Contents

1 Data Structures

Binary Search Tree

A Binary Search Tree is Binary Tree which stores keys or key-value pairs so that when you travel in-order all keys are sorted ascending.

AVLTree

An AVLTree is a special kind of Binary Search Tree. The AVL-Tree is self balancing and has therefore better running time guaranties than the normal Binary Search Tree. The tree rebalance itself when the $abs(balancefactor) > 1$. The balance factor is calculated with: `BalanceFactor(Node) = Height(Node.LeftSubtree) - Height(Node.RightSubtree)`
For rebalancing exists two methods:

- Cut/Link Algorithm aka. Trinode restructuring
- Rotation Balancing

Running time:

- find: $O(\log(n))$
- insert: $O(\log(n))$
  - at first find, after that restructure $O(1)$
- delete: $O(\log(n))$
  - at first find, after that restructure $O(\log(n))$

Cut/Link Algorithm This algorithm is used by some Binary Search Tree for rebalancing.

Input A position x of a binary search tree T that has both a parent y and a grandparent z
Output Tree t after a trinode restructuring.

1. Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y and z not rooted at x, y and z.

2. Replace the subtree rooted at z with a new subtree rooted at b
   (a) Let a be the left child of b and let T_1 and T_2 be the left and right subtrees of a.
   (b) Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c.

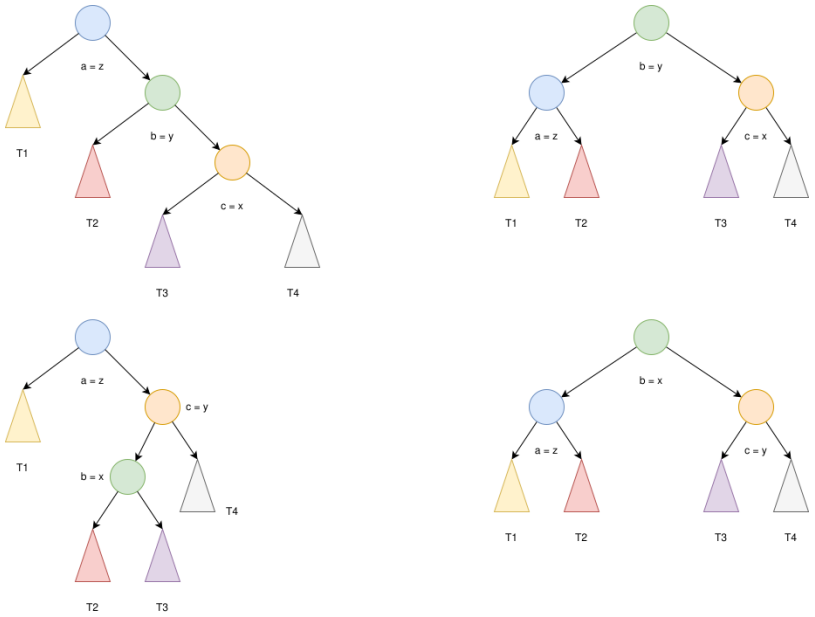


Figure 1: Cut/Link Example

Rotation Balancing
This algorithm is used by some Binary Search Trees to restructure itself. The problem with this technique is that you have to check what is required, a left rotation or a right rotation. Some times a double rotation is required.

Input A position x of a binary search tree T that has both a parent y and a grandparent z
Output Tree t after a trinode restructuring.

