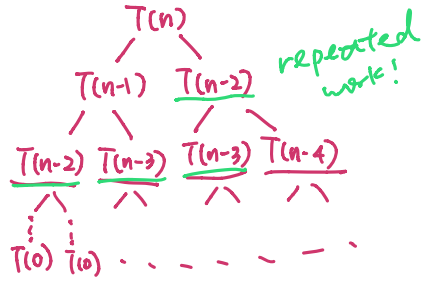
Lecture 1

1. Fibonacci Number

V1

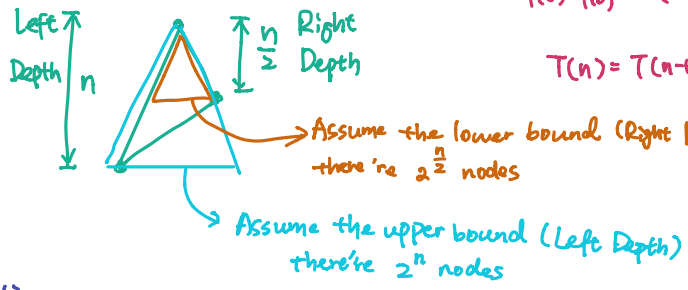
```
fib(int n){
    if (n==0 || n==1)
        return n
    return fib(n-1)+fib(n-2);
}
```

$T(n) = O(2^n)$



$\Rightarrow 2^{\frac{n}{2}} < T(n) < 2^n$

$(\sqrt{2})^n < T(n) < 2^n$



$T(n) = T(n-1) + T(n-2) + 1$

V2

```
fib(int n){
    A[0..... n]; // linear space
    A[0] = 0; A[1] = 1;
    for (i=2; i<=n; i++){
        A[i] = A[i-1]+A[i-2];
    }
}
```

$T(n) = O(n)$

V3

```
fib(int n){
    A[2]; // constant space
    A[0] = 0; A[1] = 1;
    for (i=2; i<=n; i++){
        A[i%2] = A[(i-1)%2] + A[(i-2)%2];
    }
    return A[n%2];
}
```

$T(n) = O(n)$

A: [a_1 | a_0]

e.g. i=2, A[0] = A[1] + A[0]

2%2 1%2 0%2

A: [a_2 | a_1]

i=3, A[1] = A[0] + A[1]

3%2 2%2 1%2

A: [a_2 | a_3]

V4

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a_2 & a_1 \\ a_1 & a_0 \end{bmatrix} \quad A^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} a_3 & a_2 \\ a_2 & a_1 \end{bmatrix}$$

$$A^3 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} a_4 & a_3 \\ a_3 & a_2 \end{bmatrix}$$

$$\vdots$$

$$A^n = \begin{bmatrix} a_{n+1} & a_n \\ a_n & a_{n-1} \end{bmatrix}$$

Anything in the form of matrix multiplication:

$A^n \Rightarrow O(\log n)$ ✓ $O(n)$ ×

e.g. $A^{64} = (A^{32})^2 = (A^{16})^2)^2 = (((A^8)^2)^2)^2 = ((((A^4)^2)^2)^2)^2 = (((((A^2)^2)^2)^2)^2)^2$

$A^{65} = (((((A^2)^2)^2)^2)^2)^2 \cdot A$

$$f(n) = \begin{cases} f(\frac{n}{2})^2 & , n \text{ is even number} \\ f(\frac{n-1}{2})^2 \cdot A & , n \text{ is odd number} \end{cases}$$

$\therefore O(\log n)$

f(n)
|
f(n/2)
|
f(n/4)
|
f(n/8)
⋮
f(1)

2. Asymptotic Notation: compare growth of functions

O	$f(n) = O(g(n))$	$\leq$ ∃ constants c and n_0, s.t. if $n > n_0$, then $f(n) \leq c \cdot g(n)$
o	$f(n) = o(g(n))$	$<$
Θ		$=$
Ω		$\geq$
ω		$>$

e.g. $f(n)=n \quad g(n)=2^n$

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

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

Alt. Method $\lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0 & f(n)=O(g(n)),\ f(n)=o(g(n)) \\ \text{constant} & f(n)=\Theta(g(n)),\ f(n)=O(g(n)),\ f(n)=\Omega(g(n)) \ (\text{eg. } 5\le 5,\ 5\ge 5) \\ \infty & f(n)=\Omega(g(n)),\ f(n)=\omega(g(n)) \end{cases}$

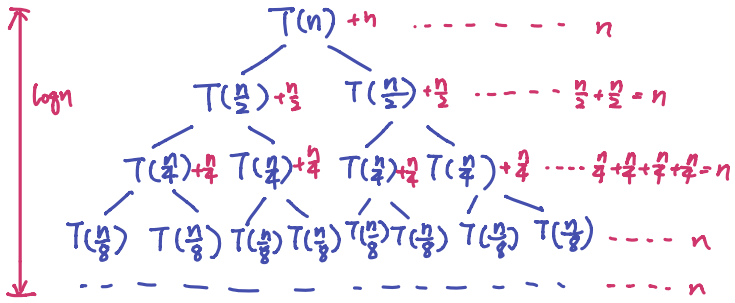
3. Recursion

Merge Sort


```
merge_sort (A[1 ..... n]) {
  if (n ≤ 1) return A;
  left = merge_sort(A[1, ..... n/2]
  right = merge_sort (A[n/2 +1, ..... n])
  return merge(left, right);
```

