

Design and Analysis of Algorithm (KCS503)

Introduction to Algorithms and its analysis mechanism

Lecture - 1

Overview

- Provide an overview of algorithms and analysis.
- Try to touch the distinguishing features and the significance of analysis of algorithm.
- Start using frameworks for describing and analysing algorithms.
- See how to describe algorithms in pseudo code in the context of real world software development.
- Begin using asymptotic notation to express running-time analysis with Examples.

What is an Algorithm ?

- An algorithm is a finite set of rules that gives a sequence of operations for solving a specific problem
- An algorithm is any well defined computational procedure that takes some value or set of values, as input and produces some value or set of values as output.
- We can also view an algorithm as a tool for solving a well specified computational problem.

Characteristics of an Algorithm

- **Input:** provided by the user.
- **Output:** produced by algorithm.
- **Definiteness:** clearly define.
- **Finiteness:** It has finite number of steps.
- **Effectiveness:** An algorithm must be effective so that it's output can be carried out with the help of paper and pen.

Analysis of an Algorithm

- **Loop Invariant** technique was done in three steps:
 - Initialization
 - Maintenance
 - Termination
- It deals with predicting the resources that an algorithm requires to its completion such as memory and CPU time.
- To main measure for the efficiency of an algorithm are Time and space.

Complexity of an Algorithm

- Complexity of an Algorithm is a function, $f(n)$ which gives the running time and storage space requirement of the algorithm in terms of size n of the input data.
- Space Complexity: Amount of memory needed by an algorithm to run its completion.
- Time complexity: Amount of time that it needs to complete itself.

Cases in Complexity Theory

- Best Case: Minimum time
- Worst Case: Maximum amount of time
- Average Case: Expected / Average value of the function $f(n)$.

Example 1

```
#include <stdio.h>

int main()
{
    printf("Hello PSIT");
    return 0;
}
```

Complexity of Example 1

```
#include <stdio.h>

int main()
{
    printf("Hello PSIT");
    return 0;
}
```

$$T(n) = O(1)$$

Example 2

```
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello PSIT !!!\n");
    }
}
```

Complexity of Example 2

```
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello PSIT !!!\n");
    }
}
```

$$T(n) = O(n)$$

Example 3

```
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        printf("Hello PSIT !!!\n");
    }
}
```

Complexity of Example 3

```
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i=i*2) {
        printf("Hello PSIT !!!\n");
    }
}
```

$$T(n) = \log_2 n$$

Example 4

```
#include <stdio.h>
#include <math.h>
void main()
{
    int i, n = 8;
    for (i = 2; i <= n; i=pow(i,2)) {
        printf("Hello PSIT !!!\n");
    }
}
```

Complexity of Example 4

```
#include <stdio.h>
#include <math.h>
void main()
{
    int i, n = 8;
    for (i = 2; i <= n; i=pow(i,2)) {
        printf("Hello PSIT !!!\n");
    }
}
```

$$T(n) = O(\log_2 (\log_2 n))$$

Example 5

```
A()
{
    int i;
    for(i=1;i<=n;i++)
    {
        printf("ABCD");
    }
}
```

Complexity of Example 5

```
A()
{
    int i;
    for(i=1;i<=n;i++)
    {
        printf("ABCD");
    }
}
```

$$T(n) = O(n)$$

Example 6

```
A()
{
    int i=1 ,s=1;
    scanf("%d", &n);
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("abcd");
    }
}
```

Complexity of Example 6

```
A()
{
    int i=1 ,s=1;
    scanf("%d", &n);
    while(s<=n)
    {
        i++;
        s=s+i;
        printf("abcd");
    }
}
```

$$T(n) = O(\sqrt{n})$$

Example 7

```
A()
{
    int i=1;
    for(i=1; i2<=n; i++)
    {
        printf("abcd");
    }
}
```

Complexity of Example 7

```
A()
{
    int i=1;
    for(i=1; i2<=n; i++)
    {
        printf("abcd");
    }
}
```

$$T(n) = O(\sqrt{n})$$

Example 8

```
A()
{
    int i=1;
    for (i=1; i≤n; i++)
    {
        for (j=1; j≤ i2; j++)
        {
            for (k=1; k≤n/2; k++)
            {
                printf("abcd");
            }
        }
    }
}
```

Complexity of Example 8

$$T(n) = O(n^4)$$

Explanation:

$$I = 1, 2, 3, 4, 5, \dots \dots \dots \dots \dots \dots$$

$$J = 1, 4, 9, 16, 25, \dots \dots \dots \dots \dots \dots$$

$$K = \frac{n}{2}, \frac{4n}{2}, \frac{9n}{2}, \frac{16n}{2}, \dots \dots \dots \dots \dots$$

Complexity of Example 8

$$T(n) = O(n^4)$$

Explanation:

$$I = 1, 2, 3, 4, 5, \dots \dots \dots \dots \dots \dots$$

$$J = 1, 4, 9, 16, 25, \dots \dots \dots \dots \dots \dots$$

$$K = \frac{n}{2}, \frac{4n}{2}, \frac{9n}{2}, \frac{16n}{2}, \dots \dots \dots$$

Sum of square of Natural Number.
= $[n(n + 1)(2n + 1)]/6$

Design and Analysis of Algorithm (KCS503)

Analysis of Linear Search

Lecture - 2

Searching

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching

Searching Algorithms are a family of algorithms used for the purpose of searching.

The searching of an element in the given array may be carried out in the following two ways-

- Linear Searching
- Binary Searching

Linear Searching

To understand the working of linear search algorithm, let's take an unsorted array. It will be easy to understand the working of linear search with an example.

Let the elements of array are :

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

Let the element to be searched is $K = 41$

Linear Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

The value of K=41, is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.

Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 40$

Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 40$

Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 40$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 11$

Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 70$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 40$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 11$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

↑
 $K \neq 57$

Searching

Now, start from the first element and compare **K=41** with each element of the array.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 70$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 40$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K = 41$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 30$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 11$

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

$K \neq 57$

Searching

Now, the element to be searched is found. So, algorithm will return the index of the element matched.

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K ≠ 70

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K ≠ 40

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K = 41

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K ≠ 30

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K ≠ 11

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K ≠ 57

0	1	2	3	4	5	6	7	8
70	40	30	11	57	41	25	14	52

K = 41

Linear Searching

What Is Linear Search?

- A linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as Sequential Search.

Linear Searching

When Linear Search Algorithm is applied-

- No information is given about the array.
- The given array is unsorted, or the elements are unordered.
- The list of data items is smaller.

Linear Searching

Linear Search Algorithm –

```
Linear_Search (a , n , item , loc)
```

```
Begin
```

```
for i = 0 to (n - 1) by 1 do
```

```
    if (a[i] = item) then
```

```
        set loc = i
```

```
        exit
```

```
    endif
```

```
end for
```

```
set loc = -1
```

```
End
```

Linear Searching

Time Complexity Analysis- (Best case)

In the best possible case,

- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes $O(1)$ operations.

Linear Searching

Time Complexity Analysis- (Worst Case)

In the worst possible case,

- The element being searched may be present at the last position or not present in the array at all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus in worst case, linear search algorithm takes $O(n)$ operations.

Linear Searching

Time Complexity Analysis- (Worst Case)

In the worst possible case,(Mathematically)

$$T(n) = O(1) + T(n - 1)$$

Linear Searching

Time Complexity Analysis- (Worst Case)

In the worst possible case,(Mathematically)

$$\begin{aligned}T(n) &= O(1) + T(n - 1) \\&= O(1) + O(1) + T(n - 2)\end{aligned}$$

Linear Searching

Time Complexity Analysis- (Worst Case)

In the worst possible case,(Mathematically)

$$\begin{aligned} T(n) &= O(1) + T(n - 1) \\ &= O(1) + O(1) + T(n - 2) \\ &= O(1) + O(1) + O(1) + T(n - 3) \\ &= O(1) + O(1) + O(1) + O(1) + T(n - 4) \\ &\quad \text{after } n \text{ times of iteration} \\ &= O(1) + O(1) + O(1) + O(1) + \dots + O(1) + T(0) \\ &= O(n) + O(0) \\ &= O(n) \end{aligned}$$

Linear Searching

Time Complexity Analysis-

Thus, we can say that in general:

The time Complexity of Linear Search Algorithm is $O(n)$. where, n is the number of elements in the linear array.

Design and Analysis of Algorithm (KCS503)

**Definition of Sorting problem through
exhaustive search and analysis of
Selection Sort through iteration Method**

Lecture -3

A Sorting Problem (Exhaustive Search Approach)

Input: A sequence of n numbers A_1, A_2, \dots, A_n .

Output: A permutation (reordering) A_1, A_2, \dots, A_n of the input sequence such that $A_1 \leq A_2 \leq \dots \leq A_n$.

The sequences are typically stored in arrays.

A Sorting Problem (Exhaustive Search Approach)

Let there be a set of **four** digits and note that there are multiple possible permutations for the four digits. They are:

1 2 3 4

1 2 4 3

1 3 2 4

1 3 4 2

1 4 2 3

1 4 3 2

2 1 3 4

2 1 4 3

2 3 1 4

2 3 4 1

2 4 3 1

2 4 1 3

3 1 2 4

3 1 4 2

3 2 1 4

3 2 4 1

3 4 1 2

3 4 2 1

4 1 2 3

4 1 3 2

4 2 1 3

4 2 3 1

4 3 1 2

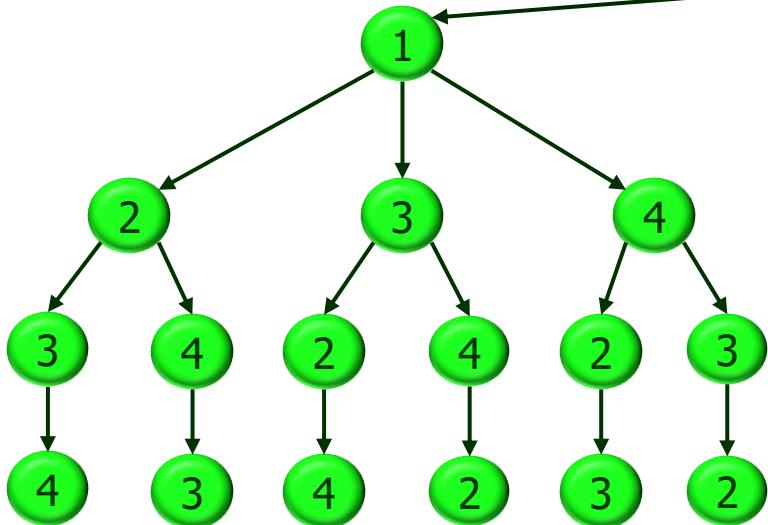
4 3 2 1

- There are 24 different permutations possible. (as shown above)
- Only one of these permutations meets our criteria.
(i.e. $A_1 \leq A_2 \leq \dots \leq A_n$) .



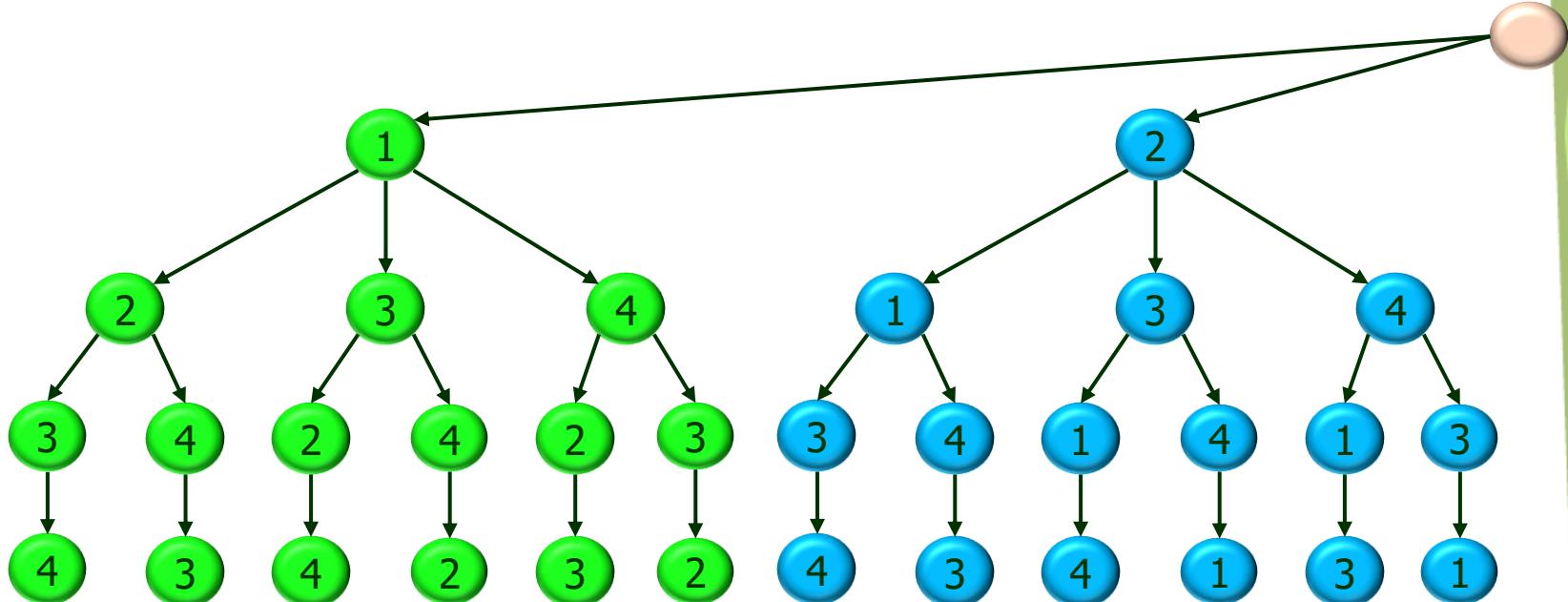
A Sorting Problem (Exhaustive Search Approach)

How to generate the 24 permutation ?



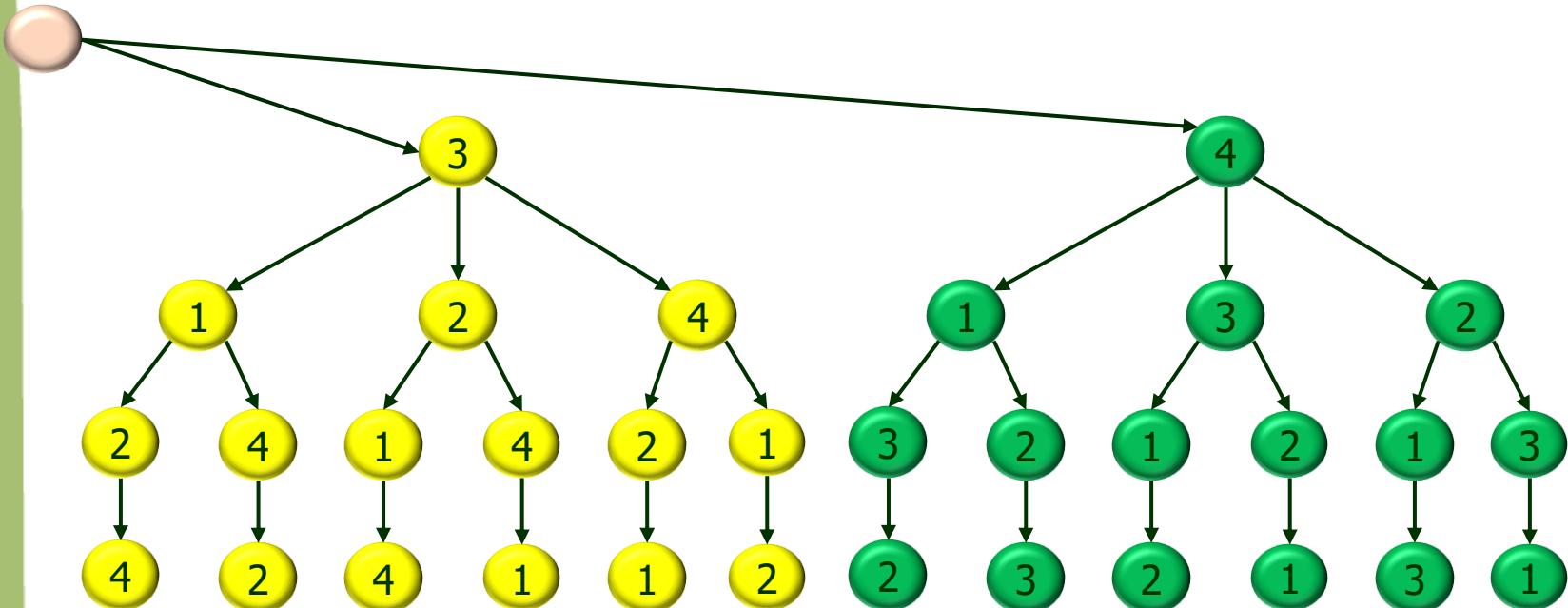
A Sorting Problem (Exhaustive Search Approach)

How to generate the 24 permutation ?



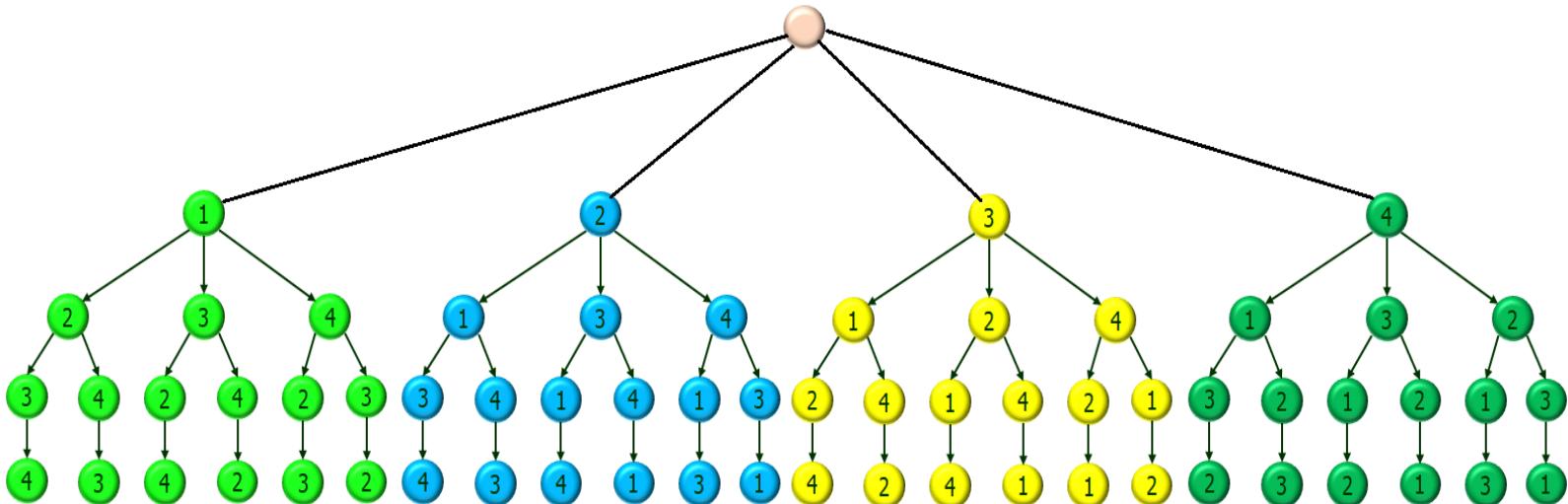
A Sorting Problem (Exhaustive Search Approach)

How to generate the 24 permutation ?



A Sorting Problem (Exhaustive Search Approach)

An in-depth look at the analysis of the 24 permutations of four digits



A Sorting Problem (Exhaustive Search Approach)

Let there be a set of **four** digits and note that there are multiple possible permutations for the four digits. They are:

1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 3 1	3 4 1 2	4 3 1 2
1 4 3 2	2 4 1 3	3 4 2 1	4 3 2 1

- There are 24 different permutations possible. (as shown above)
- Only one of these permutations meets our criteria.
(i.e. $A_1 \leq A_2 \leq \dots \leq A_n$) . **(1 2 3 4)**

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a_1 \leq a_2 \leq \dots \leq a_n$).

Step 3: Once we get it , we got the victory.

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a_1 \leq a_2 \leq \dots \leq a_n$).

Step 3: Once we get it , we got the victory.

How we do this in algo based?

For each permutation $P \in$ set of $n!$ permutations:

```
if ( $a_1 \leq a_2 \leq \dots \leq a_n$ ) == permutation set[p]:  
    print (permutation set[p])
```

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a_1 \leq a_2 \leq \dots \leq a_n$).

Step 3: Once we get it , we got the victory.

**Exhaustive
Search**

How we do this in algo based?

For each permutation $P \in$ set of $n!$ permutations:

```
if ( $a_1 \leq a_2 \leq \dots \leq a_n$ ) == permutation set[p]:  
    print (permutation set[p])
```

A Sorting Problem (Exhaustive Search Approach)

How we do this ?

Step 1: Generate all the permutation and store it.

Step 2: **Check all the permutation one by one and find which permutation is satisfying the required condition (i.e. $a_1 \leq a_2 \leq \dots \leq a_n$).**

Step 3: Once we get it , we got the victory.

**Exhaustive
Search**

How we do this in algo based?

For each permutation $P \in$ set of $n!$ permutations:

```
if ( $a_1 \leq a_2 \leq \dots \leq a_n$ ) == permutation set[p]:  
    print (permutation set[p])
```

**Complexity
= $O(n! * n)$ time**

A Sorting Problem (Selection Sort Approach)

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

A Sorting Problem (Selection Sort Approach)

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

Lets consider the following array as an example:
 $A [] = (7, 4, 3, 6, 5)$.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:

$A [] = (7, 4, 3, 6, 5)$.

- For the first position in the sorted array, the whole array is traversed from index 0 to 4 sequentially. After going through the entire array, it is evident that 3 is the lowest value, with 7 being stored at the first position.
- Thus, replace 7 with 3. At the end of the first iteration, the item with the lowest value, in this case 3, at position 2 is most likely to be at the top of the sorted list.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:

$A [] = (7, 4, 3, 6, 5)$.

- 1st Iteration

Value	7	4	3	6	5
index	0	1	2	3	4
Value	7	4	3	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4

A Sorting Problem (Selection Sort Approach)

Lets consider the updated array as an example:

$A [] = (3, 4, 7, 6, 5)$.

- For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.
- Using the traversal method, we determined that the value 12 is the second-lowest in the array and thus should be placed in the second position. **So no need of swapping.**

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:

$A [] = (7, 4, 3, 6, 5)$.

- 2nd Iteration

Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4

A Sorting Problem (Selection Sort Approach)

Lets consider the updated array as an example:

$A [] = (3, 4, 7, 6, 5)$.

- For the third position, where 7 is present, again traverse the rest of the array in a sequential manner.
- Using the traversal method, we determined that the value 5 is the third-lowest in the array and thus should be placed in the third position.

A Sorting Problem (Selection Sort Approach)

Lets consider the following array as an example:

$A [] = (7, 4, 3, 6, 5)$.

- 3rd Iteration

Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	7	6	5
index	0	1	2	3	4
Value	3	4	5	6	7
index	0	1	2	3	4
Value	3	4	5	6	7
index	0	1	2	3	4

A Sorting Problem (Selection Sort Approach)

Lets consider the updated array as an example:

$A [] = (3, 4, 5, 6, 7)$.

- Similarly we execute it for fourth and fifth iteration and finally the sorted array is looks like as below:

Value	3	4	5	6	7
index	0	1	2	3	4

A Sorting Problem

(Selection Sort Algorithm)

SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for
i = 0 to n-1

Step 2: CALL SMALLEST(arr, i, n,
pos)

Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]

Step 4: EXIT

SMALLEST (arr, i, n, pos)

Step 1: [INITIALIZE] SET SMALL = arr[i]

Step 2: [INITIALIZE] SET pos = i

Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
 SET SMALL = arr[j]

 SET pos = j

[END OF if]

[END OF LOOP]

Step 4: RETURN pos

Use C programming language to convert the above into a programme

A Sorting Problem (Selection Sort Complexity)

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

$$(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$$

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

i.e., $O(n^2)$

A Sorting Problem

(Selection Sort Complexity)

Output;

$$(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$$

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

i.e., $O(n^2)$

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of $O(n^2)$ for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is $O(n^2)$, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also $O(n^2)$, which occurs when we sort the descending order of an array into the ascending order.



Thank You

Design and Analysis of Algorithm (KCS503)

Insertion Sort and its Analysis

Lecture - 4

A Sorting Problem (Incremental Approach)

Input: A sequence of n numbers a_1, a_2, \dots, a_n .

Output: A permutation (reordering) a_1, a_2, \dots, a_n of the input sequence such that $a_1 \leq a_2 \leq \dots \leq a_n$.

The sequences are typically stored in arrays.

We also refer to the numbers as **keys**. Along with each key may be additional information, known as **satellite data**. (You might want to clarify that .satellite data. does not necessarily come from a satellite!)

We will see several ways to solve the sorting problem. Each way will be expressed as an **algorithm**: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

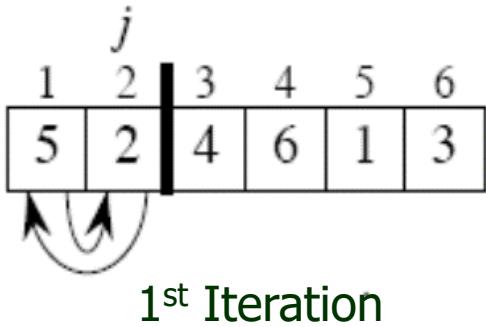
Insertion sort

- A good algorithm for sorting a small number of elements.
- It works the way you might sort a hand of playing cards:
 - Start with an empty left hand and the cards face down on the table.
 - Then remove one card at a time from the table, and insert it into the correct position in the left hand.
 - To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
 - At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

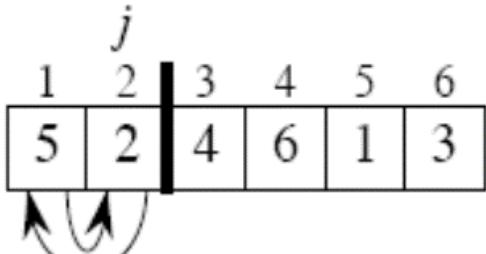
Insertion sort (Example)



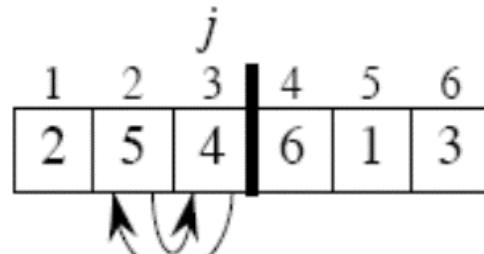
Insertion sort (Example)



Insertion sort (Example)

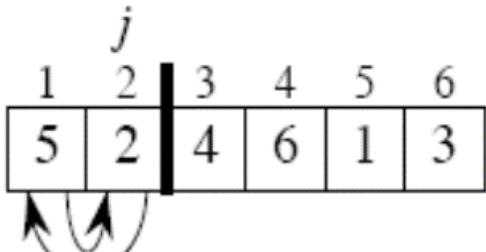


1st Iteration

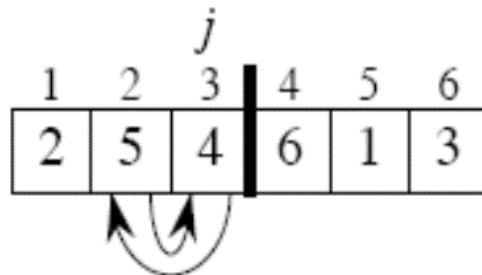


2nd Iteration

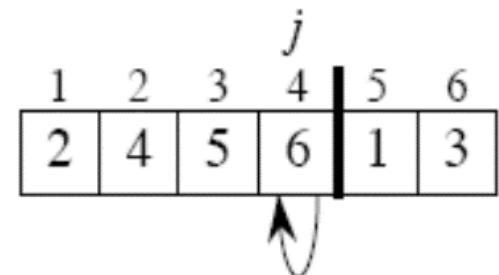
Insertion sort (Example)



1st Iteration

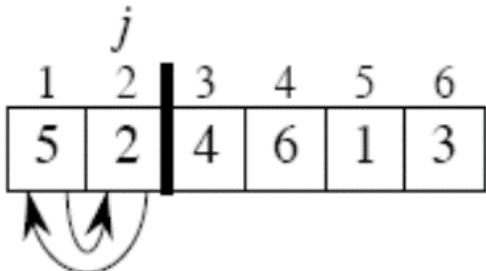


2nd Iteration

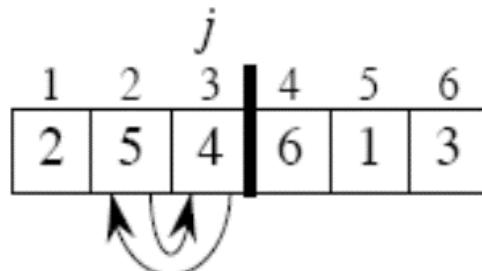


3rd Iteration

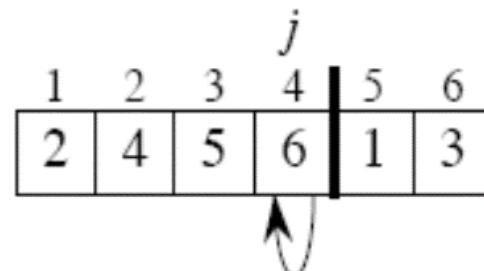
Insertion sort (Example)



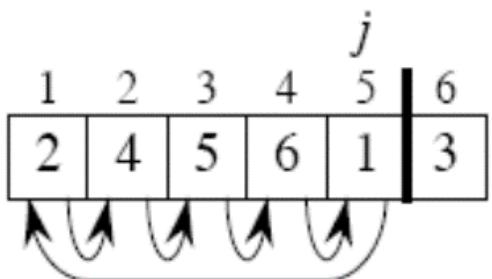
1st Iteration



2nd Iteration

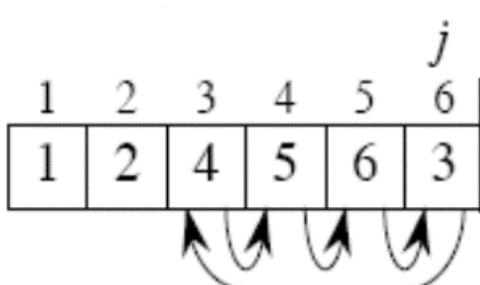
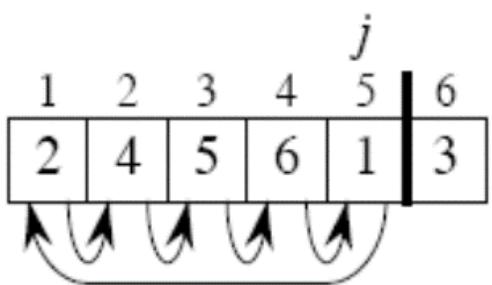
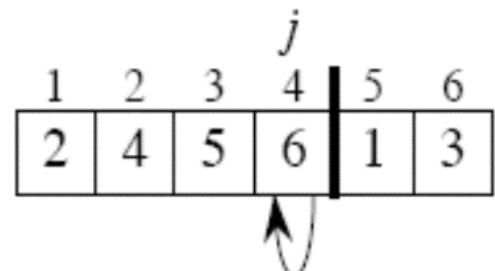
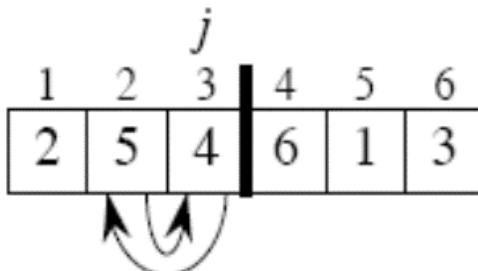
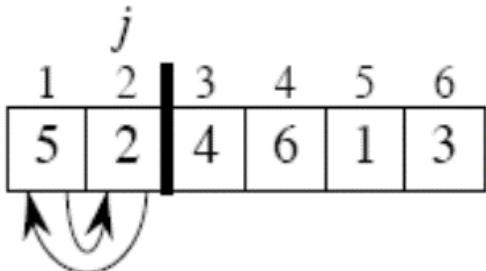


3rd Iteration

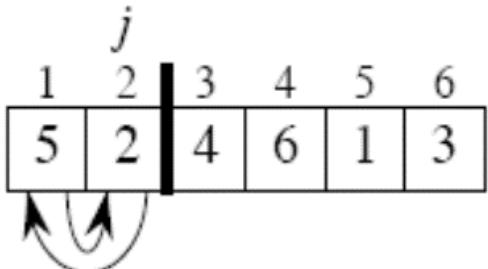


4th Iteration

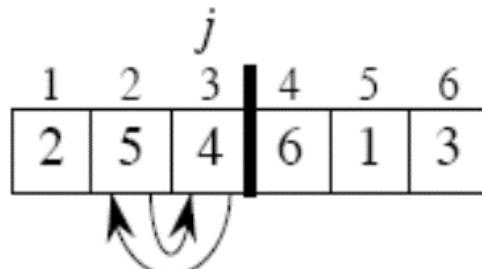
Insertion sort (Example)



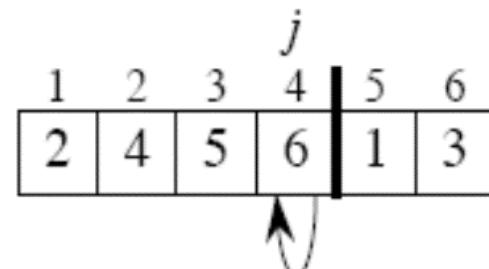
Insertion sort (Example)



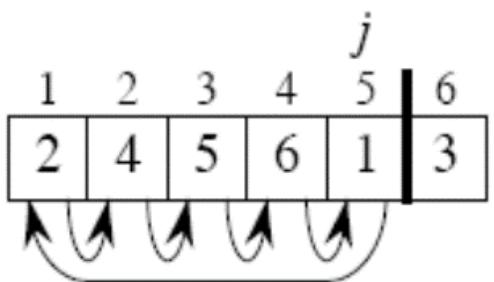
1st Iteration



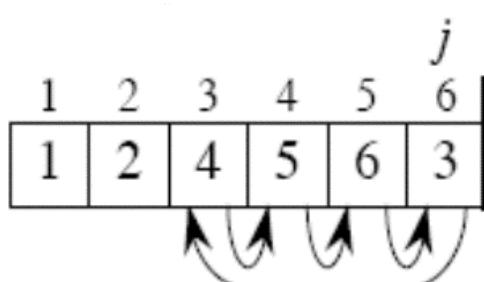
2nd Iteration



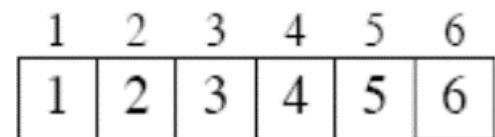
3rd Iteration



4th Iteration



5th Iteration



6th Iteration

Insertion sort (Example)

Insertion sort (Algorithm)

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** n

do $key \leftarrow A[j]$

 ▷ Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

Insertion sort (Algorithm)

INSERTION-SORT(A)

for $j \leftarrow 2$ **to** n

do $key \leftarrow A[j]$

 ▷ Insert $A[j]$ into the sorted sequence $A[1..j - 1]$.

$i \leftarrow j - 1$

while $i > 0$ and $A[i] > key$

do $A[i + 1] \leftarrow A[i]$

$i \leftarrow i - 1$

$A[i + 1] \leftarrow key$

cost times

c_1 n

c_2 $n - 1$

c_3 $n - 1$

c_4 $n - 1$

c_5 $\sum_{j=2}^n t_j$

c_6 $\sum_{j=2}^n (t_j - 1)$

c_7 $\sum_{j=2}^n (t_j - 1)$

c_8 $n - 1$

Correctness Proof of Insertion Sort

- **Initialization:** Just before the first iteration, $j = 2$. The sub array $A[1 \dots j - 1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.
- **Maintenance:** To be precise, we would need to state and prove a loop invariant for the “inner” **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j - 1], A[j - 2], A[j - 3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.
- **Termination:** The outer **for** loop ends when $j > n$; this occurs when $j = n + 1$. Therefore, $j - 1 = n$. Plugging n in for $j - 1$ in the loop invariant, the sub array $A[1 \dots n]$ consists of the elements originally in $A[1 \dots n]$ but in sorted order. In other words, the entire array is sorted!

How do we analyze an algorithm's running time?

- ***Input size:*** Depends on the problem being studied.
 - Usually, the number of items in the input. Like the size n of the array being sorted.
 - But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
 - Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

- ***Running time:*** On a particular input, it is the number of primitive operations (steps) executed.
 - Want to define steps to be machine-independent.
 - Figure that each line of pseudo code requires a constant amount of time.
 - One line may take a different amount of time than another, but each execution of line i takes the same amount of time c_i .
 - This is assuming that the line consists only of primitive operations.
 - If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
 - If the line specifies operations other than primitive ones, then it might take
 - more than constant time.

Analysis of insertion sort

- Assume that the i th line takes time c_i , which is a constant. (Since the third line is a comment, it takes no time.)
- For $j = 2, 3, \dots, n$, let t_j be the number of times that the **while** loop test is executed for that value of j .
- Note that when a **for** or **while** loop exits in the usual way-due to the test in the loop header-the test is executed one time more than the loop body.

Running Time of Insertion Sort

The running time of the algorithm is

$$\sum_{\text{all statements}} (\text{cost of statement}) \cdot (\text{number of times statement is executed}) .$$

Let $T(n)$ = running time of INSERTION-SORT.

$$\begin{aligned} T(n) &= c_1n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) \\ &\quad + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1) . \end{aligned}$$

The running time depends on the values of t_j . These vary according to the input.

Best case: The array is already sorted.

- Always find that $A[i] \leq key$ upon the first time the **while** loop test is run (when $i = j - 1$).
- All t_j are 1.
- Running time is

$$\begin{aligned} T(n) &= c_1n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- Can express $T(n)$ as $an + b$ for constants a and b (that depend on the statement costs c_i) $\Rightarrow T(n)$ is a *linear function* of n .

Worst case: The array is in reverse sorted order.

- Always find that $A[i] > key$ in while loop test.
- Have to compare key with all elements to the left of the j th position \Rightarrow compare with $j - 1$ elements.
- Since the while loop exits because i reaches 0, there's one additional test after the $j - 1$ tests $\Rightarrow t_j = j$.
- $\sum_{j=2}^n t_j = \sum_{j=2}^n j$ and $\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n (j - 1)$.
- $\sum_{j=1}^n j$ is known as an *arithmetic series*, and equation (A.1) shows that it equals $\frac{n(n + 1)}{2}$.
- Since $\sum_{j=2}^n j = \left(\sum_{j=1}^n j\right) - 1$, it equals $\frac{n(n + 1)}{2} - 1$.

- Letting $k = j - 1$, we see that $\sum_{j=2}^n (j - 1) = \sum_{k=1}^{n-1} k = \frac{n(n - 1)}{2}$.
- Running time is

$$\begin{aligned}
 T(n) &= c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \left(\frac{n(n + 1)}{2} - 1 \right) \\
 &\quad + c_6 \left(\frac{n(n - 1)}{2} \right) + c_7 \left(\frac{n(n - 1)}{2} \right) + c_8(n - 1) \\
 &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\
 &\quad - (c_2 + c_4 + c_5 + c_8).
 \end{aligned}$$

- Can express $T(n)$ as $an^2 + bn + c$ for constants a, b, c (that again depend on statement costs) $\Rightarrow T(n)$ is a *quadratic function* of n .

Home Assignment

- Write the algorithm of **Bubble sort** and calculate the time complexity.

Thank U

Design and Analysis of Algorithm

Growth of Functions

Lecture –5,6, and 7

Overview

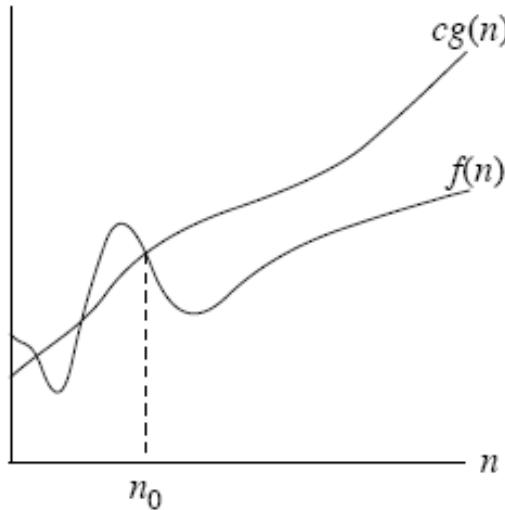
- A way to describe behaviour of functions *in the limit*. We're studying **Asymptotic** efficiency.
- Describe *growth* of functions.(*i.e.* The order of growth of the running time of an algorithm)
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.
- A way to compare "sizes" of functions through different notations (**i.e. Asymptotic Notations**):

O	\approx	\leq
Ω	\approx	\geq
Θ	\approx	$=$
o	\approx	$<$
ω	\approx	$>$

Asymptotic notation (Big Oh)

O -notation

$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$.



- Another view, probably easier to use i.e.
 $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$, where $c > 0$ and $n_0 \geq 1$

$g(n)$ is an *asymptotic upper bound* for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$ (will precisely explain this soon).

Asymptotic notation (Big Oh)

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

$$n^2$$

$$n^2 + n$$

$$n^2 + 1000n$$

$$1000n^2 + 1000n$$

Also,

$$n$$

$$n/1000$$

$$n^{1.99999}$$

$$n^2 / \lg \lg \lg n$$

Asymptotic notation (Big Oh)

Example 1

Prove that $f(n) = 2n + 3 \in O(n)$

Asymptotic notation (Big Oh)

Example 1

Prove that $f(n) = 2n + 3 \in O(n)$

$$\Rightarrow 2n + 3 \leq cg(n)$$

$$\Rightarrow 2n + 3 \leq c(n)$$

$$\Rightarrow 2n + 3 \leq 5n, \quad n \geq 1$$

Hence $f(n) = O(n)$

Asymptotic notation (Big Oh)

Example 1

Prove that $f(n) = 2n + 3 \in O(n)$

$$\Rightarrow 2n + 3 \leq cg(n)$$

$$\Rightarrow 2n + 3 \leq c(n)$$

$$\Rightarrow 2n + 3 \leq 5n, \quad n \geq 1$$

Hence $f(n) = O(n)$

For, $f(n) = O(n^2)$ is also true

For, $f(n) = O(2^n)$ is also true

Asymptotic notation (Big Oh)

Example 1

Prove that $f(n) = 2n + 3 \in O(n)$

$$\Rightarrow 2n + 3 \leq cg(n)$$

$$\Rightarrow 2n + 3 \leq c(n)$$

$$\Rightarrow 2n + 3 \leq 5n, \quad n \geq 1$$

Hence $f(n) = O(n)$

For, $f(n) = O(n^2)$ is also true

For, $f(n) = O(2^n)$ is also true

But for, $f(n) = O(\lg n)$ is not true

Because

$$1 < \lg n < n\sqrt{n} < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

Asymptotic notation (Big Oh)

Example 1 (Second Method)

Prove that $f(n) = 2n + 3 \in O(n)$

Hear $f(n) = 2n + 3$ and $g(n) = n$

So as per the definition of Big Oh

If $f(n) = O(g(n))$, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq C \quad \forall c > 0, n_0 \geq 1$$

Asymptotic notation (Big Oh)

Example 1 (Second Method)

Prove that $f(n) = 2n + 3 \in O(n)$

Hear $f(n) = 2n + 3$ and $g(n) = n$

So as per the definition of Big Oh

If $f(n) = O(g(n))$, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq C \quad \forall c > 0, n_0 \geq 1$$

$$\lim_{n \rightarrow \infty} \frac{2n + 3}{n}$$

$$= \lim_{n \rightarrow \infty} \frac{n(2 + \frac{3}{n})}{n} = \lim_{n \rightarrow \infty} (2 + \frac{3}{n}) = \lim_{n \rightarrow \infty} (2 + \frac{3}{\infty}) = \lim_{n \rightarrow \infty} 2 + 0 = \lim_{n \rightarrow \infty} 2$$

Is a constant, Hence Proved

Asymptotic notation (Big Oh)

Example 2

Prove that $f(n) = 2n^2 + 3n + 4 \in O(n^2)$

Asymptotic notation (Big Oh)

Example 2

Prove that $f(n) = 2n^2 + 3n + 4 \in O(n^2)$

$$\Rightarrow 2n^2 + 3n + 4 \leq 2n^2 + 3n^2 + 4n^2$$

$$\Rightarrow 2n^2 + 3n + 4 \leq 11n^2 , \text{where } c = 11 \text{ and } n \geq 1$$

Hence $f(n) = O(n^2)$

Asymptotic notation (Big Oh)

Example 3

If $f(n) = 2^{n+1}$ and $g(n) = 2^n$ the prove that $f(n) \in O(g(n))$

Asymptotic notation (Big Oh)

Example 3

If $f(n) = 2^{n+1}$ and $g(n) = 2^n$ the prove that $f(n) \in O(g(n))$
 $\Rightarrow 2^{n+1} = 2^n \cdot 2$

So, as per the definition of Big Oh

$$f(n) \leq cg(n)$$

Hence

$$\Rightarrow 2^{n+1} \leq 2^n \cdot 2$$

$$\Rightarrow 2^{n+1} \leq 2 \cdot 2^n \text{ for all } n \geq 1 \text{ and } c > 1$$

Hence, $f(n) \in O(g(n))$

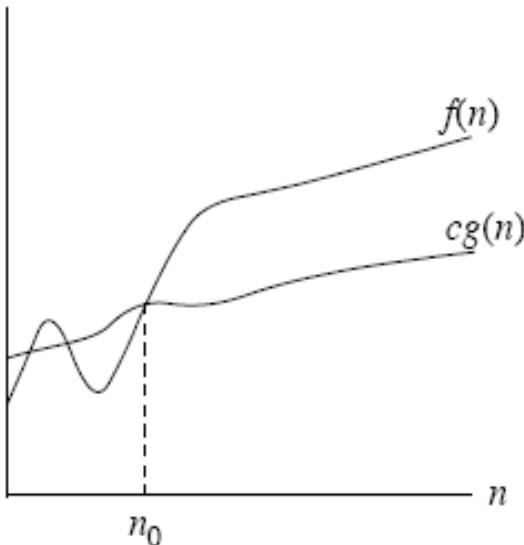
Asymptotic notation (Big Omega)

Ω -notation

$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that}$
 $0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$.

- Another view, probably easier to use i.e.

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \geq c, \text{ where } c > 0 \text{ and } n_0 \geq 1$$



$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Asymptotic notation (Big Omega)

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

$$n^2$$

$$n^2 + n$$

$$n^2 - n$$

$$1000n^2 + 1000n$$

$$1000n^2 - 1000n$$

Also,

$$n^3$$

$$n^{2.00001}$$

$$n^2 \lg \lg \lg n$$

$$2^{2^n}$$

Asymptotic notation (Big Omega)

Example 4

Prove that $f(n) = 2n^2 + 3n + 4 \in \Omega(n^2)$

Asymptotic notation (Big Omega)

Example 4

Prove that $f(n) = 2n^2 + 3n + 4 \in \Omega(n^2)$

$$\Rightarrow 2n^2 + 3n + 4 \geq 1 * n^2$$

Hence $f(n) = \Omega(n^2)$ where $c = 1$ and $n \geq 1$

Asymptotic notation (Big Omega)

Example 5

If $f(n) = 3n + 2$, $g(n) = n^2$ show that $f(n) \notin \Omega(g(n))$

Asymptotic notation (Big Omega)

Example 5

If $f(n) = 3n + 2$, $g(n) = n^2$ show that $f(n) \notin \Omega(g(n))$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{3n + 2}{n^2} > 0$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{n\left(3 + \frac{2}{n}\right)}{n^2} > 0$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{\left(3 + \frac{2}{n}\right)}{n} > 0$$

$\Rightarrow 0 > 0$ is false, Hence $f(n) \notin \Omega(g(n))$

Asymptotic notation (Big Omega)

Example 6

If $f(n) = 2^n + n^2$ and $g(n) = 2^n$ show that $f(n) \in \Omega(g(n))$

Asymptotic notation (Big Omega)

Example 6

If $f(n) = 2^n + n^2$ and $g(n) = 2^n$ show that $f(n) \in \Omega(g(n))$

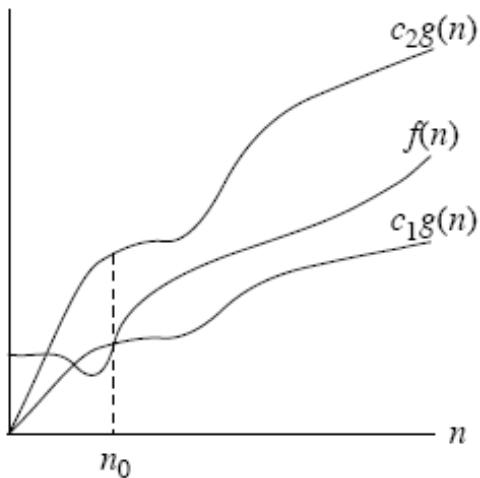
$\Rightarrow 2^n + n^2 > 2^n$ for all $n \geq 1$ and $c = 1$

Hence, $f(n) \in \Omega(g(n))$ is true

Asymptotic notation (Theta)

Θ -notation

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$.



- Another view, probably easier to use i.e.
$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, \text{ where } c > 0 \text{ and } n_0 \geq 1$$

$g(n)$ is an *asymptotically tight bound* for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Asymptotic notation (Theta)

Example 7

Show that $f(n) = 10n^3 + 5n^2 + 17 \in \Theta(n^3)$

Asymptotic notation (Theta)

Example 7

Show that $f(n) = 10n^3 + 5n^2 + 17 \in \Theta(n^3)$

As per the definition of θ notation $C_1g(n) \leq f(n) \leq C_2g(n)$

$$\Rightarrow 10n^3 \leq 10n^3 + 5n^2 + 17 < 10n^3 + 5n^3 + 17n^3$$

$$\Rightarrow 10n^3 \leq 10n^3 + 5n^2 + 17 < 32n^3$$

So, $C_1 = 10$ and $C_2 = 32$ for all $n \geq 1$

Hence, Proved

Asymptotic notation (Theta)

Example 8

Show that $f(n) = (n + a)^b \in \Theta(n^b)$

Asymptotic notation (Theta)

Example 8

Show that $f(n) = (n + a)^b \in \Theta(n^b)$

As per the definition of θ notation $\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ for all $n \geq 1$ and $c > 0$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{(n + a)^b}{n^b}$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{n^b \left(1 + \frac{a}{n}\right)^b}{n^b}$$

$$\Rightarrow \lim_{n \rightarrow \infty} \left(1 + \frac{a}{n}\right)^b \quad \therefore \frac{a}{\infty} = 0$$

$\Rightarrow 1$ which is a constant

Hence, $f(n) = (n + a)^b \in \Theta(n^b)$ is true

Asymptotic notation (Little Oh)

o-notation

$o(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$.

$$n^{1.9999} = o(n^2)$$

$$n^2 / \lg n = o(n^2)$$

$$n^2 \neq o(n^2) \text{ (just like } 2 \neq 2)$$

$$n^2 / 1000 \neq o(n^2)$$

Asymptotic notation (Little Oh)

Example 9

If $f(n) = 2n$, $g(n) = n^2$ Prove that $f(n) = o(g(n))$

Asymptotic notation (Little Oh)

Example 9

If $f(n) = 2n$, $g(n) = n^2$ Prove that $f(n) = o(g(n))$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{2n}{n^2}$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{2}{n}$$

$$\Rightarrow 0$$

Which is True, Hence $f(n) = o(g(n))$

Asymptotic notation (Little Oh)

Example 10

If $f(n) = 2n^2$, $g(n) = n^2$ Prove that $f(n) \neq o(g(n))$

Asymptotic notation (Little Oh)

Example 10

If $f(n) = 2n^2$, $g(n) = n^2$ Prove that $f(n) \neq o(g(n))$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{2n^2}{n^2}$$

$$\Rightarrow \lim_{n \rightarrow \infty} 2$$

$$\Rightarrow 2 \neq 0$$

Which is True, Hence $f(n) \neq o(g(n))$

Asymptotic notation (Little omega)

ω -notation

$\omega(g(n)) = \{f(n) : \text{for all constants } c > 0, \text{there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}.$

Another view, again, probably easier to use: $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$.

$$n^{2.0001} = \omega(n^2)$$

$$n^2 \lg n = \omega(n^2)$$

$$n^2 \neq \omega(n^2)$$

Asymptotic notation (Little omega)

Example 11

If $f(n) = 2n^2 + 16$ and $g(n) = n^2$ show that $f(n) \neq \omega(g(n))$

Asymptotic notation (Little omega)

Example 11

If $f(n) = 2n^2 + 16$ and $g(n) = n^2$ show that $f(n) \neq \omega(g(n))$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{n^2 \left(2 + \frac{16}{n^2}\right)}{n^2}$$

$$\Rightarrow \lim_{n \rightarrow \infty} \left(2 + \frac{16}{n^2}\right)$$

$$\Rightarrow \lim_{n \rightarrow \infty} (2 + 0)$$

$$\Rightarrow \lim_{n \rightarrow \infty} 2$$

So $2 \neq \infty$ is true ,Hence $f(n) \neq \omega(g(n))$

Asymptotic notation (Little Oh omega)

Example 12

If $f(n) = n^2$ and $g(n) = \log n$ show that $f(n) \in \omega(g(n))$

Asymptotic notation (Little Oh omega)

Example 12

If $f(n) = n^2$ and $g(n) = \log n$ show that $f(n) \in \omega(g(n))$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{n^2}{\log n}$$

Apply L Hospital Rule

$$\Rightarrow \lim_{n \rightarrow \infty} \frac{2n}{1/n}$$

$$\Rightarrow \lim_{n \rightarrow \infty} 2n^2$$

$$\Rightarrow \lim_{n \rightarrow \infty} \infty$$

Which is true as per ω – notation, Hence $f(n) \in \omega(g(n))$

Growth of the Functions

- Basically, divided into following Categories:
 1. Decrement Functions
 2. Constant Functions
 3. Logarithmic Functions
 4. Polynomial Functions
 5. Exponential Functions

Growth of the Functions

1. Decrement Functions (Smallest among all categories)

$$f(n) = \frac{10}{n}, \frac{1}{n^2}, \frac{1}{n^3}, \frac{n}{2^n}$$

$$\frac{10}{n} > \frac{1}{n^2} > \frac{1}{n^3} > \frac{n}{2^n}$$

Growth of the Functions

2. Constant Functions (Second Smallest among all categories)

$$f(n) = \text{Constant}$$

$$\frac{10}{n} = 0 \text{ (Constant) [if } n \rightarrow \infty]$$

Growth of the Functions

3. Logarithmic Functions (Third Smallest among all categories)

Example:

$$\log n > \log \log n > \log \log \log n$$

$$(\log n)^k > (\log \log n)^k > (\log \log \log n)^k$$

$$(\log \log \log n) < (\log \log \log n)^k$$

$$(\log \log n) < (\log \log n)^k$$

$$(\log n) < (\log n)^k$$

Combining above three inequalities, we have

$$(\log \log \log n) < (\log \log \log n)^k < (\log \log n) < (\log \log n)^k < (\log n) < (\log n)^k$$

for large value of n

Growth of the Functions

4. Polynomial Functions (Fourth Smallest among all categories) $[n^k : k \text{ is constant}]$

Example:

$$n^1, n^2, n^3, \dots, n^k$$

$$n^1 < n^2 < n^3 < \dots < n^k$$

Growth of the Functions

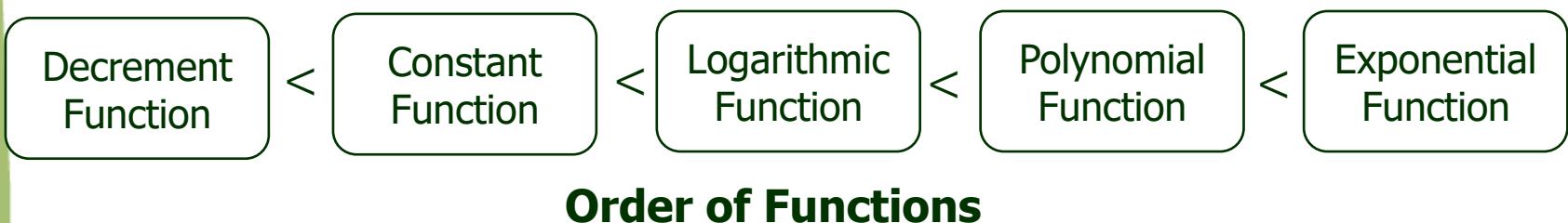
5. Exponential Functions (Fifth Smallest among all categories) $[a^n : \text{if } a > 1 \text{ or } 0 < a < 1]$

Example:

$$2^n, 3^n, 5^n, \dots, 100^n$$

$$2^n < 3^n < 5^n < \dots < 100^n$$

Growth of the Functions



Example:

$$\frac{1}{n} < 1 < \lg n < n\sqrt{n} < n^2 < n^3 < \dots < 2^n < 3^n < \dots < n^n$$

Comparisons of functions

Transitivity:

$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n)).$
Same for O , Ω , o , and ω .

Reflexivity:

$f(n) = \Theta(f(n)).$
Same for O and Ω .

Symmetry:

$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n)).$

Transpose symmetry:

$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n)).$
 $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n)).$

Comparisons:

- $f(n)$ is *asymptotically smaller* than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is *asymptotically larger* than $g(n)$ if $f(n) = \omega(g(n))$.

No trichotomy. Although intuitively, we can liken O to \leq , Ω to \geq , etc., unlike real numbers, where $a < b$, $a = b$, or $a > b$, we might not be able to compare functions.

Example: $n^{1+\sin n}$ and n , since $1 + \sin n$ oscillates between 0 and 2.

Standard notations and common functions

Monotonicity

- $f(n)$ is *monotonically increasing* if $m \leq n \Rightarrow f(m) \leq f(n)$.
- $f(n)$ is *monotonically decreasing* if $m \geq n \Rightarrow f(m) \geq f(n)$.
- $f(n)$ is *strictly increasing* if $m < n \Rightarrow f(m) < f(n)$.
- $f(n)$ is *strictly decreasing* if $m > n \Rightarrow f(m) > f(n)$.

Exponentials

Useful identities:

$$a^{-1} = 1/a ,$$

$$(a^m)^n = a^{mn} ,$$

$$a^m a^n = a^{m+n} .$$

Can relate rates of growth of polynomials and exponentials: for all real constants a and b such that $a > 1$,

$$\lim_{n \rightarrow \infty} \frac{n^b}{a^n} = 0 ,$$

which implies that $n^b = o(a^n)$.

A surprisingly useful inequality: for all real x ,

$$e^x \geq 1 + x .$$

As x gets closer to 0, e^x gets closer to $1 + x$.

