Algorithms and Data structures Summary

Thijs van Loenhout

January 2020

Contents

1 Introduction

This is a summary for the course *Algorithms and Data structures* (NWI-IBC027) based on the year 2019 - 2020. I'll cover everything that is covered in the lectures, though I'll organize it slightly differently. Firstly, I'll discuss the various algorithms, then the various data structures and finally some problem solving strategies.

2 Big-O notation

This course is about analyzing algorithms to assess their correctness and efficiency. To achieve this, some notation for efficiency is needed: **Big-O notation**. For example, the function $f(n) = 2n^2 + 3$ would be written as $f \in \mathcal{O}(n^2)$. This represents an *asymptotic bound* for the function: the function f does not grow faster than n^2 does. It is also said that f has *order of magnitude* $\mathcal{O}(n^2)$. In practice, the order of magnitude is equal to the largest component of a function.

How would you know that $f(n) = 2n^2 + 3 \in \mathcal{O}(n^2)$? $\mathcal{O}(g)$ is defined as the following set of functions:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq c * g(n)\}$$

For our example: take $g(n) = n^2$. Notice that $f(n) = 2n^2 + 3 \leq 5n^2$ for all n, so that $c = 5$ would be a sufficient choice. Thus, f is in the set defined above and we can conclude that it has order of magnitude $\mathcal{O}(n^2)$.

A similar set of functions can be defined, but for asymptotic lower bounds:

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c * g(n) \leq f(n)\}$$

If a function is in the set $\Omega(g)$, it grows at a lower speed than g does. If f is both in $\mathcal{O}(g)$ and in $\Omega(g)$, then we say that f is in the set $\Theta(g)$: f and g grow at about the same rate. This set is not trivial: if f is some function of n, then there are loads of orders of complexity f belongs to. By convention, the smallest order of complexity f belongs to, is the one that is assigned to it.

So, we are talking about some function f that is dependent on a variable n. What is this variable? When analyzing the complexity of a function, all operations have a certain cost. In most cases, n will be the size of the input of your algorithm. When reasoning about your algorithm, you should figure out how much the total cost is (in a worst case scenario). An example: say we use the function `check_function` below to check if an element of the array `some_array` (with length `n`) occurs again later on by checking all other elements with a greater index.

```
bool check_function (some_array) {
    for (int i = 0; i < n-1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (some_array[i] == some_array[j]) { return true; }
        }
    }
    return false;
}
```

This function checks for all elements in `some_array` if any of the elements after it are the same. The 'for all elements' part (which corresponds to the outer for-loop) indicates n steps have to be made. Each step is a loop over all elements with a greater index (the inner for-loop), so for index i this means $n - i$ checks. We can easily compute the total cost by using Gauss' formula for sums:

$$n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2}$$

So that the complexity of this function would be in the order of n^2.

3 Algorithms

In this section, various algorithms are discussed. Algorithms are ‘recipes’ describing how something must be done. Generally, there is some kind of problem you want to solve with this algorithm.

3.1 Graphs

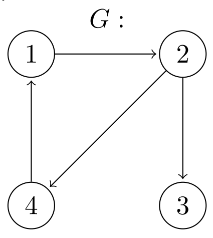
A lot of problems can be represented using graphs. A graph G is an ordered pair (V, E), where:

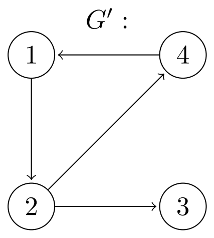
- V is a set of vertices. The size of V is denoted as $|V|$. I'll use ‘node’ as a synonym for ‘vertex’.
- $E \subset V \times V$ is a set of edges. An edge is a pair with two elements that represents a connection between two vertices. A pair $u, v \in E$ is a directional edge *from u to v*. Graphs with undirected edges have a pair $(u, v) \in E$ iff $(v, u) \in E$.

  Notice that the maximum number of edges is $\frac{n(n-1)}{2}$, so that $E \in \mathcal{O}(|V|^2)$

  Edges can have *weights* given by a wight function $w : E \to \mathbf{R}$. More on this topic in 3.1.1

An example of a graph: say $G = (\{1, 2, 3, 4\}, \{(1, 2), (2, 3), (2, 4), (4, 1)\})$. Then G can be presented like this:





The presentation of a graph does not matter. The graph G is the same as G'. Some definitions:

- If $(u, v) \in E$, then v is **adjacent** to u. v is also called a **neighbour** of u.
- A **path** is a sequence $(v_0, v_1, \ldots, v_k)$ of distinct vertices where each $(v_i, vi + 1)$ is an edge in E.
- G is called **connected** if there is a path between every pair of vertices

Examples of how a graph could be stored, is by use of a adjacency list or an adjacency matrix. An adjacency matrix of our example graph G would look like this:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	0	0	0
4	1	0	0	0

Table 1: An adjacency matrx

For an adjacency list, one can use a set of linked list where each list contains those vertices adjacent to the vertex that list represents. An adjecency list is more space efficient then a matrix, but determining whether a certain edge is part of the graph is not very efficient, while this is $\mathcal{O}(1)$ in a matrix.

3.1.1 Weighted graphs

As mentioned earlier, a graph $G = (V, E)$ can be a weighted graph. This means that each edge has a certain weight. Formally, there is a weight function $w : E \to \mathbf{R}$ that maps a value to each edge $(v_i, v_j) \in E$. The weight of a path $p = (v_0, \dots, v_k)$ is defined as the sum of weights of the edges it contains:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

A **shortest path** $\delta(u, v)$ from u to v in a weighted graph is defined as the following:

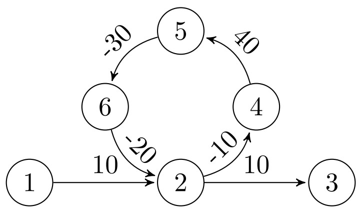
$$\delta(u, v) = min\{w(p) \mid p \text{ is a path from } u \text{ to } v\}$$

Also, $\delta(u, v) = \infty$ is there is no path from u to v.

A simple lemma:

Lemma (optimal substructure): A subpath from of a shortest path is a shortest path. (That is: if $p = (v_0, \dots, v_i, \dots, v_j \dots, v_k)$ is a shortest path from v_0 to v_k, then is $q = (v_i \dots, v_j)$ is a shortest path from v_i to v_j)

Notice that a shortest path is not always well-defined. Consider this example with a cycle of weight:



Notice that we can always find a shorter way to move from 1 to 3 (this may not seem like a problem if you take the 'distinct nodes' part of the definition of paths strictly. However, this problem was still addressed in the lectures and a intuition for this may be insightful)

Now, we'll look at some graph-searching algorithms. The first two will focus on unweighted graphs (though they could easily be expanded to weighted graphs).

