Overview

- Long numbers
- Gallery of sorting algorithms
- Median in a linear time
- Graphs and their representation,
- Graph-algorithms.

Long numbers

are when

- we have a number that does not fit into any numerical data-type.

Long numbers

are when

- we have a number that does not fit into any numerical data-type.
- We have to implement it ourselves.

Long numbers

are when

- we have a number that does not fit into any numerical data-type.
- We have to implement it ourselves.
- We may use arrays (where possible), or dynamic data-structures.

Long numbers

are when

- we have a number that does not fit into any numerical data-type.
- We have to implement it ourselves.
- We may use arrays (where possible), or dynamic data-structures.
- We want to implement mainly addition, subtraction, multiplication and division.

Long numbers

are when

- we have a number that does not fit into any numerical data-type.
- We have to implement it ourselves.
- We may use arrays (where possible), or dynamic data-structures.
- We want to implement mainly addition, subtraction, multiplication and division.
- Problems with non-integer numbers and with negative numbers!

Implementing long numbers

- We focus on dynamic structures.

Implementing long numbers

- We focus on dynamic structures.
- We can use a bidirectional list.

Implementing long numbers

- We focus on dynamic structures.
- We can use a bidirectional list.
- With one-directional list we would have to decide on its endian-ness!

Details

Shown on the black-board.

Heap

- A heap is a tree where for each vertex (node) it holds that the value of the parent is at most as large as the value of (all) sons.

Heap

- A heap is a tree where for each vertex (node) it holds that the value of the parent is at most as large as the value of (all) sons.
- A binary leftist heap is a binary tree that fulfils the heap-property in such that in all the levels except of the bottom-most two all vertices have two sons. In the bottom most level no vertces hav any sons. And in the second-to-last level, there exists at most one entry with exactly one son, and this son is to the left. Moreover, all the vertices to the left from this entry have two sons, and the vertices to the right have none.

Heap

- A heap is a tree where for each vertex (node) it holds that the value of the parent is at most as large as the value of (all) sons.
- A binary leftist heap is a binary tree that fulfils the heap-property in such that in all the levels except of the bottom-most two all vertices have two sons. In the bottom most level no vertces hav any sons. And in the second-to-last level, there exists at most one entry with exactly one son, and this son is to the left. Moreover, all the vertices to the left from this entry have two sons, and the vertices to the right have none.
- Idea: The heap is being filled from the top and the depth increases only if all the previous levels are already full. The bottom-most level is filled "from the left to the right".

Heap

- A heap is a tree where for each vertex (node) it holds that the value of the parent is at most as large as the value of (all) sons.
- A binary leftist heap is a binary tree that fulfils the heap-property in such that in all the levels except of the bottom-most two all vertices have two sons. In the bottom most level no vertces hav any sons. And in the second-to-last level, there exists at most one entry with exactly one son, and this son is to the left. Moreover, all the vertices to the left from this entry have two sons, and the vertices to the right have none.
- Idea: The heap is being filled from the top and the depth increases only if all the previous levels are already full. The bottom-most level is filled "from the left to the right".
- With a heap we use the operations makeheap and

Heapsort

- Heapsort can be described very easily:

Heapsort

- Heapsort can be described very easily:
- `H:=Makeheap;`

Heapsort

- Heapsort can be described very easily:
- `H:=Makeheap;`
- `for i:=1 to n do output(extractmin(H))`

Heapsort

- Heapsort can be described very easily:
- `H:=Makeheap;`
- `for i:=1 to n do output(extractmin(H))`
- Now, how can one implement those functions?

Implementing `Makeheap`

- Naive approach: Insert one element after another.

Implementing `Makeheap`

- Naive approach: Insert one element after another.
- Use function `bubble_upwards` for heap-consolidation.

Implementing `Makeheap`

- Naive approach: Insert one element after another.
- Use function `bubble_upwards` for heap-consolidation.
- Complexity is $\Theta(n \log n)$.

Makeheap in linear time

- Tarjan's algorithm: Building from the bottom to the top!

Makeheap in linear time

- Tarjan's algorithm: Building from the bottom to the top!
- The bottom-most levels already form a set of heaps. In each step we add a new vertex on the next higher level that combines two of these heaps. The heap property can only be violated in the root node.

