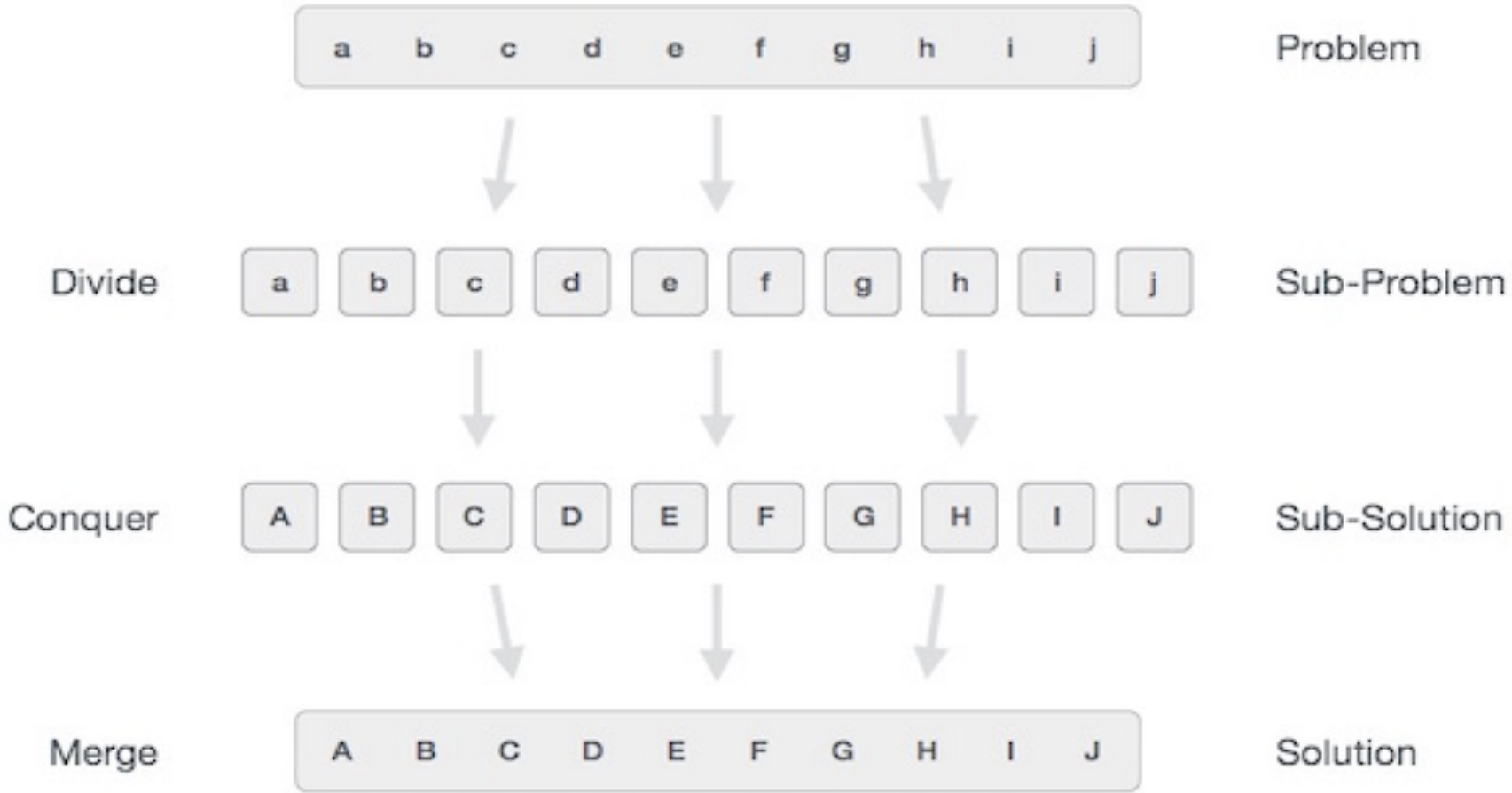
UNIT-2

- Introduction
- Divide and Conquer
- Maximum Subarray Problem

Introduction

- In divide and conquer method, the problem is divided into smaller sub-problems and then each problem is solved independently.
- When the subproblems is divided into even smaller sub-problems, it should reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved.
- The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide-and-conquer method in a three-step process:



Three parts of Divide-and-conquer method:

Divide: This involves dividing the problem into some sub problem.

Conquer: Sub problem by calling recursively until sub problem solved.

Combine: The Sub problem Solved so that we will get find problem solution.

The following are some standard algorithms that follows Divide and Conquer algorithm.

1. Binary Search is a searching algorithm. In each step, the algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of the middle. Otherwise, if x is less than the middle element, then the algorithm recurs for left side of the middle element, else recurs for the right side of the middle element.
2. Quicksort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element.

3.Merge Sort is also a sorting algorithm. The algorithm divides the array in two halves, recursively sorts them and finally merges the two sorted halves.

4.Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane. The problem can be solved in O(n^2) time by calculating distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(nLogn) time.

5.Strassen's Algorithm is an algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3).

Divide And Conquer algorithm :

```
DAC(a, i, j)
{
    if(small(a, i, j))
      return(Solution(a, i, j))
    else
      m = divide(a, i, j)                  // f1(n)
      b = DAC(a, i, mid)                    // T(n/2)
      c = DAC(a, mid+1, j)                // T(n/2)
      d = combine(b, c)                     // f2(n)
   return(d)
}
```

Recurrence Relation for DAC algorithm :

This is recurrence relation for above program.

O(1) if n is small

$T(n) = f1(n) + 2T(n/2) + f2(n)$

Example:

To find the maximum in a given array.

Input: { 100,70, 60, 50, 80, 10, 1}

Output: The maximum number in a given array is : 100

For Maximum:

The recursive approach is used to find maximum where only two elements are left using condition i.e. if(a[index]>a[index+1].)

In a program line a[index] and a[index+1]) condition will ensure only two elements in left.

```
if(index >= l-2)
{
if(a[index]>a[index+1])
{
// (a[index] // the last element will be maximum in a given array.
}
else
{
//(a[index+1] // last element will be maximum in a given array.
} }
```

Recursive function to check the right side at the current index of an array.

```
max = DAC_Max(a, index+1, l); // Recursive call


// Right element will be maximum.
if(a[index]>max)
return a[index];
// max will be maximum element in a given array.
else
return max;
}
```

- Maximum Subarray Problem

Generic problem:

Find a maximum subarray of A[low...high] with initial call: low = 1 and high = n

DC strategy:

1. Divide A[low...high] into two subarrays of as equal size as possible by finding the midpoint mid
2. Conquer:
   (a) finding maximum subarrays of A[low...mid] and A[mid + 1...high]
   (b) finding a max-subarray that crosses the midpoint
3. Combine: returning the max of the three

Correctness:

This strategy works because any subarray must either lie entirely in one side of midpoint or cross the midpoint.

- # Maximum Subarray Problem

Given an array **arr[ ]** of integers, the task is to find the maximum sum sub-array among all the possible sub-arrays.
Examples:

Input: *arr[ ] = {-2, 1, -3, 4, -1, 2, 1, -5, 4}*
Output: *6*
{4, -1, 2, 1} is the required sub-array.
Input: *arr[ ] = {2, 2, -2}*
Output: *4*

- Maximum Subarray Problem

Divide and conquer algorithms generally involves dividing the problem into sub-problems and conquering them separately. For this problem, a structure is used to store the following values:

1. Total sum for a sub-array.
2. Maximum prefix sum for a sub-array.
3. Maximum suffix sum for a sub-array.
4. Overall maximum sum for a sub-array.(This contains the max sum for a sub-array).

- Maximum Subarray Problem

  - During the recursion(Divide part) the array is divided into 2 parts from the middle. The left node structure contains all the above values for the left part of array and the right node structure contains all the above values. Having both the nodes, now we can merge the two nodes by computing all the values for resulting node.
  - The max prefix sum for the resulting node will be maximum value among the maximum prefix sum of left node or left node sum + max prefix sum of right node or total sum of both the nodes (which is possible for an array with all positive values).

- # Maximum Subarray Problem

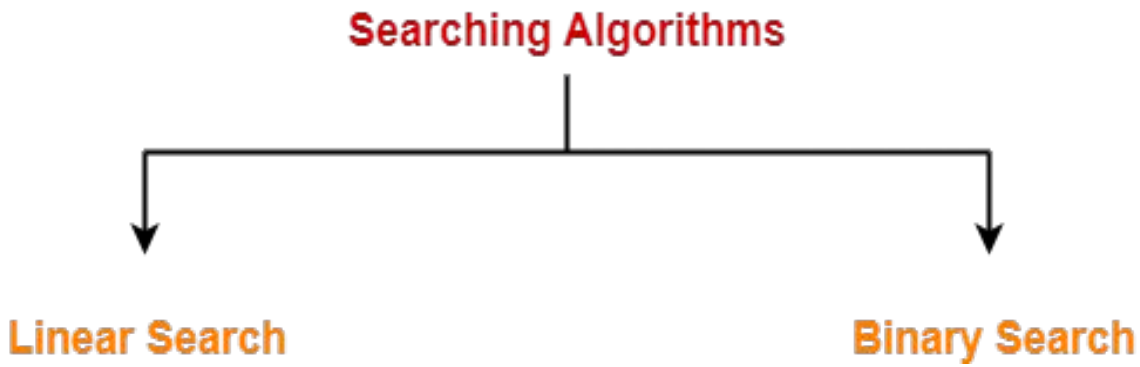
  - Similarly the max suffix sum for the resulting node will be maximum value among the maximum suffix sum of right node or right node sum + max suffix sum of left node or total sum of both the nodes (which is again possible for an array with all positive values).
  - The total sum for the resulting node is the sum of both left node and right node sum. Now, the max subarray sum for the resulting node will be maximum among prefix sum of resulting node, suffix sum of resulting node, total sum of resulting node, maximum sum of left node, maximum sum of right node, sum of maximum suffix sum of left node and maximum prefix sum of right node.