3.2 Breadth-first search

Breadth-first search is an algorithm that takes a graph and traverses it breadth-first. This means that there is a set of nodes we've already looked at, and in the next step we'll look at the neighbours of nodes in this set.

- **input**:
  - A graph $G = (V, E)$ (either directed or undirected).
  - A source vertex $s \in V$.

- **output**:
  - A distance function $d[v]$, the length of the shortest path from s to v. $d[v] = \infty$ means that v is not reachable from s.
  - A predecessor function $\pi[v] = u$ such that $(v, u) \in E$ is the last edge in the shortest path from s to v. u is called the predecessor of v.

In this course, undiscovered nodes are labeled 'white', discovered nodes 'gray' and finished nodes 'black'. A node is finished when all of its neighbours are discovered. Pseudocode looks like this:

```
BFS(Graph G, source s)
  for each vertex u in V[G]\{s}
    do color[u] ← white
       d[u] ← $\infty$
       $\pi$[u] ← nil
  color[s] ← gray
  d[s] ← 0
  $\pi$[s]← nil
  Queue Q ← $\emptyset$
  while Q $\neq \emptyset$
    do u ← dequeue(Q)
      for each v $\in$ Adj[u]
        do if color[v] = white
          then color[v] ← gray
               d[v] ← d[u] + 1
               $\pi$[v]← u
               enqueue(Q,v)
      color[u] ← black
```

Let's apply BFS to the example given in 3.1. Say the source in node 1. The last graph is a 'BFS graph'. The number inside the nodes represent their predecessor. The numbers on the edges represent the distances to the source

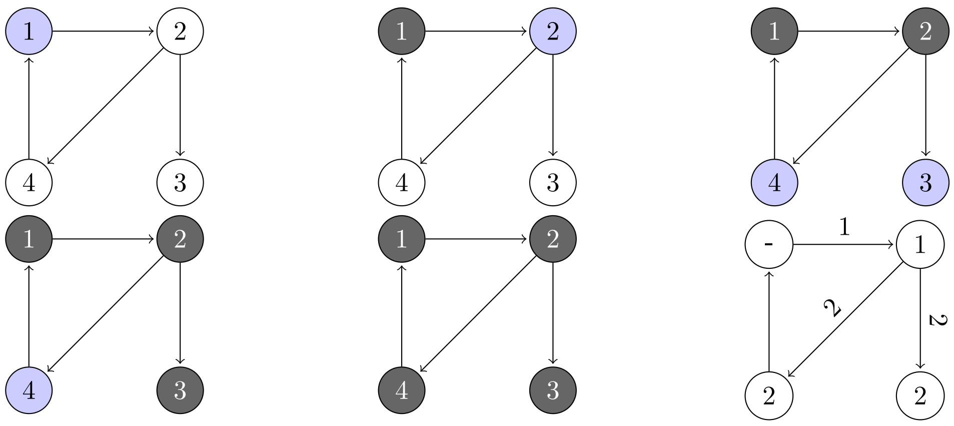


Figure 1: BFS applied to the example given in 3.1.

An advantage of breadth-first search, is that the moment you reach a node, you can be certain that you have taken a shortest path from the source to that node.

BFS has a worst-case runtime complexity of $\mathcal{O}(|V| + |E|)$ (verify this for yourself!), where V is the set of vertices and E the set of edges.

3.3 Depth-first search

Depth-first search is very similar to breadth-first search. This algorithm also takes a graph, but instead of traversing it breadth-first, it traverses it (you guessed it) depth-first. This means that instead of expanding the 'frontier' of our discovered set equally on all fronts, one path is traversed until there is no more vertex to add to it. When such a dead end is reached, the last node is finished. This can be recorded by a finishing time. A node is discovered the first time it is reached.

- **input**:
  - A graph $G = (V, E)$ (either directed or undirected)
- **output**:
  - A timestamp $d[v]$ on each vertex: the discovery time of v
  - A timestamp $f[v]$ on each vertex: the finishing time of v
  - A predecessor function $\pi[v] = u$ such that v was discovered during the scan of u's neighbours.

In pseudocode, DFS looks like this:

```
1  DFS(G)
2    for each vertex u ∈ V[G]
3      do color[u] ← white
4         π[u] ← nil
5    time ← 0
6    for each vertex u ∈ V[G]
7      do if color[u] = white
8         then DFS_visit(u)
```

```
1  DFS_visit(u)
2    color[u] ← gray
3    time ← time + 1
4    d[u] ← time
5      for each v ∈ Adj[u]
6        do if color[v] = white
7             then π[v] ← u
8                  DFS_visit(v)
9      color[u] ← black
10     time ← time + 1
11     f[u] ← time
```

Notice that BFS needs a start point s from which you expand. DFS as presented here (and in the course), does not have this argument: it continues until all nodes have been discovered (lines 6 to 8 in `DFS(G)`). However, when you discover a node with BFS, you are guaranteed that a shortest path from the source to this node was taken. Aside from not having a source, expanding DFS-style from a node does not guarantee that every node you visit along the way is on a shortest path from where you started to this node.

Altough DFS is very similar to BFS, it has a few addition characteristics in the classification of edges:

- **Tree edge**: an edge to an undiscovered node.
- **Back edge**: an edge to a discovered node
- **Forward edge**: an edge to a finished node
- **Cross edge**: any other edge

Just like BFS, DFS has a worst-case runtime complexity of $\mathcal{O}(|V| + |E|)$.

3.3.1 Topological sort

Say we want to model a structure with a partial order: $a > b$ and $b > c$, then $a > c$, but there may exist a and b so that neither $a > b$ nor $b > a$. This structure can be represented in a directed, acyclic graph (**DAG**). Canonical examples are about things that must happen before other things, like the order in which to put on clothes.

Say we want to determine an order of all nodes, so that none of the set orderings are violated. For this, one can use topological sort. It is very straightforward:

1. Run DFS
2. When a node is set to finished, add it in the front of a linked list
3. Return the list

The result is a list A with elements ordered in such a way that if $a < b$, a occurs before b in A. Topological sort has the same complexity as DFS.

3.3.2 Strongly connected components

In a graph, a **strongly connected component** (SCC) in a graph $G = (V, E)$ is a subset of nodes S so that for every pair of vertices (u, v) in S, u is reachable from v and vice versa. It may be useful to determine the strongly connected components in a graph. This can be done in the following manner:

1. Run DFS on $G = (V, E)$ and record the finishing times $f(u)$ for every node $u \in V$
2. Compute $G^T = (V, E^T) = (V, \{(u, v) : (v, u) \in E\})$
3. Run DFS on G^T, but consider vertices in order of decreasing $f(u)$ (as computed in thep 1)
4. Output the vertices in each tree of the depth-first forest formed in the second run of DFS as a seperate SCC.

