

Algorithms and Data Structures

Sorting 1

Marcin Sydow

Topics covered by this lecture:

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

- The Problem of Sorting and its importance
- Selection-sort
- Insertion-sort
- Merge-sort
- Linked Lists

Sorting

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Input: S - sequence of elements that can be ordered (according to some binary total-order relation $R \subseteq S \times S$); len - the length of sequence (natural number)

Output: S' - non-decreasingly sorted sequence consisting of elements of multi-set of the input sequence S (e.g.

$$\forall_{0 < i < len} (S[i-1], S[i]) \in R)$$

In this course, for simplicity, we assume sorting natural numbers, but all the discussed algorithms which use comparisons can be easily adapted to sort any other ordered universe.

The Importance of Sorting

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Sorting is one of the most important and basic operations in any real-life data processing in computer science. For this reason it was very intensively studied since the half of the 20th century, and currently is regarded as a well studied problem in computer science.

Examples of very important applications of sorting:

- acceleration of searching
- acceleration of operations on relations “by key”, etc. (e.g. in databases)
- data visualisation
- computing many important statistical characteristics

And many others.

Selection Sort

The idea is simple. Identify the minimum (*len* times) excluding it from the further processing and putting on the next position in the output sequence:

```
selectionSort(S, len){
    i = 0
    while(i < len){
        mini = indexOfMin(S, i, len)
        swap(S, i, mini)
        i++
    }
}
```

where:

`indexOfMin(S, i, len)` - return index of minimum among the elements $S[j]$, where $i \leq j < len$

`swap(S, i, mini)` - swap the positions of $S[i]$ and $S[mini]$

What is the invariant of the above loop?

Selection Sort - Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Dominating Operation: comparing 2 elements in array
Data Size: length of sequence (len)

The external loop is iterated $len - 1$ times, and in each i -th iteration of this loop we seek for minimum in a sequence of $(len-i)$ elements ($len - i$ comparisons)

$$W(len) = \sum_{i=1}^{len-1} i = \frac{len(len-1)}{2} = \Theta((len)^2)$$

Thus, the algorithm has **square complexity**.

Notice, that the average complexity $A(len)$ is the same as $W(len)$ - for a fixed length of the sequence the algorithm always executes the same number of operations. Even for **already sorted sequence!**

Insertion Sort

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

```
insertionSort(arr, len){  
    for(next = 1; next < len; next++){  
        curr = next;  
        temp = arr[next];  
        while((curr > 0) && (temp < arr[curr - 1])){  
            arr[curr] = arr[curr - 1];  
            curr--;  
        }  
        arr[curr] = temp;  
    }  
}
```

What is the invariant of the external loop?

Insertion Sort - Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort

Insertion Sort

Merge Sort

Linked Lists

Summary

(dominating operation and data size n is the same for all the algorithms discussed in this lecture)

What is the pessimistic case?

Insertion Sort - Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

(dominating operation and data size n is the same for all the algorithms discussed in this lecture)

What is the pessimistic case?

When the data is **invertedly** sorted. Then the complexity is:

$$W(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 + \Theta(n) = \Theta(n^2)$$

This algorithm is much more “intelligent” than the previous, because it adapts the amount of work to the degree of sortedness of the input data - the more sorted input the less number of comparisons (and swaps). In particular, for **already sorted** data it needs only $n-1$ comparisons (is linear in this case - very fast!).

Average Time Complexity Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists
Summary

Let's assume a simple model of input data - each permutation of input elements is equally likely. Then, for i -th iteration of the external loop the algorithm will need (on average):

$$\frac{1}{i} \sum_{j=1}^i j = \frac{1}{i} \frac{(i+1)i}{2} = \frac{i+1}{2}$$

comparisons. Thus, we obtain:

$$A(n) = \sum_{i=1}^{n-1} \frac{i+1}{2} = \frac{1}{2} \sum_{k=2}^n k = \frac{1}{4} n^2 + \Theta(n) = \Theta(n^2)$$

Analysis, cont.

