Advanced Algorithms Analysis and Design

By

Nazir Ahmad Zafar

Lecture No 12

Design of Algorithms using Brute Force Approach

Today Covered

Brute Force Approach,

- Checking primality
- Sorting sequence of numbers
- Knapsack problem
- Closest pair in 2-D, 3-D and n-D
- Finding maximal points in n-D

Primality Testing

(given number is n binary digits)

First Algorithm for Testing Primality

Brute Force Approach

```
Prime (n)
    for i ← 2 to n-1
        if n ≡ 0 mod i then
            "number is composite"
    else
        "number is prime"
```

- The computational cost is $\Theta(n)$
- The computational cost in terms of binary operation is $\Theta(2^n)$

Refined Algorithm for Testing Primality

```
Prime (n)
    for i $\leftarrow$ 2 to n/2
        if n $\equiv$ 0 mod i then
            "number is composite"
    else
        "number is prime"
```

- The computational cost is $\Theta(n/2)$
- The computational cost in terms of binary operation is $\Theta(2^{n-1})$, not much improvement

Algorithm for Testing Primality

- We are not interested, how many operations are required to test if the number n is prime
- In fact, we are interested, how many operations are required to test if a number with n digits is prime.
- RSA-128 encryption uses prime numbers which are 128 bits long. Where 2^{128} is:

  340282366920938463463374607431768211456
- Therefore, to prime test a number with n binary digits by brute force, we need to check 2^n numbers.
- Thus brute-force prime testing requires *exponential time* with the n number of digits.
- Which is not accepted in this case

Lemma

Statement

- If $n \in N$, $n > 1$ is not prime then n is divisible by some prime number $p \leq$ square root of n.

Proof

- Since n is not prime hence it can be factored as

  $n = x.y$ where $1 < x \leq y < n$

- x or y is a prime, if not it can be further factored out.
- Also suppose without loss of generality that $x \leq y$
- Now our claim is that $x \leq sq(n)$
- This is because otherwise $x.y > n$, a contradiction
- We only require to check till sqr(n) for primality test.

Refined Algorithm for Testing Primality

```
Prime (n)
    for i ← 2 to sqr(n)
        if n ≡ 0 mod i then
            "number is composite"
    else
        "number is prime"
```

- The computational cost is Θ(sqr(n)), much faster
- The computational cost in terms of binary operation is $\Theta(2^{\text{squareroot(n)}})$, still exponential
- Computation cost can be decreased using number theoretic concepts, which will be discussed later on.

Sorting Sequence of Numbers

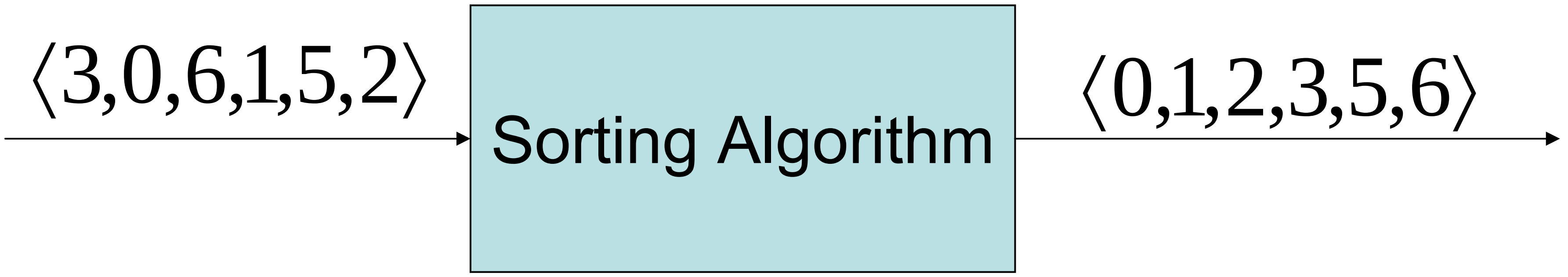
An Example of Algorithm

- Input : A sequence of n numbers (distinct)

$$\langle a_1, a_2, \ldots, a_n \rangle$$

- Output : a permutation, $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that

$$a'_1 \le a'_2 \le \ldots \le a'_n$$



Sorting Algorithm: Brute Force Approach

Sort the array [2, 4, 1, 2] in increasing order

s1 = [4,3,2,1], s2 = [4,3,1,2], s3 = [4,1,2,3]

