

Contents

1	Lecture 1	<2017-09-05 Tue>	3
1.1	Algorithm		3
1.1.1	Examples of algorithms		3
1.2	Teacher's website		4
1.3	Stable matching		4
1.3.1	Ex: $n = 4$		4
1.4	Stable Matching Algorithm		5
1.4.1	For our example:		5
1.4.2	Modified example		6
1.4.3	Why isn't this infinite?		6
2	Lecture 2	<2017-09-07 Thu>	7
2.1	Announcements		7
2.2	Recall:		7
2.2.1	Ex: $n = 4$		7
2.2.2	Last lecture we proved:		8
2.3	Stable Matching Algorithm		8
2.3.1	Does this algorithm produce a stable marriage?		8
2.3.2	Is this algorithm better for men or women?		9
2.4	Notes on problems		9
2.5	Some example problems		10
2.5.1	Interval scheduling		10
2.5.2	Weighted Interval Scheduling		11
3	Lecture 3	<2017-09-12 Tue>	13
3.1	Running Time Analysis		13
3.1.1	Questions		13
3.1.2	Efficiency		14
3.1.3	Big-O notation		16
3.1.4	Ω -notation:		17
4	Lecture 4	<2017-09-14 Thu>	18
4.1	Recall:		18
4.2	Examples		19
4.3	Θ -notation:		19
4.3.1	Examples		19
4.4	Theorem		19
4.4.1	Proof		19

4.5	Little o and Little omega	20
4.6	Theorem	20
4.7	Stable Marriage	21
5	Lecture 5 <2017-09-21 Thu>	22
5.1	Graphs	22
5.1.1	Undirected Graphs	22
5.1.2	Example applications	22
5.1.3	Ways of implementing in a program	23
5.1.4	Paths and connectivity	23
5.1.5	Cycles	24
5.1.6	Trees	24
5.1.7	Connectivity	24
5.1.8	Breadth-first search	25
6	Lecture 6 <2017-09-26 Tue>	25
6.1	Stable Marriage	25
6.2	Priority Queue	27
6.2.1	Heap Data Structure	28
7	Lecture 7 <2017-09-28 Thu>	33
7.1	Graph Exploration Algorithms	33
7.1.1	Breadth-First-Search (BFS)	33
7.1.2	Depth-First-Search (DFS)	35
7.2	Data Structure for Graphs	35
7.3	Bipartites	36
7.3.1	Testing for bipartites	37
7.4	Directed Graphs	38
7.4.1	Data Structure:	38
8	Lecture 8 <2017-10-03 Tue>	39
8.1	Directed Graphs	39
8.1.1	Graph search	39
8.1.2	Strong Connectivity	40
8.2	Directed Acyclic Graphs	40
8.2.1	Lemma	41
8.2.2	Lemma	41
8.2.3	Lemma	41

9	Lecture 9	<2017-10-05 Thu>	42
9.1	Greedy Algorithm		42
9.1.1	Interval scheduling		43
9.1.2	TODO clean up this section		44
9.1.3	Theorem		45
9.1.4	Satisfying requests		46

1 Lecture 1 <2017-09-05 Tue>

1.1 Algorithm

- Al-Khwarizmi (9th Century)
- Algorismus (Latin)
- Arithmos (Greek)
 - Greek + Latin > Algorithm

A set of step by step instructions

1. Every step simple and precise
2. Produces an answer in finite time (not run forever)

This course will be very rigorous, lots of proofs, but it will take 2-3 months to formally define algorithms, so we'll just have to be satisfied with this definition. Formalized in 1930's by Turing and Church.

- Covered in COMP 330

Even though the concept of an algorithm is very simple and intuitive, it's not very obvious to prove things.

- Algorithms are an old concept, have been studied forever. Some examples are really old

1.1.1 Examples of algorithms

- Recipes
- 1600 BC Babylonians (Factorization and square roots)
- Euclid's Algorithm (200 BC)

- Finding greatest common divisor of 2 numbers

Field of theoretical computer science is much older than first computers.

- Really mature field.
- Computers are just a device that helps us use these things.
- Theoretical computer science is part of math and science and has been studied for millennia

1.2 Teacher's website

<http://www.cs.mcgill.ca/~hatami/>

- He will be following the textbook.

1.3 Stable matching

n men: m_1, m_2, \dots, m_n n women: w_1, w_2, \dots, w_n

Every man and woman has a ranking of people of other gender.

1.3.1 Ex: $n = 4$

$m_1 : w_3 > w_1 > w_2 > w_4$
 $m_2 : w_1 > w_4 > w_2 > w_3$
 $m_3 : w_1 > w_2 > w_4 > w_3$
 $m_4 : w_2 > w_3 > w_4 > w_1$

$w_1 : m_4 > m_2 > m_1 > m_3$
 $w_2 : m_1 > m_2 > m_3 > m_4$
 $w_3 : m_2 > m_1 > m_3 > m_4$
 $w_4 : m_4 > m_1 > m_2 > m_3$

1. A pairing

- $m_1 + w_2$
- $m_2 + w_4$
- $m_3 + w_1$
- $m_4 + w_3$

What is unstable about this? The last pair, m_4 and w_3 .

- m_1 and w_3 prefer each other
2. Unstability: If there is a pair (m, w) such that
 - (a) m prefers w to his current partner
 - (b) w prefers m to her current partner
 - (c) Selfish agents, everyone wants to be with the best possible partner they can find
 3. Problem: Can we find a stable matching?

1.4 Stable Matching Algorithm

while \exists a free man \underline{m}

- m proposes to the highest-ranked woman \underline{w} that he has not proposed yet
- If \underline{w} is free or prefers m to her current partner, she gets engaged to \underline{m} and her current partner becomes free

else

- She rejects \underline{m} and \underline{m} remains free

End while

1.4.1 For our example:

- m_1 proposes to w_3 , accepts $> m_1 + w_3$
- m_2 proposes to w_1 , accepts $> m_2 + w_1$
- m_3 proposes to w_1 , rejects
 - m_3 proposes to w_2 , accepts $> m_3 + w_2$
- m_4 proposes to w_2 , rejects
 - m_4 proposes to w_3 , rejects
 - m_4 proposes to w_4 , accepts $> m_4 + w_4$

Simple example, no one broke up. Let's change the example a bit.

1.4.2 Modified example

$m_1 : w_3 > w_1 > w_2 > w_4$
 $m_2 : w_1 > w_4 > w_2 > w_3$
 $m_3 : w_1 > w_2 > w_4 > w_3$
 $m_4 : w_2 > w_3 > w_4 > w_1$

$w_1 : m_4 > m_2 > m_1 > m_3$
 $w_2 : m_1 > m_2 > m_3 > m_4$
 $w_3 : m_2 > m_4 > m_3 > m_1$
 $w_4 : m_4 > m_1 > m_2 > m_3$

- m_1 proposes to w_3 , accepts
- m_2 proposes to w_1 , accepts
- m_3 proposes to w_1 , rejects
 - m_3 proposes to w_2 , accepts
- m_4 proposes to w_2 , rejects
 - m_4 proposes to w_3 , accepts, breaks up with m_1
- m_1 proposes to w_1 , rejects
 - m_1 proposes to w_2 , accepts, breaks up with m_3
- m_3 proposes to w_4 , accepts

1.4.3 Why isn't this infinite?

$P(t)$: Number of pairs (m, w) such that m has not proposed to w yet at time t (number of iterations of while loop). $P(0) = n^2$ $P(1) = n^2 - 1$ Is it possible that a man proposes to a woman more than once? No.

1. Fact: No man proposes to the same woman more than once.
Some of these proposals may never happen.
The quantity P will never go negative.
2. Fact: $P(t)$ decreases by 1 at every iteration.
3. Lemma: The algorithm terminates after at most n^2 iterations. There will be no free men at the end.

4. Fact: Once a woman gets a proposal, she is never free again.
 - \implies If a man m remains free by the end of the alg it means that at the end all women are engaged. \implies Since there are n men and n women this means that all men are engaged as well.
 - \implies At the end every person is engaged.
- This algorithm gives us a pairing.
 - But we need to show that this is a good pairing, that it's stable.

2 Lecture 2 <2017-09-07 Thu>

2.1 Announcements

- Lectures will be recorded.
- Assignment 1 to come out soon, probably early next week.