1. Let (a, b, c) the inorder listing of the positions x, y, z
2. Do the required rotations until b is at the top.

```
// right rotation
BinaryNode rotateWithLeftChild(BinaryNode k2) {
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

// left rotation
BinaryNode rotateWithRightChild( BinaryNode k1 ) {
    BinaryNode k2 = k1.right; // Hilfsknoten
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

Splay Tree
A Splay Tree is a Binary Search Tree which moves the accessed note after the operation to the root. Even after update or search. Moving the accessd note to the root is achived using rotation. You rotate until the access element is at the top. Splay Trees are usefull when you search often for the same thing.

Graph
A graph consist of **Vertices** and **Edges**. It exists directed and undirected edges / graphs. A directed edge has one start node and one end node. An undirected edge has two nodes which are whether start nor end node. In a directed graph (Digraph) all edges are directed. In an undirected graph all edges are undirected.
A Graph is called connected when between every pair of vertices a path exists.

Graph Terminology
The Terminology used when working with graphs (Graph).

- Edges are **incident** (finished) on a vertex

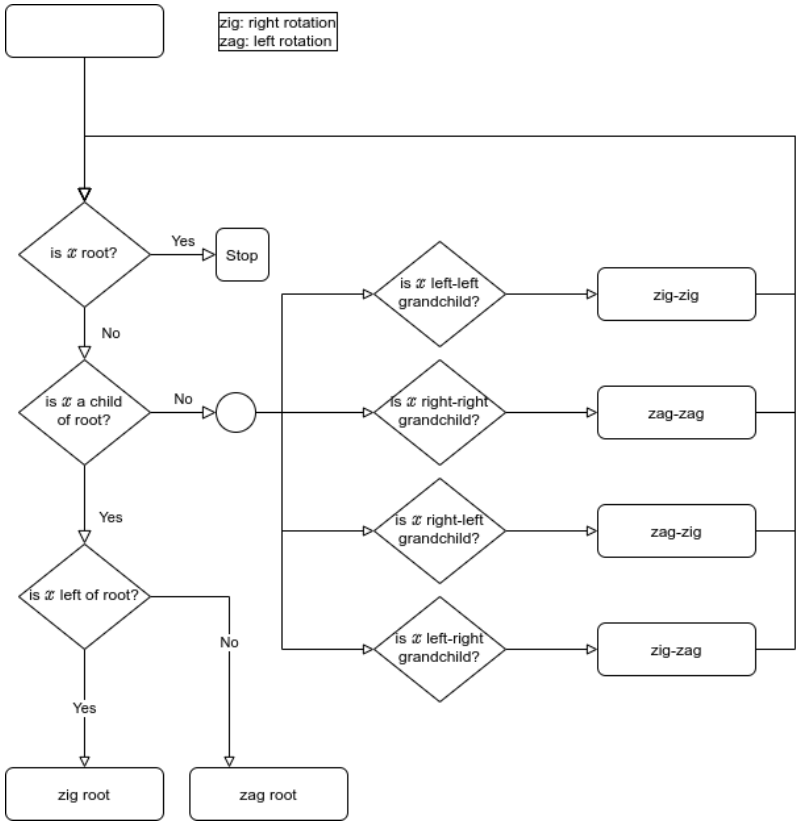


Figure 2: Splaying Algorithm

  - a, d, b are incident in V
- **Adjacent** (neighboring) vertices
  - U, V are adjacent
- **Degree** of a Vertex: The number of incident edges
  - X has degree 5
- **Parallel edges**
  - h, i are parallel
- **Loop**
  - j is a loop
- **Path**
  - Sequence of vertices and edges
  - starts with a vertex and ends with a vertex
- **Simple Path**
  - all vertices and edges are unique
- **Cycle**
  - cyclic sequence of vertices and edges
- **Simple Cycle**
  - all vertices and edges are unique
- **Strong Connectivity**
  - every vertex can reach every other vertex
- **Distance**
  - The distance of a vertex v to a vertex w is the length of the shortest path between v and w

How to Store a graph

To store a graph you have a few possibilities.

- Edge List Structure
- Adjacent List Structure
- Adjacent Matrix Structure

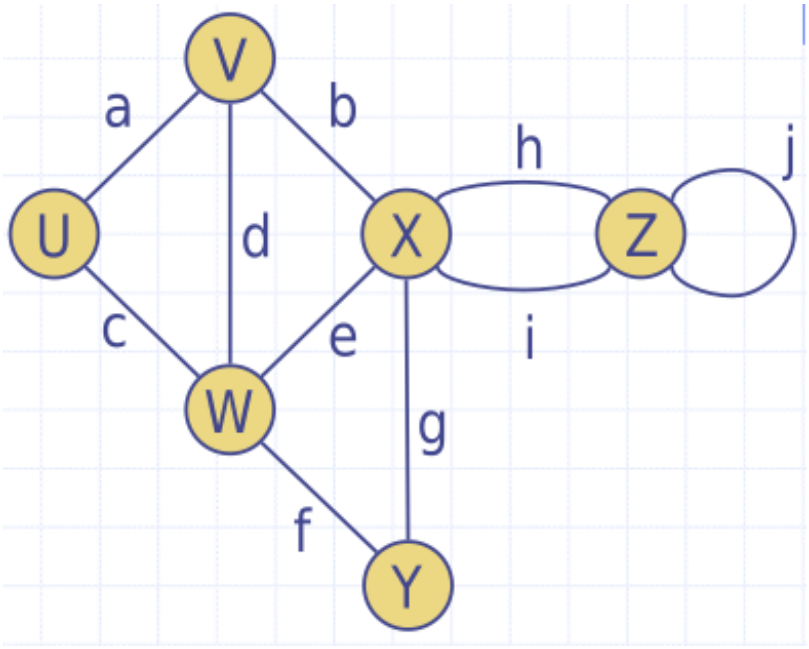


Figure 3: Graph Example

◆ n Vertizes, m Kanten ◆ keine parallelen Kanten ◆ keine Schleifen	Kanten Liste	Adjazenz Liste	Adjazenz Matrix
Space	$n+m$	$n+m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Figure 4: Performance Summary for Graph Data Structures

Edge List Structure

To represent a Graph you could use the Edge list Structure.
The Edge List Structure stores the nodes in a list. The Edge objects are also stored in a list. Each edge object holds a reference to the start vertex and to the end reference.

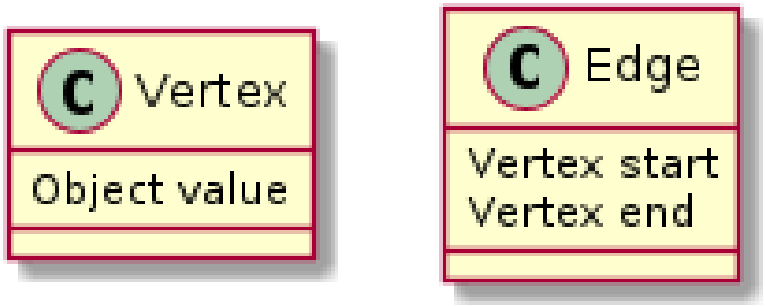
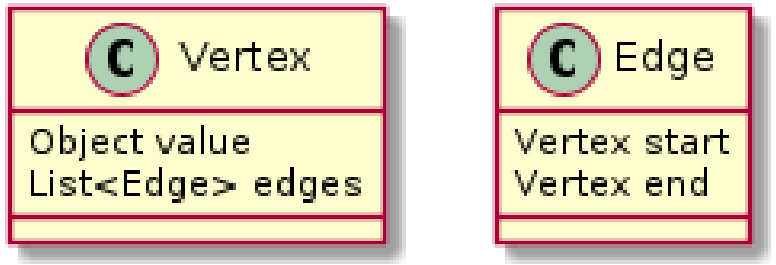


Figure 5: Test

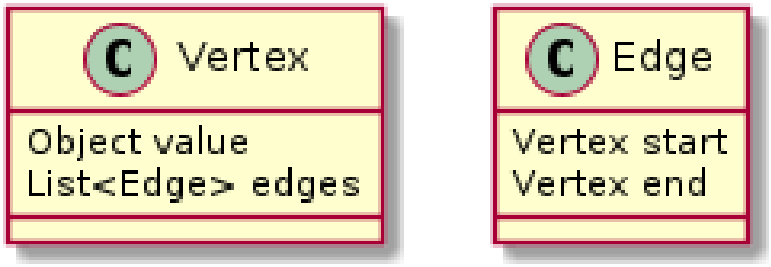
Adjacent list Structure

To represent a Graph you could use the Adjacent list Structure.
The Adjacent List Structure stores the nodes in a list. The Edge objects are also stored in a list. Each edge object holds a reference to the start vertex and to the end reference. The Vertex object holds a list with references to the edge objects.



Adjacent Matrix Structure

To represent a Graph you could use the Adjacent Matrix Structure. The Adjacent Matrix Structure stores the nodes in a list. The Edge objects are also stored in a list. Each edge object holds a reference to the start vertex and to the end reference. Additionally you have a $n \times n$ matrix / array. The indexes represent vertexies in the vertex list. If a edge exists between node 0 and node 1 then in $M[0][1]$ a edge object is stored. If no connection exists the null value is there.



Subgraph
A **Subgraph** S of G is Graph which contains a set subset of vertices and edges of the Graph G. A **Spanning Subtree** S of G is a graph which contains all vertices of the graph G

Special Kind of Graph / Trees
A free tree T is a undirected graph (Graph) so that T is conected an has no cycles. A forest is a graph without cycles A Spanning Tree of a connected graph is a Subgraph which is also tree

Transitory Clousre
The Transitory closure of a Digraph (Graph) G is another Digraph $G*$ which:

- has the same vertices as G
- when in G a directed path betwenn u and v exists ($u \neq v$) then has $G*$ a directed edge from u to v

To calculate the Transitory Closure the The Floyd-Warshall Algorithm is used.

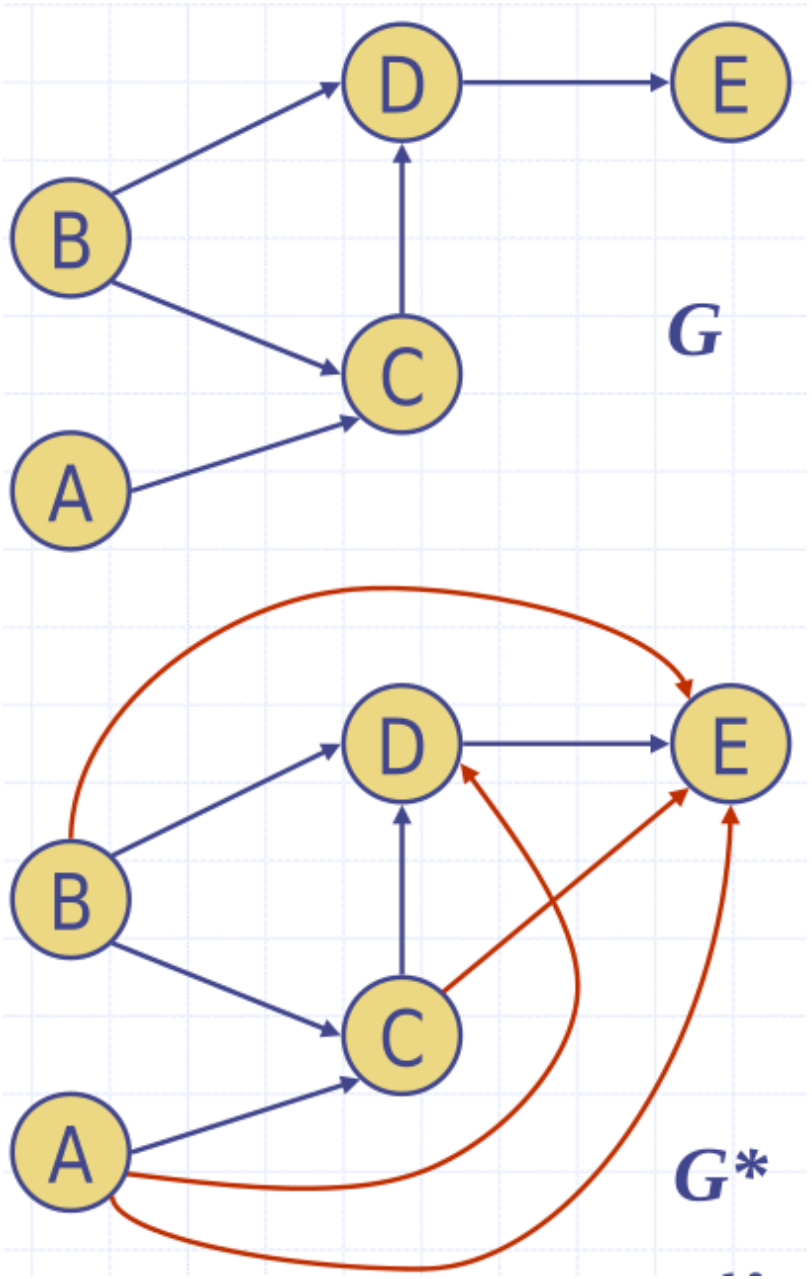


Figure 6: Transitory Clousre

DAG
A DAG is directed acyclic graph. A Graph without directed cycles.

2 Algorithms

Merge Sort

Running time $O(n \log(n))$

Merge Sort uses the Divide-and-Conquer principle. It divides the list into 2 splits and the each list again into two lists until the list length is 1. After that the two lists are merged. The resulting lists are merged again until you have the sorted list.