The complexity of this algorithm is $\Theta(|V| + |E|)$

3.4 Dijkstra's algorithm

On the topic of shortest paths, say we want to find them in a weighted graph (Section 3.1.1. Moreover, say we want to find the shortest distance from a source node s to any other node in the graph. A problem like this can be answered by Dijkstra's algorithm.

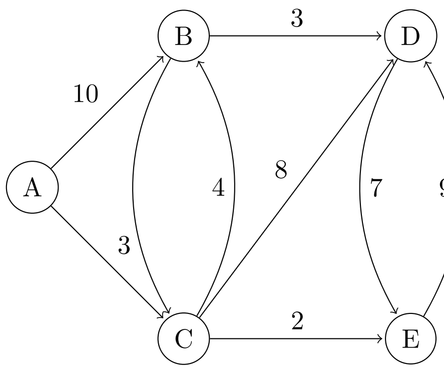
- **input:**
  - A *weighted* graph $G = (V, E)$
  - Source node $s \in V$
- **output:**
  - For all $v \in V$, a shortest path $\delta(s, v)$ from s to v.

In pseudocode, Dijkstra's algorithm looks like this:

```
$d[s] \leftarrow$ 0
for each $vinV \setminus \{s\}$
    do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow$S
while $Q \neq \emptyset$
    do $u \leftarrow$ Extraxt-Min($Q$)
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$
            do if $d[v]$ > $d[u]$ + $w(u, v)$
                then $d[v] \leftarrow d[u] + w(u, v)$
```

The idea of the algorithm is to initialize every node with a distance of ∞. Then, it will take the source node as current node and update these values according to the edge weight. Then, it will look at the take a look at the updated node with the lowest value and considers that nodes neighbours, and so on. When a node is chosen as new current node, it is moved from Q to S, indicating that we do not want to consider it again. In applying Dijkstra's algorithm, it is often very useful to keep a table of values. As an example (which you can look up in the slides for a step-by-step walkthrough), verify that the following matrix with starting node A would end up with the table next to it:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	



The run-time complexity of Dijkstra's algorithm happens to be dependant on the type of data structure you use. The initialization loop is about $|V|$ operations, the `while` loop (line 6) runs about V times. The complexity of the loop is, however, affected by that of `Extract-Min` and the decreasing of the values. The worst-case if $\mathcal{O}(|E| + |V| \log |V|)$.

3.5 Bellman-Ford

As mentioned in the section on Dijkstra's algorithm, 3.4, this algorithm has a weakness: it does not work properly when there are negative weight cycles. The Bellman-Ford algorithm serves to also find all shortest paths from a source node s to all other nodes *or* determines that there is a negative weight cycle.

- **input**:
  - A weighted graph $G = (V, E)$
  - Source node $s \in V$
- **output**:

  - A notion whether G contains a negative weight cycle and if not, for all $v \in V$ a shortest path $\delta(s, v)$ from s to v.

I will not present pseudocode for this algorithm, as I think it would blur the idea. This is because it is actually very simple: maintain a list of distances just like with Dijkstra's algorithm. For every node, update this list by considering all edges. Start this process with node s. If you're checking edges for the last node (so the $|V|$-th time you pass all edges) and can update a distance, then there is a negative weight cycle in the graph.

As you need to do a pass trhough all nodes for every node, the complexity is $\mathcal{O}(n^2)$.

3.6 Heapsort

BFS and DFS were algorithms we used to traverse graphs. We used Dijkstra's algorithm and Bellman-Ford were used to find shortest paths. Now we turn to heapsort as an example of an algorithm that sorts something: a heap (what a convenient name the algorithm has, huh?). You can read about binary heaps in section 4.2. We'll be using max heaps.

- **input**:
  - An unsorted array A of size n
- **output**:
  - Array A, but sorted

The algorithm works in the following manner:

1. Build a max-heap from A using `BuildMaxHeap`
2. Swap the first and last element of the array
3. Reconstruct the tree so that it remains a max-heap using `MaxHeapify`
4. Unless the array is sorted, perform step 2 on an array without the last element (so the array you sort shrinks 1 in size every iteration)

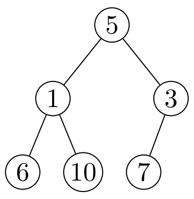
Pseudocode is given below. `MaxHeapify` may look scary, but it is actually very simple. For a node i let's assume the left and right branches are max-heaps on their own. `MaxHeapify` just determines the largest of i and i's left and right children and then swap i with this one if necessary. This may in turn damage the max-heap of the subtree that i was swapped to, so we need to fix it, again using `MaxHeapify`.

```
MaxHeapify(A,i)
    $l \leftarrow left(i)$
    $r \leftarrow right(i)$
    if $l \leq heapsize[A]$ and $A[l] > A[i]$
        then $largest \leftarrow l$
        else $largest \leftarrow i$
    if $r \leq heapsize[A]$ and $A[r] > A[largest]$
        then $largest \leftarrow r$
    if $largest \neq i$
        then exchange $A[i] \iff A[largest]$
             MaxHeapify($A, largest$)
```

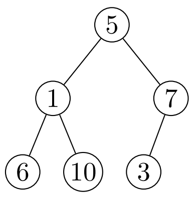
```
BuildMaxHeap($A$)
    $heapsize[A] \leftarrow length[A]$
    for $i \leftarrow \lfloor length[A]/2 \rfloor$ downto 1
        do Maxheapify($A, i$)
```

An example. Say we want to sort $A = [5, 1, 3, 6, 10, 7]$. Let's follow the algorithm step by step:

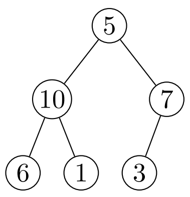
1. The first step is building a heap of A. As the size of A is 6, `BuildHeap` triggers 3 calls of `MaxHeapify`: first on 3, then on 2 and finally on 1 (note that these refer to the positions in the array, not the values that it contains). The (incorrect) heap beforehand would look like this:



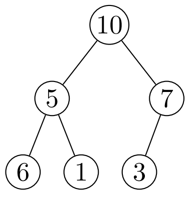
2. `MaxHeapify(A,3)` is called. The third node is the one that contains the value 3. The largest of 3 and 7 is 7, so 3 is swapped with 7 (node 6). `MaxHeapify(A,6)` is called, but this node has no children, so nothing changes. The heap now looks like this:



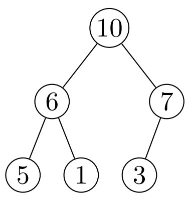
3. `MaxHeapify(A,2)` is called. The largest of 1, 6 and 10 is 10, so 1 is swapped with 10 (node 5). `MaxHeapify(A,5)` is called but again nothing changes. The heap now looks like this:



4. `MaxHeapify(A,1)` is called. The largest of 5, 10 and 7 is 10, so 5 is swapped with 10 (node 2), resulting in the following heap:



   This time, we do need to fix the heap. `MaxHeapify(A,2)` is called. The largest of 5, 6 and 1 is 6, so 5 is swapped with 6 (node 4):



`MaxHeapify(A,4)` is called, but this does not change anything.