2.2 Recall:

Stable matching n men n women.

- Not a fundamental problem, but contains many of the elements we'll see later in this course

2.2.1 Ex: $n = 4$

Man	Preference	1	2	3	4
m_1 :	$w_3 >$		$w_1 >$	$w_2 >$	w_4
m_2 :	$w_1 >$		$w_4 >$	$w_2 >$	w_3
m_3 :	$w_1 >$		$w_2 >$	$w_4 >$	w_3
m_4 :	$w_2 >$		$w_3 >$	$w_4 >$	w_1

Woman	Pref	1	2	3	4
w_1 :	$m_4 >$	$m_2 >$	$m_1 >$	m_3	
w_2 :	$m_1 >$	$m_2 >$	$m_3 >$	m_4	
w_3 :	$m_2 >$	$m_4 >$	$m_3 >$	m_1	
w_4 :	$m_4 >$	$m_1 >$	$m_2 >$	m_3	

Same example as last lecture, see matching/use of algorithm in lecture

1. Matching becomes:

$$\begin{array}{cccc} m_1 & m_2 & m_3 & m_4 \\ \hline w_2 & w_1 & w_4 & w_3 \end{array}$$

Top matched with bottom. Does w_1 have a tendency to break up and go with m_3 ? No.

2.2.2 Last lecture we proved:

1. The algorithm always terminates.
 - Easy to see from the list of preferences, because we go down the list of the men's preferences, they always go down their list and never go back
2. When the algorithm terminates everybody has a partner.
 - Won't end up with a situation where a man proposes to everyone and gets rejected
 - Women will never be free once they are initially proposed to
 - A man can't be free at the end, because that means all women we're proposed to and all women are married
 - But same amount of women and men

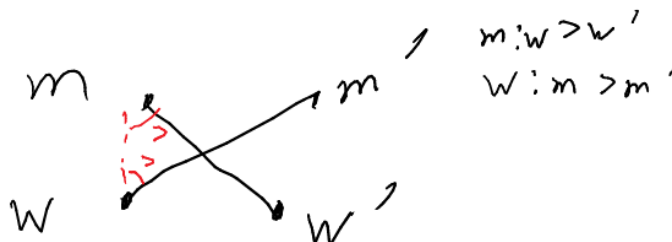
2.3 Stable Matching Algorithm

2.3.1 Does this algorithm produce a stable marriage?

It remains to show that the output is stable.

1. Observation 1
 - Throughout the algorithm every man's partner gets worse and worse
2. Observation 2
 - However, for women it is the opposite
 - They accept the first proposal
 - But every partner gets better and better
3. Theorem: The final matching is stable.

- (a) Proof: Suppose not! Then there exists engaged pairs as in the following:



But in this case m would have proposed to w before proposing to w' , and as a result we know that w would not have ended up with someone worse than m .

2.3.2 Is this algorithm better for men or women?

Let's say (m, w) is valid if there exists some stable matching that pairs m and w .

- Fact: This algorithm matches every man with their most preferred valid w and every woman with their least preferred valid m .
 - For men, they start ambitiously and go for their most preferred partner and go down the list if needed
 - For women, they start at whatever is first given and only improve if needed
 - Formal proof in textbook, won't do it in class as to spend less time on this problem

2.4 Notes on problems

- Formulate the problem as a precise mathematical problem.
 - What is the input?
 - What is the goal?
 - Conditions we want to satisfy?
 - Everything must be precise or else we won't be able to satisfy all these things.

- Design an algorithm
- Analyze the algorithm:
 - It always terminates
 - * Show that, no matter the input, it will always stop, no infinite loop
 - It outputs the correct output!
 - * In stable marriage, we showed that it is always stable
 - Running time
 - * How long does it take to terminate?
 - For stable marriage, we could brute force and try all possible combinations and see if they're stable or not, but that would be $n!$

Professor won't do much on first point, about formulating problem as math. Textbook often presents the problem in a bunch of sentences for some real life thing and we need to extract the mathematical problem from there, which the professor isn't a big fan of.

2.5 Some example problems

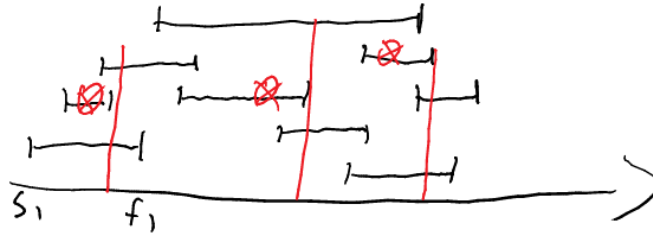
2.5.1 Interval scheduling

- Let's say you have a room and want to rent it out
- Bunch of offers that say the person wants to use the room from a start time to an end time
- Want to accomodate as many people as possible, but we can't have overlap
 - Maximize number of offers without overlap

1. Input:

- n requests
- Starting time s_1, s_2, \dots, s_n
- Finishing time f_1, f_2, \dots, f_n
- Such that $s_i < f_i$

2. Problem We want to pick the max number of these tasks s.t. no two overlap. (Maximum bookings, not maximum time, not charging per hour)



(a) Algorithm?

- What algorithm is good for this?
- Pick next available room that finishes the earliest and keep going
- **Greedy algorithm**

2.5.2 Weighted Interval Scheduling

- Now every offer comes with some value.
- v_1, \dots, v_n
 - where v_i is the value we get from accomodating the i^{th} offer.
- s_1, \dots, s_n
- f_1, \dots, f_n

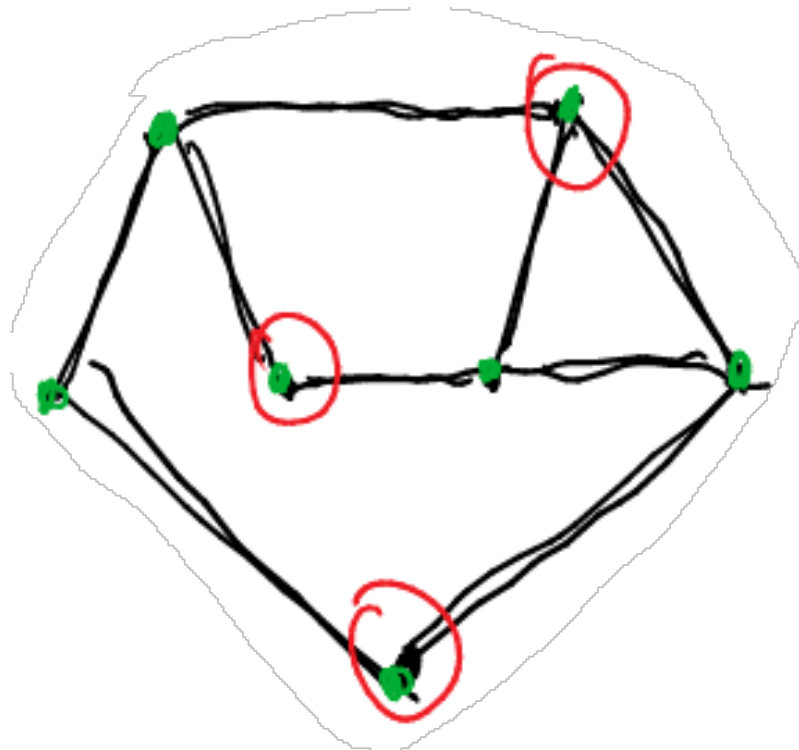
Want higher value, rather than most matchings Why is this harder to solve than the previous problem? Because the previous one is a special case of the first.

- Reduction = reducing this problem to the previous to get an answer
- Setting $v_1 = \dots = v_n = 1$ solves the previous problem.

We will solve this using **Dynamic Programming**

- Create huge table, keep filling it up as you process input

- Solve solution for smaller version of problem and keep expanding based on that
 - Let's say $A[t] = \text{max value if we stop at time } t$.
1. Independent set problem



Independent set: A set of nodes, no two are adjacent. Find the largest independent set.

- Obvious way of doing it without concerning ourselves with time?
 - Brute force
 - Without that? You can do some heuristics, but,
 - It is widely believed that every algorithm for this problem is of brute force nature: It is more or less checking all the possible subsets?
- (a) P vs NP?

