Fast sorting algorithms

Raimondas Čiegis

Matematinio modeliavimo katedra, e-paštas: rc@vgtu.lt

November 1, 2023

In this lecture we consider three fast sorting algorithms. The complexity of them is close to the optimal estimate $O(N \log N)$.

Quicksort algorithm

Quicksort is an efficient, general-purpose sorting algorithm. It is still a very popular and commonly used in different applications algorithm.

We will show that its average complexity is $\mathcal{O}(N \log N)$, and Quicksort can be done in-place, requiring only small additional amounts of memory to perform the sorting.

Quicksort is a divide-and-conquer type algorithm.

A partition produces a division into two consecutive non empty sub-sets, in such a way that no element of the first sub-set is greater than any element of the second sub-set.

Quicksort is a divide-and-conquer type algorithm.

A partition produces a division into two consecutive non empty sub-sets, in such a way that no element of the first sub-set is greater than any element of the second sub-set.

After applying this partition, Quicksort then recursively sorts the sub-sets.

Partitioning step.

We partition a set A into two sub-sets.

Partitioning step.

We partition a set A into two sub-sets.

First, a key element a_j is selected. It is called a division point or pivot.

Partitioning step.

We partition a set A into two sub-sets.

First, a key element a_j is selected. It is called a division point or pivot.

Next we reorder elements of A so that all elements with values less than the pivot come before the division point,

while all elements with values greater than the pivot come after it.

Partitioning step.

We partition a set A into two sub-sets.

First, a key element a_j is selected. It is called a division point or pivot.

Next we reorder elements of A so that all elements with values less than the pivot come before the division point,

while all elements with values greater than the pivot come after it.

Elements that are equal to the pivot can go either way.

Sorting of sub-sets

If the sub-set has fewer than two elements, return.

Otherwise, apply Quicksort to this sub-set (recursion).

Sorting of sub-sets

If the sub-set has fewer than two elements, return.

Otherwise, apply Quicksort to this sub-set (recursion).

A popular modification selects a small number M.

If the sub-set has fewer than M elements, sort it by some simple sorting algorithm, e.g. Insert sort.

Determination of the solution

Since no element of the first sub-set is greater than any element of the second sub-set, thus by sorting sub-sets we finish sorting all elements of A.

No computations are done at this stage.

Quicksort algorithm

```
QuickSort (l, r)
begin
  (1)  if  ( l < (r - M) )  then
  (2)       Partition ( l, r, m );
  (3)       QuickSort ( l, m-1 );
  (4)       QuickSort ( m+1, r );
       else
  (5)       if ( l < r ) SelectionSort (l, r);
       end if
end QuickSort
```

Partition (l, r, m)

begin

(1) $v = a_l$;

(2) i = l; j = r;

(3) **while** ($i < j$) **do**

(4) **while** (($a_j \geqslant v$) && ($i < j$)) $j = j - 1$;

(5) **if** ($i \neq j$) **then**

(6) $a_i = a_j$; i++;

end if

(7) **while** (($a_i \leqslant v$) && ($i < j$)) $i = i + 1$;

(8) **if** ($i \neq j$) **then**

(9) $a_j = a_i$; j−−;

end if

end do

(10) $a_i = v$; m = i;

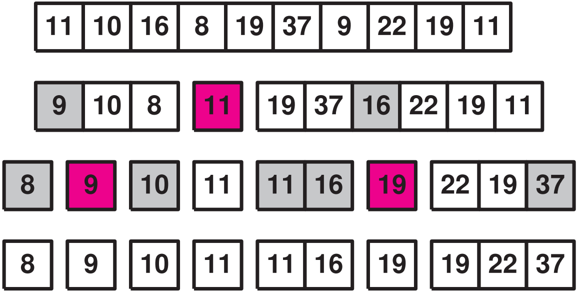
end Partition

Let's sort a list

$$A = (11, 10, 16, 8, 19, 37, 9, 22, 19, 11).$$

Let's sort a list

$$A = (11, 10, 16, 8, 19, 37, 9, 22, 19, 11).$$



The first element of any sub-set is used as a pivot.
Pivots are colored red, grey colored elements are swaped during partition steps.

Complexity of Quicksort algorithm

We are interested to find a number of comparisons L_N required to sort a given set of N elements.

Complexity of Quicksort algorithm

We are interested to find a number of comparisons L_N required to sort a given set of N elements.

During a partition step each element is compared with a pivot.

Thus a total number of comparisons depends only on sizes of produced sub-sets.