- Maximum Subarray Problem
  - **Time Complexity:** The recursive function generates the following recurrence relation.
    T(n) = 2 * T(n / 2) + O(1)

BINARY SEARCH

- Binary Search
- Complexity of binary search

BINARY SEARCH



Binary Search

Session Learning Outcome-SLO-Solve problems using divide and conquer approaches

- Motivation of the topic

  Binary Search is one of the fastest searching algorithms.
  - It is used for finding the location of an element in a linear array.
  - It works on the principle of divide and conquer technique.

*Binary Search Algorithm can be applied only on **Sorted arrays**.*

- So, the elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
- To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.

- Binary search is an efficient searching method. While searching the elements using this method the most essential thing is that the elements in the array should be a sorted one.
- An element which is to be searched from the list of elements stored in the array A[0---n- 1] is called as Key element.
- Let A[m] be the mid element of array A. Then there are three conditions that need to be tested while searching the array using this method.
- They are given as follows
  - ❖ If key==A[m] then desired element is present in the list.
  - ❖ Otherwise if key <=A[m] then search the left sub list
  - ❖ Otherwise if Key>=A[m] then search the right sub list The following algorithm explains about binary search.

Recursive Binary search algorithm

Algorithm

Input: A array A[0...n-1] sorted in ascending orderand search key K.

Output: An index of array element which is equal to k

Low<- 0;high <-n-1

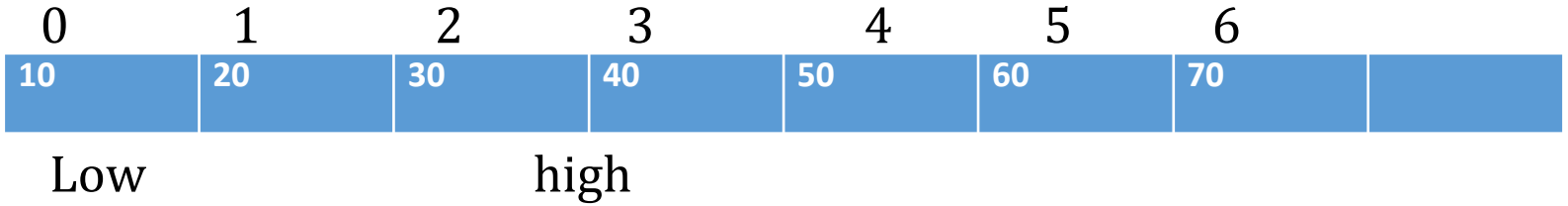
while low ≤ high do

    mid <- (low+high)/2
    if K=A[mid] return mid
    else if K,A[m] high <- mid-1
    else low<-mid+1

return

EXAMPLE FOR BINARY SEARCH

Step:1 Consider a list of elements sorted in array A as



The Search key element is key=60

Now to obtain middle element we will apply formula

mid=(low+high)/2

mid=(0+6)/2

mid=3

Then check A[mid]==key, A[3]=40

A[3]!=60 Hence condition failed

Then check key>A[mid],A[3]=40

60>A[3] Hence condition satisfied so search the Right Sublist

Step 2:

- The Right Sublist is

- Now we will again divide this list to check the mid element

- mid=(low+high)/2
- mid=(4+6)/2
- mid=5
- Check if A[mid]==key
- (i.e)A[5]==60.Hence condition is satisfied. The key element is present in position 5.
- **The number is present in the Array A[ ] at index position 5.**

Thus we can search the desired number from the list of the elements.

ANALYSIS

- The basic operation in binary search is comparison of search key with array elements.
- To analyze efficiency of binary search we must count the number of times the search gets compared with the array elements.
- The comparison is also called three way comparisons because the algorithm makes the comparison to determine whether key is smaller, equal to or greater than A[m].
- In this algorithm after one comparison the list of n elements is divided into n/2 sub list.
- The worst case efficiency is that the algorithm compares all the array elements for searching the desired element.
- In this method one comparison is made and based on the comparison array is divided each time in n/2 sub list.

- Hence worst case time complexity is given by

  Cworst(n) = Cworst (n/2) + 1 for n > 1

  Time required to one comparison made

  Compare left or with middle element

  Right sub list

  Also Cworst(1) =1

- But as we consider the rounded down values when array gets divided the above equation can be written as

  Cworst (n) = Cworst (n / 2) +1 for n>1

  Cworst (1)= 1

- We can analyse the best case , Worst case and Average case . The time complexity of binary search is given as follows

Best case	Average case	Worst Case
$\theta(1)$	$\theta(logn)$	$\theta(logn)$

- In conclusion we are now able to completely describe the computing time of binary search by giving formulas that describe best, average and worst cases
- Successful searches unsuccessful searches

$\theta(1)$ $\theta(logn)$ $\theta(logn)$ $\theta(logn)$

- best average worst best, average, worst

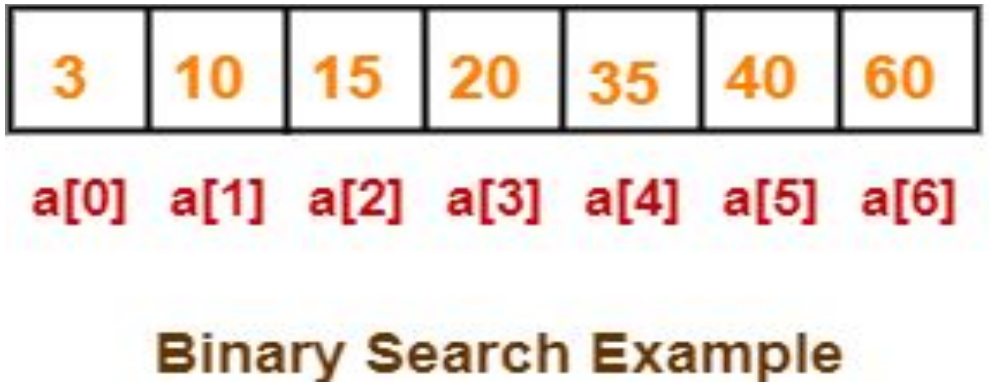
- **Advantages of Binary search:**
  - Binary search is an optimal searching algorithm using which we can search the desired element very efficiently
- **Disadvantages of binary Search :**
  - This Algorithm requires the list to be sorted . Then only this method is applicable
- **Applications of binary search:**
  - The binary search is an efficient searching method and is used to search desired record from database
  - For solving with one un known this method is used

Summary:

- Binary Search time complexity analysis is done below-
  - In each iteration or in each recursive call, the search gets reduced to half of the array.
  - So for n elements in the array, there are $\log_2 n$ iterations or recursive calls.
- **Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.**
  - Here, n is the number of elements in the sorted linear array.

Home assignment:

- Search the Element 15 from the given array using Binary Search Algorithm.



Binary Search Example

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Merge Sort with Time complexity analysis

Design and Analysis of Algorithms

Session Learning Outcome-SLO

- At the end of this session, you should be able to perform merge sort and also evaluate its time complexity

Introduction

- Attributed to a Hungarian mathematician John van Neumann
- Used for sorting unordered arrays
- Uses divide-and-conquer strategy
- **1st phase** is to **divide** the array of numbers into 2 equal parts
  - If necessary, these subarrays are divided further
- **2nd phase** is the **conquer** part
  - Involves sorting of subarrays recursively and combine the sorted arrays to give the final sorted list

Merge sort – informal procedure

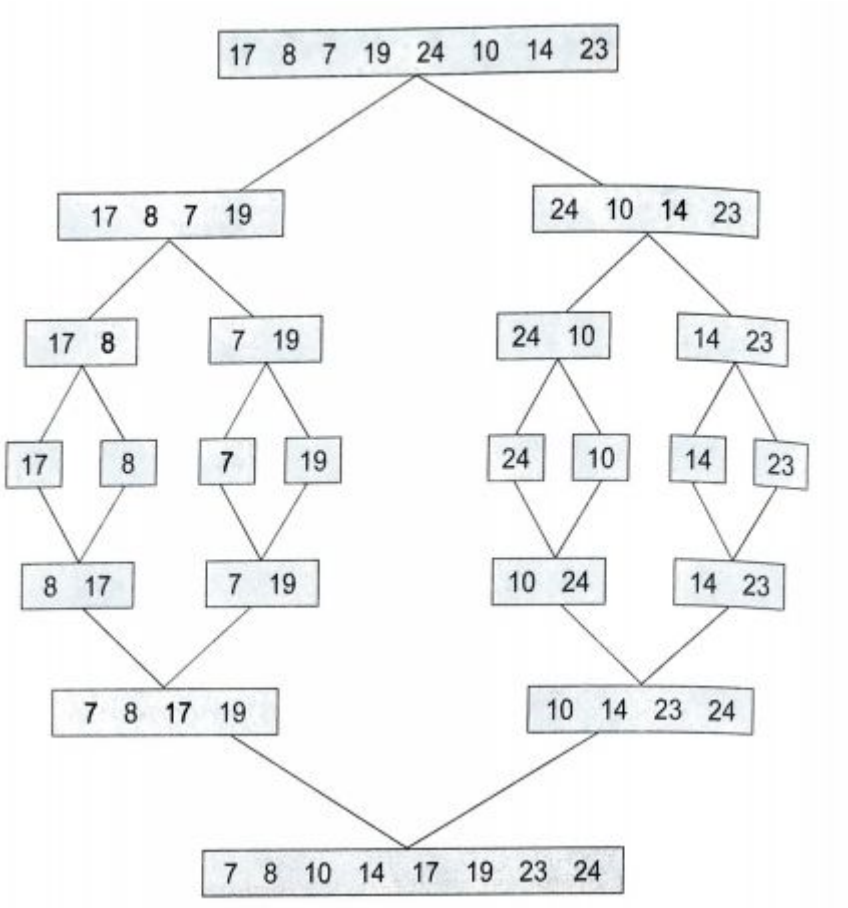
Step1: Divide an array into subarrays B and C

