University of Science

Sorting Algorithms - An Overview

Data Structures and Algorithms

Huynh Minh Tuan - 20120024@student.hcmus.edu.vn

November 2021

Abstract

Sorting is nothing but alphabetizing, categorizing, arranging, or putting items in an ordered sequence. It is a key fundamental operation in the field of computer science. It is of extreme importance because it adds usefulness to data. In this report, I have compared eleven common sorting algorithms (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort). I have developed a program in C++, Python and experimented with several input sizes 10,000, 30,000, 50,000, 100,000, 300,000, and 500,000 elements. The performance and efficiency of these algorithms in terms of CPU time consumption as well as the number of comparisons that have been recorded and presented in tabular and graphical form.

1. Introduction

Sorting is not a leap but it has emerged in parallel with the development of the human mind. In computer science, alphabetizing, arranging, categorizing, or putting data items in an ordered sequence on the basis of similar properties is called sorting. Sorting is of key importance because it optimizes the usefulness of data. We can observe plenty of sorting examples in our daily life, e.g. we can easily find required items in a shopping mall or utility store because the items are kept categorically.

The items to be sorted may be in various forms i.e. random as a whole, already sorted, very small or extremely large in numer, sorted in reverse order etc. There is no algorithm that is best for sorting all types of data. We must be familiar with sorting algorithms in terms of their suitability in a particular situation.

In this paper, I am going to compare eleven common sorting algorithms (Selection Sort, Insertion Sort, Bubble Sort, Shaker Sort, Shell Sort, Heap Sort, Merge Sort, Quick Sort, Counting Sort, Radix Sort, and Flash Sort) for their CPU time consumption and number of compared operations on four different data arrangements (Sorted data (in ascending order), Nearly sorted data, Revherse sorted data, and Randomized data).

2. Algorithm presentation

2.1. Selection Sort

Idea

The Selection Sort is based on the idea of finding the minimum element in an unsorted array and then putting it in its correct position in a sorted array.

Pseudo code

Algorithm 2.1: Selection Sort

```
   Input: a_1, a_2, ..., a_N
   Output: a_1, a_2, ..., a_N (in sorted)
1  for i ← 1 to N do
2  |  minIndex ← i
3  |  for j ← i + 1 to N do
4  |  |  if a_minIndex > a_j then
5  |  |  |  minIndex ← j
6  |  |  end
7  |  end
8  |  swap(a_minIndex, a_i)
9  end
```

Complexity

Best case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.2. Insertion Sort

Idea

The main idea of insertion sort is that array is divided in two parts which left part is already sorted, and right part is unsorted. Values from the unsorted part are picked and placed at the correct position in the sorted part. So, at every iteration sorted part grows by one element which is called key. During an iteration, if compared element is greater than key then compared element has to shift to right to open a position for key.

Pseudo code

Algorithm 2.2: Insertion Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
for i ← 2 to N do
    k ← i − 1
    key ← a_i
    while a_k > key and k ≥ 0 do
        a_{k+1} ← a_k
        k ← k − 1
    end
    a_{k+1} ← key
end
```

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.3. Bubble Sort

Idea

Bubble sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

Pseudo code

Algorithm 2.3: Bubble Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
for i ← N to 1 do
    isSwap ← False
    for j ← 1 to i − 1 do
        if a_j > a_{j+1} then
            isSwap ← True
            swap(a_j, a_{j+1})
        end
    end
    if isSwap = False then
        stop algorithm
    end
end
```

In this paper, I implemented bubble sort with a flag *isSwap* to stop the algorithm early when the array is sorted.

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$

Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.4. Shaker Sort

Idea

Shaker sort is a bidirectional version of bubble sort. The Bubble sort algorithm always traverses elements from left and moves the largest element to its correct position in first iteration and second largest in second iteration and so on. Shaker sort orders the array in both directions. Hence every iteration of the algorithm consists of two phases. In the first one, the lightest bubble ascends to the end of the array, in the second phase the heaviest bubble descends to the beginning of the array.

Pseudo code

Algorithm 2.4: Shaker Sort

```
   Input: a_1, a_2, ..., a_N
   Output: a_1, a_2, ..., a_N (in sorted)
 1 left ← 0
 2 right ← N − 1
 3 k ← 0
 4 for i ← left to right do
       // phase 1
 5     isSwap ← False
 6     for j ← left to right − 1 do
 7         if a_j > a_{j+1} then
 8             isSwap ← True
 9             swap(a_j, a_{j+1})
10             k ← j
11         end
12     end
13     if isSwap = False then
14         stop algorithm
15     end
16     right ← k
       // phase 2
17     isSwap ← False
18     for j ← right to left + 1 do
19         if a_j < a_{j−1} then
20             isSwap ← True
21             swap(a_j, a_{j−1})
22             k ← j
23         end
24     end
25     if isSwap = False then
26         stop algorithm
27     end
28     left ← k
29 end
```

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N^2)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.5. Shell Sort

Idea

Shell sort is a generalized version of the insertion sort algorithm. It first sorts elements that are far apart from each other and successively reduces the interval between the elements to be sorted.