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

① Recursion Tree $T(n)=\Theta(n\log n)=O(n\log n)$



② Master Theorem

$T(n)=a\cdot T(\frac{n}{b})+n^c$

e.g. $T(n)=2T(\frac{n}{2})+n$ ✓ $T(n)=T(n-1)+T(n-2)+1$ ✗

$T(n)=T(n-3)+1$ ✗ $T(n)=T(\frac{n}{3})+T(\frac{2n}{3})+n$ ✗

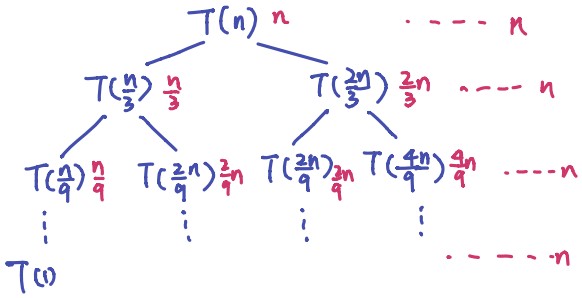
$\log_b a$ v.s. c

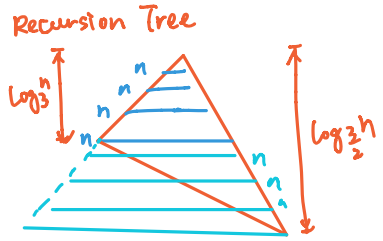
#1 $\log_b a > c$ $T(n)=O(n^{\log_b a})$

#2 $\log_b a = c$ $T(n)=O(n^c \log n)$

#3 $\log_b a < c$ $T(n)=O(n^c)$

Run time of $T(n)=T(\frac{n}{3})+T(\frac{2}{3}n)+n$?





$n\cdot\log_3 n \le T(n) \le n\cdot\log_{\frac{3}{2}} n$

e.g. $T(n)=2T(\frac{n}{2})+1$ $T(n)=\Theta(n)$

$T(n)=2T(\frac{n}{2})+n^2$ $T(n)=\Theta(n^2)$

$T(n)=4T(\frac{n}{2})+n^2$ $T(n)=\Theta(n^2\log n)$

$T(n)=8T(\frac{n}{2})+n^3$ $T(n)=\Theta(n^3\log n)$

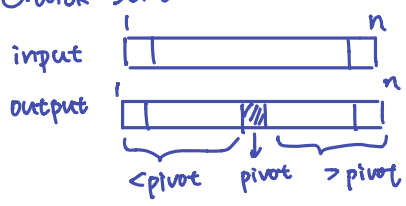
$T(n)=7T(\frac{n}{2})+n^2$ $T(n)=\Theta(n^{\log_2 7})$

Compare $(\log n)^2$, $n^{\frac{1}{2}}$.

$f(n)=2^n$ $g(n)=3^n$. $\lim_{n\to\infty}\frac{2^n}{3^n}=\lim_{n\to\infty}(\frac{2}{3})^n=0$ $\therefore f(n)=O(g(n))$

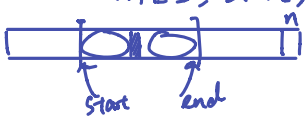
$f(n)=\log_2 n$, $g(n)=\log_3 n$ $\lim_{n\to\infty}\frac{\log_2 n}{\log_3 n}=\frac{\frac{\log n}{\log 2}}{\frac{\log n}{\log 3}}=\frac{\log 3}{\log 2}$ (constant) $\therefore f(n)=\Theta(g(n))$

4. Quick Sort



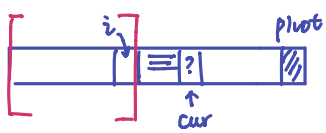
partition

partition (A[ ], start, end) → pivot position



O(n)

```
quick_sort (A[ ], start, end) {
  if (start ≥ end) return;
  pivot_pos = partition (A[ ], start, end);
  quick_sort (A[ ], start, pivot_pos - 1);
  quick_sort (A[ ], pivot_pos, end);
}
```



① A[cur] ≥ pivot cur+=1

② A[cur] < pivot swap(A[i+1], A[cur])

i+=1 cur+=1

Ideally: partition

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

Scenario 1: Draw recursion tree (imbalanced)

$T(n) = T(\frac{n}{3}) + T(\frac{2}{3}n) + n = O(n \log n)$

Scenario 2: (More imbalanced)

$T(n) = T(\frac{n}{10}) + T(\frac{9}{10}n) + n = O(n \log n)$

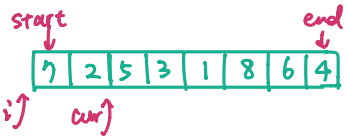
Scenario 3: 

