CS 260
Design and Analysis of Algorithms
1. Introduction (Search and Sorting)

Mikhail Moshkov

Computer, Electrical and Mathematical Sciences & Engineering Division
King Abdullah University of Science and Technology

Preface

This course is devoted to the consideration of some approaches to design and analysis of algorithms.

Each algorithm $\mathcal{A}$ transforms input data x into output data y.

The run of $\mathcal{A}$ on the input data x can be characterized by some complexity measure $L_{\mathcal{A}}(x)$ (the number of steps of the algorithm, the size of used memory).

Preface

The understanding of the behavior of $L_{\mathcal{A}}(x)$ for different x is a very complicated problem.

Therefore we consider the behavior of $L_{\mathcal{A}}(x)$ in the worst case or the behavior of $L_{\mathcal{A}}(x)$ in the average case.

Preface

We fix some natural parameter n which characterizes the complexity of the input x representation, and we study maximum value of $L_{\mathcal{A}}(x)$ or average value of $L_{\mathcal{A}}(x)$ among all inputs x for which the complexity of x is equal to n (or is at most n).

So instead of $L_{\mathcal{A}}(x)$ we study a function $L_{\mathcal{A}}(n)$ which characterized the worst-case or average-case complexity.

For such functions we are able to obtain interesting and useful results.

Preface

In this course, we study only time complexity of algorithms.

We concentrate on algorithms, but from time to time we discuss also some useful data structures.

We begin from problems of search and sorting.

Introduction (Search and Sorting)

In the introduction, we consider problems of search and sorting, $O, \Omega, \Theta, o, \omega$ notation, and the notion of 2-3 trees.

Search in Ordered Array

Let we have an ordered array of elements from some linearly ordered set: $a_1 < a_2 < \ldots < a_n$.

The input of the algorithm of *search* is an element $x \in \{a_1, \ldots, a_n\}$.

The output of the algorithm is an index $j \in \{1, \ldots, n\}$ such that $x = a_j$. However, it will be more convenient for us to assume that the output of the algorithm is an element $a_j \in \{a_1, \ldots, a_n\}$ such that $x = a_j$.

Each step is a comparison of x with some a_i, $i \in \{1, \ldots, n\}$.

We will consider only deterministic sequential algorithms.

Search in Ordered Array

Each such algorithm can be represented in the form of a *decision tree* in which terminal nodes (leaves) are labeled with elements from $\{a_1, \ldots, a_n\}$. These are results of the algorithm work.

Each nonterminal node is labeled with a pair $x : a_i$ where a_i belongs to $\{a_1, \ldots, a_n\}$. In this node we compare x and a_i.

Exactly two edges start from the considered nonterminal node. These edges are labeled with $x \leq a_i$ and $x > a_i$, respectively.

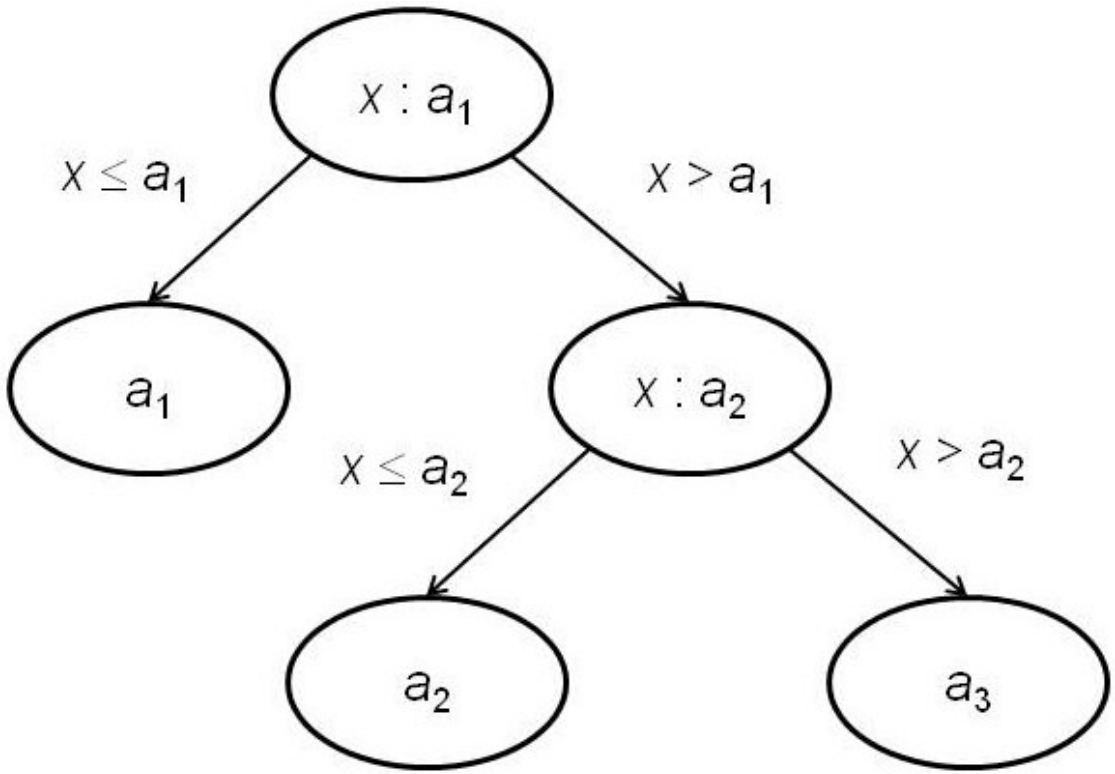
Search in Ordered Array

We will say that a node of decision tree is *realizable* if there exists an input for which the computation passes through this node.

Later we will assume that in the considered decision trees all nodes are realizable.