Makeheap in linear time

- Tarjan's algorithm: Building from the bottom to the top!
- The bottom-most levels already form a set of heaps. In each step we add a new vertex on the next higher level that combines two of these heaps. The heap property can only be violated in the root node.
- So for the added element we call the function `bubble_down`.

Makeheap in linear time

- Tarjan's algorithm: Building from the bottom to the top!
- The bottom-most levels already form a set of heaps. In each step we add a new vertex on the next higher level that combines two of these heaps. The heap property can only be violated in the root node.
- So for the added element we call the function `bubble_down`.
- Complexity:
$$\sum_{i=1}^{\log n}(i-1)\left(\frac{n}{2^i}\right) \leq \frac{n}{2}\cdot\sum_{i=0}^{\infty} i\cdot\left(\frac{1}{2^i}\right) = \frac{n}{2}\cdot\sum_{i=0}^{\infty}\sum_{j=i}^{\infty}\frac{1}{2^j} = \frac{n}{2}\sum_{i=0}^{\infty}\frac{1}{2^{i-1}} = 2n,$$
i.e., linear!

Implementing `extractmin`

- Observation: Minimum is in the root.

Implementing `extractmin`

- Observation: Minimum is in the root.
- Remove the root and replace by...

Implementing `extractmin`

- Observation: Minimum is in the root.
- Remove the root and replace by...
- ... the last element in the heap!

Implementing `extractmin`

- Observation: Minimum is in the root.
- Remove the root and replace by...
- ... the last element in the heap!
- Consolidate the heap, i.e., call `bubble_down`

Implementing `extractmin`

- Observation: Minimum is in the root.
- Remove the root and replace by...
- ... the last element in the heap!
- Consolidate the heap, i.e., call `bubble_down`
- Complexity: n-times call `bubble_down`, i.e., a function with the complexity bounded by the depth of the heap, i.e., altogether $O(n \log n)$.

How to represent a heap?

- We could use dynamic data structures, but an array is much more efficient:

How to represent a heap?

- We could use dynamic data structures, but an array is much more efficient:
- The root gets mapped onto position 1,

How to represent a heap?

- We could use dynamic data structures, but an array is much more efficient:
- The root gets mapped onto position 1,
- sons of the ith element appear on positions $2i$ and $2i + 1$.

How to represent a heap?

- We could use dynamic data structures, but an array is much more efficient:
- The root gets mapped onto position 1,
- sons of the ith element appear on positions $2i$ and $2i+1$.
- In this way we fill the array, root is at the beginning, the last element at the end.

How to represent a heap?

- We could use dynamic data structures, but an array is much more efficient:
- The root gets mapped onto position 1,
- sons of the ith element appear on positions $2i$ and $2i+1$.
- In this way we fill the array, root is at the beginning, the last element at the end.
- This representation is very efficient.

Mergesort

- Another algorithm from the family of *divide et impera* techniques.

Mergesort

- Another algorithm from the family of *divide et impera* techniques.
- This time we do not care for the values, we always divide into halves.

Mergesort

- Another algorithm from the family of *divide et impera* techniques.
- This time we do not care for the values, we always divide into halves.
- Recursively we may obtain two sorted sequences and we have to merge them.

Mergesort

- Another algorithm from the family of *divide et impera* techniques.
- This time we do not care for the values, we always divide into halves.
- Recursively we may obtain two sorted sequences and we have to merge them.
- Merging can be performed in time linear w.r.t. length of the input.

Mergesort

- Another algorithm from the family of *divide et impera* techniques.
- This time we do not care for the values, we always divide into halves.
- Recursively we may obtain two sorted sequences and we have to merge them.
- Merging can be performed in time linear w.r.t. length of the input.
- When implementing Mergesort, we do not employ recursion, we rewrite it into several cycles.

Complexity-analysis

- We proceed by summing over the "levels", i.e., how long does it take to merge sequences of length k into sequences of length $2k$?

Complexity-analysis

- We proceed by summing over the "levels", i.e., how long does it take to merge sequences of length k into sequences of length $2k$?
- Yes, linearly. How many lengths are needed (i.e., how many levels take place)?

Complexity-analysis

- We proceed by summing over the "levels", i.e., how long does it take to merge sequences of length k into sequences of length $2k$?
- Yes, linearly. How many lengths are needed (i.e., how many levels take place)?
- Logarithmically (between two levels we double the length of the sorted sequence).