5. The heap has been built! So the transformed array is $A = [10, 6, 7, 5, 1, 3]$. Swap 10 and 3 to get $A = [3, 6, 7, 5, 1, 10]$. Now, 10 is part of the sorted part of the array. We now want to repeat this process on the unsorted part. You can image (or even better: verify) that this will result in 7 as largest element, so that the sorted part consists on $[7, 10]$. Repeat this process until no more elements need to be sorted.

So, heapsort requires a lot of shuffling of nodes. The nice thing however, is that this can all be done in place: there is no extra space needed to perform the algorithm! `MaxHeapify` has a complexity of $\mathcal{O}\log n$, `BuildHeap` of $\mathcal{O}(n)$. So, the total runtime complexity is $\mathcal{O}(n \log n)$.

3.7 Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a bottom-up algorithm for calculating shortest paths between all vertices in a graph. It uses dynamic programming (5.2).

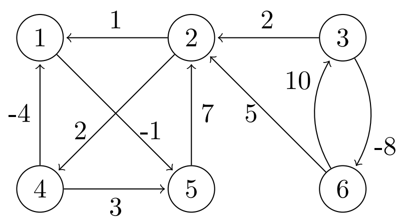
- **Input:**
  - A weighted graph $G = (V, E)$ with no negative weight cycles
- **Output:**
  - A matrix D^n with shortest paths between all pairs of vertices.

A naive way of obtaining this result, would be to run Bellman-Ford (3.5) on every node. However, this would result in a complexity of $\mathcal{O}(n^4)$. Instead, we're going to use **memoization** (see 5.2) to avoid double caclulations. In a matrix $D^{(k)}$, let an entry $d_{ij}^(k)$ denote the length of the shortest path from i to j that only goes through vertices $\{1, 2, \ldots, k\}$. Thus, the matrix with $k = n$ has the costs for the shortest path for the solution. Start with $D^{(0)}$ (the adjacency matrix that just records edge weights) and work your way towards $D^{(n)}$ using the following recurrence equation:

$$d_{ij}^k = min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

This may look daunting. This equation basically says 'hey, in our previous result, what where the shortest paths from i to k and from k to j? If we construct a path by glueing these parts together, would it be shorter than the current path that does not pass through k?'

Here is an example. Say we want to compute the shortest paths between all nodes for the following graph:



Here is what $D^{(0)}$, $D^{(1)}$ and $D^{(n)} = D^{(6)}$ would look like (calculating the intermediate steps is a good exercise!)

$$D^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{pmatrix}$$

$$D^{(6)} = \begin{pmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ -2 & -3 & 0 & 1 & -3 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 5 & 0 \end{pmatrix}$$

Constructing these matrices is really labor intensive, but if you stick to the recurrence equation of d_{ij}^k, you should be fine. As you may have noticed, the final matrix only record the lengths of the shortest paths. It is easy to extend the matrix to store predecessors as well, so that a path can easily be traced by looking at the next predecessor.

3.8 Minimum spanning trees

I will make the discussion of minumum spanning trees (MST) brief. First, the definition:

- A **minimum spanning tree** is a subgraph G' of a weighted, connected graph G by taking a subset of the edges of G such that:
  - All vertices are connected
  - There are no cycles (there is a tree structure)
  - The sum of weights of the edges in G' is minimal, while still satisfying the previous requirements

Two algorithms to obtain a MST have been discussed. Both are relatively simple, so I will mention them, but will not go into great detail. The slides provide a demo that is very clear, so I recommend to look at it. Note that a MST is not always unique.

3.8.1 Prim's algorithm

The first MST algorithm is Prim's. It works in the following manner:

1. Take a weighted, connected graph $G = (V, E)$. Pick a random starting vertex v and define a set of discovered vertices S as $\{v\}$.

2. Look at the set of edges with one point, say v_1, in S and the other, say v_2, not in S. Add the edge with the lowest weight to a set E' and add v_2 to S.
3. Repeat step 2 until $|E'| = n - 1$, where n is the total number of edges from G.
4. The graph $G' = (V, E')$ is a MST

3.8.2 Kruskal's algorithm

Kruskal's algorithm works in much the same manner as Prim's, and the result is also a MST:

1. Take a weighted, connected graph $G = (V, E)$. Initialize the set E' as empty
2. Look at the edges $\{e \in E \setminus E'\}$. Select the one with the following properties:
   - It is of minimum weight in $E \setminus E'$
   - Adding it to E' will not produce a cycle
3. Repeat step 2 until $|E'| = n - 1$, where n is the total number of edges from G.
4. The graph $G' = (V, E')$ is a MST

3.9 Network flow

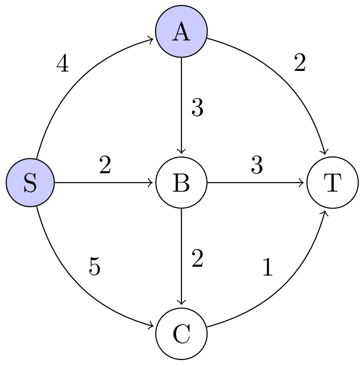
The algorithms presented in this section concern a special type of graphs: flow networks. A flow network is basically directed, weighted graph $G = (V, E)$ with a source s and a sink t. Edge weights are referred to as **capacity** and should be non-negative. There are many things one could image as a flow network. An intuitive example is one with literal flow: image a network of edges as network of pipes. Not every pipe is as small or large as others, so their capacity, the amount of water that can pass through them, is different. When you're pumping water through these pipes, not all have to be used to their full potential: an edge can let an amount n of flow pass through it, as long as $0 \leq n \leq$ its capacity.

Some definitions:

- A **st-cut** is a partition $A \cup B$ of the vertices with $s \in A$ and $t \in B$
- The capacity of a cut is the sum of capacities of edges from A to B: $cap(A, B) = \sum_{(v,w):v \in A, w \in B} c((v, w))$. (Where $c : E \to \mathbf{N}$ is the capacity function)
- An **st-flow** f is a function that satisfies:
  - For each $e \in E : 0 \leq f(e) \leq c(e)$ (no edge exceeds its capacity)
  - For each $v \in V \setminus \{s, t\} : \sum_{(e,v) \in E} f((e, v)) = \sum_{(v,e) \in E} f((v, e))$ (the value that flows into a node must equal that what flows out of it. s and t are special cases for which this does not hold)
- The **Value** of a flow f is defined as: $val(f) = \sum_{e:(s,e)} f(e)$, the sum of values that flows out of the source s

Algorithms concerning flow networks are mainly trying to solve two problems (that are actually the same). These are: the minimum-cut problem (finding a cut of minimum capacity) and the max-flow problem (finding a flow of maximum value). Luckily, an answer to one of the problems also provides you with an answer for the other.

