Annotation

- Simple sorting algorithms,
- Units,
- Pointers.

Sorting – the motivation

- We have read the data,
- we want to process them in a monotone ordering.
- How to do that? Sort, process.
- Let us consider the data have been read into an array.

The problem of sorting – simple sorting algorithms

- BubbleSort,
- InsertSort,
- SelectSort,
- QuickSort.

Bubblesort

- Geometric interpretation:
  Bubbles in a liquid tend to ascend.

Bubblesort

- Geometric interpretation:
  Bubbles in a liquid tend to ascend.
- The idea: We are comparing pairs of consecutive numbers from the first pair to the last one. If they are incorrectly ordered, we swap their positions.

Bubblesort

- Geometric interpretation:
  Bubbles in a liquid tend to ascend.
- The idea: We are comparing pairs of consecutive numbers from the first pair to the last one. If they are incorrectly ordered, we swap their positions.
- Individual elements are "bubbling" in a correct direction.

Bubblesort

- Geometric interpretation:
  Bubbles in a liquid tend to ascend.
- The idea: We are comparing pairs of consecutive numbers from the first pair to the last one. If they are incorrectly ordered, we swap their positions.
- Individual elements are "bubbling" in a correct direction.
- We iterate this process until no swap takes place.

Bubblesort in pseudocode

```
weswapped:=true;
while weswapped do
begin
    for i:=1 to length - 1 do
    begin
        weswapped:=false;
        if numbers[i]>numbers[i+1] then
        begin swap(numbers[i],numbers[i+1]);
              weswapped:=true;
        end;
    end;
end;
```

Complexity of bubble-sort

- How many times we have to iterate the outer (`while`-)cycle?

Complexity of bubble-sort

- How many times we have to iterate the outer (`while`-)cycle?
- In the i-th iteration the i-th largest element reaches its position!

Complexity of bubble-sort

- How many times we have to iterate the outer (`while`-)cycle?
- In the i-th iteration the i-th largest element reaches its position!
- Thus it suffices to perform at most n iterations. Complexity of one iteration is also linear ($O(n)$).

Complexity of bubble-sort

- How many times we have to iterate the outer (`while`-)cycle?
- In the i-th iteration the i-th largest element reaches its position!
- Thus it suffices to perform at most n iterations. Complexity of one iteration is also linear ($O(n)$).
- Thus altogether $O(n^2)$.

Complexity of bubble-sort

- How many times we have to iterate the outer (`while`-)cycle?
- In the i-th iteration the i-th largest element reaches its position!
- Thus it suffices to perform at most n iterations. Complexity of one iteration is also linear ($O(n)$).
- Thus altogether $O(n^2)$.
- We may implement the algorithm when in odd iterations we bubble from left to right while in even iterations from right to left. This is called **Shakesort**. Its complexity is the same.

Insert- and Select-sort

Selectsort:

- Repeat until the array to sort is empty:
- Find a minimum in the array to sort and add it to the sorted array.

Insertsort:

- Repeat until the array to sort is empty:
- Take the first element of the array to sort and place it onto the correct position in the target array, i.e.:
  find the position where this element should be in the target array, add it there and the rest of the target array move one position further.

Complexity-analysis: We iterate the process $n$times. One iteration takes at most cn steps (for some constant c). Therefore altogether $O(n^2)$.

Quicksort

sorting using the recursion – the idea

- Sorting one-element-array is trivial (don't do anything, it is already sorted), i.e., just return the input sequence.

Quicksort

sorting using the recursion – the idea

- Sorting one-element-array is trivial (don't do anything, it is already sorted), i.e., just return the input sequence.
- In a nontrivial array A take a pivot p (element that we use for pivoting).

Quicksort

sorting using the recursion – the idea

- Sorting one-element-array is trivial (don't do anything, it is already sorted), i.e., just return the input sequence.
- In a nontrivial array A take a pivot p (element that we use for pivoting).
- Divide the array A into arrays B and C. B consists of the elements smaller than p, C consists of elements larger than p.

Quicksort

sorting using the recursion – the idea

- Sorting one-element-array is trivial (don't do anything, it is already sorted), i.e., just return the input sequence.
- In a nontrivial array A take a pivot p (element that we use for pivoting).
- Divide the array A into arrays B and C. B consists of the elements smaller than p, C consists of elements larger than p.
- Employ recursion on B, employ recursion on C

Quicksort

sorting using the recursion – the idea

- Sorting one-element-array is trivial (don't do anything, it is already sorted), i.e., just return the input sequence.
- In a nontrivial array A take a pivot p (element that we use for pivoting).
- Divide the array A into arrays B and C. B consists of the elements smaller than p, C consists of elements larger than p.
- Employ recursion on B, employ recursion on C
- Output the array B, output pivot p (as many times as it was in A), output C.

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.
- How long takes each "recursion-level"?

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.
- How long takes each "recursion-level"?
- Linearly w. r. t. n (because each element get operated constantly many times).

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.
- How long takes each "recursion-level"?
- Linearly w. r. t. n (because each element get operated constantly many times).
- Altogether, again, $O(n^2)$.

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.
- How long takes each "recursion-level"?
- Linearly w. r. t. n (because each element get operated constantly many times).
- Altogether, again, $O(n^2)$.
- The average-case complexity is $\Theta(n \log n)$ and the algorithm can be improved to gain this complexity by choosing pivot in a smarter way.

