CSI15_386_Paper.doc

by

FILE	CSI15_386_PAPER.DOC (472K)		
TIME SUBMITTED	14-AUG-2015 12:02PM	WORD COUNT	2641
SUBMISSION ID	561202240	CHARACTER COUNT	11554

DN Searching Technique

Deep Chandra Andola
Assistant Professor
Amrapali Institute of Technology & Sciences,
Haldwani, India
deep.andola@gmail.com

Nitin Deepak
Assistant Professor
Amrapali Institute of Technology & Sciences,
Haldwani, India
nitin.d12@gmail.com

Abstract – **Searching in computer science is the method which finds the required item in the given data. Linear Search is the method for finding a particular value in the list that checks each element in sequence until the desired element is found or the list is exhausted. The list needs not to be ordered. The complexity for the linear search is θ (n).**
In this paper, the algorithm is discussed takes less time with respect to linear search. Time taken by the DN Search algorithm is equal to time taken by Binary Search Algorithm, but this algorithm also works on unsorted data.

Keywords – searching, complexity, Linear, Binary Search

I. INTRODUCTION TO ALGORITHMS

In mathematics, an algorithm is a self-contained step-by-step set of operations to be performed. An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function [1] [2].

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s) for the target machines to produce output from given input (perhaps null). In addition every algorithm must satisfy the following criteria:

- **Input:** there are one or more quantities which are externally supplied.
- **Output:** At least one quantity is produces.
- **Definiteness:** Each instruction must be clear and unambiguous.
- **Finiteness:** If we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after the finite number of steps.
- **Effectiveness:** Every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it is also be feasible.

II. UNDERSTANDING COMMUNCATION OF TIME & SPACE TRADE-OFF FOR ALGORITHM

A problem may have numerous solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run, and choose the better of the two. You don't need to know how many minutes and seconds they will take, but you do need some way to compare algorithms against one another [1] [4].

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized (not comparable) units of computational work. Consider, for example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the square of the number of cards in the deck. This quadratic behavior is the main term in the complexity formula, it says, e.g., if you double the size of the deck, then the work is roughly quadrupled.

Now let us consider how we would go about comparing the complexity of two algorithms. Let f (n) be the cost, in the worst case, of one algorithm, expressed as a function of the input size n, and g (n) be the cost function for the other algorithm. E.g., for sorting algorithms, f (10) and g (10) would be the maximum number of steps that the algorithms would take on a list of 10 items. If, for all values of n >= 0, f (n) is less than or equal to g(n), then the

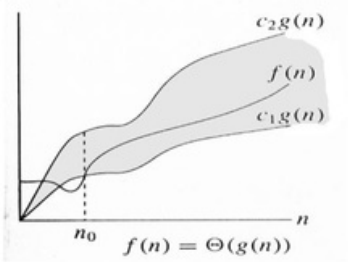
algorithm with complexity function f is strictly faster. But, generally speaking, our concern for computational cost is for the cases with large inputs; so the comparison of f (n) and g(n) for small values of n is less significant than the "long term" comparison of f(n) and g(n), for n larger than some threshold.

The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.

1. θ Notation: The theta notation bounds functions from above and below, so it defines exact asymptotic behavior.

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants.

For a given function g (n), we denote **θ** (g (n)) is following set of functions.

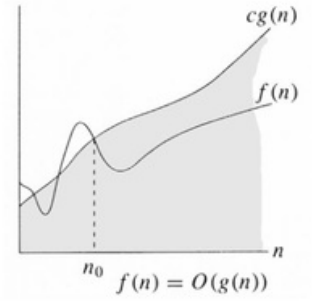


θ (g(n)) = {f (n): there exist positive constant c1, c2 and n_0 such that

0 <= c1*g (n) <= f (n) <= c2*g(n) for all n >= n_0

2) Big 'Oh' (O) Notation: The Big *O* notation defines an upper bound of an algorithm; it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O (n^2). Note that O (n^2) also covers linear time.

The Big O notation is useful when we only have upper bound on time complexity of an algorithm.

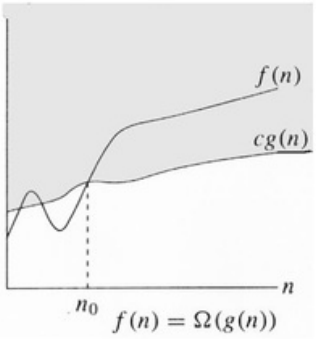


O (g(n)) = {f (n): there exist positive constant c and n_0 such that

0 <= f (n) <= c*g(n) for all n >= n_0

3) Ω Notation: Just as Big *O* notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation can be useful when we have lower bound on time complexity of an algorithm.



Ω (g (n)) = {f (n): there exist positive constant c and n_0 such that

0 <= c*g (n) <= f (n) for all n >= n_0

III. LINEAR SEARCH

In computer science, linear search or sequential search is a method for finding a particular value in a list that checks each element in sequence until the desired element is found or the list is exhausted. The list need not be ordered. Linear search is the simplest search algorithm; it is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list. Its expected cost is also proportional to the number of elements if all elements are searched equally.