An example is given below. The cut is $\{S, A\} \cup \{B, C, T\}$, with capacity 13. This cut is not minimal, for example the partition $\{S, A, B, C\} \cup \{T\}$ has capacity 6. which is smaller than 13. You can verify that 6 is also the maximum flow.

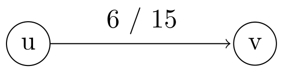


3.9.1 Ford-Fulkerson algorithm

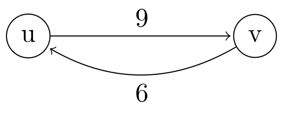
The example given in section 3.9 was small enough to see by eye what the maximum flow the network should be. This can be hard for larger network though. The Ford-Fulkerson max-flow algorithm is here to aid you with this. Problems concerning networks tend to look daunting, as they often have a big graph with lots of numbers. They are, however, a lot more friendly than their appearance would lead you to believe and the Ford-Fulkerson algorithm is very forgiving in how you apply it.

Firstly, some concepts you ought to get familiar with:

- For an edge $e = (u, v) \in E$, the capacity $c(e)$ is the maximum flow $f(e)$ that can pass through it. For a flow of 6 and a capacity of 15:



- A **residual edge** serves as a way to take flow back. If $e = (u, v)$ is an edge with capacity $c(e)$, then its corresponding residual edge is an edge $r = (v, u)$. Presented as labels on the edges are the residual capacity on e and the flow on r. For an edge (u, v) with capacity of 15 and a flow of 9:



- The **residual graph** $G_f = (V, E_f)$ of a graph $G = (V, E)$ is G alongside the residual edges with a value greater than 0 (including them won't affect anything, but omitting them produces a cleaner, easier-to-use graph)
- An **augmenting path** is a path $P = (s, \ldots, t)$ in the residual graph G_f

Now with those out of the way, let's talk about the algorithm. It is rather straightforward, but requires a lot of drawing, erasing and redrawing (or: tikz-ing and re-tikz-ing). Be aware of this: most mistakes in Ford-Fulkerson are made by not looking carefully at what you augment!

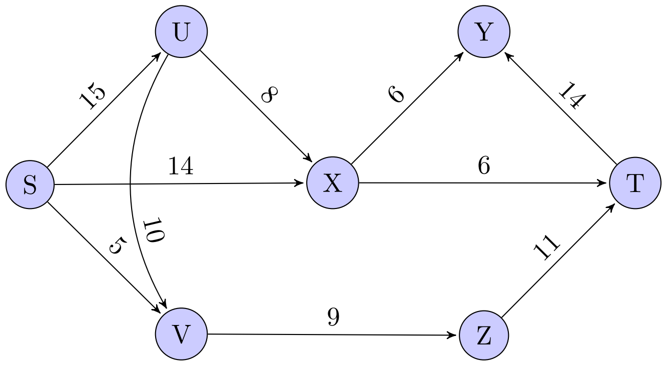
The algorithm:

1. For a graph $G = (V, E)$ with known capacities, initialize the flow $f(e)$ of every edge to be 0

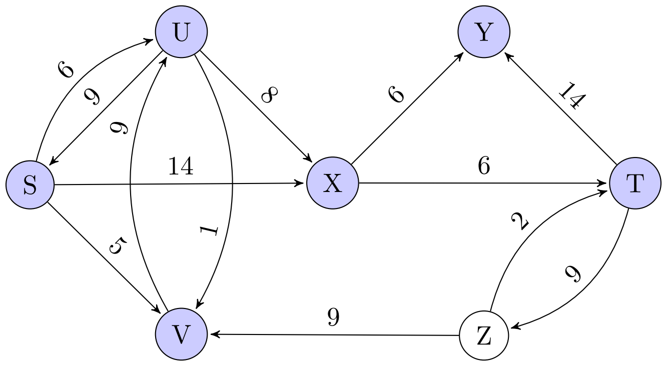
2. Construct (or update) the residual graph G_f
3. Find an augmenting path P in G_f
4. Augment the flow along path P
5. Repeat until you can no longer find a new path P

The result of this algorithm is a residual graph in which there is no path from source S to T. A minimum-cut can be made by dividing V in the partition $A \cup B$, where A contains all vertices that are reachable from S in G_f, and B all vertices that are not reachable from S in G_f. The capacity of T (in G, not in G_f!) is the maximum flow.

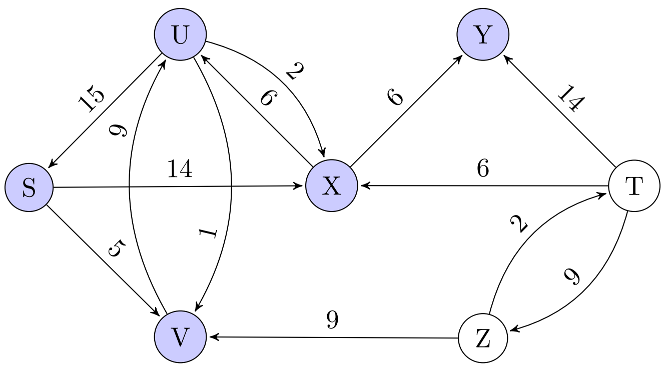
Here is an example. Consider the following graph G with source S and sing T. I will color all nodes we can reach from S, so we can easily see when we're done.



We must first look for a path from S to T. Any is fine and your choice won't affect the final result of the algorithm. For example, take the path $P_1 = (S, E, V, Z, Z)$. Notice how this path has a **bottleneck** of 9? This is the maximum flow we can send through this path. The residual graph would look like this:



Now, we need to find another path. Take $P_2 = (S, U, X, T)$ for example. This path has a bottleneck capacity of 6. The residual graph looks liek this:



There is no new path to be found, which means we're done! We can now decide what the minimum-cut and the maximum flow are. The minimum-cut is the partition $\{S, U, V, X, Y\} \cup \{T, Z\}$. So far, I've only shown the residual graph. However, every time we send flow trough an edge we could've updated G accordingly by adding this number. When we send flow trough a residual edge, we should have subtracted it. However, we did not use residual edges in this example, so the flow just equals the capacities of the two paths P_1 and P_2: $9+6=15$. (It is still a good exercise to verify this by updating G)

The complexity of Ford-Fulkerson is $\mathcal{O}(m*n*C)$, where m the number of edges, n the number of vertices and C the maximum capacity. This is a corollary of the **capacity-scaling theorem**.

