unit-3

BRUTE FORCE

There are several methods to design an algorithm for a given problem. Brute-Force is one such technique. It is a straight forward method to solve a given problem based on the statement of the problem and definitions of the concepts involved. For example, if we want to compute x^n then without any other work, we will just multiply x by itself for n number of times. The definition based matrix multiplication algorithm, the consecutive integer checking algorithm for finding gcd etc are few examples of brute force.

We may come across a question - in which situation, the brute force method is applicable? To answer, we can consider the following facts -

* Brute force algorithm is useful for solving small-size instances of a problem.
* It is useful for many important algorithms such as computing the sum of n numbers, finding largest element in list etc.

* For the problems like searching, sorting, matrix multiplication, string matching etc, the brute-force method results in reasonable algorithms of some practical value with no limitation on input size.
* Suppose, only few instances of a problem must be solved. Then, there is no meaning in designing most efficient algorithm by spending much time. In such case, brute-force gives a simple methodology to solve the problem with reasonable speed.
* Brute-force can be used as a yard-stick with which other alternate problems can be solved & efficient algorithms can be chosen.

Now, we will discuss some of the algorithms that designed using brute-force technique.

Selection Sort:

This is a sorting algorithm where we compare the first element of the list with all other elements. If first element is found to be greater than the compared element, then they are exchanged. In the second iteration, the second element -

is compared with all other elements starting from third element. The process continues and till the sorted list is available.
So, if there are n elements in the list, there will be n-1 iterations. Let us number the iterations from 0 to n-2. Then, at the i^{th} iteration, the $(n-i-1)^{th}$ element is compared with the last $(n-i)$ elements and exchange happens if required.

The algorithm is as below -

```
ALGORITHM Selection(A[0..n-1])
// Sorting a list by selection sort
// Input: An array A[0..n-1]
// Output: Array A[0..n-1] in ascending order
for i ← 0 to n-2 do
    pos ← i
    for j ← i+1 to n-1 do
        if A[j] < A[pos]
            pos ← j
    swap A[i] and A[pos].
```

Chetana Hegde
9886830185

Analysis:

1. The parameter is obviously, the input size n.
2. The basic operation is comparison.
3. The number of times the basic operation gets executed depends only on input size.

So,

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [n-1-(i+1)+1]$$

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

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

$$= n(n-2-0+1) - \frac{(n-2)(n-1)}{2} - (n-2-0+1)$$

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

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

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

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

$\therefore C(n) \in \Theta(n^2)$.

[NOTE: Find proper c_1, c_2 and n_0].

Bubble Sort :-

In this algorithm, the first element is compared with the second element and if first element is greater than second, they are exchanged. Then second element is compared with the third & so on. At the end of first iteration, the largest element will be placed in its proper position. In the second iteration again we start with comparing first and second & so on till the last-but-one element. The process is continued till we get the sorted list.

Thus, for a list of n elements, we need (n-1) iterations. The algorithm is as below –

```
ALGORITHM Bubble(A[0..n-1])
// Sorting an array using bubble sort method
// Input : An array A[0...n-1]
// Output: A sorted array.
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j]
            swap A[j] and A[j+1]
```

Analysis:

1. The parameter is the input size n.
2. The basic operation is comparison.
3. The basic operation depends only on the input size n.

And, there is one comparison for each value of i and for each value of j.

$$\therefore C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

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

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

$$= \frac{n(n-1)}{2} \quad \text{(As in selection sort)}$$

$$\therefore C(n) \in \Theta(n^2)$$

Chelana Heade
9th#8301894

NOTE: We can observe that the complexity of both selection sort and bubble sort algorithm is $\Theta(n^2)$. But, in the worst-case, selection sort requires $(n-1)$ exchanges and bubble sort requires $\frac{n(n-1)}{2}$ exchanges. Thus, there is a difference between these algorithms.