- Most important problem in computer science
- Common belief: $P \neq NP$
- This is an example of a problem which is believed to be NP

So that is essentially a small instruction about the types of problems we'll be seeing in this course. Next lecture we'll be formally going through running time.

3 Lecture 3 <2017-09-12 Tue>

3.1 Running Time Analysis

- We will be talking about running time of an algorithm.

3.1.1 Questions

Thinking back without knowledge of running time, what questions can we pose?

- How should we measure the running time of an algorithm?
- How can we compare the efficiency of two algorithms?
- What should we call an efficient algorithm?
 - Brute force isn't efficient for finding a matching.
 - Was our algorithm for stable matchings efficient?
 - We want to understand the concept of efficiency for an algorithm.

1. One option:

- Call an algorithm efficient if it performs "fast" on "real world" inputs.
 - What is a real world input?
 - * Without a good/rigorous definition, then this isn't a good option.
 - * Not precise, so this option doesn't work.

2. Option II:

- Take the set of all inputs of a certain size and take the average running time of our algorithm on them.

- Maybe the inputs we care about are quite sparse in the set of all inputs.
- Random inputs might be quite trivial
 - * May lead us to think we defined a good algorithm
 - * But in reality what we care about is harder

3. Example: Algorithm for prime numbers

- Input: integer n
- Output: Is n a prime number?
 - Alg 1:


```

for  $i = 2$  to  $n - 1$  do
  if  $n \pmod i = 0$  then return False
end if
end for
return true
          
```
 - Look at all the numbers between $1, \dots, N$
 - How many are divisible by 2, 3, 4, 5, 6, 7? $1 - \frac{1}{2} \times \frac{1}{3} \times \frac{1}{5} \times \frac{1}{7} > 99\%$
 - On average performs well
 - Worst case (prime numbers) does not perform well.
 - While this notion of average time complexity is useful, because the majority of inputs dominate the worst case ones, it is not a very good definition.
- Better to just care about the worst case

4. Worst case time analysis We measure the running time against the worst input of a given size

- Want to be independent of implementation:
 - We will count the number of "simple steps" (e.g. If " $a > b$ ", $a := b \times c$)

3.1.2 Efficiency

1. Def. We call an algorithm **efficient** if its running time is bounded by a polynomial $P(n)$ for every input of size (in number of bits) n
 - n efficient

- n^2 good
- $n \log n$ good
- 2^n bad
- $n \log n < n^2$

Remember that you need $\log n$ bits to store a number n .

- Objection: n^{100} is considered efficient while it is not practical!
- Answer: Usually the exponents are better. (Rarely see n^{100} if ever)

- Scales well
 - Many interesting algorithms have polynomial time algorithms

2. Alternative Def: Efficient is running time $< n^3$ seems a better def as it overrules cases like n^{100}

- Let's say you're combining 2 algorithms, say you're running a n^2 algorithm in a n^2 for-loop
 - Suddenly you're stuck with an n^4 algorithm
 - This doesn't allow us to easily stick algorithms in for-loops and the like
- This is not very robust.
 - The choice of data structure, pseudo-code, ... can change the running time a bit and so this definition is not "robust". Result depends on implementation.

3. Example: Input: An array $A[0 \dots n - 1]$

Goal: Are all elements in $A[]$ distinct?

```

for  $i = 0$  to  $n - 2$  do
  for  $j = i + 1$  to  $n - 1$  do
    if  $A[i] == A[j]$  then
      return "False"
    end if
  end for
end for
Return "True"

```

Step	Iterations
c_1 : setting i	$n - 1$
c_2 : setting j	$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{n(n-1)}{2}$
c_3 : comparing $A[i] == A[j]$	$\frac{n(n-1)}{2}$
c_4 : return False	1
c_5 : return True	1

Running time:

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

Efficient

This much accuracy is meaningless: Each one of these commands consist of some more primitive commands and that can depend on your compiler, ...

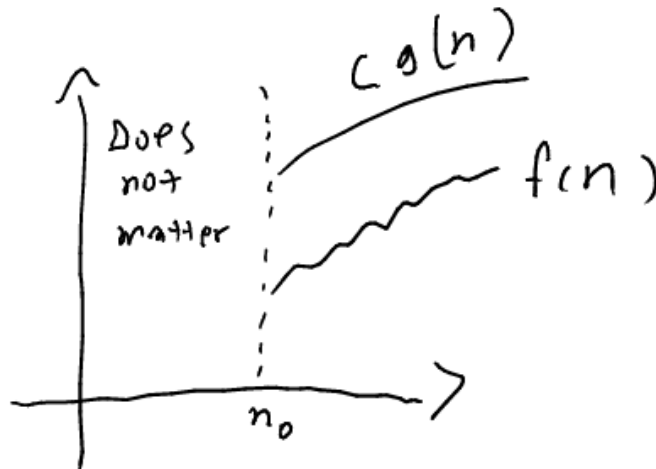
- What matters is that this is quadratic.

3.1.3 Big-O notation

Informally $O(g(n))$ is the set of all functions with smaller or same order of growth.

- You should think of it as a set, not a value.
- $n \in O(n^2)$
- $100n + 5 \in O(n^2)$
- $\frac{1}{2}n(n-1) \in O(n^2)$
- $n^3 \notin O(n^2)$

1. Def: $f(n) \in O(g(n))$ if $\exists n_0, c > 0$ such that $f(n) < cg(n) \forall n > n_0$



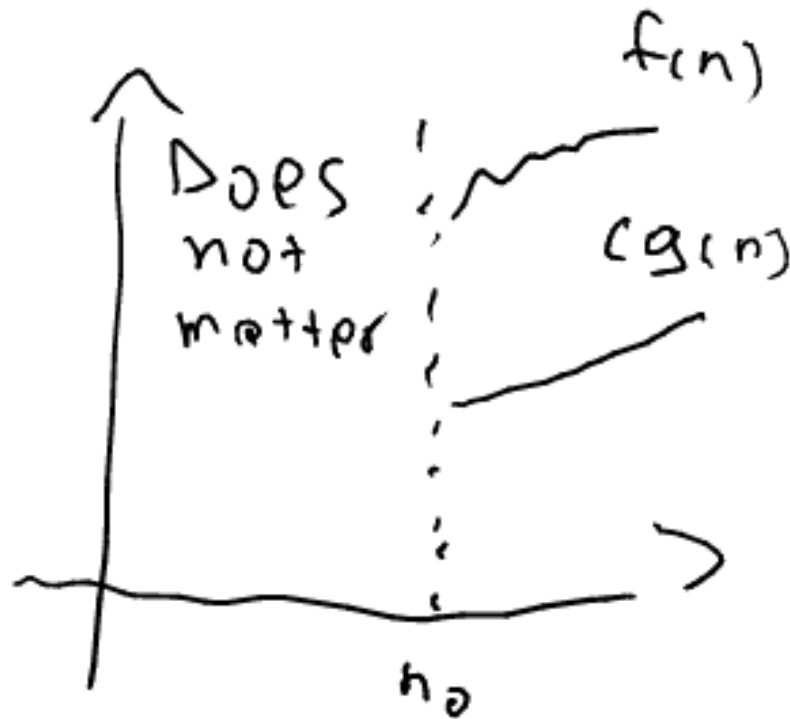
2. Ex: $100n + 5 \in O(n^2)$

(a) Proof $100n + 5 \stackrel{n \geq 5, n_0 = 5}{\leq} 100n + n \leq \underbrace{101}_{c=101} n$

3.1.4 Ω -notation:

Informally $f(n) \in \Omega(g(n))$ if $f(n)$ grows faster or the same as $g(n)$

1. Def: $f(n) \in \Omega(g(n))$ if $\exists n_0, c > 0$ such that $f(n) \geq cg(n) \forall n \geq n_0$
(Equivalently $g(n) \in O(f(n))$)



2. Ex: $\frac{n^2}{2} - 5n \in \Omega(n^2)$ $\frac{n^2}{2} - 5n \geq \frac{1}{4}n^2 \implies c = \frac{1}{4} \forall n \geq 20 = n_0$

4 Lecture 4 <2017-09-14 Thu>

4.1 Recall:

- Big-Oh
- Omega notation

4.2 Examples

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

4.3 Θ -notation:

$$f(n) \iff f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

- Grows at the same rate as $g(n)$

Alternatively:

$$\exists n_0, c_1, c_2 \forall n > n_0, \text{ s.t. } c_1 g(n) \leq f(n) \leq c_2 g(n)$$

4.3.1 Examples

- $2n^2 + 1 = \Theta(n^2)$
 1. $n^2 - 5n + 10 \leq n^2 \forall n \geq 2$
 2. $n^2 - 5n + 10 \geq \frac{n^2}{2} \forall n \geq 20$

4.4 Theorem

Let $f(n) = a_d n^d + a_{d-1} n^{d-1} + \dots + a_1 n + a_0, a_d > 0$.

