Advanced Algorithms and Data Structures (AAD) Exam

tqs326 - Anders Munkvad

January 2024

Contents

1 Max flow (x2)

Plan

- Introduction
  - Properties
  - Max-flow min-cut theorem
  - Residual network
- Example of finding the max flow:

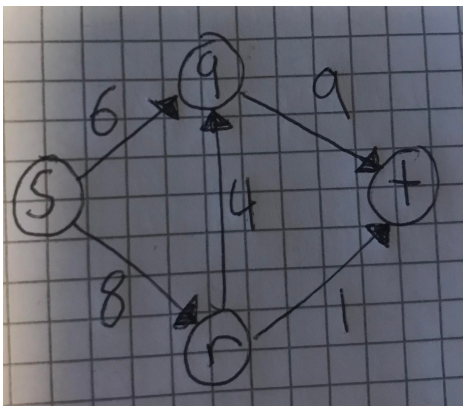

Figure 1: Example used to show how to find max flow

- Edmond Carp
  - Proof of running time

Notes

Introduction

- We have a graph $G = (V, E)$ with a $s \in V$ being the source and the sink $t \in V \setminus \{s\}$.
- We also have the capacity function that assigns the maximum amount of flow that can flow through that edge.
- **Flow network:**
  - A flow network (G, s, t, c) contains no self-loops or antiparallel edges.
  - A **flow** $f : V \times V \to R$ satisfies:
    - 1: $\forall u, v \in V : 0 \leq f(u, v) \leq c(u, v)$. (i.e. we satisfy the capacity constraints).
    - 2: $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$. (i.e. we satisfy the flow conservation.)
  - The value $|f| = \sum_{v \in V}(f(s, v) - f(v, s))$. (Just a way of expressing max flow, $|f|$, i dont understand the last part on why we minus - i guess its just if we have something going back)
  - Residual capacity c_f: $c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise} \end{cases}$.
    (Just a way of saying: We have a certain amount of flow going through and edge. The residual capacity will be whats left, i.e. how much more can we push through?)
- **Augmented Flow:**
  - Way to increase the flow in the original network G by considering an additional flow f' in the residual network G_r.
  - For an edge (u, v) in the original network E, the augmented flow is the sum of the current flow $f(u, v)$ and the additional flow $f'(u, v)$, minus the reverse flow on the residual edge $f'(v, u)$. If no edge (u, v) exist then 0. (giver mening, når du håndkører eksempel)

$$(f \uparrow f')(u, v) := \begin{cases} f(u, v) + f'(u, v) - f'(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

  - Proof of augmented flow (????)
- **Cut:**

- A cut is a partition of V into sets $S \ni s$ and $T \ni t$.

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - f(v,u)$$

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$$

(We divide the graph into two sub graphs. Each with the sink and source. The first is just the flow crossing this cut, the second will be the cpacity. I.e. the capacity will be the maximum amount that can cross this cut.)

- **Max-flow/min-cut theorem:** Let f be a flow in (G, s, t, c). Then the following 3 statements are equivalent (follows quite logically actually if you look at a given graph):
  - f is a max flow.
  - There is no augmenting path (in G_f) (i.e a path $s \to t$ in G_f).
  - There exists a cut (S,T) such that $|f| = c(S,T)$
- **Proof of max-flow/min-cut theorem:**
  - Let $S = \{v \in V \mid v\text{is reachable from } s\text{in } G_f\}$, $T = V \setminus S$. Then S, T partition $V, s \in S$ (why?) and $t \in T$ (why? no augmenting path), so (S,T) is a cut. Now let $u \in S, v \in T$. If $(u,v) \in E$ then $f(u,v) = c(u,v)$ (Why?), otherwise $c_f(u,v) > 0$ so $(u,v) \in E_f$. Since u is reachable from s in G_f that implies v is reachable from s in G_f thus $v \in S$ contradicting $v \in T = V \setminus S$. If $(v,u) \in E$ then $f(v,u) = 0$ (why?), otherwise $c_f(u,v) > 0$ and same problem. Thus:

$$|f| = f(S,T)$$

$$= \sum_{u \in S} \sum_{v \in T} (f(u,v) - f(v,u))$$

(the definition of $f(S,T)$)

$$= \sum_{u \in S} \sum_{v \in T} (c(u,v) - 0)$$

(argument above)

$$= c(S,T)$$

(Definition of $c(S,T)$ - hence it's proven).

- **Residual Network:** Introduce a residual edge that represents the additional flow capacity, given the actual current flow through the graph.

Example of finding the max flow:

Run Edmond Carp on the example.

Edmond Carp

```
EdmondsKarp(G)
1: f ← 0; G_f ← G
2: while G_f contains an s to t path P do
3:      Let P be an s to t path in Gf with the minimum number of edges.
4:      Augment f using P.
5:      Update G_f
6: end while
7: return f
```

- The running time of EdmondsKarp is $O(|V| \cdot |E|^2)$
- We want to show that at most $O(|E| \cdot |V|)$ augmentations (i.e. iterations where flow is added to the network) are used.
- In order to show this, we define a level graph:
  - This will be a subgraph of the residual graph. We construct it using BFS, which helps ensure that we get the shortest path from the source s to every other vertex.
  - Let $l(v)$ be the shortest path distance from s to v in G. (v will be some vertex in the level graph - since the level graph consists of the shortest distances from s to all other vertices, we use this notation to get that distance.)
  - The level graph of G is the subgraph with edges (u, v) such that $l(v) = l(u) + 1$.
  - By doing the above; For an edge (u, v) to be included in the level graph, the above condition must be met. The condition means that vertex v is exactly one level (as in the level graph) further from the source than vertex u, Since BFS has already ensured that $l(u)$ is the shortest distance from s to u, adding one more edge to reach v means $l(v)$ is also the shortest distance from s to v.

    (i.e. When you construct a path from s to t in the level graph using these edges, each step of the path moves exactly one level further from s. ince each of these steps is part of the shortest path from s o each intermediate vertex, the entire path from s to t is the shortest augmenting path.)
- The level graph is created from the residual graph (residual network).
- We have the directed graph G_f and the corresponding level graph.

- We observe that augmenting along a shortest path only creates G_f longer ones. When we augment flow along a shortest path in G_f, we may create longer paths in the residual graph, but we never create shorter paths than the current shortest path. This is because augmenting along the shortest path will only use up capacity or create reverse edges that do not contribute to a shorter path than the one we just used.

  (Since you use up capacity, you will potentially create a longer path than the one you are currently on. E.g. if you use up all of the capacity, hence resulting in that you have to take a longer path to get to t.)

- We then observe that the shortest path distance must increase every $|E|$ iterations. After every $|E|$ augmentations, the shortest path distance from the source to the sink in the level graph must increase. This is because each edge can be used in an augmenting path at most once before its residual capacity is reduced (since we always choose the shortest path). After $|E|$ augmentations, we must have used each edge at least once, and thus, to find a new augmenting path, we have to move to the next level, which corresponds to a longer path.

  (It intiuatively makes sense, once you pass through an edge, it will make that edge have less capacity, hence why once we have moved through all the edges, $|E|$ the shortest path must increase.)

- We also observe that there are only $|V|$ possible shortest path lengths. Since there are only $|V|$ vertices, there can be at most $|V|-1$ edges in any shortest path in a simple graph (without cycles). Therefore, the length of the shortest path can increase at most $|V|-1$ times. Since the shortest path increases after every $|E|$ augmentations, and the increase can only happen $|V|-1$ times, the total number of augmentations is $O(|E| \cdot |V|)$.

  (The length of a shortest path can only increase $|V|-1$ times. Makes sense, since if you have 4 vertices, you have 3 edges, hence once you pass through an edge, the shortest path will increase. **HENCE:** Since the shortest path can only increase every $|E|$ augmentation, i.e. we have used AT LEAST one of the edges - this is quaranteed to happen within $|E|$ augmentation (Since after running through every edge, one must have fulfilled its capacity, making it unavailable in the next shortest path), and this can only happen $|V|-1$ times, the total number of augmentations is $O(|E| \cdot |V|)$).

- **Total Running Time:** Each augmentation requires $O(|E|)$ time to find the shortest path using BFS. Give that we have at most $O(|E| \cdot |V|)$ augmentations, the total running time becomes $O(|E|) \cdot O(|E| \cdot |V|) = O(|V| \cdot |E|^2)$

2 Linear Programming and Optimization

Plan

- Objective function, constraints, forms of Linear Programming.
- Example (convert to standard form, then slack form and then Simplex).
- Simplex algorithm.
- Duality
- Example:

```
minimize -2x_1 + 3x_2
subject to x_1 + x_2 = 7
           x_1 - 2x_2 <= 4
           x_1 >= 0
```

Notes

Introduction

- **Objective function:**
- Describes the relationship between variables $(x_1, x_2, ...)$ of the problem, which we want to maximize or minimize.
- **Constraints:**
- The constrant are the values in which the variables are "constrained" within. The variables have to be withing these contraints
- **Forms of Linear Programming**
- Usually we write linear programs in standard form, or slack form, which is just different ways of representing a problem in linear programming.
- (**NOTES: Linear Objective function**:

```
min 3x_1 + 24x_2 + 13x_3 + 9x_4 + 20x_5 + 19x_6
s.t. linear constraints

Mere generel:

min/max \sum_{j=1}^{n} c_j x_j
s.t. m linear constraints
```

  - The value of the **objective function** for a particular set of values for $x_1, x_2, x_3...$ is called its **objective value**
  - If a particular set of values for $x_1, x_2, x_3, ...$ satisfies all constraints, it is said to be a **feasible solution**.
  - The set of all feasible solutions is called the **feasible region**. It can be shown to be convex (?).
  - A feasible solution that has the minimum (or maximum) objective value is called an **optimal solution.**
  - Strict inequalities are not allowed.
  - The geometric interpretation is relatively simple. All of the variables is withing a certain part of the plane. We can push these variables, and the one "furthest away" will be the optimal solution.

)

- **Geometric Interpretation in R^d: (Probably not relevant for the exam)**
  - We have d variables

- Each of our constraints define a half space in R^d. The set of feasible solutions is the intersection of these half-spaces, called **simplex**. It is convex. Can be unobunded or empty.
- The set of points in which the objective function has the same value z is a **hyperplane**.
- The value of the objective function increases or decreases as the hyperplane is translated (?)
- If the set of feasible solutions is bounded and not empty, then there is an optimal solution in an extreme vertex of the simplex.

Example

- **Standard Form:** When the problem is in standard form, all the constraints are equality constraints, and all the variables are non-negative.
- **Slack form:** In slack form, the linear programming problem is expressed with slack variables added to the inequalities to turn them into equalities.
- We have the original problem:

$$Minimize \ -2x_1 + 3x_2$$

$$s.t. \ x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$

- **Converting to standard form:**
  - In standard form it is a maximization of the linear function.
  - All variables are non-negative real-values.
  - We have m linear inequalities ("less than or equal to, $\leq$").
  - A minimization LP is converted to an equivalent maximization problem by negating the coefficients (times by -1) of the objective function, hence:

$$Max \ 2x_1 - 3x_2$$

$$s.t. \ x_1 + x_2 = 7$$

$$x_1 - 2x_2 \leq 4$$

$$x_1 \geq 0$$

  - Every variable x_j without the non-negativity constraint is replaced by two (non-negative) variables x'_j and x''_j and each occurence of x_j is replaced by $x'_j - x''_j$.