Step 2: Sort subarray D recursively; this yields B sorted subarray

Step 3: Sort subarray C recursively; this yields C sorted subarray

Step 4: Combine B and C sorted subarrays to obtain the final sorted array A

MergeSort- An Example



Algorithm mergesort(A[first .. Last])

Input: Unsorted array A with first =1 and last=n

Output: Sorted array A

```
Begin
    if (first == last) then
        return A[first]
    else
        mid = (first+last)/2
    for i ∈ {1,2, …, mid}
        B[i] = A[i]
    End for
    for i ∈ {mid +1, …, n}
        C[i] = A[i]
    End for
    mergesort(B[1 .. mid])
    mergesort(C[mid + 1 .. n])
    merge(B,C,A)
End
```

Algorithm merge(B,C,A)

Input: Two sorted arrays B and C

Output: Sorted array A

```
Begin
i=1
j=1
k=1
m = length(B)
n = length(C)
while ((i<=m) and (j <= n)) do
if(B[i] < C[j]) then
A[k] = B[i]
i=i+1
else
A[k] = C[i]
j=j+1
End if
k=k+1
end while
if (i > m) then
while k <= m + n do
A[k] = C[j]
j,k = j+1, k+1
end while
else if (j < n) then
while (k <= m+n) do
A[k] = B[j]
i,k = i+1,k+1
end while
end if
return (A)
end
```

Complexity Analysis

- T(n) = $2T\left(\frac{n}{2}\right) + n - 1, for\ n \geq 2$

  = 1, for n=2

  = 0, when n<2

Summary

- Uses divide and conquer strategy
- a comparison based sort
- out of place merge sort
- Stable Algorithm
- Merging method is used
- Variants of merge sort

  in place

  bottom up merge sort

  top down merge sort

Questions

- Why is merge sort an out-of-place sorting technique?
- Does it use recursive procedure?
- What are the best, worst and average case time complexities?

Reference

- S. Sridhar, Design and Analysis of Algorithms, Oxford University Press, 2015

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Thank You

Quick Sort

Prepared By

Dr. V. Elizabeth Jesi

Department of IT

Introduction

- An efficient sorting algorithm
- Based on partitioning of array of **data** into smaller arrays
- Developed by British computer scientist Tony Hoare in 1959 and published in 1961
- Two or three times faster than other sorting algorithms
- Uses divide and conquer strategy
- Quicksort is a comparison sort
- Sometimes called **partition-exchange sort**

Three operations performed in quick sort

1. **Divide:** Select a 'pivot' element from the array and partition the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.
2. **Conquer :** Recursively sort the two sub arrays
3. **Combine** : Combine all the sorted elements in a group to form a list of sorted elements.

How Quick sort works

- Select a PIVOT element from the array
- You can choose any element from the array as the pivot element.
- Here, we have taken the first element of the array as the pivot element.

- The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.

- Now sort the left and right arrays recursively.

Algorithm Quicksort

```
Quicksort(A, L,H)
{
// Array to be soerrted
// L – lower bound of array
// H- upper bound of array

If (L<H)
{
        P=Partition(A,L,H)
        Quicksort(A,L,P)
        Quicksort(A,P+1,H)
}
}
```

Algorithm Partition

```
Partition(A,L,H)
{
        pivot = A[L]
        i=L, j=H+1
    repeat
        repeat   i=i+1  until A[i]>=pivot
        repeat   j=j-1    until A[j]<=pivot
        swap(a[i],a[j])
    until (i>j)
    swap(A[L],A[j])
}
```

Method with Example (Partition)

- Numbers to be sorted

34	53	16	71	9	22	42	18

- Select 34(first element) as the pivot element

34	53	16	71	9	22	42	18

Find the element > 34 from the left, let its index be i

Find the element<34 from the right, let its index be j

34	**53**	16	71	9	22	42	**18**

34	**18**	16	71	9	22	42	**53**

- Do this process until the index I is greater than index j.

34	18	16	**71 i=4**	9	**22 j=6**	42	53

34	18	16	**22**	9	**71**	42	52

34	18	16	22	**9 j=5**	**71 i=6**	42	53

- I becomes > j , so exchange the pivot element and A(j)

9	18	16	22	**34**	71	42	53

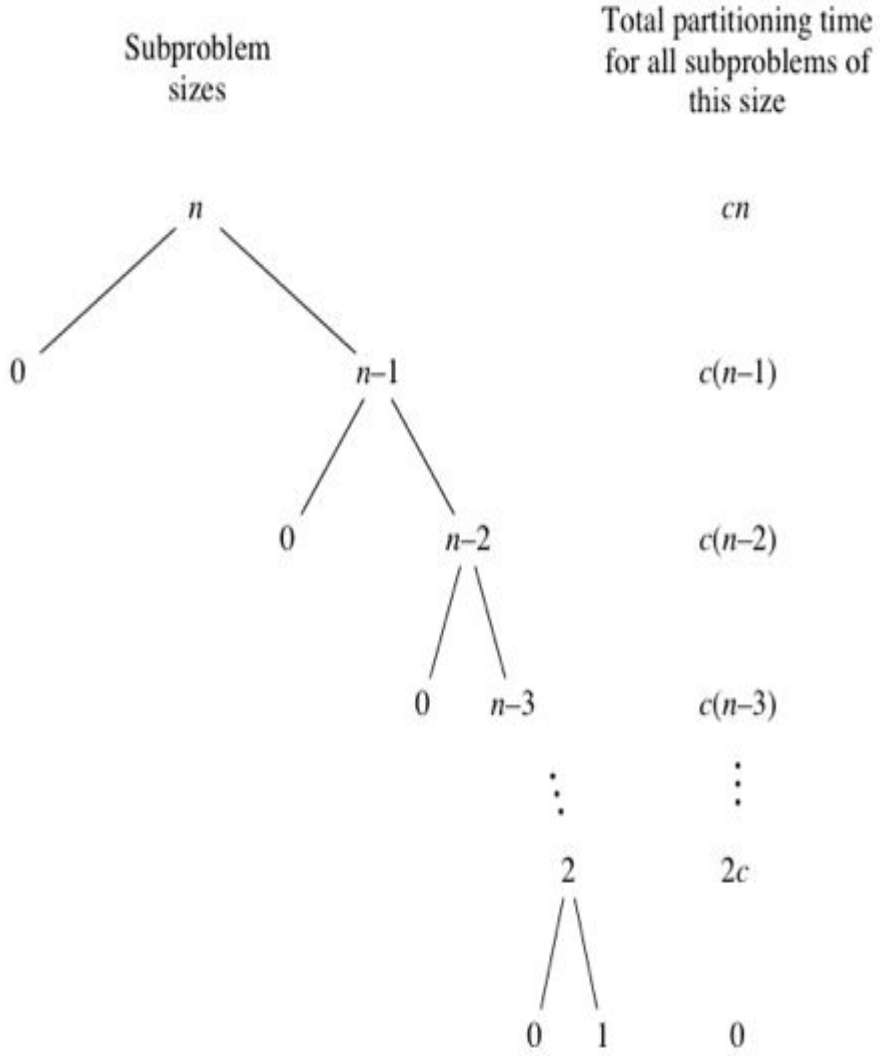
- Note all the elements to the left of pivot is lesser and the elements to the right of pivot is larger than the pivot

Analysis of Quick Sort Algorithm

- If pivot element is either the largest or the smallest element
- Then one partition will have 0 elements and the other partition will have (n-1) elements.
- Eg : if pivot element is the largest element in the array. We will have all the elements except the pivot element in the left partition and no elements in the right partition
- The recursive calls will be on arrays of size 0 and (n-1)

Quick Sort - Worst-case running time

- Let the original call takes cn time for some constant c.
- Recursive call on n-1 elements takes c(n-1)times
- Recursive call on n-2 elements takes time c(n-2) times, and so on.
- Recursive call on 2 elements will take 2c times
- Finally recursive call on 1 element will take no time.