Then $f(n) = \Theta(n^d)$

4.4.1 Proof

- $(f(n) = O(n^d))$
 - $f(n) = a_d n^d + \dots + a_1 n + a_0 \leq \underbrace{(a_d + |a_{d-1}| + \dots + |a_0|)}_c n^d, \forall n \geq 1$
 - E.g. $2n^2 - 5n + 10 \leq (2 + 5 + 10)n^2$

- $f(n) = \Omega(n^d)$
 - $a_d n^d + a_{d-1} n^{d-1} + \dots + a_1 n + a_0 \geq C n^d$
 - $c = \frac{a_d}{2}$, since a_d is controlling the growth rate of the left hand side.
 - $\frac{a_d}{2} n^d \geq -(a_{d-1} n^{d-1} + a_{d-2} n^{d-2} + \dots + a_0)$
 - $\frac{a_d}{2} n^d \geq (|a_{d-1}| + \dots + |a_0|) n^{d-1}, \forall n \geq \frac{2(|a_{d-1}| + \dots + |a_0|)}{a_d}$ (by rearranging and isolating n)
 - On the other hand:
 - * $(|a_{d-1}| + \dots + |a_0|) n^{d-1} \geq -(a_{d-1} n^{d-1} + \dots + a_0)$
 - Note that $|a_r| n^{d-1} \geq -a_r n^r, r \leq d-1$

4.5 Little o and Little omega

- Show strict upper and lower bounds, rather than equalities

$$f(n) = o(g(n))$$

- $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$
- Little oh implies big-Oh, but not the other way around

$$f(n) = \omega(g(n))$$

- $\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

$$n^{1/100} \text{ vs } \log_2(n)^5$$

$$\text{Claim: } \log_2(n)^5 = o(n^{1/100})$$

$$\text{Proof: } \lim_{n \rightarrow \infty} \frac{\log_2(n)^5}{n^{1/100}} = \lim_{n \rightarrow \infty} \frac{5 \log(n)^4 \frac{\ln(2)}{n}}{\frac{1}{100} n^{\frac{-99}{100}}} = \dots = 0 \text{ (have to do}$$

L'Hopital's 4 more times)

The lesson is that anything in log grows much slower than any polynomial.

4.6 Theorem

- $\forall r > 1, d > 0$
- $n^d = o(r^n)$ (i.e. polynomials grow much slower than exponential functions)

$$\underbrace{n^{10000}}_{\text{Better}} \text{ vs } 1.0001^n$$

4.7 Stable Marriage

Data structures we may use:

- Array $A[0 \dots n - 1]$
 - Operation times:
 - * Access $A[i] : O(1)$
 - * Insert a new entry somewhere in the middle: $O(n)$, need to shift.
 - * Delete: $O(n)$
 - * Finding an element: $O(n)$ not sorted
· $O(\log(n))$ sorted
- Linked List
 - Operation times:
 - * Access i – th entry: $O(n)$
 - * Insert-delete: $O(1)$
 - * Finding: $O(n)$

while \exists a free man m **do**

Let w be the highest-ranked woman m has not proposed to yet.

if w is free **then**

(m, w) engaged

else if w is currently engaged to m' **then**

if w prefers m to m' **then**

(m', w) engaged

m becomes free

end if

end if

end while

- Input: Two (men and women) $n \times n$ arrays (rankings)
- Reading input $\Theta(n^2)$ so at best we can hope $\Theta(n^2)$ for the alg.
- The main while loop can repeat $O(n^2)$ times \implies To have total $\Theta(n^2)$ time every iteration must be done in $O(1)$.
- How to implement?

- When do we know if a man is free?
 - * Can have an array of booleans of free men, but then you need for loop to check if there's a free man, which will be $O(n)$
 - * Solution 1: Can have a linked list of free men.
 - Delete someone from the list when they get engaged.
 - Deleting and adding is $O(1)$ (add to front)
 - * Solution 2: Using an array
 - Have a pointer to first free man and another to last free man
 - If first man gets engaged, move pointer to the right
 - If someone becomes free, then add to end and change pointer
 - Since we never have more than n people free, can use mod n

5 Lecture 5 <2017-09-21 Thu>

5.1 Graphs

5.1.1 Undirected Graphs

- Notation $G = (V, E)$
 - V = nodes (or vertices)
 - E = edges (or arcs) between pairs of nodes.
 - Captures pairwise relationship between object
 - Graph size parameters: $n = |V|, m = |E|$

5.1.2 Example applications

Graph	Node	Edge
Communication	telephone, computer	fiber optic cable
Circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	street intersection, airport	highway, airway route
internet	class C network	connection
game	board position	legal move
social relationship	person, actor	friendship, movie cast
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

5.1.3 Ways of implementing in a program

1. Adjacency matrix n -by- n matrix with $A_{uv} = 1$ if (u, v) is an edge.
 - Two representations of each edge.
 - Space proportional to n^2
 - Checking if (u, v) is an edge takes $\Theta(1)$ time
 - Identifying all edges takes $\Theta(n^2)$ time
 - It's exactly symmetric
2. Adjacency list Node-indexed array of lists
 - Two representations of each edge
 - Space is $\Theta(m + n)$
 - Checking if (u, v) is an edge takes $O(\text{degree}(u))$ time
 - Identifying all edges takes $\Theta(m + n)$ time

5.1.4 Paths and connectivity

- Def. A **path** in an undirected graph $G = (V, E)$ is a sequence of nodes v_1, v_2, \dots, v_k with the property that each consecutive pair v_{i-1}, v_i is joined by an edge in E .
- Def. A path is **simple** if all nodes are distinct.
- Def. An undirected graph is **connected** if for every pair of nodes u and v , there is a path between u and v

5.1.5 Cycles

- Def. A **cycle** is a path v_1, v_2, \dots, v_k in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.

5.1.6 Trees

- Def. An undirected graph is a **tree** if it is connected and does not contain a cycle
1. Theorem Let G be an undirected graph on n nodes. Any two of the following statements imply the third:
 - G is connected
 - G does not contain a cycle
 - G has $n - 1$ edges
 2. Rooted trees
 - Given a tree T , choose a root node r and orient each edge away from r .
 - Importance. Models hierarchical structure

5.1.7 Connectivity

- s-t connectivity problem. Given two nodes s and t , is there a path between s and t ?
- s-t shortest path problem. Given two nodes s and t , what is the length of a shortest path between s and t ?
- Applications.
 - Friendster
 - Maze traversal
 - Kevin Bacon number
 - Fewest hops in a communication network

5.1.8 Breadth-first search

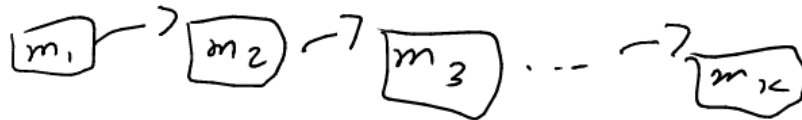
1. BFS intuition Explore outward from s in all possible directions, adding nodes one "layer" at a time. At most n layers.
2. BFS algorithm
 - $L_0 = \{s\}$
 - $L_1 =$ all neighbors of L_0
 - $L_2 =$ all nodes that do not belong to L_0 or L_1 , and that have an edge to a node in L_1
 - $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i
3. Theorem For each i , L_i consists of all nodes at distance exactly i from s . There is a path from s to t iff t appears in some layer.
4. Property Let T be a BFS tree of $G = (V, E)$, and let (x, y) be an edge of G . Then, the levels of x and y differ by at most 1.
5. Analysis
 - (a) Theorem The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.
 - (b) Proof
 - Easy to prove $O(n^2)$ running time:
 - At most n lists $L[i]$
 - Each node occurs on at most one list; for loop runs $\leq n$ times
 - When we consider node u , there are $\leq n$ incident edges (u, v) , and we spend $O(1)$ processing each edge
 - Actually runs in $O(m + n)$ time:
 - When we consider node u , there are $\text{degree}(u)$ incident edges (u, v)
 - total time processing edges is $\sum_{u \in V} \text{degree}(u) = 2m$

6 Lecture 6 <2017-09-26 Tue>

6.1 Stable Marriage

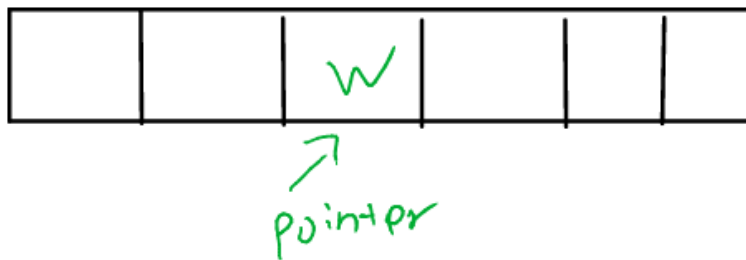
Continuation of Lecture 4: Stable Marriage algorithm analysis.