Quicksort

complexity analysis

- What's the complexity of the algorithm? How many times we could "employ the recursion"?
- Yes, n-times. If as a pivot we take the minimum, B is trivial and C is one element smaller than A.
- How long takes each "recursion-level"?
- Linearly w. r. t. n (because each element get operated constantly many times).
- Altogether, again, $O(n^2)$.
- The average-case complexity is $\Theta(n \log n)$ and the algorithm can be improved to gain this complexity by choosing pivot in a smarter way.
- To improve this algorithm we want to find a median - but we have to do it in linear time.

Method Divide et impera

alias Divide and conquer

- Already in ancient times (antiquity) it was known that if we divide enemies into several groups, we can gain control over them more easily.

Method Divide et impera

alias Divide and conquer

- Already in ancient times (antiquity) it was known that if we divide enemies into several groups, we can gain control over them more easily.
- Similar approach is used in the algorithm-design, just we divide the data.

Method Divide et impera

alias Divide and conquer

- Already in ancient times (antiquity) it was known that if we divide enemies into several groups, we can gain control over them more easily.
- Similar approach is used in the algorithm-design, just we divide the data.
- This method is specific by dividing the data in a fixed way, e.g., Quicksort.

Method Divide et impera

alias Divide and conquer

- Already in ancient times (antiquity) it was known that if we divide enemies into several groups, we can gain control over them more easily.
- Similar approach is used in the algorithm-design, just we divide the data.
- This method is specific by dividing the data in a fixed way, e.g., Quicksort.
- Technically we are designing recursive algorithms with complexity $T(n) = \sum_{i=1}^{k} T(n_i)$ where $\sum_{i=1}^{k} n_i = n$.

FIXME!!!

Here should be a quicksort implementation!

Units

how to compile parts of code separately

- Sometimes we implement functions usable in several projects (e.g., our sorting functions).

Units

how to compile parts of code separately

- Sometimes we implement functions usable in several projects (e.g., our sorting functions).
- We may copy (click'n'paste) them into the other source files (bad idea)

Units

how to compile parts of code separately

- Sometimes we implement functions usable in several projects (e.g., our sorting functions).
- We may copy (click'n'paste) them into the other source files (bad idea)
- or we store them into a separate file that gets compiled separately.

Units

how to compile parts of code separately

- Sometimes we implement functions usable in several projects (e.g., our sorting functions).
- We may copy (click'n'paste) them into the other source files (bad idea)
- or we store them into a separate file that gets compiled separately.
- The latter approach is referred as the **units.**

Units – advantages and disadvantages

- Source code gets spreaded into several files,

Units – advantages and disadvantages

- Source code gets spreaded into several files,
- it is not necessary to write the code more times when we want to share it in several projects.

Units – syntax and semantic

- Instead of word `program` we start with keyword `unit`,
- again, we place the name of the unit. This time the name must correspond with the filename. Also the keyword `unit` is compulsory.
- A unit consists of an `interface` (what's visible from the outside)
- and of `implementation` (internal part where the interface is implemented).

Units – the interface part

- Interface describes publicly visible part of the unit.

Units – the interface part

- Interface describes publicly visible part of the unit.
- Interface consists of:

Units – the interface part

- Interface describes publicly visible part of the unit.
- Interface consists of:
- variable definitions (when the variables should be publicly visible),

Units – the interface part

- Interface describes publicly visible part of the unit.
- Interface consists of:
- variable definitions (when the variables should be publicly visible),
- function (and proc.) prototypes (when the function should be publicly visible),

Units – the interface part

- Interface describes publicly visible part of the unit.
- Interface consists of:
- variable definitions (when the variables should be publicly visible),
- function (and proc.) prototypes (when the function should be publicly visible),
- prototype is the header of the function, i.e., the "first line".

Units – impelementation

- What should *not* be publicly visible, i.e.:

Units – impelementation

- What should *not* be publicly visible, i.e.:
- Function definitions,

Units – impelementation

- What should *not* be publicly visible, i.e.:
- Function definitions,
- variable definitions (for internal variables of the unit),

Units – impelementation

- What should *not* be publicly visible, i.e.:
- Function definitions,
- variable definitions (for internal variables of the unit),
- definition of any stuff that should be (publicly) invisible,

Units – impelementation

- What should *not* be publicly visible, i.e.:
- Function definitions,
- variable definitions (for internal variables of the unit),
- definition of any stuff that should be (publicly) invisible,
- definition of internal functions (not mentioned in interface).

Units – impelementation

- What should *not* be publicly visible, i.e.:
- Function definitions,
- variable definitions (for internal variables of the unit),
- definition of any stuff that should be (publicly) invisible,
- definition of internal functions (not mentioned in interface).
- We finish the unit by keyword `end.` (followed by full-stop)

Units – example

```
unit sorting;
interface
      type po=array[0..9] of integer;
      procedure bubble(var arr:array of integer);
      procedure select(var a:po);
      procedure insert(var a:po);
      procedure quicksort(var arr:array of
integer;number:integer);
      procedure output(a:array of integer);
```

Units – example (cont.)

```
...
implementation
      var inserted:integer;
      procedure bubble(var arr:array of integer);
      ...
      function extract_min(var a:po):integer;
      {This function will not be visible from
outside!}
      ...
      procedure select(var a:po):integer;
      ...
      ...
end.
```

Units – how to use them

- When using a unit, we announe it with a keyword `uses` followed by the name of the unit:
- Example: `uses sorting;`

Using the unit – example

```
program sort;
uses sorting;
var p:array [0..9] of integer;
     i:integer;
begin
     for i:=0 to 9 do
          read(p[i]);
     quicksort(p,1,10);
     output(p);
end.
```