

# Contents

<b>1</b>	<b>Lecture 1</b>	<b>&lt;2017-09-05 Tue&gt;</b>	<b>5</b>
1.1	Algorithm		5
1.1.1	Examples of algorithms		6
1.2	Teacher's website		6
1.3	Stable matching		6
1.3.1	Ex: $n = 4$		6
1.4	Stable Matching Algorithm		7
1.4.1	For our example:		8
1.4.2	Modified example		8
1.4.3	Why isn't this infinite?		9
<b>2</b>	<b>Lecture 2</b>	<b>&lt;2017-09-07 Thu&gt;</b>	<b>9</b>
2.1	Announcements		9
2.2	Recall:		9
2.2.1	Ex: $n = 4$		10
2.2.2	Last lecture we proved:		10
2.3	Stable Matching Algorithm		11
2.3.1	Does this algorithm produce a stable marriage?		11
2.3.2	Is this algorithm better for men or women?		11
2.4	Notes on problems		12
2.5	Some example problems		13
2.5.1	Interval scheduling		13
2.5.2	Weighted Interval Scheduling		14
<b>3</b>	<b>Lecture 3</b>	<b>&lt;2017-09-12 Tue&gt;</b>	<b>16</b>
3.1	Running Time Analysis		16
3.1.1	Questions		16
3.1.2	Efficiency		17
3.1.3	Big-O notation		19
3.1.4	$\Omega$ -notation:		20
<b>4</b>	<b>Lecture 4</b>	<b>&lt;2017-09-14 Thu&gt;</b>	<b>21</b>
4.1	Recall:		21
4.2	Examples		21
4.3	$\Theta$ -notation:		21
4.3.1	Examples		21
4.4	Theorem		21
4.4.1	Proof		22

4.5	Little o and Little omega . . . . .	22
4.6	Theorem . . . . .	23
4.7	Stable Marriage . . . . .	23
<b>5</b>	<b>Lecture 5 &lt;2017-09-21 Thu&gt;</b>	<b>24</b>
5.1	Graphs . . . . .	24
5.1.1	Undirected Graphs . . . . .	24
5.1.2	Example applications . . . . .	25
5.1.3	Ways of implementing in a program . . . . .	25
5.1.4	Paths and connectivity . . . . .	25
5.1.5	Cycles . . . . .	26
5.1.6	Trees . . . . .	26
5.1.7	Connectivity . . . . .	26
5.1.8	Breadth-first search . . . . .	27
<b>6</b>	<b>Lecture 6 &lt;2017-09-26 Tue&gt;</b>	<b>27</b>
6.1	Stable Marriage . . . . .	27
6.2	Priority Queue . . . . .	29
6.2.1	Heap Data Structure . . . . .	30
<b>7</b>	<b>Lecture 7 &lt;2017-09-28 Thu&gt;</b>	<b>35</b>
7.1	Graph Exploration Algorithms . . . . .	35
7.1.1	Breadth-First-Search (BFS) . . . . .	35
7.1.2	Depth-First-Search (DFS) . . . . .	37
7.2	Data Structure for Graphs . . . . .	37
7.3	Bipartites . . . . .	38
7.3.1	Testing for bipartites . . . . .	39
7.4	Directed Graphs . . . . .	40
7.4.1	Data Structure: . . . . .	40
<b>8</b>	<b>Lecture 8 &lt;2017-10-03 Tue&gt;</b>	<b>41</b>
8.1	Directed Graphs . . . . .	41
8.1.1	Graph search . . . . .	41
8.1.2	Strong Connectivity . . . . .	42
8.2	Directed Acyclic Graphs . . . . .	42
8.2.1	Lemma . . . . .	43
8.2.2	Lemma . . . . .	43
8.2.3	Lemma . . . . .	43

<b>9</b>	<b>Lecture 9</b>	<b>&lt;2017-10-05 Thu&gt;</b>	<b>44</b>
9.1	Greedy Algorithm		44
9.1.1	Interval scheduling		45
9.1.2	<b>TODO</b> clean up this section		46
9.1.3	Theorem		47
9.1.4	Satisfying requests		48
<b>10</b>	<b>Lecture 10</b>	<b>&lt;2017-10-10 Tue&gt;</b>	<b>48</b>
10.1	Recall		48
10.2	Partition scheduling		49
10.2.1	Greedy Template		49
10.3	Minimizing Lateness		52
10.3.1	Greedy Template		53
10.3.2	Optimal Alg		53
10.4	Midterm		54
<b>11</b>	<b>Lecture 11</b>	<b>&lt;2017-10-17 Tue&gt;</b>	<b>54</b>
11.1	Optimal Caching		55
<b>12</b>	<b>Lecture 12</b>	<b>&lt;2017-10-19 Thu&gt;</b>	<b>58</b>
12.1	Shortest Path in Graphs		58
12.2	Dijkstra's Algorithm		58
12.2.1	Correctness		60
<b>13</b>	<b>Lecture 13</b>	<b>&lt;2017-10-24 Tue&gt;</b>	<b>63</b>
13.1	The Minimum Spanning Tree Problems (MST)		63
13.1.1	Three Greedy Algorithms:		64
13.1.2	Correctness		67
<b>14</b>	<b>Lecture 14</b>	<b>&lt;2017-10-26 Thu&gt;</b>	<b>71</b>
14.1	Recall		71
14.2	The Kruskal Alg (Proof of Correctness)		71
14.2.1	Thm:		72
14.2.2	Pf:		72
14.3	Application		73
14.3.1	Algorithm		75
14.3.2	Remark		76
14.3.3	Thm		76
14.3.4	Proof		76

<b>15 Lecture 15</b>	<b>&lt;2017-10-31 Tue&gt;</b>	<b>76</b>
15.1	Data Compression/Huffman Codes . . . . .	76
15.1.1	Prefix property . . . . .	78
15.1.2	Prefix Codes as binary trees . . . . .	78
15.1.3	The best prefix codes . . . . .	79
15.1.4	Huffman Coding with an example . . . . .	82
<b>16 Lecture 16</b>	<b>&lt;2017-11-02 Thu&gt;</b>	<b>83</b>
16.1	Recall: Huffman Coding . . . . .	83
16.1.1	Observation 1: . . . . .	84
16.1.2	Observation 2: . . . . .	84
16.2	Huffman Coding . . . . .	84
16.2.1	Thm . . . . .	84
16.2.2	Proof . . . . .	85
16.3	Divide and Conquer . . . . .	87
16.3.1	Example: Merge Sort . . . . .	87
<b>17 Lecture 17</b>	<b>&lt;2017-11-07 Tue&gt;</b>	<b>89</b>
17.1	Recall: Merge sort . . . . .	89
17.2	Counting Inversions . . . . .	89
17.2.1	Example . . . . .	89
17.2.2	Brute-Force . . . . .	89
17.2.3	Divide and Conquer . . . . .	90
17.2.4	Applications . . . . .	92
<b>18 Lecture 18</b>	<b>&lt;2017-11-09 Thu&gt;</b>	<b>95</b>
18.1	Integer Addition . . . . .	95
18.2	Integer Multiplication . . . . .	95
18.2.1	Divide and Conquer approach . . . . .	96
18.3	Matrix Multiplication . . . . .	98
18.3.1	Divide and Conquer . . . . .	99
<b>19 Lecture 19</b>	<b>&lt;2017-11-14 Tue&gt;</b>	<b>100</b>
19.1	Dynamic Programming . . . . .	100
19.1.1	Weighted interval scheduling . . . . .	101
19.1.2	Knapsack Problem . . . . .	105
<b>20 Lecture 20</b>	<b>&lt;2017-11-16 Thu&gt;</b>	<b>107</b>
20.1	Recall . . . . .	107
20.2	Another Example . . . . .	108

20.3 RNA secondary structures . . . . .	108
<b>21 Lecture 21 &lt;2017-11-21 Tue&gt;</b>	<b>112</b>
21.1 Longest Common Subsequence . . . . .	112
21.1.1 Example . . . . .	112
21.1.2 Example 2 . . . . .	113
21.2 Edit Distance . . . . .	114
21.2.1 Example . . . . .	115
21.3 Shortest Path with Negative Weights . . . . .	115
<b>22 Lecture 22 &lt;2017-11-23 Thu&gt;</b>	<b>118</b>
22.1 Shortest Path with Negative Weights (Continuation) . . . . .	118
22.1.1 Running Time . . . . .	120
22.2 Finding Negative Cycles . . . . .	121
22.2.1 Slow fix . . . . .	122
22.2.2 Faster algorithm . . . . .	122
<b>23 Lecture 23 &lt;2017-11-28 Tue&gt;</b>	<b>123</b>
23.1 Max Flow Problem . . . . .	123
23.1.1 Ford-Fulkerson Algorithm . . . . .	129

## 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  $\underline{m}$  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:

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

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

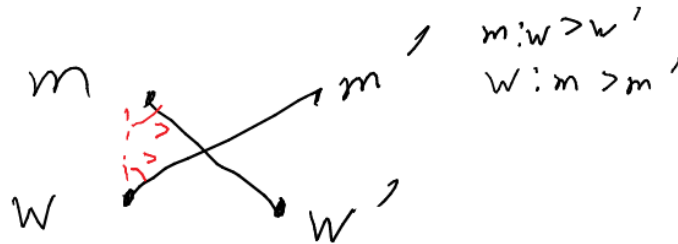
- Throughout the algorithm every man's partner gets worse and worse

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  $w$ .

1. 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

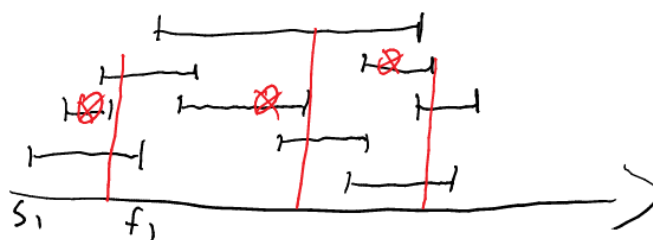
### 2.5.1 Interval scheduling

- 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.
- $s_1, \dots, s_n$
- $f_1, \dots, f_n$

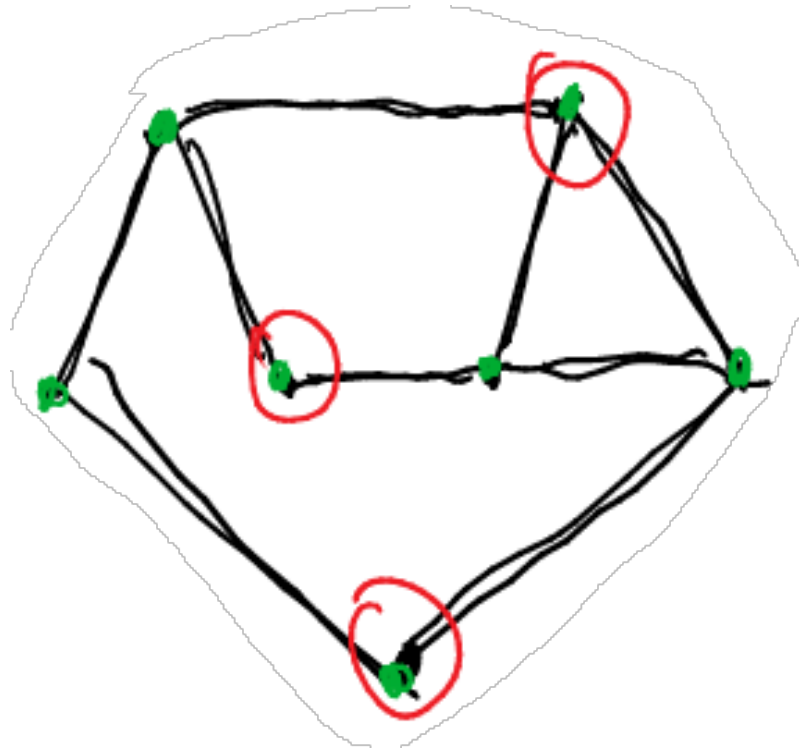
Want higher value, rather than most matchings Why is this harder to solve than the previous problem? Because the previous one is a special case of the first.

- 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  $A[t] = \max$  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, ...

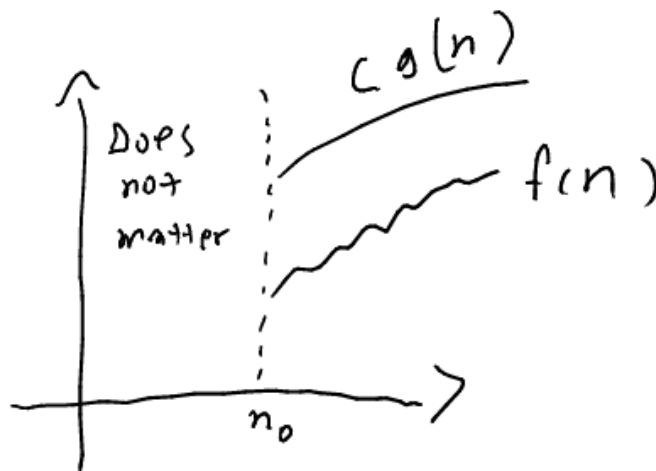
- What matters is that this is quadratic.

### 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  $f(n) < cg(n) \forall n > n_0$



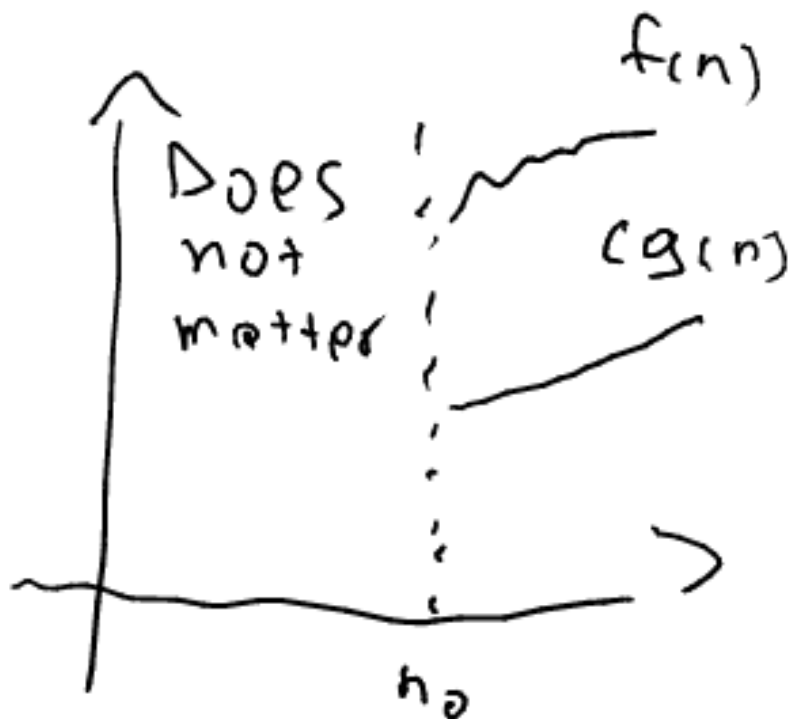
2. Ex:  $100n + 5 \in O(n^2)$

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

### 3.1.4 $\Omega$ -notation:

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

- 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 $\Theta$ -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)

---

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

- 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
- 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

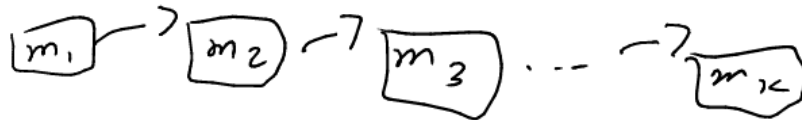
1. BFS intuition Explore outward from  $s$  in all possible directions, adding nodes one "layer" at a time. At most  $n$  layers.
2. BFS algorithm
  - $L_0 = \{s\}$
  - $L_1 =$  all neighbors of  $L_0$
  - $L_2 =$  all nodes that do not belong to  $L_0$  or  $L_1$ , and that have an edge to a node in  $L_1$
  - $L_{i+1} =$  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

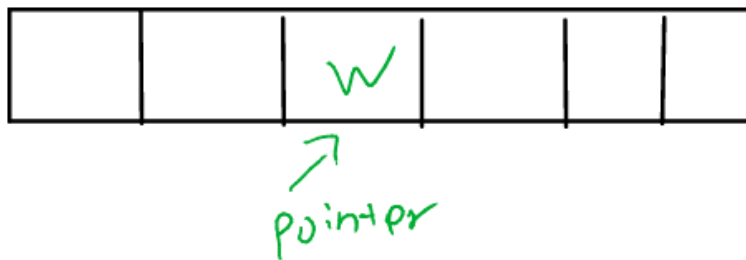
Continuation of Lecture 4: Stable Marriage algorithm analysis.

- 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.

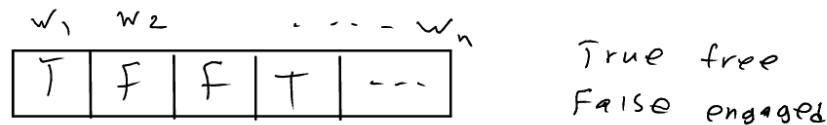


Initially all men are here. Finding a free man:  $O(1)$

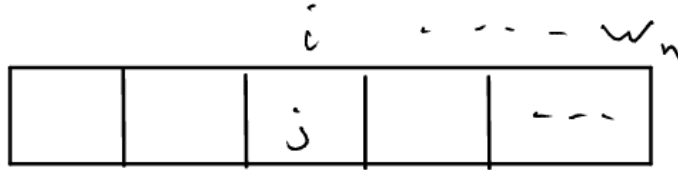
- Keep an ordered list of women sorted according to  $m$ 's preference. Keep a pointer to the first person he has not proposed to yet. (can store as a linked list, array or stack) Move pointer along to the next after proposing.



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  $S$ .

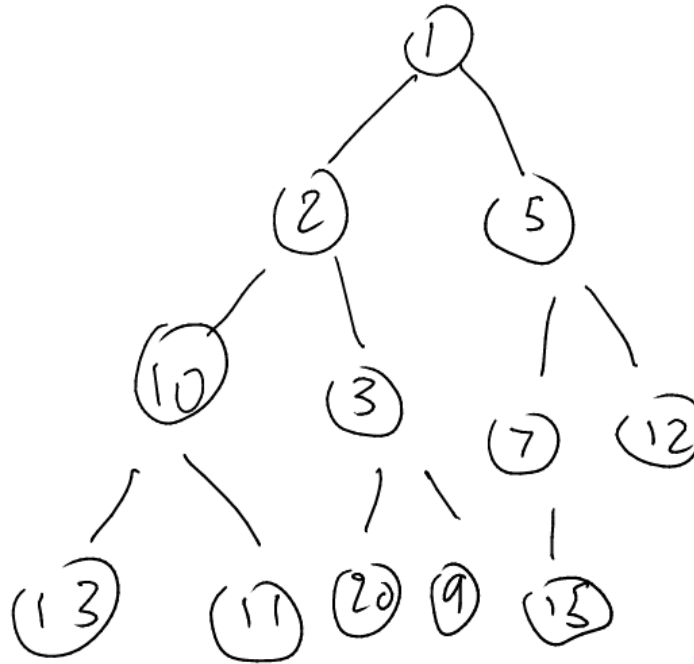
Some ideas:

- 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  $O(1)$ .

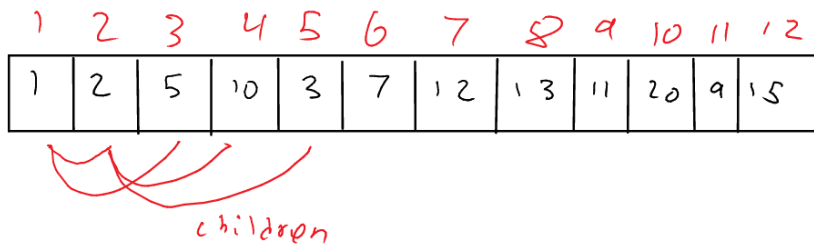
### 6.2.1 Heap Data Structure

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.

