Course overview

- 10 lectures
- Use Linux, macOS or Ubuntu app on Windows
- 6 labs — use any programming language you are familiar with
- Oral exam in zoom or E:2190: `https://calendly.com/forsete`
- You can book at any time but must have passed all labs before exam.
- If you fail, you can try again after at least one week.
- One exam booking at a time only.
- If you want to ask any question, you can also book at calendly.
- `https://jonasskeppstedt.net` has videos from 2020 but:
  - they are not official course material.
  - they are not being updated
- The course book is available from the Swedish amazon

Course purpose: algorithm design paradigms 1(2)

- Greedy: make decisions based on limited information
- Graph search: e.g. breadth first search and Tarjan's algorithm
- Dynamic programming: make decisions based on enumerating all possibilites — but avoid duplicate work
- Divide and conquer: as in quicksort and mergesort

Course purpose: algorithm design paradigms 2(2)

- Network flow: model a problem as water pipes and maximize amount of water flow
- Linear programming: inequalities and an objective function to maximize
- Integer linear programming: only integer solutions (e.g. number of persons or airplanes)
- Branch-and-bound: paradigm to solve e.g. integer linear programming

Course purpose: data structures

- More about hash tables (`dict` in Python and `Map` in Java)
- More about heaps: Hollow heap which can be faster than binary heaps from EDAA01

Course purpose: complexity

- Time complexity, or execution time, of an algorithm
- Complexity of a problem: is it possible to make a fast algorithm for a problem?
- Problem complexity classes: P and NP and NPC
- What to do if you cannot find an efficient algorithm?

Worst-case execution time with Ordo

- Paul Bachmann introduced the $O(n)$ notation in 1892
- In 1976 Knuth suggested its use in algorithm analysis.
- Let $T(n)$ be the running time of an algorithm.
- n describes the size of the input, e.g. number of array elements to sort
- Sometimes more parameters: e.g. n nodes and m edges
- Sorting 1000 integers is fast but what happens when n is large?

Worst-case execution time with Ordo

- An example: $T(n) = 123n^2 + 45n + 678$
- Ignore lower terms and the constant at n^2
- $O(n^2)$ is a set of functions with a max running time: $c \cdot n^2$ for $n \geq n_0$
- We say $T \in O(n^2)$ due to $T(n) \leq c \cdot n^2$ for $n \geq n_0$ for some c
- Let $f(n) = 124 \cdot n$ and $g(n) = 52 \cdot n^3$.
- Quiz: which of f and g is in $O(n^2)$?

Answer plus more

- Which of $f(n) = 124 \cdot n$ and $g(n) = 52 \cdot n^3$ are in $O(n^2)$?
- Only $f \in O(n^2)$ since with large n, we have $g(n) \geq c \cdot n^2$, obviously.
- When an algorithm is analyzed we want to find the smallest bound.
- If we know the runtime is at least $h(n)$ then we can use $\Omega(h(n))$
- So: $f \notin \Omega(n^2)$
- and: $g \in \Omega(n^2)$
- and: $T \in \Omega(n^2)$
- With $T(n) = 123n^2 + 45n + 678$, $T \in \Omega(n^2)$ and $T \in O(n^2)$: $c_1 n^2 \leq T(n) \leq c_2 n^2$
- We write $T \in \Theta(n^2)$
- Many use the notation $f(n) = O(h(n))$
- A trend seems to be to use $\in$ instead which I prefer so we can use normal meaning of $=$

Examples of efficient algorithms: $O(n^k)$

- An algorithm with polynomial running time is regarded as efficient.
- At least in comparison with slower algorithms.
- $O(\log n)$: searching in a sorted array
- $O(n+m)$: visiting all n nodes in a graph with m edges
- $O(n \log n)$: sorting an array
- $O(n^2)$: two for loops
- Quiz: you have points in a plane and want to find a pair of points with minimal distance. How can you do that?