Logarithms

Notations:

- $\lg n = \log_2 n$ (binary logarithm) ,
- $\ln n = \log_e n$ (natural logarithm) ,
- $\lg^k n = (\lg n)^k$ (exponentiation) ,
- $\lg \lg n = \lg(\lg n)$ (composition) .

Logarithm functions apply only to the next term in the formula, so that $\lg n + k$ means $(\lg n) + k$, and *not* $\lg(n + k)$.

In the expression $\log_b a$:

- If we hold b constant, then the expression is strictly increasing as a increases.
- If we hold a constant, then the expression is strictly decreasing as b increases.

Useful identities for all real $a > 0$, $b > 0$, $c > 0$, and n , and where logarithm bases are not 1:

$$a = b^{\log_b a},$$

$$\log_c(ab) = \log_c a + \log_c b,$$

$$\log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b},$$

$$\log_b(1/a) = -\log_b a,$$

$$\log_b a = \frac{1}{\log_a b},$$

$$a^{\log_b c} = c^{\log_b a}.$$

Factorials

$n! = 1 \cdot 2 \cdot 3 \cdot n$. Special case: $0! = 1$.

Can use *Stirling's approximation*,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right),$$

to derive that $\lg(n!) = \Theta(n \lg n)$.

Thank U

Design and Analysis of Algorithm (KCS503)

**Implementation and Analysis of
Selection Sort through Tail recursion
Approach**

Lecture -8

Objective

- Able to learn and apply **Tail recursive approach** to develop the recurrence equation $T(n) = (n) + T(n - 1), T(1) = 1$.
- Analyse and solve the recurrence by substitution method.
- Finally, Show that selection sort can be solve recursively by the recursive version of linear search.

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)



After 1st iteration



After 2nd iteration



After 3rd iteration



After 4th iteration



After 5th iteration



A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

```
SELECTION SORT(arr, i, n-1)
    if (i==n-1)
        return
    j=minIndex(arr,i,n-1)
    if I != j
        swap(arr[i],arr[j])
    SELECTION SORT(arr, i+1, n-1)
```

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

SELECTION SORT(arr, i, n-1)

```
if (i==n-1)
    return
j=minIndex(arr,i,n-1)
if I != j
    swap(arr[i],arr[j])
SELECTION SORT(arr, i+1, n-1)
```

minIndex (arr, i, n-1)

```
min_idx=i
min_val=arr[i]
for j=i+1 to n-1
    if min_val>arr[j]
        min_val=arr[j]
        min_idx=j
return min_idx
```

$$T(n)=n+T(n-1)$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

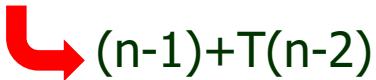
SELECTION SORT(arr, i, n-1)

```
if (i==n-1)
    return
j=minIndex(arr,i,n-1)
if I != j
    swap(arr[i],arr[j])
SELECTION SORT(arr, i+1, n-1)
```

minIndex (arr, i, n-1)

```
min_idx=i
min_val=arr[i]
for j=i+1 to n-1
    if min_val>arr[j]
        min_val=arr[j]
        min_idx=j
return min_idx
```

$$T(n)=n+T(n-1)$$


$$(n-1)+T(n-2)$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

SELECTION SORT(arr, i, n-1)

```
if (i==n-1)
    return
j=minIndex(arr,i,n-1)
if I != j
    swap(arr[i],arr[j])
SELECTION SORT(arr, i+1, n-1)
```

minIndex (arr, i, n-1)

```
min_idx=i
min_val=arr[i]
for j=i+1 to n-1
    if min_val>arr[j]
        min_val=arr[j]
        min_idx=j
return min_idx
```

$$T(n) = n + T(n-1)$$


$$(n-1) + T(n-2)$$


$$(n-2) + T(n-3)$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

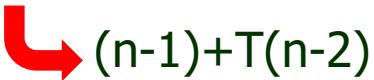
```
SELECTION SORT(arr, i, n-1)
```

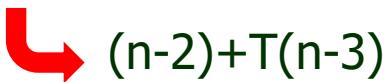
```
if (i==n-1)  
    return  
j=minIndex(arr,i,n-1)  
if I != j  
    swap(arr[i],arr[j])  
SELECTION SORT(arr, i+1, n-1)
```

```
minIndex (arr, i, n-1)
```

```
min_idx=i  
min_val=arr[i]  
for j=i+1 to n-1  
    if min_val>arr[j]  
        min_val=arr[j]  
        min_idx=j  
return min_idx
```

$$T(n)=n+T(n-1)$$


$$(n-1)+T(n-2)$$


$$(n-2)+T(n-3)$$

▪ ▪ ▪ ▪ ▪

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

SELECTION SORT(arr, i, n-1)

```
if (i==n-1)
    return
j=minIndex(arr,i,n-1)
if I != j
    swap(arr[i],arr[j])
SELECTION SORT(arr, i+1, n-1)
```

minIndex (arr, i, n-1)

```
min_idx=i
min_val=arr[i]
for j=i+1 to n-1
    if min_val>arr[j]
        min_val=arr[j]
        min_idx=j
return min_idx
```

$$T(n)=n+T(n-1)$$


$$(n-1)+T(n-2)$$


$$(n-2)+T(n-3)$$

• • • •


$$2+T(1)$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

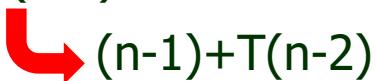
```
SELECTION SORT(arr, i, n-1)
```

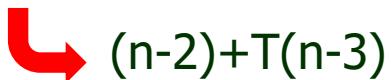
```
if (i==n-1)  
    return  
j=minIndex(arr,i,n-1)  
if I != j  
    swap(arr[i],arr[j])  
SELECTION SORT(arr, i+1, n-1)
```

```
minIndex (arr, i, n-1)
```

```
min_idx=i  
min_val=arr[i]  
for j=i+1 to n-1  
    if min_val>arr[j]  
        min_val=arr[j]  
        min_idx=j  
return min_idx
```

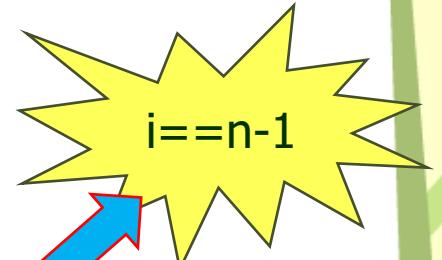
$$T(n)=n+T(n-1)$$


$$(n-1)+T(n-2)$$


$$(n-2)+T(n-3)$$

• • • •


$$2+T(1)$$


$$i==n-1$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

$$T(n) = n + T(n-1)$$

$$\quad \quad \quad \downarrow \quad \quad \quad (n-1) + T(n-2)$$

$$\quad \quad \quad \downarrow \quad \quad \quad (n-2) + T(n-3)$$

■ ■ ■ ■

$$\quad \quad \quad \downarrow \quad \quad \quad 2 + T(1)$$

$i == n-1$

$$\begin{aligned} T(n) &= n + (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1 \\ &= \frac{n(n + 1)}{2} \\ &= O(n^2) \end{aligned}$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

$$T(n) = n + T(n-1)$$

$$\quad \quad \quad \downarrow \quad \quad \quad (n-1) + T(n-2)$$

$$\quad \quad \quad \downarrow \quad \quad \quad (n-2) + T(n-3)$$

■ ■ ■ ■

$$\quad \quad \quad \downarrow \quad \quad \quad 2 + T(1)$$

$i == n-1$

$$\begin{aligned} T(n) &= n + (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1 \\ &= \frac{n(n + 1)}{2} \\ &= O(n^2) \end{aligned}$$

Hence, the recursive (Tail) method have develop the recurrence equation

$$T(n) = O(n) + T(n - 1) \in O(n^2)$$

A Sorting Problem

(Selection Sort Algorithm with Tail Recursive Approach)

SELECTION SORT(arr, i, n-1)

```
if (i==n-1)
    return
j=minIndex(arr,i,n-1)
if I != j
    swap(arr[i],arr[j])
SELECTION SORT(arr, i+1, n-1)
```

minIndex (arr, i, n-1)

```
min_indx=i
min_val=arr[i]
for j=i+1 to n-1
    if min_val>arr[j]
        min_val=arr[j]
        min_indx=j
return min_indx
```

Hence, the recursive (Tail) method have develop the recurrence equation

$$T(n) = O(n) + T(n - 1) \in O(n^2)$$



Thank You

Design and Analysis of Algorithm (KCS503)

**Implementation and Analysis of
Insertion Sort through Head recursion
Approach**

Lecture -9

Objective

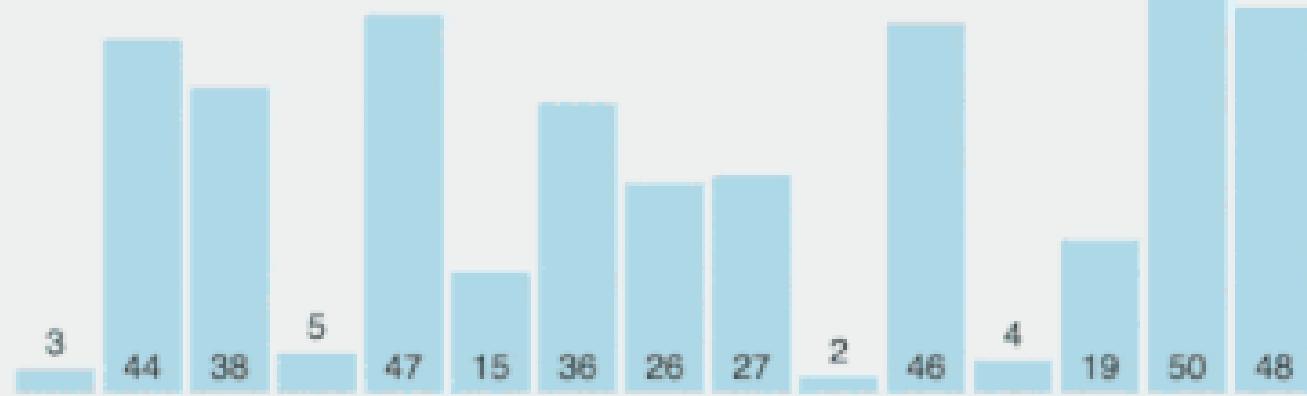
- Able to learn and apply the head recursive approach

$$(T(n) = T(n - 1) + O(n), T(1) = 1)$$

of Insertion sort with analysis and analyse its complexity.

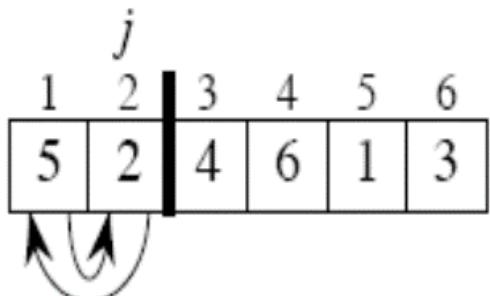
A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

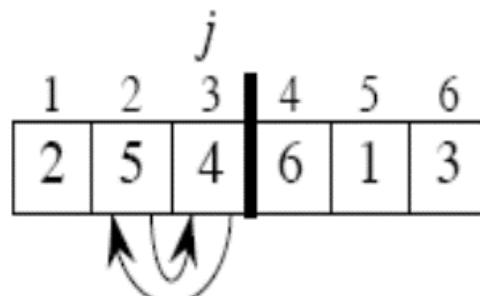


A Sorting Problem

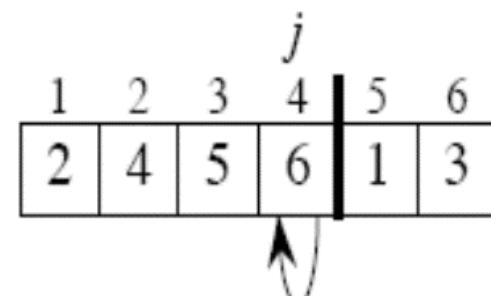
(Insertion Sort Algorithm with Head Recursive Approach)



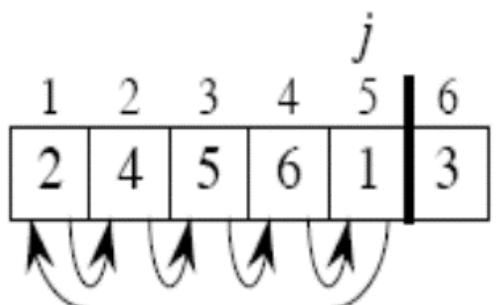
1st Iteration



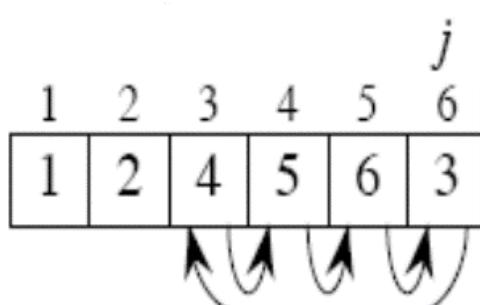
2nd Iteration



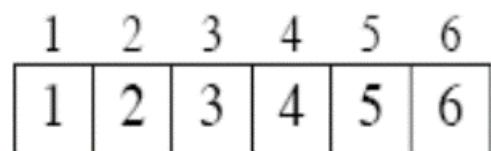
3rd Iteration



4th Iteration



5th Iteration



6th Iteration

A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

$A = [5, 2, 4, 6, 1, 3]$, size = 6

insertion-sort($A, 0$)

insert 5

return $A = [5]$

insertion-sort($A, 3$)

insert 6

return $A = [2, 4, 5, 6]$

insertion-sort($A, 1$)

insert 2

return $A = [2, 5]$

insertion-sort($A, 4$)

insert 1

return $A = [1, 2, 4, 5, 6]$

insertion-sort($A, 2$)

insert 4

return $A = [2, 4, 5]$

insertion-sort($A, 5$)

insert 3

return $A = [1, 2, 3, 4, 5, 6]$

A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

It was observed that the concept is:

Insert an element in a previously sorted array named as 'A'.

Solution Steps:

- Base Case: If array size is 1 or smaller, return.
- Recursively sort first $n-1$ elements.
- Insert the last element at its correct position in the sorted array.

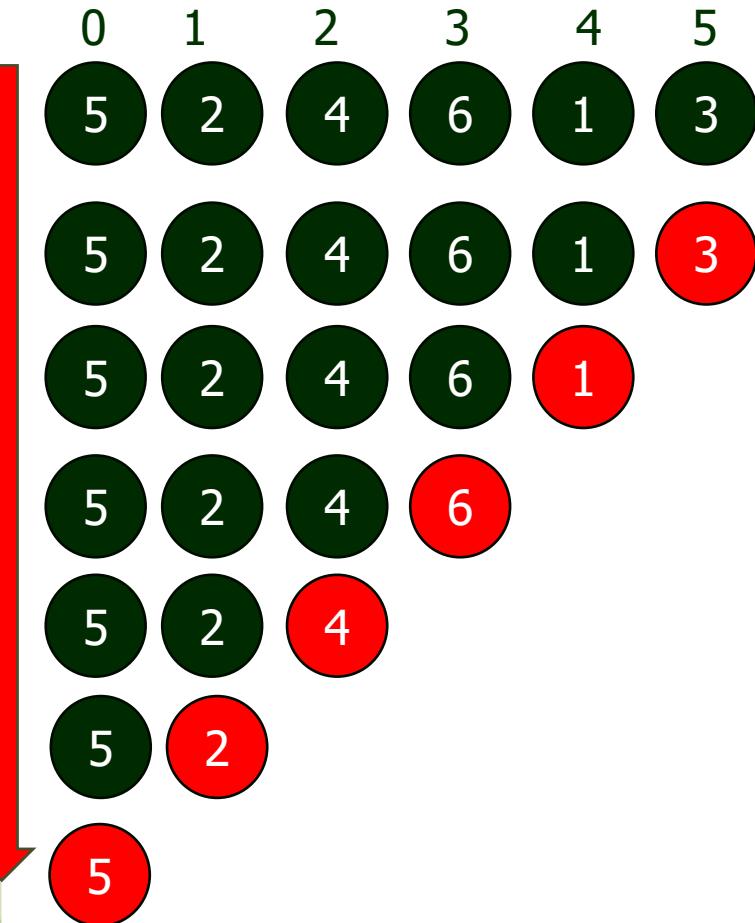
A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

```
void insertion_sort(A, j) {  
    //Initially j=length(A)  
    // Base case  
    if (j <= 1)  
        return  
    // Sort first i-1 elements  
    insertion_sort( A, j-1 )  
    insert A[j-1] into A[0...j-2]  
}
```

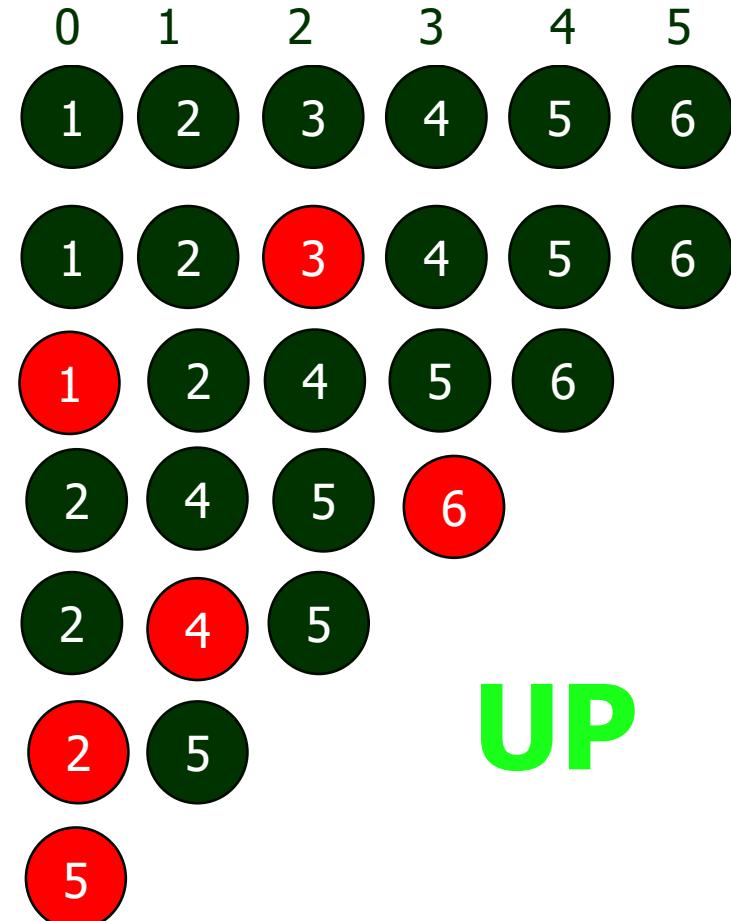
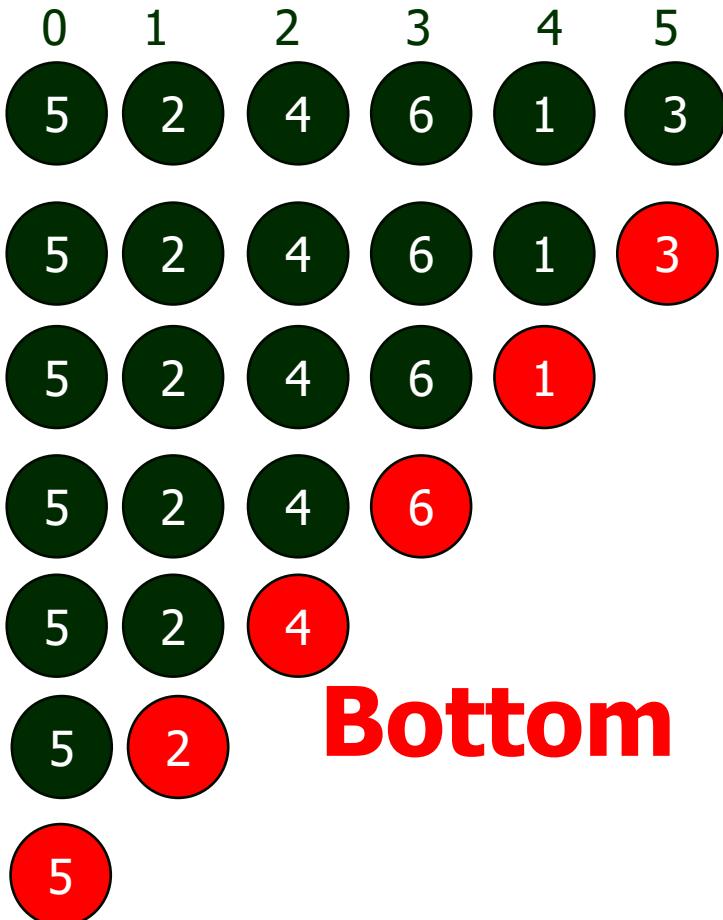
A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)



A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)



A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

```
void insertion_sort(A, j) {  
    //Initially j=length(A)  
    // Base case  
    if (j <= 1)  
        return  
    // Sort first n-1 elements  
    insertion_sort( A, j-1 )  
    val = A[j-1]  
    i = j-2  
    while (i >= 0 && A[i] > val) {  
        A[i+1] = A[i]  
        i = i - 1  
    }  
    A[i+1] = val  
}
```

A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

Complexity:

$$T(n) = T(n - 1) + (n)$$

$$T(n) = T(n - 2) + (n - 1) + (n)$$

$$T(n) = T(n - 3) + (n - 2) + (n - 1) + (n)$$

.....

.....

.....

$$T(n) = 1 + 2 + 3 + \dots + (n - 3) + (n - 2) + (n - 1) + (n)$$

Hence $T(n) = \frac{n(n + 1)}{2}$ \rightarrow $T(n) = O(n^2)$

A Sorting Problem

(Insertion Sort Algorithm with Head Recursive Approach)

```
void insertion_sort(A, j) {  
    //Initially j=length(A)  
    // Base case  
    if (j <= 1)  
        return  
    // Sort first j-1 elements  
    insertion_sort( A, j-1 )  
    val = A[j-1]  
    i = j-2  
    while (i >= 0 && A[i] > val) {  
        A[i+1] = A[i]  
        i = i - 1  
    }  
    A[i+1] = val  
}
```

Hence, the recursive (Head) method have develop the recurrence equation

$$T(n) = T(n - 1) + O(n) \in O(n^2)$$



Thank You



Design and Analysis of Algorithm

Recurrence Equation **(Solving Recurrence using Iteration Methods)**

Lecture – 10 and 11

Overview

- A **recurrence** is a function is defined in terms of
 - one or more base cases, and
 - itself, with smaller arguments.

Examples:

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n - 1) + 1 & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = n$.

Linear Decay

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

Solution: $T(n) = n \lg n + n$.

Division

$$\bullet \quad T(n) = \begin{cases} 0 & \text{if } n = 2 , \\ T(\sqrt{n}) + 1 & \text{if } n > 2 . \end{cases}$$

Solution: $T(n) = \lg \lg n$.

Changing Variable

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = \Theta(n \lg n)$.

Decision Tree

Overview

- Many technical issues:
 - Floors and ceilings

[Floors and ceilings can easily be removed and don't affect the solution to the recurrence. They are better left to a discrete math course.]
 - Exact vs. asymptotic functions
 - Boundary conditions

Overview

In algorithm analysis, the recurrence and it's solution are expressed by the help of asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \lg n)$.
 - The boundary conditions are usually expressed as $T(n) = O(1)$ for sufficiently small n .
 - But when there is a desire of an exact, rather than an asymptotic, solution, the need is to deal with boundary conditions.
 - In practice, just use asymptotics most of the time, and ignore boundary conditions.

Recursive Function

- Example

$A(n)$

{

If($n > 1$)

Return($A(n - 1)$)

}

The relation is called recurrence relation

The Recurrence relation of given function is written as follows.

$$T(n) = T(n - 1) + 1$$

Recursive Function

- To solve the Recurrence relation the following methods are used:

1. Iteration method

2. Recursion-Tree method
3. Master Method
4. Substitution Method

Iteration Method(Example 1)

- In Iteration method the basic idea is to expand the recurrence and express it as a summation of terms dependent only on 'n' (i.e. the number of input) and the initial conditions.

Example 1:

Solve the following recurrence relation by using Iteration method.

$$T(n) = \begin{cases} T(n - 1) + 1 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Iteration Method (Example 1)

It means $T(n) = T(n - 1) + 1$ if $n > 1$

and $T(n) = 1$ when $n = 1$ ----- (1)

Put $n = n - 1$ in equation 1, we get

$$T(n - 1) = T(n - 2) + 1$$

Put the value of $T(n - 1)$ in equation 1, we get

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

Iteration Method (Example 1)

Put $n = n - 2$ in equation 1, we get

$$T(n - 2) = T(n - 3) + 1$$

Put the value of $T(n - 2)$ in equation 2, we get

$$T(n) = T(n - 3) + 3 \text{ -----} -(3)$$

.....

$$T(n) = T(n - k) + k \text{ -----} -(k)$$

Iteration Method (Example 1)

Let $T(n - k) = T(1) = 1$

(As per the base condition of recurrence)

So $n - k = 1$

$\Rightarrow k = n - 1$

Now put the value of k in equation k

$$T(n) = T(n - (n - 1)) + n - 1$$

$$T(n) = T(1) + n - 1$$

$$T(n) = 1 + n - 1 \quad [\because T(1) = 1]$$

$$T(n) = n$$

$$\therefore T(n) = \Theta(n)$$

Iteration Method (Example 2)

Example 2:

Solve the following recurrence relation by using Iteration method.

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + 3n^2 & \text{if } n > 1 \\ 11 & \text{if } n = 1 \end{cases}$$

Iteration Method (Example 2)

It means $T(n) = 2T\left(\frac{n}{2}\right) + 3n^2$ if $n > 1$ and $T(n) = 11$ when $n = 1$ --(1)

Put $n = \frac{n}{2}$ in equation 1, we get

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 3\left(\frac{n}{2}\right)^2$$

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

Put the value of $T\left(\frac{n}{2}\right)$ in equation 1, we get

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

$$T(n) = 2^2T\left(\frac{n}{2^2}\right) + 2 \cdot 3 \cdot \frac{n^2}{4} + 3n^2$$

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

Iteration Method (Example 2)

Put $n = \frac{n}{4}$ in equation 1, we get

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 3\left(\frac{n}{4}\right)^2$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{2^3}\right) + 3\left(\frac{n}{4}\right)^2$$

Put the value of $T\left(\frac{n}{4}\right)$ in equation 2, we get

$$T(n) = 2^2 \left[2T\left(\frac{n}{8}\right) + 3\frac{n^2}{16} \right] + 3\frac{n^2}{2} + 3n^2$$

$$T(n) = 2^2 \left[2T\left(\frac{n}{2^3}\right) + 3\left(\frac{n}{4}\right)^2 \right] + 3\frac{n^2}{2} + 3n^2$$

$$T(n) = 2^3 T\left(\frac{n}{2^3}\right) + 4 \cdot 3 \frac{n^2}{16} + 3\frac{n^2}{2} + 3n^2$$

Iteration Method (Example 2)

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

$$T(n) = 2^i T\left(\frac{n}{2^i}\right) + \dots + \dots + \dots + 3 \frac{n^2}{2^2} + 3 \frac{n^2}{2} + 3n^2 \quad \dots \dots \quad (i^{th} \text{ term})$$

and the series terminate when $\frac{n}{2^i} = 1$

$$\Rightarrow n = 2^i$$

Taking log both side

$$\Rightarrow \log_2 n = i \log_2 2$$

$$\Rightarrow i = \log_2 n \quad (\text{because } \log_2 2 = 1)$$

Iteration Method (Example 2)

Hence we can write the i^{th} term as follows

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

$$\Rightarrow T(n) = 3n^2 + 3\frac{n^2}{2} + 3\frac{n^2}{2^2} + \dots + \dots + \dots + 2^{\log_2 n} T(1)$$

$$\Rightarrow T(n) = 3n^2 + 3\frac{n^2}{2} + 3\frac{n^2}{2^2} + \dots + \dots + \dots + 2^{\log_2 n} \cdot 11$$

$$\Rightarrow T(n) = 3n^2 + 3\frac{n^2}{2} + 3\frac{n^2}{2^2} + \dots + \dots + \dots + n^{\log_2 2} \cdot 11 \quad [\text{As } \log_2 2 = 1]$$

$$\Rightarrow T(n) = 3n^2 + 3\frac{n^2}{2} + 3\frac{n^2}{2^2} + \dots + \dots + \dots + n \cdot 11$$

$$\Rightarrow T(n) = \left[3n^2 + 3\frac{n^2}{2} + 3\frac{n^2}{2^2} + \dots + \dots + \dots \right] + 11 \cdot n$$

$$\Rightarrow T(n) = 3n^2 \left[1 + \frac{1}{2} + \frac{1}{2^2} + \dots + \dots + \dots \right] + 11 \cdot n$$

Iteration Method (Example 2)

As we know that Sum of infinite Geometric series is

$$= a + ar + ar^2 + \dots + ar^{(n-1)} = \sum_{i=0}^{\infty} ar^i = a \left(\frac{1}{1-r} \right) = \frac{a}{1-r}$$

Hence,

$$\Rightarrow T(n) \leq 3n^2 \left[\frac{1}{1 - \frac{1}{2}} \right] + 11 n$$

$$\Rightarrow T(n) \leq 3n^2 \cdot 2 + 11 n$$

$$\Rightarrow T(n) \leq 6n^2 + 11 n$$

Hence $T(n) = O(n^2)$

Iteration Method (Example 3)

Example 3:

Solve the following recurrence relation by using Iteration method.

$$T(n) = \begin{cases} 8T\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Iteration Method (Example 3)

It means $T(n) = 8T\left(\frac{n}{2}\right) + n^2$ if $n > 1$ and $T(n) = 1$ when $n = 1$ ----- (1)

Put $n = \frac{n}{2}$ in equation 1, we get

$$T\left(\frac{n}{2}\right) = 8T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2$$

Put the value of $T\left(\frac{n}{2}\right)$ in equation 1, we get

$$T(n) = 8 \left[8T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \right] + n^2$$

$$T(n) = 8^2 T\left(\frac{n}{4}\right) + 8 \frac{n^2}{4} + n^2 ----- (2)$$

Put $n = \frac{n}{4}$ in equation 1, we get

$$T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2$$

Iteration Method (Example 3)

Put the value of $T\left(\frac{n}{4}\right)$ in equation 2, we get

$$T(n) = 8^2 \left[8T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 \right] + 8\frac{n^2}{4} + n^2$$

$$T(n) = 8^3 T\left(\frac{n}{8}\right) + 8^2 \frac{n^2}{4^2} + 8\frac{n^2}{4} + n^2 \quad \dots \dots \quad -(3)$$

$$T(n)$$

$$= 8^k T\left(\frac{n}{2^k}\right) + 8^{k-1} \frac{n^2}{4^{k-1}} + \dots + \dots + \dots + 8^2 \frac{n^2}{4^2} + 8\frac{n^2}{4} + n^2 \quad \dots \quad (k^{\text{th}} \text{ term})$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2 \left[\frac{8^{k-1}}{4^{k-1}} + \frac{8^{k-2}}{4^{k-2}} \dots + \dots + \dots + \frac{8^2}{4^2} + \frac{8}{4} + 1 \right]$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2 [2^{k-1} + 2^{k-2} \dots + \dots + \dots + 2^2 + 2 + 1] \quad \dots \quad -(4)$$

Iteration Method (Example 3)

Put the value of $T\left(\frac{n}{4}\right)$ in equation 2, we get

$$T(n) = 8^2 \left[8T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 \right] + 8\frac{n^2}{4} + n^2$$

$$T(n) = 8^3 T\left(\frac{n}{8}\right) + 8^2 \frac{n^2}{4^2} + 8\frac{n^2}{4} + n^2 \quad \dots \dots \quad -(3)$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + 8^{k-1} \frac{n^2}{4^{k-1}} + \dots + \dots + 8^2 \frac{n^2}{4^2} + 8\frac{n^2}{4} + n^2 \quad \dots \quad (k^{\text{th}} \text{ term})$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2 \left[\frac{8^{k-1}}{4^{k-1}} + \frac{8^{k-2}}{4^{k-2}} \dots + \dots + \dots + \frac{8^2}{4^2} + \frac{8}{4} + 1 \right]$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2 [2^{k-1} + 2^{k-2} \dots + \dots + \dots + 2^2 + 2 + 1] \quad \dots \quad -(4)$$

Iteration Method (Example 3)

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2[2^{k-1} + 2^{k-2} \dots + \dots + 2^2 + 2 + 1] \quad \text{--- (4)}$$

$$T(n) = 8^k T\left(\frac{n}{2^k}\right) + n^2[1 + 2 + 2^2 + \dots + \dots + 2^{k-2} + 2^{k-1}] \quad \text{--- (5)}$$

and the series terminate when $\frac{n}{2^k} = 1$

$$\Rightarrow n = 2^k$$

Taking log both side

$$\Rightarrow \log_2 n = k \log_2 2$$

$$\Rightarrow k = \log_2 n \quad (\text{because } \log_2 2 = 1)$$

Now, apply the value of $k = \log_2 n$ and $\frac{n}{2^k} = 1$ in equation 5

Iteration Method (Example 3)

$$T(n) = 8^{\log_2 n} T(1) + n^2 [1 + 2 + 2^2 + \dots + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}] - \quad (6)$$

$$= n^{\log_2 8} \cdot 1 + n^2 [1 + 2 + 2^2 + \dots + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}]$$

$$= n^3 + n^2 [1 + 2 + 2^2 + \dots + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}]$$

Is a G.P Series, but in this case no need of evaluation. Because the highest order polynomial is n^3 . So no need to calculate n^2 .

$$= n^3$$

Hence the complexity is $T(n) = n^3$

Iteration Method (Example 3)

$$\begin{aligned}T(n) &= 8^{\log_2 n} T(1) + n^2 [1 + 2 + 2^2 + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}] - (6) \\&= n^{\log_2 8} \cdot 1 + n^2 [1 + 2 + 2^2 + \dots + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}] \\&= n^3 + n^2 [1 + 2 + 2^2 + \dots + \dots + 2^{\log_2 n-2} + 2^{\log_2 n-1}]\end{aligned}$$

Is a G.P Series, but in this case no need of evaluation. Because the highest order polynomial is n^3 . So no need to calculate n^2 .

$$= n^3$$

Hence the complexity is $T(n) = O(n^3)$

Sum of finite Geometric Progression series is

$$= a + ar + ar^2 + \dots + ar^n = \sum_{i=0}^n ar^i = a \left(\frac{r^{n+1}-1}{r-1} \right)$$

Iteration Method (Example 4)

Example 4:

Solve the following recurrence relation by using Iteration method.

$$T(n) = \begin{cases} 7T\left(\frac{n}{2}\right) + n^2 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

(i.e. Strassen Algorithm)

Iteration Method (Example 4)

It means $T(n) = 7T\left(\frac{n}{2}\right) + n^2$ if $n > 1$ and $T(n) = 1$ when $n = 1$ ----- (1)

Put $n = \frac{n}{2}$ in equation 1, we get

$$T\left(\frac{n}{2}\right) = 7T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2$$

Put the value of $T\left(\frac{n}{2}\right)$ in equation 1, we get

$$T(n) = 7 \left[7T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2 \right] + n^2$$

$$T(n) = 7^2 T\left(\frac{n}{4}\right) + 7 \frac{n^2}{4} + n^2 ----- (2)$$

Put $n = \frac{n}{4}$ in equation 1, we get

$$T\left(\frac{n}{4}\right) = 7T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2$$

Iteration Method (Example 4)

Put the value of $T\left(\frac{n}{4}\right)$ in equation 2, we get

$$T(n) = 7^2 \left[7T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 \right] + 7 \frac{n^2}{4} + n^2$$

$$T(n) = 7^3 T\left(\frac{n}{8}\right) + 7^2 \frac{n^2}{4^2} + 7 \frac{n^2}{4} + n^2 \quad \dots \dots \quad (3)$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + 7^{k-1} \frac{n^2}{4^{k-1}} + \dots + \dots + \dots + 7^2 \frac{n^2}{4^2} + 7 \frac{n^2}{4} + n^2 \quad \dots \quad (k^{\text{th}} \text{ term})$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + n^2 \left[\frac{7^{k-1}}{4^{k-1}} + \frac{7^{k-2}}{4^{k-2}} \dots + \dots + \dots + \frac{7^2}{4^2} + \frac{7}{4} + 1 \right]$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + n^2 \left[\sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \right] \quad \dots \dots \quad (4)$$

Iteration Method (Example 4)

Put the value of $T\left(\frac{n}{4}\right)$ in equation 2, we get

$$T(n) = 7^2 \left[7T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2 \right] + 7 \frac{n^2}{4} + n^2$$

$$T(n) = 7^3 T\left(\frac{n}{8}\right) + 7^2 \frac{n^2}{4^2} + 7 \frac{n^2}{4} + n^2 \quad \dots \dots \quad (3)$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + 7^{k-1} \frac{n^2}{4^{k-1}} + \dots + \dots + \dots + 7^2 \frac{n^2}{4^2} + 7 \frac{n^2}{4} + n^2 \quad \dots \quad (k^{th} \text{ term})$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + n^2 \left[\frac{7^{k-1}}{4^{k-1}} + \frac{7^{k-2}}{4^{k-2}} \dots + \dots + \dots + \frac{7^2}{4^2} + \frac{7}{4} + 1 \right]$$

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + n^2 \left[\sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \right] \quad \dots \dots \quad (4)$$

Iteration Method (Example 4)

$$T(n) = 7^k T\left(\frac{n}{2^k}\right) + n^2 \left[\sum_{i=0}^{k-1} \left(\frac{7}{4}\right)^i \right] \quad \text{---(4)}$$

and the series terminate when $\frac{n}{2^k} = 1$

$$\Rightarrow n = 2^k$$

Taking log both side

$$\Rightarrow \log_2 n = k \log_2 2$$

$$\Rightarrow k = \log_2 n \quad (\text{because } \log_2 2 = 1)$$

Now, apply the value of $k = \log_2 n$ and $\frac{n}{2^k} = 1$ in equation 4

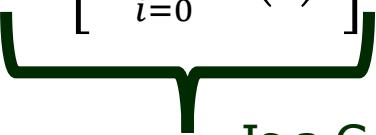
Iteration Method (Example 4)

$$T(n) = 7^{\log_2 n} T(1) + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4}\right)^i \right] \quad \text{---(5)}$$

$$= n^{\log_2 7} \cdot 1 + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4}\right)^i \right]$$

$$= n^{\log_2 7} \cdot 1 + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4}\right)^i \right]$$

$$= n^{2.8} + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4}\right)^i \right]$$



Is a G.P Series, but in this case no need of evaluation. Because the highest order polynomial is n^3 . So no need to calculate n^2 .

Iteration Method (Example 4)

$$= n^{2.8} + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4} \right)^i \right]$$

Is a G.P Series, but in this case no need of evaluation. Because the highest order polynomial is $n^{2.8}$. So no need to calculate n^2 .

$$= n^{2.8}$$

Hence the complexity is $T(n) = n^{2.8}$

Iteration Method (Example 4)

$$= n^{2.8} + n^2 \left[\sum_{i=0}^{\log_2 n - 1} \left(\frac{7}{4} \right)^i \right]$$

Is a G.P Series, but in this case no need of evaluation. Because the highest order polynomial is $n^{2.8}$. So no need to calculate n^2 .

$$= n^{2.8}$$

Hence the complexity is $T(n) = O(n^{2.8})$

Sum of finite Geometric Progression series is

$$= a + ar + ar^2 + \dots + ar^n = \sum_{i=0}^n ar^i = a \left(\frac{r^{n+1} - 1}{r - 1} \right)$$

Iteration Method (Example 5)

Example 5:

Solve the following recurrence relation by using Iteration method.

$$T(n) = \begin{cases} T(n - 1) + \log n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Iteration Method (Example 5)

It means $T(n) = T(n - 1) + \log n$ if $n > 1$ and $T(n) = 1$ when $n = 1$ ---(1)

Put $n = n - 1$ in equation 1, we get

$$T(n - 1) = T(n - 2) + \log(n - 1)$$

Put the value of $T(n - 1)$ in equation 1, we get

$$\begin{aligned} T(n) &= T(n - 2) + \log(n - 1) + \log n \\ &= T(n - 3) + \log(n - 2) + \log(n - 1) + \log n \\ &= T(n - 4) + \log(n - 3) + \log(n - 2) + \log(n - 1) + \log n \end{aligned}$$

.....

.....

Hence the k^{th} order is :

$$T(n) = T(n - k) + \log(n - (k - 1)) + \dots + \log(n - 2) + \log(n - 1) + \log n$$

$$T(n) = T(n - k) + \log(n - k + 1) + \dots + \log(n - 2) + \log(n - 1) + \log n$$

Iteration Method (Example 5)

Hence the k^{th} order is :

$$T(n) = T(n - k) + \log(n - k + 1) + \dots + \log(n - 2) + \log(n - 1) + \log n$$

As per the assumption $n - k = 1$

So $k = n - 1$

The k^{th} order can be written as:

$$\begin{aligned} T(n) &= T(1) + \log(n - n + 1 + 1) + \dots + \log(n - 2) + \log(n - 1) + \log n \\ &= 1 + \log(2) + \dots + \log(n - 2) + \log(n - 1) + \log n \\ &= 1 + \log(2) + \log(3) + \log(4) \dots + \log(n - 2) + \log(n - 1) + \log n \\ &= 1 + \log(2.3.4.5 \dots \dots \dots .n) \\ &= 1 + \log(n!) \end{aligned}$$

Hence the complexity is : $O(\log n!)$

Iteration Method (Practice)

$$Q1. T(n) = \begin{cases} T\left(\frac{7n}{10}\right) + n & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$$Q2. T(n) = \begin{cases} T(n - 1) + (n - 1) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$$Q3. T(n) = \begin{cases} T(n - 1) + n^2 & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Thank U

Design and Analysis of Algorithm

Recurrence Equation (Solving Recurrence using Recursion Tree Methods)

Lecture – 12 and 13

Overview

- A **recurrence** is a function is defined in terms of
 - one or more base cases, and
 - itself, with smaller arguments.

Examples:

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n - 1) + 1 & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = n$.

Linear Decay

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

Solution: $T(n) = n \lg n + n$.

Division

$$\bullet \quad T(n) = \begin{cases} 0 & \text{if } n = 2 , \\ T(\sqrt{n}) + 1 & \text{if } n > 2 . \end{cases}$$

Solution: $T(n) = \lg \lg n$.

Changing Variable

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = \Theta(n \lg n)$.

Decision Tree

Overview

- Many technical issues:
 - Floors and ceilings

[Floors and ceilings can easily be removed and don't affect the solution to the recurrence. They are better left to a discrete math course.]
 - Exact vs. asymptotic functions
 - Boundary conditions

Overview

In algorithm analysis, the recurrence and it's solution are expressed by the help of asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \lg n)$.
 - The boundary conditions are usually expressed as $T(n) = O(1)$ for sufficiently small n .
 - But when there is a desire of an exact, rather than an asymptotic, solution, the need is to deal with boundary conditions.
 - In practice, just use asymptotics most of the time, and ignore boundary conditions.

Recursive Function

- Example

$A(n)$

{

If($n > 1$)

Return $\left(A\left(\frac{n}{2}\right) \right)$

}

The relation is called recurrence relation

The Recurrence relation of given function is written as follows.

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

Recursive Function

- To solve the Recurrence relation the following methods are used:
 1. Iteration method
 - 2. Recursion-Tree method**
 3. Master Method
 4. Substitution Method

Recursion Tree Method

- Recursion Tree is another method for solving recurrence relations. This method work on two steps. These are
 - First, A set of pre level costs are obtained by sum the cost of each level of the tree and the height of the tree.
 - Second, to determine the total cost of all level of recursion, we sum all the pre level cost.
- This method is best used for good guess.
- For generating good guess, we can ignore floors($\lfloor x \rfloor$) and ceiling ($\lceil x \rceil$) when solving the recurrences. Because they usually do not affect the final guess.

Recursion Tree Method (Example 1)

Example 1

Solve the recurrence $T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2)$ by using recursion tree method.

Recursion Tree Method (Example 1)

Answer:

We start by focusing on finding an upper bound for the solution by using good guess. As we know that floors and ceilings usually do not matter when solving the recurrences, we drop the floor and write the recurrence equation as follows:

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2, c > 0$$

The term cn^2 , at the root represent the costs incurred by the subproblems of size $\frac{n}{4}$.

Recursion Tree Method (Example 1)

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2 \quad , c > 0 \text{ is a constant}$$

$T(n)$

Fig -(a)

Figure (a) shows $T(n)$, which progressively expands in (b) and (d) to form recursion tree.

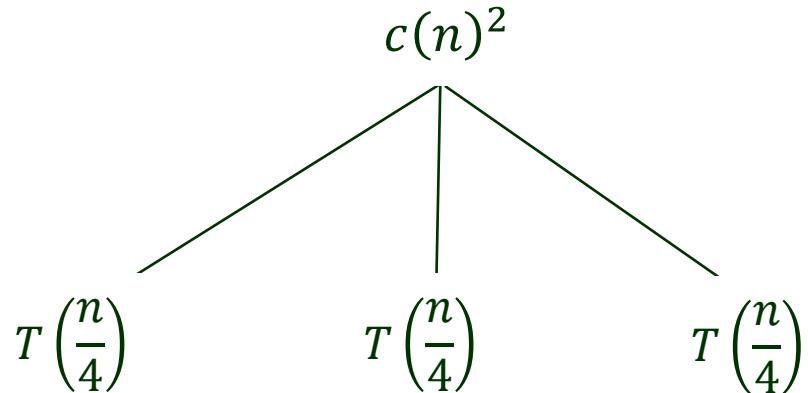


Fig -(b)

Recursion Tree Method (Example 1)

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2 \quad , c > 0 \text{ is a constant}$$

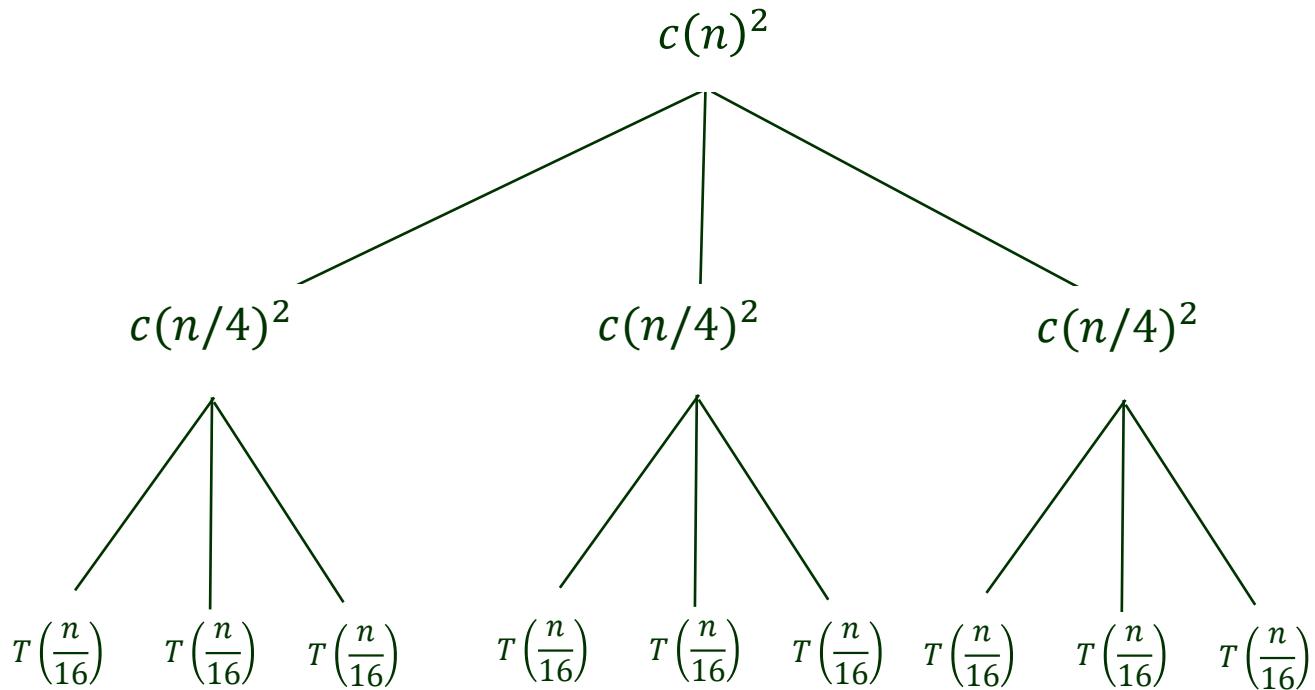


Fig -(c)

Recursion Tree Method (Example 1)

$$T(n) = 3T\left(\frac{n}{4}\right) + cn^2 \quad , c > 0 \text{ is a constant}$$

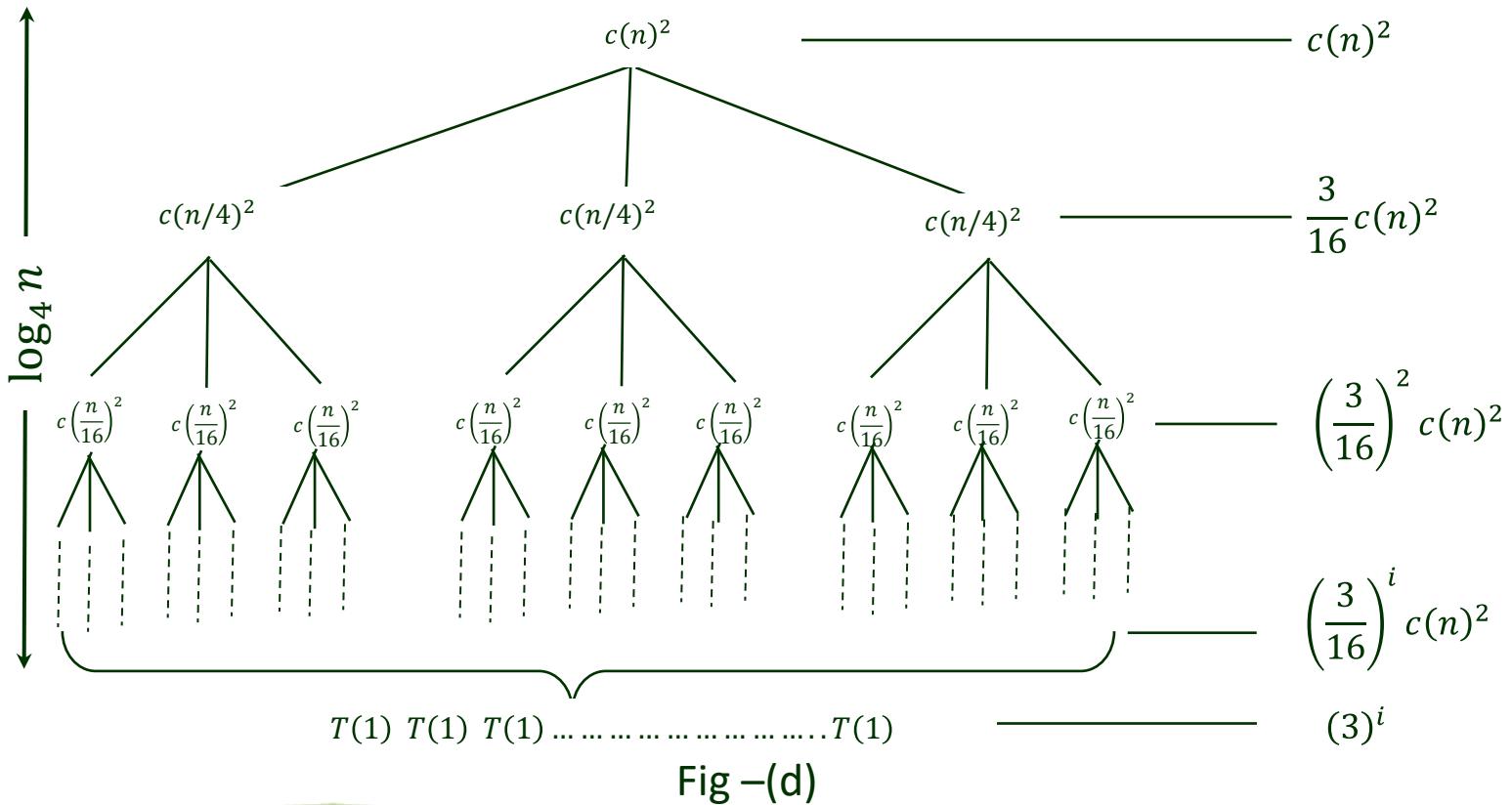


Fig -(d)

Recursion Tree Method (Example 1)

Analysis

First, we find the height of the recursion tree

Observe that a node at depth ' i ' reflects a subproblem of size $\frac{n}{(4)^i}$.

i.e. the subproblem size hits $n = 1$, when $\frac{n}{(4)^i} = 1$

So, if $\frac{n}{(4)^i} = 1$

$$\Rightarrow n = (4)^i \quad (\text{Apply Log both side})$$

$$\Rightarrow \log n = \log(4)^i$$

$$\Rightarrow i = \log_4 n$$

So the height of the tree is $\log_4 n$.

Recursion Tree Method (Example 1)

Second, we determine the cost of each level of the tree.

The number of nodes at depth ' i ' is $(3)^i$. It was observed that

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \dots + \left(\frac{3}{16}\right)^i cn^2 + (3)^i$$

So, each node at depth ' i ' (i.e. $i = 0, 1, 2, 3, 4, \dots, \log_4 n - 1$) has cost $c\left(\frac{n}{4^i}\right)^2$.

Hence the total cost at level ' i ' is $3^i c\left(\frac{n}{4^i}\right)^2$

$$\Rightarrow 3^i \cdot c \cdot \left(\frac{n}{4^i}\right)^2 \Rightarrow 3^i \cdot c \cdot \frac{n^2}{16^i} \Rightarrow \frac{3^i}{16^i} \cdot c \cdot n^2 \Rightarrow \left(\frac{3}{16}\right)^i \cdot c \cdot n^2$$

Recursion Tree Method (Example 1)

However, the bottom level is special. Each of the bottom node has contribute cost = $T(1)$

Hence the cost of the bottom level is = 3^i

$$\Rightarrow 3^{\log_4 n} \quad (\text{as } i = \log_4 n \text{) the height of the tree}$$

$$\Rightarrow n^{\log_4 3}$$

So, the total cost of entire tree is

$T(n)$

$$= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 + \dots + \dots + \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = \sum_{i=0}^{\log_4 n} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

Recursion Tree Method (Example 1)

The left term is just a sum of Geometric series. So the value of $T(n)$ is as follows.

$$T(n) = \left(\frac{\left(\frac{3}{16}\right)^{\log_4 n} - 1}{\left(\frac{3}{16} - 1\right)} \right) cn^2 + \Theta(n^{\log_4 3})$$

The above equation looks very complicated. So, we use an infinite geometric series as an upper bound. Hence the new form of the equation is given below:

$$T(n) = \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) \leq \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) \leq \frac{1}{1 - \left(\frac{3}{16}\right)} cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) \leq \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = O(n^2)$$

Sum of finite Geometric Progression

$$S_n = a + ar + ar^2 + \dots + ar^n$$

$$S_n = \sum_{i=0}^n ar^i = a \left(\frac{r^{n+1} - 1}{r - 1} \right)$$

Recursion Tree Method (Example 2)

Example 2

Solve the recurrence $T(n) = 4T\left(\frac{n}{2}\right) + n$ by using recursion tree method.

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \quad , c > 0$$

The term cn , at the root represent the costs incurred by the subproblems of size $\frac{n}{2}$.

Construction of Recursion tree

$$T(n)$$

Fig -(a)

Figure (a) shows $T(n)$, which progressively expands in (b) to form recursion tree.

Recursion Tree Method (Example 2)

$$T(n) = 4T\left(\frac{n}{2}\right) + cn \quad , c > 0$$

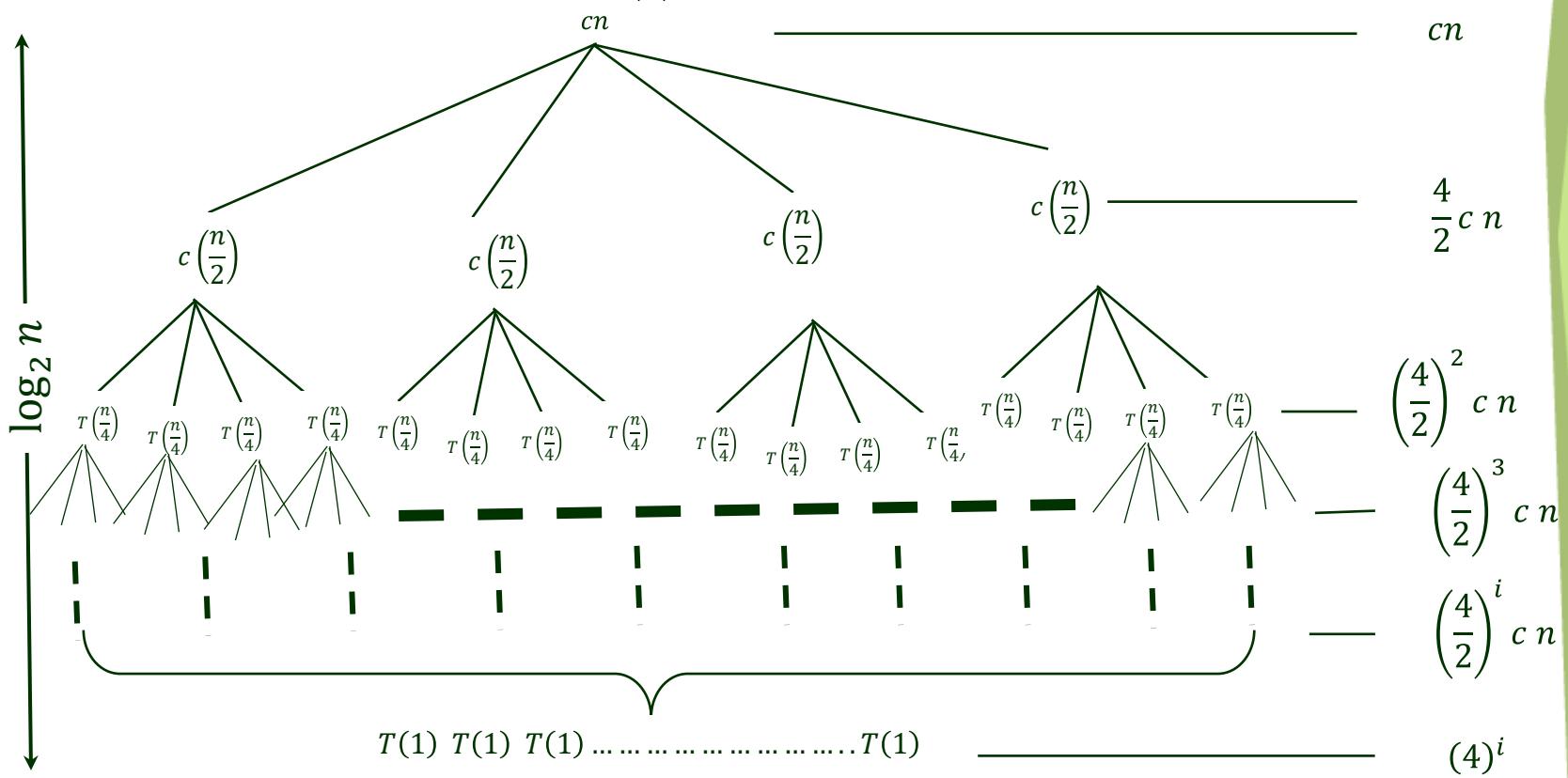


Fig –(b)

Recursion Tree Method (Example 2)

Analysis

First, we find the height of the recursion tree

Observe that a node at depth ' i ' reflects a subproblem of size $\frac{n}{(2)^i}$.

i.e. the subproblem size hits $n = 1$, when $\frac{n}{(2)^i} = 1$

So, if $\frac{n}{(2)^i} = 1$

$$\Rightarrow n = (2)^i \quad (\text{Apply Log both side})$$

$$\Rightarrow \log n = \log(2)^i$$

$$\Rightarrow i = \log_2 n$$

So the height of the tree is $\log_2 n$.