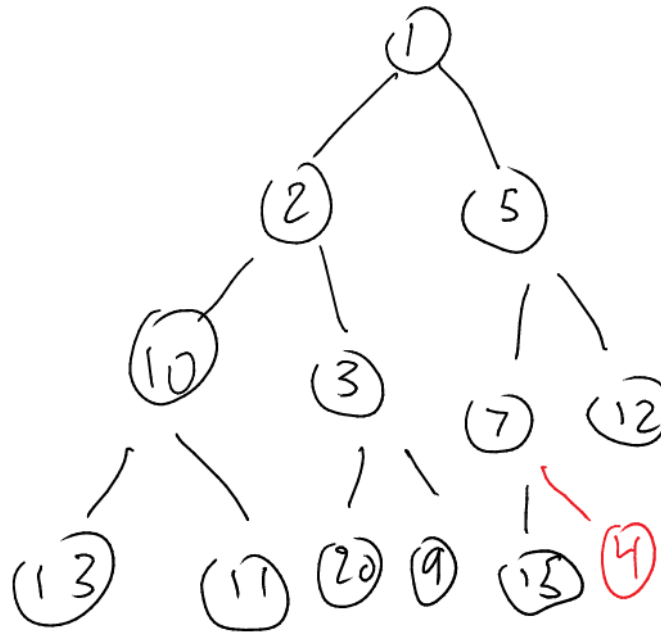


Where are the children of entry  $i$ ?  $2i, 2i + 1$  (convenient)

---

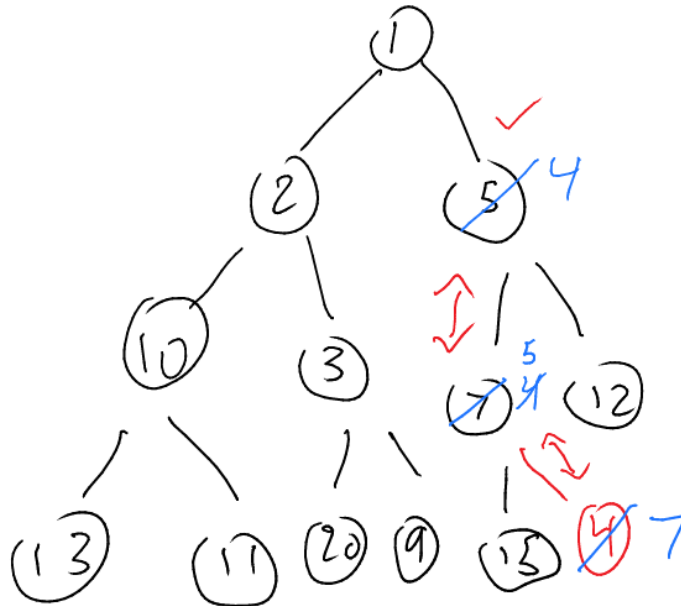
What do we do when a new number arrives? Say insert(4)

- 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**

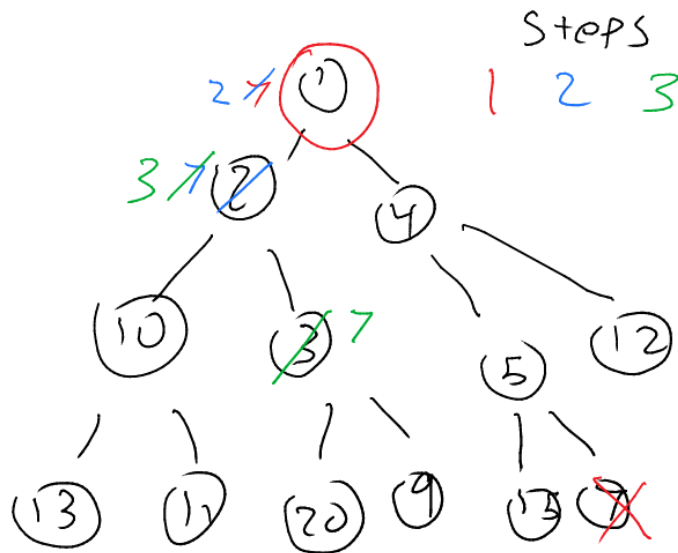
**end if**

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.

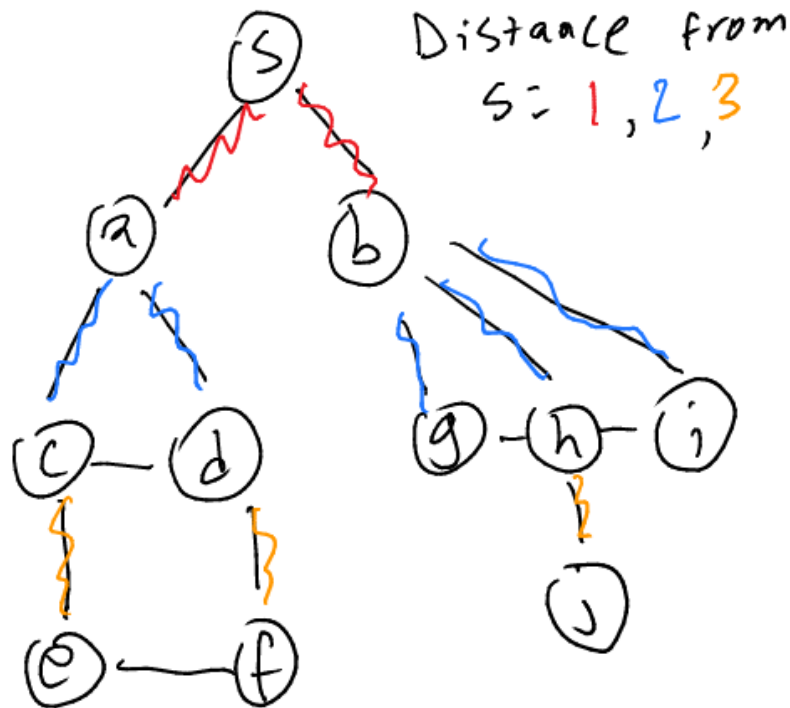
- Running time?  $2nO(\log n)$

–  $O(n \log n)$

## 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)

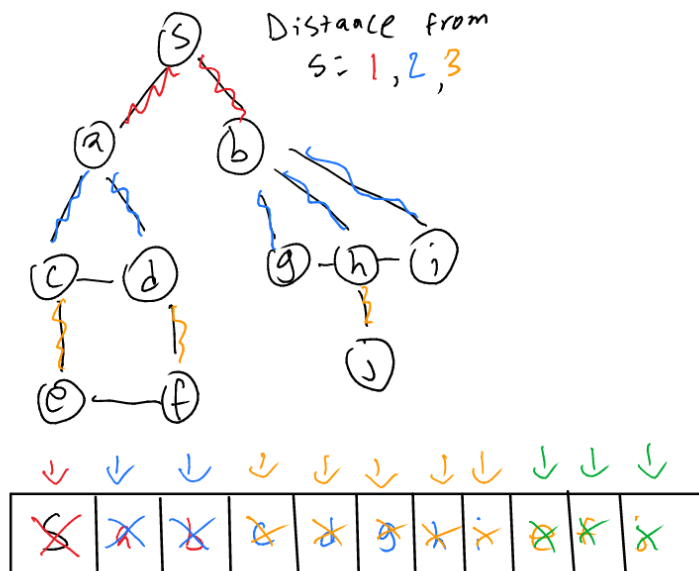
Add v to the queue

**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**

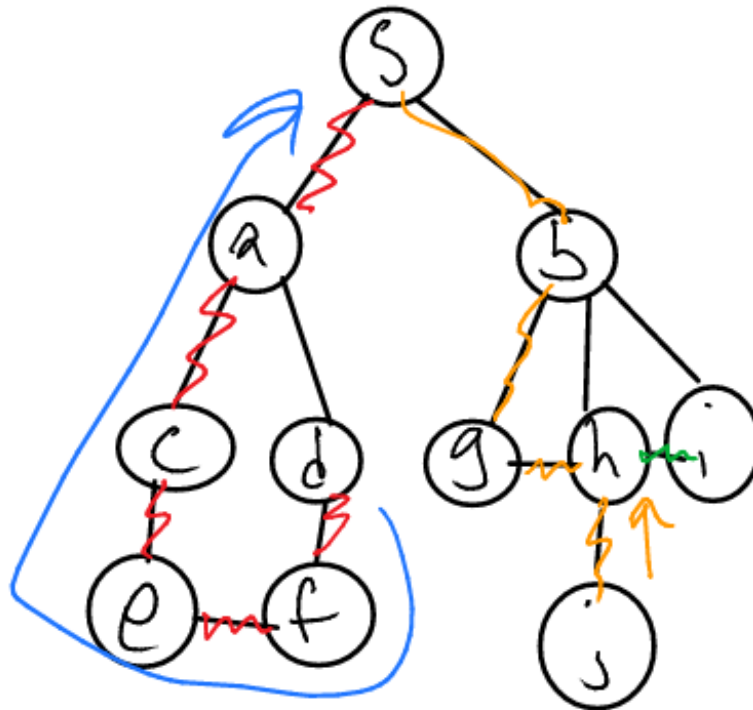


### 7.1.2 Depth-First-Search (DFS)

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

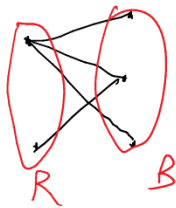
$$A[u, v] = \begin{cases} 1 & \text{if } (u, v) \in E \\ 0 & \text{if } (u, v) \notin E \end{cases}$$

- 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.

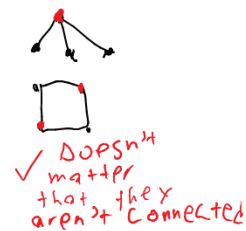
- For every vertex  $v$  we keep a list of all edges  $(u, v)$  incident to  $v$
- Pros: easy to find the neighbors
  - Doesn't take much memory if the graph is sparse
- Cons: Takes  $O(n)$  to see if  $u$  is adjacent to  $v$ .

### 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:

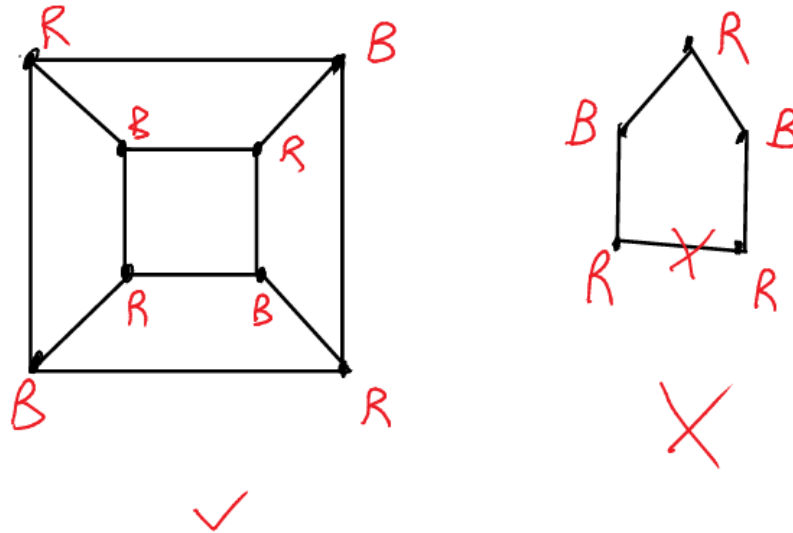


✓ DOESN'T  
matter  
that they  
aren't connected

### 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  $t$ ?

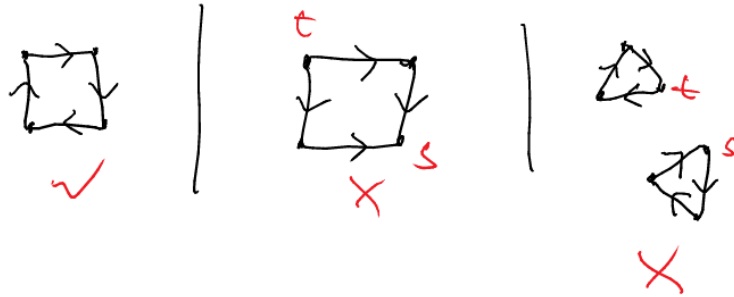
- say  $s=a$   $t=d$
- yes in the graph
- But there is no path from  $d$  to  $a$ .

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

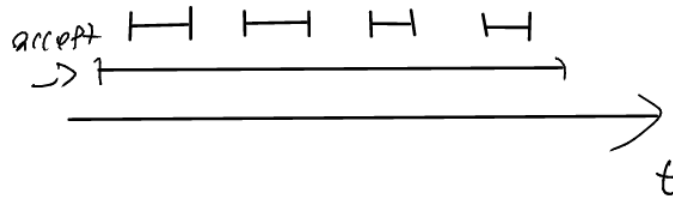
- In every step, it tries to be myopic and optimize its current goal/step
- Doesn't care about the future

### 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

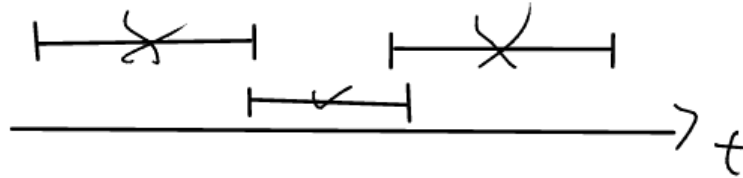
1. Pick  $s(i)$  with earliest request

- 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.



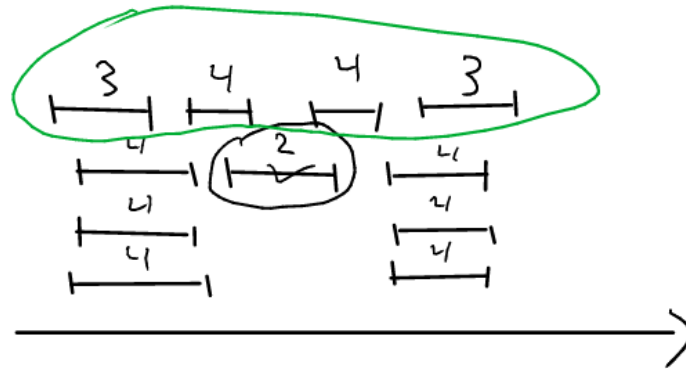
2.  $f(i) - s(i)$  is the smallest

- 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

```

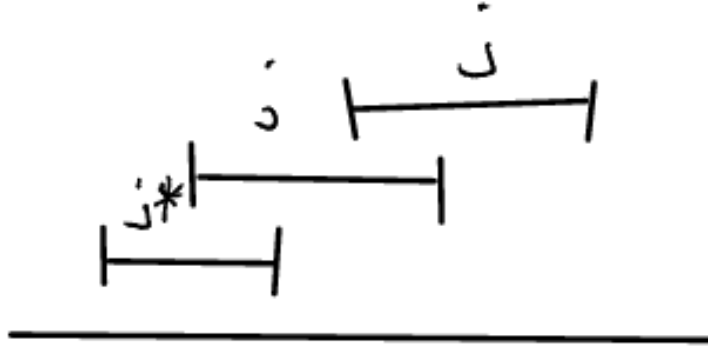
A = ∅
for j = 1 to n do
  if j is compatible with A then
    A ← A ∪ {j}
  end if
end for
return A

```

### 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.

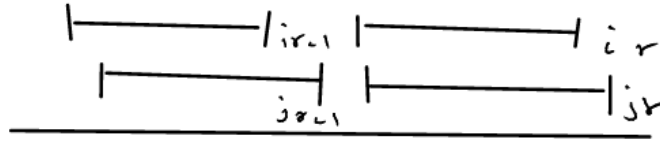
$$\underbrace{|A|}_{\text{optimal}} = |O| - \text{optimal subset}$$

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

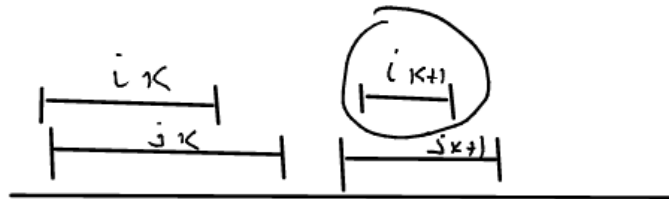
1. Lemma. For all  $r \leq k$ ,  $f(i_r) \leq f(j_r)$

(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  $d$ .

## 10 Lecture 10 <2017-10-10 Tue>

### 10.1 Recall

Interval scheduling

- Input: Lectures  $s_j, f_n$  (start and finish)  $j = 1, \dots, n$



- Goal: Find the largest non-overlapping set.
- Alg: Always pick the job with earliest finish time.

## 10.2 Partition scheduling

Now we really want to accommodate all these jobs. How many rooms/resources do we need?

- Input: Same as above
- Goal: Smallest number of rooms that can accommodate all the lectures.

### 10.2.1 Greedy Template

Consider lectures in some **natural order**. Assign each lecture to an available room (how?). If none is available open a new room.

---

Earliest-Start-Time-first

- $(n, s_1, \dots, s_n, f_1, \dots, f_n)$

Sort the lectures so that  $s_1 \leq s_2 \leq \dots \leq s_n$

$d = 0$  (number of rooms)

```

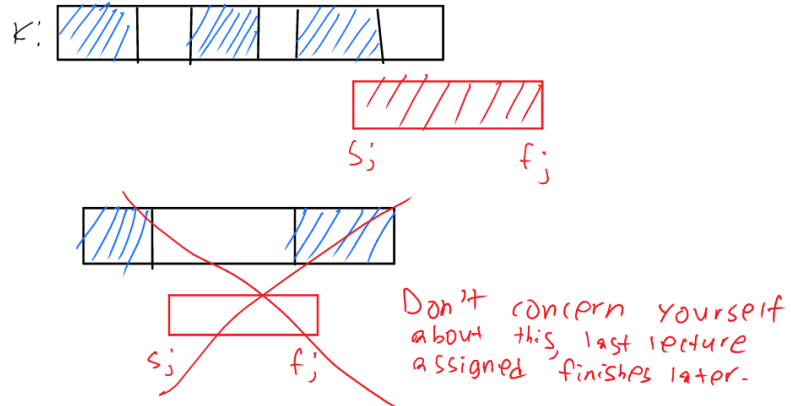
for  $j = 1, \dots, n$  do
  if lecture  $j$  is compatible with room  $k$  then
    Assign  $j$  to room  $k$ 
  else Assign  $j$  to room  $d + 1$ 
    set  $d = d + 1$ 
  end if
end for

```

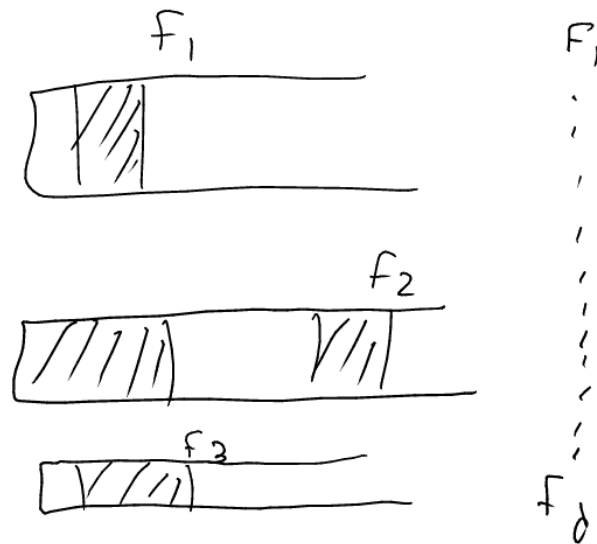
Now that we have the algorithm, we need to analyze its correctness and its running time.

#### 1. Running Time

- Sorting:  $O(n \log n)$
- For loop runs  $n$  times. Each time we check if a lecture is compatible with a room, so we must do this fast.
- To see if lecture  $j$  is compatible with a room  $k$  we only need to compare  $s_j$  with the finishing time of the last lecture assigned to that room. (Since we know that none of the lectures in the room start after time  $s_j$ )



So for each room we keep a variable which tells us when the room becomes available.



We need to see if  $s_j > \min(F_1, \dots, F_d)$ . (How to do this quickly? Priority queue.)

**if** Yes **then** The room with minimum  $F_k$  is available  
**else** Open a new room  
**end if**

This is the priority queue problem: Always want to know the minimum (we can add or delete numbers from the list). Using a heap this can be implemented so that all insertions and deletions can be done in  $O(\log n)$

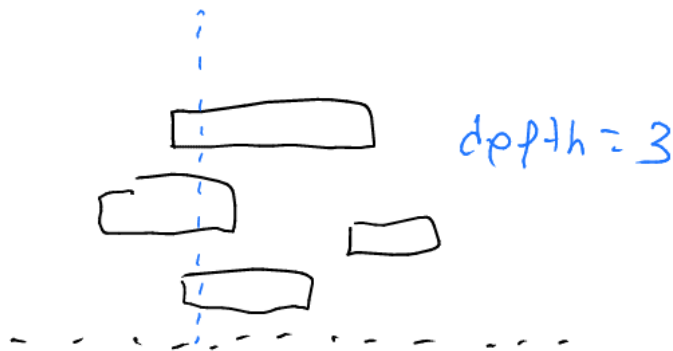
---

Running Time:

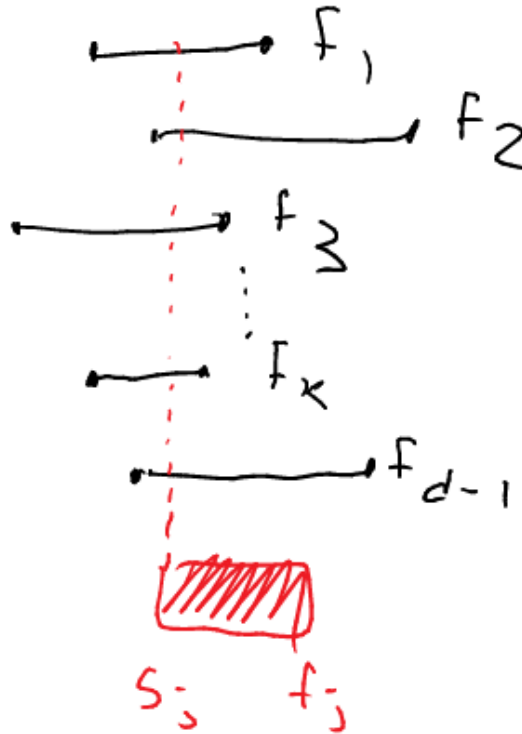
$$\bullet \underbrace{O(n \log n)}_{\text{Sort}} + \underbrace{n}_{\text{For loop}} \times \underbrace{O(\log n)}_{\text{Priority queue}} = O(n \log n)$$

2. Correctness Why does this alg output the best solution?

- Depth: Max number of intervals that contain any point on the timeline



- Obviously: Optimal  $\geq$  depth
- Claim: When the algorithm opens a new room  $d$  then depth  $\geq d$
- Proof: Room  $d$  is opened since lecture  $j$  was incompatible with  $d - 1$  other rooms.



