Unit 5 Brute Force Method

Structure:

5.1 Introduction

In the earlier unit you studied about the mathematical analysis of algorithms. In this unit you will study about brute force method in detail with algorithms.

Brute force is a problem solving technique wherein we compute a series of possible answers and test each possible answer for accuracy. It simply tries all possibilities until a satisfactory solution is found. Once it finds the value for the best solution it stops. In this unit we will discuss various algorithms which make use of the brute force method.

Objectives:

After studying this unit, you should be able to:

- define brute force method
- describe and analyze selection sort and bubble sort

- analyze and compute sequential search and brute force string matching
- explain and discuss the exhaustive search

5.2 Brute Force

Let us first define brute force.

Definition – Brute force is defined as a primitive programming technique where the programmer relies on the processing strength of the computing system rather than his own intelligence to simplify the problem. Here the programmer applies appropriate methods to find a series of possible answers and tests each possible answer for accuracy.

Let us now discuss the brute force algorithm.

5.2.1 Brute force algorithm

Brute force algorithm is a basic algorithm which works through every possible answer until the correct one is found. We can solve a particular problem easily using brute force algorithm rather than wasting time on creating a more elegant solution, especially when the size of the problem is small.

A good brute force algorithm should have the following factors in it.

- Small number of sub solutions
- Specific order in the sub solutions
- Sub solutions must be evaluated quickly

Let us take an example of counting change.

Consider a cashier who counts some amount of currency with a collection of notes and coins of different denominations. The cashier must count a specified sum using the smallest possible number of notes or coins.

Let us now analyze the problem mathematically.

Consider n as number of notes or coins and the set of different denominations of currency $P = \{p_1, p_2, ... p_n.\}$.

Let d_i = denomination of p_i

In the Indian system p_i = { Rs 1, Rs 2, Rs 5, Rs 10,..} the d_i = { 1, 2,5, 10..}
If we have to count a given sum of money A we need to find the smallest subset of P such that $\sum_{pi \in S} d_i$ = A.

Let us represent the subset S with n variables as $X = \{x_1, x_2, ...x_n.\}$

Such that

$$x_i = \begin{cases} 1 & P_i \in S \\ 0 & P_i \notin S \end{cases}$$

For {d1,d2…..dn} we have to minimize as

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

Such that

$$\sum_{i=1}^{n} d_i x_i = A$$

Since each element of X, $X = \{x_1, x_2, ...x_n.\}$ is either equal to zero or one, there will be 2^n possible values for any X in an algorithm. The best solution for brute force algorithm to solve a problem is to compute all the possible values, given any variable X.

For each possible value of X, we check whether the constraint $\sum_{i=1}^{n} d_i x_i = A$

is satisfied for it or not. A value that satisfies the constraint is called as a feasible solution. An objective function $\sum_{i=1}^{n} x_i$ is associated with an optimization problem determining how good a solution is.

Since there are 2^n possible values of X, we assume that the running time of brute-force solution is $\Omega(2^n)$. The running time needed to determine whether a possible value of a feasible solution is O(n) and the time required to compute the objective function is also O(n) is $O(n2^n)$.

Activity 1

Write a brute force algorithm to compare 25 text characters in an array and match with one character.

Self Assessment Questions

1. A value that satisfies the constraint is called a _____.
2. _____ is a function that is associated with an optimization problem determining how good a solution is.
3. The running time needed to determine whether a possible value of a feasible solution is O(n) and the time required to compute the objective function is also O(n) is _____.

5.3 Selection Sort and Bubble Sort

In the previous section we analyzed how brute force algorithm works. Now in this section let us analyze and implement brute force algorithms for selection sort and bubble sort.

5.3.1 Selection sort

First, let us define selection sort.

Selection sort is one of the simplest and performance oriented sorting techniques that work well for small files. It has time complexity as $O(n^2)$ which is unproductive on large lists.

Let us see an example for selection sort. In figure 5.1, we use selection sort to sort five names in alphabetical order.