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

On average, the algorithm is **twice faster** than Selection-sort, but still has **square** time complexity.

Anyway, square complexity for sorting is considered too high for large data (imagine sorting 1 million records in this way)

Is it possible to accelerate sorting?

Insertion-sort with Binary Search

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

It seems possible to take advantage of the invariant of the external loop in the Insertion-sort algorithm. It assures that in each i - th iteration of the external loop **subsequence** between the first and i - th element is **already sorted**.

Thus, if the sequence is kept in a random-access structure (RAM) - e.g. in an array, it is possible to apply the **binary search** algorithm to find the “correct” position (the external loop) instead of sequential search.

Notice, that in this case the algorithm needs the same number of comparisons for each input data.

Analysis of the “modified” Insertion-sort

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Let's compute the time complexity for the modified (with binary searching) algorithm (dominating operation: comparing 2 elements in the sequence, data size: the length of the input sequence - n). The analysis is as follows:

$$A(n) = W(n) = \sum_{i=1}^{n-1} \log_2 i = \log_2(\prod_{i=1}^{n-1} i) = \Theta(\log_2(n!)) = \Theta(n \cdot \log(n))$$

Thus, we seemingly improved the complexity of this algorithm. Notice, however, that we count **only comparisons** (this was a good choice for the original algorithm). However, the dominating operation set should be now **extended** by the **insert operation** which costs **linear** number of assigns (each) in case of array implementation - finally we did not improve the rank which is still **square**.

But we can improve it. In different way...

Divide and conquer sorting (1) - Merge Sort

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Let's apply the “divide and conquer” approach to the sorting problem.

- 1 divide the sequence into 2 halves
- 2 **sort** each half separately
- 3 merge the sorted halves

This approach is successful because sorted subsequences can be merged very quickly (i.e. with merely linear complexity)

Moreover, let's observe that sorting in point 2 can be recursively done with the same method (until the “halves” have zero lengths)

Thus, we have a working **recursive** sorting scheme (by merging).

Merge Sort - Scheme

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

```
mergeSort(S, len){  
  if(len <= 1) return S[0:len]  
  m = len/2  
  return merge(mergeSort(S[0:m], m), m  
               mergeSort(S[m:len], len-m), len-m)  
}
```

where:

- denotation $S[a:b]$ means the subsequence of elements $S[i]$ such that $a \leq i < b$
- the function $\text{merge}(S1, \text{len1}, S2, \text{len2})$ merges 2 (sorted) sequences $S1$ and $S2$ (of lengths len1 and len2) and **returns** the merged (and sorted) sequence.

Merge Function

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

input: a1, a2 - sorted sequences of numbers (of lengths len1, len2)

output: return merged (and sorted) sequences a1 and a2

```
merge(a1, len1, a2, len2){  
    i = j = k = 0;  
    result[len1 + len2] // memory allocation  
  
    while((i < len1) && (j < len2))  
        if(a1[i] < a2[j]) result[k++] = a1[i++];  
        else result[k++] = a2[j++];  
  
    while(i < len1) result[k++] = a1[i++];  
  
    while(j < len2) result[k++] = a2[j++];  
  
    return result;  
}
```


Analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort

Insertion Sort

Merge Sort

Linked Lists

Summary

Because the sequence is always divided into halves, the depth of recursion is $\log_2(len)$ (similarly as in binary search).

Furthermore, on each level of recursion the `merge(s1, len1, s2, len2)` function is called for a pair of arrays of the total length being len (independently on the recursion level).

Thus, the complexity of the Merge-sort algorithm depends on the complexity of the `merge` function which merges 2 sorted subsequences.