Let's consider the worst case, when the smallest element is selected as a pivot.

Then we get the following equation

$$L_B(N) = L_B(N-1) + N - 1 \, .$$

If a set contains only one element then it is already sorted:

$$L(1) = 0 \, .$$

Let's consider the worst case, when the smallest element is selected as a pivot.

Then we get the following equation

$$L_B(N) = L_B(N-1) + N - 1\,.$$

If a set contains only one element then it is already sorted:

$$L(1) = 0\,.$$

By applying this relation $(N-1)$ times, we get

$$L_B(N) = \sum_{i=2}^{N} (i-1) = \sum_{j=1}^{N-1} j = \frac{N^2 - N}{2}\,.$$

Let's consider the worst case, when the smallest element is selected as a pivot.

Then we get the following equation

$$L_B(N) = L_B(N-1) + N - 1\,.$$

If a set contains only one element then it is already sorted:

$$L(1) = 0\,.$$

By applying this relation $(N-1)$ times, we get

$$L_B(N) = \sum_{i=2}^{N}(i-1) = \sum_{j=1}^{N-1} j = \frac{N^2 - N}{2}\,.$$

Thus in the worst case this algorithm is not faster than Insert sort or Select sort algorithms.

The most un-expected conclusion is that such a result follows for already sorted sets (when the first element is selected as a pivot).

Let's consider the best case, when at each partition step we select the pivot element which divides a set into two sub-sets of equal sizes.

Let's consider the best case, when at each partition step we select the pivot element which divides a set into two sub-sets of equal sizes.

Take $N = (2^m - 1)$. Then the number of comparisons satisfy the relation:

$$L_G(2^m - 1) = \begin{cases} 2L_G(2^{m-1} - 1) + 2^m - 2, & \text{when} \quad m > 1, \\ 0, & \text{when} \quad m = 1\,. \end{cases}$$

Applying it $(m-2)$ times we get

$$\begin{aligned} L_G(N) &= 2^m - 2 + 2\cdot(2^{m-1}-2) + 2^2\cdot(2^{m-2}-2) + \dots \\ &\quad + 2^{m-2}\cdot(2^2-2) \\ &= (m-1)2^m + 2^m - 2 \\ &= (N+1)\log(N+1) - 2. \end{aligned}$$

Applying it $(m-2)$ times we get

$$\begin{aligned} L_G(N) &= 2^m - 2 + 2 \cdot (2^{m-1} - 2) + 2^2 \cdot (2^{m-2} - 2) + \dots \\ &\quad + 2^{m-2} \cdot (2^2 - 2) \\ &= (m-1)2^m + 2^m - 2 \\ &= (N+1)\log(N+1) - 2. \end{aligned}$$

We note that for the Insert sort algorithm the complexity of the best case is even better N.

Applying it $(m-2)$ times we get

$$
\begin{aligned}
L_G(N) &= 2^m - 2 + 2 \cdot (2^{m-1} - 2) + 2^2 \cdot (2^{m-2} - 2) + \dots \\
&+ 2^{m-2} \cdot (2^2 - 2) \\
&= (m-1)2^m + 2^m - 2 \\
&= (N+1)\log(N+1) - 2.
\end{aligned}
$$

We note that for the Insert sort algorithm the complexity of the best case is even better N.

But only for the best case.

Quicksort algorithm is so popular since in the average case its complexity is also very close to the best case

$$L_V(N) = 1,386 N \log N + \mathcal{O}(N) \, .$$

Quicksort algorithm is so popular since in the average case its complexity is also very close to the best case

$$L_V(N) = 1,386 N \log N + \mathcal{O}(N) \, .$$

Sorting is done in-place.

It was explained above that a complexity of Quicksort algorithm is only $\mathcal{O}(N^2)$, when

a set of elements is almost sorted,

and the first element of a sub-set is selected as a pivot.

It was explained above that a complexity of Quicksort algorithm is only $\mathcal{O}(N^2)$, when

a set of elements is almost sorted,

and the first element of a sub-set is selected as a pivot.

Thus the following two modfications of the base algorithm are recommended:

It was explained above that a complexity of Quicksort algorithm is only $\mathcal{O}(N^2)$, when

a set of elements is almost sorted,

and the first element of a sub-set is selected as a pivot.

Thus the following two modfications of the base algorithm are recommended:

1. At each recursion stage three elements of A are selected in random a_k, a_l and a_m and they are sorted.

Then a mid element is taken as a pivot.