The interval between the elements is reduced based on the sequence used. Some of the optimal sequences that can be used in the shell sort algorithm are:

- Shell's original sequence
- Knuth's increments
- Sedgewick's increments
- Hibbard's increments
- ...

In this paper, I only implemented the algorithm with optimal sequence based on Shell's original sequence.

Pseudo code

Algorithm 2.5: Shell Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
1  interval ← N/2
2  while interval > 0 do
3      for i ← interval to N do
4          temp ← a_i
5          j ← i
6          while interval ≤ j and a_{j−interval} > temp do
7              a_j ← a_{j−interval}
8              j ← j − interval
9          end
10     end
11     a_j ← temp
12     interval ← interval/2
13 end
```

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.6. Heap Sort

Idea

Heap sort is a comparison-based sorting algorithm. Heap sort can be thought of as an improved selection sort: like selection sort, heap sort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region.

Unlike selection sort, heapsort does not waste time with a linear-time scan of the unsorted region; rather, heap sort maintains the unsorted region in a **heap data structure** to more quickly find the largest element in each step.

Pseudo code

Algorithm 2.6: Heap Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
1  Function HeapRebuild(a, pos, N):
2      while 2 · pos + 1 <= N do
3          j = 2 · pos + 1
4          if j < N then
5              if a_j < a_{j+1} then
6                  j ← j + 1
7              end
8          end
9          if a_pos ≥ a_j then
10             return
11         end
12         swap(a_pos, a_j)
13         pos ← j
14     end
15 Function HeapConstruct(a, N):
16     for i ← N/2 to 0 do
17         HeapRebuild(a, i, n)
18     end
19 Function HeapSort(a, N):
20     HeapConstruct(a, N)
21     r ← N
22     while r > 0 do
23         swap(a_1, a_N)
24         HeapRebuild(a, 1, r)
25         r ← r − 1
26     end
```

Complexity

Best case time complexity: $O(N \log N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N \log N)$
Worst case space complexity: $O(1)$

2.7. Merge Sort

Idea

Merge sort is a recursive sorting algorithm based on a "divide and conquer" approach. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

Pseudo code

Algorithm 2.7: Merge Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
1  Function Merge(a, first, mid, last):
2      n_1 ← mid − first + 1
3      n_2 ← last − mid
4      L ← a_first, a_first+1, ..., a_mid
5      R ← a_mid+1, a_mid+2, ..., a_last
       // merge
6      i ← 0
7      j ← 0
8      k ← first
9      while i < n_1 and j < n_2 do
10         if L_i < R_j then
11             a_k ← L_i
12             i ← i + 1
13         else
14             a_k ← R_j
15             j ← j + 1
16         end
17         k ← k + 1
18     end
19     while j < n_2 do
20         a_k ← R_j
21         k ← k + 1
22         j ← j + 1
23     end
24     while i < n_1 do
25         a_k ← L_i
26         k ← k + 1
27         i ← i + 1
28     end
29 Function MergeSort(a, first, last):
30     if first < last then
31         mid ← first + (last − first)/2
32         MergeSort(a, first, mid)
33         MergeSort(a, mid + 1, last)
34         Merge(a, first, mid, last)
35     end
```

Complexity

Best case time complexity: $O(N \log N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N \log N)$
Worst case space complexity: $O(N)$

2.8. Quick Sort

Idea

Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Pick first element as pivot.
- Pick last element as pivot
- Pick a random element as pivot.
- Pick median as pivot.

In this paper, I implemented the algorithm with pivot is a median of array.

Pseudo code

Algorithm 2.8: Quick Sort

Input: $a_1, a_2, ..., a_N$
Output: $a_1, a_2, ..., a_N$ (in sorted)

```
1  Function Partition(a, l, r):
2      pivot ← a_{(l+r)/2}
3      while l ≤ r do
4          while a_l < pivot do
5              l ← l + 1
6          end
7          while a_r > pivot do
8              r ← r − 1
9          end
10         if l ≤ r then
11             swap(a_l, a_r)
12             l ← l + 1
13             r ← r − 1
14         end
15     end
16     return l
17 Function QuickSort(a, l, r):
18     if l < r then
19         mid ← Partition(a, l, r)
20         QuickSort(a, l, mid − 1)
21         QuickSort(a, mid, r)
22     end
```

Complexity

Best case time complexity: $O(N)$
Average case time complexity: $O(N \log N)$
Worst case time complexity: $O(N^2)$
Worst case space complexity: $O(1)$

2.9. Counting Sort

2.10. Radix Sort

2.11. Flash Sort