Recursion Tree Method (Example 2)

Second, we determine the cost of each level of the tree.

The number of nodes at depth ' i ' is $(4)^i$.

So, each node at depth ' i '(i.e. $i = 0,1,2,3,4, \dots, \log_2 n - 1$) has cost

$$c \frac{n}{2^i}$$

Hence the total cost at level ' i ' is $4^i c \frac{n}{2^i}$

$$\Rightarrow 4^i \cdot c \cdot \frac{n}{2^i}$$

$$\Rightarrow 4^i \cdot c \cdot \frac{n}{2^i}$$

$$\Rightarrow \frac{4^i}{2^i} \cdot cn$$

$$\Rightarrow \left(\frac{4}{2}\right)^i \cdot cn$$

Recursion Tree Method (Example 2)

However, the bottom level is special. Each of the bottom node has contribute cost = $T(1)$

Hence the cost of the bottom level is = 4^i

$$\Rightarrow 4^{\log_2 n} \quad (as i = \log_2 n) \text{ the height of the tree}$$
$$\Rightarrow n^{\log_2 4}$$

So, the total cost of entire tree is

$$T(n) = cn + \frac{4}{2}cn + \left(\frac{4}{2}\right)^2 cn + \left(\frac{4}{2}\right)^3 cn + \dots + \dots + \left(\frac{4}{2}\right)^i cn + \Theta(n^{\log_2 4})$$
$$T(n) = \sum_{i=0}^{\log_2 n} \left(\frac{4}{2}\right)^i cn + \Theta(n^{\log_2 4})$$

Recursion Tree Method (Example 2)

The left term is just a sum of Geometric series. So the value of $T(n)$ is as follows.

$$T(n) = \left(\frac{\left(\frac{4}{2}\right)^{\log_2 n} - 1}{\left(\frac{4}{2} - 1\right)} \right) cn + \Theta(n^{\log_2 4})$$

$$T(n) = \left(\frac{(2)^{\log_2 n} - 1}{(2 - 1)} \right) cn + \Theta(n^2)$$

$$T(n) = \left(\frac{(n)^{\log_2 2} - 1}{(2 - 1)} \right) cn + cn^2$$

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

$$T(n) = cn^2 - cn + cn^2$$

$$T(n) = 2cn^2 - cn$$

$$\text{Hence, } T(n) = \Theta(n^2)$$

Sum of finite Geometric Progression is

$$S_n = a + ar + ar^2 + \dots + ar^n$$
$$S_n = \sum_{i=0}^n ar^i = a \left(\frac{r^{n+1} - 1}{r - 1} \right)$$

Recursion Tree Method (Example 3)

Example 3

Solve the recurrence $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ by using recursion tree method.

$$T(n) = 2T(n/2) + cn, c > 0$$

The term cn , at the root represent the costs incurred by the subproblems of size $\frac{n}{2}$.

Construction of Recursion tree

$$\begin{array}{c} T(n) \\ \text{Fig -(a)} \end{array}$$

Figure (a) shows $T(n)$, which progressively expands in (b) to form recursion tree.

Recursion Tree Method (Example 3)

$$T(n) = 2T(n/2) + cn \quad , c > 0$$

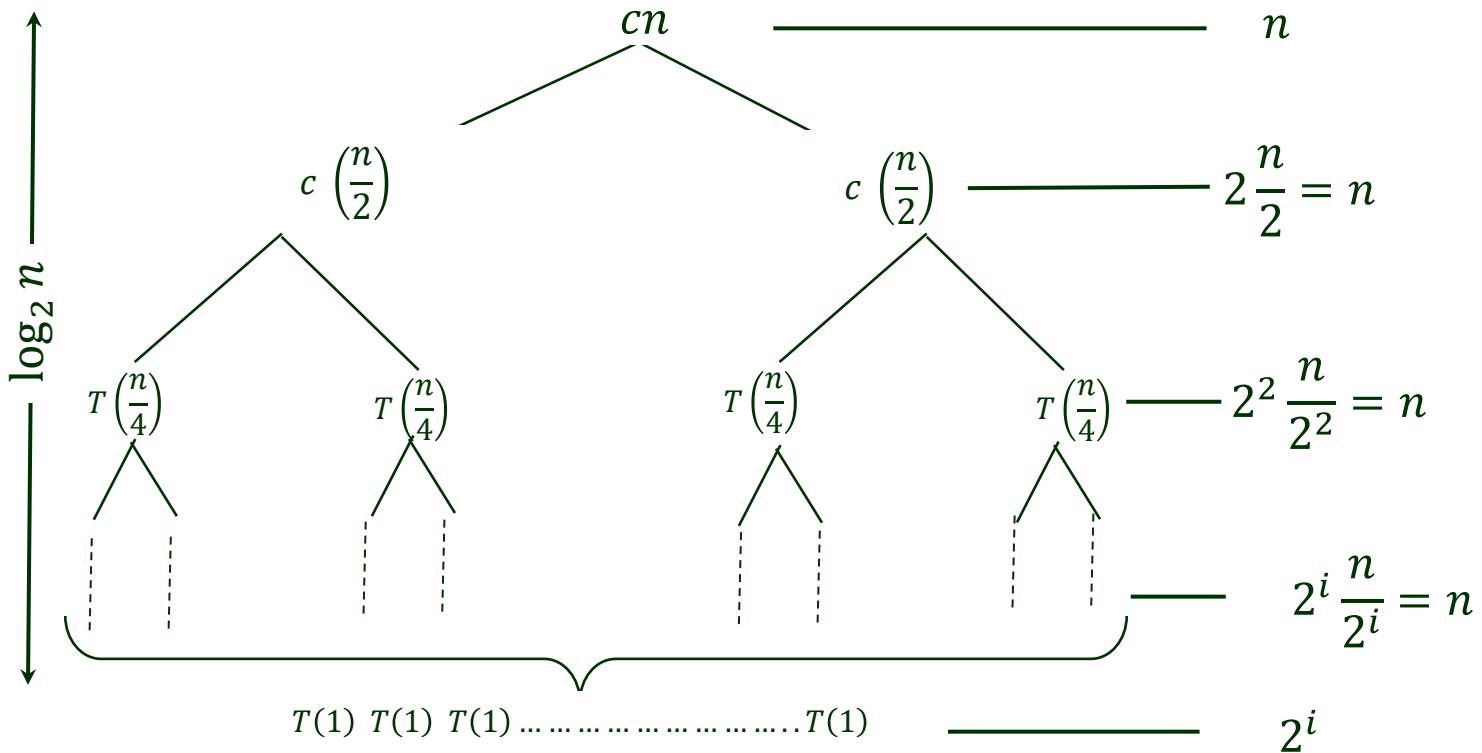


Fig -(b)

Recursion Tree Method (Example 3)

Analysis

First, we find the height of the recursion tree

Observe that a node at depth ' i ' reflects a subproblem of size $\frac{n}{(2)^i}$.

i.e. the subproblem size hits $n = 1$, when $\frac{n}{(2)^i} = 1$

So, if $\frac{n}{(2)^i} = 1$

$$\Rightarrow n = (2)^i \quad (\text{Apply Log both side})$$

$$\Rightarrow \log n = \log(2)^i$$

$$\Rightarrow i = \log_2 n$$

So the height of the tree is $\log_2 n$.

Recursion Tree Method (Example 3)

Second, we determine the cost of each level of the tree.

The number of nodes at depth ' i ' is $(4)^i$.

So, each node at depth ' i ' (*i.e.* $i = 0, 1, 2, 3, 4, \dots, \log_2 n - 1$) has cost $c\left(\frac{n}{2^i}\right)$.

Hence the total cost at level ' i ' is $2^i c\left(\frac{n}{2^i}\right) = cn$

However, the bottom level is special. Each of the bottom node has contribute cost = $T(1)$

Hence the cost of the bottom level is $= 2^i$

$$\Rightarrow 2^{\log_2 n} \quad (\text{as } i = \log_2 n \text{) the height of the tree}$$

$$\Rightarrow n^{\log_2 2}$$

$$\Rightarrow n$$

Recursion Tree Method (Example 3)

So, the total cost of entire tree is

$$T(n) = cn + cn + cn + cn + \dots + \dots + cn$$

$$T(n) = cn \sum_{i=0}^{\log_2 n} 1^i .$$

$$T(n) = cn (\log_2 n + 1)$$

$$T(n) = cn \log_2 n + cn$$

$$\text{Hence, } T(n) = \Theta(n \log_2 n)$$

Recursion Tree Method (Example 4)

Example 4

Solve the recurrence $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + \Theta(n)$ by using recursion tree method.

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + cn, c > 0$$

The term cn , at the root represent the costs incurred by the subproblems of size $\frac{n}{2}$, $\frac{n}{4}$, and $\frac{n}{8}$.

Construction of Recursion tree

$$\begin{array}{c} T(n) \\ \text{Fig -(a)} \end{array}$$

Figure (a) shows $T(n)$, which progressively expands in (b) to form recursion tree.

Recursion Tree Method (Example 4)

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + cn \quad , c > 0$$

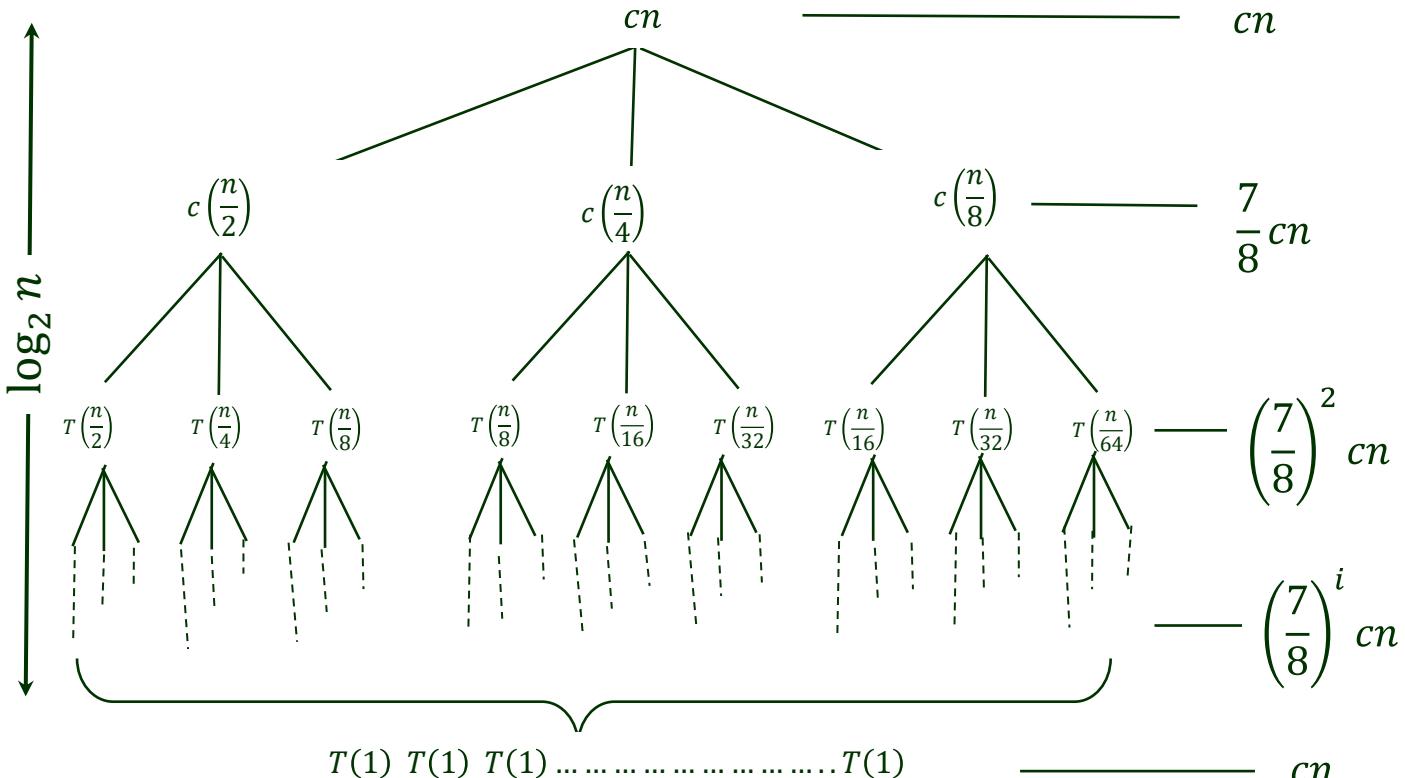


Fig -(b)

Recursion Tree Method (Example 4)

Analysis

First, we find the height of the recursion tree

Hear the problem divide into three subproblem of size $\frac{n}{2^i}, \frac{n}{4^i},$ and $\frac{n}{8^i}.$

For calculating the height of the tree, we consider the longest path of the tree. It has been observed that the node on the left-hand side is the longest path of the tree.

Hence the node at depth ' i ' reflects a subproblem of size $\frac{n}{2^i}.$

i.e. the subproblem size hits $n = 1$, when $\frac{n}{2^i} = 1$

So, if $\frac{n}{2^i} = 1$

$\Rightarrow n = 2^i$ (*Apply Log both side*)

$\Rightarrow \log n = \log(2)^i$

$\Rightarrow i = \log_2 n$

So the height of the tree is $\log_2 n.$

Recursion Tree Method (Example 4)

Second, we determine the cost of the tree in level ' i ' = $\left(\frac{n}{2^i} + \frac{n}{4^i} + \frac{n}{8^i}\right)$

So, the total cost of the tree is:

$$T(n) = cn + \frac{7}{8}cn + \left(\frac{7}{8}\right)^2 cn + \left(\frac{7}{8}\right)^3 cn + \dots + \dots$$

For simplicity we take ∞ Geometric Series

$$T(n) = cn + \frac{7}{8}cn + \left(\frac{7}{8}\right)^2 cn + \left(\frac{7}{8}\right)^3 cn + \dots + \infty$$

$$T(n) \leq \sum_{i=0}^{\infty} \left(\frac{7}{8}\right)^i cn$$

$$T(n) \leq cn \left[\frac{1}{1 - \frac{7}{8}} \right]$$

$$T(n) \leq cn \left[\frac{8}{1} \right]$$

$$T(n) \leq 8cn$$

$$\text{Hence, } T(n) = O(n)$$

Thank U

Design and Analysis of Algorithm

(Heap Sort)

Lecture -14 - 17

Overview

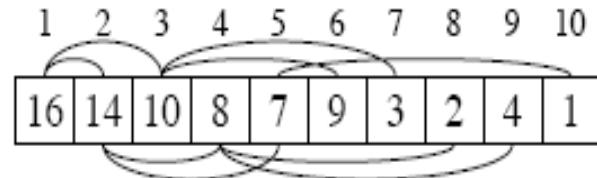
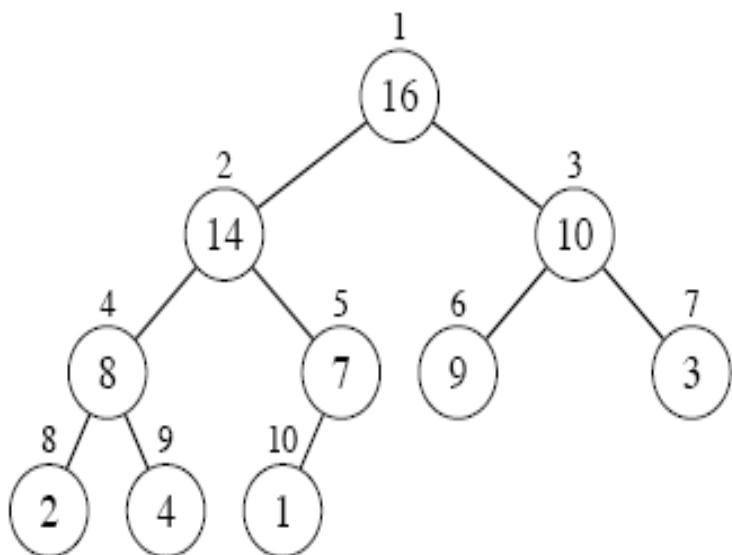
- $O(n \lg n)$ worst case time complexity.
- Sorts in place-like insertion sort.
- $O(n)$ is the tight bound analysis of Build Heap.

Heap data structure

- Heap A (*not* garbage-collected storage) is a nearly complete binary tree.
 - *Height* of node = # of edges on a longest simple path from the node down to a leaf.
 - *Height* of heap = height of root = $\Theta(\lg n)$.
- A heap can be stored as an array A .
 - Root of tree is $A[1]$.
 - Parent of $A[i] = A[\lfloor i/2 \rfloor]$.
 - Left child of $A[i] = A[2i]$.
 - Right child of $A[i] = A[2i + 1]$.
 - Computing is fast with binary representation implementation.

Example

Example: of a max-heap. [Arcs above and below the array on the right go between parents and children. There is no significance to whether an arc is drawn above or below the array.]



Example

- Given an array of size N. The task is to sort the array elements by using Heap Sort.
 - Input:
 - N=10
 - Arr[]:{16, 4, 10, 14, 7, 9, 3, 2, 8, 1}
 - Output: 1 2 3 4 7 8 9 10 14 16

Example

- Given an array of size N. The task is to sort the array elements by using Heap Sort.
 - Input:
 - N = 10
 - arr[] = {10,9,8,7,6,5,4,3,2,1}
 - Output: 1 2 3 4 5 6 7 8 9 10

Heap property

- For max-heaps (largest element at root),
max-heap property: for all nodes i ,
excluding the root, $A[\text{PARENT}(i)] \geq A[i]$.
- For min-heaps (smallest element at root),
min-heap property: for all nodes i ,
excluding the root, $A[\text{PARENT}(i)] \leq A[i]$.

Maintaining the heap property

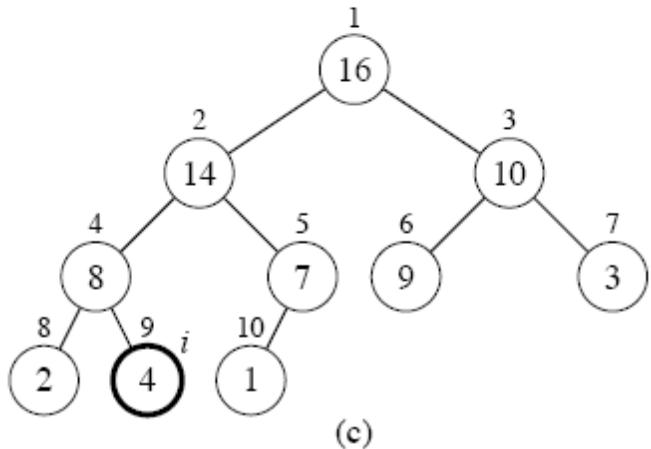
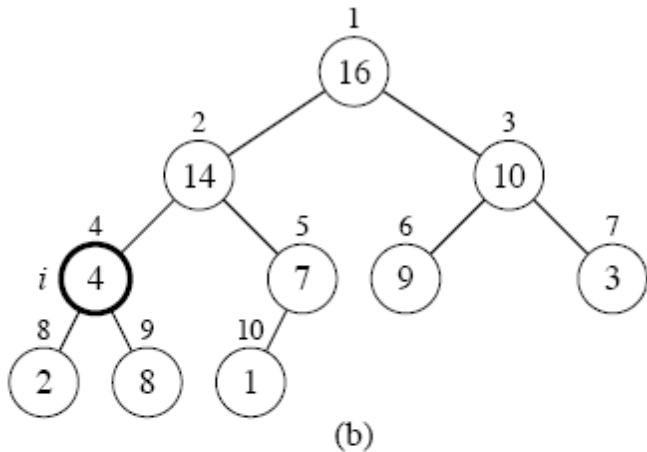
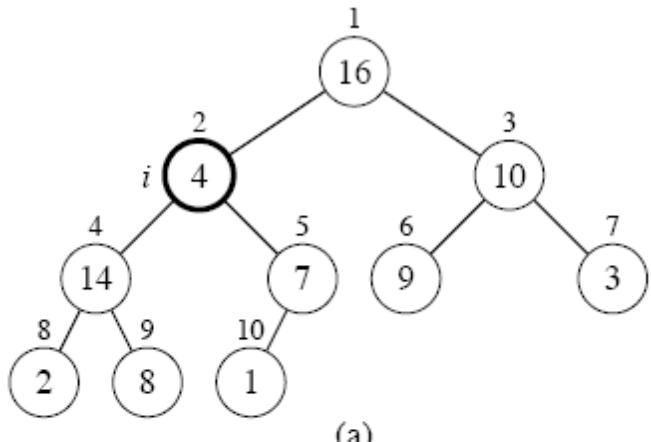
MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

- Before MAX-HEAPIFY, $A[i]$ may be smaller than its children.
- Assume left and right subtrees of i are max-heaps.
- After MAX-HEAPIFY, subtree rooted at i is a max-heap.

The way MAX-HEAPIFY works:

- Compare $A[i]$, $A[\text{LEFT}(i)]$, and $A[\text{RIGHT}(i)]$.
- If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
- Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

Run MAX-HEAPIFY on the following heap example.



MAX-HEAPIFY(A, i, n)

$l \leftarrow \text{LEFT}(i)$

$r \leftarrow \text{RIGHT}(i)$

if $l \leq n$ and $A[l] > A[i]$

then $\text{largest} \leftarrow l$

else $\text{largest} \leftarrow i$

if $r \leq n$ and $A[r] > A[\text{largest}]$

then $\text{largest} \leftarrow r$

if $\text{largest} \neq i$

then exchange $A[i] \leftrightarrow A[\text{largest}]$

 MAX-HEAPIFY($A, \text{largest}, n$)

```
MAX-HEAPIFY( $A, i, n$ )
 $l \leftarrow \text{LEFT}(i)$ 
 $r \leftarrow \text{RIGHT}(i)$ 
if  $l \leq n$  and  $A[l] > A[i]$ 
    then  $\text{largest} \leftarrow l$ 
    else  $\text{largest} \leftarrow i$ 
if  $r \leq n$  and  $A[r] > A[\text{largest}]$ 
    then  $\text{largest} \leftarrow r$ 
if  $\text{largest} \neq i$ 
    then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 
        MAX-HEAPIFY( $A, \text{largest}, n$ )
```

The Time Complexity
of MAX-HEAPIFY (A,i,n)
is $O(\log n)$.
[Because any element
insert in a tree
required maximum
 $\log n$ time]

Building a heap

```
BUILD-MAX-HEAP( $A, n$ )
for  $i \leftarrow \lfloor n/2 \rfloor$  downto 1
    do MAX-HEAPIFY( $A, i, n$ )
```

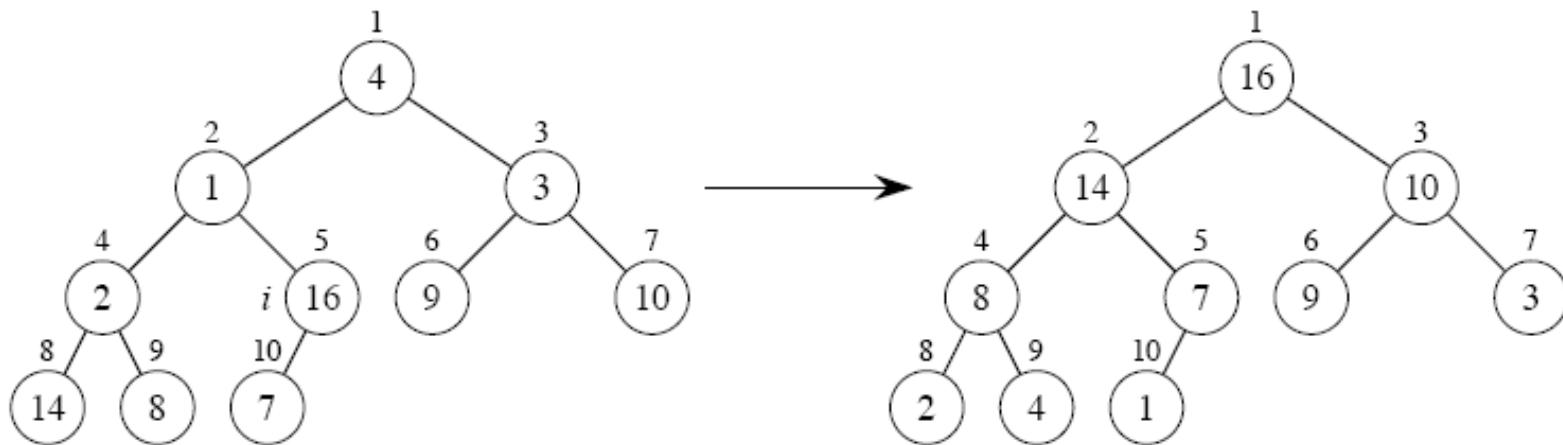
[Parameter n replaces both attributes $\text{length}[A]$ and $\text{heap-size}[A]$.]

Example

Building a max-heap from the following unsorted array results in the first heap example.

- i starts off as 5.
- MAX-HEAPIFY is applied to subtrees rooted at nodes (in order): 16, 2, 3, 1, 4.

A	1	2	3	4	5	6	7	8	9	10
	4	1	3	2	16	9	10	14	8	7



- Node 2 violates the max-heap property.
- Compare node 2 with its children, and then swap it with the larger of the two children.
- Continue down the tree, swapping until the value is properly placed at the root of a subtree that is a max-heap. In this case, the max-heap is a leaf.

Time: $O(\lg n)$.

Correctness: [Instead of book's formal analysis with recurrence, just come up with $O(\lg n)$ intuitively.] Heap is almost-complete binary tree, hence must process $O(\lg n)$ levels, with constant work at each level (comparing 3 items and maybe swapping 2).

Correctness

Initialization:

Initialization: we know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of the **for** loop, the invariant is initially true.

Maintenance: Children of node i are indexed higher than i , so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \dots, n$ are all roots of max-heaps, MAX-HEAPIFY makes node i a max-heap root. Decrementing i re-establishes the loop invariant at each iteration.

Termination: When $i = 0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis

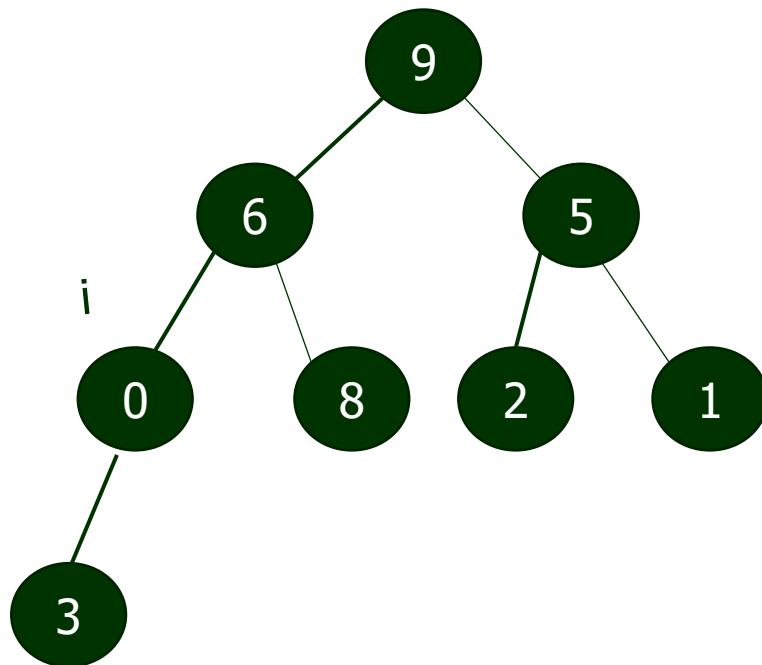
- **Simple bound:** $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$.
- **Tighter analysis observation:**
An n element heap has height $\lfloor \log n \rfloor$ and at most $\left\lceil \frac{n}{2^h + 1} \right\rceil$ nodes of any height h .

Tighter analysis Proof

BUILD-MAX-HEAP(A, n)

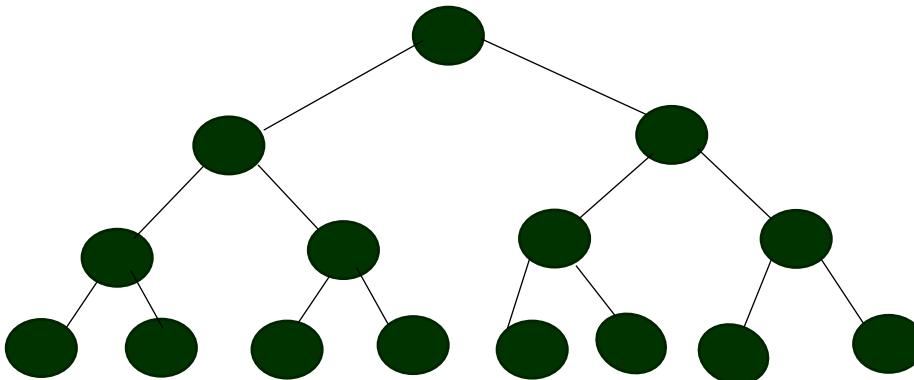
for $i \leftarrow \lfloor n/2 \rfloor$ *downto* 1
 do MAX-HEAPIFY (A, i, n)

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



Tighter analysis Proof

- For easy understanding, Let us take a complete binary Tree,



- The height of a node is the number of edges from the node to the deepest leaf.
- The depth of a node is the no of edges from the root of the node.

Tighter analysis Proof

- All the leaves are of height 0, therefore there are 8 nodes at height 0.
- 4 numbers of nodes at height 1.
- 2 numbers of nodes at height 2.
- and one node at height 3.

Tighter analysis Proof

- Hence the question is how many nodes are there at height 'h' in a complete binary tree?
- The answer is :
- If there are n nodes in tree, then at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes are available at height h.

Tighter analysis Proof

- Now if we apply MAX-HEAPIFY() on any node of any level, then the time taken by MAX-HEAPIFY() is the height of the node.(i.e. $\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$)
- Hence in case of root the time taken is $\log n$.
- Hence

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$$

Work done by
Max-Heapify()
on the number
of nodes of
height h.

Tighter analysis Proof

$$\begin{aligned} T(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) \\ &\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &\leq O\left(\frac{n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &\leq O\left(\frac{n}{2} \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h\right) \\ &\leq O\left(\frac{n}{2} \left[\frac{1/2}{(1-1/2)^2}\right]\right) \Rightarrow O\left(\frac{n}{2} 2\right) \end{aligned}$$

$$T(n) \Rightarrow O(n)$$

Hence the running time of **BUILD-MAX-HEAP(A,n)** is $O(n)$ in tight bound .

This is a Harmonic Series. By Integrating and Differentiating the series we get the value is 2.

(Refer to next slide or cormen Appendix A.8)

Tighter analysis Proof

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad [\text{value of Infanite G P Series}]$$

$$\sum_{k=0}^{\infty} x^k = (1-x)^{-1}$$

Differentiate both side:

$$\sum_{k=0}^{\infty} k \cdot x^{k-1} = (-1)(1-x)^{-2}(-1) = \frac{1}{(1-x)^2}$$

Multiply x both side

$$\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$$

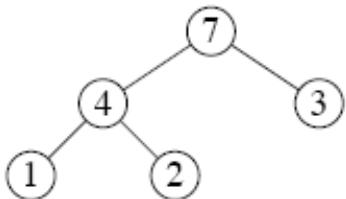
The heapsort algorithm

Given an input array, the heapsort algorithm acts as follows:

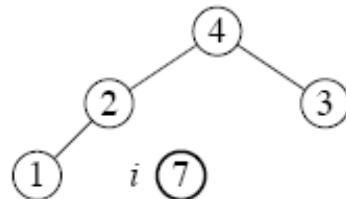
- Builds a max-heap from the array.
- Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
- “Discard” this last node (knowing that it is in its correct place) by decreasing the heap size, and calling MAX-HEAPIFY on the new (possibly incorrectly-placed) root.
- Repeat this “discarding” process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

```
HEAPSORT( $A, n$ )
BUILD-MAX-HEAP( $A, n$ )
for  $i \leftarrow n$  downto 2
    do exchange  $A[1] \leftrightarrow A[i]$ 
    MAX-HEAPIFY( $A, 1, i - 1$ )
```

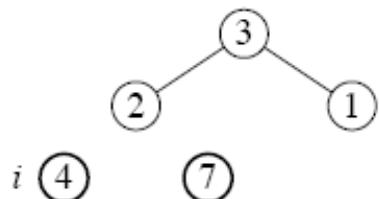
Example



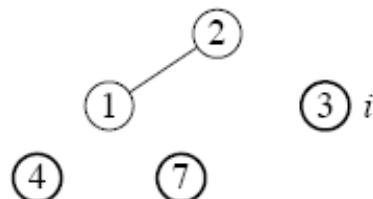
(a)



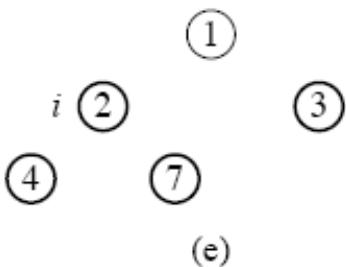
(b)



(c)



(d)



(e)

A

1	2	3	4	7
---	---	---	---	---

Analysis

- BUILD-MAX-HEAP: $O(n)$
- **for** loop: $n - 1$ times
- exchange elements: $O(1)$
- MAX-HEAPIFY: $O(\lg n)$

Total time: $O(n \lg n)$.

Heap implementation of priority queue

- Heaps efficiently implement priority queues. These notes will deal with max priority queues implemented with max-heaps. Min-priority queues are implemented with min-heaps similarly.
- A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take $O(\lg n)$ time.

Priority queue

- Maintains a dynamic set S of elements.
- Each set element has a **key**-an associated value.
- Max-priority queue supports dynamic-set operations:
 - $\text{INSERT}(S, x)$: inserts element x into set S .
 - $\text{MAXIMUM}(S)$: returns element of S with largest key.
 - $\text{EXTRACT-MAX}(S)$: removes and returns element of S with largest key.
 - $\text{INCREASE-KEY}(S, x, k)$: increases value of element x 's key to k . Assume $k \geq x$'s current key value.
- Example max-priority queue application: schedule jobs on shared computer.

- Min-priority queue supports similar operations:
 - $\text{INSERT}(S, x)$: inserts element x into set S .
 - $\text{MINIMUM}(S)$: returns element of S with smallest key.
 - $\text{EXTRACT-MIN}(S)$: removes and returns element of S with smallest key.
 - $\text{DECREASE-KEY}(S, x, k)$: decreases value of element x 's key to k . Assume $k \leq x$'s current key value.
- Example min-priority queue application:
event - driven simulator.

Finding the maximum element

Getting the maximum element is easy: it's the root.

HEAP-MAXIMUM(A)

return $A[1]$

Finding the maximum element

Getting the maximum element is easy: it's the root.

```
HEAP-MAXIMUM( $A$ )
```

```
    return  $A[1]$ 
```

Time: $O(1)$.

Extracting max element

Given the array A :

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

Extracting max element

Given the array A :

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)

if $n < 1$

then error .heap underflow.

$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY($A, 1, n - 1$) remakes heap

return max

Extracting max element

Given the array A :

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

HEAP-EXTRACT-MAX(A, n)

if $n < 1$

then error .heap underflow.

$max \leftarrow A[1]$

$A[1] \leftarrow A[n]$

MAX-HEAPIFY($A, 1, n - 1$) remakes heap

return max

Time Complexity
is $O(\log n)$

- **HEAP-INCREASE-KEY(A,i,key)**
 1. If $\text{key} < A[i]$
 2. "error" new key is smaller than the current key".
 3. $A[i] = \text{key}$
 4. While $i > 1$ and $A[\text{parent}(i)] < A[i]$
 5. $\text{swap}(A[\text{parent}(i)], A[i])$
 6. $i = \text{parent}(i)$

- **HEAP-INCREASE-KEY(A,i,key)**

1. If $\text{key} < A[i]$
2. "error" new key is smaller than the current key".
3. $A[i] = \text{key}$
4. While $i > 1$ and $A[\text{parent}(i)] < A[i]$
5. $\text{swap}(A[\text{parent}(i)], A[i])$
6. $i = \text{parent}(i)$

The running time of **HEAP-INCREASE-KEY(A,i,key)** is $O(\log n)$

- Example(from own)

- MAX-HEAP-INSERT(A, key)
 1. $A.heap-size = A.heap-size + 1$
 2. $A[heap-size] = -\infty$
 3. HEAP-INCREASE-KEY($A, heap-size, key$)

- MAX-HEAP-INSERT(A, key)
 1. $A.heap-size = A.heap-size + 1$
 2. $A[heap-size] = -\infty$
 3. HEAP-INCREASE-KEY($A, heap-size, key$)

The running time of MAX-HEAP-INSERT(A, key) is $O(\log n)$.

Thank U

Design and Analysis of Algorithm

Linear Time Sorting (Counting Sort)

Lecture -18

Overview

- Running time of counting sort is $O(n+k)$.
- Required extra space for sorting.
- Is a stable sorting.

Counting Sort

- Counting sort is a type of sorting technique which is based on keys between a specific range.
- It works by counting the number of objects having distinct key values (i.e. one kind of hashing).

Counting Sort

- Consider the input set : 4, 1, 3, 4, 3. Then $n=5$ and $k=4$.
- Counting sort determines for each input element x , the number of elements less than x .
- This information is used to place element x directly into its position in the output array.
- For example if there exists 17 elements less than x then x is placed into the 18th position into the output array.

Counting Sort

- **Assumptions:**
 - n records
 - Each record contains keys or data
 - All keys are in the range of 0 to k , where k is the highest key value of the array.
- **Space:**

For coding this algorithm uses three array:

- **Input Array:** $A[1..n]$ store input data , where n is the length of the array.
- **Output Array:** $B[1..n]$ finally store the sorted data
- **Temporary Array:** $C[0..k]$ store data temporarily

Counting Sort

- Let us illustrate the counting sort with an example.
Apply the concept of counting sort on the given array.

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

Counting Sort

- Let us illustrate the counting sort with an example.
Apply the concept of counting sort on the given array.

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

- First create a new array $C[0....k]$, where k is the highest key value. And initialize with 0(i.e. zero)

for i=0 to k
 $C[i] = 0;$

C	0	1	2	3	4	5
	0	0	0	0	0	0

Counting Sort

- Let us illustrate the counting sort with an example.
Apply the concept of counting sort on the given array.

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

- Find the frequencies of each object and store it in C array.

for j=1 to A.length

C[A[j]] = C[A[j]] + 1;

C	0	1	2	3	4	5
	2	0	2	3	0	1

Counting Sort

- Let us illustrate the counting sort with an example.
Apply the concept of counting sort on the given array.

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

- Find the frequencies of each object and store it in C array.

for $j=1$ to $A.length$

$C[A[j]] = C[A[j]] + 1;$

C	0	1	2	3	4	5
	2	0	2	3	0	1

- And then cumulatively add C array.

for $i=1$ to k

$C[i] = C[i] + C[i-1];$

C	0	1	2	3	4	5
	2	2	4	7	7	8

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

B	1	2	3	4	5	6	7	8
						3		

C

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

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

B	1	2	3	4	5	6	7	8
						3		

C	0	1	2	3	4	5		
	2	2	4	7	7	8		

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

j

B	1	2	3	4	5	6	7	8
						3		

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

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

B

1	2	3	4	5	6	7	8
	0					3	

C

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

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

B

1	2	3	4	5	6	7	8
	0					3	

C

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

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

j

B

1	2	3	4	5	6	7	8
	0					3	

C

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

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

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

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

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

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

j

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

C	0	1	2	3	4	5		
	1	2	4	5	7	8		

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	1	2	4	5	7	8		

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	1	2	4	5	7	8		

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

j

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

C	0	1	2	3	4	5		
	1	2	3	5	7	8		

Counting Sort

for j=A.length down to 1

 B[C[A[j]]] = A[j];

 C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	1	2	3	5	7	8		

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
j								

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

C	0	1	2	3	4	5		
	1	2	3	5	7	8		

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
j								

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

C	0	1	2	3	4	5		
	0	2	3	5	7	8		

Counting Sort

for j=A.length down to 1

 B[C[A[j]]] = A[j];

 C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	0	2	3	5	7	8		

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	0	2	3	5	7	8		

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

j

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

C	0	1	2	3	4	5		
	0	2	3	4	7	8		

Counting Sort

for j=A.length down to 1

 B[C[A[j]]] = A[j];

 C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	0	2	3	4	7	8		

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

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

j

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

C	0	1	2	3	4	5		
	0	2	3	4	7	8		

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

j

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

C	0	1	2	3	4	5		
	0	2	3	4	7	7		

Counting Sort

for j=A.length down to 1

 B[C[A[j]]] = A[j];

 C[A[j]] = C[A[j]] - 1;

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
j								

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

C	0	1	2	3	4	5
	0	2	3	4	7	7

Counting Sort

for j=A.length down to 1

B[C[A[j]]] = A[j];

C[A[j]] = C[A[j]] - 1;

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
j								

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

C	0	1	2	3	4	5
	0	2	3	4	7	7

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
j								

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

C	0	1	2	3	4	5
	0	2	2	4	7	7

Counting Sort

Counting-Sort(A, B, k)

1. Let C[0.....k] be a new array
2. for i=0 to k
3. C[i]= 0;
4. for j=1 to A.length
5. C[A[j]] = C[A[j]] + 1;
6. for i=1 to k
7. C[i] = C[i] + C[i-1];
8. for j=A.length down to 1
9. B[C[A[j]]] = A[j];
10. C[A[j]] = C[A[j]] - 1;

Counting Sort

Counting-Sort(A, B, k)

1. Let C[0.....k] be a new array
2. for i=0 to k **[Loop 1]**
3. C[i]= 0;
4. for j=1 to A.length **[Loop 2]**
5. C[A[j]] = C[A[j]] + 1;
6. for i=1 to k **[Loop 3]**
7. C[i] = C[i] + C[i-1];
8. for j=A.length down to 1 **[Loop 4]**
9. B[C[A[j]]] = A[j];
10. C[A[j]] = C[A[j]] - 1;

Complexity Analysis

Counting-Sort(A, B, k)

1. Let C[0.....k] be a new array
2. for i=0 to k [Loop 1] **O(*k*) times**
3. C[i]= 0;
4. for j=1 to A.length [Loop 2] **O(*n*) times**
5. C[A[j]] = C[A[j]] + 1;
6. for i=1 to k [Loop 3] **O(*k*) times**
7. C[i] = C[i] + C[i-1];
8. for j=A.length down to 1 [Loop 2] **O(*n*) times**
9. B[C[A[j]]] = A[j];
10. C[A[j]] = C[A[j]] - 1;

Complexity Analysis

- So the counting sort takes a total time of: $O(n + k)$
- Counting sort is called ***stable sort***.
(A sorting algorithm is ***stable*** when numbers with the same values appear in the output array in the same order as they do in the input array.)

Pro's and Con's of Counting Sort

- Pro's
 - Asymptotically very Fast - $O(n + k)$
 - Simple to code
- Con's
 - Doesn't sort in place.
 - Requires $O(n + k)$ extra storage space.

Thank U

Design and Analysis of Algorithm

Linear Time Sorting (Radix Sort and Bucket Sort)

Lecture -19



Linear Time Sorting

(Radix Sort)

Overview

- Running time of counting sort is $\Theta(d(n + k))$
- Required extra space for sorting.
- Is a stable sorting.

Radix Sort

- Radix sort is non comparative sorting method
- Two classifications of radix sorts are least significant digit (LSD) radix sorts and most significant digit (MSD) radix sorts.
- LSD radix sorts process the integer representations starting from the least digit and move towards the most significant digit. MSD radix sorts work the other way around.

Radix Sort (Algorithm)

Radix_Sort(A,d)

for $i \leftarrow d$ *down to* 1

*Use a stable sort to sort the array A on digit i
(i.e. Counting Sort)*

Radix Sort

- In input array A , each element is a number of d digit.

$\text{Radix_Sort}(A, d)$

for $i \leftarrow 1$ to d

do "use a stable sort to sort array A on digit i ;

329

457

657

839

436

720

355

Radix Sort

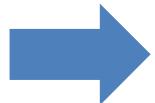
- In input array A , each element is a number of d digit.

$\text{Radix_Sort}(A, d)$

$\text{for } i \leftarrow 1 \text{ to } d$

$\text{do "use a stable sort to sort array } A \text{ on digit } i;$

329	720
457	355
657	436
839	457
436	657
720	329
355	839



i

Radix Sort

- In input array A , each element is a number of d digit.

$\text{Radix_Sort}(A, d)$

for $i \leftarrow 1$ to d

do "use a stable sort to sort array A on digit i ;

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657


i i

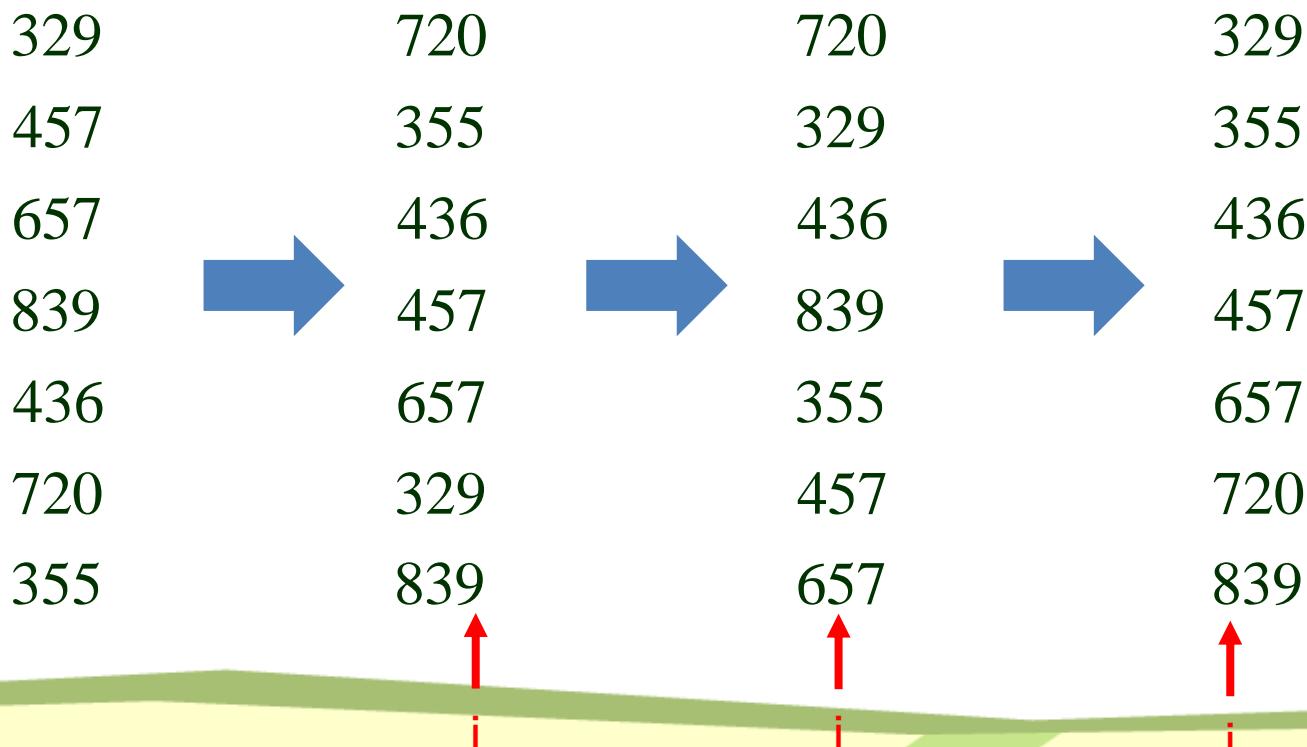
Radix Sort

- In input array A , each element is a number of d digit.

$\text{Radix_Sort}(A, d)$

for $i \leftarrow 1$ to d

do "use a stable sort to sort array A on digit i ;



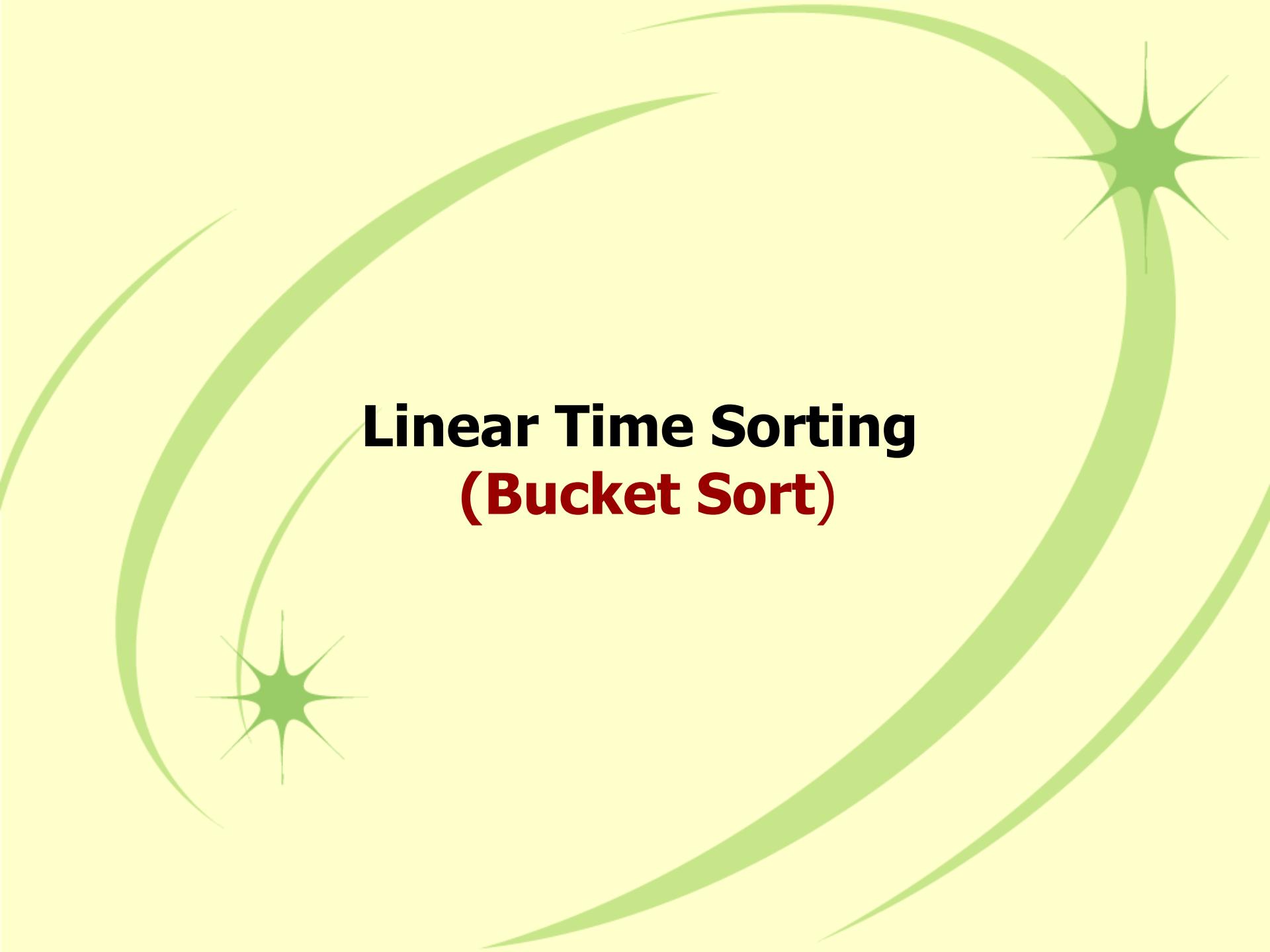
Radix Sort (Analysis)

`Radix_Sort(A,d)`

for $i \leftarrow d$ *down to* 1

*Use a stable sort to sort the array A on digit i
(i.e. Counting Sort)*

- *Here Counting Sort execute for d times.*
- *The running time of Counting Sort is $\Theta(n + k)$*
- *Hence the running time complexity of Radix Sort is $\Theta(d(n + k))$*



Linear Time Sorting

(Bucket Sort)

Overview

- The average time complexity is $O(n + k)$.
- The worst time complexity is $O(n^2)$.
- Required extra space for sorting.
- Is a stable sorting.

Bucket Sort

- Bucket sort is a comparison sort algorithm that operate on elements by dividing them into different bucket and return the result.
- Buckets are assigned based on each element's search key.
- At the time of returning the result, First concatenate each bucket one by one and then return the result in a single array.

Bucket Sort

- Some variations
 - Make enough buckets so that each will only hold one element, use a count for duplicates.
 - Use fewer buckets and then sort the contents of each bucket.
- The more buckets you use, the faster the algorithm will run but it uses more memory.

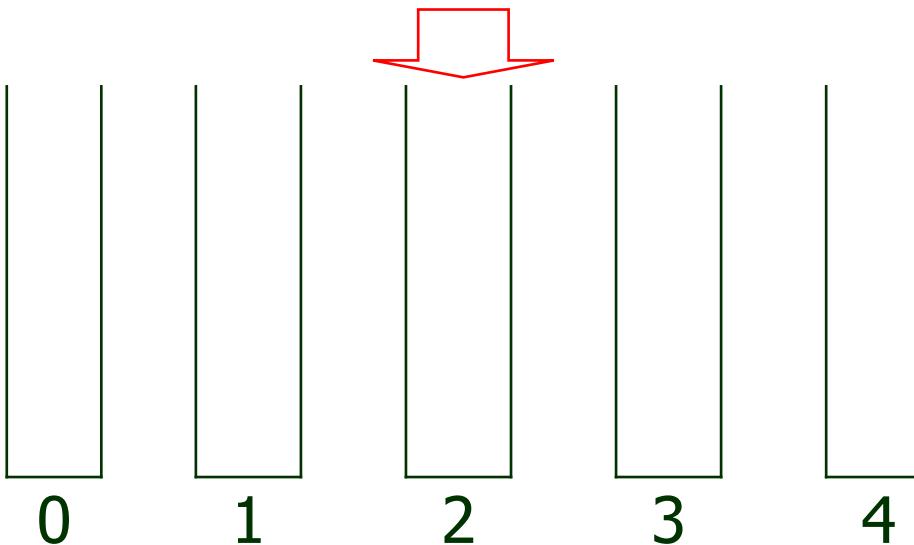
Bucket Sort

- Time complexity is reduced when the number of items per bucket is evenly distributed and it is closed to one item per bucket.
- As buckets require extra space, This algorithm trading increased space consumption for a lower time complexity.
- In general, Bucket Sort beats all other sorting techniques in time complexity but can require a huge of space.