- Good data structure to tell if someone is free or not?
 - Can have a linked list of all the free men, remove them when they're no longer free.

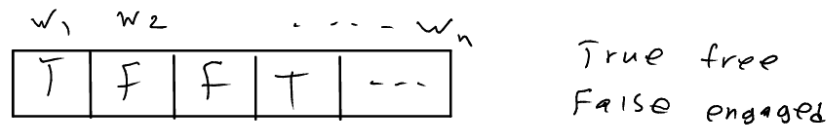


Initially all men are here. Finding a free man: $O(1)$

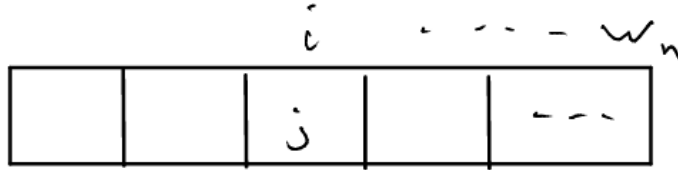
- Keep an ordered list of women sorted according to m 's preference. Keep a pointer to the first person he has not proposed to yet. (can store as a linked list, array or stack) Move pointer along to the next after proposing.



Now we need to know if the woman is free. Make a boolean array of women with true or false.



- Array telling whom w is engaged to (j^{th} entry contains who w_j is engaged to)



- We keep a matrix $w[i, j] =$ the rank of m_j in the eye of w_i
- Example: $w_2 : m_4 > m_3 > m_5 > m_2 \dots$, $w[2, 5] = 3$ (don't need to do linear time)
- If w_i prefers m_j to $m_k \iff w[i, j] < w[i, k]$
 - Do some "preprocessing" in the beginning to make it easier during the algorithm

With all these data structures, our algorithm can run in $O(n^2)$

6.2 Priority Queue

Say we're running a clinic and new patients come. A nurse assesses them and gives them a priority so that we know who we should see next.

Dynamic Scenario

- Get elements with different priorities in an "online" matter (sometimes you get new data, not all given to you in the beginning)
- Once in awhile we can serve the element with the highest priority (and remove from the set)

We have a set S .

- Initially $S = \emptyset$
- At every step either
 - A new number is added to S .
 - or the smallest number is removed from S .

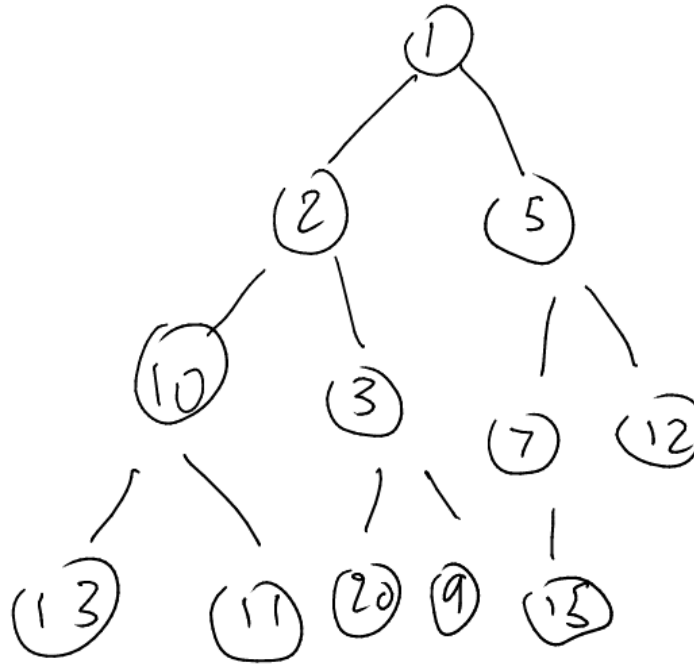
Some ideas:

- An unsorted list:
 - Inserting a new element $O(1)$
 - Removing the minimum: $O(n)$ (n elements in the list, have to find smallest)
 - Too costly, not good.
- Sorted list:
 - Inserting a new element $O(n)$
 - * With an array, need to shift all elements.
 - * Linked list (no binary search)
 - Removing the smallest $O(1)$.

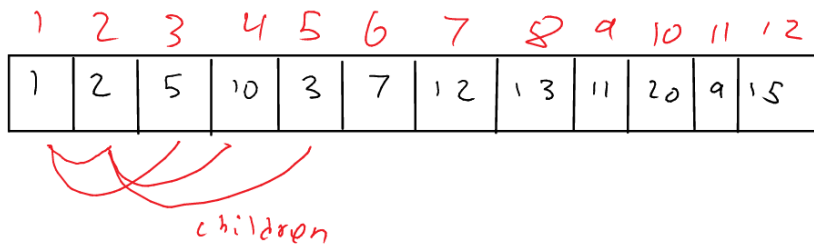
6.2.1 Heap Data Structure

A balanced binary tree

- All levels are full except the last level which is filled **from left to right**
- Every node is \geq its parent
- Ex:



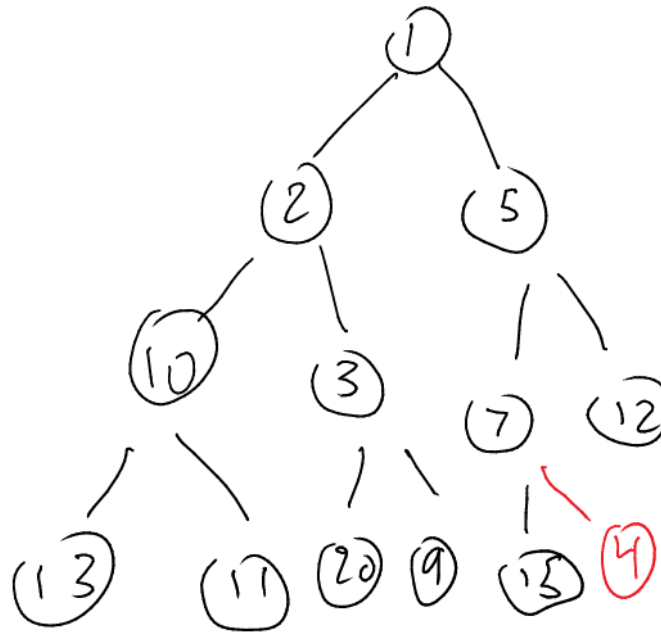
Can be implemented with an array. Fill left to right.



