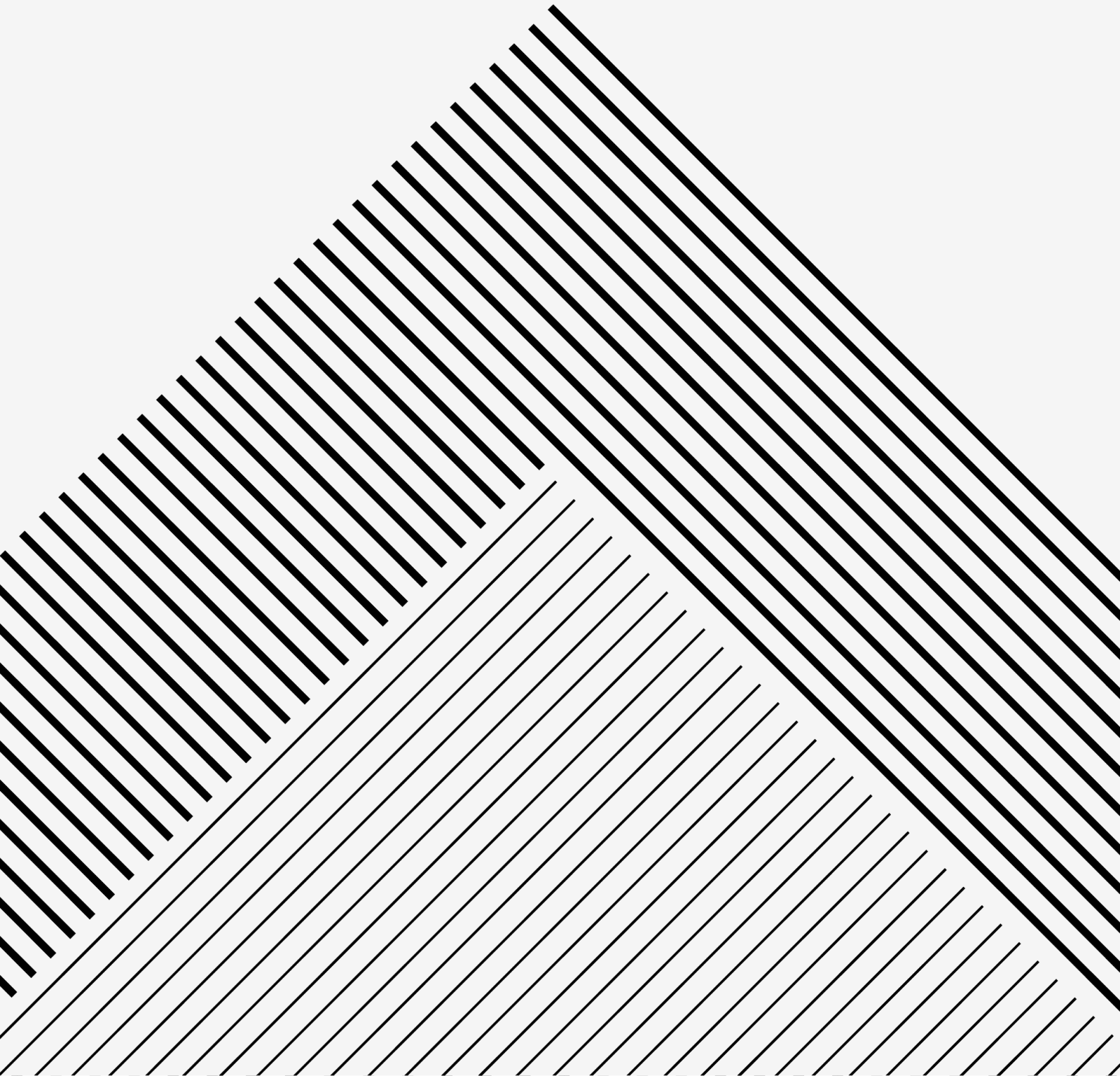
# AlgoDat

# ZSF



# AlgoDat - ZSF

## Checklist

### Algorithms, Problems and Datastructures

- [x] Karatsuba's Algorithm
- [ ] Maximum Subarray
- [x] Binary Search
- [ ] Interpolationssuche
- [x] Linear Search
- [x] Is Sorted
- [x] Bubble Sort
- [x] Selection Sort
- [x] Insertion Sort
- [x] Heap Sort
- [x] Merge Sort
- [x] Quick Sort
- [x] Längste aufsteigende Teilfolge
- [x] Längste gemeinsame Teilfolge
- [x] Minimale Editierdistanz
- [x] Subset Sum
- [x] Knapsack Problem
- [x] Stack
- [x] Queue
- [x] Priority Queue
- [x] Search Tree
- [x] AVL-Tree
- [ ] Königsberger Brücken
- [ ] Euler Walk
- [x] DFS (-Visit)
- [x] Adjazenzmatrix
- [x] Adjazenzliste
- [x] BFS
- [ ] Ford's Algorithmus
- [x] Bellman-Ford Algorithmus
- [x] Dijkstra's Algorithmus
- [x] Boruvka's Algorithmus
- [x] Prim's Algorithmus
- [x] Kruskal's Algorithmus
- [x] Floyd-Warshall
- [x] Johnson's Algorithmus
- [x] Selection Problem
- [x] All-Pair Shortest Path Problem
- [ ] WindTurbines (VLW)
- [ ] ArtGallery (FS19-VLW)

- [ ] AVL-Tree Augment. (FS19-VLW)
- [ ] Square (HS18-VIS)
- [ ] Structure (HS18-VIS)
- [ ] AlgoTower (HS18-VLW)
- [ ] BST-Reordering (HS18-VLW)
- [ ] Grid (HS19-VLW)
- [ ] Structure (HS19-VLW)

# Karatsuba's Algorithm

- Useful for multiplication of big numbers (in $n = 2^k$)

## Pseudocode:

- split: $a = a_1 \cdot 10^{n/2} + a_0$ & $b = b_1 \cdot 10^{n/2} + b_0$
- calculate: $(a_0 \cdot b_0)$; $(a_1 \cdot b_1)$; $(a_1 - a_0) \cdot (b_1 - b_0)$
- return: $(a_1 \cdot b_1 \cdot 10^n) + (a_0 \cdot b_0 + a_1 \cdot b_1 - (a_1 - a_0) \cdot (b_1 - b_0) \cdot 10^{n/2} + a_0 \cdot b_0$

## Runtime:

- $T(2^k) = 3 \cdot T(2^{k-1}) = \ldots = 3^k \cdot T(2^0) = 3^k$
- since $n = 2^k$, we have $3^k = n^{\log_2(3)} \leq n^{1.58}$

---

# Binary Search

- Useful for searching in sorted arrays
- Divide-and-conquer approach

```
BINARY-SEARCH(A[]; b)
    left = 1; right = 1
    while (left ≤ right) do
        middle = ⌊(left + right)/2⌋
        if (A[middle] == b)
            return middle
        else if (A[middle] > b
            right = middle - 1
        else
            left = middle + 1
    return „not found"
```

### Example

- $A = \{1, 2, 11, 20, 35, 50, 75, 81\}$
- $b = 75$

i.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 11 | 20 | 35 | 50 | 75 | 81 |

ii.

| 5 | 6 | 7 | 8 |
|---|---|---|---|
| 35 | 50 | 75 | 81 |

iii.

| 7 | 8 |
|---|---|
| 75 | 81 |

iv. return 7;

## Runtime

- $T(n) = \begin{cases} c & \text{if } n = 0 \\ T(n/2) + d & \text{if } n > 0 \end{cases} \longrightarrow T(n) \in O(\log(n))$

# Linear Search

- Useful for searching in unsorted arrays

```
LINEAR-SEARCH (A[n]; b)
    for (i = 1,..., n) do
        if (A[i] == b)
            return i
    return „not found"
```

## Runtime

- $T(n) \in O(n)$ comparisons

## Example

- A = { 1, 3, 7, 11, 18, 23, 32
- b = 18

i.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 3 | 7 | 11 | 18 | 23 | 32 |

ii.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 3 | 7 | 11 | 18 | 23 | 32 |

iii.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 3 | 7 | 11 | 18 | 23 | 32 |

iv.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 3 | 7 | 11 | 18 | 23 | 32 |

v.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 1 | 3 | 7 | 11 | 18 | 23 | 32 |

vi. return 5;

---

# IsSorted

- Useful to check if an array is sorted

```
Is-SORTED (A[n])
    for (i = 1,..., n-1) do
        if (A[i] > A[i+1])
            return false
    return true
```

## Runtime

- $T(n) \in O(n)$

## Example

- A = { 1, 3, 7, 5, 9 }

i.