```
public static <T extends Comparable<? super T>> T[] mergeSort(
    T[] s) {
  int n = s.length;
  if (n > 1) {
    T[] s1 = Arrays.copyOfRange(s, 0, n/2);
    T[] s2 = Arrays.copyOfRange(s, n/2, n);
    s1 = mergeSort(s1);
    s2 = mergeSort(s2);
    s = merge(s1, s2);
  }
  return s;
}

static <T extends Comparable<? super T>> T[] merge(T[] a, T[]
    b) {
  T[] s = newInstance(a, a.length * 2);
  int ai = 0; // First Element in 'Sequence' A
  int bi = 0; // First Element in 'Sequence' B
  int si = 0; // Last Element in 'Sequence' S
  while (!(ai == a.length) && !(bi == b.length)) {
    if (a[ai].compareTo(b[bi]) < 0) {
      s[si++] = a[ai++];
    }
    else {
      s[si++] = b[bi++];
    }
  }
  while (!(ai == a.length)) {
    s[si++] = a[ai++];
```

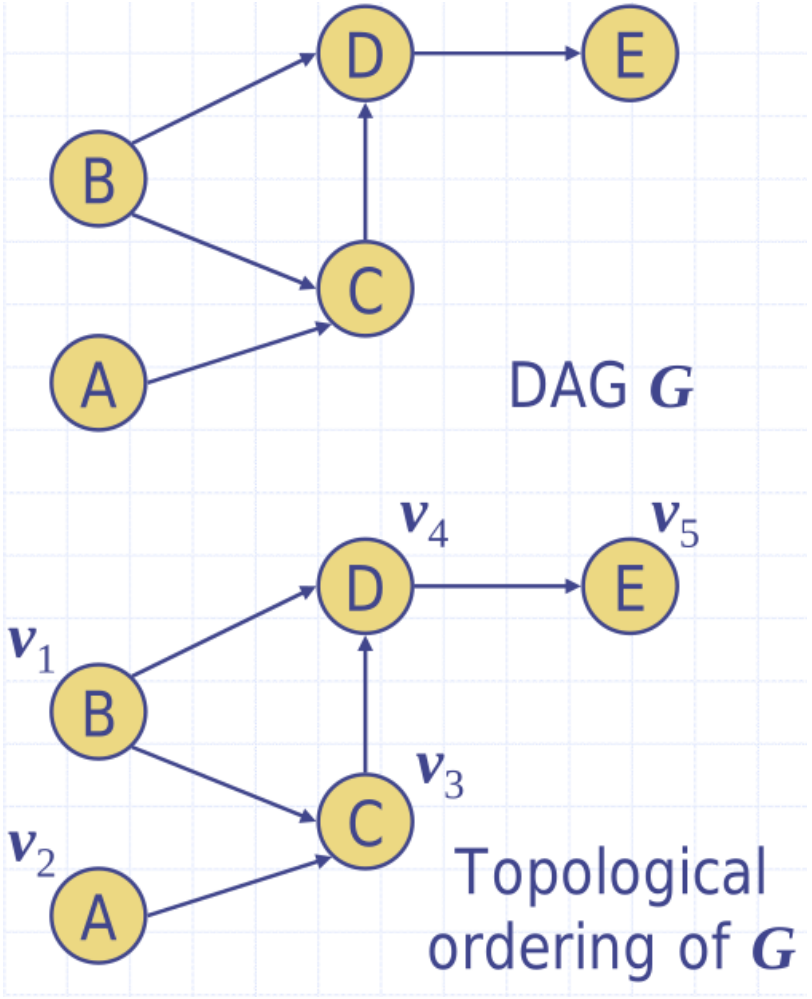


Figure 7: DAG Example