- We see that x_2 does not have the non-negativity constraint, hence we replace it:

$$Max\ 2x_1 - 3x_2' + 3x_2''$$
$$s.t.\ x_1 + x_2' - x_2'' = 7$$
$$x_1 - 2x_2' + 2x_2'' \leq 4$$
$$x_1, x_2', x_2'' \geq 0$$

- Each equality constraint is replaced by a pair of "opposite" inequality constraints (See how $= 7$ is converted):

$$Max\ 2x_1 - 3x_2' + 3x_2''$$
$$s.t.\ x_1 + x_2' - x_2'' \leq 7$$
$$x_1 + x_2' - x_2'' \geq 7$$
$$x_1 - 2x_2' + 2x_2'' \leq 4$$
$$x_1, x_2', x_2'' \geq 0$$

- However, we can only have $\leq$, so we multiply by -1 on both sides to turn the inequality around:

$$Max\ 2x_1 - 3x_2' + 3x_2''$$
$$s.t.\ x_1 + x_2' - x_2'' \leq 7$$
$$-x_1 - x_2' + x_2'' \leq -7$$
$$x_1 - 2x_2' + 2x_2'' \leq 4$$
$$x_1, x_2', x_2'' \geq 0$$

- You **can/don't have to** rename variables, but we instead of e.g. x_2'' we can say x_3:

$$Max\ 2x_1 - 3x_2 + 3x_3$$
$$s.t.\ x_1 + x_2 - x_3 \leq 7$$
$$-x_1 - x_2 + x_3 \leq -7$$
$$x_1 - 2x_2 + 2x_3 \leq 4$$
$$x_1, x_2, x_3 \geq 0$$

- It is now in standard form, and you can convert to slack form.

- **Converting to Slack Form:**
  - We convert it to slack form, as we need it when doing the Simplex algorithm.

- (**NOTES:** When considering constraints, we realise that for every feasible solution, the value of the left-hand side is at most the value of the right hand side (since we have $\leq$). We call the difference between the two values the **slack**.)
- We denote **slack** by introducing another variable $x_{\#\text{variables}+1}$
- By requiring that $x_{\#\text{variables}+1} \geq 0$ we can replace the inequality, $\leq$, with $=$.
- We take the standard form and introduce slack variables:

$$Max\ 2x_1 - 3x_2 + 3x_3$$

$$s.t.\ x_1 + x_2 - x_3 + x_4 = 7$$

$$-x_1 - x_2 + x_3 + x_5 = -7$$

$$x_1 - 2x_2 + 2x_3 + x_6 = 4$$

- We then convert it, so the slack variables are on the left hand side (fortegn skifter undtagen den første):

$$z = 0 + 2x_1 - 3x_2 + 3x_3$$

$$x_4 = 7 - x_1 - x_2 + x_3$$

$$x_5 = -7 + x_1 + x_2 - x_3$$

$$x_6 = 4 - x_1 + 2x_2 - 2x_2$$

- **Basic/non-basic:**
  - Any solution of LP in standard form yields a solution of LP in the corresponding slack form (with the same objective value) and vice versa.
  - Setting the right-hand side variables of the slack form to 0 yields a **basic solution**.
  - The left hand side variables are called **basic**. Right-hand side variables are called **nonbasic**. Above z, x_4, x_5, x_6 are basic, rest are nonbasic.
  - The basic variables are said to constitute a **basis** (?).
  - Note that a basic solution does not need to be feasible. E.g. if we set all right variables to 0 above, we would not get a feasible solution, as $x_5 = -7$ and we need a non negative result.

Simplex

- We need the problem in slack form - take the example from before.
- We set of all the **nonbasic variables** (right-hand side) to 0.
- Compute the values for the **basic values**: On the example above $x_4 = 7, x_5 = -7, x_6 = 4$.
- Compute the objective value $z = 0$.
- This gives the solution $(0, 0, 0, 7, -7, 4)$ - the values of $(x_1, x_2, x_3, x_4, x_5, x_6)$.
- **Not feasible** as we have a negative value slack variable.
- **Pivoting:** How much can we change/increase our variables, e.g. x_1, without violating feasibility (when will our slack variables become negative?)
  - If x_1 is increased beyond 7, then x_4 becomes negative.
  - If x_1 is increased less than 7, then x_5 becomes negative.
  - If x_1 is increased beyond 4, then x_6 becomes negative.
  - We see that the constraint defining x_6 is **binding** (it is the first to reach 0, when increasing x_1).
  - ----------
  - So x_1 can be increased to 4 without losing feasibility. The feasible solution is $(4, 0, 0, 3, -3, 0)$ and $z = 8$
  - We rewrite the slack form to an equivalent slack form with x_1, x_4, x_5 as the basic variables and with $(4, 0, 0, 3, -3, 0)$ being its feasible basic solution.
  - This rewriting is called **Pivoting**.
  - The binding constraint defining x_6 is rewritten so that it has x_1 on its left-hand side.
  - All other occurences of x_1 in other constraints and in the objective function are replaced by the right-hand side of the binding constraint.
  - so:

$$z = 0 + 2(4 - x_6 - 2x_2 + 2x_3) - 3x_2 + 3x_3$$
$$x_4 = 7 - (4 - x_6 - 2x_2 + 2x_3) - x_2 + x_3$$
$$x_5 = -7 + (4 - x_6 - 2x_2 + 2x_3) + x_2 - x_3$$
$$x_1 = 4 - x_6 - 2x_2 + 2x_3$$

  - After simplification:

$$z = 8 - 7x_2 + 7x_3 - 2x_6$$
$$x_4 = 3 + x_2 - x_3 + x_6$$

$$x_5 = -3 - x_2 + x_3 - x_6$$

$$x_1 = 4 - x_6 - 2x_2 + 2x_3$$

New basic variables $x_1 = 4, x_4 = 3, x_5 = -3$ and objective value $z = 8$. Feasible basic solution $(4, 0, 0, 3, -3, 0)$.

- We then pick the next one with non-negative coefficient, namely the one that would not decrease the objective value. Here we would pick x_3 as it has a positive impact on the objective value (do the same as before).
- If x_3 is increased beyond 3, then x_4 becomes negative.
- If x_3 is increased less than 3, then x_5 becomes negative (er det ikke less than 4, nej...)
- If x_3 is increased beyond 2, then x_1 becomes negative.
- Constraint x_1 is **binding**
- So x_3 can be increased by 2, making the feasible solution: $(4, 0, 2, 1, -1, 0)$ and objective value $z = 22$.
- We rewrite the slack form to an equivalent slack form with x_3, x_4, x_5 as the basic variables and with $(4, 0, 2, 1, -1, 0)$ being its feasible basoc solution.
- We rewrite the binding constraint x_1 to have x_3 on its left side.
- All other occurences of x_3 in other constraints and in the objective function are replaced by the right-side of the binding coinstraint.
- so: **PERFORM IT - make sure that it is feasible, i.e. a slack variable is not negative.**
- **unfinished**

Duality

- Also called dual of a linear program.
- We can use this to show that the SIMPLEX algorithm actually computes the optimal solution to the problem.
- The dual of a given linear programming problem is formed by a specific set of rules. These rules transform the objective function, constraints, and variables of the primal (the linear program) problem into a new set of constraints, variables and an objective function for the dual problem.
- if the primal is a minimization problem, then the dual is maximization problem, and vice versa.
- The constraints of the primal (like $\leq$, $\geq$, $=$) correspond to the variables of the dual (with each becoming $\leq$ becoming a $\geq$, and vice versa)
- **Weak and strong duality:**

- **Weak Duality:** The value of the objective function of the dual problem provides bounds for the value of the objective function of the primal solution. E.g. in a minimization problem primal problem, any feasible solution of the dual will give an upper bound to the primal's objective function.
- **Strong duality:** If both the primal and the dual have feasible solutions, then they both have optimal solutions and these optimal solutions are equal in value. This means that if you find the optimal solution to both the primal and the dual problems, and these values are equal, then you have indeed found the optimal solution.

Examples on how we can use LP to e.g. solve Max flow?

3 Randomized Algorithms

Plan

- Motivation as to why it sometimes makes sense to use Randomized Algorithms
- Monte Carlo and Las Vegas
- Example: RandQS (Runtime analysis)
- Example: Min-Cut algorithm

Notes

Motivation

- Faster. but weaker guarantess
- Simpler code, but harder to analyze.

Monte Carlo and Las Vegas

- **Las Vegas:**
  - Always returns correct answer.
  - Number of steps used is a random variable.
- **Monte Carlo:**
  - Some probability of error.
  - Number of steps used may be random or not.
- Converting into eachother (Just know the logic)

```
1: function RandQS(S = {s1, . . . ,sn})
Assumes all elements in S are distinct.
2:  if |S| <= 1 then
3:      return S
4:  else
5:      Pick pivot x in S, uniformly at random
6:      L ← {y in S | y < x}
7:      R ← {y in S | y > x}
8:      return RandQS(L)+[x]+RandQS(R)
```

RandQS - example

RandQS - runtme analysis:

- We want to prove the theorem:

$$E[\#comparisons] \in O(nlogn)$$

  (Since this is the running time: When we talk about running time for randomized algorithms, we look at the expected one).

- Let $[S_{(1)}, ..., S_{(n)}] := RandQS(S)$.

  (I.e. $S_{(i)}$ will be the ith element in the final sorted order.)

- For $i < j$ let X_{ij} be the number of times that $S_{(i)}$ and S_j are compared. We can then compute:

$$\#comparisons = \sum_{i<j} X_{ij}$$

(We just introduce a random variable that indicates the number of times S_i and S_j are compared. This will $X_{ij} \in \{0, 1\}$ because once we compare two elements, they will not get compared again, hence it will either be 1 or 0. They wont be compared to eachother, unless one of them is chosen as the pivot, as this is what we compare against)

(Shortnote: Note that $\sum_{i<j}$ is shorthand for $\sum_{1\leq i<j\leq n}$. We just use this notation as a way of saying "for all the elements in the final sorted array".)

- We can get the expectance of this as:

$$E[\#comparisons] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}]$$

(We use linearity of expectations above - $E[A + B] = E[A] + E[B]$)

- Since we have from the above, that $X_{ij} \in \{0, 1\}$ it is an indicator variable for the event that $S_{(i)}$ and $S_{(j)}$ are compared. We let p_{ij} be the probability of this event happening. (The below is just the definition of Expectation of a discrete random variable):

$$E[X_{ij}] = \sum_{x\in 0,1} Pr[X_{ij} = x] \cdot x$$

$$= (1 - p_{ij}) \cdot 0 + p_{ij} \cdot 1 = p_{ij}$$

The expectation of an indicator variable equals the probability of the indicated event :-) Therefore:

$$E[\#comparisons] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} p_{ij}$$

(Just show the below one, but understand why - shouldn't be rocketscience)

- **Lemma:** $S_{(i)}$ and $S_{(j)}$ are compared iff $S_{(i)}$ or $S_{(j)}$ is first of $S_{(i)}, ..., S_{(j)}$ to be chosen as pivot. (As mentioned before - there is a proof of this, it essentially just states that the two things are compared if they end up in a sublist together (i.e. left or right recursion)).

- p_{ij} is the conditional probability of picking $S_{(i)}$ or $S_{(j)}$ given that the pivot, denoted as c, is picked unformly at random in $\{S_{(i)}, S_{i+1}, ..., S_j\}$:

$$p_{ij} = Pr[c \in \{i, j\} \mid c \in \{i, i + 1, ..., j\} u.a.r.]$$

$$= \frac{2}{|\{i, i+1, ..., j\}|} = \frac{2}{j+1-i}$$

(Remember, $j+1-i$ just calculates the total number of elements in the set $|\{i, i+1, ..., j\}|$, which we obviously needs to determine the probability)

- It follows that:

$$E[\#comparisons] = \sum_{i<j} p_{ij} = \sum_{i<j} \frac{2}{j+1-i}$$

- The rest is just a bunch of sum arithmetic, fuck me.

- We then expand the $\sum_{i<j}$ notation. (Remember, it was just a fancy way of saying, all the elements in the sorted array, instead we expand it for both i and j):

$$E[\#comparisons] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j+1-i}$$

- The inner summation index is changed from j to k by setting $k = j+1-i$. Which simplifies the expression inside the sum:

$$E[\#comparisons] = \sum_{i=1}^{n-1} \sum_{k=2}^{n+1-i} \frac{2}{k} < \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k}$$

- The outer sum runs n times, and the inner sum is independent of i, so it's factored out:

$$E[\#comparisons] = 2n \sum_{k=2}^{n} \frac{1}{k} = 2n((\sum_{k=1}^{n} \frac{1}{k}) - 1) = 2n(H_n - 1)$$

- The harmonic series H_n is known to be approximated by the integral of $1/x$ from 1 to n, which gives $ln\ n$. This step uses an integral to approximate the sum, which is valid, since the harmonic series grows logarhitimically:

$$E[\#comparisons] \leq 2n \int_1^n \frac{1}{x} dx = 2n\ ln\ n \in O(n\ log\ n)$$

- Finally, since $2n\ ln\ n$ is propertional to $n\ log\ n$, the expected number of comparisons is $O(n\ log\ n)$.

