e-ISSN: 2458-8091

Journal Homepage: http://dergipark.gov.tr/ejsdr

EJSDR

European Journal of Sustainable Development Research

DergiPark AKADEMİK

CODEN: EJSDR

Empirical Analysis on the Running Time of a Searching Algorithm, Chunk Algorithm

Gazmend Xhaferi[1*], Florinda Imeri[2], Flamure Sadiku[3], Agon Memeti[4]

[1]*Gazmend Xhaferi, University of Tetova, Faculty of Math. &Nat. Sciences, Department of Informatics,1200, Tetovo, MK*

[2]*Florinda Imeri, University of Tetova, Faculty of Math.& Nat. Sciences, Department of Informatics,1200, Tetovo, MK*

[3]*Flamure Sadiki, University of Tetova, Faculty of Math. &Nat. Sciences, Department of Mathematics,1200, Tetovo, MK*

[4]*Agon Memeti, University of Tetova, Faculty of Math. &Nat. Sciences, Department of Informatics, 1200,Tetovo, MK,*

** Corresponding Author email: gazmend.xhaferi@unite.edu.mk*

Abstract

Searching and sorting, by no doubt, represent two of the most fundamental and widely encountered problems in computer science. Given a collection of objects, the goal of search is to find a particular object in this collection or to recognize that the object does not exist in the collection. A major goal of computer sciences is to understand and develop a solution for the particular problem. Typically solving the problem involves at least four steps: (1) design an algorithm, (2) analyze the correctness and efficiency of the procedure, (3) implement that procedure in some programming language, and (4) test that implementation. An important issue is to describe the efficiency of a given procedure for solving a problem. Informally, usually we speak in terms of "fast" or "slow" programs, but the absolute execution time of an algorithm depends on many factors such as: the size of the input, the programming language used to implement the algorithm, the quality of the implementation and the machine on which the code is run (a supercomputer is faster than a laptop). In this paper we will analyze the performances of a searching algorithm, precisely the chunk algorithm. In analyzing the efficiency of chunk algorithm, we will only concentrate on searching items, using the Chunk-Search Algorithm, on one-dimensional arrays with integers. We wanted to see how does different chunk size, input size (i.e., the "speed" of the algorithm as a function of the size of the input on which it is run), and the machine on which the code is run.

Key words

Chunk algorithm, computer performance, input size, chunk size

1. INTRODUCTION

There are some very common problems that we use computers to solve: searching through a lot of records for a specific record or set of records and sorting, or placing records in a desired order. At times we need to use both of these techniques as part of solving the same problem. There are numerous algorithms to perform searches and sorts.

A question you should always ask when selecting a search algorithm is "How fast does the search have to be?" The reason is that, in general, the faster the algorithm is, the more complex it is.

The concept of efficiency (or complexity) is important when comparing algorithms. For long lists and tasks, like searching, that are repeated frequently, the choice among alternative algorithms becomes important because they may differ in efficiency.

Before we can compare different methods of searching we need to think a bit about the time requirements for the algorithm to complete its task. We could also compare algorithms by the amount of memory needed.

An algorithm can require different times to solve different problems of the same size (a measure of efficiency). For example, the time it takes an algorithm to search for the integer '1' in an array of 100 integers depends on the nature of the array.

How can one describe the efficiency of a given procedure for solving some problem? Informally, one often speaks of "fast" or "slow" programs, but the absolute execution time of an algorithm depends on many factors:

- the size of the input (searching through a list of length 1,000 takes longer than searching through a list of length 10),
- the algorithm used to solve the problem (Unordered-Linear-Search is inherently slower than Binary-Search),
- the programming language used to implement the algorithm (interpreted languages such as Basic are typically slower than compiled languages such as C++),
- the quality of the actual implementation (good, tight code can be much faster than poor, sloppy code), and
- the machine on which the code is run (a supercomputer is faster than a laptop).