- Sum up the partitioning times, we get

  cn + c(n-1) + c(n-2) + ….+ 2c

  = c(n + (n-1) + (n-2) + …. + 2)

  = c((n+1)(n/2) - 1)

[Note : 1+2+…+(n-1)+n=(n+1)n/2

So 2+…+(n-1)+n=((n+1)n/2)-1]

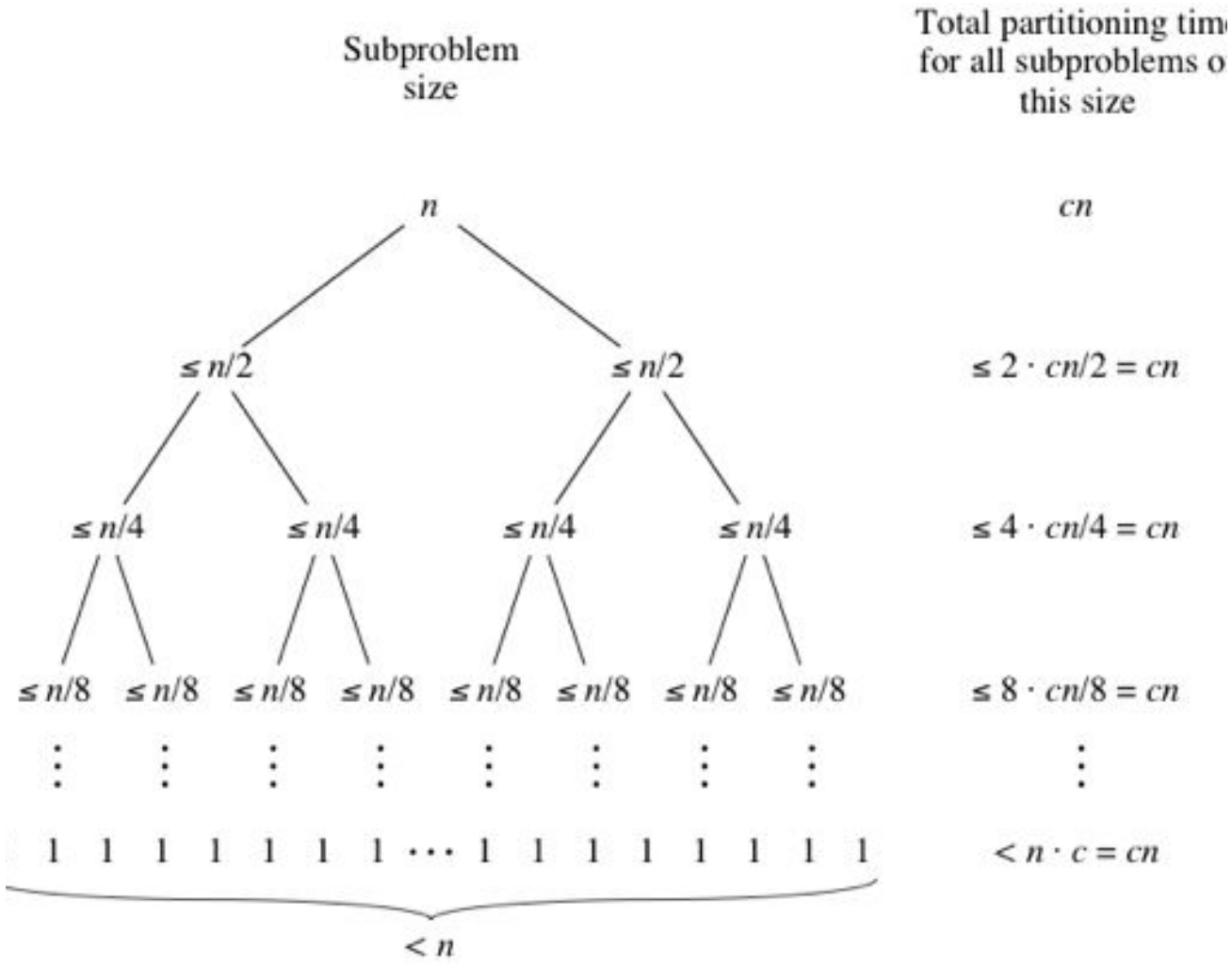
quicksort's worst-case running time is $\Theta(n^2)$

Quick Sort : Best-case running time

Occurs when the partitions are evenly partitioned.

quicksort's Best-case running time is

$$\Theta(n \log_2 n)$$



Quick Sort : Average-case running time

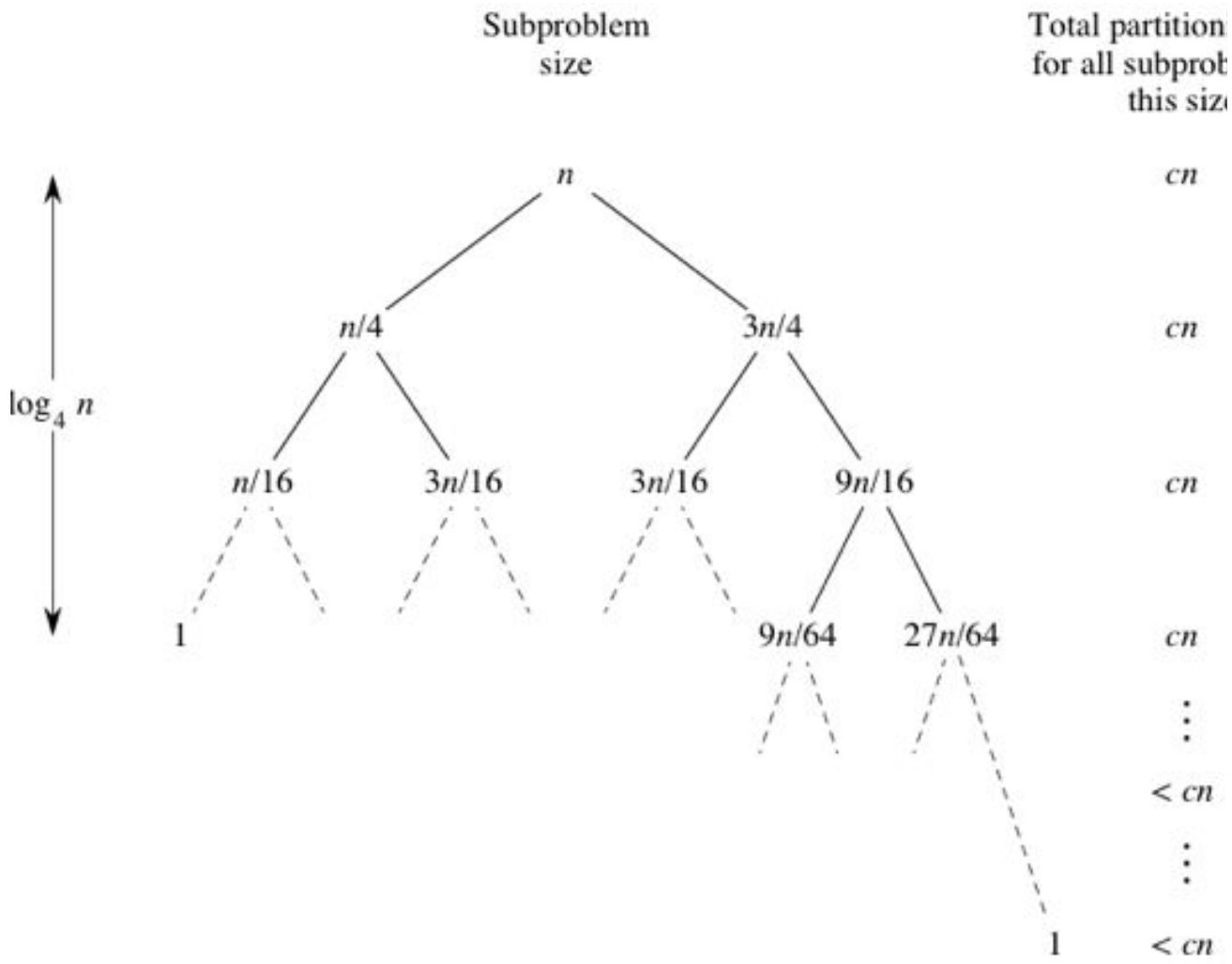
imagine that we don't always get evenly balanced partitions

but that we always get at worst a 3-to-1 split

That is, imagine that each time we partition, one side gets 3n/4 elements and the other side gets n/4 elements

quicksort's Average-case running time is

$$\Theta(n \log_2 n)$$



Home assignment

- Sort the following numbers using Quick sort

- 21, 45, 32, 76, 12, 83, 47, 153, 52
- 75, 34, 64, 82,35,79,12,53,40,61

Master Theorem

Master theorem

- Master theorem is used to determine running time of algorithms in terms of asymptotic notations.
- The master theorem is a formula for solving recurrences of the form T (n) = aT (n/b) + f (n), where a ≥ 1 and b > 1 and f(n) is asymptotically positive. (Asymptotically positive means that the function is positive for all sufficiently large n.)
- This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size n/b, and solves them recursively.

Master Theorem - Proof

The theorem is as follows:

Master Theorem:

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Theorem

- The master theorem compares the function $n^{\log_b a}$ to the function f(n). Intuitively, if $n^{\log_b a}$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(n^{\log_b a})$. If f(n) is larger (by a polynomial factor), then the solution is $T(n) = \Theta(f(n))$. If they are the same size, then we multiply by a logarithmic factor.
- Be warned that these cases are not exhaustive – for example, it is possible for f(n) to be asymptotically larger than $n^{\log_b a}$, but not larger by a polynomial factor (no matter how small the exponent in the polynomial is). For example, this is true when $f(n) = n^{\log_b a} \log n$. In this situation, the master theorem would not apply, and you would have to use another method to solve the recurrence.

