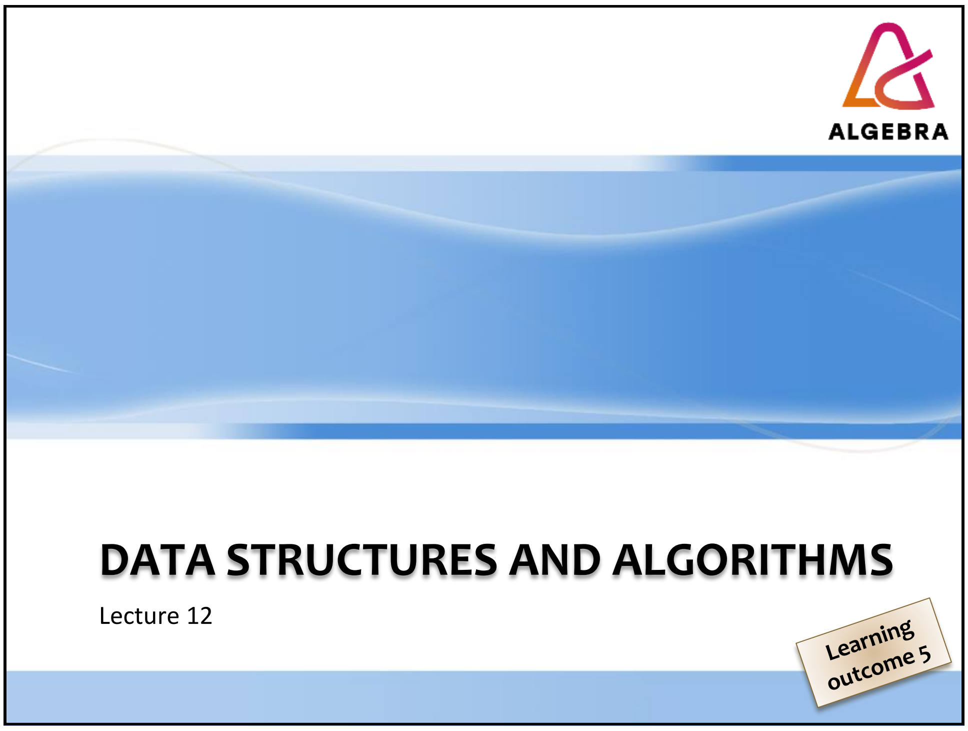
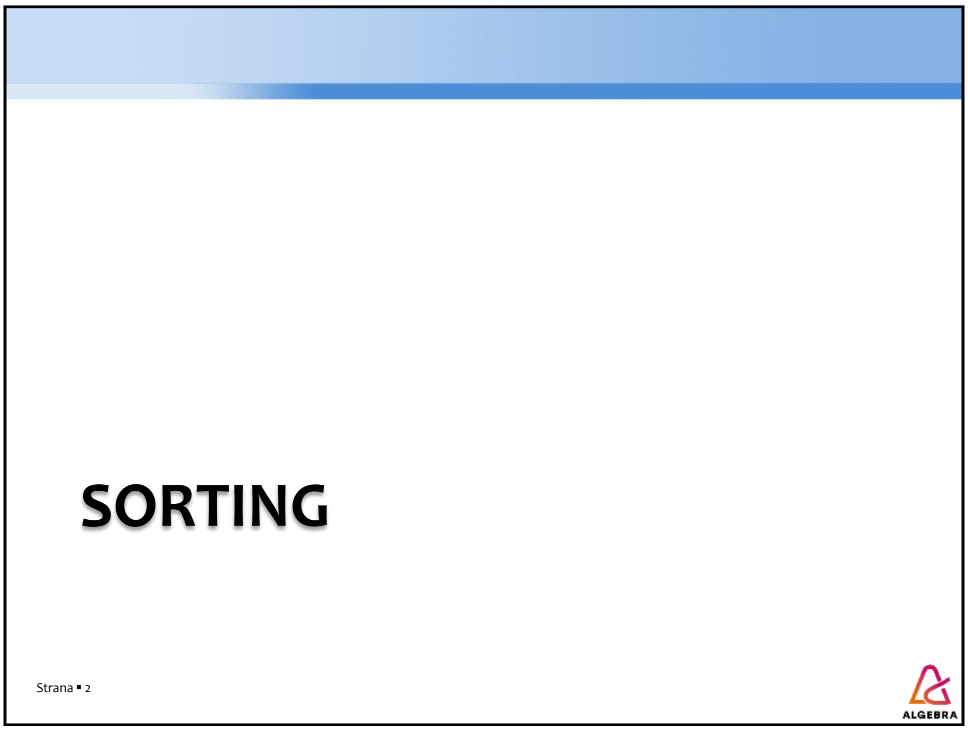


ALGEBRA

DATA STRUCTURES AND ALGORITHMS

Lecture 12

Learning outcome 5



SORTING

Strana ▪ 2

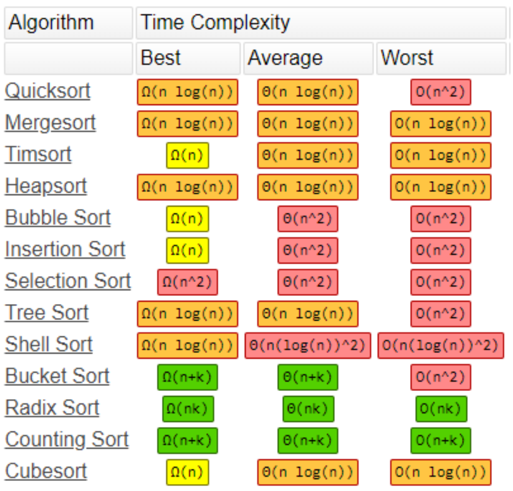
ALGEBRA

Introduction to sorting

- Sorting is the process of arranging a series of elements in ascending or descending order
- When sorting, we must take into account the following:
  - Sorting criteria - what we sort by and in what order
    - For example, we can sort people alphabetically by first and last name
  - Algorithm complexity (table on the next slide)
  - How much memory does the algorithm need to perform sorting
    - A good measure is the number of swaps
  - Is the algorithm stable
    - A stable algorithm does not change the mutual order of two equal elements

