

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 01 – Introduction

# Outline

- Course Administration
  - Break 1, Clicker Distribution (~ 6.30-6.45pm)
- Competitive Programming Book, Chapter 1
  - Competitive Programming: **Live Demo**
  - Tips to be Competitive: **Hands on ☺, join me**
  - Break 2 (~ 7.50-8.00pm)
- Mooshak: First **Mock Contest & Discussion**
  - 45 minutes contest: 3 “easy” problems

# CS3233 Lecturer History

- Initiated by **Prof Andrew Lim** (now CUHK): 1999-2001
  - Vacuum in AY 2002/03... ☹
- Between 2004-2006, CS3233 was taught by **A/P Leong Hon Wai** and **A/P Ooi Wei Tsang**
  - Another vacuum in AY 2007/08... ☹
- Revived again\* on semester 2, 2008/09 ☺
- Note: Each lecturer has different style...
  - Mine is geared towards **ICPC preparation**

# SoC Teams Performance History (1)

- ACM ICPC World Finals
  - 1999: Joint-18
  - 2000: Joint-22
  - 2001: Joint-29
  - 2003: Joint-13
  - 2005: Melvin, Junbin, Yunsong: Hon. Mention
  - 2009: Duc, Tien\*, Phong: Hon. Mention
  - 2010: Duc, Tien\*, Phong: Hon. Mention
  - 2011: Miss out by 2 ranks ☹
  - 2012: Zi Chun\*, Harta\*, Phuong\* ☺
  - 2013: You?

# SoC Teams Performance History (2)

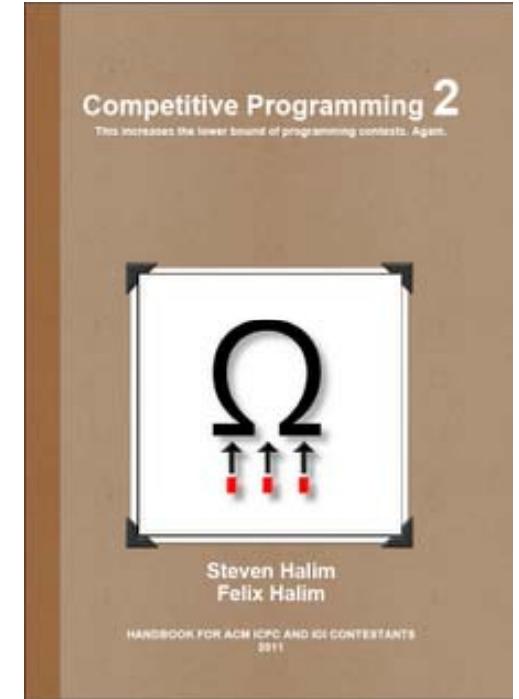
- Recent ACM ICPC Regional Contests
  - 2008: 6<sup>th</sup> in Amritapuri; 3<sup>rd</sup> in Kanpur;  
Joint-15<sup>th</sup> & Joint-16<sup>th</sup> in Kuala Lumpur
  - 2009: 7<sup>th</sup> & 10<sup>th</sup> in Jakarta; 3<sup>rd</sup> in Manila;  
2<sup>nd</sup> and 10<sup>th</sup> in Phuket
  - 2010: 10<sup>th</sup> in Daejeon; 6<sup>th</sup>++ in Kuala Lumpur; 10<sup>th</sup> in Tokyo
  - 2011: 7<sup>th</sup>++ in Phuket; 5<sup>th</sup>++ in Kuala Lumpur
  - 2012: YOUR TURN for World Finals 2013!
- More history in:
  - <http://algorithmics.comp.nus.edu.sg/wiki/>

# SoC Current Strengths

- Teaching Staffs and Seniors:
  - A/P Tan, Dr Steven, Duc, Tien\*, Phong, Felix\*, Su Zhan\*, Suhendry, Victor, Zi Chun\*, Harta\*, Phuong\*, etc
    - \* ex/current-World Finalists currently in SoC
  - Singapore IOI Teams 2010-2011
    - 2 Golds, 2 Silvers, and 4 Bronzes by this team over the past 2 years
- Current Students:
  - Many potential students (**YOU ALL**)...

# Textbook

- Competitive Programming **2**
- COMPULSORY!!
- 25 SGD/copy
- Buy tonight (15 copies are available)
- (Private) lecture material is **not** uploaded...
  - Reason: To keep NUS advantage over other universities ☺
- Public version, without the starred slides, is in:  
<http://sites.google.com/site/stevenhalim/home/material>



# Clicker Distribution (15 Mins Break)

- During the break:
  - Distribute clickers
  - Discuss administrative issues with me  
(i.e. should I take this module or not?, etc)
    - In NUS, drop with ‘W’ grade is after Week 02!
  - Buy “Competitive Programming 2” textbook

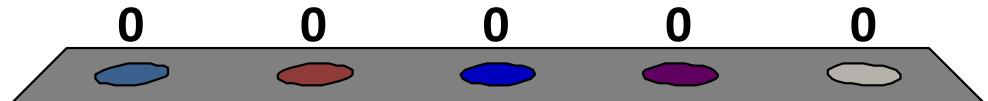


This increases the lower bound of programming contests. Again.

# **COMPETITIVE PROGRAMMING 2**

# How many of you have read CP2?

1. I have just bought it...  
So 0 page so far...
2. Only a few pages so far
3. Most of chapter 4 and a bit  
of chapter 3 due to  
CS2020/CS2010
4. I have read most of it, but I  
know you have other tricks  
not written in CP2 yet 😊
5. I have mastered CP2 **and**  
**beyond** 😊



# Competitive Programming

- Given well-known Computer Science problems, solve them as fast as possible!
  - Not about “software engineering”
- Well-known = not research problems!
  - Problems in our target contest: ACM ICPC & IOI have this characteristic!



# Demo (UVa 11849 – CD)

- This exaggerated demo illustrate contestant's type:
  - A. The blurry one
  - B. Give up
  - C. Slow
  - D. Competitive programmer
  - E. Very competitive programmer

# **TIPS TO BE COMPETITIVE**

# Tip 1: Type Fast & Correct

- No kidding, this can be important!
- Let's try
  - <http://www.typingtest.com>
    - ZEBRA – Africa's Striped Horse
    - Mine: ~85-90 wpm
    - Felix's: ~55-65 wpm
- Familiarize yourself with the positions of the following keyboard keys:
  - (,), {}, [ ], <, >, ; ; , &, |, !, etc



# Tip 2: Quickly Identify Problem Types

- Ad Hoc
- Complete Search
- Divide and Conquer
- Greedy
- Dynamic Programming
- Graph
- Mathematics
- String Processing
- Comp. Geometry
- Some Harder Ones

# Tip 3: Do Algorithm Analysis

- This is taught in more details in CS3230!
- In this module, we will just learn the basics required for dealing with ICPC/IOI problems
  - See the constraints in the problem statement
  - Conjure the simplest algorithm that works!
  - Do some basic analysis to convince that it will work *before* we start coding...

# Tip 4: Master Programming Languages

- You should master at least one **(preferably more)** programming languages
  - Reduce the amount of time looking at references
  - Use shortcuts, macros, avoid comments
  - Use libraries whenever possible
- Idea: Once you figure out a solution for a problem, you are able to translate it into a bug-free code, and do it fast!

# Tip 5: Master the Art of Testing Code

- Ultimately, we want “Accepted (AC)” verdict 😊
  - i.e. Our code passes the judge’s secret test data
- However, we may instead be given: 😞
  - Presentation Error (PE)
  - Wrong Answer (WA)
  - Time Limit Exceeded (TLE)
  - Memory Limit Exceeded (MLE)
  - Runtime Error (RTE)

# Tip 6: Practice and More Practice

- Online Judges for CS3233
  - MAIN: University of Valladolid (UVa) Online Judge
    - <http://uva.onlinejudge.org> (Open with Firefox!)
  - MISC: ACM ICPC Live Archive
    - <http://livearchive.onlinejudge.org/>
  - MISC: TopCoder
    - <http://www.topcoder.com>
  - MISC: USACO
    - <http://train.usaco.org>



# We will mainly use UVa online judge this semester. I have...

1. Registered a free account but have not solve anything
2. Have solved  $\geq 10$  UVa problems
3. Have solved  $\geq 40$  UVa problems
4. Have solved  $\geq 100$  UVa problems



# Tip 7: Team Work (ICPC Only)

- Practice coding on a blank paper
- Submit and print strategy
- Prepare test data challenges
- The X-factor

Let's start

# **THE AD HOC PROBLEMS**

# Ad Hoc Problems (1)

- Definition from USACO + modifications
  - 'Ad Hoc' problems are those whose algorithms do not fall into standard categories with **well-studied solutions**
  - Each Ad Hoc problem is **different**...
    - No specific or general techniques exist to solve them
  - This makes the problems the 'fun' ones (and sometimes frustrating), since each one presents a new challenge
  - The solutions might require a novel data structure or an unusual set of loops or conditionals

# Ad Hoc Problems (2)

- Definition from USACO + modifications (Cont)
  - Sometimes they require special combinations that are rare or at least rarely encountered
  - It usually require careful problem description reading and usually yield to an attack that revolves around carefully sequencing the instructions given in the problem
  - Ad Hoc problems can still require reasonable optimizations and at least a degree of analysis that enables one to avoid loops nested five deep, for example

# Ad Hoc Problems (3)

- Ad Hoc problems usually appear in ACM ICPC problem set (1 or 2 per set)
  - Can be of “general type” (listed in Chapter 1), using simple Data Structures (Chapter 2), Mathematical (Chapter 5), String-related (Chapter 6), or Basic Geometry (Chapter 7)
- Sadly, solving only Ad Hoc problems will not give your team a good result in ICPC...
  - We will learn more problem types in subsequent classes

# Ad Hoc Problems (4)

- We further sub-divide Ad Hoc problems into the following sub-categories:
  - Super Easy; Easy; Medium
  - Game (Card); Game (Chess); Game (Others), Easier; Game (Others), Harder (more tedious)
  - Josephus; Palindrome; Anagram
  - Interesting Real Life Problems, Easier; Interesting Real Life Problems, Harder;
  - Time; ‘Time Waster’
  - Just Ad Hoc
  - Ad Hoc problems in other Chapters (2, 5, 6, 7)

# Next Week

- **CH2: Data Structures and Libraries**
  - Focus on bit manipulation and  
Binary Indexed (Fenwick) Tree

# 10 Minutes Break

- In the last part of our first introductory class, you will familiarize yourself with **Linux controlled environment** in this PL6 and the **Mooshak system**, the internal online judge used for CS3233 this semester
  - Mock contest containing 3 “simple problems”
  - Not graded yet ☺, enjoy it for fun  
(and to help you decide if you should take CS3233)

# Mooshak

- Let's try this system
  - Open with **Firefox**  
(does not work well with most other browsers ☹):
    - <http://algorithmics.comp.nus.edu.sg>
  - Click “Online Judge (Login)” at the top left corner

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 02 – Data Structures & Libraries

Focus on Bit Manipulation & Binary Indexed Tree

# Outline

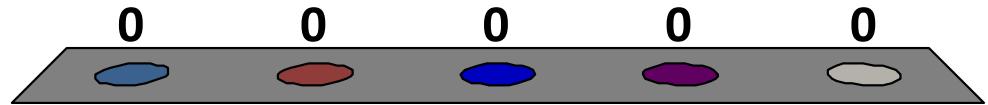
- Mini Contest 1 + Break (discussion of A/B)
- Some Admins
- Data Structures With Built-in Libraries
  - Just a quick walkthrough
    - Read/experiment with the details on your own
  - Linear Data Structures (CS1010/1<sup>st</sup> half of CS2020)
  - Non Linear Data Structures (CS2010/2<sup>nd</sup> half of CS2020)
    - Focus on the **red highlights**
- “Top Coder” Coding Style (overview) + Break
- Data Structures With Our-Own Libraries
  - Focus on Binary Indexed (Fenwick) Tree

Basic knowledge that all ICPC/IOI-ers must have!

# **LINEAR DATA STRUCTURES WITH BUILT-IN LIBRARIES**

# I am...

1. A pure C coder
2. A pure C++ coder
3. A mix between  
C/C++ coder
4. A pure Java coder
5. A multilingual  
coder: C/C++/Java



# Linear DS + Built-In Libraries (1)

1. Static Array, built-in support in C/C++/Java
  2. Resize-able: C++ STL **vector**, Java **Vector**
    - Both are very useful in ICPCs/IOIs
- 
- There are 2 very common operations on Array:
    - Sorting
    - Searching
    - Let's take a look at efficient ways to do them

Two “fundamental” CS problems

# **SORTING + SEARCHING INVOLVING ARRAY**

# Sorting (1)

- Definition:
  - Given unsorted stuffs, sort them... \*
- Popular Sorting Algorithms
  - $O(n^2)$  algorithms: Bubble/Selection/Insertion Sort
  - $O(n \log n)$  algorithms: Merge/Quick<sup>^</sup>/Heap Sort
  - Special purpose: Counting/Radix/Bucket Sort
- Reference:
  - [http://en.wikipedia.org/wiki/Sorting\\_algorithm](http://en.wikipedia.org/wiki/Sorting_algorithm)

# Sorting (2)

- In ICPC, you can “forget” all these...
  - In general, if you need to sort something..., just use the  $O(n \log n)$  sorting library:
    - C++ STL **algorithm:: sort**
    - Java **Collections.sort**
- In ICPC, sorting is either used as *preliminary step* for more complex algorithm or to *beautify output*
  - Familiarity with sorting libraries is a must!

# Sorting (3)

- Sorting routines in C++ STL algorithm
  - sort – a bug-free implementation of *introsort*\*
    - Fast, it runs in  $O(n \log n)$
    - Can sort basic data types (ints, doubles, chars), **Abstract Data Types (C++ class)**, **multi-field sorting ( $\geq 2$  criteria)**
  - partial\_sort – implementation of *heapsort*
    - Can do  $O(k \log n)$  sorting, if we just need top-k sorted!
  - stable\_sort
    - If you need to have the sorting ‘stable’, keys with same values appear in the same order as in input

# Searching in Array

- Two variants:
  - When the array is sorted versus not sorted
- Must do  $O(n)$  linear scan if not sorted - trivial
- Can use  $O(\log n)$  binary search when sorted
  - PS: must run an  $O(n \log n)$  sorting algorithm once
- Binary search is ‘tricky’ to code!
  - Instead, use C++ STL **algorithm::lower\_bound**

# Linear DS + Built-In Libraries (2)

## 3. Array of Boolean: C++ STL **bitset**

- Faster than **array of bools** or **vector<bool>**!
- No specific API in Java that is similar to this

## 4. Bitmask

- **a.k.a. lightweight set of Boolean or bit string**
- **Explanation via:**

<http://www.comp.nus.edu.sg/~stevenha/visualization/bitmask.html>



# Linear DS + Built-In Libraries (3)

## 5. Linked List, C++ STL **list**, Java **LinkedList**

- Usually not used in ICPCs/IOIs
- If you need a resizeable “list”, just use **vector**!

## 6. Stack, C++ STL **stack**, Java **Stack**

- Used by default in Recursion, Postfix Calculation, Bracket Matching, etc

## 7. Queue, C++ STL **queue**, Java **Queue**

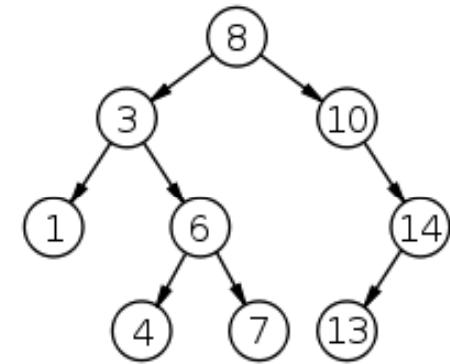
- Used in Breadth First Search, Topological Sort, etc
- **PS: Deque, used in ‘Sliding Window’ algorithm**



More efficient data structures

# **NON-LINEAR DATA STRUCTURES WITH BUILT-IN LIBRARIES**

# Binary Search Tree (1)

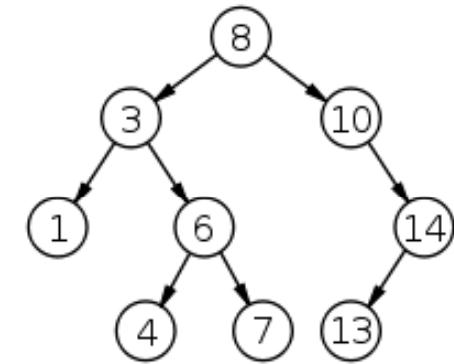


A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

- ADT Table (key → data)
- Binary Search Tree (BST)
  - Advertised  $O(\log n)$  for insert, search, and delete
  - Requirement: the BST must be **balanced!**
    - AVL tree, Red-Black Tree, etc... \*argh\*
- Fret not, just use: C++ STL **map** (Java **TreeMap**)
  - UVa [10226](#) (Hardwood Species)\*

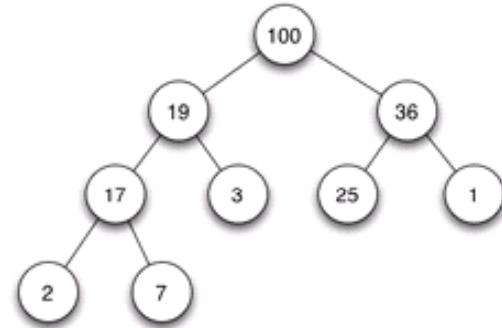
# Binary Search Tree (2)

- ADT Table (key exists or not)
- Set (Single Set)
  - C++ STL **set**, similar to C++ STL **map**
    - map stores a **(key, data)** pair
    - set stores just the **key**
  - In Java: **TreeSet**
- Example:
  - UVa [11849](#) – CD



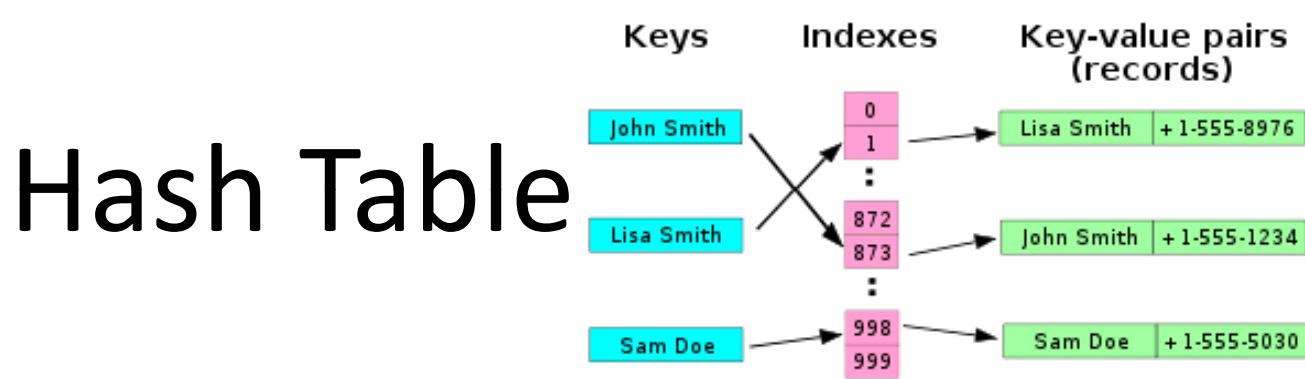
A binary search tree of size 9 and depth 3, with root 8 and leaves 1, 4, 7 and 13

# Heap



Example of a full binary max heap

- **Heap**
  - C++ STL **algorithm** has some heap algorithms
    - `partial_sort` uses `heapsort`
  - C++ STL **priority\_queue** (Java **PriorityQueue**) is heap
    - Prim's and Dijkstra's algorithms use priority queue
- But, we rarely see pure heap problems in ICPC



A small phone book as a hash table.

- Hash Table
  - Advertised  $O(1)$  for insert, search, and delete, but:
    - The hash function must be good!
    - There is no Hash Table in C++ STL ( $\exists$  in Java API)
  - Nevertheless,  $O(\log n)$  using **map** is usually ok
- Direct Addressing Table (DAT)
  - Rather than hashing, we more frequently use DAT
  - UVa [11340](#) (Newspaper)

Top Coder Coding Style

# **SUPPLEMENTARY**

# Top Coder Coding Style (1)

- You may want to follow this coding style (C++)

## 1. Include **important** headers ☺

- #include <algorithm>
- #include <cmath>
- #include <cstdio>
- #include <cstring>
- #include <iostream>
- #include <map>
- #include <queue>
- #include <set>
- #include <string>
- #include <vector>
- using namespace std;

Want More?

Add libraries that you frequently use into this template, e.g.:

ctype.h  
bitset

etc

# Top Coder Coding Style (2)

## 2. Use shortcuts for common data types

- `typedef long long ll;`
- `typedef vector<int> vi;`
- `typedef pair<int, int> ii;`
- `typedef vector<ii> vii;`

## 3. Simplify Repetitions/Loops!

- `#define REP(i, a, b) for (int i = int(a); i <= int(b); i++)`
- `#define REPN(i, n) REP (i, 1, int(n))`
- `#define REPD(i, a, b) for (int i = int(a); i >= int(b); i--)`
- `#define TRvi(c, it) \  
for (vi::iterator it = (c).begin(); it != (c).end(); it++)`
- `#define TRvii(c, it) \  
for (vii::iterator it = (c).begin(); it != (c).end(); it++)`

Define your own loops  
style and stick with it!

# Top Coder Coding Style (3)

## 4. More shortcuts

- `for (i = ans = 0; i < n; i++)...` // do variable assignment in for loop
- `while (scanf("%d", n), n) { ...}` // read input + do value test together
- `while (scanf("%d", n) != EOF) { ...}` // read input and do EOF test

## 5. STL/Libraries all the way!

- `isalpha (ctype.h)`
  - `inline bool isletter(char c) { return (c>='A'&&c<='Z') || (c>='a'&&c<='z'); }`
- `abs (math.h)`
  - `inline int abs(int a) { return a >= 0 ? a : -a; }`
- `pow (math.h)`
  - `int power(int a, int b) { int res=1; for (; b>=1; b--) res*=a; return res; }`
- Use STL data structures: `vector`, `stack`, `queue`, `priority_queue`, `map`, `set`, etc
- Use STL algorithms: `sort`, `lower_bound`, `max`, `min`, `max_element`, `next_permutation`, etc

# Top Coder Coding Style (4)

## 6. Use I/O Redirection

- int main() {
- // freopen("input.txt", "r", stdin); // don't retype test cases!
- // freopen("output.txt", "w", stdout);
- scanf and printf as per normal; // I prefer scanf/printf than  
// cin/cout, C style is much easier

## 7. Use memset/assign/constructor effectively!

- memset(dist, 127, sizeof(dist));  
// useful to initialize shortest path distances, set INF to 127!
- memset(dp\_memo, -1, sizeof(dp\_memo));  
// useful to initialize DP memoization table
- memset(arr, 0, sizeof(arr)); // useful to clear array of integers
- vector<int> dist(v, 2000000000);
- dist.assign(v, -1);

# Top Coder Coding Style (5)

## 8. Declare (large) static DS as global variable

- All input size is known, declare data structure size **LARGER** than needed to avoid silly bugs
  - Avoid dynamic data structures that involve pointers, etc
  - Use global variable to reduce “stack size” issue
- 
- Now our coding tasks are much simpler ☺
  - Typing less code = shorter coding time  
= better rank in programming contests ☺

# Quick Check

1. I can cope with this pace...
2. I am lost with so many new information in the past few slides



# 5 Minutes Break

- One data structures *without* built-in libraries will be discussed in the last part...
  - Binary Indexed (Fenwick) Tree
  - Graph, Union-Find Disjoint Sets, and Segment Tree are not discussed in this year's CS3233 Week02
    - Graph DS is covered in details in CS2010/CS2020
    - UFDS is covered briefly in CS2010/CS2020
    - Please study Segment Tree on your own
      - We try not set any contest problem involving Segment Tree

[Graph](#) (not discussed today, revisited in Week05/08)

[Union-Find Disjoint Sets](#) (not discussed today, read Ch2 on your own)

[Segment Tree](#) (not discussed today, read Ch2 on your own)

[Fenwick Tree](#) (discussed today)

# **DATA STRUCTURES WITHOUT BUILT-IN LIBRARIES**

# Fenwick Tree (1)

- Cumulative Frequency Table
  - Example,  $s = \{2, 4, 5, 5, 6, 6, 6, 7, 7, 8\}$  (already sorted)

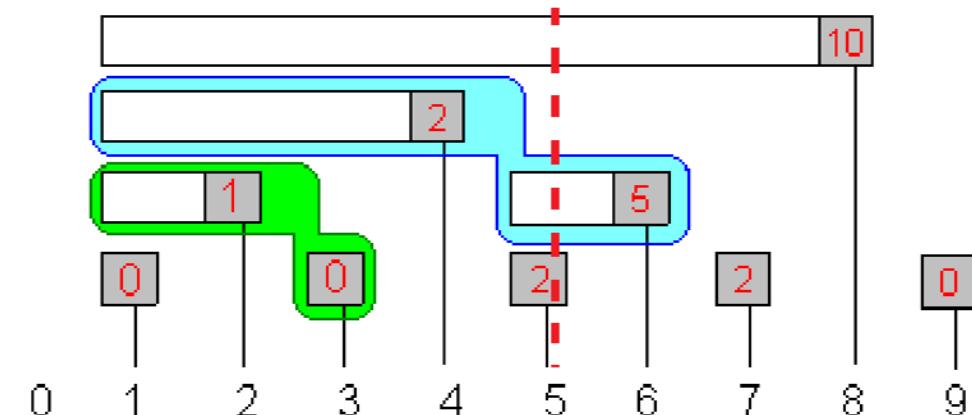
Index/Score/Symbol	Frequency	Cumulative Frequency
0	0	0
1	0	0
2	1	1
3	0	1
4	1	2
5	2	
6	3	
7	2	
8	1	

# Fenwick Tree (2)

- Fenwick Tree (inventor = Peter M. Fenwick)
  - Also known as “**Binary Indexed Tree**”, very *aptly* named
  - Implemented as an **array**, let call the array name as **ft**
    - Each **index** of **ft** is responsible for certain **range** (see diagram)

Key/Index	Binary	Range	F	CF	FT
0	0000	N/A	N/A	N/A	N/A
1	0001	1	0	0	0
2	0010	1..2	1	1	1
3	0011	3	0	1	0
4	0100	1..4	1	2	2
5	0101	5	2	4	2
6	0110	5..6	3	7	5
7	0111	7	2	9	2
8	1000	1..8	1	10	10
9	1001	9	0	10	0

Do you notice  
any particular pattern?



# Fenwick Tree (3)

- To get the cumulative frequency from index 1 to b, use `ft_rsq(ft, b)`
  - The answer is the sum of sub-frequencies stored in array `ft` with indices related to b via this formula  $b' = b - \text{LSOne}(b)$ 
    - Recall that  $\text{LSOne}(b) = b \& (-b)$ 
      - » That is, strip **the least significant bit** of b
  - Apply this formula iteratively until b is 0

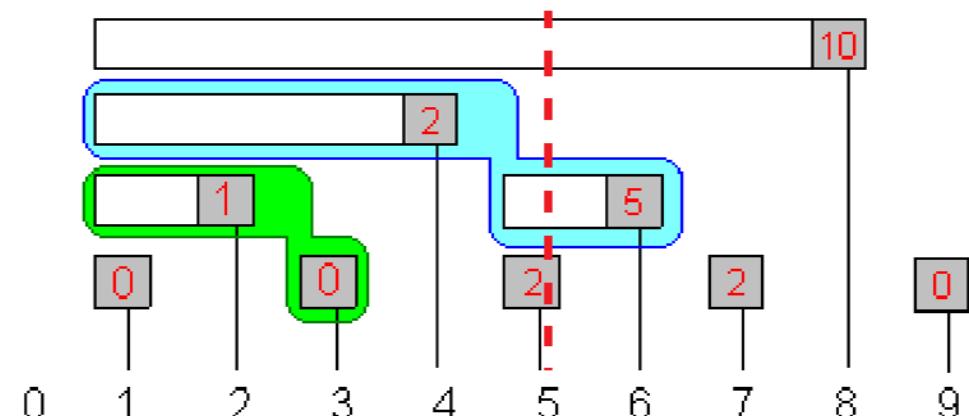
- Example: `ft_rsq(ft, 6)`

»  $b = 6 = 0110$ ,  $b' = b - \text{LSOne}(b) = 0110 - 0010$ ,  $b' = 4 = 0100$

»  $b' = 4 = 0100$ ,  $b'' = b' - \text{LSOne}(b') = 0100 - 0100$ ,  $b'' = 0$ , **stop**

- Sum  $\text{ft}[6] + \text{ft}[4] = 5 + 2 = 7$

(see the blue area  
that covers range  
[1..4] + [5..6] = [1..6])



Analysis:  
This is  
 $O(\log n)$

Why?

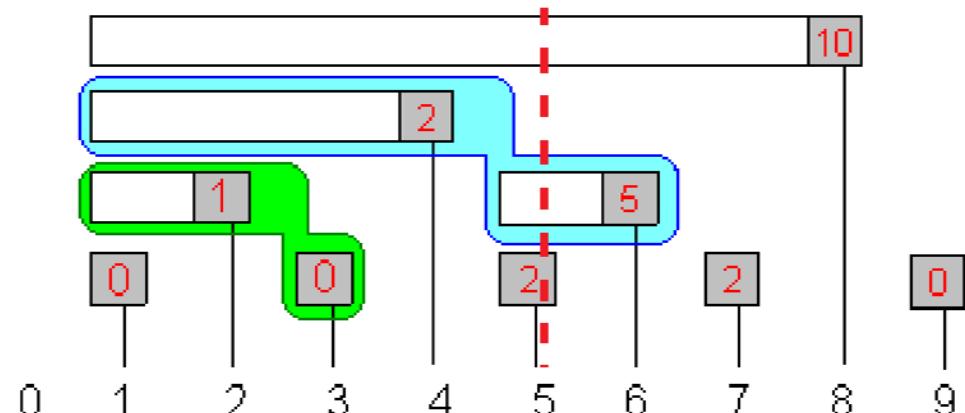
# Fenwick Tree (4)

- To get the cumulative frequency from index a to b, use `ft_rsq(ft, a, b)`
  - If a is **not one**, we can use:  
`ft_rsq(ft, b) - ft_rsq(ft, a - 1)` to get the answer

Analysis:  
This is  
 $O(2 \log n) =$   
 $O(\log n)$

Why?

- Example: `ft_rsq(ft, 3, 6) =`  
`ft_rsq(ft, 6) - ft_rsq(ft, 3 - 1) =`  
`ft_rsq(ft, 6) - ft_rsq(ft, 2) =`  
**blue area** minus **green area** =  
**(5 + 2) - (0 + 1) =**  
**7 - 1 = 6**



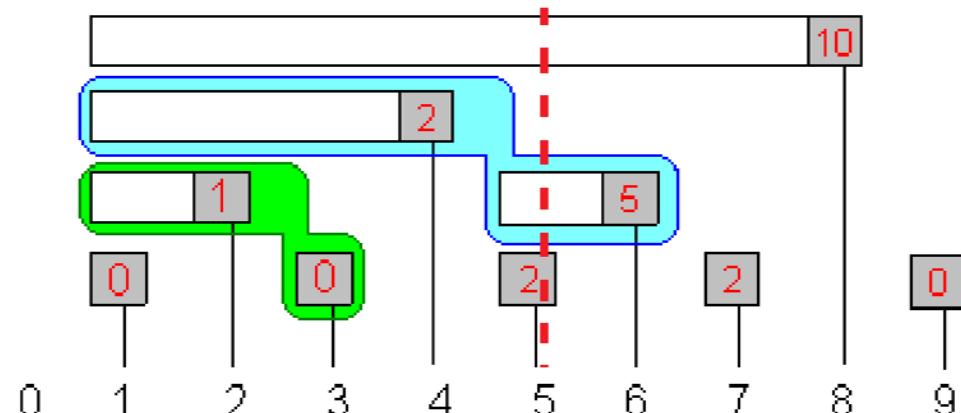
# Fenwick Tree (5)

- To update the frequency of an key/index  $k$ , by  $v$  (either positive or negative), use `ft_adjust(ft, k, v)`
  - Indices that are related to  $k$  via  $k' = k + \text{LSOne}(k)$  will be updated by  $v$  when  $k < \text{ft.size()}$ 
    - Example: `ft_adjust(ft, 5, 2)`
      - »  $k = 5 = 0101$ ,  $k' = k + \text{LSOne}(k) = 0101 + 0001 = 0110$
      - »  $k' = 6 = 0110$ ,  $k'' = k' + \text{LSOne}(k') = 0110 + 0010 = 1000$
      - » And so on while  $k < \text{ft.size()}$

Analysis:  
This is also  
 $O(\log n)$

Why?

- Observe that the **dotted red line** in the figure below **stabs through** the ranges that are under the responsibility of indices 5, 6, and 8
  - $\text{ft}[5]$ , 2 updated to 4
  - $\text{ft}[6]$ , 5 updated to 7
  - $\text{ft}[8]$ , 10 updated to 12



# Fenwick Tree (6) – Library

```
typedef vector<int> vi;
#define LSOne(S) (S & (-S))

void ft_create(vi &ft, int n) { ft.assign(n + 1, 0); } // init: n+1 zeroes

int ft_rsq(const vi &ft, int b) { // returns RSQ(1, b)
    int sum = 0; for (; b; b -= LSOne(b)) sum += ft[b];
    return sum; }

int ft_rsq(const vi &t, int a, int b) { // returns RSQ(a, b)
    return ft_rsq(t, b) - (a == 1 ? 0 : ft_rsq(t, a - 1)); }

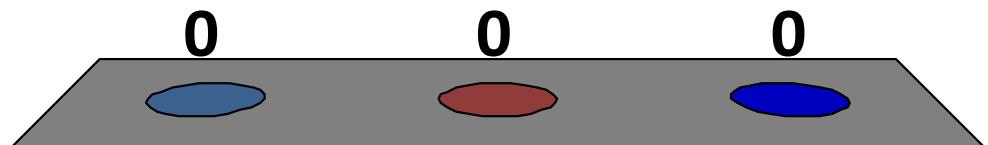
// adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
void ft_adjust(vi &ft, int k, int v) {
    for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
```

# Fenwick Tree (7) – Application

- Fenwick Tree is very suitable for *dynamic* RSQs (cumulative frequency table) where each update occurs on a certain index only
- Now, think of potential real-life applications!
  - <http://uhunt.felix-halim.net/id/32900>
  - Consider code running time of [0.000 - 9.999] for a particular UVa problem
    - There are up to 9+ million submissions/codes
      - About thousands submissions per problem
    - If your code runs in 0.342 secs, what is your rank?
- How to use Fenwick Tree to deal with this problem? 