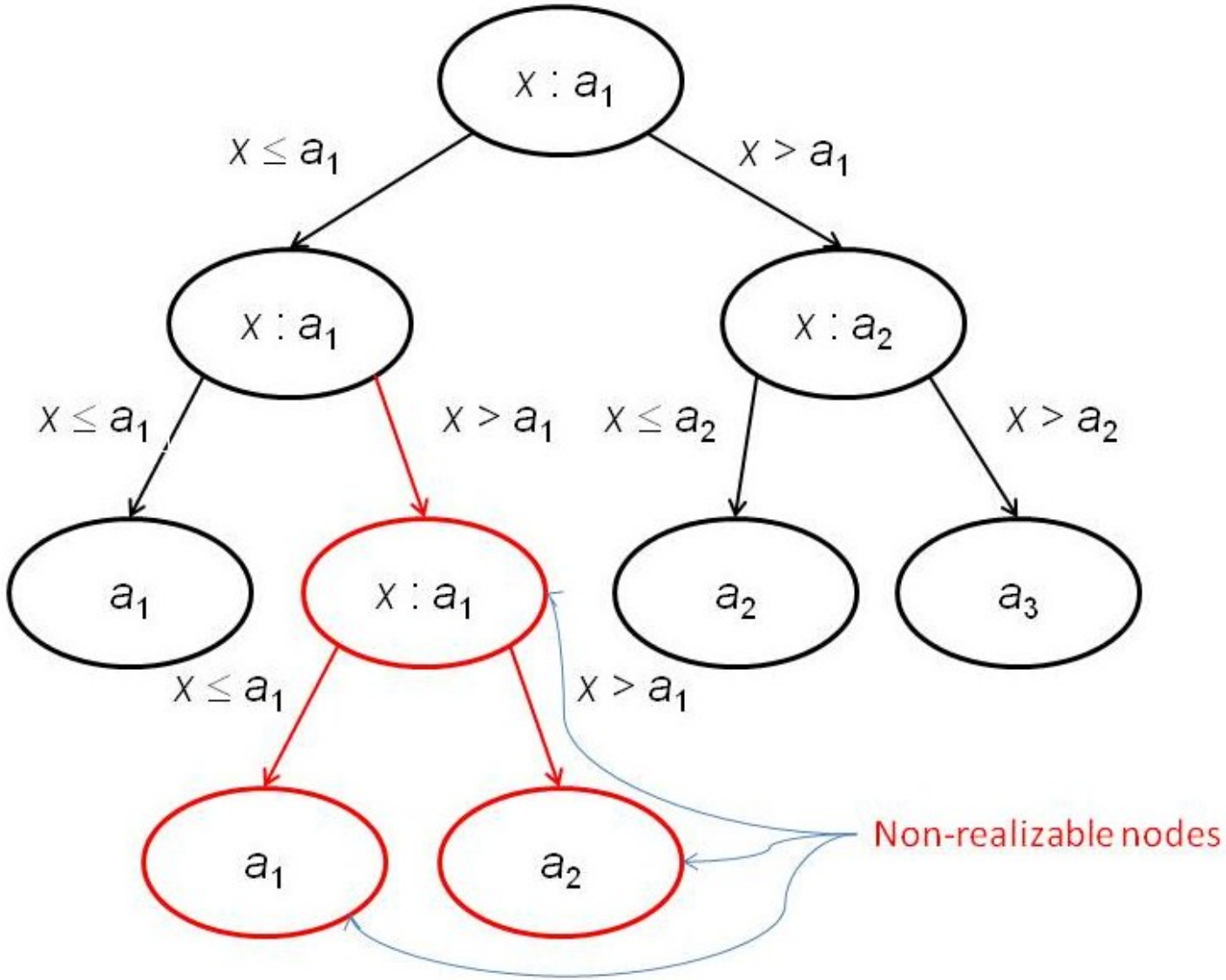
Search in Ordered Array

Example 1.1. Decision tree with only realizable nodes for the search in ordered array $a_1 < a_2 < a_3$, $x \in \{a_1, a_2, a_3\}$



Search in Ordered Array

Decision tree with non-realizable nodes:



Search in Ordered Array

Let us fix an algorithm $\mathcal{A}$ for search in ordered array with n elements. The worst-case complexity of the algorithm $\mathcal{A}$ is

$$L_{\mathcal{A}}(n) = \max\{L(a_i) : i = 1, \ldots, n\},$$

where $L(a_i)$ is the number of steps (comparisons) of the algorithm $\mathcal{A}$ if $x = a_i$.

The average-case complexity of the algorithm $\mathcal{A}$ is

$$L_{\mathcal{A}}^{avg}(n) = \frac{L(a_1) + \ldots + L(a_n)}{n}.$$

It is clear that $L_{\mathcal{A}}^{avg}(n) \leq L_{\mathcal{A}}(n)$.

Search in Ordered Array

Theorem 1.2. *There exists an algorithm $\mathcal{A}$ for search in ordered array with n elements for which $L_{\mathcal{A}}^{avg}(n) \leq L_{\mathcal{A}}(n) \leq \lceil \log_2 n \rceil$.*

Search in Ordered Array

Proof. We will describe recursively the *binary search* algorithm.

If $n = 1$ then $x = a_1$.

Let $n \geq 2$. We compute $k = \left\lfloor \frac{n}{2} \right\rfloor$ and compare x and a_k.

If $x \leq a_k$ then we (recursively) look for x in the array $a_1 < a_2 < \ldots < a_k$.

If $x > a_k$ then we (recursively) look for x in the array $a_{k+1} < a_{k+2} < \ldots < a_n$.

Search in Ordered Array

The length of the obtained array is either $\left\lfloor \frac{n}{2} \right\rfloor$ or $\left\lceil \frac{n}{2} \right\rceil$, and

$$L_{\mathcal{A}}(n) \leq 1 + \max\left(L_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right), L_{\mathcal{A}}\left(\left\lceil \frac{n}{2} \right\rceil\right)\right).$$

Search in Ordered Array

It is clear that $L_{\mathcal{A}}(1) = 0$.

Let us prove by induction on m that, for each natural n such that $2^{m-1} < n \leq 2^m$, the inequality $L_{\mathcal{A}}(n) \leq m = \lceil \log_2 n \rceil$ holds.

If $m = 0$ then $n = 1$ and $L_{\mathcal{A}}(1) = 0$.

Let the considered statement be true for some m and all nonnegative integers which are at most m. Let $2^m < n \leq 2^{m+1}$. Then $2^{m-1} \leq \left\lfloor \frac{n}{2} \right\rfloor \leq \left\lceil \frac{n}{2} \right\rceil \leq 2^m$ and, by inductive hypothesis, $L_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \leq m$ and $L_{\mathcal{A}}\left(\left\lceil \frac{n}{2} \right\rceil\right) \leq m$.

Therefore $L_{\mathcal{A}}(n) \leq 1 + \max\left(L_{\mathcal{A}}\left(\left\lfloor \frac{n}{2} \right\rfloor\right), L_{\mathcal{A}}\left(\left\lceil \frac{n}{2} \right\rceil\right)\right) \leq m + 1$, i.e., the statement is true for $m + 1$.

Thus, the inequality $L_{\mathcal{A}}(n) \leq \lceil \log_2 n \rceil$ is true for any n. Note that $L^{avg}_{\mathcal{A}}(n) \leq L_{\mathcal{A}}(n)$. □

Search in Ordered Array

Example 1.3. Search in ordered array
$1 < 3 < 6 < 7 < 8 < 9 < 10 < 11 < 12 < 14$, input $x = 3$

$n = 10$, $k = \lfloor \frac{10}{2} \rfloor = 5$
$x \leq 8$
$1 < 3 < 6 < 7 < 8$

$n = 5$, $k = \lfloor \frac{5}{2} \rfloor = 2$
$x \leq 3$
$1 < 3$

$n = 2$, $k = \lfloor \frac{2}{2} \rfloor = 1$
$x > 1$

The answer is 3

Search in Ordered Array

Each decision tree is a *binary tree with root* (*rooted binary tree*). In such a tree, each nonterminal node has two children.

To obtain lower bounds on $L_{\mathcal{A}}(n)$ and $L_{\mathcal{A}}^{avg}(n)$, we should study some properties of binary decision trees with root.

Let D be a binary tree with root.

By $h(D)$ we denote the *depth* of D – the maximum length of a path from the root to a terminal node.

The *length* of a path is the number of edges in this path.