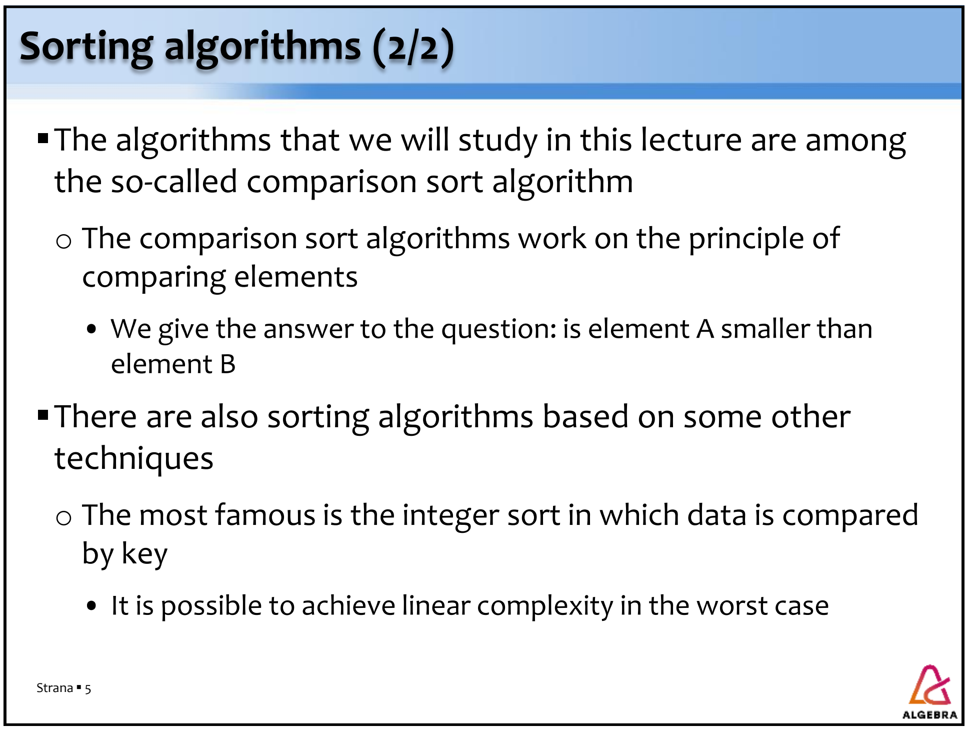
Sorting algorithms (1/2)

- Among the most well-known sorting algorithms are:



Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))

Taken from:
bigocheatsheet.com

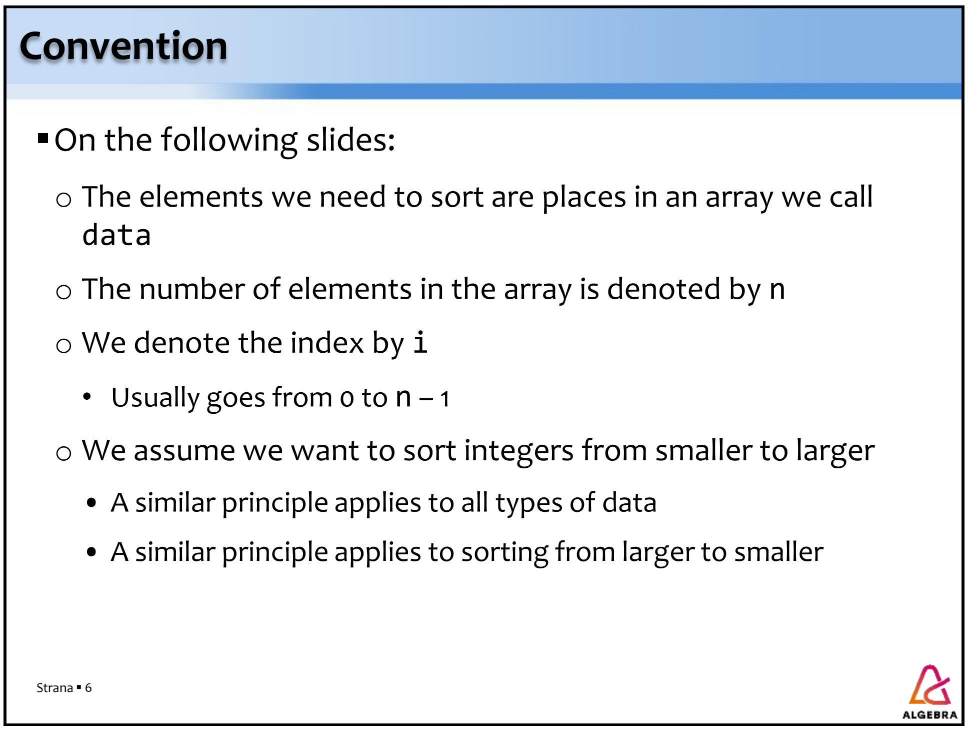


Sorting algorithms (2/2)

- The algorithms that we will study in this lecture are among the so-called comparison sort algorithm
  - The comparison sort algorithms work on the principle of comparing elements
    - We give the answer to the question: is element A smaller than element B
- There are also sorting algorithms based on some other techniques
  - The most famous is the integer sort in which data is compared by key
    - It is possible to achieve linear complexity in the worst case

Strana ▪ 5

ALGEBRA



Convention

- On the following slides:
  - The elements we need to sort are places in an array we call `data`
  - The number of elements in the array is denoted by `n`
  - We denote the index by `i`
    - Usually goes from 0 to `n` – 1
  - We assume we want to sort integers from smaller to larger
    - A similar principle applies to all types of data
    - A similar principle applies to sorting from larger to smaller

Strana ▪ 6

ALGEBRA

BOGO SORT

Bogo sort

- Also known as: stupid sort, slowsort, random sort, monkey sort, …
- Extremely inefficient, never used in practice:
  - Random version: there is no limiting function
  - Deterministic version: $O((n+1)!)$
- Tactics:
  1. Check that the items are sorted (if so, end) => $\Omega(n)$
  2. If not, shuffle the elements in random order and go to the previous step
     - The deterministic version says to use the next permutation instead of shuffling in the random order