Master Theorem

Master's theorem solves recurrence relations of the form-

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta\left(n^k \log^p n\right)$$

Master's Theorem

Case-01:

If $a > b^k$, then $T(n) = \theta\left(n^{\log_b a}\right)$

Case-02:

If $a = b^k$ and

If $p < -1$, then $T(n) = \theta\left(n^{\log_b a}\right)$

If $p = -1$, then $T(n) = \theta\left(n^{\log_b a}.\log^2 n\right)$

If $p > -1$, then $T(n) = \theta\left(n^{\log_b a}.\log^{p+1} n\right)$

Case-03:

If $a < b^k$ and

If $p < 0$, then $T(n) = O\left(n^k\right)$

If $p >= 0$, then $T(n) = \theta\left(n^k \log^p n\right)$

Master Theorem

Example 1:

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

We compare the given recurrence relation with

$T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-

$a = 3, b = 2, k = 2, p = 0$

Now, $a = 3$ and $b^k = 2^2 = 4$.

Clearly, $a < b^k$.

So, we follow case- 03.

Since $p = 0$, so we have-

$T(n) = \theta (n^k \log^p n)$

$T(n) = \theta (n^2 \log^0 n)$

Thus, **$T(n) = \theta (n^2)$**

Example 2:

$$T(n) = 2T(n/2) + n\log n$$

We compare the given recurrence relation with

$T(n) = aT(n/b) + \theta (n^k \log^p n)$.

Then, we have-

$a = 2, b = 2, k = 1, p = 1$

Now, $a = 2$ and $b^k = 2^1 = 2$.

Clearly, $a = b^k$.

So, we follow case-02.

Since $p = 1$, so we have-

$T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$

$T(n) = \theta (n^{\log_2 2} . \log^{1+1} n)$

$T(n) = \theta (n \log^2 n)$

Master Theorem

Example 3

$T(n) = 8T(n/4) – n^2 \log n$

- The given recurrence relation does not correspond to the general form of Master's theorem.
- So, it can not be solved using Master's theorem.

Example 4

$T(n) = 3T(n/3) + n/2$

- We write the given recurrence relation as $T(n) = 3T(n/3) + n$.
- This is because in the general form, we have θ for function f(n) which hides constants in it.
- Now, we can easily apply Master's theorem.

We compare the given recurrence relation with $T(n) = aT(n/b) + \theta (n^k \log^p n)$.
Then, we have-

$a = 3, b = 3, k = 1, p = 0$

Now, $a = 3$ and $b^k = 3^1 = 3$.
Clearly, $a = b^k$.
So, we follow case-02.

Since $p = 0$, so we have-

$T(n) = \theta (n^{\log_b a} . \log^{p+1} n)$
$T(n) = \theta (n^{\log_3 3} . \log^{0+1} n)$
$T(n) = \theta (n^1 . \log^1 n)$
$T(n) = \theta (n \log n)$

Master Theorem

Exercise Problems:

1. $T(n)=4T(n/2)+n2 \Rightarrow T(n)=\Theta(n2logn)$(Case2)
2. $T(n)=T(n/2)+2n \Rightarrow \Theta(2n)$(Case3)
3. $T(n) = 2nT(n/2) + nn \Rightarrow$ Does not apply (a is not constant)
4. $T(n)=16T(n/4)+n \Rightarrow T(n)=\Theta(n2)$(Case1)
5. $T(n)=2T(n/2)+nlogn \Rightarrow T(n)=nlog2n$(Case2)

Finding Minimum and Maximum

- **Session Learning Outcome-SLO:** Able to Solve Finding Minimum and Maximum using divide and conquer approaches

- **Motivation of the topic:** The problem is to find the ‘maximum’ and ‘minimum’ items in a set of ‘n’ elements

- Algorithm:
- *Algorithm MaxMin(A, n, max, min)*// **DIRECT APPROACH**
  // Set *max* to the maximum and *min* to the minimum of A[1..n]
  {
  max = min = A[1];
  for(i = 2 to n) do
  {
  if (A[i]>max) then max = A[i];
  if (A[i]<min) then min = A[i];
  }
  }
- The above algorithm requires 2(n-1) element comparisons in the best, average and worst cases.

- A *divide-and-conquer* algorithm for this problem would proceed as follows: Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list. Let small(P) be true when $n \leq 2$. In this case, the maximum and minimum are a[i] if n = 1. If n = 2, the problem can be solved by making one comparison.

- If the list has more than two elements, P has to be divided into smaller instances. For example, we might divide P into the two instances P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]). After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

- Now the question is How can we combine the Solutions for P1 and P2 to obtain the solution for P?
- If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

- MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately.

- For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of this sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

Algorithm for maximum and minimum using divide-and-conquer

```
MaxMin(i, j, max, min)
// a[1:n] is a global array. Parameters i and j are integers,   // 1≤i≤j≤n. The effect is to set max and min to the largest and  // smallest values in a[i:j].
{
    if (i=j) then max := min := a[i]; //Small(P)
    else if (i=j-1) then // Another case of Small(P)
        {
            if (a[i] < a[j]) then max := a[j]; min := a[i];
            else max := a[i]; min := a[j];
        }
    else
    {
        // if P is not small, divide P into sub-problems.
        // Find where to split the set.
        mid := ( i + j )/2;
        // Solve the sub-problems.
        MaxMin( i, mid, max, min );
        MaxMin( mid+1, j, max1, min1 );
        // Combine the solutions.
        if (max < max1) then max := max1;
        if (min > min1) then min := min1;
    }
```

Analysis

what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

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

When n is a power of two, $n = 2^k$

-for some positive integer k, then

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

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

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

.

.

$= 2^{k-1} T(2) + \sum(1 \le i \le k-1) 2^k$

$= 2^{k-1} + 2^k - 2$

$= 3n/2 - 2 = O(n)$

Note that $3n/2 - 2$ is the best, average, worst case number of comparison when n is a power of two.

Comparisons with Straight Forward Method:

Compared with the $2n - 2$ comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than $3n/2 - 2$ comparisons.

- Activity:
- The minimum number of comparisons required to find the minimum and the maximum of 100 numbers is _____.

Closest pair problem

- Closest-pair problem calls for finding the two closest points in a set of *n* Points.
- Cluster Analysis in statistics deals with closest pair problem
- Euclidian distance is used to calculate the closest pair problem

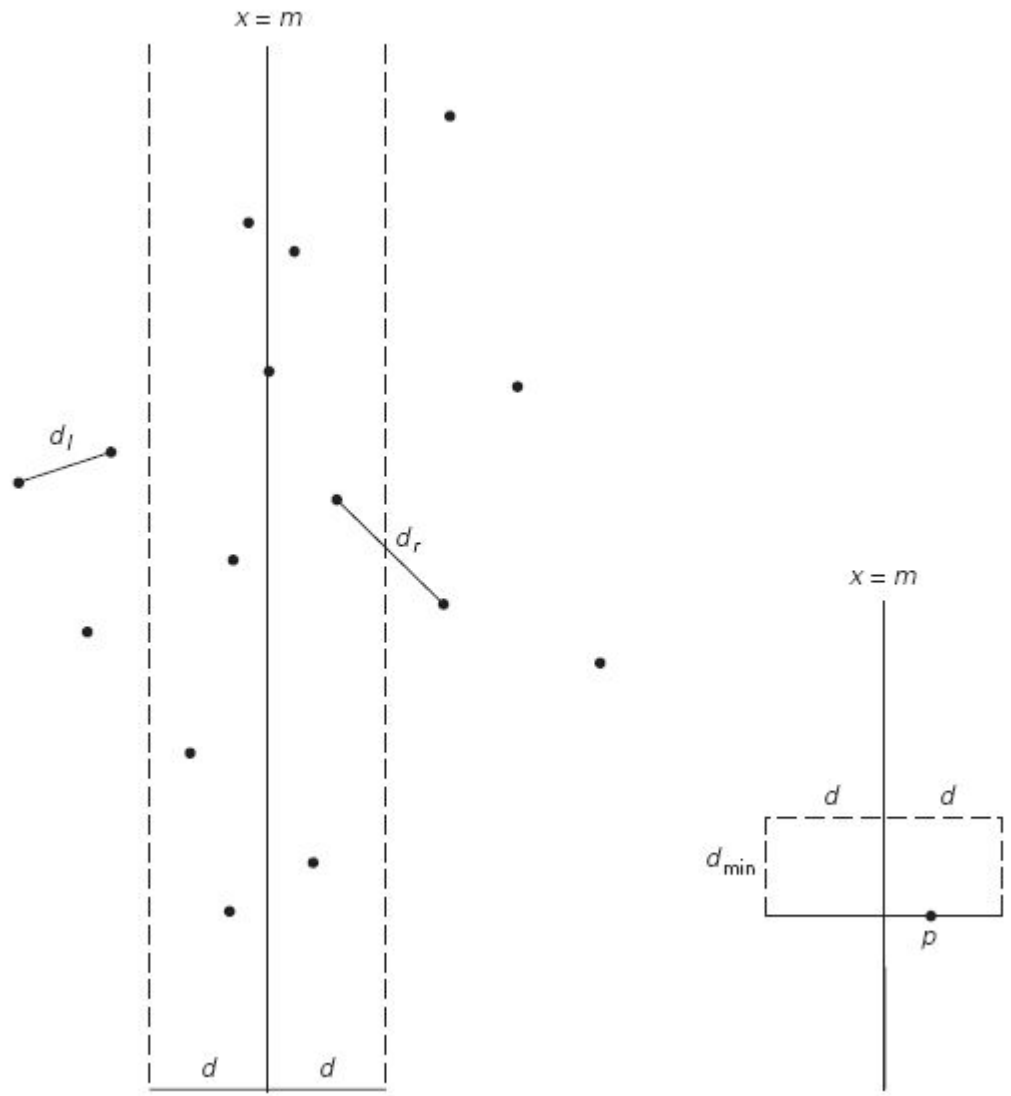
$$d_{xi,xj)} = \sqrt{(xi - xj)^2 + (yi - yj)^2}$$