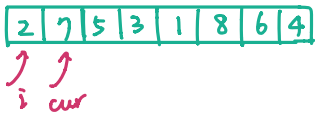
$T(n) = T(n-1) + n = O(n^2)$

Scenario 4: Every time we choose the C^{th} largest as pivot

$T(n) = T(n-c) + T(c) + n = O(n^2)$ ($T(c)$: constant)

```
partition(A[], start, end){
    i = start - 1;
    pivot = A[end];
    for(cur = start; cur < end; cur++){
        if(A[cur] < pivot){
            swap(A[i+1], A[cur]);
            i += 1;
        }
    }
    swap(A[i+1], A[end]);
    return i+1;
}
```

e.g. A 



Worst case: $O(n^2)$

Average case: $O(n \log n)$ as long as the input is somewhat random

If input is sorted → worst case running time

Good strategy: ① randomize the selection of pivot

∵ picking the smallest / largest element is $\frac{1}{n}$ (worst case)

throughout the whole sort, every time we pick the smallest/largest element is $\frac{1}{n} \cdot \frac{1}{n-1} \cdot \frac{1}{n-2} \cdots\cdots$ which is very small

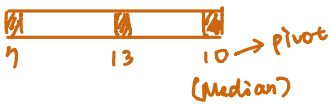
If there are lots of duplicated values:

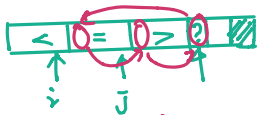
#1 partition  (original)

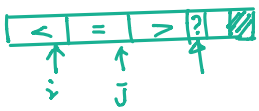
#2 partition 

(Return 2 positions)

② Median of three (start, mid, end), if the input is almost sorted

e.g. 





① A[cur] < pivot swap(A[j+1], A[cur]) swap(A[i+1], A[j+1])
i++; j++; cur++;

② A[cur] = pivot swap(A[j+1], A[cur])
j++; cur++;

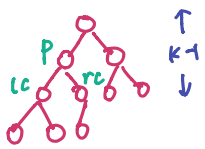
③ A[cur] > pivot cur++

Lecture 2

Heap Sort

Max Heap: insert(E) O(lgn)
get_max() O(1)
delete_max() O(lgn)

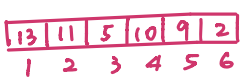
Logical View

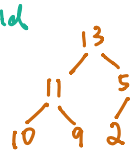


"Almost full" Binary Tree

parent → left child
parent → right child

Physical view

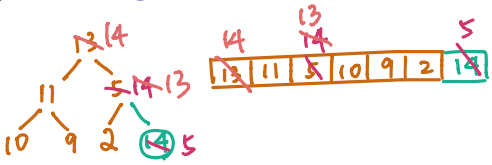
A 



parent(i) → i/2
lc(i) → 2i
rc(i) → 2i+1

get_max() ⇒ A[1]

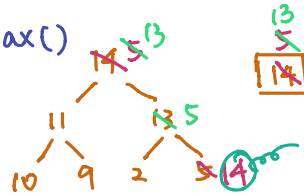
insert(14)

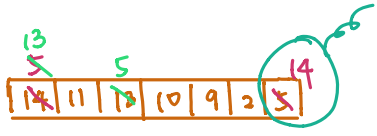


O(lgn)

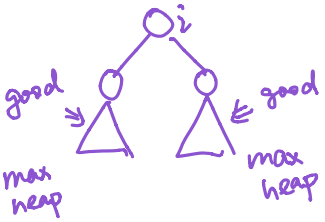
```
insert(A[], k){
    n = n+1, A[n] = k
    while(parent(i) > 0 && A[parent(i)] < A[i]){
        swap(A[parent(i)], A[i]);
        i = parent(i);
    }
}
```

delete_max()



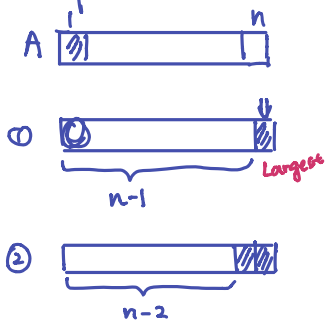



```
delete_max(A[]){
    swap(A[1], A[n])
    n = n-1;
    heapify(A[], 1);
}
```




```
heapify(A[], i){   O(logn)
    largest = i;
    if(lc(i) ≤ n && A[lc(i)] > A[largest]){ largest = lc(i); }
    if(rc(i) ≤ n && A[rc(i)] > A[largest]){ largest = rc(i); }
    if(largest == i) return;
    swap(A[i], A[largest]);
    heapify(A[], largest);
}
```

Heap



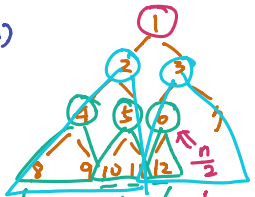

```
heapsort(A[]){
    make_heap(A[])   O(n)
    for(i=1 .... n-1)
        delete_max(A[])   } O(nlogn)
}
```

⇒ O(nlogn)

make_heap

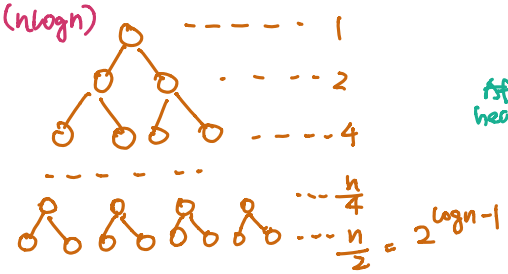
① for(i=1 .... n) insert(A[], A[i]) Θ(nlogn)
eg. (6, 5, 4, ... 1)