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 3 | 7 | 5 | 9 |

ii.

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 3 | 7 | 5 | 9 |

iii.

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 3 | 7 | 5 | 9 |

iv. return false;

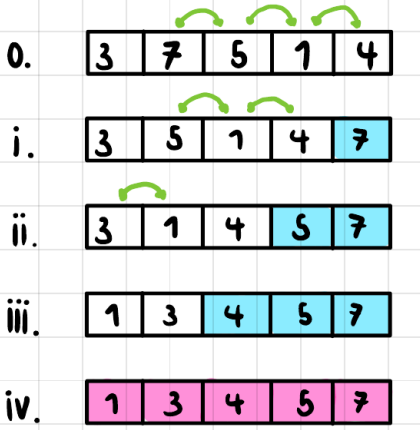
# Bubble Sort

- Sorting arrays with modified IsSorted method

```
BUBBLE-SORT(A[n])
  for (i = 1, ..., n-1) do
    for (j = 1, ..., n-i-1) do
      if (A[j] > A[j+1])
        swap(A[j], A[j+1])
```

## EXAMPLE

- A = {3, 7, 5, 1, 4}




## RUNTIME

- $T(n) \in O(n^2)$ comparisons
- $T(n) \in O(n^2)$ swaps

---

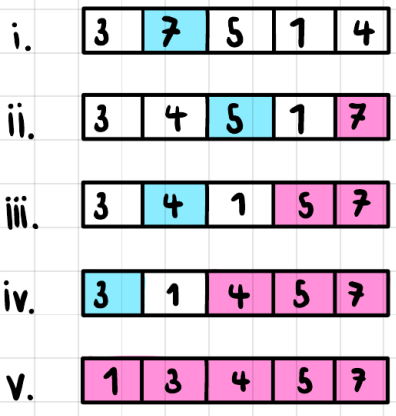
# Selection Sort

- Sorting arrays inductively

```
SELECTION-SORT(A[])
  for (k = n ... 2) do
    get index of max key in ... A[k]
    swap(i, k)
```

## EXAMPLE

- A = {3, 7, 5, 1, 4}




## RUNTIME

- $T(n) \in O(n^2)$ comparisons
- $T(n) \in O(n)$ swaps

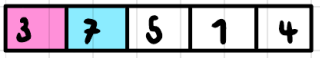
# Insertion Sort

- Sorting array inductively, but with different invariant

```
INSERTION-SORT(A[n])
  for(i = 1... (n-1)) do
    use BinarySearch to find index j
    where A[i+1] needs to be put
    b = A[k+1]
    for(k = i... j) do
      A[k+1] = A[k]
    A[j] = b
```

## EXAMPLE

- A = {3, 7, 5, 1, 4}

i. 


ii. 


iii. 


iv. 


v. 


## RUNTIME

- $T(n) \in O(n \cdot \log(n))$ comparisons
- $T(n) \in O(n^2)$ swaps

# Heap Sort

- Sorting arrays with data structure where searching Max is cheaper.

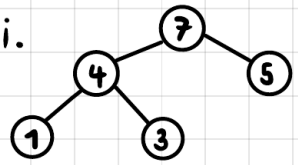
```
HEAP-SORT(A[n])
    for (i = ⌊n/2⌋ ... 1) do
        RestoreHeapCondition(A, i, n)
    for (n = n ... 2) do
        swap(1, m)
        RestoreHeapCondition(A, 1, m-1)
```

```
RESTORE HEAPCONDITION(A, i, m)
    while (2·i ≤ m) do
        j = 2·i
        if (j+1 ≤ m) do
            if (A[j] < A[j+1]) do
                j = j+1
            if (A[i] ≥ A[j]) do STOP
            swap(i, j)
            i = j
```

## EXAMPLE

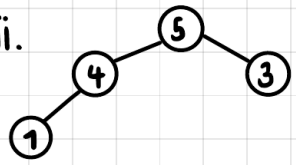
- A = {3, 7, 5, 1, 4}

i.



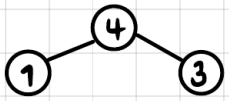

| 7 | 4 | 5 | 1 | 3 |
|---|---|---|---|---|

ii.



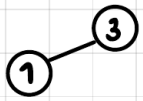

| 5 | 4 | 3 | 1 | 7 |
|---|---|---|---|---|

iii.




| 4 | 1 | 3 | 5 | 7 |
|---|---|---|---|---|

iv.




| 3 | 1 | 4 | 5 | 7 |
|---|---|---|---|---|

v.


| 1 | 3 | 4 | 5 | 7 |
|---|---|---|---|---|

vi.

| 1 | 3 | 4 | 5 | 7 |
|---|---|---|---|---|

# Merge Sort

- Divide and conquer approach to sorting an array

```
MERGE-SORT(A[n], le, ri)
  if(le < ri) do
    middle = ⌊(le+re)/2⌋
    MergeSort(A[], le, middle)
    MergeSort(A[], middle+1, ri)
    Merge(A, le, ri, middle)
```

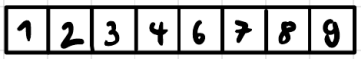
## Example

- $A = \{9, 7, 3, 2, 1, 8, 4, 6\}$

i. 9 7 3 2 1 8 4 6

ii. 

iii. 

iv. 

```
MERGE(A[n], le, ri, mi)
  B[] = new Array[ri-le+1]
  i = le ; j = mi+1 ; k = 1
  while(i ≤ mi && j ≤ ri) do
    if(A[i] ≤ A[j]) do
      B[k] = A[i] ; i++
    else
      B[k] = A[j] ; j++
    k++
  while(i ≤ mi) do
    B[k] = A[i] ; i++ ; k++
  while(j ≤ ri) do
    B[k] = A[j] ; j++ ; k++
  for(k = le ... ri) do
    A[k] = B[k-le+1]
```

## Runtime

- $T(n) \in O(n \cdot \log(n))$ comp.
- $T(n) \in O(n \cdot \log(n))$ swaps

# Quick Sort

- Sorting array with most of work done in partitioning the array into subarrays

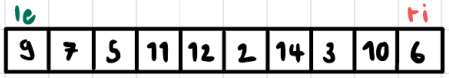
```
QUICK-SORT(A[n], le, ri)
    K = Partition(A[], le, ri)
    QuickSort(A, le, K-1)
    QuickSort(A, K+1, ri)
```

```
PARTITION(A[], le, ri)
    i = le ; j = re - 1 ; p = A[re]
    Do
        while(i < re && A[i] < p)
            i++
        while(j > le && A[j] > p)
            j--
        if(i < j) do
            swap(i, j)
    while(i ≥ j)
    swap(i, re)
    return i
```

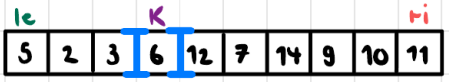
## EXAMPLE

- A = {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

i.

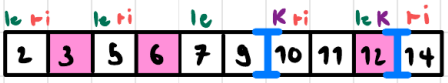



ii.




iii.


iv.




v.


vi.


## RUNTINE

- $T(n) \in O(n \log(n))$
- $T(n) \in O(n^2)$ if pivot = A[1]

# Längste aufsteigende Teilfolge

- Find the longest increasing subsequence in a given sequence
- We go through the sequence and decide for each element A[i] between the two cases:
  - case 1: A[i] fits → new length
  - case 2: doesn't fit → lower the ending value of a subsequence which's ending value > A[i] and the ending value of the one lower subsequence is < A[i]

↳ LENGTH(i) = 1 + max(L(j)) where $0 < j < i$ and $A[j] < A[i]$

## EXAMPLE

- A = { 4, 9, 8, 13, 10, 11, 7, 3, 16 }

| Length: | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Ending value: | 4 | ~~9~~ 8 | ~~13~~ 10 | 11 | 16 |
| | 3 | 7 | | | |

## RUNTIME

- $T(n) \in O(n \cdot \log(n))$ OP's
- $T(n) \in O(n)$ memory

---

# Längste gemeinsame Teilfolge

- Finding the longest common subsequence of two sequences
- We write the two sequences over each other and shift the elements with spaces to find matchings. We have 4 cases:
  - i. $\frac{x}{\_}$ → LGT(n,m) = LGT(n-1, m)
  - ii. $\frac{\_}{x}$ → LGT(n,m) = LGT(n, m-1)
  - iii. $\frac{x}{y}$ → LGT(n,m) = LGT(n-1, m-1)
  - iv. $\frac{x}{x}$ → LGT(n,m) = LGT(n-1, m-1) + 1

↳ LGT(i,j) = max(LGT(i-1, j), LGT(i, j-1), LGT(i-1, j-1) + 1)

## EXAMPLE

| LGT | - | T | I | G | E | R |
|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 |
| Z | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 1 | 1 | 1 | 1 |
| E | 0 | 0 | 1 | 1 | 2 | 2 |
| G | 0 | 0 | 1 | 2 | 2 | 2 |
| E | 0 | 0 | 1 | 2 | 3 | 3 |

## RUNTIME

- $T(n) \in O(n \cdot m)$ OP's
- $T(n) \in O(n \cdot m)$ memory

# Minimale Editierdistanz

- Given two sequences, what is the minimal number of editing operations to make them equal
- We can perform 3 operations, delete, insert and change.