Answer

- One can use two for-loops.
- For each point, find the distance to every other point.
- $O(n^2)$
- This is "efficient" according to theory.
- It is too slow in practice for large number of points.
- Quiz: how long time would it take to find the closest pairs if there are 10^9 pairs?
- An hour or a day? Any guess?

Examples of inefficient algorithms

- $O(2^n)$: all subsets of n objects
- $O(n!)$: all permutations of n objects

A model of a 4 GHz modern CPU

n	n	$n \log n$	n^2	n^3	1.5^n	2^n	$n!$
10	2.5 ns	8.3 ns	25.0 ns	250.0 ns	14.4 ns	256.0 ns	907.2 μs
11	2.8 ns	9.5 ns	30.2 ns	332.8 ns	21.6 ns	512.0 ns	10.0 ms
12	3.0 ns	10.8 ns	36.0 ns	432.0 ns	32.4 ns	1.0 μs	119.8 ms
13	3.2 ns	12.0 ns	42.2 ns	549.2 ns	48.7 ns	2.0 μs	1.6 s
14	3.5 ns	13.3 ns	49.0 ns	686.0 ns	73.0 ns	4.1 μs	21.8 s
15	3.8 ns	14.7 ns	56.2 ns	843.8 ns	109.5 ns	8.2 μs	5 min
16	4.0 ns	16.0 ns	64.0 ns	1.0 μs	164.2 ns	16.4 μs	1 hour
17	4.2 ns	17.4 ns	72.2 ns	1.2 μs	246.3 ns	32.8 μs	1.0 days
18	4.5 ns	18.8 ns	81.0 ns	1.5 μs	369.5 ns	65.5 μs	18.5 days
19	4.8 ns	20.2 ns	90.2 ns	1.7 μs	554.2 ns	131.1 μs	352.0 days
20	5.0 ns	21.6 ns	100.0 ns	2.0 μs	831.3 ns	262.1 μs	19 years
30	7.5 ns	36.8 ns	225.0 ns	6.8 μs	47.9 μs	268.4 ms	10^{15} years
40	10.0 ns	53.2 ns	400.0 ns	16.0 μs	2.8 ms	5 min	10^{31} years
50	12.5 ns	70.5 ns	625.0 ns	31.2 μs	159.4 ms	3.3 days	10^{47} years
100	25.0 ns	166.1 ns	2.5 μs	250.0 μs	3 years	10^{13} years	10^{141} years
1000	250.0 ns	2.5 μs	250.0 μs	250.0 ms	10^{159} years	10^{284} years	huge
10^4	2.5 μs	33.2 μs	25.0 ms	4 min	huge	huge	huge
10^5	25.0 μs	415.2 μs	2.5 s	2.9 days	huge	huge	huge
10^6	250.0 μs	5.0 ms	4 min	8 years	huge	huge	huge
10^7	2.5 ms	58.1 ms	7 hour	10^4 years	huge	huge	huge
10^8	25.0 ms	664.4 ms	28.9 days	10^7 years	huge	huge	huge
10^9	250.0 ms	7.5 s	8 years	10^{10} years	huge	huge	huge

The choice of algorithm is more important than CPU, language or compiler.
But for a given algorithm, they certainly can matter a lot.

From a book by two famous theoretical computer scientists

Sedgewick and Flajolet in "An Introduction to the Analysis of Algorithms":

The quality of the implementation and properties of compilers, machine architecture, and other major facets of the programming environment have dramatic effects on performance.

Matchings

- Most of the rest of this lecture is about **matchings** and Lab 1
- Given two sets $X = \{x_1, x_2, ..., x_n\}$ and $Y = \{y_1, y_2, ..., y_n\}$.
- A matching M is a set of pairs (x_i, y_j) such that an $x \in X$ and an $y \in Y$ appear in at most one pair.
- Matchings can be used for many things:
  - university admission: n students and n places at universities
  - medical training: n medical students and n internships
  - not so realistic but lab 1 is about summer jobs: students and companies
- The size of M may be less than n.
- If all are matched, it is called a **perfect matching**.