It was explained above that a complexity of Quicksort algorithm is only $\mathcal{O}(N^2)$, when

a set of elements is almost sorted,

and the first element of a sub-set is selected as a pivot.

Thus the following two modfications of the base algorithm are recommended:

1. At each recursion stage three elements of A are selected in random a_k, a_l and a_m and they are sorted.

Then a mid element is taken as a pivot.

2. Before starting the Quicksort algorithm we swap all elements of A in random.

There is a big probability that sorting costs of such perturbed set will be close to the average complexity of Quicksort.

Median of an unsorted array

Median of a sorted array of size N is defined as the middle element.

Median of an unsorted array

Median of a sorted array of size N is defined as the middle element.

The following tasks are solved in many applications:

- Given an unsorted array of elements A of length N, the task is to find the median of this array;

Median of an unsorted array

Median of a sorted array of size N is defined as the middle element.

The following tasks are solved in many applications:

- Given an unsorted array of elements A of length N, the task is to find the median of this array;
- Find k-th element according to a sorted order.

Median of an unsorted array

Median of a sorted array of size N is defined as the middle element.

The following tasks are solved in many applications:

- Given an unsorted array of elements A of length N, the task is to find the median of this array;
- Find k-th element according to a sorted order.

Median of an unsorted array

Median of a sorted array of size N is defined as the middle element.

The following tasks are solved in many applications:

- Given an unsorted array of elements A of length N, the task is to find the median of this array;
- Find k-th element according to a sorted order.

In fact the task to find the median is a particular case of a more general second task

$$k = N/2.$$

Both tasks can be solved easily when we have a sorted list of elements.

Both tasks can be solved easily when we have a sorted list of elements.

Fast sorting algorithms exist. Thus both new tasks can be solved in $O(N \log N)$ operations.

Both tasks can be solved easily when we have a sorted list of elements.

Fast sorting algorithms exist. Thus both new tasks can be solved in $O(N \log N)$ operations.

The main challenge is to solve these tasks faster.

Both tasks can be solved easily when we have a sorted list of elements.

Fast sorting algorithms exist. Thus both new tasks can be solved in $O(N \log N)$ operations.

The main challenge is to solve these tasks faster.

Now we will construct a fast algorithm by using the same divide-and-conquer method.

It is sufficient to modify a partition part of Quicksort algorithm.

Quick search algorithm

```
int QuickFind (l, r, k)    #  l ≤ k ≤ r
begin
 (1)  if  ( l == r )  then
 (2)      return (l);
      else
 (3)      Partition (l, r, m);
 (4)      if  ( m > k )  then
 (5)          QuickFind ( l, m-1, k );
          else
 (6)          if  ( m == k )  then
 (7)              return (m);
              else
 (8)              QuickFind ( m + 1, r, k );
      end if
end QuickFind
```

This implementation of the algorithm is based on recursion.

Still at each stage only one recursion function is activated.

This implementation of the algorithm is based on recursion.

Still at each stage only one recursion function is activated.

It is recommended to present also an iterative version of this algorithm.

We restrict to the complexity analysis of the best case.

After each step of the partition algorithm the size of sub-sets is reduced twice, thus we get the following equation

$$L_G(N) = N + \frac{N}{2} + \frac{N}{4} + \ldots + 2 + 1 = 2N + \mathcal{O}(1)\,.$$

We restrict to the complexity analysis of the best case.

After each step of the partition algorithm the size of sub-sets is reduced twice, thus we get the following equation

$$L_G(N) = N + \frac{N}{2} + \frac{N}{4} + \ldots + 2 + 1 = 2N + \mathcal{O}(1)\,.$$

Thus the median is computed $\frac{1}{2} \log N$ times faster than by using the Quicksort algorithm.

Merge sort

Now we consider one more fast sorting algorithm. Even for the worst case it has a complexity $\mathcal{O}(N \log N)$.

Merge sort

Now we consider one more fast sorting algorithm. Even for the worst case it has a complexity $\mathcal{O}(N \log N)$.

This algorithm is based on a well known fact that it is possible to merge two already sorted sets very efficiently.

Therefore this algorithm is called Merge sort.

Merge sort

Now we consider one more fast sorting algorithm. Even for the worst case it has a complexity $\mathcal{O}(N \log N)$.

This algorithm is based on a well known fact that it is possible to merge two already sorted sets very efficiently.

Therefore this algorithm is called Merge sort.

It is interesting to note that divide-and-conquer method is again used to construct Merge sort.

Let's consider how three main steps of the divide-and-conquer method are implemented for this algorithm.