Complexity-analysis

- We proceed by summing over the "levels", i.e., how long does it take to merge sequences of length k into sequences of length $2k$?
- Yes, linearly. How many lengths are needed (i.e., how many levels take place)?
- Logarithmically (between two levels we double the length of the sorted sequence).
- Thus altogether: $O(n \log n)$.

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.
- It is fast but relatively small.

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.
- It is fast but relatively small.
- External memory is a hard-disk or good old tape.

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.
- It is fast but relatively small.
- External memory is a hard-disk or good old tape.
- It is large but slow. Here, it is an advantage when we read as long block as possible (and we do not skip too much).

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.
- It is fast but relatively small.
- External memory is a hard-disk or good old tape.
- It is large but slow. Here, it is an advantage when we read as long block as possible (and we do not skip too much).
- Usually we are sorting huge amounts of data that do not necessarily fit into the internal memory.

Sorting in external memory

out of core sorting

- Internal memory is usually on-chip RAM.
- It is fast but relatively small.
- External memory is a hard-disk or good old tape.
- It is large but slow. Here, it is an advantage when we read as long block as possible (and we do not skip too much).
- Usually we are sorting huge amounts of data that do not necessarily fit into the internal memory.
- For sorting data in external memory, a mergesort can be performed.

Sorting in external memory

improving the algorithm

- First we perform a heapsort in internal memory to obtain a sorted block that can be written to external memory.

Sorting in external memory

improving the algorithm

- First we perform a heapsort in internal memory to obtain a sorted block that can be written to external memory.
- We modify this algorithm so that we do not just build the heap and extract by units, but if the next sorted element is large enough, we add it to the current heap.

Sorting in external memory

improving the algorithm

- First we perform a heapsort in internal memory to obtain a sorted block that can be written to external memory.
- We modify this algorithm so that we do not just build the heap and extract by units, but if the next sorted element is large enough, we add it to the current heap.
- Large enough means at least as large as was the last element we wrote.

Sorting in external memory

improving the algorithm

- First we perform a heapsort in internal memory to obtain a sorted block that can be written to external memory.
- We modify this algorithm so that we do not just build the heap and extract by units, but if the next sorted element is large enough, we add it to the current heap.
- Large enough means at least as large as was the last element we wrote.
- If the element is too small (smaller than the last element), we finish this heap and start a new one.

Sorting in external memory

improving the algorithm

- First we perform a heapsort in internal memory to obtain a sorted block that can be written to external memory.
- We modify this algorithm so that we do not just build the heap and extract by units, but if the next sorted element is large enough, we add it to the current heap.
- Large enough means at least as large as was the last element we wrote.
- If the element is too small (smaller than the last element), we finish this heap and start a new one.
- Next we perform mergesort. Note that repeating heapsort brings no improvement!

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.
- How to distribute the blocks onto tapes? It is an advantage when we merge from all (remaining) tapes onto one.

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.
- How to distribute the blocks onto tapes? It is an advantage when we merge from all (remaining) tapes onto one.
- By generalized Fibonacci numbers. For 3 tapes:

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.
- How to distribute the blocks onto tapes? It is an advantage when we merge from all (remaining) tapes onto one.
- By generalized Fibonacci numbers. For 3 tapes:
- Last configuration $(0,0,1)$, previous $(1,1,0),(2,0,1),(0,2,3)$, i.e.,

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.
- How to distribute the blocks onto tapes? It is an advantage when we merge from all (remaining) tapes onto one.
- By generalized Fibonacci numbers. For 3 tapes:
- Last configuration $(0, 0, 1)$, previous $(1, 1, 0), (2, 0, 1), (0, 2, 3)$, i.e.,
- $(0, f_{i-2}, f_{i-3}), (f_{i-2}, f_{i-1}, 0), (f_i, 0, f_{i-1}), (0, f_i, f_{i+1})$.

Sorting on tapes

- We have k tapes and we want to perform a mergesort on them.
- How to distribute the blocks onto tapes? It is an advantage when we merge from all (remaining) tapes onto one.
- By generalized Fibonacci numbers. For 3 tapes:
- Last configuration $(0, 0, 1)$, previous $(1, 1, 0), (2, 0, 1), (0, 2, 3)$, i.e.,
- $(0, f_{i-2}, f_{i-3}), (f_{i-2}, f_{i-1}, 0), (f_i, 0, f_{i-1}), (0, f_i, f_{i+1})$.
- With more tapes, generalized Fibonacci numbers appear.