- Let *P* be a set of *n* > 1 points in the Cartesian plane , we assume points are distinct
- Assume that the points are ordered in nondecreasing order of their *x* coordinate and y coordinate

Closest pair problem

- Closest-pair problem calls for finding the two closest points in a set of *n* Points.
- Divide the points into equal halves
- After dividing the points into two equal halves find the closest point in each equal halves
- Find the distances among each closest point in each halves using Euclidean distance
- Store the distances calculated in a array
- Find the smallest distance in the array an the return the point for which we received minimum distance is closest pair

Closest pair problem -Example



Closest pair problem -Motivation

- When number of points or number of items are available we have to find the closest items or points available
- We find the distance between each pair of distinct points and find a pair with the smallest distance
- A naive algorithm of finding distances between all pairs of points in a space of dimension d and selecting the minimum requires $O(n^2)$ time
- But while we go for calculating distance between two points using Euclidean distance , The total time taken is O(nlogn)
- So Euclidean distance is used in closest pair using divide and conquer mechanism
- Time taken for computation using Brute Force approach is $O(n^2)$

Closest pair problem -Algorithm

if $n \leq 3$
    return the minimal distance found by the brute-force algorithm
else
    copy the first $\_n/2\_$ points of P to array Pl
    copy the same $\_n/2\_$ points from Q to array Ql
    copy the remaining $\_n/2\_$ points of P to array Pr
    copy the same $\_n/2\_$ points from Q to array Qr
    $dl \leftarrow EfficientClosestPair(Pl, Ql)$
    $dr \leftarrow EfficientClosestPair(Pr, Qr)$
    $d \leftarrow \min\{dl, dr\}$
    $m \leftarrow P[\_n/2\_ - 1].x$
    copy all the points of Q for which $|x - m| < d$ into array $S[0..num - 1]$
    $dminsq \leftarrow d2$
for $i \leftarrow 0$ to $num - 2$ do
    $k \leftarrow i + 1$
while $k \leq num - 1$ and $(S[k].y - S[i].y)2 < dminsq$
    $dminsq \leftarrow \min((S[k].x - S[i].x)2 + (S[k].y - S[i].y)2, dminsq)$
$k \leftarrow k + 1$
return $sqrt(dminsq)$

Closest pair problem -Analysis

- Running time of the algorithm (without sorting) is:

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in \Theta(n)$$

- By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in \Theta(n \log n)$$

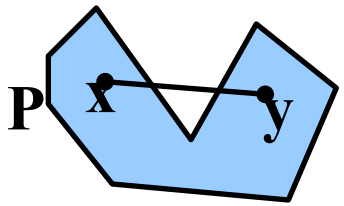
- So the total time is $\Theta(n \log n)$.

Real time applications

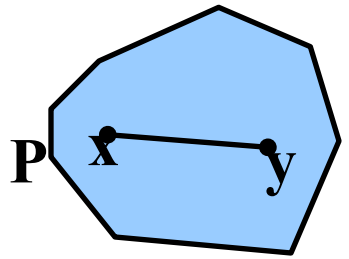
- Air/land/water traffic control system
- Collision avoidance

Convex vs. Concave

- A polygon P is convex if for every pair of points x and y in P, the line xy is also in P; otherwise, it is called concave.



concave



convex

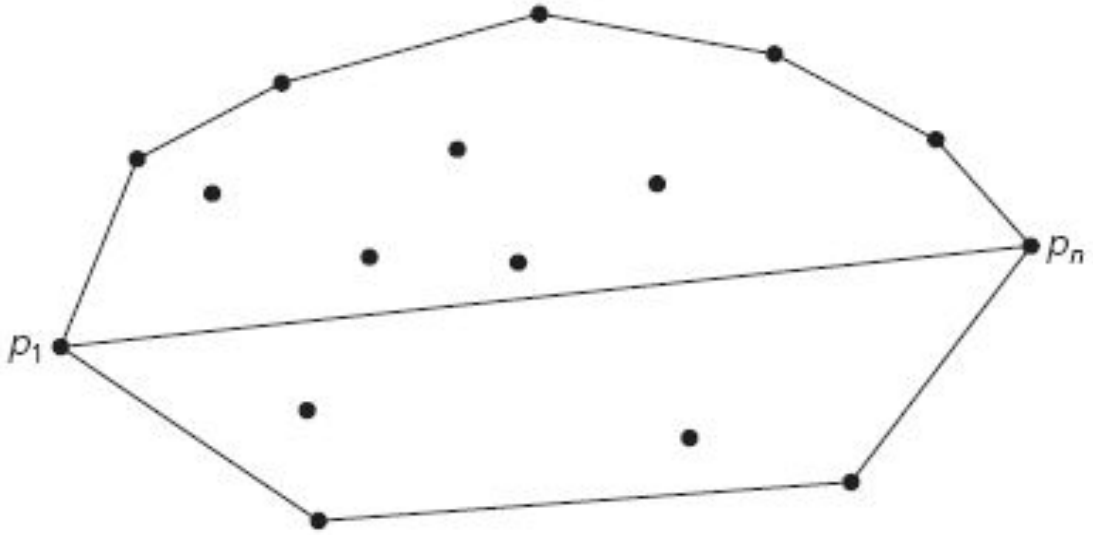
Convex hull Problem

- Convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it

- Convex Hull is the line completely enclosing a set of points in a plane so that there are no concavities in the line

Convex hull Problem

- Let *S* be a set of*n*>1 points *p*1*(x*1*, y*1*), . . . , pn(xn, yn)* in the Cartesian plane



- Splitted in upper and lower half

Convex hull Problem -Algorithm

```
vector<pair<int, int>> divide(vector<pair<int, int>> a)
{
    if (a.size() <= 5)
       return bruteHull(a);
   vector<pair<int, int>>left, right;
   for (int i=0; i<a.size()/2; i++)
       left.push_back(a[i]);
   for (int i=a.size()/2; i<a.size(); i++)
       right.push_back(a[i]);
    vector<pair<int, int>>left_hull = divide(left);
    vector<pair<int, int>>right_hull = divide(right);
    return merger(left_hull, right_hull);
}
```

Convex Hull - Time Complexity

- Time complexity If points are not initially sorted O(n log n)
- Time efficiency: T(n) = T(n-1) + O(n)

  T(n) = T(x) + T(y) + T(z) + T(v) + O(n), where x + y + z +v <= n.

  worst case: $\Theta(n^2)$

  average case: $\Theta(n)$

Real time applications

- Collision avoidance

Assignment

- Implementation of Closest pair problem
- Implementation of Convex hull problem

Strassen's Matrix Multiplication

Dr. Anand M
Id: 102763
Assistant Professor
Department of Computer Science and Engineering
SRM Institute of Science and Technology

Basic Matrix Multiplication

Suppose we want to multiply two matrices of size N x N: for example A x $B = C$.

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

$$\mathbf{C_{11} = a_{11}b_{11} + a_{12}b_{21}}$$

$$\mathbf{C_{12} = a_{11}b_{12} + a_{12}b_{22}}$$

$$\mathbf{C_{21} = a_{21}b_{11} + a_{22}b_{21}}$$

$$\mathbf{C_{22} = a_{21}b_{12} + a_{22}b_{22}}$$

2x2 matrix multiplication can be accomplished in 8 multiplication.($\mathbf{2^{\log_2 8} = 2^3}$)

Basic Matrix Multiplication

```
void matrix_mult (){
  for (i = 1; i <= N; i++) {
        for (j = 1; j <= N; j++) {
        compute C_{i,j};
                                }
  }}
```

algorithm

Time analysis

$$C_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$$

$$\text{Thus } T(N) = \sum_{i=1}^{N} \sum_{j=1}^{N} \sum_{k=1}^{N} c = cN^3 = O(N^3)$$