# Quick Check

1. I am lost with Fenwick Tree
2. I understand the basics of Fenwick Tree, but since this is new for me, I may/may not be able to recognize problems solvable with FT
3. I have solved several FT-related problems before



# Summary

- There are a lot of great Data Structures out there
  - We need the most efficient one for our problem
    - Different DS suits different problem!
- Many of them have **built-in libraries**
  - For some others, we have to build **our own (focus on FT)**
    - Study these libraries! Do not rebuild them during contests!
- From Week03 onwards and future ICPCs/IOIs,  
use C++ STL and/or Java API and our built-in libraries!
  - Now, your team should be in rank 30-45 (from 60)  
(still solving ~1-2 problems out of 10, but faster)

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 03 – Problem Solving Paradigms  
(Complete Search; Divide & Conquer; Greedy)

# Outline

- Mini Contest 2 + Break + Discuss Last Week's C, Today's A/B
- Admins
- Complete Search
  - Iterative: (Nested) Loops, Permutations, Subsets
  - Recursive Backtracking
  - Some Tips
- Greedy Algorithm (we will discuss this first before D&C)
  - Ingredients and some examples
- Divide and Conquer (D&C)
  - Focus on **Binary Search the Answer**

Iterative: (Nested) Loops, Permutations, Subsets

Recursive Backtracking

Some Tips

# **COMPLETE SEARCH**

# Iterative Complete Search (1)

- UVa 725 – Division
  - Find two 5-digits number s.t.  $\rightarrow \text{abcde} / \text{fghij} = N$
  - **abcdefgij** must be all different,  $2 \leq N \leq 79$
- Iterative Complete Search Solution (Nested Loops):
  - Try all possible **fghij** (one loop)
  - Obtain **abcde** from **fghij \* N**
  - Check if **abcdefgij** are all different (*another* loop)

# Iterative Complete Search (2)

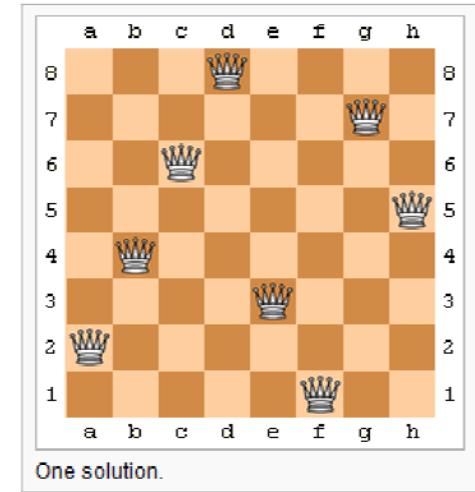
- UVa 11742 – Social Constraints
  - There are  $0 < n \leq 8$  movie goers
  - They will sit in the front row with  $n$  consecutive open seats
  - There are  $0 \leq m \leq 20$  seating constraints among them, i.e. **a** and **b** must be at most (or at least) **c** seats apart
  - How many possible seating arrangements are there?
- Iterative Complete Search Solution (Permutations):
  - Set counter = 0 and then try all possible  **$n!$  permutations**
  - Increase counter if a permutation satisfies all **m** constraints
  - Output the final value of counter

# Iterative Complete Search (3)

- UVa 12346 – Water Gate Management
  - A dam has  $1 \leq n \leq 20$  water gates to let out water when necessary, each water gate has **flow rate** and **damage cost**
  - Your task is to manage the opening of the water gates in order to get rid of *at least* the specified **total flow rate** condition that the **total damage cost** is minimized!
- Iterative Complete Search Solution (Subsets):
  - Try all possible  $2^n$  subsets of water gates to be opened
  - For each subset, check if it has sufficient flow rate
    - If it is, check if the total damage cost of this subset is smaller than the overall minimum damage cost so far
      - If it is, update the overall minimum damage cost so far
  - Output the minimum damage cost

# Recursive Backtracking (1)

- UVa 750 – 8 Queens Chess Problem
  - Put 8 queens in 8x8 Chessboard
  - No queen can attack other queens
- Naïve way (Time Limit Exceeded)
  - Generate  $8^8 = 17M$  possible 8 queens position and filter the infeasible ones...



# Recursive Backtracking (2)

- Better way, recursive backtracking
  - Insight 1: **all-different constraint** for row (or col)
    - Search space goes down from  $8^8 = 17M$  to  $8! = 40K!$
  - Insight 2: diagonal check
    - Another way to prune the search space
    - Queen A ( $i, j$ ) attacks Queen B ( $k, l$ ) iff
$$\text{abs}(i - k) == \text{abs}(j - l)$$
- Scrutinize the sample code of recursive backtracking!

# Tips

1. Filtering versus Generating
2. Prune Infeasible Search Space Early
3. Utilize Symmetries
4. Pre-Computation a.k.a. Pre-Calculation
5. Try Solving the Problem Backwards
6. Optimizing Your Source Code
7. Use Better Data Structure & Algorithm ☺

# More Search Algorithms...

- Depth Limited Search
- Iterative Depth Limited Search
- A\*
- Iterative Deepening A\* (IDA\*)
- Memory Bounded A\*
- Branch and Bound (BnB)
- Maybe in Week12 ☺...

Ingredients and Non Classical Example

# **GREEDY ALGORITHM**

# Coin Change (Special Case)

- Given many coins of different denominations = {25, 10, 5, 1} cents, make X cents with the *least number of coins* used!
  - If  $X = 42$ , solution:  $25 + \underline{10 + \textcolor{red}{5+1+1}} = 42$  (5 coins)
  - If  $X = 17$ , solution:  $\underline{10 + \textcolor{red}{5+1+1}} = 17$  (4 coins)
  - If  $X = 7$ , solution:  $\textcolor{red}{5+1+1} = 7$  (3 coins)
- Observe “optimal sub-structure” and “Greedy property”!

# Greedy Algorithm – Basic Idea

- Always make **locally optimal** choice that looks best at the moment, hoping that this choice will lead to a **globally optimal** solution
- Does not always work, but sometimes it does
  - When it does, you have a very efficient solution that can be coded quickly

# Coin Change (General Case)

- Given many coins of different denominations =  $\{4, 3, 1\}$  cents, make  $X$  cents with the *least number of coins* used!
  - If  $X = 6$ , greedy solution:  $4+1+1 = 6$  (3 coins)
  - But the optimal solution is:  $3+3 = 6$  (2 coins)
- This problem will be revisited in “Dynamic Programming” topic

# Few Classic Greedy Problems

- Coin Change \*certain denominations only\*
- Fractional Knapsack
- Load Balancing
- Greedy Assignment
- Interval Covering/Activity Selection
- Huffman's Coding
- Minimum Spanning Tree
- +ve Weighted Single-Source Shortest Paths

# Designing Greedy Algorithm (5)

- Tips (Not rule of thumb!)
  - In ICPC, if the input size is too large\* for Complete Search or even DP, Greedy may be the (only) way
    - But if the input size is small, avoid greedy!  
Try complete search or DP first!
      - If our Complete Search or DP always select one particular (best) sub problem at each step, greedy strategy may work
  - If no obvious greedy strategy seen, try to sort the data first or introduce some tweaks
    - See if greedy strategy emerges

# Greedy in Other Algorithms

- Greedy strategy is embedded in more complex algorithms:
  - +ve Weighted Single-Source Shortest Path
    - Dijkstra's
  - Minimum Spanning Tree
    - Kruskal's
    - Prim's
  - Combined with binary search the answer
    - See the next section
  - Etc

Interesting Usages of Binary Search

# DIVIDE AND CONQUER

# Divide and Conquer Paradigm

- Idea:
  - Divide: Split original problem into sub-problems
  - Conquer: Solve sub-problems and then use some (or all) the results of sub-problems to solve the original problem
- Data Structures/Algorithms with D&C flavor:
  - Merge/Quick/Heap Sort, findKthIndex, Binary Search, BST/Heap/Segment/Fenwick Tree, etc
- See if your problem can be solved using D&C!

# Focus on Binary Search the Answer

- Ordinary Usage
  - On Static Sorted Array
    - C++ STL algorithm::lower\_bound (Java Collections.binarySearch)
- Not so common (read textbook)
  - On weighted tree with root-to-leaf path sorted!
- Bisection (Numerical) Method (read textbook)
  - Find root  $x$  of a function s.t.  $f(x) = 0$ 
    - Let computer do the work
- Binary Search the Answer (discussed next)

# Binary Search the Answer (1)

- UVa 714 – Copying Books
  - Given  $m$  books with different pages  $p_i \{p_1, p_2, \dots, p_m\}$
  - Make a copy of each book with  $k$  scribes
  - Each book can only be assigned to a single scribe
  - Every scribe must get a continuous sequence of books, i.e.  
$$0 = b_0 < b_1 < b_2 < \dots < b_{k-1} < b_k = m$$
  - Minimize the maximum number of pages assigned to a scribe
  - $1 \leq k, m \leq 500, 1 \leq p_i \leq 10000000$
- Example:
  - $m = 9, k = 3, p = \{100, 200, 300, 400, 500, 600, 700, 800, 900\}$
  - Answer: 1700 pages: 100, 200, 300, 400, 500 | 600, 700 | 800, 900

# Binary Search the Answer (2)

- Can we “guess” this answer in binary search fashion?
  - Yes
  - If we guess answer = 1000 pages, we have no solution
    - Greedy assignment!
      - Give as many consecutive books to current scribe until “overflow”
    - $100,200,300,400|500|600|\{700,800,900\}$  (unassigned)
    - Answer is too low, we have to increase the workload per each scriber
  - If we guess answer = 2000 pages, we have a solution + slack
    - $100,200,300,400,500|600,700|800,900$  (slack 500|700|300)
    - Answer is too high, we can decrease the workload per each scriber
  - And so on until we hit answer = 1700 pages (optimal)

# Summary

- We have seen:
  - Complete Search: Iterative and Recursive Backtracking
  - Greedy
  - Divide and Conquer: Focus on Binary Search the Answer
- Next week, we will see (revisit) the fourth paradigm:
  - Dynamic Programming

# References

- **Competitive Programming**, chapter 3
  - Steven, Felix ☺
- **Introduction to Algorithms**, p370-404
  - Cormen, Leiserson, Rivest, Stein
- **Algorithm Design**, p115-208
  - Tardos, Kleinberg
- **Algorithms**, p127-155
  - Dasgupta, Papadimitriou, Vazirani

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 04 – Problem Solving Paradigms  
(Dynamic Programming 1)

# Outline

- Mini Contest 3 + Break + Discussion
- Admins
- Dynamic Programming – Introduction
  - Treat this as **revision** for ex CS2010/CS2020 students
  - **Listen carefully** for other group of students!
- Dynamic Programming
  - Some Classical Examples
- PS: I will use the term **DP** in this lecture
  - OOT: DP is NOT Down Payment!

Wedding Shopping

# **EXAMPLE 1**

# Motivation

- How to solve UVa [11450](#) (Wedding Shopping)?
  - Given  $1 \leq C \leq 20$  classes of garments
    - e.g. shirt, belt, shoe
  - Given  $1 \leq K \leq 20$  different models for each class of garment
    - e.g. three shirts, two belts, four shoes, ..., each with its own price
  - Task: Buy **just one** model of **each class** of garment
  - Our budget  $1 \leq M \leq 200$  is limited
    - We cannot spend more money than it
    - But we want to spend the maximum possible
  - What is our maximum possible spending?
    - Output “no solution” if this is impossible

- Budget M = 100

– Answer: 75

Garment	Model 0	1	2	3
0	8	6	4	
1	5	10		
2	1	3	3	7
C = 3	50	14	23	8

- Budget M = 20

– Answer: 19

- Alternative answers are possible

Garment	Model 0	1	2	3
0	4	6	8	
1	5	10		
C = 2	1	3	5	5

- Budget M = 5

– Answer: no solution

Garment	Model 0	1	2	3
0	6	4	8	
1	10	6		
C = 2	7	3	1	7

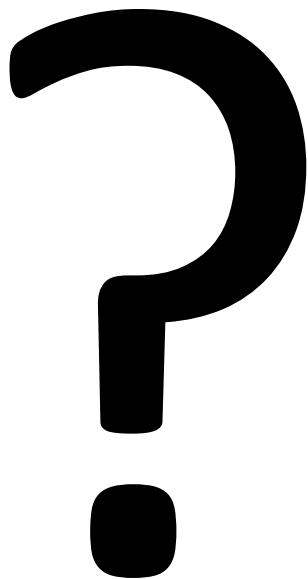
# Greedy Solution?

- What if we buy the most expensive model for each garment which still fits our budget?
- Counter example:
  - $M = 12$
  - Greedy will produce:
    - no solution
  - **Wrong answer!**
    - The correct answer is 12
    - (see the **green dotted highlights**)
      - Q: Can you spot one more potential optimal solution?

Garment	Model 0	Model 1	Model 2	Model 3
0	6	4	8	
1	5	10		
$C = 2$	1	5	3	5

# Divide and Conquer?

- Any idea?



# Complete Search? (1)

- What is the potential state of the problem?
  - $g$  (which garment?)
  - $id$  (which model?)
  - money (money left?)
- Answer:
  - $(\text{money}, g)$  or  $(g, \text{money})$
- Recurrence:

```
shop(money, g)
    if (money < 0) return -INF
    if (g == C - 1) return M - money
    return max(shop(money - price[g][model], g + 1), ∀model ∈ [1..K])
```

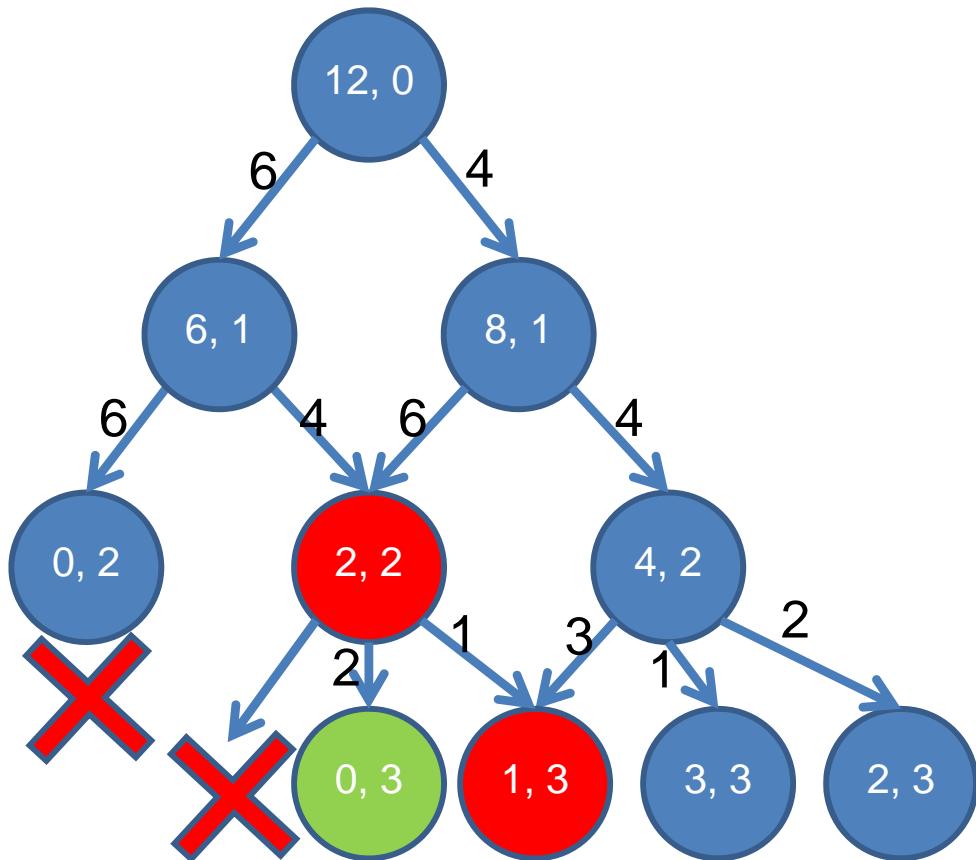
# Complete Search? (2)

- But, how to solve this?
  - $M = 1000$
- Time Complexity:  $20^{20}$ 
  - Too many for 3s time limit 😞

Garment	Model	0	1	...	19
0	32	12			55
1	2	53			4
2	1	3			7
3	50	14			8
4	3	1			5
5	4	3			1
6	5	2			5
...					
19	22	11			99

# Overlapping Sub Problem Issue

- In simple  $20^{20}$  Complete Search solution, we observe **many overlapping sub problems!**
  - Many ways to reach state (money, g), e.g. see below,  $M = 12$



Model Garment	0	1	2
0	6	4	
1	4	6	
C = 2	1	2	3

# DP to the Rescue (1)

- DP = Dynamic Programming
  - Programming here is not writing computer code, but a **“tabular method”!**
    - a.k.a. **table** method
  - A programming paradigm that you must know!
    - And hopefully, master...

# DP to the Rescue (2)

- Use DP when the problem exhibits:
  - Optimal sub structure
    - Optimal solution to the original problem contains optimal solution to sub problems
      - This is similar as the requirement of Greedy algorithm
      - If you can formulate complete search recurrences, you have this
  - Overlapping sub problems
    - Number of **distinct sub problems** are actually “small”
    - But they are **repeatedly computed**
      - This is different from Divide and Conquer

# DP Solution – Implementation (1)

- There are two ways to implement DP:
  - Top-Down
  - Bottom-Up
- Top-Down (Demo):
  - Recursion as per normal + **memoization table**
    - It is just a simple change from backtracking (complete search) solution!

# Turn Recursion into Memoization

initialize memo table in main function (use ‘memset’)

```
return_value recursion(params/state) {  
    if this state is already calculated,  
        simply return the result from the memo table  
    calculate the result using recursion(other_params/states)  
    save the result of this state in the memo table  
    return the result  
}
```

# Dynamic Programming (Top-Down)

- For our example:

```
shop(money, g)
    if (money < 0) return -INF
    if (g == C - 1) return M - money
    if (memo[money][g] != -1) return memo[money][g];
    return memo[money][g] = max(shop(money - price[g][model], g + 1),
        ∀model ∈ [1..K]
```

- As simple as that ☺

# DP Solution – Implementation (2)

- Another way: Bottom-Up:
  - Prepare a table that has size equals to the number of distinct states of the problem
  - Start to fill in the table with base case values
  - Get the topological order in which the table is filled
    - Some topological orders are natural and can be written with just (nested) loops!
  - Different way of thinking compared to Top-Down DP
- Notice that both DP variants use “table”!

# Dynamic Programming (Bottom-Up)

- For our example:
  - Start with with table **can\_reach** of size 201 (money) \* 20 (g)
    - Initialize all entries to 0 (false)
    - Fill in the first column with money left (row) reachable after buying models from the first garment
  - Use the information of current column c to update the values at column c + 1

	0	1	2		0	1	2		0	1	2
0	0	0	0		0	0	0	0	0	0	0
1	0	0	0		1	0	0	0	1	0	0
2	0	0	0		2	0	1	0	2	0	1
3	0	0	0		3	0	0	0	3	0	0
4	0	0	0		4	0	1	0	4	0	1
5	0	0	0		5	0	0	0	5	0	0
6	0	0	0		6	0	1	0	6	0	1
7	0	0	0		7	0	1	0	7	0	1
8	0	0	0		8	0	0	0	8	0	0
9	0	0	0		9	0	1	0	9	0	1
10	0	0	0		10	0	0	0	10	0	0
11	0	0	0		11	0	1	0	11	0	1
12	1	0	0		12	1	0	0	12	1	0
13	0	0	0		13	0	0	0	13	0	0
14	1	0	0		14	1	0	0	14	1	0
15	0	0	0		15	0	0	0	15	0	0
16	1	0	0		16	1	0	0	16	1	0
17	0	0	0		17	0	0	0	17	0	0
18	0	0	0		18	0	0	0	18	0	0
19	0	0	0		19	0	0	0	19	0	0
20	0	0	0		20	0	0	0	20	0	0

# Top-Down or Bottom-Up?

- Top-Down
  - Pro:
    - Natural transformation from normal recursion
    - Only compute sub problems when necessary
  - Cons:
    - Slower if there are many sub problems due to recursive call overhead
    - Use exactly  $O(\text{states})$  table size (MLE?)
- Bottom Up
  - Pro:
    - Faster if many sub problems are visited: no recursive calls!
    - Can save memory space ^
  - Cons:
    - Maybe not intuitive for those inclined to recursions?
    - If there are X states, bottom up visits/fills the value of all these X states

Flight Planner (study this on your own)

## **EXAMPLE 2**

# Motivation

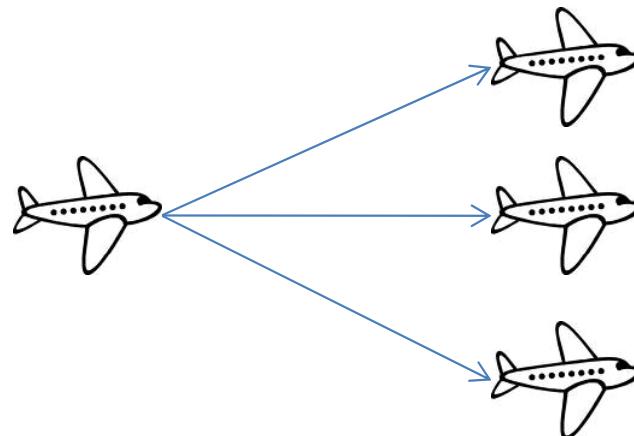


- How to solve this: [10337](#) (Flight Planner)?
  - Unit: 1 mile altitude and 1 (x100) miles distance
  - Given wind speed map
  - Fuel cost: {**climb** (+60), **hold** (+30), **sink** (+20)} - wind speed  $wsp[alt][dis]$
  - Compute **min** fuel cost from (0, 0) to (0, X = 4)!

1	1	1	1	9
1	1	1	1	8
1	1	1	1	7
1	1	1	1	6
1	1	1	1	5
1	1	1	1	4
1	1	1	1	3
1	1	1	1	2
1	9	9	1	1
1	-9	-9	1	0
=====				4 (x100)
0	1	2	3	4 (x100)

# Complete Search? (1)

- First guess:
  - Do complete search/brute force/**backtracking**
  - Find *all possible* flight paths and pick the one that yield the minimum fuel cost



# Complete Search? (2)

- Recurrence of the Complete Search

- **fuel**(alt, dis) =  
$$\min(60 - \text{wsp}[\text{alt}][\text{dis}] + \text{fuel}(\text{alt} + 1, \text{dis} + 1),$$
$$30 - \text{wsp}[\text{alt}][\text{dis}] + \text{fuel}(\text{alt}, \text{dis} + 1),$$
$$20 - \text{wsp}[\text{alt}][\text{dis}] + \text{fuel}(\text{alt} - 1, \text{dis} + 1))$$

- Stop when we reach final state (base case):

- alt = 0 and dis = X, i.e.  $\text{fuel}(0, X) = 0$

- Prune infeasible states (also base cases):

- alt < 0 or alt > 9 or dis > X!, i.e. return INF\*

- Answer of the problem is **fuel(0, 0)**

# Complete Search Solutions (1)

- Solution 1

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1 →	-9	-9	1 →		0
=====					
0	1	2	3	4 (x100)	

$$29+ 39+ 39+ 29 = \mathbf{136}$$

- Solution 2

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1 →	-9	-9	1 →		0
=====					
0	1	2	3	4 (x100)	

$$29+ 39+ 69+ 19 = \mathbf{156}$$

# Complete Search Solutions (2)

- Solution 3

1	1	1	1	1	9
1	1	1	1	1	8
1	1	1	1	1	7
1	1	1	1	1	6
1	1	1	1	1	5
1	1	1	1	1	4
1	1	1	1	1	3
1	1	1	1	1	2
1	9	9	1	1	1
1 →	-9	-9	1 →	0	0
=====					
0	1	2	3	4 (x100)	

$$29+ 69+ 11+ 29 = \mathbf{138}$$

- Solution 4

1	1	1	1	1	9
1	1	1	1	1	8
1	1	1	1	1	7
1	1	1	1	1	6
1	1	1	1	1	5
1	1	1	1	1	4
1	1	1	1	1	3
1	1	1	1	1	2
1	9	9	1	1	1
1 →	-9	-9	1 →	0	0
=====					
0	1	2	3	4 (x100)	

$$59+ 11+ 39+ 29 = \mathbf{138}$$

# Complete Search Solutions (3)

- Solution 5

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4 (x100)	

$$29+ 69+ 21+ 19 = \mathbf{138}$$

- Solution 6

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0
=====					
0	1	2	3	4 (x100)	

$$59+ 21+ 11+ 29 = \mathbf{120(OPT)}$$

# Complete Search Solutions (4)

- Solution 7

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0

$$59 + 21 + 21 + 19 = \mathbf{120} (\text{OPT})$$

- Solution 8

1	1	1	1		9
1	1	1	1		8
1	1	1	1		7
1	1	1	1		6
1	1	1	1		5
1	1	1	1		4
1	1	1	1		3
1	1	1	1		2
1	9	9	1		1
1	-9	-9	1		0

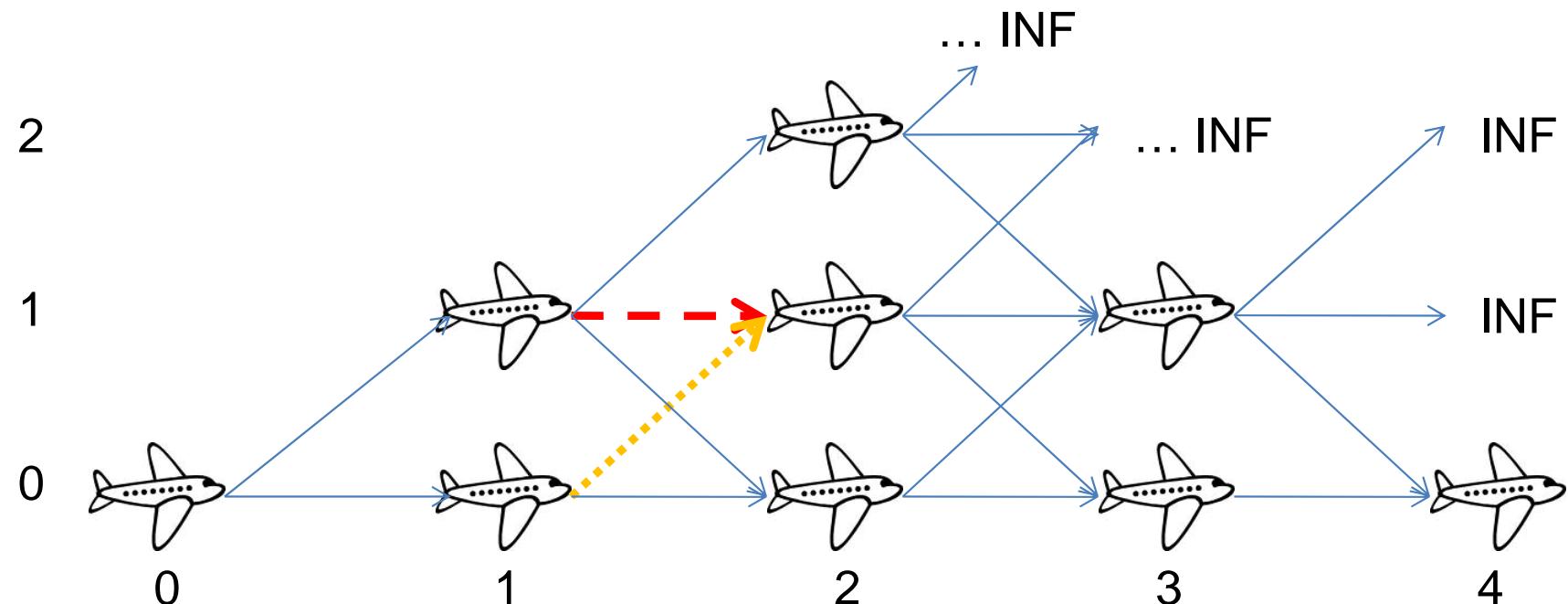
$$59 + 51 + 19 + 19 = \mathbf{148}$$

# Complete Search? (3)

- How large is the search space?
  - Max distance is 100,000 miles  
Each distance step is 100 miles  
That means we have **1,000** distance columns!
    - Note: this is an example of “coordinate compression”
  - Branching factor per step is 3... (climb, hold, sink)
  - That means complete search can end up performing  $3^{1,000}$  operations...
  - Too many for 3s time limit ☹

# Overlapping Sub Problem Issue

- In simple  $3^{1,000}$  Complete Search solution, we observe many **overlapping sub problems!**
  - Many ways to reach coordinate (alt, dis)



# DP Solution

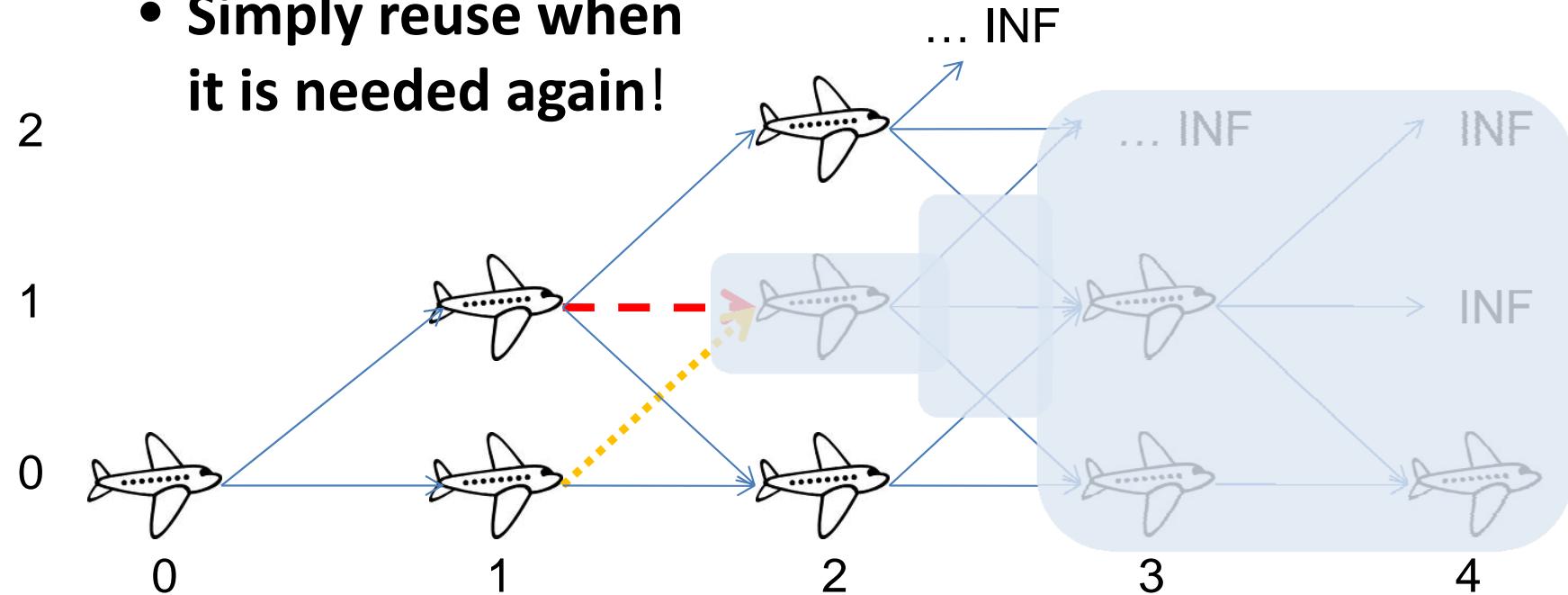
- Recurrence\* of the Complete Search
  - `fuel(alt, dis) =`  
`min3(60 - wsp[alt][dis] + fuel(alt + 1, dis + 1),`  
`30 - wsp[alt][dis] + fuel(alt , dis + 1),`  
`20 - wsp[alt][dis] + fuel(alt - 1, dis + 1))`
- Sub-problem `fuel(alt, dis)` can be **overlapping!**
  - There are only 10 alt and 1,000 dis = **10,000** states
  - A lot of time saved if these are not re-computed!
    - Exponential  $3^{1,000}$  to polynomial  $10^*1,000!$

alt > 2 not shown

2	-1	-1	-1	$\infty$	$\infty$	
1	-1	-1	<b>40</b>	<b>19</b>	$\infty$	
0	-1	-1	-1	<b>29</b>	<b>0</b>	
	0	1	2	3	4	

# DP Solution (Top Down)

- Create a 2-D table of size  $10 * (X/100)$   $\leftarrow$  Save Space!
  - Set “-1” for unexplored sub problems (memset)
  - Store the computation value of sub problem
    - **Simply reuse when it is needed again!**



# DP Solution (Bottom Up)

```
fuel(alt, dis) =
    min3(20 - wsp[alt + 1][dis - 1] + fuel(alt + 1, dis - 1),
          30 - wsp[alt      ][dis - 1] + fuel(alt      , dis - 1),
          60 - wsp[alt - 1][dis - 1] + fuel(alt - 1, dis - 1))
```

1	1	1	1	1	9
1	1	1	1	1	8
1	1	1	1	1	7
1	1	1	1	1	6
1	1	1	1	1	5
1	1	1	1	1	4
1	1	1	1	1	3
1	1	1	1	1	2
1	9	9	1	1	1
1	-9	-9	1	1	0
=====					
0	1	2	3	4	(x100)

Tips:  
(space-saving trick)

We can reduce one storage dimension by only keeping 2 recent columns at a time...

But the time complexity is unchanged:  
 $O(10 * X / 100)$

2	$\infty$	$\infty$			
1	$\infty$	59			
0	0	29			
	0	1	2	3	4

2	$\infty$	$\infty$	110		
1	$\infty$	59	80		
0	0	29	68		
	0	1	2	3	4

2	$\infty$	$\infty$	110	131	
1	$\infty$	59	80	101	
0	0	29	68	91	
	0	1	2	3	4

2	$\infty$	$\infty$	110	131	-
1	$\infty$	59	80	101	-
0	0	29	68	91	120
	0	1	2	3	4

# If Optimal Solution(s) are Needed

- Although not often, sometimes this is asked!
- As we build the DP table,  
record which option is taken in each cell!
  - Usually, this information is stored in different table
  - Then, do recursive scan(s) to output solution
    - Sometimes, there are more than one solutions!

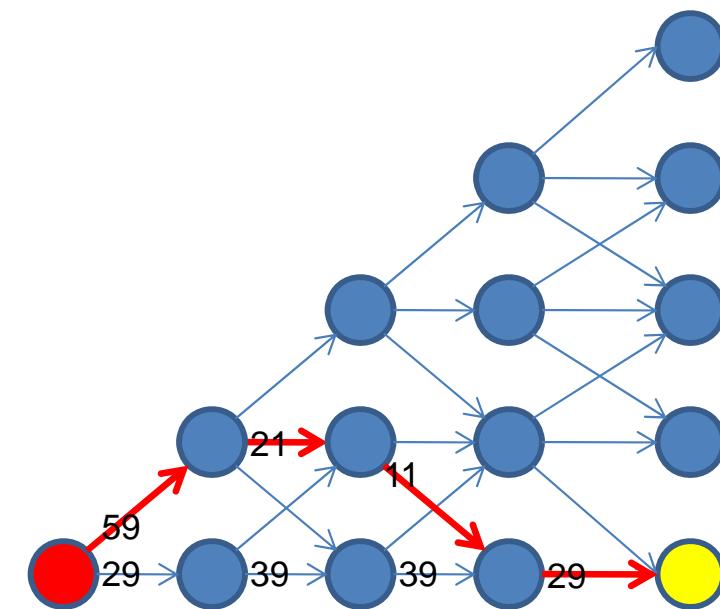
2	$\infty$	$\infty$	110	131	-
1	$\infty$	59	80	101	-
0	0	29	68	91	120
	0	1	2	3	4

# Shortest Path Problem? (1)

- Hey, I have alternative solution:
  - Model the problem as a **DAG**
  - Vertex is each position in the unit map
  - Edges connect vertices reachable from vertex  $(\text{alt}, \text{dis})$ , i.e.  $(\text{alt}+1, \text{dis}+1)$ ,  $(\text{alt}, \text{dis}+1)$ ,  $(\text{alt}-1, \text{dis})$ 
    - Weighted according to flight action and wind speed!
    - Do not connect infeasible vertices
      - $\text{alt} < 0$  or  $\text{alt} > 9$  or  $\text{dis} > X$

# Visualization of the DAG

1	1	1	1	1	9
1	1	1	1	1	8
1	1	1	1	1	7
1	1	1	1	1	6
1	1	1	1	1	5
1	1	1	1	1	4
1	1	1	1	1	3
1	1	1	1	1	2
1	9	9	1	1	1
1	-9	-9	1	1	0
=====					
0	1	2	3	4	(x100)



Source

What is the  
**shortest path**  
from source  
to destination?

# Shortest Path Problem? (2)

- The problem: find the **shortest path** from vertex  $(0, 0)$  to vertex  $(0, X)$  on this DAG...
- $O(V + E)$  solution exists!
  - $V$  is just  $10 * (X / 100)$
  - $E$  is just  $3V$
  - Thus this solution is as good as the DP solution

# Break

- In the next part, we will strengthen our DP concepts by looking at **more** examples...
  - For this lecture: Classical Ones First
    - Longest Increasing Subsequence (LIS)
    - Max Sum (1-D and 2-D)
    - 0-1 Knapsack / Subset Sum
    - Coin Change (General Case)
    - Traveling Salesman Problem (TSP)

Let's discuss several problems that are solvable using DP  
First, let's see some classical ones...

## LEARNING VIA EXAMPLES

# Longest Increasing Subsequence (1)



- Problem Description (Abbreviated as LIS):
  - As implied by its name....