Bucket Sort

- One Value per bucket:

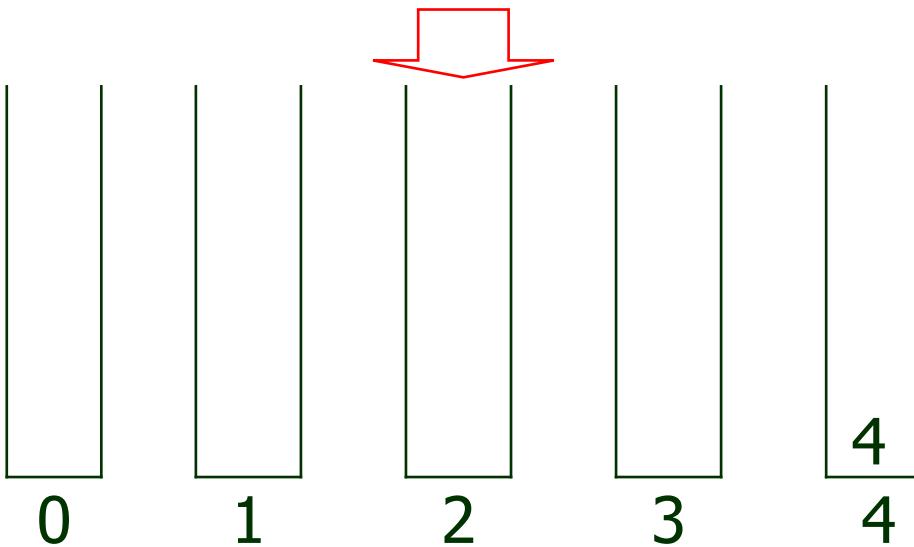
4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



Bucket Sort

- One Value per bucket:

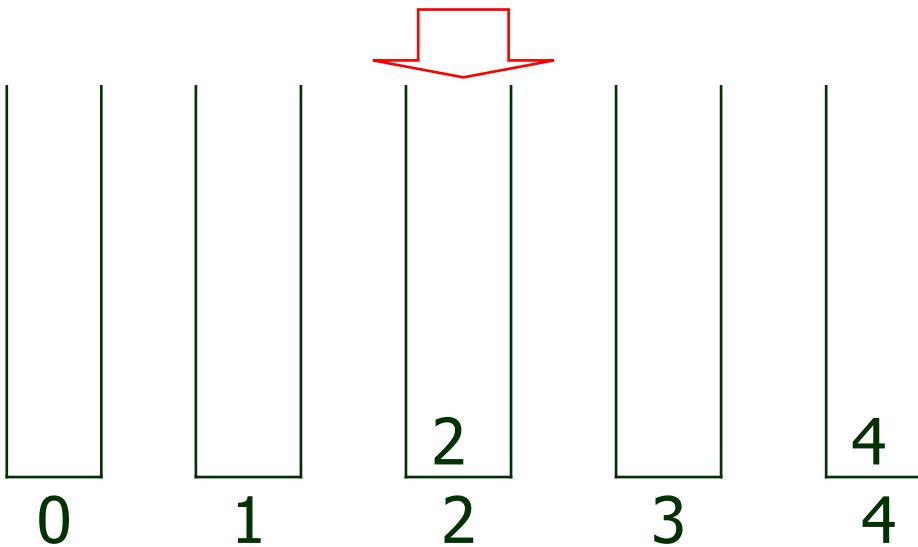
4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



Bucket Sort

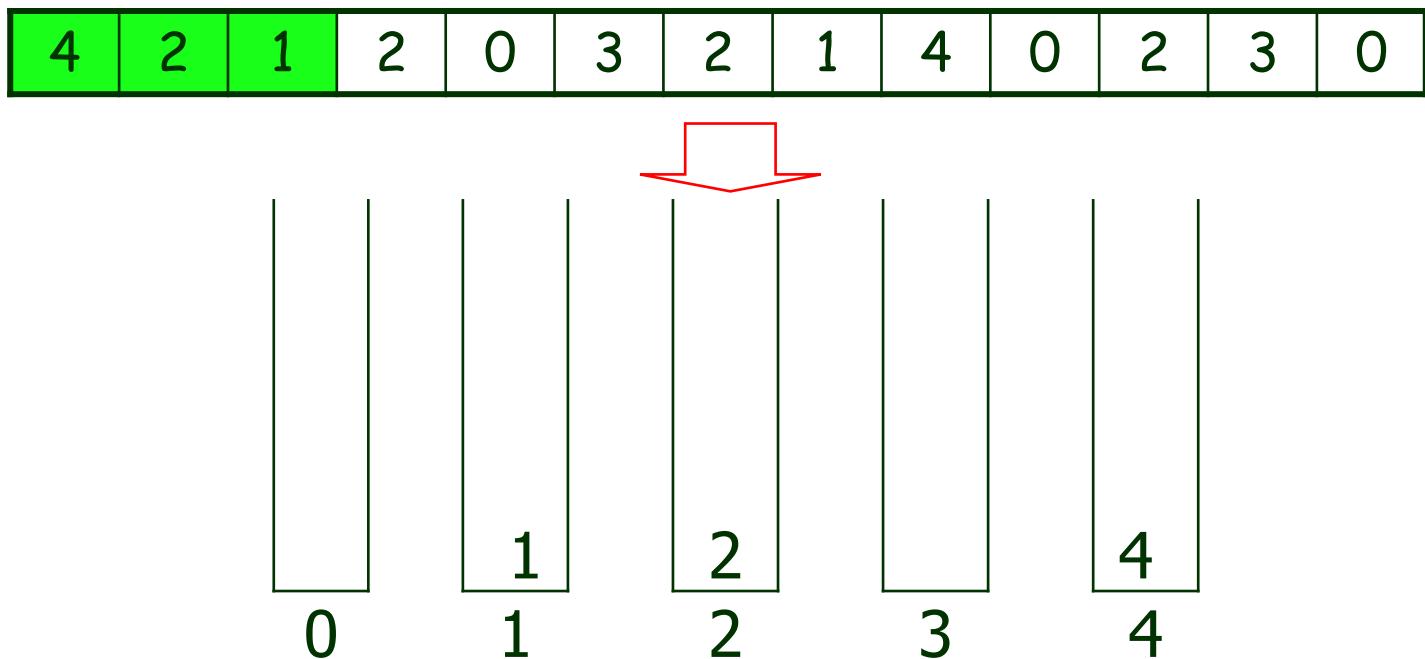
- One Value per bucket:

4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



Bucket Sort

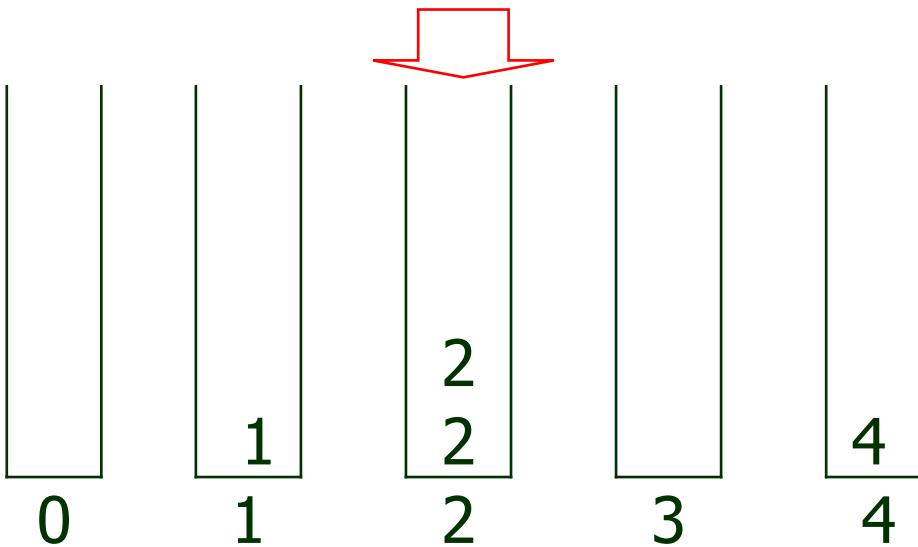
- One Value per bucket:



Bucket Sort

- One Value per bucket:

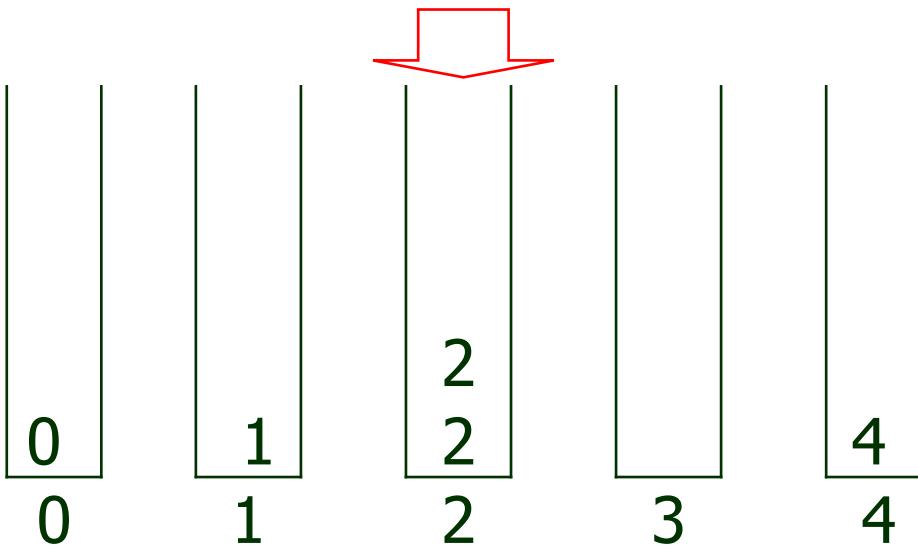
4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



Bucket Sort

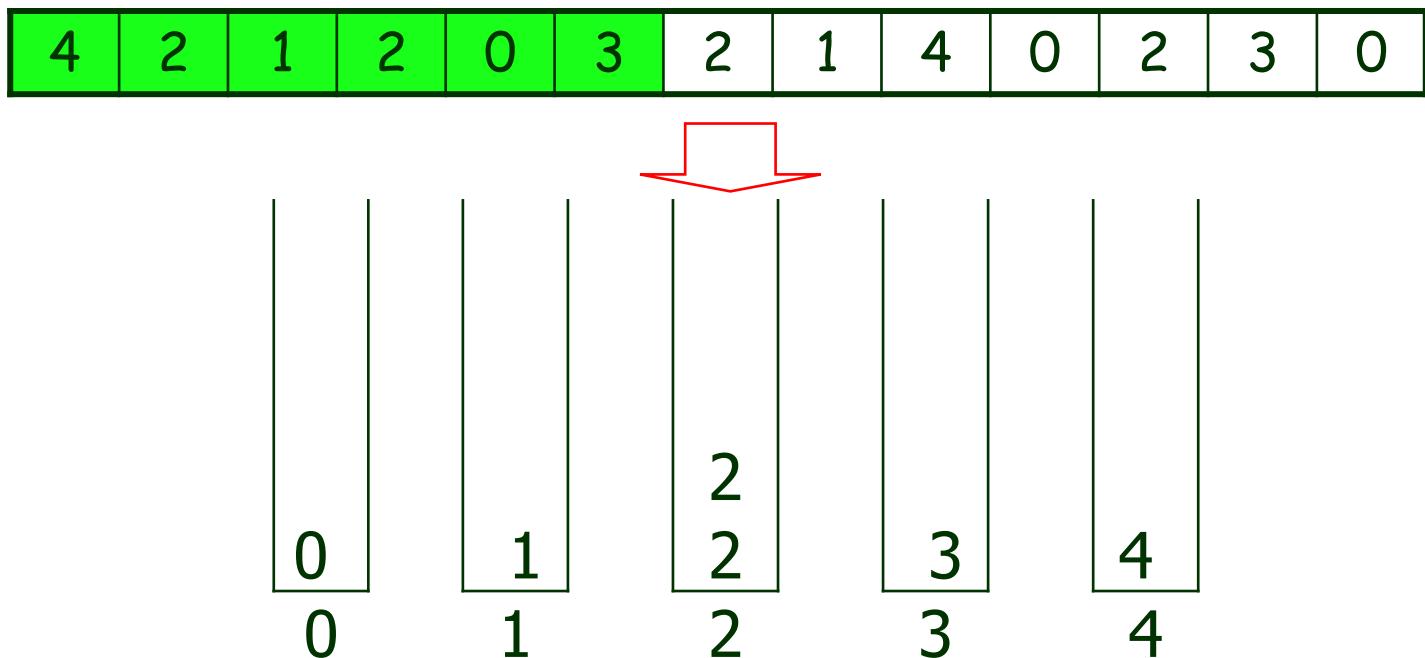
- One Value per bucket:

4	2	1	2	0	3	2	1	4	0	2	3	0
---	---	---	---	---	---	---	---	---	---	---	---	---



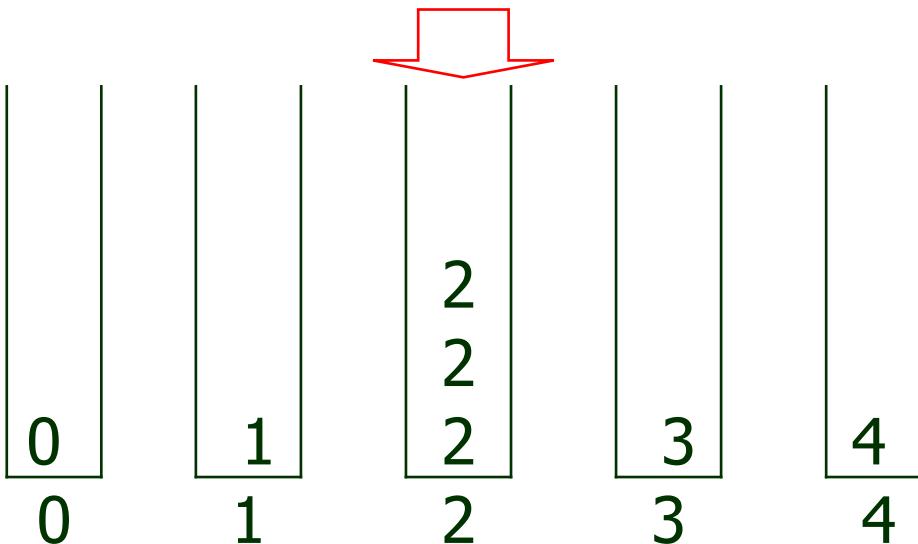
Bucket Sort

- One Value per bucket:



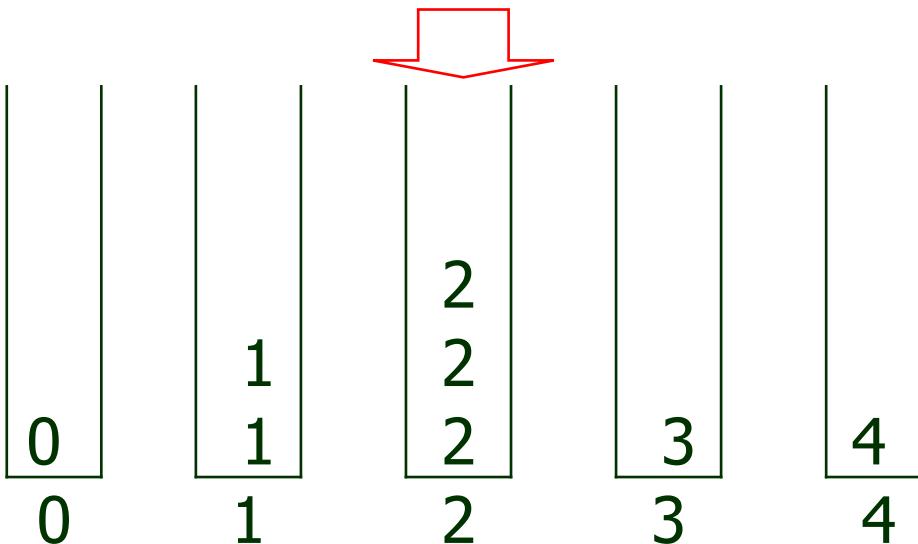
Bucket Sort

- One Value per bucket:



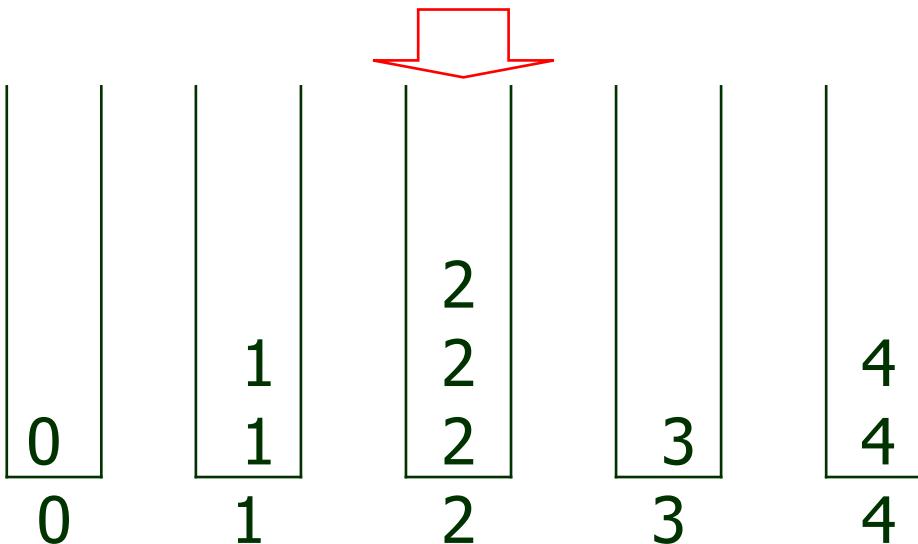
Bucket Sort

- One Value per bucket:



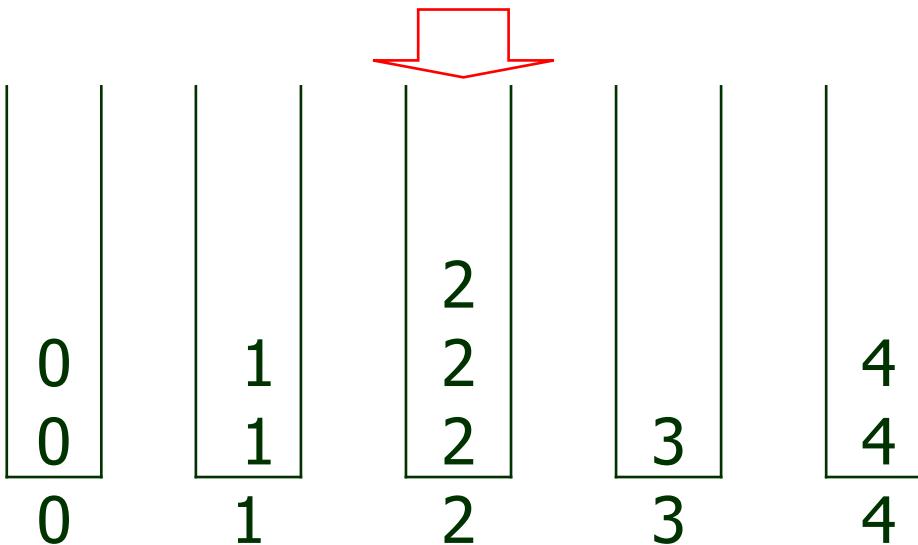
Bucket Sort

- One Value per bucket:



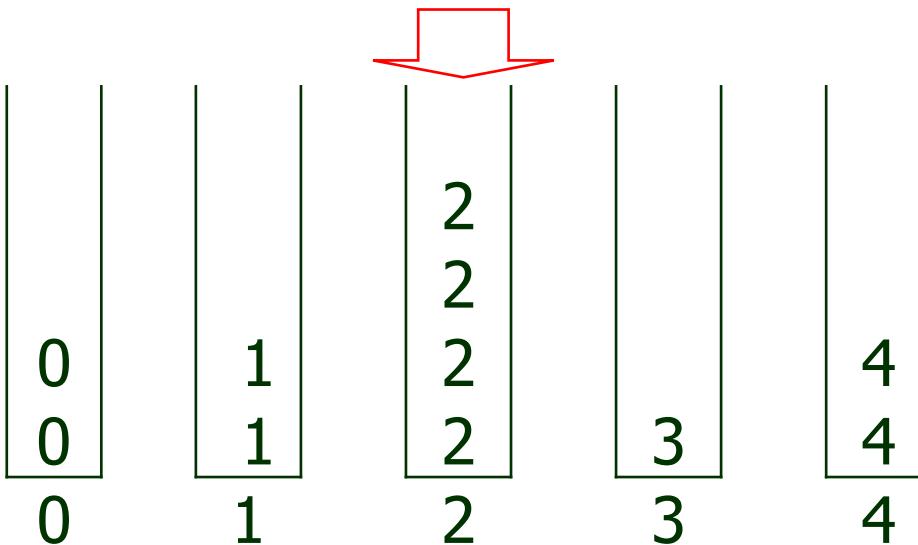
Bucket Sort

- One Value per bucket:



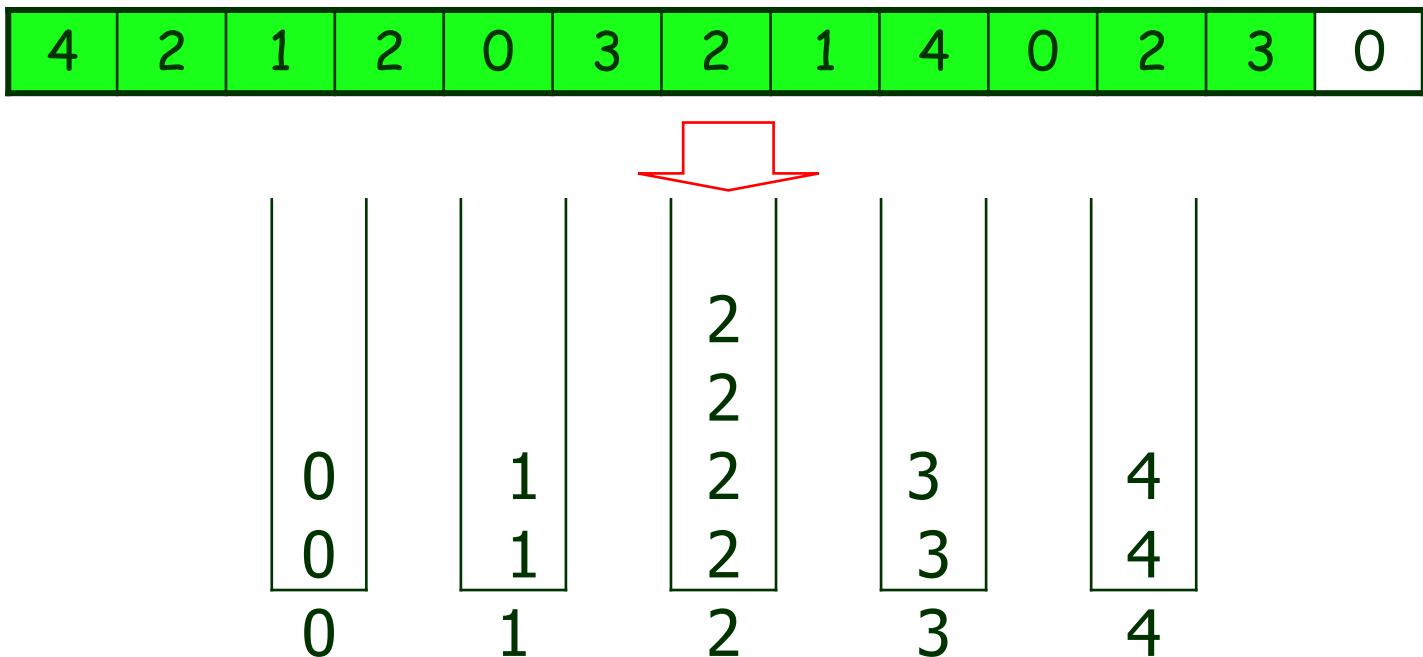
Bucket Sort

- One Value per bucket:



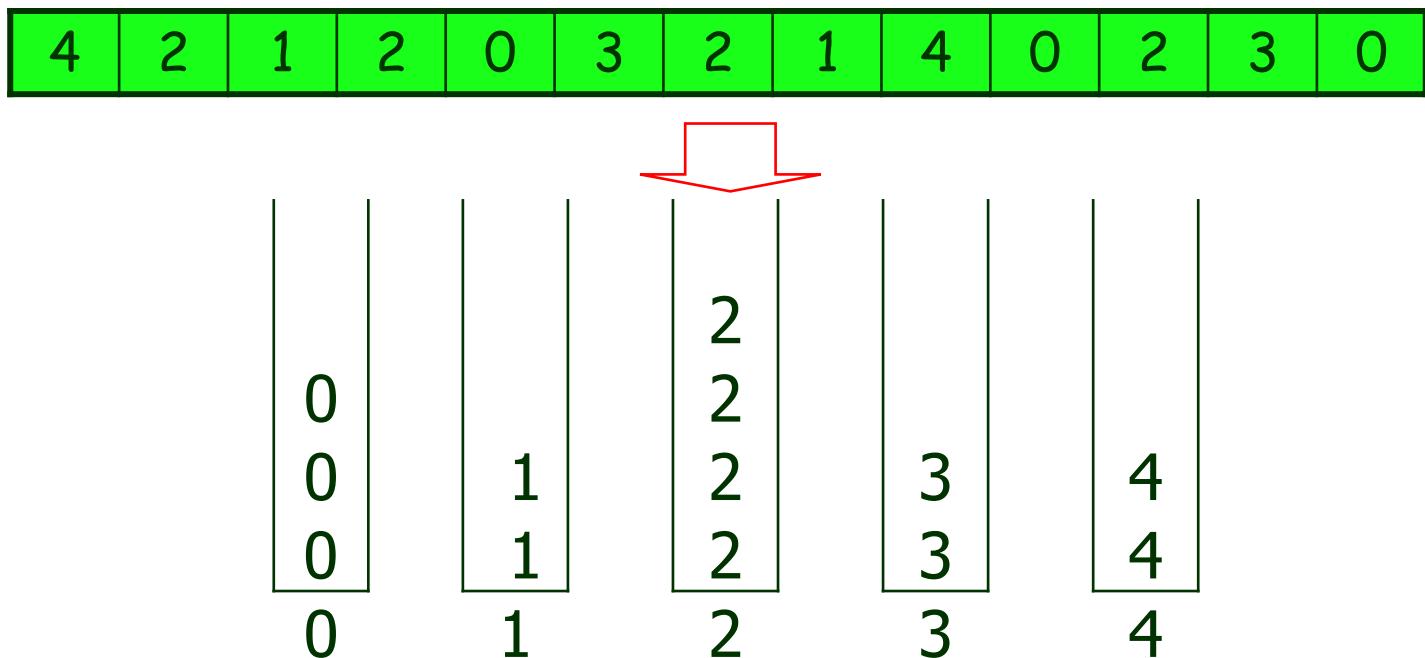
Bucket Sort

- One Value per bucket:



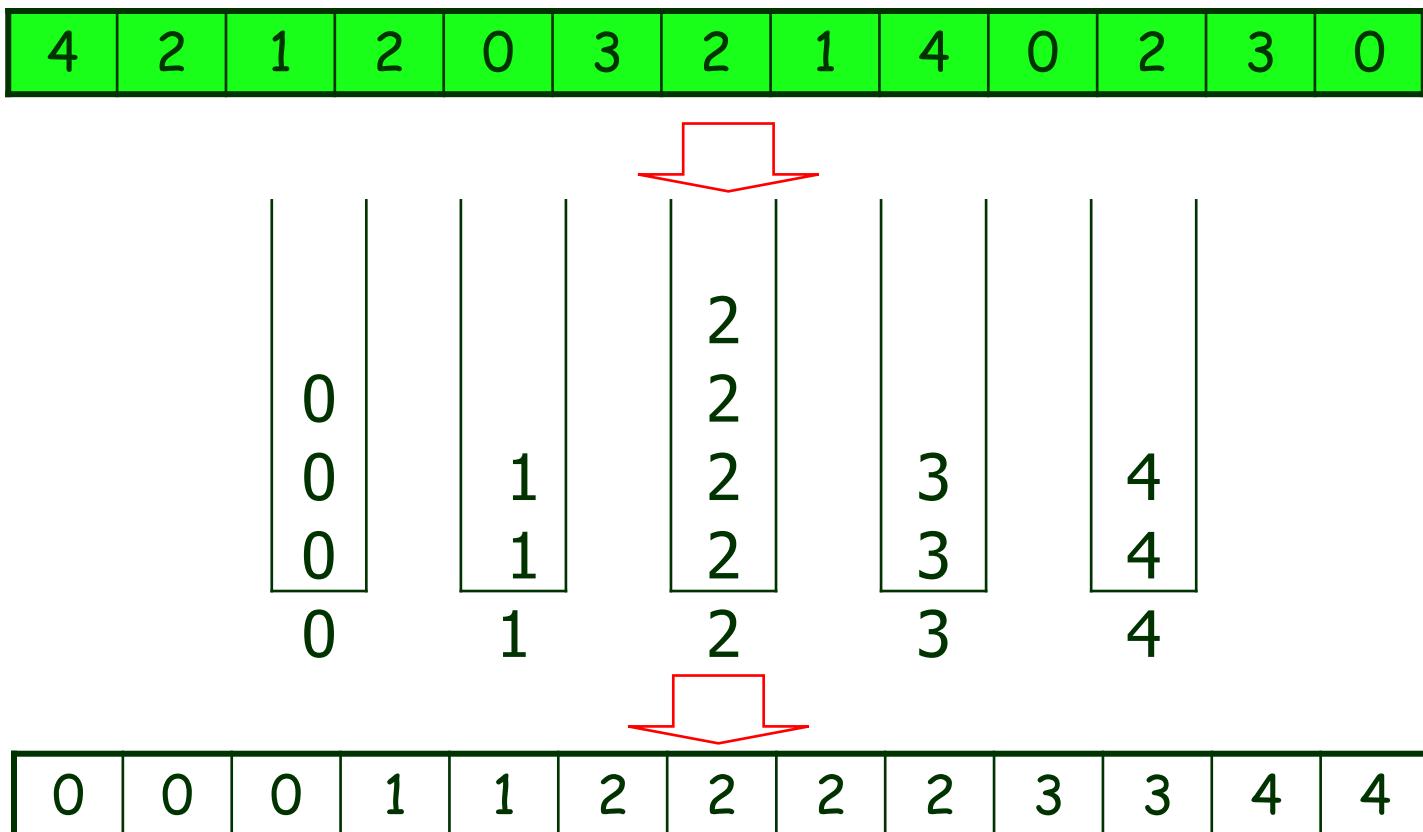
Bucket Sort

- One Value per bucket:



Bucket Sort

- One Value per bucket:



Bucket Sort

- One Value per bucket:



Bucket Sort

- One Value per bucket:

Algorithm BucketSort(S)

(values in S are between 0 and m-1)

for $j \leftarrow 0$ to $m-1$ do // initialize m buckets

$b[j] \leftarrow 0$

for $i \leftarrow 0$ to $n-1$ do // place elements in their

$b[S[i]] \leftarrow b[S[i]] + 1$ // appropriate buckets

$i \leftarrow 0$

for $j \leftarrow 0$ to $m-1$ do // place elements in buckets

 for $r \leftarrow 1$ to $b[j]$ do // back in S (Concatination)

$S[i] \leftarrow j$

$i \leftarrow i + 1$

Bucket Sort

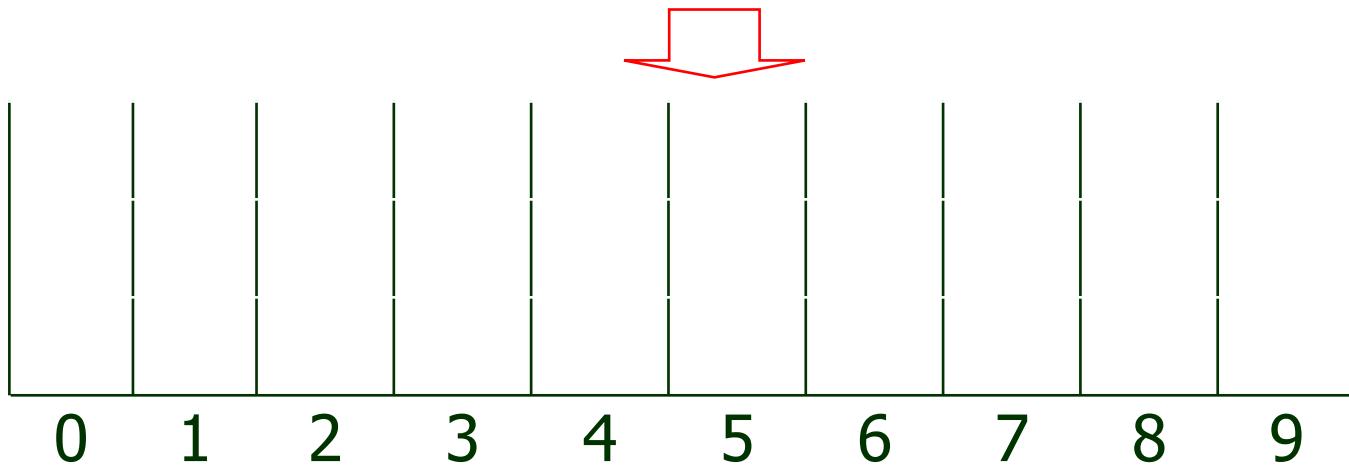
One Value per bucket (Analysis)

- Bucket initialization: $O(m)$
- From array to buckets: $O(n)$
- From buckets to array: $O(n)$
 - Due to the implementation of dequeue.
- Since m will likely be small compared to n , Bucket sort is $O(n)$
- Strictly speaking, time complexity is $O(n + m)$

Bucket Sort

- Multiple items per bucket:

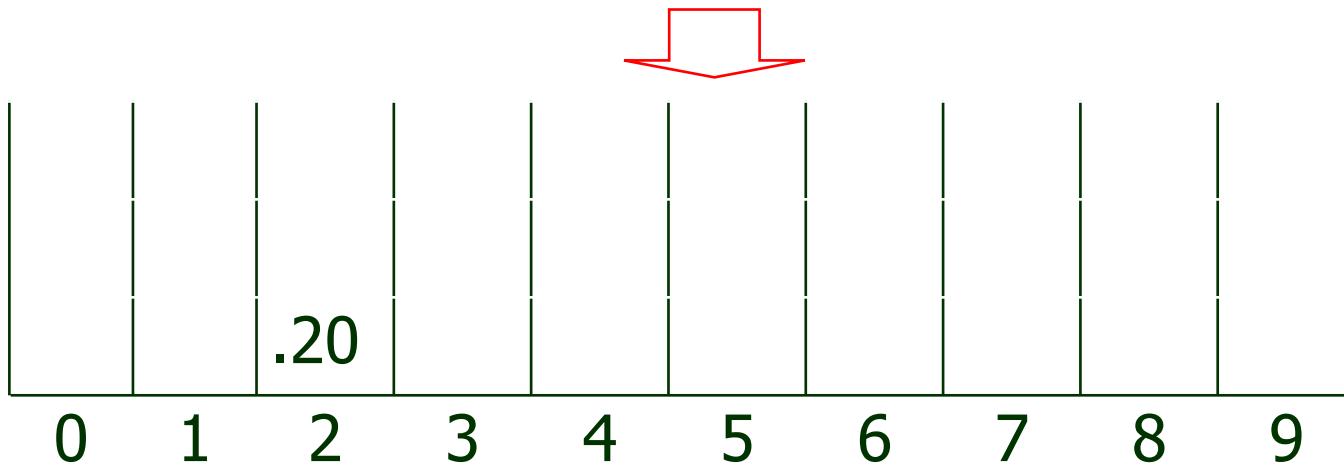
.20	.12	.58	.63	.64	.36	.37	.47	.52	.18	.88	.09	.99
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Bucket Sort

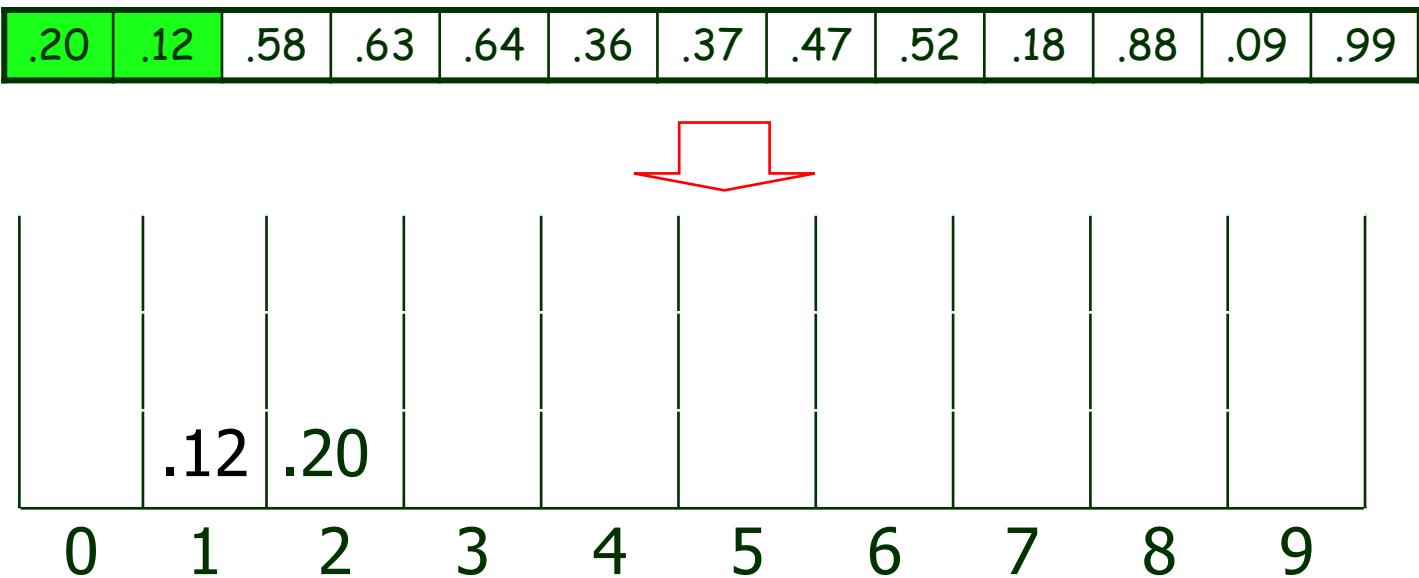
- Multiple items per bucket:

.20	.12	.58	.63	.64	.36	.37	.47	.52	.18	.88	.09	.99
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



Bucket Sort

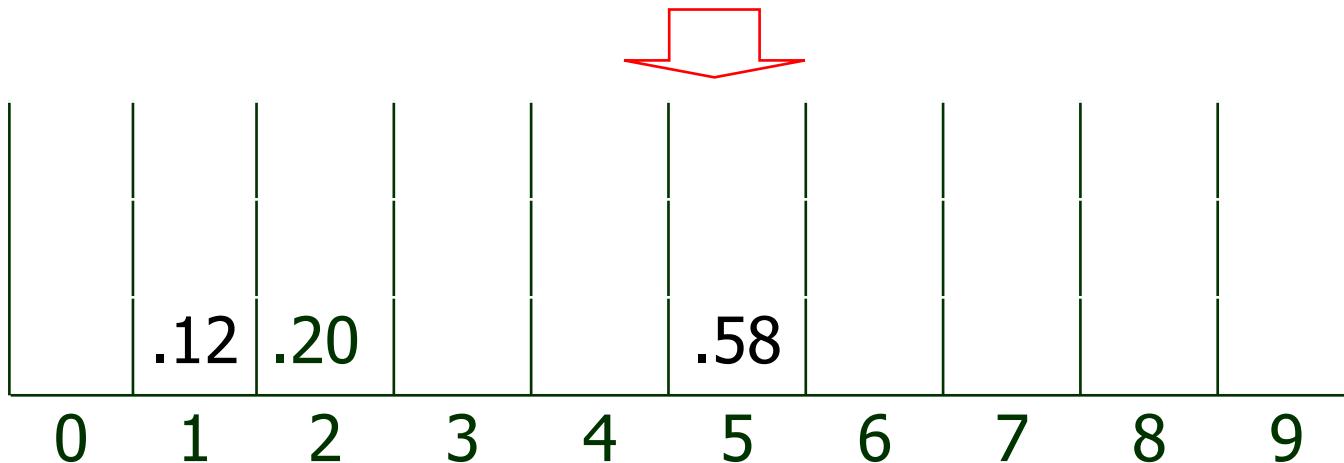
- Multiple items per bucket:



Bucket Sort

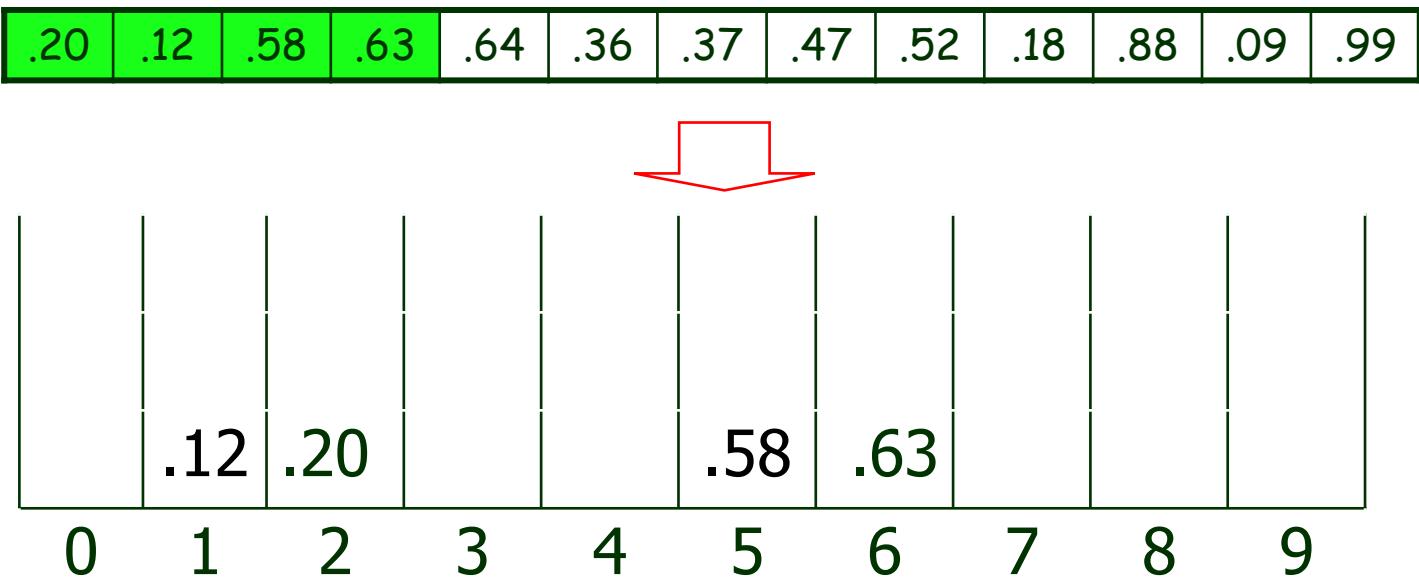
- Multiple items per bucket:

.20	.12	.58	.63	.64	.36	.37	.47	.52	.18	.88	.09	.99
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



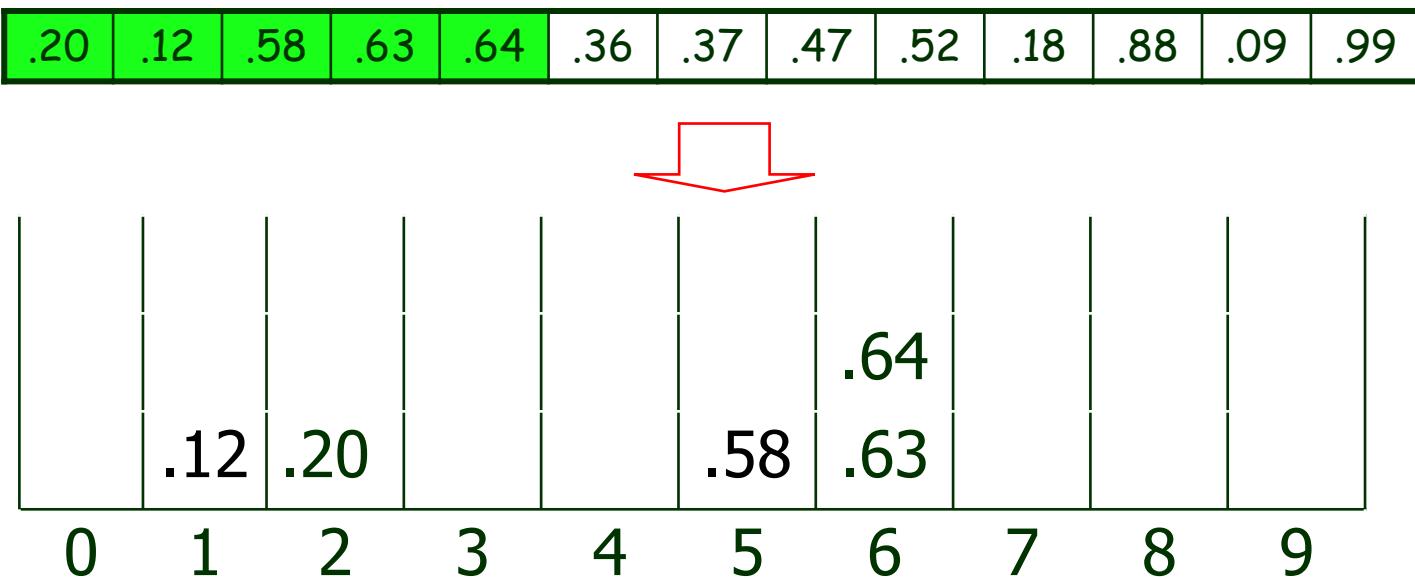
Bucket Sort

- Multiple items per bucket:



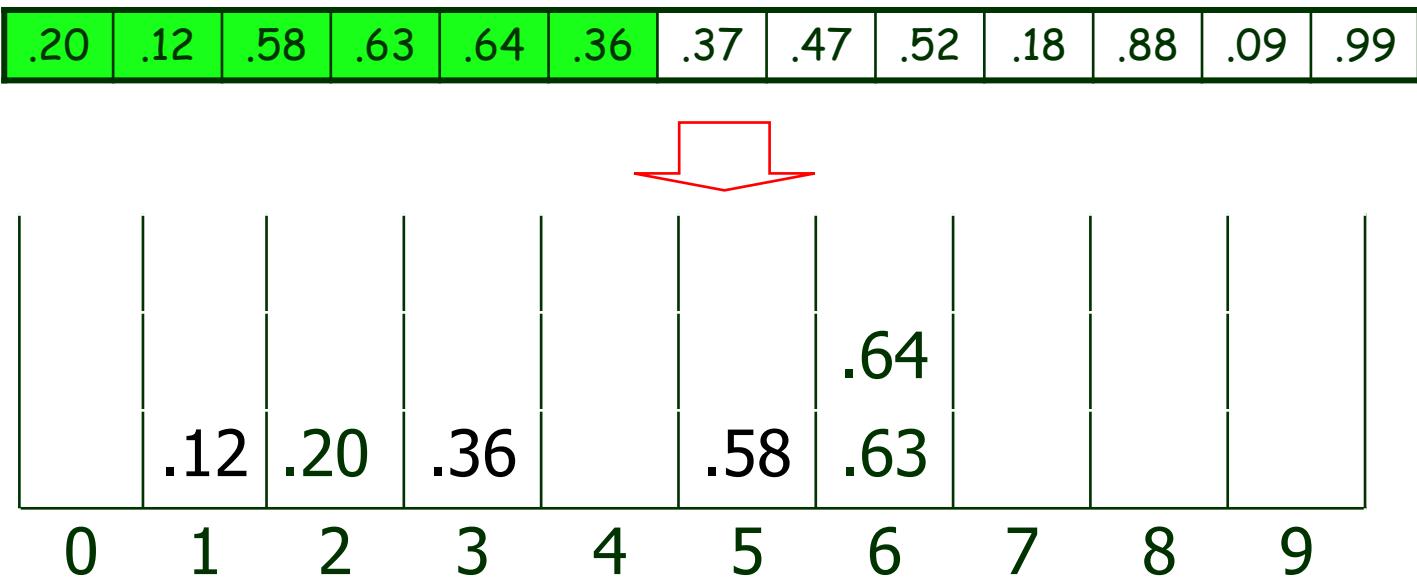
Bucket Sort

- Multiple items per bucket:



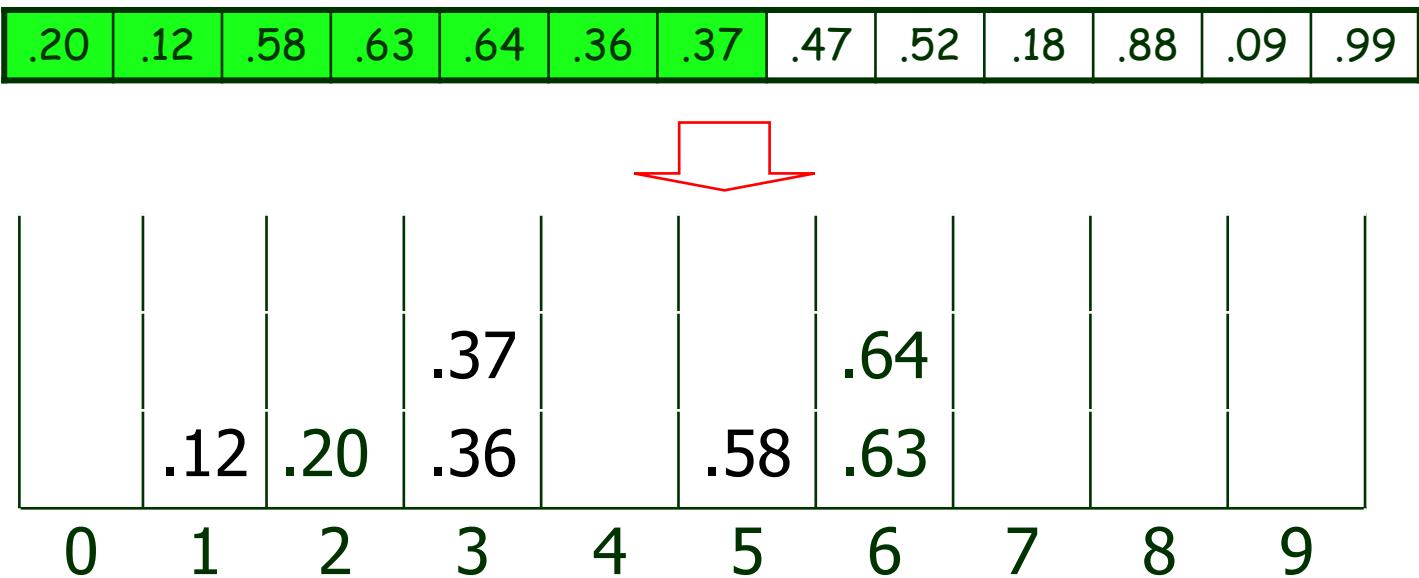
Bucket Sort

- Multiple items per bucket:



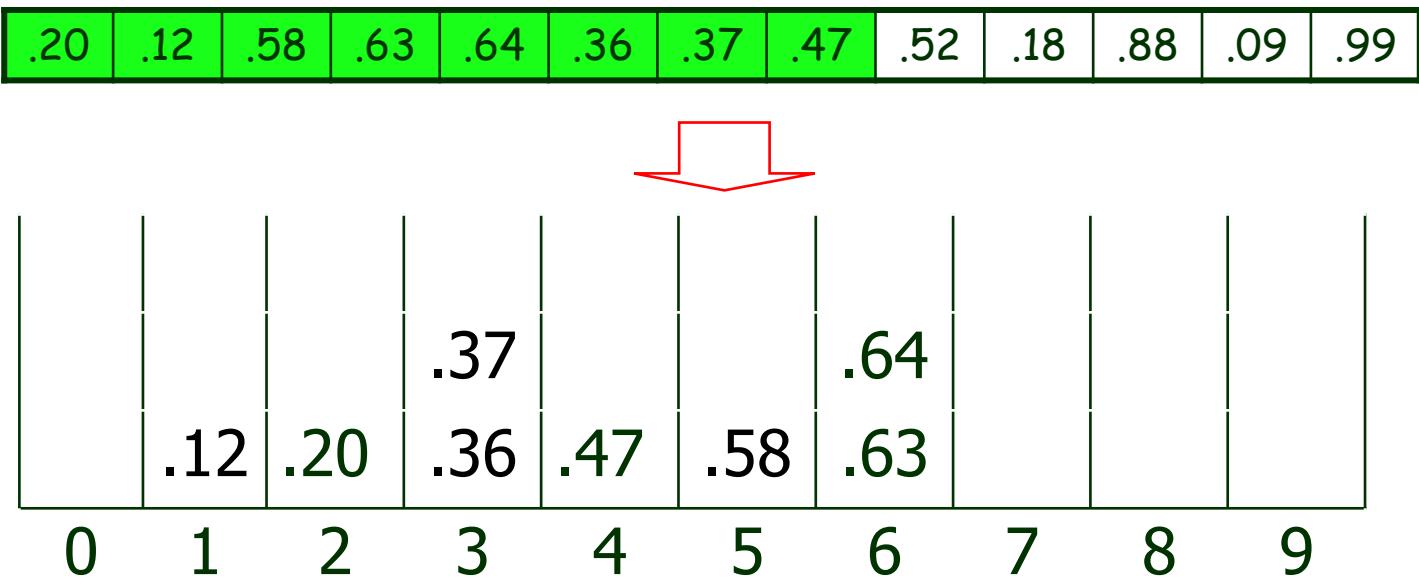
Bucket Sort

- Multiple items per bucket:



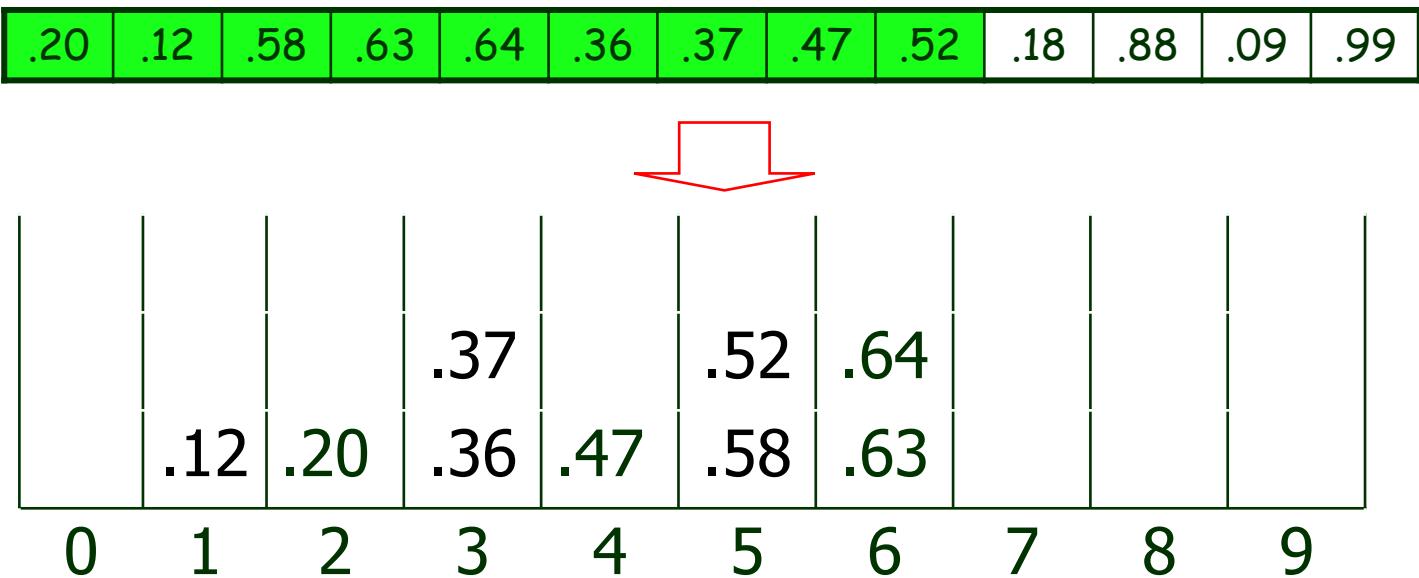
Bucket Sort

- Multiple items per bucket:



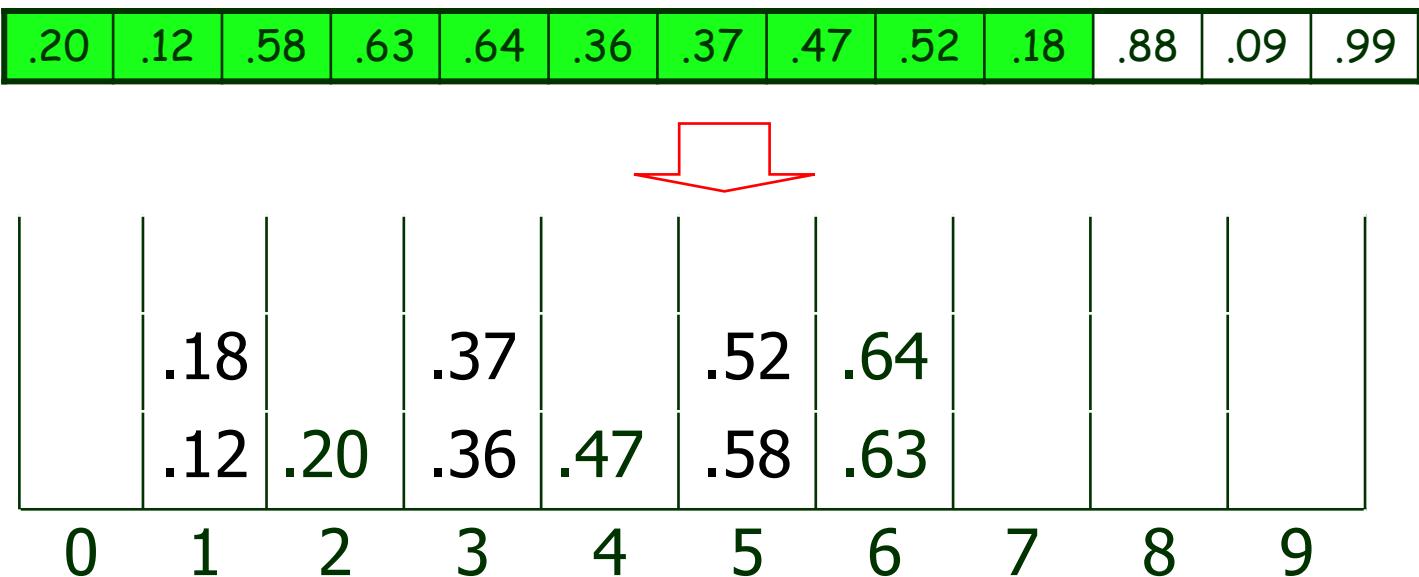
Bucket Sort

- Multiple items per bucket:



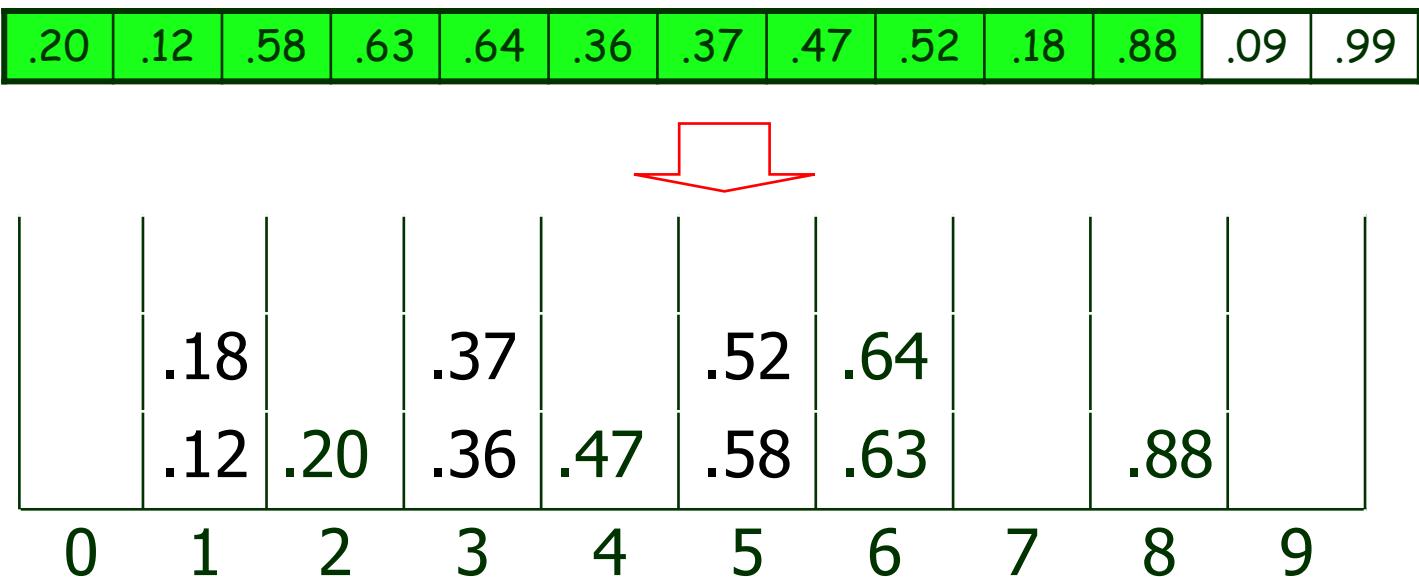
Bucket Sort

- Multiple items per bucket:



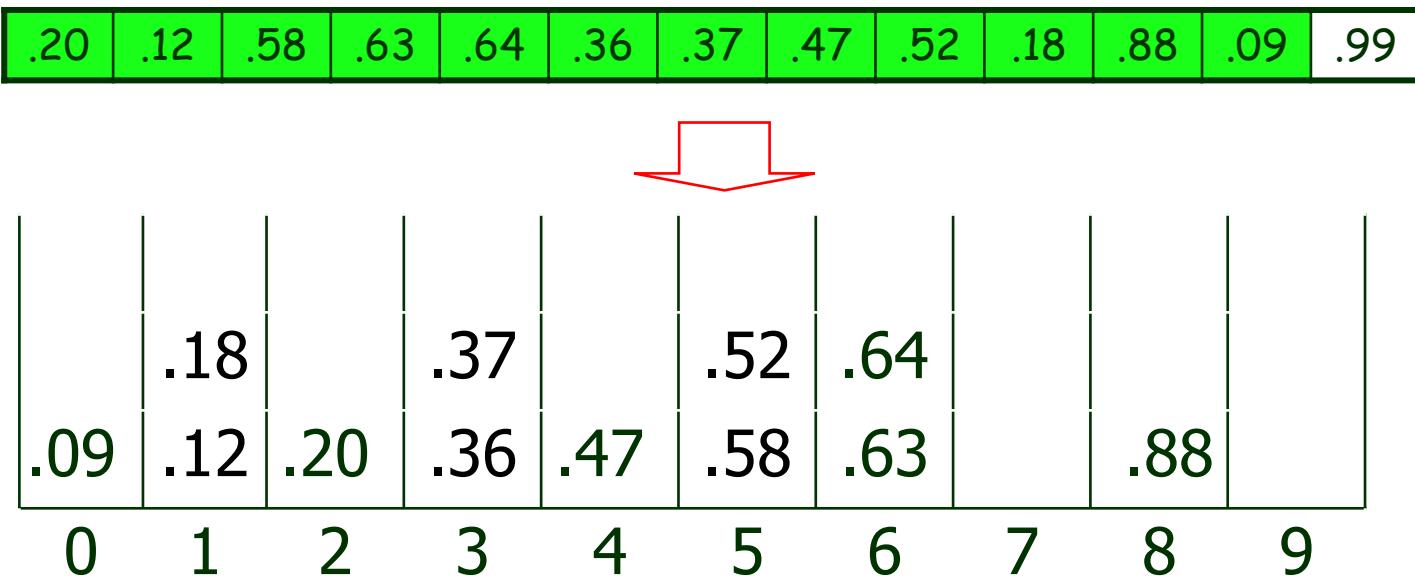
Bucket Sort

- Multiple items per bucket:



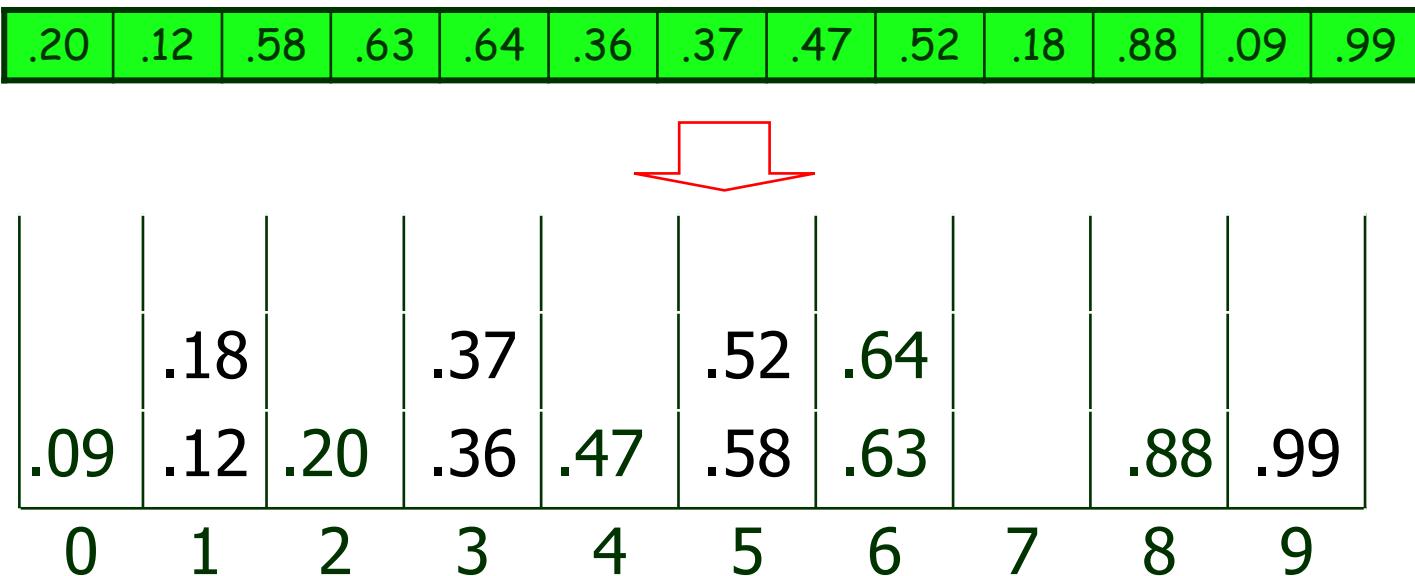
Bucket Sort

- Multiple items per bucket:



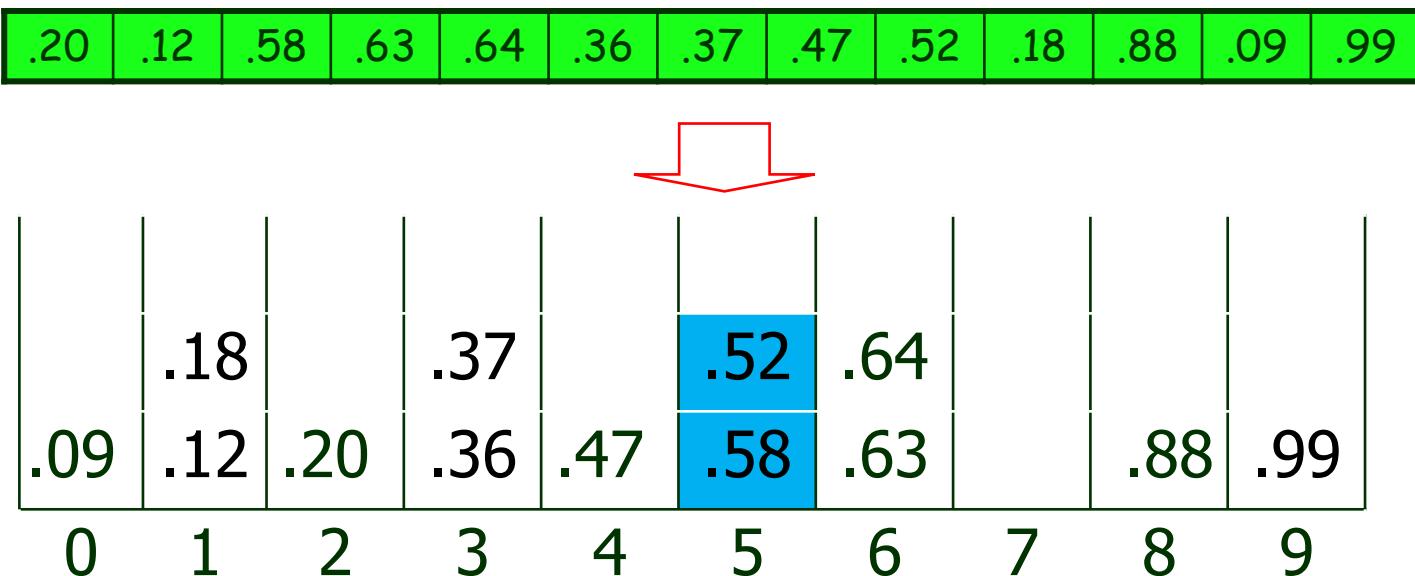
Bucket Sort

- Multiple items per bucket:



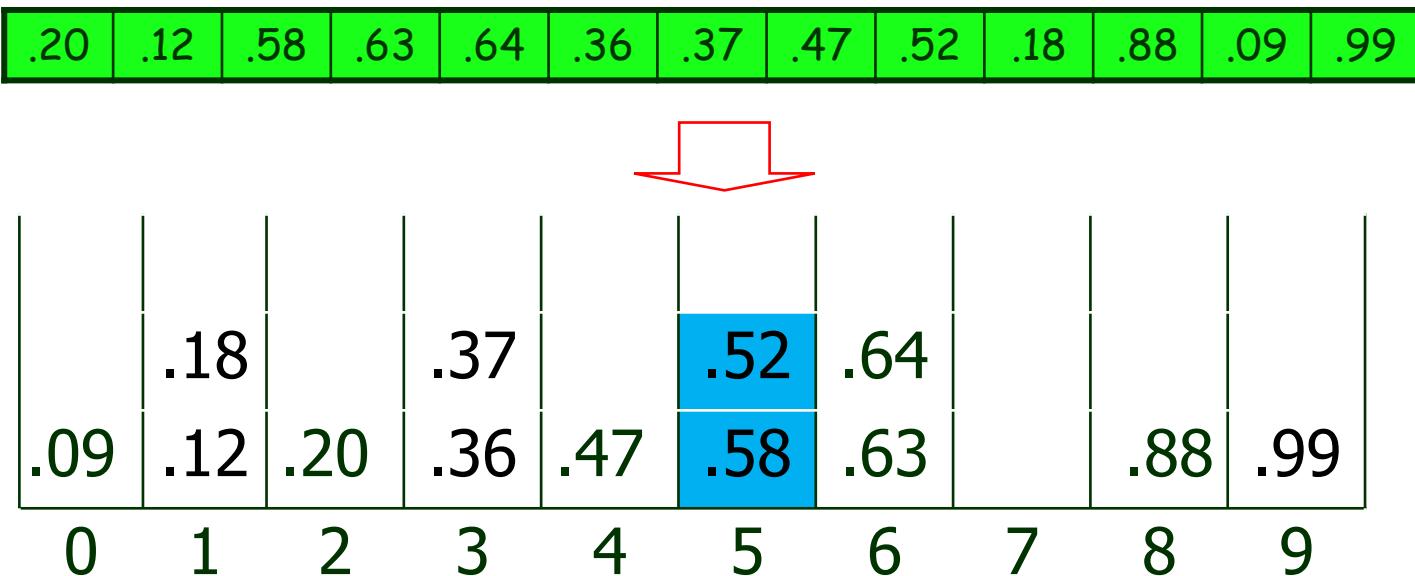
Bucket Sort

- Multiple items per bucket:



Bucket Sort

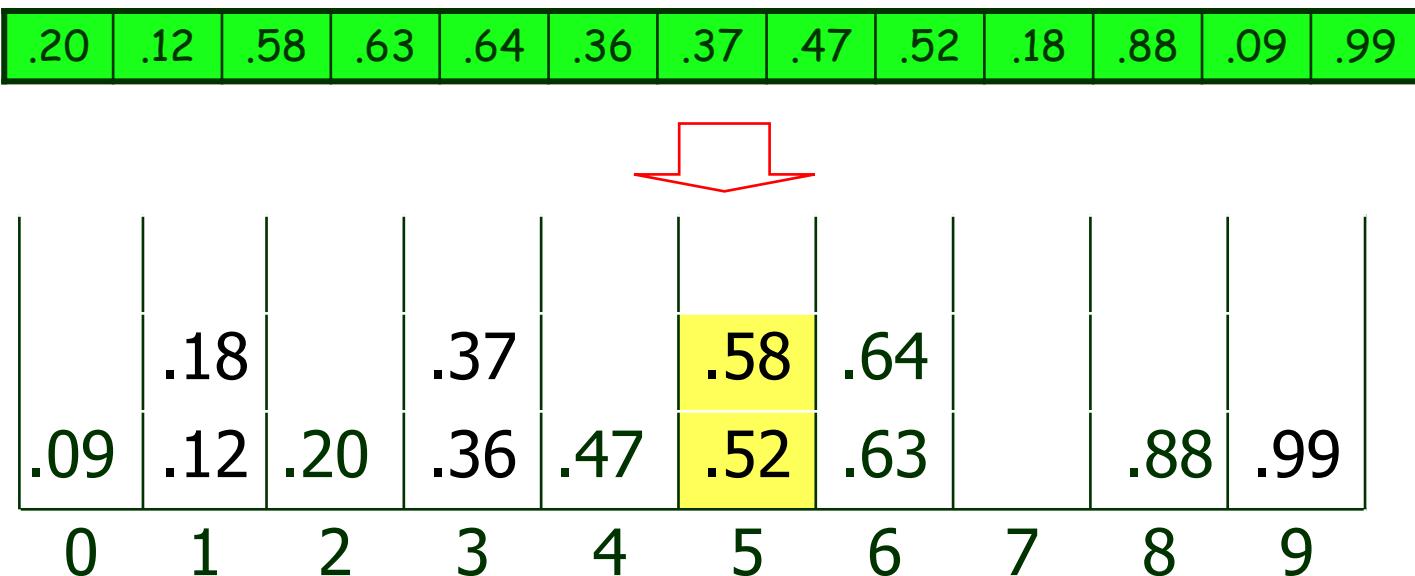
- Multiple items per bucket:



Apply Internal
sorting(stable)
on highlighted
data

Bucket Sort

- Multiple items per bucket:

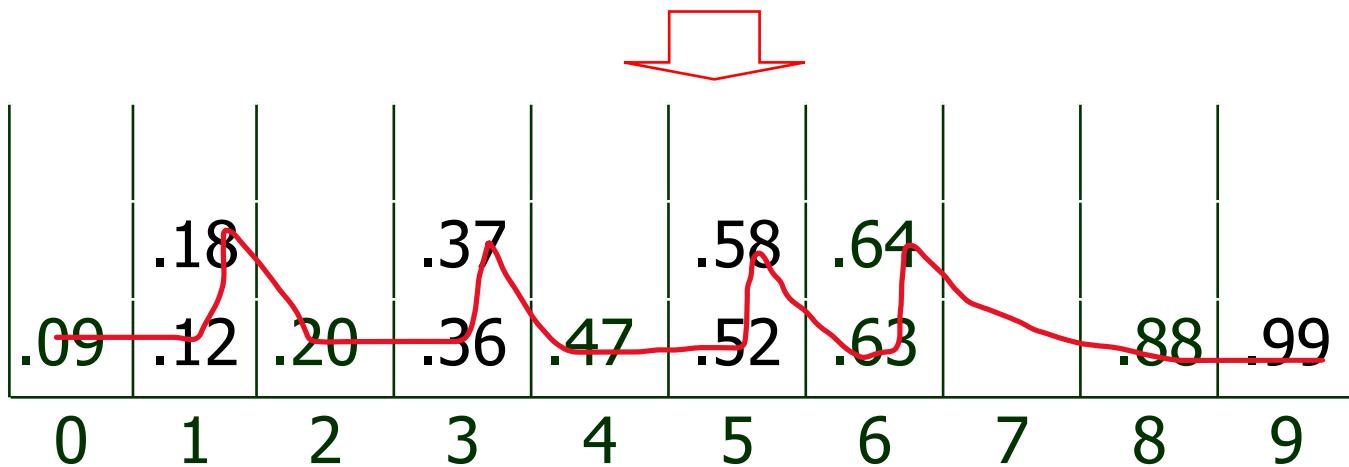


After Internal
sorting(stable)
on highlighted
data

Bucket Sort

- Multiple items per bucket:

.20	.12	.58	.63	.64	.36	.37	.47	.52	.18	.88	.09	.99
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----



.09	.12	.18	.20	.36	.37	.47	.52	.58	.63	.64	.88	.99
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Bucket Sort

- Multiple items per bucket:

Algorithm BucketSort(S)

1. Let $B[0..(n - 1)]$ be a new array.
2. $n \leftarrow A.length$
3. for $i \leftarrow 0$ to $n - 1$
4. make $B[i]$ an empty list
5. for $i \leftarrow 1$ to n
6. insert $A[i]$ into list $B[n * A[i]]$
7. for $i \leftarrow 0$ to $n - 1$
8. sort list $B[i]$ with a stable sorting(insertion sort)
9. Concatenate the list $B[0], B[1], B[2], \dots, B[n - 1]$ together in order.

Bucket Sort

- Multiple items per bucket:

Algorithm BucketSort(S)

1. Let $B[0..(n - 1)]$ be a new array. $O(1)$
2. $n \leftarrow A.length$ $O(1)$
3. for $i \leftarrow 0$ to $n - 1$
4. make $B[i]$ an empty list } $O(n)$
5. for $i \leftarrow 1$ to n
6. insert $A[i]$ into list $B[n * A[i]]$ } $O(n)$
7. for $i \leftarrow 0$ to $n - 1$
8. sort list $B[i]$ with a stable sorting(insertion sort) } $O(n^2)$
9. Concatenate the list $B[0], B[1], B[2], \dots, B[n - 1]$
together in order. } $O(n)$

if all the
elements
belongs to
one bucket.

Bucket Sort

Multiple items per bucket (Analysis)

- It was observed that except line no 8 all other lines take $O(n)$ time in worst case.
- Line no. 8 (i.e. insertion sort) takes $O(n^2)$, if all the elements belongs to one bucket.
- The average time complexity for Bucket Sort is $O(n + k)$ in uniform distribution of data.

Bucket Sort

Characteristics of Bucket Sort

- Bucket sort assumes that the input is drawn from a uniform distribution.
- The computational complexity estimates involve the number of buckets.
- Bucket sort can be exceptionally fast because of the way elements are assigned to buckets, typically using an array where the index is the value.

Bucket Sort

Characteristics of Bucket Sort

- This means that more auxiliary memory is required for the buckets at the cost of running time than more comparison sorts.
- The average time complexity is $O(n + k)$.
- The worst time complexity is $O(n^2)$.
- The space complexity for Bucket Sort is $O(n + k)$.

Thank U

Design and Analysis of Algorithm

Linear Time Sorting (Shell Sort)

Lecture -20

Overview

- Running time of Shell sort in worst case is $O(n^2)$ or float between $O(n \log n)$ and $O(n^2)$.
- Running time of Shell sort in best case is $O(n \lg n)$. And $O(n)$ if the total number of comparisons for each interval (or increment) is equal to the size of the array.
- Is not a stable sorting.

Shell Sort

- Designed by Donald Shell and named the sorting algorithm after himself in 1959.
- Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared.
- Shell sort is also known as **diminishing increment sort**.

Shell Sort

- Shell sort improves on the efficiency of **insertion sort** by quickly shifting values to their destination.
- This algorithm tries to decreases the distance between comparisons (i.e. gap) as the sorting algorithm runs and reach to its last phase where, the adjacent elements are compared only.

Shell Sort

- The distance of comparisons (i.e. gap) is maintained by the following methods:
 - divide by 2(Two) [Designed by Donald Shell)
 - Knuth Method(*i.e. gap* \leftarrow *gap* * 3 + 1)
(initially the gap starts with 1)

Shell Sort

- Let's execute an example with the help of Knuth's gap method on the following array.

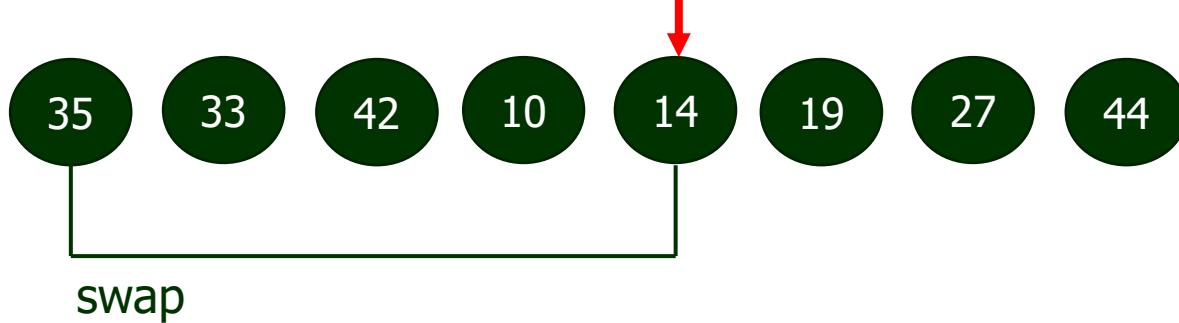


- At the beginning the gap is initialized as 1
- Hence the new gap value for iteration 1 is calculated as follows:

$$\begin{aligned} \textit{gap} &= \textit{gap} * 3 + 1 \\ &= 1 * 3 + 1 = 4 \end{aligned}$$

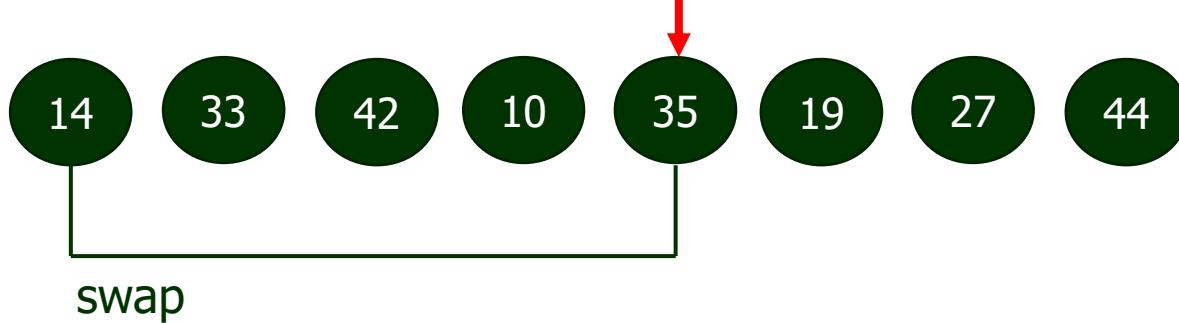
Shell Sort

Swap count =0



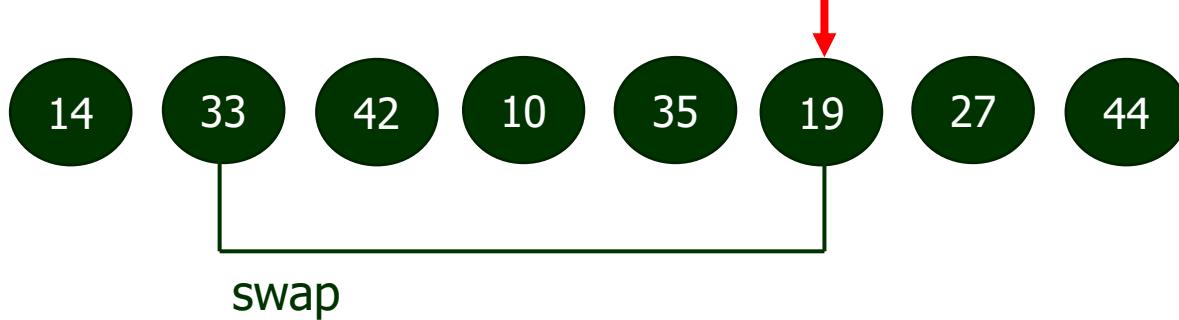
Shell Sort

Swap count = 1



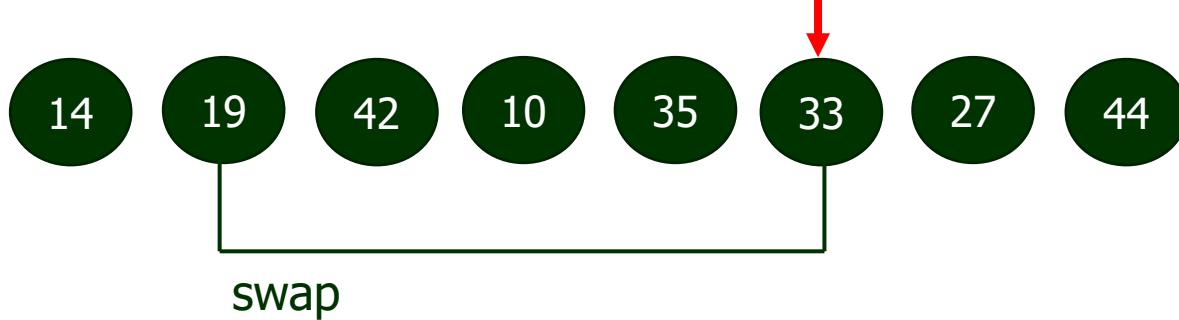
Shell Sort

Swap count = 1



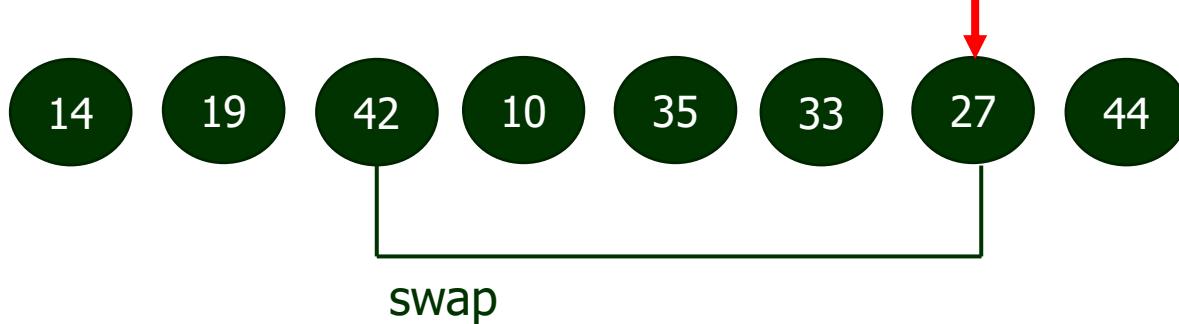
Shell Sort

Swap count =2



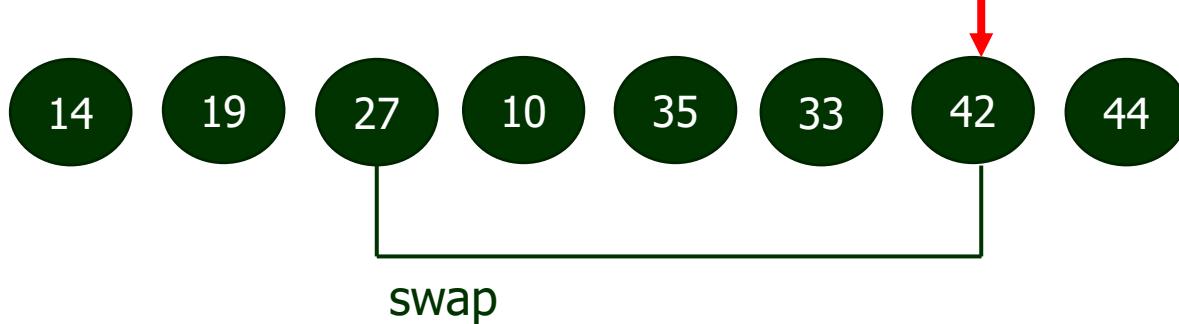
Shell Sort

Swap count =2



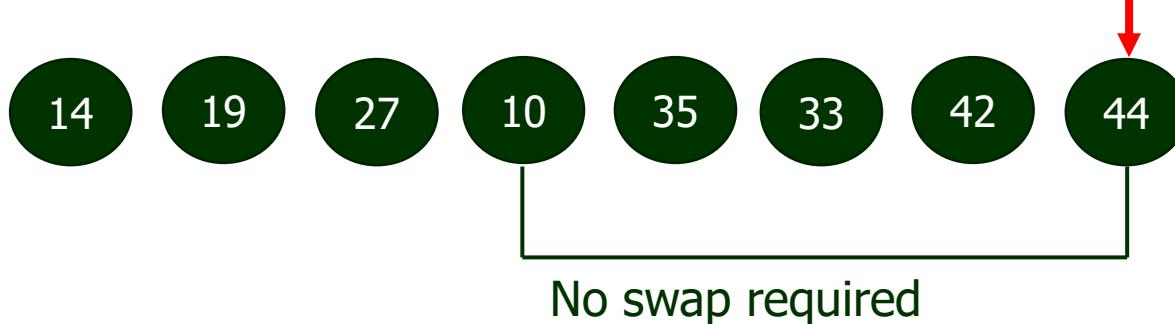
Shell Sort

Swap count =3



Shell Sort

Swap count =3



Shell Sort

- After the first iteration the array looks like as follows.



- Again we finding the gap value for nest iteration.

$$gap = gap * 3 + 1$$

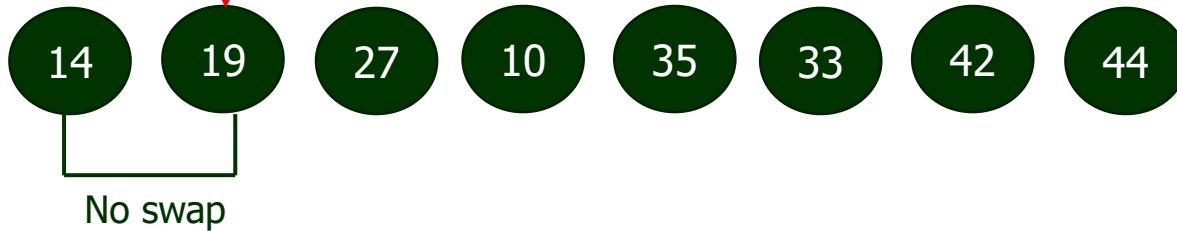
- We can write the above equation as follows:

$$gap = \frac{gap - 1}{3}$$

- So, the new gap is $gap = \frac{gap - 1}{3} = \frac{4 - 1}{3} = 1$

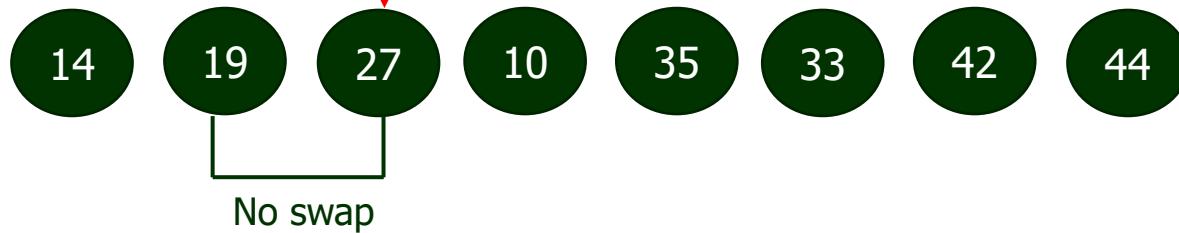
Shell Sort

Swap count =3



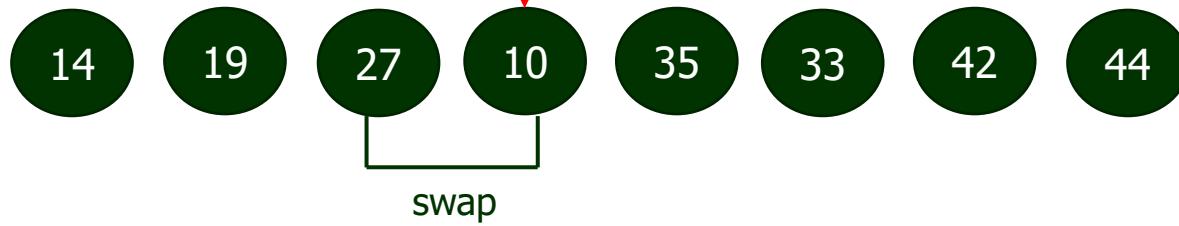
Shell Sort

Swap count =3



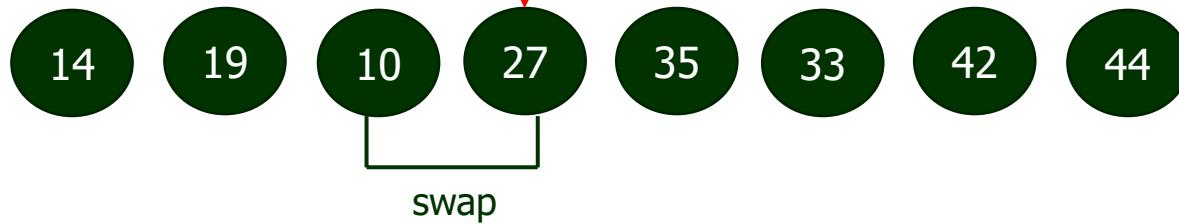
Shell Sort

Swap count =3



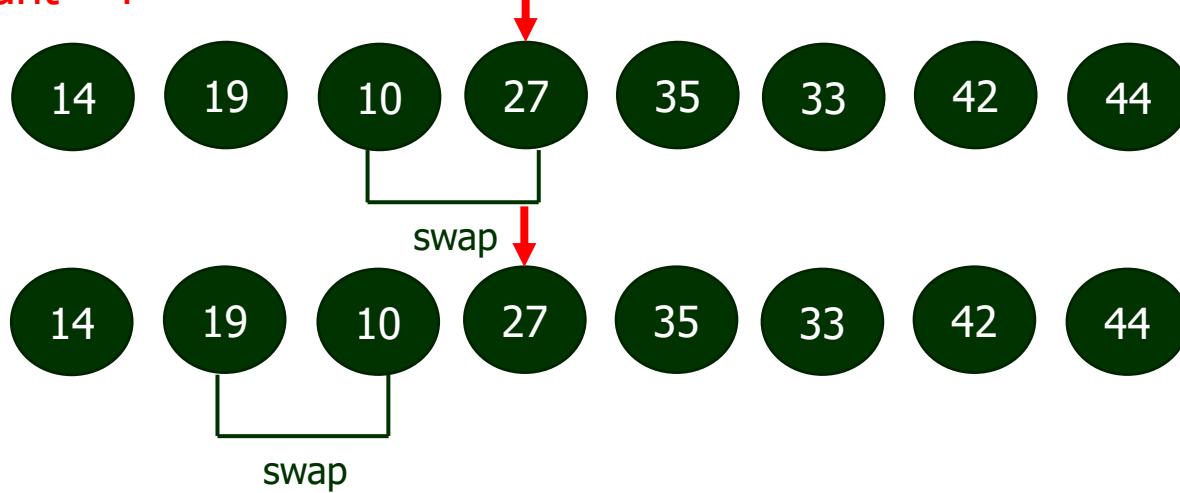
Shell Sort

Swap count =4



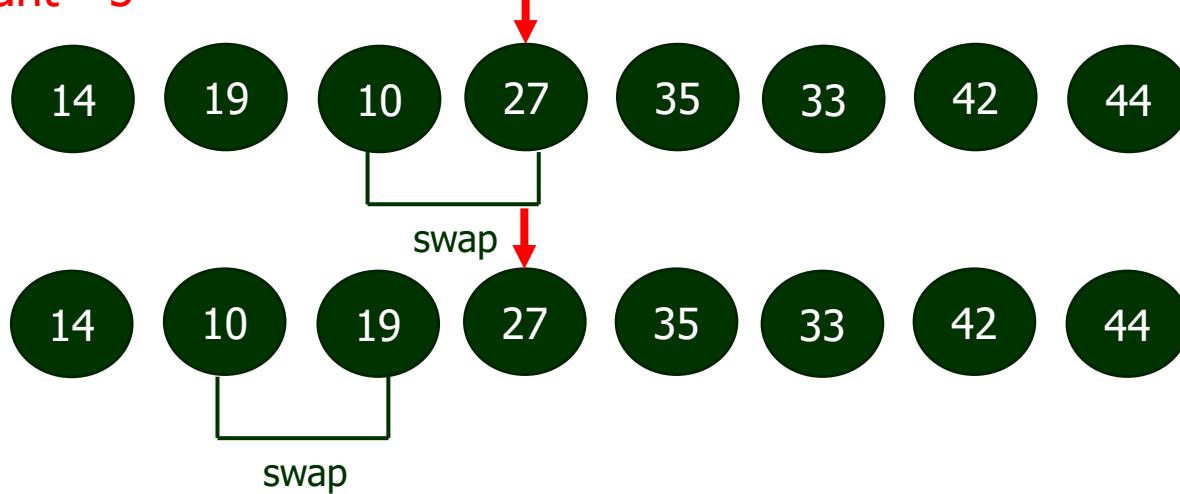
Shell Sort

Swap count =4



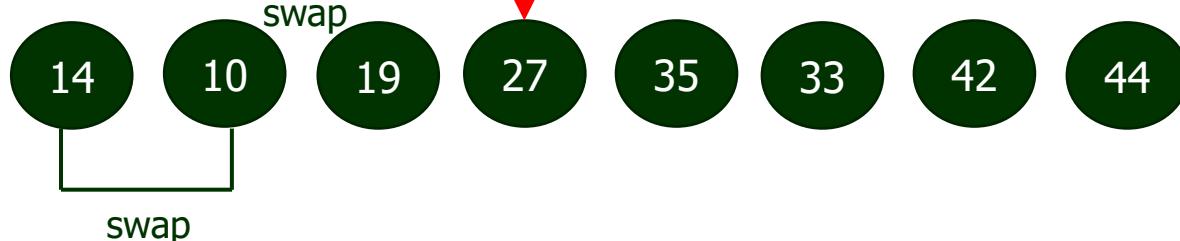
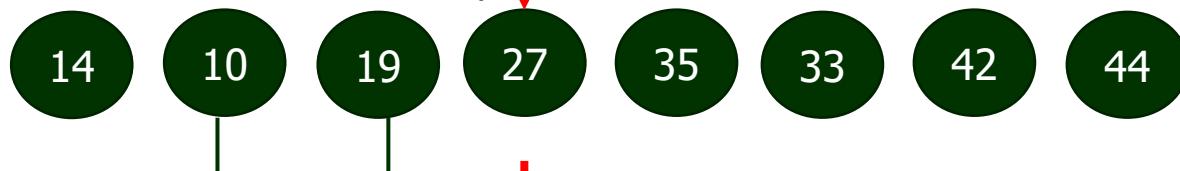
Shell Sort

Swap count = 5



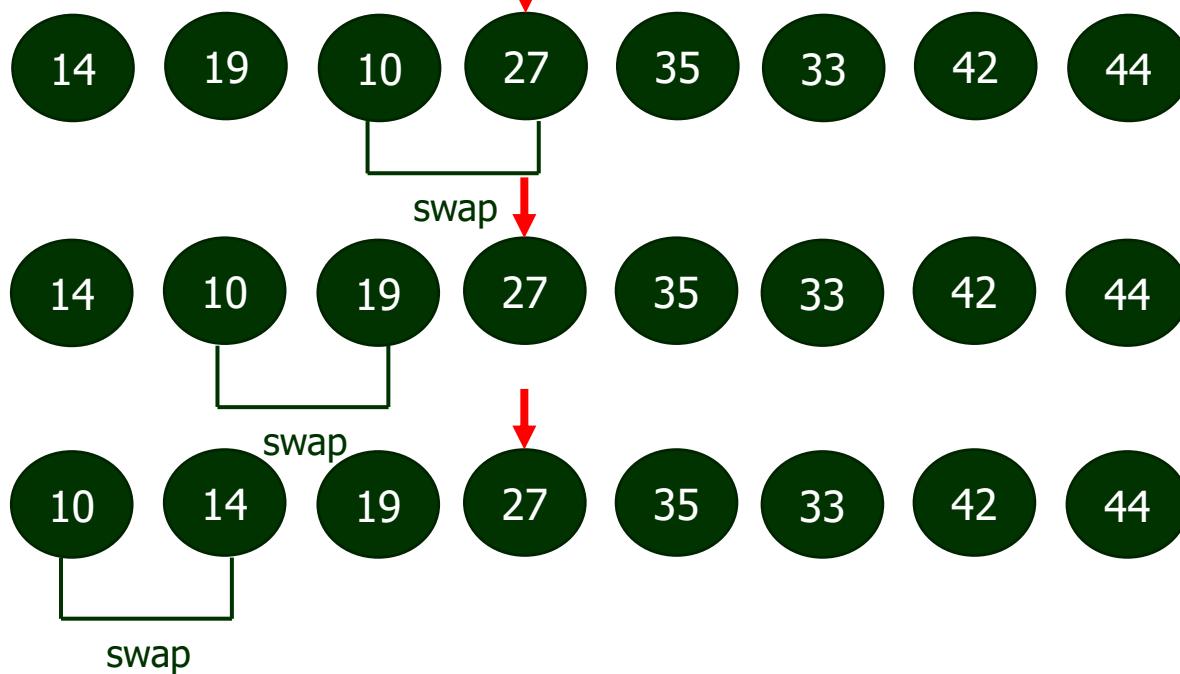
Shell Sort

Swap count = 5



Shell Sort

Swap count =6



Shell Sort

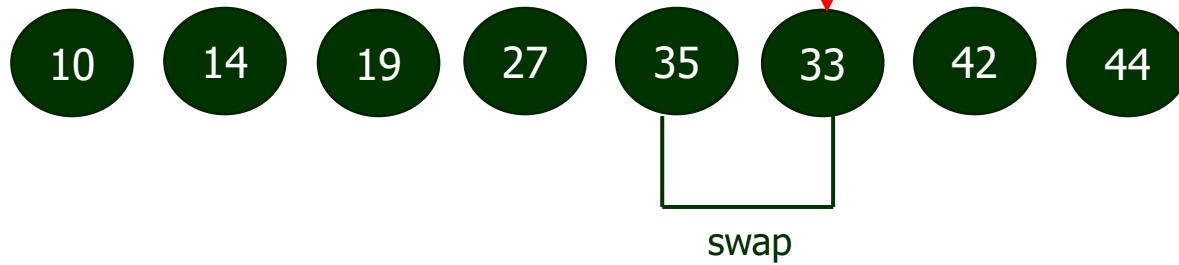
Swap count =6



No swap

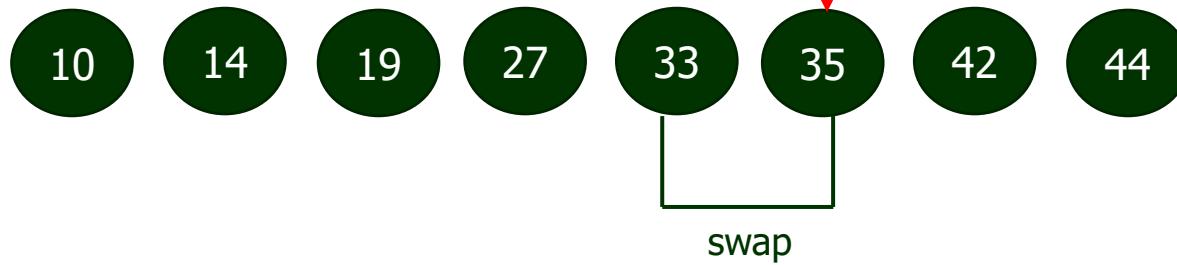
Shell Sort

Swap count =6



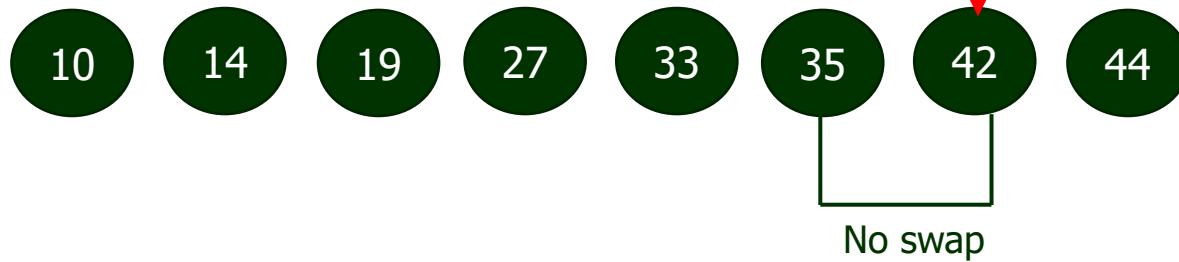
Shell Sort

Swap count = 7



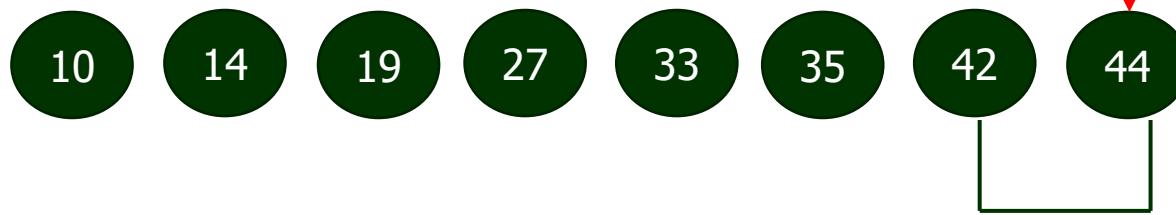
Shell Sort

Swap count = 7



Shell Sort

Swap count = 7



Shell Sort

Swap count =7



- Hence ,total number of swap required in
 - 1st iteration= 3
 - 2nd iteration= 4
- So total 7 numbers of swap required to sort the array by shell sort.

Shell Sort

Algorithm Shell sort (Knuth Method)

```
1. gap=1
2. while(gap < A.length/3)
3.   [ gap=gap*3+1
4.   while( gap>0)
5.     for(outer=gap; outer<A.length; outer++)
6.       [ Ins_value=A[outer]
7.         inner=outer
8.         while(inner>gap-1 && A[inner-gap]≥ Ins_value)
9.           [ A[inner]=A[inner-gap]
10.             inner=inner-gap
11.             A[inner]=Ins_value
12.           ] gap=(gap-1)/3
```

Shell Sort

- Let us dry run the shell sort algorithm with the same example as already discussed.



At the beginning

A.length=8 and gap=1

After first three line execution the gap value changed to 4

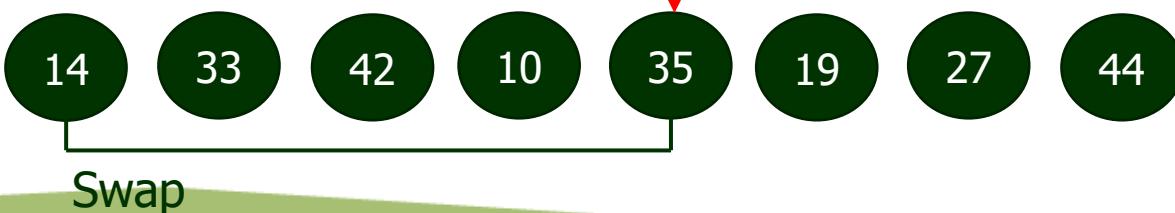
Now, gap>0 (i.e. 4>0)

Now in for loop outer=4;outer<8;outer++

Ins_value=A[outer]=A[4]=14

inner=outer i.e. inner=4

Now the line no 8 is true \Rightarrow change occurred and the updated array is looked as follow



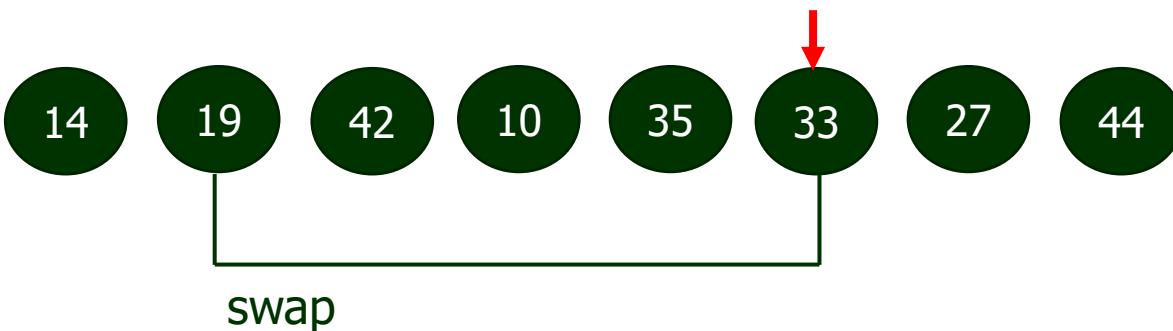
Shell Sort

Now in for loop outer=5 ;outer<8; outer++

Ins_value=A[outer]=A[5]=19

inner=outer i.e. inner=5

Now the line no 8 is true \Rightarrow *change occurred* and the updated array is looked as follow



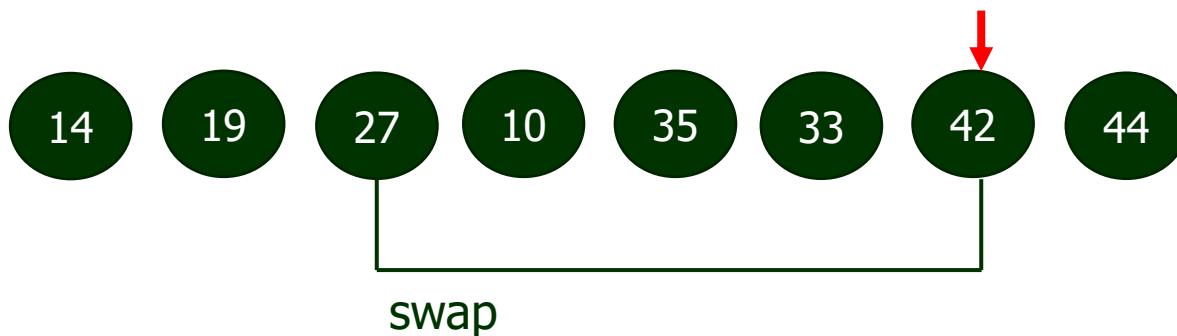
Shell Sort

Now in for loop outer=6 ;outer<8; outer++

Ins_value=A[outer]=A[6]=27

inner=outer i.e. inner=6

Now the line no 8 is true \Rightarrow *change occurred* and the updated array is looked as follow



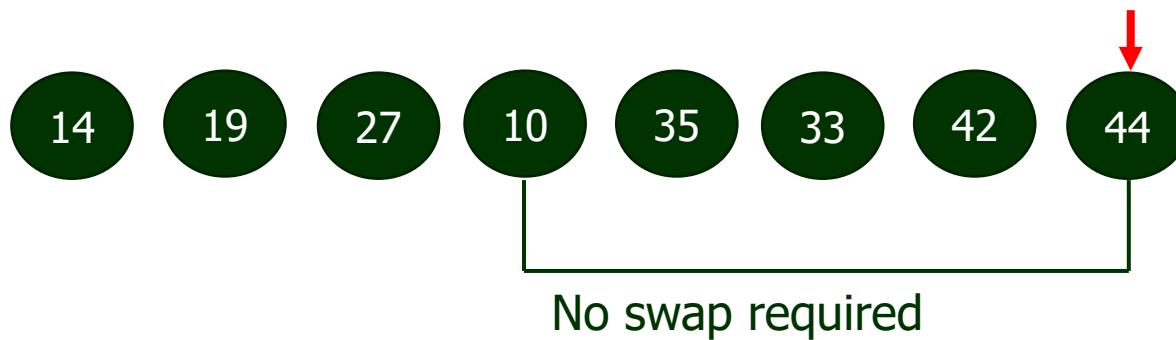
Shell Sort

Now in for loop outer=7 ;outer<8; outer++

Ins_value=A[outer]=A[7]=44

inner=outer i.e. inner=7

Now the line no 8 is **False** \Rightarrow no change in array and the updated array is looked as follow



Shell Sort

- Now again gap value will be calculated .
- The new gap value is 1. And again the same procedure will be continued .

Shell Sort

- Now again gap value will be calculated .
- The new gap value is 1. And again the same procedure will be continued .



Home
Assignment

Shell Sort

- Now again gap value will be calculated .
- The new gap value is 1. And again the same procedure will be continued . And finally the sorted array looks as given below with 7(seven) number of swap



Shell Sort

Analysis:

- Shell sort is efficient for medium size lists.
- For bigger list, this algorithm is not the best choice.
- But it is the fastest of all $O(n^2)$ sorting algorithm.
- The best case in shell sort is when the array is already sorted in the right order i.e. $O(n)$
- The worst case time complexity is based on the gap sequence. That's why various scientist give their gap intervals. They are:
 1. Donald Shell give the gap interval $\frac{n}{2}$. $\Rightarrow O(n \log n)$
 2. Knuth give the gap interval $gap \leftarrow gap * 3 + 1$ $\Rightarrow O(n^{3/2})$
 3. Hibbard give the gap interval 2^{k-1} $\Rightarrow O(n \log n)$

Shell Sort

Analysis:

In General

- Shell sort is an unstable sorting algorithm because this algorithm does not examine the elements lying in between the intervals.
- **Worst Case Complexity:** less than or equal to $O(n^2)$ or float between $O(n \log n)$ and $O(n^2)$.
- **Best Case Complexity:** $O(n \log n)$
When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to $O(n)$ i.e. the size of the array.
- Average Case Complexity: $O(n \log n)$
It is around $O(n^{1.25})$.

Shell Sort

Analysis:

In General

- Shell sort is an unstable sorting algorithm because this algorithm does not examine the elements lying in between the intervals.
- **Worst Case Complexity:** less than or equal to $O(n^2)$ or float between $O(n \log n)$ and $O(n^2)$.
- **Best Case Complexity:** $O(n \log n)$
When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to $O(n)$ i.e. the size of the array.
- Average Case Complexity: $O(n \log n)$
It is around $O(n^{1.25})$.

(Remark: Accurate model not yet been discovered)

Thank U

Design and Analysis of Algorithm

Divide and Conquer strategy (Merge Sort)

Lecture -21

Overview

- Learn the technique of “divide and conquer” in the context of merge sort with analysis.

A Sorting Problem (Divide and Conquer Approach)

- **Divide** the problem into a number of sub problems.
- **Conquer** the sub problems by solving them recursively.
 - ***Base case:*** If the sub problems are small enough, just solve them by brute force.
- **Combine** the sub problem solutions to give a solution to the original problem.

Merge sort

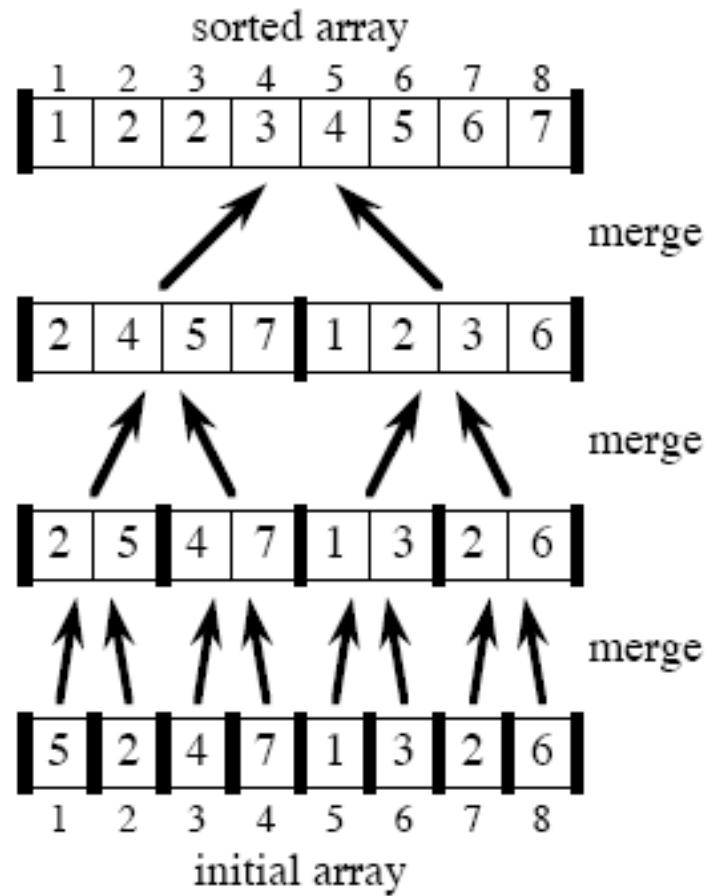
- A sorting algorithm based on divide and conquer. Its worst-case running time has a lower order of growth than insertion sort.
- Because we are dealing with sub problems, we state each sub problem as sorting a sub array $A[p \dots r]$.
- Initially, $p = 1$ and $r = n$, but these values change as we recurse through sub problems.

To sort $A[p \dots r]$:

- **Divide** by splitting into two sub arrays $A[p \dots q]$ and $A[q + 1 \dots r]$, where q is the halfway point of $A[p \dots r]$.
- **Conquer** by recursively sorting the two sub arrays $A[p \dots q]$ and $A[q + 1 \dots r]$.
- **Combine** by merging the two sorted sub arrays $A[p \dots q]$ and $A[q + 1 \dots r]$ to produce a single sorted sub array $A[p \dots r]$. To accomplish this step, we'll define a procedure $\text{MERGE}(A, p, q, r)$.

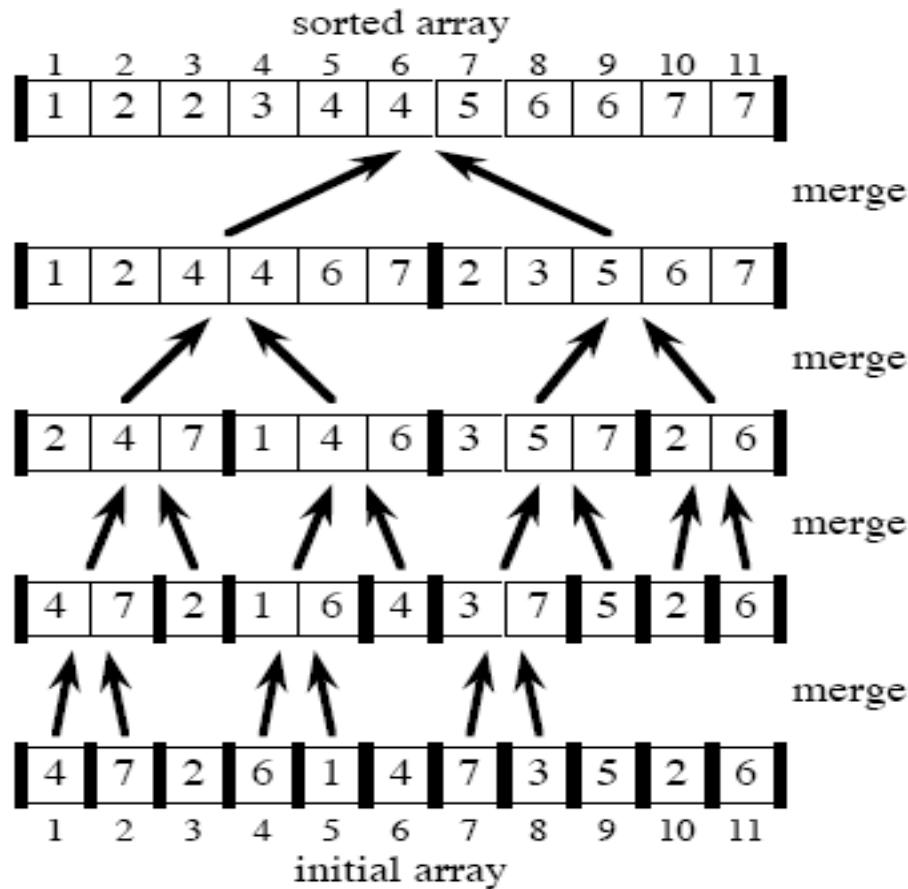
Example

Bottom-up view for $n = 8$: [Heavy lines demarcate subarrays used in subproblems.]