Permutations

- Permutation is the name for rearranging elements into another order
  - For example, we have elements: [1, 2, 3, 4, 5]
  - One of possible permutation is: [2, 5, 4, 3, 1]
  - If we have *n* elements, then there are *n*! permutations
- STL comes with several functions that simplify working with permutations
  - `lexicographical_compare` checks whether the first sequence is lexicographically smaller than the second
  - `next_permutation` calculates the next permutation for the given sequence

Lexicographic comparison

- Used to alphabetically sort words in dictionaries
- Suppose we compare two words
  - The characters are compared, starting from the beginning
    - If we take the first two characters that are different, the smaller word is the one whose character alphabetically comes first
    - If we get to the end of one word and all the characters were equal, the smaller word is the one with fewer characters
- For example, let's compare words „Mario" and „Marko"
  - „Mario" is smaller because a letter „i" comes before „k"
- For example, compare the words „Java" and „JavaScript"
  - „Java" is smaller because first four letters are the same, and „Java" is a shorter word

Permutation functions (1/2)

- `bool lexicographical_compare(be1, end1, be2, end2)`
  - Returns true if the first range is lexicographically smaller than the second
  - Example:

```
string s1 = "Mario";
string s2 = "Marko";

cout << lexicographical_compare(
             s1.begin(), s1.end(), s2.begin(), s2.end()) << endl;

vector<int> v1 = { 1, 2, 3, 4, 5 };
vector<int> v2 = { 1, 2, 3, 4, 6 };

cout << lexicographical_compare(
             v1.begin(), v1.end(), v2.begin(), v2.end()) << endl;
```

Permutation functions (2/2)

- `bool next_permutation(begin, end)`
  - Rearranges elements into next larger lexicographic permutation:
    - Returns true if there is a larger lexicographic permutation
      - In the largest lexicographic permutation elements are sorted descending
    - Otherwise it returns false and rearranges the elements to the smallest permutation
      - In the smallest lexicographic permutation elements are sorted ascending
  - Example:

```
vector<int> v({ 5, 3, 1, 2, 4 });
int n = nfact(v.size());
for (int i = 0; i < n; i++) {
    std::cout << v[0] << ' ' << v[1] << ' ' << v[2] << ' ' <<
                                           v[3] << ' ' << v[4];
    cout << " (" << next_permutation(v.begin(), v.end()) << ")"
                                                  << endl;
};
```

BUBBLE SORT

Bubble sort

Bubble Sort $\Omega(n)$ $\Theta(n^2)$ $O(n^2)$

- Tactics: compare two adjacent elements and if they are not in correct order, replace them
  - After the first pass, the biggest element ended up at the end of the array and we don’t touch it anymore because it makes up the **sorted part of the array**
    - We say that he bubbled up to the end of the array
  - We repeat the procedure for the remaining elements
  - The algorithm stops after n - 1 passes or after the pass without replacement

Performance

- Easiest to implement, but has the worst efficiency
  - At worst, but also in the average case it is O(n^2)
  - The problem is also the large number of swaps
- The only advantage is the built-in ability to detect that the array has already been sorted
- Not used in practice

Bubble sort example

- Example:

```
 Start: [16, 32, 42, 10, 20]
Pass 1: [16, 32, 10, 20, 42]
Pass 2: [16, 10, 20, 32, 42]
Pass 3: [10, 16, 20, 32, 42]
Pass 4: [10, 16, 20, 32, 42]
   End: [10, 16, 20, 32, 42]
```

- Resources:
  - cs.usfca.edu/~galles/visualization/ComparisonSort.html
  - youtube.com/watch?v=lyZQPjUT5B4

Bubble sort implementation

```
for (int i = 0; i < n - 1; i++) {
    bool sorted = true;

    for (int j = 0; j < n - 1 - i; j++) {
        if (data[j] > data[j + 1]) {
            swap(data[j], data[j + 1]);
            sorted = false;
        }
    }

    if (sorted) {
        break;
    }
}
```

SELECTION SORT

Selection sort

Selection Sort Ω(n^2) Θ(n^2) O(n^2)

- Tactics:
  - Let the array have a sorted and an unsorted part
  - Find the smallest element in the unsorted part:
    - Replace it with the first element in the unsorted part
    - Declare that this element now belongs to the sorted part
  - Repeat until the whole array is sorted

ALGEBRA

Selection sort example

- Example:

```
 Start: [16, 32, 42, 10, 20]
Pass 1: [10, 32, 42, 16, 20]
Pass 2: [10, 16, 42, 32, 20]
Pass 3: [10, 16, 20, 32, 42]
Pass 4: [10, 16, 20, 32, 42]
   End: [10, 16, 20, 32, 42]
```