Strassens's Matrix Multiplication

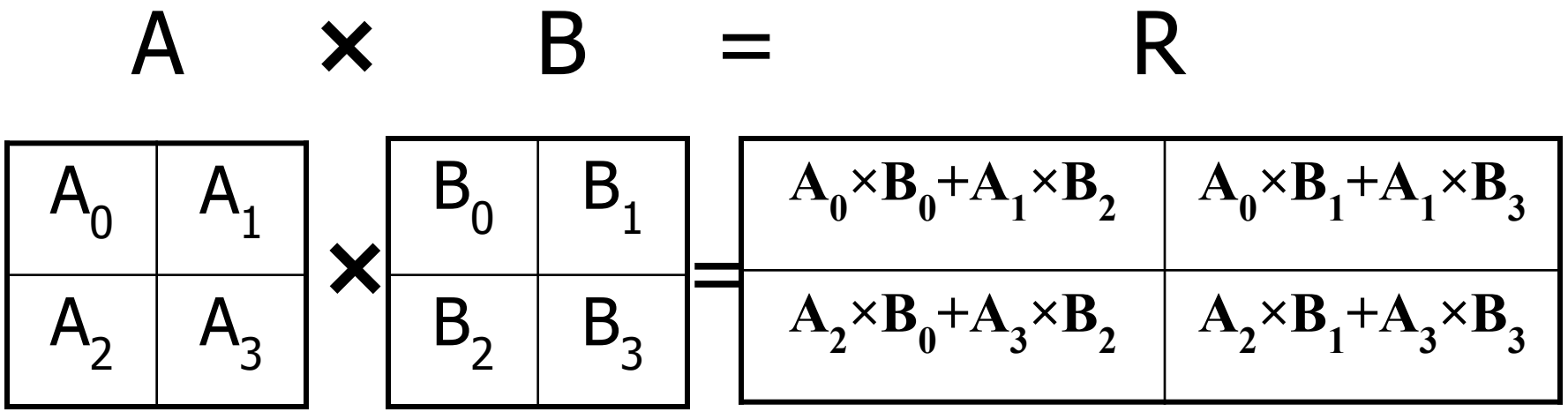
- Strassen showed that 2x2 matrix multiplication can be accomplished in 7 multiplication and 18 additions or subtractions. .($2^{\log_2 7} = 2^{2.807}$)

- This reduce can be done by Divide and Conquer Approach.

Divide-and-Conquer

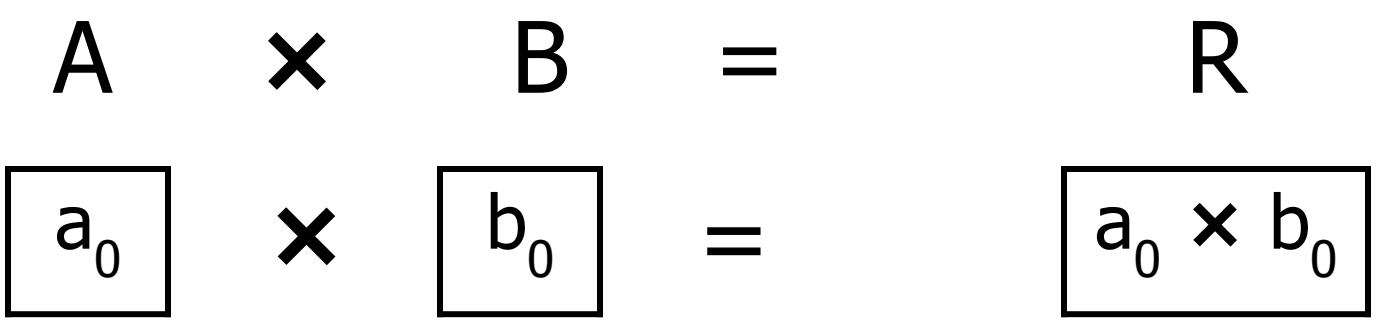
- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $\boldsymbol{S}$ in two or more disjoint subsets $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …
  - Recur: solve the subproblems recursively
  - Conquer: combine the solutions for $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, into a solution for $\boldsymbol{S}$
- The base case for the recursion are subproblems of constant size
- Analysis can be done using **recurrence equations**

Divide and Conquer Matrix Multiply



- Divide matrices into sub-matrices: A_0 , A_1, A_2 etc
- Use blocked matrix multiply equations
- Recursively multiply sub-matrices

Divide and Conquer Matrix Multiply



- Terminate recursion with a simple base case

Strassens's Matrix Multiplication

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

$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$
$P_2 = (A_{21} + A_{22}) * B_{11}$
$P_3 = A_{11} * (B_{12} - B_{22})$
$P_4 = A_{22} * (B_{21} - B_{11})$
$P_5 = (A_{11} + A_{12}) * B_{22}$
$P_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$
$P_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{12} = P_3 + P_5$
$C_{21} = P_2 + P_4$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Comparison

$$
\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
&= (A_{11} + A_{22})(B_{11} + B_{22}) + A_{22} * (B_{21} - B_{11}) - (A_{11} + A_{12}) * B_{22} + \\
&\quad (A_{12} - A_{22}) * (B_{21} + B_{22}) \\
&= A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} - \\
&\quad A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22} \\
&= A_{11} B_{11} + A_{12} B_{21}
\end{aligned}
$$

Strassen Algorithm

```
void matmul(int *A, int *B, int *R, int n) {
  if (n == 1) {
       (*R) += (*A) * (*B);
  } else {
       matmul(A, B, R, n/4);
       matmul(A, B+(n/4), R+(n/4), n/4);
       matmul(A+2*(n/4), B, R+2*(n/4), n/4);
       matmul(A+2*(n/4), B+(n/4), R+3*(n/4), n/4);
       matmul(A+(n/4), B+2*(n/4), R, n/4);
       matmul(A+(n/4), B+3*(n/4), R+(n/4), n/4);
       matmul(A+3*(n/4), B+2*(n/4), R+2*(n/4), n/4);
       matmul(A+3*(n/4), B+3*(n/4), R+3*(n/4), n/4);
  }
```

Divide matrices in sub-matrices and recursively multiply sub-matrices

Time Analysis

$$T(1) = 1 \qquad (\text{assume } N = 2^k)$$

$$T(N) = 7T(N/2)$$

$$T(N) = 7^k T(N/2^k) = 7^k$$

$$T(N) = 7^{\log N} = N^{\log 7} = N^{2.81}$$

Assignment Work

1. Verify the formulas of Strassen's algorithm for multiplying 2 matrices.
2. Apply Strassen's algorithm to compute.

$$\begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Largest Subarray Problem

Dr. Anand M
Id: 102763
Assistant Professor
Department of Computer Science and Engineering
SRM Institute of Science and Technology

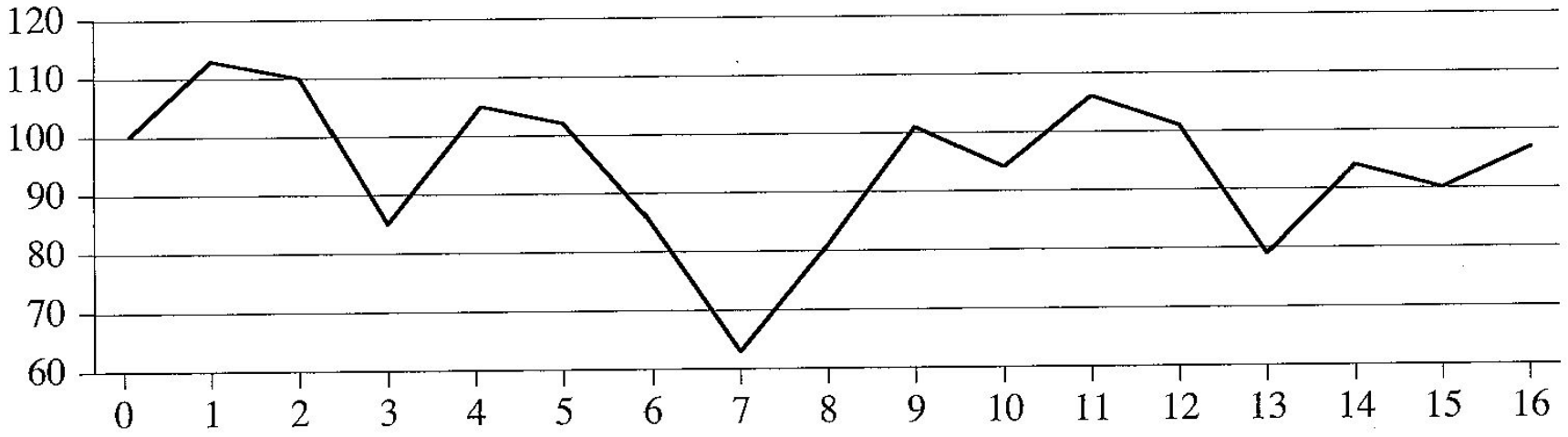
Largest Subarray Problem

Problem: given an array of n numbers, find the (a) contiguous subarray whose sum has the largest value.

Application: an unrealistic stock market game, in which you decide when to buy and see a stock, with full knowledge of the past and future. ***The restriction*** is that you can perform just one ***buy*** followed by a ***sell***. The buy and sell both occur right after the close of the market.

The interpretation of the numbers: each number represents the stock value at closing on any particular day.

Largest Subarray Problem

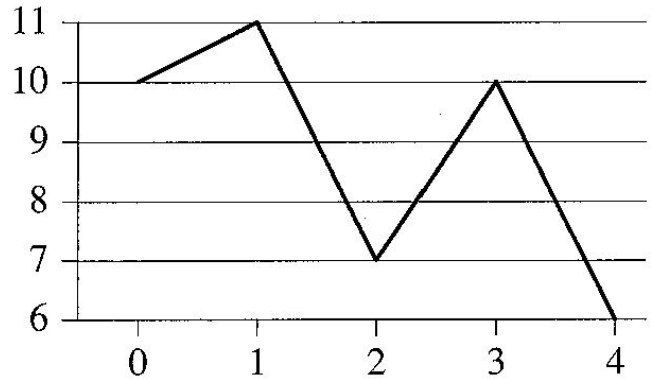


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Largest Subarray Problem

Another Example: buying low and selling high, even with perfect knowledge, is not trivial:



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of $3 per share would be earned by buying after day 2 and selling after day 3. The price of $7 after day 2 is not the lowest price overall, and the price of $10 after day 3 is not the highest price overall.

