low<high

// sort only if there are more than two elements in the array

pivot_pos=partition(a,low,high) //*pivot_pos* is the Split position

QuickSort(a,low, *pivot_pos*-1)

QuickSort(a, *pivot_pos*+1,high)

end if

QuickSort Algorithm (Contd…)

ALGORITHM: partition(a[0….n-1],low,high)

//partition the array into parts such that elements towards the left of the *pivot* element are
//less than *pivot* element and elements towards right of the *pivot* element are greater than *pivot* element
//Input: An array a[0….n-1] is unsorted from index position *low* to *high*
//Output: A partition of a[0…n-1] with split position returned as this function's value

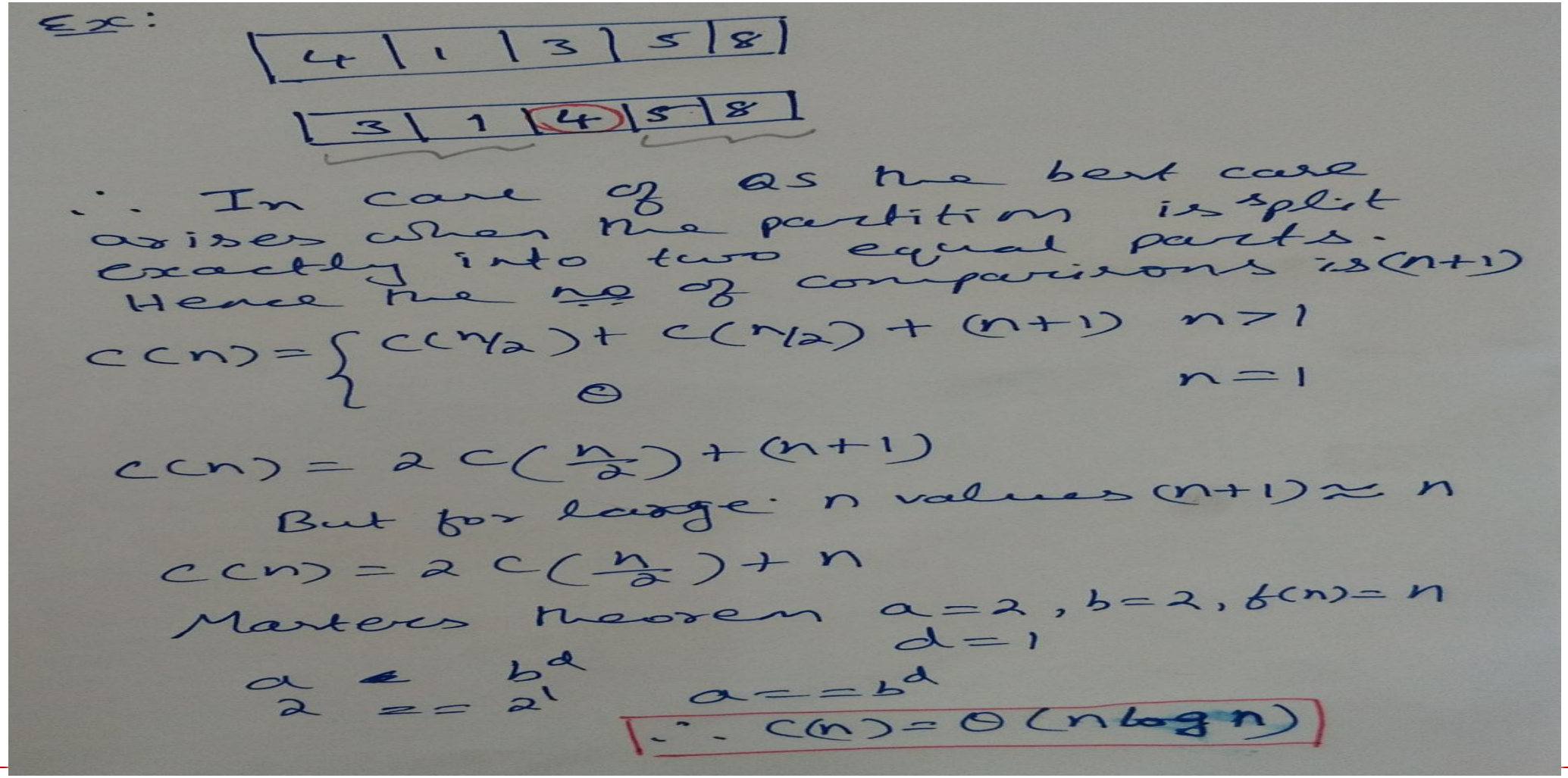
```
pivot=a[low]
i=low+1
j=high
while(1) {
      while (a[i]<=pivot and i<=high) { i=i+1 }
      while (a[j]> pivot and j>=low ) { j=j-1 }
       if (i<j)
          swap a[i] and a[j]
       else {
          a[low]=a[j]; a[j]=pivot
          return j
           }
}
```