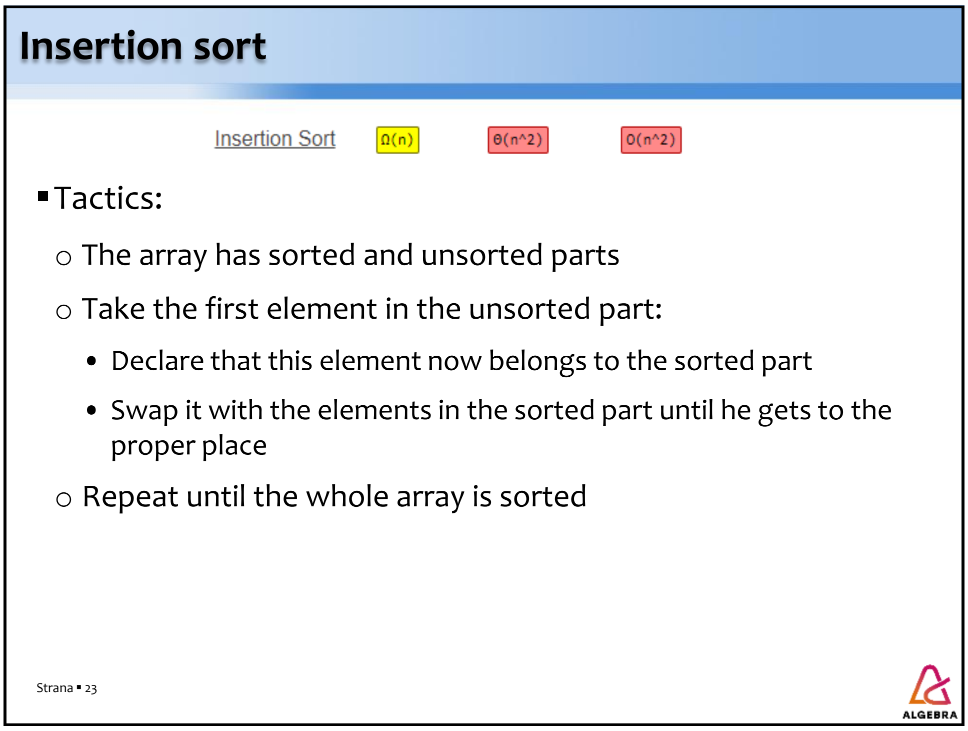
- Look at the example at cs.usfca.edu/~galles/visualization/ComparisonSort.html

ALGEBRA

Selection sort implementation

```
for (int i = 0; i < n - 1; i++) { // Unsorted part
    int min_index = i;
    for (int j = i + 1; j < n; j++) {
        if (data[j] < data[min_index]) {
            min_index = j;
        }
    }
    swap(data[min_index], data[i]);
}
```

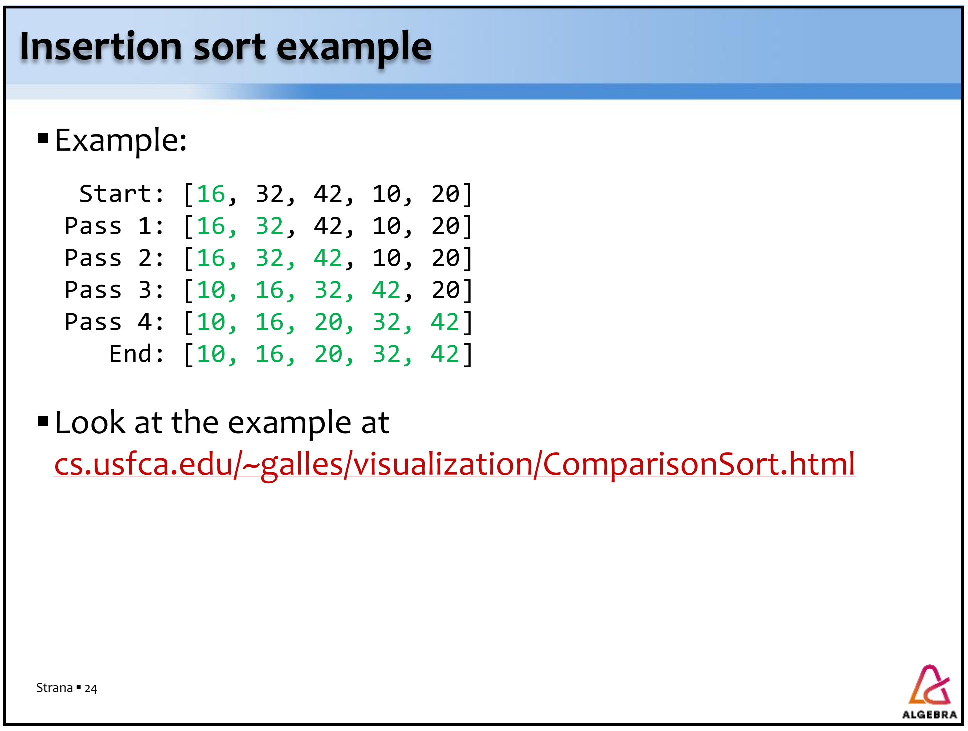
INSERTION SORT



Insertion sort

Insertion Sort $\Omega(n)$ $\Theta(n^2)$ $O(n^2)$

- Tactics:
  - The array has sorted and unsorted parts
  - Take the first element in the unsorted part:
    - Declare that this element now belongs to the sorted part
    - Swap it with the elements in the sorted part until he gets to the proper place
  - Repeat until the whole array is sorted



Insertion sort example

- Example:

```
 Start: [16, 32, 42, 10, 20]
Pass 1: [16, 32, 42, 10, 20]
Pass 2: [16, 32, 42, 10, 20]
Pass 3: [10, 16, 32, 42, 20]
Pass 4: [10, 16, 20, 32, 42]
   End: [10, 16, 20, 32, 42]
```

- Look at the example at cs.usfca.edu/~galles/visualization/ComparisonSort.html

Insertion sort implementation

```
for (int i = 1; i < n; i++) { // Skip the first one.
    for (int j = i; j > 0 && data[j - 1] > data[j]; j--) {
        swap(data[j], data[j - 1]);
    }
}
```

SHELL SORT

Shell sort

Shell Sort $\Omega(n \log(n))$ $\Theta(n(\log(n))^2)$ $O(n(\log(n))^2)$

- Named after the author, Donald Shell
- A variation of an insertion sort
  - Solves the problem of insertion sort: small elements at the end of the array require a lot of swaps
- Tactics: we will prepare the array so that the insertion sort works on it faster. The preparation is done by sorting smaller parts of the array (in order to bring some extreme elements to their place before the insertion sort would happen)
- Efficiency depends on the algorithm for selecting smaller parts of the array

A more detailed description

- A more detailed description of shell sort:
  - Split the array to h_1 subarrays, and every subarray is made from the following elements:
    - 1st subarray: `data`[0], `data`[h_1], `data`[2h_1], …
    - 2nd subarray: `data`[1], `data`[h_1+1], `data`[2h_1+1], …
    - h_1th subarray: `data`[h_1-1], `data`[2h_1-1], `data`[3h_1-1], …
  - Each subarray is sorted by insertion sort
  - We take number $h_2 < h_1$ and repeat the procedure until we reach $h_k = 1$
  - Sequence $h_1, h_2, \ldots, h_k = 1$ is called a gap sequence
    - $h_k = 1$ is an insertion sort of the entire array