s4 = [4,2,3,1], s5 = [4,1,3,2], s6 = [4,2,1,3]

s7 = [3,4,2,1], s8 = [3,4,1,2], s9 = [3,1,2,4]

s10 = [3,2,4,1], s11 = [3,1,4,2], s12 = [3,2,1,4]

s13 = [2,3,4,1], s14 = [2,3,1,4], s15 = [2,1,4,3]

s16 = [2,4,3,1], s17 = [2,1,3,4], s18 = [2,4,1,3]

s19 = [1,3,2,4], s20 = [1,3,1,4], s21 = [1,4,2,3]

s22 = [1,2,3,4], s23 = [1,4,3,2], s24 = [1,2,4,3]

There are 4! = 24 number of permutations.

For n number of elements there will be n! number of permutations. Hence cost of order n! for sorting.

Generating Permutations

```
Permute (i)                    \\initial call Permute(1)
    if i == N
        output A[N]
    else
        for j = i to N do
            swap(A[i], A[j])
            permute(i+1)
            swap(A[i], A[j])
```

- There are 4! = 24 number of permutations.
- For n number of elements there will be n! number of permutations. Hence cost of order n! for sorting.

Generating Permutations

Theorem

Prove, by mathematical induction, that computational cost is n!

0-1 Knapsack Problem

0-1 Knapsack Problem Statement

The knapsack problem arises whenever there is resource allocation with no financial constraints

Problem Statement

- You are in Japan on an official visit and want to make shopping from a store (Best Denki)
- You have a list of required items
- You have also a bag (knapsack), of fixed capacity, and only you can fill this bag with the selected items
- Every item has a value (cost) and weight,
- And your objective is to seek most valuable set of items which you can buy not exceeding bag limit.

0-1 Knapsack Example

Input

- Given *n* items each
  - weight w_i
  - value v_i
- Knapsack of capacity *W*

Output: Find most valuable items that fit into the knapsack

Example:

knapsack capacity W = 16

item	*weight*	*value*
1	2	20
2	5	30
3	10	50
4	5	10

0-1 Knapsack Problem Example

	Subset	*Total weight*	*Total value*
1.	∅	*0*	*0*
2.	{1}	2	20
3.	{2}	5	30
4.	{3}	10	50
5.	{4}	5	10
6.	{1,2}	7	50
7.	{1,3}	12	70
8.	{1,4}	7	30
9.	{2,3}	15	80
10.	{2,4}	10	40
11.	{3,4}	15	60
12.	{1,2,3}	17	not feasible
13.	{1,2,4}	12	60
14.	{1,3,4}	17	not feasible
15.	{2,3,4}	20	not feasible
16.	{1,2,3,4}	22	not feasible

#	*W*	*V*
1	2	20
2	5	30
3	10	50
4	5	10

0-1 Knapsack Algorithm

```
Knapsack-BF (n, V, W, C)
    Compute all subsets, s, of S = {1, 2, 3, 4}
    forall s ∈ P(S),  s ⊆ S
        weight = Compute sum of weights of these items
        if weight > C
              then not feasible
        else new solution = sum of values of these items
    solution = solution ∪ {new solution}
    Return maximum of solution
```

0-1 Knapsack Algorithm Analysis

Approach

- In brute force algorithm, we go through all combinations and find the one with maximum value and with total weight less or equal to $W = 16$

Complexity

- Cost of computing subsets $O(2^n)$ for n elements
- Cost of computing weight = $O(2^n)$
- Cost of computing values = $O(2^n)$
- Total cost in worst case: $O(2^n)$

The Closest Pair Problem

Finding Closest Pair

Problem

The closest pair problem is defined as follows:

- Given a set of n points, determine the two points that are closest to each other in terms of distance. Furthermore, if there are more than one pair of points with the closest distance, all such pairs should be identified.

Input :

is a set of n points

Output

- is a pair of points closest to each other,
- there can be more then one such pairs

Finding Closest Pair

Closest Pair Problem in 2-D

- A point in 2-D is an ordered pair of values (x, y).
- The Euclidean distance between two points

  $P_i = (x_i, y_i)$ and $P_j = (x_j, y_j)$ is

  $d(p_i, p_j) = sqr((x_i - x_j)^2 + (y_i - y_j)^2)$
- The closest-pair problem is finding the two closest points in a set of n points.
- The brute force algorithm checks every pair of points.
- Assumption: We can avoid computing square roots by using squared distance.
  - This assumption will not loose correctness of the problem.