Analysis of Quick Sort

- Best Case
- Worst Case
- Average Case

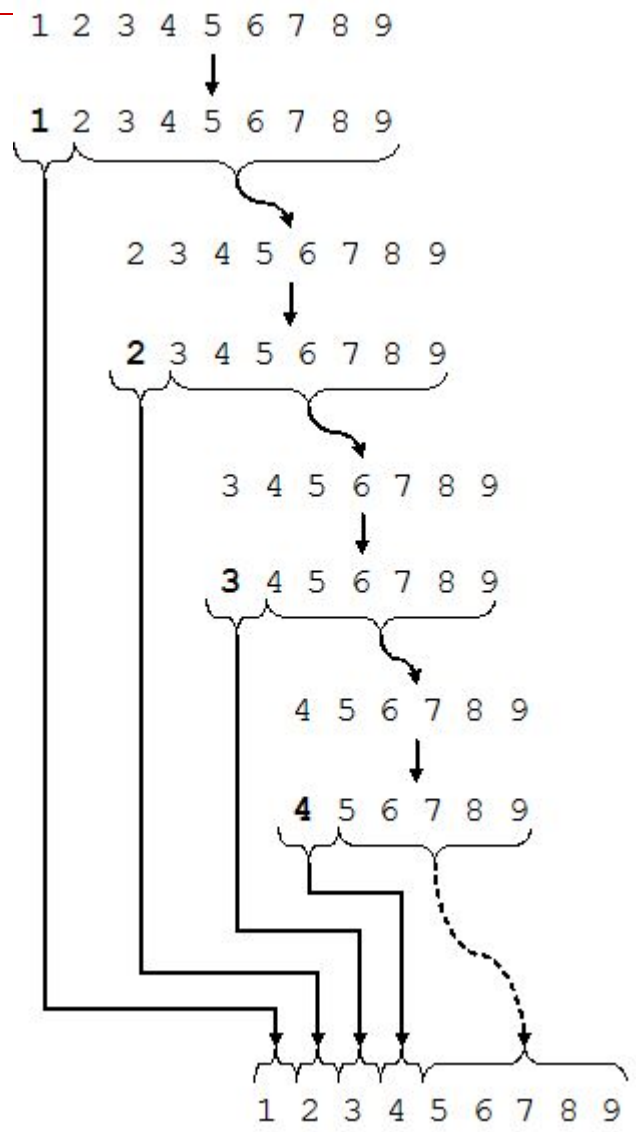
Analysis of Quick Sort: Best Case

- Best case: If the partition splits into two equal parts



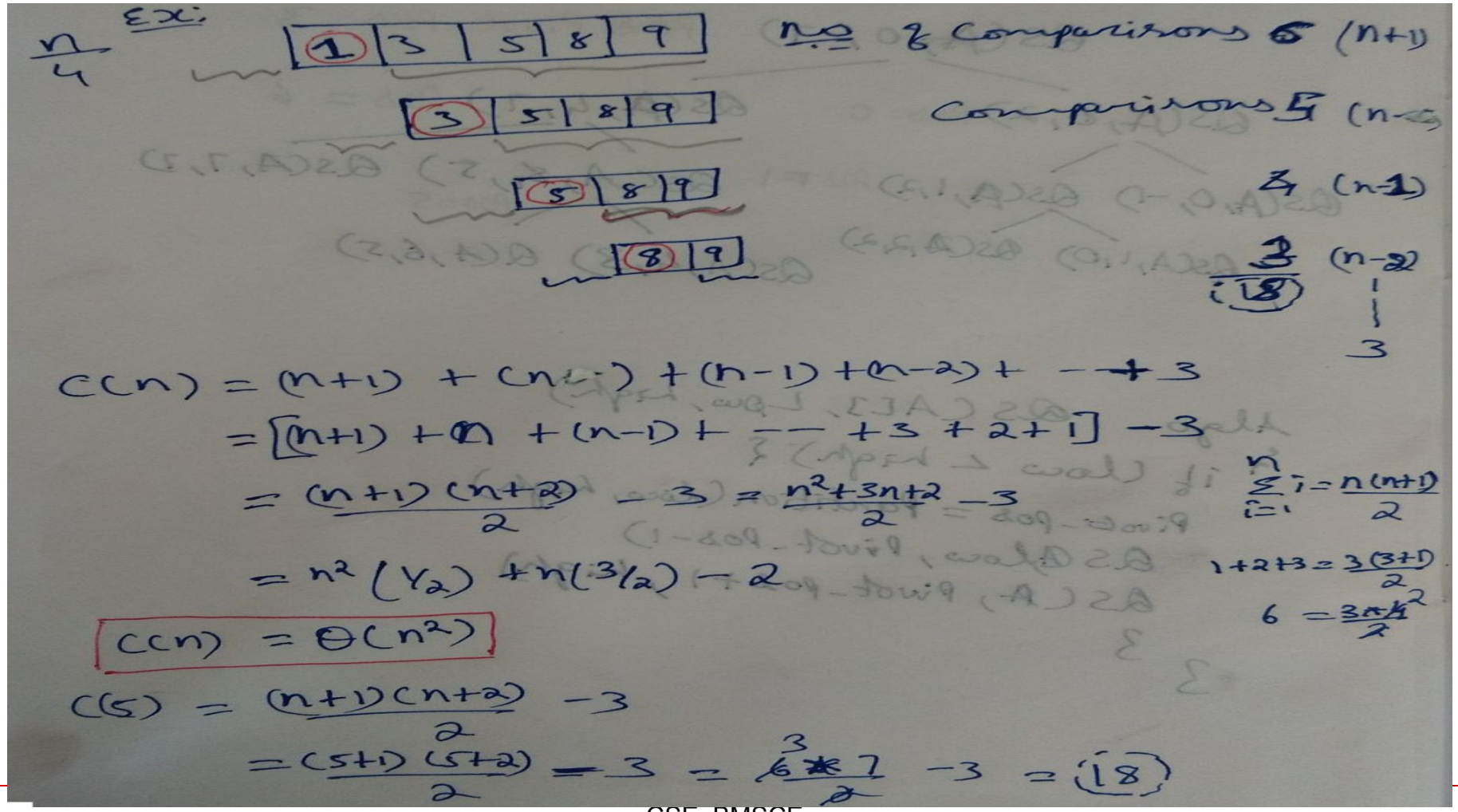
QuickSort

Example of QuickSort for the case when the input is already sorted



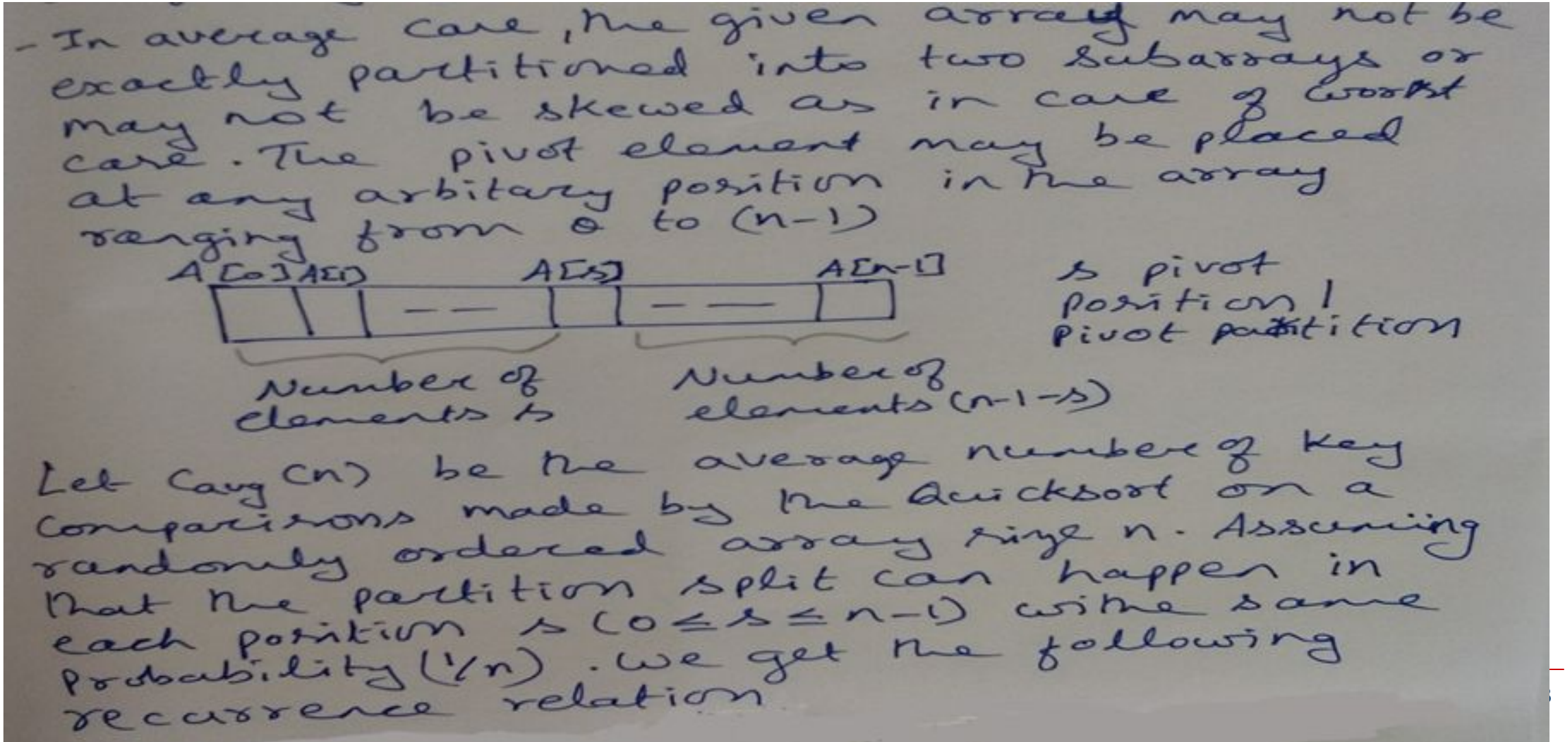
Analysis of Quick Sort: Worst Case

- Worst case: Worst case happens when the input is already sorted



Analysis of Quick Sort: Average Case

- Average case arises when the partition is unequally divided



Analysis of Quick Sort: Average Case (Contd…)