```
  }
  while (!(bi == b.length)) {
    s[si++] = b[bi++];
  }
  return s;
}
```

Algorithmus	Zeitverhalten	Bemerkungen
selection-sort	$O(n^2)$	in-place langsam (gut für kleine Inputs)
insertion-sort	$O(n^2)$	in-place langsam (gut für kleine Inputs)
quick-sort	$O(n \log n)$ expected	in-place schnellster (gut für grosse Inputs)
heap-sort	$O(n \log n)$	in-place schnell (gut für grosse Inputs)
merge-sort	$O(n \log n)$	sequentieller Datenzugriff schnell (gut für riesige Inputs)

Figure 8: Runtime Summary

Quick Sort

running time $O(n \log(n))$ expected

Quick Sort uses the Divide-and-Conquer principle.

1. A random element x is choosen (the **pivot**).
2. Divide
   (a) All elements smaller than x go to S
   (b) All elements greater than x go to G
   (c) All elements equal x go to E
3. Repeate 2 for each list until list length 1
4. merge from lower to equal to higher

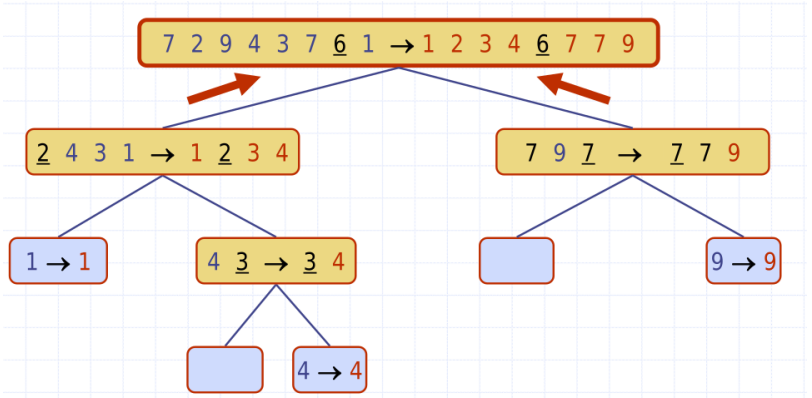


Figure 9: Quick Sort Tree

```
/*
 * @param a
 *          The leftbound of the part that shall be sorted.
 * @param b
 *          The rightbound of the part that shall be sorted.
 */
public <T> void inPlaceQuickSort(T[] sequence, Comparator<T>
    comp, int a, int b) {
    T temp;
    if (a >= b)
        return;
    T pivot = sequence[b];
    int l = a;
    int r = b - 1;
    while (l <= r) {
        while (l <= r && comp.compare(sequence[l], pivot) <=
            0) {
            l++;
        }
        while (r >= l && comp.compare(sequence[r], pivot) >=
            0) {
            r--;
        }
        if (l < r) {
            temp = sequence[l];
            sequence[l] = sequence[r];
            sequence[r] = temp;
```

```
            }
        }
        temp = sequence[l];
        sequence[l] = sequence[b];
        sequence[b] = temp;
        /*  Move index if sequence of equal elements. E.g.:
         *  ├──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┤
         *  |  |  | 5 | 5 | 2 | 8 | 8 | 8 | 8 |  |  |  |
         *  ├──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┼──┤
         *    a                  l <──────── l          b
         */
        while ((l > a) && (comp.compare(sequence[l], sequence[l
            -1]) == 0)) {
            l--;
        }
        while ((r < b) && (r > a) && (comp.compare(sequence[r],
            sequence[r+1]) == 0)) {
            r++;
        }
        inPlaceQuickSort(sequence, comp, a, l - 1);
        inPlaceQuickSort(sequence, comp, r + 1, b);
}
```

Lower Bound Search Algorithm

Every sorting algorithm which uses comparing has a minimal running time of $\log(n!)$. In O notation this is $O(n \log(n))$.

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\left(\frac{n}{2}\right) \tag{1}$$

$$n! \geq \sqrt{2\pi \cdot n}\left(\frac{n}{2}\right)^2 \to O(n \log n) \tag{2}$$

Bucket Sort

running time $O(n + N)$

Bucket Sort is Sorting algorithm which does not use any compares to achieve the sorting.
To sort something with bucket sort you need a key. This key must have a specific range $[0, N-1]$. Then you create an array of N empty lists. Then you move the value from initial unsorted list into the array. The key of the element specifies the location in the array. After the initial list is cleared you filled again by iterating over the array.

```
public List<T> bucketSort(List<T> S, int N) {
    List<T>[] buckets = new ArrayList[N];

    while (!S.isEmpty()) {
        T f = S.first();
        (k,o) = S.remove(f);
        buckets[k].insertLast((k,o));
    }

    for (int i = 0; i < N; i++) {
        while (!buckets[i].isEmpty()) {
            T f = buckets[i].first();
            (k,o) = B[i].remove(f);
            S.insertLast((k,o));
        }
    }
}
```

Radix Sort

Radix Sort is a sorting algorithm for lexicographical sorting based on Bucket Sort.

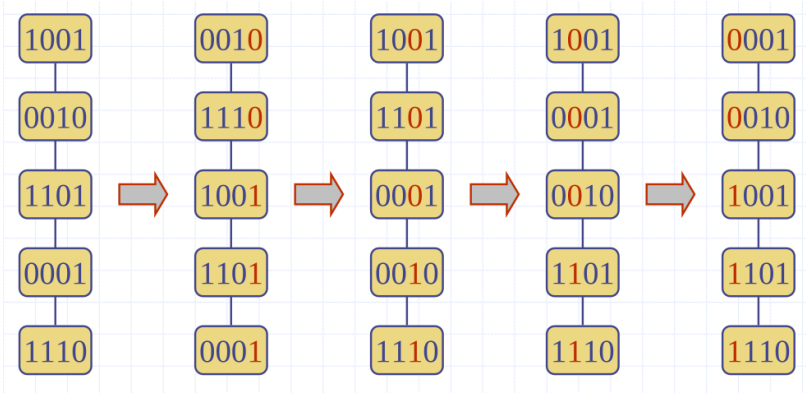


Figure 10: Radix Sort of a 4-Bit Integer

```
public void radixSort(String[] data) {
  // find max string-length
```

```
  int maxLength = -1;
  OptionalInt max = Arrays.stream(data).mapToInt(str -> str.
      length()).max();
  if (max.isPresent()) {
    maxLength = max.getAsInt();
  } else {
    return;
  }

  // bucketsort from max index to first index
  for (int i = maxLength - 1; i >= 0; i--) {
    bucketSort(data, i);
  }

}

protected void bucketSort(String[] data, int index) {
  Arrays.stream(data).forEachOrdered(str -> {
    if (str.length() <= index) {
      buckets[0].addLast(str);
    } else {
      buckets[str.charAt(index) - 'a' + 1].addLast(str);
    }
  });