Search in Ordered Array

Let $v_1, \ldots, v_n$ be all terminal nodes (leaves) of D. We denote by $h(v_i)$ the length of the path from the root of D to v_i.

The value $\sum_{i=1}^{n} h(v_i)$ is the *external path length* of D, and the value $h_{avg}(D) = \sum_{i=1}^{n} h(v_i)/n$ is the *average depth* of D.

The tree D is called *complete* if $h(v_i) = h(D)$ for $i = 1, \ldots, n$.

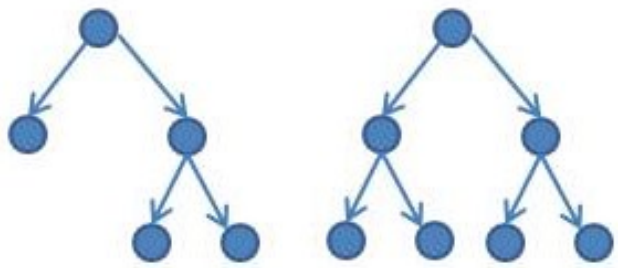
Search in Ordered Array

Lemma 1.4. *For any rooted binary tree D with n terminal nodes (leaves), the following inequalities hold:* 1) $h(D) \geq \lceil \log_2 n \rceil$ *and* 2) $h_{avg}(D) \geq \log_2 n$.

Search in Ordered Array

Proof. 1). Each rooted binary tree of the depth h has at most 2^h terminal nodes.

To prove this statement it is enough to extend the considered tree up to complete binary tree of the depth h, in which the length of each path from the root to a terminal node is equal to h. During this extension, the number of terminal nodes cannot decrease.



Search in Ordered Array

The number of terminal nodes in the complete binary tree is equal to 2^h. Then $n \leq 2^{h(D)}$ and $h(D) \geq \log_2 n$.

Since $h(D)$ is a nonnegative integer, we have $h(D) \geq \lceil \log_2 n \rceil$.

Search in Ordered Array

2). Let $v_1, \ldots, v_n$ be all terminal nodes of D.

For $i = 1, \ldots, n$, instead of the node v_i we add to D a complete rooted binary tree with depth equal to $h(D) - h(v_i)$ (the number of terminal nodes in added tree is equal to $2^{h(D)-h(v_i)}$).

As a result, we obtain a complete rooted binary tree D' with $2^{h(D)}$ terminal nodes.

Therefore $\sum_{i=1}^{n} 2^{h(D)-h(v_i)} = 2^{h(D)}$ and $\sum_{i=1}^{n} 2^{-h(v_i)} = 1$.

Search in Ordered Array

Using well known inequality for arithmetic and geometric means we obtain

$$\frac{1}{n} = \frac{1}{n}\sum_{i=1}^{n} 2^{-h(v_i)} \geq \sqrt[n]{\prod_{i=1}^{n} 2^{-h(v_i)}} = \sqrt[n]{2^{-\sum_{i=1}^{n} h(v_i)}}.$$

Therefore

$$2^{\sum_{i=1}^{n} h(v_i)} \geq n^n$$

and

$$\frac{1}{n}\sum_{i=1}^{n} h(v_i) \geq \log_2 n.$$

□

Search in Ordered Array

Theorem 1.5. *For any algorithm $\mathcal{A}$ for search in ordered array with n elements, $L_{\mathcal{A}}(n) \geq \lceil \log_2 n \rceil$ and $L^{avg}_{\mathcal{A}}(n) \geq \log_2 n$.*

Search in Ordered Array

Proof. The considered algorithm can be represented as a decision tree D which is a rooted binary tree.

It is clear that $L_{\mathcal{A}}(n) = h(D)$ and $L_{\mathcal{A}}^{avg}(n) = h_{avg}(D)$.

Search in Ordered Array

Any element from $\{a_1, \ldots, a_n\}$ can be a result of the algorithm $\mathcal{A}$ work. Therefore, the tree D has n terminal nodes (leaves).

Using Lemma 1.4, we obtain $h(D) \geq \lceil \log_2 n \rceil$ and $h_{avg}(D) \geq \log_2 n$.

Therefore $L_{\mathcal{A}}(n) \geq \lceil \log_2 n \rceil$ and $L_{\mathcal{A}}^{avg}(n) \geq \log_2 n$. $\square$

Search in Ordered Array

The worst-case complexity of search in ordered array with n elements is $L_{search}(n) = \min L_{\mathcal{A}}(n)$ where minimum is considered among all algorithms $\mathcal{A}$ of search in ordered array with n elements for each of which each step is a comparison of the input x and an element a_i from the array.

The average-case complexity of search in ordered array with n elements is $L^{avg}_{search}(n) = \min L^{avg}_{\mathcal{A}}(n)$ where minimum is considered among all algorithms $\mathcal{A}$ of search in ordered array with n elements for each of which each step is a comparison of the input x and an element a_i from the array.

From Theorems 1.2 and 1.5 it follows that $L_{search}(n) = \lceil \log_2 n \rceil$ and $\log_2 n \leq L^{avg}_{search}(n) \leq \lceil \log_2 n \rceil$.

Search in Ordered Array

One can show that $L^{avg}_{search}(n) = \varphi(n)/n$, where

$$\varphi(n) = (\lceil \log_2 n \rceil + 1)n - 2^{\lceil \log_2 n \rceil}$$

is the minimum external path length of a rooted binary tree with n terminal nodes.

Details can be found in the book

D.E. Knuth, Sorting and Searching, second ed., The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, MA, 1998.

Search in Ordered Array

A rooted binary tree with n terminal nodes has the external path length equal to $\varphi(n)$ if and only if $2^q - n$ terminal nodes are in the layer $q-1$ and $2n - 2^q$ terminal nodes are in the layer q where $q = \lceil \log_2 n \rceil$ and the root of the tree is in layer 0.

It is possible to construct such decision tree for the search in ordered array with n elements. This tree will have also minimum depth.

Sorting

Let us consider the *problem of sorting* on linearly ordered set.

Input: a sequence of elements $a_1, a_2, \ldots, a_n$ of some infinite linearly ordered set (for simplicity, we will assume that $a_i \neq a_j$ if $i \neq j$) .

Output: a permutation $(i_1, i_2, \ldots, i_n)$ of numbers $1, 2, \ldots, n$ such that $a_{i_1} < a_{i_2} < \ldots < a_{i_n}$.

We denote by P_n the set of all permutations of numbers $1, 2, \ldots, n$. The cardinality of the set P_n is equal to $n!$.

Sorting

We will study algorithms for this problem solving for each of which each step is a comparison $a_i : a_j$ of any two elements a_i and a_j. There are two possible answers: $a_i < a_j$ or $a_i > a_j$.