The Stable Matching Problem

- *The Swedish National Bank's Prize in Economic Sciences in Memory of Alfred Nobel* year 2012 was awarded for solving a problem called the **Stable Matching Problem** — this problem was called something else until recently.
- Wikipedia and other sources call it the Stable Marriage Problem but it is an overly simplified model for winning somebody's heart.
- In the videos there is an example from Röde Orm who has fallen in love with princess Ylva, daughter of King Harald Blåtand.

Lab 1: students and summer jobs

- Assume each company has exactly one job offer
- Each company has a preferred list of students, sorted in descending order, and similarly for students.
- We assume a student s_i applies to a company c_j which answers yes or no — if yes then (s_i, c_j) are matched temporarily in a pair
- If later another student s_k applies to c_j and it says yes, a new pair is created and the old no longer exists
- How can we create a perfect matching with no pairs wanting to split (ie quit the job or reject the student)?

Three famous Swedish companies

Stora	1288	world's oldest company that is still active	forest
Uddeholm	1720	founded as Sunnemo bruk 1640	steel
Spotify	2006		

Stable and unstable matchings

- Split: a company C in one pair and a student S in another pair quit/reject their matching and create (S, C)
-
who	preference lists with most liked first
Harald	Stora, Uddeholm
Ingrid	Uddeholm, Stora
Stora	Harald, Ingrid
Uddeholm	Ingrid, Harald

- It is easy to create a perfect and stable matching: $S = \{$(Harald, Stora),(Ingrid,Uddeholm)$\}$
- In $U = \{$(Harald, Uddeholm),(Ingrid,Stora)$\}$ both pairs want to split
- U is called an **unstable matching**

Stable and unstable matchings

- So, a matching is **unstable** if it contains two pairs (s_i, c_j) and (s_k, c_l) such that at least one of the following is true:
  - s_i prefers c_l and c_l prefers s_i, or
  - c_j prefers s_k and s_k prefers c_j.
- A stable matching is a perfect matching with no unstable pairs.
- Is it always possible?
- We are not trying to find a matching in which every person is paired with their favorite company or partner — most likely impossible
- The reason the Nobel prize winners worked on this problem was to make matchings for medical students simple and without chaotic change requests

The problem

- So how can we find a perfect matching which is stable?
- Or, how can we efficiently find a matching without any unstable pairs?
- We will next show an algorithm for finding stable perfect matchings
- We will then analyze its time complexity
- After that we will show it is correct

The Gale-Shapley algorithm

procedure $GS(S, C)$
    /* S is a set of n students and C is a set of n companies */
    add each student $s \in S$ to a list p
    while $p \neq null$
        $s \leftarrow$ take out the first element from p
        $c \leftarrow$ the company s prefers the most **and**
            s has not yet applied to
        if c has no student **then**
            (s, c) becomes a pair
        else if c prefers s over its current student s_c **then**
            remove the pair (s_c, c)
            (s, c) becomes a pair
            add s_c to p
        else
            add s to p

Sorted preference lists

- Recall both students and companies have a sorted list of preferred matchings
- For a student to find the next company to apply to, it needs just to remember where in the list it currently is.
- So the list can be an array and an index variable is used to find c and then that index variable is incremented. One operation.
- But for a company to answer yes or no, it must check who of s and s_c comes first in its preference list.
- It seems it must go through its list each time somebody applies which obviously takes more time. With n students, this search may need n operations.

Time complexity

- How fast is then the GS algorithm?
- We want an estimatation based on the input size parameter n
- We have no obvious answer based on e.g. two for-loops.
- GS is more complicated since we can put back a student in the list!
- Will this algorithm even terminate?

Algorithm termination

- When it is not obvious to determine the number of iterations, we should try to find what kind of **progress** is made each iteration

Lemma

The GS algorithm terminates after at most n^2 iterations.

Proof.