  String[] phase2Array = Arrays.asList(buckets).stream().
      flatMap(List::stream)
      .toArray(String[]::new);
  System.arraycopy(phase2Array, 0, data, 0, phase2Array.length
      );
  Arrays.stream(buckets).parallel().forEach(list -> list.clear
      ());
}
```

Boyere-Moore

running time $O(n \cdot m + s)$

The Boyere-Moore Algorithm is used for Pattern Matching. Instead to compare the pattern from the beginning it starts at the end.

1. align start character of search string with pattern
2. compare the n-th character of pattern with search string
3. chars equal: $n = n - 1$ and go to 2.
4. not equal: move pattern forward
   (a) mismatched character occurs: move pattern forward to mismatched (last occurence function)
   (b) else: move whole pattern after the mismatched position

The last occurence funciton returns the index of the where the charachter x occurs last in the pattern (11).

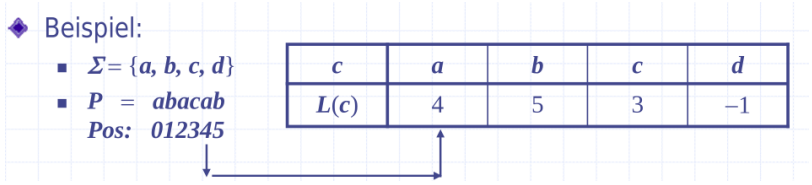


Figure 11: Last Occurrence Function

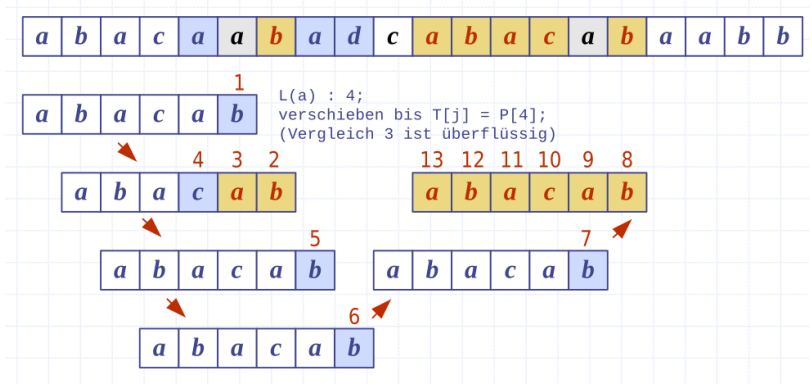


Figure 12: Boyere-Moore Algorithm

KMP

running time O(m + n)

The KMP Algorithm is a Pattern Matching Algorithm. KMP compares the pattern with text from left to right. But it moves the pattern smarter than Brute-Force.

Algorithm 1 Boyer-Moore Algorithm

```
1:  procedure BoyerMooreMatch(T, P, Σ)
2:      L ← lastOccurenceFunction(P, Σ)
3:      i ← m − 1
4:      j ← m − 1
5:      repeat
6:          if T[i] = P[j] then
7:              if j = 0 then
8:                  return i { match at i }
9:              else
10:                 i ← i − 1
11:                 j ← j − 1
12:             end if
13:             { charchter-jump }
14:             l ← L[T[i]]
15:             i ← i + m − min(j, 1 + l)
16:             j ← m − 1
17:         end if
18:     until i > n − 1
19:     return b { The gcd is b }
20: end procedure
```

The Algorithm has two phases:

1. calculating the **failure function**
   - Calculating the Failure Function for KMP
2. the actual pattern matching

1. If $T[i] = P[i]$
   (a) is pattern found, then return start position
   (b) otherwise increment i and j
2. otherwise if $j > 0$
   (a) then j will become the value from the $failureFunction(j)$
   (b) else $i = i + $

Example:
KMP according to figure 13:

1. the first 5 comparisons are matching
2. 6 does not match
   (a) check if 5 could be an prefix (failure function)
   (b) yes, the first character of the pattern must not be compared
3. 7 does not match
   (a) check error function
   (b) 0 → align pattern to current position
4. and so on

Failure Function KMP

running time O(m)

The Failure Function for KMP (KMP Algorithm) is defined as follows:

> $F(j)$ is defined as the longest prefix from P[0..j] so that this also is the suffix from P[1..j].

Example according to figure 14:

Algorithm 2 KMP Algorithm

```
1:  procedure KMPMatch(T, P)
2:      F ← failureFunction(P)
3:      i ← 0 { Location in search string }
4:      j ← 0 { Location in pattern }
5:      while i < m − 1 do
6:          if T[i] = P[i] then
7:              if j = m − 1 then { m length of pattern }
8:                  return i − m + 1 { match }
9:              else
10:                 i ← i + 1
11:                 j ← j + 1
12:             end if
13:         else
14:             if j > 0 then
15:                 j ← F[j − 1]
16:             else
17:                 i ← i + 1
18:             end if
19:         end if
20:     end whilereturn -1 { no match }
21: end procedure
```

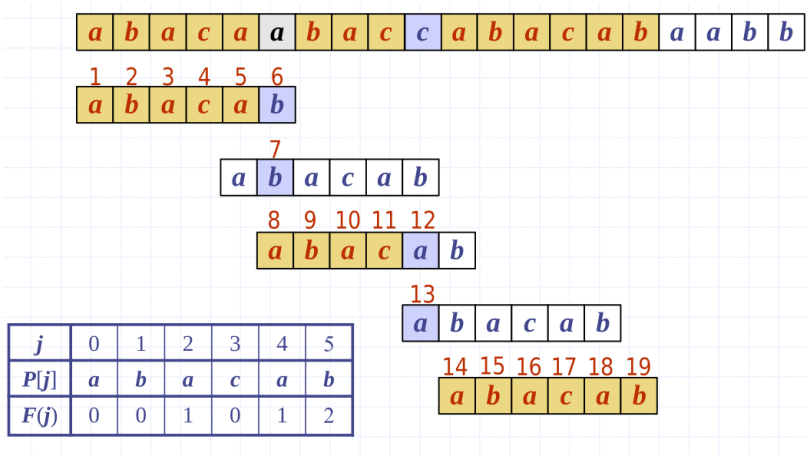


Figure 13: KMP Example

j = 0 a is only prefix, but not suffix
j = 1 b is only a suffix
j = 2,3 a is suffix (0) and a prefix
j = 4 Now the suffix is **ab**, but this could be also the prefix
j = 5 Now the suffix is **aba**, but this could be also the prefix

j	0	1	2	3	4	5
P[j]	a	b	a	a	b	a
F(j)	0	0	1	1	2	3

Figure 14: Failure Function for KMP

Tries
A **Trie** is a data structure (a tree Tree) to represent a set of strings. Tries are used to perform fast pattern matching.

Algorithm 3 KMP Failure Function