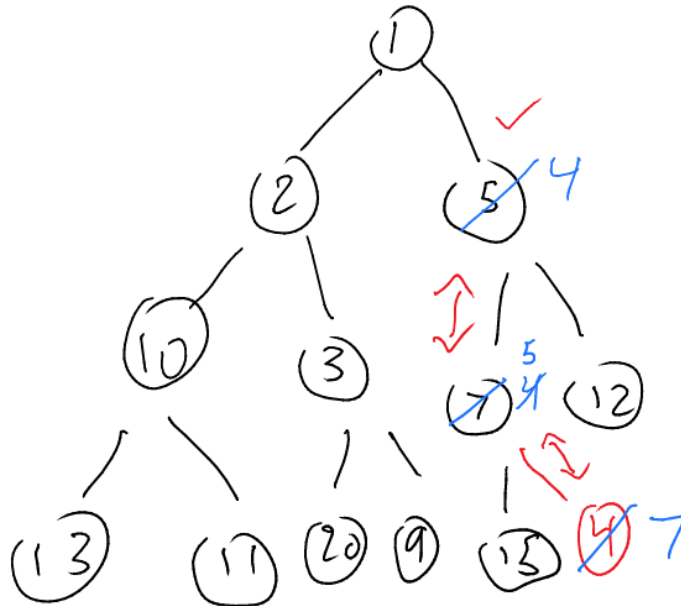
Where are the children of entry i ? $2i, 2i + 1$ (convenient)

What do we do when a new number arrives? Say insert(4)

- Naturally we want to put it in the next available place " n^{th} " if n is the updated # of nodes



But 4 is smaller than its parent. How to fix? Swap with parent.



We will call this operation Heapify-Up.

Heapify-Up(H, i) // i is index

if $i > 1$ **then**

 let $j = \text{parent}(i) = \lfloor \frac{i}{2} \rfloor$

if $H[i] < H[j]$ **then**

 swap($H[i], H[j]$)

 Heapify-Up(H, j)

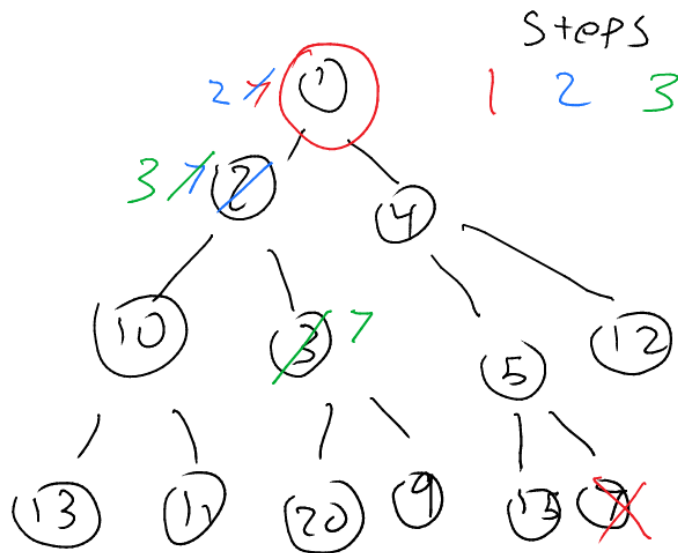
end if

end if

Running time of Heapify-Up: $O(\log n) = O(\text{Height of the tree})$

How do we remove the minimum?

- Insert last element at head and then swap with smallest child until the tree is balanced



```

Heapify-down( $H, i$ )
 $n = \text{length}(H)$ 
if  $2i > n$  then // Elements  $> n/2$  have no children
    Terminate
else if  $2i + 1 \leq n$  then
     $left = 2i, right = 2i + 1$ 
    if  $H[left] < H[right]$  then
         $j = left$ 
    else
         $j = right$ 
    end if
else // ( $n = 2i$ )
     $j = left = 2i$ 
end if
if  $H[j] < H[i]$  then
    swap( $H[j], H[i]$ )
    Heapify-down( $H, j$ )
end if

```

Q: How can we use this data structure to sort a list of n numbers?

Answer: Insert the elements one by one and then extract the minimums one by one.

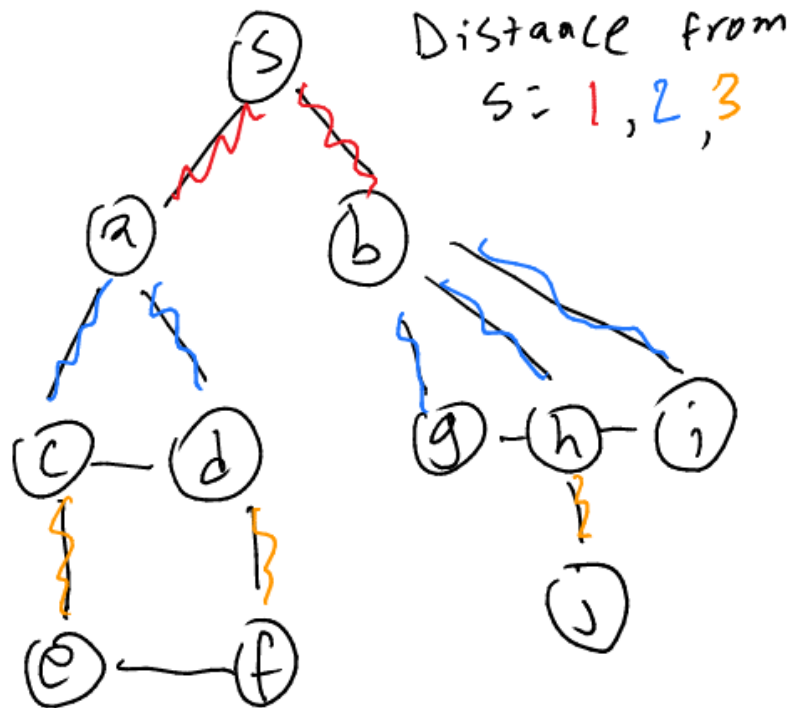
- Running time? $2nO(\log n)$

– $O(n \log n)$

7 Lecture 7 <2017-09-28 Thu>

7.1 Graph Exploration Algorithms

7.1.1 Breadth-First-Search (BFS)



Also tells you length of shortest path from s to any vertex.

- We explore according to the distance from s .
- How to implement this?

BFS(G)

```
for every vertex  $v$  in  $G$  do
  if  $v$  is unexplored then
    Mark  $v$  as explored
    BFS.vertex( $v$ )
```

```

        connected-comp++
    end if
end for

```

BFS-vertex(v)

Make a list of all the unexplored neighbors of v.

Mark every vertex in this list as explored

for every u in this list **do**

 BFS-Vertex(u)

end for

Recursive way above does not work?

A good way to implement this is to keep the newly discovered vertices in a queue (FIFO, first in first out).

BFS-Vertex(v)

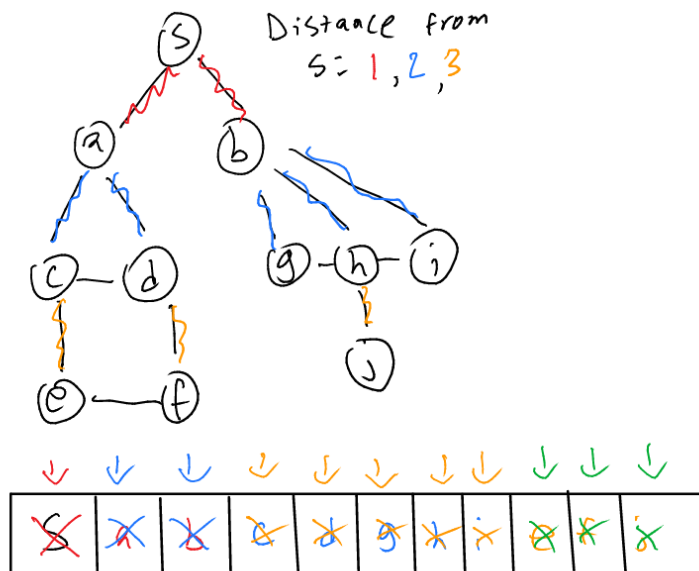
Add v to the queue

while queue is not empty **do**

 Pick the first vertex u in the queue.

