Nadar Saraswathi College of Engineering & Technology

NSCET E-LEARNING PRESENTATION

LISTEN … LEARN… LEAD…

COMPUTER SCIENCE AND ENGINEERING

II YEAR / IV SEMESTER

CS8451 – DESIGN AND ANALYSIS OF ALGORITHMS

S ARUL JOTHI M.E.,MISTE,

ASSISTANT PROFESSOR

Nadar Saraswathi College of Engineering & Technology,

Vadapudupatti, Annanji (PO), Theni – 625531.

UNIT 2-Brute Force and Divide-and-Conquer

Brute Force

- Computing a^n
- String Matching
- Closest Pair and Convex Hull Problems
- Exhaustive search
- Traveling salesman Problem
- Knapsack Problem
- Assignment Problem

UNIT 2-Brute Force and Divide-and-Conquer

Divide-and-Conquer

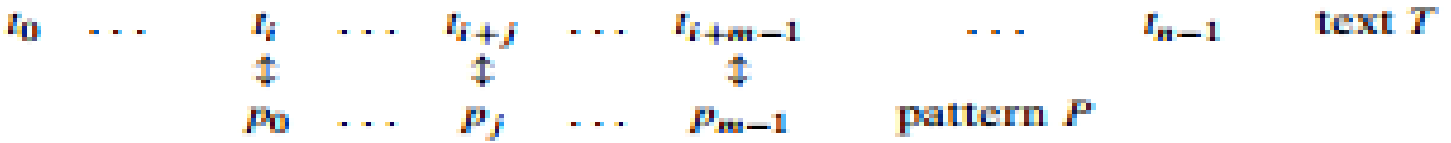
- Binary Search
- Merge sort
- Quick sort
- Heap sort
- Multiplication of large Integers
- Closest Pair and Convex Hull Problems

Brute Force-Computing a^n

- Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved
- consider the exponentiation problem: compute an for a nonzero number a and a nonnegative integer n.
- a^n= a * … * a multiplying n times
- Ex. Computing a^5 here a=2 and n=5 means multiply by 2 for 5 times , The result is 2*2*2*2*2=32

Brute Force-String Matching

- Given a string of n characters called the text and a string of m characters (m $\leq$ n) called the pattern, find a substring of the text that matches the pattern.
- to find i—the index of the leftmost character of the first matching substring in the text such that $t_i = p_0,\ldots,t_{i+j} = p_j,\ldots,t_{i+m-1} = p_{m-1}$:



Brute Force-String Matching

```
ALGORITHM BruteForceStringMatch(T [0..n – 1], P[0..m – 1])
//Implements brute-force string matching
//Input: An array T [0..n – 1] of n characters representing a text and
// an array P[0..m – 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
// matching substring or –1 if the search is unsuccessful
for i ⟵ 0 to n – m do
    j ⟵ 0
        while j<m and P[j ] = T [i + j ] do
            j ⟵ j + 1
        if j = m return i
return –1
```

Brute Force-String Matching

Complexity:

- The worst case is much worse: the algorithm may have to make all m comparisons before shifting the pattern, and this can happen for each of the n – m + 1 tries. m(n – m + 1) character comparisons, which puts it in the O(nm) class.
- for searching in random texts, it has been shown to be linear,

  i.e.Θ(n).

Brute Force-Closest Pair and Convex Hull Problems

- Closest-Pair Problem-finding the two closest points in a set of n points.
- One of the important applications of the closest-pair problem is cluster analysis in statistics.
- This metric is usually the Euclidean distance; for text and other nonnumerical data, metrics such as the Hamming distance

Brute Force-Closest Pair

ALGORITHM BruteForceClosestPair(P)