```
1:  procedure FAILUREFUNCTION(P)
2:      F[0] ← 0
3:      i ← 1 { Location in pattern }
4:      j ← 0
5:      while i < m do
6:          if P[i] = P[j] then
7:              F[i] ← j + 1
8:              i ← i + 1
9:              j ← j + 1
10:         else if j > 0 then
11:             j ← F[j − 1]
12:         else
13:             F[i] ← 0
14:             i ← i + 1
15:         end if
16:     end whilereturn -1 { no match }
17: end procedure
```

The Default-Trie / Standard-Trie has:

- each node except the root node has exactly one symbol
- Every path from the root to a leaf is a word in the set

The Default-Trie stores every symbol in a node. But sometimes a node has only one next node. This symbol and the next node can be merged together in one node.

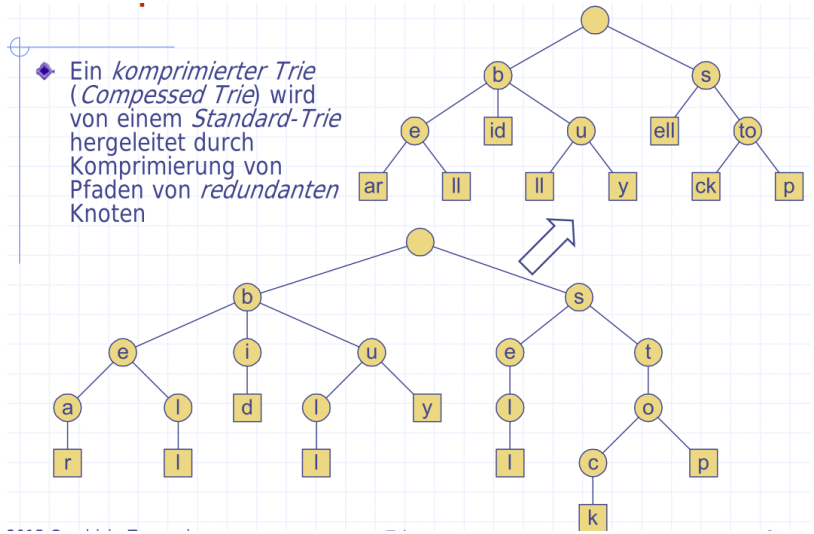


Figure 15: Example of Tries

The Suffix-Trie for a string is a compress trie (Tries) which stores all suffixes.

DFS

Depth-First search is a technique to traverse a Graph. As the name says you go deep and then wide (see 17).

DFS for finding path

DFS can be used to find a path between two vertices in a Graph. Every time you discovered a new edge, the edge is added to a stack. If you come back from this edge (return from the recursion) you remove it

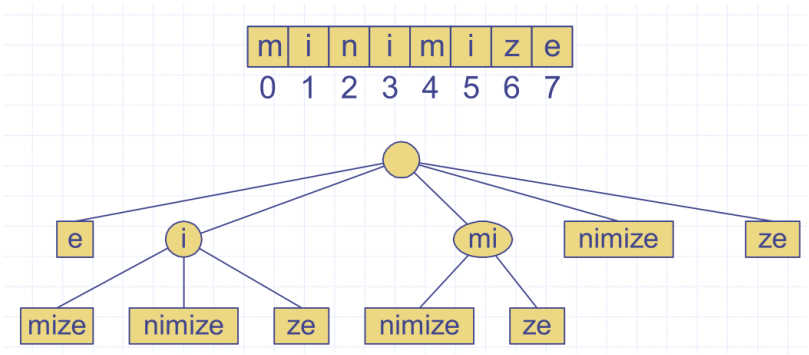


Figure 16: Suffix Trie

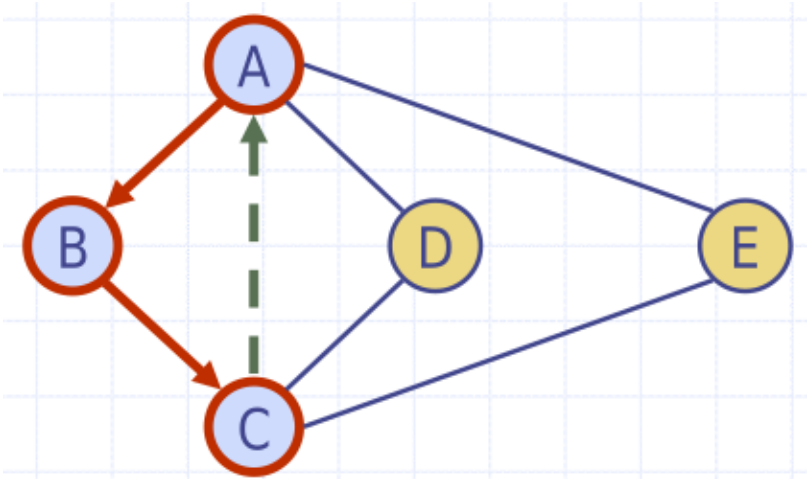


Figure 17: DFS Example

Algorithm 4 DFS Algorithm

```
1:  procedure DFS(G)
2:      for all u ∈ G.vertices() do
3:          setLabel(u, UNEXPLORED)
4:      end for
5:      for all e ∈ G.edges() do
6:          setLabel(e, UNEXPLORED)
7:      end for
8:      for all v ∈ G.vertices() do
9:          if getLabel(v) = UNEXPLORED then
10:             DFS(G, v)
11:         end if
12:     end for
13: end procedure
```

```
1:  procedure DFS(G, v)
2:      setLabel(v, VISITED);
3:      for all e ∈ G.incidentEdges(v) do
4:          if getLabel(e) = UNEXPLORED then
5:              w ← opposite(v, e)
6:              if getLabel(w) = UNEXPLORED then
7:                  setLabel(e, DISCOVERED)
8:                  DFS(G, w)
9:              else
10:                 setLabel(e, BACK)
11:             end if
12:         end if
13:     end for
14: end procedure
```

from the stack. If you finally found the searched vertex the path to this is stored in the stack.

Algorithm 5 Find path with DFS

```
1:  procedure pathDFS(G, v, z)
2:      setLabel(v, VISITED)
3:      S.push(v) { Stores the path }
4:      if v = z then
5:          finish: result is S.elements()
6:      end if
7:      for all e ∈ G.incidentEdges(v) do
8:          if getLabel(e) = UNEXPLORED then
9:              w ← opposite(v, e)
10:             if getLabel(w) = UNEXPLORED then
11:                 setLabel(e, DISCOVERY)
12:                 S.push(e)
13:                 pathDFS(G, w, z)
14:                 S.pop()
15:             else
16:                 setLabel(e, BACK
17:             end if
18:         end if
19:     end for
20:     S.pop()
21: end procedure
```

DFS For finding cycleso

DFS can be used to find cycles in a Graph. Using a stack you store the path from the start to the end vertex. DFS has found a cycle as soon as you would mark a edge with **BACK**.

BFS

Breath-First Search is a technique to traverse a graph. BFS visits one level after another. The start vertex is marked as visited. Then all adjacent vertexes are visited. BFS uses a Queue to store which vertex

Algorithm 6 Find Cycles with DFS