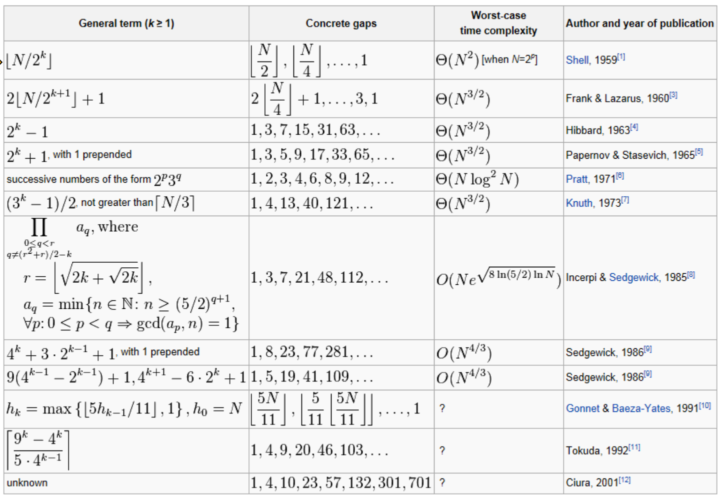
Shell sort example

- See example at cs.usfca.edu/~galles/visualization/ComparisonSort.html
- We have an array of 50 elements
- *h_1* = 25 (therefore, in the first pass we have 25 subarrays of 2 elements that we sort by insertion sort)
- *h_2* = 12
- *h_3* = 6
- *h_4* = 3
- *h_5* = 1 – classic insertion sort, but of an array that has no extremes, so the efficiency is great

ALGEBRA

Gap sequences

- Source: wikipedia



General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
$\lfloor N/2^k \rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta(N^2)$ [when N=2^p]	Shell, 1959[1]
$2\lfloor N/2^{k+1} \rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta(N^{3/2})$	Frank & Lazarus, 1960[3]
$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta(N^{3/2})$	Hibbard, 1963[4]
$2^k + 1$, with 1 prepended	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta(N^{3/2})$	Papernov & Stasevich, 1965[5]
successive numbers of the form $2^p 3^q$	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta(N \log^2 N)$	Pratt, 1971[6]
$(3^k - 1)/2$, not greater than $\lceil N/3 \rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta(N^{3/2})$	Knuth, 1973[7]
$\prod\limits_{\substack{0 \leq q < r \\ q \neq (r^2+r)/2-k}} a_q$, where $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$, $a_q = \min\{n \in \mathbb{N} \colon n \geq (5/2)^{q+1}, \forall p \colon 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\}$	$1, 3, 7, 21, 48, 112, \ldots$	$O(Ne^{\sqrt{8 \ln(5/2) \ln N}})$	Incerpi & Sedgewick, 1985[8]
$4^k + 3 \cdot 2^{k-1} + 1$, with 1 prepended	$1, 8, 23, 77, 281, \ldots$	$O(N^{4/3})$	Sedgewick, 1986[9]
$9(4^{k-1} - 2^{k-1}) + 1, 4^{k+1} - 6 \cdot 2^k + 1$	$1, 5, 19, 41, 109, \ldots$	$O(N^{4/3})$	Sedgewick, 1986[9]
$h_k = \max\{\lfloor 5h_{k-1}/11 \rfloor, 1\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11} \left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	?	Gonnet & Baeza-Yates, 1991[10]
$\left\lceil \frac{9^k - 4^k}{5 \cdot 4^{k-1}} \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	?	Tokuda, 1992[11]
unknown	$1, 4, 10, 23, 57, 132, 301, 701$	?	Ciura, 2001[12]

ALGEBRA

Shell sort implementation

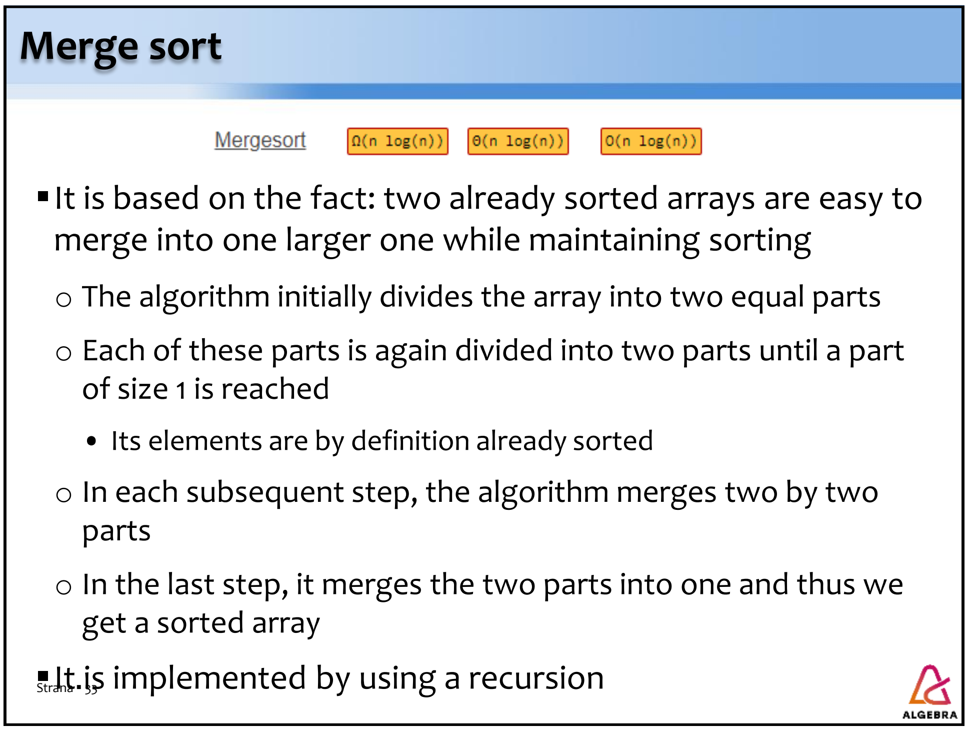
- Shell sort implementation with original gap sequence from 195

```
for (int step = n / 2; step > 0; step /= 2) {

   for (int i = step; i < n; i++) {
      int temp = data[i];
      for (int j = i; j >= step && data[j - step] > temp; j -= step) {
         swap(data[j], data[j - step]);
      }
   }

}
```