//Finds distance between two closest points in the plane by brute force

//Input: A list P of n ($n \geq 2$) points p1(x1, y1), . . . , $p_n(x_n, y_n)$

//Output: The distance between the closest pair of points

$d \leftarrow \infty$

for i $\leftarrow$ 1 to n – 1 do

    for j $\leftarrow$ i + 1 to n do

        d $\leftarrow$ min(d, sqrt((xi – xj)2 + (yi – yj)2)) //sqrt is square root

return d

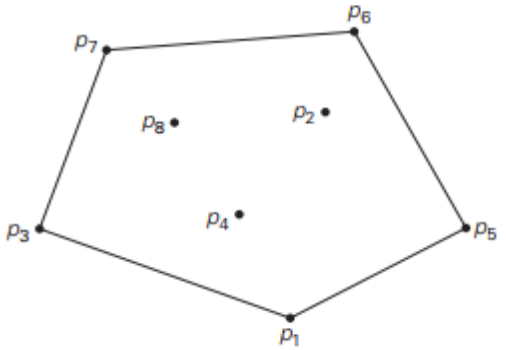
Brute Force-Closest Pair

$$C(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2 = 2 \sum_{i=1}^{n-1} (n-i)$$

$$= 2[(n-1) + (n-2) + \cdots + 1] = (n-1)n \in \Theta(n^2).$$

Brute Force Convex-Hull Problem

- The convex-hull problem is the problem of constructing the convex hull for a given set S of n points.
- an extreme point of a convex set is a point of this set that is not a middle point of any line segment with :endpoints in the set.



Complexity

The time efficiency is $O(n^3)$: for each of $n(n-1)/2$ pairs of distinct points

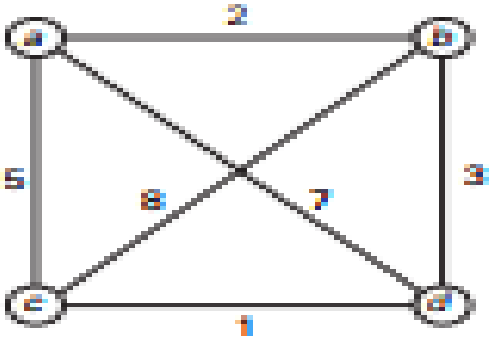
Exhaustive search

- Exhaustive search is simply a brute-force approach to combinatorial problems.
- It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element
- Exhaustive search can be applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

Exhaustive search- Traveling salesman problem

- The problem can be modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.
- Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph.
- A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once.
- The total number of permutations needed is 1/2(n – 1)!,which makes the exhaustive-search approach impractical for all but very small values of n.

Exhaustive search- Traveling salesman problem



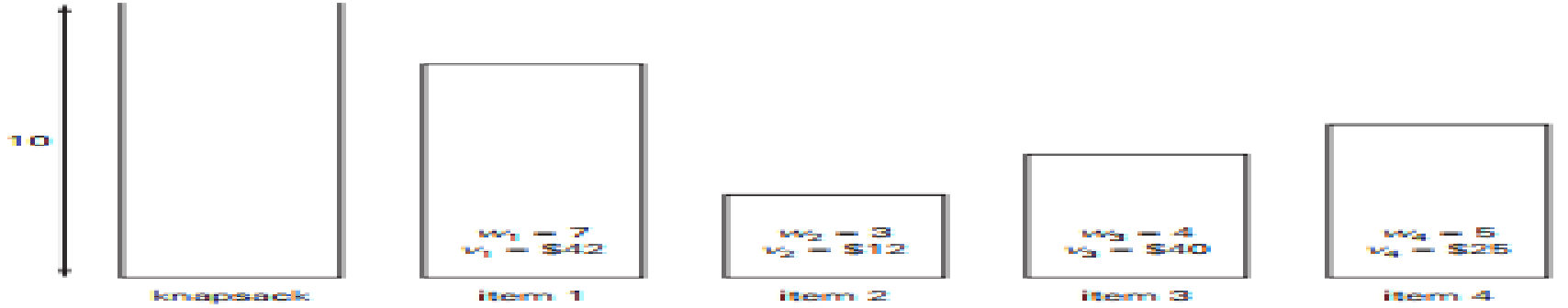
Tour	Length	
$a \to b \to c \to d \to a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \to b \to d \to c \to a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \to c \to b \to d \to a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \to c \to d \to b \to a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \to d \to b \to c \to a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \to d \to c \to b \to a$	$l = 7 + 1 + 8 + 2 = 18$	

Exhaustive search- Knapsack Problem

- Given n items of known weights w1, w2,...,w_n and values v1, v2,...,v_n and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack
- The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets
- the number of subsets of an n-element set is 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm
- This search leads to algorithms that are extremely inefficient on

  every input.