Merge Function - analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

`merge(a1, len1, a2, len2)`

Dominating operation: comparing 2 elements or indexes

Data size: the total length of 2 subsequences to be merged
 $n = len1 + len2$

Time complexity: $W(n) = A(n) =$

Merge Function - analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists
Summary

`merge(a1, len1, a2, len2)`

Dominating operation: comparing 2 elements or indexes

Data size: the total length of 2 subsequences to be merged
 $n = len1 + len2$

Time complexity: $W(n) = A(n) = \Theta(n)$

Unfortunately, if the sequences are represented as arrays the space complexity is high:

$S(n) =$

Merge Function - analysis

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists
Summary

`merge(a1, len1, a2, len2)`

Dominating operation: comparing 2 elements or indexes

Data size: the total length of 2 subsequences to be merged
 $n = len1 + len2$

Time complexity: $W(n) = A(n) = \Theta(n)$

Unfortunately, if the sequences are represented as arrays the space complexity is high:

$$S(n) = \Theta(n)$$

(this problem can be avoided by representing sequences as **linked lists**)

Time Complexity Analysis of Merge-sort

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

`mergeSort(S, len)`

Dominating operation: comparing 2 elements or indexes

Data size: length of sequence (len)

Thus, we have $\Theta(\log_2(len))$ levels of recursion and on each level we have some calls of the `merge` function with total aggregate complexity of $\Theta(len)$.

Thus, we obtain:

$$W(len) = A(len) = \Theta(len \cdot \log(len))$$

- the complexity is **linear-logarithmic**.

Short Summary of SS, IS and MS

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort

Insertion Sort

Merge Sort

Linked Lists

Summary

Selection and Insertion Sort:

quite simple algorithms of **square** ($\Theta(n^2)$) time complexity.

Merge sort is faster: **linear-logarithmic** ($\Theta(n \log_2(n))$), but would need quite much memory if sequences are implemented as arrays, and additional memory for recursion.

Is the difference in speed between $\Theta(n \log_2(n))$ and $\Theta(n^2)$ really **significant**?

Example: square vs linear-logarithmic complexity

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Consider 100 million logs of some server to be sorted

Assume a machine speed: **billion** comparisons in a second

Example: square vs linear-logarithmic complexity

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Consider 100 million logs of some server to be sorted

Assume a machine speed: **billion** comparisons in a second

How much time does it take for Insertion Sort?

Example: square vs linear-logarithmic complexity

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort

Insertion Sort

Merge Sort

Linked Lists

Summary

Consider 100 million logs of some server to be sorted

Assume a machine speed: **billion** comparisons in a second

How much time does it take for Insertion Sort?

How much for merge sort?

Linked Lists

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

To save memory (avoid unnecessary array allocation and element copying with each call of the merge function) a **different** (than arrays) representation of sequences.

Linked List:

It consists of **nodes**

Each node contains some data (e.g. the element of sequence) and **link** to the next node.

e.g.:

head-> (2)-> (3)-> (5)-> (8)-> null

Linked Lists

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

Linked List is the simplest representant of the **important family** of **linked data structures**. There are many variants of linked lists:

- uni-directional
- bi-directional
- cyclic

Also various graph-like data structures (e.g. trees) can be conveniently represented as linked structures.

Linked Lists vs Arrays

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

arrays:

- advantages: very fast random access, less memory consumption
- disadvantages: inserting has a linear time complexity (in terms of the array lengths)

linked lists:

- disadvantages: slower access (only through the “list head”), additional memory needed for storing links
- advantages: very fast modification operations - inserting, deleting of subsequences, and other reorganization operations

Linked Lists in MergeSort

Algorithms
and Data
Structures

Marcin
Sydow

Sorting
Selection Sort
Insertion Sort
Merge Sort


Linked Lists

Summary

If uni-directional lists are applied in MergeSort, the code should be slightly modified, but the idea is the same.

The time complexity will not change.

However, the space complexity will be improved if the input sequence is represented as a linked list. It will be **constant** in such a case: $S(n) = O(1)$ ¹

¹assuming the recursion implementation issue is neglected 

Questions/Problems:

Algorithms
and Data
Structures

Marcin
Sydow

Sorting

Selection Sort
Insertion Sort
Merge Sort

Linked Lists

Summary

- The sorting problem
- Selection Sort (idea, pseudo-code, analysis)
- Insertion Sort (as above)
- Merge Sort (as above)
- Linked Lists and comparison with arrays

Thank you for your attention!