Complexity of the problem

sorting by comparison

- Let us recall that the complexity of a problem is the complexity of the optimal algorithm solving this problem.

Complexity of the problem

sorting by comparison

- Let us recall that the complexity of a problem is the complexity of the optimal algorithm solving this problem.
- We have seen several algorithms for sorting, several of them with complexity $O(n \log n)$. Can we do better or is there some obstruction?

Complexity of the problem

sorting by comparison

- Let us recall that the complexity of a problem is the complexity of the optimal algorithm solving this problem.
- We have seen several algorithms for sorting, several of them with complexity $O(n \log n)$. Can we do better or is there some obstruction?
- If we can gain information about the input only by comparison...

Complexity of the problem

sorting by comparison

- Let us recall that the complexity of a problem is the complexity of the optimal algorithm solving this problem.
- We have seen several algorithms for sorting, several of them with complexity $O(n \log n)$. Can we do better or is there some obstruction?
- If we can gain information about the input only by comparison...
- ... then we cannot do better.

Lower bound

- For any deterministic sorting algorithm we can define a decision-tree.

Lower bound

- For any deterministic sorting algorithm we can define a decision-tree.
- The tree describes the comparisons (as the algorithm can only decide based on these comparisons). A leaf means that we know how to permute the input.

Lower bound

- For any deterministic sorting algorithm we can define a decision-tree.
- The tree describes the comparisons (as the algorithm can only decide based on these comparisons). A leaf means that we know how to permute the input.
- We calculate the depth of this tree, i.e., depth of the deepest leaf.

Lower bound

- For any deterministic sorting algorithm we can define a decision-tree.
- The tree describes the comparisons (as the algorithm can only decide based on these comparisons). A leaf means that we know how to permute the input.
- We calculate the depth of this tree, i.e., depth of the deepest leaf.
- How many leaves must there be?

Lower bound – cont.

- At least one for each input permutation (different permutations have different inverse-permutations), i.e., altogether at least $n!$ leaves.
- The decision-tree is a binary tree with $n!$ leaves. What's the minimum depth?
- $\log n! \geq \log(n^{\frac{n}{2}}) = \frac{n}{2} \log n$, i.e., $\Omega(n \log n)$.

Another proof

- Different permutations require different computations.

Another proof

- Different permutations require different computations.
- How many computations are there with $o(n \log n)$ comparisons?

Another proof

- Different permutations require different computations.
- How many computations are there with $o(n \log n)$ comparisons?
- $o(2^{n \log n})$, i.e., $o(n!)$ – and it does not suffice to sort even all possible permutations!

Another proof

- Different permutations require different computations.
- How many computations are there with $o(n \log n)$ comparisons?
- $o(2^{n \log n})$, i.e., $o(n!)$ – and it does not suffice to sort even all possible permutations!
- Among other, we showed the lower bound for Mergesort, Heapsort and even for A-sort as they belong to the family of algorithms sorting by comparison.

Bucketsort

sorting in a linear time

- Considering that the input consists of integers in a decimal notation, we may split the values into "buckets" based on a value in a particular position.

Bucketsort

sorting in a linear time

- Considering that the input consists of integers in a decimal notation, we may split the values into "buckets" based on a value in a particular position.
- Bucketsort starts this splitting based on the last digit. Then we pick the content of individual buckets, put them one after another (in a correct ordering) and continue splitting based on the previous digit up to the beginning of the (longest) numbers.

Bucketsort

sorting in a linear time

- Considering that the input consists of integers in a decimal notation, we may split the values into "buckets" based on a value in a particular position.
- Bucketsort starts this splitting based on the last digit. Then we pick the content of individual buckets, put them one after another (in a correct ordering) and continue splitting based on the previous digit up to the beginning of the (longest) numbers.
- Details on the blackboard.

Bucketsort

complexity analysis

- Considering the length of numbers is constant, what is the complexity?

Bucketsort

complexity analysis

- Considering the length of numbers is constant, what is the complexity?
- $c \times n$ where c is a constant (depending on the length of numbers).

Bucketsort

complexity analysis

- Considering the length of numbers is constant, what is the complexity?
- $c \times n$ where c is a constant (depending on the length of numbers).
- We are sorting in a linear time. How is it possible (w.r.t. the lower bound)?

Median in a linear time

motivation

- Quicksort was dividing the input based on a pivot.