In this case every room  $1, \dots, d-1$  has a lecture that ends after  $s_j$  (and starts before  $s_j$ , due to the way the algorithm works). These together with  $j$  show  $\text{depth} \geq d$

---

What do we know?  $\text{depth} \geq \text{Output of algorithm d (just showed)} \geq \text{optimal} \geq \text{depth (earlier)} \implies \text{depth} = \text{optimal} = \text{output of alg}$

### 10.3 Minimizing Lateness

- Input:  $n$  tasks.
  - Processing times:  $t_1, \dots, t_n$
  - Deadline:  $d_1, \dots, d_n$
- Goal: We have a single processor. Ideally we want to schedule all tasks so that they all finish before their deadlines.

Each task will be scheduled for some time  $s_j = f_j - t_k$  to  $f_j$  (to finish at  $f_j$  we need to start at  $f_j - t_k$ ).

- Lateness =  $\max_j \{f_j - d_j\}$  (Time we finish - deadline for job)
- Goal: Minimize the lateness

### 10.3.1 Greedy Template

Sort the jobs according to some order and assign them to the processor according to this order.

Shortest job first?

- This doesn't work.

Process Time	Deadline
1	100
10	10

Optimal is  $f = 10, f = 11$ . But this alg gives us  $f = 1, f = 11$ .

Smallest slack ( $d_j - t_j$ ) first. But this might give us huge lateness.

t	d
1	2
10	10

Optimal:  $f = 1, f = 11$ , lateness = 1

Alg:  $f = 10, f = 11$ , lateness = 9

### 10.3.2 Optimal Alg

Sort by the deadline:  $d_1 \leq d_2 \leq \dots d_n$

Set  $f \leftarrow 0$

**for**  $i = 1 \dots n$  **do**

    Assign job  $j$  to  $[f, f + t_j]$

$f = f + t_j$

**end for**

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

---

Why is this optimal? Suppose the optimal is not sorted according to deadlines. Then we will have  $i$  and  $j$ :

What will switching these two jobs do? It can only improve the lateness.

## 10.4 Midterm

Next class is the midterm, will be split into 2 rooms.

- Topics: Everything until today
- Format: Similar to assignments, 3-4 questions like on the assignments
- No crib sheets

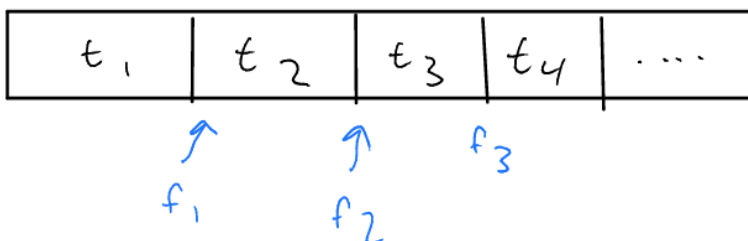
## 11 Lecture 11 <2017-10-17 Tue>

- Recall: Minimize Lateness
  - Input: Jobs  $(t_i, d_i), i = 1, \dots, n$ , where  $t_i$  is the process time and  $d_i$  is the deadline.
  - In which order should we proceed them in order to minimize
  - Lateness =  $\max_i f_i - d_i$ 
    - \* Where  $f_i$  is the finishing time of job  $i$

---

Greedy alg: Process these jobs in increasing order of their deadlines

- (Earliest deadline first)
- Sort  $d_1 \leq d_2 \leq \dots \leq d_n$

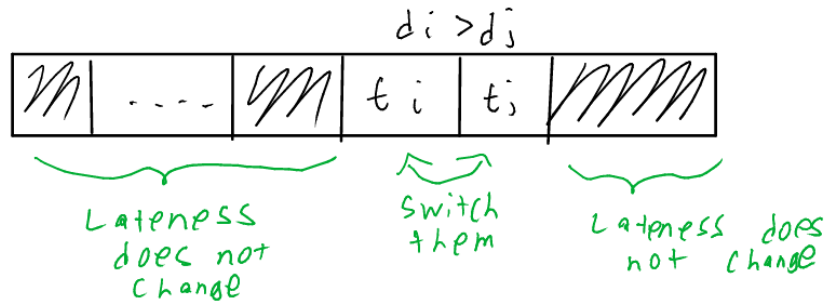


---

How do we show that this is optimal?

- Most greedy algorithm proofs are similar, start with the optimal solution and then show that the algorithm keeps with it

Consider an optimal solution. If different from the output of the algorithm (not sorted), then we can find 2 jobs that are not sorted in order of deadline



- How does the lateness of the jobs we switch change?
- $f_i = T + t_i \rightarrow f'_i = T + t_i + t_j$
- $f_j = T + t_i + t_j \rightarrow f'_j = T + t_i$
- $f'_j$  has better lateness than before (smaller lateness), but  $f'_i$  lateness might increase
  - new lateness =  $T + t_i + t_j - d_i$
- Why won't this increase lateness? This is smaller than the original lateness of the  $j^{th}$  job =  $T + t_i + t_j - d_j$  (since  $d_i$  is larger than  $d_j$ )

A different way of writing this proof: Among all optimal solutions pick the one that agrees with the greedy algorithm for the longest period.

## 11.1 Optimal Caching

(Very complicated)

- Cache with some capacity to store items
- If someone requests an item that we have in the cache, we can show it to them
- If they request something we don't have in the cache, then we have to remove it from the cache
- Sequence of  $m$  requests:  $d_1, d_2, \dots, d_m$

- Cache hit: The item is in the cache.
- Cache miss: Item not in cache when requested. (Must bring the item to the cache and evict some existing item) This is a costly operation.
- We want to make the cache optimal given the schedule beforehand
- We assume that we start with a full cache.

Example:  $k = 2$ , initial cache  $|a|b|$

- Requests:

	cache	
1 ✓ a	ab	
2 ✓ b	ab	
3 miss × c	cb	$a \leftarrow c$
4 ✓ b	cb	
5 ✓ c	cb	
6 miss × a	ab	$c \leftarrow a$
7 ✓ a	ab	
8 ✓ b	ab	

We managed to do this one with 2 cache misses. How do we optimize this?

Greedy Alg: Evict the item that is needed farthest in the future. In the above example, in step 3, we see that a is needed in step 6 but b is needed in step 4, so we evict a.

	1 ✓ a	abc	
	2 ✓ b	abc	
	3 ✓ c	abc	
	4 × d	abd	$c \leftarrow d$
	5 ✓ a	abd	
Example: cache abc	6 ✓ d	a <b>b</b> d	Can do anything for steps
	7 × e	aed	$b \leftarrow e$
	8 ✓ a	aed	
	9 ✓ d	aed	
	10 × b	bed	$a \leftarrow b$
	11 × c	ced	$b \leftarrow c$

10 & 11, but are steps 4 and 7 unique? No, we can do  $b \leftarrow d$  at step 4 instead. So the greedy algorithm is one solution, but it isn't the only solution, making it harder to prove.

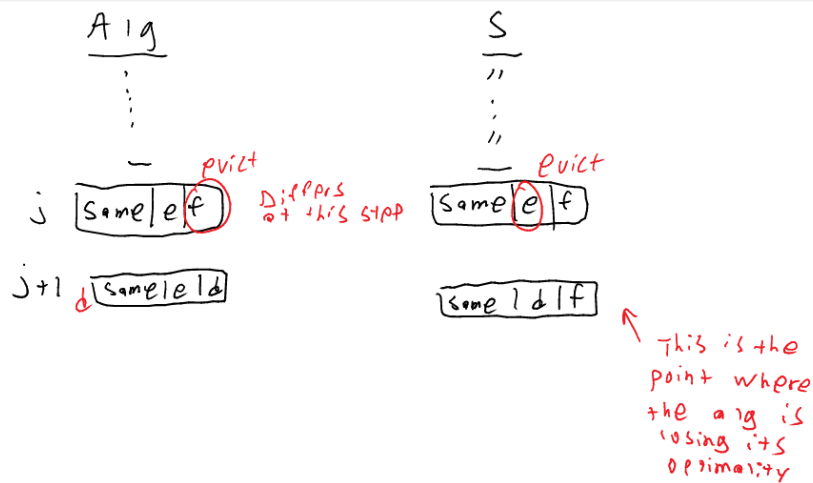


---

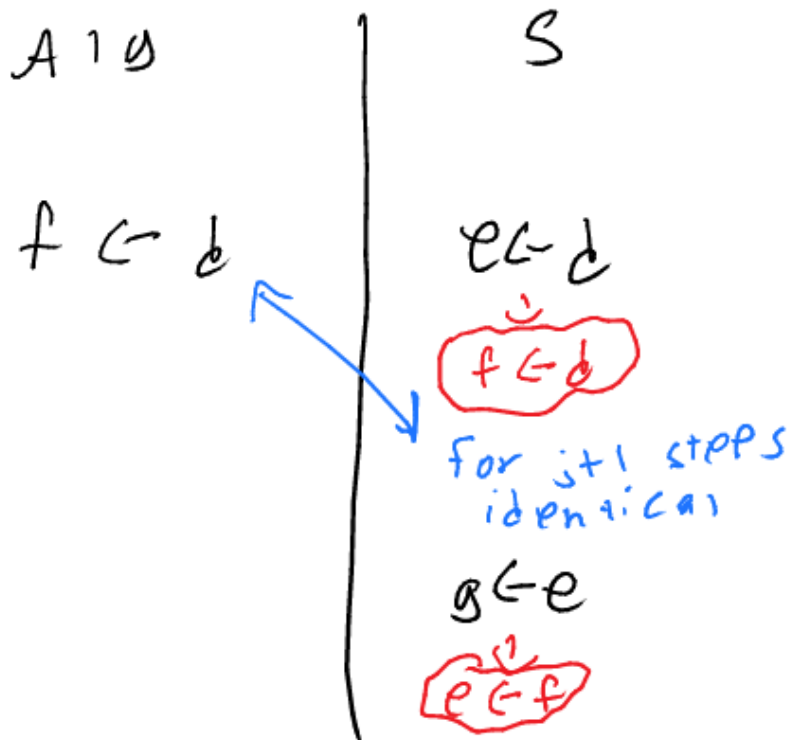
Reminder: We will assume that we only evict items if there is a request that is not in the cache. (won't preemptively remove something)

- Read the book: There is no disadvantage in doing this

Proof: Among all the optimal solutions, pick the one that agrees with our algorithm for the longest period (assuming they all diverge eventually), call it solution S.



- From the algorithm, we know that  $e$  is requested earlier than  $f$ , say at step  $n$ .
- As for the optimal solution, at step  $n$  it must have  $e$ . Let  $t$  be the first time after  $j$  that  $S$  has  $g \leftarrow e$  for some  $g$ .
  - $t$  cannot be later than  $n$  so  $t \leq n$
  - How do we satisfy  $t$  without increasing the number of cache misses in  $S$  and making the solution closer to our algorithm? Evict  $f$  at  $j$  instead of  $e$



So for the proof, either assume that there's an optimal solution that remains stays the same for  $j$  steps and reach a contradiction showing that it is the same for  $j + 1$  steps or show that it keeps going on

## 12 Lecture 12 <2017-10-19 Thu>

### 12.1 Shortest Path in Graphs

- Input: Directed graph  $G = (V, E)$ , source  $s$ , destination  $t$
- $\forall e, \ell_e = \text{length of edge } e$
- Goal: Find the length of the shortest path from  $\underline{s}$  to  $\underline{t}$ .

### 12.2 Dijkstra's Algorithm

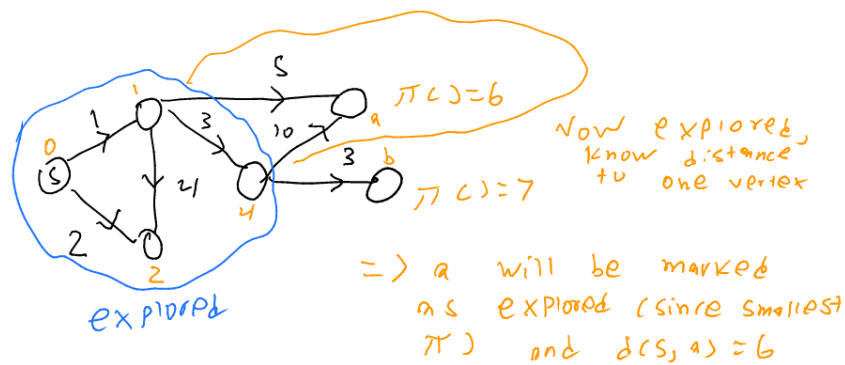
It will find shortest paths from  $s$  to all the other nodes in one go.

- Idea: We keep a list of all vertices (initially includes source)

- We already know the lengths of the shortest paths from  $s$  to all the explored vertices
- At the next step we choose the vertex with smallest

$$\pi(v) = \min_{\ell=(u,v) \text{ } u \text{ is explored}} d(s, u) + \ell e$$

and mark that as explored and set  $d(s, v) = \pi(v)$




---

Alg:  $S$  = set of explored vertices

- $d(u)$  = distance from  $s$  to  $u$  for explored  $u$

set  $S = \{s\}, d(s) = 0$

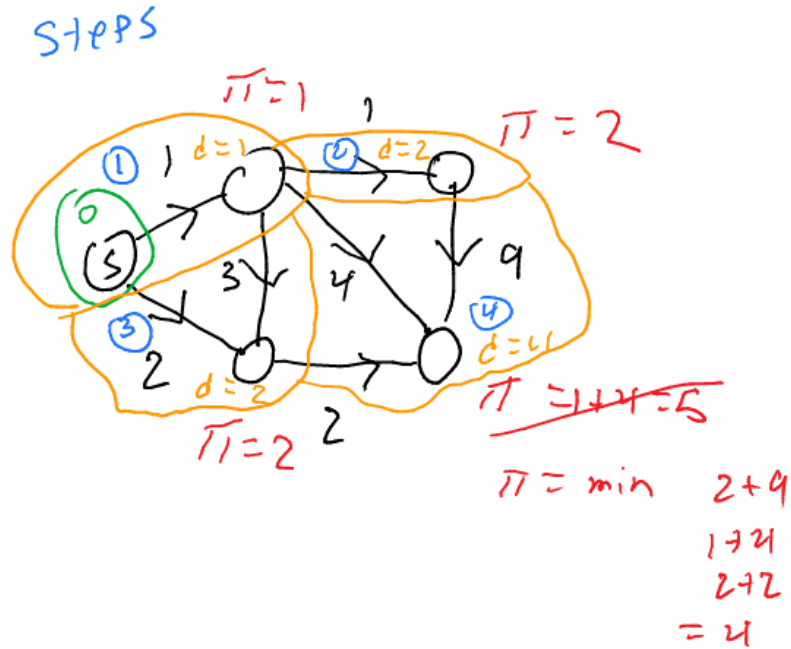
**while**  $S \neq V$  **do** choose  $w \in V - S$  with minimum  $\pi(w) = \min_{\ell=(u,w) \text{ } u \in S} d(u) + \ell e$

$S \leftarrow S \cup \{w\}$

$d(w) = \pi(w)$

**end while**

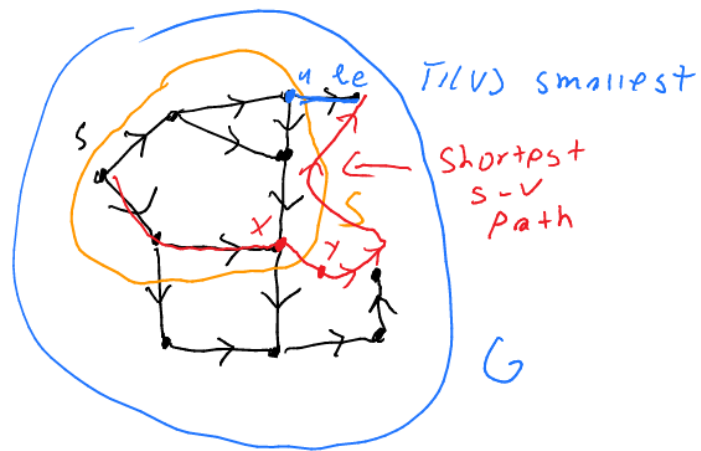
- Example:



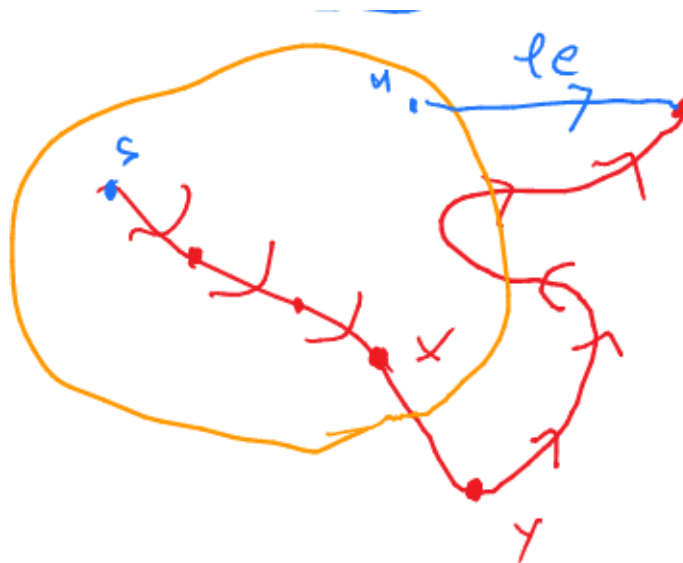
### 12.2.1 Correctness

Claim: During the execution of the algorithm for every  $u \in S$ ,  $d(u)$  is the length of the shortest path from  $s$  to  $u$

- Proof: We use induction on size of  $S$ .
  - Base: Trivial,  $S = \{s\}$ ,  $d(s) = 0$
  - Induction Hypothesis: The claim remains true after adding next  $v$ .



- If  $\pi(v)$  is not the length of the shortest  $s - v$  path
  - \* Consider the shortest  $s - v$  path on the red path
  - \* Consider first vertex  $y$  outside  $S$  on the path. Let  $x$  be the previous vertex.



- $\pi(y) \leq d(x) + \ell_{xy} \leq \text{length of the red path} < \pi(v)$  (because we assumed  $\pi(v)$  is not the shortest path from  $s$  to  $v$ )
- Contradiction as we assumed  $\pi(v)$  was the smallest (we want to

pick smallest  $\pi$  outside of explored area and we showed that  $\pi(y)$  is clearly smaller)

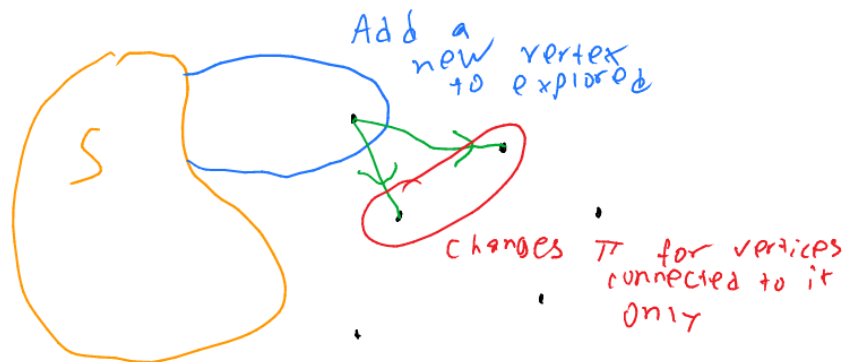
---

Runtime of implementation

```
set  $S = \{s\}, d(s) = 0$   
while  $S \neq V$  do choose  $w \in V - S$  with minimum  $\pi(w) = \min_{\ell=uw, u \in S} d(u) + \ell_e$   
     $S \leftarrow S \cup \{w\}$   
     $d(w) = \pi(w)$   
end while
```

---

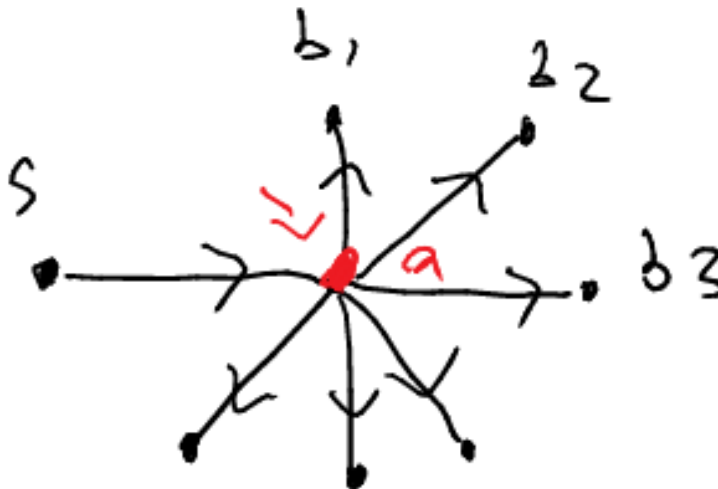
- While:  $|V| = n$  iterations
  - Computing  $\pi(w) \forall w \in V - S \rightarrow O(m) \rightarrow O(mn)$  after multiplying loop iterations
    - \* Might be costly to calculate one  $\pi$ , as we can have many incoming edges to a vertex, up to  $m$  incoming edges
  - Taking their min



When we add  $v$  to  $S$  we only need to update the  $\pi$  value for all  $w \in S - v$  with  $vw \in E$

- If we use a binary heap to implement a priority queue for  $\pi$  values then
- Finding  $\min \pi : O(\log n)$ 
  - (Extracting min from a binary heap)