ALGEBRA

MERGE SORT

ALGEBRA



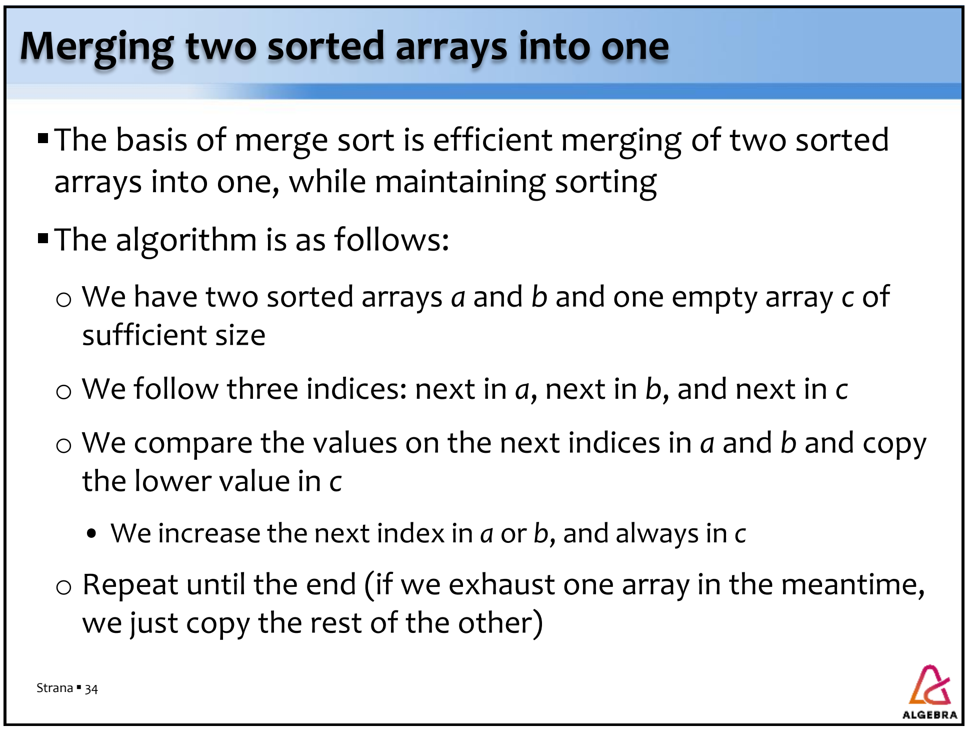
Merge sort

Mergesort `Ω(n log(n))` `Θ(n log(n))` `O(n log(n))`

- It is based on the fact: two already sorted arrays are easy to merge into one larger one while maintaining sorting
  - The algorithm initially divides the array into two equal parts
  - Each of these parts is again divided into two parts until a part of size 1 is reached
    - Its elements are by definition already sorted
  - In each subsequent step, the algorithm merges two by two parts
  - In the last step, it merges the two parts into one and thus we get a sorted array
- It is implemented by using a recursion

Strana ▪ 33

ALGEBRA



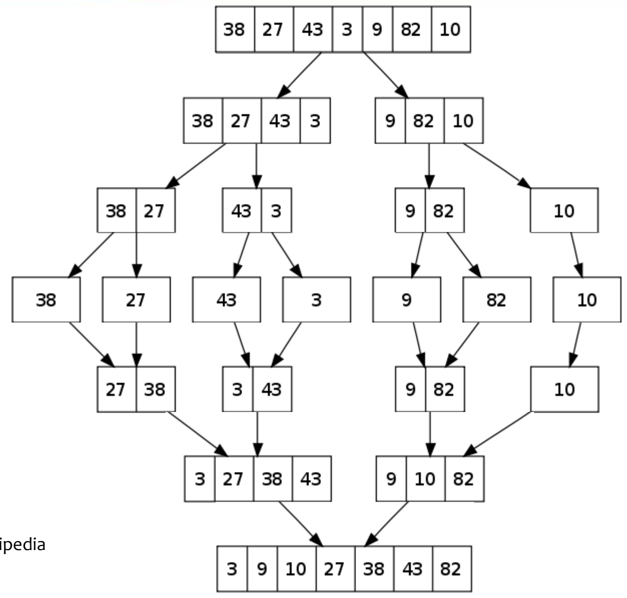
Merging two sorted arrays into one

- The basis of merge sort is efficient merging of two sorted arrays into one, while maintaining sorting
- The algorithm is as follows:
  - We have two sorted arrays *a* and *b* and one empty array *c* of sufficient size
  - We follow three indices: next in *a*, next in *b*, and next in *c*
  - We compare the values on the next indices in *a* and *b* and copy the lower value in *c*
    - We increase the next index in *a* or *b*, and always in *c*
  - Repeat until the end (if we exhaust one array in the meantime, we just copy the rest of the other)

