

Comparative Study on Different Algorithms

Raduan Ahamad 191-15-12938

Faraz Ahamed 191-15-12948

Sazid Nawas Shovon 191-15-12929

M Shahriar Ishtiaque 191-15-12938

Sakib Rokoni 191-15-12961

Comparative Study on Different Algorithms

Introduction

Algorithms.

An algorithm is a finite sequential set of instructions which, if followed, accomplish a particular task or a set of tasks in a finite time. Complexity: The complexity of an algorithm is a function $g(n)$ that gives the upper bound of the number of operations performed by an algorithm when the input size is n . Complexity are divided in two ways. Time complexity is the amount of time the computer requires to execute the algorithm & Space complexity of an algorithm is the amount of memory space the computer requires, completing the execution of the algorithm. In case of algorithm searching is the process to finding to location of the given data elements in the data structure. The different types of searching techniques are Linear search, Binary search. In a modern time where hardware and computing powers are in rapid advancement and invention are often, then why we need algorithms.

For a concrete example [1], let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million numbers, computer A takes

2. $(10^7)^2$ instructions = 20,000 seconds (more than 5.5 hours)

10^{10} instruction/second

While computer B takes

5. $(10^7) \lg 10^7$ instructions = 1163 1163 seconds (less than 20 minutes)

10^7 instruction/second

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.

Linear (Sequential) Search is the basic and simple method of searching: It is a method where the search begins at the end of the list, scans the elements of the list from left to right until the desired record is found. In Binary search the entire sorted list is divided into two parts. We first compare our input item with the mid element of the list & then restrict our attention to only the first or second half of the list depending on whether the input item comes left or right of the mid element. In this way we reduce the length of the list to be searched by half. Less time is taken by binary search to search an element from the sorted list of elements. So, we can conclude that binary search method is more efficient than the linear search. Binary search algorithm is efficient because it is based on divide-and-conquer strategy; which divides the list into two parts and searches one part of the list thereby reducing the search time

CONCEPT BEHIND SEARCHING Algorithms

CONCEPT BEHIND SEARCHING PROCESS In linear search, each element of an array is read one by one sequentially and it is compared with the desired element. A search will be unsuccessful if all the elements are read and the desired element is not found. Where Binary search is an extremely efficient algorithm when it is compared to linear search. Binary search technique searches —data in minimum possible comparisons. Suppose the given array is a sorted one, otherwise first we have to sort the array elements. Then apply the following conditions to search a —data. 1) Find the middle element of the array (i.e., $n/2$ is the middle element if the array or the sub-array contains n elements). 2) Compare the middle element with the data to be searched and then there are following three cases. a) If it is a desired element, then search is successful. b) If it is less than desired data, then search only the first half of the array, i.e., the elements which come to the left side of the middle element. c) If it is greater than the desired data, then search only the second half of the array, i.e., the elements which come to the right side of the middle element. Repeat the same steps until an element are found or exhaust the search area. Again, for searching an ordered array, Interpolation search is used. This method is even more

efficient than binary search, if the elements are uniformly distributed (or sorted) in an array A. Interpolation search is a method of retrieving a desired record by key in an ordered file by using the value of the key and the statistical distribution of the Keys [3]. Consider an array A of n elements and the elements are uniformly distributed (or the elements are arranged in a sorted array). Initially, as in binary search, low is set to 0 and high is set to n – 1. Now we are searching an element key in an array between A[low] and A[high]. The key would be expected to be at mid, which is an approximately position. $mid = low + (high - low) \times ((key - A[low]) / (A[high] - A[low]))$. If key is lower than A[mid], reset high to mid–1; else reset low to mid+1. Repeat the process until the key has found or low > high.

PROCEDURE OF SEARCHING

A. Linear or Sequential Search:

Unsorted array

5 4 21 16 25 3 15
 5 4 21 16 25 3 15
 5 4 21 16 25 3 15
 5 4 21 16 25 3 15
 5 4 21 16 25 3 15
 5 4 21 16 25 3 15
 5 4 21 16 25 3 15

Conclusion: The element 15 is not present inside that array.

Sorted array

4 5 9 11 13 10
 4 5 9 11 13 10
 4 5 9 11 13 10
 4 5 9 11 13 10
 4 5 9 11 13 10

Conclusion: The element 10 is not present inside that

B. Binary Search:

Suppose we have an array of 7 elements

9	10	25	30	40	45	70
0	1	2	3	4	5	6

Following steps are generated if we binary search a data = 45 from the above array.