- Updating the key ( $\pi - value$ ) for all  $w \in V-S$  with  $vw \in E$ : Updating each one at these  $w$ 's costs  $O(\log n)$  (either heapify-up or heapify-down, depending on if we're increasing or lowering key)
  - $n \log n$  since a vertex might have linear amount of outward edges to unvisited vertices
- Note that each edge  $vw \in E$  is causing at most one of those updates. It will never be visited again. Therefore total # of these key updates is at most  $m = |E|$



So all these updates cost  $O(m \log n)$

- Binary heap implementation  $O(m \log n + n \log n)$
- Fibonacci Heap:  $O(m + n \log n)$  (Won't be looking at this in this course as it's much more complicated)

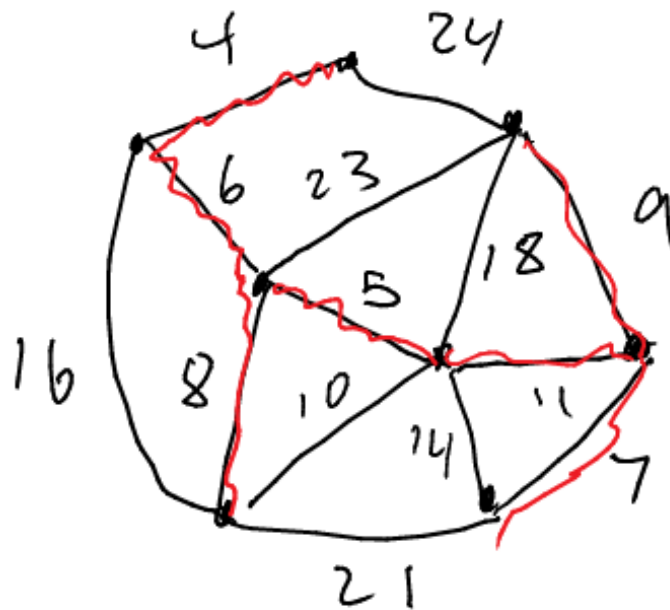
## 13 Lecture 13 <2017-10-24 Tue>

### 13.1 The Minimum Spanning Tree Problems (MST)

- Input: Undirected Connected Graph  $G = (V, E)$

- To every edge  $e$  a positive cost  $c_e > 0$  is assigned
- Goal: Find a spanning tree in  $G$  (i.e. a tree that includes all the vertices of  $G$ ) with minimum cost.

$$cost = \sum_{e \text{ is an edge of the tree}} c_e$$



$$cost = 4 + 6 + 5 + 8 + 11 + 9 + 7$$

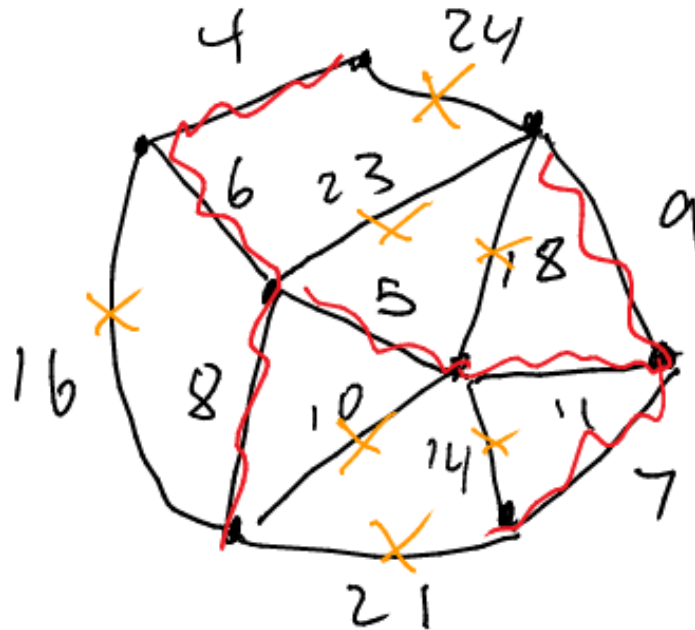
- Why not check all the spanning trees? Very costly.
- Cayley's Thm: Complete graph on  $n$  vertices have  $n^{n-2}$  spanning trees
- So checking all the spanning trees requires exponential time  $\Omega(n^{n-2})$

### 13.1.1 Three Greedy Algorithms:

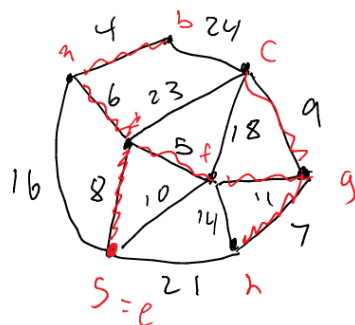
- Kruskal: Start with  $T = \{\emptyset\}$ . At each step add the edge with minimum cost that does not create a cycle until we find a spanning tree (i.e.  $n-1$  edges are added)







- Prims: Start with a node  $s$  (root) and greedily grow a tree from  $s$  outward by adding the cheapest edge that leaves  $T$ .



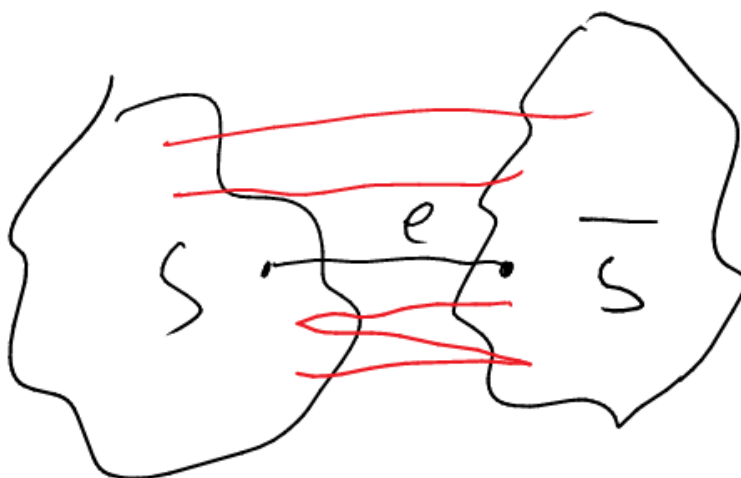
$T = \{ed\}$   
 $\bar{T} = \{a, b, d, f\}$   
 $T = \{ed, df, ad\}$   
 $\bar{T} = \{e, d, df, ad, ab\}$   
 $\bar{T} = \{ed, df, ad, ab, fg\}$   
 $\bar{T} = \{ed, df, ad, ab, fg, gh\}$   
 $\bar{T} = \{ed, df, ad, ab, fg, gh, ac\}$

### 13.1.2 Correctness

Why do they all find the MST?

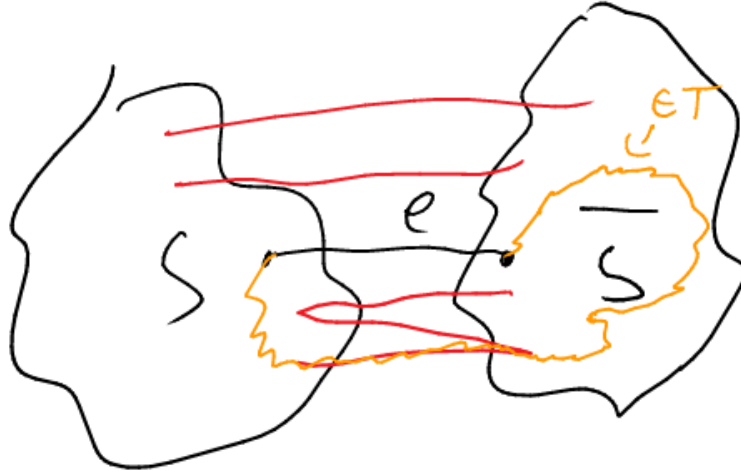
- Simplifying assumption: we assume that all  $c_e$  are different (just to simplify the presentation of the proof)

Cut Property: Let  $S$  be a subset of nodes and  $\underline{e}$  be the minimum cost edge from  $S$  to  $\bar{S}$ . Then  $\underline{e}$  is in every minimum spanning tree.



Proof: Suppose not. Let  $T$  be an MST that does not include  $\underline{e}$

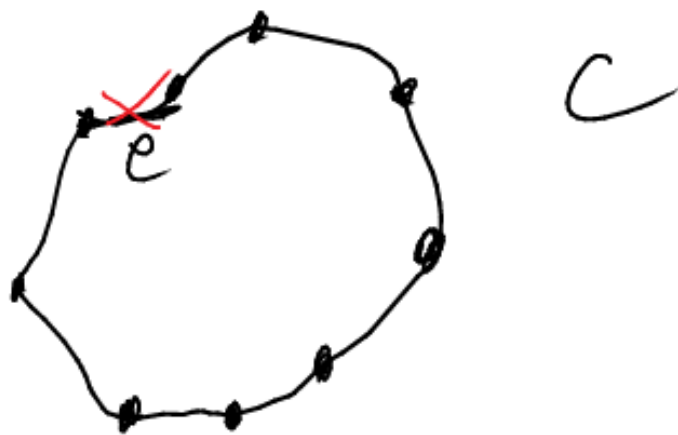
- Consider the path that connects  $u$  to  $v$  in  $T$ .



Pick an edge on this path that goes from  $S$  to  $\bar{S}$  and replace it with  $e$ . Thus way we find another spanning tree with smaller cost. This contradicts the assumption that  $T$  is a MST.

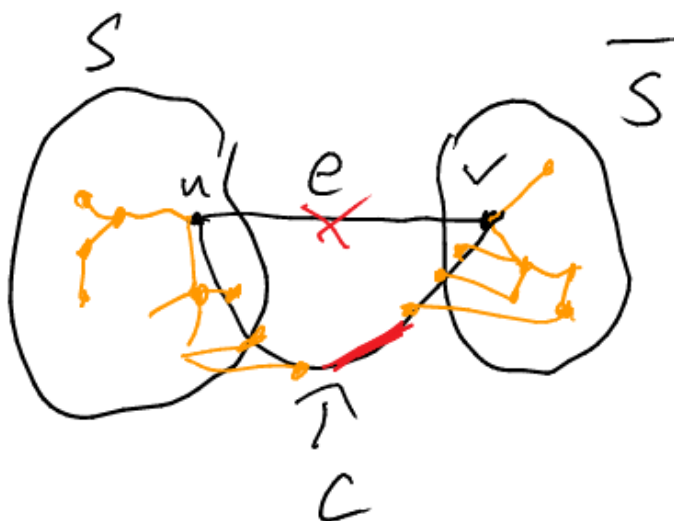
- Why am I allowed to do this? Why doesn't it create a cycle?
  - Can this create a cycle? If adding  $e$  made a cycle, then we had a cycle in the original  $T$ .
  - If there were two paths from  $u$  to  $v$  (such that adding  $e$  makes a cycle), then  $T$  already had a cycle.

Cycle Property: Let  $C$  be a cycle in  $G$  and let  $e$  be the most costly edge on this cycle. Then  $e$  does not belong to any spanning tree.



Proof: Suppose not. There is a MST " $T$ " that contains  $e$ .

- Remove  $e$  from  $T$ . This will break  $T$  into two components  $S$  and  $\bar{S}$ .



Since  $C$  is a cycle it crosses the cycle at some other edge  $e'$ . Adding  $e'$  instead of  $e$  creates a better spanning tree. A contradiction!

---

Prims: Each time add the smallest edge from  $T$  to the rest of the graph (starting from a root  $s$ ).

- Theorem: If all costs are different then there's a unique MST and Prims Alg finds it.
- Pf: Consider a step of the alg. Let  $S$  be the component of the current  $T$ .

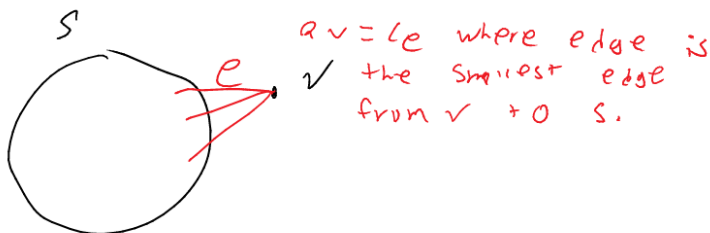


Prim's alg picks the smallest edge  $e$  between  $S$  and  $\bar{S}$  and adds it to  $T$ . By cut property  $e$  is in every MST. So our alg indeed only picks edges that are in every MST. This finishes the proof of the theorem.

---

Implementation: We maintain  $S$  (initially  $S = \{s\}$ )

- For each  $v \notin S$  maintain an attachment cost.
- $a_v =$  The cost of the cheapest edge from  $S$  to  $v$



- At each step we add the vertex with smallest attachment cost to  $S$  and update attachment costs.

- Using a priority queue seems like a good idea.

---

Running time of updating attachment costs once  $v$  is added to  $S$ . Only vertices in  $\bar{S}$  with edges to  $v$  need updates: There are  $\leq \deg(v)$  of these vertices  $w$ . If we use a binary heap to keep attachment costs then the updates have running time  $\deg(v)$ .

$$\deg(v) \times O(\log_n)$$

- $\log_n$  for updating key in binary heap
- So total running time:

$$\sum_{v \in V} \deg(v) (\log n) = O(\log(n)) \times \sum_{v \in V} \deg(v) = O(m \log n)$$

## 14 Lecture 14 <2017-10-26 Thu>

### 14.1 Recall

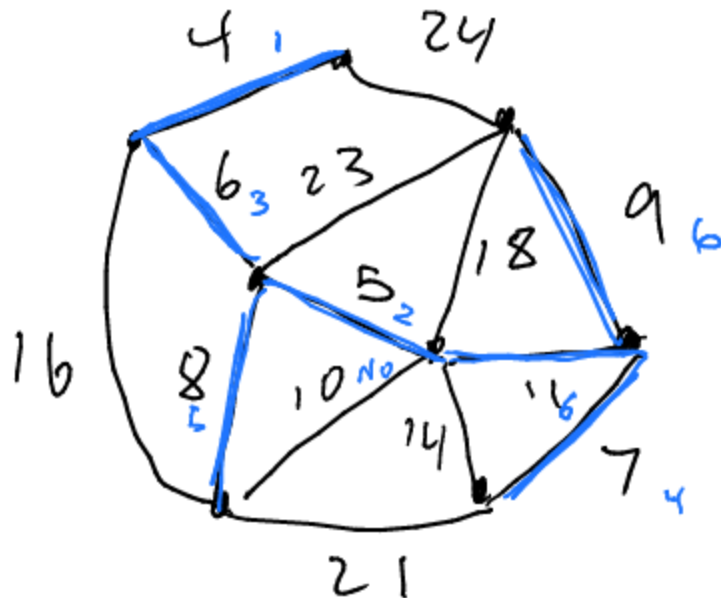
Minimum Spanning Tree Problem. (MST)

- Input: Undirected Graph  $G = (V, E)$ 
  - Every edge  $\underline{e}$  has weight  $c_e > 0$
- Goal: Find a spanning tree in  $G$  with minimum weight
- Cut Property: If  $S, \bar{S}$  is a "cut" in  $G$  and  $\underline{e}$  is the cheapest edge between  $S$  and  $\bar{S}$  then  $\underline{e}$  is in every MST.
  - Shows that Prim's algorithm is optimal
- Cycle Property: If  $e$  has the max cost edge in a cycle  $C$  then  $e$  is not in any MST.

### 14.2 The Kruskal Alg (Proof of Correctness)

Sort the edges from lowest cost to highest.

- Add these edges one by one to the tree (skipping the ones that create a cycle)



#### 14.2.1 Thm:

If costs are different then the Kruskal Algorithm finds the (unique) minimum spanning tree.

#### 14.2.2 Pf:

(Let's look at the edges that the algorithm skips and what we can say about those edges)

- Note that the edges are added in the increasing order according to their costs.
- Let's consider the point in the execution of the algorithm where we are deciding whether to include  $e$  or to skip it.
- We know that the edges included so far are all cheaper than  $e$





We will skip  $e$  only if  $e$  creates a cycle with the currently included edges. In this case  $e$  is the most expensive edge in that cycle and thus by the cycle property, has to be excluded. This shows that all the excluded edges do not belong to any MST. So the included edges form the unique MST. (In contrast with Prim's, here we're showing that every edge we exclude has to be excluded, whereas in the proof for Prim's we showed that every edge added had to be added)

### 14.3 Application

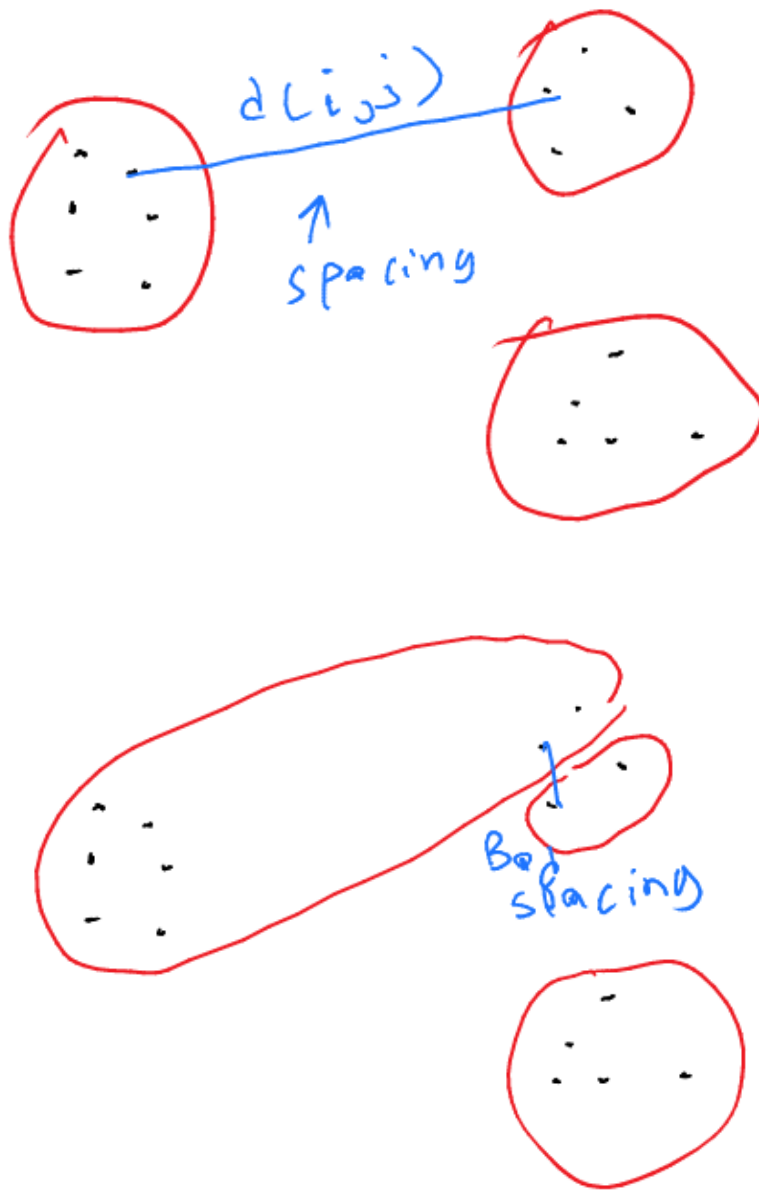
A clustering problem. (Analyzing data, saying what is similar)

$k$ -clustering problem: Given  $n$  points with pairwise distances  $d(i, j) = d(j, i) \geq 0$  (distance between  $i$  and  $j$ )

- $d(i, j) = 0 \iff i = j$
- Not necessarily geometric distances, may be similarities
  - Does not satisfy triangle inequality (or else it would be a metric)
- Input:  $d(i, j) \forall i, j$ , a para

meter  $k \in \mathbb{N}$

- Goal: "Partition" the set of points into  $k$  sets so that the "spacing" is maximized where
  - spacing =  $\min_{i, j \text{ in different clusters}} d(i, j)$
  - Ex.  $k = 3$



Want minimum distance between clusters to be big, or else you're saying two points are different even though they're quite close/similar to each other.

### 14.3.1 Algorithm

An algorithm for this:

- Remark: Note that the "worst" clustering puts the closest two points in different clusters.
- Now among all clusterings that put the two closest points in the same cluster, which one is the worst?



- Answer: Any clustering that separates the next pair of closest points




---

We are basically running the Kruskal algorithm until we have  $k$  connected components and then we stop. Connected components are the desired clusters. Here costs are the distances.

### 14.3.2 Remark

Another way to think about this algorithm is that "we find the MST and then remove the  $(k - 1)$  most expensive edges

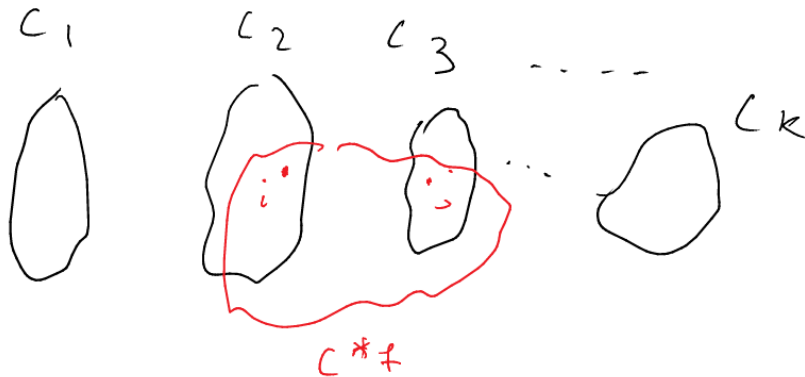
### 14.3.3 Thm

Let  $C^*$  denote the clustering  $C_1^*, \dots, C_k^*$  obtained by deleting the  $(k - 1)$  most expensive edges from the MST. Then  $C^*$  is the  $k$ -cluster with largest spacing,

### 14.3.4 Proof

Let  $C_1, \dots, C_k$  be a different  $k$ -clustering.