② for(i = n/2 .... 1) Θ(n)
heapify(A[], i);



after heapify: valid heap

the 1st node that doesn't have 2 children



Geometric Series

$1, \frac{1}{2}, \frac{1}{2^2}, \frac{1}{2^3}, \cdots, \frac{1}{2^n} = \frac{1-(\frac{1}{2})^{n+1}}{1-\frac{1}{2}}$

Arithmetic Series

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

$S = \frac{n}{2}\times 1 + \frac{n}{4}\times 2 + \frac{n}{8}\times 3 + \cdots + 1\times\log n$

$\frac{1}{2}S = \frac{n}{4}\times 1 + \frac{n}{8}\times 2 + \frac{n}{16}\times 3 + \cdots + \frac{2}{2}\cdot(\log n - 1) + \frac{1}{2}\cdot\log n$

$S = O(n)$

Theorem: any comparison based sorting algorithms run in O(nlogn)

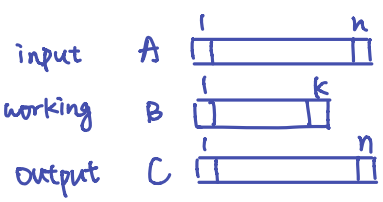
"Linear" time sorting algorithm

- counting sort

$\forall i, 1 \le A[i] \le k$

```
for(i=1 ..... n)
    B[A[i]] += 1
for(i=2 ..... k)
    B[i] = B[i-1] + B[i]
for(i=n ..... 1)   Why not from 1 .... n?
    C[B[A[i]]] = A[i]
    B[A[i]] -= 1
```

Because n .... 1 preserves the relative ordering, making counting sort stable



e.g.

input A: 1 1 3 2 1 5 2 4 1 5

working B: 4 2 1 1 2 — counting the occurrence

B: 4 6 7 8 10 — B[i] = B[i-1] + B[i]

output C: 1 1 1 1 2 2 3 4 5 5

B: 4 6 7 8 8

Time: O(n+k)!!

Space: O(n+k)

- radix sort (digit sort)

least significant ⇒ most significant digit.

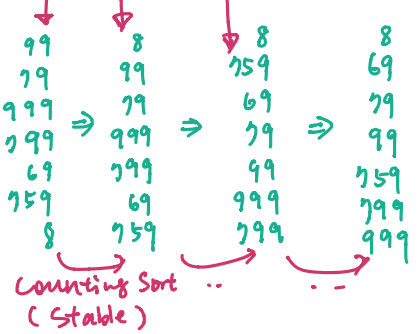
counting sort on all numbers

$O(dn)$

$O[d(n+k)] = O(dn + 10d) = O(dn)$

(10: [0 .... 9])

e.g.



Lecture 3 D&C - Merge Sort - Quick Sort

① Order statistics

input: a list of numbers

Q: median? 25th percentile? 20th largest/smallest number?

⇓

kth smallest number ← A[1 ····· n], k

#1
- Sort A
- A[k]

$O(n \lg n)$

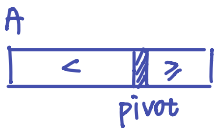
#2
- Heap
- make_min_heap $O(n)$
- delete k times $O(k \lg n)$

$O(k \lg n + n)$

if k is median → $\frac{n}{2}$

25th → $\frac{n}{4}$

#3
- partition (quick select)



case 1 pivot == k

case 2 pivot_pos > k : ignore right, recurse on left

case 3 pivot_pos < k : ignore left, recurse on right

#4 CLRS. $O(n)$

e.g. [7|3|5|1|6|2|4] k = 6

① partition (right)

[3|1|2|4|6|5|7] k = 2

↓ pivot_pos = 4

② partition k = 2

[6|5|7]

↓ #5 ↓ pivot_pos = #7

rank = 7 − 5 + 1 = 3 , k < rank

③ partition

[6|5] ⇒ [5|6] k = 1

↑ pivot_pos = #5 , rank = 5 − 5 + 1 = 1 < k = 2

```
kth(A[], begin, end, k){
    pivot_pos = partition(A[], begin, end)
    rank = pivot_pos - begin + 1;
    if (rank == k) return A[pivot_pos];
    if (k < rank) return kth(A[], begin, pivot_pos - 1, k);
    (if k > rank) return kth(A[], pivot_pos + 1, end, k - rank);
}
```

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

$T(n) = T(\frac{2}{3}n) + n = O(n)$

$T(n) = T(\frac{99}{100}n) + n = O(n)$

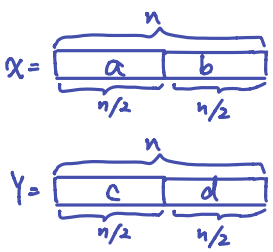
} master's theorem

$T(n) = T(n-1) + n = O(n^2)$

Depends on how good we can partition the array

② Integer Multiplication

$a = [13][73] = [13] \times 10^2 + [73]$



n-digit

$x \cdot y = (a \cdot 10^{n/2} + b)(c \cdot 10^{n/2} + d)$ ($\frac{n}{2}$ digit)

$= \underbrace{ac}_{P_1} \cdot 10^n + \underbrace{(bc + ad)}_{P_3 - P_1 - P_2} \cdot 10^{n/2} + \underbrace{bd}_{P_2}$

shifting

Let $p_1 = ac$, $p_2 = bd$, $p_3 = (a+c)(b+d)$

$T(n) = 4T(\frac{n}{2}) + O(n)$

$= O(n^2)$

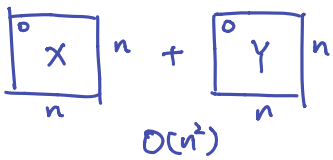
$\Rightarrow T(n) = 3T(\frac{n}{2}) + O(n)$