Given a sequence  $\{X[0], X[1], \dots, X[N-1]\}$ , determine the Longest Increasing Subsequence

    - Subsequence is not necessarily contiguous
    - Example:  $N = 8$ , sequence =  $\{-7, 10, 9, 2, 3, 8, 8, 1\}$ 
      - LIS is  $\{-7, 2, 3, 8\}$  of length 4
    - Variants:
      - Longest Decreasing Subsequence
      - Longest Non Decreasing<sup>^</sup> Subsequence

# Longest Increasing Subsequence (2)

- Let  $LIS(i)$  be the LIS ending in index  $i$
- Complete Search Recurrence:
  - $LIS(0) = 1$  // base case
  - $LIS(i) = \text{longestLIS}$ 
    - ```
int longestLIS = 1;
for (int j = 0; j < i; j++)
    if (x[i] > x[j])
        longestLIS = max(longestLIS, 1 + LIS(j))
```
  - The answer is  $\max(LIS(k))$ , for  $k \in [0..N-1]$

# Longest Increasing Subsequence (3)

- Many overlapping sub problems,  
but there are only N states,  
LIS ending at index i, for all  $i \in [0..N-1]$

| Index  | 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 |
|--------|----|----|---|---|---|---|---|---|
| X      | -7 | 10 | 9 | 2 | 3 | 8 | 8 | 1 |
| LIS(i) | 1  | 2  | 2 | 2 | 3 | 4 | 4 | 2 |

- This is  $O(n^2)$  algorithm
  - Solutions can be reconstructed: follow the arrows

# Max Sum (1D)

- Given array  $X = \{10, -1, 6, 3, 2, -2, -6, 7, 1\}$  of length  $n$ 
  - Find a **contiguous** sequence with **maximum sum**
    - The answer is **{6, 3, 2}** with max sum  $6 + 3 + 2 = 11$
  - Naïve solution in  $O(N^3)$ 
    - Try all starting and ending indices  $i & j$ ,  $O(N^2)$
    - Sum the values between index  $i$  and  $j$ ,  $O(N)$
    - Report the best starting and ending indices
  - DP solution in  $O(N^2)$ 
    - Try all starting and ending indices  $i & j$ ,  $O(N^2)$
    - Get the range sum between index  $i$  and  $j$  in  $O(1)$** 
      - Use Fenwick Tree idea (Week02) =  $RSQ(i, j) = RSQ(0, j) - RSQ(0, i - 1)$
    - Report the best starting and ending indices
  - Q: Do you know the Greedy solution (Kadane's algorithm) in  $O(N)$ ?



# Max Sum (2D)

- What if we are given 2D matrix instead?
  - Use similar technique as in 1D matrix
  - But this time we divide the matrix into 4 sub regions
    - See the diagrams below for explanation

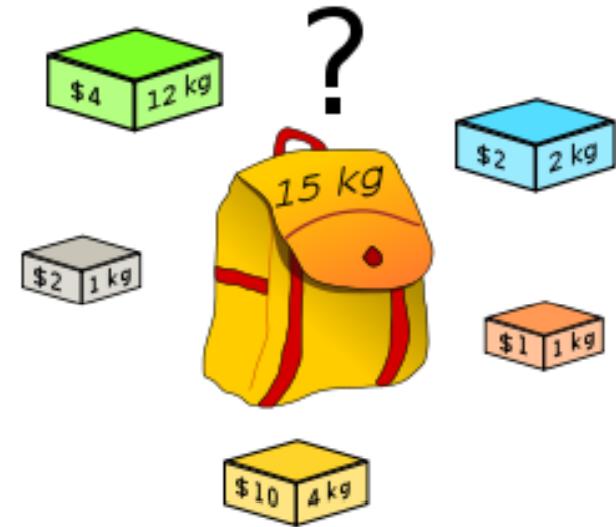
|    |   |    |    |   |
|----|---|----|----|---|
| A  | 0 | -2 | -7 | 0 |
| 9  | 2 | -6 | 2  |   |
| -4 | 1 | -4 | 1  |   |
| -1 | 8 | 0  | -2 |   |

|   |    |     |    |    |
|---|----|-----|----|----|
| B | 0  | -2  | -9 | -9 |
| 9 | 9  | -4  | 2  |    |
| 5 | 6  | -11 | -8 |    |
| 4 | 13 | -4  | -3 |    |

|   |    |     |    |    |
|---|----|-----|----|----|
| C | 0  | -2  | -9 | -9 |
| 9 | 9  | -4  | 2  |    |
| 5 | 6  | -11 | -8 |    |
| 4 | 13 | -4  | -3 |    |

# 0-1 Knapsack / Subset Sum

- Standard state:
  - $\text{value}(id, w)$
- Standard transition:
  - Take item ‘id’ (only if  $W[id] \leq w$ )
    - Go to state:  $V[id] + \text{value}(id + 1, w - W[id])$
  - Ignore item ‘id’
    - Go to state:  $\text{value}(id + 1, w)$
  - Stop when
    - $id = N$  (all items taken care of)
    - $w = 0$  (cannot carry anything else)



# Coin Change (1)



- Problem Description:
  - Given an amount  $V$  cents and a list of  $N$  coins:
    - We have  $\text{coinValue}[i]$  for coin type  $i \in [0..N-1]$
  - What is the **minimum number of coins** that we must use to obtain amount  $V$ ?
  - Assume that we have **unlimited** supply of coins of any type!
  - UVa: [166](#)

# Coin Change (2)

- Example:
  - $V = 10, N = 2, \text{coinValue} = \{1, 5\}$
  - We can use:
    - **Ten** 1 cent coins =  $10 * 1 = 10$ 
      - Total coins used = 10
    - **One** 5 cents coin + **Five** 1 cent coins =  $1 * 5 + 5 * 1 = 10$ 
      - Total coins used = 6
    - **Two** 5 cents coins =  $2 * 5 = 10$ 
      - Total coins used = 2 → Optimal
  - Is Greedy Algorithm Possible? ^

# Coin Change (3)

- Complete Search Recurrence:
  - $\text{coin}(0) = 0$  // 0 coin to produce 0 cent
  - $\text{coin}(<0) = \text{INFINITY}$  (infeasible solution)
  - $\text{coin}(\text{value}) = 1 + \min(\text{coin}(\text{value} - \text{coinValue}[i]))$  for all  $i \in [0..N-1]$
- How many possible states of parameter value?
  - Only  $O(V)$  and  $V$  is usually small

|          |   |   |   |   |   |   |   |   |   |   |    |                                                   |
|----------|---|---|---|---|---|---|---|---|---|---|----|---------------------------------------------------|
| <0       | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | $V = 10, N = 2,$<br>$\text{coinValue} = \{1, 5\}$ |
| $\infty$ | 0 | 1 | 2 | 3 | 4 | 1 | 2 | 3 | 4 | 5 | 2  |                                                   |

Both bottom up and top down DP happen to produce the same table...

# Traveling Salesman Problem (TSP)

- Standard state:
  - $dp(pos, \text{bitmask})$
- Standard transition:
  - If every cities have been visited
    - $tsp(pos, 2^N - 1) = dist[pos][0]$
  - Else, try visiting unvisited cities one by one
    - $tsp(pos, \text{bitmask}) = \min(dist[pos][nxt] + tsp(nxt, \text{bitmask} | (1 << nxt)))$   
for all possible  $nxt$  in  $[0..N-1]$ ,  $nxt \neq pos$ ,  
and  $\text{bitmask} \& (1 << nxt)$  is off



# Summary

- We have seen:
  - Basic DP concepts
  - DP on some **classical** problems
- We will see more DP in Week06:
  - DP and its relationship with DAG
  - DP on **non classical** problems
  - Plus some other “cool” DP techniques

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 05 – Graph 1 (“Basics”)

# Outline (1)

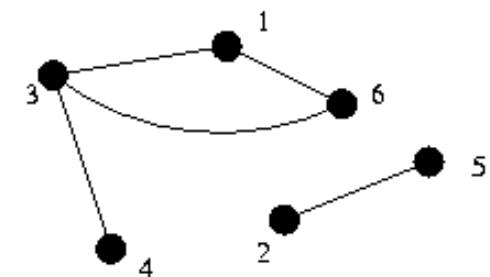
- Mini Contest #4 + Break + Discussion
- Admins
- CS2010/2020 Reviews (not discussed in details except red ones)
  - Graph: Preliminary & Motivation
  - Graph Traversal Algorithms
    - DFS/BFS: **Connected Components**/Flood Fill
    - DFS only: Toposort/Cut Vertex+Bridges/**Strongly Connected Components**
  - Minimum Spanning Tree Algorithm
    - Kruskal's and Prim's: Plus Various Applications

# Outline (2)

- CS2010/2020 Reviews (not discussed in details except **red ones**)
  - Single-Source Shortest Paths
    - BFS: SSSP on Unweighted graph/Variants
    - Dijkstra's: SSSP on Weighted (no -ve cycle) graph/Variants
  - All-Pairs Shortest Paths
    - Floyd Warshall's + Variants
  - Special Graphs Part 1
    - Tree, Euler Graph, Directed Acyclic Graph

# Graph Terms – Quick Review

- Vertices/Nodes
- Edges
- Un/Weighted
- Un/Directed
- In/Out Degree
- Self-Loop/Multiple Edges  
(Multigraph) vs Simple  
Graph
- Sparse/Dense
- Path, Cycle
- Isolated, Reachable
- (Strongly) Connected  
Component
- Sub Graph
- Complete Graph
- Directed Acyclic Graph
- Tree/Forest
- Euler/Hamiltonian  
Path/Cycle
- Bipartite Graph



# Kaohsiung 2006

## Standings & Felix's Analysis

- |                                    |                          |
|------------------------------------|--------------------------|
| A. Print Words in Lines            | A. DP                    |
| B. The Bug Sensor Problem          | B. Graph, MST            |
| C. Pitcher Rotation                | C. DP + memory reduction |
| D. Lucky and Good Months...        | D. Tedious Ad Hoc        |
| E. Route Planning                  | E. Search?               |
| Shift Cipher (N/A in Live Archive) | Complete Search + STL    |
| F. A Scheduling Problem            | F. Greedy?               |
| G. Check the Lines                 | G. ?                     |
| H. Perfect Service                 | H. DA Graph, DP on Tree  |

# Singapore 2007

## Standings & Felix's Analysis

- |            |                                      |
|------------|--------------------------------------|
| A. MODEX   | A. Math, Modulo Arithmetic           |
| B. JONES   | B. DP                                |
| C. ACORN   | C. DP + memory reduction             |
| D. TUSK    | D. Geometry                          |
| E. SKYLINE | E. DS, Segment Tree                  |
| F. USHER   | F. Graph, APSP, Min Weighted Cycle++ |
| G. RACING  | G. Graph, Maximum Spanning Tree      |

# Jakarta 2008

## Standings & Suhendry's Analysis

- |                                 |                                |
|---------------------------------|--------------------------------|
| A. Anti Brute Force Lock        | A. Graph, MST                  |
| B. Bonus Treasure               | B. Recursion                   |
| C. Panda Land 7: Casino Island  | C. Trie?                       |
| D. Disjoint Paths               | D. DA Graph, DP on Tree        |
| E. Expert Enough?               | E. Complete Search (is enough) |
| F. Free Parentheses             | F. DP, harder than problem I   |
| G. Greatest K-Palindrome Sub... | G. String                      |
| H. Hyper-Mod                    | H. Math                        |
| I. ICPC Team Strategy           | I. DP, medium                  |
| J. Jollybee Tournament          | J. Ad Hoc, Simulation          |

# Kuala Lumpur 2008

- |                               |                                  |
|-------------------------------|----------------------------------|
| A. ASCII Diamondi             | A. Ad Hoc                        |
| B. Match Maker                | B. DP, Stable Marriage Problem?  |
| C. Tariff Plan                | C. Ad Hoc                        |
| D. Irreducible Fractions      | D. Math                          |
| E. Gun Fight                  | E. Graph, MCBM (AlternatingPath) |
| F. Unlock the Lock            | F. Graph, SSSP (BFS)             |
| G. Ironman Race in Treeland   | G. DA Graph, Likely DP on Tree?  |
| H. Shooting the Monster       | H. Comp Geo                      |
| I. Addition-Substraction Game | I. ?                             |
| J. The Great Game             | J. ?                             |
| K. Triangle Hazard            | K. Math                          |

# Daejeon 2010

- |                       |                                   |
|-----------------------|-----------------------------------|
| A. Sales              | A. Brute Force                    |
| B. String Popping     | B. Recursive Backtracking         |
| C. Password           | C. Recursive Backtracking         |
| D. Mines              | D. Geometry + SCCs (Graph)        |
| E. Binary Search Tree | E. Graph, BST, Math, Combinatoric |
| F. Tour Belt          | F. Graph, MST (modified)          |
| G. String Phone       | G. ?                              |
| H. Installations      | H. ?                              |
| I. Restaurant         | I. ?                              |
| J. KTX Train Depot    | J. ?                              |

Depth-First Search (DFS)

Breadth-First Search (BFS)

Reachability

**Finding Connected Components**

Flood Fill

Topological Sort

Finding Cycles (Back Edges)

Finding Articulation Points & Bridges

**Finding Strongly Connected Components**

# GRAPH TRAVERSAL ALGORITHMS

# Motivation (1)

- How to solve these UVa problems:
  - [469](#) (Wetlands of Florida)
    - Similar problems: 260, 352, 572, 782, 784, 785, etc
  - [11504](#) (Dominos)
    - Similar problems: 1263, 11709, etc
- Without familiarity with **Depth-First Search** algorithm and its variants, they look “hard”

# Motivation (2)

- How to solve these UVa problems:
  - [336](#) (A Node Too Far)
    - Similar problems: 383, 439, 532, 762, 10009, etc
- Without familiarity with **Breadth-First Search** graph traversal algorithm, they look “hard”

# Graph Traversal Algorithms

- Given a graph, we want to traverse it!
- There are 2 major ways:
  - Depth First Search (DFS)
    - Usually implemented using recursion
    - More natural
    - Most frequently used to traverse a graph
  - Breadth First Search (BFS)
    - Usually implemented using queue (+ map), use STL
    - Can solve special case\* of “shortest paths” problem!

# Depth First Search – Template

- $O(V + E)$  if using Adjacency List
- $O(V^2)$  if using Adjacency Matrix

```
typedef pair<int, int> ii; typedef vector<ii> vi;

void dfs(int u) { // DFS for normal usage
    printf(" %d", u); // this vertex is visited
    dfs_num[u] = DFS_BLACK; // mark as visited
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j]; // try all neighbors v of vertex u
        if (dfs_num[v.first] == DFS_WHITE) // avoid cycle
            dfs(v.first); // v is a (neighbor, weight) pair
    }
}
```

# Breadth First Search (using STL)

- Complexity: also  $O(V + E)$  using Adjacency List

```
map<int, int> dist; dist[source] = 0;
queue<int> q; q.push(source); // start from source

while (!q.empty()) {
    int u = q.front(); q.pop(); // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j]; // for each neighbours of u
        if (!dist.count(v.first)) {
            dist[v.first] = dist[u] + 1; // unvisited + reachable
            q.push(v.first); // enqueue v.first for next steps
        }
    }
}
```

# 1<sup>st</sup> Application: Connected Components

- DFS (and BFS) can find connected components
  - A call of `dfs(u)` visits only vertices connected to **u**

```
int numComp = 0;
dfs_num.assign(V, DFS_WHITE);
REP(i, 0, V - 1) // for each vertex i in [0..V-1]
    if (dfs_num[i] == DFS_WHITE) { // if not visited yet
        printf("Component %d, visit", ++numComp);
        dfs(i); // one component found
        printf("\n");
    }
printf("There are %d connected components\n", numComp);
```



Finding Topological Sort (see text book/CS2010/CS2020)

Finding Articulation Points and Bridges (see text book)

**Finding Strongly Connected Component**

# TARJAN'S DFS ALGORITHMS

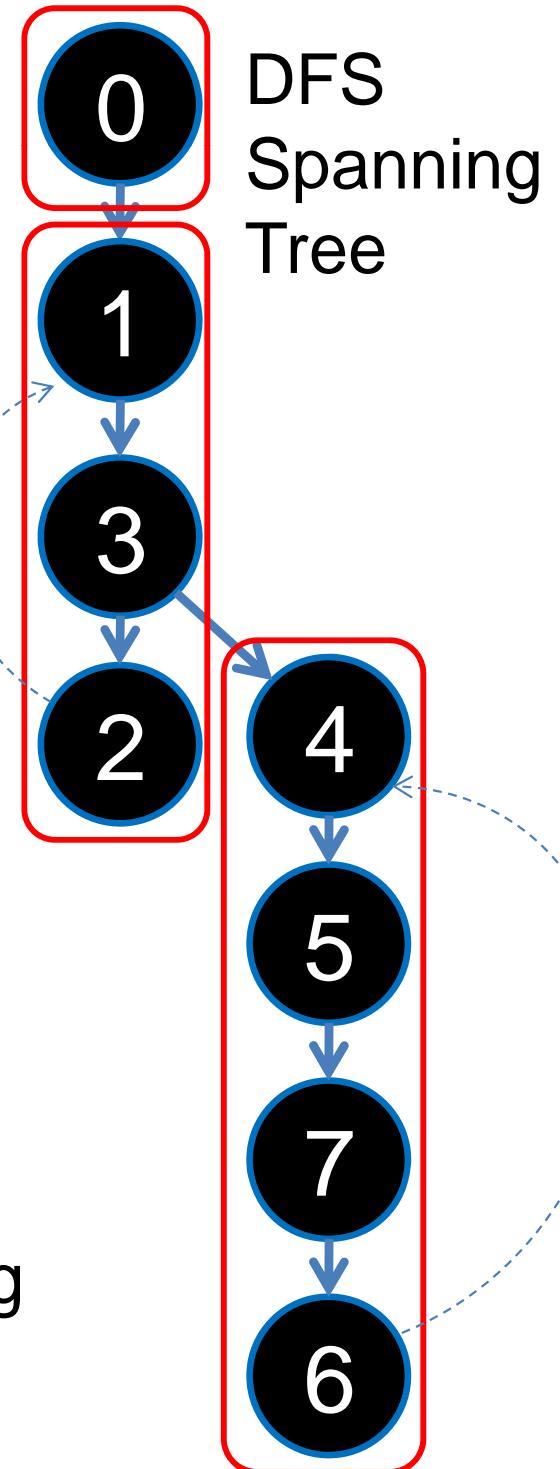
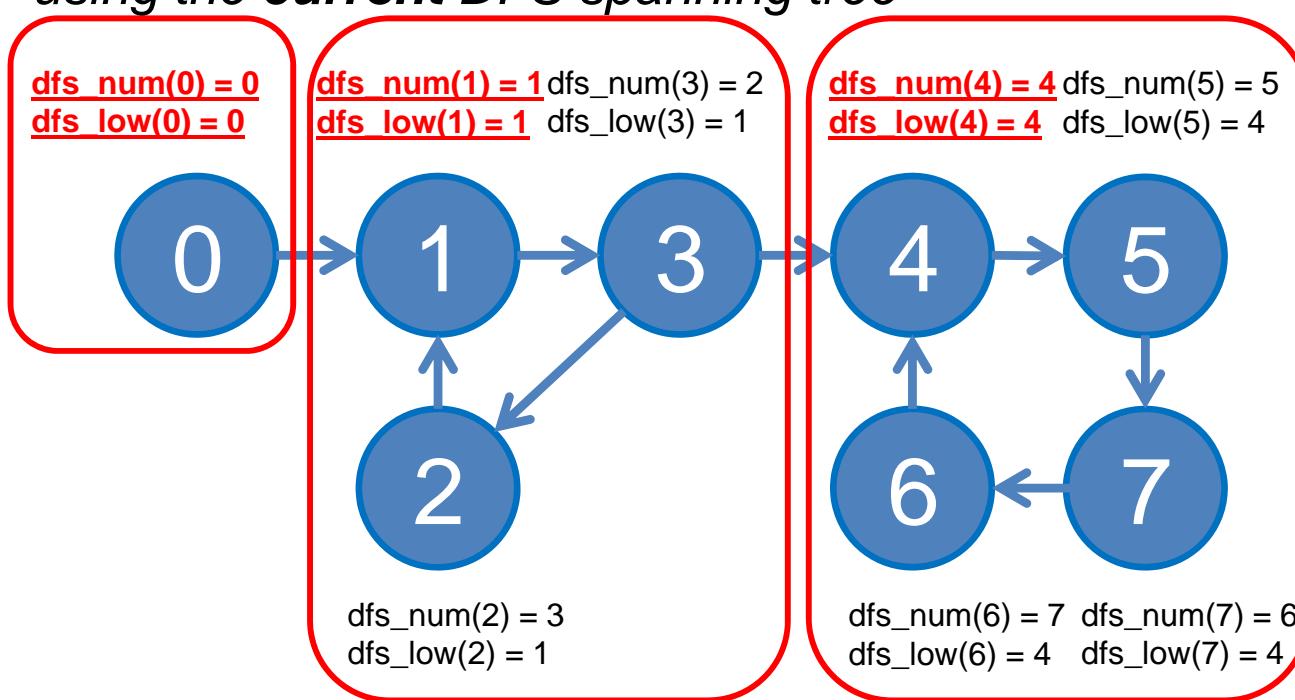
# Tarjan's SCC

Input: A Directed Graph

dfs\_num: visitation counter

dfs\_low: lowest dfs\_num reachable from that vertex

*using the **current** DFS spanning tree*



# Code: Tarjan's SCC (not in IOI syllabus)

```
vi dfs_num, dfs_low, S, visited; // global variables

void tarjanSCC(int u) {
    dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
    S.push_back(u); // stores u in a vector based on order of visitation
    visited[u] = 1;
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii = AdjList[u][j];
        if (dfs_num[v.first] == DFS_WHITE) // a tree edge
            tarjanSCC(v.first);
        if (visited[v.first]) // condition for update
            dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); // update dfs_low[u]
    }
    if (dfs_low[u] == dfs_num[u]) { // if this is a root (start) of an SCC
        printf("SCC %d: ", ++numSCC); // this part is done after recursion
        while (1) {
            int v = S.back(); S.pop_back(); visited[v] = 0;
            printf(" %d", v);
            if (u == v) break;
        }
        printf("\n");
    }
}
```

# Graph Traversal Comparison

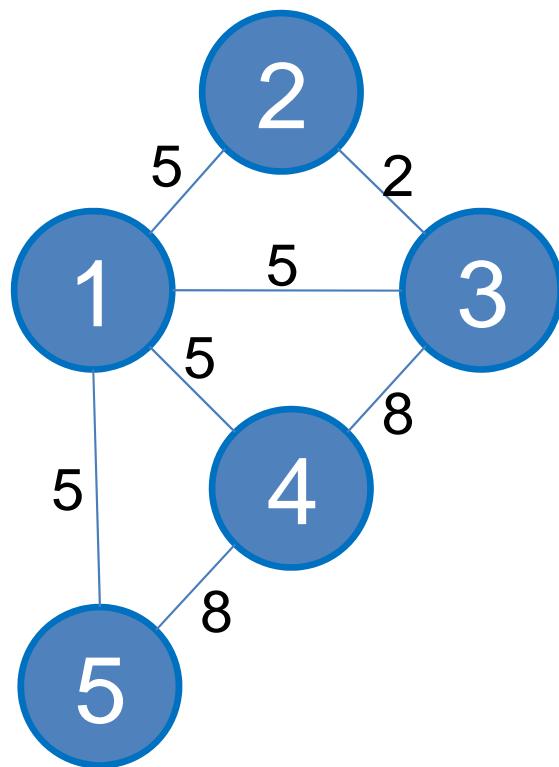
- DFS
- Pros:
  - Slightly easier to code
  - Use less memory
- Cons:
  - Cannot solve SSSP on unweighted graphs
- BFS
- Pros:
  - Can solve SSSP on unweighted graphs (discussed later)
- Cons:
  - Slightly longer to code
  - Use more memory

Prim's algorithm → Read textbook on your own 😊  
(or revise CS2010/CS2020 material)

# KRUSKAL'S ALGORITHM FOR MINIMUM SPANNING TREE

# How to Solve This?

- Given this graph, select some edges s.t  
the graph is connected  
but with minimal total weight!



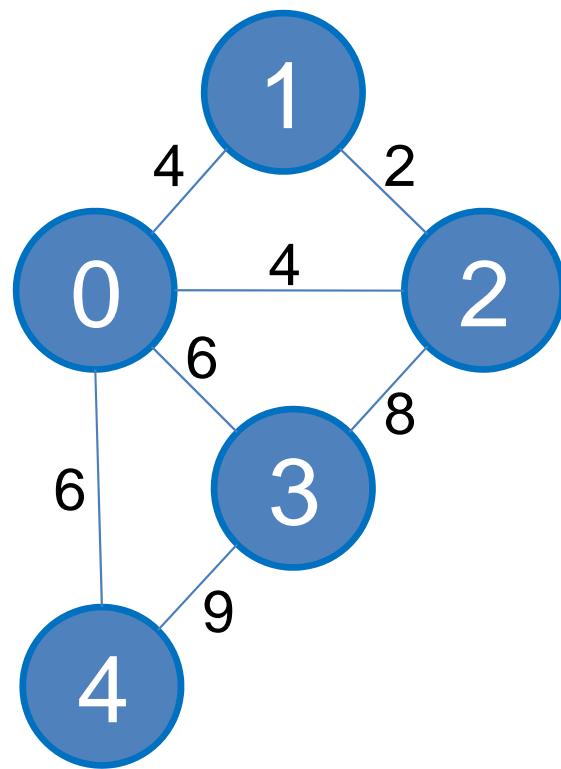
- MST!

# Spanning Tree & MST

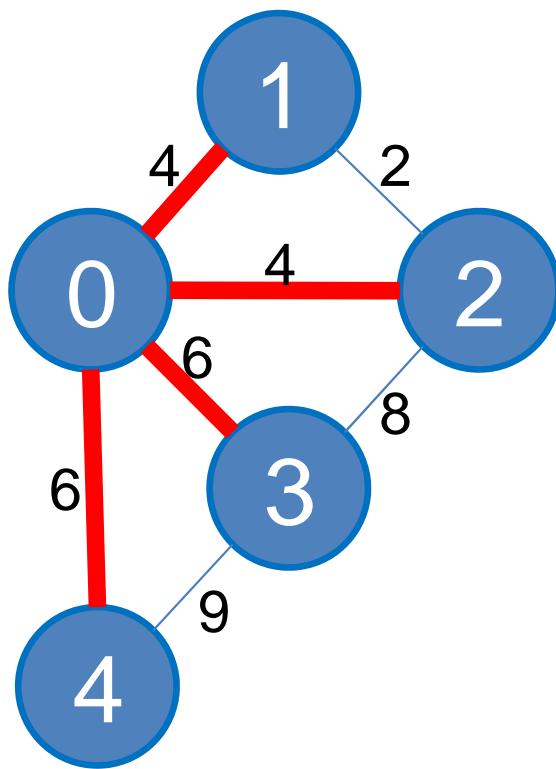
- Given a **connected undirected** graph  $G$ , select  $E \in G$  such that a tree is formed and this tree **spans** (covers) all  $V \in G$ !
  - No cycles or loops are formed!
- There can be **several** spanning trees in  $G$ 
  - The one where total cost is minimum is called the **Minimum Spanning Tree (MST)**
- UVa: [908](#) (Re-connecting Computer Sites)

# Example

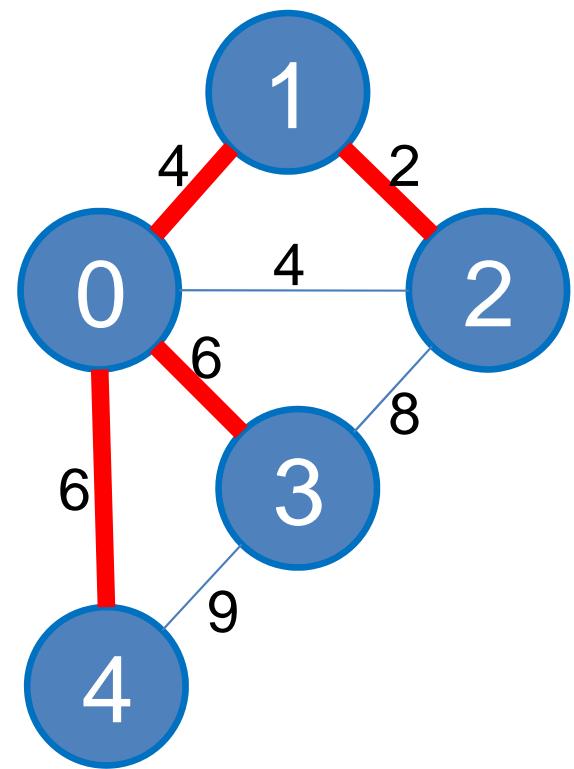
The Original Graph



A Spanning Tree  
Cost:  $4+4+6+6 = 20$



An MST  
Cost:  $4+6+6+2 = 18$



# Algorithms for Finding MST

- Prim's (Greedy Algorithm)
  - At every iteration, choose an edge with minimum cost that does not form a cycle
    - “grows” an MST from a root
- Kruskal's (also Greedy Algorithm)
  - Repeatedly finds edges with minimum costs that does not form a cycle
    - forms an MST by connecting forests
- Which one is easier to code?

# Kruskal's Algorithm



- In my opinion, Kruskal's algorithm is simpler

sort edges by increasing weight  $O(E \log E)$

**while** there are unprocessed edges left  $O(E)$

    pick an edge  $e$  **with minimum cost**

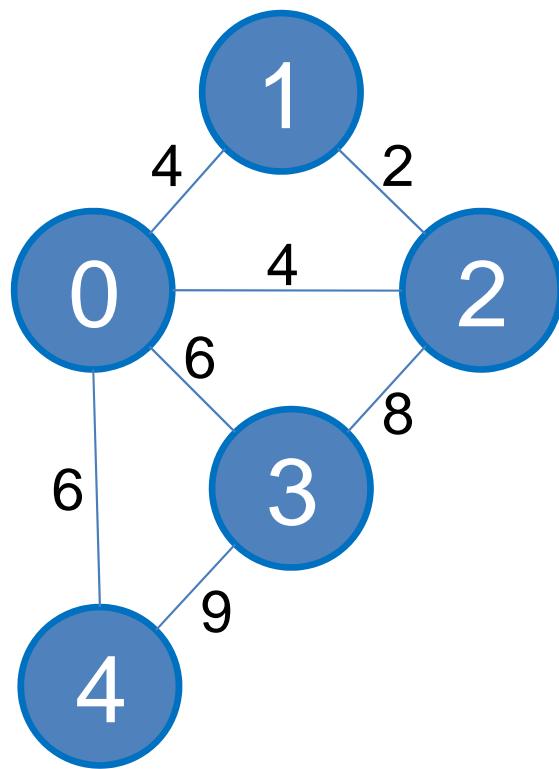
**if** adding  $e$  to MST **does not form a cycle**

        add  $e$  to MST

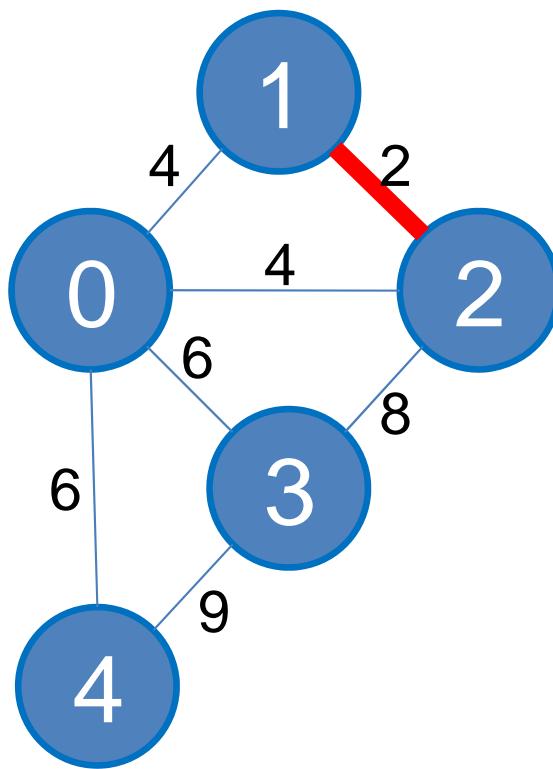
- Simply store the edges in an array of Edges (EdgeList) and sort them, or use Priority Queue
- Test for cycles using Disjoint Sets (Union Find) DS

# Kruskal's Animation (1)

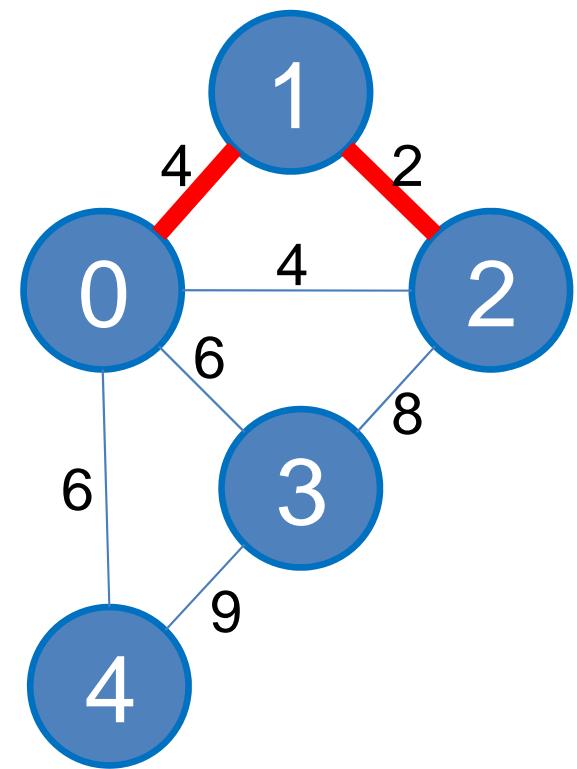
The original graph,  
no edge is selected



Connect 1 and 2  
As this edge is smallest



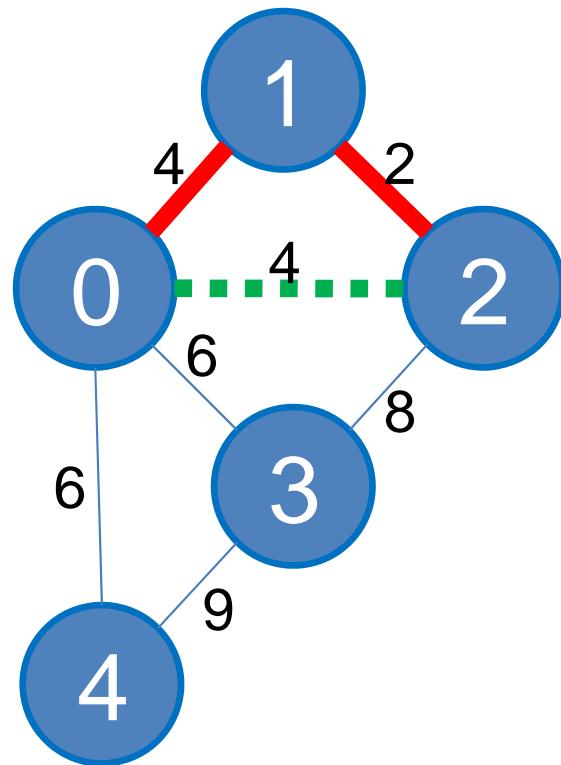
Connect 1 and 0  
No cycle is formed



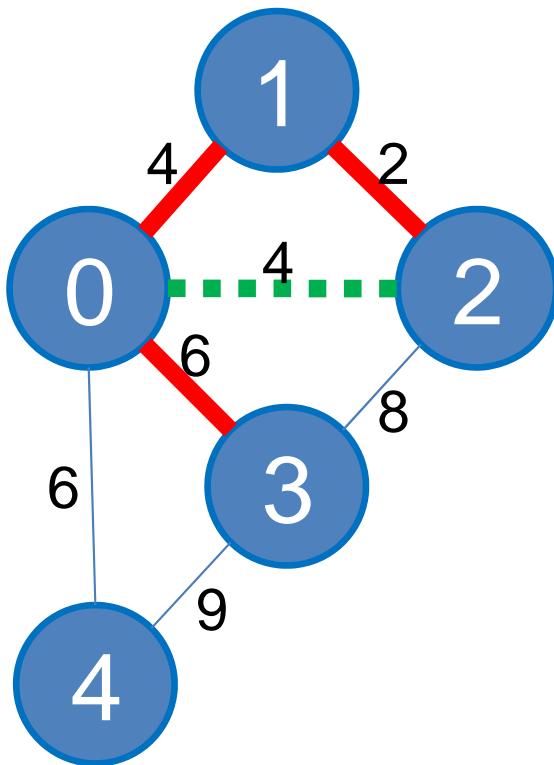
Note: The sorted order of the edges determines how the MST formed. Observe that we can also choose to connect vertex 2 and 0 also with weight 4!

# Kruskal's Animation (2)

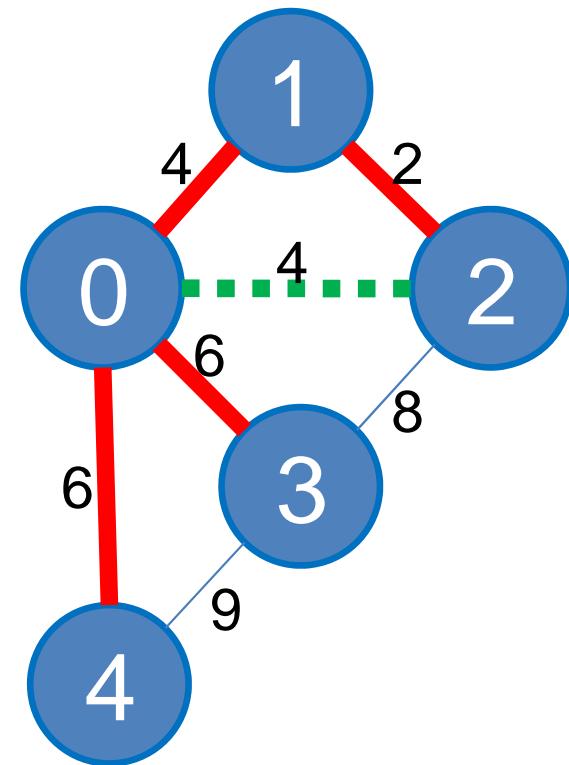
Cannot connect 0 and 2  
As it will form a cycle



Connect 0 and 3  
The next smallest edge



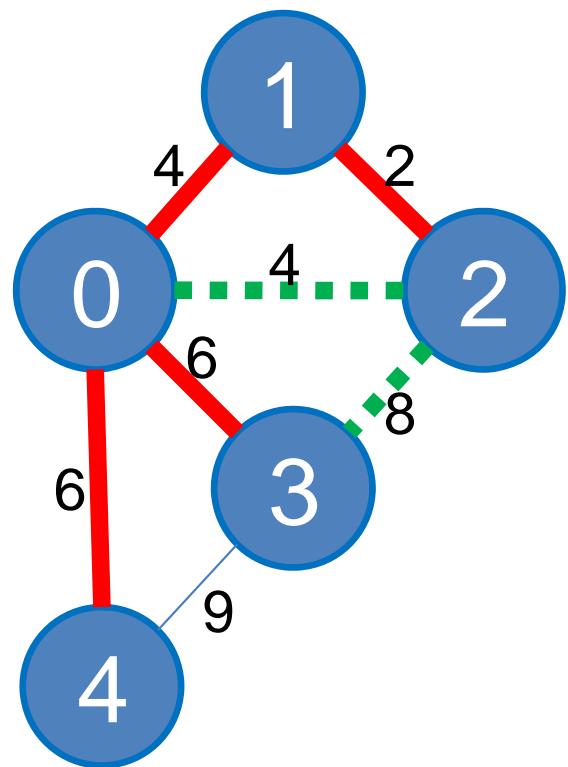
Connect 0 and 4  
MST is formed...



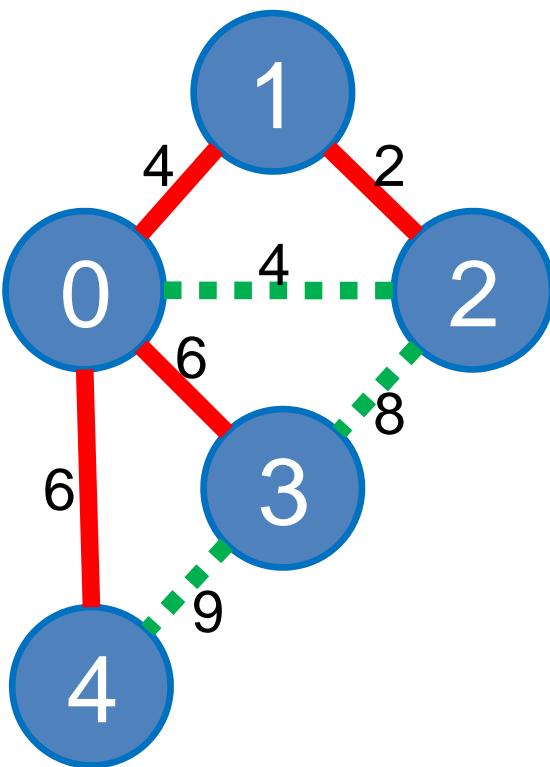
Note: Again, the sorted order of the edges determines how the MST formed; Connecting 0 and 4 is also a valid next move

# Kruskal's Animation (3)

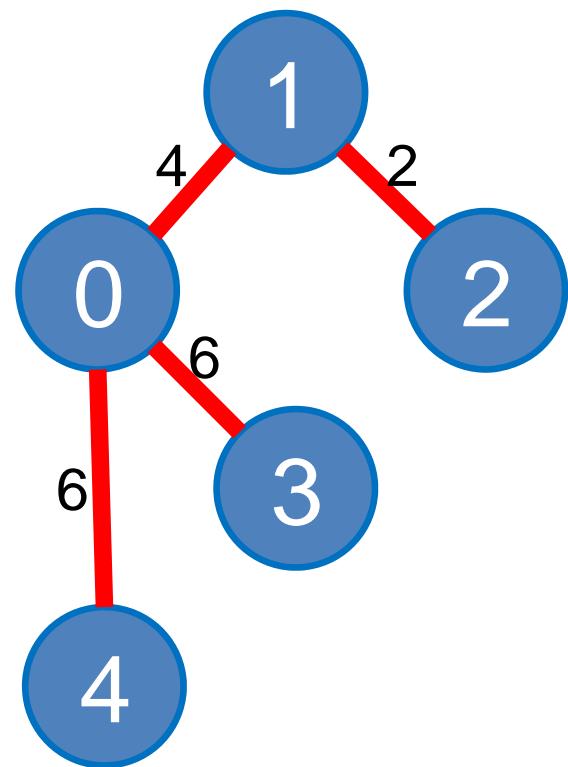
But (standard) Kruskal's algorithm will still continue



However, it will not modify anything else



This is the final MST with cost 18



# Kruskal's Algorithm (Sample Code)