Exhaustive search- Knapsack



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	9	$65
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

Exhaustive search- Assignment Problem

- Each person is assigned to exactly one job and each job is assigned to exactly one person
- The cost that would accrue if the i^{th} person is assigned to the j^{th} job is a known quantity C[i,j] for each pair i, j = 1, 2,...,n. The problem is to find an as

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

Exhaustive search- Assignment Problem

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

Since the number of permutations to be considered for the general case of the assignment problem is n!, exhaustive search is impractical for all but very small instances of the problem

Exhaustive search

- The term "exhaustive search" can also be applied to two very important algorithms that systematically process all vertices and edges of a graph.
- Depth-first search (DFS) and breadth-first search (BFS) are two principal graph-traversal algorithms
- Both algorithms have the same time efficiency:

$(|V|^2)$ for the adjacency matrix representation and $(|V|+|E|)$ for the adjacency list representation.

Divide-and-conquer

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several sub problems of the same type, ideally of about equal size.
2. The sub problems are solved (typically recursively)
3. If necessary, the solutions to the sub problems are combined to get a solution to the original problem.

Divide-and-conquer

Master Theorem If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in recurrence (5.1), then

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

$n=2^k$

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

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

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

Divide-and-conquer- Mergesort

- Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n – 1] by dividing it into two halves
- A[0..n/2 – 1] and A[n/2..n – 1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

Divide-and-conquer- Mergesort

ALGORITHM Mergesort(A[0..n – 1])

//Sorts array A[0..n – 1] by recursive mergesort

//Input: An array A[0..n – 1] of orderable elements

//Output: Array A[0..n – 1] sorted in non decreasing order

if n > 1

    copy A[0..n/2 – 1] to B[0..n/2 – 1]

    copy A[n/2..n – 1] to C[0..n/2 – 1]

    Mergesort(B[0..n/2 – 1])

    Mergesort(C[0..n/2– 1])

Divide-and-conquer- Mergesort

ALGORITHM Merge(B[0..p − 1], C[0..q − 1], A[0..p + q − 1])

//Merges two sorted arrays into one sorted array

//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted

//Output: Sorted array A[0..p + q − 1] of the elements of B and C

i ⟵ 0; j ⟵ 0; k ⟵ 0

while i<p and j<q do

    if B[i] ≤ C[j]