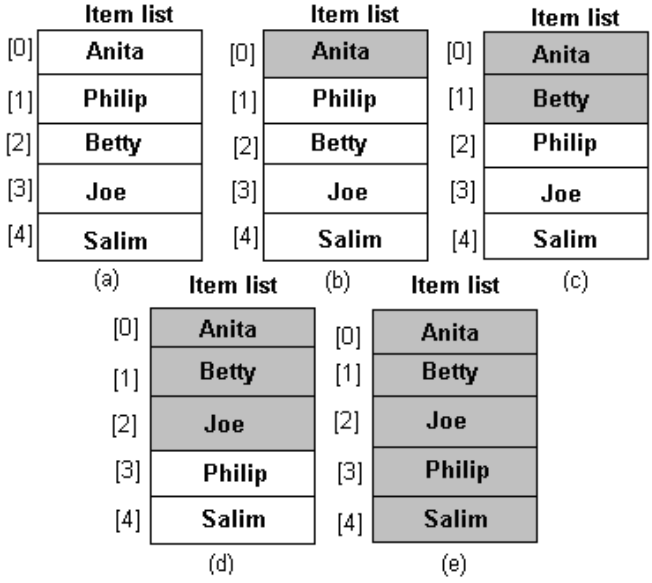

Figure 5.1: Example of Selection Sort

Here we compare consecutive names and exchange it if not in order.

Implementation of selection sort

The Selection sort spends most of its time in finding out the minimum element in the "unsorted" part of the array. Selection sort is quadratic in both worst and average case and needs no extra memory.

For each i from 1 to n - 1, there exists one exchange and n - i comparisons. So there will be total of n - 1 exchanges and (n - 1) + (n - 2) + . . . + 2 + 1 = n (n - 1)/2 comparisons. These observations will not bother about what input is given as data. In the worst case, we assume that time complexity of selection sort is quadratic, but in the average case, we consider that time complexity is O(n log n). This implies that the running time of selection sort is quite insensitive to the input.

Pseudocode implementation

Let us find the smallest element in the array and exchange it with the element which is in first position. Similarly we shall find the second smallest element and exchange it with the element that is present in the second position. We continue this process until the entire array is sorted. Let us see the pseudocode implementation for selection sort.

Pseudocode for selection sort (ascending order)

```
Selection_Sort (A)
for i ← 1 to n-1 do
   min j ← i;
   min y ← A[i]
   for j ← i + 1 to n do
     If A[j] < min y then
        min j ← j
          min y ← A[j]
   A[min j] ← A [i]
   A[i] ← min y
```

If the array is already sorted in descending order then the worst case occurs. "If A[j] < min y" is computed exactly the same number of times in every case then the variation in time is only due to the number of times the "then" part (i.e., min j ← j; min y ← A[j] of this test is executed.

Analysis

We can analyze selection sort very easily compared to other sorting algorithms since none of the loops depend on the data that is present in the array. For selecting the lowest element in the array we scan all n elements (this takes n − 1 comparisons) and then swap it with the first position. For finding the next lowest element we scan the remaining n − 1 element and so on for (n − 1) + (n − 2) + ... + 2 + 1 = n (n − 1) / 2 Є $\Theta(n^2)$ comparisons. Each of these scans require one swap for n − 1 elements (because the final element is already in place).

Comparison with other sorting algorithms

Amongst simple average-case $\Theta(n^2)$ algorithms, selection sort always outperforms bubble sort, insertion sort and gnome sort. Insertion sort's main advantage is that it can only scan as many elements as it needs in order to place the k + 1st element, while selection sort scans all remaining elements to find the k + 1st element.

Another key difference which we can observe is that selection sort always performs $\Theta(n)$ swaps while insertion sort performs $\Theta(n^2)$ swaps in the average and worst cases because generally swaps require writing to the array. In such case, selection sort is more preferable.

5.3.2 Bubble sort

Definition – A bubble sort is a sorting algorithm that continuously moves through a list swapping the items till they appear in a correct order. Bubble sort is the simplest sorting algorithm.

We can execute bubble sort by iterating it down an array (that has to be sorted) from the first element to the last and compare with each pair of elements and switch their positions if required. We should repeat this process till the array is sorted.