$= O(n^{\log_2 3})$

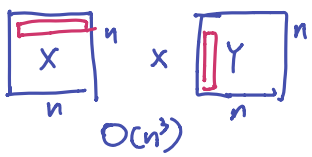
$\simeq O(n^{1.6})$

③ Strassen's Algorithm

Square matrix multiplication



$O(n^2)$



$O(n^3)$

Addition $O(n^2)$

$T(n) = 8 \cdot T(\frac{n}{2}) + O(n^2) = O(n^3)$

⇓

$T(n) = 7 \cdot T(\frac{n}{2}) + O(n^2) = O(n^{\log_2 7})$

Block Matrix

$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}_{n \times n}$ $Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}_{n \times n}$

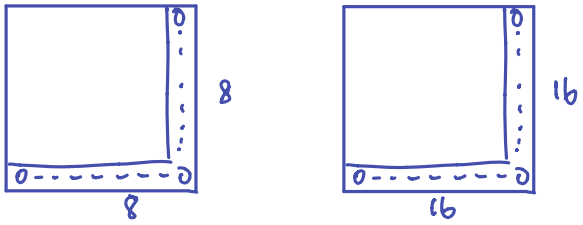
$X \cdot Y = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \times \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$

$P_4+P_5+P_6-P_2$ P_1+P_2

P_3+P_4 $P_1+P_5-P_3-P_7$

$p_1 = A \cdot (F - H)$
$p_2 = (A+B) \cdot H$
$p_3 = (C+D) \cdot E$
$p_4 = D \cdot (G - E)$
$p_5 = (A+D)(E+H)$
$p_6 = (B-D) \cdot (G+H)$
$p_7 = (A-C) \cdot (E+F)$

What if n is odd?
e.g. 7×7 padding 11×11

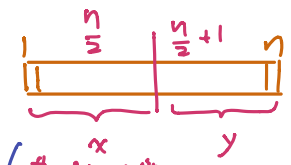


④ Counting Inversions

Input : a list of distinct numbers
A[2,3,1] e.g. (2,1), (3,1) $A[i] > A[j]$ && $i < j$
Output : # of inversions
a pair of numbers

```
for(i=1 ..... n)
  for(j=i+1 ..... n)
    if(A[i] > A[j]) res+=1
```
$O(n^2)$

```
count_inversion(A[]){
  x = count_inversion(A[1 : n/2])
  y = count_inversion(A[n/2+1 : n])
  sort(A[1 : n/2])
  sort(B[n/2+1 : n])          } O(nlgn)
  z = cross_inversions(A[1 : n/2], A[n/2+1 : n])   O(n)
  return x+y+z
}
```
$T(n) = 2T(\frac{n}{2}) + nlgn = O(nlg^2n)$



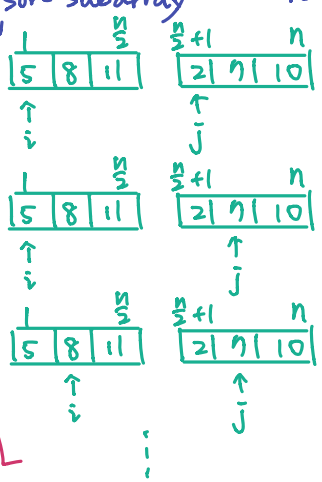
# of inversion

Cross Inversion

```
for(i=1 .... n/2)
  for(j=n/2+1 ..... n)
    if(A[i] > A[j]) z+=1
```
$O(n^2)$. not efficient.

Sort subarray

return x+y+z



① $5 > 2$, so [all elements after 5] > 2
# of inversions += $\frac{n}{2}$

② $5 < 7$, move i

③ $8 > 7$
# of inversions += $\frac{n}{2} - i$

☆ Update : let count_inversion tell us # of inversions and sort A
☆ A = merge(A[1 : n/2], A[n/2+1 : n])
$T(n) = 2T(\frac{n}{2}) + O(n)$

⑤ Top k

⇒ { } top k smallest numbers

#1 Sort(A)
return A[1:k]
$O(nlgn)$ time, $O(1)$ space

#2
```
make_min_heap(A[1:n])
for(i=1 ---- k)
  delete_min()  # move min to the end
return A[n-k+1, n]
```
$O(n + klgn)$, $O(1)$
$O(n)$ if k is small
$O(nlgn)$ if $k \to n$

#3
```
kth(A[1:n], k)
return A[1:k]
```
$O(n)^*$, $O(1)$

#4 max heap of size k

```
make_max_heap(A[1:k])
for(i=k+1 ... n){
  if(A[i] < get_max())
    delete_max()
    insert(A[i])
}
```
$O(k) + O((n-k)lgk)$, $O(1)$
$O(k + (n-k)lgk) \Rightarrow O(n)$ if k is small
($(n-k)lgk \approx nlgk$)
$O(n)$ if $k \to n$

Space Complexity

Quick Sort

#1 O(1) – does not count recursion calls
⇒ #2 O(lgn)* – recursion calls take space on call stack

} Overall, O(1) + O(lgn)

Lecture 4

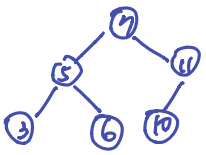
BST (Binary Search Tree)
Hashing (Set<key>, Map<key, value> { TreeMap / HashMap insert / delete / look up

	Insert	Delete	LookUp	
BST (balanced)	O(lgn)	O(lgn)	O(lgn)	Time
Hashing	O(1)*	O(1)*	O(1)*	Time
	O(n) ↳ worstcase	O(n)		
BST (balanced)				Space
Hashing				Space

* If we need to traverse the elements in order, must use tree map.