Algorithm for Linear Search

Here '*A*' is the linear array with '***n***' elements and '***item***' is the given item of information. This algorithm find the location '***loc***' of item in *A*, or set loc = 0 if the search is unsuccessful. [3].

linear search (A, item)

1. [insert item at the end of data, i.e. 'A']
   Set A [n+1] = item
2. [initialize counter]
   set loc = 1
3. [search for item]
   Repeat while A [loc] =! Item
   set loc = loc + 1

4. [successful]
   if loc = n + 1,
   else set loc = 0
5. Exit

alysis:

For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed. Thus worst cost of Linear Search is **θ (n)**.

IV. BINARY SEARCH

In computer science, a binary search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index, or position, is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

gorithm for Binary Search

Here '*A*' is sorted array with lower bound '*LB*' and upper bound '*UB*' and '*item*' is a given element of information. The variables ***beg***, ***end*** and ***mid*** denote the beginning, end and the middle location of the segment of element *A*. This algorithm finds the location ***loc*** of *item* in *A*. [3].

ary_search (A, UB, LB, item)

1. [initialize segment variables]
   set beg = LB
   end = UB
   mid = ((beg + end) / 2)
2. Repeat step 3 & 4
   While beg <= end and A[mid] =! item
3. if item < A[mid], then
   set end = mid -1
   else
   set beg = mid + 1
   [end of if structure]
4. set mid = int ((beg + end) / 2)
   [end of step 2 loop]
5. if A[mid] = item
   then set loc = mid
   else
   set loc = NULL
6. Exit

Anal *:*

Log (N) −1 is the expected number of probes in an average successful search, and the worst case is log (N), just one more probe. If the list is empty, no probes at all are made. Thus binary search is a logarithmic algorithm and executes in **θ (log n)** time. In most cases it is considerably faster than a linear search.

As Linear & Binary Search Techniques are discussed and their average time can be given as:

SNo	Technique	Data List Format	Average e
1.	Linear Search	Any	T (n) = **θ (n)**
2.	Binary Search	Sorted	T (n) = **θ (log n)**

Table1: Comparison between Linear & Binary Search

Since, time taken by the Binary Search is less than that of Linear Search. But the constraint in binary search is that list should be sorted.

V. DN SEARCH

DN Searching Algorithm is the searching technique. This algorithm is similar to Linear Search algorithm, i.e. it works on unsorted data. But DN searching technique takes less time then Linear Search. Time taken by this algorithm is equal to the time take by the Binary Search Algorithm, other than Binary searching technique should apply on sorted data only, if data or list is in unsorted then Binary searching technique is not valid. So, DN Searching technique can be applied on sorted or unsorted data or list. Algorithm for DN Searching Technique is discussed below:

Algorithm for DN Algorithm

Here, '*A*' is the list with '***n***' element
'***num***' item to search in *A*
'*B*' is the array of length 2

B [1] store '***1***' for TRUE and

B [2] store position of the result

'***search()***' function is used to compare

```
        DN_Search (A, num)
Line1:      for (i = 1 to n-1)
Line 2:     {       sum = A[i] + A[i+1];
Line3:              if ((sum – num) >= 0)
Line4:                      B = search (A, i,
            i+1, num);
Line5:              if (B[1] = = 1)
Line6:                      break;
Line7:              i = i + 2; }
Line 8:     if (B [1] != 1 && i <= len)
Line9:      {       if (num = =A[i])
Line10:     B[1]=1;
Line11:     B[2]=i;         }
Line12: if ( B[1] = = 1)
Line13: {       pos = B[2];
Line14:     then print "element is at A[pos]
        position";      }       // successful
Line15: else
Line16:     print "NOT FOUND";
            // unsuccessful


        search (A, strt, end, num)
Line1s:     {for (k = strt to end)
Line2s:     {       if (num = = A[i])
Line3s:             x[1]=1;  x[2]=i;
Line4s:                     return x; }
Line5s:             return 0;       }
```

Analysis:

Best Case: Suppose the array A is:

A = {6, 5, 7, 3, 8, 9, 1}

Item to be searched is:

num = 5

for i = 1, (**Line1**)

sum = 11 (**Line2**) // sum = A[i] + A[i+1]

for **Line3**, condition will be TRUE, since (sum - num) >=0

Now, in **Line4** ***search()*** will be call as A = {6,5}, strt = 1, end = 2, num = 5,

In **Line2s**, num == A[end]

➔ x = {1, 2} and it will return to **Line4**

In **Line4** ➔ B = {1, 2} and in next line compiler will break from *for* loop and moves to **Line8** and given conditions will be false, so compiler will directly moves to **Line12** where the condition is true.