- Let  $i$  and  $j$  be two points that are in the same cluster in  $C^*$  but in different clusters in  $C$ .



$$spacing \leq d(i, n)$$

- It suffices to show  $d(i, j) < \text{spacing of } C^*$

## 15 Lecture 15 <2017-10-31 Tue>

### 15.1 Data Compression/Huffman Codes

- Very essential and important result in coding theory
- Wouldn't have the digital world without it

- Trying to compress a file that consists of characters. Want to make it into a smaller file, optimize it. How small can you make it? How redundant is it? Is it the same character over and over or random characters?

Question: Consider a file that uses certain characters (say 32 characters). How can we encode this in bits?

- The easiest way is to assign a unique 5-bit string to each one of these characters (There are  $2^5 = 32$  such strings)
- Example:  $c(a) = 00000, c(b) = 00001, \dots, c(z) = 11001, \dots$

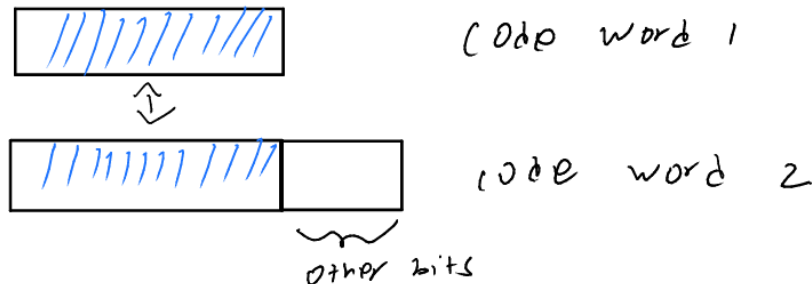
Now given a string of characters we can "code" it and easily "decode" it back

- $abb \xrightarrow{\text{code}} 000000000100001$
- $00001, 00000 \xrightarrow{\text{decode}} b \quad a$

---

Efficiency: Suppose that some letters in the file are way more frequent than the others

- In this case it is more efficient to assign fewer number of bits to frequent characters and more to others
- Doing this in an arbitrary way can cause a problem!
  - Say  $c(a) = 1, c(b) = 01, c(c) = 010$
  - Consider 0101. How can we decode this? 01,01 -> bb, 010,1 -> ca
    - \* Won't know original string here, ambiguity
    - \* Didn't face this problem in the original approach
- How can we avoid this?
  - What property of the code words can guarantee that we won't run into this problem?
  - We do not want any code word to be a prefix of another one.



cw1 is a prefix of cw2.

- Example 01101 is a prefix of 0011010010

### 15.1.1 Prefix property

Def: We say that an encoding has prefix property if no code word is a prefix of another

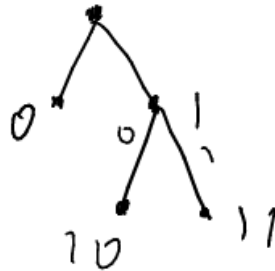
- Ex:  $c(a) = 0, c(b) = 10, c(c) = 11$  has prefix property
- Ex:  $c(a) = 0, c(b) = 10, c(c) = 110, c(d) = 111$  has prefix property
- When you make one code short (like a), you're paying by making the others longer

For this example

- 001011110110
- 0, 0, 10, 111, 10, 110
- a,a,b,d,b,c

### 15.1.2 Prefix Codes as binary trees

- The two codes in the previous two examples can be represented by the leaves of the following trees



This has prefix property because all the codes are only on leaves

Given a binary tree let "Left" be 0 and "Right" be 1

- Now every path from the root to a leaf corresponds to a zero-one string and these strings have the prefix property as the prefix of a code leads to an internal node and not a leaf

### 15.1.3 The best prefix codes

Consider the text and assume that for every letter  $x$ ,  $f_x$  is its frequency.

$$f_x = \frac{\# \text{ of times } x \text{ appears in text}}{\text{total } \# \text{ of characters in text}}$$

Example: *abbacddaa*

- $f_a = \frac{4}{10} = 0.4$

- $f_b = \frac{3}{10} = 0.3$
- $f_c = \frac{1}{10} = 0.1$
- $f_d = \frac{2}{10} = 0.2$

Def: Average bit per letter of a prefix code is

$$ABL(c) = \sum_x f_x \cdot |c(x)| = \frac{\# \text{ of bits in the coded version}}{\text{Size of the original text}}$$

Goal: Minimize # bits in the coded version. Equivalently minimize  $ABL(c)$

Observation: In the binary tree representation, need as many bits as depth of the leaf to represent that number. So

$$ABL(c) = \sum_{\text{leaves } x} f_x \cdot \text{depth}(x)$$

Def: A binary tree is called **full** if every node has 0 or 2 children.

- Claim: The optimal prefix code has a full binary tree.
- Proof: If tree is not full then we can improve it in the following way.



We have the same set of leaves and the depth of some creases so  $ABL(c) = \sum f_x \text{depth}(x)$  has decreased. So the initial tree was not optimal.

How do we come up with the codes? Want more frequent characters to have less bits, put higher up on the tree.



- Observation: Since we are trying to minimize  $\sum f_x \cdot \text{depth}(x)$  We want to assign letters with larger frequencies to leaves at lower depths and letters with smaller frequencies to leaves at higher depths.
- Claim: There is an optimal in which the two least frequent letters are at the highest depth and are siblings.



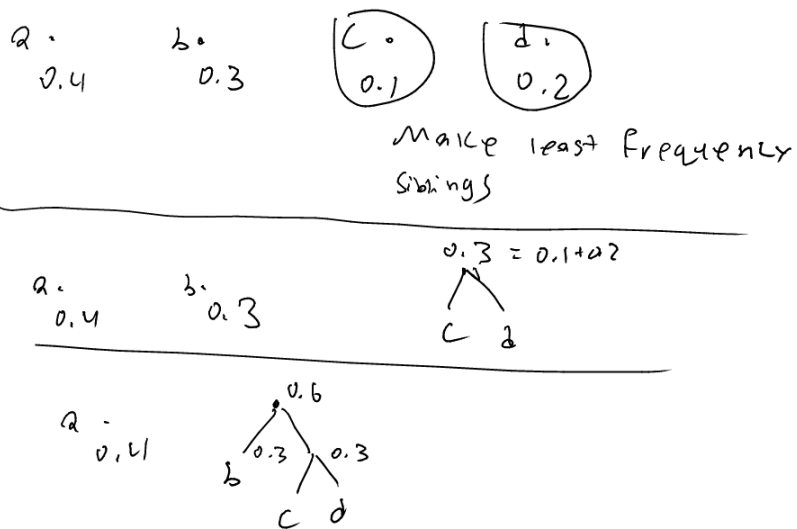
- If one of them was higher up (lower depth) then we could have swapped it with a more frequent letter at higher depth and decreasing  $ABL(c)$ .
- Example:  $f_z = 0.1, f_a = 0.2$ ,  $z$  (least frequent) is on the  $3^{rd}$  layer, then swapping it from 2 to 5 gives you:

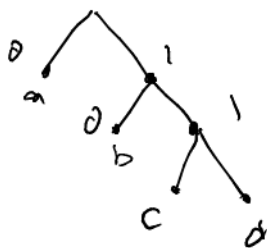
$$0.1 \times 3 + 0.2 \times 5 \implies 0.1 \times 5 + 0.2 \times 3$$

- Shuffling letters at the same depth does not change  $ABL(c)$ . So we can bring the least frequent # letters next to each other

### 15.1.4 Huffman Coding with an example

- $f_a = 0.4, f_b = 0.3, f_c = 0.1, f_d = 0.2$
- At each step we take the two least frequent nodes and make them the children of a new node and assign the summation of the frequencies to this new word. (Bottom up construction)





ex with  $f_a = f_b = f_c = f_d = 0.25$



## 16 Lecture 16 <2017-11-02 Thu>

### 16.1 Recall: Huffman Coding

- Frequencies:  $f_x = \frac{\# \text{ appearances of } x}{\# \text{ of characters}}$
- Goal: Find the best prefix code. Minimize the number of bits. Equivalently minimize

$$ABL(c) = \sum_x f_x |c(x)| = \frac{\# \text{ bits}}{\# \text{ of characters}}$$

- Idea was to see how frequent strings are and to replace frequent strings with shorter codes and rarer strings with longer codes
- With prefix codes they're easy to decode, no ambiguity
  - When you see a code word you know that you're not looking at any other code word

### 16.1.1 Observation 1:

The optimal binary tree is full. i.e. every node has zero or two children

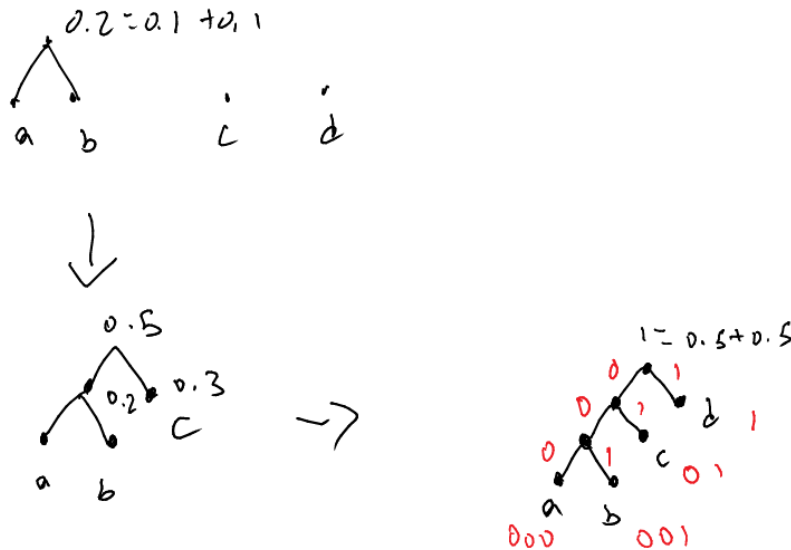
### 16.1.2 Observation 2:

The two least frequent letters appear at max depth and we can assume that they are siblings (if they aren't you can make them siblings without making your code worse).

## 16.2 Huffman Coding

At each step take the two least frequent nodes and make them siblings by creating a new node as their parent and assign the sum of the frequencies to the parent.

- $f_a = 0.1, f_b = 0.1, f_c = 0.3, f_d = 0.5$



### 16.2.1 Thm

Huffman code has the optimal ABL among all prefix codes.

### 16.2.2 Proof

Let's consider an optimal code  $c$  that satisfies the properties mentioned in observations I and II.

The proof is by induction.

- Base case: Only one letter. In this case the best we can do is to assign a 1-bit string to this letter and that is what Huffman Code does.
- I.H. Huffman code is optimal if we have  $m$  letters.
- I. Step: We want to show that Huffman Code is optimal for  $m + 1$  letters

Let  $c$  be an optimal code as described above. Consider the tree of  $c$ :



The two least frequent letters  $a, b$  are as in the picture.

- Consider the same text but replace occurrences of both  $a$  and  $b$  with a new character  $[ab]$ . The new text has  $m$  characters.
- ex:  $a\ b\ c\ c\ c\ a\ b \rightarrow [ab]\ [ab]\ c\ c\ c\ [a]\ [b]$



- Adding the red part gives us the Huffman tree for the original text.

$$ABL(\text{Huffman original}) = ABL(\text{Huffman for the } []) + f_a + f_b$$

I.H.  $ABL(\text{Huffman for } []) \leq ABL(c')$

- (Showed earlier:)  $ABL(c') = ABL(c) + f_a + f_b \rightarrow ABL(\text{Huffman original}) \leq ABL(c)$

### 16.3 Divide and Conquer

- Break up the input into several parts.
- Solve each part recursively.
- Combine the solution to sub-problem into a solution for the original problem

#### 16.3.1 Example: Merge Sort

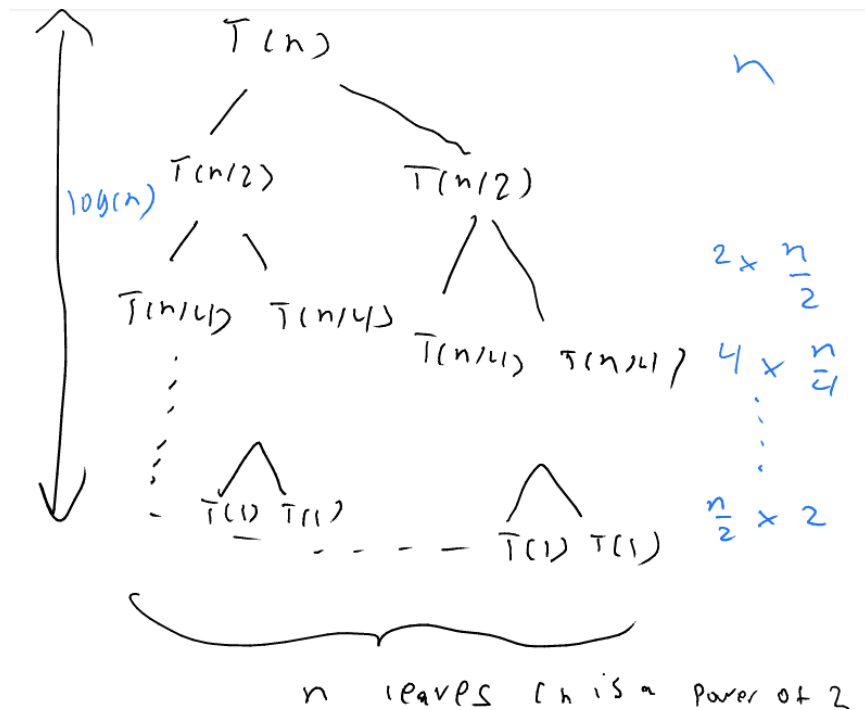
- Divide the array into two equal parts.
- Sort each part recursively
- Merge the two parts into one sorted array.

---

Sort the letters of ALGORITHMS

- ALGOR | ITHMS, Divide  $O(1)$
- AGLOR | HIMS, recursive and sort params  $2T(n/2)$
- AGHILMORST, merge  $O(n)$

$T(n) = 2T(n/2) + O(n)$ , recursive formula, how fast is it really?



- Merging cost =  $n \log n$
- Running time =  $O(n \log n + n) = O(n \log n)$  More formal way to prove using induction:

1. Thm

$$T(n) \leq \begin{cases} 2T(n/2) + cn & \text{if } n > 1 \\ c & \text{if } n = 1 \end{cases}$$

Then  $T(n) \leq cn \log_2 n$  ( $n$  is power of 2).

2. Pf. Assume  $n = 2^m$ ,  $m \in \mathbb{N} \cup \{0\}$

- We use induction on  $m$ .
- Base:  $m = 0 : T(2^0) = T(1) \leq c$
- I.H.  $T(2^m) \leq c2^m \log 2^m = c2^m m$
- I. Step:  $\underbrace{T(2^{m+1})}_n \leq \underbrace{2T(2^m)}_{n/2} + \underbrace{c2^{m+1}}_n \stackrel{I.H.}{\leq} 2c2^m m + c2^{m+1} = c2^{m+1} m + c2^{m+1} = c2^{m+1}(m+1) = cn \log n$



## 17 Lecture 17 <2017-11-07 Tue>

### 17.1 Recall: Merge sort

- $T(n) = 2T(n/2) + O(n)$  (n from merging) =  $\Theta(n \log n)$

### 17.2 Counting Inversions

- Input: A sequence of  $n$  distinct numbers  $a_1, a_2, \dots, a_n$
- Output: How many  $i < j$  satisfy  $a_i > a_j$  (number of pairs that are in the wrong order)?

#### 17.2.1 Example

Array: 5, 2, 1, 4, 3

- (5,2)
- (5,1)
- (5,4)
- (5,3)
- (2,1)
- (4,3)
- 6 inversions

---

1,2,3,4,5 has 0 inversions

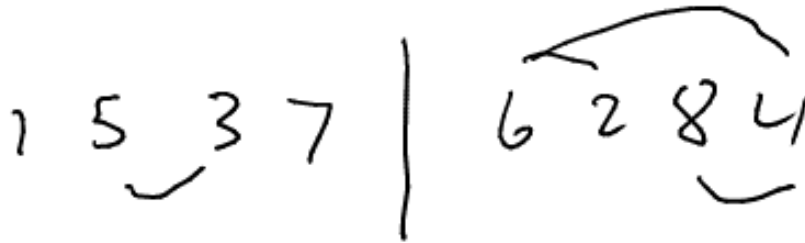
#### 17.2.2 Brute-Force

```
for  $i = 1, \dots, n$  do
  for  $j = i + 1, \dots, n$  do
    if  $a_i > a_j$  then counter++
  end if
end for
end for
```

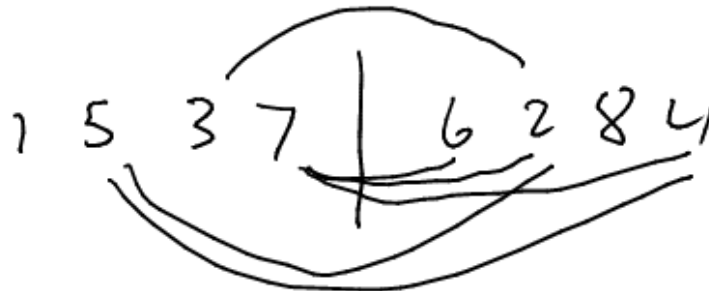
This is  $\Theta(n^2)$ .

### 17.2.3 Divide and Conquer

- Divide: Separate the sequence into two halves  $O(1)$
- Conquer: Count the inversions in each part recursively  $2T(\frac{n}{2})$  (solving 2 halves)



- Combine: Counting number of inversions where  $i$  and  $j$  are in different halves



Naive Combine:

```

for  $i = 1, \dots, \frac{n}{2}$  do
  for  $j = \frac{n}{2} + 1, \dots, n$  do
    if  $a_i > a_j$  then counter++
  end if
end for
end for

```

This is  $\Theta(\frac{n}{2}) = \Theta(n^2)$ , bad.

- $T(n) = 2T(\frac{n}{2}) + \Theta(n^2)$
- $T(n) = \Theta(n^2)$ , no improvement over brute force

---

Idea: Suppose each part is sorted. Then combine can be done faster. -  
 Example

$1 \quad 3 \quad 5 \quad 7$      $1 \quad 2 \quad 4 \quad 6 \quad 8$   
                   ↓    ↓  
                   x  
                           3 2 1 0

$3 \quad 7 \quad 10 \quad 14 \quad 18 \quad 19 \quad | \quad 2 \quad 11 \quad 16 \quad 17 \quad 23 \quad 25$   
   6 3 2 2 2

$O(n)$  if they were sorted.

---

Key Observation: Sorting the two halves does not change the number of inversions in the combine step.



- If  $(i, j)$  is an inversion as in the picture,  $(a_i, a_j)$  will remain one.

---

Attempt:

- Divide into two halves  $O(1)$

- Conquer recursively  $2T(\frac{n}{2})$
- Sort ( $O(n \log n)$ )
- Combine  $O(n)$

$T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2 n)$ . Want to get rid of  $n \log n$

---

Final Alg:

- Divide into two halves  $O(1)$
- Conquer: "Sort" and "count inversions"
- Combine: "Merge"  $O(n)$  the sorted arrays and count inversions  $O(n)$

Running Time:

- $T(n) = 2T(\frac{n}{2}) + O(n)$
- $T(n) = O(n \log n)$

#### 17.2.4 Applications

Closest Pair of Points

- Input: The coordinates of  $n$  points on the plane
- Goal: Find a pair with smallest distance between them

Brute-force,  $O(n^2)$

```

for  $i = 1, \dots, n$  do
  for  $j = i + 1, \dots, n$  do
    if  $dis(P_i, P_j) < min$  then update min
  end if
end for
end for

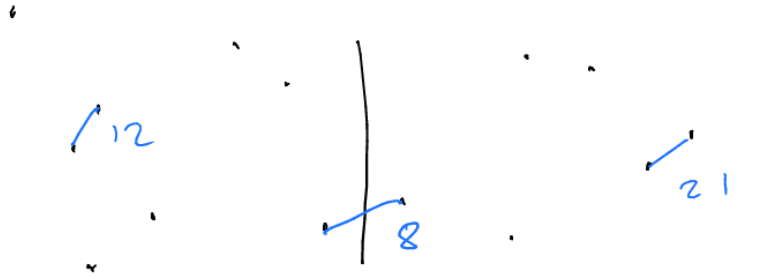
```

Divide-and-conquer: Find a vertical line that separates the points into two equal size halves. Assumption: Points have distinct  $x$  coordinates.