(**NOTES:** Don't do all of the sum arithmetic in the exam, i cant be bothered. Just know why it actually sums up to be the harmonic series.)

RandQS - runtime (weird way of doing it)

- Theorem:
$$\mathbb{E}[\#comparisons] \in O(n\ log\ n)$$
- Slides 50-60
- We have that the worst case runtime of quicksort is $O(n^2)$ (Always picks pivot to be the first element of the array).
- In order to fix this runtime, we use randomized QuickSort.
- In Randomized QuickSort we pick the pivot to a random element in the array $A[x_1, x_2, ..., x_n]$ (Pick random pivot between x_1 and x_n).
- The element we pick, let's call it m, as pivot will be swapped with the first element in the array
- $A[1] \longleftrightarrow A[m]$
- We then pick the next pivot to be $pivot = A[1]$
- Next, just do this recursively untila array is sorted.
- **Proof:**
  - (i det næste når vi snakker om elementerne, snakker vi om elementerne i det array vi ender ud med efter sorteringen)
  - Pairs of element will have to be compared (altså hvis to elementer er ved siden af hindanden i det array vi ender ud med, kan man med garanti sige, at de to er blevet sammenlignet - ellers ville det være umuligt for dem at være ved siden af hindanden, da vi ikke ville vide at de var ved siden af hindanden, uden at sammenligne dem (hvilken rækkefølge de skulle stå i))
  - Elements that are further apart may be compared (Hvis den ene af dem bliver valgt som pivot på et eller andet tidspunkt i kørslen - However, hvis et element der befinder sig imellem de to elementer bliver valgt som pivot, er det muligt at de aldrig bliver sammenlignet - grundet at de vil ende i enten den venstre eller højre side af rekursionen. Altså vil der ikke være behov for at sammenligne).
  - The probability that two elements will be compared in an array of e.g. længth 6 is $\frac{2}{6}$.
  - We therefore have an expectation of if two elements are compared given the final sort.
  - The expectation of pairs getting compared is $\frac{2}{2}$ - and how many pairs do we have? We have $n-1$ pairs in a sorted array.
    - **NOTE:** Forstå nedenstående er essentielt fordi det er det du bruger, når du summer expectance?

    - $\frac{2}{2}$ expectance of comparisons for pairs.
    - $\frac{2}{3}$ expectance of comparisons for elements with 1 element between them - $n-1$ have this expectance.
    - $\frac{2}{4}$ expectance of comparisons for elements with 2 elements between them - $n-2$ have this expectance.
    - $\frac{2}{5}$ expectance of comparisons for elements with 3 elements between them - $n-3$ have this expectance.
    - $\frac{2}{n}$ - 1 have this expectance.
- We could like to sum up all of these expectations.
- We can factor out the 2.
- Harmonic series?
- Summing these up: $\frac{1}{2} \cdot n - \frac{1}{2} + \frac{1}{3} \cdot n - \frac{2}{3} + \frac{1}{4} \cdot n - \frac{3}{4}, ..., \frac{1}{n} \cdot n - \frac{n-1}{n}$.
- There is a harmonic series multiplied by n - we factor n out:

$$sum = n(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n})$$

- We are still missing the things we minus, hence we realise that these range from $\frac{1}{2}n$ to $1 \cdot n$, so it is a constant difference. Hence we just add a $-n$ to resemble this constant difference.

$$sum = n(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}) - n$$

- We know that the harmonic series sums up to $logn$ so - and remember the factor by 2:

$$2(n(logn) - n)$$

  We dont care about the constant difference, so fuck that, hence $O(nlogn)$.

Min-cut algorithm

Mangler

4 Hashing

Plan

- Why do we care about hashing?
- Definitions: Universal and strongly universal hashing.
- Example: Hash table with chaining.
- Hashing with chaining proof.

Notes

Motivation

- Don't confuse it with cryptographic hashing.
- The generel idea behind hashing is that we have universe U of keys that we want to map, however the universe is too large.
- Hence we wish to map it to a smaller range $m = \{0, ..., m-1$ of hash values.
- **Definition:** (hash function): *A hash function $h : U \to [m]$ is a random variable in the class of all functions $U \to [m]$, that is, it consists of a random varible h(x) for each $x \in U$*
- (**NOTE:** The definition above essentially just means that we have a given hash function that maps values in the universe to some integers ranging from 0 to $m-1$. Meaning, no matter the size of the universe, we can always map to these values. The random variable part, just means that the function is not a static function, but a part of a larger collection of functions. And that the output is not deterministic, but can vary, depending on the nature of the hash function.)
- We mainly care about 3 things when discussing this topic:
  - **Space:** Size of the random seed that is needed to calculate $h(x)$ given x.

    (**NOTE:** The random seed is what we use to generate the hash function $h(x)$. It is just something we use to generate random numbers - these can impact memory, so that's why we care)
  - **Speed:** Time to calculate $h(x)$ given x.

    (**NOTE:** Directly affects the performance of data retrieval or storage operations. A faster hash function enables quicker access to data in e.g. a hash table)
  - **The properties of the random variable:** Certain random variables have desirable properties such as universal hash functions, or strongly universal hash functions.

    (**NOTE:** Some random variable reduce % of collisions, see next section.)

Universal and strongly universal hashing

- **Definiton:** (Universal and c-universal hash function). *let $h : U \to [m]$ by a random hash function from a key universe U to a set of hash values $[m] =$*

$0, ..., m-1$. *Think of h as a random variable following some distribution over functions* $U \to [m]$. *Such a hash function is* ***universal*** *if for any given distinct keys* $x, y \in U$, *when h is picked at random independently of x and y, we have a low collision probability:*

$$Pr_h[h(x) = h(y)] \leq \frac{1}{m}$$

*Also, h is called c-****universal*** *if, for some c = O(1), we have:*

$$Pr_h[h(x) = h(y)] \leq \frac{c}{m}$$

(**NOTE:** Pretty self-explanatory. We want distinct keys to avoid collisions when doing lookups. You probably should not need to actually write all this during the exam, but good to actually understand)

- **Definition:** (Strong universality). *For* $h : [u] \to [m]$, *we consider pair-wise events of the form that given distinct keys* $x, y \in [u]$ *and possibly non-distinct hash values* $q, r \in [m]$ *we have* $h(x) = q$ *and* $h(y) = r$. *We say a random hash function* $h : [u] \to [m]$ *is* ***strongly universal*** *if the probability of every pair-wise event is* $\frac{1}{m^2}$. *We note that if h is strongly universal, it is also universal since:*

$$Pr[h(x) = h(y)] = \sum_{q \in [m]} Pr[h(x) = qh(y) = q] = \frac{m}{m^2} = \frac{1}{m}$$

(**NOTE:** $[u]$ is the set of all possible keys, $[m]$ is the range of all the hash values the function can produce. Here we focus on pair-wise events with two **distinct** keys $x, y \in [u]$ and two possibly non-distinct hash values $q, r \in [m]$. The thing we are essentially looking for is that the hash function maps x to q and y to r. What the definition is saying, is that the hash function is strongly universal iff the probability of each of these pair-wise events is exactly $\frac{1}{m^2}$ - **Essentially, the probability of causing a collision, namely picking the same hash value, is** $1/m^2$ **which is much lower** $1/m$ **than making strongly universal much better)**.

Hash table with chaining

- **Generel Idea:**
- Classic application of universal hashing is hash tables with chaining.
- We want to store a set $S \subseteq U$ of keys such that we can expect to find a key from S in constant time.
- We let $|S| = n$ and $m \geq n$ (de værdier vi kan hashe til er større eller lig antallet i vores sæt.)
- We pick a universal hash function $h : U \to [m]$.

- We then create an array L of m lists/chains so that for $i \in [m]$, $L[i]$ us the list of keys that hash to i.
- **Logic:**
- In order to decide if a key $x \in U$ is in S, we only have to check if x is in the list $L[h(x)]$.
- This takes time propertional to $1 + |L[h(x)]|$, since we spent 1 on look-up, and then potentially going through the entirety of $L[h(x)]$.
- In order to show this is universal, we assume that $x \notin S$ and that h is universal. We let $I(y)$ be an indicator variable which is 1 if $h(x) = h(y)$ (If there is a collision) and 0 otherwise. Then we have that the expected number of elements in $L[h(x)]$ is:

$$E[|L[h(x)]|] = E[\sum_{y \in S} I(y)] = \sum_{y \in S} E[I(y)] = \sum_{y \in S} Pr[h(x) = h(y)] = \frac{n}{m} \leq 1$$

- Each operation takes $O(|L[h(x)]| - 1)$ time. (**NOTE:** The probability is pretty self-explanatory. If we have n (which is $|S|$) the probability of causing a collision will be $\frac{n}{m}$, as m is the values that are hashed. Why we even say that this is less or equal to 1, idk. Just keep in mind the different rules of going through the first expectance to the probability in the exam)

Hashing with chaining proof:

- **Theorem:** For $x \notin S$, $E_h[|L[h(x)]|] \leq 1$.
  (The theorem essentially states, for any element x not in S, which is the list where we store our elements, the expected length of the list at the position where x would be inserted is less than or equal to 1. So, we expect the list, where we need to put x, to be less or equal to 1, since if it would be 1 or more, it would cause a collision. This is a statement about the efficiency of the hashing algorithm: ideally, you want the elements to be distributed uniformly across the hash table to minimize the length of these chains (lists), as longer chains mean longer search times.)
- **Proof:** We use the fact that by definition: $L[i] = \{y \in S \mid h(y) = i\}$

  $$E_h[|L[h(x)]|] = E_h[|\{y \in S \mid h(y) = h(x)\}|]$$

  We rewrite - $[h(y) = h(x)]$ becomes an indicator varible for the vent $h(y) = h(x)$:

  $$= E_h[\sum_{y \in S} [h(y) = h(x)]]$$

  Using linearity of expectations:

  $$= \sum_{y \in S} E_h[[h(y) = h(x)]]$$

Using expectation of an indicator varible:

$$= \sum_{y \in S} Pr_h[[h(y) = h(x)]]$$

Since we have that $x \notin S$ and $y \in S$, we have that $x \neq y$. Then by definition of a hash function: $h : U \to [m], Pr_h[h(y) = h(x)] \leq \frac{1}{m}$.

$$\leq |S|\frac{1}{m} \leq \frac{n}{m} \leq 1$$

($x \notin S$ and $y \in S$, i.e. y has already been hashed, and we want to hash x. Under the assumption that we have a good hashing scheme, the probability of causing a collision is $\frac{1}{m}$, i.e. causing a collision with y. The inequality $Pr_h[h(y) = h(x)] \leq |S| \cdot \frac{1}{m}$ sums this probability over all elements in S. The final step involves the assumption $|S| \leq n$, with n being all the elements that could be in the set, and $n \leq m$, i.e. the number of slots in the hash table is at least as large as the number of elements. This leads to the above conclusion, implying that the expected length of the list/slot where x needs to be inserted is less than or equal to 1.)