A brute-force solution

- $O(n^2)$ Solution
- Considering ${C_n}^2$ pairs
- Not a pleasant prospect if we are rummaging through long time-series (Who told you it was easy to get rich???), even if you are allowed to post-date your stock options…

A Better Solution: Max Subarray

Transformation: Instead of the daily price, let us consider the daily change: A[i] is the difference between the closing value on day i and that on day i-1.

The problem becomes that of finding a contiguous subarray the sum of whose values is maximum.

- On a first look this seems even worse: roughly the same number of intervals (one fewer, to be precise), and the requirement to add the values in the subarray rather than just computing a difference: $\Omega(n^3)$?

Max Subarray

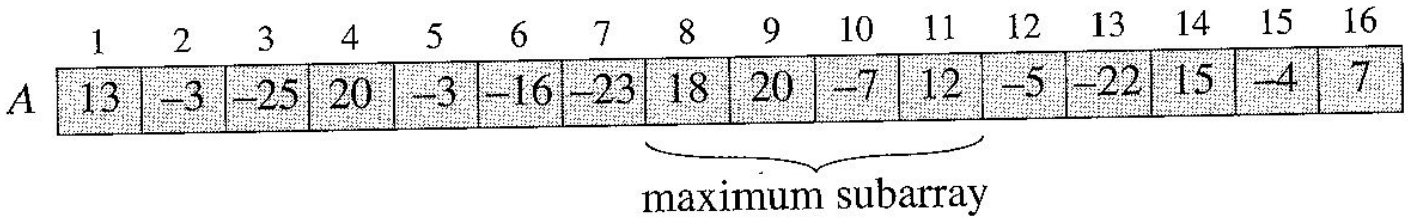


Figure 4.3 The change in stock prices as a maximum-subarray problem. Here, the subarray $A[8..11]$, with sum 43, has the greatest sum of any contiguous subarray of array A.

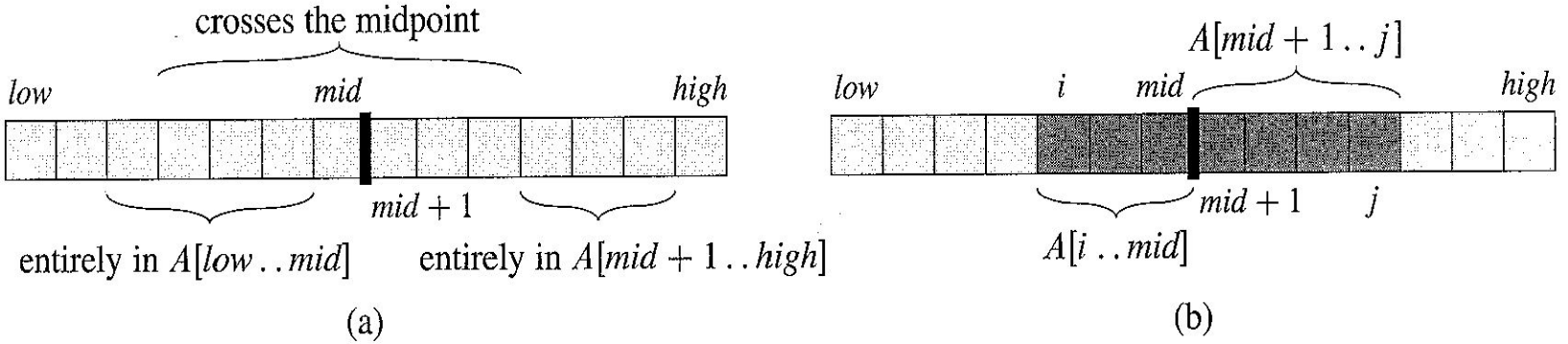


Figure 4.4 **(a)** Possible locations of subarrays of $A[low..high]$: entirely in $A[low..mid]$, entirely in $A[mid + 1..high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low..high]$ crossing the midpoint comprises two subarrays $A[i..mid]$ and $A[mid + 1..j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max Subarray

How do we divide?

We observe that a maximum contiguous subarray A[i…j] must be located as follows:

1. It lies entirely in the left half of the original array: [low…mid];
2. It lies entirely in the right half of the original array: [mid+1…high];
3. It straddles the midpoint of the original array: i ≤ mid < j.

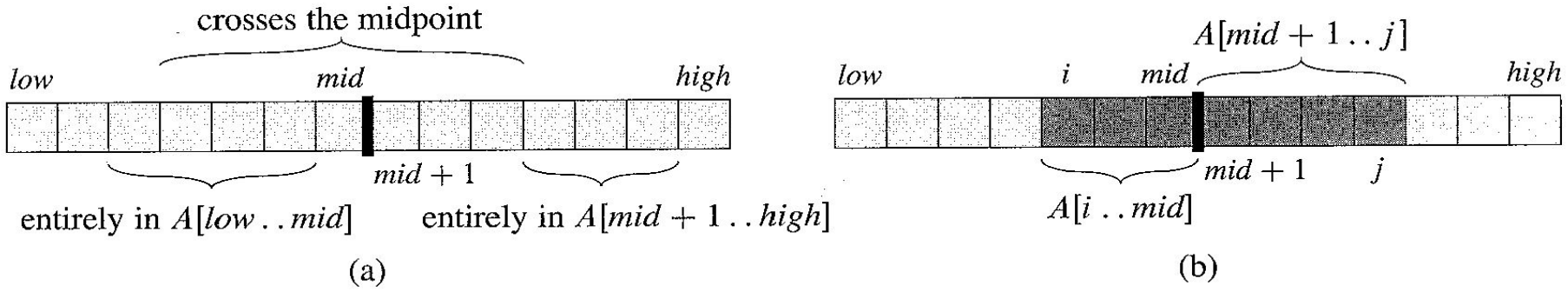


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint mid. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

Max Subarray: Divide & Conquer

- The “left” and “right” subproblems are smaller versions of the original problem, so they are part of the standard Divide & Conquer recursion.
- The “middle” subproblem is not, so we will need to count its cost as part of the “combine” (or “divide”) cost.
  - The crucial observation (and it may not be entirely obvious) is that **we can find the maximum crossing subarray in time linear in the length of the A[low…high] subarray**.

How? A[i,…,j] must be made up of A[i…mid] and A[m+1…j] – so we find the largest A[i…mid] and the largest A[m+1…j] and combine them.

The middle subproblem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Algorithms: Max Subarray

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 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 ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

Max Subarray: Algorithm Efficiency

We finally have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution $T(n) = \Theta(n \lg n)$.

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

Assignment Work

1. Write a pseudocode for a divide-and-conquer algorithm for finding the position of the largest element in an array of n numbers.
2. Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.
3. How does this algorithm compare with the brute-force algorithm for this problem?

Time complexity analysis of Largest sub-array sum

Dr. Anand M
Id: 102763
Assistant Professor
Department of Computer Science and Engineering
SRM Institute of Science and Technology

Max Subarray: Algorithm Efficiency

We finally have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The recurrence has the same form as that for MERGESORT, and thus we should expect it to have the same solution $T(n) = \Theta(n \lg n)$.

This algorithm is clearly substantially faster than any of the brute-force methods. It required some cleverness, and the programming is a little more complicated – but the payoff is large.

Time complexity analysis

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

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + cn \\ &\leq 2[2T(\frac{n}{4}) + c\frac{n}{2}] + cn = 4T(\frac{n}{4}) + 2cn \quad \text{1st expansion} \\ &\leq 4[2T(\frac{n}{8}) + c\frac{n}{4}] + 2cn = 8T(\frac{n}{8}) + 3cn \quad \text{2nd expansion} \\ &\vdots \\ &\leq 2^k T(\frac{n}{2^k}) + kcn \quad \text{k}^{\text{th}} \text{ expansion} \end{aligned}$$

The expansion stops when $2^k = n$

$$\begin{aligned} T(n) &\leq nT(1) + cn\log_2 n \\ &= O(n) + O(n\log n) \\ &= O(n\log n) \end{aligned}$$

Time complexity analysis

- Theorem

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & \text{if } n \geq 2 \end{cases} \Rightarrow T(n) = O(n \log n)$$

- Proof
  - There exists positive constant a, b s.t. $T(n) \leq \begin{cases} a & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n & \text{if } n \geq 2 \end{cases}$
  - Use induction to prove $T(n) \leq 2b \cdot n \log_2 n + a \cdot n$
    - n = 1, trivial
    - n > 1, $\lceil \frac{n}{2} \rceil \leq \frac{n}{\sqrt{2}}$

$$\begin{aligned} T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n \\ &\overset{\text{Inductive hypothesis}}{\leq} 2b \cdot (\lceil n/2 \rceil \log_2 \lceil n/2 \rceil) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \frac{n}{\sqrt{2}}) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \frac{n}{\sqrt{2}}) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &= 2b \cdot n(\log n - \log_2 \sqrt{2}) + a \cdot n + b \cdot n = 2b \cdot n \log_2 n + a \cdot n \end{aligned}$$