 Mark all unexplored neighbors of u as explored and add them to the queue

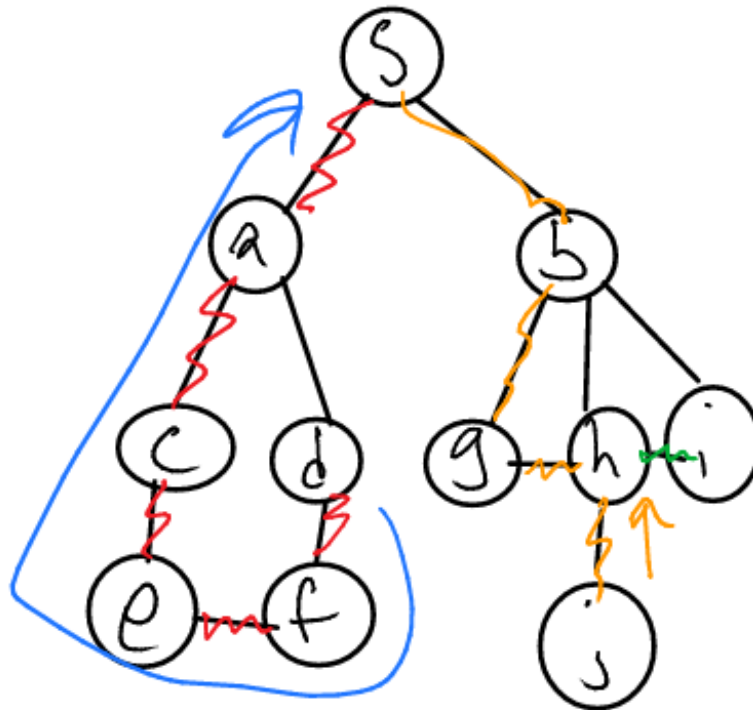
end while



7.1.2 Depth-First-Search (DFS)

We go in a path discovering new vertices until we reach a dead-end, and then we step back ...

```
DFS(u)
for every edge (u,v) do
  if v is unexplored then
    mark v as explored
    DFS(v)
  end if
end for
```



Non-recursive DFS: Every time we discover a new vertex we put it at the top of a stack (FILO, first in last out).

7.2 Data Structure for Graphs

What data structure to use for graphs?

- Adjacency Matrix

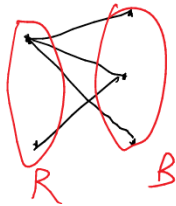
$$A[u, v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{if } (u, v) \notin E \end{cases}$$

- Pros: very easy to see if u is connected to v
- Cons: IF the graph has few edges it is wasteful. $O(n^2)$ bits of memory.

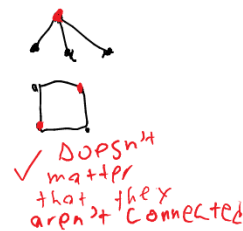
- For every vertex v we keep a list of all edges (u, v) incident to v
- Pros: easy to find the neighbors
 - Doesn't take much memory if the graph is sparse
- Cons: Takes $O(n)$ to see if u is adjacent to v .

7.3 Bipartites

An undirected graph is called bipartite if we can partition the vertices into two parts R and B such that all the edges are between R and B



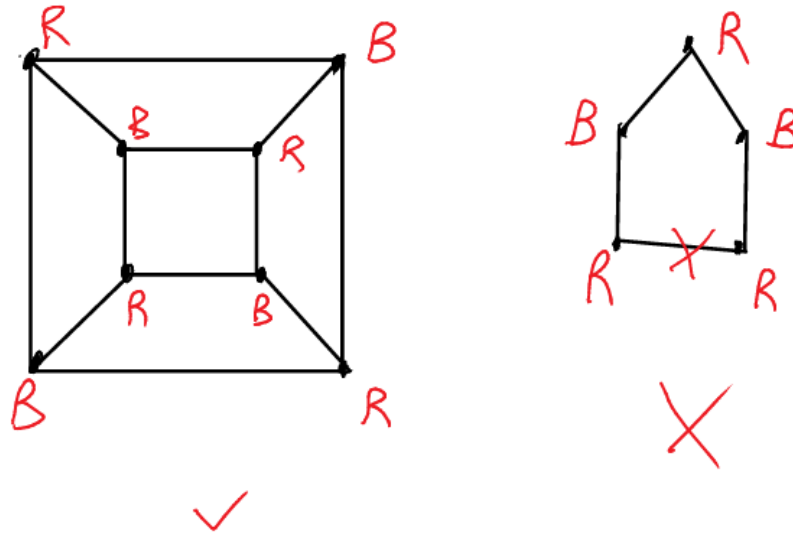
Ex:



7.3.1 Testing for bipartites

How can we test to see if G is bipartite? Label one vertex in R then:

- Look at neighbors to see if they're supposed to be R or B



```
DFS_Bipartite(G)
for every vertex u in G do
  if u is not explored then
    color[u] = "R"
    mark u as explored
    DFS(u)
  end if
end for
if not declared "non-bipartite" yet then
  declare "bipartite"
end if
```

```
for each edge (u,v) do
  if v is not explored then
    Mark v as explored
    color v differently from color[u]
    DFS(v)
```

```

    else if color[u]=color[v] then
        declare "non-bipartite"
    end if
end for

```

This is called proper two coloring of a graph.

7.4 Directed Graphs

Every edge has an orientation.



Sometimes
we allow



7.4.1 Data Structure:

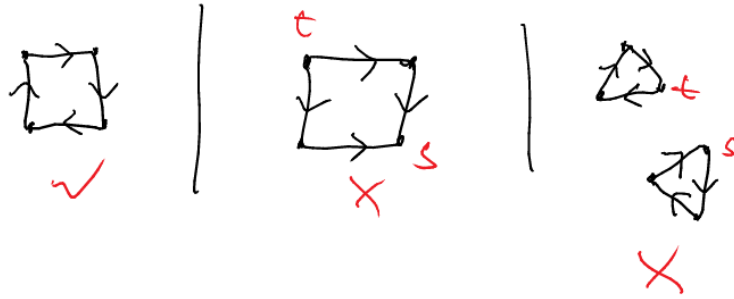
For every vertex keep two lists: the edge going out, the edges coming into that vertex Given two vertices, s and t , is there a path from s to t ?

- say $s=a$ $t=d$
- yes in the graph
- But there is no path from d to a .

We can use the "directed" version of DFS to solve this problem: We run DFS(s) if t is explored then such a path exists otherwise it doesn't.

Def: A directed graph is called strongly connected if for every u and v there is a path from u to v . (can go from anywhere to anywhere)

Ex:



Q: Given G , how can we tell if it is strongly connected?

- Pick a vertex s
- Run DFS(s) in G
- If there is any unexplored vertex then "not strongly connected"
- Run DFS(s) in G^{rev} (same as G , but with directions reversed)
- If \exists any unexplored vertex then "not strongly connected"
- Otherwise declare " G is strongly connected"

8 Lecture 8 <2017-10-03 Tue>

8.1 Directed Graphs

- Each edge has a direction (seen last class)
- Not symmetric, edge from u to v means no edge from v to u .

8.1.1 Graph search

- Directed reachability
 - Find all nodes reachable from a given node
- Directed s-t shortest path problem

- Given two nodes, what is length of shortest path between them
- BFS extends naturally to directed graphs
- Web crawler
 - Start from web page s . Find all web pages linked from s

8.1.2 Strong Connectivity

- Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u .
 - A graph is **strongly connected** if every pair of nodes is mutually reachable.
1. Lemma Let s be any node. G is strongly connected iff every node is reachable from s , and s is reachable from every node.
 - Proof: \implies Follows from definition
 - \impliedby Path from u to v : concatenate u - s path with s - v path
 - Path from v to u : concatenate v - s path with s - u path
 2. Algorithm
 - (a) Theorem Can determine if G is strongly connected in $O(m + n)$ time.

Proof:

 - Pick any node s
 - Run BFS from s in G
 - Run BFS from s in G^{rev}
 - Return true iff all nodes reached in both BFS executions
 - Correctness follows immediately from previous lemma
 - Has running time of BFS $O(m + n)$

8.2 Directed Acyclic Graphs

- A **DAG** is a directed graph that contains no directed cycles
 - Good for modeling dependencies, like a course's prerequisites
- Ex. Precedence constraints: edge (v_i, v_j) means v_i must precede v_j .

– Precedence constraints imply no cycle

- A **topological order** of a directed graph $G = (V, E)$ is an ordering of its nodes as v_1, v_2, \dots, v_n so that for every edge (v_i, v_j) we have $i < j$

8.2.1 Lemma

If G has a topological order, the G is a DAG.

Proof (by contradiction)

- Suppose G has a topological order v_1, \dots, v_n and that G also has a directed cycle C .
- Let v_i be the lowest-indexed node in C and let v_j be the node just before v_i : thus (v_j, v_i) is an edge.
- By our choice of i , we have $i < j$
- On the other hand, since (v_j, v_i) is an edge and v_1, v_2, \dots, v_n is a topological order, we must have a contradiction. \nexists

8.2.2 Lemma

If G is a DAG, then G has a node with no incoming edges.

