Sorting algorithms

Raimondas Čiegis

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

October 30 d., 2023

Main Problems

As it was noted in previous lectures, efficient sorting is important for optimizing the efficiency of other algorithms such as search.

Main Problems

As it was noted in previous lectures, efficient sorting is important for optimizing the efficiency of other algorithms such as search.

We intend to study most popular sorting algorithms and analyse their complexity (mainly average and worst cases).

Main Problems

As it was noted in previous lectures, efficient sorting is important for optimizing the efficiency of other algorithms such as search.

We intend to study most popular sorting algorithms and analyse their complexity (mainly average and worst cases).

There are quite many sorting algorithms and in applications it is important to select the one best fitted for the specific problem.

Problem formulation

Let us consider a set of elements (data) $A = (a_1, a_2, \dots, a_N)$.

Problem formulation

Let us consider a set of elements (data) $A = (a_1, a_2, \dots, a_N)$.

We assume that it is possible to compare elements, i.e. to check if the following estimate is valid

$$a_i < a_j, \quad \text{if} \quad i \neq j.$$

Problem formulation

Let us consider a set of elements (data) $A = (a_1, a_2, \ldots, a_N)$.

We assume that it is possible to compare elements, i.e. to check if the following estimate is valid

$$a_i < a_j, \quad \text{if} \quad i \neq j.$$

The output $A^{'} = (a_1^{'}, a_2^{'}, \ldots, a_N^{'})$ of any sorting algorithm must satisfy two conditions:

1. The output is in monotonic order (each element is no smaller/larger than the previous element, according to the required order)

$$a_i^{'} \leq a_{i+1}^{'}, \quad \text{if} \quad i = 0, 1, \ldots, N-1.$$

2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

Complexity of comparison sorting algorithms

In this lecture we consider only comparison based algorithms, when it is possible

Complexity of comparison sorting algorithms

In this lecture we consider only comparison based algorithms, when it is possible

- to compare two elements of A ,
- to swap (permute) two elements.

Complexity of comparison sorting algorithms

In this lecture we consider only comparison based algorithms, when it is possible

- to compare two elements of A ,
- to swap (permute) two elements.

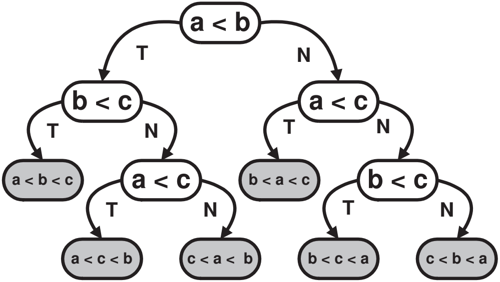
Such algorithms can be presented as binary tree data structure.

Each vertex denotes a comparison operation of two elements, and its leaves define a result of this comparison.

Let us consider a simple example, when we need to sort three elements $A = (a, b, c)$.

Let us consider a simple example, when we need to sort three elements $A = (a, b, c)$.

The sorting process and all possible sorted sets are presented in the figure.



T denotes an edge, when the condition is "true" and N otherwise.

It follows from the presented figure that three comparisons are required in the worst case.

It follows from the presented figure that three comparisons are required in the worst case.

Thus such a number of comparisons is necessary for any sorting algorithm when three elements are sorted.

It follows from the presented figure that three comparisons are required in the worst case.

Thus such a number of comparisons is necessary for any sorting algorithm when three elements are sorted.

A more important conclusion is that the number of leaves of binary tree should be not smaller than a total number of possible permutations.

A full binary tree of height h has 2^h leaves.

For N elements it is possible to construct $N!$ different permutations, thus for any sorting algorithm a number of comparisons K must be not smaller than a solution of the inequality

$$2^K \geq N!$$

A full binary tree of height h has 2^h leaves.

For N elements it is possible to construct $N!$ different permutations, thus for any sorting algorithm a number of comparisons K must be not smaller than a solution of the inequality

$$2^K \geq N!$$

In the case of three elements we get $K = 3$ since

$$2^3 \geq 3!$$

For a general number of elements N we can use Stirling's approximation that

$$K \geq N \log N - N \log e \, .$$

For a general number of elements N we can use Stirling's approximation that

$$K \geq N \log N - N \log e \,.$$

Thus for any comparison based sorting algorithm the worst case complexity is estimated as

$$W_b \geq N \log N.$$

We already constructed one efficient sorting algorithm.

We already constructed one efficient sorting algorithm.

Let us assume that elements of A are stored in a binary searching tree.

Then we print elements by using InOrder algorithm and get a sorted set.

We already constructed one efficient sorting algorithm.

Let us assume that elements of A are stored in a binary searching tree.

Then we print elements by using InOrder algorithm and get a sorted set.

InOrder is a recursive algorithm and it requires to make $O(N)$ operations.

We already constructed one efficient sorting algorithm.

Let us assume that elements of A are stored in a binary searching tree.

Then we print elements by using InOrder algorithm and get a sorted set.