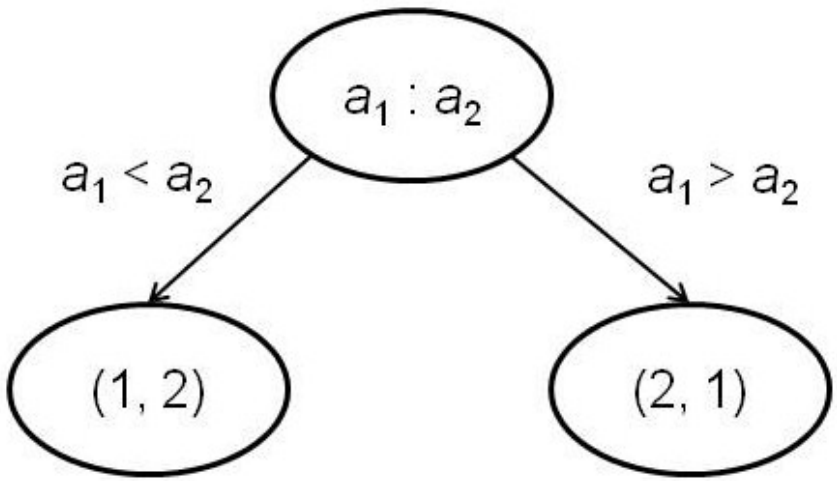
We will consider only deterministic sequential algorithms for sorting. Such an algorithm can be represented as a decision tree.

Sorting

Each terminal node of the decision tree is labeled with a permutation $(i_1, i_2, \ldots, i_n)$.

Each nonterminal node is labeled with a pair of elements $a_i : a_j$, $i \neq j$.

Exactly two edges start from this node. These edges are labeled with $a_i < a_j$ and $a_i > a_j$ respectively.



Sorting

Let $\mathcal{A}$ be an algorithm for sorting of n elements $a_1, \ldots, a_n$, and $L_{\mathcal{A}}(a_1, \ldots, a_n)$ be the number of steps (comparisons) which $\mathcal{A}$ makes during the sorting of $a_1, \ldots, a_n$.

Let $b_1, \ldots, b_n$ be pairwise different elements from the considered linearly ordered set.

The worst-case time complexity of $\mathcal{A}$ is equal to

$$L_{\mathcal{A}}(n) = \max\{L_{\mathcal{A}}(b_{i_1}, \ldots, b_{i_n}) : (i_1, \ldots, i_n) \in P_n\}.$$

The average-case time complexity of $\mathcal{A}$ is equal to

$$L_{\mathcal{A}}^{avg}(n) = \frac{\sum_{(i_1,\ldots,i_n)\in P_n} L_{\mathcal{A}}(b_{i_1}, \ldots, b_{i_n})}{n!}.$$

It is clear that $L_{\mathcal{A}}^{avg}(n) \leq L_{\mathcal{A}}(n)$.

Sorting

Theorem 1.6. For any algorithm $\mathcal{A}$ for sorting of n elements, the following inequalities hold: $L_{\mathcal{A}}(n) \geq \lceil \log_2 n! \rceil$ and $L_{\mathcal{A}}^{avg}(n) \geq \log_2 n!$.

Proof. The considered algorithm $\mathcal{A}$ can be represented as a decision tree D.

Each permutation $(i_1, i_2, \ldots, i_n)$ can be an output of $\mathcal{A}$.

So for each permutation $(i_1, i_2, \ldots, i_n)$ there exists a terminal node of D which is labeled with this permutation.

Therefore the number of terminal nodes in D is equal to $n!$.

Sorting

Using Lemma 1.4 we obtain $h(D) \geq \lceil \log_2 n! \rceil$ and $h_{avg}(D) \geq \log_2 n!$.

It is clear that $L_{\mathcal{A}}(n) = h(D)$ and $L_{\mathcal{A}}^{avg}(n) = h_{avg}(D)$.

Thus $L_{\mathcal{A}}(n) \geq \lceil \log_2 n! \rceil$ and $L_{\mathcal{A}}^{avg}(n) \geq \log_2 n!$. □

Sorting

has ready prove

Corollary 1.7. *For any algorithm $\mathcal{A}$ for sorting of n elements, the following inequalities hold:* $L_{\mathcal{A}}(n) \geq (1 - o(1))n \log_2 n$ and $L_{\mathcal{A}}^{avg}(n) \geq (1 - o(1))n \log_2 n$

Here $o(1)$ means a nonnegative for large enough n function $f(n)$ for which $\lim_{n\to\infty} f(n) = 0$.

Sorting

To prove this corollary it is enough to use inequalities that follow from Stirling formula:

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n < n! < \sqrt{2\pi n}\left(\frac{n}{e}\right)^n e^{\frac{1}{12n}}.$$

We have $\log_2 n! > n\log_2 n - n\log_2 e + \frac{1}{2}\log_2(2\pi) + \frac{1}{2}\log_2 n = (1 - o(1))n\log_2 n$, where

$$o(1) = \frac{n\log_2 e - \frac{1}{2}\log_2(2\pi) - \frac{1}{2}\log_2 n}{n\log_2 n}.$$

Sorting

We now consider two sorting algorithms for which time complexity is close to the obtained lower bounds.

Insertion Sort

We will sequentially solve subproblems: to sort $a_1, \ldots, a_k$ for $k = 1, 2, \ldots, n$.

If $k = 1$ then the solution is trivial and equal to (1).

When $k = n$ we obtain the solution of the whole problem.

Insertion Sort

The passage from the subproblem with $k-1$ elements to the subproblem with k elements it the following.

We have an ordered sequence $a_{i_1} < a_{i_2} < \ldots < a_{i_{k-1}}$ and the element a_k.

There are k possible positions for a_k: before a_{i_1}, between a_{i_1} and a_{i_2} ,... , after $a_{i_{k-1}}$.

To find the proper position we can use a bit modified binary search which requires $\lceil \log_2 k \rceil$ steps (comparisons).

Insertion Sort

Example 1.8. Insertion sort of array
$2, 7, 4, 6, 3, 1$

1). $-2-$, $\quad 7 > 2$

2). $-2-7-$, $\quad 4 > 2,\ 4 < 7$

3). $-2-4-7-$, $\quad 6 > 4,\ 6 < 7$

4). $-2-4-6-7-$, $\quad 3 < 4,\ 3 > 2$

5). $-2-3-4-6-7-$, $\quad 1 < 4,\ 1 < 2$

6). $-1-2-3-4-6-7-$

The answer is $1, 2, 3, 4, 6, 7$ (instead of permutation we consider sorted array)

Insertion Sort

Let $L_{ins}(n)$ be the time complexity (the number of comparisons) in the worst-case for the insertion sort under the sorting of n elements.

Let $L_{ins}^{avg}(n)$ be the time complexity (the number of comparisons) in the average-case for the insertion sort under the sorting of n elements.

Theorem 1.9. *The time complexity of insertion sort satisfies the following inequalities:* $L_{ins}(n) \leq \log_2 n! + n - 1$ *and* $L_{ins}^{avg}(n) \leq \log_2 n! + n - 1$.

Insertion Sort

Proof. For $k = 1$ we do not make any comparisons.

For $k = 2, \ldots, n$, when we pass from the subproblem with $k - 1$ elements to the subproblem with k elements, we make at most $\lceil \log_2 k \rceil$ comparisons (steps).

