ISSN 1512-1232

QUICKIN SORT-A NEW SORTING APPROACH

[1] Rajesh Ramachandran, [2] Dr.E.Kirupakaran

[1] Sr. Lecturer, Dept. of Computer Science,
Naipunnya Institute of Management and Information Technology, Pongam, Kerala; Email: ryanrajesh@hotmail.com
[2] Sr. DGM (Outsourcing) BHEL, Trichy; Email: e_kiru@yahoo.com

Abstract

QuickIn Sort is a sorting algorithm that, makes O (n log n) (Big Oh notation) comparisons to sort n items. Typically, QuickIn Sort is significantly faster in practice than other O(n log n) algorithms , because its inner loop can be efficiently implemented on most architectures .This sorting method requires data movement, but less than that of insertion sort. This data movement can be reduced by implementing the algorithm using linked list. Major advantage of this sorting method is its behaviour pattern is same for all cases, i.e. time complexity of this method is same for best, average and worst case.

Introduction

Sorting is any process of arranging items in some sequence and/or in different sets, and accordingly, it has two common, yet distinct meanings:

1. ordering: arranging items of the same kind, class, nature, etc. in some ordered sequence,
2. Categorizing: grouping and labeling items with similar properties together (by sorts).

In computer science and mathematics, a **Sorting Algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly.

To analyze an algorithm is to determine the amount of resources (such as time and storage) necessary to execute it. Most algorithms are designed to work with inputs of arbitrary length. Usually the efficiency or complexity of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity). Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem. These estimates provide an insight into reasonable directions of search for efficient algorithms. In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, omega notation and theta notation are used to this end.

Time complexity

Time efficiency estimates depend on what we define to be a step. For the analysis to correspond usefully to the actual execution time, the time required to perform a step must be guaranteed to be bounded above by a constant. In mathematics, computer science, and related fields, Big Oh notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. Big O notation allows its users to simplify functions in order to concentrate on their growth rates: different functions with the same growth rate may be represented using the same O notation.

Although developed as a part of pure mathematics, this notation is now frequently also used in computational complexity theory to describe an algorithm's usage of computational resources:

ISSN 1512-1232
the worst case or average case running time or memory usage of an algorithm is often expressed as a function of the length of its input using big O notation.

Space complexity

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its space complexity is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too. Definition (Space complexity)

The space complexity of a program (for a given input) is the number of elementary objects that this program needs to store during its execution. This number is computed with respect to the size n of the input data.

There is often a time-space-tradeoff involved in a problem, that is, it cannot be solved with few computing time and low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

In addition to varying complexity, sorting algorithms also fall into two basic categories — comparison based and non-comparison based. A comparison-based algorithm orders a sorting array by weighing the value of one element against the value of other elements. Algorithms such as quick sort, merge sort, heap sort, bubble sort, and insertion sort are comparison based. Alternatively, a non-comparison based algorithm sorts an array without consideration of pair wise data elements. Radix sort is a non-comparison based algorithm that treats the sorting elements as numbers represented in a base-M number system, and then works with individual digits of M.

Another factor, which influences the performance of sorting method, is the behavior pattern of the input. In computer science, **best**, **worst** and **average cases** of a given algorithm express what the resource usage is *at least*, *at most* and *on average*, respectively. Usually the resource being considered is running time, but it could also be memory or other resources.

QuickIn Sort is a new sorting algorithm that, makes O (n log n) (Big Oh notation) comparisons to sort n items. Typically, QuickIn Sort is significantly faster in practice than other O (n log n) algorithms, because its inner loop can be efficiently implemented on most architecture. This sorting method requires data movement but less than that of insertion sort. This data movement can be reduced by implementing the algorithm using linked list. Major advantage of this sorting method is its behavior pattern is same for all cases, ie time complexity of this method is same for best, average and worst case.