Let's consider how three main steps of the divide-and-conquer method are implemented for this algorithm.

Partitioning step.

The full set of elements A are divided into two sub-sets.

The first $N/2$ elements are saved in the left sub-set, the remaining elements belong to the right sub-set.

Let's consider how three main steps of the divide-and-conquer method are implemented for this algorithm.

Partitioning step.

The full set of elements A are divided into two sub-sets.

The first $N/2$ elements are saved in the left sub-set, the remaining elements belong to the right sub-set.

No computations are done at this step!

Sorting of sub-sets

If no more than one element belong to this sub-set, then this task is solved.

Otherwise a sub-set is sorted by using Merge sort algorithm.

Sorting of sub-sets

If no more than one element belong to this sub-set, then this task is solved.

Otherwise a sub-set is sorted by using Merge sort algorithm.

Thus once again the recursion method is used.

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Both input sets are already sorted.

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Both input sets are already sorted.

We compare the first alements of both sets and a smaller element is stored in a sorted list.

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Both input sets are already sorted.

We compare the first alements of both sets and a smaller element is stored in a sorted list.

Next this algorithm is repeated for all elements of sub-sets.

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Both input sets are already sorted.

We compare the first alements of both sets and a smaller element is stored in a sorted list.

Next this algorithm is repeated for all elements of sub-sets.

These comparisons are continued till one set becomes empty. The remaining elements are moved to the sorted list in-order (since both sub-sets were already sorted).

Merge of two sorted sets

Merge sort's most common implementation does not sort in place; therefore, the memory size of the input must be allocated for the sorted output to be stored in.

Both input sets are already sorted.

We compare the first alements of both sets and a smaller element is stored in a sorted list.

Next this algorithm is repeated for all elements of sub-sets.

These comparisons are continued till one set becomes empty. The remaining elements are moved to the sorted list in-order (since both sub-sets were already sorted).

It is important to note that the Merge sort is a stable sort algorithm.

Let's apply the Merge sort for the following set of elements

$$A = (11,\ 10,\ 16,\ 8,\ 19,\ 37,\ 9,\ 22,\ 19,\ 11).$$

Let's apply the Merge sort for the following set of elements

$$A = (11,\ 10,\ 16,\ 8,\ 19,\ 37,\ 9,\ 22,\ 19,\ 11).$$

11	10	16	8	19	37	9	22	19	11

11	10	16	8	19

37	9	22	19	11

11	10	16

8	19

37	9	22

19	11

10	11	16

8	19

9	22	37

11	19

8	10	11	16	19

9	11	19	22	37

8	9	10	11	11	16	19	19	22	37

Selection sort is used to sort grey color elements.

Complexity analysis of Merge sort

Since all comparisons are done at merge stage, thus it is sufficient to analyse only this part of the algorithm.

Complexity analysis of Merge sort

Since all comparisons are done at merge stage, thus it is sufficient to analyse only this part of the algorithm.

Let us assume that merging two sorted sub-sets of length N_1 and N_2 we compare $c(N_1 + N_2)$ elements, where $c \leq 1$.

Complexity analysis of Merge sort

Since all comparisons are done at merge stage, thus it is sufficient to analyse only this part of the algorithm.

Let us assume that merging two sorted sub-sets of length N_1 and N_2 we compare $c(N_1 + N_2)$ elements, where $c \leq 1$.

For simplicity of analysis we take $N = 2^m$. Then we get the following equation

$$L(N) = \begin{cases} 2\,L\big(\frac{1}{2}N\big) + cN, & \text{if } \ N > 2, \\ 1, & \text{if } \ N = 2\,. \end{cases}$$

By applying this equation $(m-1)$ time, we get the total number of comparisons

$$\begin{aligned} L(N) &= 2L\left(\frac{1}{2}N\right) + cN \\ &= 4L\left(\frac{1}{4}N\right) + 2cN \\ &= \cdots = \frac{N}{2}L(2) + (m-1)cN \\ &= cN\log N + (0.5-c)N\,. \end{aligned}$$

Merge sort's best case takes about half as many comparisons as its worst case.

Merge sort's best case takes about half as many comparisons as its worst case.

For large N and a randomly ordered input list, Merge sort's expected (average) number of comparisons of one step approaches αN fewer than the worst case, where $\alpha = 0.2645$.

Merge sort's best case takes about half as many comparisons as its worst case.

For large N and a randomly ordered input list, Merge sort's expected (average) number of comparisons of one step approaches αN fewer than the worst case, where $\alpha = 0.2645$.