```
procedure cyclesDFS(G, v)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v) do
        if getLabel(e) = UNEXPLORED then
            w ← opposite(v, e)
            S.push(e)
            if getLabel(w) = UNEXPLORED then
                setLabel(e, DISCOVERY)
                cycleDFS(G, w)
                S.pop()
            else
                T ← newemptystack
                repeat
                    o ← S.pop()
                    T.push(0)
                until o = w
                finish: result in T.elements()
            end if
        end if
    end for
    S.pop()
end procedure
```

should be visited next.

DFS vs. BFS
A comparision of DFS and BFS.

Floyd-Warshall
The Floyd-Warshall algorithm is used to calculate the Transitory closure of a Graph G.
From v_5 to v_4 exists a directed path over v_1. There for are $j = 4, i = 5, k = 1$.

Topological Sorto
Topological Sort is used to sort / enumerate vertices so that for an edge (u, v) $u < v$ true is.
Topological Sort is just a DFS where you label the vertices.

Dijkstra
The Dijkstra Algorithm is used to calculate the distance to all vertices from a start vertex s in a Graph. Dijkstra has a few requirements:

- The graph is connected
- the edges are undirected
- the edge weight is not negative

You build a cloud of vertices. Beginning with the start vertex s you add one vertex after one to the cloud. With each vertex v we store a property which holds the distance from s to v. After each iteration we add the vertex with the lowest distance and update the distance of all adjacent vectors.

Bellman-Ford
The Bellman-Ford algorithm is use to caculate the distance from a vertex to every other vertex in a Graph. But other then the Dijkstra Algorithm the Bellman-Ford algorithm can also work with negative edge weights. Requirements:

- directed edges
- no negative weighted cycles

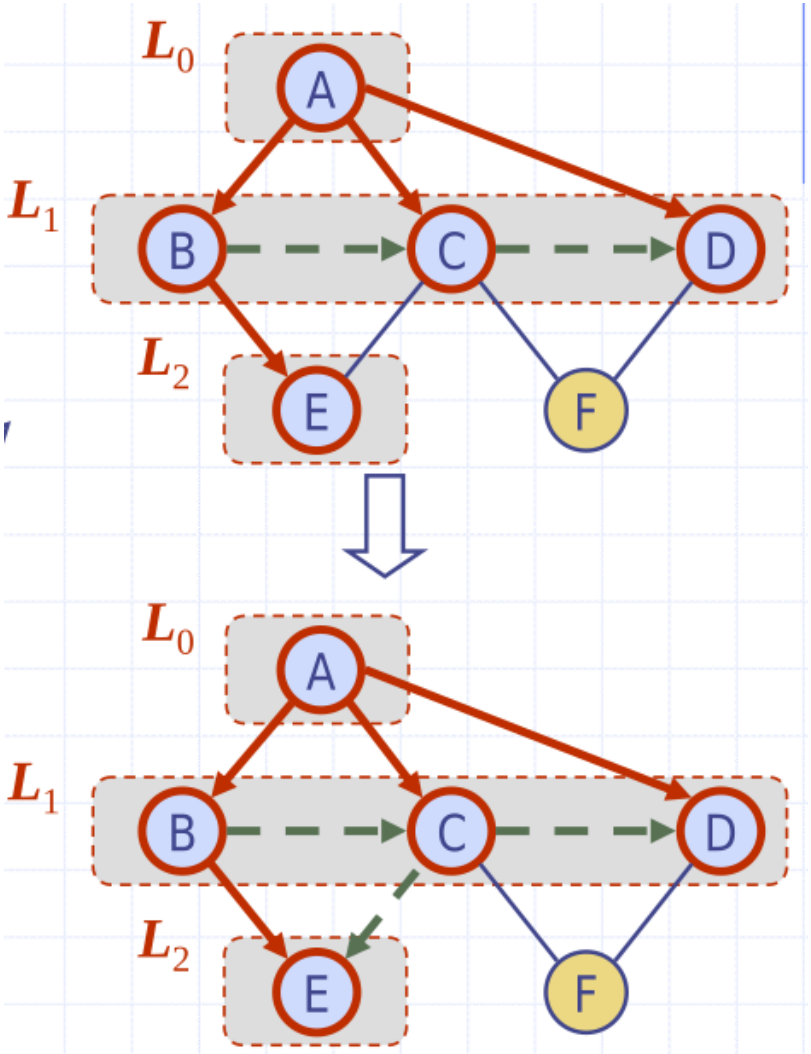


Figure 18: BFS Example

Algorithm 7 BFS Algorithm

```
procedure BFS(G)
    for all u ∈ G.vertices() do
        setLabel(u, UNEXPLORED)
    end for
    for all e ∈ G.edges() do
        setLabel(e, UNEXPLORED)
    end for
    for all v ∈ G.evertices() do
        if getLabel(v) = UNEXPLORED then
            DFS(G, v)
        end if
    end for
end procedure
procedure BFS(G, s)
    L_0 ← newemptysequence
    L_0.insertLast(s)
    setLabel(s, VISITED)
    i ← 0
    while ¬L_i.isEmpty() do
        L_{i+1}newemptysequence
        for all v ∈ L_i.elements() do
            for all e ∈ G.incidentEdges(v) do
                if getLabel(e) = UNEXPLORED then
                    w ← opposite(v, e)
                    if getLabel(w) = UNEXPLORED then
                        setLabel(e, DISCOVERY)
                        setLabel(w, VISITED)
                        L_{i+1}.insertLast(w)
                    else
                        setLabel(e, CROSS)
                    end if
                end if
            end for
        end for
    end while
    i ← i + 1
end procedure
```

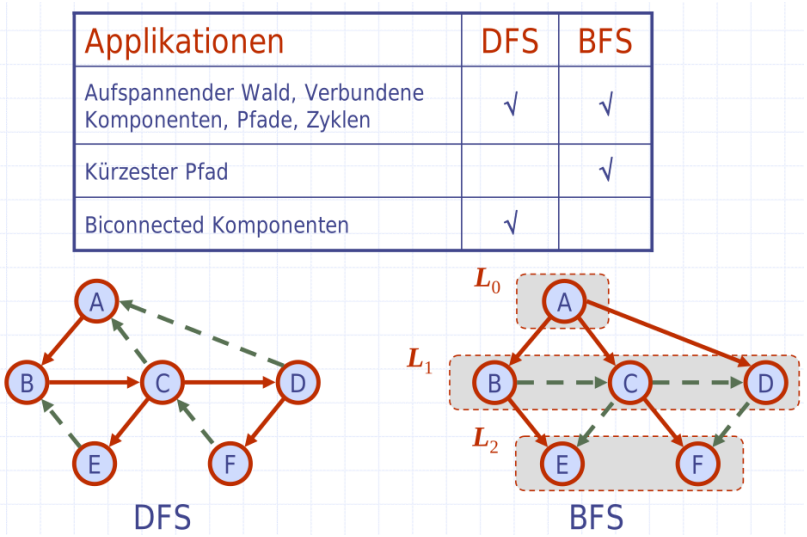


Figure 19: DFS vs. BFS

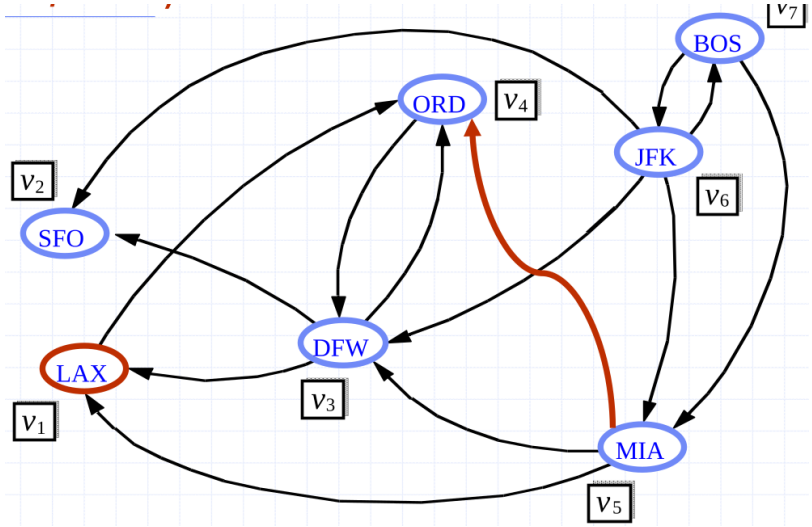


Figure 20: Floyd-Warshall Example

Algorithm 8 Floyd-Warshall's Algorithm