Performance of algorithm

Let us now analyze the performance of bubble sort algorithm. For analyzing the algorithm, we assume an array containing elements to be sorted. Now we will look through the array and pick the smallest element to put it in position 1. This is the first pass. For second pass, we need to consider the remaining list from the second element to the last in order to put the next smallest element in position 2 and so on till we sort all the elements.

For instance, let us consider the array of numbers as A[4] = {4, 5, 3, 2}. (A[0]=4, A[1]=5, A[2]=3, A[3]=2) Pictorial representation of how sorting (first pass and second pass) is performed is shown in figure 5.2.

In the first pass of figure 5.2, we check for the smallest element in the array. The smallest element is located at .A[3] i.e. '2'. Now we swap '2' with '3' (that is located at A[2]). Let us compare A[1] and A[2]. We find '2' as the smallest element. So we swap '2' with '5' (that is present in A[1]). Let us now compare A[0] with A[1]. After comparison we swap 2 with the element that is present in A[0]='4'. Hence the order of elements in first pass is 2, 4, 3, 5.

In second pass of figure 5.2, we take the order of elements obtained from the first pass i.e. 2 4 3 5. Let us compare A[3] with A[2] and swap the smallest element. So '3' is swapped with '5'. In the next step we compare A[2] with A[1] and swap '2' with '5'.In the last step, we compare A[1] with A[0] and swap '2' with '4'. Hence after sorting the elements, the order of the elements is 2 3 4 5.

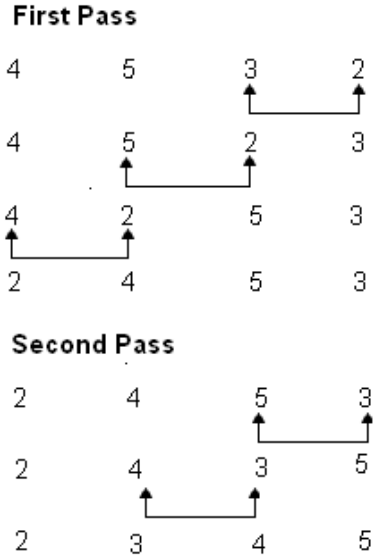

Figure 5.2: Bubble Sort

Example code for sorting the array elements

```
for i = n down to 2
   for j = 1 to i-1
      if A[j] < A[j+1]
         swap(A,i,j)
```

Algorithm analysis

In the above example, the outer loop runs n times. The complexity in this analysis lies in the inner loop. If we run the inner loop for a single time, we get a simple bound by noting that it can never loop more than n times. Since the outer loop makes the inner loop to complete n times, we cannot compare more than $O(n^2)$ times. This seems to be a very high bound because when we run the inner loop for last time, it will only make one comparison which is less than n. When we run the inner loop for first time, it will make n-1 comparisons, then next time it will make n-2 comparisons; and so on. Therefore, the actual number of comparisons is $\sum_{k=1}^{n-1} k$ that has a value of (n-1)n/2 which is also $O(n^2)$. Thus bubble sort has worst, best and average case run-time of $O(n^2)$.

Let us now discuss the pseudocode of bubble sort for sorting an array of integers.

Pseudocode for bubble sort to sort an integer array

```
procedure bubble sort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length(A) - 2 inclusive do:
      if A[i] > A[i+1] then
        swap ( A[i], A[i+1] )
        swapped: = true
      end if
    end for
  while swapped
end procedure
```

Optimizing bubble sort

We can optimize bubble sort easily after each pass by observing the smallest element that will always move up the array. Let us assume a list of size n. The nth element will be in its final place. Hence we can sort the remaining n - 1 elements. Now after this pass, the n - 1^{st} element will be in its final place. This allows us to skip over a lot of the elements and helps in tracing of the "swapped" variable. This will ultimately lead to a worst case of

50% improvement in iteration count but will have no improvement in swap counts.

We optimize bubble sort for sorting an integer array using the below pseudocode implementation.