In analyzing the efficiency of an algorithm, one typically focuses on the first two of these factors i.e., the "speed" of the algorithm as a function of the input size and the machine on which it is run. Finally, when analyzing the efficiency of an algorithm, one often performs a worst case and/or an average case analysis.

A worst case analysis aims to determine the slowest possible execution time for an algorithm. For example, if one were searching through a list, then in the worst case, one might have to go through the entire list to find (or not find) the object in question. A worst case analysis is useful because it tells you that no matter what, the running time of the algorithm cannot be slower than the bound derived. An algorithm with a "good" worst case running time will always be "fast." On the other hand, an average case analysis aims to determine how fast an algorithm is "on average" for a "typical" input. It may be the case that the worst case running time of an algorithm is quite slow, but in reality, for "typical" inputs, the algorithm is much faster: in this case, the "average case" running time of the algorithm may be much better than the "worst case" running time, and it may better reflect "typical" performance.

Average case analyses are usually much more difficult than worst case analyses. In actual practice, the average case running time of an algorithm is usually only a constant factor (often just 2) faster than the worst case running time. Since worst case analyses are (1) interesting in their own right, (2) easier to perform than average case analyses, and (3) often indicative of average case performance, worst case analyses tend to be performed most often.

With this as motivation, we now analyze the performances of the algorithm Chunk-Search.

2. A REVIEW OF THE CHUNK-SEARCH ALGORITHM

Given an ordered list, one need not (and one typically does not) search through the entire collection one-by-one. Consider searching for a name in a phone book or looking for a particular exam in a sorted pile: one might naturally grab 50 or more pages at a time from the phone book or 10 or more exams at a time from the pile to quickly determine the 50 page (or 10 exam) "chunk" in which the desired data lies. One could then carefully search through this chunk using an ordered linear search. Let c be the chunk size used (e.g., 50 pages or 10 exams), and assume that we have access to a slightly generalized algorithm for ordered linear search, Encoding the above ideas; we have the chunk search algorithm [2].

Input: ordered objects array A, the number of objects n, chunk size c, key value being sought x.

Output: if found, return position i, if not, return message "x not found"

a. Cut array A into chunks of size c.
b. Compare x with the last elements of each chunk, except the last chunk! See if x is GREATER than that element.
c. If yes, check the next chunk
d. If no, that means x should be in that chunk
e. Execute Ordered Linear Search inside the chunk

A pseudo code for the algorithm chunk-search is as below:

```
Chunk-Search [A, n, c, x]
        high ← c
        while high < n and A[high] < x
        do high ← high + c
        high ← min{high, n}
        low ← max{high − c + 1, 1}
return Ordered-Linear-Search[A, low , high, x]
```

The call to Ordered-Linear-Search will be performed on a list whose size is at most c, and thus at most 2c additional comparisons will be performed (as described above). We therefore have

$$T(n) = n/c + 2c. \quad [1]$$

Note that the running time of Chunk-Search depends on both n and c. What does this analysis tell us? We can use this analysis, and specifically equation [1], in order to determine the optimal chunk size c; i.e., the chunk size which would minimize the overall running time of Chunk-Search (in the worst case).

Suppose that one were to run Chunk-Search using a very small value of c. Our chunks would be small, so there would be lots of chunks. Much of the time would be spent trying to find the right chunk.

Consider the extreme case of c = 1: in the worst case, n/c = n/1 = n comparisons would be spent trying to find the right chunk while only 2c = 2 compares would be spent searching within a chunk for a total of n + 2 compares (in the worst case). This is worse than Ordered-Linear-Search (though it is still linear)[1].

Now consider using a very large value of c. Our chunks would be big, so there would be few of them, and very few comparisons would be spent finding the right chunk. However, searching for the element in question within a very large chunk would require many comparisons. Consider the extreme case of c = n: in the worst case, n/c = n/n = 1 comparison would be spent "finding" the right chunk (our chunk is the entire list) while 2c=2n compares would be spent searching within a chunk for a total of 2n + 1 compares (in the worst case). This is worse than either Unordered-Linear-Search or Ordered-Linear-Search (though, again, it is still linear).