So the whole number of steps is at most

$$\begin{aligned} & \lceil \log_2 2 \rceil + \lceil \log_2 3 \rceil + \ldots + \lceil \log_2 n \rceil \\ \leq \quad & \log_2 2 + \log_2 3 + \ldots + \log_2 n + (n-1) = \log_2 n! + n - 1. \end{aligned}$$

Therefore $L_{ins}(n) \leq \log_2 n! + n - 1$ and $L_{ins}^{avg}(n) \leq \log_2 n! + n - 1$. □

Insertion Sort

Corollary 1.10. $L_{ins}(n) \leq (1 + o(1))n \log_2 n$ and $L_{ins}^{avg}(n) \leq (1 + o(1))n \log_2 n$.

Insertion Sort

The worst-case complexity of sorting of n elements is $L_{sort}(n) = \min L_{\mathcal{A}}(n)$, and the average-case complexity of sorting of n elements is $L^{avg}_{sort}(n) = \min L^{avg}_{\mathcal{A}}(n)$ where minimum is considered among all algorithms $\mathcal{A}$ of sorting n elements for each of which each step is a comparison of any two elements a_i and a_j.

From Corollaries 1.7 and 1.10 it follows that

$$\lim_{n\to\infty} \frac{L_{sort}(n)}{n \log_2 n} = 1 \quad \text{and} \quad \lim_{n\to\infty} \frac{L^{avg}_{sort}(n)}{n \log_2 n} = 1.$$

Merge Sort

We will describe the merge sort of n elements recursively. If $n = 1$ then the problem is trivial.

If $n \geq 2$ then we divide the sequence $a_1, a_2, , \ldots, a_n$ into the subsequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$.

We sort these subsequences by the same algorithm, and then merge two sorted subsequences $\alpha = (a_{i_1} < a_{i_2} < \ldots < a_{i_{\lfloor \frac{n}{2} \rfloor}})$ and $\beta = (a_{j_1} < a_{j_2} < \ldots < a_{j_{n - \lfloor \frac{n}{2} \rfloor}})$ to obtain the whole sorted sequence $\gamma = (a_{k_1} < a_{k_2} < \ldots < a_{k_n})$.

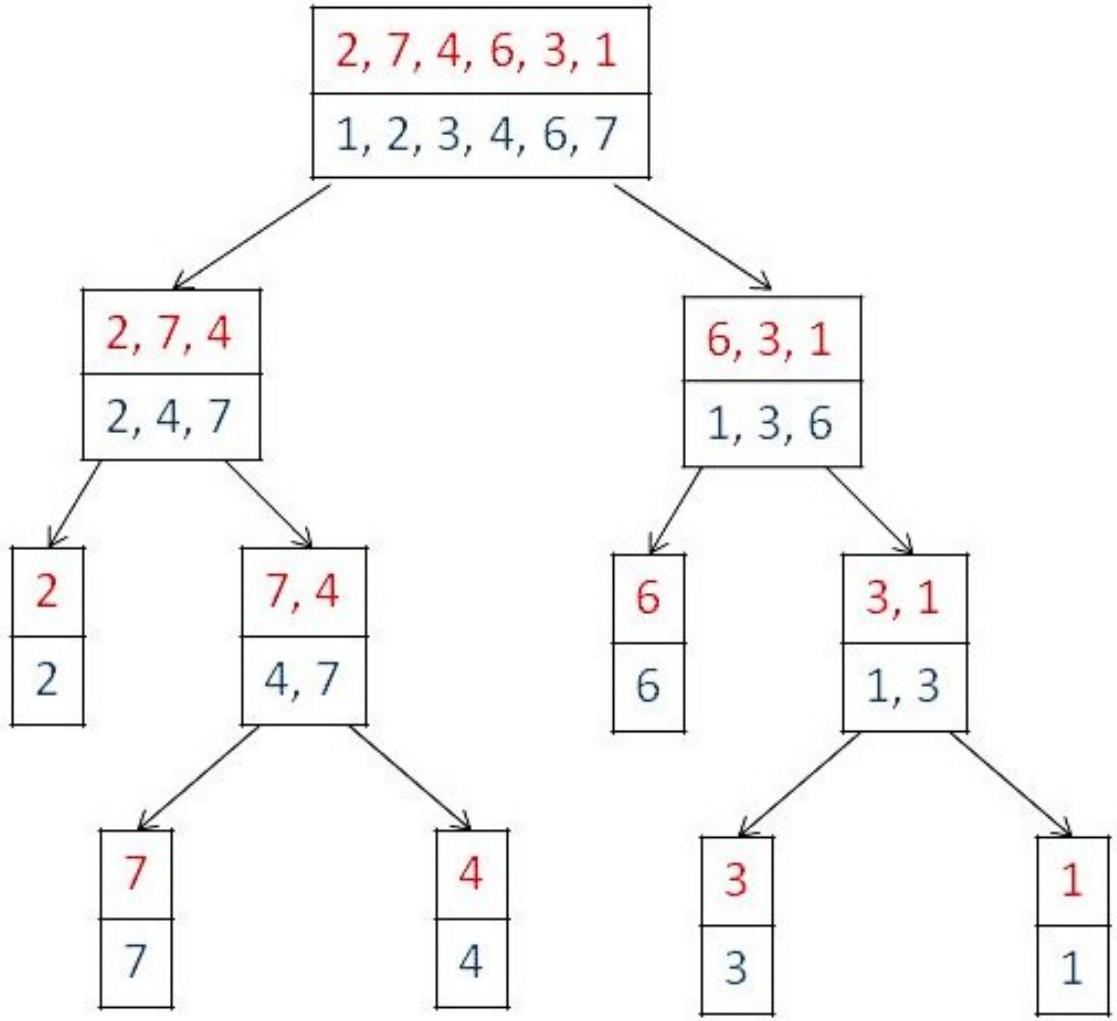
Merge Sort

During each step we compare first elements in α and β, and transfer the smallest one to γ as the next element.

If α or β becomes empty then we transfer the remained elements to γ in the same order.

Merge Sort

Example 1.11. Merge sort of array $2, 7, 4, 6, 3, 1$



Merge Sort

Let $L_{mer}(n)$ be the time complexity (the number of comparisons) in the worst-case for the merge sort under the sorting of n elements.

Then $L_{mer}(1) = 0$ and

$$L_{mer}(n) \leq L_{mer}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + L_{mer}\left(\left\lceil \frac{n}{2} \right\rceil\right) + n - 1$$

if $n \geq 2$ since the merge of two subsequences requires $n - 1$ comparisons in the worst case.

Merge Sort

Lemma 1.12. $L_{mer}(n) \leq n \log_2 n - n + 1$ *for* $n = 2^k$ *where* k *is any nonnegative integer.*

Merge Sort

Proof. We prove the statement by induction on k. If $k = 0$ then the considered statement holds since $L_{mer}(1) = 0$. Let the considered statement be true for each k, $0 \leq k \leq m-1$, where m is a natural number.

Then, for $k = m$, we have

$$\begin{aligned} L_{mer}(2^m) &\leq 2L_{mer}(2^{m-1}) + 2^m - 1 \\ &\leq 2(2^{m-1}(m-1) - 2^{m-1} + 1) + 2^m - 1 \\ &= m2^m - 2^m + 1, \end{aligned}$$

i.e., for $k = m$, the considered statement is true.