Pseudocode for optimized bubble sort for sorting an integer array

```
procedure bubble Sort( A : list of sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    n := n – 1
    for each i in 0 to n  do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

We then do bubbling passes on smaller parts in an increasing order in the list. To be more precise, instead of doing n^2 comparisons and swaps, we can use (n-1) + (n-2) + ... + 1 comparison which will sum up to n(n - 1) / 2 comparisons.

Activity 2

Write an algorithm to sort four elements in an array list.

Self Assessment Questions

4. Selection sort is one of the simplest and _____ sorting techniques.
5. Bubble sort has _____, best and average case run-time of O(n2).
6. _____ is the simplest sorting algorithm.

5.4 Sequential Search and Brute Force String Matching

The previous section helped us to analyze the implementation of selection sort and bubble sort algorithms. In this section we will deal with the implementation of sequential search and brute-force string matching algorithms.

5.4.1 Sequential search

Let us first define sequential search.

Definition – Sequential search is a process for finding a particular value in a list that checks every element (one at a time) in sequence till the desired element is found.

Sequential search is the simplest brute force search algorithm. It is also known as linear search. This search is a special case of brute-force search. The algorithm's worst case cost is proportional to the number of elements in the list and its expected cost. Therefore, if the list contains more than a few elements then other methods like binary search or hashing becomes more efficient.

Consider an example as given in figure 5.3.

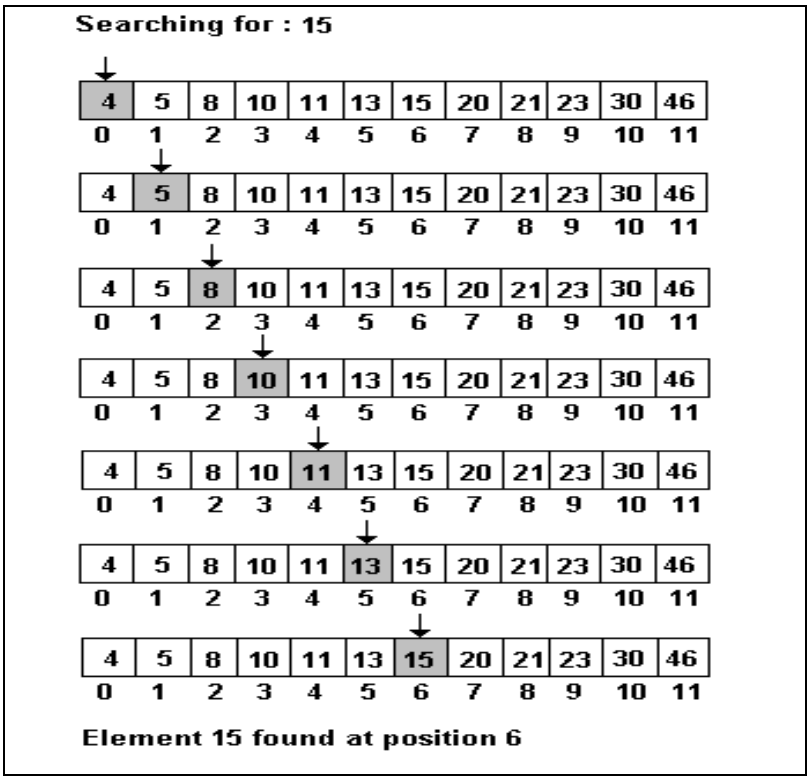

Figure 5.3: Example for Sequential Search

Here we are searching for the element 15 in a sorted list sequentially, by comparing every element with the given value.

Implementation of the algorithm

We can implement sequential search very easily. This search is very practical when we perform a single search in an unordered list or when list contains only few elements.

Let us analyze how pseudocode implementation is performed in various ways.

Onward iteration of the algorithm

The pseudocode portrays a typical variant of sequential search where the result of the search is assumed to be either the location of the list item where the desired value is present or an invalid location (Λ) to show that the desired element does not occur in the list.

Pseudocode for onward iteration of the sequential search algorithm