$$C_{avg}(n) = (n+1) + \frac{1}{n}\sum_{s=0}^{(n-1)} [C_{avg}(s) + C_{avg}(n-1-s)]$$

Time for average number of comparisons required for partitioning

Average time required to sort left part of the array

Average time required to sort right part of the array

$$C_1(n) = (n+1) + \frac{1}{n}\sum_{s=0}^{(n-1)} C_1(s) + C_1(n-1-s)$$

$$C(n) = (n+1) + \frac{1}{n}[C(0) + C(1) \text{ -- } \text{ -- } + C(n-1) + C(n-1) + C(n-2) + \text{ -- } + C(0)]$$

$$C(n) = (n+1) + \frac{1}{n}[2(C(0) + C(1) + \text{ -- } + C(n-1))]$$

multiply by n

$$nC(n) = n(n+1) + 2[C(0) + C(1) + \text{ -- } + C(n-1)] \quad (1)$$

Replace n by $(n-1)$ in (1)

$$(n-1)C(n-1) = n(n-1) + 2[C(0) + C(1) + \text{ -- } + C(n-2)] \quad (2)$$

Subtract (1) with (2)

$$nC(n) - (n-1)C(n-1) = 2n + 2C(n-1)$$

Analysis of Quick Sort: Average Case (Contd..)

$$n\,C_1(n) = (n-1)\,C_1(n-1) + 2n + 2\,C_1(n-1) \quad \text{---(2)}$$

$$= n\,C_1(n-1) + C_1(n-1) + 2n$$

$$= (n+1)\,C_1(n-1) + 2n \longrightarrow (3)$$

Divide (3) by $n(n+1)$ both sides

$$\frac{C_1(n)}{(n+1)} = \frac{C_1(n-1)}{n} + \frac{2}{(n+1)}$$

Expand $C_1(n-1)$

$$= \left[\frac{C_1((n-1)-1)}{(n-1)} + \frac{2}{((n-1)+1)}\right] + \frac{2}{(n+1)}$$

$$= \frac{C_1(n-2)}{(n-1)} + \frac{2}{n} + \frac{2}{(n+1)}$$

Expand $C_1(n-2)$

$$\frac{C_1(n)}{(n+1)} = \frac{C(n-3)}{(n-2)} + \frac{2}{(n-1)} + \frac{2}{n} + \frac{2}{(n+1)}$$