Multiply-shift is 2-universal (Not part of the exam)

- Touch on multiply-mod-prime (pretty similar to below, just slightly less complicated)
- **Definition** *Multiply-shift is a practical universal hashing scheme. It generally addresses hashing from w-bit integers to l-bit integers. Pick a uniformly random odd w-bit integer a, and then compute* $h_a : [2^w] \to [2^l]$ *as:*

  $$h_a(x) = \lfloor \frac{(ax \ mod \ 2^w)}{2^{w-l}} \rfloor$$

  (**NOTES:** We first pick an odd w-bit integer a - acts as the multiplier in the hash function. We pick an odd, as it helps ensure a good distribution of different hash values. In terms of $h_a(x)$: Our input value x is multiplied by our odd integer a. This product is then taken modulo 2^w to ensure the result stays withing w bits. If we did not take module, the product could potentially be larger than w bits. We then divide by 2^{w-l} which shifts the result $w - l$ bits to the right - Meaning, that after multiplication we have a w bit integer. In order to get l bit integer we shift it by dividing (We have the input size w, and the desired output size l). The result is an l bit integer, which is the hash value of the original w bit integer x.)
- This exploits that these operations are fast on computers.
- **Proof that Mupliply-shift is 2-universal - namely $\frac{2}{m}$:**

- **Logic used in proof:**
- Think of bits of a number as indexed with bit 0 at the least significant bit. The scheme is simply extracting bits $w-l, ..., w-1$ from the product ax. That is, it extracts the l most significant bits of $ax \ mod \ 2^w$, ensuring that h_a is indeed $[2^w] \rightarrow [2^l]$. (Essentially just saying that we map correctly, as in the notes above described.)
- We have $h_a(x) = h_a(y)$ (collision) if and only if ax and $ay = ax + a(y-x)$ agree on bits $w-l, ..., w-1$.

  (**NOTES:** IFF ax and ay has the same higher order bits (i.e. agree on bits), we will have a collision. This is because it is the higher order bits that determine the hash value after bit shifting.)
- In order for them to match it requires that bits $w-l, ..., w-1$ of $a(y-x)$ are either all 0s or 1s. More precisely, if we get no carry from bits $0, ..., w-l$ then we add $a(y-x)$ to ax, then $h_a(x) = h_a(y)$ exacylu when bits $w-l, ..., w-1$ of $a(y-x)$ are all 0s.
- Similarly, if we get a carry, then $h_a(x) = h_a(y)$ when the same bits are all 1s, since the carry is then carried all the way through the bits, leaving all 0s behind. It is thus sufficient to show that the probability of the bits being all 0 or 1s is at most $\frac{2}{2^l}$:
- **Proof:**
- We use the fact that any odd number z is relatively prime to any power of two:

  $$\text{if } \alpha \text{ is odd and } \ \beta \in [2^q]_+ \text{ then } \alpha\beta \neq 0 \ (\text{mod } 2^q)$$
- Define b such that $a = 1 + 2b$. Then b is uniformly distributed in 2^{w-1}, since a is chosen unformly at random. Moreover, define z to be odd number satisfying $(y-x) = z2^i$. Then

  $$a(y-x) = (1+2b)(y-x) = z2^i + bz2^{i+1}$$
- Next we show that $bz \ mod \ 2^{w-1}$ is uniformly distributed in $[2^{w-1}]$. There is a 1 to 1 correspondence between the $b \in [2^{w-1}]$ and the products $bz \ mod 2^{w-1}$; If not, then there would be a b' such that $b'z = bz(\ mod 2^{w-1}) \Leftrightarrow z(b-b') = 0(\ mod 2^{w-1})$.
- This is a contradiction to the fact that odd numbers are relatively prime to any power of two, since z is odd. But then the uniform distribution on b implies that $bz \ mod 2^{w-1}$ is uniformly distributed.
- We can now say that $a(y-x)$, as specified above, has 0 in bits $0, ..., i-1$, because of shiting with at least 2^i; it has 1 at index i, since z is odd, and thus we have that $z2^i$ has a 1 at bit index i; and finally, a uniform

distribution on bits $i+1, ..., i+w-1$, because bz is uniformly distributed and shifted to index 2^{i+1}.

- In conclusion, if $i \geq w - l$, then $h_a(x) \neq h_a(y)$, since ax and ay are always different in bit i; however, if $i < w - l$, then, because of cariies, we could have $h_a(x) = h_a(y)$ if bits $w - l, ..., w - 1$ of $a(y - x)$ are all either 0s or 1s. Because of the uniform distribution, either event happens with probability $\frac{1}{2^l}$, for a combined probability bounded by $\frac{2}{2^l}$, which is what we just proved.

- **NOTES:** Intitial thought process: We chose to prove that if all are 0s or 1s as this will result in collision. We then check the probability of this happens, due to different known facts - do however understand each point, especially with how bits carries.

5 van Emde Boas

Plan

- Motivation: Why are van Emde Boas Trees interesting
- What is it?
- Build example using $\{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\}$.
- Running time analysis.

Notes

What is it?

- **Problem:** Given a universe $U = [u]$ where $u = 2^w$, maintain subsets $S \subseteq U$, $S = n$ under:
  - **Member(x, S):** return $[x \in S]$.
  - **Insert(x,S):** Add x to S (assumes $x \notin S$).
  - **Delete(x, S):** Remove x from S (assumes $x \in S$).
  - **empty(S):** Return $[S = \emptyset]$.
  - **min(S):** Return min S (assumes $S \neq \emptyset$).
  - **max(S):** Return max S (assumes $S \neq \emptyset$).
  - **Predecessor(x, S):** Return $\max\{y \in S | y < x\}$ (assumes $\{y \in S | y < x\}$, i.e $S \neq \emptyset$ and $x > min(S)$

    (the above shows how we return the maximum y which is smaller than x, i.e. being the one right next to it)
  - **Succeessor(x, S):** $\max\{y \in S | y > x\}$ (assumes $\{y \in S | y > x\}$, i.e $S \neq \emptyset$ and $x < max(S)$

    (same as above, just the other way around).
- We split each key into high and low parts (e.g. when we represent in bits, i.e. in 1011_2, 10 will be the high part, and 11 will be the low part):
- Recall that $U = [2^w]$, and define:

$$hi_w(x) = \lfloor \frac{x}{2^{\lceil w/2 \rceil}} \rfloor$$

$$lo_w(x) = x \ mod \ 2^{\lceil w/2 \rceil}$$

$$index_w(h, l) = h \cdot 2^{\lceil w/2 \rceil} + l$$

- See how we define the summary and clusters later.
- A **van Emde Boas Tree** is a data structure that provides fast operations for a set of integers.
- It supports **search**, **successor**, **predecessor**, **insert** and **delete** in $O(lglg|U|)$ time, where U is the size of the universe.
- This is faster than e.g. priority queue, binary search tree etc.
- We also have that van emde boas trees works with $O(1)$ time complexity for minimum and maximum query.

- **NOTE:** Van Emde Boas data strcuture's keys set must be defined over a range of 0 to n (n being a positive integer of the form 2^k) and it works when euplicate keys are not allowed.
- **Definition:** (van Emde Boas tree). *The **van Emde Boas tree**, denoted **vEB tree**, is a recursive data structure on a universe of keys, whose size u is any exact power of 2. Each vEB(u) contains the universe size u, elements min and max, a pointer **summary** to a $vEB(\uparrow\sqrt{u})$ tree, and an array cluster $[0, ..., \uparrow\sqrt{u}]$ of $\uparrow\sqrt{u}$ to $vEB(\downarrow\sqrt{u})$.*

  (**NOTES:** vEB is just an abbreviation of van emde boas trees. $vEB(\downarrow\sqrt{u})$ is just an abbreviation of a vEB containing u keys. Each cluster corresponds to a subset of the universe and has a universe size of $\sqrt{u}$. This recursive division continues until the size of the universe is 2. This is done to keep the depth of the tree small, which we can use in the running time analysis. The up and down arrows are used to handle the case where $\sqrt{u}$ is not a perfect square. I.e. if the $\sqrt{u}$ is not a power of 2, you round up to the nearest power of 2. This is used for the size of the **summary** vEB tree and also determines the number of clusters. It ensures that each cluster and the summary tree are dealing with universe sizes that are powers of 2, which is a requirement for the vEB tree. If $\sqrt{u}$ is not a power of 2, you round down to the nearest power of 2. This is used to determine the universe size for each of the vEB trees in the cluster array.)
- We redefine **summary** and **clusters** to recursively store the sets:

$$summary := \{hi_w(x) \mid w \in S \setminus \{min, max\}\}$$

$$clusters[h] := \{l \in [2^{\lceil w/2 \rceil}] \mid index_w(h, l) \in S \setminus \{min, max\}\} \; \forall h \in [2^{\lceil w/2 \rceil}]$$

- **Summary:** A smaller vEB tree that keeps track of which cluster have at least one element. It's defined to store the **high bits** of all elements excluding the min and max.
- **Clusters:** The clusters are an array of vEB trees, each responsible for a subrange of the universe. They store the **low bits** of the elements, again excluding the min and max.

Example:

- We have a van Emde Boas tree that already stores $\{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\}$.
- **Important:** Each cluster defines a range of characters. The cluster contains the min and max from withing that range.
  - The size of the universe is determined as follows: The largest number above is 13. In order for our universe to accomodate this value, we need atleast 4 bits. Therefore, the next power of 2 that can represent this range is $2^4 = 16$ - therefore $U = 16$.

- We then split the universe into $\sqrt{U}$ cluster.
- $\sqrt{U} = \sqrt{16} = 4$, so we have 4 clusters.
- The clusters are organized as the following:
  - The first cluster covers the range from 0 to $\sqrt{U} - 1$, which will be 0 to 3 in the above example.
  - The second cluster covers the range from $\sqrt{U}$ to $2\sqrt{U} - 1$, which will be 4 to 7.
  - The third cluster covers the range from $2\sqrt{U} - 1$ to $3\sqrt{U} - 1$, which will be 8 to 11.
  - The fourth cluster covers the range from $3\sqrt{U}$ to $4\sqrt{U} - 1$, which is 12 to 15.
- Indsæt billede
- We start by setting $1 = 0001_2$ as the "global" minimum.
- We then look at the next number, which is $4 = 0100_2$. We put its higher order bits in the summary (as it is the "next" smallest element). We update cluster's min value. (Cluster 2 since it ranges from 4-7).
- The next element, $5 = 0101_2$, should also be placed in cluster 2 for obvious reasons. However, since it is not the minimum or maximum of this cluster, its 0 (highest order bit) is placed in summary, while 1 (lower order bit) gets placed in cluster.
- We then see that the next elemement is $7 = 0111_2$. This is also placed in cluster 2. We see that this is max, hence max of cluster 2 will be updated.
- We see that the next element is $10 = 1010_2$, hence it needs to be placed in cluster 3. Since this is the only element in cluster its lower order bits will be both max and min.
- Remember that the lower order bits are always placed in the clusters, while the higher order bits are placed in summary.
- We can simply find $13 = 1101_2$ by looking at the first max.
- Hence, cluster 1 and 3 will be empty, while the others will consists of the elements form the above.

Running time analysis

- **Theorem:** The recursion depth of this structure, when used on the universe $U = [2^w]$ is $\lceil log_2 w \rceil = O(log\ log\ |U|)$. (I.e. this is the running time, since the recursion depth is what decides the running time.)
- **Proof:** Let $d(w)$ be the recursion depth when working with a universe of size 2^w. We will prove by induction that $d(w) = \lceil log_2 w \rceil$.

Base case: Base case is $w = 1$ where there is no recursion, $d(1) = 0 = \lceil log_2(1) \rceil$. (i.e. both equals 0, so base case holds)

Induction step: We suppose $w > 1$ and that $d(w') = \lceil log_2(w') \rceil$ for all $w' \in [w]_+$. Now let $w' = \lceil w/2 \rceil \in [w]_+$, then the largest universe size used in the recursion is $2^{w'}$, and (by induction) $d(w) = 1 + d(w') = 1 + \lceil log_2(w') \rceil = \lceil log_2(2w') \rceil$. What remains is to show $\lceil log_2(2w') \rceil = \lceil log_2(w) \rceil$.

If w is even, $2w' = w$ and we are done. Otherwise $w > 1$ is odd so $w \geq 3$ and the smallest integer $k = \lceil log_2(w) \rceil$ such that $2^k \geq w$ satisfies $k \geq 1$. Thus, 2^k is even and so must satisfy $2^k \geq w+1 = 2w'$ and therefore also $k = \lceil log_2(2w') \rceil = \lceil log_2(w) \rceil$.