Proof (by contradiction)

- Suppose G is a DAG and every node has at least one incoming edge.
- Pick any node v , begin following edges backward from v . Since v has at least one incoming edge (u, v) we can walk backward to u .
- Since u has at least one incoming edge (x, u) we can walk backward to x
- Repeat until we visit a node, say w , twice.
- Let C denote the sequence of nodes encountered between successive visits to w . C is a cycle. \nexists

8.2.3 Lemma

If G is a DAG, then G has a topological ordering.

Proof (by induction on n)

- Base case: true if $n = 1$

- Given DAG on $n > 1$ nodes, find a node v with no incoming edges
- $G \setminus \{v\}$ is a DAG, since deleting v cannot create cycles
- By inductive hypothesis, $G \setminus \{v\}$ has a topological ordering.
- Place v first in topological ordering: then append nodes of $G \setminus \{v\}$ in topological order. This is valid since v has no incoming edges.

1. Algorithm To compute a topological ordering of G

- Find a node v with no incoming edges and order it first
- Delete v from G
- Recursively compute a topological ordering of $G \setminus \{v\}$ and append this order after v
- Running time: $O(n)$ for each call, calling exactly n times. So algorithm runs in $O(n^2)$. Lots of running time if the graph is sparse, not many edges. If we reimplement this more carefully, we can get $O(m + n)$, with m being the number of edges. Note that making an algorithm run faster usually requires more space.'

2. Theorem Algorithm finds a topological order in $O(m + n)$ time

Proof:

- Maintain the following information:
 - $count[w]$ = remaining number of incoming edges
 - S = set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete v
 - Remove v from S
 - Decrement $count[w]$ for all edges from v to w and add w to S if $count[w]$ hits 0
 - This is $O(1)$ per edge

9 Lecture 9 <2017-10-05 Thu>

9.1 Greedy Algorithm

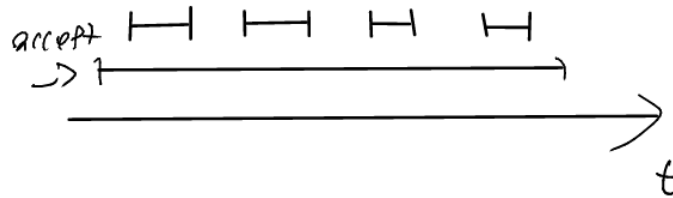
- In every step, it tries to be myopic and optimize its current goal/step
- Doesn't care about the future

9.1.1 Interval scheduling

- Have a class room and a microscope
 - Every request has a starting time and finishing time $\{1, \dots, n\}, (s_i, f_i)$
 - Def. i and j are compatible ($i + j$) when $f_i \leq s_j$ or $f_j \leq s_i$
 - Subset of requests is compatible if every point of requests are compatible.
 - Maximum sized compatible subset is the optimal subset
-

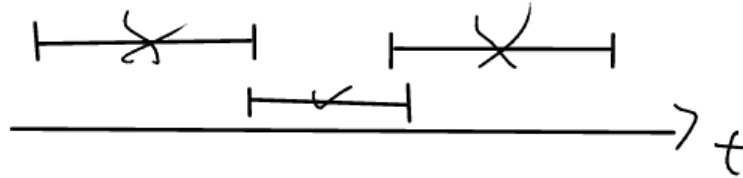
1. Pick $s(i)$ with earliest request

- Might not give an optimal solution if the request that begins the earliest goes until the end, not allowing any of the other requests to be fulfilled.



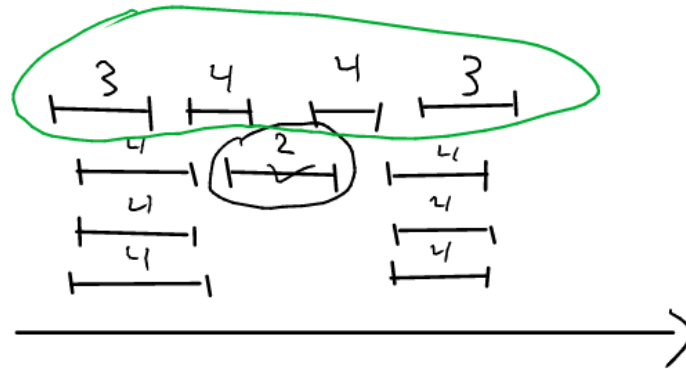
2. $f(i) - s(i)$ is the smallest

- Can be problematic if the smallest is in between 2



3. For each request compute the # of requests it overlaps with. Pick the one with the smallest number.

- Still problematic



4. Accept (greedy rule) requests i for which $f(i)$ is the smallest.

- Sort requests so that $f(i_1) \leq f(i_2) \leq \dots \leq f(i_n)$
- This one works

```

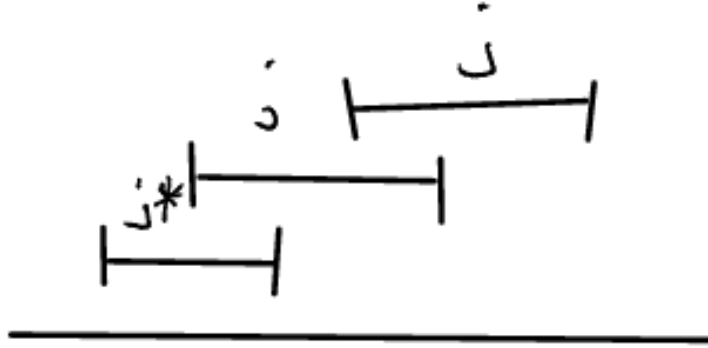
A = ∅
for j = 1 to n do
  if j is compatible with A then
    A ← A ∪ {j}
  end if
end for
return A

```

9.1.2 TODO clean up this section

Running time of method 4:

- Sort : $O(n \log n)$
- $f(j) \geq f(j^*) \forall i \in A, f(i) \leq f(j^*) \leftarrow O(n)$



9.1.3 Theorem

This greedy algorithm returns the optimal subset.

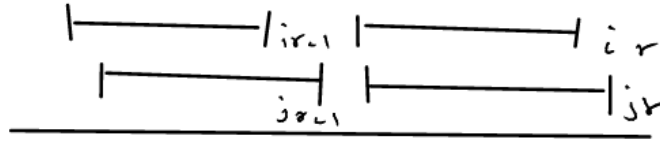
$$\underbrace{|A|}_{\text{optimal}} = |O| - \text{optimal subset}$$

- "stays ahead"
- $|A| = k, |O| = n$, assume $k < m$
- O is ordered by their starting and finishing time for every $j \in O$, $f(i_1 \in A) \leq f(j)$

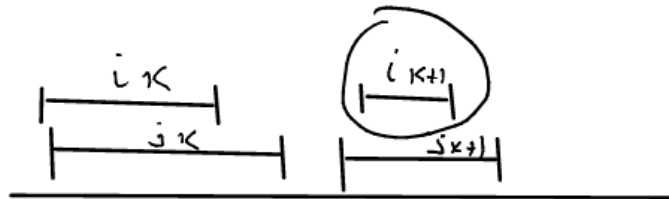
1. Lemma. For all $r \leq k$, $f(i_r) \leq f(j_r)$

(a) Proof

- $r = 1$ $f(\underbrace{i_1}_A) \leq f(\underbrace{j_1}_O)$ Works
- This greedy algorithm returns the optimal subset $r - 1$ i.e. $f(i_{r-1}) \leq f(j_{r-1})$.
- But this contradicts $f(i_r) \geq f(j_r) \implies f(i_r) \leq f(j_r)$



- $A : i_1 \dots i_k$
- $O : j_1 \dots j_k j_{k+1} \dots$
- Apply the lemma with $r = k$ so $f(i_k) \leq f(j_k)$



This contradicts $k < m$! Thus $m = k$

- Sort O by starting time, it's also sorted by finishing time

9.1.4 Satisfying requests

Given requests, how many resources do we need to satisfy all of them?

- Def. depth is the maximum number of requests that have a common point in the time line.
1. Claim The # of resources is at least d . I_1, \dots, I_d - requests with depth d .