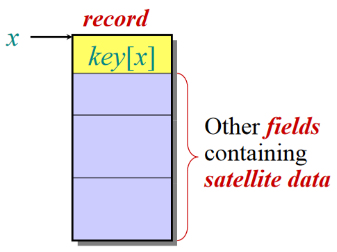
I mentioned before that it does not matter which augmenting paths you choose when applying Ford-Fulkerson. This is true, but during the exam, time may be of the essence, so are there ways to choose augmenting paths that are 'better' than others. 2 options were briefly discussed:

- The **capacity-scaling algorithm**. It is based on the intuitive idea to choose the augmenting path with the highest bottleneck capacity
- Shortest augmenting path: choose an augmenting path with the fewest number of edges. This is known as the **Edmonds-Karp algorithm**

And there are many others. However, they can all affect running time. In the lectures, we also discussed some applications of network flow. In particular, this can help you find a matching of maximum cardinality in a bipartite graph.

3.10 Hashing

Say we want a **symbol-table** S holding n records that looks like the figure to the right. How should the data structure S be organized? A solution would be a **direct access table**: suppose keys are drawn from a **universe of keys** $U \subset \{0, 1, \dots, m-1\}$, with distinct keys. Define an array $T[0 \dots m-1]$ such that:



$$T[k] = \begin{cases} x & \text{if } key(x) = k \\ NIL & \text{otherwise} \end{cases}$$

In other words, each key is distinct and is paired with a value in T: a pair (k, v) is mapped such that $T[k] = v$. A problem is that the range of keys can be incredibly large.

Hashing functions provide a solution: they map a value from U to a table T that is not necessarily the same size as or bigger than U. However, this has potential collisions as a result. It may be desirable to fix these, which is the main thing we'll discuss in this section. But first, how to choose a hash function? The *desirata*:

- A good hash function should distribute the keys uniformly into the slots of the table
- Regularity in the key distribution should not affect this uniformity

A very common theme in hash function is the usage of the modulo operation. Some hash functions were discussed in the lecture, but I will not state them here.

3.10.1 Resolving collisions

Two methods of resolving collisions were discussed in the lecture:

- **Resolving collisions by chaining**. A simple way to avoid collisions (or rather, to deal with them), is by making a slot in the table T a linked list. If a key maps to a slot that is already occupied, then we just chain it's value after the one that is already present. This has the disadvantage of having a poor access time of $\Theta(n)$, as every key could map to the same slot. However, if you're dealing with a decently uniform hash function, then the *average* complexity will be $\Theta(1+\alpha)$, where $\alpha = \frac{n}{m}$. This α is called the **load-factor**.
- **Resolving collisions by open addressing**. In this case, a hash function takes an additional argument alongside the key. A function h now maps $U \times \{0, 1, \ldots, m-1\} \rightarrow \{0, 1, \ldots, m-1\}$. The sequence $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1)\rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$. So, how do we avoid collision? By simply incrementing the second counter each time you **probe** unsuccessfully until you can insert into T. So, for example, if $T = [42, 37, NIL]$ and $h(1729, 0) = 1$, the probe is unsuccessful, as $T[1]$ is already occupied. So, increment the counter. Maybe $h(1729, 1) = 2$, so that we can than insert it in $T : T = [42, 37, 1729]$. A simple example of such a hash function would be: $h(k, i) = (k + i) \mod 3$. A more general format would be $h(k, i) = (h'(k) + i) \mod m$, where h' is an ordinary hash function.

  To delete something from this table, we cannot simply remove the entry, as this would destroy searches (image your looking for 1729 in our example after you have deleted 37. You'd figure that 1729 is not present, as $h(1729, 0) = 1$ and $T[1]$ is empty). The 'solution' is by replacing the to-be-deleted value with a special 'deleted'-marker (sometimes called a **tombstone**)

  In open addressing, the expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$, where α is again the load-factor.

3.11 Greedy algorithms

Greedy algorithms are not so much a strategy, but rather a category of algorithms. This category has an overlapping strategy though, so I've decided to include it in this section. The core is very simple: take a 'greedy' approach to reaching your goal. There is really not much more for me to say here. Generally, a greedy algorithm is very simple and straightforward. It's proving its correctess that is the hard part.

3.12 Randomized algorithms

As for greedy algorithms, randomized algorithms form more of a category of algorithms than a strategy to create them. The common thread through these algorithms is that there is some sort of randomness.

There are two kinds of randomized algorithms:

- A **Monte Carlo** algorithm always runs fast, but the output has a small chance of being incorrect
- A **Las Vegas** algorithm always outputs the correct answer. The expected running time is (still) small

The example given in the lectures was an $\mathcal{O}(n)$ algorithm (on average) for determining the k-th smallest element in a list. The random factor was randomly choosing a 'pivot' on which a split in the list was made.

It is not uncommon to combine this strategy (or any really) with others. The example in the lecture used divide and conquer aside from the random factor.

4 Data structures

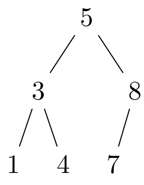
In the preceding section, we've talked about various algorithms. In the following sections, we'll turn our focus to data structures: ways to store information.

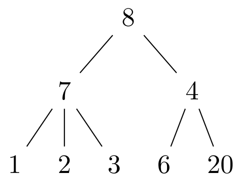
4.1 Binary search trees

Binary search trees are a special type of graphs that adhere to a specific structure:

- The graph is a (directed) tree
- Every node has at most 2 children
- Nodes in the left subtree of a node n contain values smaller than n. Nodes in the right subtree of n contain values larger than n

The result is a structure with a relatively small storage size on which we can perform simple operations. We can use the knowledge of the structure to our advantage. Here is an example of what it and what is **not** a binary search tree:





The left tree *is* a binary tree, while right one is not. For example, the latter has a node with 3 children, has the value 4 and 6 in the right subtree of a node with value 8, and more.

A binary search tree is called **complete** if all but the lowest layer of the tree is filled and the lowest layer is filled from left-to-right (so, the example on the left earlier is a complete binary search tree)

Some function on binary search trees were discussed. I will skip some (printing them and finding the minimum and maximum values), as they are rather obvious and do not help you understand these structures better. I will discuss the important ones: searching, inserting and deleting.