Strana ▪ 34

ALGEBRA

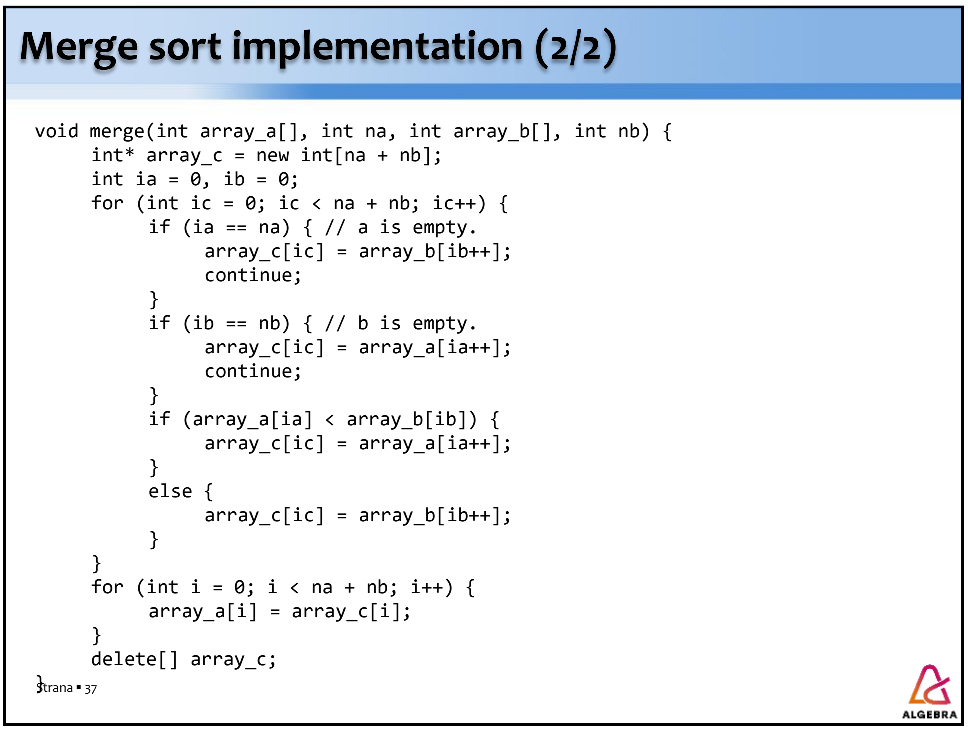
Merge sort example



Source: wikipedia

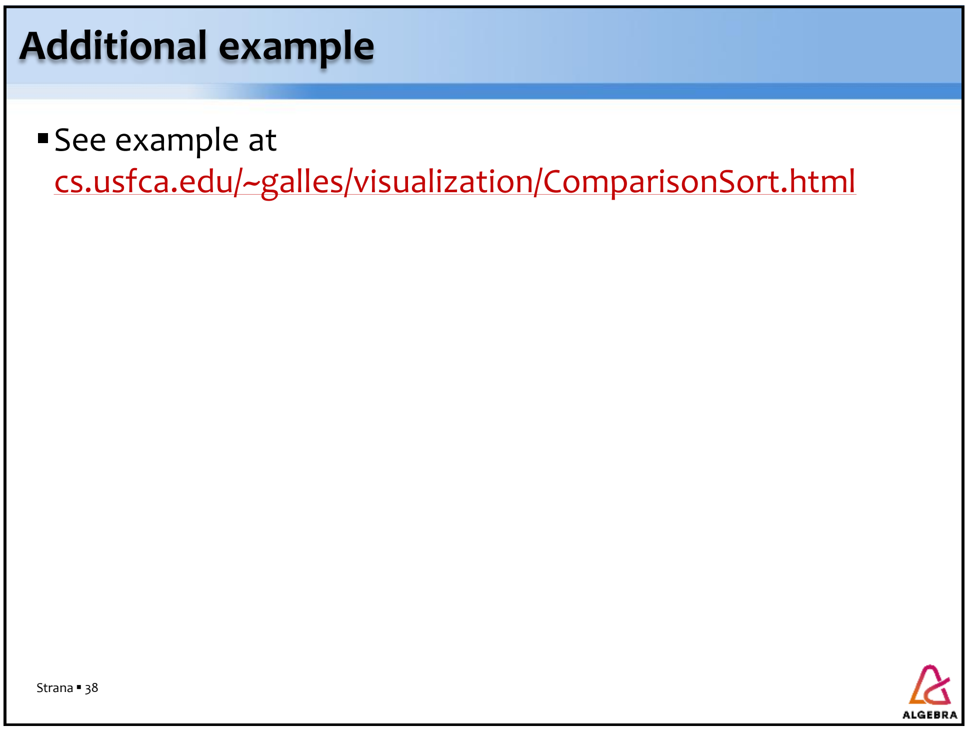
Merge sort implementation (1/2)

```
void merge_sort(int data[], int from, int to) {
       if (from == to) { // Stop condition.
              return;
       }

       int mid = (from + to) / 2;
       merge_sort(data, from, mid);
       merge_sort(data, mid + 1, to);

       merge(data + from, mid - from + 1, data + mid + 1,
         to - mid);
}
```



Merge sort implementation (2/2)

```
void merge(int array_a[], int na, int array_b[], int nb) {
     int* array_c = new int[na + nb];
     int ia = 0, ib = 0;
     for (int ic = 0; ic < na + nb; ic++) {
          if (ia == na) { // a is empty.
               array_c[ic] = array_b[ib++];
               continue;
          }
          if (ib == nb) { // b is empty.
               array_c[ic] = array_a[ia++];
               continue;
          }
          if (array_a[ia] < array_b[ib]) {
               array_c[ic] = array_a[ia++];
          }
          else {
               array_c[ic] = array_b[ib++];
          }
     }
     for (int i = 0; i < na + nb; i++) {
          array_a[i] = array_c[i];
     }
     delete[] array_c;
}
```



Additional example

- See example at cs.usfca.edu/~galles/visualization/ComparisonSort.html

QUICK SORT

Quick sort

Quicksort Ω(n log(n)) Θ(n log(n)) O(n^2)

- Efficient algorithm, consumes little resources
- Tactics:
  - Determine the pivot element
  - Move elements smaller than pivot to the left, larger ones to the right. Now the pivot is in the final place, so we have a part with smaller elements and a part with bigger elements
    - We can put equals on one side or the other
  - Recursively make the previous steps on the part with smaller and on the part with bigger elements
- Recursive algorithm
  - Array with 0 or 1 element does is already sorted (base case)