```
For each item in the list:
    if particular item has desired value,
      stop the search and return to the location where the item is present.
Return Λ.//Invalid location
```

In this pseudocode implementation, we execute the last line only after all list items are examined with none matching.

If we store the list as an array data structure then the location of the item at the index of the list will be usually between 1 and n, or 0 and n−1. In such case the invalid location (Λ) will be any index before the first element (such as 0 or −1, respectively) or after the last element (n+1 or n, respectively).

Recursive version

We will next describe the recursive algorithm of sequential search.

Pseudocode for recursive version of sequential search algorithm

```
If the list is empty, return Λ;// Invalid location
else
      if the first item of the list has the desired value, return its location;
      else search the value in the remainder of the list and return the
result.
```

Analysis

A list with n items has best case when the value of n is equal to the first element of the list and we do not need to do any comparisons in this case.

The worst case happens when the value is not in the list or appears only once at the end of the list and in this case we need n comparisons.

When the value which we require occurs k times in the list then the estimated number of comparisons are asymptotic. Hence O(n) is the worst-case cost and also the expected cost of sequential search.

Searching the array elements

We program sequential search in an array by stepping up an index variable until it reaches the last index. We normally require two comparison instructions for each list item. One, we use for checking whether the index has reached the end of the array and another for checking whether the item contains the desired value.

Let us consider an array A with elements indexed from1 to n. We have to search for a value x in the array. We can perform a forward search using pseudocode and this code returns n + 1 if the value is not found. Let us now see the pseudocode for forward search.

Pseudocode implementation for forward search

```
Set i to 1.
Repeat this loop:
    If i > n, then exit the loop.
      If A[i] = x, then exit the loop.
        Set i to i + 1.
Return i.
```

Let us now search array using pseudocode in the reverse order and return 0 when the element is not found.

Pseudocode implementation for reverse order

```
Set i to n.
Repeat this loop:
    If i ≤ 0, then exit the loop.
    If A[i] = x, then exit the loop.
     Set i to i − 1.
Return i.
```

We will next discuss the brute force string matching algorithm.

5.4.2 Brute Force string matching

If we are given a string of n characters known as text and a string of m characters (m <= n) known as pattern then we can find a substring of the text that matches the pattern using string matching algorithm. This implies that we find i – the index of the leftmost character of the first matching substring.

Let us now discuss an example of brute-force string matching algorithm.

Brute-force string matching algorithm

```
Algorithm BruteForceStringMatch (A [0…n-1], B [0…m-1])
// executes brute-force string matching
//Input: An array A [0…n-1] of n characters for a text, an array B [0…m-1]
of m characters for a pattern
//Output: The index of first character in the text starts a matching string or
-1 if the search is not successful\
For i ← 0 to n-m do
      j ← 0
        while j<m and B[j] = A[i+j] do
                j ← j+1
        if j=m return i
return -1
```

Let us trace brute-force string matching algorithm

Algorithm tracing for brute-force string matching

```
A[ ]= {S H I N E  B I K E}
B[ ]= {I N E}
n=9
m=3
For i ← 0 to 9-3 do
      j ← 0
        while 0<3 and B[0] = A[0] do
                0 ← 0+1
        if 0=3 return 0 //the final value returned is 2.
return -1
```

Let us consider a string of characters and analyze the string matching algorithm.

“SHINE BIKE” is the character string here. Let us see how the algorithm computes. INE is our pattern here with which we are going to match our string.

S H I N E _B I K E

I N E

I N E

I N E

In the above example, first the pattern I N E points to the first letter of the text which is ‘S’. As ‘I’ which is first letter of pattern I N E is not matched with ‘S’ , I N E checks for the next letter of the text i.e. ‘H’ which is not a matching string again. Now the pattern I N E checks for the third letter in the text which is ‘I’. Text letter ‘I’ matches with pattern’s first letter i.e. ‘I’. It then checks if the next letter of the pattern ‘N’ matches with the next character of the text which is also ‘N’. And thus the text I N E (last three letters of S H I N E) matches with the pattern I N E.

Let us discuss the Naïve string matching algorithm which is a type of brute force string matching algorithm.

Naïve String Matching algorithm

The naïve approach tests all the possible placements of Pattern A [1 . . . p] relative to text T [1 . . . q]. , we try to shift s = 0, 1. . . q - p, successively and for each shift compare T[s +1 . . . s + p] to A [1 . . p]. Let us now discuss the naïve string matching algorithm.

We can interpret naïve string-matching procedure graphically as a sliding pattern A [1 . . . p] over the text T [1 . . . q] and note down for which shift all of the characters in the pattern are matching the corresponding characters in the text.

In this execution, we use notation A [1 . . . j] to represent the substring of A from index i to index j. i.e., A [1 . . . j] = A[i] A [i +1] . . . A[j]. Let us analyze the algorithm when there is no substring of T matching A.

Naïve String Matching algorithm