Example

Bottom-up view for $n = 11$: [Heavy lines demarcate subarrays used in subproblems.]

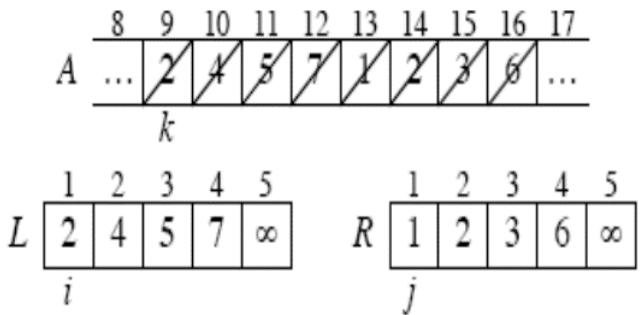


Merge Sort (Algorithm)

The recursion bottoms out when the subarray has just 1 element, so that it's trivially sorted.

```
MERGE-SORT( $A, p, r$ )
if  $p < r$                                 ▷ Check for base case
  then  $q \leftarrow \lfloor(p + r)/2\rfloor$       ▷ Divide
        MERGE-SORT( $A, p, q$ )                 ▷ Conquer
        MERGE-SORT( $A, q + 1, r$ )              ▷ Conquer
        MERGE( $A, p, q, r$ )                  ▷ Combine
```

Example [A call of MERGE(9, 12, 16)]



Example [A call of MERGE(9, 12, 16)]

A	8	9	10	11	12	13	14	15	16	17	...
	...	2	4	5	7	1	2	3	6	...	
		<i>k</i>									

L	1	2	3	4	5	...
	2	4	5	7	∞	
	<i>i</i>					

R	1	2	3	4	5	...
	1	2	3	6	∞	
	<i>j</i>					

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	4	5	7	1	2	3	6	...	
		<i>k</i>									

L	1	2	3	4	5	...
	2	4	5	7	∞	
	<i>i</i>					

R	1	2	3	4	5	...
	1	2	3	6	∞	
	<i>j</i>					

Example [A call of MERGE(9, 12, 16)]

A	8	9	10	11	12	13	14	15	16	17	...
	...	2	4	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i			j		

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	4	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i			j		

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	5	7	1	2	3	6	...	
	k										

L	1	2	3	4	5	
	2	4	5	7	∞	
	i			j		

Example [A call of MERGE(9, 12, 16)]

A	8	9	10	11	12	13	14	15	16	17	...
	...	2	4	5	7	1	2	3	6	...	
											k

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	
	1	2	3	6	∞	
	j					

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	4	5	7	1	2	3	6	...	
											k

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	
	1	2	3	6	∞	
	j					

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	5	7	1	2	3	6	...	
											k

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	
	1	2	3	6	∞	
	j					

A	8	9	10	11	12	13	14	15	16	17	...
	...	1	2	2	7	1	2	3	6	...	
											k

L	1	2	3	4	5	
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	
	1	2	3	6	∞	
	j					

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	1	2	3	6	\dots	
											k

L	1	2	3	4	5	\dots
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	\dots
	1	2	3	6	∞	
		j				

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	4	2	3	6	\dots
											k

L	1	2	3	4	5	\dots
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	\dots
	1	2	3	6	∞	
		j				

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	3	6	\dots	
											k

L	1	2	3	4	5	\dots
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	\dots
	1	2	3	6	∞	
		j				

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	6	\dots	
											k

L	1	2	3	4	5	\dots
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	\dots
	1	2	3	6	∞	
		j				

A	8	9	10	11	12	13	14	15	16	17	\dots
	...	1	2	2	3	4	5	6	7	\dots	
											k

L	1	2	3	4	5	\dots
	2	4	5	7	∞	
	i					

R	1	2	3	4	5	\dots
	1	2	3	6	∞	
		j				

Merging

Input: Array A and indices p, q, r such that

- $p \leq q < r$.
- Subarray $A[p \dots q]$ is sorted and subarray $A[q + 1 \dots r]$ is sorted. By the restrictions on p, q, r , neither subarray is empty.

Output: The two subarrays are merged into a single sorted subarray in $A[p \dots r]$.

We implement it so that it takes $\Theta(n)$ time, where $n = r - p + 1$ = the number of elements being merged.

Pseudocode (Merging)

MERGE(A, p, q, r)

$n_1 \leftarrow q - p + 1$

$n_2 \leftarrow r - q$

create arrays $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$

for $i \leftarrow 1$ **to** n_1

do $L[i] \leftarrow A[p + i - 1]$

for $j \leftarrow 1$ **to** n_2

do $R[j] \leftarrow A[q + j]$

$L[n_1 + 1] \leftarrow \infty$

$R[n_2 + 1] \leftarrow \infty$

$i \leftarrow 1$

$j \leftarrow 1$

for $k \leftarrow p$ **to** r

do if $L[i] \leq R[j]$

then $A[k] \leftarrow L[i]$

$i \leftarrow i + 1$

else $A[k] \leftarrow R[j]$

$j \leftarrow j + 1$

Analyzing divide-and-conquer algorithms

Use a *recurrence equation* (more commonly, a *recurrence*) to describe the running time of a divide-and-conquer algorithm.

Let $T(n)$ = running time on a problem of size n .

- If the problem size is small enough (say, $n \leq c$ for some constant c), we have a base case. The brute-force solution takes constant time: $\Theta(1)$.
- Otherwise, suppose that we divide into a subproblems, each $1/b$ the size of the original. (In merge sort, $a = b = 2$.)
- Let the time to divide a size- n problem be $D(n)$.
- There are a subproblems to solve, each of size $n/b \Rightarrow$ each subproblem takes $T(n/b)$ time to solve \Rightarrow we spend $aT(n/b)$ time solving subproblems.
- Let the time to combine solutions be $C(n)$.
- We get the recurrence

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c , \\ aT(n/b) + D(n) + C(n) & \text{otherwise .} \end{cases}$$

Analyzing merge sort

For simplicity, assume that n is a power of 2 \Rightarrow each divide step yields two subproblems, both of size exactly $n/2$.

The base case occurs when $n = 1$.

When $n \geq 2$, time for merge sort steps:

Divide: Just compute q as the average of p and $r \Rightarrow D(n) = \Theta(1)$.

Conquer: Recursively solve 2 subproblems, each of size $n/2 \Rightarrow 2T(n/2)$.

Combine: MERGE on an n -element subarray takes $\Theta(n)$ time $\Rightarrow C(n) = \Theta(n)$.

Since $D(n) = \Theta(1)$ and $C(n) = \Theta(n)$, summed together they give a function that is linear in n : $\Theta(n) \Rightarrow$ recurrence for merge sort running time is

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

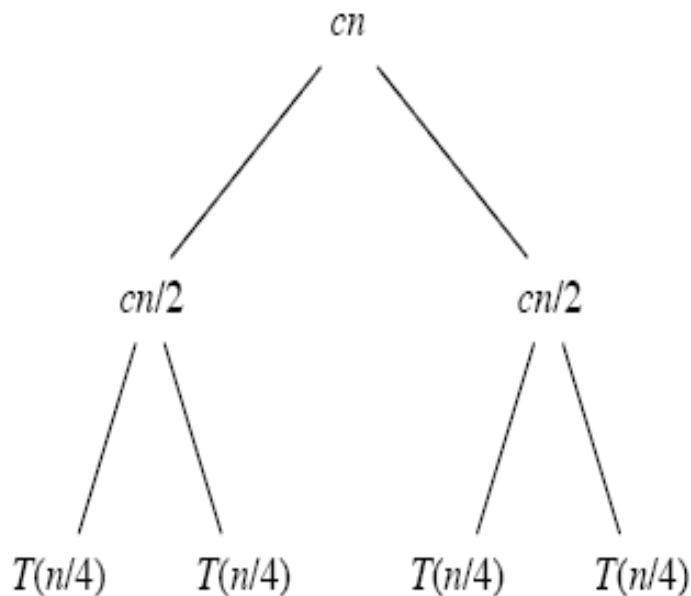
Recursion tree (Step 1)

- For the original problem, we have a cost of cn , plus the two subproblems, each costing $T(n/2)$:



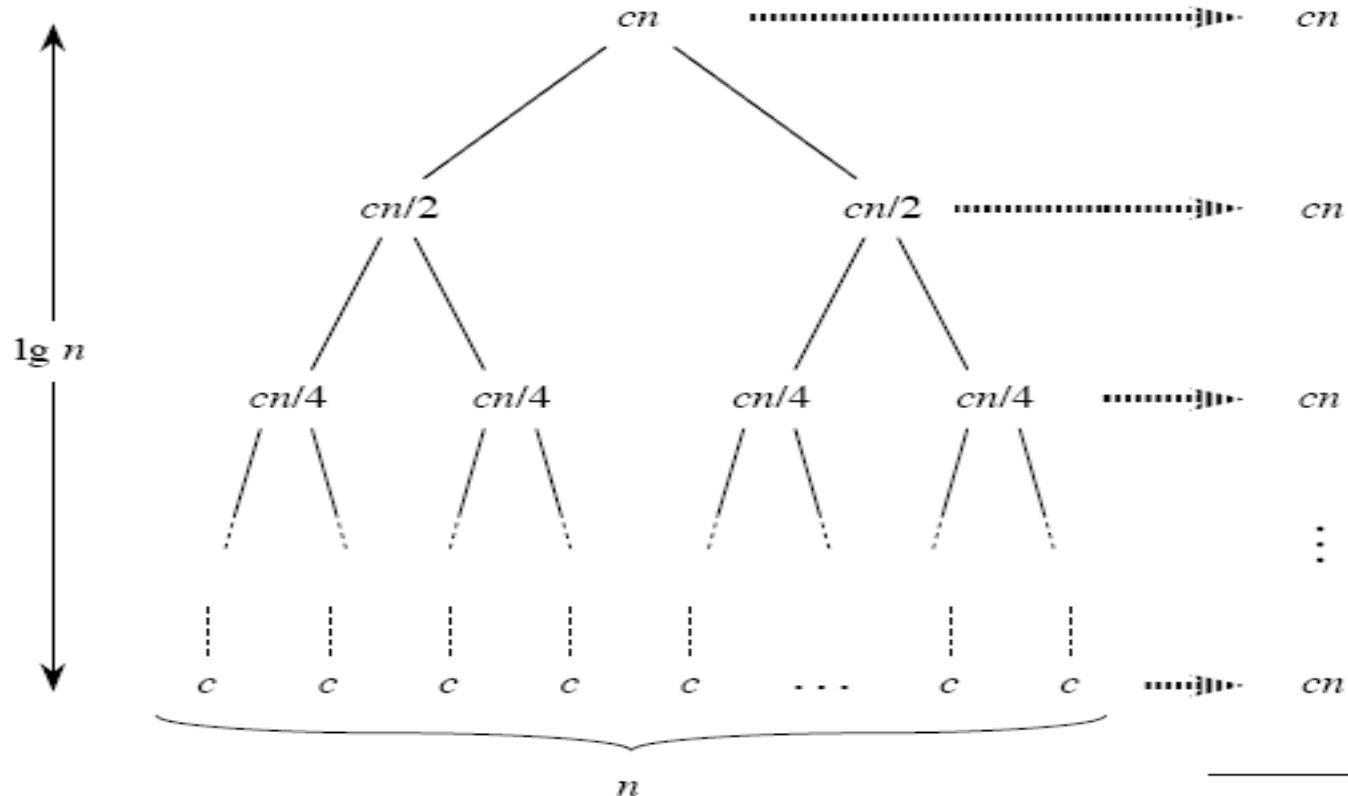
Recursion tree (Step 2)

- For each of the size- $n/2$ subproblems, we have a cost of $cn/2$, plus two subproblems, each costing $T(n/4)$:



Recursion tree (Step n)

- Continue expanding until the problem sizes get down to 1:



Total: $cn \lg n + cn$

Home Assignment

- Solve the Recurrence of Merge Sort with the help of Iteration method.

Thank U

Design and Analysis of Algorithm

Divide and Conquer strategy (Quick Sort)

Lecture -22

Overview

- Worst-case running time: $\Theta(n^2)$
- Expected running time: $\Theta(n \lg n)$
- Constants hidden in $\Theta(n \lg n)$ are small.
- Sorts in place.

Description of quicksort

Quicksort is based on the three-step process of divide-and-conquer.

- To sort the subarray $A[p \dots r]$:

Divide: Partition $A[p \dots r]$, into two (possibly empty) subarrays $A[p \dots q - 1]$ and $A[q + 1 \dots r]$, such that each element in the first subarray $A[p \dots q - 1]$ is $\leq A[q]$ and $A[q]$ is \leq each element in the second subarray $A[q + 1 \dots r]$.

Conquer: Sort the two subarrays by recursive calls to QUICKSORT.

Combine: No work is needed to combine the subarrays, because they are sorted in place.

- Perform the divide step by a procedure PARTITION, which returns the index q that marks the position separating the subarrays.

```
QUICKSORT( $A, p, r$ )
if  $p < r$ 
    then  $q \leftarrow \text{PARTITION}(A, p, r)$ 
        QUICKSORT( $A, p, q - 1$ )
        QUICKSORT( $A, q + 1, r$ )
```

Initial call is $\text{QUICKSORT}(A, 1, n)$.

Partitioning

Partition subarray $A[p \dots r]$ by the following procedure:

```
PARTITION( $A, p, r$ )
 $x \leftarrow A[r]$ 
 $i \leftarrow p - 1$ 
for  $j \leftarrow p$  to  $r - 1$ 
    do if  $A[j] \leq x$ 
        then  $i \leftarrow i + 1$ 
            exchange  $A[i] \leftrightarrow A[j]$ 
exchange  $A[i + 1] \leftrightarrow A[r]$ 
return  $i + 1$ 
```

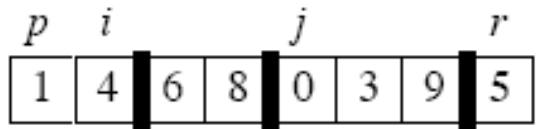
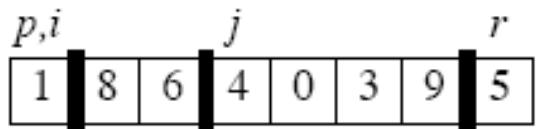
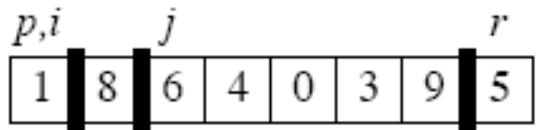
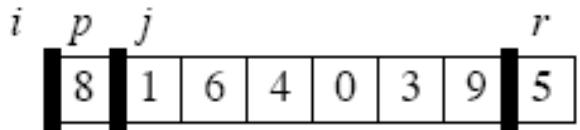
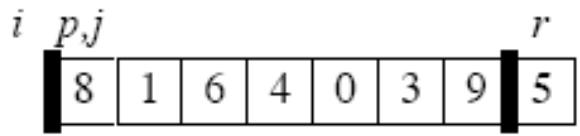
- PARTITION always selects the last element $A[r]$ in the subarray $A[p \dots r]$ as the *pivot*—the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:

Loop invariant:

1. All entries in $A[p \dots i]$ are \leq pivot.
2. All entries in $A[i + 1 \dots j - 1]$ are $>$ pivot.
3. $A[r] = \text{pivot}$.

It's not needed as part of the loop invariant, but the fourth region is $A[j \dots r - 1]$, whose entries have not yet been examined, and so we don't know how they compare to the pivot.

Example: On an 8-element subarray.



- ↑ $A[r]$: pivot
- ↑ $A[j .. r-1]$: not yet examined
- ↑ $A[i+1 .. j-1]$: known to be $>$ pivot
- ↑ $A[p .. i]$: known to be \leq pivot

$$\begin{array}{ccccccc} p & & i & & j & & r \\ \boxed{1} & \boxed{4} & \boxed{0} & \boxed{8} & \boxed{6} & \boxed{3} & \boxed{9} \end{array}$$
$$\begin{array}{ccccccc} p & & i & & j & & r \\ \boxed{1} & \boxed{4} & \boxed{0} & \boxed{3} & \boxed{6} & \boxed{8} & \boxed{9} \end{array}$$
$$\begin{array}{ccccccc} p & & i & & & & r \\ \boxed{1} & \boxed{4} & \boxed{0} & \boxed{3} & \boxed{6} & \boxed{8} & \boxed{9} \end{array}$$
$$\begin{array}{ccccccc} p & & i & & & & r \\ \boxed{1} & \boxed{4} & \boxed{0} & \boxed{3} & \boxed{5} & \boxed{8} & \boxed{9} \end{array}$$

Correctness: Use the loop invariant to prove correctness of PARTITION:

Initialization: Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot and the subarrays $A[p \dots i]$ and $A[i + 1 \dots j - 1]$ are empty.

Maintenance: While the loop is running, if $A[j] \leq \text{pivot}$, then $A[j]$ and $A[i + 1]$ are swapped and then i and j are incremented. If $A[j] > \text{pivot}$, then increment only j .

Termination: When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases: $A[p \dots i] \leq \text{pivot}$, $A[i + 1 \dots r - 1] > \text{pivot}$, and $A[r] = \text{pivot}$.

The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays. This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i + 1]$ and $A[r]$.

Time for partitioning: $\Theta(n)$ to partition an n -element subarray.

Performance of quicksort

The running time of quicksort depends on the partitioning of the subarrays:

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and $n - 1$ elements in the other subarray.
- Get the recurrence

$$\begin{aligned}T(n) &= T(n - 1) + T(0) + \Theta(n) \\&= T(n - 1) + \Theta(n) \\&= \Theta(n^2).\end{aligned}$$

- Same running time as insertion sort.
- In fact, the worst-case running time occurs when quicksort takes a sorted array as input, but insertion sort runs in $O(n)$ time in this case.

Best case

- Occurs when the subarrays are completely balanced every time.
- Each subarray has $\leq n/2$ elements.
- Get the recurrence

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\&= \Theta(n \lg n).\end{aligned}$$

Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence

$$\begin{aligned}T(n) &\leq T(9n/10) + T(n/10) + \Theta(n) \\&= O(n \lg n).\end{aligned}$$

Randomized version of quicksort

- We have assumed that all input permutations are equally likely.
- This is not always true.
- To correct this, we add randomization to quicksort.
- We could randomly permute the input array.
- Instead, we use ***random sampling***, or picking one element at random.
- Don't always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted.

We add this randomization by not always using $A[r]$ as the pivot, but instead randomly picking an element from the subarray that is being sorted

```
RANDOMIZED-PARTITION( $A, p, r$ )
```

```
 $i \leftarrow \text{RANDOM}(p, r)$ 
```

```
exchange  $A[r] \leftrightarrow A[i]$ 
```

```
return PARTITION( $A, p, r$ )
```

Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

```
RANDOMIZED-QUICKSORT( $A, p, r$ )
```

```
if  $p < r$ 
```

```
    then  $q \leftarrow \text{RANDOMIZED-PARTITION}(A, p, r)$ 
```

```
        RANDOMIZED-QUICKSORT( $A, p, q - 1$ )
```

```
        RANDOMIZED-QUICKSORT( $A, q + 1, r$ )
```

Analysis of quicksort

We will analyze

- the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT (the same), and
- the expected (average-case) running time of RANDOMIZED-QUICKSORT is $O(n \lg n)$.

Thank U

Design and Analysis of Algorithm

**Divide and Conquer strategy
(Maximum Sub-array Problem)**

Lecture -23

Overview

- Learn the technique of “divide and conquer” in the context of the maximum sub-array with analysis.

The Maximum subarray Problem (A Divide and Conquer Approach)

- **Divide** the problem into a number of sub problems.
- **Conquer** the sub problems by solving them recursively.
 - ***Base case:*** If the sub problems are small enough, just solve them by brute force.
- **Combine** the sub problem solutions to give a solution to the original problem.

The Maximum subarray problem

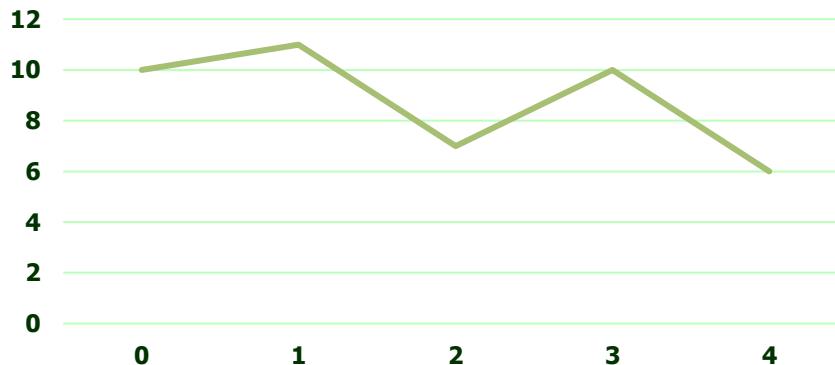
- **Problem:** In a share market you can buy a unit of stock, only one time, then sell it at a later date
 - Buy/sell at end of day
- **Strategy:** buy low, sell high
 - The lowest price may appear after the highest price
 - Assume you know future prices
- **Objective:** Can you maximize profit by buying at lowest price and selling at highest price?

The Maximum subarray problem

- Example 1:

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

Daywise stock price information



Concept: Buy lowest sell highest
Objective : Maximize the profit

The Maximum subarray problem

- Transformation of Example 1
 - Find sequence of days so that:
 - the net change from last to first is maximized
 - Look at the daily change in price
 - $\text{Change on day } i = \text{price on day}(i) - \text{price day } (i - 1)$
 - We now have an array of changes (numbers),

Day	0	1	2	3	4
Price	10	11	7	10	6
Changes		1	-4	3	-4

- Hence the changes are : 1, -4, 3, -4
- Find contiguous subarray with largest sum
- maximum subarray—E.g.: buy after day 2, sell after day 3

The Maximum subarray problem

➤ Example 2:

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



Concept: Buy lowest sell highest

Objective : Maximize the profit

The Maximum subarray problem

- Transformation of Example 2:
 - Find sequence of days so that:
 - the net change from last to first is maximized
 - Look at the daily change in price
 - $\text{Change on day } i = \text{price on day}(i) - \text{price day } (i - 1)$
 - We now have an array of changes (numbers),

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
Changes		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7

- Hence the changes are : 13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, and 7
- Find contiguous subarray with largest sum
- maximum subarray-E.g.: buy after day 7, sell after day 11

The Maximum subarray problem

➤ **Question**

- How many buy/sell pairs are possible over ‘n’ days?
(i.e. search every possible pair of buy and sell dates in which the buy date precedes the sell date)

➤ **Brute force Approach**

- Evaluate each pair and keep track of maximum.

The Maximum subarray problem

➤ Brute force Approach

The Maximum subarray problem

➤ Brute force Approach

Gap between day=2

The Maximum subarray problem

➤ Brute force Approach

Gap between day=3

The Maximum subarray problem

➤ Brute force Approach

Gap between day=4

The Maximum subarray problem

➤ Brute force Approach

The Maximum subarray problem

➤ Brute force Approach

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
SUB STRING ARRAY (i.e. S Array)																	
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	
[1]	13	10	-15	5	2	-14	-37	-19	1	-6	6	1	-21	-6	-10	-3	
[2]		-3	-28	-8	-11	-27	-50	-32	-12	-19	-7	-12	-34	-19	-23	-16	
[3]			-25	-5	-8	-24	-47	-29	-9	-16	-4	-9	-31	-16	-20	-13	
[4]				20	17	1	-22	-4	16	9	21	16	-6	9	5	12	
[5]					-3	-19	-42	-24	-4	-11	1	-4	-26	-11	-15	-8	
[6]						-16	-39	-21	-1	-8	4	-1	-23	-8	-12	-5	
[7]							-23	-5	15	8	20	15	-7	8	4	11	
[8]								18	38	31	43	38	16	31	27	34	
[9]									20	13	25	20	-2	13	9	16	
[10]										-7	5	0	-22	-7	-11	-4	
[11]											12	7	-15	0	-4	3	
[12]												-5	-27	-12	-16	-9	
[13]													-22	-7	-11	-4	
[14]														15	11	18	
[15]															-4	3	
[16]																7	

Hence, maximum subarray—E.g.: buy after day **7**, sell on day **11**

The Maximum subarray problem

- **Brute force Approach**
 - The total number of pairs (Combinations) are $\binom{n}{2}$.
Hence the complexity is $\Theta(n^2)$
 - Can we do better?

The Maximum subarray problem

- **Brute force Approach**
 - The total number of pairs (Combinations) are $\binom{n}{2}$.
Hence the complexity is $\Theta(n^2)$
 - Can we do better?

Let's rewrite the problem again:

The Maximum subarray problem

The maximum sum subarray problem is the task to find a contiguous subarray with the largest sum of a given one-dimensional array $Arr[1..n]$ of numbers. The task is to find indices ' i ' and ' j ' with the condition $1 \leq i \leq j \leq n$, such that:

$$\sum_{x=i}^j Arr[x]$$

Is as large as possible.

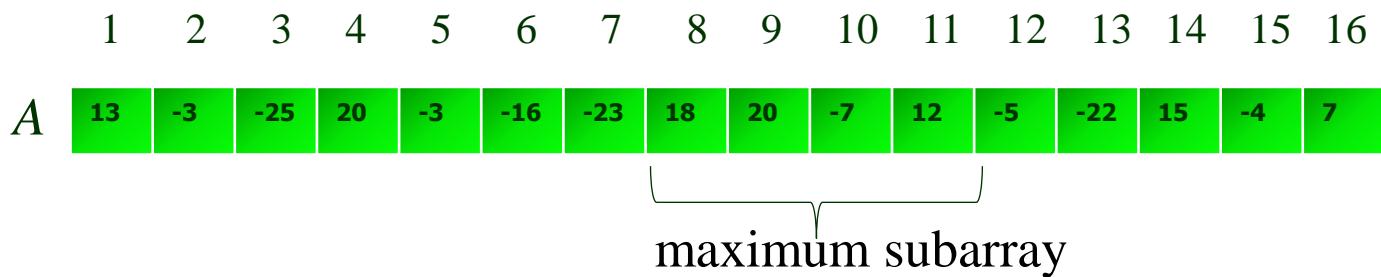
(Note: The number of the input array may be positive, negative or zero)

The Maximum sum subarray problem

- **Input:** an array $A[1..n]$ of n numbers
 - Assume that some of the numbers are negative, because this problem is trivial when all numbers are nonnegative
- **Output:** a nonempty subarray $A[i..j]$ having the largest sum

$$S[i, j] = A_i + A_{i+1} + \dots + A_j$$

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
Changes		13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7



The Maximum subarray problem

➤ Divide and Conquer Approach

- **Subproblem:** Find a maximum subarray of $A[\text{low} .. \text{high}]$
In initial call, $\text{low} = 1$ and $\text{high} = n$.
- **Divide:** the subarray into two subarrays of as equal size as possible. Find the midpoint mid of the subarrays, and consider the subarrays $A[\text{low} .. \text{mid}]$ and $A[\text{mid}+1 .. \text{high}]$.
- **Conquer:** by finding the maximum subarrays of $A[\text{low} .. \text{mid}]$ and $A[\text{mid}+1..\text{high}]$.
- **Combine:** by finding a maximum subarray that crosses the midpoint, and using the best solution out of the three (the subarray crossing the midpoint and the two solutions found in the conquer step).

The Maximum subarray problem

➤ Divide and Conquer Approach

Possible locations of a maximum subarray $A[i..j]$ of $A[low..high]$, where $mid = \lfloor (low+high)/2 \rfloor$

- entirely in $A[low..mid]$ ($low \leq i \leq j \leq mid$)
- entirely in $A[mid+1..high]$ ($mid < i \leq j \leq high$)
- crossing the midpoint ($low \leq i \leq mid < j \leq high$)

The Maximum subarray problem

➤ Divide and Conquer Approach

crosses the midpoint

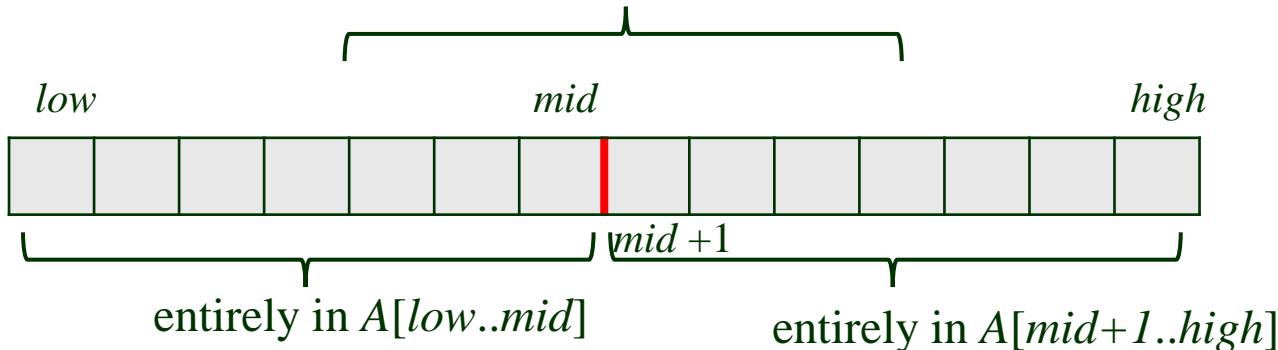


Fig (a): Possible locations of subarrays of $A[low..high]$

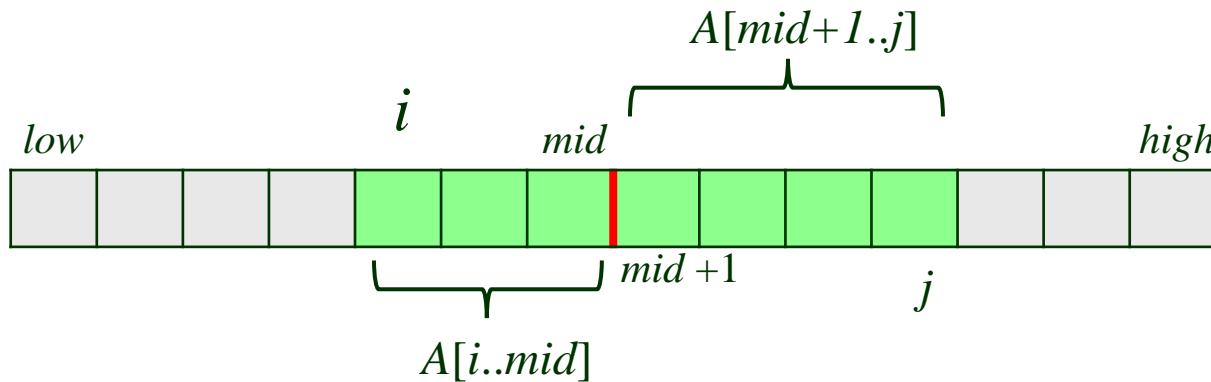
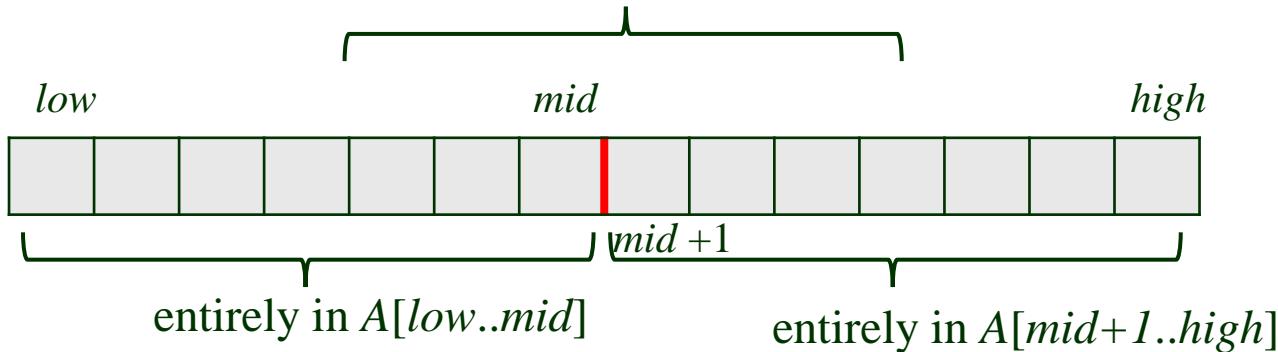


Fig (b): $A[i..j]$ comprises two subarrays $A[i..mid]$ and $A[mid+1..j]$

The Maximum subarray problem

➤ Divide and Conquer Approach

crosses the midpoint

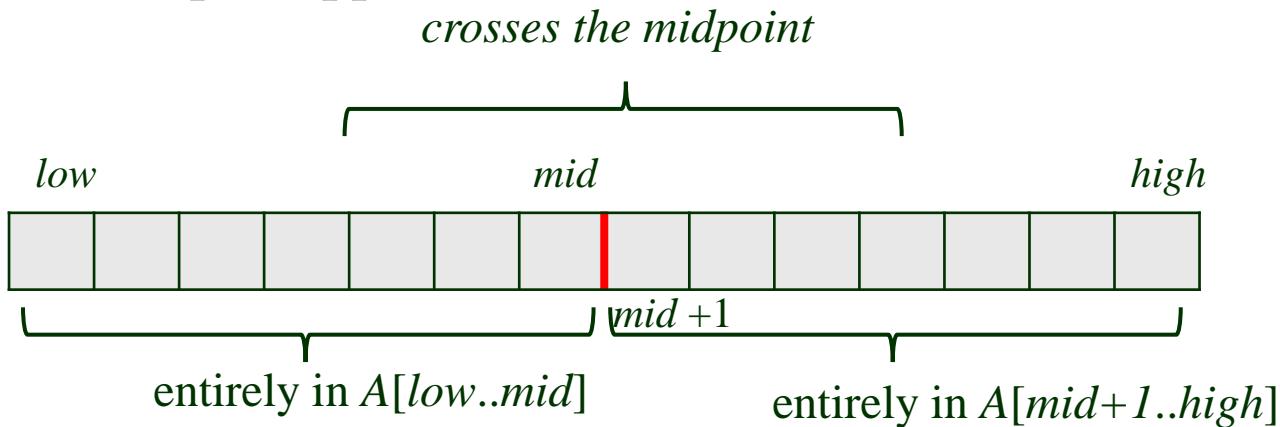


For example :

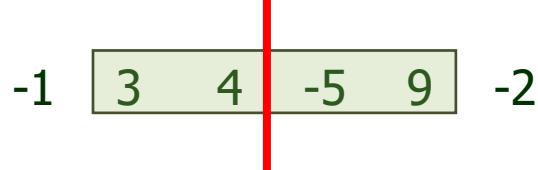
-1 3 4 -5 9 -2

The Maximum subarray problem

➤ Divide and Conquer Approach



For example :



$$\text{Left Sum} = -1 + 3 + 4 = 6$$

$$\text{Right Sum} = -5 + 9 + -2 = 2$$

$$\text{Cross Midpoint Sum} = 3 + 4 + -5 + 9 = 11$$

Hence, Max sum =11 and sequence is (3 , 4, -5, 9)

The Maximum subarray problem

Changes(A)	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
indices	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S[8..8]								max - left \Rightarrow	18	20						S[9..9]
S[7..8]									-5		13					S[9..10]
S[6..8]									-21			25	\Leftarrow max - right			S[9..11]
S[5..8]									-24			20				S[9..12]
S[4..8]									-4				-2			S[9..13]
S[3..8]									-29				13			S[9..14]
S[2..8]									-32				9			S[9..15]
S[1..8]									-19					16		S[9..16]

\Rightarrow maximum subarray crossing mid is $S[8..11] = 18 + 25 = 43$

Find-Max-Crossing-Subarray(A , low , mid , $high$)

Find maximum subarray of the form $A[low..mid]$

```

left-sum=-∞
sum=0
for  $i=mid$  downto  $low$ 
    sum=sum+A[i]
    if sum>left-sum
        left-sum=sum
        max-left=i
    
```

Changes(A)	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7	
indices	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
S[8..8]								max - left \Rightarrow	18	20							S[9..9]
S[7..8]									-5		13						S[9..10]
S[6..8]									-21		25				$\Leftarrow max - right$		S[9..11]
S[5..8]									-24		20						S[9..12]
S[4..8]									-4					-2			S[9..13]
S[3..8]									-29					13			S[9..14]
S[2..8]									-32					9			S[9..15]
S[1..8]									-19					16			S[9..16]

Find-Max-Crossing-Subarray(A , low , mid , $high$)

Find maximum subarray of the form $A[low..mid]$

Find maximum subarray of the form $A[mid+1..high]$

```

left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i

right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j

// Return the indices and
return(max-left, max-right)

```

Changes (A)	13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
indices	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
S[8..8]								18	20							S[9..9]
S[7..8]								-5		13						S[9..10]
S[6..8]								-21			25				$\Leftarrow \max - right$	S[9..11]
S[5..8]								-24			20					S[9..12]
S[4..8]								-4						-2		S[9..13]
S[3..8]								-29					13			S[9..14]
S[2..8]								-32					9			S[9..15]
S[1..8]								-19					16			S[9..16]

The Maximum subarray problem

FIND-MAXIMUM-SUBARRAY($A, low, high$)

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

The Maximum subarray problem

Initial call: FIND-MAXIMUM-SUBARRAY(A,1,n)

- Divide by computing *mid*.
- Conquer by the two recursive calls to FIND-MAXIMUM-SUBARRAY.
- Combine by calling FIND-MAX-CROSSING-SUBARRAY and then determining which of the three results gives the maximum sum.
- Base case is when the subarray has only 1 element.

The Maximum subarray problem(Analysis)

FIND-MAXIMUM-SUBARRAY($A, low, high$) $\longrightarrow T(n)$

if $high == low$ $\longrightarrow \Theta(1)$

return ($low, high, A[low]$) $//$ base case: only one element

else $mid = \lfloor (low + high)/2 \rfloor$ $\longrightarrow \Theta(1)$

 ($left-low, left-high, left-sum$) =

 FIND-MAXIMUM-SUBARRAY(A, low, mid) $\longrightarrow T\left(\frac{n}{2}\right)$

 ($right-low, right-high, right-sum$) =

 FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$) $\longrightarrow T\left(\frac{n}{2}\right)$

 ($cross-low, cross-high, cross-sum$) =

 FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$) $\longrightarrow \Theta(n)$

if $left-sum \geq right-sum$ and $left-sum \geq cross-sum$

return ($left-low, left-high, left-sum$)

elseif $right-sum \geq left-sum$ and $right-sum \geq cross-sum$

return ($right-low, right-high, right-sum$)

else return ($cross-low, cross-high, cross-sum$)

$\longrightarrow \Theta(1)$

The Maximum subarray problem(Analysis)

FIND-MAXIMUM-SUBARRAY($A, low, high$) $\longrightarrow T(n)$

if $high == low$ $\longrightarrow \Theta(1)$

return ($low, high, A[low]$) $//$ base case: only one element

else $mid = \lfloor (low + high)/2 \rfloor$ $\longrightarrow \Theta(1)$

 ($left-low, left-high, left-sum$) =

 FIND-MAXIMUM-SUBARRAY(A, low, mid) $\longrightarrow T\left(\frac{n}{2}\right)$

 Type equation here.

 ($right-low, right-high, right-sum$) =

 FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$) $\longrightarrow T\left(\frac{n}{2}\right)$

 ($cross-low, cross-high, cross-sum$) =

 FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$) $\longrightarrow \Theta(n)$

if $left-sum \geq right-sum$ and $left-sum \geq cross-sum$

return ($left-low, left-high, left-sum$)

elseif $right-sum \geq left-sum$ and $right-sum \geq cross-sum$

return ($right-low, right-high, right-sum$)

else return ($cross-low, cross-high, cross-sum$)

$\longrightarrow \Theta(1)$

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

The Maximum subarray problem(Analysis)

FIND-MAXIMUM-SUBARRAY($A, low, high$) $\longrightarrow T(n)$

if $high == low$ $\longrightarrow \Theta(1)$

return ($low, high, A[low]$) $//$ base case: only one element

else $mid = \lfloor (low + high)/2 \rfloor$ $\longrightarrow \Theta(1)$

($left-low, left-high, left-sum$) =

FIND-MAXIMUM-SUBARRAY(A, low, mid) $\longrightarrow T\left(\frac{n}{2}\right)$

Type equation here.

($right-low, right-high, right-sum$) =

FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$) $\longrightarrow T\left(\frac{n}{2}\right)$

($cross-low, cross-high, cross-sum$) =

FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$) $\longrightarrow \Theta(n)$

if $left-sum \geq right-sum$ and $left-sum \geq cross-sum$

return ($left-low, left-high, left-sum$)

elseif $right-sum \geq left-sum$ and $right-sum \geq cross-sum$

return ($right-low, right-high, right-sum$)

else return ($cross-low, cross-high, cross-sum$)

$\longrightarrow \Theta(1)$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow \Theta(n \lg n)$$

Analysing Maximum subarray problem

Simplifying assumption: Original problem size is a power of 2, so that all subproblem sizes are integer. [We made the same simplifying assumption when we analyzed merge sort.]

Let $T(n)$ denote the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements.

Base case: Occurs when $high$ equals low , so that $n = 1$. The procedure just returns $\Rightarrow T(n) = \Theta(1)$.

Recursive case: Occurs when $n > 1$.

- Dividing takes $\Theta(1)$ time.
- Conquering solves two subproblems, each on a subarray of $n/2$ elements. Takes $T(n/2)$ time for each subproblem $\Rightarrow 2T(n/2)$ time for conquering.
- Combining consists of calling FIND-MAX-CROSSING-SUBARRAY, which takes $\Theta(n)$ time, and a constant number of constant-time tests $\Rightarrow \Theta(n) + \Theta(1)$ time for combining.

Analysing Maximum subarray problem

Recurrence for recursive case becomes

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \quad (\text{absorb } \Theta(1) \text{ terms into } \Theta(n)) . \end{aligned}$$

The recurrence for all cases:

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

Same recurrence as for merge sort. Can use the master method to show that it has solution $T(n) = \Theta(n \lg n)$.

Thus, with divide-and-conquer, we have developed a $\Theta(n \lg n)$ -time solution.
Better than the $\Theta(n^2)$ -time brute-force solution.

Analysing Maximum subarray problem

A complete example given in next slide for easy understanding.

The Maximum subarray problem

➤ Divide and Conquer Approach

Start Division

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The Maximum subarray problem

➤ Divide and Conquer Approach

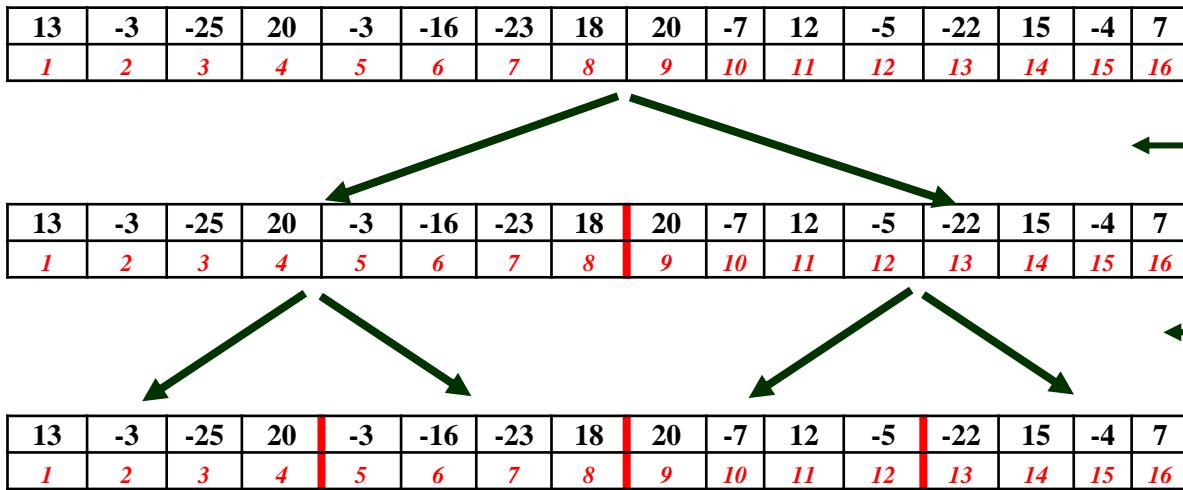
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Divide

The Maximum subarray problem

➤ Divide and Conquer Approach



The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

← Divide

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

← Divide

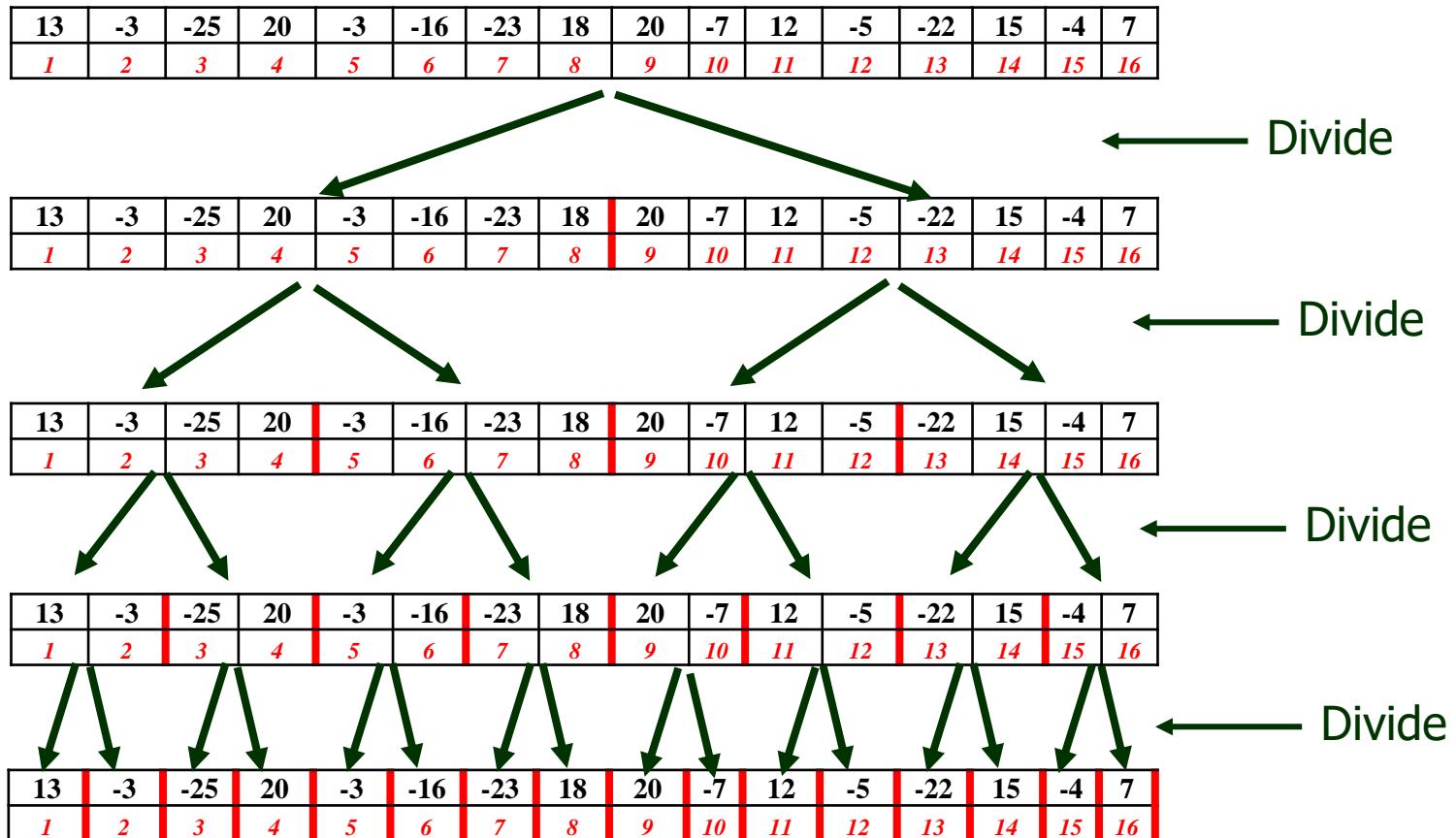
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

← Divide

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The Maximum subarray problem

➤ Divide and Conquer Approach



The Maximum subarray problem

➤ Divide and Conquer Approach

Start Conquer

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
LS=13	RS=-3	LS=-25	RS=20	LS=-3	RS=16	LS=-19	RS=-5	LS=-23	RS=18	LS=20	RS=-7	LS=12	RS=-5	LS=-22	RS=15	LS=-4	RS=7

Conquer

[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

LS=13
RS=-3
CS=10

← Conquer

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16		
LS=13	RS=-3	LS=-25	RS=20	LS=-3	RS=16	LS=-19	RS=-5	LS=-23	RS=18	LS=20	RS=-7	LS=12	RS=-5	LS=-22	RS=15	LS=-4	RS=7

Conquer

[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

The Maximum subarray problem

➤ Divide and Conquer Approach

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

LS=20 RS=18 Conquer

CS=17

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

LS=13 RS=20 Conquer

CS=5

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

LS=-3 RS=18 Conquer

CS=-21

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

LS=20 RS=12 Conquer

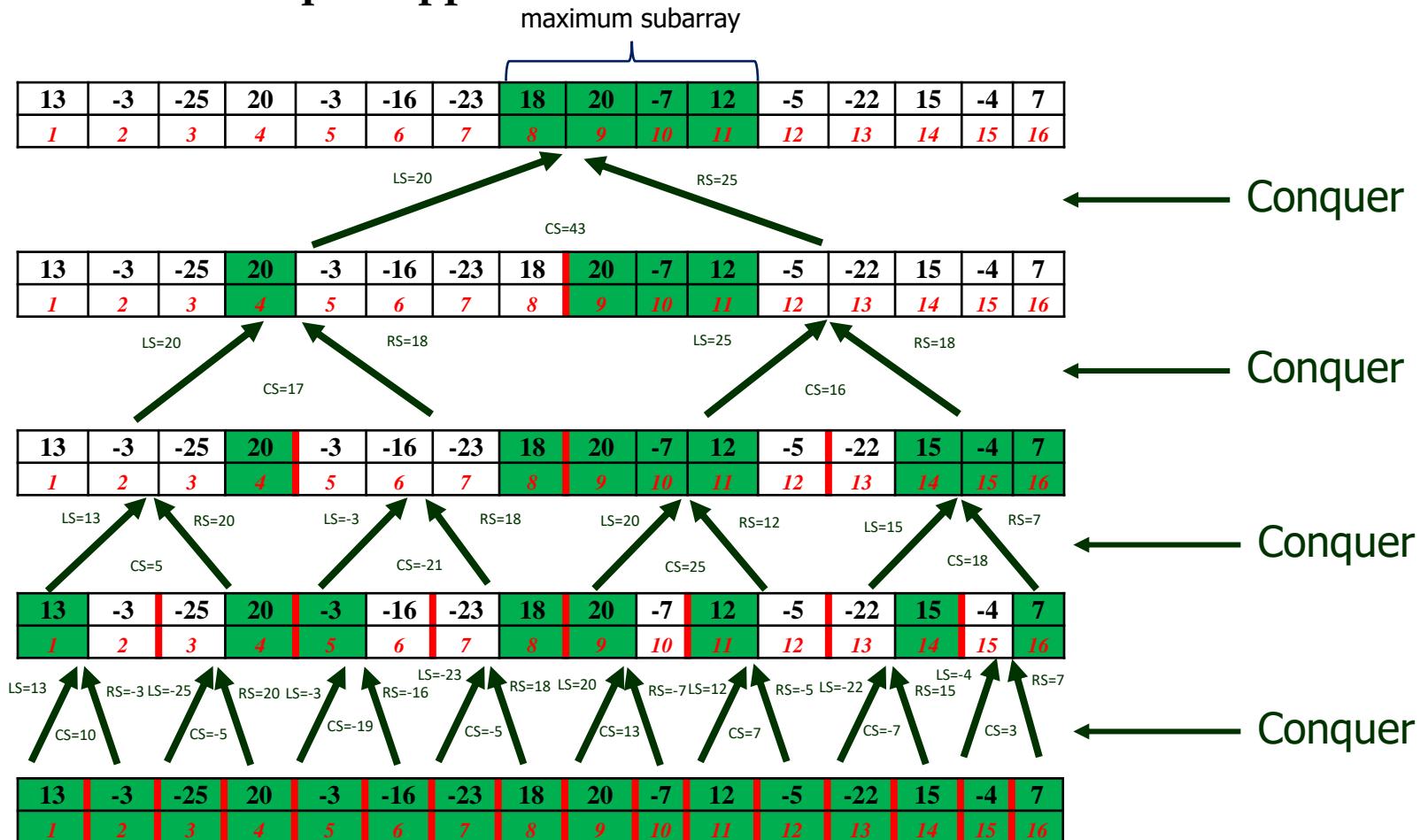
CS=25

13	-3	-25	20	-3	-16	-23	18	20	-7	12	-5	-22	15	-4	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

The Maximum subarray problem

➤ Divide and Conquer Approach



[Note: Where LS (Left Sum), RS (Right Sum) and CS (Cross Sum)]

Home Assignment

- Solve the Maximum Subarray problem in $\Theta(n)$ time.

Thank U

Design and Analysis of Algorithm

**Divide and Conquer strategy
(Matrix Multiplication
by Strassen's Algorithm)**

Lecture -24

Overview

- *Learn the implementation techniques of “divide and conquer” in the context of the Strassen’s Matrix multiplication with analysis.*
- *Conventional strategy $\Rightarrow O(n^3)$.*
- *Divide and Coques strategy $\Rightarrow O(n^3)$.*
- *Strassen’s strategy $\Rightarrow O(n^{2.81})$.*

Matrix Multiplication

- Problem definition:

Input: Two $n \times n$ (square) matrices, $A = (a_{ij})$ and $B = (b_{ij})$.

Output: $n \times n$ matrix $C = (c_{ij})$, where $C = A \cdot B$, i.e.,

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

for $i, j = 1, 2, \dots, n$.

Need to compute n^2 entries of C . Each entry is the sum of n values.

Matrix Multiplication

- Conventional strategy:

SQUARE-MAT-MULT(A, B, n)

```
let  $C$  be a new  $n \times n$  matrix
for  $i = 1$  to  $n$ 
    for  $j = 1$  to  $n$ 
         $c_{ij} = 0$ 
        for  $k = 1$  to  $n$ 
             $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$ 
return  $C$ 
```

Analysis: Three nested loops, each iterates n times, and innermost loop body takes constant time $\Rightarrow \Theta(n^3)$.

Matrix Multiplication

- Question is
Is $\Theta(n^3)$ is the best or we can multiply the matrix in $o(n^3)$ time?
(i.e. can we solve it in $< \Theta(n^3)$)
- Let's see with Divide and Conquer strategy.....

Matrix Multiplication

- Divide-and-conquer strategy :
 - As with the other divide-and-conquer algorithms, assume that n is a power of 2(i.e. 2^n).
 - Partition each of A,B, C into four $\frac{n}{2} \times \frac{n}{2}$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

For multiplication we can write $C = A . B$ as

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

Matrix Multiplication

- Divide-and-conquer strategy :

For multiplication we can write $C = A \cdot B$ as

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

Which create four equations. They are

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

Each of these equations multiplies two $\frac{n}{2} \times \frac{n}{2}$ matrices and then adds their $\frac{n}{2} \times \frac{n}{2}$ products.

Matrix Multiplication

- Divide-and-conquer strategy :

By using the equations of previous slide we can write the Divide and conquer algorithm.

REC-MAT-MULT (A, B, n)

Let C be a $n \times n$ matrix

If $n == 1$

$$C_{11} = A_{11} \times B_{11}$$

else partition A,B, and C into $\frac{n}{2} \times \frac{n}{2}$ submatrices.

$$C_{11} = \text{REC-MAT-MULT}(A_{11}, B_{11}, N/2) + \text{REC-MAT-MULT}(A_{12}, B_{21}, N/2)$$

$$C_{12} = \text{REC-MAT-MULT}(A_{11}, B_{12}, N/2) + \text{REC-MAT-MULT}(A_{12}, B_{22}, N/2)$$

$$C_{21} = \text{REC-MAT-MULT}(A_{21}, B_{11}, N/2) + \text{REC-MAT-MULT}(A_{22}, B_{21}, N/2)$$

$$C_{22} = \text{REC-MAT-MULT}(A_{21}, B_{12}, N/2) + \text{REC-MAT-MULT}(A_{22}, B_{22}, N/2)$$

Return C

Matrix Multiplication

- Analysis of Divide-and-conquer strategy :

Let $T(n)$ be the time to multiply two $\frac{n}{2} \times \frac{n}{2}$ matrices.

Base Case: $n=1$. Perform one scalar multiplication: $\Theta(1)$.

Recursive Case: $n>1$

- Dividing takes $\Theta(1)$ time, using index calculations.
- Conquering makes 8 recursive calls, each multiplying $\frac{n}{2} \times \frac{n}{2}$ matrices.
(i.e. $8T(n/2)$)
- Combining Takes $\Theta(n^2)$ time to add $\frac{n}{2} \times \frac{n}{2}$ matrices four items.

Hence the Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

The Complexity is $\Theta(n^3)$ (Apply Master Method)

Matrix Multiplication

- Analysis of Divide-and-conquer strategy :

Let $T(n)$ be the time to multiply two $\frac{n}{2} \times \frac{n}{2}$ matrices.

Base Case: $n=1$. Perform one scalar multiplication: $\Theta(1)$.

Recursive Case: $n>1$

- Dividing takes $\Theta(1)$ time, using index calculations.
- Conquering makes 8 recursive calls, each multiplying $\frac{n}{2} \times \frac{n}{2}$ matrices.
(i.e. $8T(n/2)$)
- Combining Takes $\Theta(n^2)$ time to add $\frac{n}{2} \times \frac{n}{2}$ matrices four items.

Hence the Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 8T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

The Complexity is $\Theta(n^3)$ (Apply Master Method)

Can we do better?

Matrix Multiplication

- Strassen's strategy :

The Idea:

- Make the recursion tree less bushy.
- Perform only 7(seven) recursive multiplications of $n/2 \times n/2$ matrices, rather than 8(Eight).

Matrix Multiplication

- Strassen's strategy :

The Algorithm:

1. As in the recursive method, partition each of the matrices into four $\frac{n}{2} \times \frac{n}{2}$ submatrices. Time: $\Theta(1)$
2. Compute 7 matrix products P, Q, R, S, T, U, V for each $\frac{n}{2} \times \frac{n}{2}$.
3. Compute $\frac{n}{2} \times \frac{n}{2}$ submatrices of C by adding and subtracting various combinations of the P_i . Time: $\Theta(n^2)$.