Comparison of algorithms

S.No	Name	Average	Worst	Memory	Method
1	Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)	Exchange
2	Selection Sort	$O(n^2)$	$O(n^2)$	O(1)	Selection
3	Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)	Insertion
4	Shell Sort	---	O(n log n)	O(1)	Insertion

ISSN 1512-1232

5	Binary Tree Sort	O(n log n)	O(n log n)	O(n)	Insertion
6	Merge Sort	O(n log n)	O(n log n)	O(n)	Merging
7	Heap Sort	O(n log n)	O(n log n)	O(1)	Selection
8	Quick Sort	O(n log n)	$O(n^2)$	O(log n)	Partitioning

Methodology

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.

A problem may have numerous algorithmic 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. Alternatively, 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 do not need to know how many minutes and seconds they will take, but you do need some way to compare algorithms against one another.

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized 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.

The exact formula for the cost is more complex, and contains more details than are needed to understand the essential complexity of the algorithm. With our deck of cards, in the worst case, the deck would start out reverse-sorted, so our scans would have to go all the way to the end. The first scan would involve scanning 52 cards, the next would take 51, etc. So the cost formula is 52 + 51 + ... + 1. Generally, letting N be the number of cards, the formula is 1 + 2 + ... + N, which equals ((N + 1) * (N) / 2) = (N2 + N) / 2 = (1 / 2) N2 + N / 2. However, the N^2 term dominates the expression, and this is what is key for comparing algorithm costs. (This is in fact an expensive algorithm; the best sorting algorithms run in sub-quadratic time.)

Asymptotically speaking, in the limit as N tends towards infinity, 1 + 2 + ... + N gets closer and closer to the pure quadratic function (1/2) N^2. In addition, what difference does the constant factor of 1/2 make, at this level of abstraction? Therefore, the behavior is said to be O(n2).

In typical usage, the formal definition of Big O notation is not used directly; rather, the Big O notation for a function f(x) is derived by the following simplification rules:

If f(x) is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.

ISSN 1512-1232

If f(x) is a product of several factors, any constants (terms in the product, that do not depend on x) are omitted.

For example, let f(x) = 6x4 − 2x3 + 5, and suppose we wish to simplify this function, using O notation, to describe its growth rate as x approaches infinity. This function is the sum of three terms: 6x4, −2x3, and 5. Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of x, namely 6x4. Now one may apply the second rule: 6x4 is a product of 6 and x4 in which the first factor does not depend on x. Omitting this factor results in the simplified form x4. Thus, we say that f(x) is a big-oh of (x4) or mathematically we can write f(x) = O(x4).

Results and Discussions

QuickIn sort is a sorting algorithm that, makes O *(*n log *n)* (Big Oh notation) comparisons to sort *n* items. Typically, QuickIn sort is significantly faster in practice than other *O* (*n* log *n*) algorithms , because its inner loop can be efficiently implemented on most architectures and its behavior pattern is same for all cases

How it sorts

From the given set of unsorted numbers, take the first number name it as key K1 Read all the remaining numbers one by one. Compare the number with K1. If the number is greater than or equal to K1 then place the number right of K1 else place the number left of K1.Conitnue the same process for all the remaining numbers in the list. Finally, we will get two sub lists. One list with numbers less than K1 and the other with numbers greater than or equal to K1. Repeat the same process for each sub list. Continue this process until the sub list contains zero elements or one element.

Algorithm

QuickIn (N: Array of Numbers, K1, A: integers,)

Step1. Read the first number from N,

Let K1

Step2. Read the next number, let A

Step3 Compare A with K1

Step4 If A is greater than or equal to K1

Then

Place a right of K1

Else

Place a left of K1

Step5 If the list contains any more

Elements go to step 2

Step 6 Now we have 2 Sub list.

- ✓ First list with all values less than K1.
- ✓ Second list with values greater than or equal to K1.