Each student has n companies in its preference list, so it can make at most n applications. In each loop iteration it can apply to one company. There are n students so we have at most n^2 loop iterations. $\square$

- We assumed an application is a quick operation
- If a company must check its list each time, we would get $O(n^3)$
- But at least the algorithm terminates after at most n^2 applications

Constant time reply

- An obvious way to check which of two students a company prefers is to search its preference list to see who comes first.
- But how can it determine this without searching through her preference list?
- Any suggestions?

Hint for lab 1

- Assume a preference list is: $4, 2, 1, 3$. Student 4 is most preferred.
- The companies should not store students as a preference list.
- Instead the position in the above list should be stored for each student.
- Thus: $3, 2, 4, 1$. This says student number 1 comes at position 3 above, and student number 4 at position 1.
- Just look at some array[k] to see how much a company wants s_k

Algorithm output: a stable matching

Facts

- *A company is matched from the point a student first applies to it.*
- *A company is matched with increasingly preferred students.*
- *A student is matched with decreasingly preferred companies.*

Lemma

If a student is free, there remains a company it has not applied to.

Proof.

Assume in contradiction s is free and has already applied to all n companies. Since every company is matched all n students are also matched, which is a contradiction since we assumed s is not matched and there are n students. $\square$

Perfect matching

Lemma

The GS algorithm produces a perfect matching.

Proof.

Assume in contradiction the while loop terminates with a student s free due to it has applied to every company. This cannot happen since it contradicts the previous lemma. Therefore GS terminates with a perfect matching. □

Stable matching

Lemma

The GS algorithm produces a stable matching M.

Motivation — see book for a more formal looking proof.

- Assume: M is not stable due to $\{(Harald, Uddeholm), (Ingrid, Stora)\} \subseteq M$ but Harald and Stora **both** want to be matched with each other.
- Then we have two cases:
  1. If Harald did not apply to Stora then he does not like Stora
     - Uddeholm comes before Stora in Harald's preference list
  2. If Harald did apply to Stora then Stora does not like him
     - Stora either said no to Harald or rejected him later for somebody else
     - Eventually Stora accepted and employed Ingrid
- In either way M is not unstable due to Harald and Stora (or any others)
- This may look like an example only but if we treat the above names as variables, it is a normal proof.

Valid and best company

- For a student s a company c is **valid** if (s, c) is a pair in a stable matching.
- The **best** company c is the company most preferred by s which is valid for it.

Theorem

The GS algorithm produces the stable matching $\{\ (s, c) \mid c \ =\ best(s)\ \}$.

- In other words, the matching is unique. So it does for instance not matter in which order the students are put in a list initially.

Proof sketch by contradiction

- Assume $(s, c) \in S$ but $c \neq best(s)$ for some student.
- This s was rejected by $best$(s) otherwise s would be matched with it
- Consider the first time *any* student, say Harald, is rejected by a company c valid for it
- Harald was either rejected when he applied or later
- c must be $best(Harald)$
- Why?
  - because Harald applies according to his preference list
  - $best(Harald)$ is first valid company who rejected him
  - So no other valid company could have rejected him before $best(Harald)$
- From that point c is matched with a student s_c which c prefers over Harald (c either was already matched with s_c or replaced s with s_c).
- Let c be Uddeholm and s_c be Ingrid

Continued

- What we know so far about the preference lists:
  Uddeholm: ... Ingrid ... Harald ...
  Harald: ... Uddeholm ...
  Ingrid: ... Uddeholm ...
- Since Uddeholm is a valid matching for Harald, $(Harald, Uddeholm)$ is a matching in some other stable matching T
- In T, Ingrid is not matched with Uddeholm
- Assume $(Ingrid, Spotify) \in T$
- Which of the following?
  Ingrid: ... Spotify ... Uddeholm ...
  Ingrid: ... Uddeholm ... Spotify ...
- Does Ingrid prefer Spotify or Uddeholm and in that case why?

Continued

- Recall: in S the rejection of Harald by Uddeholm was the *first* rejection by a company valid for any student
- So in S Ingrid cannot have been rejected by Spotify before Harald was
- Since Ingrid applied to Uddeholm before applying to Spotify in S, it must be the case that Ingrid prefers Uddeholm over Spotify.