- Notes: In simpler terms, it's saying that the number of times you need to recursively call the structure's operations (like search, insert, or delete) is proportional to the logarithm of the logarithm of the size of the universe. The universe here is the range of values the data structure can handle, and w represents the number of bits needed to represent the largest number in this universe. w' is a smaller version of the universe, half to be exact, becase $w' = \lceil w/2 \rceil$. If our universe is 2^w, then the smaller universe is $2^{w'}$.

6 NP-completeness (x2)

Plan

- Outline: What are hard problems?
- Definition of NP and reduction.
- Definition of NP-completeness.
- Show NP-completeness for vertex cover.
- Show NP-completeness for TSP.

Notes

Outline

- **Definition:** (Problem). *Consider a set I of instances and a set S of solutions. An abstract **problem** is a binary relation between I and S, i.e., a subset of I x S*

  (**NOTES:** Example: SHORTEST-PATH-PROBLEM: An instance would be a triple $<G, s, t>$ where G is the graph, s is the source node and t is the sink node. A solution would be a sequence of vertices forming a shortest s-to-t path.)

- **Definition:**(Decision Problems). "Yes" or "no" problems, hence $S = \{0, 1\}$, i.e. "yes" $= 1$; no $= 0$. E.g. $PATH<G, s, t, k> = 1$ if there is a path from s-to-t with at most k edges. A decision problem can be seen as a mapping from instances to $S = \{0, 1\}$. Instances with solution 1 are called **yes**-instances. Pretty obvious what solution 0 are called.

Optimization problems

- Most optimization problems can be turned into decision problems.
- E.g. if you want to find an optimal path of min length, then iterate over the decision problem starting with the smallest k possible, do $k + 1$ until the answer to the decision problem is 1.
- **Definition:** (Polynomial-time solvable problems). *We assume that instances of a problem are encoded as binary strings. A given algorithm **solves** a problem in time $O(T(n))$ if for any instance of length n, the algorithm return a solution (i.e 1 or 0) in time $O(T(n)$. If $T(n) = O(n^k)$ for some constant k, the problem is **polynomial-time solvable.***

  (Giver vel rimelig meget sig selv, ved ikke hvorfor man behøver en definition på det)

- $\langle x \rangle$ is used to refer to a chosen encoding of an instance x of a problem (Here this will always be a binary string).
- **Definition:** (Languages). *An **alphabet** is a finite set Σ and symbols. A **language** L over an alphabet Σ is a set of strings of symbols from Σ. E.g.: $\Sigma = \{a, b, c\}$ and $L = \{a, ba, cab, bbac, ...\}$. We also allow the empty string and denote it ϵ. The empty language is denoted $\emptyset$ and it does not contain ϵ. Σ^* denotes the language of all strings (including ϵ). Any language L over Σ is a subset of Σ^**

Instances of decision problems/Accepting algorithm/deciding algorithms:

- As mentioned, decision problems are encoded as binary strings, and a decision problem can be seen as a mapping $Q(x)$ from instances to $\Sigma = \{0, 1\}$. We can specify Q to be the binary strings that encode yes-instances of the problem. Hence, we can view Q as a language L:

  $$L = \{x \in \Sigma^* | Q(x) = 1\}$$

  (**NOTES:** Essentially just saying that L consists of all the yes-instances of the given problem mapping Q.)

- **Definition:** (Accepting algorithm, deciding algorithm). *Let A be an algorithm for a decision problem and denote it by $A(x) \in \{0, 1\}$ its output given input x. Then A* ***accepts*** *a string x if $A(x)$=1 and* ***rejects*** *a string if $A(x)$= 0. The language accepted by A is*:

  $$L = \{x \in \{0, 1\}^* | A(x) = 1\}$$

  (**NOTES:** Same as above?? Except is not all possible strings, i guess)

- If we suppose in addition that all strings not in L are rejected by A, i.e $A(x) = 0 \forall x \in \{0, 1\}^* \setminus L$. Then we say that L is **decided** by A. Deciding a language is stronger than accepting it.

  (**NOTES:** It is just saying that a language L is decided by algorithm A if A accepts all strings in L (hence the yes-instances) and rejects all string not in L (the no-instances) - so both acceptance and rejection. Acception does not take into account rejecting all not in L.)

- We say that a language L is accepted by an algorithm A in polynomial time if A accepts L and runs in polynomial time on strings from L. Similarly, L is decided by A in polynomial time A decides L and runs in polynomial time on all strings.

  (**NOTES:** Relates to complexity classes. E.g. if both yes and no-instances can be decided in polynomial time, they belong in **P**. If we can verify the yes-instances on polynomial time, it belongs in **NP**. P = decided, NP = accepted)

Complexity classes:

- A **complexity class** can be seen as a set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithm that determines whether a given string x belongs to language L.

- **Definition:** (Complexity class P).

  $P = \{L \subseteq \{0,1\}^* | \textit{there exists an algorithm } A \textit{ that decides } L \textit{ in polynomial time}\}\}$

- **Lemma:** (P in terms of acceptance).

  $P = \{L \subseteq \{0,1\}^* | \textit{there exists an algorithm } A \textit{ that decides } L \textit{ in polynomial time}\}\}$

  $P = \{L \subseteq \{0,1\}^* | \textit{there exists an algorithm } A \textit{ that accepts } L \textit{ in polynomial time}\}\}$

- We may not have an efficient algorithm that accepts a language L. If we e.g. consider an algorithm A taking two parameters, $x, c \in \Sigma^*$. Instead of trying to find a given solution to x, which can take a long time, A instead **verifies** that c is a solution to x. (Making it a decision problem, because the answer is either yes or no, right?)

- **Example (Not needed for the exam, but good for understanding:)** (The HAM-CYCLE problem). *An undirected graph G is hamiltonian if it containts a simple cycle containing every verted of G. We define:*

  $$HAM - CYCLE = \{\langle G \rangle \mid G \textit{ is } Hamiltonian\}$$

  It is open whether HAM-CYCLE can be decided in polynomial time (hence, they are clinically insane if they ask this at the exam). However, it is easy to show that HAM-CYCLE can be verified in polynomial time.

  If we consider an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$. A_{ham} checks that $\langle G \rangle$ and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once. If so, $A_{ham} = 1$, otherwise $A_{ham} = 0$. Designing such an algorithm is simple (Simply check if C is a hamiltonian cycle. This will scale with the size of G, hence being polynomial and not exponential).

- **Definition** (Verification of languages). *A **verification algorithm** A is an algorithm taking two arguments $x, y \in \{0,1\}^*$, where y is the **certificate** and x is a string. Algorithm A **verifies** x if there is a certificate y such that $A(x, y) = 1$. The language verified by A is:*

  $$L = \{x \in \{0,1\}^* \mid \textit{There is a } y \in \{0,1\}^* \textit{ such that } A(x, y) = 1\}$$

  If we e.g. take the previous example:

  $$HAM{-}CYCLE = \{x \in \{0,1\}^* \mid \textit{There is a } y \in \{0,1\}^* \textit{ such that } A_{ham}(x, y) = 1\}$$

- **Definition:** (Complexity class NP). *NP is the class of languages that can be verified in polynomial time. More precisely, $L \in NP$ íf and only if there is a polynomial-time verification algorithm A and a constant c such that*

  $$L = \{x \in \{0,1\}^* \mid \textit{There is a } y \in \{0,1\}^* \textit{ with } |y| = O(|x|^c) \textit{ such that } A(x, y) = 1\}$$

We have that $P \subseteq NP$, but do not know if $P = NP$.

(Bare at vi kan verify i poly tid, ligesom vi kunne i ham-cycle, altså er den i NP.)

- **Definition:** (Complexity class co-NP). *co-NP is the class of languages L such that* $\bar{L} \in NP$

  (**NOTES:** $\bar{L}$ refers to the complement of a language L, meaning all strings that are not in L. Recaæææ that a language is in NP if we can verify yes-instances in polynomial time. In contrast, a language is in co-NP if we can verify no-instances in polynomial time)

- It is not known if $NP = co - NP$. I.e. for the HAM-CYCLE problem, given a graph, we can we easily verify that it does *not* have a simple cycle containing every vertex of G (Being the complement of the previous example)? What we do know is $P \subseteq NP \cap co_N P$.

  (**NOTES:** Recall P is both yes and no instances. Hence the intersect of NP and its complements, i.e. co-NP (both the yes and no instances), will essentially be equal to P (or $\subseteq$))

- We have problems in NP that are the "most difficult" in that class. If any of those problems can be solved in polynomial time, then *every* problem in NP can be solved in polynomial time. These problems are called NP-complete.

Reducibility and NP-complete

- **Definition:** (Reducibility). *A language* L_1 *is polynomial-time* ***reducible*** *to language* L_2 *if there is a polynomial-time computible function* $f : \{0,1\}^* \to \{0,1\}^*$ *such that for all* $x \in \{0,1\}^*$

  $$x \in L_1 \leftrightarrow f(x) \in L_2$$

  In this case, we write $L_1 \leq_p L_2$.

  (**NOTES:** The above means, that x is a yes-instance of L_1 if and only if $f(x)$ is a yes-instance of L_2. The function can do this reduction in polynomial time, hence not causing much more runtime. The goal is to show, that one problem is at least as hard as another problem (i.e. solving L_2 would also imply a solution for L_1). This is used to prove that problems are NP-hard or NP-complete)

- If $L_1 \leq_p L_2$ then L_1 is in a sense no harder to solve than L_2. More precisely:

  $$L_1 \leq_p L_2 \wedge L_2 \in P \to L_1 \in P$$

This follows since any instance I_1 of L_1 can be solved by transforming (reducing?) it in polynomial time to an instance I_2 of L_2 and then solving I_2 with a polynomial time algorithm for L_2.

- **Definition** (NP-complete languages). *A language L is NP-complete if*

$$1.\ L \in NP \text{ and}$$

$$2.\ L' \leq_p L \text{ for every } L' \in NP$$

L is NP-hard if property 2 holds (and possibly not property 1). The class of NP-complete languages id denoted NPC. If some language of NPC belongs to P then $P = NP$. This is because any language in NP can then be transformed to this language in polynomial time, and then solved on polynomial time (making all NP languages solvable in polynomial time).

(**NOTES:** The first property is that the language must be in NP, hence there is a polynomial time algorithm to verify yes-instances of L (We can check if an answer is correct in polynomial time). The second property says, that for every langauge L' in NP, L' is polynomial time reducible to L. Meaning any problem in NP can be transformed to a problem in L in polynomial time, essentially implying that L is at least as hard as any problem in NP. So the point is, it is NP-complete because if it's solvable, we can solve every problem in NP. Because if you can solve L, then you can also solve L', hence all problems in NP)

- **Technique for showing NP-completeness for a language L:** Suppose L' is an NP-complete language. If $L' \leq_p L$ then L is NP-hard, because all other languages in NP can be reduced to L', which can in turn be reduced to L, i.e. all languages in NP can in this be reduced to L. If it is also the case that $L \in NP$, then L is NP-complete (by the defintion of NP-complete languages).

  The method for $L \in NPC$ (L is NP-compelete) is thus as follows:

  - Prove $L \in NP$.
  - Select a known NP-complete language L'
  - Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
  - Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
  - Prove that the algorithm computing f runs in polynomial time.

- Problems in NP (-complete?): CIRCUIT-SAT, SAT, 3-CNF-SAT, SUBSET-SUM, CLIQUE, VERTEX-COVER, HAM-CYCLE, TRAVELLING-SALESMAN-PROBLEM.

Showing that VERTEX-COVER is NP-complete:

Before showing that **VERTEX-COVER** is NP-complete, we first define the problem. A **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then either $u \in V'$ or $v \in V'$ or both (i.e. if u and v share an edge, then one of them or both must be in the subset V'). That is, each $v \in V'$ covers all its incident edges, and a vertex cover for G covers all the edges E. (**GOAL:** Cover **all** edges using the least (or max i guess, depending on the problem) number of vertices possible). The size of a vertex cover is the number of vertices in it. The **vertex-cover problem** is to find a vertex cover of minimum size in a given graph (i.e. how do we cover all edges, using the minimum amount of vertices). This is an optimization problem. The corresponding decision problem is to determine whether a graph has a vertex cover of size k. We define VERTEX-COVER as the following language:

$$\{\langle G, k\rangle \ : \ \text{graph } G \text{ has a vertex cover of size } k\}$$

(I.e. the decision problem is ask whether or not this solution exists, hence being a vertex cover that covers all edges, with k vertices.)