Matrix Multiplication

- Strassen's strategy :

Details of Step 2:

Compute 7 matrix products:

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \quad U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$Q = (A_{21} + A_{22}) \cdot B_{11} \quad V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$R = A_{11} \cdot (B_{12} - B_{22})$$

$$S = A_{22} \cdot (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) \cdot B_{22}$$

Matrix Multiplication

- Strassen's strategy :

Details of Step 3:

Compute C with 4 adding and subtracting :

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Matrix Multiplication

- Strassen's strategy :

Analysis:

The Recurrence is

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 7T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1 \end{cases}$$

The Complexity is $\Theta(n^{\log_2 7}) = \Theta(n^{2.81})$ (By using Master Method)

Matrix Multiplication

Example 1

- Compute Matrix multiplication of the following two matrices with the help of Strassen's strategy

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Matrix Multiplication

Example 1

- Compute Matrix multiplication of the following two matrices with the help of Strassen's strategy

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \text{ and } B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Ans:

$$A_{11}=1, A_{12}=2, A_{21}=3, \text{ and } A_{22}=4$$

$$B_{11}=5, B_{12}=6, B_{21}=7, \text{ and } B_{22}=8$$

Matrix Multiplication

Example 1

Calculate the value of P, Q, R, S, T, U and V

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) = (1+4)(5+8) = 5 \times 13 = 65$$

$$Q = (A_{21} + A_{22}) \cdot B_{11} = (3+4)5 = 7 \times 5 = 35$$

$$R = A_{11} \cdot (B_{12} - B_{22}) = 1(6-8) = 1 \times -2 = -2$$

$$S = A_{22} \cdot (B_{21} - B_{11}) = 4(7-5) = 4 \times 2 = 8$$

$$T = (A_{11} + A_{12}) \cdot B_{22} = (1+2)8 = 3 \times 8 = 24$$

$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12}) = (3-1)(5+6) = 2 \times 11 = 22$$

$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) = (2-4)(7+8) = -2 \times 15 = -30$$

Matrix Multiplication

Example 1

Compute C_{11} , C_{12} , C_{21} , and C_{22} :

$$C_{11} = P + S - T + V = 65 + 8 - 24 - 30 = 19$$

$$C_{12} = R + T = -2 + 24 = 22$$

$$C_{21} = Q + S = 35 + 8 = 43$$

$$C_{22} = P + R - Q + U = 65 - 2 - 35 + 22 = 50$$

Hence,

$$C = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

Matrix Multiplication

Example 2

Compute Matrix multiplication of the following two matrices with the help of Strassen's strategy.

$$A = \begin{bmatrix} 4 & 2 & 0 & 1 \\ 3 & 1 & 2 & 5 \\ 3 & 2 & 1 & 4 \\ 5 & 2 & 6 & 7 \end{bmatrix}, B = \begin{bmatrix} 2 & 1 & 3 & 2 \\ 5 & 4 & 2 & 3 \\ 1 & 4 & 0 & 2 \\ 3 & 2 & 4 & 1 \end{bmatrix}$$

Matrix Multiplication

Example 2

First we partition the input matrices into sub matrices as shown below:

$$A_{11} = \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix}, A_{12} = \begin{bmatrix} 0 & 1 \\ 2 & 5 \end{bmatrix}$$

$$B_{11} = \begin{bmatrix} 2 & 1 \\ 5 & 4 \end{bmatrix}, B_{12} = \begin{bmatrix} 3 & 2 \\ 2 & 3 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 3 & 2 \\ 5 & 2 \end{bmatrix}, A_{22} = \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix}$$

$$B_{21} = \begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix}, B_{22} = \begin{bmatrix} 0 & 2 \\ 4 & 1 \end{bmatrix}$$

Matrix Multiplication

Example 2

Calculate the value of P, Q, R, S, T, U and V

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$= \begin{bmatrix} 5 & 6 \\ 9 & 8 \end{bmatrix} \begin{bmatrix} 2 & 3 \\ 9 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 64 & 45 \\ 90 & 67 \end{bmatrix}$$

$$Q = (A_{21} + A_{22}) \cdot B_{11}$$

$$= \begin{bmatrix} 4 & 6 \\ 11 & 9 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 5 & 4 \end{bmatrix}$$

$$= \begin{bmatrix} 38 & 28 \\ 67 & 47 \end{bmatrix}$$

Matrix Multiplication

Example 2

$$R = A_{11} \cdot (B_{12} - B_{22})$$

$$= \begin{bmatrix} 4 & 2 \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ -2 & 2 \end{bmatrix}$$

$$= \begin{bmatrix} 8 & 4 \\ 7 & 2 \end{bmatrix}$$

$$S = A_{22} \cdot (B_{21} - B_{11})$$

$$= \begin{bmatrix} 1 & 4 \\ 6 & 7 \end{bmatrix} \begin{bmatrix} -1 & 3 \\ -2 & -2 \end{bmatrix}$$

$$= \begin{bmatrix} -9 & -5 \\ -20 & 4 \end{bmatrix}$$

Matrix Multiplication

Example 2

$$T = (A_{11} + A_{12}) \cdot B_{22}$$

$$= \begin{bmatrix} 4 & 3 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} 0 & 2 \\ 4 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 12 & 11 \\ 24 & 16 \end{bmatrix}$$

$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$

$$= \begin{bmatrix} -1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 5 & 3 \\ 7 & 7 \end{bmatrix}$$

$$= \begin{bmatrix} -5 & -3 \\ 17 & 13 \end{bmatrix}$$

$$V = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$= \begin{bmatrix} -1 & -3 \\ -4 & -2 \end{bmatrix} \begin{bmatrix} 1 & 6 \\ 7 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} -22 & -15 \\ -18 & -30 \end{bmatrix}$$

Matrix Multiplication

Example 2

Now, Compute C_{11} , C_{12} , C_{21} , and C_{22} :

$$C_{11} = P + S - T + V$$

$$C_{11} = \begin{bmatrix} 64 & 45 \\ 90 & 67 \end{bmatrix} + \begin{bmatrix} -9 & -5 \\ -20 & 4 \end{bmatrix} \cdot \begin{bmatrix} 12 & 11 \\ 24 & 16 \end{bmatrix} + \begin{bmatrix} -22 & -15 \\ -18 & -30 \end{bmatrix} = \begin{bmatrix} 21 & 14 \\ 28 & 25 \end{bmatrix}$$

$$C_{12} = R + T$$

$$C_{12} = \begin{bmatrix} 8 & 4 \\ 7 & 2 \end{bmatrix} + \begin{bmatrix} 12 & 11 \\ 24 & 16 \end{bmatrix} = \begin{bmatrix} 20 & 15 \\ 31 & 18 \end{bmatrix}$$

Matrix Multiplication

Example 2

Now, Compute C_{11} , C_{12} , C_{21} , and C_{22} :

$$C_{21} = Q + S$$

$$C_{21} = \begin{bmatrix} 38 & 28 \\ 67 & 47 \end{bmatrix} + \begin{bmatrix} -9 & -5 \\ -20 & 4 \end{bmatrix} = \begin{bmatrix} 29 & 23 \\ 47 & 51 \end{bmatrix}$$

$$C_{22} = P + R - Q + U$$

$$C_{22} = \begin{bmatrix} 64 & 45 \\ 90 & 67 \end{bmatrix} + \begin{bmatrix} 8 & 4 \\ 7 & 2 \end{bmatrix} \cdot \begin{bmatrix} 38 & 28 \\ 67 & 47 \end{bmatrix} + \begin{bmatrix} -5 & -3 \\ 17 & 13 \end{bmatrix} = \begin{bmatrix} 29 & 18 \\ 47 & 35 \end{bmatrix}$$

Matrix Multiplication

Example 2

So the values of C_{11} , C_{12} , C_{21} , and C_{22} are:

$$C_{11} = \begin{bmatrix} 21 & 14 \\ 28 & 25 \end{bmatrix}, C_{12} = \begin{bmatrix} 20 & 15 \\ 31 & 18 \end{bmatrix}, C_{21} = \begin{bmatrix} 29 & 23 \\ 47 & 51 \end{bmatrix} \text{ and } C_{22} = \begin{bmatrix} 29 & 18 \\ 47 & 35 \end{bmatrix}$$

Hence the resultant Matrix C is =

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} 21 & 14 & 20 & 15 \\ 28 & 25 & 31 & 18 \\ 29 & 23 & 29 & 18 \\ 47 & 51 & 47 & 35 \end{bmatrix}$$

Thank U

Design and Analysis of Algorithm

Divide and Conquer strategy (Convex Hull Problem)

Lecture -25

Overview

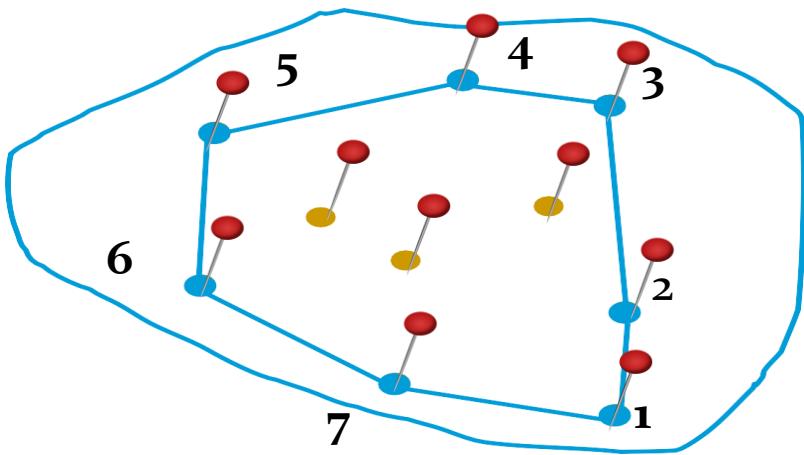
- *Learn the implementation techniques of “divide and conquer” in the context of the Convex Hull Problem with analysis.*

Convex Hull

- Given a set of pins on a pinboard, and a rubber band around them .
- How does the rubber band look when it snaps tight? Just imagine.

Convex Hull

- Given a set of pins on a pinboard, and a rubber band around them .
- How does the rubber band look when it snaps tight? Just imagine.

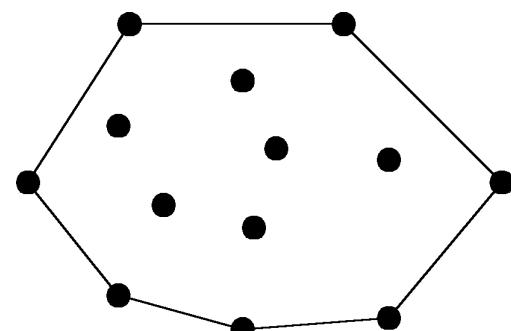


- We represent the convex hull as the sequence of points on the convex hull polygon, in counter-clockwise order.

Convex Hull

- Definition:

- Informal: Convex hull of a set of points in plane is the shape taken by a rubber band stretched around the nails pounded into the plane at each point.
- Formal: The convex hull of a set of planar points is the smallest convex polygon containing all of the points.



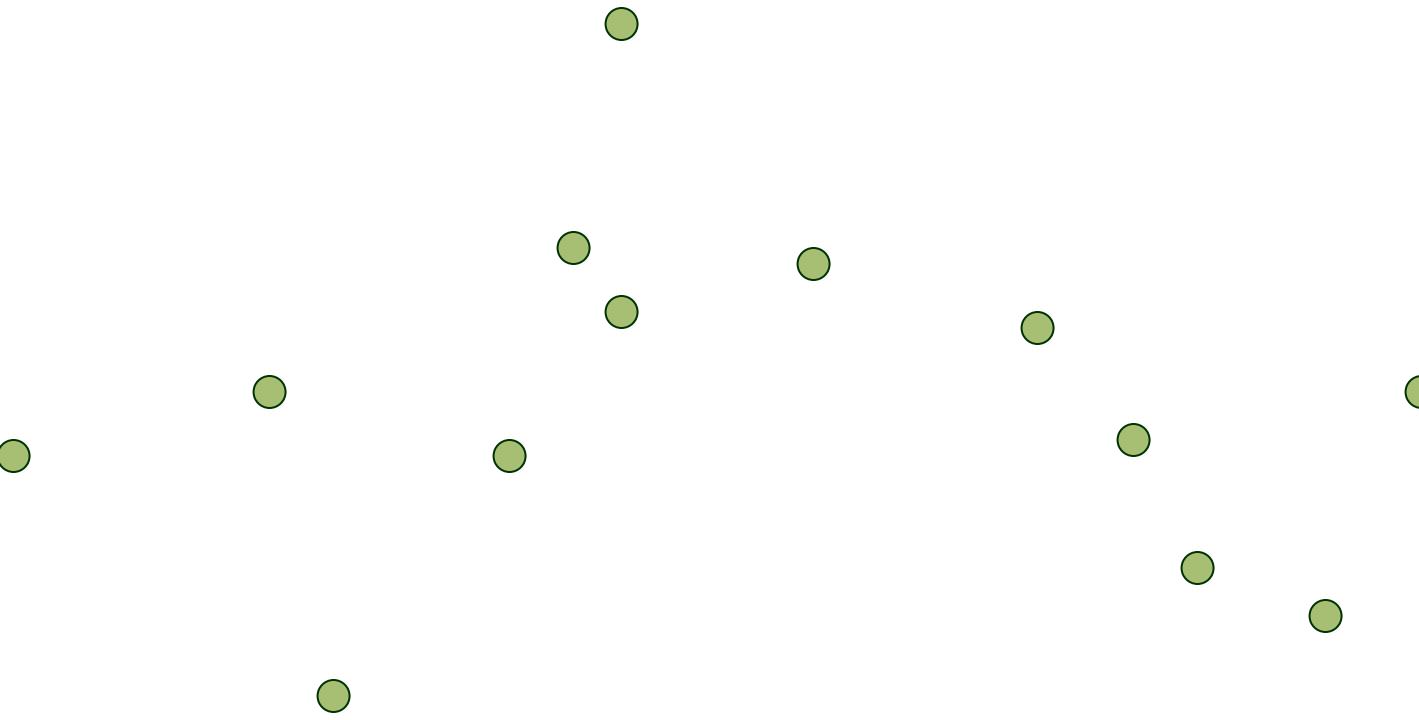
Graham Scan

- Concept:
 - Start at point guaranteed to be on the hull.
(the point with the minimum y value)
 - Sort remaining points by polar angles of vertices relative to the first point.
 - Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

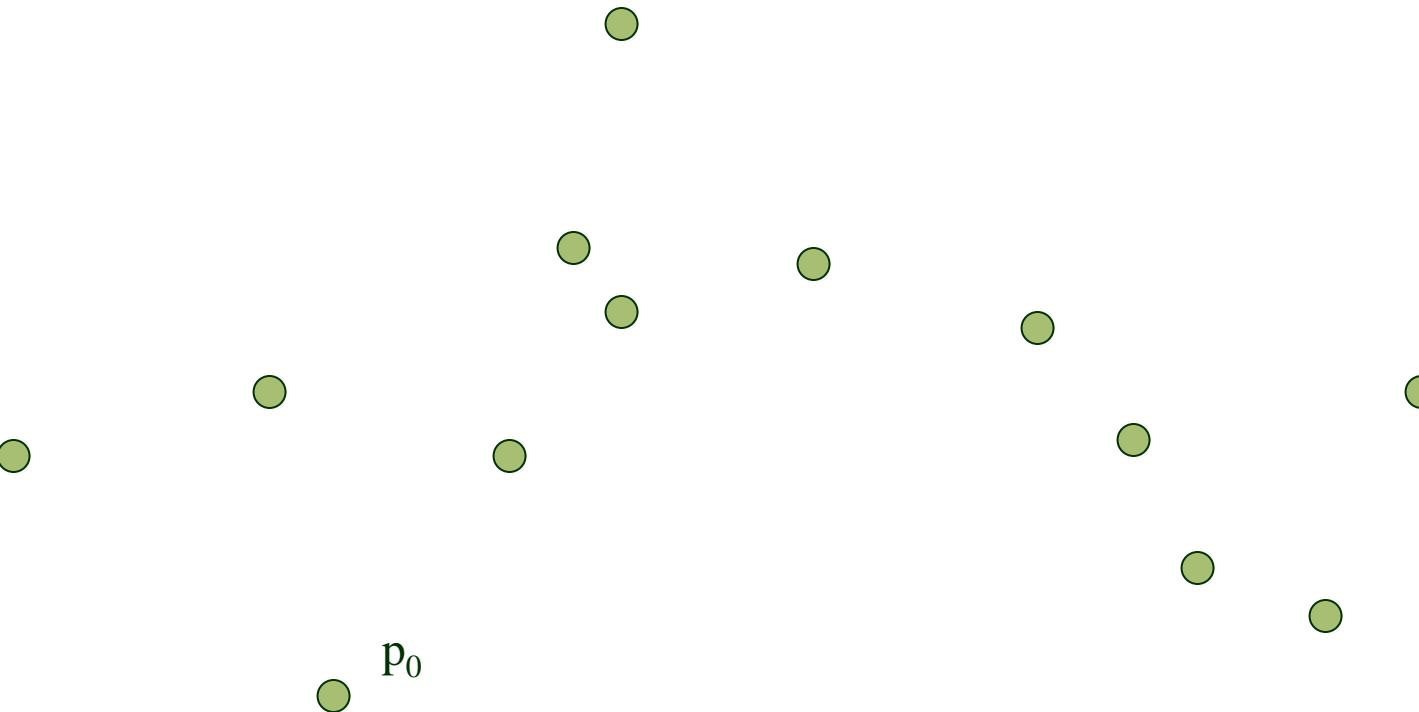
Graham Scan

- Concept:
 - Start at point guaranteed to be on the hull.
(the point with the minimum y value)
 - Sort remaining points by polar angles of vertices relative to the first point.
 - Go through sorted points, keeping vertices of points that have left turns and dropping points that have right turns.