↳ $ED(i,j) = \min(ED(i-1,j)+1,\ ED(i,j-1)+1,\ ED(i-1,j-1)+1)$

## Example

- | ED | – | T | I | G | E | R |
|---|---|---|---|---|---|---|
| – | 0 | 1 | 2 | 3 | 4 | 5 |
| Z | 1 | 1 | 2 | 3 | 4 | 5 |
| I | 2 | 2 | 1 | 2 | 3 | 4 |
| E | 3 | 3 | 2 | 2 | 2 | 3 |
| G | 4 | 4 | 3 | 2 | 3 | 3 |
| E | 5 | 5 | 4 | 3 | 2 | 3 |

## Runtime

- $T(n) \in O(m \cdot n)$ OP's
- $T(n) \in O(m \cdot n)$ memory

---

# Subset Sum

- Given an array $A[n]$ and a sum $b$, we try to find the elements such that $\Sigma A[i] = b$.
- For each element of the array, we distinguish two cases:
  - case 1: $b$ is subset sum of $A[1 \dots n-1]$
  - case 2: $b - A[n]$ is subset sum of $A[1 \dots n-1]$

↳ $TS(i,s) = TS(i-1,s)$ or $TS(i-1, s-A[i])$

## Example

- | SS | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 7 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 3 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

## Runtime

- $T(n) \in O(b \cdot n)$ OP's
- $T(n) \in O(b \cdot n)$ memory

# Knapsack Problem

- Given a backpack with weightlimit $W$ and $n$ objects each with value $v_i$ and weight $w_i$, we search for $I \subseteq \{1, \dots, n\}$ such that $\sum w_i \leq W$ and $\sum v_i$ is maximal.
- We have that the optimal solution is either solution for $1 \dots n-1$ with $W$ or for $1 \dots n-1$ with $W - w_n$.

↳ $T(i, w) = \max(T(i-1, w), T(i-1, w-w_i))$

### Runtime

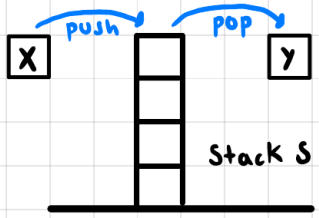
- $T(n) \in O(n \cdot W)$ OP's
- $T(n) \in O(n \cdot W)$ memory

- We can make the algorithm faster by approximating the optimal solution
  - Input-Approx: $w_i, \lfloor v_i / K \rfloor, W$
  - Factor $K = \varepsilon / n \cdot v_{max}$
  - Runtime $T(n) \in O(n^3 / \varepsilon)$

---

# Stack

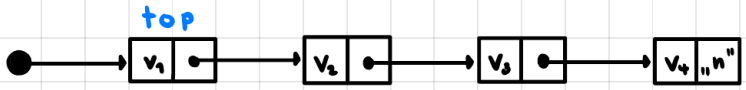
- For a ADT-Stapel (Stack) we require the following methods:
  - push(x, S) ⟶ Put x onto the stack S
  - pop(S) ⟶ Remove the top element of the stack S
  - top(S) ⟶ Returns the top element of the stack S
- Visualization:




### Runtime

- push(x, S) $\in O(1)$
- pop(S) $\in O(1)$
- top(S) $\in O(1)$

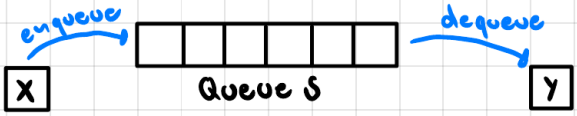
- Structure: Linked List



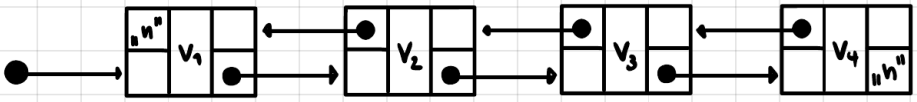
# Queue

- For an ADT - Schlange (Queue) we require the following methods:
  - enqueue (x, S) ⟶ Add x to the Queue S
  - dequeue (S) ⟶ Remove the first element of the Queue S

- Visualization:




- Structure: Doubly LinkedList




### Runtime

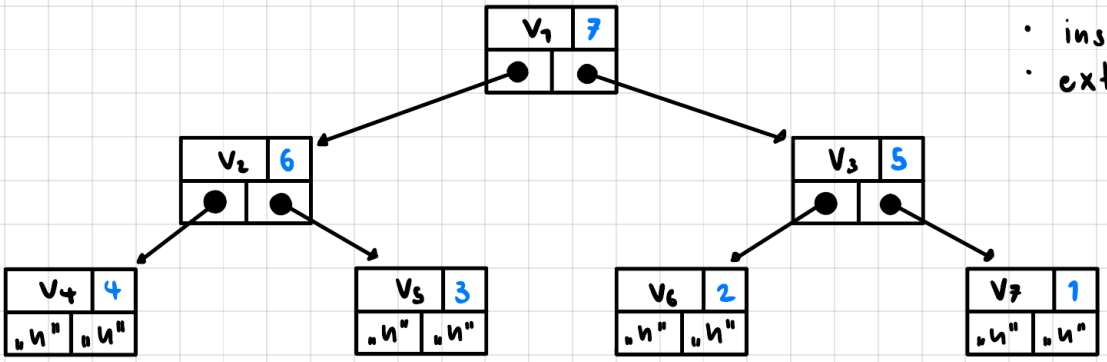
- enqueue (x, S) ∈ O(1)
- dequeue (S) ∈ O(1)

---

# PriorityQueue

- For an ADT - Prioritätsschlange (Priority Queue) we require the following methods:
  - insert (x, p, P) ⟶ Inserts x with priority p into the Queue P
  - extract Max (P) ⟶ Extracts the element with maximal priority

- Structure: Max Heap

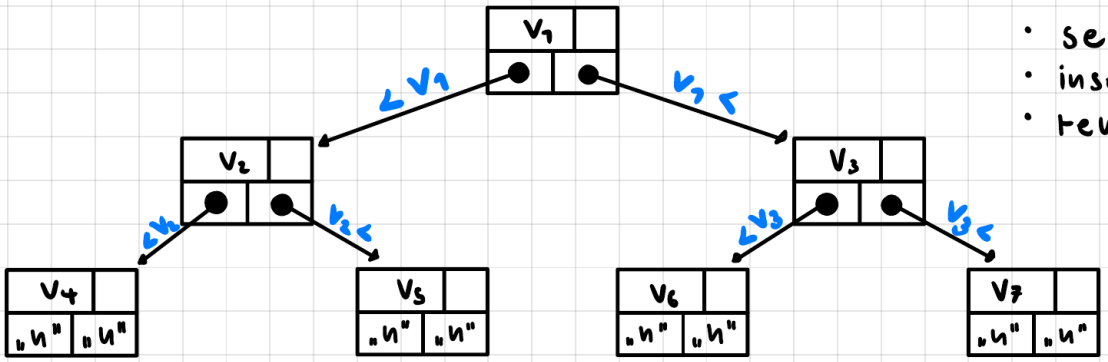



### Runtime

- insert (x, p, P) ∈ O(log (n))
- extractMax (P) ∈ O(log (n))

# Dictionary_

- For an ADT - Wörterbuch (DICTIONARY) we require the following methods:
  - search(x, W) ⟶ Find if x is in dictionary W
  - insert(x, W) ⟶ Insert x into the dictionary W
  - remove(x, W) ⟶ Remove x from the dictionary W
- Structure: SEARCH TREE



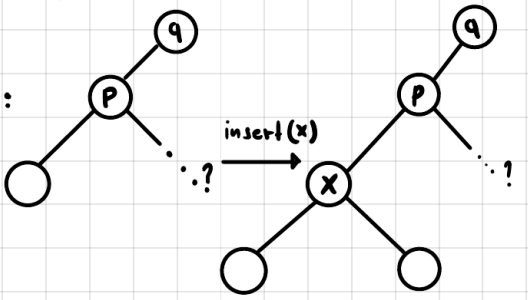

**RUNTIME**

- search(x, W) ∈ O(h)
- insert(x, W) ∈ O(h) + O(1)
- remove(x, W) ∈ O(h) + O(1)

---

# AVL Tree

- The problem with a search tree is, that the height could go to h = n, and each operation would then run in O(n).
- For an AVL-Tree we define the balance of a vertice as $bal(v) = h(T_r(v)) - h(T_l(v))$. For a search tree to fulfill the AVL-condition, we need $bal(v) \in \{-1, 0, 1\}$.
- We differ 3 states before inserting a node:
  - bal(p) = -1: not possible
  - bal(p) = 0 $\xrightarrow{insert(x)}$ bal(x) = 0; bal(p) = -1; upin(p)!
  - bal(p) = +1 $\xrightarrow{insert(x)}$ bal(x) = 0; bal(p) = 0; done!