```
// sorted by edge cost
vector< pair<int, ii> > EdgeList;
// insert edges in format (weight, (u, v)) to EdgeList
sort(EdgeList.begin(), Edgelist.End());

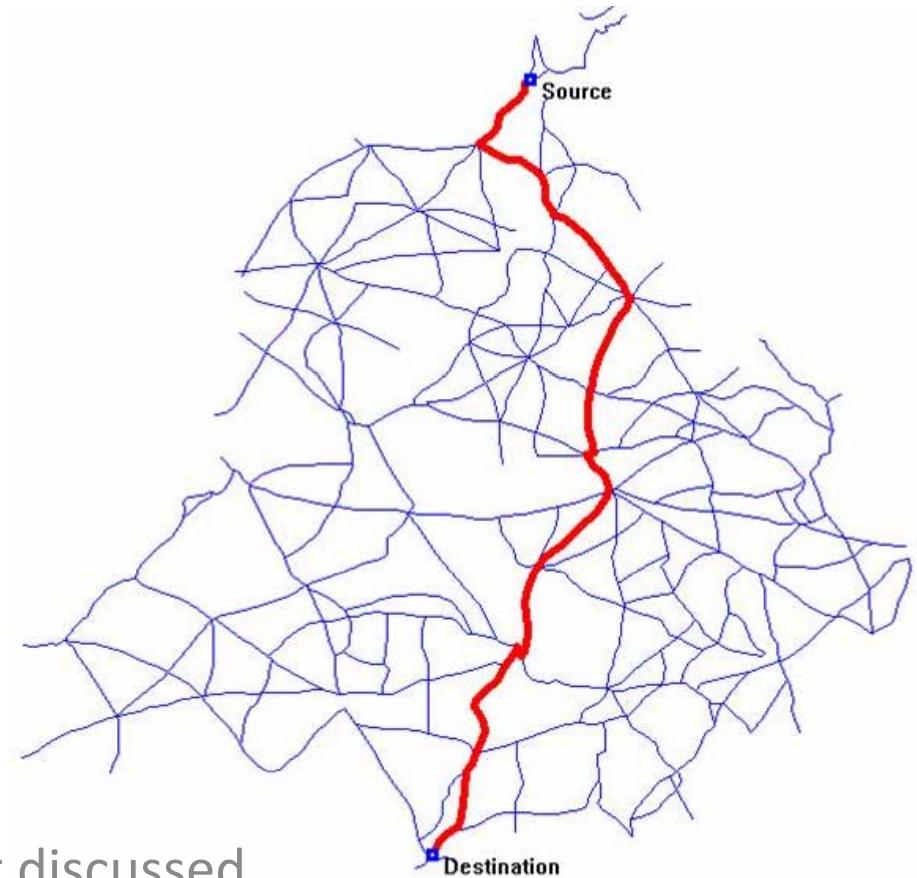
mst_cost = 0; initSet(V); // all V are disjoint initially
for (int I = 0; I < E; i++) { // while ∃ more edges
    pair<int, ii> front = EdgeList[i];
    if (!isSameSet(front.second.first, front.second.second)) {
        // if adding e to MST does not form a cycle
        mst_cost += front.first; // add the weight of e to MST
        unionSet(front.second.first, front.second.second);
    }
}
```

# But...

- You have not teach us Union Find DS in CS3233??
  - It is also only covered briefly in CS2010/CS2020
- Yeah, we choose to skip that DS in CS3233...
- If you want to solve MST problems,  
learn Union Find DS on your own (Sec 2.3.2)
- To be fair, I will **not** set any MST problems in CS3233  
mini contests problems A + B ☺
  - I do not say anything about problem C or mid/final contest

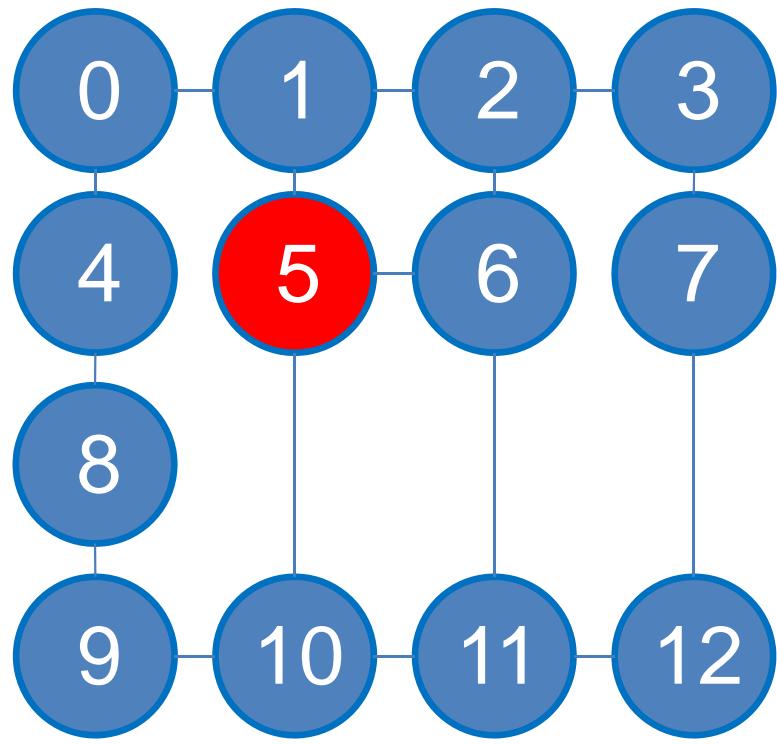
BFS (unweighted)  
Dijkstra's (non -ve cycle)  
Bellman Ford's (may have -ve cycle), not discussed  
Floyd Warshall's (all-pairs

# SHORTEST PATHS



# BFS for Special Case SSSP

- SSSP is a classical problem in Graph theory:
  - Find shortest paths from **one source** to the rest^
- Special case: [UVa 336](#) (A Node Too Far)
- Problem Description:
  - Given an **un-weighted** & un-directed Graph, a starting vertex  $v$ , and an integer TTL
  - Check how many nodes are un-reachable from  $v$  or has distance  $>$  TTL from  $v$ 
    - i.e.  $\text{length}(\text{shortest\_path}(v, \text{node})) > \text{TTL}$



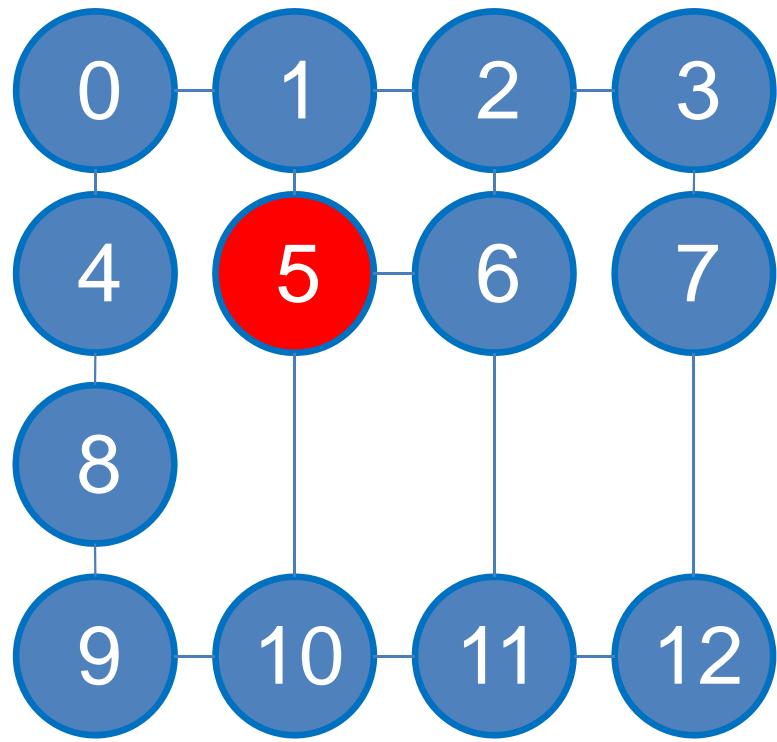
# Example (1)

$$Q = \{5\}$$

$$D[5] = 0$$

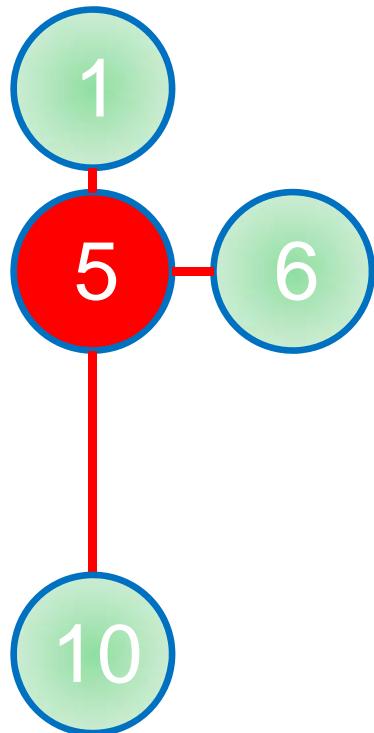


## Example (2)

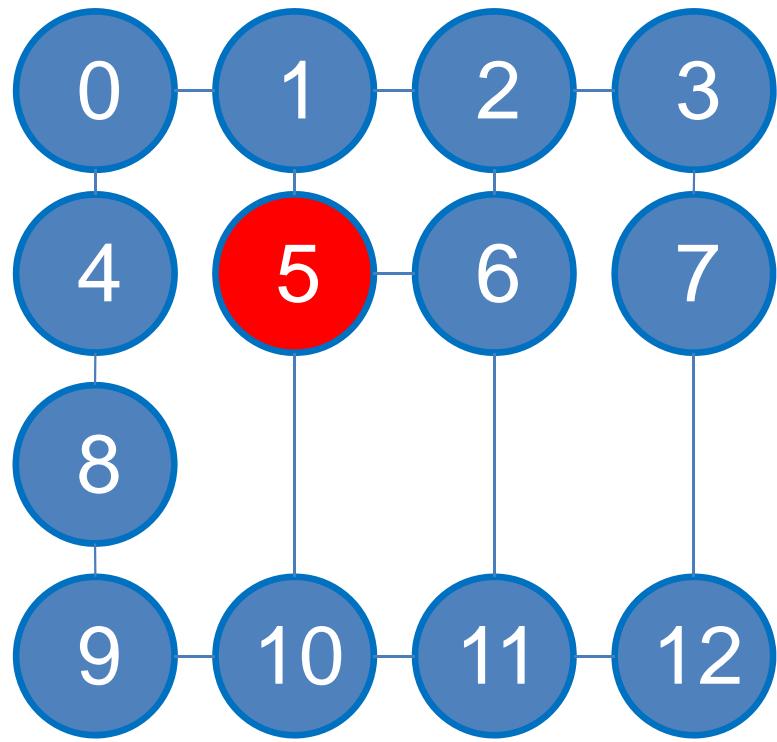


$$\begin{aligned} Q &= \{5\} \\ Q &= \{1, 6, 10\} \end{aligned}$$

$$\begin{aligned} D[5] &= 0 \\ D[1] &= D[5] + 1 = 1 \\ D[6] &= D[5] + 1 = 1 \\ D[10] &= D[5] + 1 = 1 \end{aligned}$$



# Example (3)



$$Q = \{5\}$$

$$Q = \{1, 6, 10\}$$

$$Q = \{6, 10, \textcolor{red}{0}, \textcolor{red}{2}\}$$

$$Q = \{10, 0, 2, \textcolor{red}{11}\}$$

$$Q = \{0, 2, 11, \textcolor{red}{9}\}$$

$$D[5] = 0$$

$$D[1] = D[5] + 1 = 1$$

$$D[6] = D[5] + 1 = 1$$

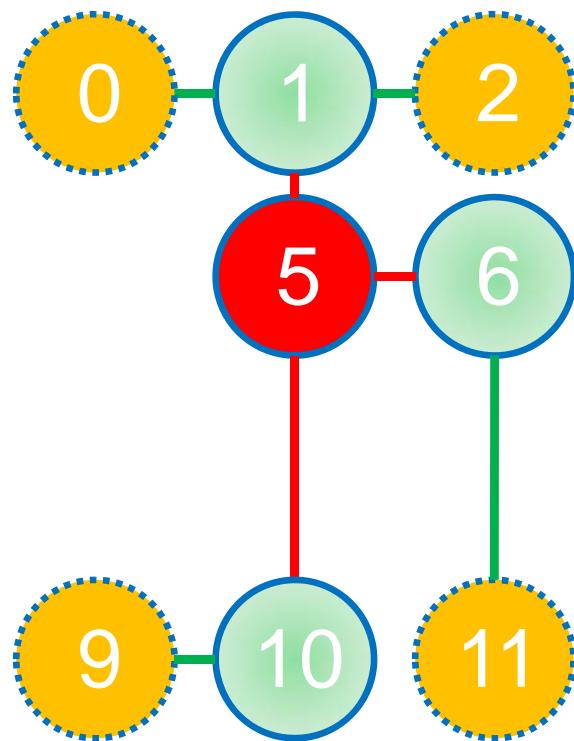
$$D[10] = D[5] + 1 = 1$$

$$D[0] = D[1] + 1 = 2$$

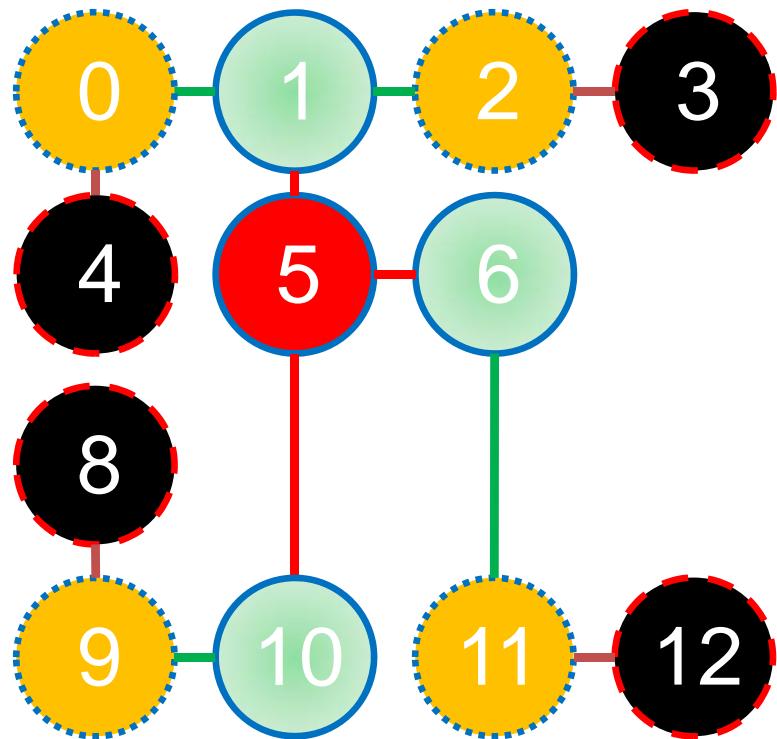
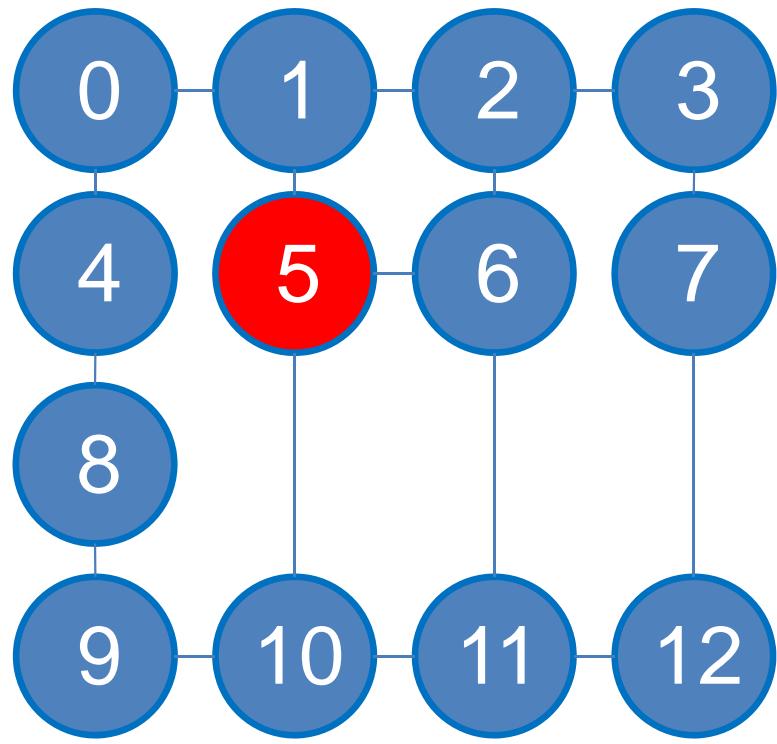
$$D[2] = D[1] + 1 = 2$$

$$D[11] = D[6] + 1 = 2$$

$$D[9] = D[10] + 1 = 2$$



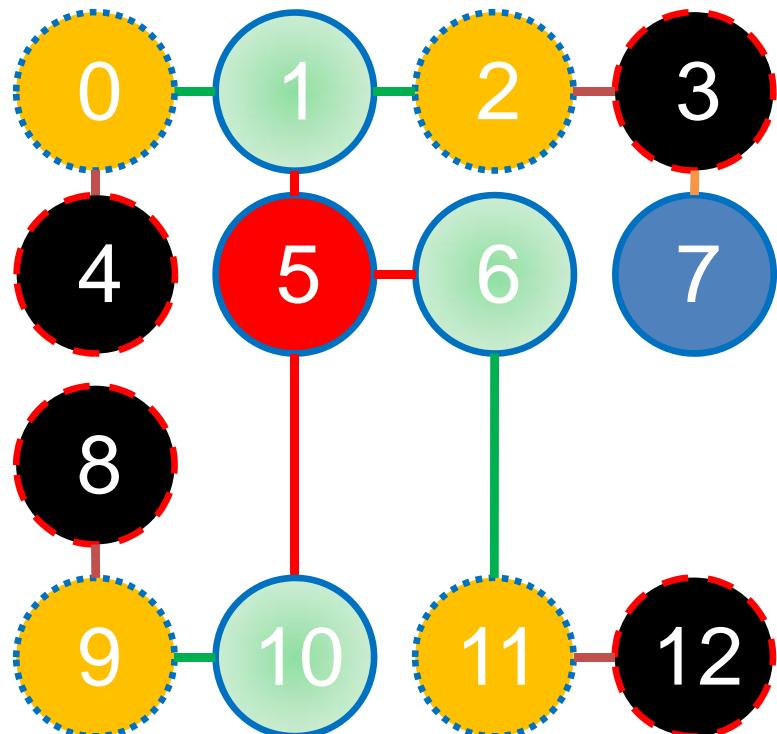
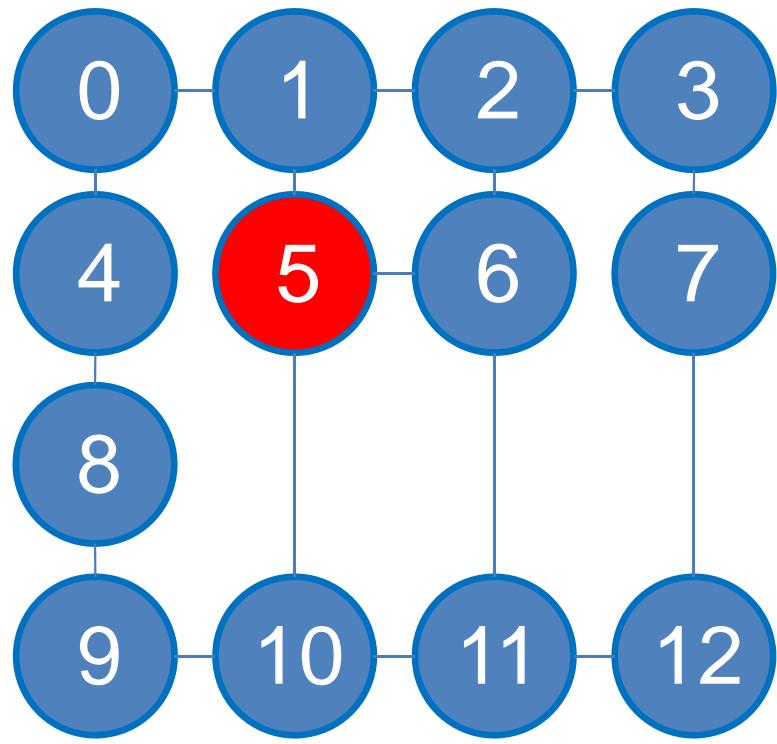
# Example (4)



$Q = \{5\}$   
 $Q = \{1, 6, 10\}$   
 $Q = \{6, 10, \textcolor{red}{0}, \textcolor{red}{2}\}$   
 $Q = \{10, 0, 2, \textcolor{red}{11}\}$   
 $Q = \{0, 2, 11, \textcolor{red}{9}\}$   
 $Q = \{2, 11, 9, \textcolor{red}{4}\}$   
 $Q = \{11, 9, 4, \textcolor{red}{3}\}$   
 $Q = \{9, 4, 3, \textcolor{red}{12}\}$   
 $Q = \{4, 3, 12, \textcolor{red}{8}\}$

$D[5] = 0$   
 $D[1] = D[5] + 1 = 1$   
 $D[6] = D[5] + 1 = 1$   
 $D[10] = D[5] + 1 = 1$   
 $D[0] = D[1] + 1 = 2$   
 $D[2] = D[1] + 1 = 2$   
 $D[11] = D[6] + 1 = 2$   
 $D[9] = D[10] + 1 = 2$   
 $D[4] = D[0] + 1 = 3$   
 $D[3] = D[2] + 1 = 3$   
 $D[12] = D[11] + 1 = 3$   
 $D[8] = D[9] + 1 = 3$

# Example (5)



$Q = \{5\}$   
 $Q = \{1, 6, 10\}$   
 $Q = \{6, 10, \textcolor{red}{0}, \textcolor{red}{2}\}$   
 $Q = \{10, 0, 2, \textcolor{red}{11}\}$   
 $Q = \{0, 2, 11, \textcolor{red}{9}\}$   
 $Q = \{2, 11, 9, \textcolor{red}{4}\}$   
 $Q = \{11, 9, 4, \textcolor{red}{3}\}$   
 $Q = \{9, 4, 3, \textcolor{red}{12}\}$   
 $Q = \{4, 3, 12, \textcolor{red}{8}\}$   
 $Q = \{3, 12, 8\}$   
 $Q = \{12, 8, \textcolor{red}{7}\}$   
 $Q = \{8, 7\}$   
 $Q = \{7\}$   
 $Q = \{\}$

$D[5] = 0$   
 $D[1] = D[5] + 1 = 1$   
 $D[6] = D[5] + 1 = 1$   
 $D[10] = D[5] + 1 = 1$   
 $D[0] = D[1] + 1 = 2$   
 $D[2] = D[1] + 1 = 2$   
 $D[11] = D[6] + 1 = 2$   
 $D[9] = D[10] + 1 = 2$   
 $D[4] = D[0] + 1 = 3$   
 $D[3] = D[2] + 1 = 3$   
 $D[12] = D[11] + 1 = 3$   
 $D[8] = D[9] + 1 = 3$   
 $D[7] = D[3] + 1 = 4$

This is the **BFS = SSSP  $\odot$  spanning tree** when BFS is started from vertex 5

For SSSP on Weighted Graph but without Negative Weight Cycle

# DIJKSTRA's

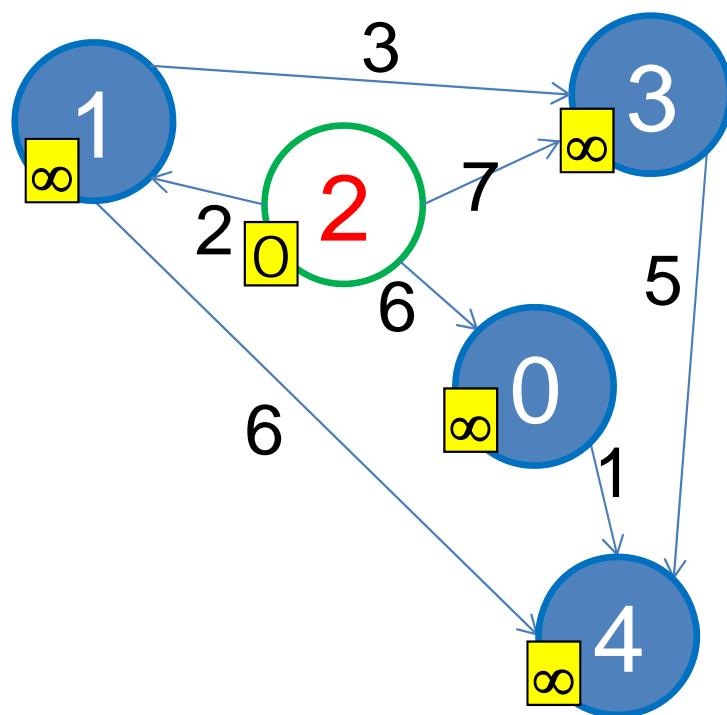


CS3233 - Competitive Programming,  
Steven Halim, SoC, NUS

# Single-Source Shortest Paths (1)

- If the graph is un **weighted**, we can use BFS
  - But what if the graph is **weighted**?
- [UVa 341](#) (Non Stop Travel)
- Solution: Dijkstra  $O((V+E) \log V)$ 
  - A Greedy Algorithm
  - Use Priority Queue

# Modified Dijkstra's – Example (1)

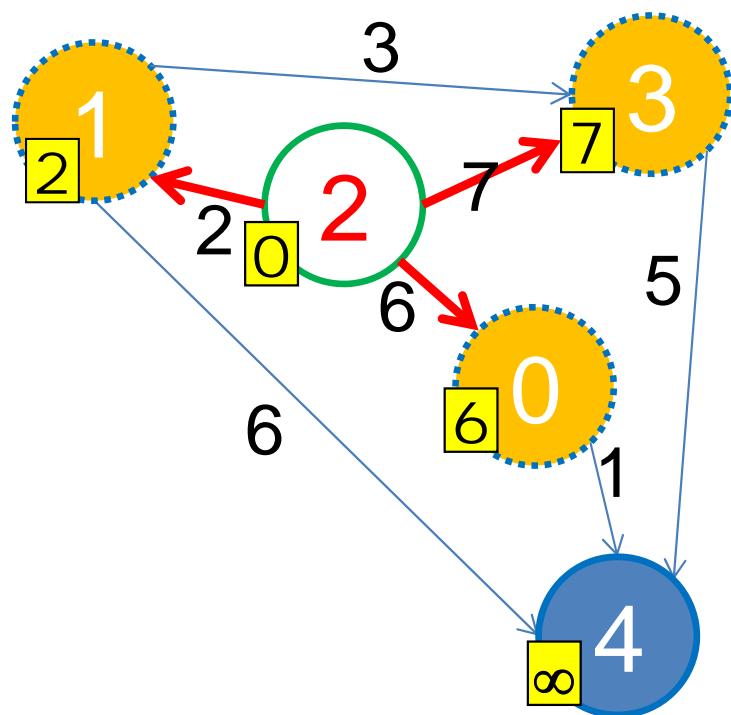


$$pq = \{(0, 2)\}$$

We store this pair of information to the priority queue:  $(D[\text{vertex}], \text{vertex})$ , sorted by increasing  $D[\text{vertex}]$ , and then if ties, by vertex number

See that our priority queue is “clean” at the beginning of (modified) Dijkstra’s algorithm, it only contains (0, the source s)

# Modified Dijkstra's – Example (2)



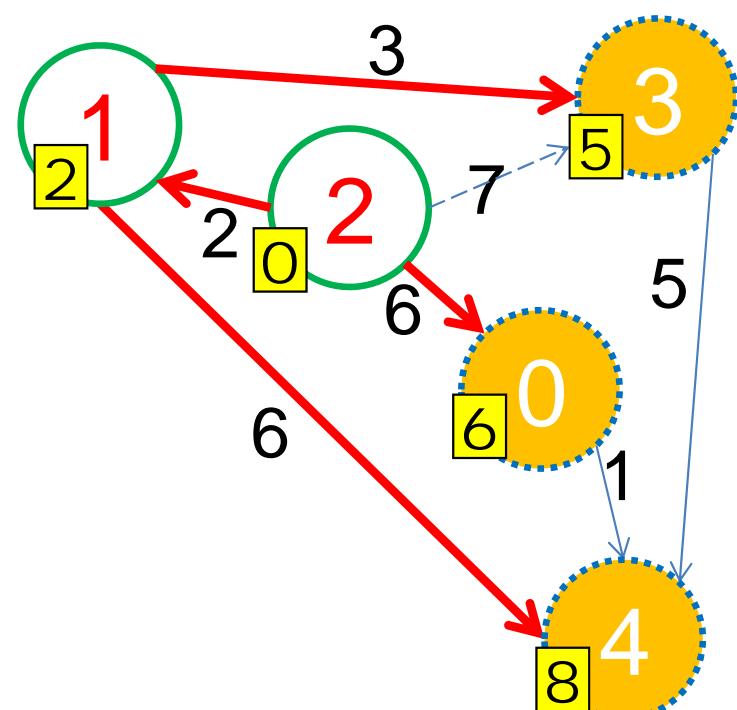
$$pq = \{(0, 2)\}$$

$$pq = \{(2, 1), (6, 0), (7, 3)\}$$

We greedily take the vertex in the front of the queue (here, it is vertex 2, the source), and then successfully relax all its neighbors (vertex 0, 1, 3).

Priority Queue will order these 3 vertices as 1, 0, 3, with shortest path estimate of 2, 6, 7, respectively.

# Modified Dijkstra's – Example (3)



Vertex 3 appears twice in the priority queue, but this does not matter, as we will take only the first (smaller) one

$$pq = \{(0, 2)\}$$

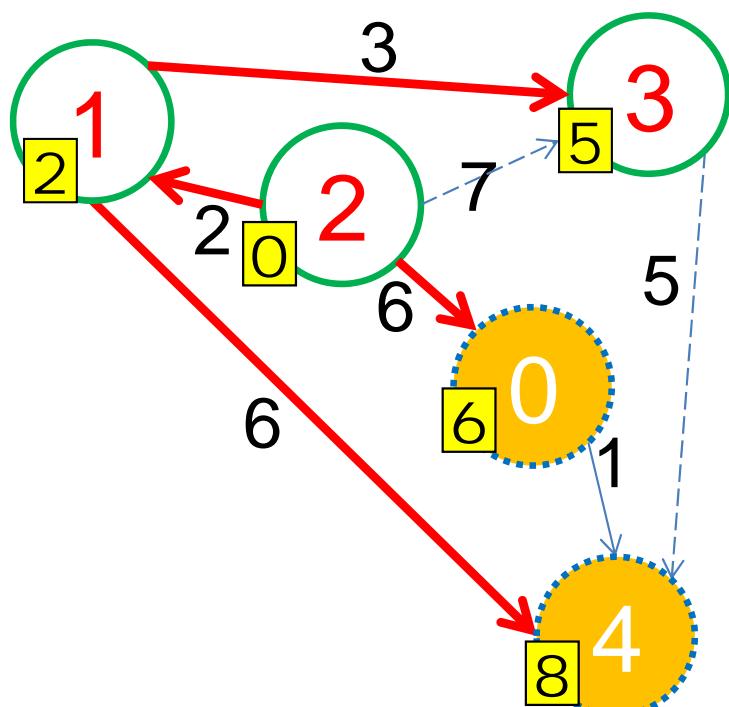
$$pq = \{(2, 1), (6, 0), (7, 3)\}$$

$$pq = \{(5, 3), (6, 0), (7, 3), (8, 4)\}$$

We greedily take the vertex in the front of the queue (now, it is vertex 1), then successfully relax all its neighbors (vertex 3 and 4).

Priority Queue will order the items as 3, 0, 3, 4 with shortest path estimate of 5, 6, 7, 8, respectively.

# Modified Dijkstra's – Example (4)



$$pq = \{(0, 2)\}$$

$$pq = \{(\cancel{2}, 1), (6, 0), (7, 3)\}$$

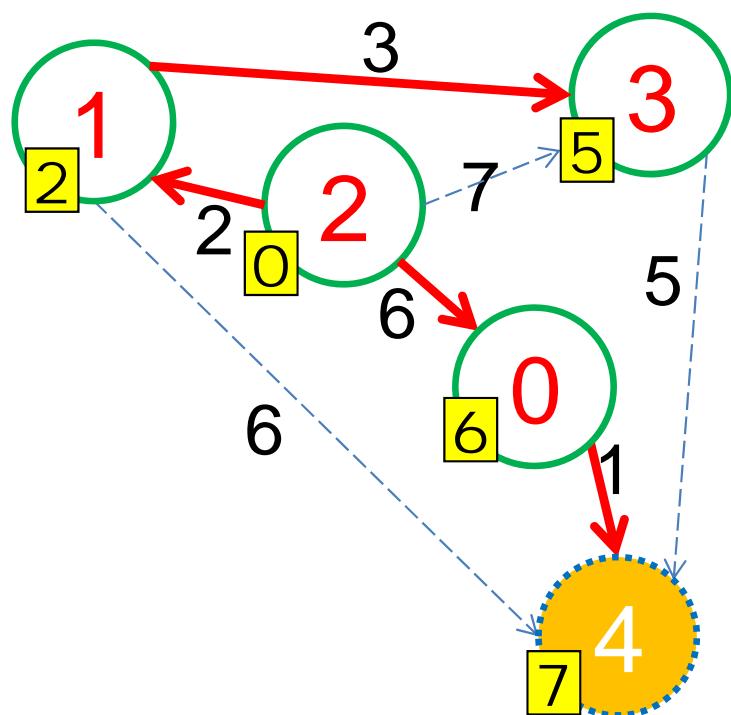
$$pq = \{(\cancel{5}, 3), (6, 0), (7, 3), (8, 4)\}$$

$$pq = \{(6, 0), (7, 3), (8, 4)\}$$

We greedily take the vertex in the front of the queue (now, it is vertex 3), then try to relax all its neighbors (only vertex 4). However  $D[4]$  is already 8. Since  $D[3] + w(3, 4) = 5 + 5$  is worse than 8, we do not do anything.

Priority Queue will now have these items 0, 3, 4 with shortest path estimate of 6, 7, 8, respectively.

# Modified Dijkstra's – Example (5)



$$pq = \{(0, 2)\}$$

$$pq = \{(2, 1), (6, 0), (7, 3)\}$$

$$pq = \{(5, 3), (6, 0), (7, 3), (8, 4)\}$$

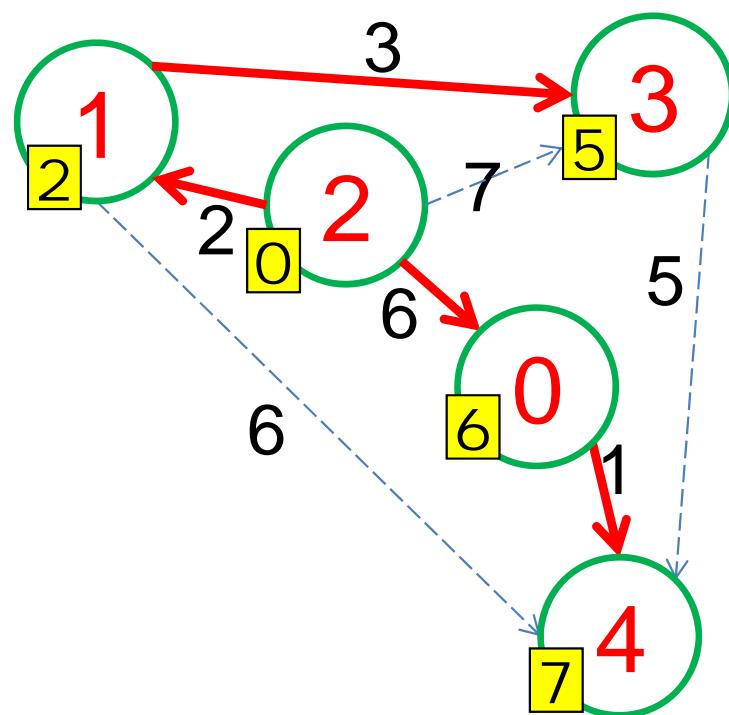
$$pq = \{(6, 0), (7, 3), (8, 4)\}$$

$$pq = \{(7, 3), (7, 4), (8, 4)\}$$

We greedily take the vertex in the front of the queue (now, it is vertex 5), then successfully relax all its neighbors (only vertex 4).

Priority Queue will now have these items 3, 4, 4 with shortest path estimate of 7, 7, 8, respectively.

# Modified Dijkstra's – Example (6)



Remember that vertex 3 appeared twice in the priority queue, but this Dijkstra's algorithm will only consider the first (shorter) one

$$pq = \{(0, 2)\}$$

$$pq = \{(2, 1), (6, 0), (7, 3)\}$$

$$pq = \{(5, 3), (6, 0), (7, 3), (8, 4)\}$$

$$pq = \{(6, 0), (7, 3), (8, 4)\}$$

$$pq = \{(7, 3), (7, 4), (8, 4)\}$$

$$pq = \{(7, 4), (8, 4)\}$$

$$pq = \{(8, 4)\}$$

$$pq = \{\}$$

Similarly for vertex 4. The one with shortest path estimate 7 will be processed first and the one with shortest path estimate 8 will be ignored, although nothing is changed anymore

# Dijkstra's Algorithm (using STL)