Is Chunk-Search doomed to be no faster than linear search? No! One must optimize the value of c in order to minimize the total number of comparisons, and this can be accomplished by choosing a value of c which balances the time (number of comparisons) spent finding the right chunk and the time spent searching within that chunk. Suppose that we wish to spend precisely equal amounts of time searching for the correct chunk and then searching within that chunk; what value of c should we pick? Our goal is then to find a c such that n/c (the time spent searching for a chunk) is equal to 2c (the time spent searching within a chunk)[4].

$$n/c = 2c$$
$$n = 2c^2$$
$$n/2 = c^2$$
$$c = \sqrt{n/2}$$

Thus for $c = \sqrt{n/2}$

$$T(n) = \frac{n}{c} + 2c = 2\sqrt{2n}$$

By this we answer the above question that for n/c = 2c, T(n) is optimized

Note that for sufficiently large n, this is much faster than a linear search. For example, if n=1,000,000, Ordered-Linear-Search would require 2,000,000 comparisons in the worst case, while Chunk-Search would require approximately 2,828 comparisons in the worst case— Chunk-Search would be approximately 707 times faster (in the worst case).

Do even better values of c exist? One can show through the use of calculus that $c = \sqrt{n/2}$ is optimal. We essentially have a function (n/c + 2c) which we wish to minimize with respect to c. Taking the derivative with respect to c, setting this derivative to zero, and solving for c yields $c = \sqrt{n/2}$.

3. RESULTS AND DISCUSSION

In this study we will only concentrate on searching items, using the Chunk-Search Algorithm, on one-dimensional arrays with integers. We wanted to see how does different

- chunk size, input size, and hardware (computers)

  influence the speed.

The tests were run in computers with different characteristics:

1. **CPU: Intel(R) Core (TM) i3 2.2 GHz, 6 GB RAM (PC1)**
2. **CPU: Intel® Core™ i7-4720HQ CPU @ 2.60GHz; 8,00 GB RAM (PC2)**

In order to test the speed of different input size, chunk size and different computer performances, we made a C++ program which runs Chunk-Search Algorithm several times for randomly-generated arrays of different size: 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 and 100000 items (integers). Measurements for the Chunk-Search algorithm, where the array will be divided into chunk size of **c = 1, c = 50, c = 100, c = 250, c = 500, c = 750, c = 1000**

The first experiment was conducted for different sizes of chunks, on different array size. Below are shown results

Table 1. The execution time for Chunk-Search algorithm, for different chunk and array size

	Input size/chunk size	c=1	c=50	c=100	c=250	c=500	c=750	c=1000
PC 1	10000	0.027	0.031	0.055	0.054	0.053	0.031	0.030
PC 2		**0.008**	**0.004**	**0.008**	**0.005**	**0.006**	**0.008**	**0.004**
PC 1	50000	0.037	0.026	0.056	0.054	0.057	0.025	0.028
PC 2		**0.011**	**0.012**	**0.011**	**0.012**	**0.028**	**0.008**	**0.012**
PC 1	75000	0.033	0.035	0.032	0.027	0.028	0.018	0.025
PC 2		**0.012**	**0.012**	**0.012**	**0.012**	**0.009**	**0.021**	**0.009**
PC 1	100000	0.017	0.019	0.021	0.027	0.015	0.012	0.018
PC 2		**0.015**	**0.012**	**0.012**	**0.008**	**0.008**	**0.012**	**0.013**