InOrder is a recursive algorithm and it requires to make O(N) operations.

This result is very unexpected, since the obtained complexity estimate O(N) is better than the lower estimate which was proved above for ANY sorting algorithm.

Where a mistake was done?

Where a mistake was done?

In fact no mistakes were done in our analysis.

We must add costs of constructing an initial balanced search tree. As it was shown in previous lectures the costs of best algorithms are equal to $O(N \log N)$.

Next we consider two simple sorting algorithms. They are very useful when a small number of elements is sorted.

Next we consider two simple sorting algorithms. They are very useful when a small number of elements is sorted.

Note, that the complexity analysis of these algorithms is also simple and it helps us to formulate main steps of such a task very clearly.

Selection sort

Selection sort is an in-place comparison sorting algorithm.

This algorithm is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations.

Selection sort

Selection sort is an in-place comparison sorting algorithm.

This algorithm is noted for its simplicity and has performance advantages over more complicated algorithms in certain situations.

The Selection sort divides the input list into two parts:

a sorted sublist of items which is built up from left to right at the front (left) of the list

and a sublist of the remaining unsorted items that occupy the rest of the list.

Initially, the sorted sublist is empty and the unsorted sublist is the entire input list, thus the boundary point of both sublists $i = 0$.

Initially, the sorted sublist is empty and the unsorted sublist is the entire input list, thus the boundary point of both sublists $i = 0$.

The algorithm proceeds by finding the smallest (or largest) element in the unsorted sublist $i, i + 1, \ldots, N$,

exchange (swap) it with the leftmost unsorted element (putting it in sorted order), if required,

and move the sublist boundaries one element to the right, $i = i + 1$.

Selection sort algorithm

```
SelectionSort ()
begin
  (1)  for ( i = 1;  i < N ;  i++ ) do
  (2)      k = i;
  (3)      for ( j = i+1;  j <= N;  j++ ) do
  (4)          if ( a_j < a_k )  k = j;
           end do
  (5)      if ( k > i )  swap ( a_i, a_k );
       end do
end SelectionSort
```

101	17	33	2	24

a)

2	17	33	101	24

b) $i = 1, \ k = 4$

2	**17**	33	101	24

c) $i = 2, \ k = 2$

2	**17**	**24**	101	33

d) $i = 3, \ k = 5$

2	**17**	**24**	**33**	**101**

e) $i = 4, \ k = 5$

Sorting of integer numbers: i is the iteration number , k is the index of the smallest number in an unsorted sublist, items a_i and a_k are swapped, if required

Complexity of the selection sort algorithm

Two main operations are important in any sorting algorithm:

- comparison of two items a_i and a_j,
- swapping of a_i and a_j.

Complexity of the selection sort algorithm

Two main operations are important in any sorting algorithm:

comparison of two items a_i and a_j,

swapping of a_i and a_j.

Let $L(N)$ denotes a total number of comparisons and $S(N)$ a number of swappings when a set A of the size N is sorted.

In the selection sort algorithm all comparisons are done at each iteration and for any distribution of elements, thus

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

In the selection sort algorithm all comparisons are done at each iteration and for any distribution of elements, thus

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

This estimate is much worse than the bound $N \log N$.

The number of swappings depends on the initial distribution of items.

The number of swappings depends on the initial distribution of items.

In the best case when all elements are already sorted no need to make any swapping, i.e. $S_G(N) = 0$.

The number of swappings depends on the initial distribution of items.

In the best case when all elements are already sorted no need to make any swapping, i.e. $S_G(N) = 0$.

In the worst case such swappings are done at each iteration thus $S_B(N) = N - 1$.

The number of swappings depends on the initial distribution of items.

In the best case when all elements are already sorted no need to make any swapping, i.e. $S_G(N) = 0$.

In the worst case such swappings are done at each iteration thus $S_B(N) = N - 1$.

Therefore the selection sort algorithm is recommended when a number of elements of A is not big, but the value field of each element is large.

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

Insertion sort provides several advantages:

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

Insertion sort provides several advantages:

More efficient in practice than most other simple quadratic algorithms such as selection sort;

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

Insertion sort provides several advantages:

More efficient in practice than most other simple quadratic algorithms such as selection sort;

Efficient for data sets that are already substantially sorted: the time complexity is $O(kN)$, when each element in the input is no more than k places away from its sorted position;

Insertion sort algorithm

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time by comparisons.

Insertion sort provides several advantages:

More efficient in practice than most other simple quadratic algorithms such as selection sort;

Efficient for data sets that are already substantially sorted: the time complexity is $O(kN)$, when each element in the input is no more than k places away from its sorted position;

It is stable, i.e. don't change the relative order of elements with equal keys.

Insertion sort iterates, taking one input element each repetition, and grows a sorted output list.

Insertion sort iterates, taking one input element each repetition, and grows a sorted output list.

At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.

Insertion sort iterates, taking one input element each repetition, and grows a sorted output list.

At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there.

Iterations are repeated until no input elements remain.

The sorting is done in-place.

The sorting is done in-place.