```
vi dist(V, INF); dist[s] = 0; // INF = 2B
priority_queue< ii, vector<ii>, greater<ii> > pq;
pq.push(ii(0, s)); // sort based on increasing distance
while (!pq.empty()) { // main loop
    ii top = pq.top(); pq.pop(); // greedy
    int d = top.first, u = top.second;
    if (d == dist[u]) {
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
            ii v = AdjList[u][j]; // all outgoing edges from u
            if (dist[u] + v.second < dist[v.first]) {
                dist[v.first] = dist[u] + v.second; // relax
                pq.push(ii(dist[v.first], v.first));
            } // enqueue this neighbor regardless it is
        } // already in pq or not
    }
}
```

For All-Pairs Shortest Paths

# FLOYD WARSHALL's



CS3233 - Competitive Programming,  
Steven Halim, SoC, NUS

# UVa 11463 – Commandos (1)

Al-Khawarizmi, Malaysia National Contest 2008

- Given:
  - A table that stores the amount of minutes to travel between buildings (there are **at most 100** buildings)
  - 2 special buildings: startB and endB
  - K soldiers to bomb all the K buildings in this mission
  - Each of them start at the same time from startB, choose one building B that has not been bombed by other soldier (bombing time negligible), and then gather in (destroyed) building endB.
- What is the minimum time to complete the mission?

# UVa 11463 – Commandos (2)

Al-Khawarizmi, Malaysia National Contest 2008

- How long do you need to solve this problem?
- Solution:
  - The answer is determined by sp from starting building, detonate **furthest building**, and sp from that furthest building to end building
    - $\max(\text{dist}[\text{start}][i] + \text{dist}[i][\text{end}])$  for all  $i \in V$
- How to compute **sp** for **many** pairs of vertices?

# UVa 11463 – Commandos (3)

Al-Khawarizmi, Malaysia National Contest 2008

- This problem is called: All-Pairs Shortest Paths
- Two options to solve this:
  - Call SSSP algorithms multiple times
    - Dijkstra  $O(V * (V+E) * \log V)$ , if  $E = V^2 \rightarrow O(V^3 \log V)$
    - Bellman Ford  $O(V * V * E)$ , if  $E = V^2 \rightarrow O(V^4)$
    - Slow to code
  - Use Floyd Warshall, a clever **DP** algorithm
    - $O(V^3)$  algorithm
    - Very easy to code!
    - In this problem,  $V$  is  $\leq 100$ , so Floyd Warshall is DOABLE!!

# Floyd Warshall – Template

- $O(V^3)$  since we have three nested loops!
- Use adjacency matrix:  $G [ MAX\_V ] [ MAX\_V ]$ ;
  - So that weight of edge( $i, j$ ) can be accessed in  $O(1)$