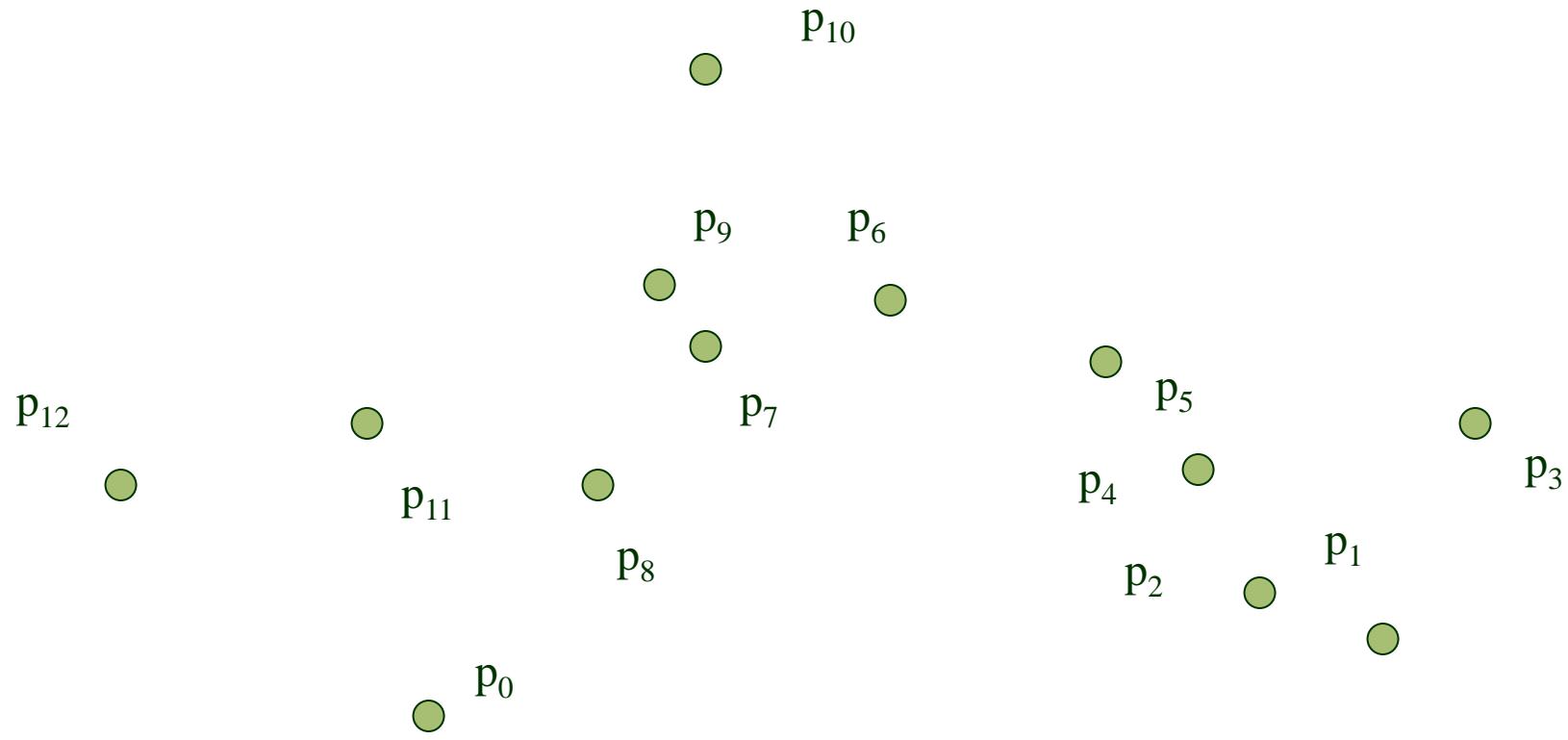
Graham Scan - Example



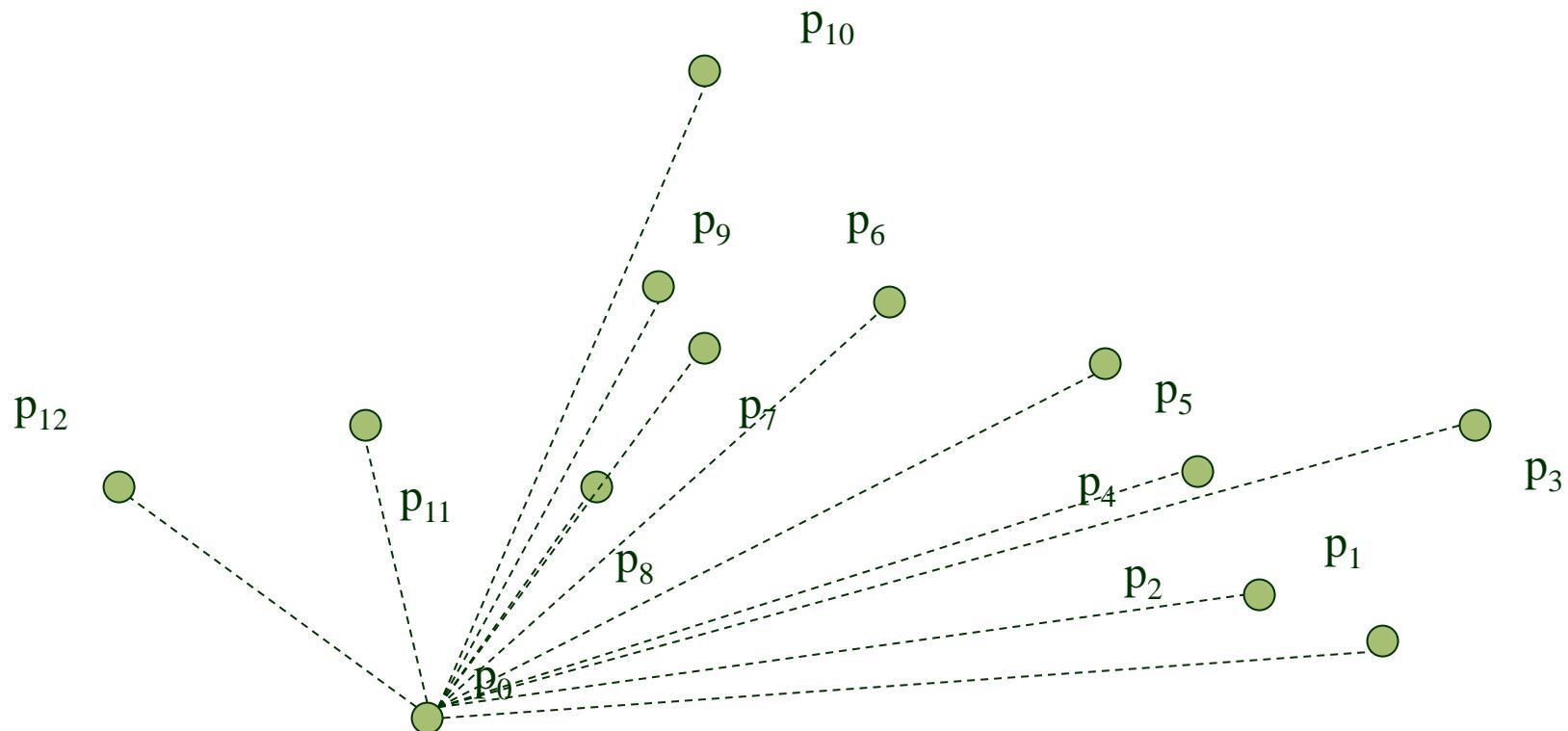
Graham Scan - Example



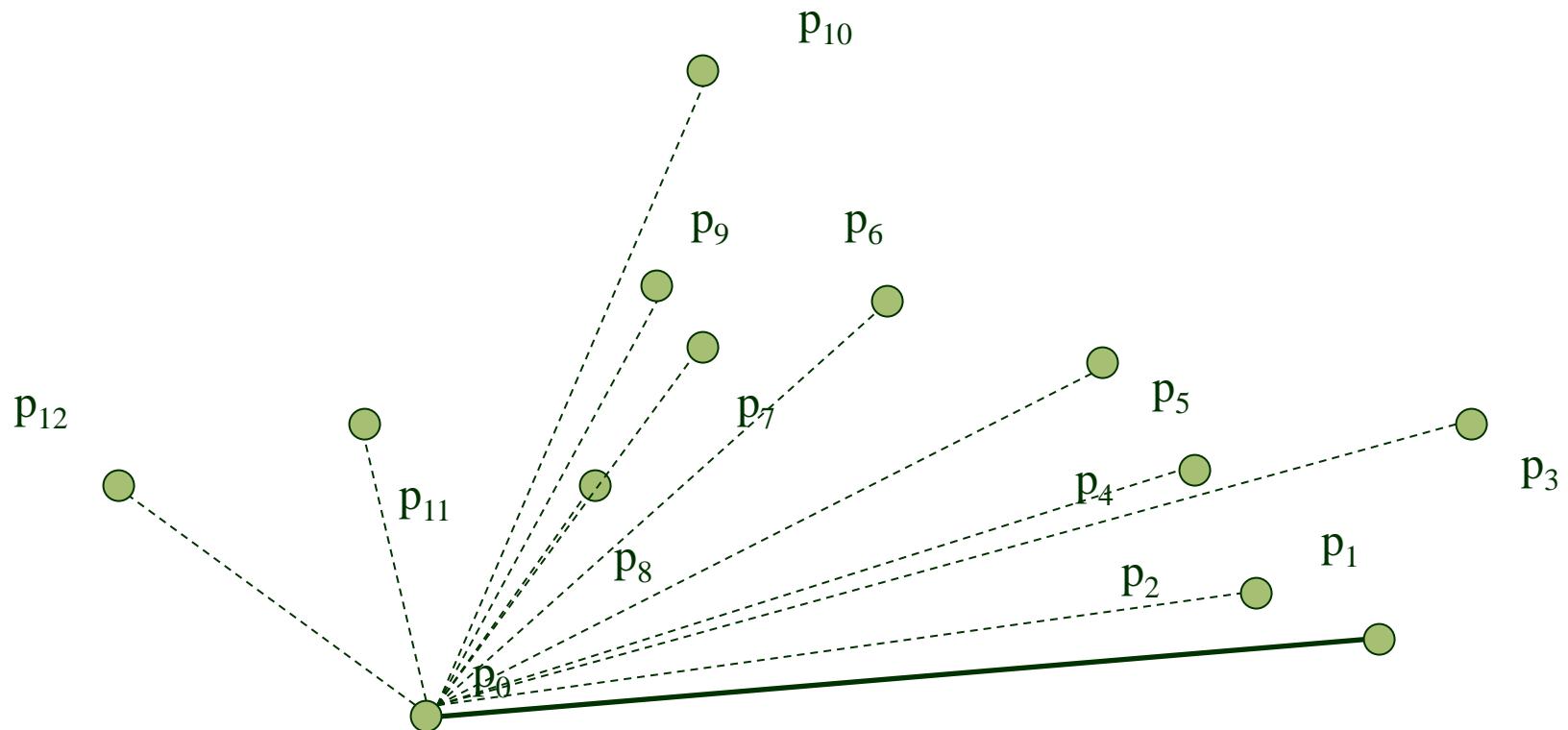
Graham Scan - Example



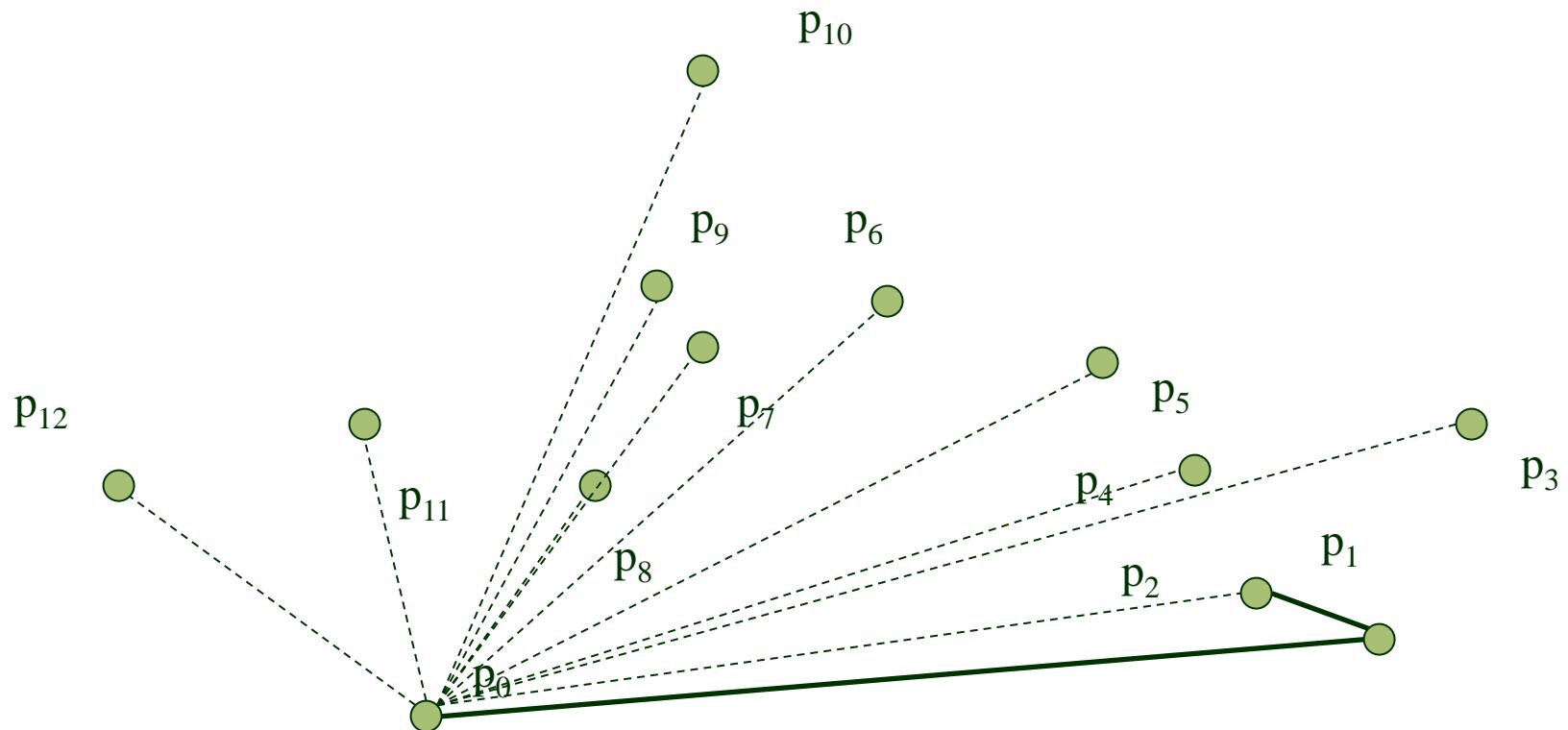
Graham Scan - Example



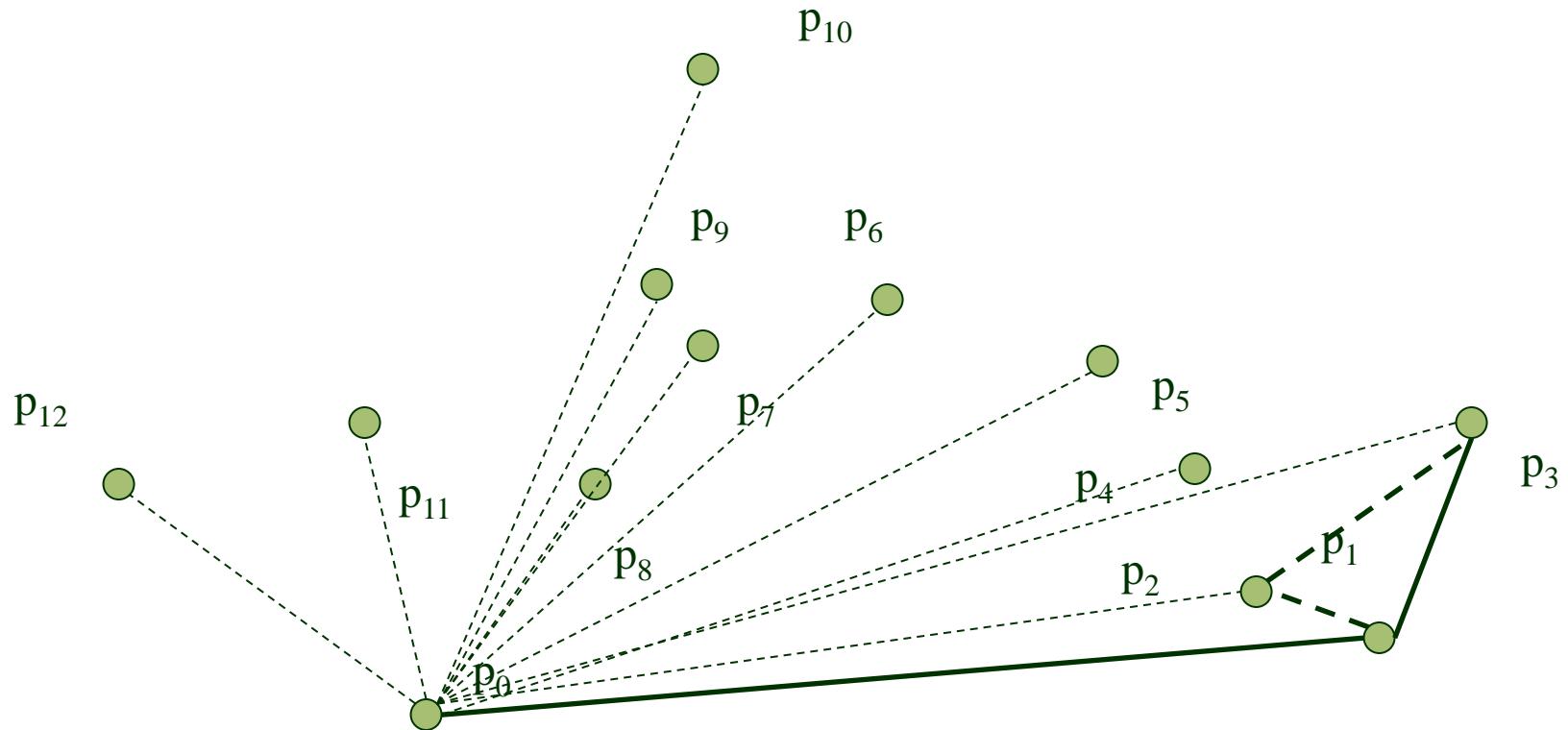
Graham Scan - Example



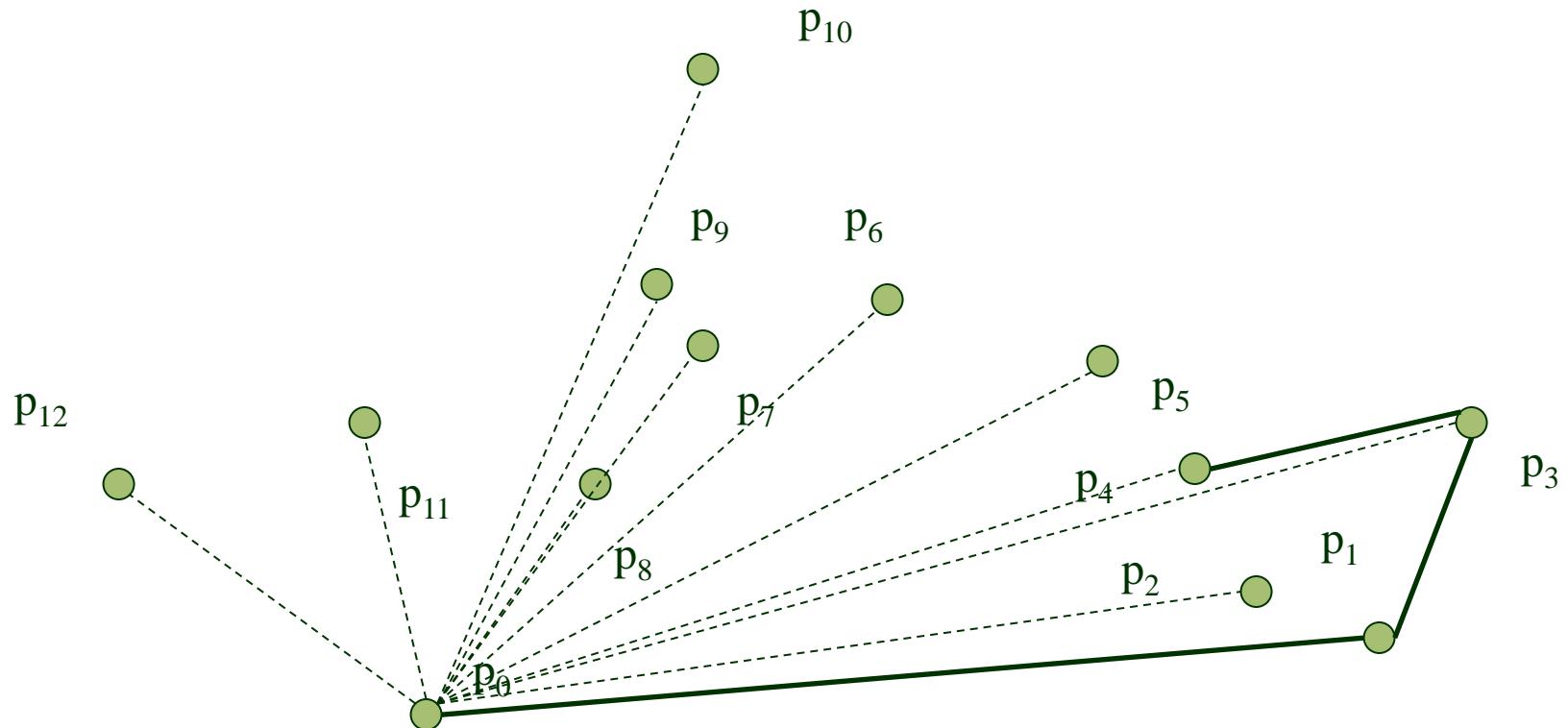
Graham Scan - Example



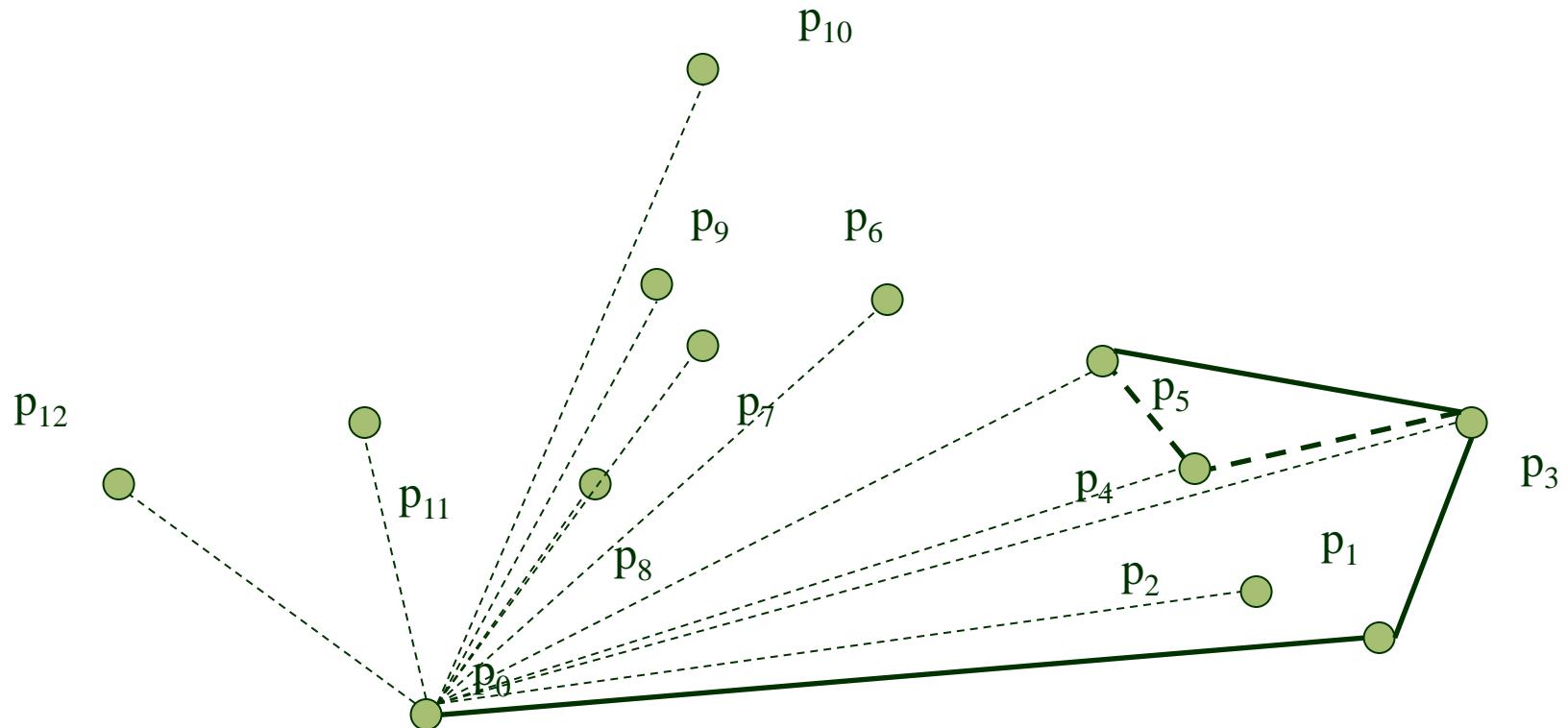
Graham Scan - Example



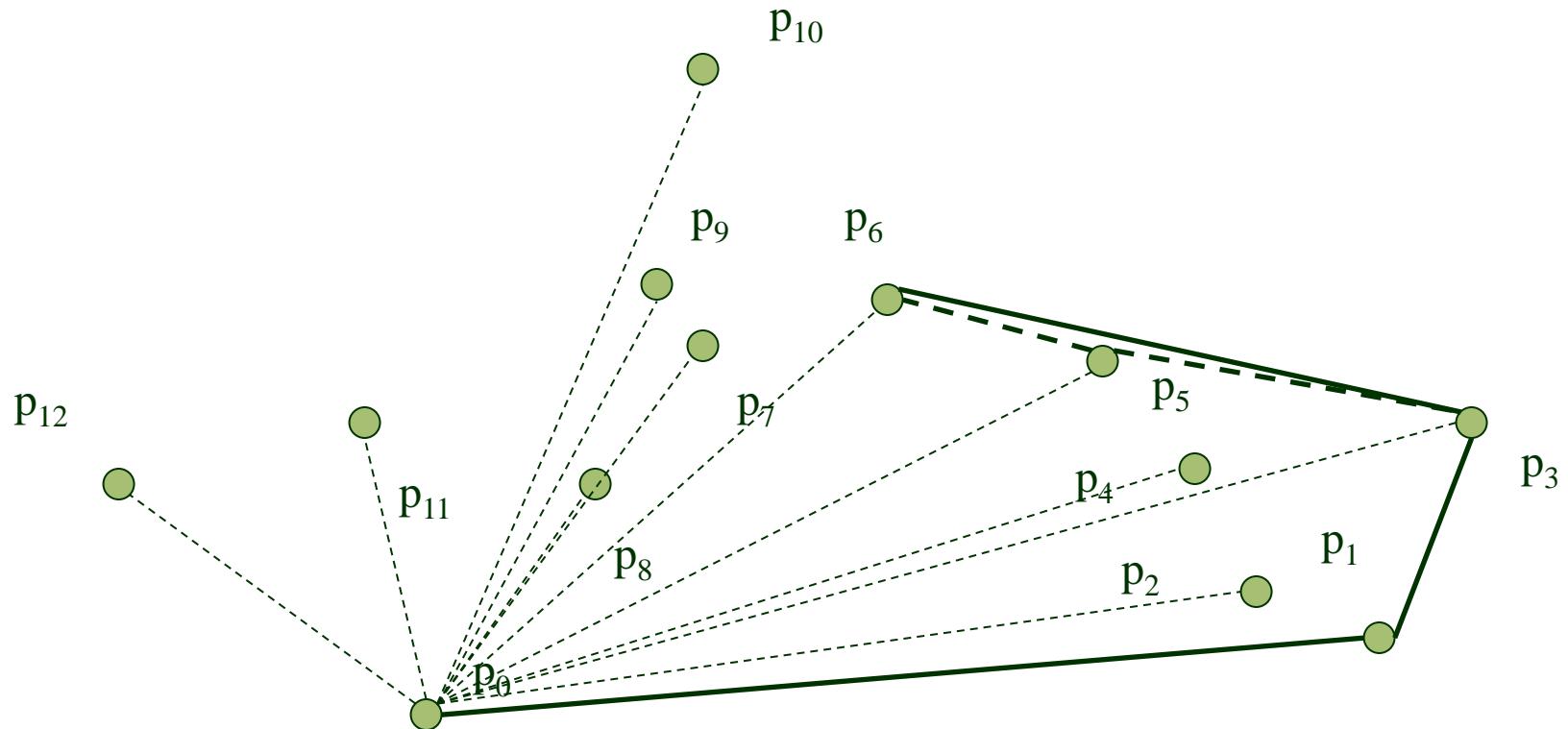
Graham Scan - Example



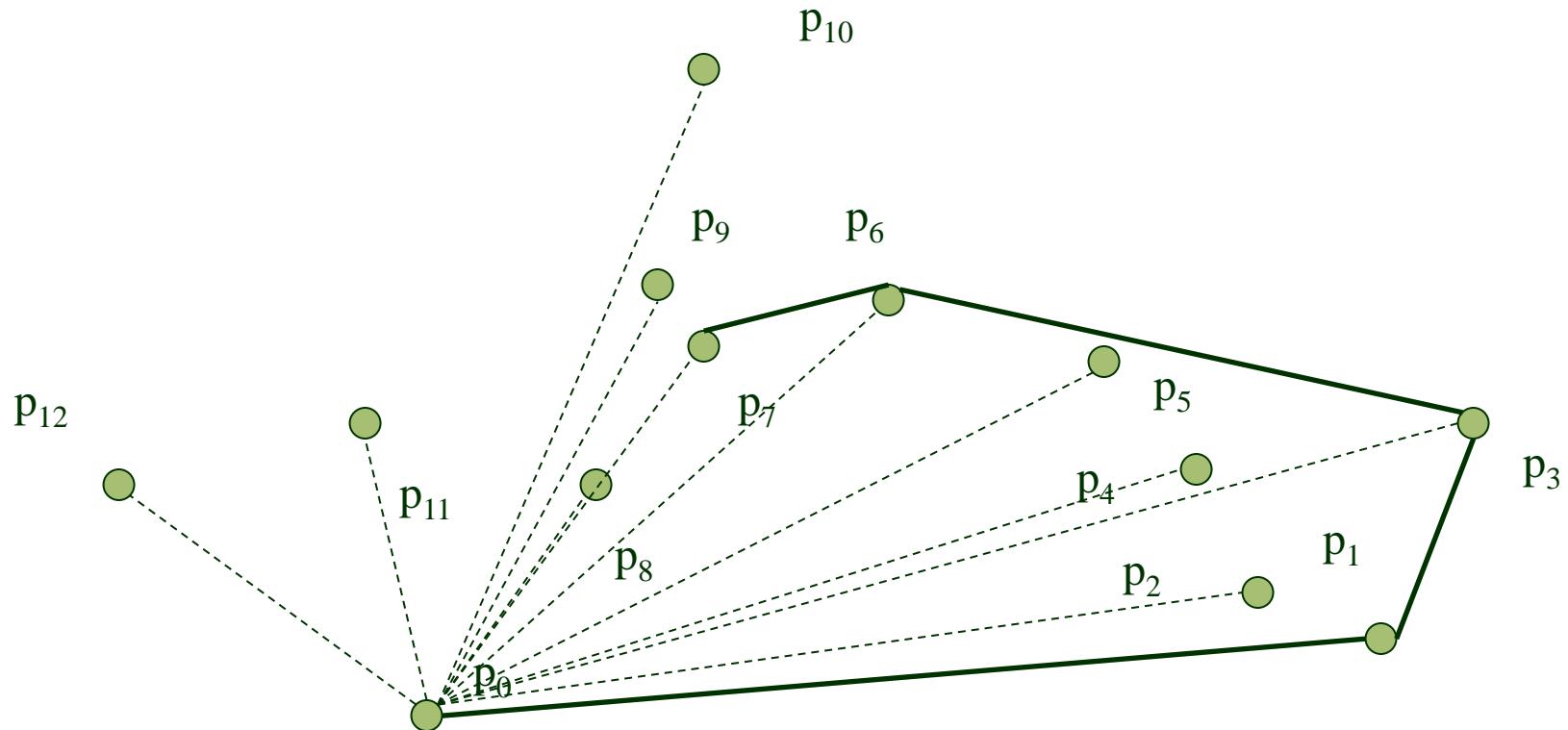
Graham Scan - Example



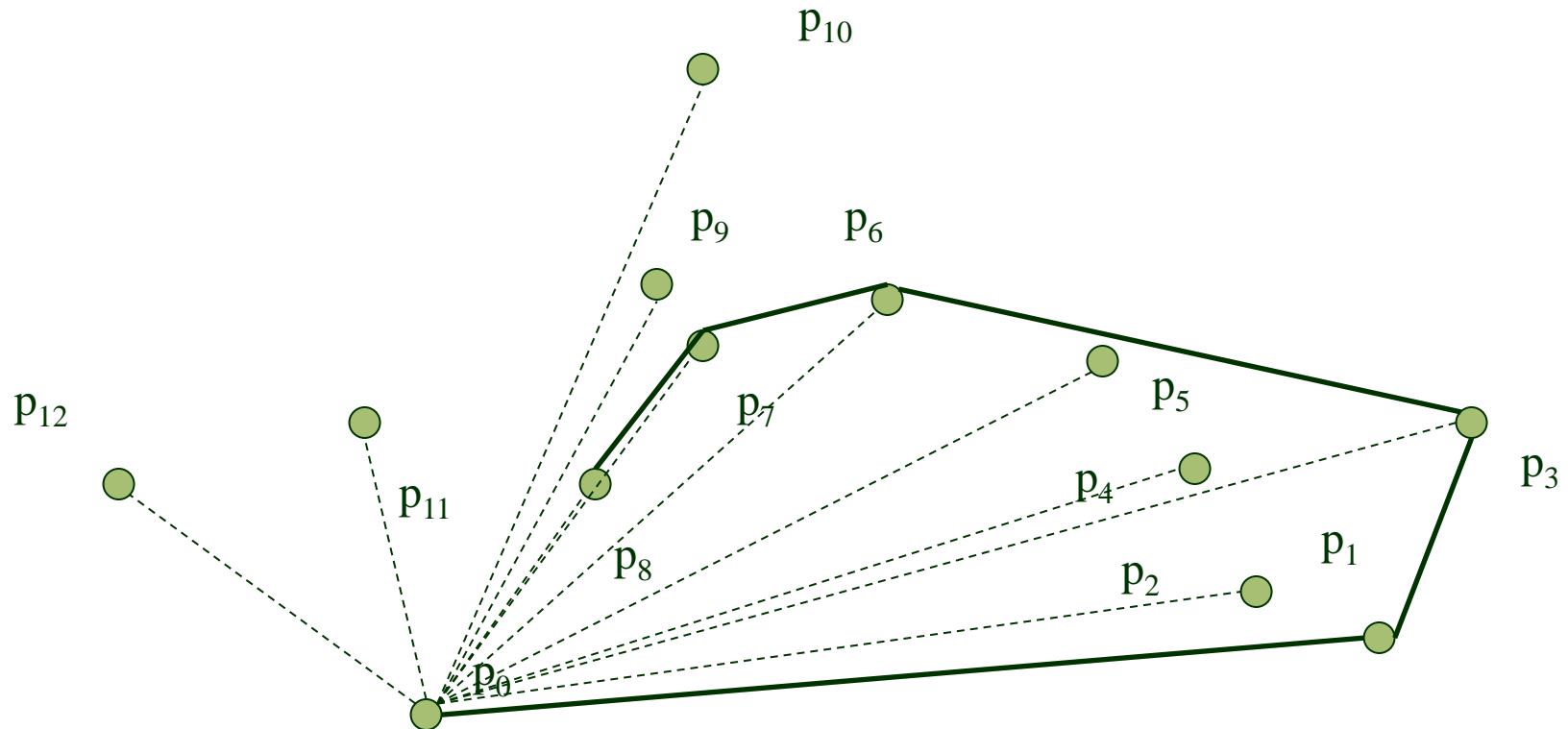
Graham Scan - Example



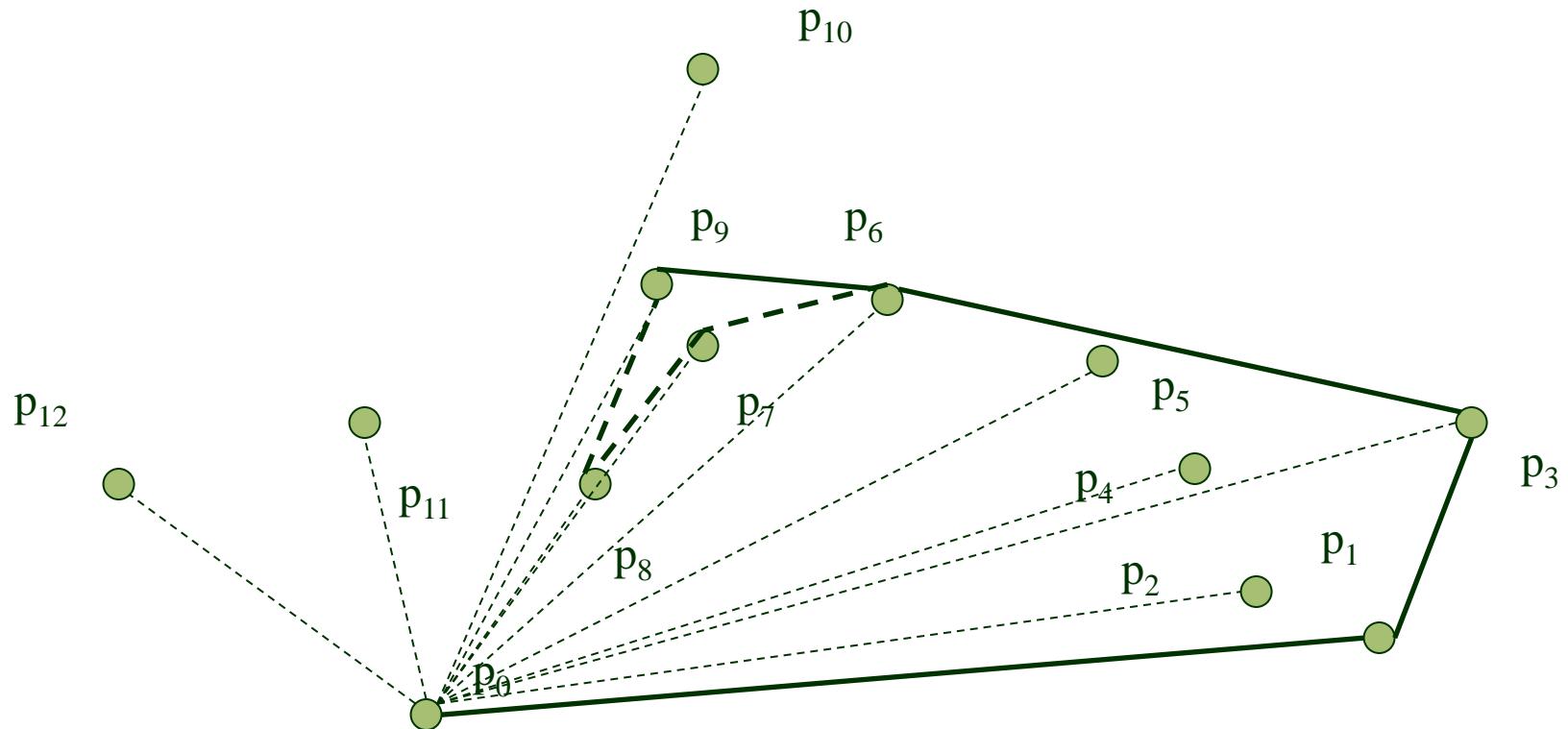
Graham Scan - Example



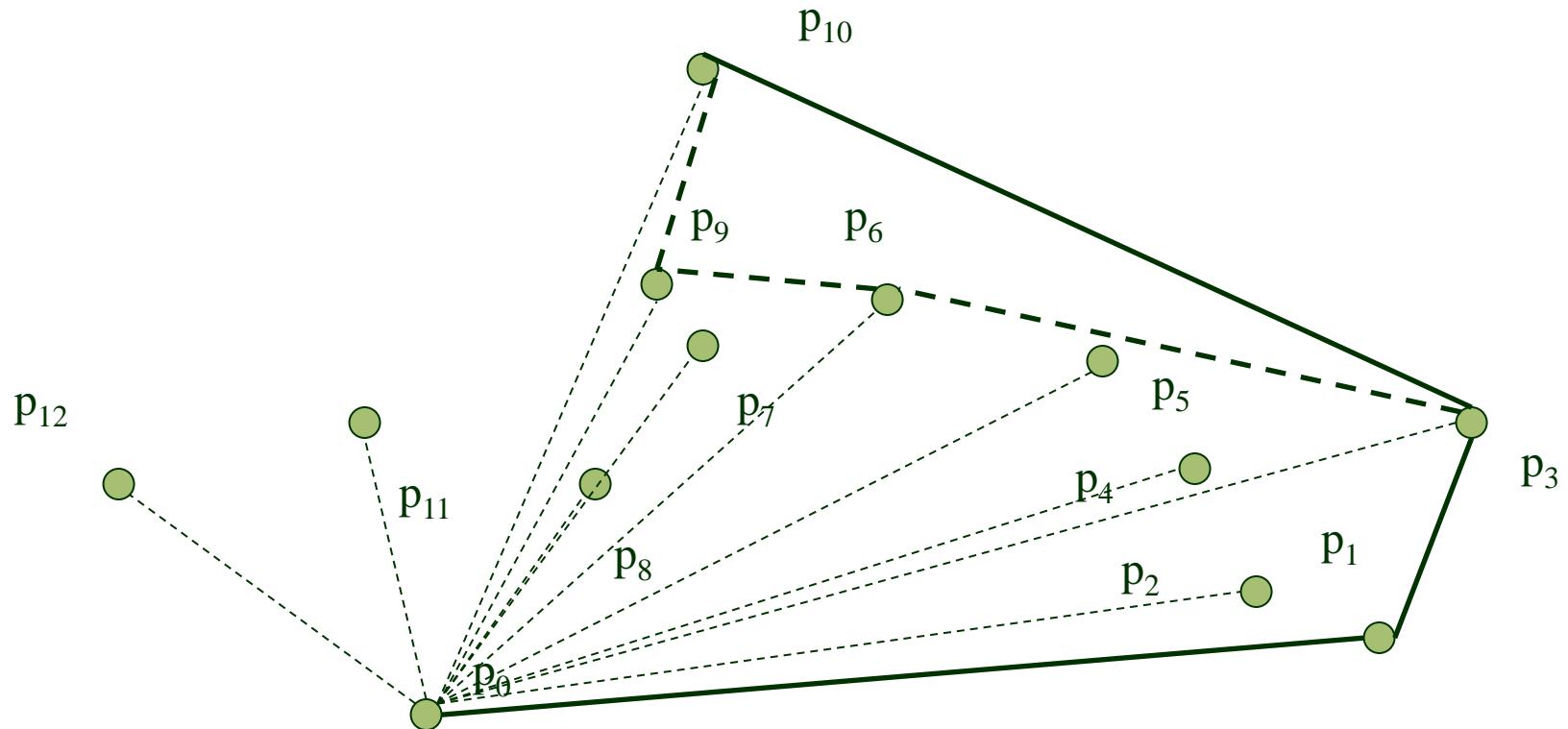
Graham Scan - Example



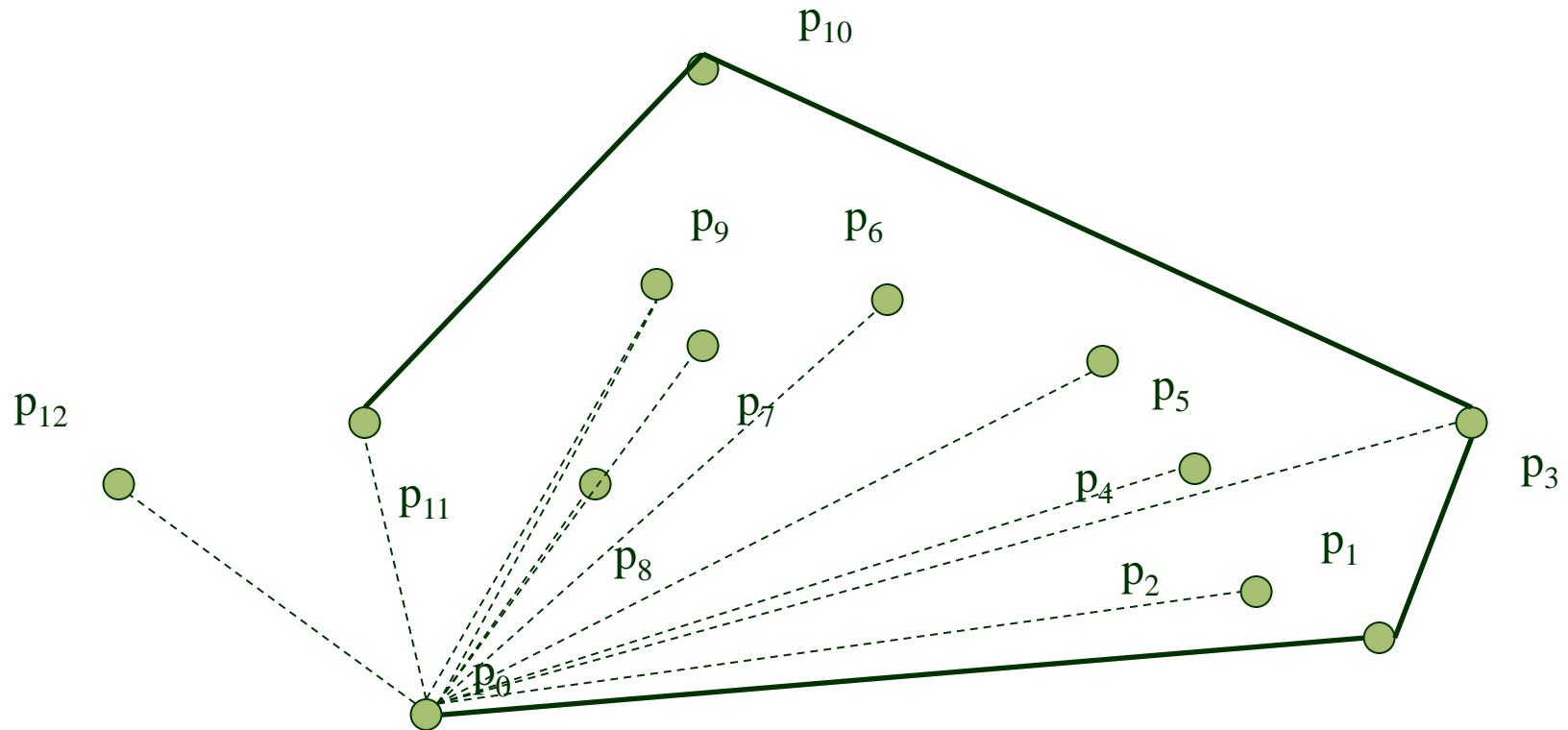
Graham Scan - Example



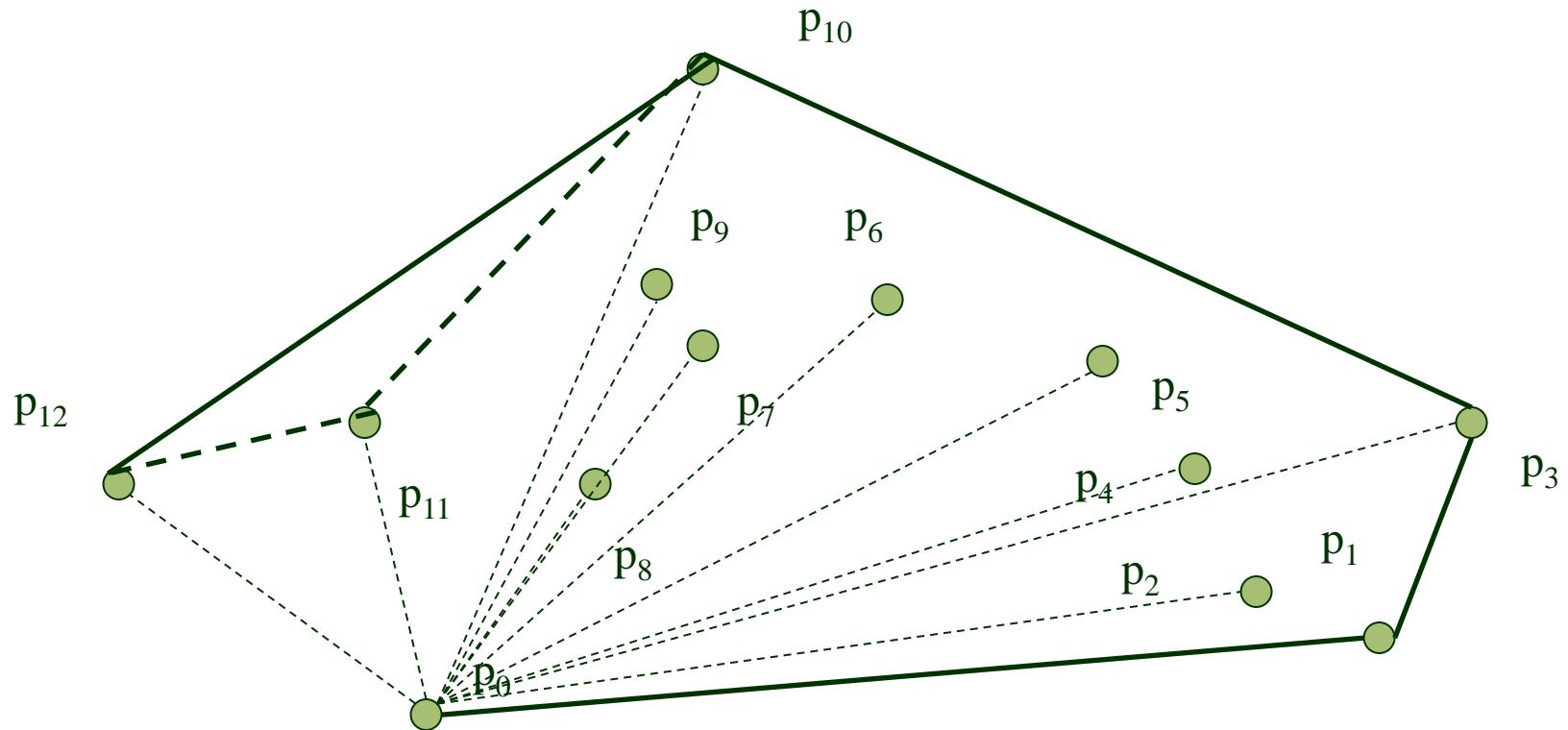
Graham Scan - Example



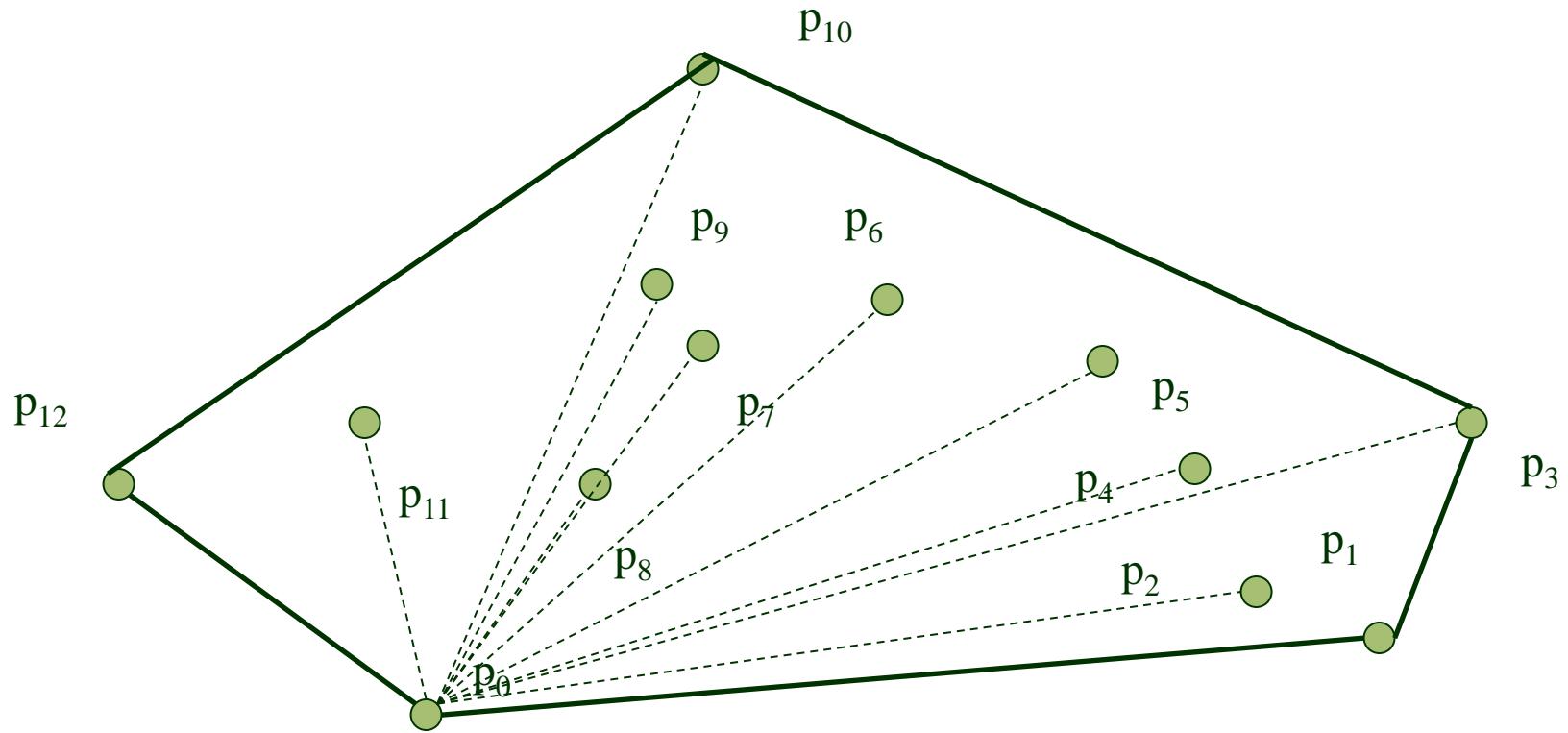
Graham Scan - Example



Graham Scan - Example



Graham Scan - Example



Graham Scan - Algorithm

GRAHAM-SCAN(Q)

- 1 let p_0 be the point in Q with the minimum y -coordinate,
or the leftmost such point in case of a tie
- 2 let $\langle p_1, p_2, \dots, p_m \rangle$ be the remaining points in Q ,
sorted by polar angle in counterclockwise order around p_0
(if more than one point has the same angle, remove all but
the one that is farthest from p_0)
- 3 PUSH(p_0, S)
- 4 PUSH(p_1, S)
- 5 PUSH(p_2, S)
- 6 **for** $i \leftarrow 3$ **to** m
- 7 **do while** the angle formed by points NEXT-TO-TOP(S), TOP(S),
and p_i makes a nonleft turn
- 8 **do** POP(S)
- 9 PUSH(p_i, S)
- 10 **return** S

Graham Scan - Algorithm

GRAHAM-SCAN(Q)

- 1 let p_0 be the point in Q with the minimum y -coordinate,
or the leftmost such point in case of a tie ----- $O(n)$
- 2 let $\langle p_1, p_2, \dots, p_m \rangle$ be the remaining points in Q ,
sorted by polar angle in counterclockwise order around p_0
(if more than one point has the same angle, remove all but
the one that is farthest from p_0) ----- $O(n \log n)$
- 3 PUSH(p_0, S) ----- $O(1)$
- 4 PUSH(p_1, S) ----- $O(1)$
- 5 PUSH(p_2, S) ----- $O(1)$
- 6 **for** $i \leftarrow 3$ **to** m
- 7 **do while** the angle formed by points NEXT-TO-TOP(S), TOP(S),
and p_i makes a nonleft turn ----- $O(n)$
- 8 **do** POP(S)
- 9 PUSH(p_i, S)
- 10 **return** S

Graham Scan - Algorithm

GRAHAM-SCAN(Q)

- 1 let p_0 be the point in Q with the minimum y -coordinate,
or the leftmost such point in case of a tie $\text{----- } O(n)$
- 2 let $\langle p_1, p_2, \dots, p_m \rangle$ be the remaining points in Q ,
sorted by polar angle in counterclockwise order around p_0
(if more than one point has the same angle, remove all but
the one that is farthest from p_0) $\text{----- } O(n \log n)$
- 3 PUSH(p_0, S) $\text{----- } O(1)$
- 4 PUSH(p_1, S) $\text{----- } O(1)$
- 5 PUSH(p_2, S) $\text{----- } O(1)$
- 6 **for** $i \leftarrow 3$ **to** m
- 7 **do while** the angle formed by points NEXT-TO-TOP(S), TOP(S),
and p_i makes a nonleft turn
- 8 **do** POP(S)
- 9 PUSH(p_i, S)
- 10 **return** S

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

$$T(n) = O(n \lg n) + O(n)$$

Graham Scan - Algorithm

- Time complexity of Graham's scan:
 - $O(n \log n)$ time required to sort of angles in step 2.
 - $O(n)$ time required for visiting n points.(step 6 to step 9)

Hence we can write the complexity of Graham's scan as:

$$T(n) = O(n \lg n) + O(n) = O(n \lg n)$$

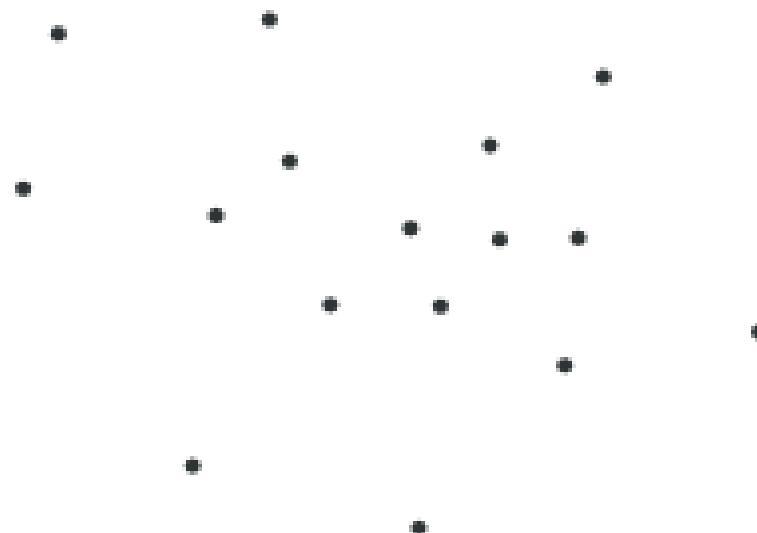
Divide and Conquer (Quickhull)

- QuickHull uses a Divide and Conquer approach similar to the Quick Sort algorithm.
- Benchmarks showed it is quite fast in most average cases.
- Recursive nature allows a fast and yet clean implementation.

Divide and Conquer (Quickhull)

Initial Input

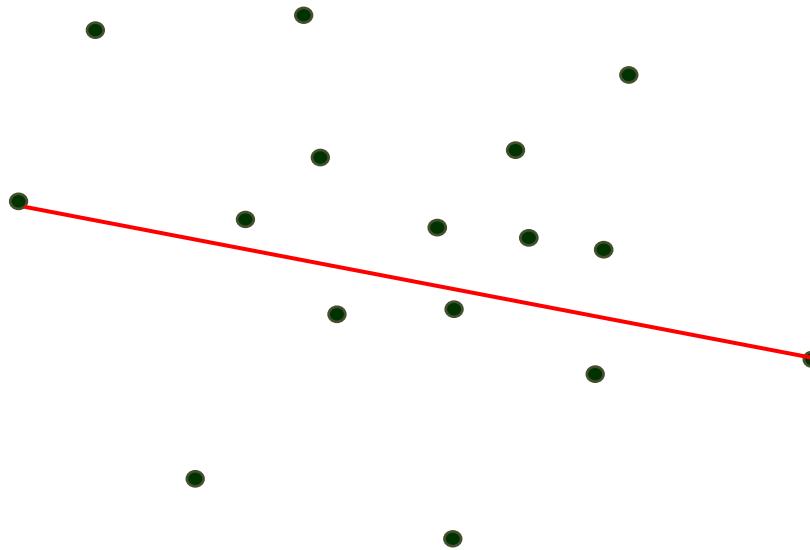
- The initial input to the algorithm is an arbitrary set of points as shown in the figure.



Divide and Conquer (Quickhull)

First Two Points on the Convex Hull

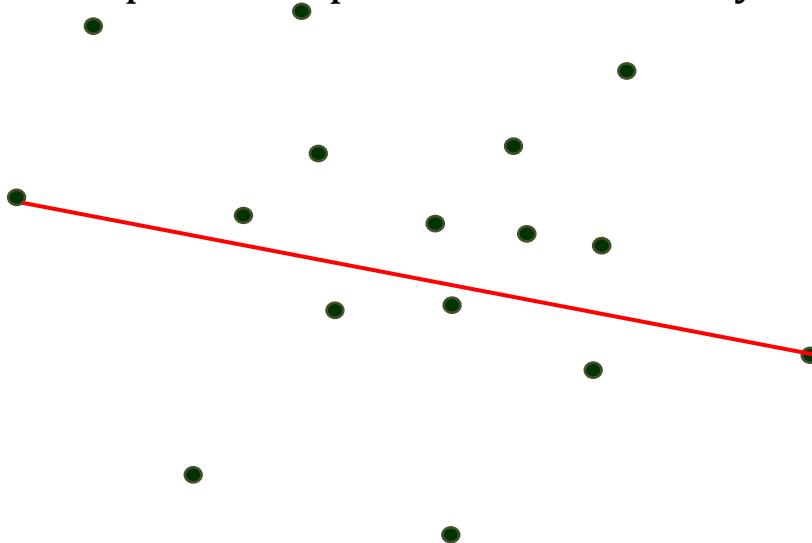
- Starting with the given set of points the first operation done is the calculation of the two maximal points on the horizontal axis. i.e.



Divide and Conquer (Quickhull)

Recursively Divide

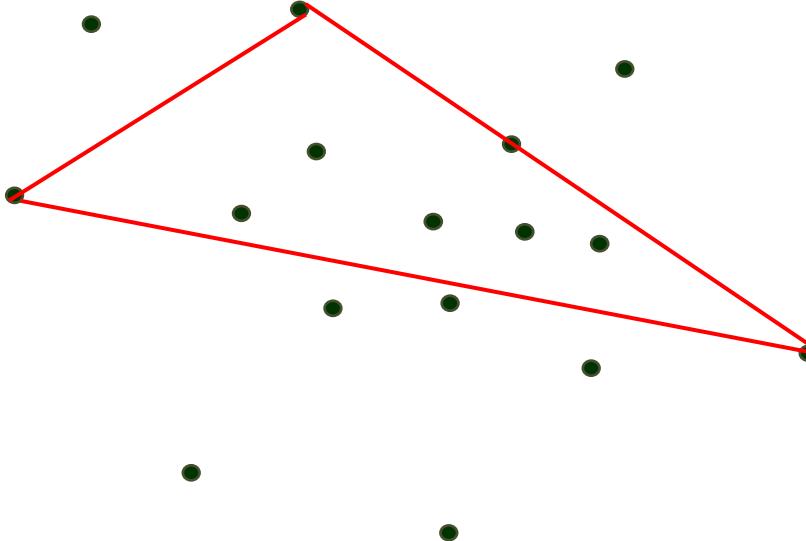
- Next the line formed by these two points is used to divide the set into two different parts.
- Everything left from this line is considered one part, everything right of it is considered another one.
- Both of these parts are processed recursively.



Divide and Conquer (Quickhull)

Max Distance Search

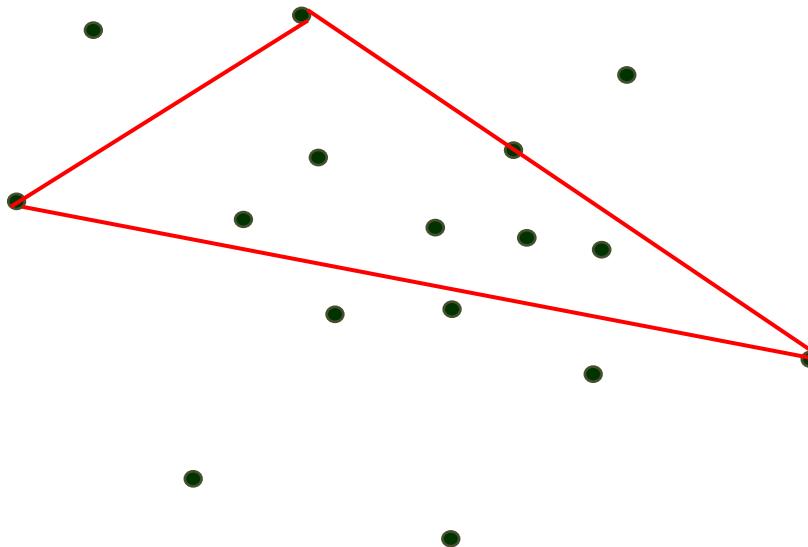
- To determine the next point on the convex hull a search for the point with the greatest distance from the dividing line is done.
- This point, together with the line start and end point forms a triangle.



Divide and Conquer (Quickhull)

Point Exclusion

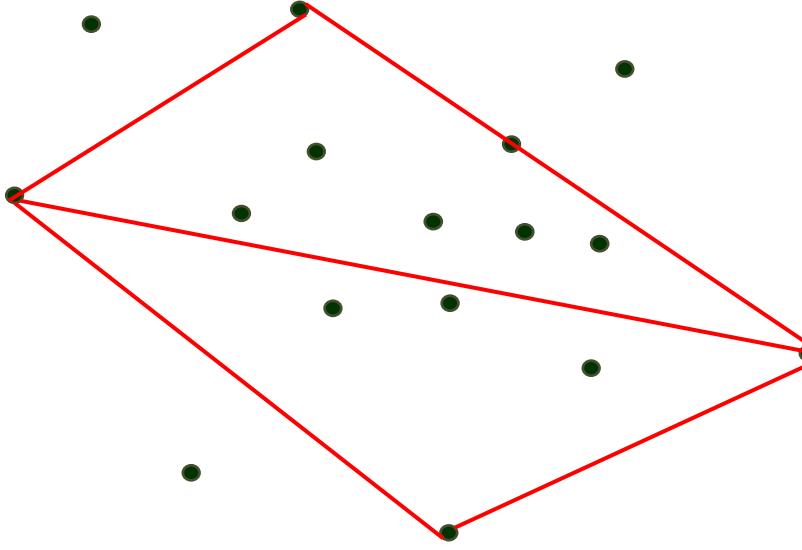
- All points inside this triangle can not be part of the convex hull polygon, as they are obviously lying in the convex hull of the three selected points.
- Therefore, these points can be ignored for every further processing step.



Divide and Conquer (Quickhull)

Recursively Divide

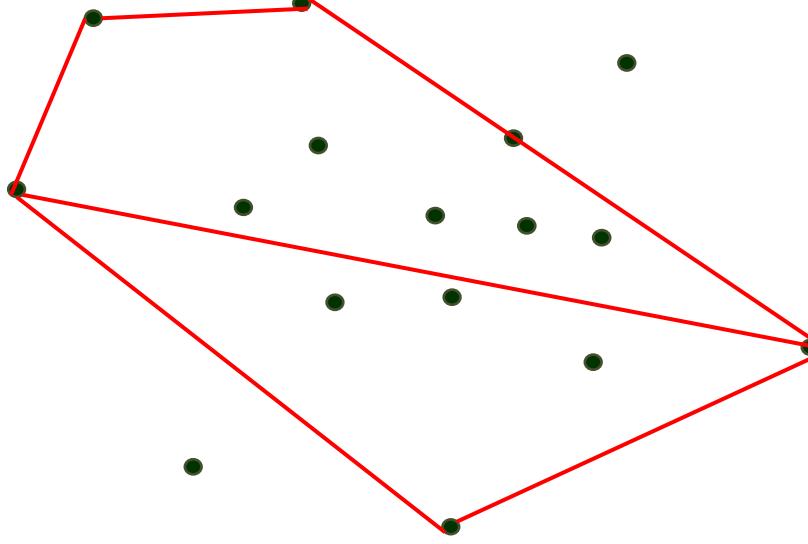
- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



Divide and Conquer (Quickhull)

Recursively Divide

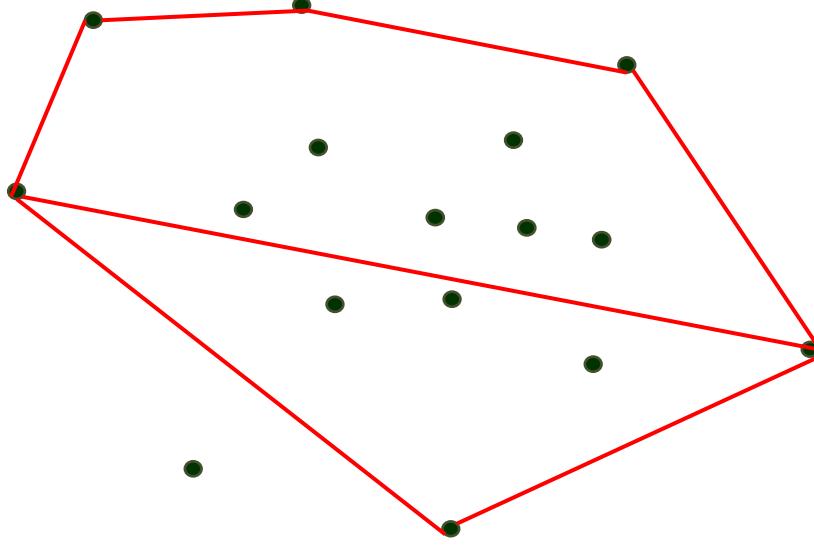
- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



Divide and Conquer (Quickhull)

Recursively Divide

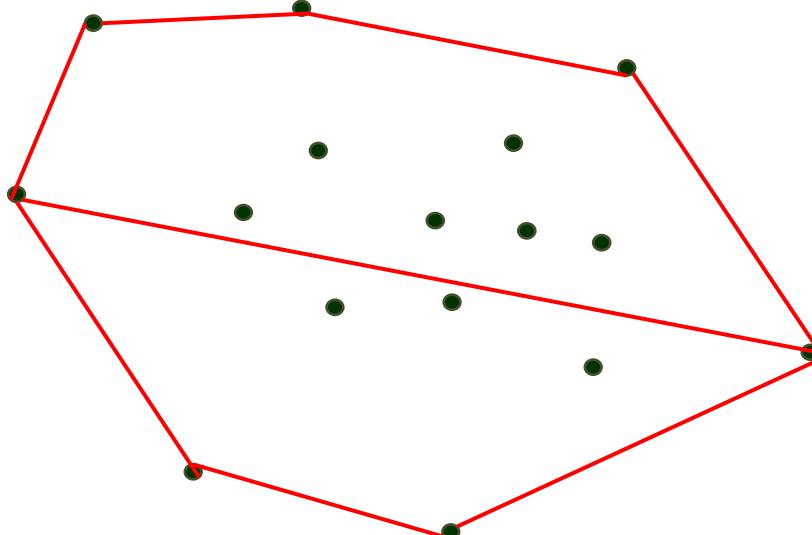
- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



Divide and Conquer (Quickhull)

Recursively Divide

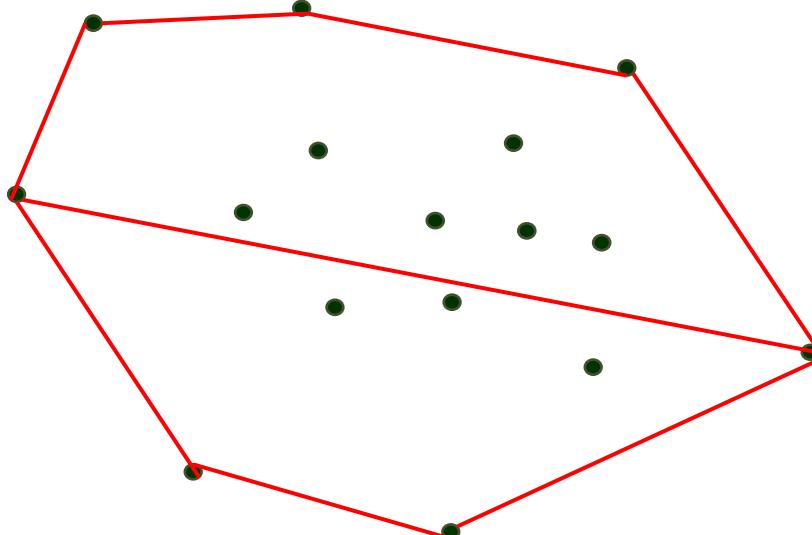
- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



Divide and Conquer (Quickhull)

Recursively Divide

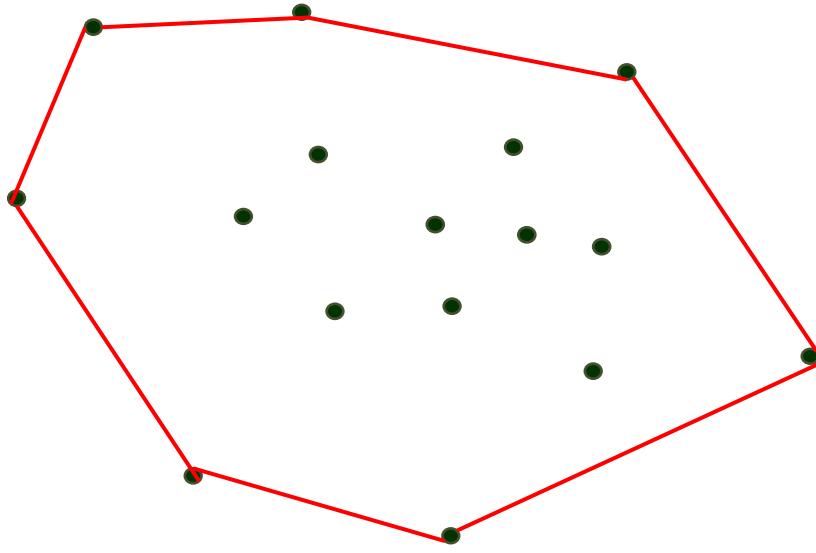
- Having this in mind the recursive processing can take place again.
- Everything right of the triangle is used as one subset, everything left of it as another one.



Divide and Conquer (Quickhull)

Abort Condition

- At some point the recursively processed point subset does only contain the start and end point of the dividing line.
- If this is case this line has to be a segment of the searched hull polygon and the recursion can come to an end.



Divide and Conquer (Quickhull)

Algorithm

Quick Hull(S)

//Find convex hull from the set S on n points Convex
Hull= {}//

1. Find left and right most points, say A & B and add \overline{AB} to Convex Hull.
2. Segment \overline{AB} divides the remaining (n-2) points into two groups S_1 and S_2 . Where S_1 are points in S that are on the right side of the oriented line from A to B. And S_2 are points in S that are on the right side of the oriented line from B to A.
3. Find Hull (S_1 , A, B)
4. Find Hull (S_2 , B, A)

Divide and Conquer (Quickhull)

Algorithm

Find Hull(S_k , P, Q)

//{Find points on Convex Hull from the set S_k points, that are on the right side of the oriented from P to Q}//

If S_k has no point

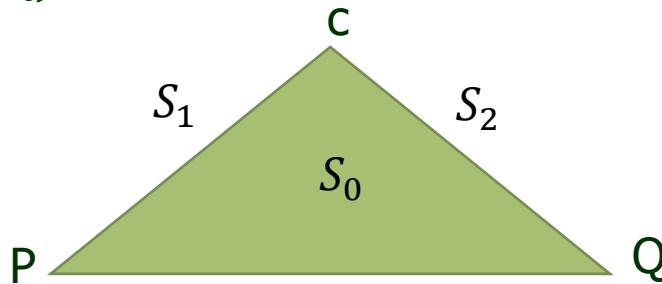
 then return

- From the given set of points in S_k , find farthest point, say C. from segment PQ.
- Add point C to the Convex Hull at the location between P and Q. Three points P, Q, and C partition the remaining points of S_k into three subsets S_0 , S_1 , and S_2 .

Divide and Conquer (Quickhull)

Algorithm

- Where S_0 are points inside triangle PCQ, and S_1 are points on the right side of the oriented line from P to C, and S_2 are the points on the right side of the oriented line from C to Q.
- Find Hull (S_1, P, C)
- Find Hull (S_2, C, Q)



Divide and Conquer (Quickhull)

Time Complexity of Quickhull

- The running time of Quickhull, as with QuickSort, depends on how evenly the points are split at each stage.
- If we assume that the points are ``evenly'' distributed, the running time will solve to $O(n \log n)$.
- if the splits are not balanced, then the running time can easily increase to $O(n^2)$.

Divide and Conquer (Quickhull)

Time Complexity of Quickhull

$$T(n) = T(l) + T(n - l) + O(n)$$

Where,

- $T(l)$ → Point in left side of AB.
- $T(n - l)$ → Point in right side of AB.
- $O(n)$ → To find the farthest point.

Assume that $T(l)$ contain $(n/2)$ points and $T(n-l)$ contain $(n/2)$ points.
Hence ,

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

After applying Master Method

$$T(n) = \Theta(n \lg n) \text{ in average case}$$
$$T(n) = \Theta(n^2) \text{ in worst case.}$$

Thank U

Algorithm Analysis and Design

Recurrence Equation (Solving Recurrence using Master Method)

Lecture – 26 and 27

Overview

- A **recurrence** is a function is defined in terms of
 - one or more base cases, and
 - itself, with smaller arguments.

Examples:

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n - 1) + 1 & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = n$.

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

Solution: $T(n) = n \lg n + n$.

$$\bullet \quad T(n) = \begin{cases} 0 & \text{if } n = 2 , \\ T(\sqrt{n}) + 1 & \text{if } n > 2 . \end{cases}$$

Solution: $T(n) = \lg \lg n$.

$$\bullet \quad T(n) = \begin{cases} 1 & \text{if } n = 1 , \\ T(n/3) + T(2n/3) + n & \text{if } n > 1 . \end{cases}$$

Solution: $T(n) = \Theta(n \lg n)$.

Overview

- Many technical issues:
 - Floors and ceilings

[Floors and ceilings can easily be removed and don't affect the solution to the recurrence. They are better left to a discrete math course.]
 - Exact vs. asymptotic functions
 - Boundary conditions

Overview

In algorithm analysis, the recurrence and it's solution are expressed by the help of asymptotic notation.

- Example: $T(n) = 2T(n/2) + \Theta(n)$, with solution $T(n) = \Theta(n \lg n)$.
 - The boundary conditions are usually expressed as $T(n) = O(1)$ for sufficiently small n .
 - But when there is a desire of an exact, rather than an asymptotic, solution, the need is to deal with boundary conditions.
 - In practice, just use asymptotics most of the time, and ignore boundary conditions.

Recursive Function

- Example

$A(n)$

{

If($n > 1$)

Return $\left(A\left(\frac{n}{2}\right) \right)$

}

The relation is called recurrence relation

The Recurrence relation of given function is written as follows.

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

Recursive Function

- To solve the Recurrence relation the following methods are used:
 1. Iteration method
 2. Recursion-Tree method
 - 3. Master Method**
 4. Substitution Method

Master Method

The master method provides a "cookbook" method for solving recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. The master method requires memorization of three cases.

"The beauty of Master Method is the solution of many recurrences can be determined quite easily, often without pencil and paper."

Master Method

- Definition

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 $T(n)$ can be bounded asymptotically as follows.

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 $af\left(\frac{n}{b}\right) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$

Master Method

Example 1

Solve the following recurrence by using Master Method

$$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Master Method

Example 1

Solve the following recurrence by using Master Method

$$T(n) = 5T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Hear, $a = 5, b = 2$ and $f(n) = n^2$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_2 5} = n^{2.32}$$

$$\text{but, } f(n) = n^2$$

$$\text{Therefore, } n^{\log_b a} > f(n), \quad (\varepsilon = 0.32)$$

Hence as per the definition of master theorem Case 1

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{2.32})$$

Master Method

Example 2

Solve the following recurrence by using Master Method

$$T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n)$$

Master Method

Example 2

Solve the following recurrence by using Master Method

$$T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n)$$

Hear, $a = 9, b = 3$ and $f(n) = n$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_3 9} = n^2$$

but, $f(n) = n$

$$\text{Therefore, } n^{\log_b a} > f(n), \quad (\varepsilon = 1)$$

Hence as per the definition of master theorem Case 1

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

Master Method

Example 3

Solve the following recurrence by using Master Method

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

Master Method

Example 3

Solve the following recurrence by using Master Method

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

Hear, $a = 1, b = \frac{3}{2}$ and $f(n) = 1$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

and, $f(n) = 1$

Therefore, $n^{\log_b a} = f(n)$,

Hence as per the definition of master theorem Case 2

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\lg n)$$

Master Method

Example 4

Solve the following recurrence by using Master Method

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

Master Method

Example 4

Solve the following recurrence by using Master Method

$$T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$$

Hear, $a = 3, b = 4$ and $f(n) = n \lg n$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_4 3} = n^{0.793}$$

but, $f(n) = n \lg n$

Since, $f(n) = \Omega(n^{\log_4 3 + \varepsilon})$ where $\varepsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n ,

$$\Rightarrow af(n/b) \leq cf(n)$$

$$\Rightarrow 3(n/4)\lg(n/4) \leq (3/4)n \lg n \quad (\text{for } c = 3/4)$$

Which is true by Master Method Case 3

Hence, the solution to the recurrence is $T(n) = \Theta(f(n)) = \Theta(n \lg n)$

Master Method

Example 5

Solve the following recurrence by using Master Method

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

Master Method

Example 5

Solve the following recurrence by using Master Method

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

Hear, $a = 2, b = 2$ and $f(n) = n \lg n$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

So, $n^{\log_b a} = n^{\log_2 2} = n^1 = n$

but, $f(n) = n \lg n$

Which looks, $n^{\log_b a} < f(n)$, and we might mistakenly think that case 3 of master method should apply. But The problem is that it is not polynomially larger.

Because the ratio (*i.e.* $\frac{f(n)}{n^{\log_b a}} = \frac{n \log n}{n} = \log n$) is asymptotically less than n^ε for any positive constant ε .

Hence master method is not applicable to the recurrence.

Master Method

Example 6

Solve the following recurrence by using Master Method

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

Master Method

Example 6

Solve the following recurrence by using Master Method

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

Hear, $a = 4, b = 2$ and $f(n) = n$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_2 4} = n^2$$

$$\text{but, } f(n) = n^2$$

$$\text{Therefore, } n^{\log_b a} > f(n), \quad (\varepsilon = 1)$$

Hence as per the definition of master theorem Case 1

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

Master Method

Example 7

Solve the following recurrence by using Master Method

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$$

Master Method

Example 7

Solve the following recurrence by using Master Method

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$$

Here, $a = 2, b = 4$ and $f(n) = \sqrt{n}$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } n^{\log_b a} = n^{\log_4 2} = n^{\frac{1}{2}} = \sqrt{n}$$

and, $f(n) = \sqrt{n}$

Therefore, $n^{\log_b a} = f(n)$,

Hence as per the definition of master theorem Case 2

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(\sqrt{n} \lg n)$$

Master Method

- **Recurrence (Changing Variable)**

Example 8

Solve the following recurrence by using Master Method

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n$$

Due to, a little algebraic manipulation the above recurrence looks very difficult.

These recurrences can be simplified by using change of variable. For convenience, we shall not worry about rounding off values, such as \sqrt{n} , to be integers.

First, Renaming $m = \log n$

$$\Rightarrow n = 2^m$$

Put the value of n and m on the above recurrence.

Hence the above recurrence can be written as follows

$$T(2^m) = 2T(\lfloor \sqrt{2^m} \rfloor) + m$$

$$\Rightarrow T(2^m) = 2T(2^{m^{\frac{1}{2}}}) + m$$

$$\Rightarrow T(2^m) = 2T(2^{m/2}) + m$$

Master Method

We can now rename

$$S(m) = T(2^m)$$

$$\Rightarrow S(m/2) = T(2^{m/2})$$

Now put these values in above equation

$$S(m) = 2S(m/2) + m$$

Now apply master method for solve the above equation

Hear, $a = 2, b = 2$ and $f(m) = m$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } m^{\log_b a} = m^{\log_2 2} = m^1 = m$$

$$\text{and, } f(m) = m$$

Master Method

Hence as per the definition of master theorem Case 2

$$S(m) = \Theta(m^{\log_b a} \lg m)$$

$$\Rightarrow S(m) = \Theta(m \lg m)$$

$$\Rightarrow S(m) = \Theta(m \lg m)$$

$$\Rightarrow T(2^m) = \Theta(m \lg m) \quad \text{as } S(m) = T(2^m)$$

$$\Rightarrow T(n) = \Theta(\log n \lg \log n) \quad \text{as } n = 2^m, \text{ and } m = \log n$$

Hence the complexity of above recurrence is $\Theta(\log n \lg \log n)$

Master Method

Example 9

Solve the following recurrence by using Master Method

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + 1$$

Master Method

Example 9

Solve the following recurrence by using Master Method

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + 1$$

Due to, a little algebraic manipulation the above recurrence looks very difficult. These recurrences can be simplified by using change of variable. For convenience, we shall not worry about rounding off values, such as \sqrt{n} , to be integers.

First, Renaming $m = \log n$

$$\Rightarrow n = 2^m$$

$$\Rightarrow n^{1/2} = 2^{m/2}$$

Put the value of n on the above recurrence.

Hence the above recurrence can be written as follows

$$T(2^m) = 2T(\lfloor \sqrt{2^m} \rfloor) + 1$$

$$\Rightarrow T(2^m) = 2T(2^{m^{1/2}}) + 1$$

$$\Rightarrow T(2^m) = 2T(2^{m/2}) + 1$$

Master Method

We can now rename

$$S(m) = T(2^m)$$

$$\Rightarrow S(m/2) = T(2^{m/2})$$

Now put these values in above equation

$$S(m) = 2S(m/2) + 1$$

Now apply master method for solve the above equation

Hear, $a = 2, b = 2$ and $f(m) = 1$

First we calculate $n^{\log_b a}$ and then compare with $f(n)$

$$\text{So, } m^{\log_b a} = m^{\log_2 2} = m^1 = m \quad \text{and, } f(m) = 1$$

Hence as per the definition of master theorem Case 1:

$$m^{\log_b a} > f(m)$$

$$\Rightarrow m > 1$$

$$\Rightarrow m^{1-\varepsilon} = 1 \quad \text{where } \varepsilon = 1$$

Master Method

$$\text{Hence, } S(m) = \Theta(m^{\log_b a})$$

$$\Rightarrow S(m) = \Theta(m^{\log_b a}) = \Theta(m)$$

$$\Rightarrow S(m) = \Theta(m)$$

$$\Rightarrow T(2^m) = \Theta(\log n) \quad \text{as } S(m) = T(2^m) \text{ and } m = \log n$$

$$\Rightarrow T(n) = \Theta(\log n) \quad \text{as } n = 2^m$$

Hence the complexity of above recurrence is $\Theta(\log n)$

Master Method

Problems Solved by Students:

$$Q1. T(n) = T(n/2) + 2^n.$$

$$Q2. T(n) = 2^n T(n/2) + n^n.$$

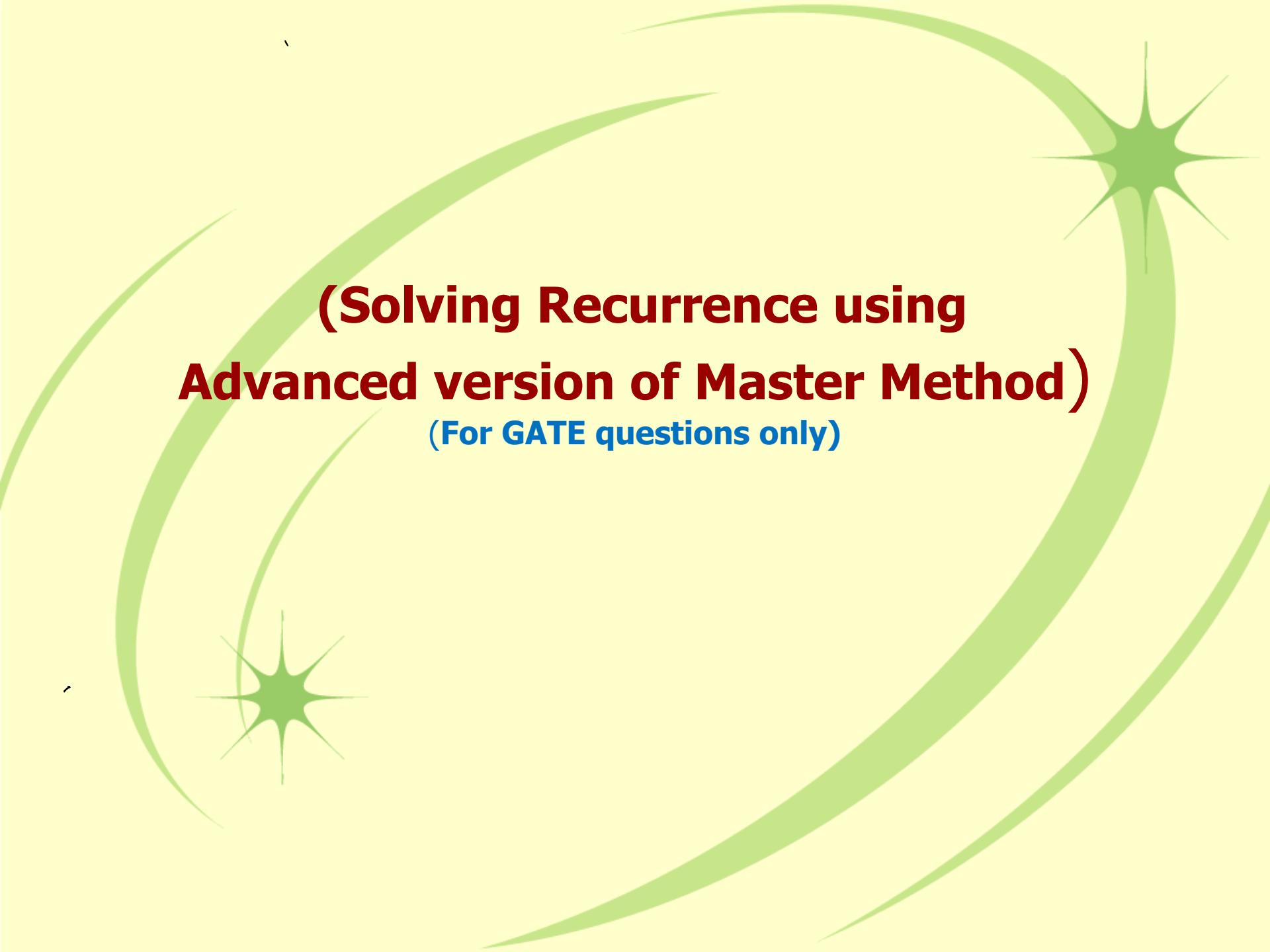
$$Q3. T(n) = 3T(n/2) + n^2.$$

$$Q4. T(n) = 16T(n/4) + n.$$

$$Q5. T(n) = 3T(n/2) + n^2 \log n.$$

$$Q6. T(n) = 2T(n/2) + \frac{n}{\log n}.$$

$$Q7. T(n) = 2T(\sqrt{n}) + \frac{\log_2 2^n}{\log \log_2 2^n}.$$



(Solving Recurrence using Advanced version of Master Method)

(For GATE questions only)

Master Method (GATE)

Definition (Advance Version)

Let $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real number and let $T(n)$ be defined on the nonnegative integers by the recurrence

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

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$.

Then $T(n)$ can be bounded asymptotically by comparing a with b^k as follows.

1. if $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$

2. If $a = b^k$ and

Option 1 : if $p < -1$, then $T(n) = \Theta(n^{\log_b a})$

Option 2 : if $p = -1$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^2 n)$

Option 3 : if $p > -1$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$

3. If $a < b^k$ and

Option 1 : if $p < 0$, then $T(n) = \Theta(n^k)$

Option 2 : if $p \geq 0$, then $T(n) = \Theta(n^k \cdot \log^p n)$

Master Method (GATE)

Example 10

Solve the following recurrence by using Master Method

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

Master Method (GATE)

Example 10

Solve the following recurrence by using Master Method

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

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

Now compare a with b^k .

So, $a = 3$ and $b^k = 2^2 = 4$

The comparision result shows that $a < b^k$

Hence as per the definition of Advanced version of Master Method case 3
(option 2)

$$T(n) = \Theta(n^k \cdot \log^p n) = \Theta(n^2 \cdot \log^0 n) = \Theta(n^2)$$

Master Method (GATE)

Example 11

Solve the following recurrence by using Master Method

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Master Method (GATE)

Example 11

Solve the following recurrence by using Master Method

$$T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2)$$

Hear, $a = 4, b = 2, k = 2, p = 0$

Now compare a with b^k .

So, $a = 4$ and $b^k = 2^2 = 4$

The comparision result shows that $a = b^k$

Hence as per the definition of Advanced version of Master Method case 2 (option 3)

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n) = \Theta(n^{\log_2 4} \cdot \log^{0+1} n) = \Theta(n^2 \cdot \log^1 n) = \Theta(n^2 \cdot \log n)$$

Master Method (GATE)

Example 12

Solve the following recurrence by using Master Method

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

Master Method (GATE)

Example 12

Solve the following recurrence by using Master Method

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

Hear, $a = 1, b = 2, k = 2, p = 0$

Now compare a with b^k .

So, $a = 1$ and $b^k = 2^2 = 4$

The comparision result shows that $a < b^k$

Hence as per the definition of Advanced version of Master Method case 3 (option 2)

$$T(n) = \Theta\left(n^k \cdot \log^p n\right) = \Theta\left(n^2 \cdot \log^0 n\right) = \Theta(n^2)$$

Master Method (GATE)

Example 13

Solve the following recurrence by using Master Method

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

Master Method (GATE)

Example 13

Solve the following recurrence by using Master Method

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

Hear, $a = 2^n, b = 2, k = n, p = 0$

The value of a must be a constant number. Which is not true in this case.

Hence Master method can't be applied here.

Master Method (GATE)

Example 14

Solve the following recurrence by using Master Method

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

Master Method (GATE)

Example 14

Solve the following recurrence by using Master Method

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

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

Now compare a with b^k .

So, $a = 2$ and $b^k = 2^1 = 2$

The comparision result shows that $a = b^k$

Hence as per the definition of Advanced version of Master Method case 2 (option 3)

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n) = \Theta(n^{\log_2 2} \cdot \log^{1+1} n) = \Theta(n \cdot \log^2 n)$$

Master Method (GATE)

Example 15

Solve the following recurrence by using Master Method

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

Master Method (GATE)

Example 15

Solve the following recurrence by using Master Method

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

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

Now compare a with b^k .

So, $a = 2$ and $b^k = 2^1 = 2$

The comparision result shows that $a = b^k$

Hence as per the definition of Advanced version of Master Method case 2
(option 2)

$$T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n) = \Theta(n^{\log_2 2} \cdot \log^2 n) = \Theta(n \cdot \log^2 n)$$

Master Method (GATE)

Example 16

Solve the following recurrence by using Master Method

$$T(n) = 2T\left(\frac{n}{4}\right) + \Theta(n^{0.51})$$

Master Method (GATE)

Example 16

Solve the following recurrence by using Master Method

$$T(n) = 2T\left(\frac{n}{4}\right) + \Theta(n^{0.51})$$

Hear, $a = 2, b = 4, k = 0.51, p = 0$

Now compare a with b^k .

So, $a = 2$ and $b^k = 4^{0.51}$

The comparision result shows that $a < b^k$

Hence as per the definition of Advanced version of Master Method case 3
(option 2)

$$T(n) = \Theta(n^k \cdot \log^p n) = \Theta(n^{0.51} \cdot \log^0 n) = \Theta(n^{0.51})$$

Master Method (GATE)

Example 17

Solve the following recurrence by using Master Method

$$T(n) = 0.5T\left(\frac{n}{2}\right) + \frac{1}{n}$$

Master Method (GATE)

Example 17

Solve the following recurrence by using Master Method

$$T(n) = 0.5T\left(\frac{n}{2}\right) + \frac{1}{n}$$

Hear, $a = 0.5, b = 2, k = -1, p = 0$

As per the definition of master method the value of 'a' should be greater than 1.

But hear the value of a is less than 1. Hence Master method can't be applied here.

Master Method (GATE)

Example 18

Solve the following recurrence by using Master Method

$$T(n) = 6T\left(\frac{n}{3}\right) + \Theta(n^2 \log n)$$

Master Method (GATE)

Example 18

Solve the following recurrence by using Master Method

$$T(n) = 6T\left(\frac{n}{3}\right) + \Theta(n^2 \log n)$$

Hear, $a = 6, b = 3, k = 2, p = 1$

Now compare a with b^k .

So, $a = 6$ and $b^k = 3^2 = 9$

The comparision result shows that $a < b^k$

Hence as per the definition of Advanced version of Master Method case 3
(option 2)

$$T(n) = \Theta(n^k \cdot \log^p n) = \Theta(n^2 \cdot \log^1 n) = \Theta(n^2 \cdot \log n)$$

Master Method (GATE)

Example 19

Solve the following recurrence by using Master Method

$$T(n) = 64T\left(\frac{n}{8}\right) - \Theta(n^2 \lg n)$$

Master Method (GATE)

Example 19

Solve the following recurrence by using Master Method

$$T(n) = 64T\left(\frac{n}{8}\right) - \Theta(n^2 \lg n)$$

This recurrence says that without any execution the problem is divided in to sub problems. Because the term ($-n^2 \lg n$) is not valid. Hence it is an invalid representation.

Master Method (GATE)

Example 20

Solve the following recurrence by using Master Method

$$T(n) = 4T\left(\frac{n}{3}\right) + \Theta(\log n)$$

Master Method (GATE)

Example 20

Solve the following recurrence by using Master Method

$$T(n) = 4T\left(\frac{n}{3}\right) + \Theta(\log n)$$

Hear, $a = 4, b = 2, k = 0, p = 1$

Now compare a with b^k .

So, $a = 4$ and $b^k = 2^0 = 1$

The comparision result shows that $a > b^k$

Hence as per the definition of Advanced version of Master Method case 1

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 4})$$

Master Method (GATE)

Example 21

Solve the following recurrence by using Master Method

$$T(n) = 27T\left(\frac{n}{3}\right) + \Theta(n^3 \lg n)$$

Master Method (GATE)

Example 21

Solve the following recurrence by using Master Method

$$T(n) = 27T\left(\frac{n}{3}\right) + \Theta(n^3 \lg n)$$

Hear, $a = 27, b = 3, k = 3, p = 1$

Now compare a with b^k .

So, $a = 27$ and $b^k = 3^3 = 1$

The comparision result shows that $a = b^k$

Hence as per the definition of Advanced version of Master Method
case 2(option 3)

$$\begin{aligned} T(n) &= \Theta(n^{\log_b a} \cdot \log^{p+1} n) = \Theta(n^{\log_3 27} \cdot \log^{1+1} n) = \Theta(n^3 \cdot \log^2 n) \\ &= \Theta(n^3 \cdot \log \log n) \end{aligned}$$

Thank U