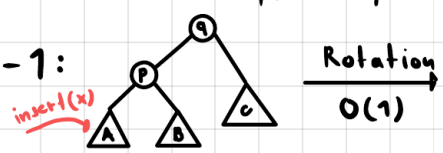



- For the function upin(q) we differ the following cases:
  - bal(q) = +1 $\xrightarrow{insert(x)}$ bal(q) = 0; done
  - bal(q) = 0 $\xrightarrow{insert(x)}$ bal(q) = -1; upin(q)
  - bal(q) = -1 $\xrightarrow{insert(x)}$ bal(p) = ±1; rebalance!

**RUNTIME**

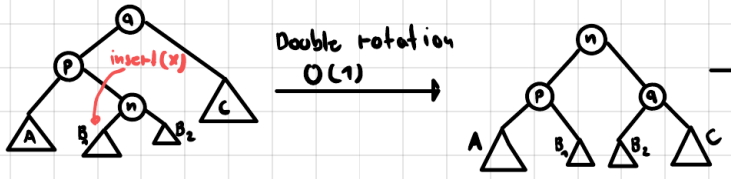
- insert(x) ∈ O(log(n))
- remove(x) ∈ O(log(n))

- bal(p) = -1:




⟶ bal(p) = 0, bal(q) = 0

- bal(q) = +1:




⟶ bal(n) = 0, bal(q) = 1, bal(p) = 0

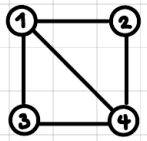
# Graph Theory

- Euler path: visit each edge once
- Hamilton path: visit each vertice once
- Degree: number of connected vertices
- Graph $G = (V, E)$
- $V$: set of vertices
- $E$: set of edges
- Walk: series of connected vertices
- Path: Walk without repeated vertices
- Closed Walk: Walk where $v_0 = v_n$
- Cycle: Closed walk without repeated vertices
- Directed graph: Edges are ordered pairs
- Ancestor: $u$ in $u \longrightarrow v$
- Successor: $v$ in $u \longrightarrow v$
- $deg_{in}(v)$: Incoming edges into $v$
- $deg_{out}(u)$: Outgoing edges from $v$
- $dist(u,v)$: minimal number of edges between the vertices $u, v$
- $relax(u,v)$: relax a spanned edge, i.e. $d[v] = d[u] + w(u,v)$
- spanned edge: $d[v] > d[u] + w(u,v)$

# Graph Representation

- **Adjacency Matrix**
  - We can represent a graph in a matrix where $A_{uv} = \begin{cases} 1 & \text{if } (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$

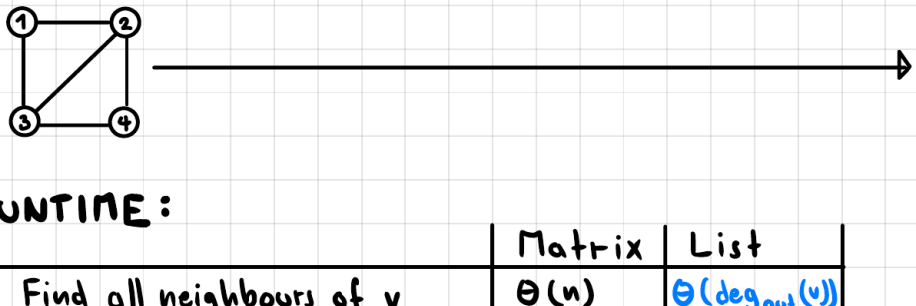
**Example**




$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \end{array}$$

- **Adjacency List**
  - We can represent a graph in an array of Linked List's, where Adj[u] holds all the neighbours of u.

**Example**




**Runtime:**

| | Matrix | List |
|---|---|---|
| Find all neighbours of $v$ | $\Theta(n)$ | $\Theta(deg_{out}(v))$ |
| Find $v \in V$ without neighbours | $O(n^2)$ | $O(n)$ |
| Is $(u,v) \in V$ ? | $O(1)$ | $O(deg_{out}(v))$ |
| Insert edge | $O(1)$ | $O(1)$ |
| Remove edge | $O(1)$ | $O(deg_{out}(v))$ |

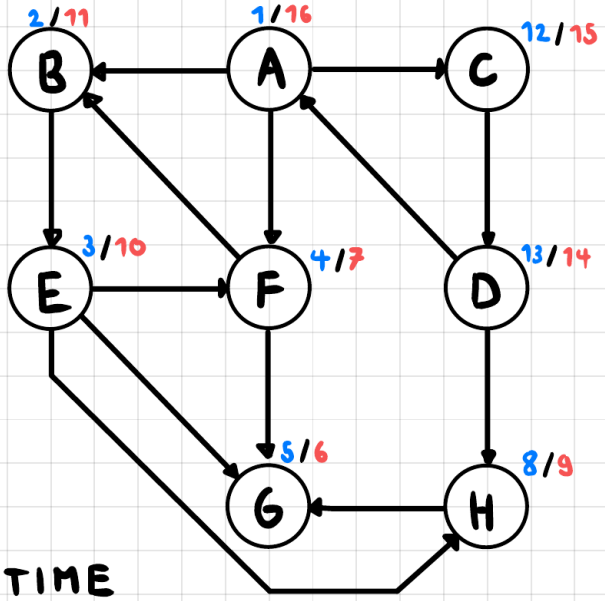
# Depth-First Search

- When wanting to make definitions over a graph (e.g if there exists a cycle etc.), we need a way to „walk through" the graph. That is where we can use a DFS (Depth-First Search).

```
DFS(G)
    for (v∈V not marked) do
        DFS-Visit(v)
```

```
DFS-Visit(v)
    t = 0
    pre[v] = t; t++
    marked[v] = true
    for ((v,u)∈V not marked)do
        DFS-Visit(u)
    post[u] = t; t++
```

### Example (pre/post)

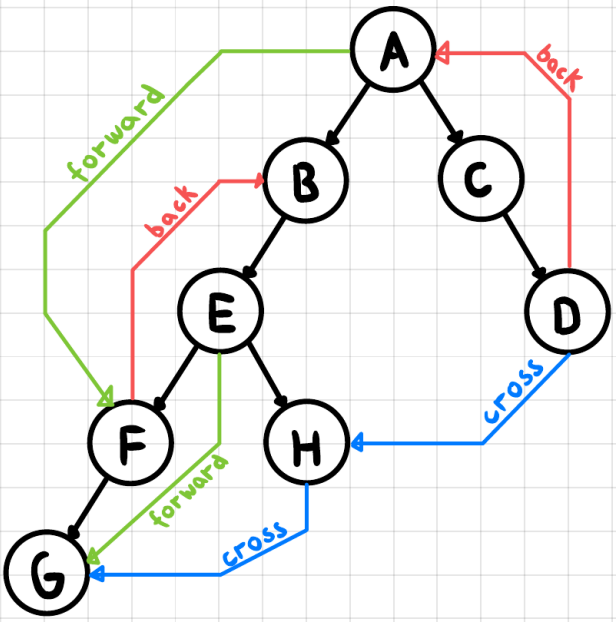



### Runtime

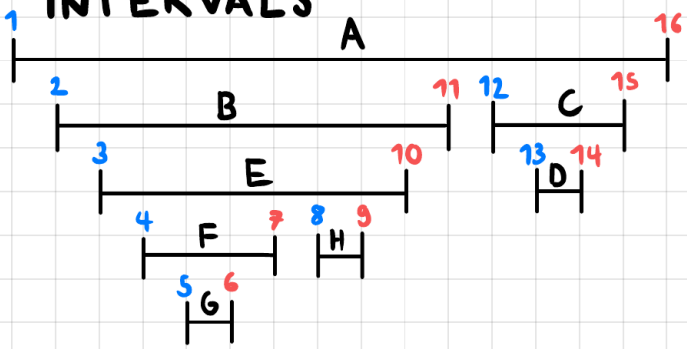
- $T(n) \in \Theta(|E|+|V|)$ OP's
- $T(n) \in O(|V|)$ memory