```
for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
```

- See more explanation of this three-liner DP algorithm in CP

Tree, Euler Graph, Directed Acyclic Graph (basics)

DAG is also re-visited (next week, Week 06)

Bipartite Graph (Week 08)

# **SPECIAL GRAPHS (Part 1)**

# Special Graphs in Contest

- 4 special graphs frequently appear in contest:
  - Tree, keywords: connected,  $E = V - 1$ , unique path!
  - Eulerian Graph, keywords: must visit each edge once
    - » Actually also rare now
  - Directed Acyclic Graph, keywords: no cycle
  - Bipartite, keywords: 2 sets, no edges within set!
    - » Currently not in IOI syllabus
- Some classical ‘hard’ problems may have *faster solution* on these special graphs
  - This allows problem setter to increase **input size!**
    - » Eliminates those who are not aware of the faster solution as solution for general graph is slower (TLE)  
or harder to code (slower to get AC)...



# TREE

CS3233 - Competitive Programming,  
Steven Halim, SoC, NUS

# Tree

- Tree is a special Graph. It:
  - Connected
  - Has  $V$  vertices and exactly  $E = V - 1$  edges
  - Has no cycle
  - Has one unique path between two vertices
  - Sometimes, it has one special vertex called “root”  
(rooted tree): Root has no parent
  - A vertex in n-ary tree has either  $\{0, 1, \dots, n\}$  children
    - $n = 2$  is called binary tree (most popular one)
  - Has many practical applications:
    - Organization Tree, Directories in Operating System, etc

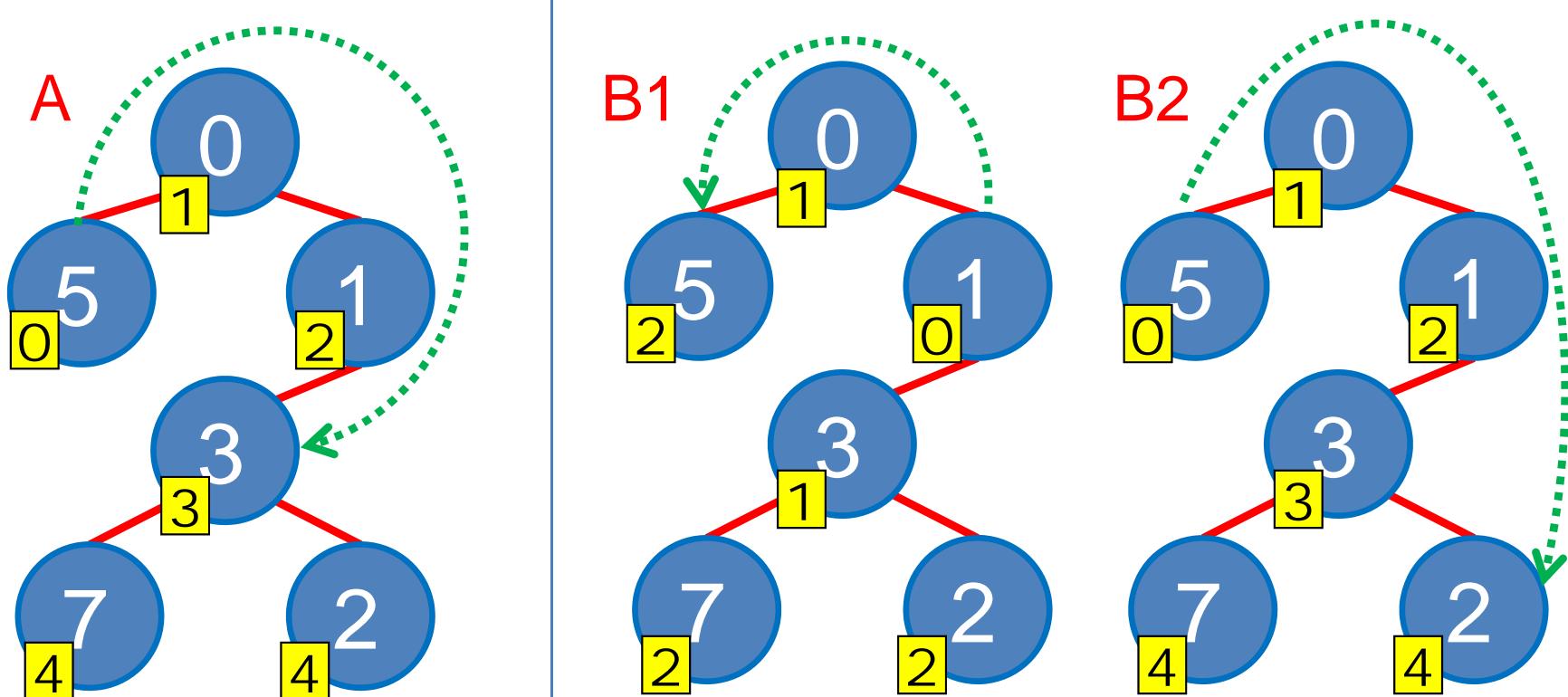
# SSSP and APSP Problems on Weighted Tree

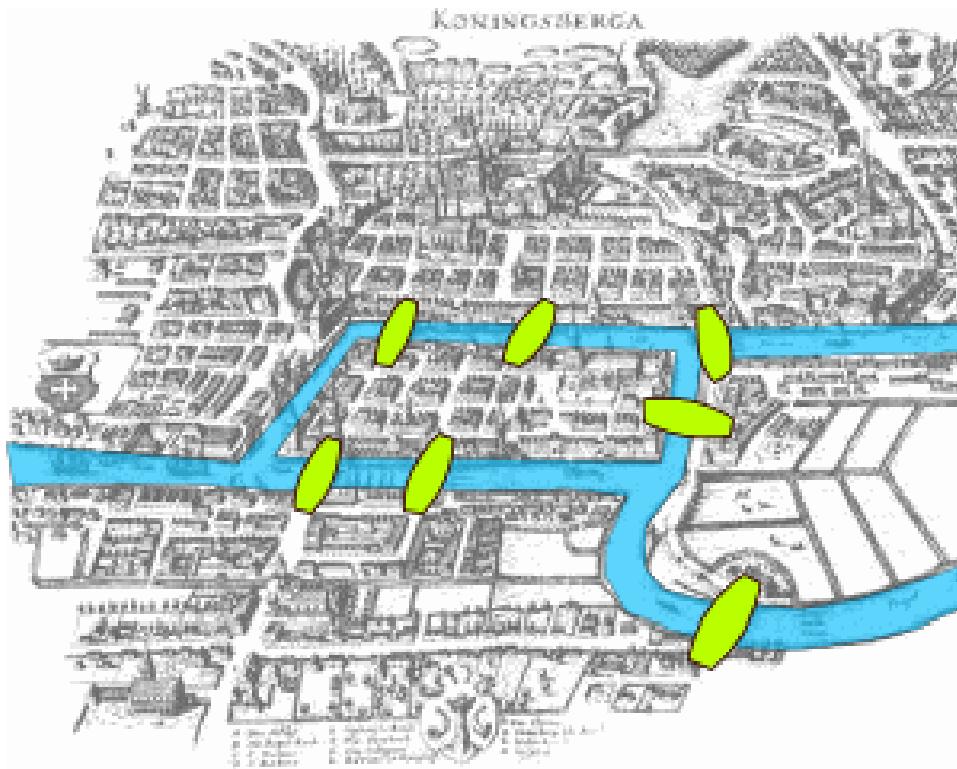
- In general weighted graph
  - SSSP problem:  $O((V+E) \log V)$  Dijkstra's or  $O(VE)$  Bellman Ford's
  - APSP problem:  $O(V^3)$  Floyd Warshall's
- In weighted tree
  - SSSP problem:  $O(V+E = V+V = V)$  DFS or BFS
    - There is only 1 unique path between 2 vertices in tree
  - APSP problem: simple  $V$  calls of DFS or BFS:  $O(V^2)$ 
    - But can be made even faster using LCA... not covered

# Diameter of a Tree

- In general weighted graph
  - We have to run  $O(V^3)$  Floyd Warshall's and pick the maximum over all  $\text{dist}[i][j]$  that is not INF
- In weighted tree
  - Do DFS/BFS twice!
    - From any vertex  $s$ , find furthest vertex  $x$  with DFS/BFS
    - Then from vertex  $x$ , find furthest vertex  $y$  with DFS/BFS
    - Answer is the path length of  $x-y$
    - $O(V+E = V+V = V)$  only – two calls of DFS/BFS

# Tree Illustration





# Euler Graph

# Eulerian Graph (1)

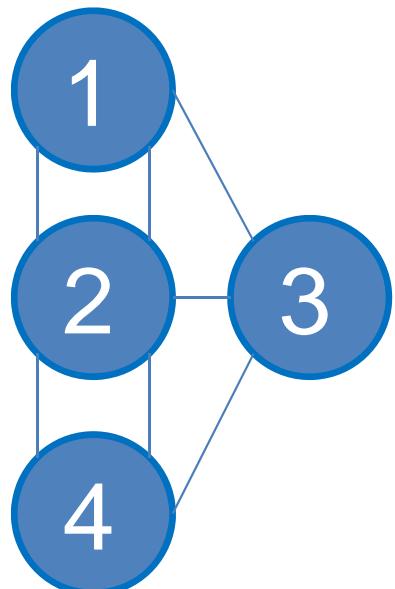
- An **Euler path** is defined as a path in a graph which visits each edge exactly once
- An **Euler tour/cycle** is an Euler path which starts and ends on the same vertex
- A graph which has either Euler path or Euler tour is called Eulerian graph

# Eulerian Graph (2)

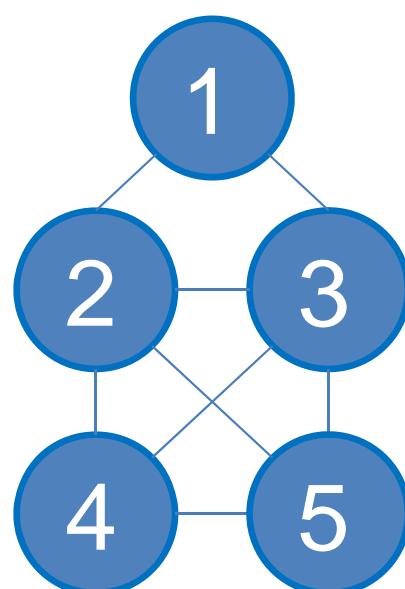
- To check whether an undirected graph has an Euler tour is **simple** 😊
  - Check if all its vertices have even degrees.
- Similarly for the Euler path
  - An undirected graph has an Euler path if all except two vertices have even degrees and at most two vertices have odd degrees. This Euler path will start from one of these odd degree vertices and end in the other
- Such degree check can be done in  $O(V + E)$ , usually done simultaneously when reading the input graph

# Eulerian Graph (3)

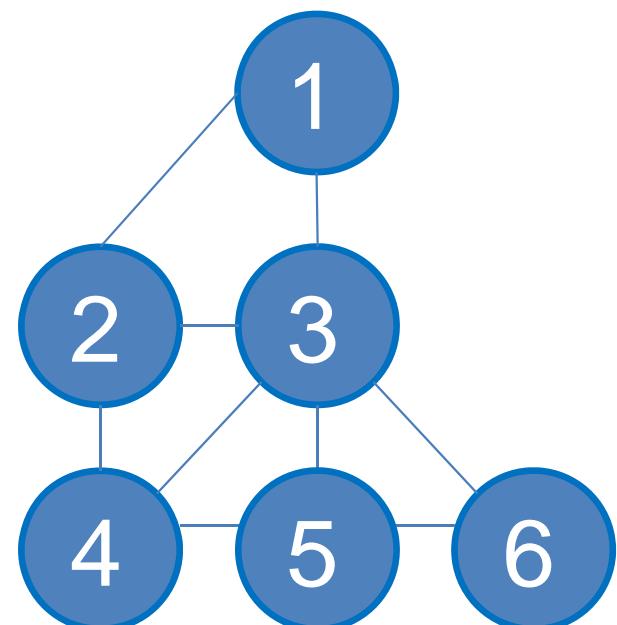
Königsberg  
Non Eulerian

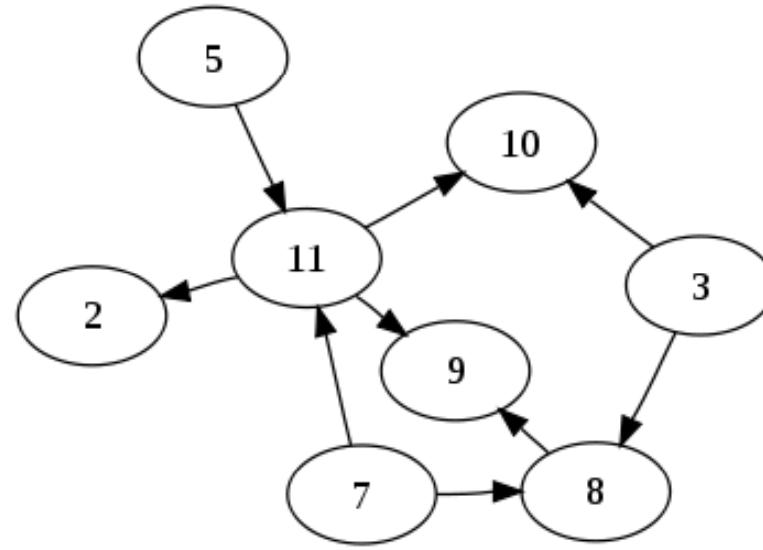


UVa 291  
Eulerian



Non Eulerian





# DIRECTED ACYCLIC GRAPH (DAG)

# Directed Acyclic Graph (DAG)

- Some algorithms become simpler when used on DAGs instead of general graphs, based on the principle of topological ordering
- For example, it is possible to find shortest paths and longest paths from a given starting vertex in DAGs in **linear time** by processing the vertices in a **topological order**, and calculating the path length for each vertex to be the minimum or maximum length obtained via any of its incoming edges
- In contrast, for arbitrary graphs the shortest path may require slower algorithms such as Dijkstra's algorithm or the Bellman-Ford algorithm, and longest paths in arbitrary graphs are NP-hard to find

# Single-Source Shortest Paths in DAG

- In general weighted graph
  - Again this is  $O((V+E) \log V)$  using Dijkstra's or  $O(VE)$  using Bellman Ford's
- In DAG
  - The fact that there is no cycle simplifies this problem substantially!
    - Simply “relax” vertices according to topological order! This ensure shortest paths are computed correctly!
    - One Topological sort can be found in  $O(V+E)$

# Single-Source Longest Paths in DAG

- In general weighted graph
  - Longest (simple) paths is an NP complete problem
- In DAG
  - The solution is the same as shortest paths in DAG, just that we have tweak the relax operator (or alternatively, negate all edge weight in DAG)

# Summary (1)

- Today, we have *quickly* gone through various well-known graph problems & algorithms
  - Depth First Search and Breadth First Search
    - Connected versus **Strongly** Connected Components
  - Kruskal's for MST (briefly)
  - Shortest Paths problems
    - BFS (unweighted), Dijkstra's (standard), Floyd Warshall's (all-pairs, three liners)
  - Special Graph: Tree, Eulerian, DAG

# Summary (2)

- Note that just knowing these algorithm will not be too useful in contest setting...
- You have to practice **using them**
  - At least code each of the algorithms discussed today on a contest problem!

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 06 – Problem Solving Paradigms  
(Dynamic Programming 2)

# Outline

- Mini Contest #5 + Break + Discussion + Admins
- A simple problem to refresh our memory about DP paradigms
- DP and its relationship with DAG (Chapter 4)
  - DP on Explicit DAG/Implicit DAG
    - These are CS2020/CS2010 materials
    - Those from CS1102 must catch up/consult Steven separately
- DP on Math Problems (Chapter 5)
- DP on String Problems (Chapter 6)
- DP + bitmask (Chapter 8)
- Compilation of Common DP States

DP Problems on Chapter 4-5-6 of CP2 Book

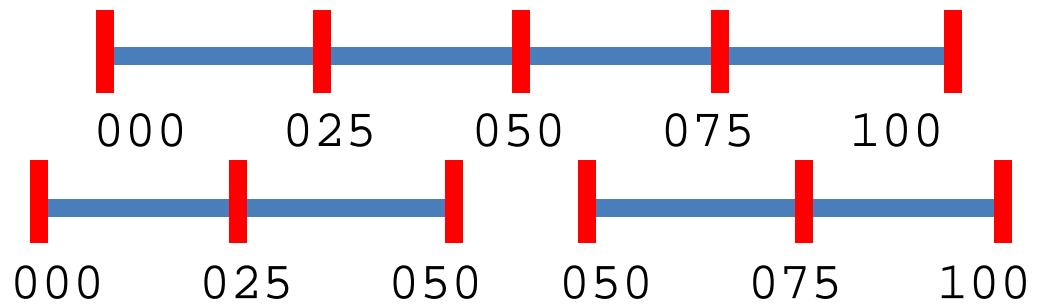
# **NON CLASSICAL DP PROBLEMS**

# Non Classical DP Problems (1)

- My definition of *Non Classical DP* problems:
  - Not the pure form/variant of LIS, Max Sum, Coin Change, 0-1 Knapsack/Subset Sum, TSP where the DP **states** and **transitions** can be “memorized”.
  - Requires **original formulation** of DP states and transitions
    - Although some formulation are still “similar” to the classical ones
  - Throughout this lecture, we will talk mostly in *DP terms*
    - **State** (to be precise: “*distinct state*”)
    - **Space Complexity** (i.e. the number of distinct states)
    - **Transition** (which entail overlapping sub problems)
    - **Time Complexity** (i.e. num of distinct states \* time to fill one state)

# Non Classical DP Problems (2)

- To refresh our memory 😊
- Example: Cutting Sticks ([UVa 10003](#)) – in CLRS 3<sup>rd</sup> ed!
  - State:  $dp[l][r]$ , Q: Why these two parameters?
  - Space Complexity: Max  $50 \times 50 = 2500$  distinct states
  - Transition: Try all possible cutting points **m** between **l** and **r**,
    - i.e. cut  $(l, r)$  into  $(l, m)$  and  $(m, r)$
  - Time Complexity: There can be up to 50 possible cutting points, thus max  $2500 \times 50 = 125000$  operations, do able 😊



# DP on DAG (1)

## Overview

- Dynamic Programming (DP) has a close relationship with (sometimes implicit) Directed Acyclic Graph (DAG)
  - The **states** are the **vertices** of the DAG
  - Space complexity: Number of vertices of the DAG
  - The **transitions** are the **edges** of the DAG
    - Logical, since a recurrence is always **acyclic**
  - Time complexity: Number of edges of the DAG
  - Top-down DP: Process each vertex just once via **memoization**
  - Bottom-up DP: Process the vertices in **topological order**
    - Sometimes, the topological order can be written by just using simple (nested) loops

# DP on DAG (2)

## On Explicit DAG (1)

- There is a DAG in this problem (can you spot it?)
  - [UVa 10285](#) (Longest Run on a Snowboard)
- Given a  $R*C$  of height matrix  $h$ ,  
find what is the longest run possible?
  - Longest run of length  $k$ :
    - $h(i_1, j_1) > h(i_2, j_2) > \dots > h(i_k, j_k)$
  - **Longest path in DAG!**
- We will learn a creative DP table filling technique  
from this short example (bottom up)

# DP on DAG (2)

## On Explicit DAG (2)

- Complete search recurrence:
  - Let  $h(i, j) = \text{height of location } (i, j)$
  - Let  $f(i, j) = \text{longest run length started at } (i, j)$
  - $f(i, j) = 1 + \max(f \text{ values of reachable neighbors: } h(i, j) > h(\text{neighbors' } i, j))$ 
    - Base case: If  $h(i, j)$  is the lowest among its neighbor, then  $f(i, j) = 1$
- Do it from all possible  $R*C$  locations!
  - Do you observe many overlapping sub problems?
- This problem is actually solvable using complete search due to “small”  $R*C$  + constraints!
  - But with DP, the solution is more efficient
    - And it can be used to solve larger  $R*C$

# DP on DAG (2)

## On Explicit DAG (3)

- Fill-in-the-Table (Bottom-Up) Trick:
  - Sort  $(i, j)$  according to  $h(i, j)$  in **increasing order**
    - i.e. Fill the bottom cells first!
    - Fill in  $f(i, j)$  in order based on the value of  $f(i-1, j), f(i+1, j), f(i, j-1), f(i, j+1)$ 
      - This is the topological sort!
  - This may actually be harder to code
    - For this problem, writing the DP solution in top-down fashion may be better/easier

# DP on DAG (3)

## Converting a General Graph → DAG (1)

- Not every graph problem has a ready algorithm for it
  - Some have to be solved with Complete Search...
  - Some are solvable with DP, as long as we can find a way to make the **DP transition acyclic**, usually by adding one (or more) parameters to each vertex :O
  - Remember that Dijkstra's is a greedy algorithm, and every greedy solution (without proof of correctness) has a chance for getting WA...
- Example: [Fishmonger \(SPOJ 101\)](#)
  - State:  $dp[pos][time\_left]$ , Q: Why these two parameters?
  - Space Complexity:  $\text{Max } 50 * 1000 = 50000$
  - Transition: Try all remaining N cities, notice that  $time\_left$  always decreases as we move from one city to another (**acyclic!!**)
  - Time Complexity:  $\text{Max } 50000 * 50 = 2.5 \text{ M}$ , doable ☺



# DP on Math Problems (Chapter 5)

- Some well-known mathematic problems involves DP
  - Some combinatorics problem have recursive formulas which entail overlapping subproblems
    - e.g. those involving Fibonacci number,  $f(n) = f(n - 1) + f(n - 2)$
  - Some probability problems require us to search the entire search space to get the required answer
    - If some of the sub problems are overlapping, use DP, otherwise, use complete search
  - Mathematics problems involving **static** range sum/min/max!
    - Use dynamic tree DS for dynamic queries

# DP on String Problems (Chapter 6)

- Some string problems involves DP
  - Usually, we do not work with the string itself
    - Too costly to pass (sub)strings around as function parameters
  - But we work with the integer indices to represent suffix/prefix/substring
- Example: [UVa 11258 – String Partition](#)
  - There are many ways to split a string of digits into a list of non-zero-leading (0 itself is allowed) 32-bit *signed* integers
    - That is, max integer is  $2^{31}-1 = 2147483647$
  - What is the maximum sum of the resultant integers if the string is split appropriately?

Last part of this lecture, from Chapter 8 of CP2 Book

# **DP + BITMASK AND TIPS/TRICKS**

# Emerging Technique: DP + bitmask

- We have seen this form earlier in DP-TSP
- Bitmask technique can be used to represent *lightweight set of Boolean* (up to  $2^{64}$  if using unsigned long long)
- Important if one of the DP parameter is a “set”
- Can be used to solve ***matching in small general graph***
- Example: Forming Quiz Teams ([UVa 10911](#))
  - State:  $dp[\text{bitmask}]$
  - Space Complexity:  $2^M \sim 65K$  distinct sub problems;  
 $\max N = 8, M = 2N$ , thus  $\max M = 16$
  - Transition: Clever version:  $O(N)$ , try all and break as soon as we find the first perfect matching, Not so clever:  $O(N^2)$
  - Time Complexity:  $O(N \cdot 2^M)$  or about 524K, doable

# Common DP States (1)

- Position:
  - Original problem:  $[x_1, x_2, \dots, x_n]$ 
    - Can be sequence (integer/double array), can be string (char array)
  - Sub problems, break the original problem into
    - Sub problem and Prefix:  $[x_1, x_2, \dots, x_{n-1}] + x_n$
    - Suffix and sub problem:  $x_1 + [x_2, x_3, \dots, x_n]$
    - Two sub problems:  $[x_1, x_2, \dots, x_i] + [x_{i+1}, x_{i+2}, \dots, x_n]$
  - Example: LIS, 1D Max Sum, Matrix Chain Multiplication (MCM), etc

# Common DP States (2)

- **Positions:**
  - This is similar to the previous slide
  - Original problem:  $[x_1, x_2, \dots, x_n]$  and  $[y_1, y_2, \dots, y_n]$ 
    - Can be two sequences/strings
  - Sub problems, break the original problem into
    - Sub problem and prefix:  $[x_1, x_2, \dots, x_{n-1}] + x_n$  and  $[y_1, y_2, \dots, y_{n-1}] + y_n$
    - Suffix and sub problem:  $x_1 + [x_2, x_3, \dots, x_n]$  and  $y_1 + [y_2, y_3, \dots, y_n]$
    - Two sub problems:  $[x_1, x_2, \dots, x_i] + [x_{i+1}, x_{i+2}, \dots, x_n]$  and  $[y_1, y_2, \dots, y_i] + [y_{i+1}, y_{i+2}, \dots, y_n]$
  - Example: String Alignment/Edit Distance, LCS, etc
  - PS: Can also be applied on 2D matrix, like 2D Max Sum, etc

# Tips: When to Choose DP

- Default Rule:
  - If the given problem is an **optimization** or **counting** problem
    - Problem exhibits optimal sub structures
    - Problem has overlapping sub problems
- In ICPC/IOI:
  - If actual solutions are not needed (only final values asked)
    - If we must compute the solutions too, a more complicated DP which stores *predecessor information* and *some backtracking* are necessary
  - The number of distinct sub problems is small enough (< 1M) and you are not sure whether greedy algorithm works (why gamble?)
  - Obvious overlapping sub problems detected :O

# Dynamic Programming Issues (1)

- Potential issues with DP problems:
  - They may be disguised as (or looks like) non DP
    - It looks like greedy can work but some cases fails...
      - e.g. problem looks like a shortest path with some constraints on graph, but the constraints fail *greedy* SSSP algorithm!
  - They may have subproblems but not overlapping
    - DP does not work if overlapping subproblems not exist
      - Anyway, this is still a good news as perhaps Divide and Conquer technique can be applied

# Dynamic Programming Issues (2)

- Optimal substructures may not be obvious
  1. Find correct “states” that describe problem
    - Perhaps extra parameters must be introduced?
  2. Reduce a problem to (smaller) sub problems (with the same states) until we reach base cases
- There can be more than one possible formulation
  - Pick the one that works!

# DP Problems in ICPC (1)

- The number of problems in ICPC that must be solved using DP are growing!
  - At least one, likely two, maybe three per contest...
- These new problems are **not** the classical DP!
  - They require deep thinking...
  - Or those that look solvable using other (simpler) algorithms but actually must be solved using DP
  - Do not think that you have “mastered” DP by only memorizing the classical DP solutions!

# DP Problems in ICPC (2)

- In 1990ies, mastering DP can make you “king” of programming contests...
  - Today, it is a must-have knowledge...
  - So, get familiar with DP techniques!
- By mastering Graph + DP, your ICPC rank is probably:
  - from top ~[25-30] (solving 1-2 problems out of 10)
    - Only easy problems
  - to top ~[10-20] (solving 3-5 problems out of 10)
    - Easy problems + some graph + greedy/DP problems

For Week 07 homework 😊  
(You can do this over recess week too)

## BE A PROBLEM SETTER

# Be a Problem Setter

- Problem Solver:
  - A. Read the problem
  - B. Think of a good algorithm
  - C. Write ‘solution’
  - D. Create tricky I/O
  - E. If WA, go to A/B/C/D
  - F. If TLE/MLE, go to A/B/C/D
  - G. If AC, stop ☺
- Problem Setter:
  - A. Write a good problem
  - B. Write good solutions
    - The correct/best one
    - The incorrect/slower ones
  - C. Set a good secret I/O
  - D. Set problem settings
- A problem setter must think from a different angle!
  - By setting good problems, you will simultaneously be a better problem solver!!

# Problem Setter Tasks (1)

- Write a good problem
  - Options:
    - Pick an algorithm, then find problem/story, or
    - Find a problem/story, then identify a good algorithm for it (harder)
  - Problem description must not be ambiguous
    - Specify input constraints
    - Good English!
    - Easy one: longer, Hard one: shorter!
- Write good solutions
  - Must be able to solve your own problem!
    - To set hard problem, one must increase his own programming skill!
  - Use the best possible algorithm with lowest time complexity
    - Use the inferior ones ‘that barely works’ to set the WA/TLE/MLE settings...

# Problem Setter Tasks (2)

- Set a good secret I/O
  - Tricky test cases to check AC vs WA
    - Usually ‘boundary case’
  - Large test cases to check AC vs TLE/MLE
    - Perhaps use input generator to generate large test case, then pass this large input to our correct solution
- Set problem settings
  - Time Limit:
    - Usually 2 or 3 times the timings of your own best solutions
    - Java slower than C++!
  - Memory Limit:
    - Check OJ setting^
  - Problem Name:
    - Avoid revealing the algorithm in the problem name

# FYI: Be A Contest Organizer

- Contest Organizer Tasks:
  - Set problems of *various* topic
    - Better set by >1 problem setter
  - Must balance the difficulty of the problem set
    - Try to make it fun
    - Each team solves some problems
    - Each problem is solved by some teams
    - No team solve all problems
      - Every teams must work until the end of contest

# Special Homework for Week07

- Create **one** problem of your choice
  - Can be of **any** problem type
    - Regardless we have studied that in CS3233 or not
    - For those who are new with Competitive Programming, just create one from these: Ad Hoc/Libraries/BF/D&C/Greedy/DP/simple Graph
    - You can do this in advance (use your mid sem break?)
- Deliverables:
  - 1 html: problem description + sample I/O
  - 1 source code: cpp/java
  - 1 test data file (of multiple instances type)
  - 1 short txt write up of the expected solution
  - Zip and upload your problem into special folder in IVLE
    - CS3233 / Homework / Be A Problem Setter

# More References

- **Competitive Programming 2**
  - Section 3.5, 4.7.1, 5.4, 5.6, 6.5, and 8.4
- **Introduction to Algorithms**, p323-369, Ch 15
- **Algorithm Design**, p251-336, Ch 6
- **Programming Challenges**, p245-267, Ch 11
- <http://www.topcoder.com/>  
tc?module=Static&d1=tutorials&d2=dynProg
- Best is practice & more practice!

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

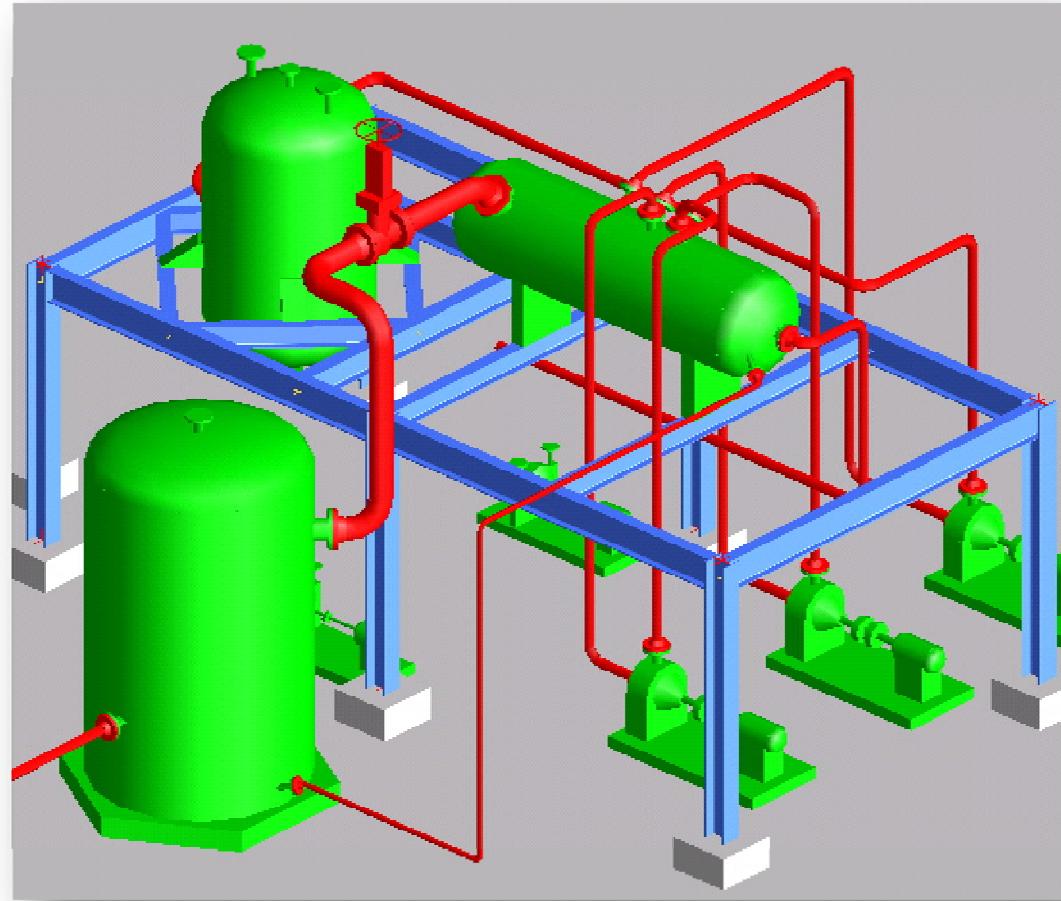
Dr. Steven Halim

Week 08 – Max Flow and Bipartite Graph  
(both are *not* in IOI syllabus 2009)

# Outline

- Mini Contest #6 + Break + Discussion + Admins
- Max Flow
  - (We will skip the many variants → read the book!)
- Special Graphs 2
  - Bipartite Graph
  - Finding the bipartite graph in a problem
  - The important Alternating Path algorithm





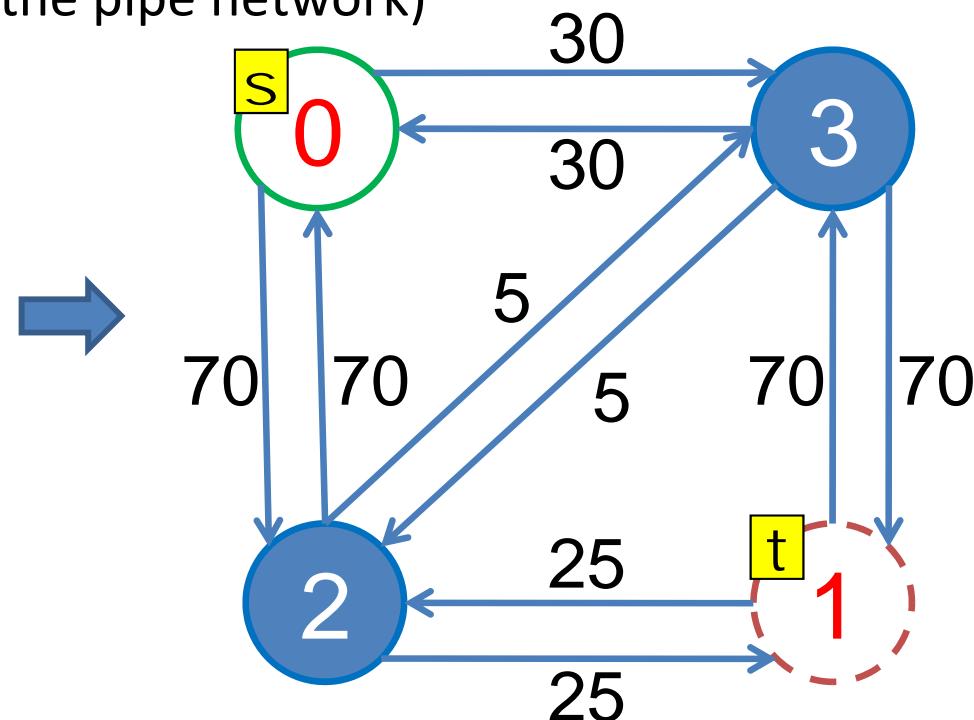
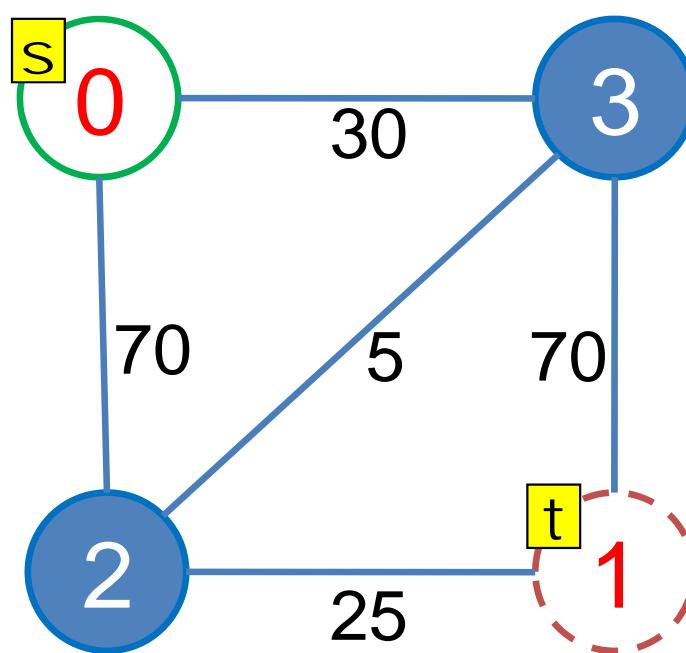
# MAX FLOW

# Motivation

- How to solve this UVa problem:
  - [820](#) (Internet Bandwidth)
    - Similar problems: 259, 753, 10092, 10480, 10511, etc
- Without **Max Flow** algorithms, they look “hard”

# Max Flow in a Network

- Imagine **connected, weighted, directed** graph as *pipe network*
  - The edges are the pipes
  - The vertices are the splitting points
  - There are also two special vertices: source **s** & sink **t**
  - What is the **max flow** (rate) from source **s** to sink **t** in this graph?  
(imagine water flowing in the pipe network)

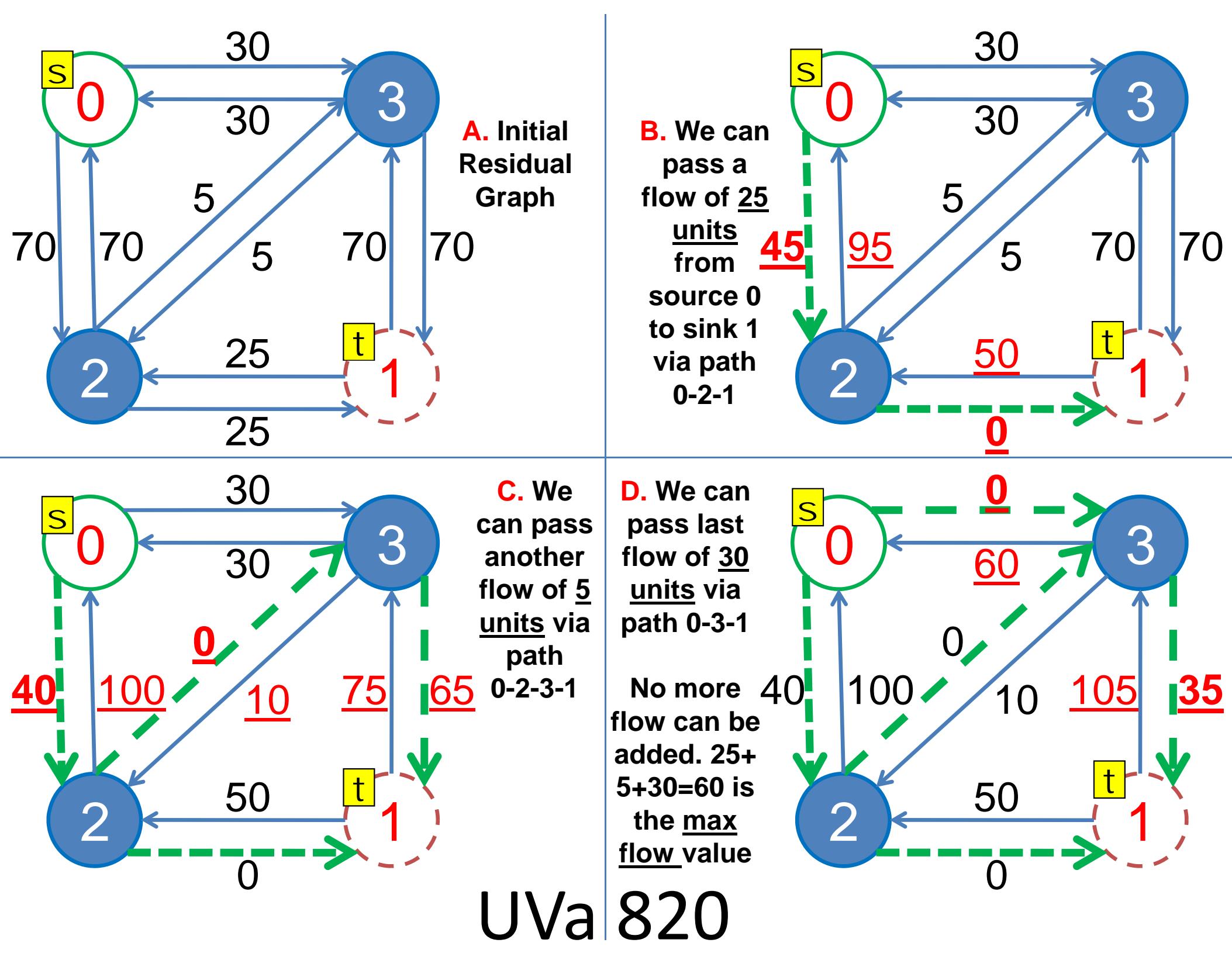


# Maximum Flow in Network

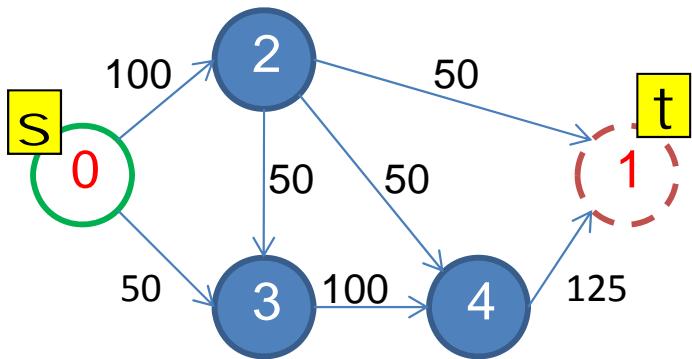
- One Solution: [Ford Fulkerson's Method](#)



- A surprisingly **simple** *iterative* algorithm
  - Send a flow through path  $p$  whenever there exists an augmenting path  $p$  from  $s$  to  $t$ 
    - *Augmenting path is a path from source  $s$  to sink  $t$  that pass through positive edges in residual graph*

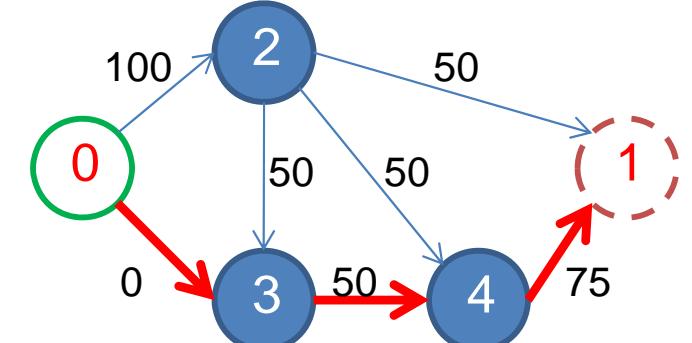


# What is the Max Flow value? (1)

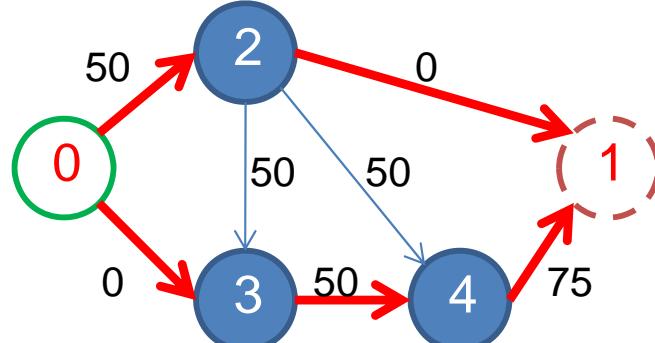


Answer: Maximum flow = 150

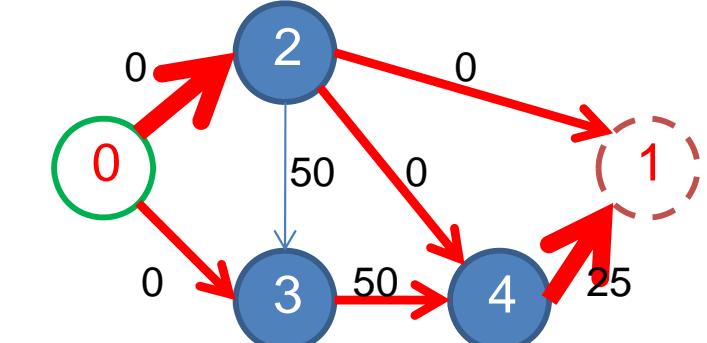
Flow so far = 50



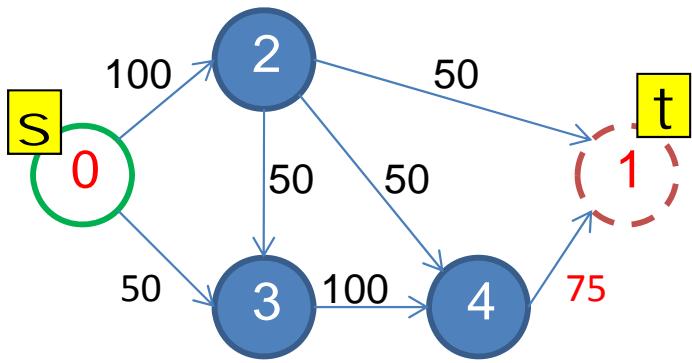
Flow so far = 100



Flow so far = 150

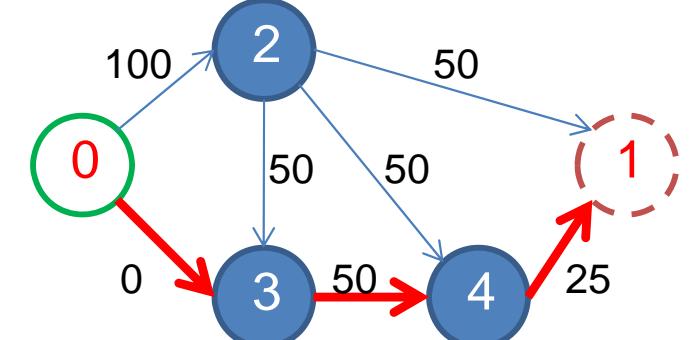


# What is the Max Flow value? (2)

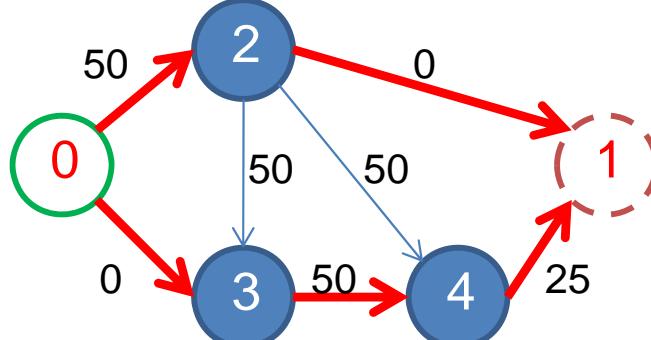


Answer: Maximum flow = 125

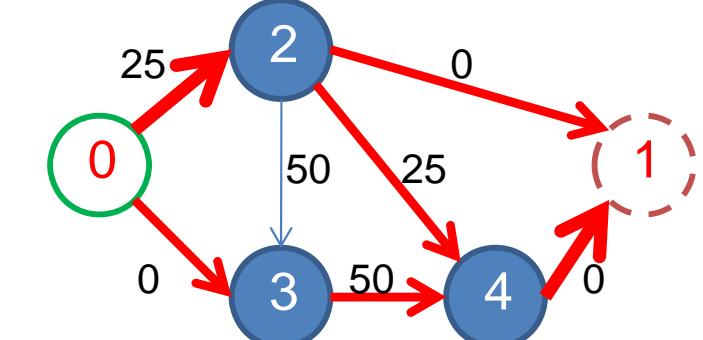
Flow so far = 50



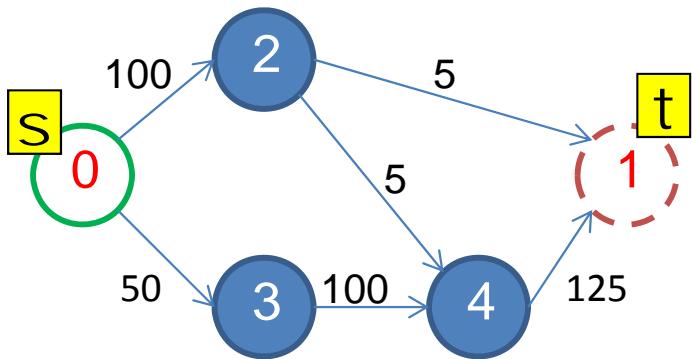
Flow so far = 100



Flow so far = 125

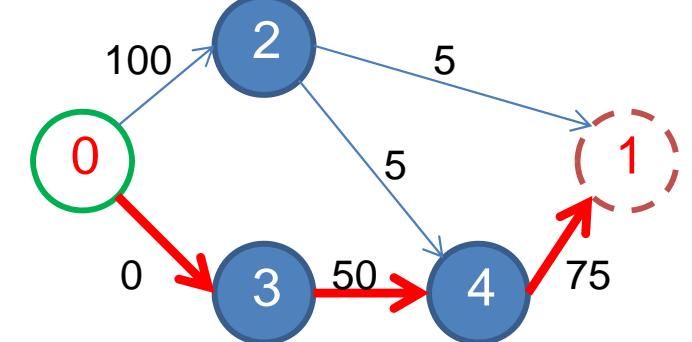


# What is the Max Flow value? (3)

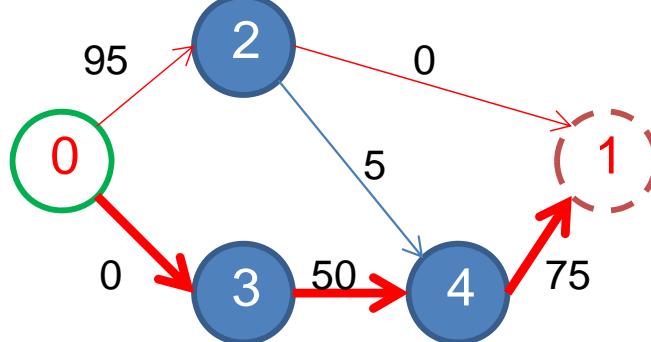


Answer: Maximum flow = 60

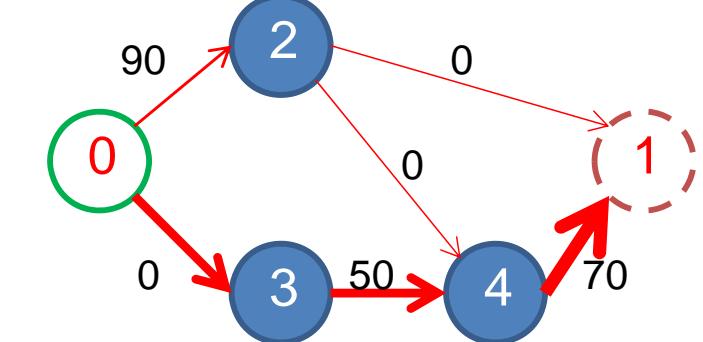
Flow so far = 50



Flow so far = 55



Flow so far = 60



# Ford Fulkerson's Method

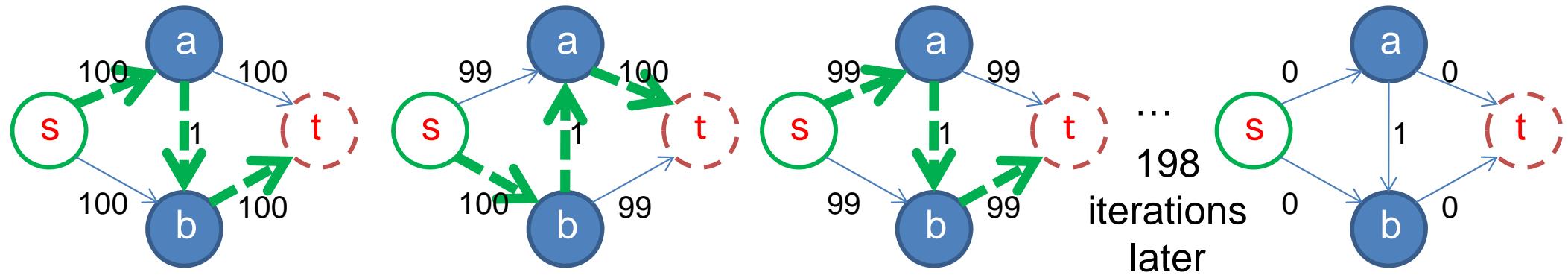
setup directed residual graph

each edge has the same weight with the original graph

```
mf = 0 // this is an iterative algorithm, mf stands for max_flow
while (there exists an augmenting path p from s to t) {
    // p is a path from s to t that pass through positive edges in residual graph
    augment/send flow f along the path p (s -> ... -> i -> j -> ... t)
    1. find f, the min edge weight along the path p
    2. decrease the weight of forward edges (e.g. i -> j) along path p by f
        reason: obvious, we use the capacities of those forward edges
    3. increase the weight of backward edges (e.g. j -> i) along path p by f
        reason: not so obvious, but this is important for the correctness of Ford
        Fulkerson's method; by increasing the weight/capacity of a backward edge
        (j -> i), we allow later iteration/flow to cancel part of weight/capacity
        used by a forward edge (i -> j) if not all capacity of edge (i -> j) is
        used in the final set of paths that produce the max flow
    mf += f // we can send a flow of size f from s to t, increase mf
}
output mf
```

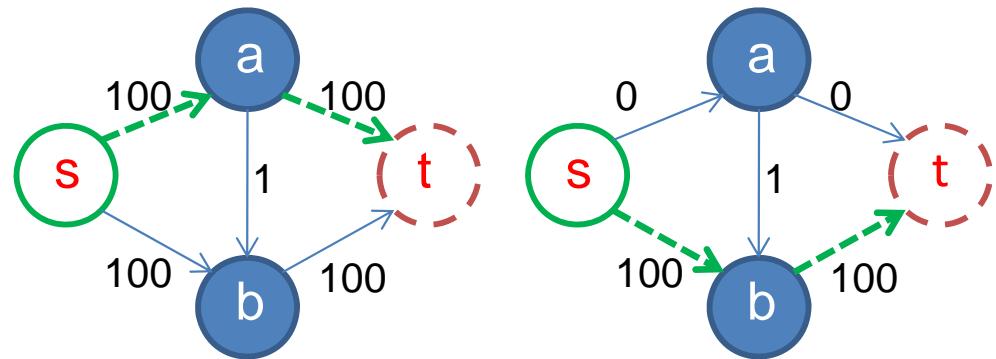
# DFS Implementation

- DFS implementation of Ford Fulkerson's method runs in  $O(|f^*|E)$  and can be very slow on graph like this:
  - Notice the presence of backward edges (drawn this time)
    - Q: What if we do not use backward edges?

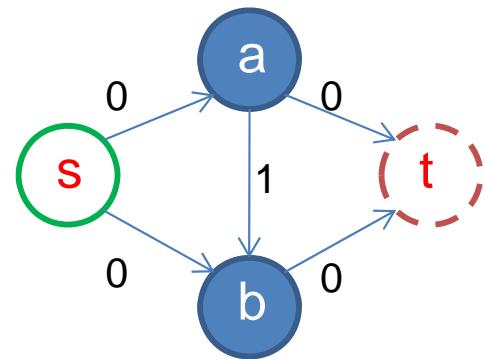


# BFS Implementation

- BFS implementation of Ford Fulkerson's method (called Edmonds Karp's algorithm) runs in  $O(VE^2)$



...  
After just 2  
iterations



# Edmonds Karp's (using STL) (1)

```
int res[MAX_V][MAX_V], mf, f, s, t; // global variables
vi p; // note that vi is our shortcut for vector<int>

// traverse the BFS spanning tree as in print_path (section 4.3)
void augment(int v, int minEdge) {
    // reach the source, record minEdge in a global variable 'f'
    if (v == s) { f = minEdge; return; }
    // recursive call
    else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
        // alter residual capacities
        res[p[v]][v] -= f; res[v][p[v]] += f;
    }
}

// in int main()
// set up the 2d AdjMatrix 'res', 's', and 't' with appropriate values
```

# Edmonds Karp's (using STL) (2)

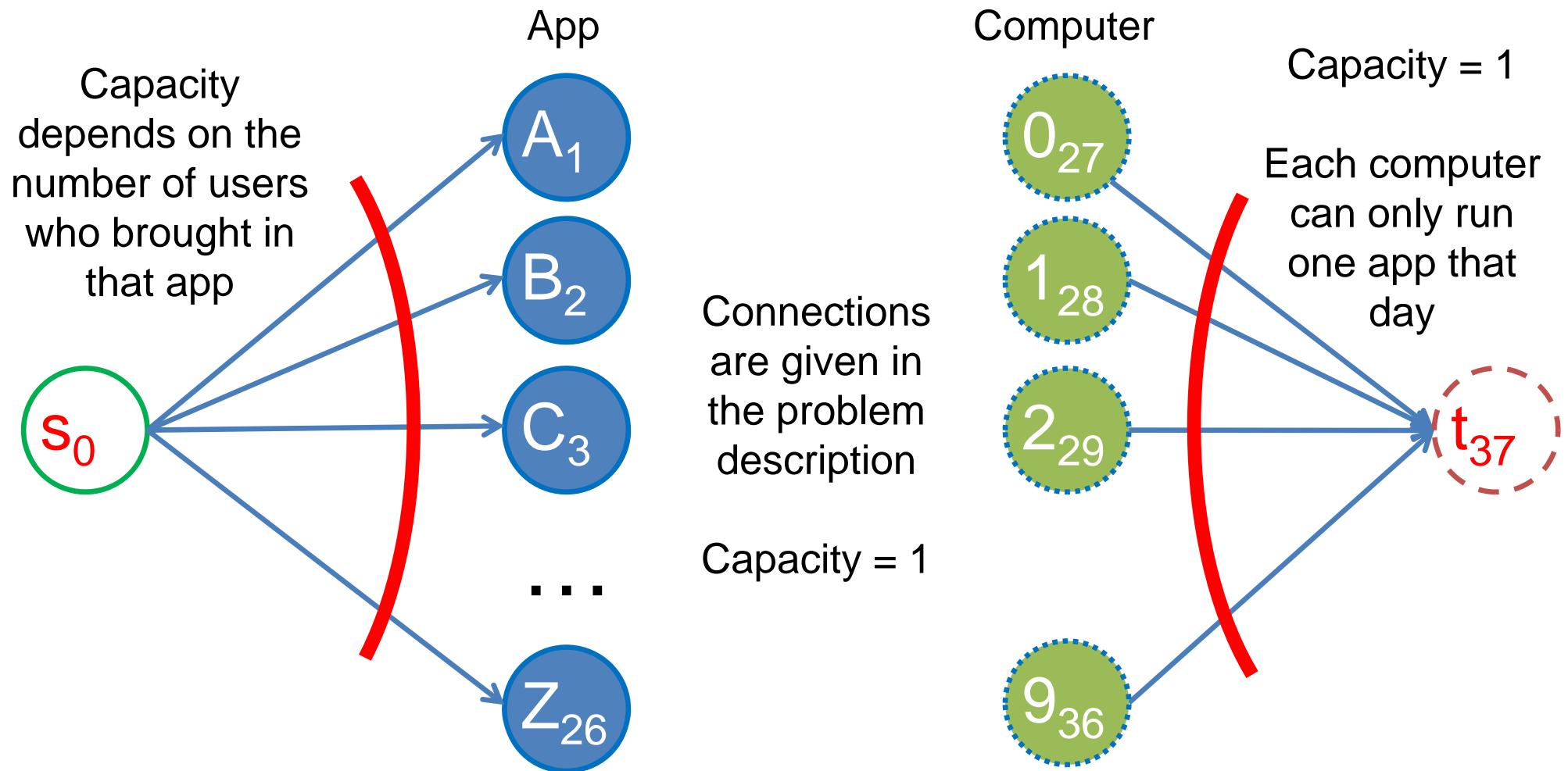
```
mf = 0;
while (1) { // run O(VE * V^2 = V^3*E) Edmonds Karp to solve the Max Flow problem
    f = 0;