Brute Force Approach: Finding Closest Pair in 2-D

ClosestPairBF(P)

```
1.  mind ← ∞
2.  for i ← 1 to n
3.  do
    4. for j ← 1 to n
    5. if i ≠ j
    6. do
    7. d ← $((x_i - x_j)^2 + (y_i - y_j)^2)$
    8. if d < minn then
        8. mind ← d
        9. mini ← i
        10.minj ← j
11.return mind, p(mini, minj)
```

Time Complexity

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} c$$

$$= \sum_{i=1}^{n} cn$$

$$= cn^2$$

$$= \Theta(n^2)$$

Improved Version: Finding Closest Pair in 2-D

ClosestPairBF(P)

```
1. mind ← ∞
2. for i ← 1 to n − 1
3. do
      4. for j ← i + 1 to n
      5. do
      6. d ← $((x_i - x_j)^2 + (y_i - y_j)^2)$
      7. if d < minn then
            8. mind ← d
            9. mini ← i
            10.minj ← j
11.return mind, p(mini, minj)
```

Time Complexity

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c\left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right)$$

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

$$= \Theta(n^2)$$

Brute Force Approach: Finding Closest Pair in 3-D

ClosestPairBF(P)

```
1.  mind ← ∞
2.  for i ← 1 to n − 1
3.  do
        4. for j ← i + 1 to n
        5. do
        6. d ← $((x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2)$
        7. if d < minn then
            8. mind ← d
            9. mini ← i
            10.minj ← j
11.return mind, p(mini), p(minj)
```

Time Complexity

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c$$

$$= \sum_{i=1}^{n-1} c(n-i)$$

$$= c\left(\sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i\right)$$

$$= \Theta(n^2)$$

Finding Maximal in n-dimension

Maximal Points

- Dominated Point in 2-D

  A point p is said to be dominated by q if

  $p.x \leq q.x$ and $p.y \leq q.y$

- Dominated Point in n-D

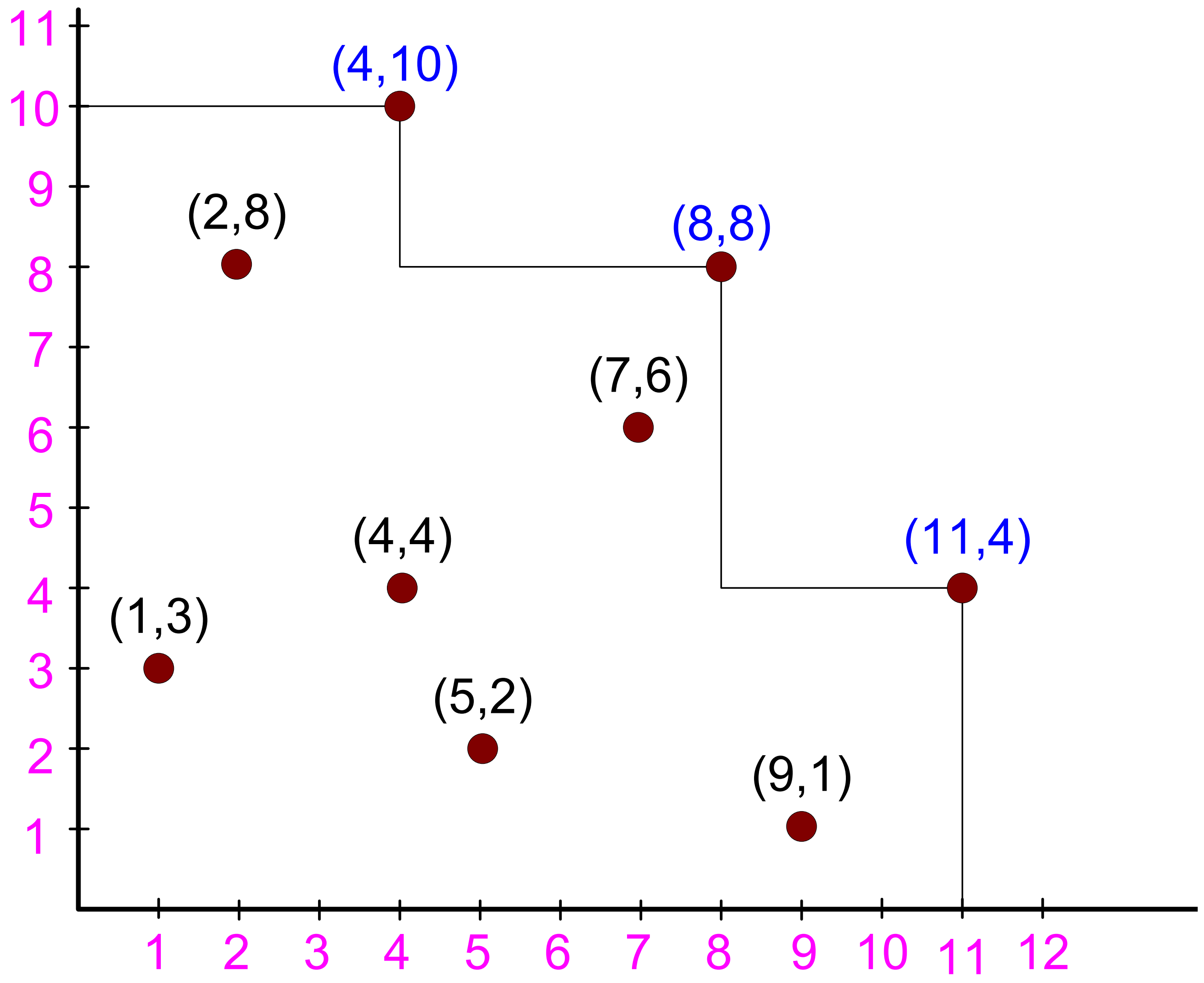
  A point p is said to be dominated by q if