Median in a linear time

motivation

- Quicksort was dividing the input based on a pivot.
- The complexity could be at most quadratic. Can we do better?

Median in a linear time

motivation

- Quicksort was dividing the input based on a pivot.
- The complexity could be at most quadratic. Can we do better?
- Yes, if we manage to find a reasonable pivot in a linear time.

Median in a linear time

motivation

- Quicksort was dividing the input based on a pivot.
- The complexity could be at most quadratic. Can we do better?
- Yes, if we manage to find a reasonable pivot in a linear time.
- We design an algorithm looking for the kth largest value (among n values) in time linear w.r.t. n.

Median in a linear time

the algorithm itself

- We divide the input into 5-tuples.

Median in a linear time

the algorithm itself

- We divide the input into 5-tuples.
- We find a median of each 5-tuple.

Median in a linear time

the algorithm itself

- We divide the input into 5-tuples.
- We find a median of each 5-tuple.
- We find the median of these medians.

Median in a linear time

the algorithm itself

- We divide the input into 5-tuples.
- We find a median of each 5-tuple.
- We find the median of these medians.
- Divide the input onto pile of smaller values and larger values.

Median in a linear time

the algorithm itself

- We divide the input into 5-tuples.
- We find a median of each 5-tuple.
- We find the median of these medians.
- Divide the input onto pile of smaller values and larger values.
- (Recursively) continue searching in the appropriate pile.

Median in a linear time

the details

- How to find the medians of 5-tuples?

Median in a linear time

the details

- How to find the medians of 5-tuples?
- By brute force (because 5 is a constant).

Median in a linear time

the details

- How to find the medians of 5-tuples?
- By brute force (because 5 is a constant).
- How to find the median of medians?

Median in a linear time

the details

- How to find the medians of 5-tuples?
- By brute force (because 5 is a constant).
- How to find the median of medians?
- Recursively (we are the function finding median in a linear time) unless there are at most five 5-tuples (i.e., 5 candidates for median of medians).

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.
- Complexity is: $T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.
- Complexity is: $T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
- We need to show: $\exists_d$ s. t. $T(n) \leq dn$. This gets verified by induction.

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.
- Complexity is: $T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
- We need to show: $\exists_d$ s. t. $T(n) \leq dn$. This gets verified by induction.
- Natural questions: Why 5-tuples? How about 3-tuples or 7-tuples?

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.
- Complexity is: $T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
- We need to show: $\exists_d$ s. t. $T(n) \leq dn$. This gets verified by induction.
- Natural questions: Why 5-tuples? How about 3-tuples or 7-tuples?
- The latter works, the former not.

Complexity analysis

- Why is the algorithm linear? Because the size of the "pile" reduces rapidly:
- The median of medians used as a pivot creates each new "pile" of size at most $\frac{7}{10}n$ (provided that the input is of size n). Details on the blackboard.
- Complexity is: $T(n) = cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
- We need to show: $\exists_d$ s. t. $T(n) \leq dn$. This gets verified by induction.
- Natural questions: Why 5-tuples? How about 3-tuples or 7-tuples?
- The latter works, the former not.
- Conclusion: Quicksort employing median in linear time has complexity $\Theta(n \log n)$.

Graph – definition

Definition

A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Graph – definition

Definition

A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

Graph – definition

Definition

A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are discussed in Discrete mathematics, so you already know some algorithms. Now we want to implement them.

Graph – definition

Definition

A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are discussed in Discrete mathematics, so you already know some algorithms. Now we want to implement them.
- The definition of a graph is nice, but it does not help much with programming. So we have to ask:

Graph – definition

Definition

A graph is an ordered pair $G = (V, E)$ where V is a set of vertices and $E \subseteq \binom{V}{2}$ is a set of edges.

Definition

An ordered pair $G = (V, E)$ is called an oriented graph with a vertex set V and an edge set E, if $E \subseteq V \times V$.

- Graphs are discussed in Discrete mathematics, so you already know some algorithms. Now we want to implement them.
- The definition of a graph is nice, but it does not help much with programming. So we have to ask:
- How do we represent a graph while programming?

Graphs – remarks to definition

- Goal: Advantages and disadvantages of individual representations.

Graphs – remarks to definition

- Goal: Advantages and disadvantages of individual representations.
- Define basic notions (walk, trail, path, connectivity, trees).

Graph Representation

- From lectures of Discrete Mathematics you know:

Graph Representation

- From lectures of Discrete Mathematics you know:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).