```
Naive_String_Macher (T, A)
q ← length [T]
p ← length [A]
for s ← 0 to q-p do
   j ← 1
   while j ≤ p and T[s + j] = A[j] do
       j ← j +1
   If j > p then
      return valid shift s
return no valid shift exist  // i.e., there is no substring of T matching A.
```

Let us trace the above algorithm.

Algorithm tracing for Naïve String Matching algorithm

```
T[ ] = [S H I N E  B I K E]
A[ ] = [I N E]
Naive_String_Matcher (T, A)
q ← 9
p ← 3
for s ← 0 to 1 do
   j = 1
  while j ≤ 3 and T[1] = A[1] do
      j = 2
   If j > 3 then
      return valid shift s
return no valid shift exist  // i.e., there is no substring of T matching A.
```

When we refer to implementation of naïve matcher, we see that the for-loop in line 3 is executed at most q - p + 1 times and the while-loop in line 5 is executed at most m times. Hence we can say that the running time of the algorithm is O((q - p +1) p) which is clearly O(pq). We say this algorithm is in the quadratic time when the length of the pattern m is roughly equal in worst case. We assume one worst case as that the text T has n number of A's and the pattern A has (p - 1) number of A's which is followed by a single B.

Self Assessment Questions

7. _____ is also known as linear search.
8. We program sequential search in an array by _____an index variable until it reaches the last index.
9. In this pseudocode implementation, we execute the _____ only after all list items are examined with none matching.

5.5 Exhaustive Search

In the above section you studied how to analyze and implement sequential search and string matching algorithms. In this section we will profoundly analyze the implementations of exhaustive search.

5.5.1 Definition

Exhaustive search or Brute force search (also called as generate and test) is a trivial but general problem-solving technique that systematically specifies all possible candidates for the solution and checks whether each candidate satisfies the problem's statement.

However, we know that its cost is proportional to the number of candidate solutions which in many practical problems tend to grow very quickly as problem size increases. Therefore, we use exhaustive search typically when the problem size is limited and when implementation is more important than speed.

5.5.2 Implementation of the algorithm

For implementing exhaustive search to a specific class of problems, we need to follow four procedures. They are first, next, valid and output. To solve the problem, these procedures will take a parameter (data p) for particular instance.

Algorithm procedure for exhaustive search

We should follow the given below algorithm procedure to implement exhaustive search.

Algorithm procedure for exhaustive search
First (P): Generating a first candidate solution for P.
Next (P, c): Generating the next candidate for P after the current one c.
Valid (P, c): Verifying whether candidate c is a solution for P.
Output (P, c): Using the solution c of P as an appropriate to the application.

We implement Next procedure when there are no candidates for the instance p after the current one. A convenient way for us is that we can return a “null candidate “(some conventional data value Λ) that is distinct from real candidate. In the same way we implement First procedure when there are no candidates for instance p and we return some conventional data value, Λ.

5.5.3 Reordering the exhaustive search

Some applications require only one solution rather than preferring all the solutions. In such cases, expected running time of exhaustive search often depends on the order in which the candidates are tested. As a general rule, we should test the most promising candidates first. For example, when we search for a proper divisor of a random number n we should enumerate the candidate divisors in increasing order from 2 to n - 1 because the probability we obtain when we divide n by c is 1/c.

5.5.4 Speeding up exhaustive search

One way by which we can speed up the exhaustive search algorithm is to reduce the search space i.e. we can give the set of candidate solutions to specific problem class by using experienced based techniques which help in problem solving.