Therefore, this statement is true for any nonnegative integer k. □

Merge Sort

Let $L_{mer}^{avg}(n)$ be the time complexity (the number of comparisons) in the average-case for the merge sort under the sorting of n elements.

Theorem 1.13. $L_{mer}^{avg}(n) \leq L_{mer}(n) < 2n \log_2 n + 1$ *for any natural* n.

Merge Sort

Proof. First, we prove that $L_{mer}(n)$ is a nondecreasing function.

Let $a_1, \ldots, a_n$ be pairwise different elements from a linearly ordered set and a be an element from this set such that $a > a_i$ for $i = 1, \ldots, n$.

We prove by induction on n that there exists $j \in \{1, \ldots, n\}$ such that

$$L_{mer}(a_1, \ldots, a_n) \leq L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n).$$

If $j = n$ then $a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n = a_1, \ldots, a_n, a$. In this case we have the inequality

$$L_{mer}(a_1, \ldots, a_n) \leq L_{mer}(a_1, \ldots, a_n, a).$$

Merge Sort

It is clear that

$$L_{mer}(a_1) \leq L_{mer}(a_1, a).$$

It means that the considered statement holds if $n = 1$.

Let this statement hold for $1, \ldots, n-1$ where $n \geq 2$.

We now show that the statement holds for n.

Merge Sort

Let n be odd. Then $\lfloor \frac{n}{2} \rfloor < n$ and, according to the inductive hypothesis, there exists $j \in \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}$ such that

$$L_{mer}(a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}) \leq L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor}).$$

It is clear that

$$L_{mer}(a_1, \ldots, a_n) = L_{mer}(a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}) + L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n) + A$$

where A is the number of comparisons during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$.

Merge Sort

One can show that

$$\begin{aligned} L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n) &= L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor}) \\ &+ L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n) + B \end{aligned}$$

where B is the number of comparisons during the merge of ordered sequences $a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$.

Merge Sort

Taking into account that $a > a_i$ for $i = 1, \ldots, n$, it is not difficult to see that during the merge of ordered sequences $a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$ the algorithm makes the same comparisons as during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$ plus, possibly, comparisons of a with some elements from $\{a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n\}$.

Therefore $A \leq B$. As a result, we have

$$L_{mer}(a_1, \ldots, a_n) \leq L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n).$$

Merge Sort

Let n be even. Then $n - \left\lfloor \frac{n}{2} \right\rfloor < n$ and, according to the inductive hypothesis, there exists $j \in \left\{ \left\lfloor \frac{n}{2} \right\rfloor + 1, \ldots, n \right\}$ such that

$$L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n) \leq L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_j, a, a_{j+1}, \ldots, a_n).$$

We know that

$$L_{mer}(a_1, \ldots, a_n) = L_{mer}(a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}) + L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n) + A$$

where A is the number of comparisons during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$.

Merge Sort

One can show that

$$L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n) = L_{mer}(a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}) + L_{mer}(a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_j, a, a_{j+1}, \ldots, a_n) + C$$

where C is the number of comparisons during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_j, a, a_{j+1}, \ldots, a_n$.

Merge Sort

Taking into account that $a > a_i$ for $i = 1, \ldots, n$, it is not difficult to see that during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_j, a, a_{j+1}, \ldots, a_n$ the algorithm makes the same comparisons as during the merge of ordered sequences $a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}$ and $a_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, a_n$ plus, possibly, comparisons of a with some elements from $\{a_1, \ldots, a_{\lfloor \frac{n}{2} \rfloor}\}$.

Therefore $A \leq C$. As a result, we have

$$L_{mer}(a_1, \ldots, a_n) \leq L_{mer}(a_1, \ldots, a_j, a, a_{j+1}, \ldots, a_n).$$

Merge Sort

Thus the considered statement holds. From this statement it follows that $L_{mer}(n)$ is a nondecreasing function.

The statement of the theorem holds for $n = 1$.

For any natural $n \geq 2$ there exists a natural k such that $2^{k-1} < n \leq 2^k$.

From the fact that $L_{mer}(n)$ is a nondecreasing function and from Lemma 1.12 it follows that

$$L_{mer}(n) \leq L_{mer}(2^k) \leq 2^k k - 2^k + 1 = 2^k(k-1) + 1 < 2n \log_2 n + 1.$$

Therefore $L^{avg}_{mer}(n) \leq L_{mer}(n) < 2n \log_2 n + 1$. □

Decision Trees with Minimum Depth

Table from the book of Knuth The Art of Computer Programming, vol. 3, Sorting and Searching.

$n =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\lceil \log_2 n! \rceil =$	0	1	3	5	7	10	13	16	19	22	26	29	33	37	41	45	49
$F(n) =$	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42	46	50

$n =$	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
$\lceil \log_2 n! \rceil =$	53	57	62	66	70	75	80	84	89	94	98	103	108	113	118	123
$F(n) =$	54	58	62	66	71	76	81	86	91	96	101	106	111	116	121	126

$F(n)$ is the depth of a decision tree for sorting of n elements constructed by Ford-Johnson algorithm (published in 1959).

$L_{sort}(12) = 30$, Wells 1965; $L_{sort}(13) = 34$, Kasai, Sawato, Iwata 1994; $L_{sort}(14) = 38$, $L_{sort}(15) = 42$, $L_{sort}(22) = 71$, Peczarski 2004, 2007; $L_{sort}(16) =?$

Decision Trees with Minimum Average Depth

n	2	3	4	5
$L_{sort}^{avg}(n)$	$2/2!$	$16/3!$	$112/4!$	$832/5!$
$\varphi(n!)/n!$	$2/2!$	$16/3!$	$112/4!$	$832/5!$
$\lvert Opt_{avg}(n)\rvert$	1	12	$27,744$	$2,418,647,040$

n	6	7	8
$L_{sort}^{avg}(n)$	$6896/6!$	$62416/7!$	$620160/8!$
$\varphi(n!)/n!$	$6896/6!$	$62\,368/7!$	$619\,904/8!$
$\lvert Opt_{avg}(n)\rvert$	1.968×10^{263}	4.341×10^{6681}	8.548×10^{326365}

$\varphi(n!) = (\lceil \log_2 n! \rceil + 1) \cdot n! - 2^{\lceil \log_2 n! \rceil}$.

$\lvert Opt_{avg}(n)\rvert$ is the number of decision trees for sorting n elements which have minimum average depth.

Césary proved in 1968 that $L_{sort}^{avg}(n) \neq \varphi(n!)/n!$ for $n = 7, 8$ and $L_{sort}^{avg}(n) = \varphi(n!)/n!$ for $n = 9, 10$.

Kollár found the number $L_{sort}^{avg}(7)$ in 1986.

$O, \Omega, \Theta, o, \omega$ Notation

Up to now we used mainly exact formulas to estimate time complexity of algorithms.

We now consider some useful notation that allow us to work with asymptotic order of growth of functions.