Sequential Search

We have discussed the algorithm and analysis of sequential search. In that, the key element is compared with every element till either match found or list is terminated. We can change the algorithm slightly by putting the key element at the end of existing list so that it always be a successful search. The algorithm is as below -

```
ALGORITHM Sequential(A[0..n], K)
// Sequential search by putting key at the end.
// Input: Array of n elements & key K.
// Output: The position of the first element
//         in A[0..n-1] whose value is equal
//         to K or -1 if no such element exists.
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i+1
if i < n
    return i
else
    return -1
```

String Matching

This problem involves with ~~matching~~ checking whether one particular string exists in other string. i.e. We will consider one string of length 'n' and call it as 'text'. Then take one more string of length 'm' (where $m \leq n$) and call it as 'pattern'. Then our problem is to check whether pattern exists in text or not. The procedure for this is as below —

Take the first character of 'text' and the first character of 'pattern'. If they matches, consider second character of 'text' and second character of 'pattern'. If they matches, take third pair & so on. Do this process until either all the character matches or an unmatched pair is found. If there is an unmatched pair, the start the process by taking second character of 'text' and first character of 'pattern', third character of 'text' and second character of 'pattern' & so on. Thus, at the i^{th} step, we will be

comparing -

$t_i = P_0$, $t_{i+1} = P_1$, ..., $t_{i+j} = P_j$, ..., $t_{i+m-1} = P_{m-1}$.

(where, t_i is i^{th} character of 'text' and P_j is j^{th} character of pattern.)

Note that for the 'text' of length n and the 'pattern' of length m, we need to consider only first (n-m) characters of 'text'. Because, after (n-m) have been completed in text, there will not be sufficient number (i.e. m) of characters remaining in the 'text' to match with 'pattern'. So, there is no meaning in checking thereafter.

Consider an example to illustrate this -

Let our text be -

Hello, How are you ?

$t_0\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ t_8\ t_9\ t_{10}\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t_{16}$

Hello, How are you ?

$t_0\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ t_8\ t_9\ t_{10}\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t_{16}\ t_{17}\ t_{18}$

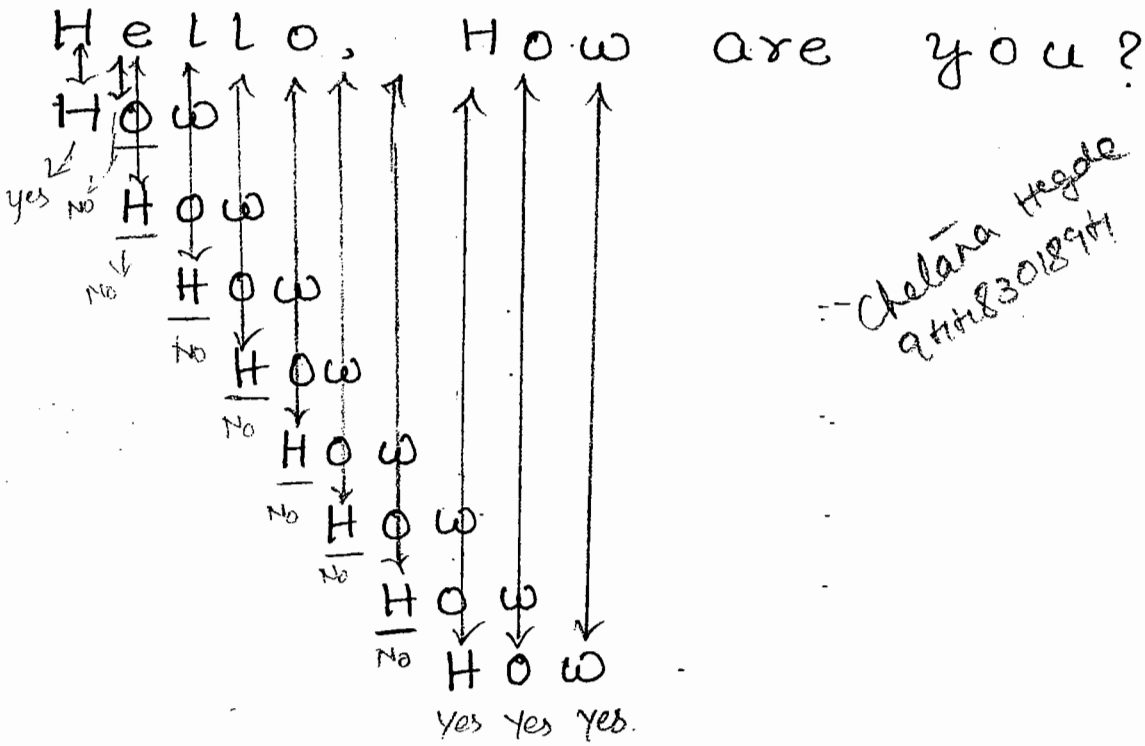
Let the pattern be -

H o w

P_0 P_1 P_2

So, $n = 19$ and $m = 3$.

The procedure is as shown —



:- Chetana Hegde
9th 8301894

Thus, by this straight-forward method we can check whether the given pattern exists in a text or not. If match is found, the position of the first character in the text that starts the first matching substring is returned otherwise, -1 is returned.