        A[k]⟵ B[i]; i ⟵ i + 1

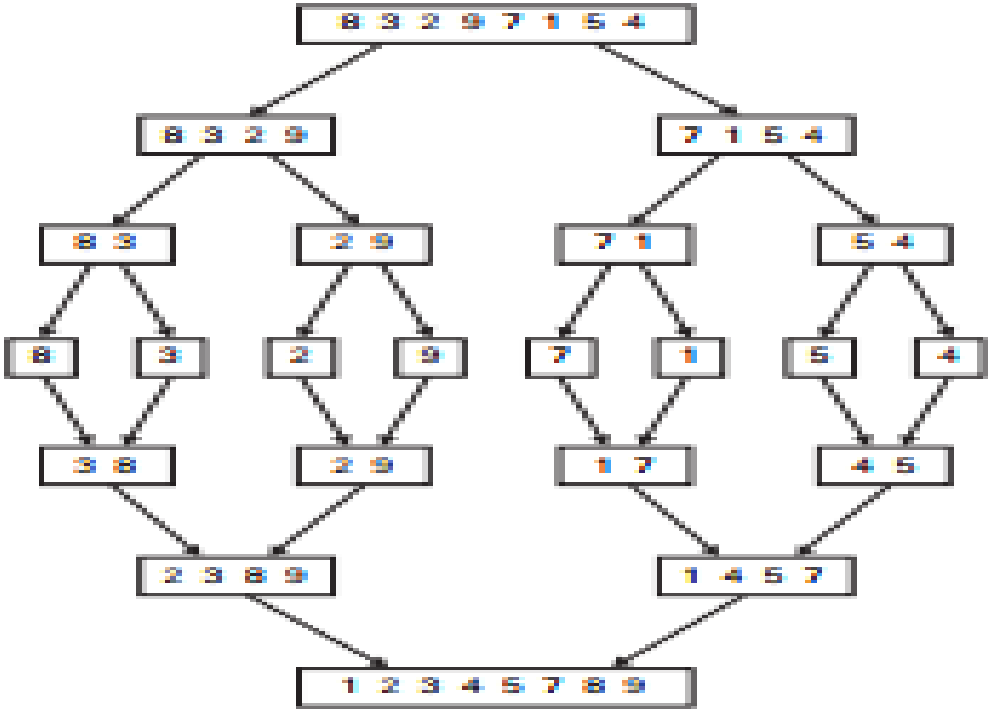
    else A[k]⟵ C[j]; j ⟵ j + 1

    k ⟵ k + 1

If i = p

Divide-and-conquer- Mergesort

- The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure



Divide-and-conquer- Mergesort

- The recurrence relation for the number of key comparisons C(n) is

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

- for the worst case, $C_{merge}(n) = n - 1$, and the recurrence

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

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

and also for average is (n log n)

Divide-and-conquer- Quicksort

- Quicksort divides the input according to their value.
- A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s] are greater than or equal to it:
- A[s] will be in its final position in the sorted array, and continue sorting the two subarrays to the left and to the right of A[s] independently (e.g., by the same method)

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

Divide-and-conquer- Quicksort

ALGORITHM Quicksort(A[l..r])

//Sorts a subarray by quicksort

//Input: Subarray of array A[0..n – 1], defined by its left and right indices l and r

//Output: Subarray A[l..r] sorted in nondecreasing order if l<r

s ⟵Partition(A[l..r]) //s is a split position

Quicksort(A[l..s – 1])

Quicksort(A[s + 1..r])

Divide-and-conquer- Quicksort

ALGORITHM HoarePartition(A[l..r])

//Partitions a subarray by Hoare's algorithm, using the first element as a pivot

//Input: Subarray of array A[0..n – 1], defined by its left and right indices l and r (l < r)

//Output: Partition of A[l..r], with the split position returned as this function's value

p ⟵ A[l]

i ⟵ l; j ⟵ r + 1

repeat

repeat i ⟵ i + 1 until A[i] ≥ p

repeat j ⟵ j – 1 until A[j] ≤ p

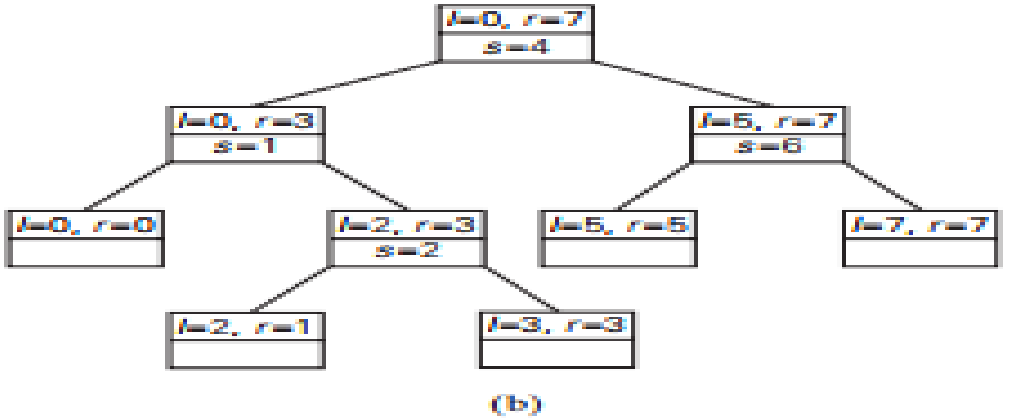
swap(A[i], A[j])

until i ≥ j

swap(A[i], A[j]) //undo last swap when i ≥ j

Divide-and-conquer- Quicksort

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



(b)

Divide-and-conquer- Quicksort

- The number of key comparisons in the best case satisfies the recurrence

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

- According to the Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields , $C_{best}(n) = n \log 2 n$.
- $C_{worst}(n) = (n + 1) + n + \ldots + 3 = \in \Theta(n^2)$.
- $Cavg(n) \approx 2n \ln n \approx 1.39n \log_2 n$

Multiplication of Large Integers

Consider two-digit integers, say, 23 and 14.