    // run BFS, please examine parts of the BFS code that is different than in Section 4.3
    queue<int> q; vi dist(MAX_V, INF); // #define INF 2000000000
    q.push(s); dist[s] = 0;
    p.assign(MAX_V, -1); // (we have to record the BFS spanning tree)
    while (!q.empty()) { // (we need the shortest path from s to t!)
        int u = q.front(); q.pop();
        if (u == t) break; // immediately stop BFS if we already reach sink t
        for (int v = 0; v < MAX_V; v++) // note: enumerating neighbors with AdjMatrix is 'slow'
            if (res[u][v] > 0 && dist[v] == INF) dist[v] = dist[u] + 1, q.push(v), p[v] = u;
    }

    augment(t, INF); // find the min edge weight 'f' along this path, if any
    if (f == 0) break; // if we cannot send any more flow ('f' = 0), terminate the loop
    mf += f; // we can still send a flow, increase the max flow!
}

printf("%d\n", mf); // this is the max flow value of this flow graph
```

# UVa 259 – Software Allocation



# Other Applications of Max Flow

- Variants:
  - Min Cut
  - Multi-source Multi-sink Max Flow
  - Max Flow with Vertex Capacities
  - Max Independent Path
  - Max Edge-Disjoint Path
  - Min Cost Max Flow (MCMF)
- On “Special Graphs”:
  - Max Cardinality Bipartite Matching (MCBM)
  - Max Independent Set on Bipartite Graph (brief)
  - Max Vertex Cover on Bipartite Graph (brief)
  - Min Path Cover on DAG → MCBM on Bipartite Graph

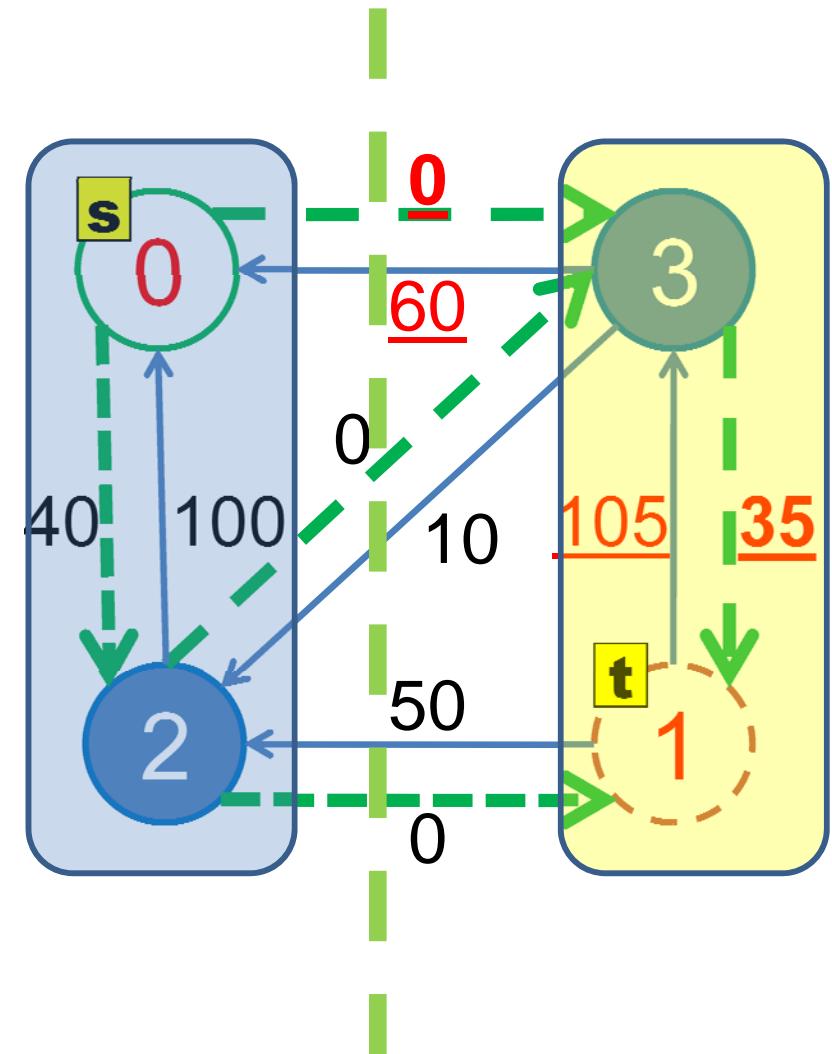
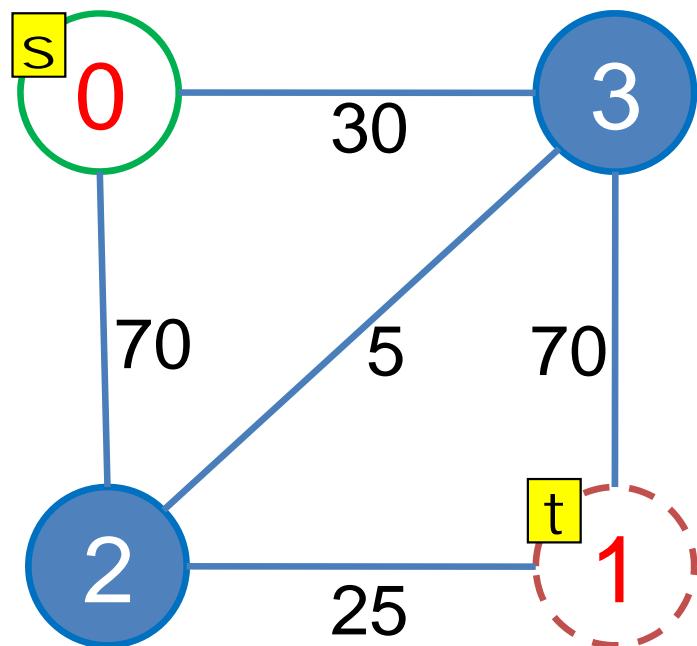
We will focus only on the **red ones** for this semester's CS3233 😞

For the rest, read the book!

# Max Flow = Min Cut (Dual)

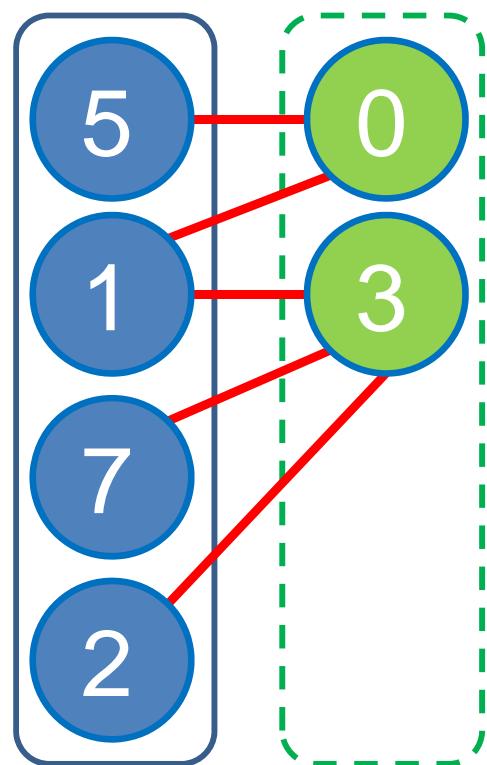
- Min Cut:
  - How to cut the edge(s) of the graph such that we minimizes sum of the edge weights being cut?
    - Note: This is ‘similar’ to finding ‘bridges’, but they are **different!**
- It happens that running max flow algorithm can give us the answer for min cut problem :O
  - Idea: Once we cannot add any more flow (max flow reached), do graph traversal from source  $s$  *one more time*
  - All vertices still reachable from  $s$  are  $\in$  first component, the rest are  $\in$  second component, and the cut is obvious
  - Detailed proof is not shown...

# Min Cut Example



Bipartite Graph

# SPECIAL GRAPHS 2



# Matching on Bipartite Graph

- In general graph
  - We need to use **Edmonds's Blossom Shrinking** (not discussed today) that is quite hard to code
  - DP + bitmask can only be used if input size is not too large ( $N \leq 20$ )
    - More than that... MLE 😞
- In bipartite graph...
  - Let's see the next few slides ☺

# PrimePairs (1)

- Group a list of numbers into pairs s.t the sum of each pair is prime.
- Given the numbers {1, 4, 7, 10, 11, 12}, you could group them as follows:
  1.  $1 + 4 = 5$        $7 + 10 = 17$        $11 + 12 = 23$
  2.  $1 + 10 = 11$        $4 + 7 = 11$        $11 + 12 = 23$
- Task: Given a `int[] numbers`, return a `int[]` of all the elements in `numbers` that could be paired with `numbers[0]` successfully as part of a *complete* pairing, sorted in ascending order.
- The answer for the example above would be {4, 10}.
  - Even though  $1 + 12$  is prime, there would be no way to pair the remaining 4 numbers.
- **Constraints:**
  - `numbers` contain an even number of elements in [2 .. 50], inclusive.
  - Each element of `numbers` will be between 1 and 1000, inclusive.
  - Each element of `numbers` will be distinct.

# PrimePairs (2)

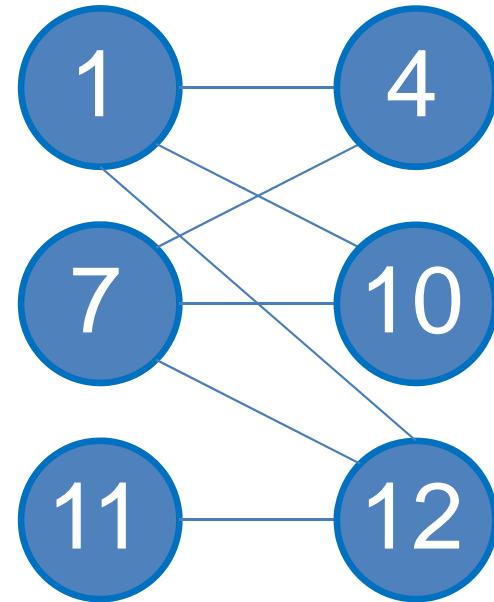
- Sample test cases (Source: TCO09 Qual 1):
  - Input: {1, 4, 7, 10, 11, 12}, Output: {4, 10}
    - This is the example from the problem statement.
  - Input: {11, 1, 4, 7, 10, 12}, Output: {12}
    - The same numbers, but in a different order.
    - In both of the 2 possible complete pairings, the 11 is paired with the 12.
  - Input: {8, 9, 1, 14}, Output: {}
    - No complete pairings are possible because none of the numbers can be paired with 1.
  - Input: {34, 39, 32, 4, 9, 35, 14, 17}, Output: {9, 39}
  - Input: {941, 902, 873, 841, 948, 851, 945, 854, 815, 898, 806, 826, 976, 878, 861, 919, 926, 901, 875, 864}, Output: {806, 926}

# PrimePairs (3)

- Is this a Math problem?
  - Yes, there is a bit Math here, we need list of prime...
  - But elements of numbers are  $\leq 1000$ !
    - This is not the major issue
- Complete Search Pairings?
  - ${}_{50}C_2$  for the first pair,  
 ${}_{48}C_2$  for the second pair, ...,  
until  ${}_2C_2$  for the last pair?
    - This is too much...
    - This is the major issue

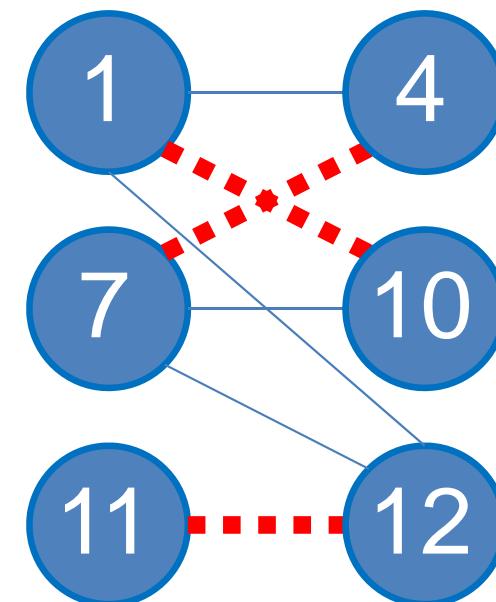
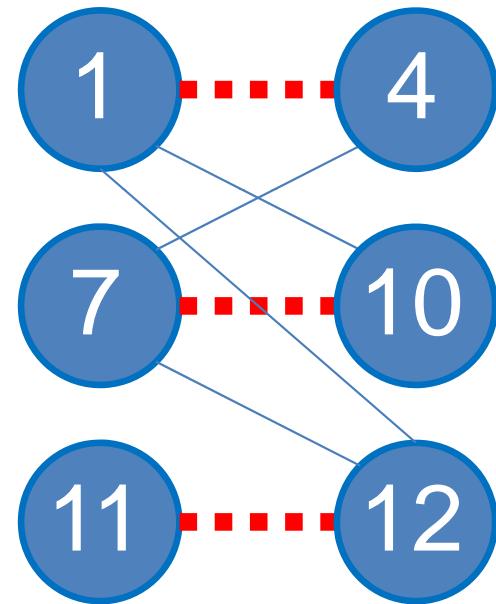
# PrimePairs (4)

- Any keyword to help?
  - Pairing = Matching, familiar?
- Is this matching bipartite^?
  - YES :D
  - To get a prime, we need to sum 1 odd + 1 even
    - 1 odd + 1 odd = even number → not a prime
    - 1 even + 1 even = even number → not a prime
  - Split odd/even numbers to left/right set
    - Give edge from left to right if  $\text{left}[i] + \text{right}[j] = \text{prime}$

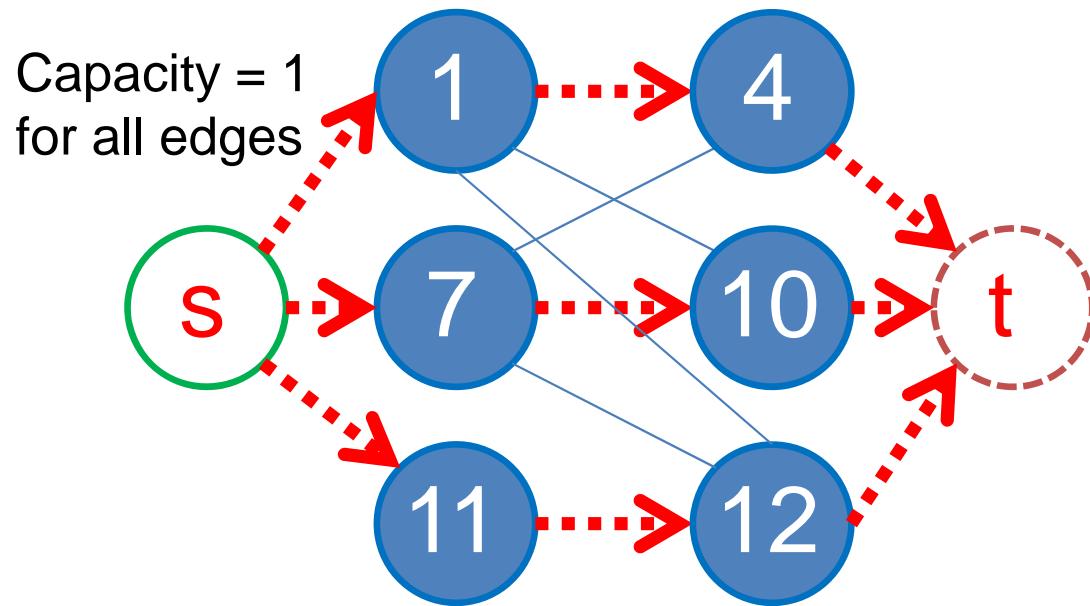


# PrimePairs (5)

- Solution is then trivial:
  - If size of even and odd set are different
    - Pairing is not possible...
  - Otherwise, if size of both sets are  $n/2$ 
    - Try to match  $\text{left}[0]$  with  $\text{right}[k]$  for  $k = [0 .. n/2 - 1]$ 
      - Do *Bipartite Matching* for the rest
      - If we obtain  $n/2 - 1$  more matchings
        - » Add  $\text{right}[k]$  to the answer
    - For this test case,  
we get 1 + 4 and 1 + 10 as the answer

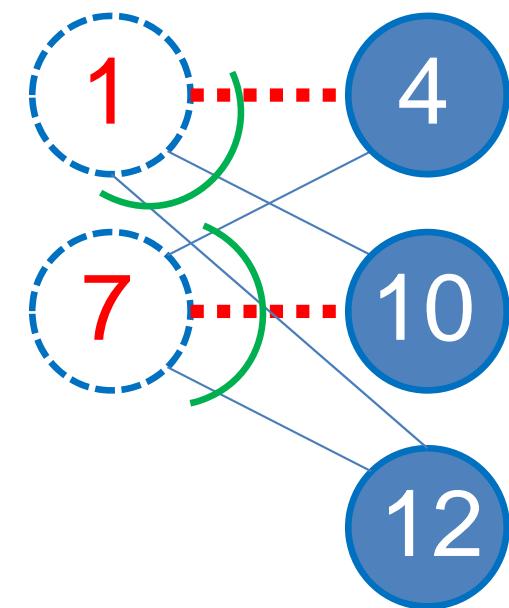
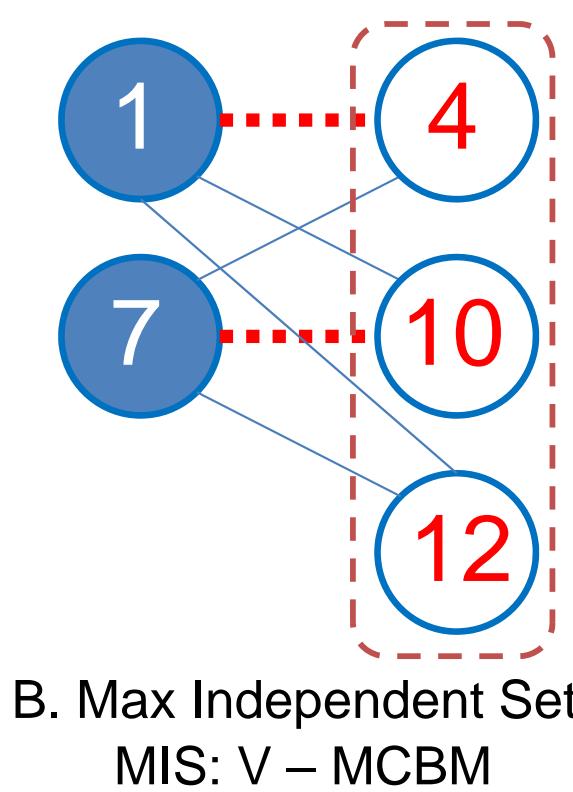
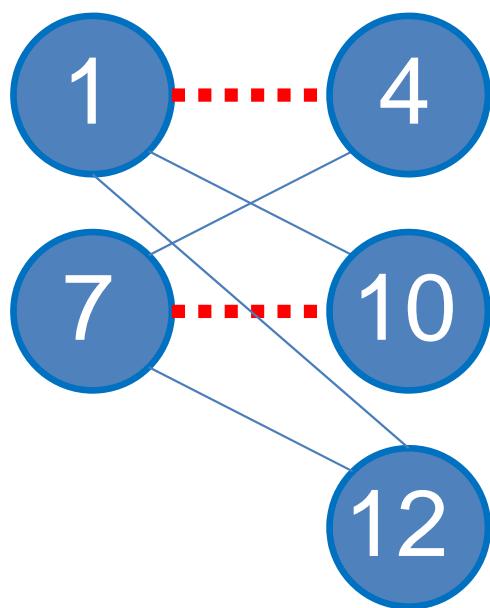


# PrimePairs (6)



# Can Be Tricky!

## Independent Set / Vertex Cover



(König's theorem)

# Graph Theory in ICPC

- Graph problems appear several times in ICPC!
  - Min 1, normally 2, can be 3 out of 10
  - Master all known solutions for classical graph problems
  - Or perhaps combined with DP/Greedy style
- This can move your team nearer to top 10
  - Perhaps rank [11-20] out of 60 now
  - Solving 3-5 problems out of 10

# References

- CP2, Chapter 4 ☺
- Introduction to Algorithms, Ch 22,23,24,25,26 (p643-698)
- Algorithm Design, Ch 3,4,6,7 (p337-450)
- Algorithms (Dasgupta et al), Ch 6 & Ch 7
- Algorithms (Sedgewick), Ch 33 & Ch 34
- Algorithms (Alsuwaiyel), Ch 16 & Ch 17
- Programming Challenges, p227-230, Ch 10
- <http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary2>
- Internet: [TopCoder Max-Flow tutorial](#), UVa Live Archive, UVa main judge, Felix's blog, Suhendry's blog, Dhaka 2005 solutions, other Max Flow lecture notes, etc...

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 09 – Mathematics  
in Programming Contests

# Outline

- Mini Contest #7 + Discussion + Break
- Admins
- Mathematics-Related Problems & Algorithms
  - Ad Hoc Mathematics Problems
    - Those that do not need specific algorithm, just basic coding/math skill
  - Java BigInteger/Decimal
  - Combinatorics
  - Number Theory
  - Others (not covered in depth this semester):
    - Probability Theory
    - Cycle-Finding
    - Game Theory
    - Powers of a (Square) Matrix

# Mathematics, CS, and ICPC/IOI (1)

- Computer Science is deeply rooted in Maths
  - Compute = Math
- It is not a surprise to see many Maths problems in ICPC (PS: IOI tasks are usually *not* Maths-specific)
  - Many of which, I do not have time to teach you...
  - Few others, I cannot teach you as I do not know them yet...
  - CS3233 is NOT a pure Mathematics module
    - Only 1 week (1.5 hours) is devoted for Mathematics-related topic
  - It is nice if we can improve our ranks by solving some mathematics problems in programming contest

# Mathematics, CS, and ICPC/IOI (2)

- Tips:
  - Revise your high school mathematics
  - (In NUS): Take MAXXXX modules as CFM :D
  - Read more references about powerful math algorithms and/or interesting number theories, etc
  - Study C++ <cmath> & Java.Util.Math/Java.Math Library
  - Try maths problems in UVa/other OJ and at [projecteuler](#)

# The Lesson Plan

- Today, we will discuss small subset of this big domain
- Plan:
  - We will skip/fast forward “not so interesting” stuffs
  - I will give several Maths-related pop-quizzes using clicker system to see how far you know these tricks...
  - We will focus on **three related subjects**: Big Integer, Prime Factors, and Modulo Arithmetic
  - You will then have to read Chapter 5 of CP2 **on your own** (it is a huge chapter btw...)



# Mathematics-Related Problems

## 1. Ad Hoc Mathematics

1. The Simple Ones
2. Mathematical Simulation (Brute Force)
3. Finding Pattern or Formula
4. Grid
5. Number Systems or Sequences
6. Logarithm, Exponentiation, Power
7. Polynomial
8. Base Number Variant
9. Just Ad Hoc: Roman Numerals, etc

## 2. Java BigInteger

1. Basic Features
2. Bonus Features

## 3. Combinatorics

1. Fibonacci Numbers
2. Binomial Coefficients
3. Catalan Numbers
4. Others

## 4. Number Theory

1. Prime Numbers: Sieve of Eratosthenes
2. GCD & LCM
3. Factorial
4. Prime Factors
5. Working with Prime Factors
6. Functions involving Prime Factors
7. Modulo Arithmetic
8. Extended Euclid/Linear Diophantine Equation
9. Others

## 5. Probability Theory

## 6. Cycle-Finding

1. Floyd's Tortoise-Hare Algorithm

## 6. Game Theory

1. Two Players Game, Minimax
2. Nim Game (Sprague Grundy Theorem)

## 7. Powers of a (Square) Matrix

1. Efficient Exponentiation

Programming problems that are from the domain of mathematics,  
but we do not need specialized data structure(s) or algorithm(s) to solve them

We will do **A QUICK SPLASH AND DASH...** Learn the details at home ☺

Section 5.2

## AD HOC MATHEMATICS

# The Simpler Ones

- Nothing to teach 😞
- They are too simple, really...
- You can get ~9 ACs in < 1 hour if you solve all problems listed in this category in CP2 😊

# Mathematical Simulation (Brute Force)

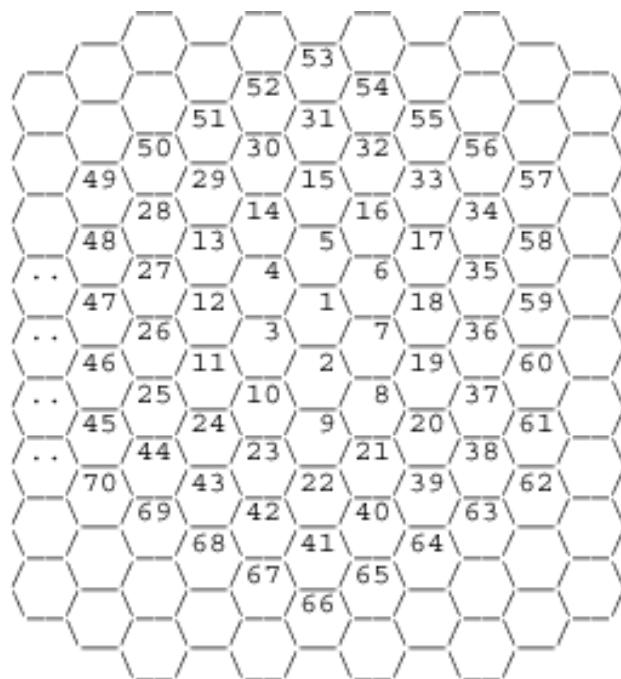
- Nothing to teach other than the ones already presented during iterative/recursive “Complete Search” topic
  - Just remember to prune the search space whenever possible...
- Note: Problems that require other technique (like number theory knowledge) and cannot be brute-force-able are NOT classified in this category

# Finding Pattern or Formula

- Nothing to teach 😞
- This requires your mathematical insights to obtain those patterns/formulas **as soon as possible** to reduce the time penalty (in ICPC setting)
- Useful trick:
  - Solve some small instances by hand
  - List the solutions and see if there is/are any pattern(s)?

# Grid

- Nothing to teach 😞
- Also about finding pattern. It requires creativity on manipulating the grid/convert it to simpler ones
- Example:



# Number Systems or Sequences

- Nothing to teach 😞
- Most of the time, carefully following the problem description is sufficient

# Logarithm, Exponentiation, Power

- In C/C++ <cmath>, we have  $\log$  (base  $e$ ) and  $\log_{10}$  (base 10)
- In Java.lang.Math, we only have  $\log$  (base  $e$ )
- To do  $\log_b(a)$  (base  $b$ ), we can use:  
 $\log(a) / \log(b)$
- Btw, what does this code snippet do?  
`(int)floor(1 + log10((double)a))`
- And how to compute the n-th root of a?



# Polynomial

- Representation: usually the coefficients of the terms in some sorted order (based on power)
- Polynomial formatting, evaluation, derivation (Horner's rule), division, remainder...
- The solution usually requires careful loops...

# Base Number Variants

- Do you know that base number conversion is now *super easy* with Java BigInteger?
- However, for some variants, we still have to go to the basics method...
  - The solution usually use base 10 (decimal) as an intermediate step

# And A Few Others...

- e.g. Roman Numerals, etc
- Best way to learn these: Via practice...

Section 5.3

A Powerful API for Programming Contests

# JAVA BIGINTEGER CLASS

# Big Integer (1)

- Range of default integer data types (C++)
  - unsigned int = unsigned long:  $2^{32}$  (9-10 digits)
  - unsigned long long:  $2^{64}$  (19-20 digits)
- Question:
  - What is “777!”, i.e. factorial of 777?
- Solution?
  - Big Integer: Use string to represent number
    - ~ number can be as long as computer memory permits
    - FYI, this is similar to how basic data types are stored in computer memory. Just that this time we do not have limitation of the number of bits (digits) used...

# Big Integer (2)

- Operations on Big Integer
  - Basic: add, subtract, multiply, divide, etc
  - Use “high school method”
    - Some examples below:

$$\begin{array}{r} 1 \leftarrow \text{carry} \\ 218 \\ 45 \\ \hline - + \\ 263 \end{array} \qquad \begin{array}{r} 218 \\ 45 \\ \hline \text{--- } x \\ 1090 \quad (218 * 5) \\ 872 \quad (218 * 4) * 10 \\ \hline \text{----- } + \\ 9810 \end{array}$$

# Big Integer (3)

- Note:
  - Writing these “high school methods” during stressful contest environment is **not a good strategy!**
- Fortunately, Java has BigInteger library
  - They are allowed to be used in contests (ICPC and CS3233)
    - So use it...
  - Note: IOI does not allow Java yet,  
and anyway, I have not seen BigInteger-related tasks in IOI...
- Or, if you insist, build your own BigInt library and bring its hardcopy to future contests!

# Java BigInteger Class

- This class is rather powerful
  - Not just it allows for basic mathematical operations involving big integers (addition, subtraction, multiplication, division, mod or remainder, and power)...
  - It also provides support for:
    - Finding GCD of big numbers
    - Finding the solution of  $x^y \bmod m$  (modulo arithmetic)
    - Very Easy Base Number Conversion ← so far the more useful one
    - See various examples in the book ☺

Section 5.4

# **COMBINATORICS**

# Combinatorics

- Given problem description,  
find some nice formula to **count something**
  - Coding is (usually very) short
  - Finding the formula is not straightforward...
    - If formula has overlapping sub problems → use DP
    - If formula yield huge numbers → use Java BigInteger
- Memorize/study the basic ones: Fibonacci-based formulas, Binomial Coefficients, Catalan Numbers...
- PS: On-Line Encyclopedia of Integer Sequences
  - Can be a good reference: <http://oeis.org/>

Programming problems that requires the knowledge of number theory, otherwise you will likely get Time Limit Exceeded (TLE) response for solving them naively...

Section 5.5

# NUMBER THEORY

# Prime Numbers

- First prime and the only even prime: 2
- First 10 primes: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}
- Primes in range:
  - 1 to 100 : 25 primes
  - 1 to 7,919 : 1,000 primes
  - 1 to 1,000 : 168 primes
  - 1 to 10,000 : 1,229 primes
- Largest prime in signed 32-bit int = 2,147,483,647
- Used/appear in:
  - Factoring
  - Cryptography
  - Many other problems in ICPC, etc

# Optimized Prime Testing

- Algorithms for testing if  $N$  is prime:  $\text{isPrime}(N)$ 
  - First try: check if  $N$  is divisible by  $i \in [2 .. N-1]$ ?
    - $O(N)$
  - Improved 1: Is  $N$  divisible by  $i \in [2 .. \sqrt{N}]$ ?
    - $O(\sqrt{N})$
  - Improved 2: Is  $N$  divisible by  $i \in [3, 5, .. \sqrt{N}]$ ?
    - One test for  $i = 2$ , no need to test other even numbers!
    - $O(\sqrt{N}/2) = O(\sqrt{N})$
  - Improved 3: Is  $N$  divisible by  $i \in \text{primes} \leq \sqrt{N}$ 
    - $O(\pi(\sqrt{N})) = O(\sqrt{N}/\log(\sqrt{N}))$ 
      - $\pi(M) = \text{num of primes up to } M$
      - For this, we need smaller primes beforehand

# Prime Generation

- What if we want to generate a list of prime numbers between [0 ... N]?

- Slow naïve algorithm:

```
Loop i from [0 ... N]
    if (isPrime(i))
        print i
```

- Can we do better?

- Yes: Sieve of Eratosthenes

# Sieve of Eratosthenes Algorithm

- Generate primes between [0 ... N]:
  - Use **bitset** of size N, set all true except index 0 & 1
  - Start from  $i = 2$  until  $k \cdot i > N$ 
    - If bitset at index  $i$  is on, cross all multiple of  $i$  (i.e. turn off bit at index  $i$ ) starting from  $i \cdot i$
  - Finally, whatever not crossed are primes
- Example:
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 51, 52, 53, 54, 55, ..., 75, 76, 77, ...
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 51, 52, 53, 54, 55, ..., 75, 76, 77, ...
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 51, 52, 53, 54, 55, ..., 75, 76, 77, ...
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 51, 52, 53, 54, 55, ..., 75, 76, 77, ...
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ..., 51, 52, 53, 54, 55, ..., 75, 76, 77, ...

# Greatest Common Divisor (GCD)

- Naïve Algorithm:
  - Find all divisors of a and b (slow)
  - Find those that are common
  - Pick the greatest one
- Better & Famous algorithm: D & C Euclid algorithm
  - $\text{GCD}(a, 0) = a$
  - $\text{GCD}(a, b) = \text{GCD}(b, a \% b)$  // problem size decreases a lot!!
- Its recursive code is easy to write:

```
int gcd(int a, int b) { return (b == 0 ? a : gcd(b, a % b)); }
```

# Lowest Common Multiple (LCM)

- $\text{lcm}(a, b) = a * b / \text{gcd}(a, b)$

```
int lcm(int a, int b) { return (a * b / gcd(a, b)); }
```

- Note for gcd/lcm of more than 2 numbers:
  - $\text{gcd}(a, b, c) = \text{gcd}(a, \text{gcd}(b, c))$ ;
- Both gcd and lcm runs in  $O(\log_{10} n)$  where  $n = \max(a, b)$

# Factorial

- What is the highest  $n$  so that **factorial( $n$ )** still fits in 64-bits unsigned long long?
  - Answer:  $n = 20$ 
    - $20! = 2432902008176640000$
    - $ull = 18446744073709551615$
    - $21! = 51090942171709440000$
- Hm... so almost all factorial related questions require Java BigInteger?



# Prime Factors

- Direct algorithm: generate list of primes (use sieve), check how many of them can divide integer N
  - This can be improved!
- Better algorithm: Divide and Conquer!
  - An integer N can be expressed as:
    - $N = PF * N'$ 
      - PF = a **prime factor**
      - $N'$  = another number which is  $N / PF$
    - If  $N' = 1$ , stop; otherwise, repeat
    - N is reduced every time we find a divisor

But if integer I is a large prime, then this is still slow.

This fact is the basis for cryptography techniques

Ok... that's enough for this semester 😊

# **THE OTHER MATHS PROBLEMS IN PROGRAMMING CONTEST**

# Not Covered in Depth this Semester

- Probability Theory (Section 5.6)
- Cycle-Finding (Section 5.7)
- Game Theory (Section 5.8)
- Powers of a (Square) Matrix (Section 5.9)
- They are already in CP2
  - Read them on your own
  - These problems will not appear as problem A or B in our mini contests

# Many More Still Undiscussed...

- Mathematics is a large field
  - Gaussian Elimination for solving system of linear equations
  - Miller-Rabin's primality test
  - Pollard's Rho integer factoring algorithm
  - Many other Prime theorems, hypotheses, conjectures
  - Chinese Remainder Theorem
  - Lots of Divisibility Properties
  - Combinatorial Games, etc
- Again CS3233 != Math Module
- Chapter 5 of CP2 has a collection of 285 UVa programming exercises... the highest among 8 chapters in CP2!

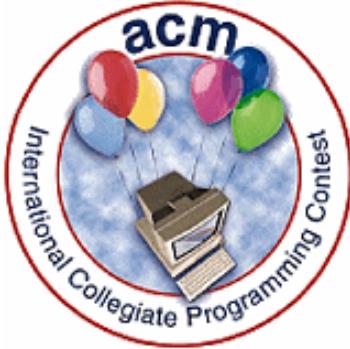
# Summary

- We have seen *some* mathematics-related problems and algorithms
  - Too many to be learned in one night...
  - Even so, many others are left uncovered  
(some are inside CP2 for you to read on your own pace)
  - Best way to learn: lots of practice
- In the next two weeks, two more new topics:
  - Week 10: String Processing (Focus on SA)
  - Week 11: (Computational) Geometry

# References

- CP2, Chapter 5 ☺
- Introduction to Algorithms, Ch 31, Appendix A/B/C
- Project Euler, <http://projecteuler.net/>

This course material is now made available for public usage.  
Special acknowledgement to School of Computing, National University of Singapore  
for allowing Steven to prepare and distribute these teaching materials.



# CS3233

# Competitive Programming

Dr. Steven Halim

Week 10 – String Processing

# Outline

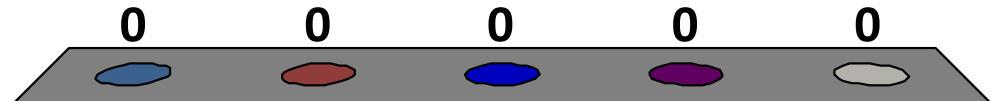
- Mini Contest #8 + Discussion + Break
- Admins
- Covered *very briefly* in class but examinable:
  - Basic String Processing Skills
- Skipped this semester:
  - Ad Hoc String Problems
  - String Matching (Knuth-Morris-Pratt's Algorithm)
- Today, focus on:
  - Suffix Trie/Tree/Array
- Note: DP on String has been discussed in Week 06

Section 6.2

# **BASIC STRING PROCESSING SKILLS**

# Which programming language is the best for processing strings?

1. C
2. C++
3. Java
4. Perl (eh what is this?)
5. It depends...



# Basics of String Processing (01)

## Data Structure

### C (top)/C++ (bottom)

- C: **null-terminated character array**

- We have to know the string length (or at least the upperbound) beforehand

```
char str[10000];
```

### Java

- **String class**

```
String str;
```

- C++: **string class**

```
#include <string>
string str;
```

# Basics of String Processing (02)

## Reading a String (a word)

### C (top)/C++ (bottom)

```
#include <stdio.h>
```

```
scanf( "%s" , &str );
// & optional
```

```
#include <iostream>
using namespace std;

cin >> str;
```

### Java

```
import java.util.*;
```

```
Scanner sc = new
Scanner(System.in);
str = sc.next();
```

# Basics of String Processing (03)

## Reading a Line of String

### C (top)/C++ (bottom)

```
gets(str);
// alternative/safer version
// fgets(str, 10000, stdin);
// but you will read extra
// '\0' at the back
```