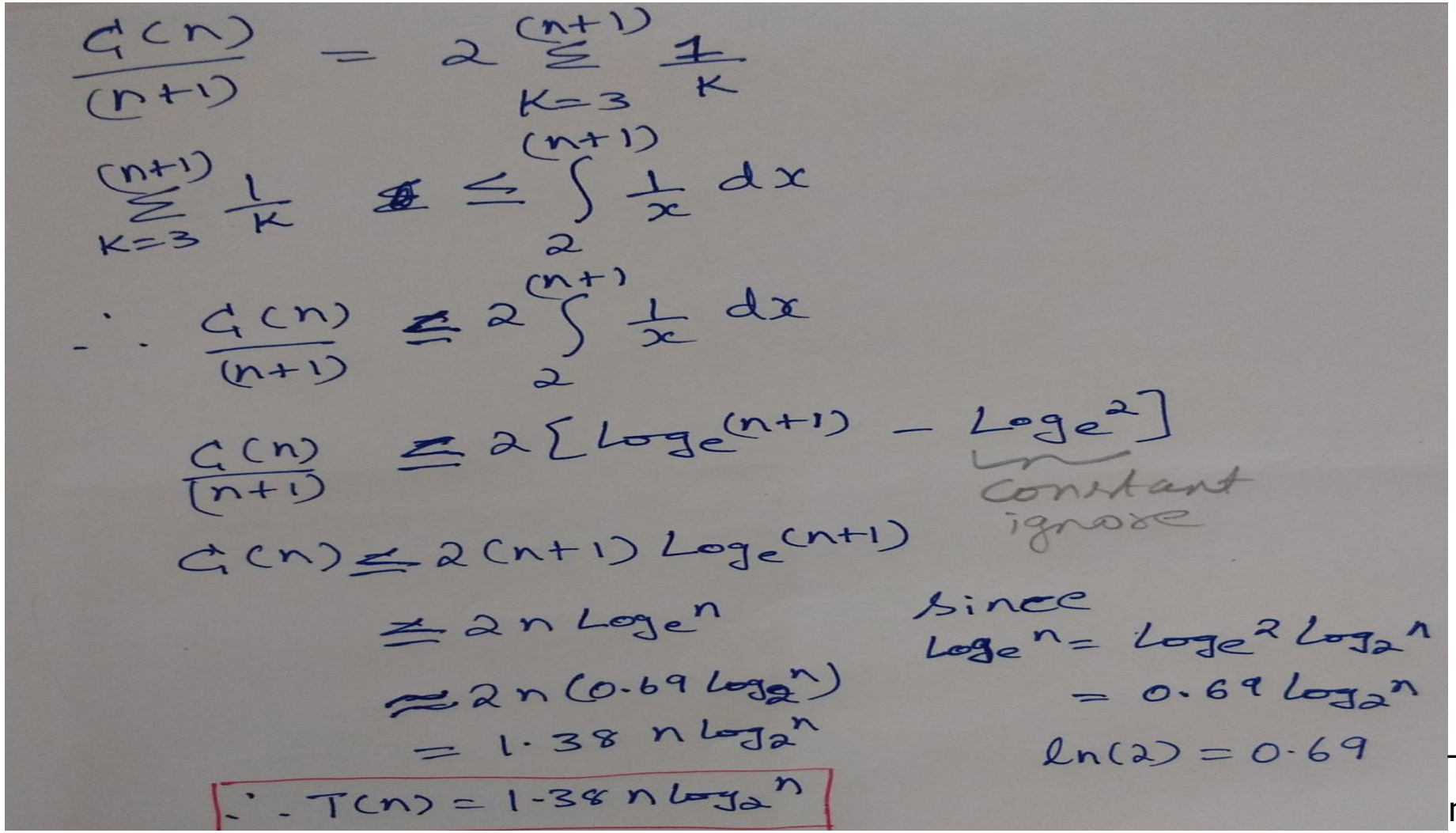
$$\vdots$$

$$= \frac{C(n-(n-1))}{n-(n-2)} + \frac{2}{n-(n-3)} + \cdots + \frac{2}{(n+1)}$$

$$= \frac{C(1)}{2} + 2\sum_{K=3}^{(n+1)} \frac{1}{K}$$

$$= \frac{0}{2} + 2\sum_{K=3}^{(n+1)} \frac{1}{K}$$

Analysis of Quick Sort: Average Case (Contd..)



Thus, on average, quicksort makes only 38% more comparisons than in best case

QuickSort

Quicksort Advantage

- Sorts in place
- Sorts O(nlog n) in average case
- Very efficient in practice, its quick

QuickSort Disadvantage

- Sorts $O(n^2)$ in worst case
- And the worst case does not happen often...

Lab Program 9

- Sort a given set of N integer elements using Quick Sort technique and compute its time taken.

Multiplication of Large Integers and Strassen's Matrix Multiplication

- We examine two algorithms for multiplying two integers and multiplying two square matrices.
- Both achieve a better asymptotic efficiency by ingenious application of the divide-and conquer technique.