Graph Representation

- From lectures of Discrete Mathematics you know:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.

Graph Representation

- From lectures of Discrete Mathematics you know:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.
- Advantages and disadvantages?

Graph Representation

- From lectures of Discrete Mathematics you know:
- Adjacency matrix A_G
  – is a square 0/1-matrix whose rows and columns are indexed by individual vertices. One corresponds to an edge between appropriate vertices (zero means no edge there).
- Incidence matrix B_G – rows index by vertices, columns by edges, one in $B[i,j]$ means that the edge j is incident with the vertex i.
- Advantages and disadvantages?
- Can we convert these representations?

Converting A_G to B_G and back

```
init_with_0s(B_G);
edge_index:=1;
for i:=1 to n do begin
      for j:=i+1 to n do begin
      if(A_G[i,j]=1) then
      begin
            B_G[i,edge_index]:=1;
            B_G[j,edge_index]:=1;
            inc(edge_index);
      end;
end;
```

B_G to A_G

Either we analyze the incidence matrix (in a similar way) or:

$A_G := B_G \times {B_G}^T$;

```
for i:=1 to n do
```

$\qquad A_G[i, i] := 0$;

Důkaz.

Exercise in Combinatorics and Graph Theory I $\square$

Further graph representations

- List of vertices and edges incident to individual vertices.

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).
- Advantages/disadvantages?

Further graph representations

- List of vertices and edges incident to individual vertices.
- I.e., we are keeping a list of edges incident to each vertex in the graph.
- We employ linked-lists. If we employ an array, what do we get?
- The adjacency matrix!

Functions necessary/sufficient to work with a graph:

- `find_neighbors(v)`,
- `vertices`,
- `edges` or `edge(u,v)` – we can find it through `vertices` and `find_neighbors`,
- further, e.g., (`vertex_weight(v), edge_weight(e)...`).
- Advantages/disadvantages?
- In oriented case we have to modify the representation.

Walk, trail, path, circle

Definition

Walk, trail, path, circle

Definition

- Walk of length k s a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.

Walk, trail, path, circle

Definition

- Walk of length k s a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- Trail is a walk where each edge occurs at most once.

Walk, trail, path, circle

Definition

- Walk of length k s a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- Trail is a walk where each edge occurs at most once.
- Path is a trail (or a walk) where each vertex occurs only once (i.e., each vertex is incident to two consecutive edges).

Walk, trail, path, circle

Definition

- Walk of length k s a sequence of edges $\{v_0, v_1\}, \{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{k-1}, v_k\}$.
- Trail is a walk where each edge occurs at most once.
- Path is a trail (or a walk) where each vertex occurs only once (i.e., each vertex is incident to two consecutive edges).
- A trail is a circle if it starts and ends in the same vertex and each vertex occurs there exactly once.

Connectivity, tree

Definition

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.
- How do we decide whether a graph is a tree?

Connectivity, tree

Definition

- Graph is connected if from any its vertex we can reach any other vertex.
- Graph is a tree if it is connected and it contains no circles.

- Definitions are fine, but how we use them while programming?
- How do we verify that a graph is connected?
- We use a suitable claim.
- How do we decide whether a graph is a tree?
- Similarly!

Graph connectivity

Graph is connected iff from one (fixed) vertex we can reach all the other vertices.

```
for i in vertices do
      unvisit(i); {so far we visited nothing}
i:=start_vertex;
queue:={i};{for reachable vertices}
while nonempty(queue) do begin
      visit(i);
      queue:=queue+unvisited_neighbors(i);
end;
connected:=true;
for i in vertices do begin
      if unvisited(i) then
            connected:=false;
```

Algorithm analysis

- for-cycle takes place at most n-times.

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,
- considering the incidence matrix, complexity may be $\Theta(mn^2)$.

Algorithm analysis

- for-cycle takes place at most n-times.
- while-cycle passes for each vertex at most once and inspects the neighbors of this vertex.
- Complexity depends on the representation (how quickly we find the neighbors of a given vertex).
- Complexity is $\Omega(m)$ (Each edge must be inspected).
- Considering a list of edges incident to each vertex, complexity is $O(m)$,
- in the adjacency matrix, complexity is $O(n^2)$,
- considering the incidence matrix, complexity may be $\Theta(mn^2)$.
- Thus a good representation yields the complexity $\Theta(m+n)$.