# Edge Classification

### Example (from DFS)

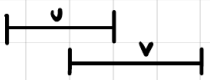
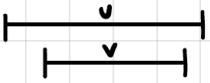
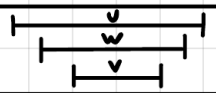
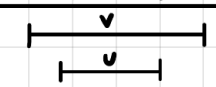
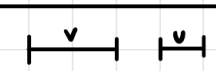
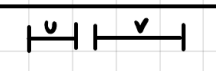



### Intervals




### Classification via Intervals

- Edge $(u,v) \in E$

| | |
|---|---|
| i.  | not possible |
| ii.  | forward or DFS-Tree |
| iii.  | forward edge |
| iv.  | back edge |
| v.  | cross edge |
| vi.  | not possible |

### Remark

- $\nexists$ back edge $\Leftrightarrow$ $\nexists$ closed walk

# Breadth-First Search

- Instead of searching through the depth of a graph first, we can also first search through all direct successors of the root with the BFS (Breadth-first-search) - Algorithm.

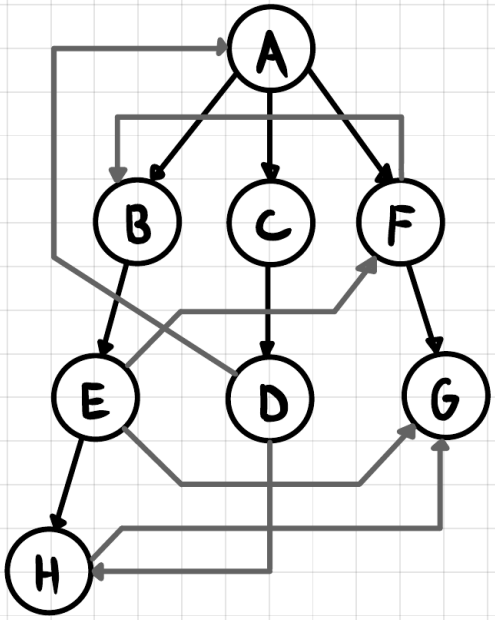
```
DFS(G)
    for (v∈V not marked) do
        DFS-Visit(v)
```

```
DFS-Visit(v)
    Q = new Queue
    active[v] = true
    enqueue(v, Q)
    while(!IsEmpty(Q)) do
        w = dequeue(Q)
        visited[w] = true
        for ((w,x)∈E) do
            if(!active[x] && !visited[x])
                active[x] = true
                enqueue(x, Q)
```

## Runtime

- $T(n) \in \Theta(|V| + |E|)$ OP's
- $T(n) \in O(|V|)$ memory

## Example (from DFS)



# Dijkastra

- To find the shortest (cheapest) path in a graph we can use the Dijkstra-Algorithm. ⟶ For graphs with non-negative weights!

```
DIJKSTRA (G, s)
    for (v ∈ V) do
        distance[v] = ∞
        parent[v] = null
    distance[s] = 0
    Q = new Queue
    insert(s, 0, Q)
    while (! Q.isEmpty) do
        u = Q.ExtractMin
        for ((u,v) ∈ E) do
            if (parent[v] == null) do
                distance[v] = distance[u] + w(u,v)
                parent[v] = u
            else if (distance[u] + w(u,v) < distance[v]) do
                distance[v] = distance[u] + w(u,v)
                parent[v] = u
                decreaseKey(v, distance[v], Q)
```

RUNTIME (depends on implementation)

- $T(n) \in O((|V|+|E|) \cdot \log(|V|))$ when implemented with a heap

# Bellman-Ford Algorithm

- For graphs with general edge-weights (positive and negative!) where we want to find the shortest (cheapest) paths, we can use the Bellman-Ford Algorithm.

```
BELLMAN-FORD(G, s)
    for(v ∈ V) do
        distance[v] = ∞
        parent[v] = null
    distance[s] = 0
    for(i = 1, 2, ..., |V|-1) do
        for((u,v) ∈ E) do
            if(distance[v] > distance[u] + w(u,v)) do
                distance[v] = distance[u] + w(u,v)
                parent[v] = u
    for((u,v) ∈ E) do
        if(distance[u] + w(u,v) < distance[v]) do
            return „negative cycle exists!"
```

## RUNTIME

- $T(n) \in O(|E| \cdot |V|)$ OP's

---

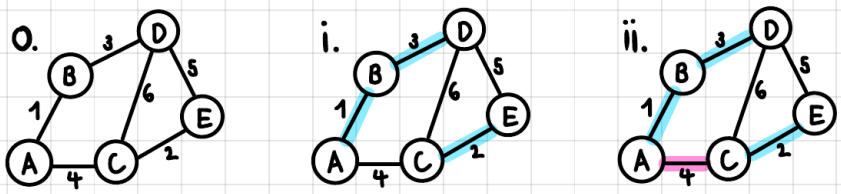
# Boruvka's Algorithm

- To find a MST in a given graph, we can use BORUVKA'S ALGORITHM.

```
BORUVKA(G)
    MST = new minimalSPtree
    components = new Set
    for(v ∈ V) do
        components.add(v)
    while(components.size > 1) do
        for(comp ∈ components) do
            edge e = connectWithMinDist(comp)
            MST.add(e)
```

## RUNTIME

- $T(n) \in \Theta((|V|+|E|) \cdot \log(|V|))$



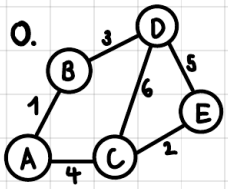
# Kruskal's Algorithm

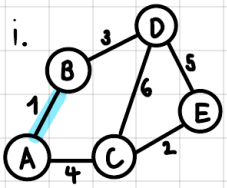
- KRUSKAL'S ALGORITH delivers another variation to find a MST in a given graph. It sorts edges by weight and adds them one-by-one as long as there is no cycle created.

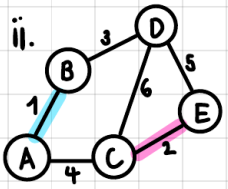
```
KRUSKAL(G)
    MST = new minimalSPtree
    E = new Set
    E.add(all Edges)
    E.sort(by weight)
    for(e ∈ E) do
        E.remove(e)
        if(∄ cycle in MST with e) do
            MST.add(e)
```

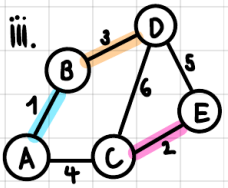
### RUNTIME

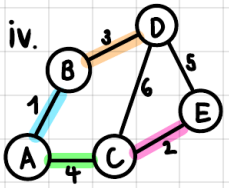
- $T(n) \in O(|E| \cdot \log|V|)$ OP's

0.



i.



ii.



iii.



iv.


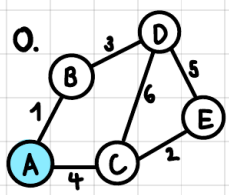

# Prim's Algorithm

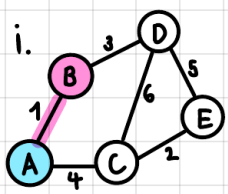
- PRIM'S ALGORITHM is an alternative to Kruskal's Algorithm. It needs a given starting vertex, from where we add one-by-one the smallest edge, outgoing from the starting vertex or the created subtree.

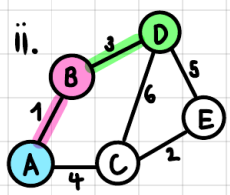
```
PRIM(G, s)
    MST = new minSPtree
    H = create-min-Heap(V, ∞)
    decreaseKey(H, s, 0)
    while(!H.IsEmpty) do
        v = extractMin(H)
        MST = MST ∪ {v}
        for((v,u) ∈ E, v ∉ MST) do
            decreaseKey(H, u, w(v,u))
```

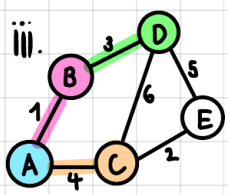
### RUNTIME

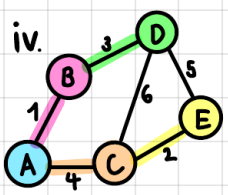
- $T(n) \in O((|V|+|E|) \cdot \log|V|)$

0.



i.



ii.



iii.



iv.