```
// set ifstream first
getline(ifstream, str);
```

### Java

```
str = sc.nextLine();
```

# Basics of String Processing (04)

## Printing and Formatting String Output

### C (top)/C++ (bottom)

- Preferred method ☺

```
printf("s = %s, l = %d\n",
      str, strlen(str));
```

- C++ version is harder ☹

```
cout << "s = " << str <<
      ", l = " << str.length()
      << endl;
```

### Java

- We can use

`System.out.print` or  
`System.out.println`, but  
the best is to use C-style  
`System.out.printf`

```
System.out.printf(
      "s = %s, l = %d\n",
      str, str.length());
```

# Basics of String Processing (05)

## Comparing Two Strings

### C (top)/C++ (bottom)

```
printf(strcmp(str, "test") ?  
      "different\n" :  
      "same\n" );
```

### Java

```
System.out.println(  
    str.equals("test"));
```

```
cout << str == "test" ?  
    "same" :  
    "different" << endl;
```

# Basics of String Processing (06)

## Combining Two Strings

### C (top)/C++ (bottom)

```
strcpy(str, "hello");
strcat(str, " world");
printf("%s\n", str);
// output: "hello world"
```

### Java

```
str = "hello";
str += " world";
System.out.println(str);
```

```
str = "hello";
str.append(" world");
cout << str << endl;
```

# Basics of String Processing (07)

## String Tokenizer: Splitting Str into Tokens

### C (top)/C++ (bottom)

```
#include <cstring>
for (
    char *p=strtok(str, " ");
    p = strtok(NULL, " ");
    printf("%s\n", p);
```

### Java

```
import java.util.*;
StringTokenizer st = new
    StringTokenizer(str, " ");
while (st.hasMoreTokens())
    System.out.println(
        st.nextToken());
```

- C++: we can use **istringstream**

# Basics of String Processing (08)

## String Matching: Finding a Substr in a Str

### C (top)/C++ (bottom)

```
char *p=strstr(str, substr);  
if (p)  
    printf("%d\n", p-str-1);
```

### Java

```
int pos=str.indexOf(substr);  
if (pos != -1)  
    System.out.println(pos);
```

```
int pos = str.find(substr);  
if (pos != string::npos)  
    cout << pos - 1 << endl;
```

# Basics of String Processing (09)

## Editing/Examining Characters of a String

### Both C & C++

```
#include <ctype.h>

for (int i = 0; str[i]; i++)
    str[i] = toupper(str[i]);
// or tolower(ch)
// isalpha(ch), isdigit(ch)
```

### Java

- Characters of a Java `String` can be accessed with `str.charAt(i)`, but Java `String` is immutable (cannot be changed)
- You may have to create new `String` or use Java `StringBuffer`

# Basics of String Processing (10)

## Sorting Characters of a String

### Both C & C++

```
#include <algorithm>

// if using C-style string
sort(s, s + (int)strlen(s));

// if using C++ string class
sort(s.begin(), s.end());
```

### Java

- Java String is immutable (cannot be changed)
- You have to break the string toCharArray() and then sort the character array

# Basics of String Processing (11)

## Sorting Array/Vector of Strings

### Preferably C++

```
#include <algorithm>
#include <string>
#include <vector>

vector<string> S;
// assume that S has items
sort(S.begin(), S.end());
// S will be sorted now
```

### Java

```
Vector<String> S =
    new Vector<String>();
// assume that S has items
Collections.sort(S);
// S will be sorted now
```

List of (simple) problems solvable with basic string processing skills

Section 6.3

Skipped this semester (do a few programming exercises on your own)

## **AD HOC STRING PROBLEMS**

# Ad Hoc String Problems (1)

- Cipher (Encode-Encrypt/Decode-Decrypt)
  - Transform string given a coding/decoding mechanism
  - Usually, we need to follow problem description
  - Sometimes, we have to guess the pattern
  - UVa 10878 – Decode the Tape
- Frequency Counting
  - Check how many times certain characters (or words) appear in the string
  - Use efficient data structure (or hashing technique)
  - UVa 902 – Password Search

# Ad Hoc String Problems (2)

- Input Parsing
  - Given a grammar (in Backus Naur Form or in other form), check if a given string is valid according to the grammar, and evaluate it if possible
  - Use **recursive** parser, Java **Pattern (RegEx)** class
  - UVa 622 – Grammar Evaluation
- Output Formatting
  - The problematic part of the problem is in formatting the output using certain rule
  - UVa 10894 – Save Hridoy

# Ad Hoc String Problems (3)

- String Comparison
  - Given two strings, are they similar with some criteria?
  - Case sensitive? Compare substring only? Modified criteria?
  - UVa 11233 – Deli Deli
- Others, not one of the above
  - But still solvable with just basic string processing skills
- Note:
  - None of these are likely appear in IOI other than as the bonus problem per contest day (no longer true in 2011)
  - In ICPC, one of these can be the bonus problem

Knuth-Morris-Pratt's Algorithm

Section 6.4

Skipped this semester (please use Suffix Array for (long) String Matching)

# **STRING MATCHING**

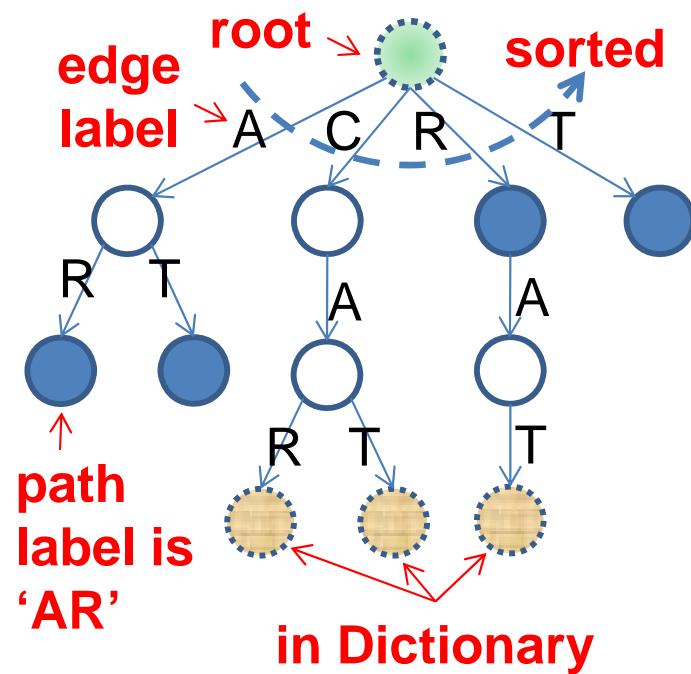
# String Matching

- Given a pattern string  $P$ ,  
can it be found in the longer string  $T$ ?
  - Do not code naïve solution
  - Easiest solution: Use string library
    - C++: `string.find`
    - C: `strstr`
    - Java: `String.indexOf`
  - In CP2 book: KMP algorithm
  - Or later: Suffix Array

This part is courtesy of A/P Sung Wing Kin, Ken from SoC, NUS  
Section 6.6

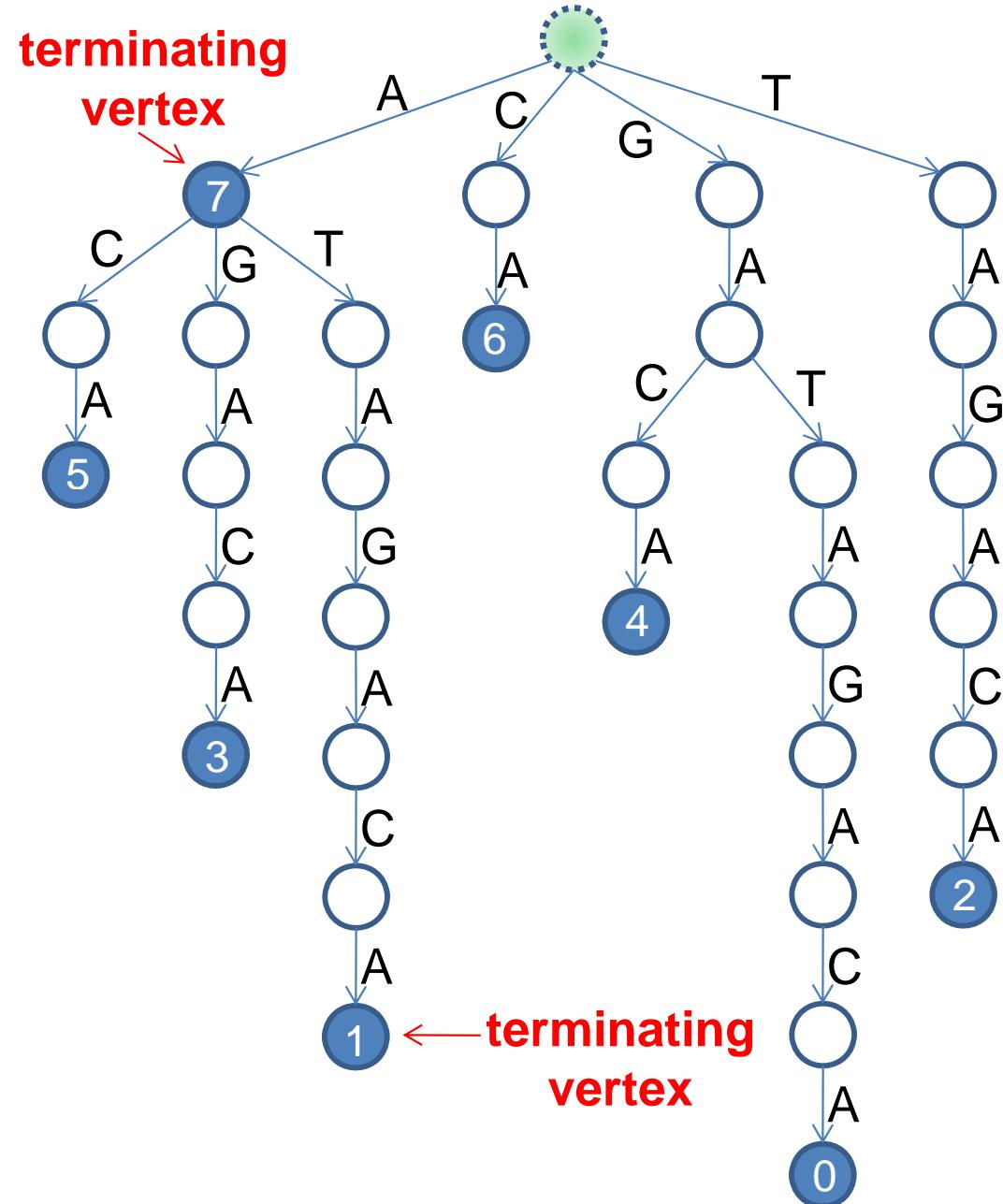
## **SUFFIX TRIE, TREE, AND ARRAY**

# Suffix Trie ('CAR', 'CAT', 'RAT')



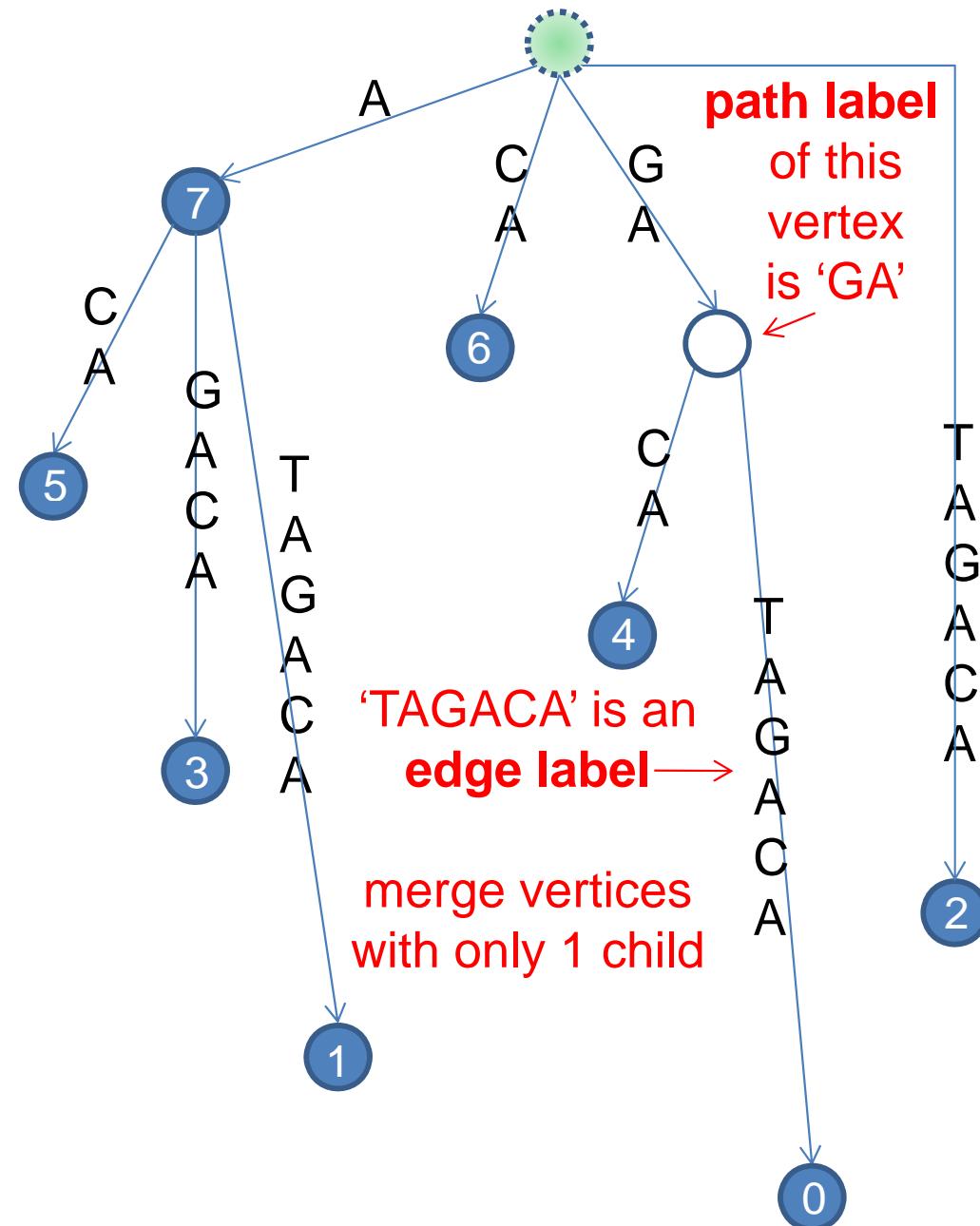
# Suffix Trie ( $T = \text{'GATAGACA'}$ )

| i | Suffix   |
|---|----------|
| 0 | GATAGACA |
| 1 | ATAGACA  |
| 2 | TAGACA   |
| 3 | AGACA    |
| 4 | GACA     |
| 5 | ACA      |
| 6 | CA       |
| 7 | A        |



# Suffix Tree ( $T = \text{'GATAGACA'}$ )

| i | Suffix   |
|---|----------|
| 0 | GATAGACA |
| 1 | ATAGACA  |
| 2 | TAGACA   |
| 3 | AGACA    |
| 4 | GACA     |
| 5 | ACA      |
| 6 | CA       |
| 7 | A        |

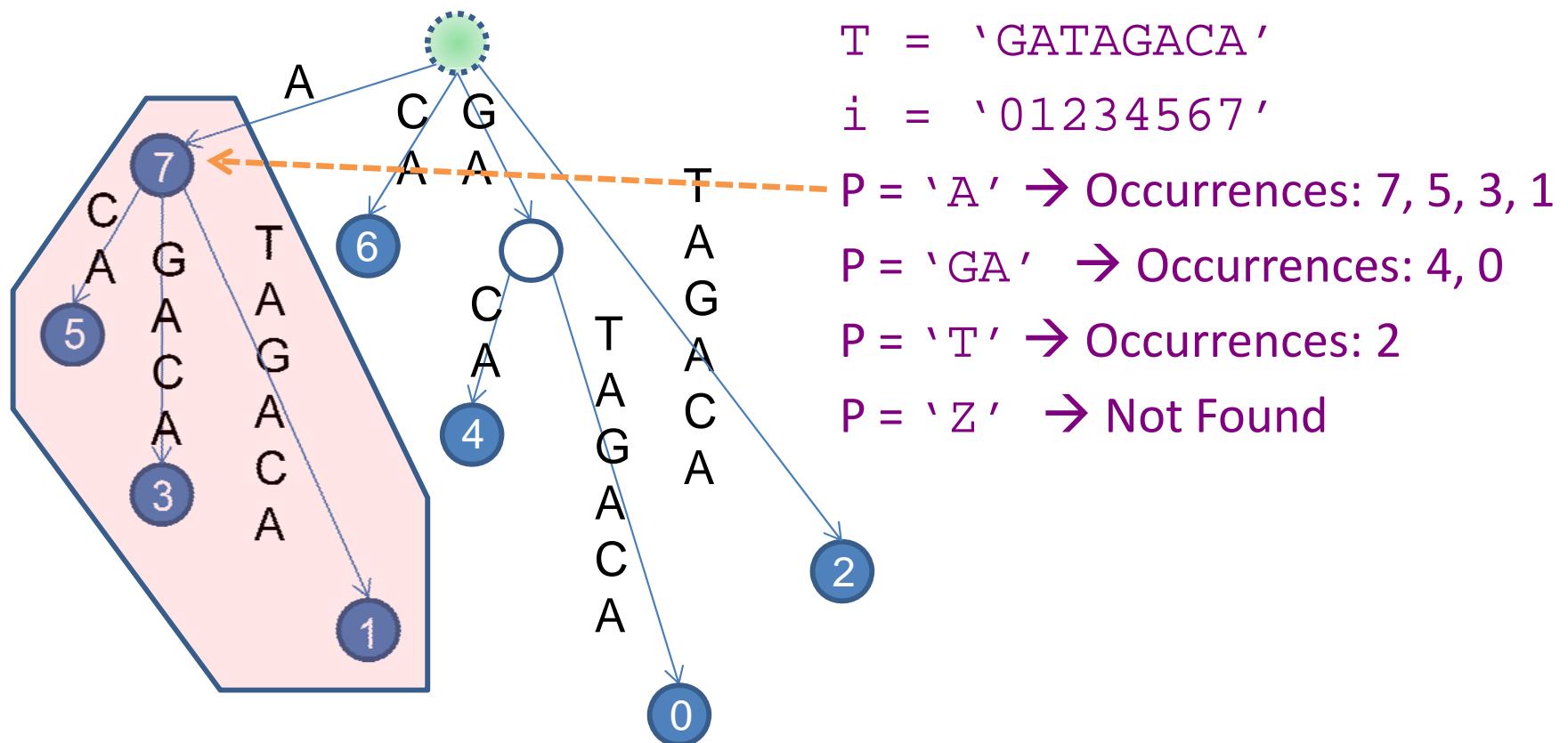


What can we do with this specialized string data structure?

## **APPLICATIONS OF SUFFIX TREE**

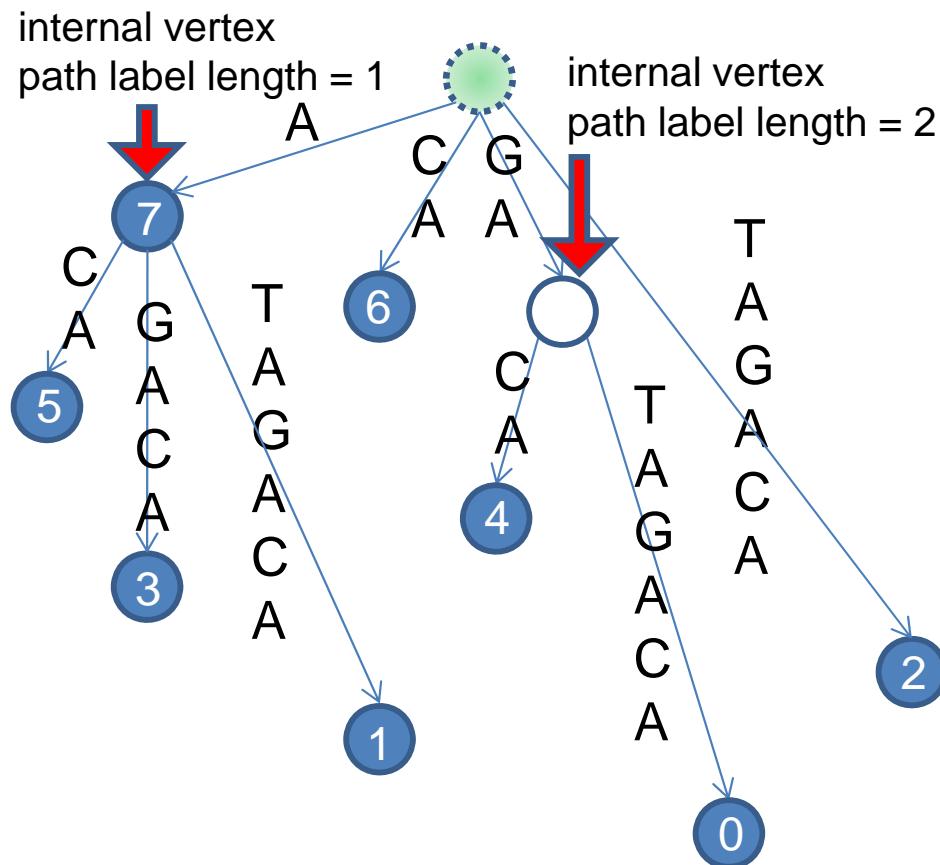
# String Matching

- To find all occurrences of **P** (of length m) in **T** (of length n)
  - Search for the vertex **x** in the Suffix Tree which represents **P**
  - All the leaves in the subtree rooted at **x** are the occurrences
- Time:  $O(m + \text{occ})$  where occ is the total no. of occurrences



# Longest Repeated Substring

- To find the longest repeated substring in T
  - Find the deepest internal node
- Time:  $O(n)$



e.g.  $T = \underline{\text{GATAGACA}}$

The longest repeated substring is 'GA' with path label length = 2

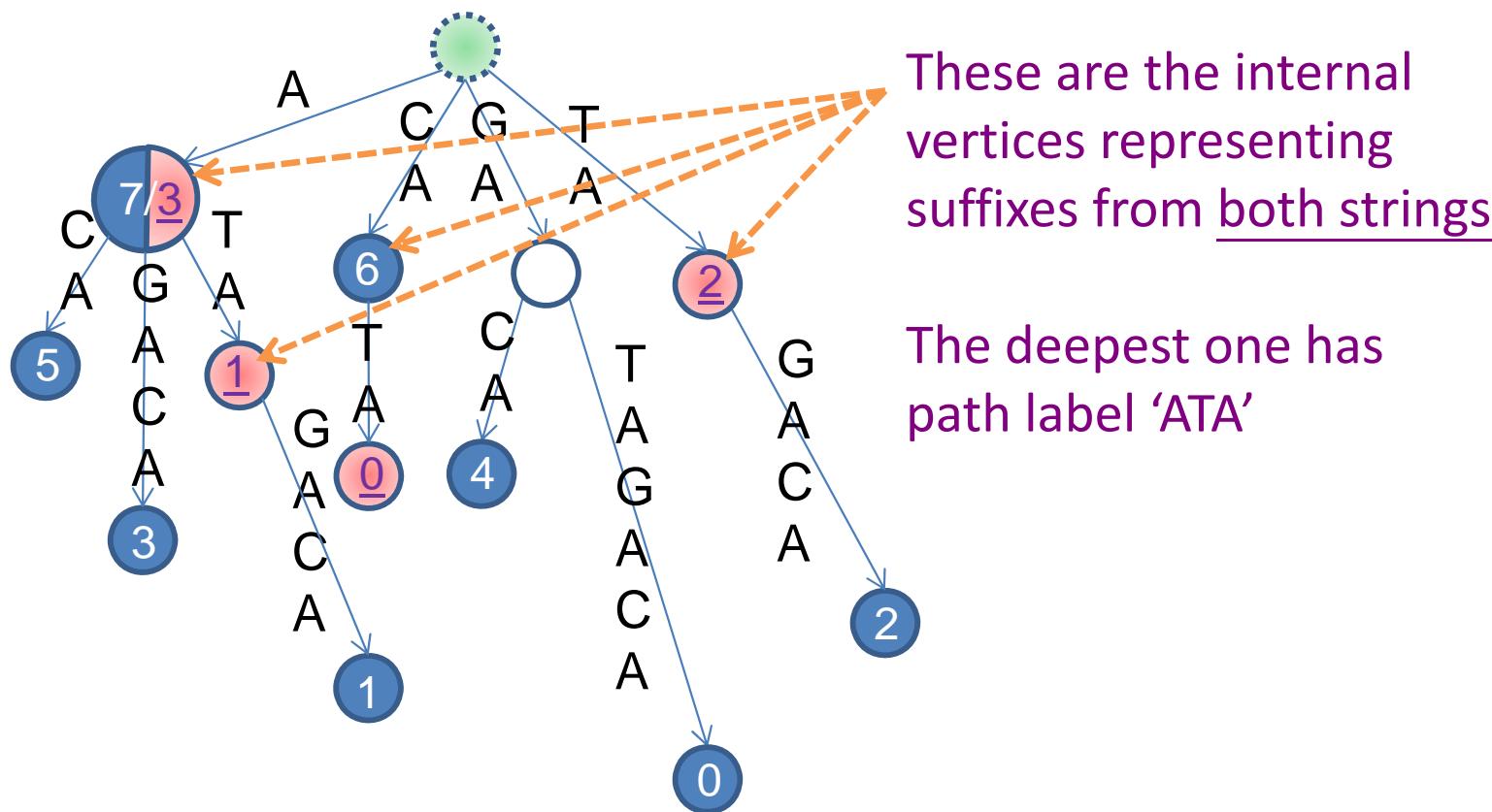
The other repeated substring is 'A', but its path label length = 1

# Longest Common Substring

- To find the longest common substring of two or more strings
  - Note: In 1970, Donald Knuth conjectured that a linear time algorithm for this problem is impossible
  - Now, we know that it can be solved in linear time
  - E.g. consider two string T1 and T2,
    - Build a **generalized** Suffix Tree for T1 and T2
      - i.e. a Suffix Tree that combines both the Suffix Tree of T1 and T2
    - Mark internal vertices with leaves representing suffixes of both T1 and T2
    - Report the deepest marked vertex

# Example of LC Substring

- $T_1 = \text{'GATAGACA'}$  (end vertices labeled with blue)  
 $T_2 = \text{'CATA'}$  (end vertices labeled with red)
  - Their longest common substring is ‘ATA’ with length 3



How to build Suffix Tree?

For programming contests, we use Suffix Array instead...

# **SUFFIX ARRAY**

# Disadvantage of Suffix Tree

- Suffix Tree is space inefficient
  - It requires  $O(n|\Sigma| \log n)$  bits
- Actual reason for programming contests
  - It is harder to construct Suffix Tree
- Manber and Myers (SIAM J. Comp 1993) proposes a new data structure, called the **Suffix Array**, which has a similar functionality as suffix tree
  - Moreover, it only requires  $O(n \log n)$  bits
- And it is much easier to implement

# Suffix Array (1)

- Suffix Array (SA) is an array that stores:
  - A permutation of  $n$  indices of sorted suffixes
  - Each integer takes  $O(\log n)$  bits, so SA takes  $O(n \log n)$  bits
- e.g. consider  $T = \text{'GATAGACA'}$

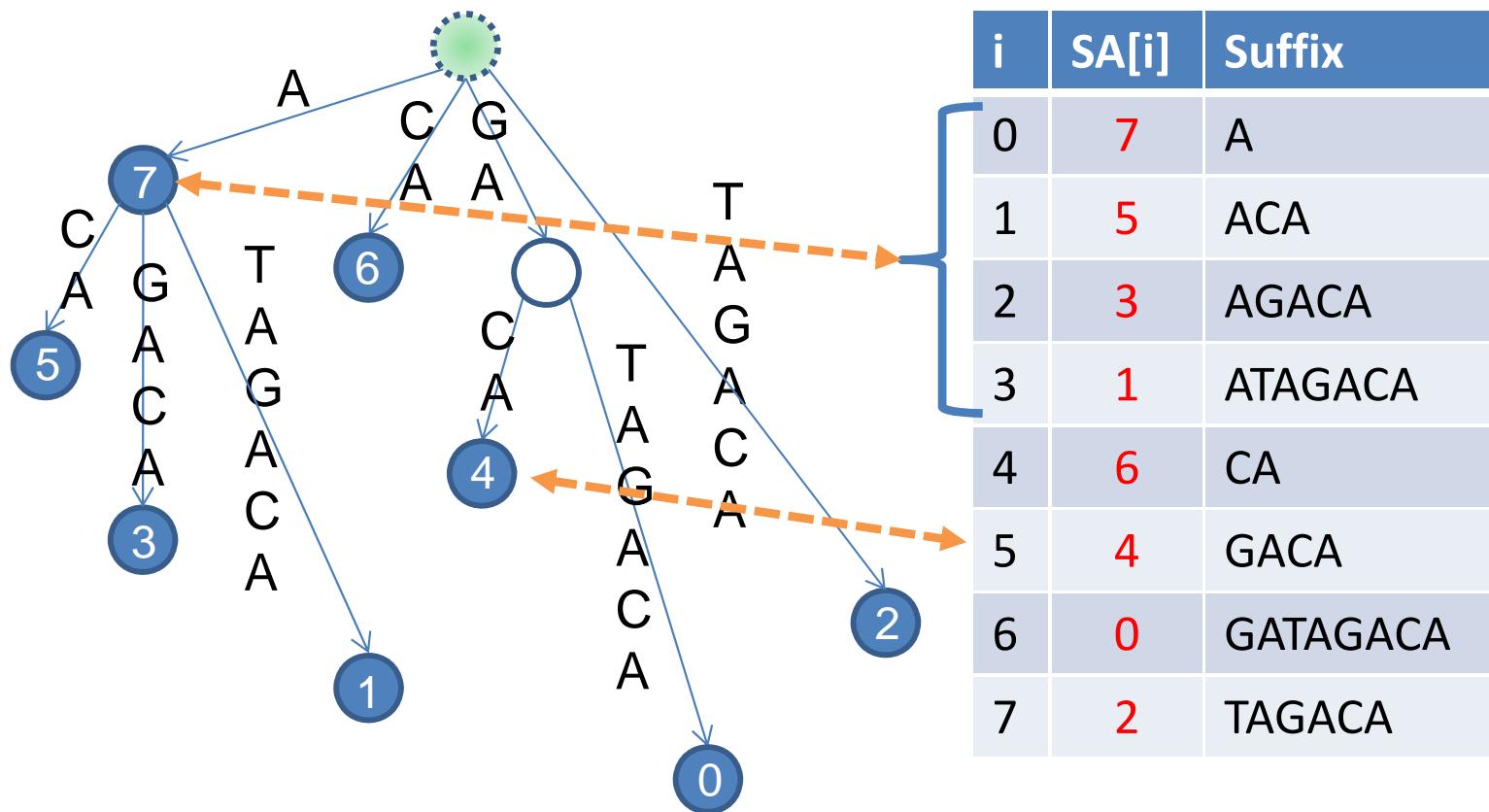
| i | Suffix   |
|---|----------|
| 0 | GATAGACA |
| 1 | ATAGACA  |
| 2 | TAGACA   |
| 3 | AGACA    |
| 4 | GACA     |
| 5 | ACA      |
| 6 | CA       |
| 7 | A        |

Sort →

| i | SA[i] | Suffix   |
|---|-------|----------|
| 0 | 7     | A        |
| 1 | 5     | ACA      |
| 2 | 3     | AGACA    |
| 3 | 1     | ATAGACA  |
| 4 | 6     | CA       |
| 5 | 4     | GACA     |
| 6 | 0     | GATAGACA |
| 7 | 2     | TAGACA   |

# Suffix Array (2)

- Preorder traversal of the Suffix Tree visits the terminating vertices in Suffix Array order
- Internal vertex in ST is a **range** in SA
  - Each terminating vertex in ST is an **individual index** in SA = a suffix



# Easy/Slow Suffix Array Construction

```
#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

char T[MAX_N]; int SA[MAX_N];

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }

int main() {
    int n = (int)strlen(gets(T));
    for (int i = 0; i < n; i++) SA[i] = i;
    sort(SA, SA + n, cmp);
}
```

This is O(N)

What is the time complexity?  
Can we do better?



Most (if not all) applications related to Suffix Tree  
can be solved using Suffix Array

With some increase in time complexity

## **APPLICATIONS OF SUFFIX ARRAY**

# String Matching

- Given a Suffix Array SA of the string T
- Find occurrences of the pattern string P
- Example
  - T = ‘GATAGACA’
  - P = ‘GA’
- Solution:
  - Use Binary Search twice
    - One to get lower bound
    - One to get upper bound

# String Matching Animation

Finding lower bound

| i | SA[i] | Suffix   |
|---|-------|----------|
| 0 | 7     | A        |
| 1 | 5     | ACA      |
| 2 | 3     | AGACA    |
| 3 | 1     | ATAGACA  |
| 4 | 6     | CA       |
| 5 | 4     | GACA     |
| 6 | 0     | GATAGACA |
| 7 | 2     | TAGACA   |

Finding upper bound

| i | SA[i] | Suffix   |
|---|-------|----------|
| 0 | 7     | A        |
| 1 | 5     | ACA      |
| 2 | 3     | AGACA    |
| 3 | 1     | ATAGACA  |
| 4 | 6     | CA       |
| 5 | 4     | GACA     |
| 6 | 0     | GATAGACA |
| 7 | 2     | TAGACA   |

# Time Analysis

- Binary search runs at most  $\log n$  comparisons
- Each comparison takes at most  $O(m)$  time
- We run binary search twice
- In the worst case,  $O(2m \log n) = O(m \log n)$

# Longest Repeated Substring

- Simply find the highest entry in LCP array
  - $O(n)$

| i | SA[i] | LCP[i] | Suffix           |
|---|-------|--------|------------------|
| 0 | 7     | 0      | A                |
| 1 | 5     | 1      | <u>A</u> CA      |
| 2 | 3     | 1      | <u>A</u> GACA    |
| 3 | 1     | 1      | <u>A</u> TAGACA  |
| 4 | 6     | 0      | CA               |
| 5 | 4     | 0      | GACA             |
| 6 | 0     | 2      | <u>G</u> ATAGACA |
| 7 | 2     | 0      | TAGACA           |

Recall:  
LCP = Longest  
Common Prefix  
between two  
successive suffixes

# Longest Common Substring

- $T_1 = \text{'GATAGACA'}$
- $T_2 = \text{'CATA'}$
- $T = \text{'GATAGACA.CATA'}$
- Find the highest number in LCP array provided that it comes from two suffices with different owner
  - Owner: Is this suffix belong to string 1 or string 2?
- $O(n)$

| i  | SA[i] | LCP[i] | Owner | Suffix               |
|----|-------|--------|-------|----------------------|
| 0  | 8     | 0      | 2/NA  | .CATA                |
| 1  | 12    | 0      | 2     | A                    |
| 2  | 7     | 1      | 1     | <u>A</u> .CATA       |
| 3  | 5     | 1      | 1     | <u>A</u> CA.CATA     |
| 4  | 3     | 1      | 1     | <u>A</u> GACACATA    |
| 5  | 10    | 1      | 2     | <u>A</u> TA          |
| 6  | 1     | 3      | 1     | <u>AT</u> AGACACATA  |
| 7  | 6     | 0      | 1     | CA.CATA              |
| 8  | 9     | 2      | 2     | <u>C</u> ATA         |
| 9  | 4     | 0      | 1     | GACA.CATA            |
| 10 | 0     | 2      | 1     | <u>G</u> ATAGACACATA |
| 11 | 11    | 0      | 2     | TA                   |
| 12 | 2     | 2      | 1     | <u>T</u> AGACACATA   |

# Summary

- In this lecture, you have seen:
  - Various string related tricks
  - Focus on Suffix Tree and Suffix Array
- But... you need to practice using them!
  - Especially, scrutinize ch6\_03\_sa.cpp/java
  - Solve one UVa problem involving SA
  - We will have SA-contest next week ☺

# References

- CP2, Chapter 6
- Introduction to Algorithms, 2<sup>nd</sup>/3<sup>rd</sup> ed, Chapter 32