- **Prove $L \in NP$:** We first prove that VERTEX-COVER is in NP, meaning that we can verify a potential solution in polynomial time. A certificate for vertex cover is the set of vertices $V' \subseteq V$. We can easily check whether V' is a vertex cover of size k in polynomial time, by first checking that $|V'| = k$ and then going through all edges in E and checking that each edge is incident to at least one vertex in V'. (The key point here being, that checking both of these things should not result in exponential time, rather we can do both in polynomial time, hence it being in NP)

- **Select a known NP-complete language L':** We select CLIQUE.

- **Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L:** The reduction from CLIQUE to VERTEX-COVER introduces the notion of a complement of a graph. Given an undirected grapg $G = (V, E)$, the **complement** of this graph G will be $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(u, v) : u, v \in V, u \neq v, and (u, v) \notin E\}$. (I.e we now have a graph that contains all of the edges exactly not in G).

  The reducation takes an instance $\langle G, k\rangle$ of CLIQUE as input. It then computes $\bar{G}$ (i.e. the complement) in polynomial time and returns the instance $\langle \bar{G}, |V| - k\rangle$ of the vertex cover problem.

  (The smart thing here is, that a CLIQUE of size k G corresponds to a vertex cover of size $|V| - 1$ in $\bar{G}$. Since in the complement graph, the edges present are those not in the original graph. So a clique in G (where every vertex is connected to every other vertex in the clique), translates

to a set of vertices in $\bar{G}$ that are not connected to eachother, hence they cover all edges of $\bar{G}$)

- **Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.:** We have to prove that: The graph G has a clique of size k if and only if the graph $\bar{G}$ has a vertex cover of size $|V| - k$. To show that the first implies the second, suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. Then $V - V'$ is a vertex cover in $\bar{G}$.

  Let (u, v) be any edge in $\bar{E}$. The $(u, v) \notin E$, which implies that at least of u or v does not belong in V', since every pair of vertices in V' is connected by an edge of E. This means that at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\bar{E}$, every edge of $\bar{E}$ is covered by a vertex in $V - V'$. Hence, the set $V - V'$ which has size $|V| - k$, forms a vertex cover of $\bar{G}$.

  Conversely, suppose that $\bar{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then for all $u, v \in V$, if $(u, v) \in \bar{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words $V - V'$ is a clique, and it has size $|V| - |V'| = k$.

- **Prove that the algorithm computing f runs in polynomial time:** We can easily compute the complement of G in polynomial time by going through $|V|^2$ iterations, choosing any two distinct vertices u and v, and adding an edge $\{u, v\}$ to $\bar{E}$ if $\{u, v\} \notin E$.

Showing that TSP is NP-complete:

Problem definition: A hamiltonian cycle is a path in an indirected graph $G = (V, E)$ where each vertex is visited exactly once. In the **travelling salesman problem**, a salesman must visits n cities. Modelling the problem as a complete graph with n vertices, we can say that the salesman wants to make a **tour**, i.e. a hamiltonian cycle, visiting each city exactly once, and finishing in the city he started in.

The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to j, and the salesman wants to make the tour whose total total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. The formal language for the corresponding decision problem is:

$$\{\langle G, c, k\rangle : G = (V, E) \text{ is a complete graph,} \tag{1}$$
$$c \text{ is a function from } V \times V \to N, \tag{2}$$
$$k \in N, \textit{and}, \tag{3}$$
$$G \text{ has a travelling-salesman tour with cost at most } k \tag{4}$$

We now want to prove that TSP is NP-complete by reducing a known NP-complete problem, namely the HAM-CYCLE problem, to TSP in polynomial time.

- **Prove $L \in NP$:** A certificate is a seuence of n vertices in a tour. The verification checks that each vertex is contained once in the sequence, sums the edge cost and checks that the sum is at most k. This is easily done in polynomial time. (I.e. we check if a given solution is a solution. None of the steps in doing so, should introduce exponential time complexity).

- **Select a known language NP-complete language L':** As mentioned we select the HAM-CYCLE problem, which we know to be NP-complete.

- **Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L:**

  Let $G = (V, E)$ be an instance of a HAM-CYCLE. We construct an instance of TSP as follows: We form the complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$ and we define the cost function c by:

  $$c(i,j) = \begin{cases} 0 & \text{if } (i,j) \in E, \\ 1 & \text{if } (i,j) \notin E. \end{cases}$$

  The instance of TSP is then $\langle G', c, 0 \rangle$.

- **Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.:**

  Suppose that graph G has a hamiltonian cycle h. Each edge in h belongs to E and thus has a cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the cost of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We can therefore conclude that h' is a hamiltonian cycle in graph G.

- **Prove that the algorithm computing f runs in polynomial time:** Clearly we can do the above in polynomial time.

7 Exact exponential algorithms and parameterized complexity

Plan

- Motivation: Why do even care about exponential algorithms?
- Example using the Travelling-Salesman problem (Exact Exponential algorithms).
- Example using k-vertex cover (parameterized complexity)

Notes

Definitions and Logic

- **Definition:** (O^* notation). *For functions f and g we write $f(n) = O^*(g(n))$ if $f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial.*
- For example, for $f(n) = 2^n n^2$ and $g(n) = 2^n$, $f(n) = O^*(g(n))$.
  (**NOTES:** its a way of focusing on the exponential growth, rather than the polynomial one. Since polynomial is essentially "Irrelevant" compared to exponential. Similar to how we ignore constants in regular O notation, here we ignore polynomial time, because it is irrelevant.)
- **Measuring quality of exact algorithms:** The running time of an algorithm is estimated by a function either of the input length or of the input "size". The input length can be defined as the number of bits in any reasonable encoding of the input over a finite alphabet; the notion of input size if problem dependent.

Exact Exponential Algorithms (EEA)

- Algorithms that run in exponential time, and there is no know solution to the given problems that run in polynomial time.
- A lot of these algorithms are designed to solve NP-complete problems. Recall that these are the problems for which no known polynomial-time algorithm exists. (Note: we can still verify them in polynomial time, see section on NP-completeness).
- Exact Exponential Algorithms do however guarantee optimal solutions. Unlike, e.g. approximation algorithms (see section on approximation.)
- Can be good to use on a small or moderate input size (this can be managed using **parameterized complexity**)

Parameterized Complexity (PC)

- **Parameterized Complexity** measures complexity not only in terms of input length but also in terms of a parameter which is a numerical value not necessarily dependent on the input length.
- Many parameterized algorithmic techniques evolved accompanied by a powerful complexity theory (okay, who cares?).
- We want to have the possibility of getting algorithms whose running time can be bounded by a polynomial function of the input length and, usually, an exponential function of the parameter.
- Most of the exact exponential algorithms studied in this course can be treated as parameterized algorithms, where the parameter can be the number of vertices in a graph, etc.

Similarities between EEA and PC:

- Many basic tehniques, such as branching dynamic programming, iterative compression and inclusion-exclusion are used in both areas.
- There is a nice connection between sub-exponential complexity and parameterized complexity.
- **Definition:** (Fixed-Parameter algorithms, or FPT). *Algorithms with running time $f(k) \cdot n^c$, for a constant c independent of both n and k, are called* ***fixed-parameter algorithms****, or* ***FPT algorithms****. Typically the goal in parameterized algorithmics is to design FTP algorithms, trying to make the f(k) factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be put in contrast with less efficient XP algorithms, where the running time os of the form $f(k) \cdot n^{g(k)}$, for some function f and g. There is a tremendous difference in the running time.*

  (**NOTES:** The key of this is that, the exponential part is confined in the $f(k)$ part, which allows us to make somewhat efficient algorithms, when k is small, even if n is large.)
- In parameterized algorithmics, k is simply a **relevant secondary measurement** that encapsulates some aspect of the input instance, be it the size of the solution sought after, or a number describing how "structured" the input instance is.
- **Definition.** (A parameterized problem). A **parameterized problem** is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the **parameter**.

  (**NOTES:** We use the parameter k to not only look at the input size, but also the algorithm more generally. E.g. In a graph problem, the total input size n might refer to the number of vertices or edges in the graph. This is a general measure of the problems size. The parameter k, such as the depth of a tree within this graph, is a more specific measure. It focuses on a prticular aspect of the problem that might a significant impact on the computational complexity. So instead of in more general O notation, where essentially all that matters is n (input size), we look at other aspects that could have a significant impact on the running time (e.g. depth of tree) (which does not depend on input size))

  (**NOTES 2:** The goal is to design algorithms where the most computationally intensive part scales with k rather than n. Such algorithms are termed Fixed-Parameter Tractable (FPT) when they have running times like $f(k) \cdot n^c$.)

Dynamic Programming for TSP - mangler forklaring:

- **Definiton:** (Travelling Salesman Problem). *Given a set of distinct cities* $\{c_1, c_2, ..., c_n\}$ *and for each pair* $c_i \neq c_j$*, the distance between* c_i *and* c_j *is denoted by* $d(c_i, c_j)$*. The task is to construct a tour of the travelling salesman of minimum total length which visits all the cities and returns to the starting point. In other word, the task is to find a permutation* π *of* $\{1, 2, ..., n\}$*, such that the following sum is minimized:*

$$(\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)})) + d(c_{\pi(n),c_{\pi(1)}})$$

  (Forklar formlen?)

- The naive brute force approach is to enumerate all possible permutations of $1, 2, ..., n$, of which there are $n!$. Using dynamic programming we can obtain a much faster algorithm.

- The idea is as follows:

  - For every pair (S, c_i), where S is a nonempty subset of $\{c_2, c_3, ..., c_n\}$ and $c_i \in S$, the algorithm computes the value $OPT[S, c_i]$, which is the minimum length of a tour which starts in c_1, visits all cities from S and ends in c_i

- We compute the values of $OPT[S, c_i]$ dynamically and bottom-up, in order of increasing cardinality of S. The computation of $OPT[S, c_i]$ in the case S contains only city is trivial, because in this case, $OPT[S, c_i] = d(c_1, c_i)$. For the case $|S| > 1$, the value $OPT[S, c_i]$ can be expressed in terms of subsets of S:

$$OPT[S, c_i] = \min\{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i) : c_j \in S \setminus \{c_i\}\}$$

- Indeed, if in some optimal tour in S termincating in c_i, the city c_j immediately preceeds c_i, then

$$OPT[S, c_i] = OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i)$$

- Thus, taking the minimum over all cities that can precede c_i, we about the minimization function given above. Finally, the value OPT of the optimal solution is the minimum of:

$$OPT[\{c_2, c_3, ..., c_n\}, c_i] + d(c_i, c_1)$$

  Where the minimum is taken over all indices $i \in \{2, 3, ..., n\}$.

- The amount of steps required to compute the minimization function, for a fixed set S of size k and all vertices $c_i \in S$ is $O(k^2)$, since for a specific c_i, we iterate over all $c_j \in S \setminus \{c_i\}$ to find the optimal preceding node.

- We compute the minimization function for every subset S of cities, and thus takes time $\sum_{k=1}^{n-1} O(\binom{n}{k})$. Therefore, the total tome to compute OPT is:

$$\sum_{k=1}^{n-1} O((\binom{k}{n})k^2) = O(n^2 2^n)$$

- The improvement from $O(n!n)$ in the trivial enumeration algorithm to $O^*(2^n)$ in the dynamic programming algorithm is quite significant.

- However, the algorithm is exponential in space as well as time, meaning that a $O(2^n)$ memory is used as well.

Bar fight prevention - Vertex Cover (k-vertex cover) - mangler forklaring

- **Problem:** You are a bouncer that wants to preemptively stop fights at a bar. Denote a graph G of n nodes, where nodes represent bar guests and edges are fights, i.e. guest a will fight guest b if they share an edge. You can at most reject k guests from the bar. Thus, this problem translates to: Is there a vertex cover of G of size at most k.