In the worst case, Merge sort uses approximately 39 percents fewer comparisons than Quicksort does in its average case.

Conclusions

QuickSort is choosen as a basic sorting algorithm for many Big Data applications when a very large amount of data should be sorted and analysed.

Conclusions

QuickSort is choosen as a basic sorting algorithm for many Big Data applications when a very large amount of data should be sorted and analysed.

We note that this selection is done by so many users despite the known theoretical result that in the worst case QuickSort is far from optimal.

Conclusions

QuickSort is choosen as a basic sorting algorithm for many Big Data applications when a very large amount of data should be sorted and analysed.

We note that this selection is done by so many users despite the known theoretical result that in the worst case QuickSort is far from optimal.

Next we formulate the main reasons why it is recommended to select QuickSort algorithm.

Conclusions

QuickSort is choosen as a basic sorting algorithm for many Big Data applications when a very large amount of data should be sorted and analysed.

We note that this selection is done by so many users despite the known theoretical result that in the worst case QuickSort is far from optimal.

Next we formulate the main reasons why it is recommended to select QuickSort algorithm.

1. QuickSort implementations are faster than Merge sort.

Conclusions

QuickSort is choosen as a basic sorting algorithm for many Big Data applications when a very large amount of data should be sorted and analysed.

We note that this selection is done by so many users despite the known theoretical result that in the worst case QuickSort is far from optimal.

Next we formulate the main reasons why it is recommended to select QuickSort algorithm.

1. QuickSort implementations are faster than Merge sort.
2. QuickSort works in-place.

Heap sort algorithm

Let's recall main properties of heap data data structure.

Two important conditions are satified for such binary trees:

Heap sort algorithm

Let's recall main properties of heap data data structure.

Two important conditions are satified for such binary trees:

1. The value of each vertex is larger or equal to values of its children.

2. The binary tree is balanced, each new level is filled one by one and from left to right.

Heap sort algorithm

Let's recall main properties of heap data data structure.

Two important conditions are satified for such binary trees:

1. The value of each vertex is larger or equal to values of its children.

2. The binary tree is balanced, each new level is filled one by one and from left to right.

3. The largest element is stored in the root.

Heap sort algorithm

Let's recall main properties of heap data data structure.

Two important conditions are satified for such binary trees:

1. The value of each vertex is larger or equal to values of its children.

2. The binary tree is balanced, each new level is filled one by one and from left to right.

3. The largest element is stored in the root.

4. The complexity of heap construction algorithm is $\Theta(N)$.

Heap sort algorithm is based on the recursion method.
Thus it is sufficient to consider one step.

Heap sort algorithm is based on the recursion method.
Thus it is sufficient to consider one step.

1. Elements a_1 and a_N are swapped.

Now the largest element is placed into the correct position

Heap sort algorithm is based on the recursion method.
Thus it is sufficient to consider one step.

1. Elements a_1 and a_N are swapped.

Now the largest element is placed into the correct position

2. The size of the heap P is reduced

$$\mathrm{size(P)} = \mathrm{size(P)} - 1.$$

Heap sort algorithm is based on the recursion method.
Thus it is sufficient to consider one step.

1. Elements a_1 and a_N are swapped.

Now the largest element is placed into the correct position

2. The size of the heap P is reduced

$$\text{size(P)} = \text{size(P)} - 1.$$

3. The properties of the heap structure are restored if they where violated by the swap operation.

In order to make these modifications we use
HeapDownOrder (1, size(P)).

Heap sort algorithm

HeapSort ()
begin
(1) MakeHeap ();
(2) **for** (i=N; i > 1; i = i-1) **do**
(3) swap (a_1, a_i);
(4) HeapDownOrder (1, i-1);
end do
end HeapSort

10	37	18	13	22	14	25	8	12	28

a)

37	**28**	**25**	**13**	**22**	**14**	**18**	**8**	**12**	**10**

b)

10	28	25	13	22	14	18	8	12	**37**

c)

28	22	25	13	10	14	18	8	12	**37**

d)

12	22	25	13	10	14	18	8	**28**	**37**

e)

25	22	18	13	10	14	12	8	**28**	**37**

f)

8	22	18	13	10	14	12	**25**	**28**	**37**

g)

22	13	18	8	10	14	12	**25**	**28**	**37**

h)

12	13	18	8	10	14	**22**	**25**	**28**	**37**

i)

18	13	14	8	10	12	**22**	**25**	**28**	**37**

j)