Multiplication of Large Integers

Consider the problem of multiplying two (large) *n*-digit integers represented by arrays of their digits such as:

A = 12345678901357986429 B = 87654321284820912836

The grade-school algorithm:

$$\begin{array}{l} a_1\ a_2 \ldots\ a_n \\ \quad b_1\ b_2 \ldots\ b_n \\ \hline \\ (d_{10})\ d_{11} d_{12} \ldots d_{1n} \\ (d_{20})\ d_{21} d_{22} \ldots d_{2n} \\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots\ \ldots \\ (d_{n0})\ d_{n1} d_{n2} \ldots d_{nn} \\ \hline \end{array}$$

1011
x1111
1011
10110
101100
+1011000
10100101

Efficiency: $\Theta(n^2)$ single-digit multiplications

Multiplication of Large Integers

- Integers that are over 100 decimal digits long are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers.
- In this section, we outline an interesting algorithm for multiplying such numbers.
- Obviously, if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n*n digit multiplications.
- The miracle of divide-and-conquer comes to the rescue to accomplish this feat in fewer than n*n digit multiplication.

Multiplication of Large Integers

- Let us start with case of multiplying:
  23 and 14
- The numbers can be represented as follows:

- $23 = 2*10^1 + 3*10^0$ and $14 = 1*10^1 + 4*10^0$

- Now let us multiply
- $23*14 = (2*10^1 + 3*10^0) * (1*10^1 + 4*10^0)$
- $23*14 = (2*1)10^2 + (2*4+3*1)10^1+(3*4)10^0$

- How many multiplications? $4 = n^2$

- The above solution uses the same four digit multiplications as the pen-and-pencil algorithm.

First Divide-and-Conquer Algorithm

Example: A * B where A = 2135 and B = 4014

$A = (21 \cdot 10^2 + 35)$, $B = (40 \cdot 10^2 + 14)$

So, $A * B = (21 \cdot 10^2 + 35) * (40 \cdot 10^2 + 14)$

$= (21 * 40) \cdot 10^4 + (21 * 14 + 35 * 40) \cdot 10^2 + (35 * 14)$

In general, if $A = A_1A_2$ and $B = B_1B_2$ (where A and B are n-digit, A_1, A_2, B_1, B_2 are $n/2$-digit numbers),

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

We've broken down the problem of multiplying 2 n-bit numbers into

4 multiplications of n/2-bit numbers plus 3 additions.

Recurrence for the number of one-digit multiplications $M(n)$:

$M(n) = 4M(n/2), \quad M(1) = 1$

Solution: $M(n) = n^2$

Multiplication of Large Integers

$$23 = 2 \cdot 10^1 + 3 \cdot 10^0 \quad \text{and} \quad 14 = 1 \cdot 10^1 + 4 \cdot 10^0.$$

Now let us multiply them:

$$23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$$
$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0.$$

- Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

  2 * 4 + 3 * 1= (2 + 3)*(1 + 4)-(2*1)-(3*4)

  = (5)*(5) – 2 – 12

  = 25 – 14 = 11

- $$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

Multiplication of Large Integers

- For any pair of two-digit numbers a = a1a0 and b = b1b0, their product c can be computed by the formula:

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Multiplication of Large Integers

- Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number.
- Applying the divide-and-conquer technique, let us divide both numbers in the middle.
- We denote the first half of the a's digits by a_1 and the second half by a_0; for b, the notations are b_1 and b_0, respectively.
- In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_110^{n/2} + b_0$.
- Therefore, taking advantage of the same trick we used for two-digit numbers, we get:

Multiplication of Large Integers

- Therefore, taking advantage of the same trick we used for two-digit numbers, we get:

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$
$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$
$$= c_2 10^n + c_1 10^{n/2} + c_0,$$

$$c = a * b = c_2 10^2 + c_1 10^1 + c_0,$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0.

- Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. The recursion is stopped when n becomes 1.

Divide-and-Conquer Algorithm

$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_2 + A_2 * B_1) \cdot 10^{n/2} + A_2 * B_2$

The idea is to decrease the number of multiplications from 4 to 3:

$$(A_1 + A_2) * (B_1 + B_2) = A_1 * B_1 + (A_1 * B_2 + A_2 * B_1) + A_2 * B_2,$$

i.e., $(A_1 * B_2 + A_2 * B_1) = (A_1 + A_2) * (B_1 + B_2) - A_1 * B_1 - A_2 * B_2,$
which requires only 3 multiplications at the expense of 2 extra addition and 2 subtraction.

Recurrence for the number of multiplications $M(n)$:

$$M(n) = 3M(n/2) + c \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + (35*14)

where c1 = (21+35) *(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1) *(4+0) - 2*4 - 1*0, etc.