Binary Tree

Properties: 1) binary tree
2) root ≥ all values in left subtree
root < all values in right subtree



Tree Traversal Algo:
- pre-order
- in-order
- post-order

```
class TreeNode {
    int key;
    TreeNode left;
    TreeNode right;
}
```

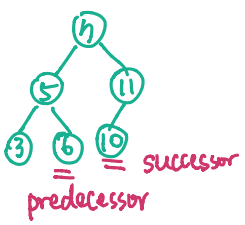
```
inorder (TreeNode root){
    if (root == null) return;
    inorder(root.left);
    print(root);
    inorder(root.right);
}
```

preorder: 7, 5, 3, 6, 11
Inorder: 3, 5, 6, 7, 11
post-order:

O(depth)

```
Insert(TreeNode root, int key) {
    if (root == null)
        return new TreeNode(key);
    if (key < root.val)
        root.left = insert(root.left, key);
    else
        root.right = insert(root.right, key);
    return root;
}
```

delete



Case 1: no children
delete it
Case 2: Single Child
Replace by this single child
Case 3: Two Children
- 3.1 find successor (left-most node in right subtree)
- 3.2 replace
- 3.3 delete successor node

```
delete (TreeNode root, int key) {
    if (root == null) return root;
    if (key < root.val)
        root.left = delete (root.left, key);
    else if (key > root.val)
        root.right = delete (root.right, key);
    else {
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        TreeNode cur = root.right;
        while (cur.left != null) {
            cur = cur.left;
        }
        root.val = cur.val;
        root.right = delete(root.right, cur.val)
    }
    return root;
}
```

Balanced BST
- AVL tree
- Red-Black tree
- 2,3 tree
- 2,3,4 tree

Find successor
- Case 1 : if have right subtree
- Case 2 :

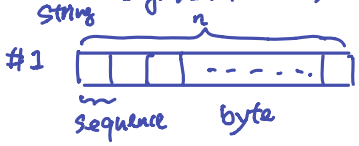
Hashing [generalized array]

String / int / ~~double~~ / user_defined data type

pre-hash: $f(x) = x$ → int (large universe, e.g. hashCode())

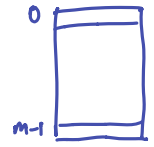
hash → int (much smaller universe)

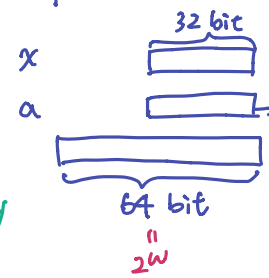
```
class Pair {
    int x;
    int y;
}
prehash(x) x 41 + prehash(y);
```

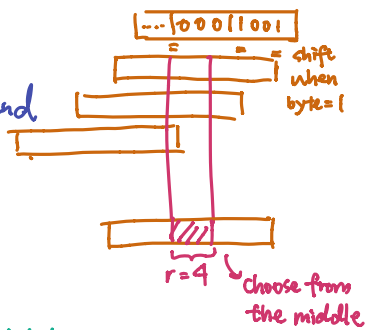


#1 $h_1(x) = x \% m$

#2 $f(x) = (a \cdot x \% 2^w) \gg (w-r)$







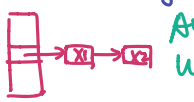
* Why don't we use prehash directly?
- sometimes hashcode from prehash could really large, it could cause a waste of space

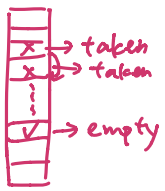
↗ if α is too large, collision is likely

Load factor $\alpha = \frac{n}{m}$ ($\alpha \uparrow$ utilization is good)

≈ 1

collision { - chaining → Average: $O(1+\alpha)$, Worst: $O(n)$ → lookup / insert / delete
- open addressing





Chaining	V.S.	Open Addressing
Implement: Easy		Hard
Space: Waste space		Save space

$O\left(1 + \frac{1}{1-\alpha}\right)$

probability of a slot that's empty

Lecture 5

- Dynamic Programming

① Optimization / Counting / Feasibility ····· need DP
② Design Recursion
③ Run time is bad (e.g. naive impl, exponential)
④ cache/remember result { Bottom up / Top down with memoization

Knapsack Problem

0 - 1

Inputs:
- n items
- ∀ items i, value[i], weight[i]
- Knapsack Capacity C

Question: max # value

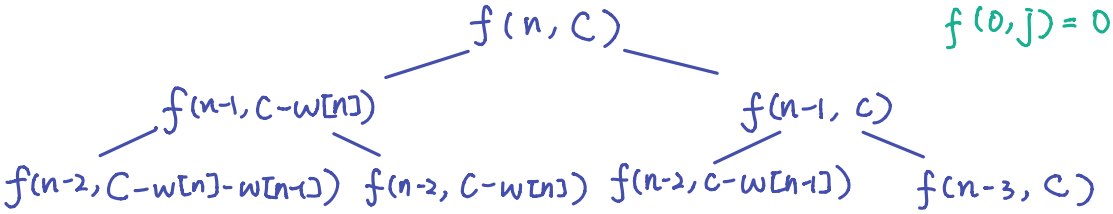
Solution (Brute force): $O(2^n)$ ↷ # of possibilities

DP: $f(i,j)$ = max # we can get if value[1 ····· i], weight[1 ····· i], knapsack cap is j.