$O, \Omega, \Theta, o, \omega$ Notation

We will say that $f(n)$ is an *asymptotically nonnegative* (*positive*) function if there exists $n_0 > 0$ such that $f(n) \geq 0$ ($f(n) > 0$) for any $n \geq n_0$.

We assume now that the considered functions $f(n)$ and $g(n)$ are asymptotically nonnegative.

$O, \Omega, \Theta, o, \omega$ Notation

We will write $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

We will write $f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that $cg(n) \leq f(n)$ for all $n \geq n_0$.

We will write $f(n) = \Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

$O, \Omega, \Theta, o, \omega$ Notation

For any two functions $f(n)$ and $g(n)$ we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

$O, \Omega, \Theta, o, \omega$ Notation

Example 1.14. Let $f(n) = pn^2 + qn + r$ and $p, q, r > 0$. We will show that $f(n) = \Theta(n^2)$.

We have $f(n) \geq pn^2$. Therefore $f(n) = \Omega(n^2)$.

Let $t = p + q + r$. Then $f(n) \leq tn^2$ for $n \geq 1$. Thus $f(n) = O(n^2)$ and $f(n) = \Theta(n^2)$.

$O, \Omega, \Theta, o, \omega$ Notation

Example 1.15.

If $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ and $a_d > 0$ then $f(n) = \Theta(n^d)$

If $a > 1$ and $d > 0$ then $\log_a n = O(n^d)$

If $r > 1$ and $d > 0$ then $n^d = O(r^n)$

If $a > 1$ and $b > 1$ then $\log_a n = \frac{\log_b n}{\log_b a}$ and $\log_a n = \Theta(\log_b n)$. We will write $O(\log n)$ instead of $O(\log_a n)$.

$O, \Omega, \Theta, o, \omega$ Notation

We will write $f(n) = o(g(n))$ if for any constant $c > 0$ there exists a positive constant n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$.

We will write $f(n) = \omega(g(n))$ if for any constant $c > 0$ there exists a positive constant n_0 such that $cg(n) < f(n)$ for all $n \geq n_0$.

Example 1.16.

$L_{ins}(n) \leq (1 + o(1))n \log_2 n \Longrightarrow L_{ins}(n) = O(n \log n)$

$L_{mer}(n) \leq 2n \log_2 n + 1 \Longrightarrow L_{mer}(n) = O(n \log n)$

$(1 - o(1))n \log_2 n \leq L_{sort}(n) \leq (1 + o(1))n \log_2 n$
$\Longrightarrow L_{sort}(n) = \Theta(n \log n)$

$L^{avg}_{sort}(n) = \Theta(n \log n)$

$O, \Omega, \Theta, o, \omega$ Notation

Let us consider some properties of O, Ω, Θ, o, and ω.

We assume now that the considered functions $f(n)$, $g(n)$ and $h(n)$ are asymptotically positive.

$O, \Omega, \Theta, o, \omega$ Notation

Transitivity: For any $B \in \{O, \Omega, \Theta, o, \omega\}$, if $f(n) = B(g(n))$ and $g(n) = B(h(n))$ then $f(n) = B(h(n))$.

Reflexivity: For any $B \in \{O, \Omega, \Theta\}$, $f(n) = B(f(n))$.

Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

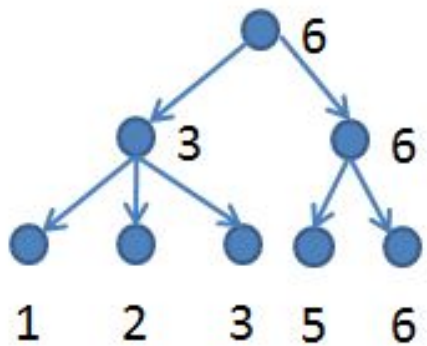
Transpose Symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$; $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

2-3 Trees

To simplify the process of sorting and search we can use balanced search trees. Here we consider a modification of 2-3 trees. We will say about this modification as about 2-3 trees.

Let we have a linearly ordered set A and n elements $a_1 < \ldots < a_n$ from A (for simplicity, we assume that $a_i \neq a_j$ if $i \neq j$).

2-3 tree is a rooted tree with n leaves (terminal nodes) which are in the same level and are labeled with $a_1, \ldots, a_n$ from the left to the right.

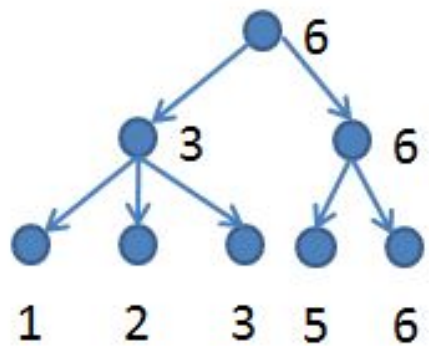


2-3 Trees

Each internal (nonterminal) node v of the tree has either two or three children.

This node v is labeled with maximum element attached to leaves of the subtree with the root v.

We will call this element a *guide*.

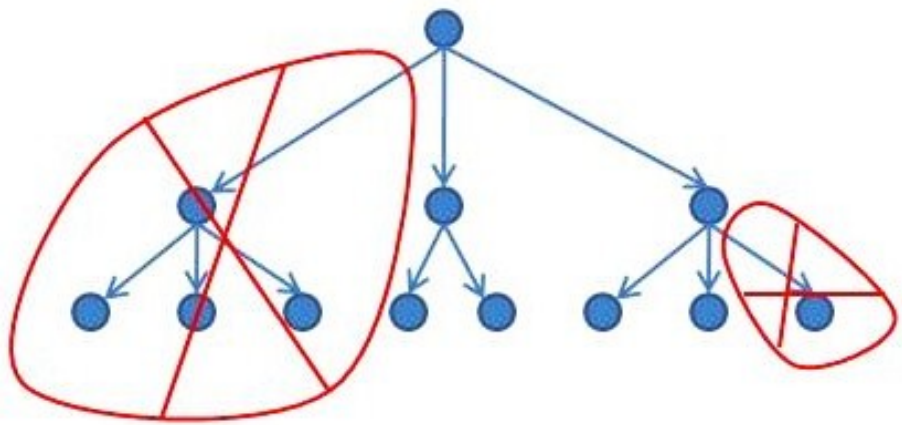


2-3 Trees

Let us show that the depth h of this tree is $O(\log n)$.

By removal of some edges and nodes from the considered tree, we can obtain a complete binary tree of the depth h.

This tree contains exactly 2^h terminal nodes. Therefore $2^h \leq n$, $h \leq \log_2 n$ and $h = O(\log n)$.



2-3 Trees

The considered tree allows us to solve the *problem of sorting* very easily: we should read labels attached to leaves from the left to the right.

2-3 Trees

Also this tree allows us to solve efficiently the following *problem of search*: for a given element $a \in A$, we should either find $a_i \in \{a_1, \ldots, a_n\}$ such that $a = a_i$, or show that $a < a_1$, or $a > a_n$, or $a_i < a < a_{i+1}$ for some $i \in \{1, \ldots, n-1\}$.