(sort the points according to their  $x$  coordinates and find the middle point,  $O(n \log n)$ )

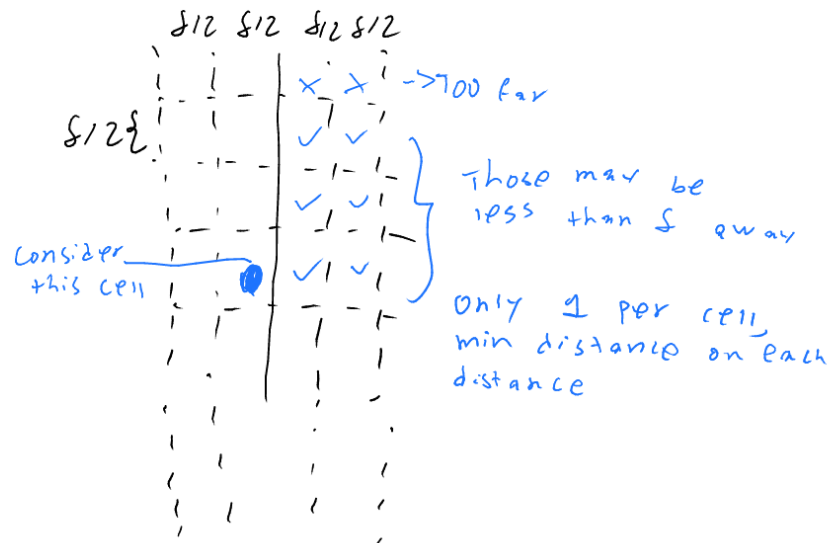
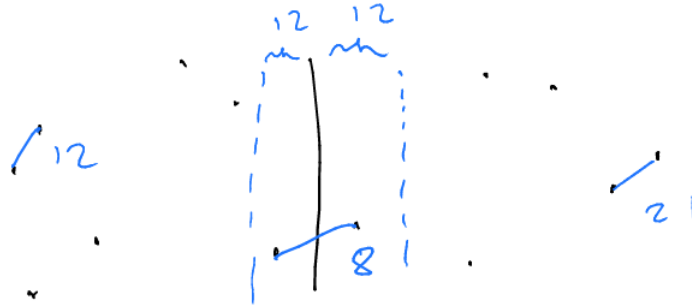
Conquer: Find the closest pair in each part recursively.  $2T(\frac{n}{2})$  Combine: Find the closest pair with each pair on one side of the line. Compare this to the pairs found in "conquer step" (distance between 2 sides we split). Return the best of the three.



Finding the closest pair with points in different parts:

- Naive approach:  $\frac{n}{2} + \frac{n}{2}$
- $T(n) = 2T(\frac{n}{2}) + \Theta(n^2) = \Theta(n^2)$ , no improvement over brute-force.

Observation: If  $\delta$  is the best distance found in the recursion, we only need to consider a window of width  $2\delta$  around the splitting line. Unfortunately all or a significant number of points can be in this  $2\delta$ -strip.



Suppose that the points are sorted according to their  $y$ -coordinates. Now for the point  $P_i$  (blue point in picture) in the  $2\delta$ -strip

- Want to find  $P_i, P_j$  with
  - $P_i$  one side  $+dist(P_i, P_j)$
  - $P_j$  other side

Claim: We only need to look at  $j \in [i - 12, i + 12]$

- Sort the points according to y-coord

```

min =  $\delta$ 
for  $i = 1, \dots, n$  do
    for  $j = i - 20$  to  $i + 20$  do
        if  $P_i, P_j$  are on different sides and  $dist(P_i, P_j) < min$  then update
        min
    end if
end for
end for

```

Running time:  $O(n \log n)$  sort,  $O(n)$  nested for loop gives you:  $T(n) = 2T(n/2) + O(n \log n) = \Theta(n^2 \log n)$

Exercise, can show  $T(n) = 2T(n/2) + O(n) = \Theta(n \log n)$

## 18 Lecture 18 <2017-11-09 Thu>

### 18.1 Integer Addition

So far we've considered these basic operations as constant operations, like adding two numbers, comparing two numbers, etc. But they really aren't, since they're sequence of bits.

- Input: Two  $n - bit$  positive integers  $a, b$
- Output:  $a + b$ ,  $n + 1 - bit$  number.
- Example:

```

11010101
+01111101

```

---

```

101010010

```

Basically need a for-loop to keep track and check if there's a carry over. This is  $O(n)$ . Cannot be improved as even reading the input requires  $\Omega(n)$ .

### 18.2 Integer Multiplication

- Input: Two  $n - bit$  integers  $a, b > 0$
- Goal: Output  $a \times b$

---


$$\begin{array}{r}
0101 \\
\times 1101 \\
\hline
0101 \\
00000 \\
010100 \\
+0101000 \\
\hline
1000001
\end{array}$$

Running time:  $(n-1)$  additions of numbers with at most  $2n$  bits  $\implies O(n^2)$ .  
Can we improve this? Yes.

### 18.2.1 Divide and Conquer approach

Split each one of  $a, b$  into two  $\frac{n}{2}$  - bit (assume  $n$  even) numbers.

- $a = 2^{n/2}a_1 + a_0$
- $b = 2^{n/2}b_1 + b_0$
- Example:  $a = \underbrace{01}_{a_1} \underbrace{01}_{a_0}$   
 $- b = \underbrace{11}_{b_1} \underbrace{01}_{b_0}$

$$ab = 2^n a_1 b_1 + 2^{n/2}(a_1 b_0 + b_1 a_0) + a_0 b_0$$

Recursively compute:  $a_0 b_0$

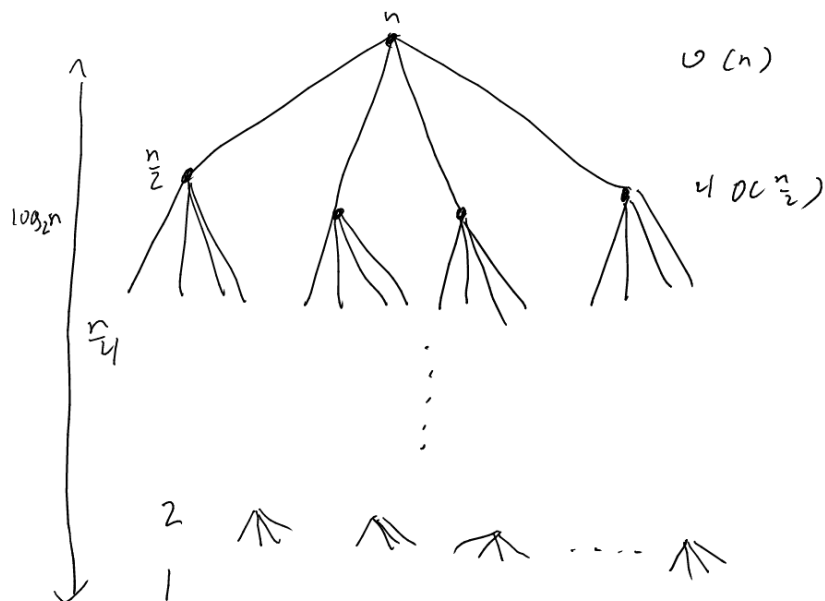
- $a_1 b_0$
- $a_0 b_1$
- $a_1 b_1$
- $4T(n/2)$



Same shifting:  $2^n a_1 b_1, \dots, O(n)$

Same additions: Three of these.  $O(n)$

$$T(n) = 4T(n/2) + O(n)$$



- Number of leaves  $4^{\log_2 n} = n^2$

Expanding this

$$T(n) = \underbrace{T(1) + \dots + T(1)}_{n^2} + O(n) + 4O\left(\frac{n}{2}\right) + \dots \implies T(n) = \Theta(n^2)$$

- So no improvement from before.

Or by using Master Theorem:

$$\log_2 4 > 1 \implies O(n^{\log_2 4}) = O(n^2)$$

---

Karatsuba 1962

- $a = 2^{n/2} a_1 + a_0$
- $b = 2^{n/2} b_1 + b_0$
- $ab = 2^n a_1 b_1 + 2^{n/2}((a_0 + a_1)(b_0 + b_1) - a_1 b_1 - a_0 b_0) + a_0 b_0$

- Different way of writing  $ab$  in the previous approach

---

Karatsuba's Alg:

- Recursively compute:
  - $a_1b_1, a_0b_0, (a_0 + a_1) \times (b_0 + b_1)$
  - This time we only have 3 multiplications instead of 4. The rest is addition and shifting, which costs  $O(n)$ .

$$T(n) = 3T(n/2) + O(n)$$

Applying Master's Theorem:

$$\log_2 3 > 1 \implies \Theta(n^{\log_2 3}) = \Theta(n^{1.585\dots})$$

- This algorithm isn't used in practice and can be improved to  $O(n \log n \log \log n)$  (Fast Fourier transforms)

### 18.3 Matrix Multiplication

- Input: Two  $n \times n$  matrices.  $A, B$
- Output:  $A \times B$

---

Simple Alg:  $C = AB$

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

```

C = [0]n×n
for i = 1, ..., n do
  for j = 1, ..., n do
    for k = 1, ..., n do
      Cij += AikBkj
    end for
  end for
end for

```

This is  $O(n^3)$  if we assume that the matrix has a very simple form (0 and 1's or if we're just counting number of multiplication rather the actual bits like before).

### 18.3.1 Divide and Conquer

Split  $A, B$  into four  $\frac{n}{2} \times \frac{n}{2}$  blocks

$$A = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix}, B = \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

where  $A_{00}, A_{01}, A_{10}, A_{11}, B_{00}, \dots, B_{11}$  are  $\frac{n}{2} \times \frac{n}{2}$  matrices.

- Example:

$$A = \begin{bmatrix} \begin{matrix} A_{00} & A_{01} \\ \begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} & \begin{bmatrix} 5 & 0 \\ 0 & 0 \end{bmatrix} \\ \begin{matrix} A_{10} & A_{11} \\ \begin{bmatrix} 0 & 2 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix} \end{matrix} \end{bmatrix}$$

$$AB = \begin{bmatrix} \underbrace{A_{00}B_{00} + A_{01}B_{10}}_{\frac{n}{2} \times \frac{n}{2}} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

$$T(n) = 8T(n/3)(\text{multiplications}) + O(n^2)(\text{additions})$$

Recursively compute:

$$A_{00}B_{00}, A_{01}B_{10}, A_{00}B_{01}, \dots$$

Using Master Theorem:

$$\log_2(8) > 2 \implies \Theta(n^3)$$

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \times \begin{bmatrix} [r]B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$\begin{aligned}
P_1 &= A_{00} \times (B_{01} - B_{11}) \\
P_2 &= (A_{00} + A_{01})B_{11} \\
P_3 &= (A_{10} + A_{11})B_{00} \\
P_4 &= A_{11}(B_{10} - B_{11}) \\
P_5 &= (A_{00} + A_{11})(B_{00} + B_{11}) \\
P_6 &= (A_{01} - A_{11})(B_{10} + B_{11}) \\
P_7 &= (A_{00} - A_{10})(B_{00} + B_{01})
\end{aligned}$$


---

$$\begin{aligned}
C_{00} &= P_5 + P_4 - P_2 - P_6 \\
C_{01} &= P_1 + P_2 \\
C_{10} &= P_3 + P_4 \\
C_{11} &= P_5 + P_1 - P_3 - P_2
\end{aligned}$$

Strassen Alg: 1971

$$T(n/2) + O(n^2)(\text{additions})$$

- Compute:  $P_1, \dots, P_7$  recursively (all size  $\frac{n}{2} \times \frac{n}{2}$ )
- Compute:  $C_{00}, C_{01}, C_{10}, C_{11}$  using them

$$T(n) = 7 \times T(n/2) + O(n^2)(\text{additions/subtractions})$$

$$\log_2 7 > 2 \implies \Theta(n^{\log_2 7}) = \Theta(n^{2.81\dots})$$

Matrix computations are everywhere, graphics do matrix computations all the time.

- Can this be improved? Has been improved to something like  $\Theta(n^{2.37\dots})$

Open Problem: Can we do matrix multiplication in  $O(n^{2+O(1)})$ ?

- Say  $O(n^2(\log n)^{1000})$
- We don't know

## 19 Lecture 19 <2017-11-14 Tue>

### 19.1 Dynamic Programming

Each instance of the problem is solved by "looking up" the solution to smaller instances of the problem. Hence often we fill up a table with solutions to

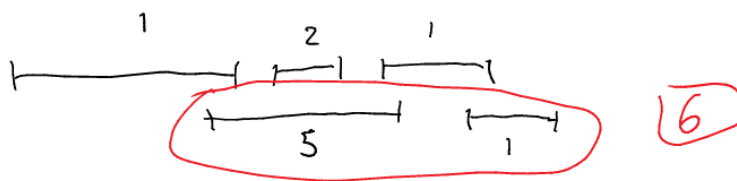
instances of the problem, starting from smaller values of a parameter, and each one looks up the earlier values in the table to compute the current entry.

Name "Dynamic" was suggested by Bellman in the 50's. Was very theoretical but not much money was given to theoretical things during the time. Wanted to make the name nicer to get more funding, even though this name doesn't fit that much.

### 19.1.1 Weighted interval scheduling

Recall: Weighted interval scheduling Input: Jobs  $(s_i, f_i, v_i)$ , where  $s$  is starting time,  $f$  is finishing time and  $v$  is the value,  $v_i \geq 0$

Goal: Find the maximum value, non-overlapping set of jobs.



Remark: Setting all  $v_i$  to 1 recovers the simple interval scheduling problem (maximizing # of jobs)

Recall: In the original interval scheduling problem, we showed that the greedy algorithm that sorts the jobs  $f_1 \leq f_2 \leq \dots \leq f_n$  and picks each job iff it is consistent with the earlier chosen jobs. In the above example, we would choose 1, 2, 1 on the top (ignoring values) and only getting 4 instead of 6.

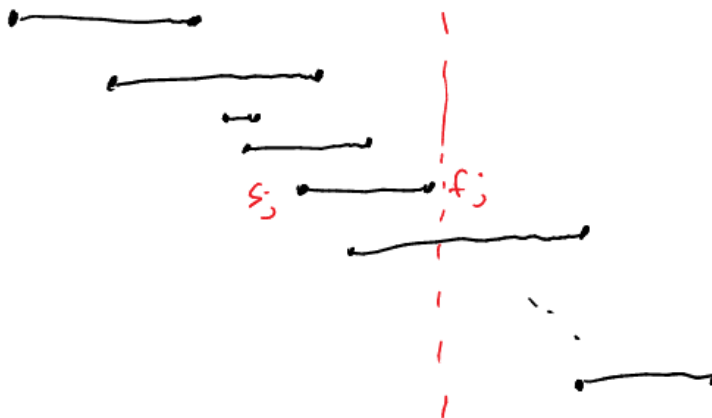


- Come up with any greedy algorithm, won't work. If you pick biggest value first, the above example won't work. If you look at ratios between time and values it won't work, etc.

---

Let us still sort them by their finishing time

$$f_1 \leq f_2 \leq \dots \leq f_n$$



- Consider the problem until time  $f_j$  -> Two cases:
  - Reject job  $j$  ->
    - \* We are optimizing value until job  $j - 1$
  - Accept job  $j$  -> gain  $v_j$  + we are optimizing value until the last job where finishing time is before  $s_j$  (call this  $P(j)$ )

$$P(j) = \max\{i | f_i \leq s_j\}$$

---

Def:  $Opt[j]$  = maximum profit that we can make from jobs  $1, \dots, j$

- We just showed  $Opt[j] = \max\{Opt[j - 1], v_j + Opt[P(j)]\}$  with base case  $Opt[0] = 0$ . Note that  $Opt[P(j)]$  considers jobs  $1, \dots, P(j)$

1. Alg

- Sort the jobs so that

$$f_1 \leq f_2 \leq \dots \leq f_n, O(n \log n)$$

- Compute the values  $P(j)$  for  $j = 1, \dots, n$ . Can be done easily in  $O(n \log n)$  via binary search, can even be possible to do in  $O(n)$  but we already have  $O(n \log n)$  from sorting above anyway.
- $Opt[0] = 0$

**for**  $j = 1, \dots, n$  **do**

$Opt[j] = \max\{Opt[j-1], v_j + Opt[P(j)]\}$

// Note that  $Opt[j-1]$  and  $Opt[P(j)]$  are already computed and in an array.

**end for**

Return  $Opt[n]$

Total  $\rightarrow O(n \log n)$

What if we implement this as a recursive program?

- Sort  $f_1 \leq \dots \leq f_n$
- Compute  $P(j)$  for  $j = 1, \dots, n$

ComputeOpt( $j$ )

**if**  $j == 0$  **then**

return 0

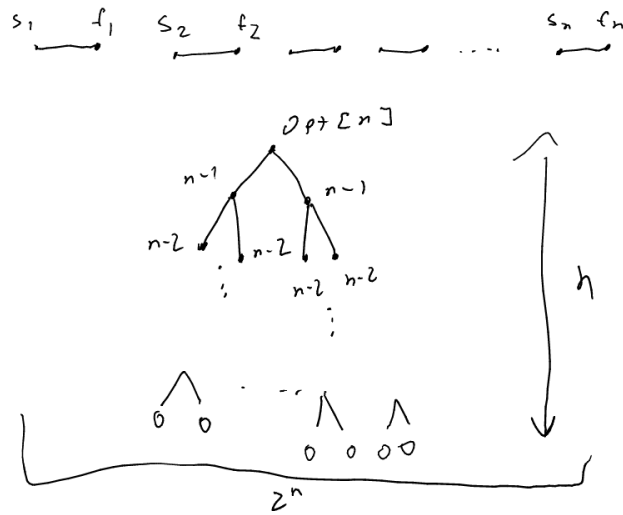
**else**

return  $\max(\text{ComputeOpt}(j-1), v_j + \text{ComputeOpt}(P(j)))$

**end if**

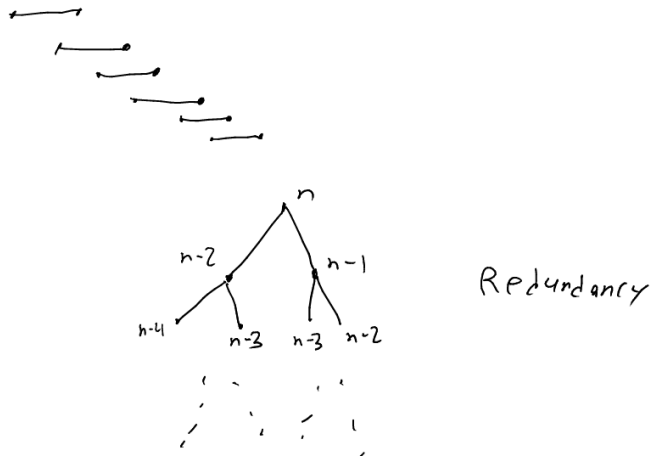
Running time? You have recursion and a binary structure that will just go down. This can get exponential.

- Example:



Running time:  $O(2^n)$ . But why? This is similar to the way we did it dynamically. We recompute opt values many times, and this leads to exponential time.

- Another example:



Storing the values in an array and not recomputing them is dynamic programming. Really just recursion except we save the values.

## 2. Memorization



```

Sort  $f_1 \leq \dots \leq f_n$ 
Compute  $P(j)$ ,  $j = 1, \dots, n$ 
for  $j = 1, \dots, n$  do
     $Opt[j] = \text{empty}$ 
end for
 $Opt[0] = 0$ 
ComputeOpt(j)
if  $Opt[j] = \text{empty}$  then
     $Opt[j] = \max\{ComputeOpt(j-1), v_j + ComputeOpt(P(j))\}$ 
else
    return  $Opt[j]$ 
end if

```

Now for every  $j$  we make two recursive calls at most once.  $\rightarrow O(2 \times n) = O(n)$  for recursive step  $\rightarrow O(n \log n)$  total time.

---

This memorization technique is useful when computing most of the entries in the table are not necessary.



### 19.1.2 Knapsack Problem

(Will be also used in COMP360 with Hatami, important)

- Given  $n$  objects and a "knapsack"
- Item  $i$  has weight  $w_i > 0$  (integers)
  - value  $v_i > 0$
- Knapsack has capacity  $W$  (in weight)

Goal: Fill the knapsack so as to maximize the value without exceeding its capacity.

- Ex:

	$w_i$	$v_i$	ratio
#1	1	1	1
#2	2	6	3
#3	5	18	$3\frac{3}{5}$
#4	6	22	$3\frac{2}{3}$
#5	7	28	4

Optimal total value: 40

---

Greedy Alg: Compute value/weight ratio and start from the items with higher ratio -> #5, #2, #1 ->  $value = 28 + 6 + 1 = 35$

Optimal #3, #4 -> 40

This is an NP complete problem. Can't solve it greedily. Let's try and solve it.

Attempt #2: Define  $Opt[i] = \max$  value if we only consider items  $1, \dots, j$

Two cases:

- Case 1: Not include item  $j$ 
  - $Opt[j - 1]$  can be achieved
- Case 2: Item  $j$  in the knapsack. Value  $v_j$  included.
  - $v_j + Opt[j - 1]$ ?
    - \* No because this will use  $w_j$  of the capacity!
  - It is  $v_j + Opt(j - 1, \text{new cap} = W - w_j)$

---

Solution:  $Opt[j, u] = \max$  value if we consider items  $1, \dots, j$  with capacity  $u$ , where  $u = 0, \dots, W$

---

```

for  $u = 0, \dots, W$  do
   $Opt[0, u] = 0$ 
end for
for  $j = 1, \dots, n$  do
   $Opt[j, u] = \max\{opt[j - 1, u], v_j + opt[j - 1, u - w_j]\}$ 
  // Only if  $w_j \leq u$ 
end for

```

Running time:  $O(nW)$ . This isn't considered polynomial.  $W$  is just a number here and may be a number to a big exponent.