# Floyd-Warshall Algorithm

- The FLOYD-WARSHALL ALGORITHM is for solving the All-Pair Shortest Path problem. This is to find the sortest distance between any two vertices of a given graph (without negative closed walks!).

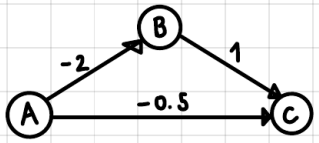
```
FLOYDWARSHALL(G)
  for(n∈V) do
    d⁰_nn = 0
  for((n,v)∈E) do
    d⁰_nv = w(n,v) ; else d⁰_nv = ∞
  for(i=1...n) do
    for(u=1...n) do
      for(v=1...n) do
        d^i_uv = min(d^(i-1)_uv , d^(i-1)_ui + d^(i-1)_iv)
```

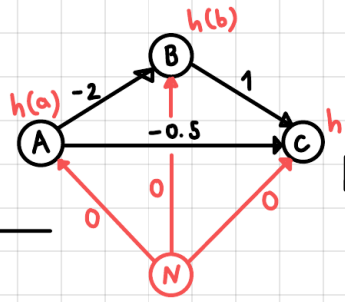
**RUNTIME**

- $T(n) \in O(|V|^3)$

# Johnson's Algorithm

- JOHNSON'S ALGORITHM is based on the idea, to first make every weight positive, and then simply use n-times Dijkstra to solve the All-Pair Shortest Path problem.



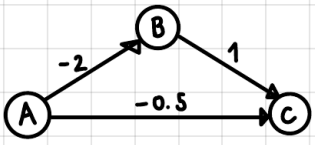



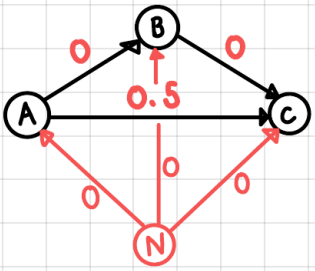

→ h(n) is the „height" of each node, equal to the shortest path from N to n.

- h(a) = 0
- h(b) = -2
- h(c) = -1

→ We know can transform each weight w(u,v) of each edge to a new one $\hat{w}(u,v)$, where each weight is positive:

- $\hat{w}(u,v) = w(u,v) + (h(u) - h(v))$



# All-Pair Shortest Path

- We now know different algorithms to solve the ALL-PAIR SHORTEST PATH problems. Those algorithms compare in the following way to each other:

| GRAPH | ALGORITHM | RUNTIME |
|---|---|---|
| $G = (V, E)$ | $n \cdot$ BFS | $O(mn + n^2)$ |
| $G = (V, E, w)$ $w: E \to \mathbb{R}^+$ | $n \cdot$ Dijkstra | $O(mn + n^2 \cdot \log(n))$ |
| $G = (V, E, w)$ $w: E \to \mathbb{R}$ | $n \cdot$ Bellman-Ford | $O(mn^2)$ |
| | Floyd-Warshall | $O(n^3)$ |
| | Johnson's Algorithm | $O(mn + n^2 \log(n))$ |

more general

# Selection Problem

- Given an array $A[] = A[1] \dots A[n]$, we want to find the $i$-th smallest element in the array.

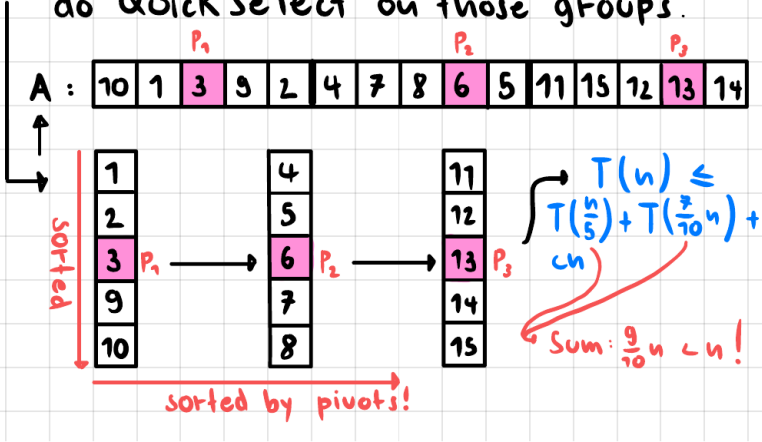
## IDEAS

i. output $i$-times the smallest element $\longrightarrow \Theta(i \cdot n)$
ii. sort, output $i$-th element $\longrightarrow \Theta(n \cdot \log(n))$
iii. Quick Select $\longrightarrow$ $O(n)$ average, $O(n^2)$ worst case

```
QUICKSELECT (A[], i)
    p = choose pivot
    move elements <p to the left of p.
    if (index(p) == i) do
        return p
    else if (i > index(p)) do
        QuickSelect(A[p+1, ..., n], i)
    else do
        QuickSelect(A[1, ..., p-1], i)
```

## IDEA

- Split array into pairs of 5 and do Quick Select on those groups.

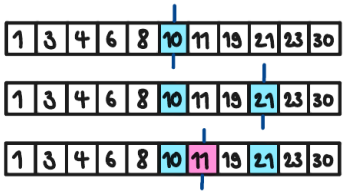


# Cheat-Sheet Sorting Algorithms

## ☐ BINARY SEARCH:

- RUNTIME: $O(\log(n))$ comp.
- USE: Searching in sorted arrays
- APPROACH: Check middle, if $x \leq$ middle, check middle of left half etc.

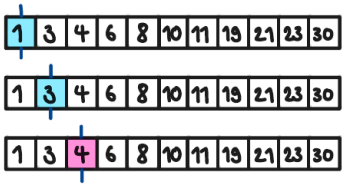
Find 11:




## ☐ LINEAR SEARCH

- RUNTIME: $O(n)$ comp.
- USE: Searching in unsorted arrays
- APPROACH: Go one-by-one through the array

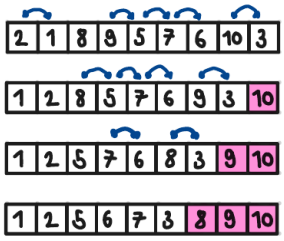
Find 4:




## ☐ BUBBLE SORT

- RUNTIME: $O(n^2)$ comp. & swap.
- APPROACH: Go $n$ times through the array, whenever a succesive element is smaller, swap it with the previous one.
- REMARK: At each step, the $n$-th largest element is at its correct position!

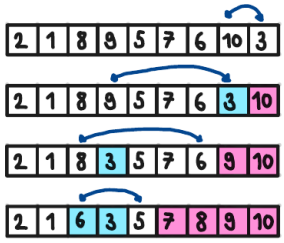
Step: 0.




## ☐ SELECTION SORT

- RUNTIME: $O(n^2)$ comp. $O(n)$ swap.
- APPROACH: Select the greatest (or least) element and put it at the correct position.
- REMARK: Elements are swapped, not shifted!

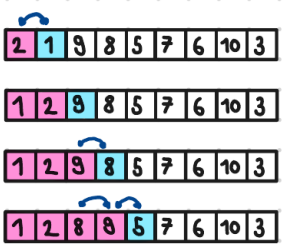
Step: 0.




## ☐ INSERTION SORT

- RUNTIME: $O(n \cdot \log(n))$ comp. $O(n^2)$ swap.
- APPROACH: Go through array once and determine for each element its correct position and place it there.
- REMARK: Elements are shifted, not swapped!

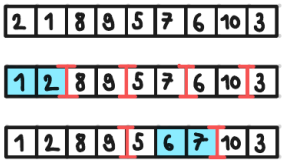
Step: 0.




## ☐ MERGE SORT

- RUNTIME: $O(n \cdot \log(n))$ comp. & swap.
- APPROACH: Array is split into pairs, then 4-tuples etc. which get sorted at each step.
- REMARK: All pairs are sorted after the first step!

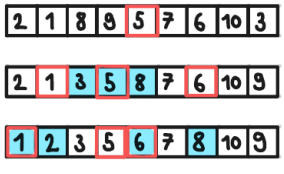
Step: 0.




## ☐ QUICK SORT

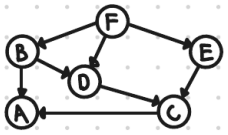
- RUNTIME: $O(n \cdot \log(n))$ avg., $O(n^2)$ worst.
- APPROACH: Each step a pivot gets selected and all remainig elements are moved such that all elements on the left of the pivot are smaller than the pivot.
- REMARK: There exists a pivot where after the first step, all element on its left side are smaller than the pivot!