  $p.x_i \leq q.x_i \ \forall \ i = 1, \ldots, n$

- Maximal Point

  A point is said to be maximal if it is not dominated by any other point.

Example: Maximal Points in 2-Dimension



Example: Buying a Car

Suppose we want to buy a car which is

- Fastest and
- Cheapest

- Fast cars are expensive. We want cheapest.
- We can't decide which one is more important
  - Speed or
  - Price.
- Of course fast and cheap dominates slow and expensive car.
- So, given a collection of cars, we want the car which is not dominated by any other.

Example: Buying a Car

Formal Problem: Problem can be modeled as:

- For each car C, we define C (x, y) where

  x = speed of car and

  y = price of car
- This problem can not be solved using maximal point algorithm.

Redefine Problem:

- For each car C’, we define C’ (x’, y’) where

  x’ = speed of car and

  y’ = negation of car price
- This problem is reduced to designing maximal point algorithm

Problem Statement

Problem Statement:

Given a set of m points, $P = \{p_1, p_2, . . . , p_m\}$, in n-dimension. Our objective is to compute a set of maximal points i.e. set of points which are not dominated by any one in the given list.

Mathematical Description:

Maximal Points =

$\{ p \in P \mid \exists\ i \in \{1,. . . , n\}\ \&\ p.x_i \geq q.x_j,$
$\forall, q \in \{p_1, p_2, . . . , p_m\}$

Brute Force Algorithm in n-dimension

```
MAXIMAL-POINTS (int m, Point P[1. . . m])
0  A = ∅;
1  for i ← 1 to m          \\ m used for number of points
2  do maximal ← true
3        for j ← 1 to m
4        do
5              if (i ≠ j) &
6              for k ← 1 to n          \\ n stands for dimension
7              do
8                    P[i].x[k] ≤ P[j].x[k]
9                          then maximal ← false; break
10       if maximal
11             then A = A ∪ P[i]
```

Plane Sweep Algorithm in n-dimension

```
MAXIMAL-PINTS (int m, int n, Point P[1. . . m])
sort P in increasing order by first component
stack s;
for i ←1 to m      \\ m used for number of points
do
      while (s.noEmpty() &
      for j ← 2 to n        \\ n stands for dimension
      do
            s.top().x[j] ≤ P[i].x[j])
      do s.pop();
            s.push(P[i]);
      output the contents of stack s;
```

Conclusion

- Brute Force approach is discussed, design of some algorithms is also discussed.
- Algorithms computing maximal points is generalization of sorting algorithms
- Maximal points are useful in Computer Sciences and Mathematics in which at least one component of every point is dominated over all points.
- In fact we put elements in a certain order
- For Brute Force, formally, the output of any sorting algorithm must satisfy the following two conditions:
  - Output is in decreasing/increasing order and
  - Output is a permutation, or reordering, of input.