Step 1:

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 0;
 UB = 6 ;
 $mid = (0 + 6) / 2 = 3$
 $A[mid] = A[3] = 30$

Step 2:

Since (A [3] < data) - i.e., 30 < 45 – reinitialize the variable

LB, UB and mid

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 3; UB = 6;
 $mid = (3 + 6) / 2 = 4$
 $A[mid] = A [4] = 40$

Step 3:

Since (A [4] < data) - i.e., 40 < 45 - reinitialize the variable LB, UB and mid

9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 4;
 UB = 6;
 $mid = (4 + 6) / 2 = 5$
 $A[mid] = A [5] = 45$

Step 4:

Since (A [5] == data) - i.e., 45 == 45 - searching is successful.

C. Interpolation Search:

Consider 7 numbers. 2, 25, 35, 39, 40, 47, 50

Step1:

Suppose we are searching 50 from the array.

Here n = 7

Key = 50

low = 0

high = n – 1 = 6

$mid = 0 + (6 - 0) \times ((50 - 2) / (50 - 2))$

= $6 \times (48 / 48) = 6$ if (key == A[mid])

$\Rightarrow key == A [6] \Rightarrow 50 == 50 \Rightarrow key$ is found.

Step 2:

Say we are searching 25 from the array

Here n = 7

Key = 25

low = 0

high = n – 1 = 6

$mid = 0 + (6 - 0) \times ((25 - 2) / (50 - 2))$

= $6 \times (23 / 48)$

= 2.875

Here we consider only the integer part of the mid, i.e.,

$mid = 2$ if (key == A[mid]) $\Rightarrow key == A[2] \Rightarrow 25 == 25 \Rightarrow key$ is found.

Step 3:

Say we are searching 34 from the array

Here n = 7

Key = 34

low = 0

}

}

Use a 2-D array to store the computed $\text{lcs}(m, n)$ value at $\text{arr}[m-1][n-1]$ because the string index starts from 0. Whenever the function with an equivalent argument m and n are called again, don't perform any longer recursive call and return $\text{arr}[m-1][n-1]$ because the previous computation of the $\text{lcs}(m, n)$ has already been stored in $\text{arr}[m-1][n-1]$, hence reducing the recursive calls that happen more than once.

Time Complexity: $O(M*N)$, where M and N is length of the first and second string respectively.

Auxiliary Space: $(M*N)$

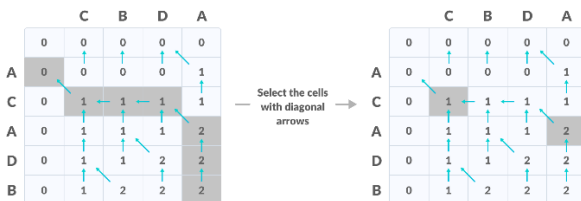
LCS with tabulation method (Bottom Up)

First, we create a table of dimension $(n+1)*(m+1)$ where n and m are the lengths of strings X and Y respectively. The first row and the first column are filled with 0.

If the character of previous rows and column are matching then fill the slot with previous diagonal slots number and adding 1 with it.

Else fill the slot with the maximum number of previous slot of rows and column.

After filling the table, the last number of last row and last column is the length of longest common subsequence. And if we want to find the sequence, we need to backtrack the table.



Here the longest common subsequence is CA.

```
void lcs(char *S1, char *S2, int m, int n) {
```

```
    int LCS_table[m + 1][n + 1];

    // Building the matrix in bottom-up way
    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                LCS_table[i][j] = 0;
            else if (S1[i - 1] == S2[j - 1])
                LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
            else
                LCS_table[i][j] = max (LCS_table[i - 1][j],
                LCS_table[i][j - 1]);
        }
    }
    int index = LCS_table[m][n];
```

```
    char lcs[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (S1[i - 1] == S2[j - 1]) {
            lcs[index - 1] = S1[i - 1];
            i--;
            j--;
            index--;
        }

        else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
            i--;
        else
            j--;
    }

    // Printing the sub sequences
    cout << "S1 : " << S1 << "\nS2 : " << S2 << "\nLCS: " << lcs << "\n";
}
```

The time complexity of this method is $O(m*N)$. m stands for rows and n stands for column. Because we need to backtrack $m*n$ element in the worst case.