Step: 0.



# Cheat-Sheet Graph Algorithms

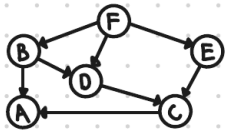
## ☐ DEPTH-FIRST SEARCH (DFS)




DFS(F): F, B, A, D, C, E

- RUNTIME: $\Theta(|E|+|V|)$
- APPROACH: We first go through the depth of each node. We might keep track of pre-/post numbers to make edge classifications.
- USES: Detect cycles (if ∃ a backward edge), path finding, topological sorting (sorting by reversed post-order), test if graph is bipartite, solving problems with one solution (mazes)

## ☐ BREADTH-FIRST SEARCH (BFS)




BFS(F): F, B, D, E, A, C

- RUNTIME: $\Theta(|V|+|E|)$
- APPROACH: We first go through all direct succesors and then move one-by-one one level deeper.
- USES: shortest path in unweighted graphs, cycle detection, test if graph is bipartite, path finding.

## ☐ DIJKSTRA ALGORITHM

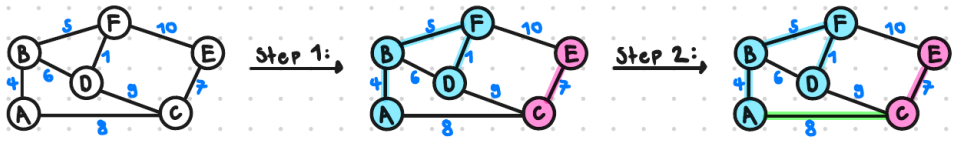
- RUNTIME: $O((|V|+|E|)\cdot\log(v))$ with adj. list & queue or with heap
- APPROACH: We put the starting node into the queue, take it out, and set the distance for all adjacent nodes and put them into the queue. We repeat, update distances and only put nodes into the queue if they weren't visited before.
- USES: find minimal-cost paths in non-negative weighted graphs.
- DATASTRUCTURES: distance[], parent[], queue, graph (adjacency list)

## ☐ BELLMAN-FORD ALGORITHM

- RUNTIME: $O(|E|\cdot|V|)$
- APPROACH: We initiate all distances with ∞. Then we go $|V|-1$ times through every edge, and test for $(u,v)\in E$, if distance[v] > distance[u] + w(u,v), if yes, we update the distance[v]. After $|V|-1$ iterations we go through each edge again, if we still can relax an edge, then there exists a negative cycle.
- USES: detect negative cycles, find minimal-cost paths in weighted graphs with negative weights.
- DATASTRUCTURES: distance[], parent[], graph (adjacency list)

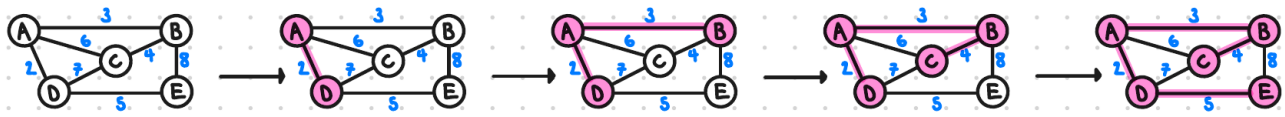
## ☐ BORUVKA'S ALGORITHM

- RUNTIME: $O((|V|+|E|)\cdot\log(|V|))$
- APPROACH: We first add all vertices to the set of components. Then as long as the size of the components-set is greater than 1, we connect each component to another with the cheapest edge.
- USES: Find MST in weighted, undirected graph.
- DATASTRUCTURES: UnionFind (MST), graph (adjacency list)



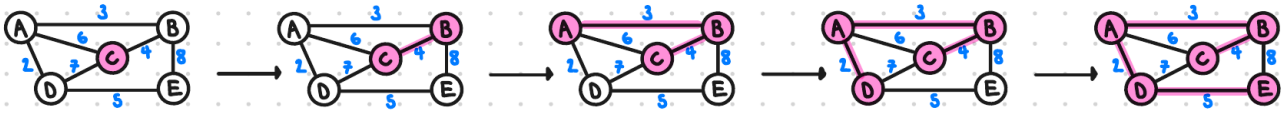

## ☐ KRUSKAL'S ALGORITHM

- RUNTIME: $O(|E|\cdot|V|)$ or $O(|E|\cdot\log(|V|))$ with UnionFind
- APPROACH: We sort the edges by weight and add them one-by-one as long as there is no cycle created.
- USES: Find MST in weighted, undirected graph.
- DATASTRUCTURES: UnionFind, graph



## ☐ PRIM'S ALGORITHM

- RUNTIME: $O((|V|+E|)\cdot \log(|V|))$
- APPROACH: We start at a given vertex and connect it to „closest" vertex. To this sub-tree we add one-by-one the cheapest edge connecting the subtree to another vertex (avoiding cycles) until all vertices are connected.
- USES: Find MST in weighted, undirected graph.
- DATASTRUCTURES: UnionFind (MST), graph

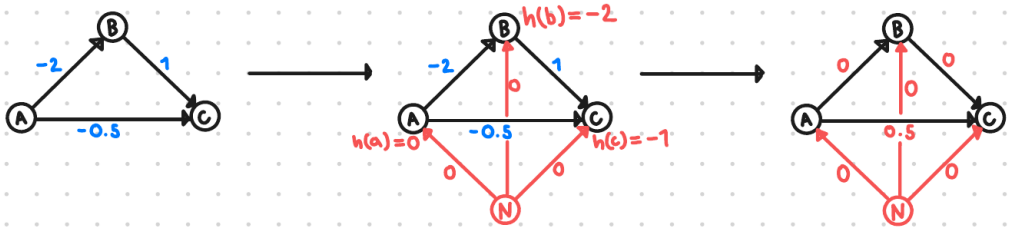



## ☐ FLOYD-WARSHALL ALGORITHM

- RUNTIME: $O(|V|^3)$
- APPROACH: DP-Algorithm in a 3D-array. It is based on the fact that in a shortest path from 1 to $k$, every path $i$ to $j$, with $i<j\leq K$ is also minimal. The recursion therefore is stated as: $d[i][u][v] = \min(d[i-1][u][v], d[i-1][u][i] + d[i-1][i][v])$.
- USES: All-to-all shortest paths in directed graph without negative closed walks.
- DATASTRUCTURES: graph (matrix), dp-table

## ☐ JOHNSON'S ALGORITHM

- RUNTIME: $O((|V|+|E|)\cdot |V| \log(|V|))$
- APPROACH: We make all edges positive and then perform n-times Dijkstra. To do this we create an additional node that is linked to each node and store for each node a height $h(x)$, where $h(x)$ is equal to the shortest path from the new node $N$ to the node $x$. The new weights are now equal to: $\hat{w}(u,v) = w(u,v) + (h(u)-h(v))$.
- USES: All-to-all shortest paths in directed graphs without negative closed walks.
- DATASTRUCTURES: distance[], parent[], queue, graph (adjacency list)
- REMARK: Faster than Floyd-Warshall in graphs with less edges.




## ☐ ONE-TO-ALL SHORTEST PATH ($|V|=n; |E|=m$)

| GRAPH | ALGORITHM | RUNTIME |
|---|---|---|
| $G=(V,E)$ | BFS | $O(m+n)$ |
| $G=(V,E,w)$<br>$w: E \to \mathbb{R}^+$ | Dijkstra | $O((m+n)\cdot \log(n))$ |
| $G=(V,E,w)$<br>$w: E \to \mathbb{R}$ | Bellman-Ford | $O(m\cdot n)$ |

## ☐ ALL-TO-ALL SHORTEST PATHS

| GRAPH | ALGORITHM | RUNTIME |
|---|---|---|
| $G=(V,E)$ | $n$x BFS | $O(mn+n^2)$ |
| $G=(V,E,w)$<br>$w: E \to \mathbb{R}^+$ | $n$x Dijkstra | $O(mn+n^2\log(n))$ |
| $G=(V,E,w)$<br>$w: E \to \mathbb{R}$ | $n$x Bellman-Ford<br>Floyd-Warshall<br>Johnson | $O(mn^2)$<br>$O(n^3)$<br>$O(mn+n^2\log(n))$ |

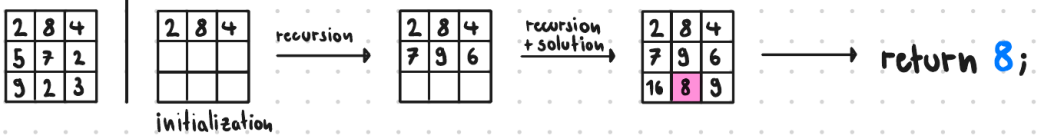
# ☐ ALL NODES ON SHORTEST PATHS

- RUNTIME: $\Theta(n+m)$ (2x BFS in $\Theta(|V|+|E|)$ and classify nodes in $\Theta(|V|)$)
- APPROACH: We want to have a list of all nodes on all shortest paths from node s to t. We do a BFS from s and from t, where we store for each node $d(s,u)$ and $d(t,u)$. All nodes occuring in a shortest path therefore are all the nodes, where $d(s,u)+d(t,u) = d(t,s)$.
- REMARK: For directed graphs we can do the same, but with 2x Dijkstra and a transposed graph $G^T$.