At i-th iteration we take element a_i and insert it among already sorted $(i - 1)$ elements.

The sorting is done in-place.

At i-th iteration we take element a_i and insert it among already sorted $(i-1)$ elements.

The comparison is started from $(i-1)$-th element.

If $a_i < a_{i-1}$, then these elements are swapped and a next element a_{i-2} is tested.

The sorting is done in-place.

At i-th iteration we take element a_i and insert it among already sorted $(i-1)$ elements.

The comparison is started from $(i-1)$-th element.

If $a_i < a_{i-1}$, then these elements are swapped and a next element a_{i-2} is tested.

This process is continued till the correct place is defined.

Insertion sort algorithm

InsertionSort ()
begin
(1) **for** (i = 2; i <= N; i++) **do**
(2) v = a_i; a_0 =v; j = i;
(3) **while** ($v < a_{j-1}$) **do**
(4) $a_j = a_{j-1}$;
(5) j = j-1;
end do
(6) **if** (i $\neq$ j) $a_j = v$;
end do
end InsertionSort

Insertion sort algorithm

InsertionSort ()
begin
(1) **for** (i = 2; i <= N; i++) **do**
(2) v = a_i; a_0 =v; j = i;
(3) **while** ($v < a_{j-1}$) **do**
(4) $a_j = a_{j-1}$;
(5) j = j-1;
end do
(6) **if** (i $\neq$ j) $a_j = v$;
end do
end InsertionSort

A barrier technique is applied when a dummy element $a_0 = a_i$ is inserted before starting iterations (3).
This trick guarantee that iterations will end successfully without cheking if the first element is already reached.

Let us sort a list of integer numbers

$$A = (101,\ 17,\ 33,\ 2,\ 24).$$

Let us sort a list of integer numbers

$$A = (101,\ 17,\ 33,\ 2,\ 24).$$

101	17	33	2	24

a)

17	**101**	33	2	24

b)

17	**33**	**101**	2	24

c)

2	**17**	**33**	**101**	24

d)

2	**17**	**24**	**33**	**101**

e)

A grey color is used to denote the element which was inserted at a given iteration.

Complexity of the insertion sort algorithm

At each iteration (1) a number of swappings is one less than the number of comparisons

$$L(N) = S(N) + N - 1.$$

Complexity of the insertion sort algorithm

At each iteration (1) a number of swappings is one less than the number of comparisons

$$L(N) = S(N) + N - 1.$$

If elements of A initially are distributed in an oposite order, then in step (1) all comparisons are done.

Thus in the *worst* case we get the complexity estimate

$$L_B(N) = \sum_{i=2}^{N} i = \frac{N^2 + N - 2}{2} = \frac{N^2}{2} + \mathcal{O}(N)\,.$$

Average case complexity depends on an initial distribution of data.

Average case complexity depends on an initial distribution of data.

We assume that during iteration (1) a new element can be inserted into any place of the already sorted list with the same probability.

Average case complexity depends on an initial distribution of data.

We assume that during iteration (1) a new element can be inserted into any place of the already sorted list with the same probability.

Then it is easy to compute that

$$L_V(N) = \frac{N^2}{4} + \mathcal{O}(N)\,.$$

Average case complexity depends on an initial distribution of data.

We assume that during iteration (1) a new element can be inserted into any place of the already sorted list with the same probability.

Then it is easy to compute that

$$L_V(N) = \frac{N^2}{4} + \mathcal{O}(N)\,.$$

This estimate is only twice better than the worst case complexity.

Let us consider an interesting modification of this algorithm when comparison costs are much larger than swapping costs.

Let us consider an interesting modification of this algorithm when comparison costs are much larger than swapping costs.

Then we can apply the divide-and-conquere method in order to find the insertion place.

Let us assume that we want to insert element a_i.

First we compare a_i with the $a_{i/2}$.

If

$$a_{i/2} \leq a_i,$$

then we repeat this process in the interval $[i/2+1, i]$,

otherwise

we test the interval $[1, i/2-1]$.

Let us assume that we want to insert element a_i.

First we compare a_i with the $a_{i/2}$.

If

$$a_{i/2} \le a_i,$$

then we repeat this process in the interval $[i/2+1, i]$,

otherwise

we test the interval $[1, i/2-1]$.

Assume that a_i must be inserted into j-th place.

Then we move elements a_j, ..., a_{i-1} to positions a_{j+1}, ..., a_i and insert the old a_i to the new position a_j.

Let us assume that we want to insert element a_i.

First we compare a_i with the $a_{i/2}$.

If

$$a_{i/2} \leq a_i,$$

then we repeat this process in the interval $[i/2+1, i]$,

otherwise

we test the interval $[1, i/2-1]$.

Assume that a_i must be inserted into j-th place.

Then we move elements a_j, ..., a_{i-1} to positions a_{j+1}, ..., a_i and insert the old a_i to the new position a_j.

The total number of comparisons for this sorting algorithm is optimal

$$L_N = \mathcal{O}(N \log N).$$