Two methods for searching in a BST:

```
recursiveSearch (x,k)
    if x == NIL or k == x.key
        return x
    if (k < x.key)
        return recursiveSearch(x.left,k)
    else
        return recursiveSearch(x.right,k)
```

```
iterativeSearch
    while x != NIL and k != x.key
        if k < x.key
            x = x.left
        else
            x = x.right
    return x
```

For insertion, simply traverse the tree like you would in the searching examples above, but insert the to-be-inserted node when you encounter `NIL` (at the bottom of the tree). Deletion is more involved, as you could disrupt the structure by removing an internal node. There are 3 cases:

- The to-be-deleted node `x` has no children (or to `NIL` children), then it can be safely deleted
- The to-be-deleted node `x` has one child, then we can safely remove `x` and make `x`'s parent the parent of `x`'s child (simply moving the branch up one node)
- The to-be-deleted node `x` has two children. Then, we need to find a successor for `x` for save deletion:

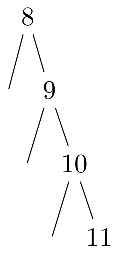
```
findSuccessor(x)
    if x.right != NIL
        return findSuccessor(x.right)
    y = x.parent
    while y != NIL and x == y.right
        x = y
        y = y.parent
    return y
```

Deletion can now safely be performed in the following manner: look at the node you want to delete (`x`). Look for its successor (`y`) and swap them. Look at `y`'s successor and swap `x` with it, and so on, until you encounter a `NIL` node. When that happens, you can safely remove `x`.

In the example from earlier: say we want to delete 1: no problem! Say we want to delete 3: 4 is its successor, so we swap 3 and 4. Now, we can safely remove 3 and the structure is still intact!

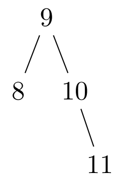
4.1.1 AVL trees

A problem with binary search trees, is that they could look something like the figure to the right (where I have drawn edges to `NIL` children as well). They can be very unbalanced, which in turn can lead to some bad (or at least: unreliable) complexities. We would like to have an extra condition that must be met, so that the trees are more balanced:



- The heights of the left and right children of a node, may differ by at most 1.

The result is a balanced binary search tree, also called an AVL tree (which stands for Adelson-Velsky and Landis). A balanced version of the tree to the right (above) would be that below it. More on the topic of balanced search trees in section 4.3.



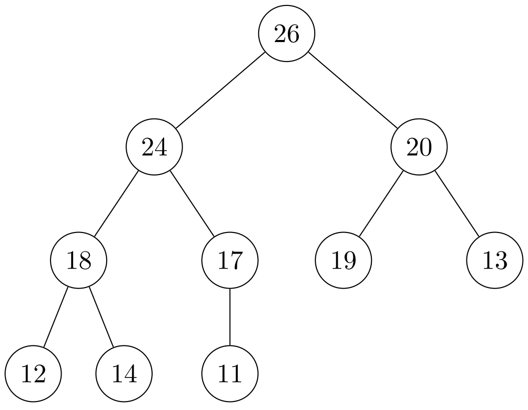
4.2 Binary Heap

Binary heaps are very similar to binary trees. They are logically a tree, but are 'physically' represented as an array, say A. This means that there is a mapping between the logical and physical representation:

- Root: $A[1]$ (we start at 1 rather than 0 for convenience reasons)
- Left of node i: $A[2i]$
- Right of node i: $A[2i+1]$
- Parent of node i: $A[\lfloor \frac{i}{2} \rfloor]$

There is a distinction to be made between to kind of heaps: a **max-heap** and a **min-heap**. For the former, the following rule must be followed: *for every node excluding the root, the value of the node is at most that of its parent.* So, the root is the largest element. A min-heap is the mirror-image: *for every node excluding the root, the value of the node is at least that of its parent.* So, the root is the smallest element.

An example: the array $[26, 24, 20, 18, 17, 19, 13, 12, 14, 11]$ has a logical representation as the following graph:



Notice that the last row is filled from left to right.

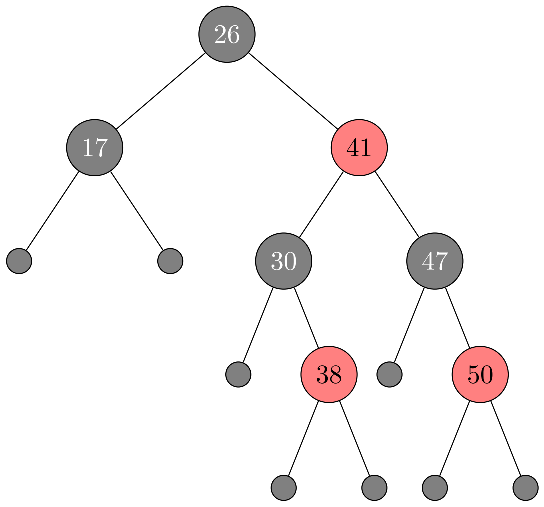
Some definitions and characteristics:

- The **height** of a node is the length of a path from that node to a leaf (so, the number of edges)
- The **height of a tree** is the height of the root.
- The **height of a heap** of an array of length n is $\lfloor \log n \rfloor$.
- The number of leaves is $\lceil \frac{n}{2} \rceil$

It may be desirable to insert and delete nodes into and from the heap. Both can be done with complexity $\mathcal{O}(\log n)$.

4.3 Red-black trees

In section 4.1.1, we discussed a balanced binary search tree: AVL trees. In this section, we'll discuss another one: red-black trees. These are trees that have the following properties:



- **All** properties of a binary search tree
- Every node is either red of black
- The root is black
- Every leaf (`NIL` node) is black
- If a node is red, then both its children are black
- For each node, all paths from it to a descendant leave contain the same number of black nodes.

An example is given to the right.
It is a good exercise to verify that this is, indeed, a red-black tree. It Is not necessary to always draw the leave nodes, but keep in mind they are still there! It is also possible to connect everything that would lead to a leave to *one* `NIL` node. Some definitions:

- The **height** $h(x)$ of a node x is the number of edges in a longest path to a leaf (just ilke with BSTs)
- The **black-height** $bh(x)$ of a node x is the number of black nodes from x to a leaf (including the leaf, but excusing x). Note that this property is well-defined beause of the last property of red-black trees
- The black-height of a red-black tree is the black-height of its root

In the example above, the height of the root is 4 and the black-height of the root is 2. So, the black-height of the tree is 2 as well. The height of the node containing the value 50 is 1. Its black-height is also 1.

You may ask: why would these kind of trees be more balanced than regular old binary search trees? It is in part because of the following lemmas:

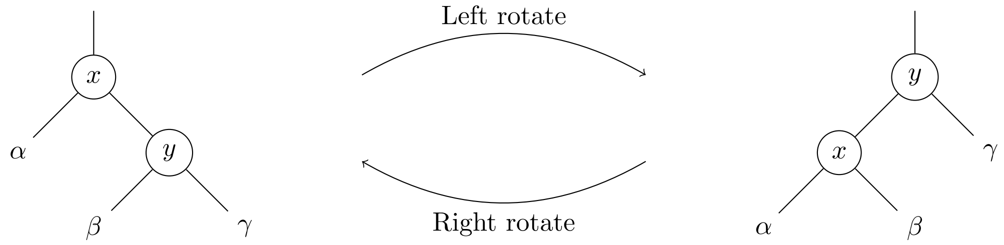
Lemma: Let x be a node in a RB tree. Then the height $h(x)$, the length of the the longest descending path from x to a leaf, is at most twice the length $s(x)$ of the shortest descending path from x to a leaf.