```
procedure FloydWarshall(G)
    i ← 1
    for all v ∈ G.vertices() do
        denotevasv_i
        i ← i + 1
    end for
    G_0 ← G
    for k ← 1 to n do
        G_k ← G_{k−1}
        for i ← 1ton(i ≠ k) do
            for j ← 1ton(j ≠ i, k) do
                if G_{k−1}.areAdjacent(v_i, v_k)∧G_{k−1}.areAdjacent(v_i, v_j)
then
                    if ¬G_k.areAdjacent(v_i, v_j) then
                        G_k.insertDirectedEdge(v_i, v_j, k)
                    end if
                end if
            end for
        end for
    end for
  return G_nend procedure
```

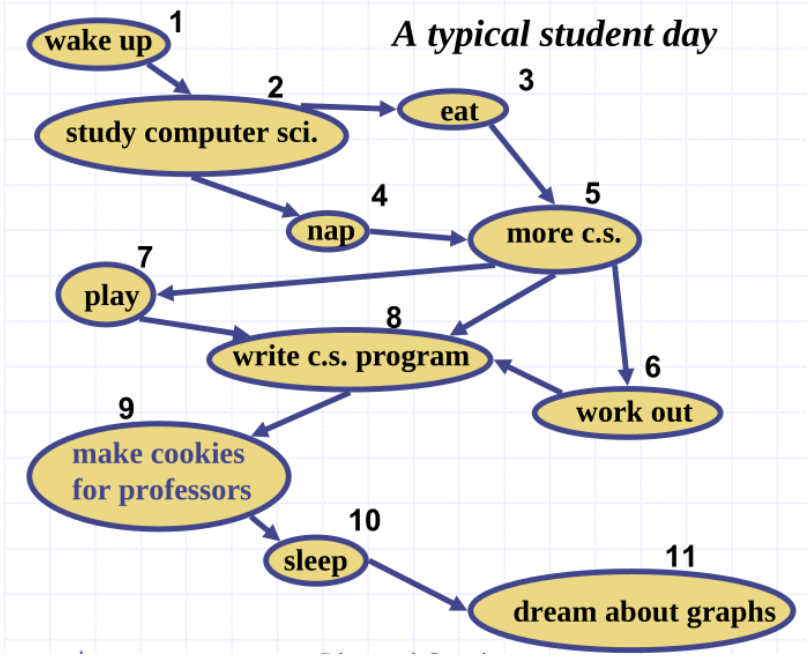


Figure 21: Topological Sort Example

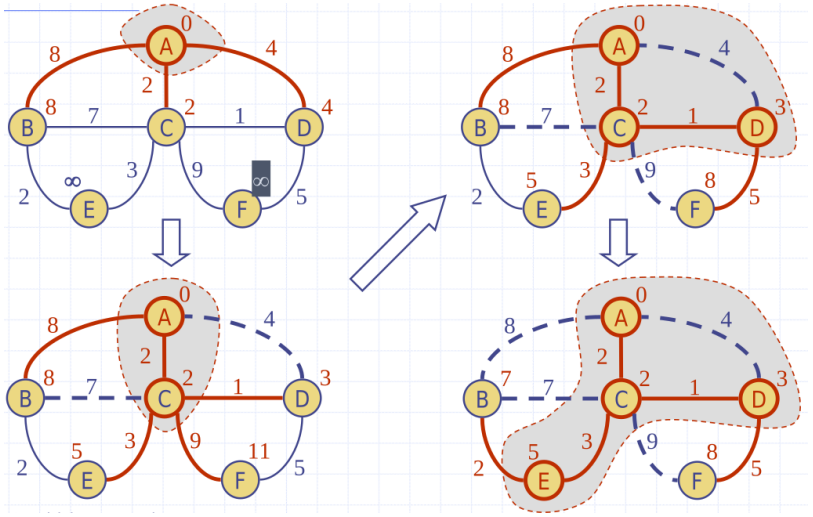


Figure 22: Dijkstra Algorithm Example

Algorithm 9 Topological Sort using DFS

```
1: procedure TOPOLOGICALDFS(G)
2:     for all u ∈ G.vertices() do
3:         setLabel(u, UNEXPLORED)
4:     end for
5:     for all e ∈ G.edges() do
6:         setLabel(e, UNEXPLORED)
7:     end for
8:     for all v ∈ G.vertices() do
9:         if getLabel(v) = UNEXPLORED then
10:            topologicalDFS(G, v)
11:        end if
12:    end for
13: end procedure
```

```
1: procedure TOPOLOGICALDFS(G, v)
2:     setLabel(v, VISITED);
3:     for all e ∈ G.incidentEdges(v) do
4:         if getLabel(e) = UNEXPLORED then
5:             w ← opposite(v, e)
6:             if getLabel(w) = UNEXPLORED then
7:                 setLabel(e, DISCOVERED)
8:                 topologicalDFS(G, w)
9:             else
10:                setLabel(e, n)
11:            end if
12:        end if
13:    end for
14:    n ← n − 1
15: end procedure
```

Algorithm 10 Dijkstra Algorithm

```
procedure DikstraDistance(G, s)
    Q ← new heap-based adaptable PQ
    { Initialization }
    for all v ∈ G.vertices() do
        if v = s then
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        end if
    end for
    while ¬Q.isEmpty() do
        u ← Q.removeMin().getValue()
        for all e ∈ G.incidentEdges(u) do
            { relax edge e}
            z ← G.opposite(u, e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z) then
                setDistance(z, r)
                Q.replaceKey(getLocator(z), r)
            end if
        end for
    end while
end procedure
```

Algorithm 11 Bellman-Ford Algorithm

```
procedure BellmanFord(G, s)
    for all v ∈ G.vertices() do
        if v = s then
            setDistance(v, 0)
        else
            setDistance(v, ∞)
        end if
    end for
    n ← G.numVertices()
    for i ← 1 to n − 1 do
        for all e ∈ G.edges() do
            u ← G.origin(e)
            z ← G.opposite(u, e)
            r ← getDistance(u) + weight(e)
            if r < getDistance(z) then
                setDistance(z, r)
            end if
        end for
    end for
end procedure
```

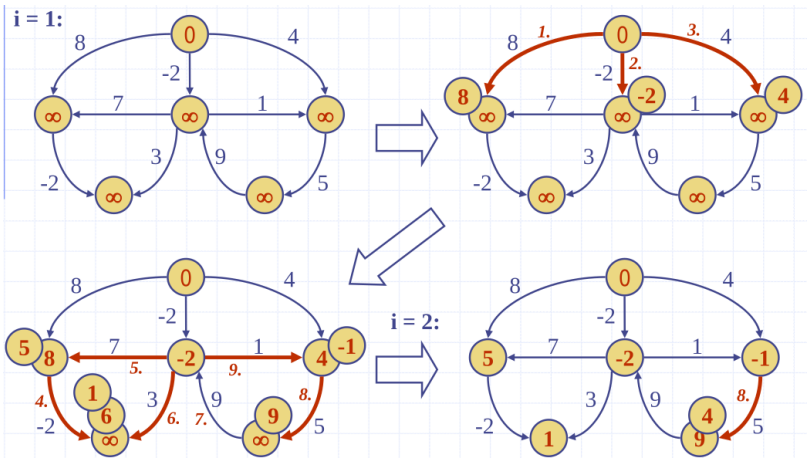


Figure 23: Bellman-Ford Example