$23 = 2 \cdot 10^1 + 3 \cdot 10^0$ and $14 = 1 \cdot 10^1 + 4 \cdot 10^0$

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

$= (2 * 1)\ 10^2 + (2 * 4 + 3 * 1)\ 10^1 + (3 * 4)\ 10^{10} = 322$

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product can be computed by the formula $c = a * b = c_2 10^2 + c_1\ 10^1 + c_0$ where

$c_2 = a1 * b1$ is the product of their first digits,

$c_0 = a0 * b0$ is the product of their second digits,

$c_1 = (a1 + a0) * (b1 + b0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0

Multiplication of Large Integers

The recurrence for the number of multiplications M(n) is

$M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$.

$M(2^k) = 3M(2^{k-1}) = 3[3M(2^{k-2})] = 3^2M(2^{k-2}) = \ldots = 3^iM(2^{k-i})$

$= \ldots = 3^kM(2^{k-k}) = 3^k$

Since $k = \log_2 n$,

$M(n) = 3^{\log_2 n}$

$= n^{\log_2 3} \approx n^{1.585}$

Strassen's Matrix Multiplication

Divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers,Such an algorithm was published by V. Strassen in 1969

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

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

where,

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

Strassen's Matrix Multiplication-Time Complexity

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

Since $n = 2^k$,

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

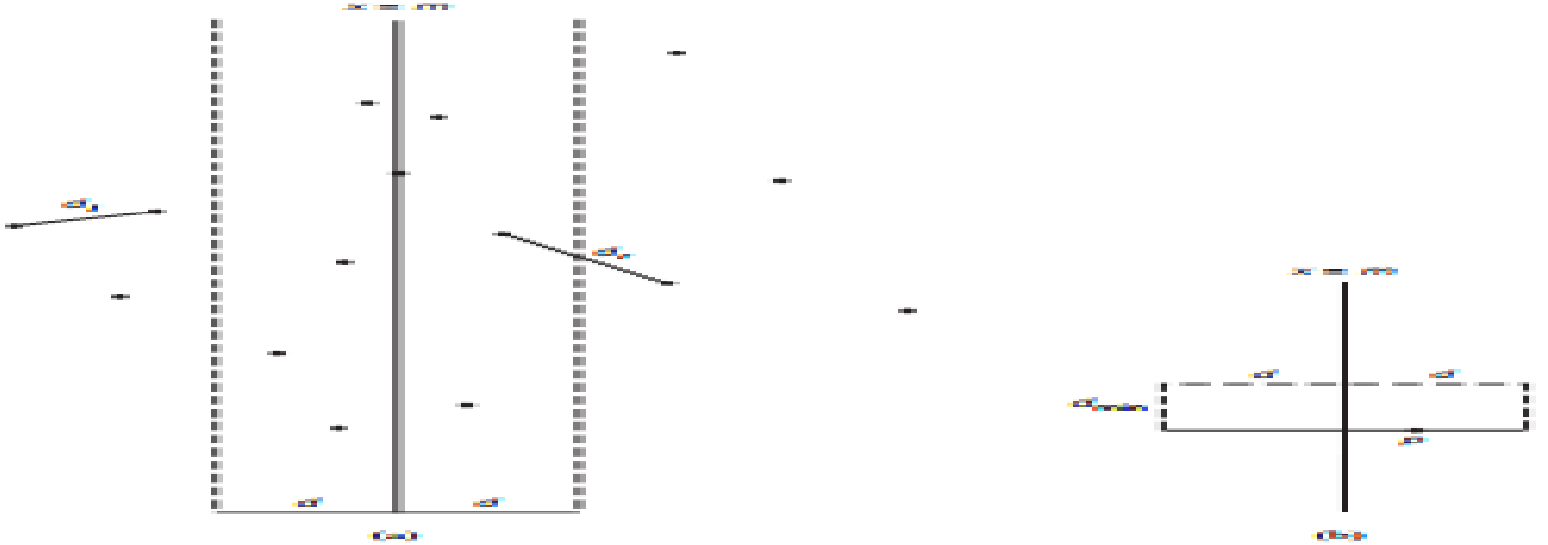
Since $k = \log_2 n$,