ALGORITHM StringMatch(T[0..n-1], P[0..m-1])

// Implements string matching by brute-force

// Input: An array T[0..n-1] of length 'n' is text.

// An array P[0..m-1] of length m is pattern.

// Output: The position of first character in the

// text that starts the first matching

// substring if successful, if -1 otherwise.

```
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
```

Analysis (Worst-Case)

1. There are two parameters m & n.
2. The basic operation is comparison.

Analysis :-

1. There are two parameters m & n.
2. The basic operation is comparison.

3. As the time complexity of the algorithm not only depends on the input size m and n, but it depends on the position of occurence of pattern in text or the non-occurence also. So, we go for both best case and worst case analysis.

Best-case: If the pattern of length m is found in the text at the very first position only ie if first m characters of the text matches with the pattern, then that will be best possibility. For this reason, we have to compare first m characters.

So, $C_{best}(n) = m$

As $C_{best}(n) \neq m$, we can easily say that-

$C_{best}(n) \geq m$

$\therefore C_{best}(n) \in \Omega(m)$

ie The time complexity in best case analysis of string matching is m.

Worst-case:

In the algorithm, the comparison of P[j] and T[i+j] is done once for

each value of j starting from 0 to m. Moreover, this is done for each value of i ranging from 0 to n-m.

Thus,

$$C(n) = \sum_{i=0}^{n-m-1} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m-1} (m-1-0+1)$$

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

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

$$= m\,(n-m-1-0+1)$$

$$= m(n-m)$$

$$= mn - m^2$$

$$\simeq mn, \quad \text{for } n >> m.$$

Thus, $C(n) \in \Theta(mn)$,

i.e. time complexity of string matching at worst case is of order mn.

Chetana Hegde
9th Aug 2020, 9 pm

Exhaustive Search

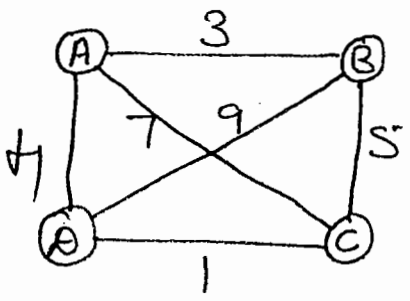
The problems involving combinatorial objects such as permutations, combinations, subsets of a set etc. requires finding an element with a special property in a domain that grows exponentially with an instance size. Such problems will usually be optimization problems ie finding out the element that maximizes or minimizes some characteristic like transportation cost, profit after production etc.

Exhaustive search is a brute-force technique for solving combinatorial problems. This technique generates each and every element of the problem's domain that are satisfying the constraints of the problem. Then the desired element as per the requirement of the problem will be selected. The implementation of exhaustive search requires an algorithm for generating certain combinatorial objects. But few of the problems that uses exhaustive search are discussed here viz. travelling salesman problem, knapsack problem and assignment problem.

Travelling Salesman Problem :-

This problem involves finding the shortest path through 'n' cities that visits each city exactly once before returning to the starting city. This can be thought of as finding the shortest Hamiltonian circuit of a weighted graph, where vertices of the graph being cities and the distance between the cities is considered as weights. We know that a Hamiltonian circuit is a sequence of $n+1$ vertices $v_0 v_1 \ldots v_n v_0$ with first and last vertices of the sequence being same and other $n-1$ vertices are being distinct.

Consider an example of TSP which can be solved by exhaustive search technique. Below shown is a graph representing the cities A, B, C and D with their mutual distances.



If we assume that the salesman starts with the city A, then the possible routes with the distances are given below –

$A \to B \to C \to D \to A$	dist. = 13
$A \to B \to D \to C \to A$	dist. = 20
$A \to D \to C \to B \to A$	dist. = 13
$A \to D \to B \to C \to A$	dist. = 25
$A \to C \to B \to D \to A$	dist. = 25
$A \to C \to D \to B \to A$	dist. = 20

Having all possible routes from A we can observe that the first and third routes results in optimal distance ie 13. So, salesperson can opt either

$A \to B \to C \to D \to A$ or

$A \to D \to C \to B \to A$.

It can easily observed that there will be $(n-1)!$ permutations or routes for n cities. So, the time complexity of TSP is of order $(n-1)!$, as we will choose optimal route only after finding all these $(n-1)!$ routes. Thus, $C(n) \in O((n-1)!)$.