Thus output will be ***element is at 2 position***, i.e. successful

Analysis: In the above algorithm, result is successful for i = 1. Thus **T (n) = 1**

Worst Case: Suppose the array A is:

A = {6, 5, 7, 4, 8, 9, 1}

Item to be searched is:

num = 15

for i = 1, (**Line1**) sum = 11 (**Line2**) // sum = A[i] + A[i+1] for **Line3**, condition will be FALSE, since (sum - num) < 0

In **Line7**, 'i' will be incremented by 2 ➔ i = 3

for i = 3, (**Line1**) sum = 10 (**Line2**) // sum = A[i] + A[i+1] for **Line3**, condition will be FALSE, since (sum - num) < 0

In **Line7**, 'i' will be incremented by 2 ➔ i = 5

for i = 5, (**Line1**) sum = 17 (**Line2**) // sum = A[i] + A[i+1] for **Line3**, condition will be TRUE, since (sum - num) >= 0

Now, in **Line4** ***search()*** will be call as A = {8, 9}, strt = 5, end = 6, num = 15,

In **Line2s**, num will not equal to A [i]

➔ this will return '0' and compiler will move to **Line4** as B = {0, 0}, thus condition of **Line5** will be false and compiler will to **Line7**

In **Line7**, 'i' will be incremented by 2 ➔ i = 7

Now, **Line1** condition will be FALSE and compiler will directly jumps to **Line8**

In **Line8**, '*if*' condition will get true and *num* will compare with last element, A [7], which is again not equal to '*num*'. Compiler will now moves to **Line15** after checking if condition in **Line12** and output will be *NOT FOUND*, i.e. unsuccessful

Analysis: In above algorithm, *num* was not found (unsuccessful) in the given list *A*. Thus, total time take by algorithm (in worst case) will **T (n) = θ (log n)**.

For '**n**' elements, algorithm will execute for maximum **(n/2) + 1** time, because of **Line7**, which increment value of 'i' by 2. Thus worst time taken by the algorithm is **θ (log n)**.

VI. RESULT

As all three searching techniques are discussed and their average time can be given as:

SNo	Technique	Data List Format	Average Time
1.	Linear Search	Any	T (n) = **θ (n)**
2.	Binary Search	Sorted	T (n) = **θ (log n)**
3.	DN Search	Any	T (n) = **θ (log n)**

Table2: Comparison between Linear, Binary Search & DN Search Technique

Since, average time taken by DN and Binary Search Technique are same. But DN Searching Technique is more powerful because it can be implemented on unsorted data or list, while Binary Search needs only sorted list.

VII. CONCLUSION

In this paper, Linear and Binary Search techniques were discussed with their average time (in Table1), in which binary search was superior to linear search with lesser time. In Table2, when new search technique (DN Search) was compared with other two, time taken by the binary and DN search was equal and better than that of linear search. But the limitation of binary search was dominated by DN search, that is, DN search technique can be applied on any type of list (sorted or unsorted).

Thus, DN Search Technique can be proved as more powerful technique than Linear & Binary Search.

REFERENCES

[1] Thomas H Cormen, Charles H Leiserson, Ronald L Rivest, Clifford Stein, "Introduction to Algorithm", 3rd Edition, Prentice Hall of India Pvt Ltd., Chapter 1 and Chapter 3.

[2] Ellis Horowitz, Suraj Sahni, Sanguthevar Rajasekaran, "Fundament of Computer Algorithms", Galgotia Publication, Chapter 1.3.

[3] Seymour Lipshutz, Schaum's Outline "Data Structures", Tata McGraw Hill, Chapter 4.7 and Chapter 4.8.

[4] Seymour Lipshutz, Marc Lars Lipson, Varsha H Patil, Schaum's Outline "Discrete Structure", Third Edition, Tata McGraw Hill, Chapter 3.9

CSI15_386_Paper.doc

ORIGINALITY REPORT

53%	53%	17%	38%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	en.wikipedia.org Internet Source	11%
2	en.wikibooks.org Internet Source	7%
3	www.docstoc.com Internet Source	6%
4	www.geeksforgeeks.org Internet Source	6%
5	dev.integralminds.co.in Internet Source	4%
6	serghei.net Internet Source	4%
7	mpi.edu.bd Internet Source	4%
8	www.codingtiger.com Internet Source	2%
9	www.expertsmind.com Internet Source	2%

10	www.cse.buffalo.edu Internet Source	2%
11	cs.anu.edu.au Internet Source	2%
12	vivekontech.blogspot.com Internet Source	1%
13	www.personal.kent.edu Internet Source	1%
14	Submitted to Gems Modern Academy Student Paper	1%
15	gateatzeal.com Internet Source	1%
16	milanalgorithms.blogspot.com Internet Source	<1%
17	tacklemantra.com Internet Source	<1%
18	thecodeaddict.wordpress.com Internet Source	<1%
19	topperspoint.in Internet Source	<1%

EXCLUDE QUOTES	OFF	EXCLUDE MATCHES	< 5 WORDS
EXCLUDE	ON		

BIBLIOGRAPHY