## 20 Lecture 20 <2017-11-16 Thu>

### 20.1 Recall

Knapsack  $\Theta(nW)$

- Input: Weights  $w_1, \dots, w_n$ 
  - Values  $v_1, \dots, v_n$
  - Capacity  $W$
- $Opt[i, u]$  = optimal value items  $1, \dots, i$  capacity  $u$

---

Weren't able to solve this greedily, but we did come up with a recursive algorithm that could be implemented through dynamic programming.

$$Opt[i, u] = \max \begin{cases} opt[i-1, u] \\ v_i + opt[i-1, u-w_i] \end{cases} \quad \text{if } w_i \leq u$$

$Opt[n, W]$  = Final answer

This is not a polynomial time algorithm. Why?

- Recall: An algorithm is polynomial time if it's running time is  $O(N^c)$  where  $c$  is a constant and  $N$  is the size of the input (in bits)

---

What is  $N$  here?

- input  $w_1, \dots, w_n$ 
  - $v_1, \dots, v_n$
  - $W$

How many bits do we need for this?  $N$ ?

$$N = O(\log w_1 + \log w_2 + \dots + \log w_n + \log v_1 + \log v_2 + \dots + \log v_n + \log W)$$

Where here logs are base 2 (for number of bits to store the number in binary)

Ex:  $w_1 = 2, w_2 = 4, v_1 = 3, v_2 = 1, w = 2^{10}$

$$10, 100, 11, 1, \underbrace{100\dots00}_1$$

---

Running Time  $\Theta(nW)$

- Let us show that this is not polynomial time:
- $W = 2^n$  and  $w_1 = \dots = w_n = v_1 = \dots = v_n = 2$
- $N = O(\log w_1 + \dots + \log w_n + \log v_1 + \dots + \log v_n + \log W) = O(n)$

Running Time:  $\Theta(nW) = \Theta(n2^n)$

## 20.2 Another Example

Input  $n$

- Goal: Is  $n$  a prime?
- ```

for  $i = 2, \dots, n - 1$  do
  if  $(n \bmod i) = 0$  then
    return "Not prime"
  end if
end for
return Prime

```

---

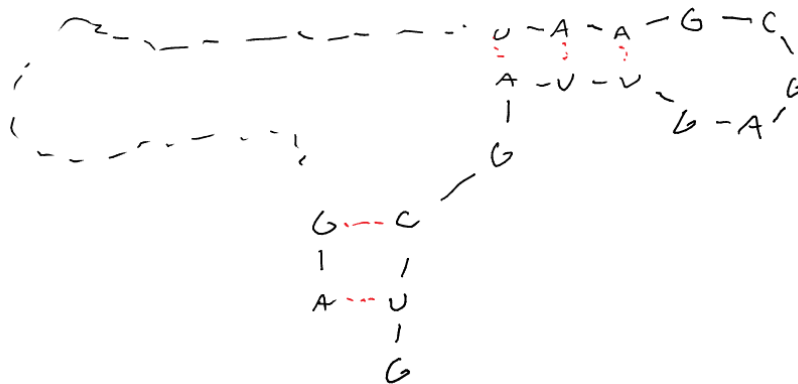
Running time  $\Omega(n)$

- Input size:  $N = \log_2 n$
- Running time:  $\Omega(n) = \Omega(2^N)$

## 20.3 RNA secondary structures

RNA: String  $B = b_1, \dots, b_n$  over alphabet  $\{A, C, G, U\}$ .

- Ex: *GUCGAUUAGC...GA*
- Secondary Structure: RNA's tend to loop back and make pairs with itself

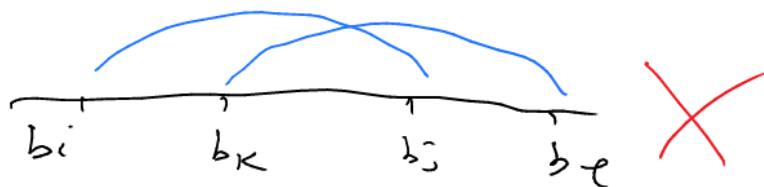


Secondary Structure: Is a set of pairs  $S = \{(b_i, b_j), \dots\}$  that satisfies three properties

[Watson-Crick (Property)]:  $S$  is a matching and each pair is a Watson-Crick complement:  $A - U, U - A, C - G, G - C$

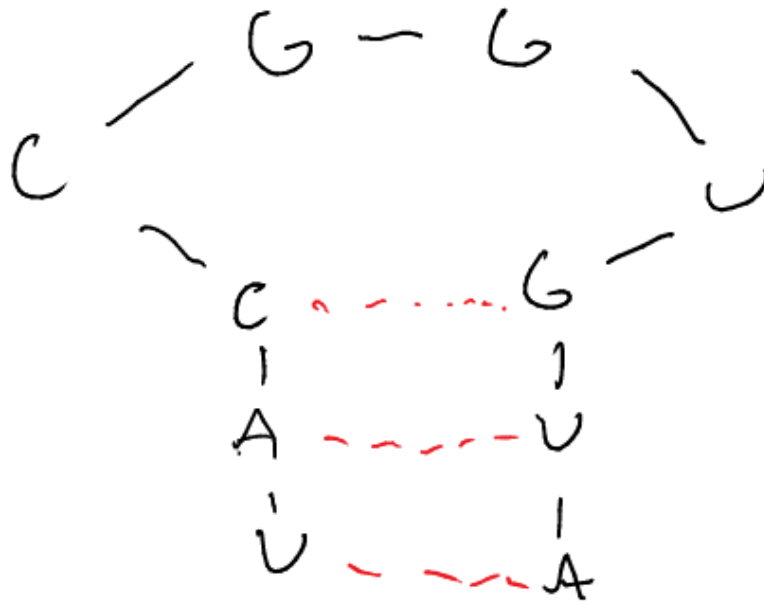
[No sharpturns]: If  $(b_i, b_j), i < j$  is a pair then  $i < j - 4$  (at least distance 4 between a pair)

[No crossings]: We can't have  $(b_i, b_j), (b_k, b_l)$  with  $i < k < j < l$

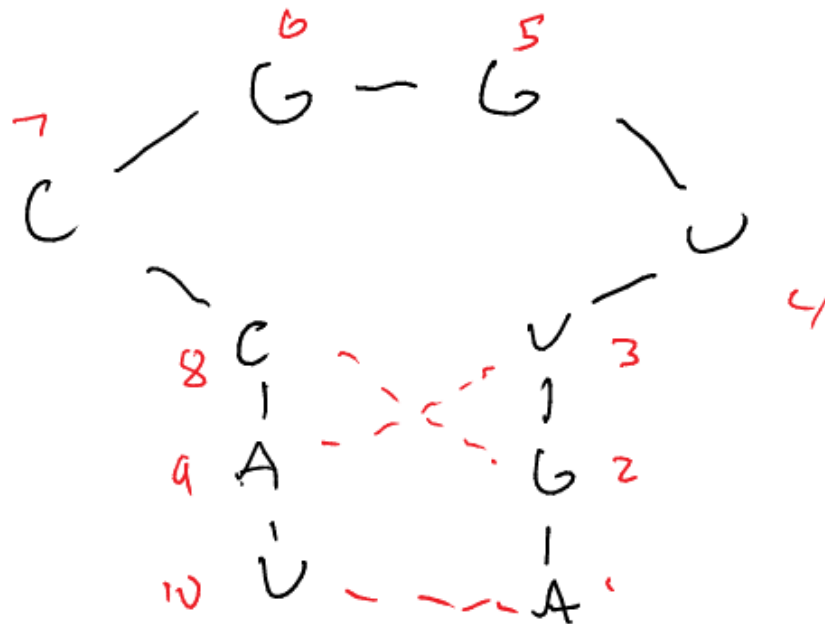


RNA molecules tend to form a structure with maximum number of pairings

- Example: *AUGUGGCCAU*



- Bad pairing:



has crossing

$(2, 8)$

$(3, 9)$

Problem: Given an RNA sequence find the maximum number of pairings.

---

Attempt #1: Let  $Opt[j]$  to be the optimal number of pairs until point  $j$  :  $b_1, \dots, b_j$

- Case 1:  $b_j$  is not in any pair

$$- Opt[j] = Opt[j - 1]$$

- Case 2:  $b_j$  is paired with  $b_t$

$$Opt[i, j] = 1 + Opt[1_t - 1] + Opt[t + 1, j - 1]$$

$$Opt[i, j] = \max\{Opt[i, t - 1] + Opt[t + 1, j - 1]\}$$

- Max is over all  $t$  such that  $(b_t, b_j)$  is a legal pair.

$$- i \leq t < j - 4$$

$$- (b_t, b_j) \in \{AU, UA, CG, GC\}$$

$Opt[i, j] = 0$  if  $i \geq j - 4$  Can be implemented using memorization (recursion)  
or can be implemented with loops

---

The second way to implement this:

```

if  $i \geq j$  then
     $Opt[i, j] = 0$ 
end if
for  $k = 5, \dots, n - 1$  do
    for  $i = 1, \dots, n - k$  do
         $j = i + k$ 
        Compute  $Opt[i, j]$  using the formula we obtained earlier.
    end for
end for

```

---

Running time  $O(n^3)$  because Compute  $Opt[i, j]$  takes  $O(n)$

## 21 Lecture 21 <2017-11-21 Tue>

### 21.1 Longest Common Subsequence

- Input: Two strings  $A = a_1 \dots a_n$ ,  $B = b_1 \dots b_m$  from some alphabet.
- Goal: Find the longest common subsequence (not necessarily consecutive, can skip letters) of the two strings

#### 21.1.1 Example

- $A = aaabcadb$
- $B = aaaaeb$
- $\rightarrow aaaab$
- Want to use Dynamic Programming
  - What will be our sub-problems?



- Attempt:  $Opt[i] =$  longest common subsequence up to position  $i$  in both strings
  - If  $a_i \neq b_i$ , we'll run into trouble. Maybe we want to match the  $i^{th}$  character with a later character in the other string, so we can't just cut off the end after  $i$  for both strings

Define  $D[i, j] =$  longest common subsequence for  $a_1 \dots a_i, b_1 \dots b_j$

$$D[i, j] = \max \begin{cases} D[i-1, j] & i > 0 \\ D[i, j-1] & j > 0 \\ 1 + D[i-1, j-1] & a_i = b_j \end{cases}$$

- 1<sup>st</sup> case, not using  $a_i$ , drop it
- 2<sup>nd</sup> case, not using  $b_j$ , drop it
- 3<sup>rd</sup> case, they are equal, use them both and add to length

Base cases:

$$\begin{aligned} D[0, j] &= 0 \forall j = 0, \dots, m \\ D[i, 0] &= 0 \forall i = 0, \dots, n \end{aligned}$$

Solution is:  $D[n, m] =$  length of the longest common subsequence.

### 21.1.2 Example 2

- $A = abc$
- $B = aceb$

| D | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 2 |
| 3 | 0 | 1 | 2 | 2 | 2 |

- $D[1, 1] = \max D[0, 1], D[1, 0], 1 + D[0, 0]$
- $D[1, 2] = \max D[0, 2], D[1, 1]$ 
  - a & ac

- $D[1, 3] = \max D[0, 3], D[1, 2]$
- $D[2, 1] = \max D[1, 1], D[2, 0]$
- $D[2, 4] = \max D[1, 4], D[2, 3], 1 + D[1, 3]$

What if we want to get the actual sub-string? We can trace it back essentially.

- Interested in  $D[3, 4]$ , where is it coming from?  $D[3, 3]$  and  $D[2, 4]$ . Since they're equal here, just arbitrarily chose one, say  $D[2, 4]$ .

– So  $D[3, 4] = D[2, 4] = 1 + D[1, 3]$

\* Since this is an increase of  $D[1, 3]$ , then our last character is the character read by  $D[2, 4]$

–  $1 + D[1, 3] = 1 + D[1, 3] = 1 + D[1, 2] = 1 + D[1, 1] = 1 + 1 + D[0, 0]$

\* Previous character is  $D[1, 1]$

- Now we get our sub-string:  $ab$
- Fill up  $D[, ]$  as it is described above.

FindString(i,j)

**if**  $a_i == b_j$  and  $(D[i, j] = 1 + D[i - 1, j - 1])$  **then**

    print " $a_i$ " (matching  $a_i$  with  $b_j$ )

**else**

**if**  $D[i, j] == D[i - 1, j]$  **then**

        FindString(i-1,j)

**else**  $(D[i, j] == D[i, j - 1])$

        FindString(i,j-1)

**end if**

**end if**

## 21.2 Edit Distance

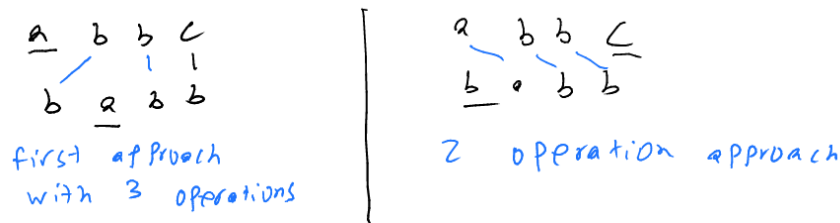
- Inputs:  $A = a_1 \dots a_n$ ,  $B = b_1 \dots b_m$  as above.
- Goal: Find the minimum number of operations needed to convert  $A$  to  $B$ . The possible operations are as follows
  - delete(i): Deletes the  $i^{th}$  letter from  $A$
  - insert(i,x): Inserts letter  $x$  right after the  $i^{th}$  position.
  - replace(i,x): Replaces the  $i^{th}$  letter with  $x$ .

### 21.2.1 Example

$A = abbc, B = babb$

One way to convert (that isn't the best way):

- $abbc \xrightarrow{del(1)} bbc \xrightarrow{ins(1,a)} abbc \xrightarrow{rep(4,b)} babb \rightarrow 3 \text{ operations}$
- $abbc \xrightarrow{del(4)} abb \xrightarrow{ins(0,b)} babb \rightarrow 2 \text{ operations}$



$D[i, j]$  = Edit distance from  $a_1 \dots a_i$  to  $b_1 \dots b_j$

$$D[i, j] = \min \begin{cases} D[i-1, j] + 1 & (del(i)) \\ D[i, j-1] + 1 & (ins(i, b_j)) \\ D[i-1, j-1] & (\text{only if } a_i = b_j) \\ D[i-1, j-1] + 1 & Rep(i, b_j) \end{cases}$$

4<sup>th</sup> case is  $D[i-1, j-1] + 1$  since we have to add one operation but the two end letters are now the same like in the 3<sup>rd</sup> case.

- Base cases:

$$D[0, j] = j \text{ for } j = 0, \dots, m$$

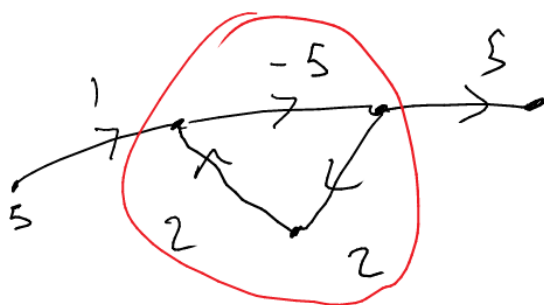
$$D[i, 0] = i \text{ for } i = 0, \dots, n$$

Either insert everything (first line) or delete everything (second line)

### 21.3 Shortest Path with Negative Weights

- Input: Directed graph  $G = (V, E)$  with edge weights  $c_{uv} \in \mathbb{R}$  (possibly negative) and a vertex  $s$  (source).
- Goal: Find the length of the shortest path from  $s$  to all other vertices (like Dijkstra's but with negative weights).

Negative cycles can cause problems. Since if we have a negative cycle then the best way to get to a vertex is not through a path and indeed it will have cost  $-\infty$

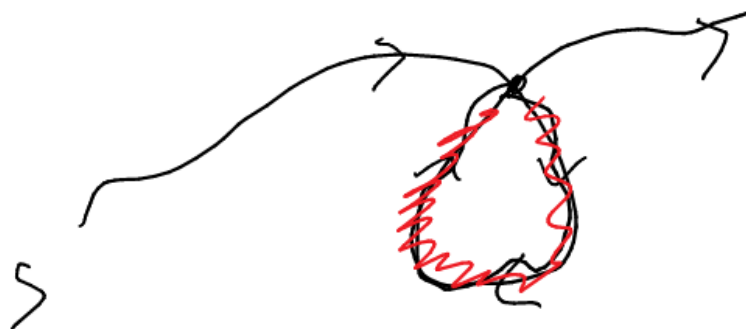


$$-5 + 2 + 2 = -1$$

$$s, \underline{-5, 2, 2}, \underline{-5, 2, 2}, \underline{-5, 2, 2}, \dots$$

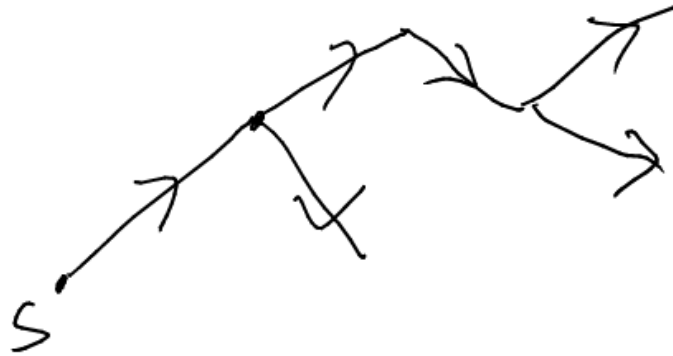
Assumption: The input does not have any negative cycles (or else it would be NP-complete as you can keep looping on the negative cycle).

Claim: Under this assumption the minimum cost way to go from  $s$  to  $t$  is through a path.

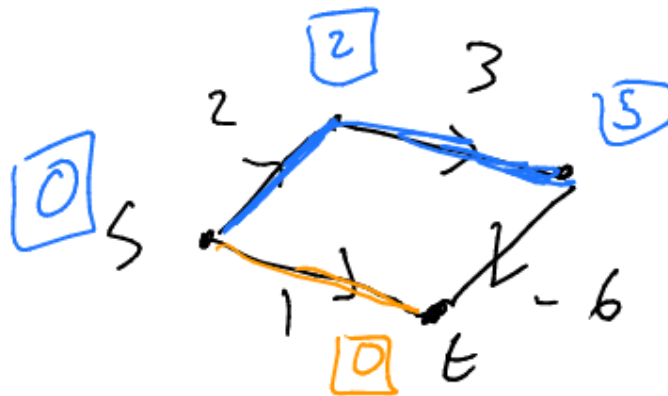


If the minimum cost way revisits any vertices, then we can remove a cycle from it and decrease its cost.  $\rightarrow$  Positive cycles are redundant.

- Does Dijkstra work under this assumption?



- No it does not work. Take the following as an example:



Maybe shift all the costs to make them positive



This doesn't work, shifting one path by more since it has more edges

## 22 Lecture 22 <2017-11-23 Thu>

### 22.1 Shortest Path with Negative Weights (Continuation)

Finding shortest path in directed graphs with no negative cycles from a vertex  $s$ . We will use dynamic programming for the algorithm.

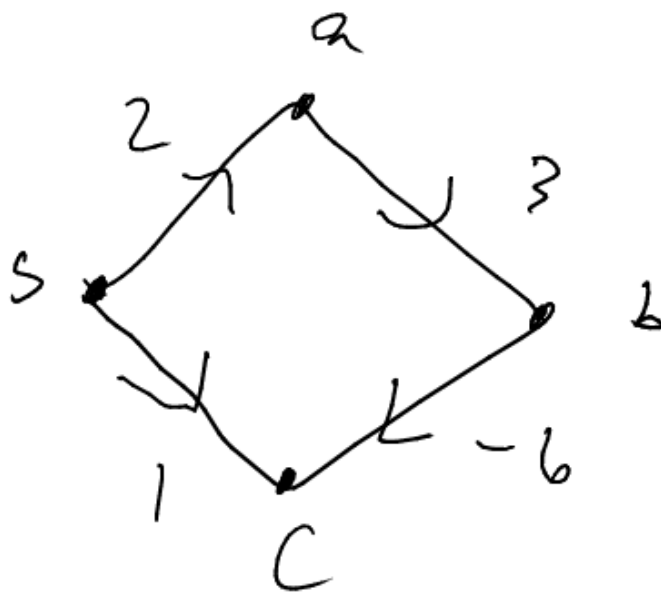
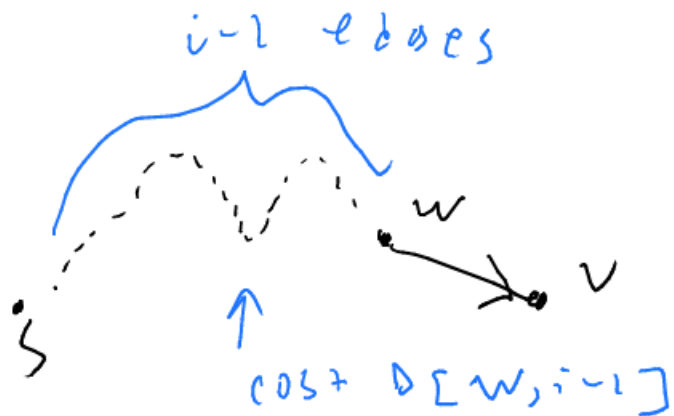
- $D[v, i]$  = minimum cost of a path from  $s$  to  $v$  that has at most  $i$  edge
- $D[v, n + 1]$  = minimum cost of a path from  $s$  to  $v$  (since a path has at most  $n - 1$  edges)

Base case

$$D[v, 0] = \begin{cases} 0 & \text{if } v = s \\ \infty & \text{if } v \neq s \end{cases}$$

Recursion

- $D[v, i] = \min_{w: wv \in E} \{c_{wv} + D[w, i-1], D[v, i-1]\}$ 
  - Use the previous  $i - 1$  edges and then take the least expensive edge going to previous vertex



|   | s | a        | b        | c        |
|---|---|----------|----------|----------|
| 0 | 0 | $\infty$ | $\infty$ | $\infty$ |
| 1 | 0 | 2        | $\infty$ | 1        |
| 2 | 0 | 2        | 5        | 1        |
| 3 | 0 | 2        | 5        | -1       |

- $D[a, 1] = \min\{D[a, 0](\infty), 2 + D[s, 0](2)\}$

- $D[b, 1] = \min D[b, 0], 3 + D[a, 0]$
- $D[c, 3] = \min\{D[c, 2](1), 1 + D[s, 2](1 + 0), -6 + D[b, 2](-6 + 5)\}$

### 22.1.1 Running Time

For every  $i$  and  $v$ ,  $D[v, i]$  costs  $O(deg^{in}v)$  where  $deg^{in}(v)$  is the number of edges pointing to  $v$ . So  $D[v, i] \forall v \in V$  but fixed  $i$  : (i.e. a row in the table)

$$O(\sum_v deg^{in}v) = O(n)$$

We are doing this for  $i = 0, 1, 2, \dots, n - 1$

- $O(nm)$

|   | s | a | b | c |
|---|---|---|---|---|
| 0 | s | x | x | x |
| 1 | s | s | x | s |
| 2 | s | s | a | s |
| 3 | s | s | a | b |

Can we compute these with less memory? Yes as we don't need to keep all  $D[v, i]$  for all  $i$ . Indeed to compute  $D[v, i]$  we only need  $D[w, i - 1]$  for all  $w$ .