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

Divide-and-conquer-The Closest-Pair Problem

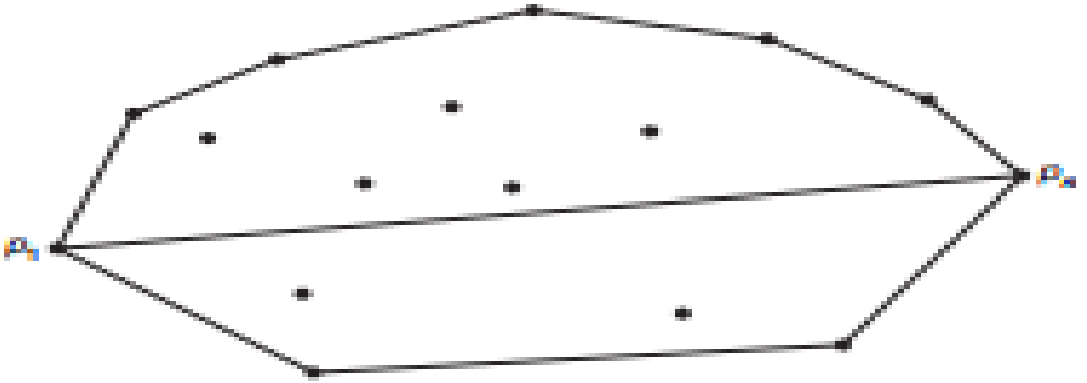
- Let P be a set of n > 1 points in the Cartesian plane.
- Assume that the points are distinct.
- Also assume that the points are ordered in nondecreasing order of their x coordinate.
- solve the closest-pair problem recursively for subsets P_l and P_r. Let d_l and d_r be the smallest distances between pairs of points in P_l and P_r, respectively, and let $d = \min\{d_l, d_r\}$.

Divide-and-conquer-The Closest-Pair Problem



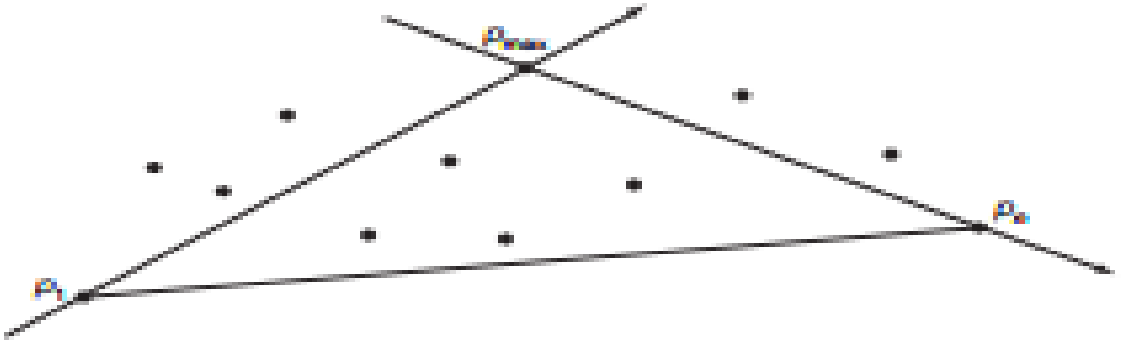
$T(n) = 2T(n/2) + f(n)$, where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in$

Divide-and-conquer-Convex-Hull Problem

- Let S be a set of n > 1 points $p_1(x_1, y_1) \ldots, p_n(x_n, y_n)$ in the Cartesian plane.
- Assume that the points are sorted in nondecreasing order of their x coordinates, with ties resolved by increasing order of the y coordinates of the points involved.
-  nts.

Divide-and-conquer-Convex-Hull Problem

The idea of quickhull as follows:



Quickhull has the same (n2) worst-case efficiency as quicksort

Binary search

- If the array is sorted first, we can then apply binary search, which requires only $\log_2 n + 1$ comparisons in the worst case. Assuming the most efficient n log n sort, the total running time of such a searching algorithm in the worst case will be
- $T(n) = T_{sort}(n) + T_{search}(n) = (n \log n) + (\log n) = (n \log n)$,