Uddeholm:	... Ingrid ... Harald ...
Harald:	... Uddeholm ...
Ingrid:	... Uddeholm ... Spotify ...

Continued

Uddeholm:	... Ingrid ... Harald ...
Harald:	... Uddeholm ...
Ingrid:	... Uddeholm ... Spotify ...

- We know that Uddeholm prefers Ingrid over Harald since it rejected Harald for Ingrid in S.
- Recall: $\{(Harald, Uddeholm), (Ingrid, Spotify)\} \subseteq T$
- T is unstable due to Uddeholm and Ingrid, and our first assumption must have been false and therefore we see that Harald is matched with $best(Harald)$.

Is Gale-Shapley fair?

- We have just proved that the GS algorithm finds the best company for students.

Theorem

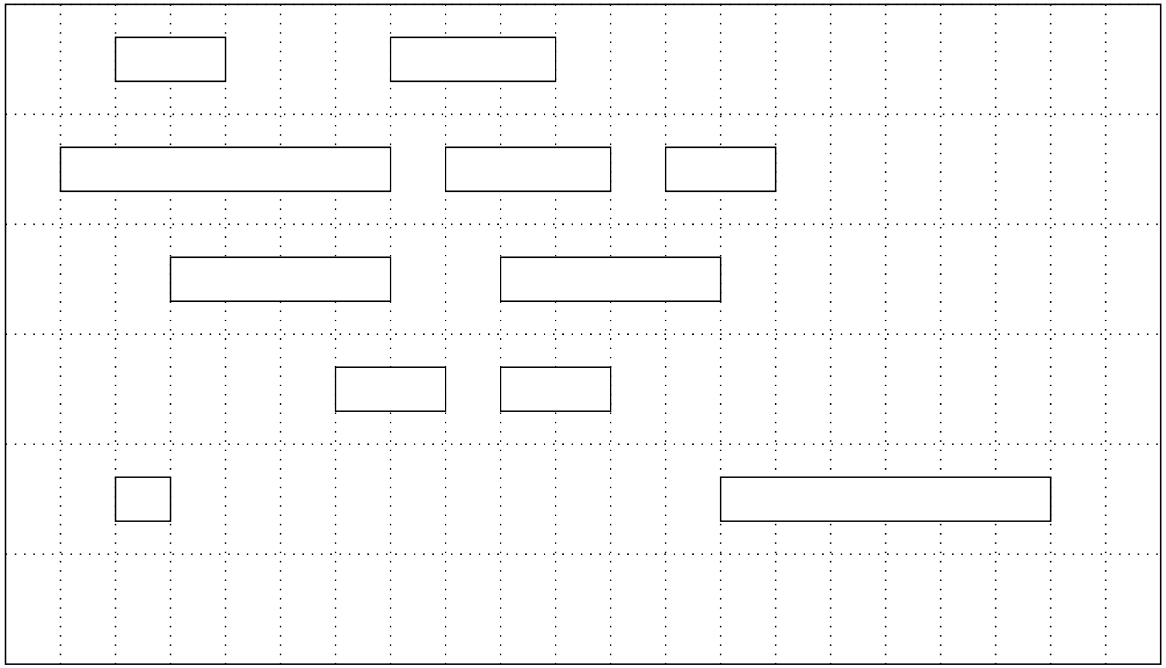
The GS algorithm produces the stable matching which is worst for companies.