- if take item i, $f(i,j) = V[i] + f(i-1, j-W[i])$
- if don't take i, $f(i,j) = f(i-1, j)$

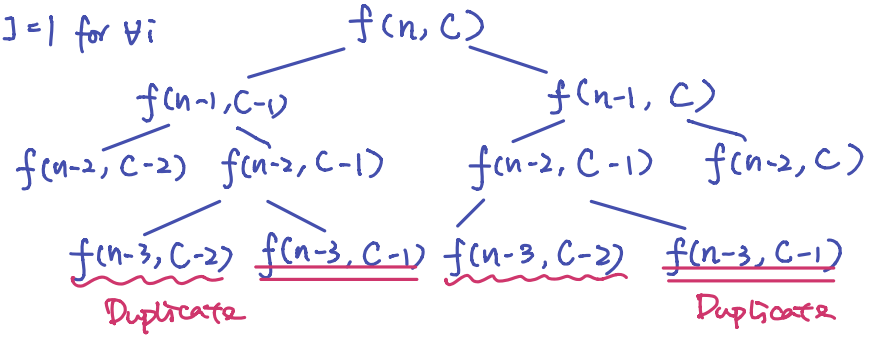
} max

$$f(i,j) = \begin{cases} V[i] + f(i-1, j-w[i]) & \text{if } w[i] \le j \\ f(i-1, j) \end{cases}$$



$f(0,j) = 0$, $f(n,0) = 0$

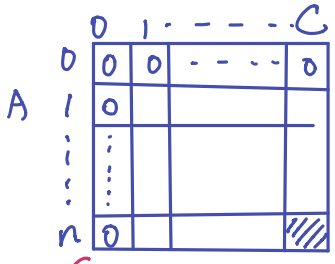
$w[i] = 1$ for $\forall i$



Bottom Up

$f(i,j)$

$A[i,j] = f(i,j)$

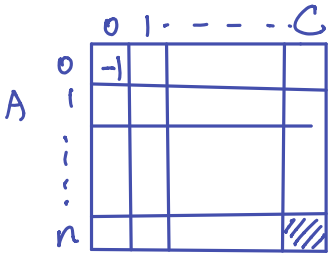


initialize 2D array with 0

```
for (i = 1 .... n) {
  for (j = 1 .... C) {
    if (w[i] ≤ j)
      A[i,j] = max(V[i] + A[i-1, j-w[i]],
                   A[i-1, j])
    else
      A[i,j] = A[i-1, j]
  }
}
return A[n, C]
```

Time & Space $O(nC)$

Top Down



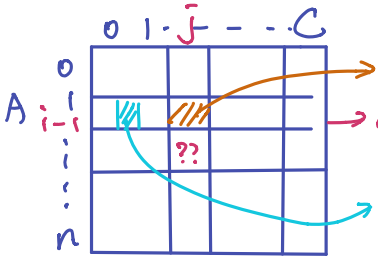
initialize 2D array with -1

```
knapsack(V[], w[], i, j) {
  if (i == 0 || j == 0) return 0;
  if (A[i,j] != -1) return A[i,j];   ↳ if has been visited
  if (w[i] ≤ j) {
      s = max { knapsack(V[], w[], i-1, j-w[i]) + V[i],
                knapsack(V[], w[], i-1, j) }
  } else {
      s = knapsack(V[], w[], i-1, j);
  }
  A[i,j] = s;
  return s;
}
```

Time & Space $O(nC)$

for every value, we visit once

Reduce Space




```
for (i = 1 .... n)
  for (j = 1 .... C)
    if (w[i] ≤ j)
      A[i%2, j] = max { V[i] + A[i-1%2, j-w[i]],
                        A[i-1%2, j] }
    else
      A[i%2, j] = A[i-1%2, j]
return A[n%2, C]
```

- knapsack w/ repetition (complete knapsack)
  - n types of item
  - each type, there are unlimited copies
  - type i, value[i], weight[i]
  - knapsack capacity is C
  - Q: max return?