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:
- DFS may get implemented using recursion (and thus without an auxiliary data-structure).

Remarks

- If we use a queue, we are implementing the wave-algorithm (BFS).
- We may use a buffer and get a DFS.
- Advantages/disadvantages:
- DFS may get implemented using recursion (and thus without an auxiliary data-structure).
- BFS visits the vertex using the shortest path.

Looking for a cycle

A graph has a cycle if we return to a particular vertex while searching the graph.

```
cycle:=false; {so far no cycle}
for i in vertices do unvisit(i);
for i in vertices do
    if unvisited(i) then{new component}
    begin queue:={i};
        while(nonempty(queue)) do
        begin dequeue_from_queue_and_assing_into(i);
            if(visited(i)) then
                cycle:=true;
            else  for j in neighbors(i) do
                begin queue:=queue+{j};
                    erase_edge({i,j});
                end;
    end; end;
```

Tree

- We may test whether the graph is connected without cycle (use previous algorithms).

Tree

- We may test whether the graph is connected without cycle (use previous algorithms).
- Or we test cycle-freeness and connectivity (one component).

Tree

- We may test whether the graph is connected without cycle (use previous algorithms).
- Or we test cycle-freeness and connectivity (one component).
- Or we test connectivity (or cycle-freeness) and an appropriate number of edges (Euler's formular).

Shortest path

When looking for the shortest path, it depends on the representation:

- Perform BFS (considering the list of vertices and edges),

Theorem

In ${A_G}^k$ position i, j gives number of walks with length k from (vertex) i to j.

Corollary

In $(A_G + I)^k$ position i, j says the number of walks of length at most k from i to j.

Shortest path

When looking for the shortest path, it depends on the representation:

- Perform BFS (considering the list of vertices and edges),
- make the power of adjacency-matrix using matrix-representation.

Theorem

In ${A_G}^k$ position i, j gives number of walks with length k from (vertex) i to j.

Corollary

In $(A_G + I)^k$ position i, j says the number of walks of length at most k from i to j.

Dijkstra's algorithm

Looks for the shortest path from a given vertex into all other vertices

Input: Graph with `nonnegatively` evaluated edges.

- We keep the "queue" for vertices ordered by the shortest so far found path.
- At the beginning we inicialize the distances to all vertices [except start] by infinity [large-enough value], distance to start is 0.
- We add `start` into the queue for reachable vertices.
- Remove the first vertex of the "queue" and inspect its neighbors.
- Repeat this while the "queue" is non-empty.

Extending the path

When extending the path, for a vertex v in distance $d(v)$ we try for each edge $\{v, w\}$ whether

$$d(w) > d(v) + length(\{v, w\}).$$

If so, let $d(w) := d(v) + length(\{v, w\})$ and correct the position of w in the "queue".

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.
- The algorithm is kind of modification of BFS!

Analysis

- The algorithm is proven in Invitation to Discrete Mathematics (and many other books).
- Finiteness: Each iteration removes one vertex from the "queue". This vertex never appears there anew (as all the edge-lengths are non-negative).
- Partial correctness uses the invariant:
  In each iteration we have the shortes paths using only the vertices that were already removed from the "queue".
- Invariant shows the correctness.
- The algorithm is kind of modification of BFS!
- Complexity depends on the representation of the graph and of the 'queue'!

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Thésseus and Minotaurus,

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Thésseus and Minotaurus,
- King (and other garbage) on chessboard with some squares prohibited,

Remarks

- If the graph is unweighted, Dijkstra's algorithm collapses into BFS!
- Applications of graph algorithms – the whole theoretical computer science.
- Examples of graph-algorithms:
- Thésseus and Minotaurus,
- King (and other garbage) on chessboard with some squares prohibited,
- ...

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.
- Edge chromaticity – we color edges and no pair of edges incident with a common vertex has the same color.

Further graph-optimization problems (algorithms)

- Minimum spanning tree – a.k.a. how to create the electric power lines.
- Eulerian graph, i.e., can we draw all the edges of the graph by one trail (without passing any edge twice)?
- Hamiltonicity – a circle that visits all the vertices (each exactly once),
- Clique – maximum complete subgraph,
- Chromaticity – minimum number of colors s. t. we can color all vertices in such a way that neighboring vertices have distinct colors.
- Edge chromaticity – we color edges and no pair of edges incident with a common vertex has the same color.
- ...

End

Thank you for your attention...