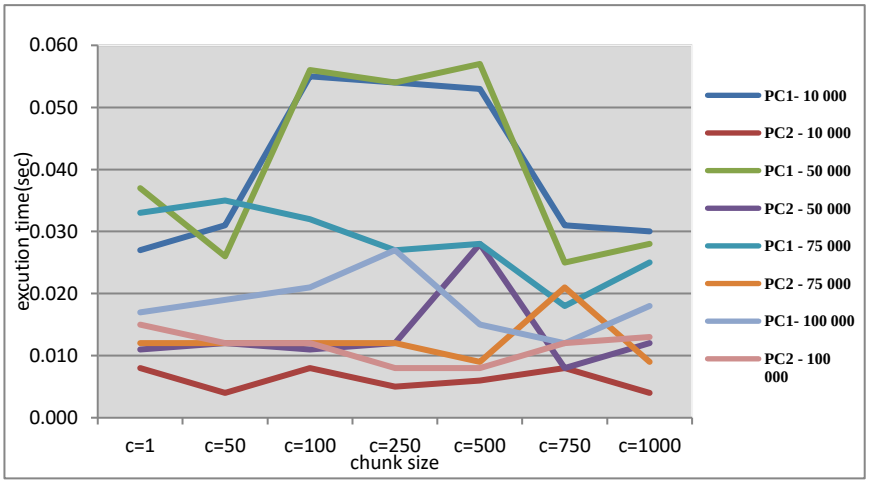

Figure 1. The graph for the Chunk-Search algorithm for different chunk and array size

The second experiment was conducted for different input size, for the chunk size which presents an optimum, **c = 1000**

Table 2. The execution time for the Chunk-Search algorithm to chunk size of 1000.

no. of elements in the array	no. of blocks	exec. time (PC 1) c=1000	exec. time (PC 2) c=1000
10 000	10	0.024	0.006
20 000	20	0.038	0.013
30 000	30	0.039	0.011
40 000	40	0.039	0.012
50 000	50	0.038	0.011
60 000	60	0.039	0.01
70 000	70	0.040	0.01
80 000	80	0.026	0.011
90 000	90	0.022	0.012
100 000	100	0.020	0.009

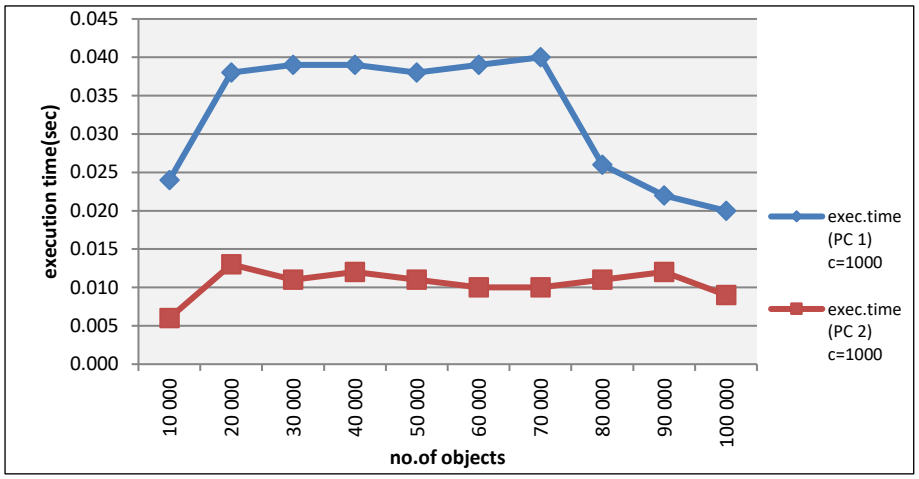

Figure 2. The graph for Chunk-Search algorithm to chunk size of 1000

The third experiment was conducted for different computer performances for the chunk size **c=100**:

Table 3. The execution time for the Chunk-Search algorithm to chunk size of 100

no. of elements in the array	no.of blocks	exec.time (PC 1) c=100	exec.time (PC 2) c=100
10 000	100	**0.029**	**0.007**
20 000	200	**0.054**	**0.015**
30 000	300	**0.053**	**0.010**
40 000	400	**0.048**	**0.010**
50 000	500	**0.038**	**0.010**
60 000	600	**0.032**	**0.011**
70 000	700	**0.020**	**0.009**
80 000	800	**0.018**	**0.011**
90 000	900	**0.018**	**0.011**
100 000	1 000	**0.019**	**0.010**