- Unfortunately, This is NP-complete, and so the naive approach, i.e. trying all possibilities, runs in $O(2^n)$ time. However, by restricting the parameter k, better solutions may be found.

- **Insigt:** We can always add a node with degree $d \geq k+1$ to the vertex cover, since if you did not, you have to add all $k+1$ neighbours to your vertex cover, breaking the bound in the cover size. (giver rimelig meget sig selv, hvis der er en node der vil "kæmpe" mod flere end der er k, altså har en højere degree, giver det mere mening at fjerne den node, end alle dens naboer.)

- After doing this, all remaining nodes will have at most degree k.

- **Insights:** Every edge has to be covered, and the only way to do this is to include one of its endpoiunts in the vertex cover. Thus, we can proceed by doing the following:

  - For a given edge $\{u, v\}$, try adding u to the vertex cover and run the algorithm recursively to check whether the remaining edges can be using at most $k-1$ vertices. If this succeeds, we have a solution.
  - If not, then replace u by v in the vertex cover and run the algorithm recursively to check whether the remaining edges can be covered using at most $k-1$ vertices.
  - If this also fails, then you are guaranteed that no vertex cover of size k exists.

- **What is the running time?**

- For each recursive call, we decrease k by 1.
- When k reaches 0, all the algorithm has to do is to check if there remaining edges in the graph not covered by the proposed vertex cover.
- Each call spawn two recursive calls, which we do at most k times, giving a total of at most 2^k recursive calls.
- Let m be the number of edges. Each call runs in linear time $O(n+m)$, since we have to if all edges are covered.
- We know something about the number of edges m. Since each node has at most k edges (because we removed nodes with more edges), there iss at most $\frac{nk}{2}$ edges in the graph. Thus, $m = O(nk)$ and $O(n + m) = O(nk)$.
- This gives the total running time $O(2^k nk)$. The naive algorithm that tries all possible subsets of k people runs in time $O(n^k)$. (But wouldnt that mean that the naive approach is better, since it is polynomial)

8 Approximation Algorithms (x2)

Plan

- Min/max approximation problems.
- Approximation Ratio
- Example: Vertex cover
- Example: MAX-CNF (or 3-CNF-SAT)

Notes

Definition and notes:

- Many problems of practical significance are NP-complete, hence they are difficult to compute (see section about NP-completeness for definition.)
- However, they are too important to abandon merely because we do not know how to find an optimal solution in polynomial time.
- We have at least three ways to get arount NP-completeness:
  - For small cases, exponential running time might be perfectly acceptable.
  - We might be able to restrict our problem to special cases which we can solve faster.
  - We might be ok with a solution that is near-optimal.
- We call an algorithm that returns a near-optimal solutions an **approximation algorithm**.
- **Definition:** (Approximation ratio, $\rho(n)$-approximation). *We say that an algorithm for a problem has an* ***approximation ratio*** *of $\rho(n)$ if, for any input if size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:*

  $$max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n)$$

  If an algorithm achives an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-**approximation algorithm**. The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply both to minimization and maximization problems.

  (**NOTES:** Approximation ratio $\rho(n)$: Essentially says how close to optimal solution we actually are - the closer to 1, the better. C is the value of the solution provided by the approximation algorithm. C^* is value of the optimal solution. So we need to show, by the definioton, that C is atleast somewhat good compared to the optimal value C^*, and we then compare to approximation ratio.)
- **Definition:** (Approximation scheme). *An* ***approximation scheme*** *for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1 + \epsilon)$-approximation algorithm. (Giver rimelig meget sig selv imo.)*
- We say that an approximation scheme is a **polynomial-time approximation scheme** if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input.

- We say that an approximation scheme is a **fully polynomial-time approximation scheme** if it is an approximation scheme and its running time is polynomial in both $\frac{1}{\epsilon}$ and the size n of the input instance.
- E.g. a scheme with running time $O((\frac{1}{\epsilon})^2 n^3)$ is such a scheme, while $O(n^{\frac{2}{\epsilon}})$ is not.

  (**NOTES:** Forklar - giver okay mening, hvis du læser)

The Vertex-Cover Problem and proof:

```
Algorithm: Approx-Vertex-Cover
Data: An undirected graph G with n vertices and edges E.
Result: A vertex cover with size no more than twice the size of the optimal vertex cover.

C = empty_set {}
E' = E
While E' != empty_set do
    let (u,v) be an arbitrary edge of E'
    C = C union {u, v}
    remove from E' every edge incident on either u or v.
return C.
```

- The above algorithm is a polynomial-time 2-approximation algorithm (Since we return a vertex cover with size nor more than twice the size of the optimal vertex cover).
- (Give an explanation as to how the algorithm works)
- **Proof:** The running time is $O(n + |E|)$, since we go through all edges E and potentially adds n nodes to C - this is obviously a polynomial running time.
- We have that the set C is a vertex cover, since the algorithm goes through all edges, only removing a given edge e after a vertex has been found (and added to C) which covers e.
- Let A denote the set of edges that are chosen arbitrarily from E' (first thing in the while loop). Any vertex cover, including the optimal cover C^*, must cover the edges in A, meaning this for each edge in A, either of its endpoints must be in the vertex cover. Since no two edges in A share endpoints, at least one node for each edge must be in a vertex cover, and we have a lower bound of:

  $$|A| \leq |C^*|$$

  In addition, by constructing we have that:

  $$|C| = 2|A|$$

From which we can now bound the approximated solution by:

$$|C| = 2|A| \leq 2|C^*|$$

(**NOTES:** FORKLAR)

Randomized Approximation Algorithm for MAX-3-CNF Satisfiability:

We say that a random algorithm for a problem has an **approximation ratio** of $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the C^* of an optimal colution. That is, we lookat expected cost in case of randomized algorithm.

- In **MAX-3-CNF satisfiability**, the input is the same as for 3-CNF satisfiability, and the goal is to return an assignement of the variables that mazimize the number of clauses evaluating to 1. (Essentially maximizing the number of boolean values that evaluate to true).
- We now show that randomly flipping a coin for each variable, giving equal chance of it being 1 or 0, yields a randomized $\frac{8}{7}$-approximation algorithm.
- **Theorem:** *Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, ..., x_n$ amd m clauses, the randomized algorithm that independently sets each variable to 1 with probability $\frac{1}{2}$ and to 0 with probability 0 with probability $\frac{1}{2}$ is a randomized $\frac{8}{7}$-approximation algorithm.*
- **Proof:** Suppose that we have independently flipped a coin for each variable. For $i = 1, 2, ..., m$, we define the indicator random variable:

$$Y_i = \begin{cases} 1 & \text{if clause } i \text{ is satisfied} \\ 0 & \text{otherwise} \end{cases}$$

  So that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in each clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. (Forklar) A clause is not satisfied only if all three of its literals are set to 0, and so Pr[clause i is not satisfied] $= (\frac{1}{2})^3 = \frac{1}{8}$. Thus, we have that $E[Y_i] = 1 - \frac{1}{8} = \frac{7}{8}$ (because it is the opposite happening.)
- We now define $Y = Y_1 + Y_2 + ... + Y_m$ (Y is basically just for all the variables, right?). Then we have:

$$E[Y] = E[\sum_{i=1}^{m} Y_i]$$

By linearity of expectation:

$$E[Y] = \sum_{i=1}^{m} E[Y_i]$$

$$E[Y] = \sum_{i=1}^{m} \frac{7}{8}$$

$$E[Y] = \frac{7m}{8}$$

- Clearly, m is an upper bound on the number of satisfied clauses, since m is the number of clauses, and thus we have $C^* \leq m$. We have also just shown that $C = \frac{7m}{8}$. Hence the approximation ratio is at most:

$$\frac{C^*}{C} \leq \frac{m}{\frac{7m}{8}} = \frac{8}{7}$$

9 Polygonal Triangulation

Plan

- Introduction.
- The art gallery problem.
- Y-monotone pieces.
- Types of vertices we meet during sweep - (Start, Stop/End, Split, Merge, Regular).
- Dividing into y-monotone pieces.
- Triangulating a polygon.
- Example:

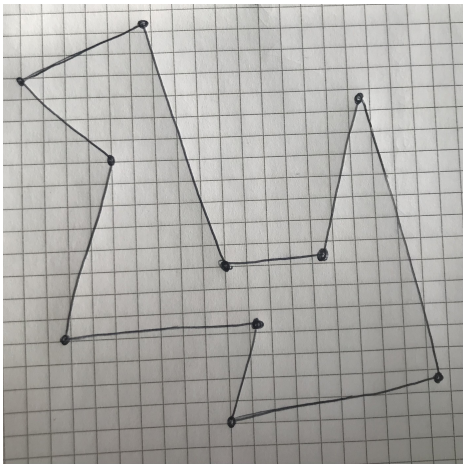

Figure 2: Polygon example

Notes - based on the lecture

Introduction:

- Triangulating a polygon?
- **Motivation:** Two points in a simple polygon can **see** each other if their connecting line segment is in the polygon. (Points are just meant as somewhere on the "problem surface" - that's my word i don't know what it's actually called)
- **The art gallery problem:** How many cameras are needed to guard a given art gallery so that every point is seen.
  - We assume that the space (room) is 2-dimensional.
  - The room can be modelled as a simple polygon (makes sense if you look at what a simple polygon is).
  - We want the minimum amout of cameras to cover the simple polygon (the room) (as to not have excessive spending).
  - In geometry terminologi: *How many points are needed in a simple polygon with n vertices so that every point in the polygon is seen. See the **motivation** as to what is meant by seen.*
  - We have that the optimization problem is computationally difficult (i.e. finding the minimum number of cameras that maximises space seen. This is an NP-hard (actually harder, but doesn't matter to us) problem, i.e. there are no polynomial solutions to it).
  - **Art Gallery Theorem:** $\lfloor n/3 \rfloor$ cameras are occasionally necessary but always sufficient.

    (**NOTES:** Note however that this is a bad case and you generelly don't want that many, however, this will always be enough.).
  - **Why are $\lfloor n/3 \rfloor$ always enough?:** Assume polygon P is **triangulated:** a decomposition of P into disjoint triangles by a maximal set of non-intersecting diagonals.

    (**NOTES:** We just make the polygon into triangles. Each of these triangles have corners, that are also corners in the polygon.)
  - **Diagonal of P:** Open line segment that connects two vertices of P and fully lies in the interior of P. (Just the diagonals making the polygon into triangles - these are not allowed to cross).
  - **Lemma:** A simply polygon with n vertices can always be triangulated, and always with $n-2$ triangles. Can be proved by induction:
    - **Proof - i don't think this is relevant for the exam though:** Induction on n. If $n = 3$, it is trivial (the polygon is already a triangle.)

* Assume $n > 3$. Consider the leftmost vertex v and its two neighbours u and w. Either, uw is a diagnoal (we have a line segment, i don't know how this is made. So it's diagonal to this line segment), this is **case 1**, or part of the boundary of P is in $\triangle uvw$ (**case 2**).
* **Case 2:** Choose the vertex t in $\triangle uvw$ farthest from the line through u and w, then $\bar{vt}$ must be a diagonal.

  (**NOTES:** fuck this proof. I think that case 1 is most likely, if it is a diagonal, we can create a triangle $\triangle uvw$. If case 2, slide to the vertex farthest away from the line through u and w. Then we land in case 2? Ahh, i think we just create case 2 to create another triangle, because if we slide to t we can just make a line through t that is diagonal to uw - if we do this we will create a triangle - with one of the triangle's side being the line through t. But fuck this proof. I don't think you need it at the exam. It's essentially just, if you go to a room, you can always see another corner. Since you can do this, we can always add another diagonal, hence making it into a triangle (Doesn't hold with circular rooms, funnily enough)).

3-coloring proof:

- **Dual graph of the triangulation:** Each face gives a node; Two nodes are connected if the faces are adjacent.