This process requires _____one-digit multiplications as opposed to 16.

Example of Large-Integer Multiplication

2135 * 4014

= (21*10^2 + 35) * (40*10^2 + 14)

= (21*40)*10^4 + c1*10^2 + (35*14)

where c1 = (21+35) *(40+14) - 21*40 - 35*14, and

21*40 = (2*10 + 1) * (4*10 + 0)

= (2*4)*10^2 + c2*10 + 1*0

where c2 = (2+1) *(4+0) - 2*4 - 1*0, etc.

This process requires 9 one-digit multiplications as opposed to 16.

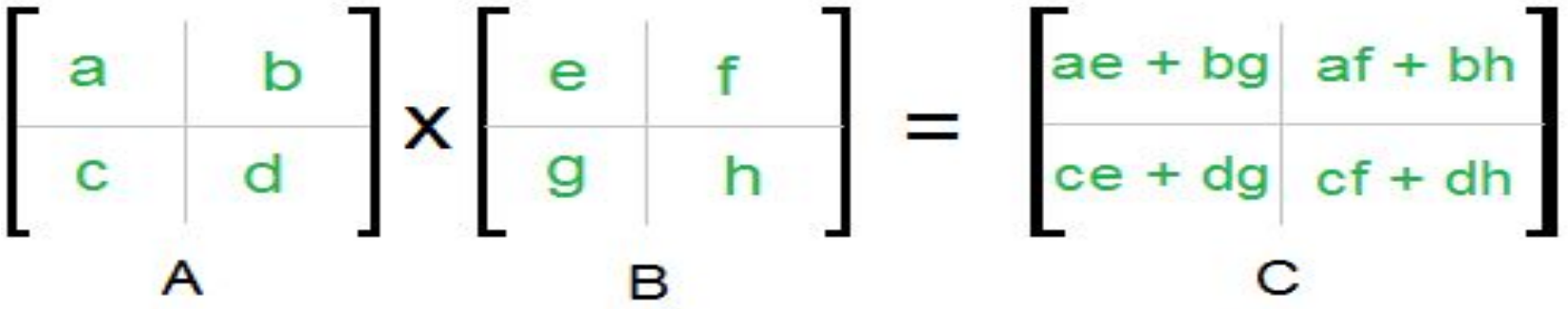
Matrix Multiplication: Naïve Method

```
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            C[i][j] = 0;
            for (int k = 0; k < N; k++)
            {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
}
```

Time Complexity of above method is **$O(N^3)$**

Matrix Multiplication: Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.
1) Divide matrices A and B in 4 sub-matrices of size N/2 x N/2 as shown in the below diagram.
2) Calculate following values recursively. ae + bg, af + bh, ce + dg and cf + dh.



A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2

- In the above method, we do 8 multiplications for matrices of size N/2 x N/2 and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as
- $T(N) = 8T(N/2) + O(N^2)$ From Master's Theorem, time complexity of above method is **$O(N^3)$** which is unfortunately same as the above naive method.

Matrix Multiplication: Divide and Conquer

Divide-and-conquer.

- Divide: partition A and B into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- Conquer: multiply 8 $\frac{1}{2}n$-by-$\frac{1}{2}n$ recursively.
- Combine: add appropriate products using 4 matrix additions.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\ C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\ C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\ C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \end{aligned}$$

$$\mathrm{T}(n) = \underbrace{8T(n/2)}_{\text{recursive calls}} + \underbrace{\Theta(n^2)}_{\text{add, form submatrices}} \Rightarrow \mathrm{T}(n) = \Theta(n^3)$$

Example: $\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\cdot\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$

use Divide-and-Conquer way to solve it as following:

$$\left[\begin{array}{cc|cc}1&1&1&0\\1&1&1&0\\\hline1&1&1&0\\0&0&0&0\end{array}\right]\cdot\left[\begin{array}{cc|cc}1&1&1&0\\1&1&1&0\\\hline1&1&1&0\\0&0&0&0\end{array}\right]=\begin{bmatrix}3&3&3&0\\3&3&3&0\\3&3&3&0\\0&0&0&0\end{bmatrix}$$

$$C_{11}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\3&3\end{bmatrix}$$

$$C_{12}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\3&0\end{bmatrix}$$

$$C_{21}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\0&0\end{bmatrix}$$

$$C_{22}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\0&0\end{bmatrix}$$

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

Strassen’s Matrix Multiplication Method

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls.

The idea of **Strassen’s method** is to reduce the number of recursive calls to 7.

Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size N/2 x N/2 as shown, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.