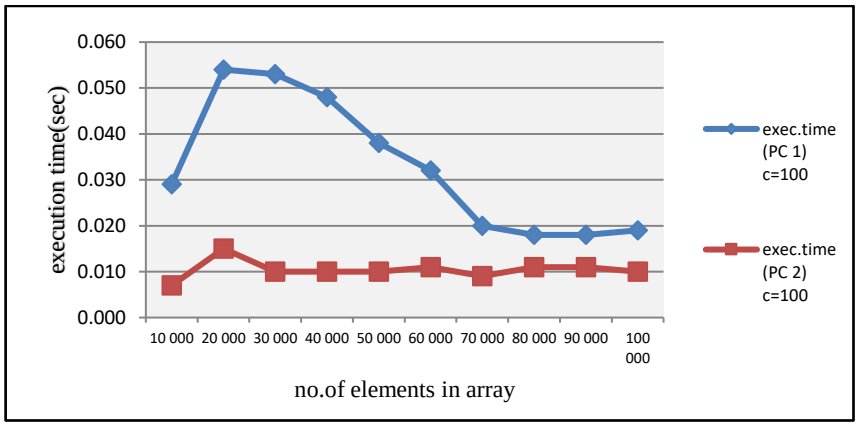

Figure 3. The graph for Chunk-Search algorithm to chunk size of 100

From the first experiment, the case 1, with the chunk size of 1, tab 1. and the fig 1., we see that with increasing of the input size, the overall execution time increases. Similarly to case 1 but with small change are the case 2. For the second case, where the chunk size is 1000, it is seen that with the increasing of the input size from 10000 to 50000 the execution time increases, and then decreases, tab.2 and fig.2. Similarly happens with the third case, where the chunk size is 100, tab.3. and fig.3.

From the empirical analysis of the algorithm and from the experimental one we can conclude that Chunk-Search algorithm has the fastest time in the cases when the chunk size is approximately equal to the number of elements in the input array. The best can be seen in the third case where the chunk size is 100 and the number of elements in the input array is

10000, where each chunk has 100 elements. So, in this case the execution time is 0.007 seconds, which is an optimal time or we can say that the algorithm performs the best.

In the third case it is also was analyzed and compared the algorithm Chunk-Search on two different computers with the different performances as regarding the CPU speed and memory capacity. Analyses were conducted for the chunk size of 100. The best is seen from table and graph above, tab.3 and fig.3, therefore, the execution time varies depending on the speed of the computer, which is with no doubt one of the factors that affect the speed of the algorithm.

4. CONCLUSIONS

From the results derived, shown in the above tables and graphs, and from the empirical analysis of algorithm Chunk-Search is seen that the execution speed of this algorithm is directly affected by the chunk size and the machine performances.

As it is shown empirically we get the optimum in the case when the chunk size and the input size $c = \sqrt{n/2}$. Also it is very clear from the third experiment that the performances of the machine affect the speed of algorithm, as the computers with the best performances suggest that the algorithm will be faster and vice versa.

5. REFERENCES

[1]. Profs. Aslam & Fell, *Analysis of Algorithms: Running Time* September 7, 2005. Available from http://www.psnacet.edu.in/courses/MCA/Design%20and%20analysis%20of%20Algorithms/Lecture1.pdf (Accessed June, 2016).
[2]. A. Levitin. *Introduction to the Design and Analysis of Algorithms*. 3e, Addison-Wesley, 2011. ISBN 978-0132316811.
[3]. Searching Arrays: Algorithms and Efficiency. Available from http://research.cs.queensu.ca/home/cisc121/2006s/webnotes/search.html (Accessed June, 2016).
[4]. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Bentley, Programming Pearls, Addison-Wesley
[5]. R. Sedgewick, *Bundle of Algorithms in C++*, Addison-Wesley, 2001, 3rd Edition.
[6]. Design and Analysis of Algorithms, Available from https://www.cs.cornell.edu/~kozen/papers/daa.pdf (Accessed October, 2015)