Knapsack Problem

The problem is stated as — There is a knapsack of capacity W. There are 'n' items of weights $w_1, w_2, \ldots, w_n$ and values $v_1 \ldots v_n$. We have to find the most valuable subset of the items that fit into the knapsack. i.e. our problem is to

maximize $\sum_{i=1}^{n} v_i a_i$

subject to the constraint - $\sum_{i=1}^{n} w_i \cdot a_i \leq W$.

Here a_i denotes the i^{th} item to be put into the knapsack.

To illustrate, consider an example —

Let the capacity of knapsack, $W = 30$
Let there be three items (i.e. $n=3$) A, B & C.
Let their weights be 18, 18 and 22.
Let their values be ₹1, 28 & 50.

Now, we have to find the subset of a set $\{A, B, C\}$ so that -

* total weight ~~must~~ can be at the most 30
* total value of selected items is maximum.

The exhaustive search approach to this problem suggests to pick-up all possible subsets of the given set & to compute the total weights of all these subsets with the values & then select the feasible subset.

So, for the given example, we proceed as given below –

Subsets	Total weight	Total value	Feasible?
{ }	0	0	Yes
{A}	18	₹1	Yes
{B}	8	28	Yes
{C}	22	50	Yes
{A, B}	18+8=26	₹1+28=69	Yes
{A, C}	18+22=₹0	₹1+50=91	No
{B, C}	8+22=30	28+50=78	Yes
{A, B, C}	18+8+22 =₹8	₹1+28+50 =119	No

Here, the weights for the subsets {A, C} and {A, B, C} are exceeding the capacity of the knapsack. So, they are not feasible.

Out of other remaining feasible subsets, the one with maximum value is the subset {B, C}, which is the solution of the problem.

Note that for any knapsack problem with n items, we have to find all the possible subsets. As we know, there will be 2^n subsets for a set containing n elements. So, for solving knapsack problem, we have to find out 2^n subsets. This means that the time complexity of this algorithm is $\Omega(2^n)$.

Chetana Hegde
9th Feb 2018

Assignment Problem

Assignment problem involves assigning n different jobs to n different people. There will be cost incurred for assigning a job to a person. We have to assign the jobs so that the total cost is minimum. The cost incurred for assigning i^{th} person to the j^{th} job is denoted by C_{ij}, for $i = 1, 2, \ldots, n$ & $j = 1, 2, \ldots, n$

~~Consider one example for illustration.~~

So, our problem is to

minimize $\sum_{i=1}^{4} \sum_{j=1}^{4} c_{ij} x_{ij}$

Subjected to constraints -

$$x_{ij} = \begin{cases} 1, & \text{if } i^{th} \text{ person is assigned } j^{th} \text{ job} \\ 0, & \text{otherwise.} \end{cases}$$

& $\sum_{j=1}^{4} x_{ij} = 1$, i.e. exactly one job is done by i^{th} person.

Now, consider one example for illustration -

	J1	J2	J3	J4
P1	7	10	13	8
P2	9	2	5	12
P3	3	15	4	9
P4	13	5	10	7

Here, J1, J2, J3, J4 are jobs and P1, .., P4 are persons. The values are costs & such a matrix is known as cost matrix.

The exhaustive search techniques says that we have to find out all possible n-tuples like $(J_1, \ldots, J_n)$ for a problem of n jobs. Then find out the cost incurred for all these tuples and select the one feasible tuple which results in minimum cost.

Hence, for the example, we have tuples like-

$(J_1, J_2, J_3, J_4) \rightarrow 7+2+4+7 = 20$

$(J_1, J_2, J_4, J_3) \rightarrow 7+2+9+10 = 28$

$(J_1, J_3, J_2, J_4) \rightarrow 7+5+15+7 = 34$

$(J_1, J_3, J_4, J_2) \rightarrow 7+5+9+5 = 26$

$(J_1, J_4, J_2, J_3) \rightarrow 7+12+15+10 = 44$

$(J_1, J_4, J_3, J_2) \rightarrow 7+12+4+5 = 28$

⋮ etc ⋮

As, in this example we are having 4 jobs, there will be $4! = 24$ tuples. We have to find one sequence with minimum cost.

Thus, in general, to get feasible solution for an assignment problem with n jobs, we have to find out n! number of permutations or n-tuples. Hence, the time complexity of the assignment problem will be $O(n!)$.

NOTE: Apart from expensive algorithm like exhaustive search technique, we have 'Hungarian method' for solving assignment problem.