For example let us consider the problem of finding all integers between 1 and 1000 that are evenly divisible by 20. A naive brute-force search solution will generate all integers in the range (from 1 to 1000) testing each of them for divisibility. However, this problem can be solved much more efficiently by starting with 20 and repeatedly adding 20 until the number exceeds 1000 which takes only 50 (i.e.1000/20) steps and no tests. This way we can speed up the search in finding the integers that are divisible by 20.

5.5.5 Alternatives to the exhaustive search

We can use experience – based techniques (heuristics) to make an early cutoff of parts of the search. One example for exhaustive search is the minimax principle for searching game trees that eliminates many sub trees in the search at an early stage.

We can reduce the search space for problems by replacing the full problem with a simplified version. For example in a computer chess game we compute a more limited tree of minimax possibilities rather than computing the full minimax tree of all possible moves for the remainder of the game.

Hope you are clear about the brute force method and the different algorithms that use this method.

Self Assessment Questions

10. Exhaustive search implementation is more important than _____.
11. Exhaustive search algorithm gives the _____ for every candidate that is a solution to the given instance P.
12. Exhaustive search is typically used when the problem size is _____.

5.6 Summary

It is very essential for us to obtain a best algorithm for any analysis. Brute force method is a mathematical proof which helps in simplifying the finite number of classes of each case and proves each case separately for analysis.

We analyzed the performance of selection sort and bubble sort algorithms and implemented the algorithms in pseudocode with suitable examples and figures.

Sequential search and brute–force string matching algorithms are the simplest algorithms to implement. In this unit we examined the performance of sequential search algorithm in a systematic way. We implemented sequential search algorithm in the following ways - forward iteration, recursive version, searched in an ordered list and reverse order search.

Exhaustive search is a method which helps us in determining all the possible candidates for the solutions and helps in verifying whether the candidates satisfy the problem's solution.

5.7 Glossary

Term	Description
Pseudocode	Pseudocode is an artificial and informal language that helps programmers to develop algorithms.
Heuristics	Heuristics is an adjective for experience-based techniques that help in problem solving, learning and discovery.
Gnome sort	Gnome sort is a sorting algorithm which is similar to insertion sort but moves an element to its proper place by a series of swaps as in bubble sort.
Substring	Substring of a string is a subset of the symbols in a string where order of the elements is preserved.

5.8 Terminal Questions

1. Explain brute-force algorithm?
2. Explain selection sort algorithm implementation with suitable example.
3. Give the pseudocode implementation of bubble sort algorithm.
4. Explain the analysis of sequential search algorithm with examples.
5. What is exhaustive search and how it is implemented?

5.9 Answers

Self Assessment Questions

1. Feasible solution
2. Objective function
3. $O(n2^n)$.
4. Performance oriented
5. Worst
6. Bubble sort
7. Sequential search
8. Stepping up
9. Last line
10. Speed
11. Output
12. Limited

Terminal Questions

1. Refer section 5.2.1 – Brute-force algorithm
2. Refer section 5.3.1 – Selection sort
3. Refer section 5.3.2 – Bubble sort
4. Refer section 5.4.1 – Sequential search
5. Refer section 5.5.1 – Definition of exhaustive search

References

- Rashid Bin Muhammad. Design and Analysis of Computer Algorithms.

E-Reference

- http://caveshadow.com/CS566/Sabin%20M.%20Thomas%20-%20String%20 Matching%20Algorithms.ppt
- http://www.cse.unl.edu/~ylu/csce310/notes/BruteForce.ppt.
- http://www.cs.miami.edu/~burt/learning/Csc517.051/notes/selection.html
- http://cprogramminglanguage.net/c-bubble-sort-source-code.aspx
- http://knol.google.com/k/bubble-sort# cs.unco.edu/course/CS101/F06/Chapter09.ppt
- http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm
- http://webspace.ship.edu/cawell/Sorting/bubanal.htm