Step7. If each list contains more than one

Element go to step1

ISSN 1512-1232

Example 1

Let unsorted numbers be

10,13,4,29,4,6,8,1,2,20,16,24,12,5,1

Step 1.
Read the first number, i.e 10
So here, key K1 is 10
Read the remaining number one by one

Read the next number, i.e. 13
Compare 13 and 10. Since 13 is greater than 10 place 13 right of K1. We get
10, 13
Read the next number, i.e. 4
Since 4 is smaller than 10 places, 4 left of 10.
We get
4, 10, 13
Read next number, ie 29, we get
4,10,13,29
Read next number, ie 4, we get
4, 4,10,13,29
Read next number, ie 6, we get
6,4,4,10,13,29
Read the next number , ie 8, we get
8,6,4,4,10,13,29
Read the next number, ie 1 , we get
1, 8,4,4,10,13,29
Read the next number, ie 2, we get
2,1, 8,4,4,10,13,29
Read the next number, ie 20, we get
2,1, 8,4,4,10,13,29,20
Read the next number, ie 16, we get
2,1, 8,4,4,10,13,29,20,16
Read the next number, ie 24, we get
2,1, 8,4,4,10,13,29,20,16,24
Read the next number, ie 12 we get
2,1, 8,4,4,10,13,29,20,16,24,12
Read the next number , ie 5, we get
5, 2,1, 8,4,4,10,13,29,20,16,24,12
Read the next number, ie 1 we get
1, 5, 2,1, 8,4,4,10,13,29,20,16,24,12

Now we get two lists one with values less than 10 and the other with values greater than or equal to 10
That is

List 1

1, 5, 2,1, 8,4,4

List 2

13,29,20,16,24,12

ISSN 1512-1232

Repeat the same process for each list.

List 1
1, 5, 2,1, 8,4,4
Here key K1 is 1. Read all the number one by one compare with K1.
We get

1, 5, 2,1, 8,4,4
Here we get two more sublists list 11 and list12
Here list11 contains zero elements so no need to process list11.Continue the same process with list12.
i.e
5, 2,1, 8,4,4
Process list12
We get
4,4,1,2,5,8
List121 is 4,4,1,2 and list 122 is 8
Repeat the same process for both we get

2,1,4,4
List 1211 is 2,1 and list 1212 is 4
Process list 1211
We get 1,2

Now the list is sorted and is
1,1,2,4,4,5,8
Continue the same process for list 2 , we get
12,13,16,20,24,29.

Therefore, the complete sorted list is
1,1,2,4,4,5,8 ,10, 12,13,16,20,24,29.

Example 2

Consider the unsorted numbers 8,6,4,2
Here the key K1 is 8
After the first comparison process w get
2,4,6,8
This is in sorted order but we have to continue the same process for each list but number of comparison here after will be less.

Example 3

Consider the unsorted numbers
2,4,6,8
Here the key K 1 is 2

After the first comparison process we get
2,4,6,8

In both examples s 2 and 3 we get immediate results if the list is already in sorted form in either ascending or descending.

ISSN 1512-1232

Suggestions

This sorting method is a combination of quick sort and insertion sort. Even though data movement in this sorting method is less compares to insertion sort, still it requires data movement when the element is less than K1.This data movement can be reduced by implementing the algorithm using linked list.

References

1. Sartaj Sahni, "Data Structures Algorithms and Applications in C++", 2nd Ed. University Press,2005
2. Aaron M Tanenbaum, Moshe J Augenstein, "Data Structures using C",Prentice Hall International Inc.,Emglewood Cliffs,NJ,1986
3. Robert L Cruse, " Data Structure and Program Design", Prentice Hall India 3rd ed.,1999
4. Sartaj Sahni,"Data Structures, Algorithms and applications in C++",University Press , 2nd Ed.,2005
5. Yedidyah Langsam,Moshe J Augenstein, Aaron M Tanenbaum " Data Structures using C and C++", Prentice Hall India, 2nd Ed. 2005
6. Mark Allen Weiss "Data Structures and Algorithm analysis in C++ ", Pearson Education, 3rd Ed., 2007
7. Ying Shi and Eushiuan Tran, 18-742 Advanced Computer Architecture, Carnegie Mellon University, Pittsburgh, Pennsylvania
8. www.en.wikipedia.org
9. http://www.leda-tutorial.org/en/official/ch02s02s03.html Space Complexity
10. Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956

Article received: 2010-01-06