Strassen's Matrix Multiplication Method

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Thus, to multiply two 2 × 2 matrices, Strassen's algorithm makes **seven multiplications and 18 additions/subtractions**, whereas the normal algorithm requires eight multiplications and four additions.

Strassen's Matrix Multiplication Method

Let ***A*** and ***B*** be two ***n*** × ***n*** matrices where ***n*** is a power of 2. (If ***n*** is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide ***A***, ***B***, and their product ***C*** into four ***n/***2 ×***n/***2 submatrices each as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}.$$

- It is not difficult to verify that one can treat these submatrices as numbers to get the correct product. For example, C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices.
- If the seven products of ***n/***2 × ***n/***2 matrices are computed recursively by the same method, we have Strassen's algorithm for matrix multiplication.

Strassen's Matrix Multiplication Method

- Let us evaluate the asymptotic efficiency of this algorithm. If ***M(n)*** is the number of multiplications made by Strassen's algorithm in multiplying two ***n*** × ***n*** matrices (where ***n*** is a power of 2), we get the following recurrence relation for it:

$$M(n) = 7M(n/2) \quad \text{for } n > 1, \quad M(1) = 1.$$

Since $n = 2^k$,

$$M(2^k) = 7M(2^{k-1}) = 7[7M(2^{k-2})] = 7^2 M(2^{k-2}) = \cdots$$
$$= 7^i M(2^{k-i}) \cdots = 7^k M(2^{k-k}) = 7^k.$$

Since $k = \log_2 n$,

$$M(n) = 7^{\log_2 n} = n^{\log_2 7} \approx n^{2.807},$$

- **$N^{2.807}$ which is smaller than n^3** required by the straight-forward algorithm.

Recurrence relation

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(n/2) & n > 1 \end{cases}$$

$T(n) = 7T(\frac{n}{2})$

$T(n) = 7 * [7T(\frac{n}{4})]$

$= 7^2 T(\frac{n}{2^2})$

$= 7^K T(\frac{n}{2^K})$ $\quad n = 2^K$

$= 7^K T(1)$

$= 7^K$ $\quad K = \log_2 n$

$= 7^{\log_2 n}$

$= n^{\log_2 7}$

$\boxed{T(n) = n^{2.807}}$ By Strassen's Multiplication

$\boxed{T(n) = n^3}$ By Normal Multiplication

Strassen's Matric Multiplication

$$C_{4\times4} = A_{4\times4} * B_{4\times4}$$

$$= \left[\begin{array}{cc|cc} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}\right] * \left[\begin{array}{cc|cc} 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \\ \hline 2 & 2 & 2 & 2 \\ 2 & 2 & 2 & 2 \end{array}\right]$$

(A_{00}, A_{01}, A_{10}, A_{11}; B_{00}, B_{01}, B_{10}, B_{11})

$$C_{4\times4} = \left[\begin{array}{cc|cc} c_{00} & c_{01} & c_{02} & c_{03} \\ c_{10} & c_{11} & c_{12} & c_{13} \\ \hline c_{20} & c_{21} & c_{22} & c_{23} \\ c_{30} & c_{31} & c_{32} & c_{33} \end{array}\right]$$

$$= \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m \end{bmatrix}$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

$M_1 = \left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\right)$

$= \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\right) * \left(\begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}\right)$

$M_1 = \begin{bmatrix} 16 & 16 \\ 16 & 16 \end{bmatrix}$

$M_2 = \left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) * \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$

$= \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\right) * \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$

$M_2 = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}$

$$M_3 = A_{00} * (B_{01} - B_{11})$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} - \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$M_3 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$M_4 = A_{11} * (B_{10} - B_{00})$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} - \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$M_4 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$M_5 = (A_{00} + A_{01}) * B_{11}$$

$$= \left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right) * \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} * \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}$$

$$M_5 = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}$$

$$M_6 = (A_{10} - A_{00}) * (B_{10} + B_{11})$$

$$= \left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

$$M_6 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$M_7 = (A_{01} - A_{11}) * (B_{10} + B_{11})$$

$$= \left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix} + \begin{bmatrix} 2 & 2 \\ 2 & 2 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} * \begin{bmatrix} 4 & 4 \\ 4 & 4 \end{bmatrix}$$

$$M_7 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$C_{11} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_4 - M_2 + M_6 \end{bmatrix}$$

$$= \begin{bmatrix} \begin{bmatrix} 16 & 16 \\ 16 & 16 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix} \\ \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 16 & 16 \\ 16 & 16 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \\ 8 & 8 & 8 & 8 \end{bmatrix}$$

Thanks for Listening