# Cheat-Sheet Algorithm Programming

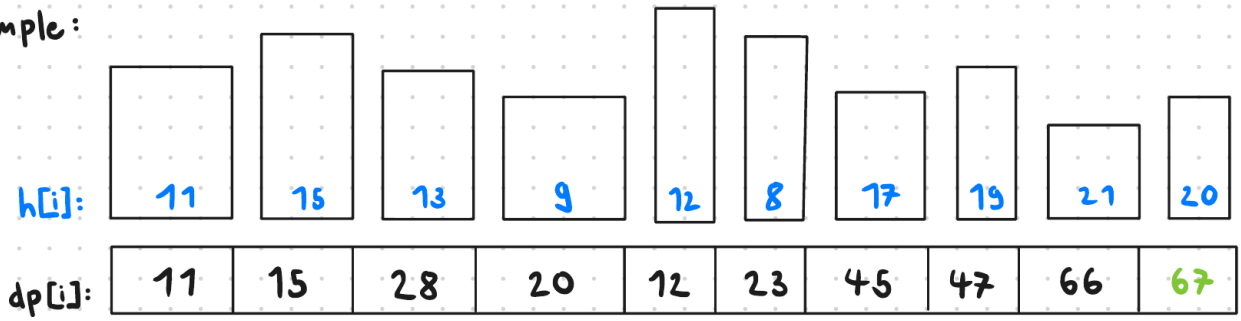
## ☐ AD19E02.GRID

- **Problem:** Given a NxN grid with each cell having a cost w(i,j), find the cost of the cheapest path from any cell in the first row, to any cell in the last row.
- **Restrictions:** Runtime: $O(N^2)$, $O(N^2 \cdot \log(N))$, $O(N \cdot 3^N)$. Possible movements: Diagonally to the left or right, downwards.
- **Recursion:** solution[i][j] = w(i,j) + min( sol.[i-1][j-1], sol.[i-1][j], sol.[i-1][j+1] )
- **Initialization:** solution[0][0...N-1] = grid[0][0...N-1]
- **Solution:** While filling out the last row, keep track of the min. element in the last row.
- **Example:** grid[][] | sol.[][]




## ☐ ALGOTOWER

- **Problem:** Given n bricks, sorted from biggest to least area with each block given a height h[i]. What is the tallest tower you can build?
- **Restrictions:** Runtime: $O(n^2)$. Blocks can only be put onto each other, if b[i+1] ≤ b[i] and l[i+1] ≤ l[i] (or 90° turned).
- **Solution:** dp[i] stores the greatest possible height with block i on top. For each block i, we try to fit it onto every block i-1...1 and store the greatest height in dp[i].
- **Example:**




## ☐ BST-REORDERING

- **Problem:** Given a preorder-sequence of a BST, return the order in which BFS traverses the tree.
- **Restrictions:** Runtime: $O(n^2)$
- **Approach:** First create a BST from the given preorder, then run BFS on it.

i. **Build tree from preorder:** We go throught the preorder, then entry 1 is the root, entry 2 the left child and entry, where i > root, is the right child. The rest we do recursive.

ii. **Run BFS** (inplace in BST)

```
Vertex queue = new Vertex[n];
int dequeue = 0;
int enqueue = 0;
int bfsOrder = new int[n];
queue[enqueue++] = root;
while (dequeue < n) {
    Vertex current = queue[dequeue++];
    bfsOrder[dequeue-1] = current.key;
    if (current.left != null) {
        queue[enqueue++] = node.left;}
    if (current.right != null) {
        queue[enqueue++] = node.right;}
}
return bfsOrder;
```

```
Vertex root = buildRec(0, n, preorder);

public Vertex buildRec(int start, int end, int[] preorder) {
    if (start > end) { return null; }
    Vertex current = preorder[start];
    for (int i = start, i≤end, i++) {
        if (preorder[i] < current.key){
            break;
    } }
    current.left = buildRec(start+1, i-1, preorder);
    current.right = buildRec(i, end, preorder);
    return current;
```

## ☐ AD18E02.SQUARE

- **Problem:** Given a matrix which's entries consist only of 0's and 1's. Return the biggest subsquare consisting of only 1's.
- **Restrictions:** $O(n \cdot m)$
- **Recursion:** $dp[i][j] = \min(dp[i][j-1], dp[i-1][j-1], dp[i-1][j]) +$ if $matrix[i][j] == 1$, $dp[i][j] = 0$ otherwise.
- **Meaning of entry:** Sidelenght of max subsquare with $dp[i][j]$ in lower right corner.
- **Initialization:** First row and column are equal to the matrix.
- **Solution:** Parallel to the recursion, keep track of the max sidelength.

## ☐ AD18E01.STRUCTURE

- **Problem:** Given a base template for a max heap structure, implement buildHeap(), insert and extractMax().
- **Restrictions:** Runtime: buildHeap $\in O(n)$, insert() and extractMax $\in O(\log(n))$
- **Solution:** See MaxHeap-DTS.

# ☐ UNION-FIND DTS

- **Idea:** Keep track of a set of elements partitioned into dis-joint subsets.
- **Required methods:**
  - `find(x)`: Determine in which subset „x" lies. Useful to check if two elements are in the same subset.
  - `union(x,j)`: Join two subsets together.
- **Implementation:** Union by rank and path compression approach:

```
public class UnionFind {
    int[] parent;
    int[] rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for(int i=0; i<n; i++) {
            parent[i] = i;
        }
    }

    int find(int x) {...}

    void unionRoots(int x, int y) {...}
}
```

```
int find(int x) {
    int root = x;

    // find root of x by following chain of pointers
    while(parent[root] != root) {
        root = parent[root];
    }

    // Path compression: Set all parents of the nodes in
      the chain to the found root
    while(x != root) {
        int nextNode = parent[x];
        parent[x] = root;
        x = nextNode;
    }
    return root;
}
```

- **Runtime:**
  - `find(x)`: O(log(N))
  - `union(x,y)`: O(log(N))

```
void unionRoots(int x, int y) {
// The method only allows to union roots! If x & y aren't roots,
   we need to use unionRoots(find(x), find(y))!
    if(rank[x] < rank[y]) {
        parent[x] = y;

    } else if(rank[x] > rank[y]) {
        parent[y] = x;

    } else {
        parent[y] = x;
        rank[x] ++;   }
}
```

# ☐ MAX-/MIN-HEAP-DTS

```
class MaxHeap {

    int N = 0   //stores current heapSize
    int[] values = new int[maxSize];

    void buildHeap() {
        int startIdx = (N/2)-1;
        for(int i = startIdx; i≥0; i--) {
            siftDown(i); } }

    void insert(int x) {
        values[N] = x;
        N++;
        siftUp(N-1); }

    int pollMax() {
        N--;
        swap(0, N)
        siftDown(0);
        return values[N] }}
```

**Helper Methods:**

```
int parentIdx(int idx) {return ((idx+1)/2)-1;}
int leftIdx(int idx) { return ((idx+1)*2)-1; }
int rightIdx(int idx) { return (idx+1)*2; }
```

```
void siftDown(int i) {
    int largest = i;
    int l = leftIdx(i);
    int r = rightIdx(i);
    if(l<N && cmp(values[largest],
       values[l]) { largest = l;}
    if(r<N && cmp(values[largest],
       values[r]) {largest = r;}
    if(largest != i){
        swap(i, largest);
        siftDown(largest)}}
```

```
void siftUp(int idx) {
    while(idx>0) {
        int parentIdx =
            parentIdx(idx);
        if(cmp(values[parent
            Idx], values[idx]))
        { swap(parentIdx, idx);
          idx = parentIdx;
        } else{
            break;
        }}}
```