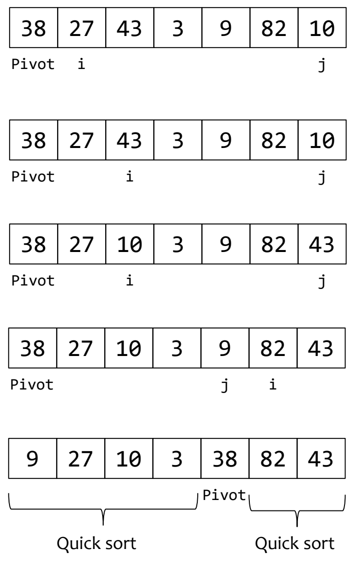
More detailed description (1/2)

- More detailed description:
  - The first element in the part we are sorting is declared a pivot
  - We start from the second element and go to the right looking for the first element larger than the pivot (i)
  - We start from the last element and go to the left looking for the first element less than or equal to the pivot (j)
  - If i < j, we swap these elements and again i goes to the right, j goes to the left
  - When i overtakes j, we swap the pivot with j
    - The pivot is in its final place now
    - All elements on the left are smaller or equal to the pivot
    - All elements on the right are larger than the pivot

More detailed description (2/2)

- We are now forming two arrays:
  - The first array consists of all the elements to the left of the pivot
  - The second array consists of all the elements to the right of the pivot
- In each array, we recursively make a quick sort
  - We stop when we get an array of size 0 or 1 because that array is already sorted

Quick sort example



Additional example

- See example on cs.usfca.edu/~galles/visualization/ComparisonSort.html

Quick sort implementation

```
void quick_sort(int* data, int left, int right) {
        if (right <= left) {
                return;
        }
        int& pivot = data[left];
        int i = left + 1;
        int j = right;
        while (i <= j && i <= right && j > left) {
                while (data[i] <= pivot && i <= right) { // Move i to right.
                        i++;
                }
                while (data[j] > pivot && j > left) { // Move j to left.
                        j--;
                }
                if (i < j) {
                        swap(data[i], data[j]);
                }
        }
        swap(pivot, data[j]);
        quick_sort(data, left, j - 1);
        quick_sort(data, j + 1, right);
}
```

Choice of a pivot

- Choosing a pivot element can significantly improve, but also spoil the efficiency of the quick sort algorithm
  - In the first (and our) versions, the leftmost element was taken
    - Problem: if the array is already sorted, we have the worst case
- There are several variants of pivot choice:
  - Take a random element
  - Take the middle element
  - Take the median (value in the middle) of the first, middle and last elements
    - The probability of the worst case being significantly reduced
      - All three elements should be among the largest or smallest in the array

HEAP SORT

Heap sort

Heapsort $\Omega(n \log(n))$ $\Theta(n \log(n))$ $O(n \log(n))$

- Very simple implementation:

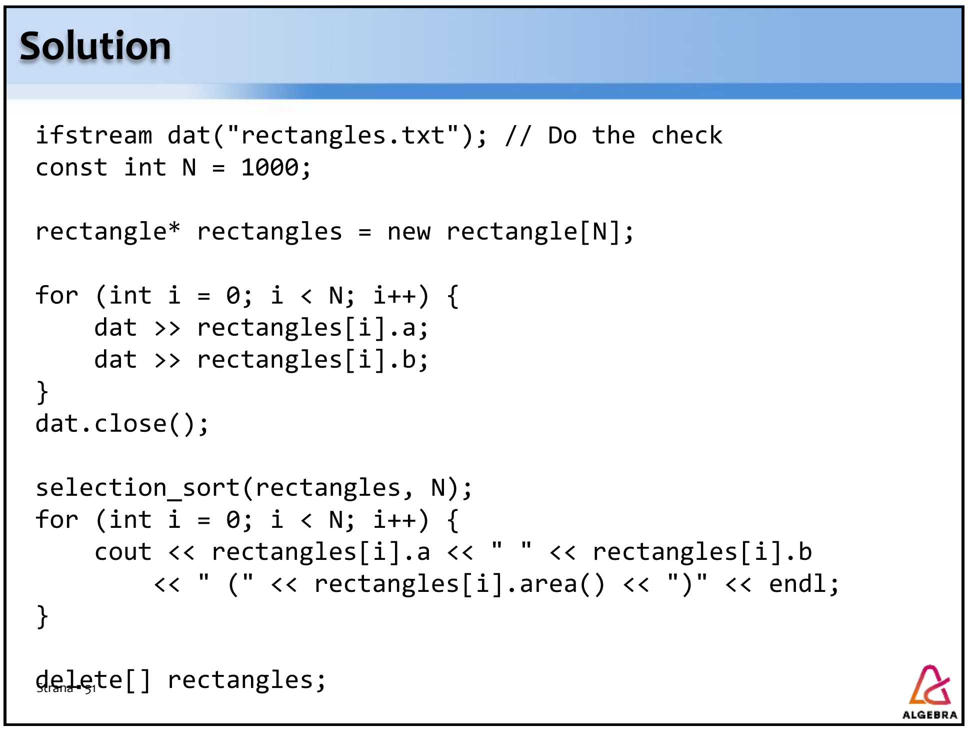
```
make_heap(data, data + n);
sort_heap(data, data + n);
```

Problem

1. Change selection sort so it sorts rectangles (width, height) descending based on their area. Load all 1000 rectangles from rectangles.txt and display them in descending order.

Solution

```
void selection_sort(rectangle data[], int n) {
  for (int i = 0; i < n - 1; i++) {
    int min_index = i;
    for (int j = i + 1; j < n; j++) {
      if (data[j].area() > data[min_index].area()) {
        min_index = j;
      }
    }
    swap(data[min_index], data[i]);
  }
}
```



Solution

```
ifstream dat("rectangles.txt"); // Do the check
const int N = 1000;

rectangle* rectangles = new rectangle[N];

for (int i = 0; i < N; i++) {
    dat >> rectangles[i].a;
    dat >> rectangles[i].b;
}
dat.close();

selection_sort(rectangles, N);
for (int i = 0; i < N; i++) {
    cout << rectangles[i].a << " " << rectangles[i].b
        << " (" << rectangles[i].area() << ")" << endl;
}

delete[] rectangles;
```

Strana • 51