- We will use contradiction again. S is a stable matching made by GS
- Assume $(Harald, Uddeholm) \in S$ and Harald is not the worst for Uddeholm
- That is: not the worst in a *stable matching*
- We know $Uddeholm = best(Harald)$ from the previous theorem
- Assume Uddeholm thinks Ingrid is worse than Harald
- Consider another matching T with $(Ingrid, Uddeholm) \in T$
- But we know Uddeholm prefers Harald over Ingrid and Uddeholm is $best(Harald)$
- Thus Uddeholm and Harald make T unstable, i.e. a contradiction

Five representative problems

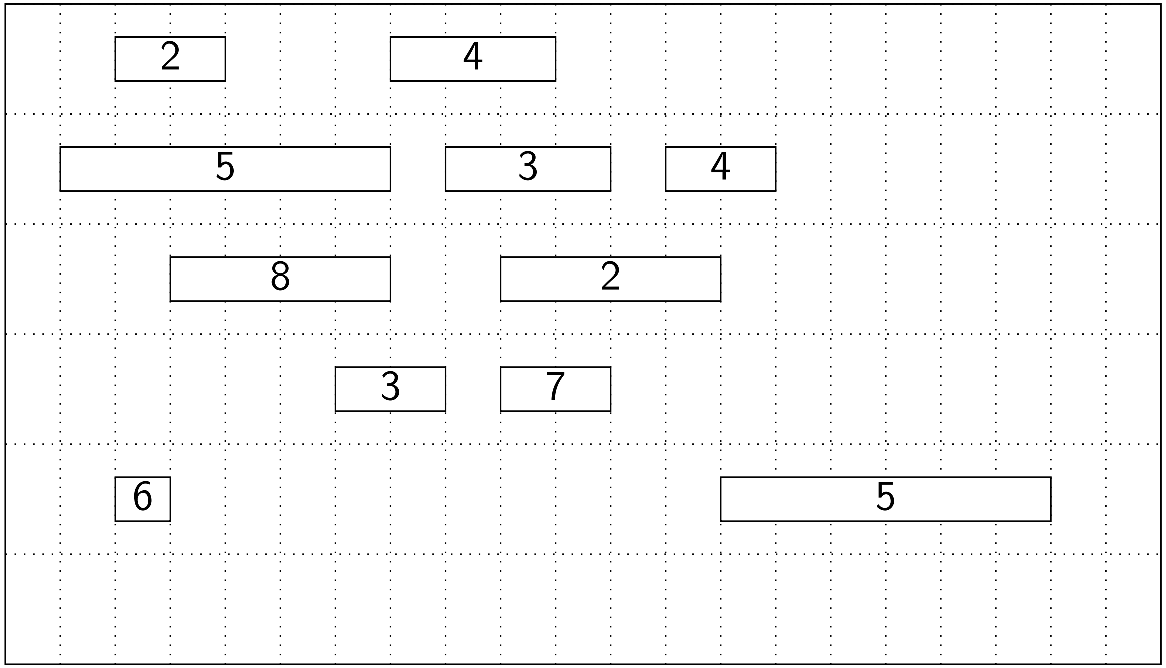
- Interval scheduling / Intervallschemaläggning
- Weighted interval scheduling / Viktad intervallschemaläggning
- Bipartite graph matching
- Independent set / Oberoende mängd
- Chess

Interval scheduling: can be solved by a greedy algorithm



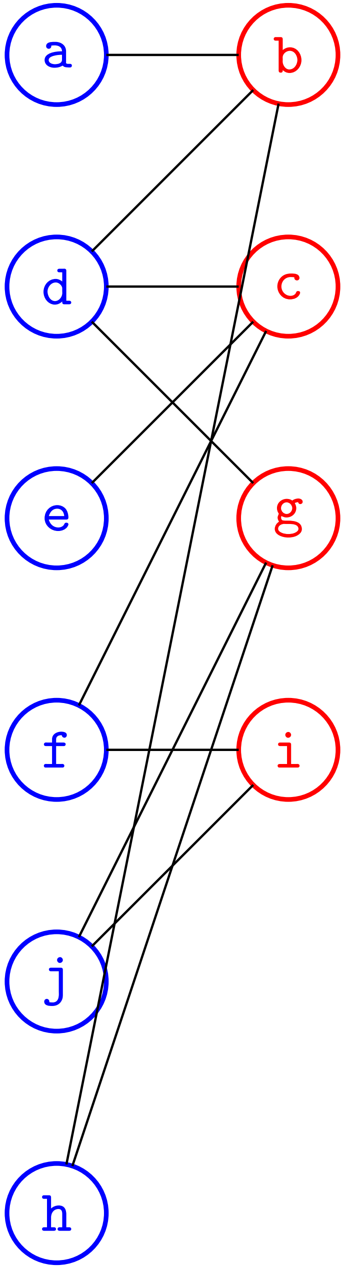
- The boxes are requests with start and finish times
- Time goes from left to right
- We want to find as many non-overlapping intervals as possible
- This problem can be solved by making simple "local" decisions
- By local is meant that it is sufficient to make a decision without analyzing the consequences for the next decision
- Topic of Lecture 3