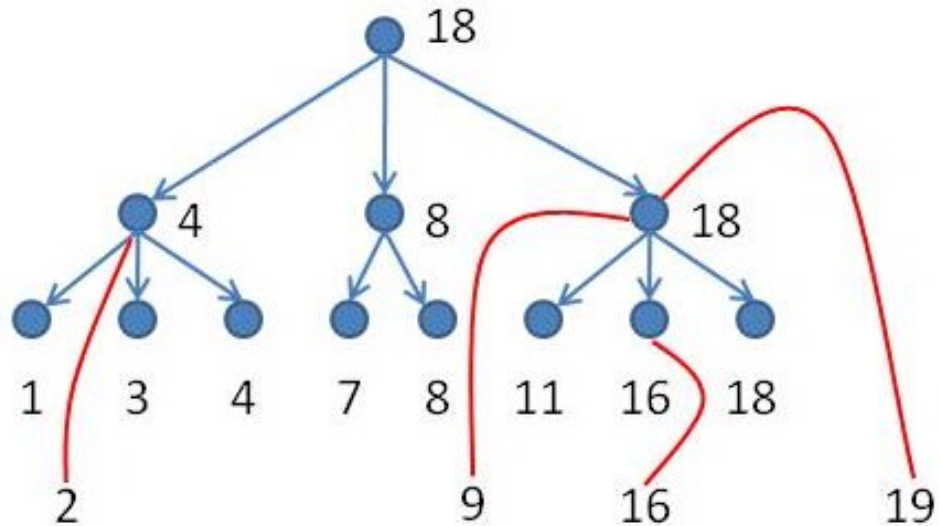
To this end we can compare a with guides in a path from the root to a non-terminal node p for which all children are leaves.

The comparison of a with elements attached to children of p gives us the solution of the search problem.

To solve this problem we should make $O(h) = O(\log n)$ comparisons.

2-3 Trees

Example 1.17. Search for elements $2, 19, 9$ and 16



2-3 Trees

To work efficiently with 2-3 trees we should have fast algorithms for operations of insertion of $a \in A$ to the tree and for the operation of the deletion of $a \in A$ from the tree.

2-3 Trees

Let us describe the *operation of insertion*. We solve the problem of search for a. Let during the solution of this problem we come to a node p for which all children are leaves.

If a is attached to one of children of p then we leave the tree untouched. Otherwise, we add a as a child of p.

If p now has four children then we split p into two nodes p_1 and p_2 each with two children.

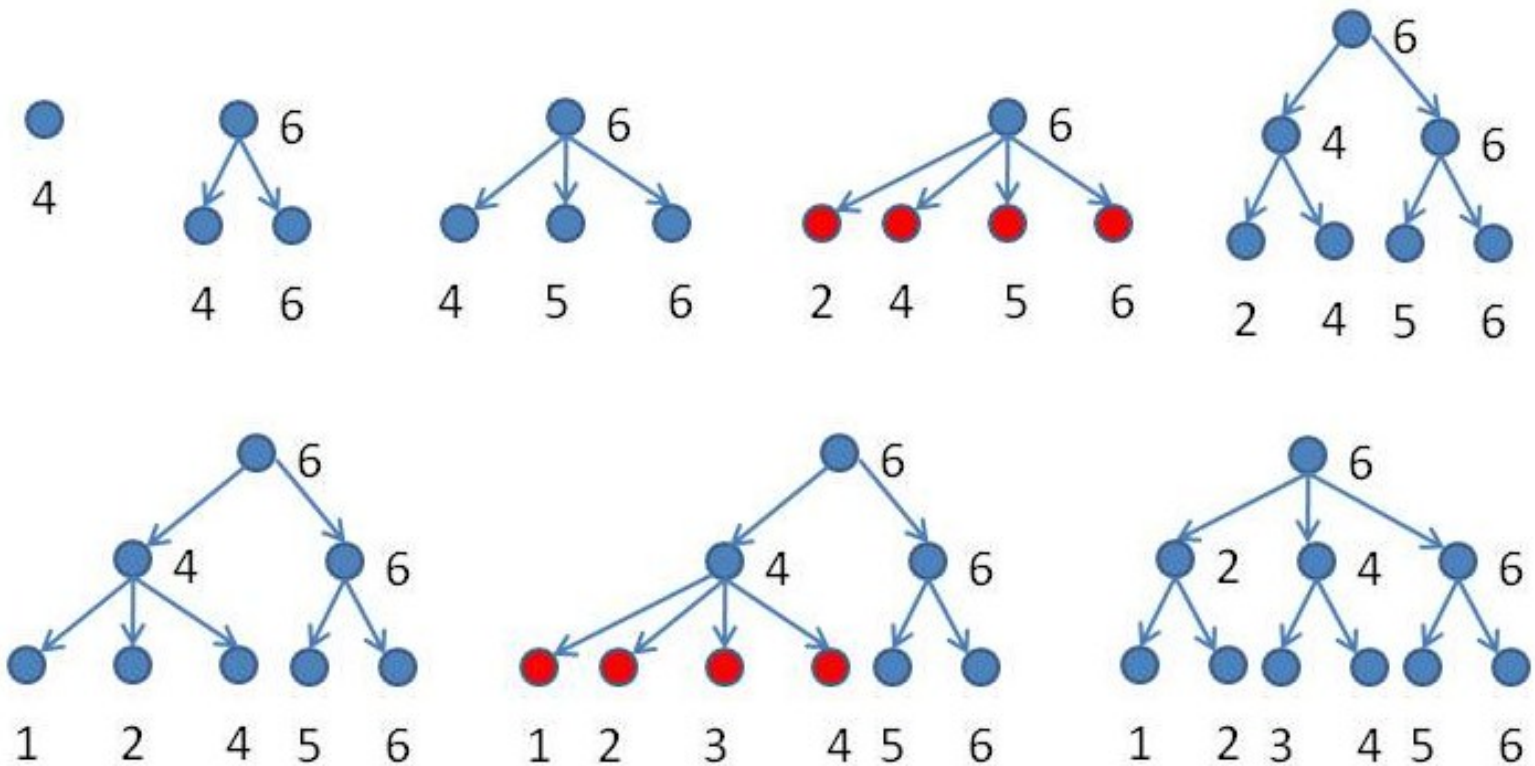
After that we process parent of p in the same way. If p has no parent (p is the root) we create new root. The depth of the new tree is the depth of initial tree plus 1.

Also we should update guides.

The needed time for the insertion operation is $O(h) = O(\log n)$.

2-3 Trees

Example 1.18. Insertion of $4, 6, 5, 2, 1$ and 3



2-3 Trees

We now describe the *operation of deletion*. We solve the problem of search for a, and come to a node p for which all children are leaves.

If a is not a label of a child of p then we leave the tree untouched. Otherwise, we remove the child of p labeled with a.

2-3 Trees

Let p have now only one child. If p is the root then we delete p. If one of immediate siblings of p has three children we borrow one for p.

Let none of immediate siblings of p have three children. Then the sibling q must have two children. We give the only child of p to q and delete p. After that we process parent of p, etc.

Also we should update guides.

The time needed for the deletion operation is $O(h) = O(\log n)$.

Example 1.19. Operation of deletion