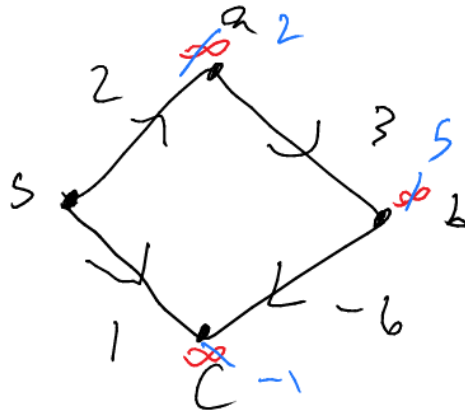
---

```

 $D[s] = 0$  and  $D[v] = \infty$  for  $v \neq s$ 
for  $i = 1, \dots, n - 1$  do
  for all  $v$  do
     $x = \min_{wv \in E} c_{wv} + D[w]$ 
    if  $x < D[v]$  then  $D[v] = x$ 
  end if
  end for
end for

```





step 0  
step 1

Local algorithm, every vertex only has to interact with their neighbors, they don't need to see the global picture.

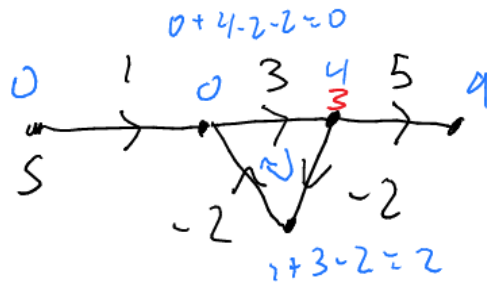
This can be used in situations where nodes only see their neighbors and not the global network

## 22.2 Finding Negative Cycles

Can we find out whether the graph has negative cycles?

- (Will be using the following in the assignment)

Without a negative cycle, the minimum should be at the end of the for loop at  $n - 1$ . If that isn't the minimum/things are still changing, we have a negative cycle.



$\bar{c} = 4$   
 $\bar{c} = 5$

What if we can't get to the cycle from  $s$ ?



### 22.2.1 Slow fix

```

for every vertex  $u$  do
  set  $s = u$ 
  Run the algorithm until  $i = n - 1$ 
  Run one more round ( $i = n$ )
  if any updates happen  $\implies$  negative cycle then
    end if
end for

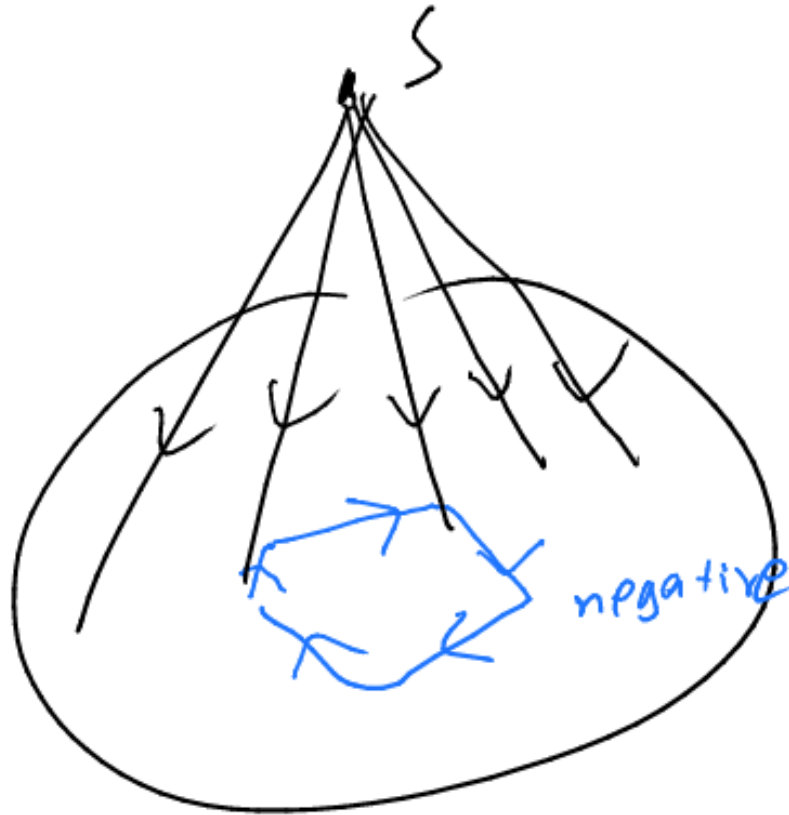
```

Running time:  $O(n \times nm)$

### 22.2.2 Faster algorithm

Add a new vertex  $s$

- Connect  $s$  to every vertex with an edge cost 0.



- This won't add a cycle

Now there is a path from  $s$  to any potential negative cycle and thus we can detect it.

- Instead of the for loop in the slow fix, we connect  $s$  to everything with cost 0 as mentioned above and then check  $n - 1$  versus  $n$ .

## 23 Lecture 23 <2017-11-28 Tue>

### 23.1 Max Flow Problem

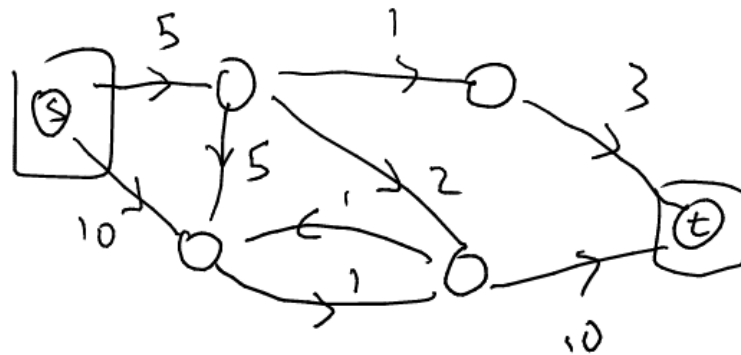
(Will be the first two lectures of 360)

- Def: A flow network is a directed graph  $G = (V, E)$  such that

1. Every edge  $e$  has some capacity  $c_e \geq 0$
2. There is a source node  $s \in V$
3. There is a sink  $t \in V$

$$(G = (V, E), s, t, \{c_e\}_{e \in E})$$

Example



Remark: To make the problem cleaner we make the following assumptions (these aren't essential assumptions, but they make things easier).

1. No edge ends in  $\underline{s}$ 
  - If any edge goes towards  $\underline{s}$ , we can just remove it, it doesn't matter for the problem
2. No edge leaves the sink  $\underline{t}$
3. All capacities are integers.
4. There is at least one edge connected to every vertex

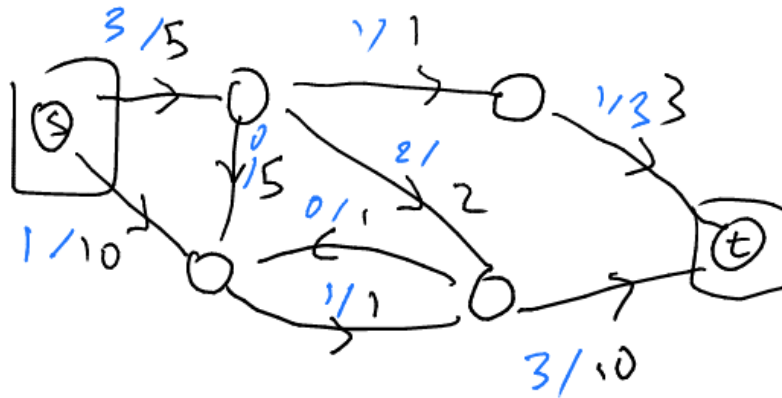
Every pipe gets some sort of traffic, want to assign a certain amount of flow per pipe.

Def: An  $s - t$  flow on this network is a function  $f : E \rightarrow \mathbb{R}^+$  (Each edge gets assigned a positive real number) satisfying

1. [capacity]  $0 \leq f(e) \leq c_e, \forall e \in E$  (don't want to exceed capacity of the edge)
2. [flow conservation] for every internal node  $v \in V$  (not source or sink)

$$\underbrace{\sum_{e \text{ into } v} f(e)}_{f^{in}(v)} = \underbrace{\sum_{e \text{ leaves } v} f(e)}_{f^{out}(v)}$$

i.e. the incoming flow is the same as the outgoing flow, no inner node generates or loses flow.

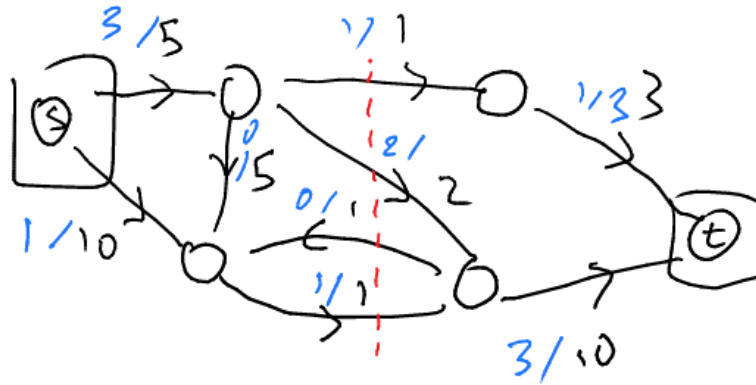


- Def: The value of a flow  $f^{out}(s)$  (amount of flow generated at source, total flow leaving source. In the above example, it is 4)

---

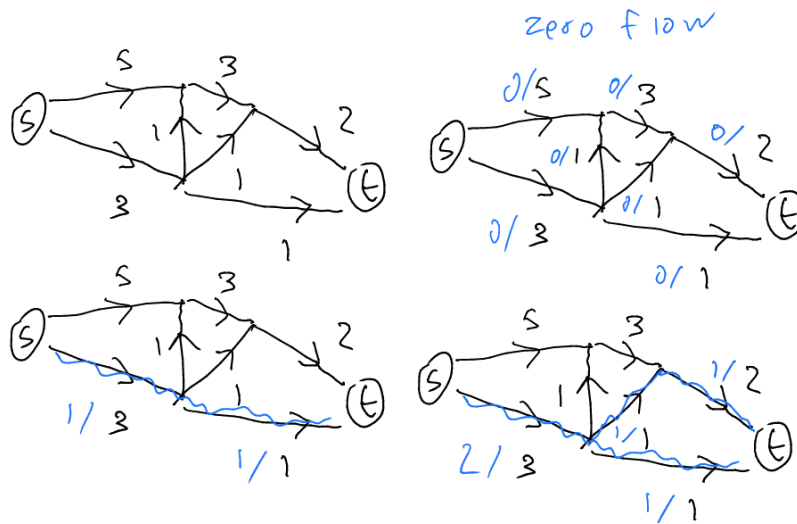
The Max Flow Problem:

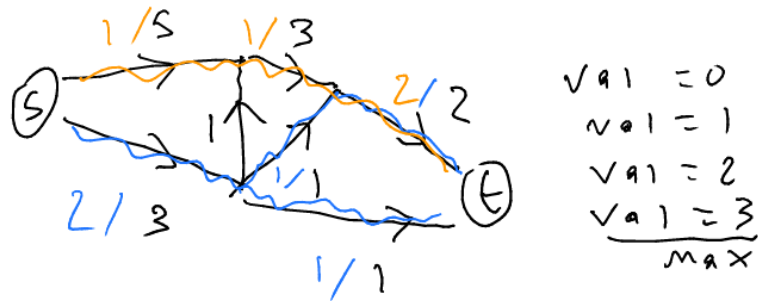
- Given a flow network  $(G, s, t, \{c_e\}_{e \in E})$
- We want to find the maximum value of an  $s - t$  - flow
- Very useful, many other problems can be modeled as flow networks, so you can use the solution of this problem for many other problems.
- In our example, we cannot do better than 4. Intuitively, we can see that we cannot pass more than 4 from the left and right side of the following dotted line. We will prove this more formally later.



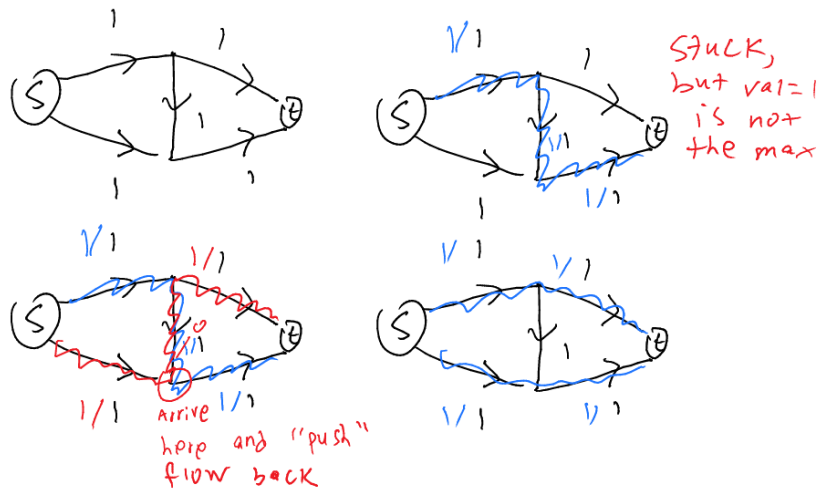
# Ford-Fulkerson Algorithm

- Basic idea: start from the all zero flow and at each step augment the flow by pushing more flow through an  $s - t$  path that has some unused capacity.





Problematic example:



- Def: [Residual graph] Given a flow network  $(G, s, t, \{c_e\}_{e \in E})$  and an  $s-t$ -flow  $f : E \rightarrow \mathbb{R}^+$  the corresponding residual graph  $G_f$  is defined as

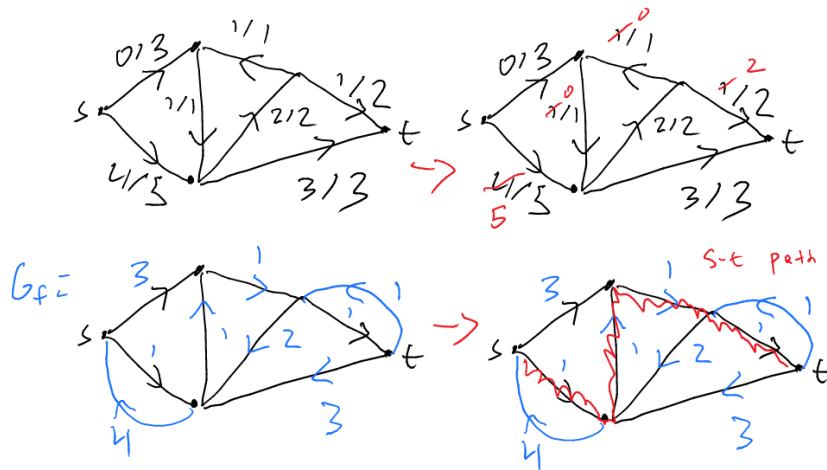
1. The nodes of  $G_f$  are the same as  $G$ .
2. For every edge  $e = (u, v)$  with  $f(e) < c_e$  we add the edge  $e$  to  $G_f$  and assign a residual capacity of  $c_e - f(e)$  (you have an edge with unused capacity, so you assign the residual capacity to it)



1. For every edge  $e = (u, v)$  with  $f(e) > 0$  then we add the opposite edge  $(v, u)$  in  $G_f$  with residual capacity  $f(e)$  (i.e. pushing back the flow)



Example.



- Def: [An Augmenting path in a residual graph] is a simple  $s-t$ -path in  $G_f$ . We defined  $\text{Bottleneck}(P, f)$  to be the minimum residual capacity of any edge on  $P$ .



- Rem:  $\text{Bottleneck}(P, f)$  is the maximum amount of new flow that can be pushed through  $P$  from  $s$  to  $t$

### 23.1.1 Ford-Fulkerson Algorithm

Initially set  $f(e) = 0, \forall e \in E$

Construct  $G_f$  (residual graph)

**while** there is an  $s - t$  path  $P$  in  $G_f$  **do**

$f' \leftarrow \text{augment}(P, f)$

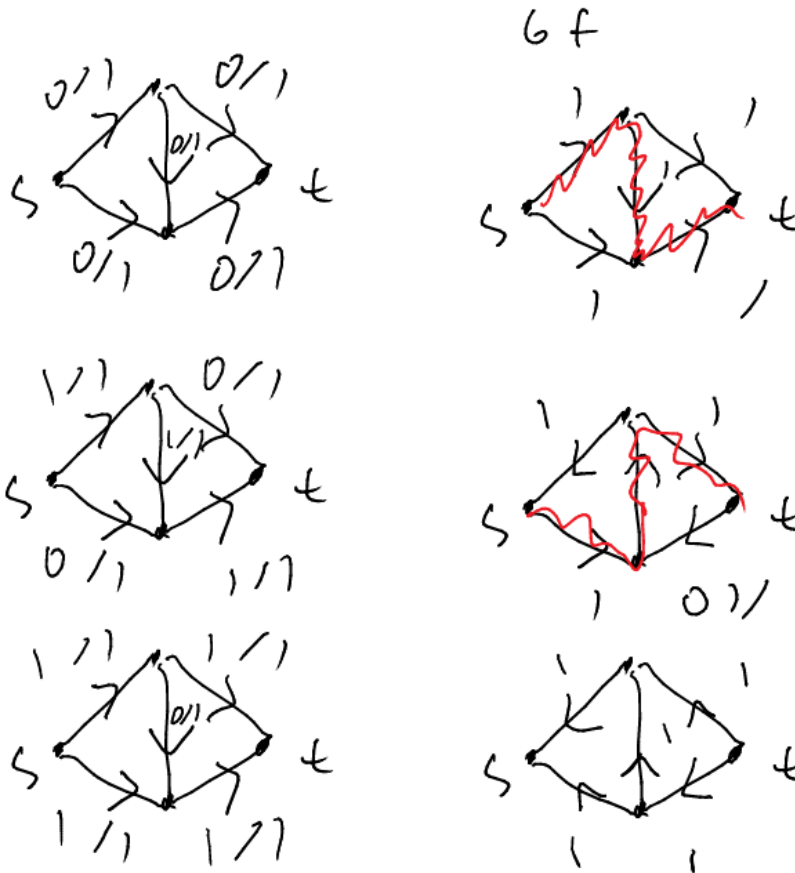
    (i.e. increase the flow by  $\text{Bottleneck}(P, f)$  using  $P$ )

    Update  $f$  to  $f'$

    Update  $G_f$

**end while**

Example.



- The  $3^{rd}G_f$  has no more  $s - t$  paths. We now have the max flow.
- But we still don't know everything. Why does this gives us the flow, let alone the max?
- Why does this terminate?
- What is the running time?
- We will answer these questions now.

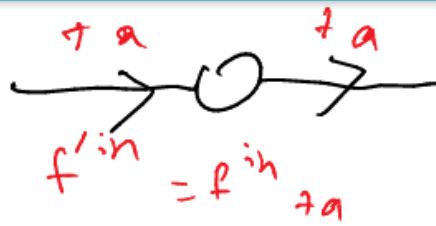
Claim: The updates result in valid flows. ( $f'$  is valid)

Proof: We need to verify that both capacity and conservation conditions are respected.

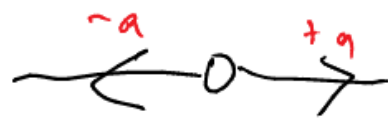
1. Capacity: The residual capacities are set so that after updating the flow we will not exceed the capacity
  - $f'(e) \leq f(e) + c_e - f(e) = c_e$

$$\overbrace{\quad\quad\quad}^{f(e)/c_e} \quad \rightarrow \quad \overbrace{\quad\quad\quad}^{c_e - f(e)}$$

2. Conservation, where  $a = \text{Bottleneck}$



$$f'in = f'in + a$$



$$f'in = f'in$$

$$f'out = f'out - a + a = f'out$$



$$f'in = f'in - a + a = f'in$$

$$f'out = f'out$$