  (**NOTES:** Really annoying way of saying: We add a vertex to each triangle in the triangulation, and an edge between the vertices if the two triangles share a diagonal. Essentially just making a graph inside of the triangulation)

- **Lemma:** The vertices of a triangulated simple polygon can always be **3-colored**.

- **Proof by induction - this proof seems easier:**
  - We do induction on the number of triangles in the triangulation. Base case: True for a triangle (makes sense.)
  - Every tree has a leaf, in particular the one that is the dual graph. Remove the corresponding triangle (the one corresponding to being the leaf) from the triangulated polygon, color its vertices, add the triangle back, and let the extra vertex have the color different from its neighbours. (Actually makes sense, compared to the other)
  - **Why does this show that it's enough?** For a 3-colored, triangulated simple polygon, one of the color classes is used at most $\lfloor n/3 \rfloor$. Place the cameras at these vertices. (E.g. pick all of the red vertices, and they will guard everything)

- The color with the least amount of vertices (being an integer), will have at most $\lfloor n/3 \rfloor$ vertices, and this color can be used to guard the room or polygon or whatever the fuck.
- This argument is called **the pigeon-hole principle**.

Two-ears for triangulation:

- **Two-ears theorem:** An ear consists if three consecutive vertices u, v, w where $u\bar{w}$ is a diagonal.
- He just straigtup ingores this algorithm because its slow

y-monotone pieces:

- **Overview:** A simple polygon is **y-montone** if and only if any horizontal line intersects it in a connected set (or not at all).

  (**NOTES:** Just means that the horizontal line only enters and leaves the polygon ONCE. E.g. if we had a polygon that was constructed in a way so we would enter it (with the line), leave it, and then enter and leave it again, it would NOT be y-montone.)
- We use plane sweep to partition the polygon into **y-monotone** polygons.
- We then triangulate each y-monotone polygon.

  (**NOTES:** We use the plane sweep algorithm to process the polygon and partion it into y-monotone pieces. This is done by adding diagonals in the polygon such that it splits the polygon into smaller polygons, each of which is y-monotone. We we have done this, we triangulate each of the y-monotone polygons. This is because triangulation works well with y-monotone polygons.)
- A y-monotone polygon will always have a top vertex, where both edges goes down, and a bottom vertex where both edges goes up (pretty self-explanatory).
- It will also contain two y-montone chains between top and bottom as its boundary (Chains are just meant as the sides of the polygon.)
- Any simple polygon with one top vertex and one bottom vertex is y-monotone (makes sense - if this is the case, the line will only enter and leave once, making it y-monotone).
- **What types of vertices does a simple polygon have - i.e. what will we encounter when we do plane sweep:**
  - **Start vertices:** Vertices where the edges goes down, and the rest of the polygon is below.

- **Stop/end vertices:** Edges goes up, and the polygon is above.
- **Split vertices:** Edges also goes down, but the there's also a part of the polygon above - the reason for the name split, is because when we pass this vertex with out line, it will split into two.
- **Merge vertices:** Edges goes up, and the polygon is below these vertices - the reason for the name merge vertex, is because when our sweep line crosses a merge vertex, the two pieces it crosses will merge together (There are two piecese the line crosses, since it is not y-monotone).
- **Regular vertices:** One edge goes up, the other down.

Algorithm to divide into y-monotone pieces (He does not mention a name of the algorithm in the lecture.)

- We use the sweep and do different things when encountering different types of vertices.
- We need to eliminate all of the merge and split verticies (as these are partly the reason why it isn't y-monotone.)
- Once these has been eliminated, we will only have y-monotone polygons left.
- A simple polygon with no split or merge vertices can have at most one start and one end vertex (i.e. it's already y-monotone).
- Find diagonals from each merge vertex down, and from each split vertex up.
- The **helper** for an edge e that has the polygon to the right of it, and a poisition of the sweep line, is the lowest vertex v above the sweep line such that the horizontal line segment connecting e and v is inside the polygon.

  (**NOTES:** What this essentially is saying is that: We have the line going through the polygon (the sweep). Whatever edge in the polygon the line **enters** will have the helper v, which will be the lowest vertex in that part of the polygon (does not have to be connected to the edge. It's the same way as saying it is the lowest vertex that can "see" the edge, if we're thinking about the camera problem. That vertex is called the **helper**.))
- When we encounter a **split vertex** we draw the diagonal to the vertex that is nearest.
- We then define a data structure (it is a binary search which we call **status**) that holds these edges and their helpers: The **status** is the set of edges intersecting the sweep line that have the polygon to their right, sorted from left to right, and each with their **helper**.

- This data structure will be updated as we go along the polygon.
- The smart thing is: Whenever we encounter a split vertex, we look at the edge directly to the left of the split vertex. We find this edge in the **status** and connect a diagonal to the helper from the split vertex.
- *The status data structure stores all edges that have the polygon to the right, with their helper, sorted from left to right in the leaves of a balanced binary search tree.*

Main algorithm - uses logic from above section:

- Initialize the event list (all vertices are sorted by decreasing y-coordinate) and the status structure (empty)
- While there are still events in the event list, remove the first (topmost) one and handle it.
- We need to specify what we will do with every type of vertex:
- **Start vertex v:** Since we arrive at a start vertex, the line going through will have that the edge to the left will have the polygon to the right, hence we add the edge with v as its helper in T (being the balanced binary search tree).
- **End vertex v:** The edge to the left will be a part of T, since we have passed at an earlier point. Delete this edge and its helper from T.
- **Regular vertex v:** Two cases:
  - Case 1: If the polygon is right of the two incident edges (i.e the edges both going to v), then replace the upper edge by the lower edge in T, and make v the helper (because we essentially just move through the polygon and we have a new edge that needs to replace the older one, since the line will now pass through this new edge).
  - Case 2: If the polygon is left of the two incident edges, then find the edge e directly left of v, and replace its helper by v (because we now have vertex v that is lower than the current helper of the edge to the left.)
- **Merge vertex v:**
  - Remove the edge clockwise from v from T (since there is a merge vertex, we have that the line enters, leaves, enters, leaves. The second edge it enters will be in T. Remove this edge, as we have passed through it.)
  - Find the edge e directly left of v, and replace its helper by v. (Since v is now lower than the current helper that edge has).

- **Split vertex v:**
  - Find the edge e directly left of v, and choose as a diagonal the edge between its helper and v. (i.e. essentially splitting the polygon, by creating a line from v to the helper of the edge to the left of v.)
  - Replace the helper of that edge e with v (since we have a new lower point that can "see" e.)
  - Insert the edge counterclockwise from v in T, with v as its helper. (Since its a split vertex, after the split vertex the line will do enter, leave, enter, leave. We need to add the second edge it enters to T with v as its helper, as its the lowest the can "see" it.)
- **Efficiency:** Sorting all of the events by y-coordinates takes $O(nlogn)$ time (e.g. by using merge sort). Every event takes $O(logn)$ time, because it only involves querying, inserting and deleting in status structure T (a balanced binary search tree's height/depth is $O(logn)$).
- **Edge cases:** Could be if several vertices had the same y-coordinates. Not relevant here, but should be kept in mind when implementing it.

Representation:

- A simple polygon with some diagonals is a subdivision $\rightarrow$ use a DCEL - a data structure, something with d, connected edge list.
- Its just: We add the diagonals, and it splits the polygon into more polygons, and then we deal with these polygons afterwards - pretty self explanatory.
- After splitting, we perform more sweeping: We sweep upwards in each of these sub polygons, where we can find a diagonal down from every merge vertex (these will essentially be a split vertex, since we are doing an upward sweep).
- We then perform the same steps when we encounter merge vertices (i.e. being the split), making all of the sub polygons y-monotone.

Result:

- **Theorem:** A simple polygon with n vertices can be partioned into y-monotone pieces in $O(nlogn)$ time. (We use $nlogn$ time to sort the corners, and use logarithmic time to update the **status** structure for each vertex.)

Triangulating a monotone polygon:

- How do we triangulate a y-monotone polygon?
- Again, we visit the vertices from top to bottom.

- Once we arrive at a new vertex, we will add as many diagonals as possible. (I.e. the first two vertices we visist, we wont be able to add any, as there will just be an edge between the two. However, the third we can add one, since there are now 3 vertices. We can add a diagonal to create a triangle.)
- If we e.g. have two vertices u and v. If u and v share an edge (i.e. are connected) we can't add a diagonal. If they don't, we can add a diagonal between the two.
- We can only add diagonals between vertices that we have visited.
- Just think of it as: We add diagonals between all vertices that can see eachother, if they don't have an edge connecting them.
- At concave (i.e. a part of the polygon that bends) corners, we can however not add an diagonal (these diagonals will however be added from different vertices when we go down the polygon - this rule just seems stupid)
- More generally: **The algorithm:**
  - Sort the vertices top-to-bottom by a merge of the two chains (chains are just the edges and vertices going down the left and right - this is just a way of saying that we sort it to be able to pick the top one, and know when we are at the bottom).
  - Initialize a stack. Push the first two vertices. (The stack contains all of the vertices where we need to connect the diagonals. The vertices on this stack will essentially form the concave chain, as we haven't added diagonals yet, but will have to later).
  - Take the next vertex v, and triangulate as much as possible, top-down, while popping the stack. (Popping the stack, since we triangulate (add diagonals), so the stack will become smaller).
  - Push v onto the stack. (We still need to add diagonal **to** v - we only have **from** v).
  - Concave chain essentially just means that we have a lot of vertices where we can't add any diagonals. This makes sense if you look at a polygon. At some point you will reach where it curves. Since everything is connecting with edges, and you can't see any of the vertices you have already visitied, you can't add diagonals.
- Again: A simply polygon with n vertices can be partitioned into y-monotone pieces in $O(nlogn)$ time. And a monotone polygon with n vertices can be triangulated in $O(n)$ time (we just add a diagonal for each vertex (i.e. each vertex will be added to the stack and removed from the stack at some point - this takes constant time, hence $O(n)$, since we do it for all vertices).)

- Can we conculde: A simple polygon with n vertices can be triangulated in $O(nlogn)$ time?? Initially we had n edges. We add at most $n-3$ diagonals in the sweeps (i.e. if we meet a merge or split, we add a diagonal). These diagonals are used on both sides as edges. So all monotone polygons together have at most $3n-6$ edges, and therefore at most $3n-6$ vertices. Hence we can conclude that triangulating all monotone polygons together takes $O(n)$ time. (Essentially just saying, when we run the algorithm, we might have more corners than the original polygon (because we add diagonals - i.e. get two corners more). For each split or merge vertex we add 1 diagonal, meaning each of these will add 2 corners to our "instance" all in all. Hence, we only add a linear amount of corners all in all. So after creating the sub polygons y-monotone, we still have $O(n)$ corners).

At the exam:

- Focus on explaining the algorithms - this is the main part.
- Don't use alot of time on the art gallery problem or proving that there exists a triangulation (thank fuck for that).

Notes from Q and A:

- Max flow spørgsmål: min-cut theorem, 3 implications that are part of the proof. 2 implies 3 proof, the long one. Possibly switch out edmond karp proof with max-flow min-cut proof. Or just know and describe the max-flow min-cut and its proof.
- Van emde boas tree: Brug slides fra ny forelæsning, diku notes er fra 2018. Ignore the others tree structures, just mention Van emde boas trees.
- Hashing: Proof: Lecture plan: We should not show that multiply shift being 2 approximately universal??? - læs lecture plan. Do not show that proof, thank fuck for that. Running time of hash table instead, proof of the expected running time. Coordinated sampling.
- exact exponential algorithms: The floored proof
- exact exponential algorithm and parameterized complexity: - Focus more on one topic, mention the other. Proof of running time of MIS - mention that it is wrong.
- Spørgsmål periode - outline of a proof is somewhat acceptable
- Have examples ready on summary (disposition)
- Main part is a large part of the grade, questions can swing grades - allegedly
- Poly: Two algorithms, choose a proof for one of them. Monotone pieces - need to mention?
- Linear Programming: Dont run a full example of simplex! Just run enough to show you know it works. There is too much you can fuck up - potentially come with some examples as to what you can use linear programming for (e.g. max flow).