Weighted interval scheduling: dynamic programming



- Each box has a weight, or value
- We want to maximize the sum of values of selected boxes.
- It is impossible to just look at a box to decide if it should be selected or not
- Two cases for each box: (1) select it, or (2) skip it
- We evaluate the optimal value for both cases and take the best
- This may sound time consuming but we will see a neat trick in Lectures 6 and 7

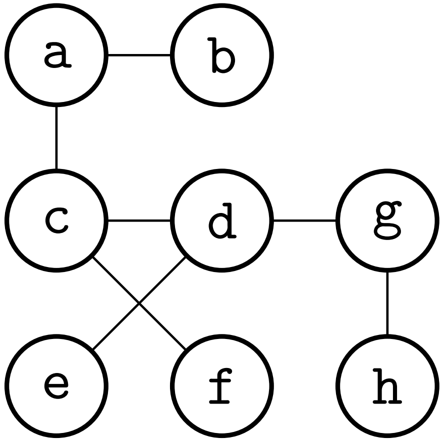
Bipartite graph matching



- In a bipartite graph the nodes can be partitioned in two sets
- No edge between nodes in the same set
- We seek a matching of blue and red nodes
- Similar to Stable Matching but fewer edges here
- If Students = blue nodes and Companies = red nodes there would have been an edge between every student and company in Stable Matching
- A matching M is a set of edges and a node must be an endpoint of at most one edge in M
- We want to find an as large matching as possible
- The algorithm design technique used for this problem is called network flow and is the topic of Lecture 8

Independent set / Oberoende mängd

- Let $G(V, E)$ be an undirected graph and $S \subseteq V$



- S is an **independent set** if for no nodes $u, v \in S$ we have $(u, v) \in E$
- The problem is to find an S with maximum size
- Two independent sets of size four:
  - $S_1 = \{b, c, e, g\}$
  - $S_2 = \{a, e, f, g\}$
- If you can write a fast program for this you win USD 1,000,000 from Clay Institute of Mathematics
- This is an NP-complete problem and the topic of Lecture 9.

NP-complete problems

- A requirement for NP-complete problems, is that a proposed solution to a problem can be checked easily
- If somebody has a solution to Independent Set, it is easy to check if any nodes in S have an edge connecting them in the original graph.
- It is not known if NP-complete problems really are impossible to solve with an efficient algorithm
- For small and special cases they can be
- You will see a very cool proof in Lecture 9 about electronic circuits and NP-completeness

Chess

- If you play a game of chess against Magnus Carlsen and he tells you he wins in 10 moves it is not easy to quickly check if that is true
- You must consider all moves you can make and all moves he can make which is more complicated than checking if S is an independent set
- Of course, for certain chess games you may only have one valid move to make in each of these 10 moves
- One can also argue that chess with about 10^{120} possible positions is a finite game and the optimal move for every position can be stored in a table, but that table would need more entries than the estimated 10^{80} atoms in the known universe
- Thus, there are problems more complicated than the NP-complete ones
- Or, we can say we create huge table with all possible chess positions but that is only possible in dreams

What to do now?

- It is a good idea to start preparing lab 1
- Download the documentation
- Either see mail or link to Tresorit at `https://cs.lth.se/edaf05`
- Think through how to fix the constant time reply to an application