This is a corollary of the last property: all paths must contain an equal number of black nodes. As a red node can only have black children, it follows that $h(x) \leq 2bh(x) \leq 2s(x)$.

Lemma: The subtree rooted at any node s has $\geq s^{bh(x)} - 1$ internal nodes.

Lemma: A red-black tree with n internal nodes has height at most $2\log(n+1)$

I'll discuss a few properties of red-black trees, all of which can be performed in $\mathcal{O}(\log n)$ time. Searching is the same as in regular BSTs. Insertion and deletion require more care, as they could lead to a violation of the properties of RB trees. First, the concept of rotations:



Insertion and deletion make use of these tree-restructuring operations. Verify that the properties of a BST are not violated by these kind of rotations. Let's first look at insertion. The basic idea is this:

1. Use the insertion of a BST to insert a node x into our RB tree T
2. Color x red
3. Fix the modified tree so that no properties are violated

Deletion and insertion both have a lot of cases with standard ways to solve them. I recommend you to look at the slides for red-black trees to see these cases in detail. (I tried fitting them all in here, but aside from being messy, the Latex compiler complained about too much Tikz pictures haha).

5 Strategies

In this section, we'll look at some strategies to solve problems. As they are strategies, I can try to point out the patterns and summarize what they are about, but this can't compete with doing some exercises for yourself! Here is a quick list of strategies with an example for each:

- **Incremental**: insertion sort
- **Divide and conquer**: merge sort
- **Greedy**: Dijkstra's algorithm
- **Dynamic programming**: all-pairs-shortest-path
- **Randomized algorithms**: randomized quicksort

In the following sections, we'll discuss divide and conquer and dynamic programming. Here is a global difference:

- **Top-down approach**: divide a big problem recursively into smaller sub-problems. Solve them and combine the results.
- **Bottom-up approach**: Start with solving a small sub-problem and keep adding to it to work you way up to a bigger problem.

Generally, the latter is less intuitive, but more efficient. Divide-and-conquer generally uses a top-down approach with independent sub-problems. Dynamic programming generally uses a bottom-up approach with overlapping sub-problems.

5.1 Divide-and-conquer

Divide and conquer is a a strategy that is based on the idea of dividing a large problem into easier to solve sub-problems. Here are three steps:

1. Divide a problem into smaller instances of the original problem
2. Conquer these sub-problems
   - ... trivial for small sizes
   - ... dividing them again if they are still too large
3. Combining the solutions of the sub-problems into a solution of the original problem

As you may suspect, an implementation is typically recursive. An example is merge sort:

1. The problem is to sort a vector of values. Divide it into two vectors of similar size
2. Sorting is trivial for size 1, so keep dividing until this size is reached. Then, Recursively sort two vectors
3. Combine two ordered vectors into a new ordered vector

Problems were you are asked to give an algorithm using divide and conquer has often to do with some kind of list. Base cases are almost always a singleton list, so that it is required to divide the list until you have such a singleton list. Most of the time, the only way to split the list in equal sub-problems, is by dividing it in half repeatedly, so that a complexity with a log somewhere in it is almost inevitable.

5.2 Dynamic programming

Dynamic programming is an algorithm design strategy that depends on the idea of eliminating duplicate work. Take the Fibonacci numbers as an example. These can be recursively defined as:

$$\begin{aligned} F_0 &= 0 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

However, you can see that this is rather inefficient: if we calculate all parts separately, $F_n = F_{n-1} + F_{n-2} = (F_{n-2} + F_{n-3}) + F_{n-2}$, you can see that we would calculate F_{n-2} twice! In turn, this means that we would do a whole chain of calculations multiple times. Yikes! How can this be prevented? Well, notice that once F_{n-2} is calculated once, all other times we'd want to use it, we need the same value (the required return value is not dependant of the time at which the calculation is made). Thus, if we could store this value somewhere, we could later just reuse it. This optimization by storing previously calculated values is called **memoization**.

(*Note:* with Fibonacci numbers, only the previous two values are required to calculate the new one, so these are the only ones that need to be stored.)

Another example of dynamic programming is the Floyd-Warshall algorithm (section 3.7).

Just like with divide and conquer, we can capture the dynamic programming strategy in a sequence of steps:

1. Characterize the structure of the optimal solution
2. Recursively define the value for the optimal solution
3. Compute the optimal solution
4. Construct the optimal solution from the computed information

Often, a problem presents itself as a computation over a list. Where we recursively divided the list in divide and conquer strategies, we'll commonly look at the prefixes when using dynamic programming. The solution is then often presented as a recurrence equation with a base case and a recursive case (that may use memoization!). This latter case frequently uses $max(a, b)$.

6 Overview

In this section, I won't present anything new. It is meant as a quick reference to some of the complexities for operations and algorithms discussed in the course.

Algorithm or operation	Complexity	Factors
Breadth-first search (3.2)	$\mathcal{O}(n+m)$	Number of vertices n Number of edges m
Depth-first search (3.3)	$\mathcal{O}(n+m)$	Number of vertices n Number of edges m
Topological sort (3.3.1)	$\mathcal{O}(n+m)$	Number of vertices n Number of edges m
Determining SCCs (3.3.2)	$\mathcal{O}(n+m)$	Number of vertices n Number of edges m
Dijkstra's Algorithm(3.4)	$\mathcal{O}(m+n\log n)$	Number of vertices n Number of edges m
Bellman-Ford (3.5)	$\mathcal{O}(n^2)$	Number of vertices n
Heapsort (3.6)	$\mathcal{O}(n\log n)$	Array length n
Floyd-Warshall (3.7)	$\mathcal{O}(n^3)$	Number of vertices n
Ford-Fulkerson (3.9.1)	$\mathcal{O}(mnC)$	Number of vertices n Number of edges m Highest capacity C
Hashing access time(3.10) - Chaining - Open addressing	 $\Theta(1+\alpha)$ $\mathcal{O}(\frac{1}{1+\alpha})$	Loading factor $\alpha = \frac{n}{m}$ Number of keys n Number of slots m
Binary search trees(4.1) - Insertion - Deletion - Searching	 $\mathcal{O}(\log n)$ $\mathcal{O}(\log n)$ $\mathcal{O}(\log n)$	 Number of nodes n
Red-Black search trees(4.3) - Insertion - Deletion - Searching	 $\mathcal{O}(\log n)$ $\mathcal{O}(\log n)$ $\mathcal{O}(\log n)$	 Number of nodes n

I wish you luck (and fun!) at the exam :)