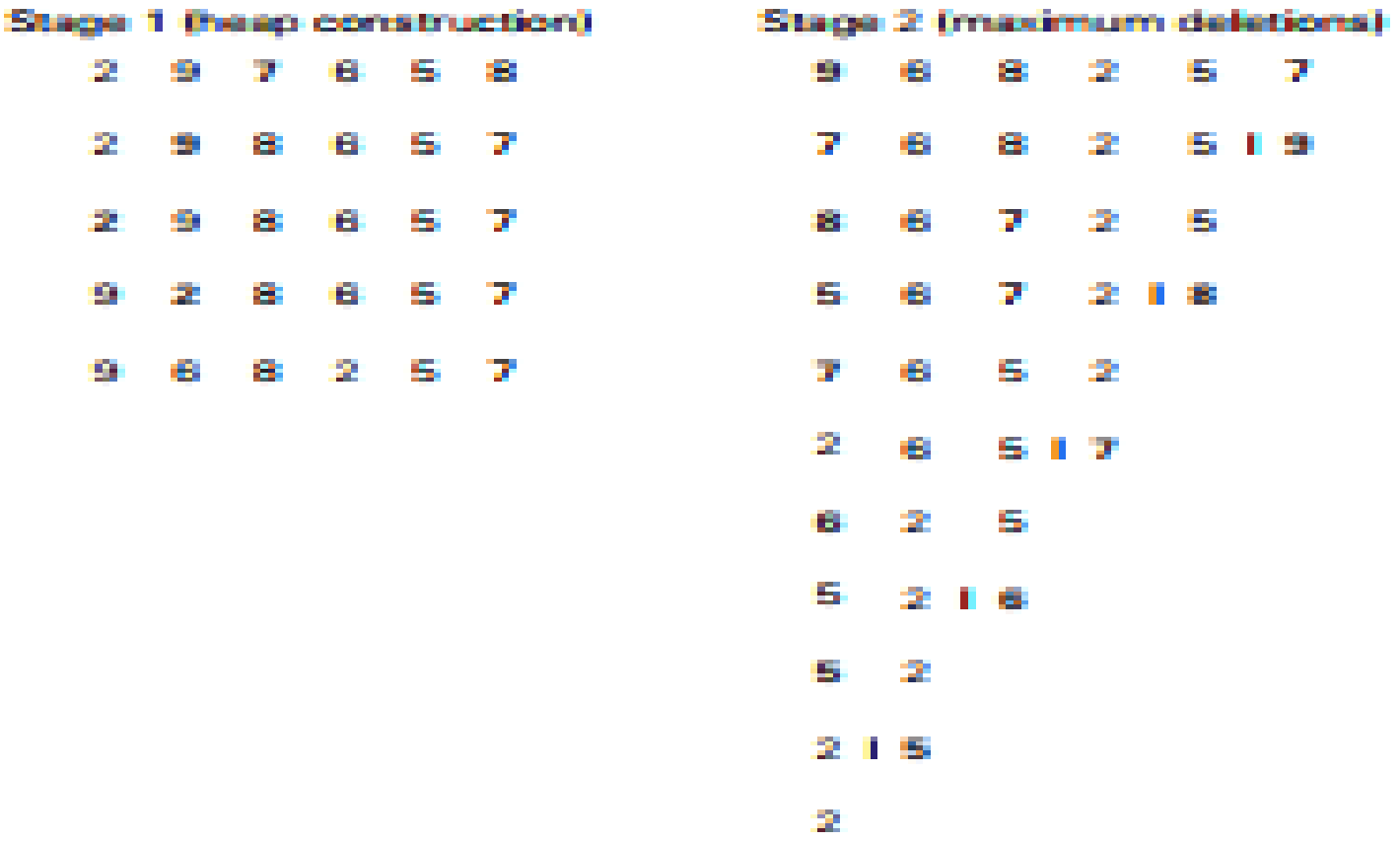
  which is inferior to sequential search. The same will also be true for the average case efficiency.

Heapsort

- an interesting sorting algorithm discovered by J. W. J. Williams This is a two-stage algorithm that works as follows.
- Stage 1 (heap construction): Construct a heap for a given array.
- Stage 2 (maximum deletions): Apply the root-deletion operation

  n – 1 times to the remaining heap.

Heapsort-Example



Heapsort-Complexity

- For the number of key comparisons, C(n), needed for eliminating the root keys from the heaps of diminishing sizes from n to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2 (n-1) \rfloor + 2\lfloor \log_2 (n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2 \sum_{i=1}^{n-1} \log_2 i$$

$$\le 2 \sum_{i=1}^{n-1} \log_2 (n-1) = 2(n-1) \log_2 (n-1) \le 2n \log_2 n.$$

- C(n) ∈ O(n log n) in second stage
- For both stage O(n) + O(n log n) = O(n log n)

THANK YOU