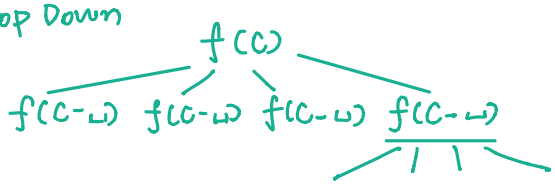
Recursion

f(i) := max $ return w/ knapsack of capacity i

f(i-1)
f(i-2)
⋮
f(2)
f(1)
f(0)

$$f(i) = \max \begin{cases} value[1] + f(i - weight[1]) \\ value[2] + f(i - weight[2]) \\ \vdots \\ value[n] + f(i - weight[n]) \end{cases}$$

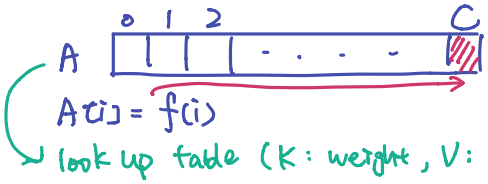
Example: Top Down



Lots of re-computation

weight[i] ≥ i

$f(i) = \max\{value[k] + f(i - weight[k])\}$



Time: O(nC)
Space: O(C)

```
for(i=1 ..... C){
  for(k=1 ..... n){
    if(weight[k] ≤ i){
      A[i] = max(A[i], value[k] + A[i-weight[k]])
    }
  }
}
return A[C]
```

- Coin Change

1, 5, 10, 25 make change of $4.21
d_1 d_2 d_3 d_4 what's minimum # of coins?

n := amount of cents to make change for
k := $d_1, d_2 \cdots d_k$

d[1 .... k], n


```
for(i=1 ..... n){
    A[i] = i                    O(nk)
    for(j=2 ... k){
        if(d[j] <= i)
            A[i] = min(A[i], 1+ A[i-d[j]])
    }
}
return A[n];
```

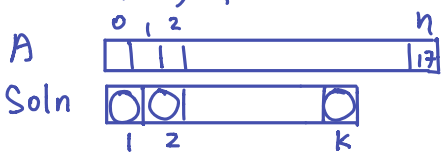
Recursion

f(i) = minimum # of coins to make changes for i

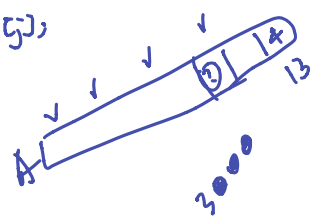
$$f(i) = \min \begin{cases} 1 + f(i-1) \\ 1 + f(i-5) \\ 1 + f(i-10) \\ 1 + f(i-25) \end{cases}$$

$f(i) = \min(1 + f(i-k))$

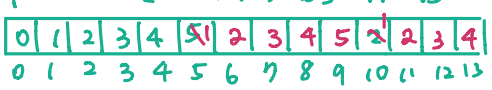
Find out how many of each coins we take:




```
while(n>0){
    for(j=1 ..... k){
        if(d[j] <= n && A[n-d[j]] = A[n]-1){
            soln[j] += 1;
            n -= d[j];
            break;
        }
    }
}
```

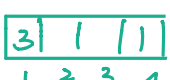


Example d=[1, 5, 10, 25] n = 13



① min(5, 1+A[5-5]) = 1
② min(6, 1+A[6-5]) = 2
③ min(7, 1+A[7-5]) = 3
⋮
④ min(10, 1+A[10-5]) = 2
  min(2, 1+A[10-10]) = 1
⑦ min(11, 1+A[11-5]) = 3
  min(3, 1+A[11-10]) = 2

Soln



① A[13-1] = A[13]-1 = 3
  soln[1] = 1
  n = 13-1 = 12

② A[12-1] = A[12]-1 = 2
  soln[1] = 1+1 = 2
  n = 12-1 = 11

③ A[11-1] = A[11]-1 = 1
  Soln[1] += 1 = 3
  n = 11-1 = 10

④ A[10-1] ≠ A[10]-1
  A[10-5] ≠ A[10]-1
  A[10-10] = A[10]-1
  soln[10] += 1 = 1 n = 10-10 = 0

- Subset Sum ⇒ Similar to 0, 1 knapsack

{ a set of ints
target T

∃ a subset of ints that sums up to T

f(i,j) := T / F given A[1 ··· i] . j

f(i,j) = f(i-1, j-A[i]) || f(i-1, j)

- LCS (Longest common subsequence)

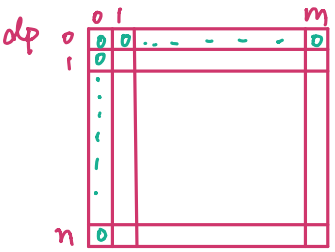
A = [5, 10, 1, 3, 7, 11] Q: what's the length of LCS?
B = [2, 5, 8, 1, 7, 10]

f(i,j) = length of LCS given A[1 ····· i], B[1 ····· j]

① A[i] ≠ B[j] f(i,j) = max{f(i-1,j), f(i,j-1)}

② A[i] = B[j] f(i,j) = f(i-1,j-1)+1

③ Base case: f(0,j) = f(i,0) = 0



|A| = n for (i = 1 ··· n)
|B| = m for (j = 1 ····· m)

O(nm)

```
if (A[i] == B[j]) dp[i,j] = dp[i-1,j-1] + 1
else dp[i,j] = max(dp[i-1,j], dp[i,j-1])
return dp[n,m]
```

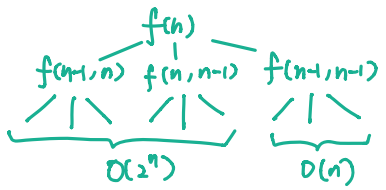
How to find an instance of LCS?

O(n+m)

```
while (n > 0 && m > 0) {
    if (A[n] == B[m])
        L.append(A[n])
        n = n-1
        m = m-1
    else
        if (dp[n-1,m] < dp[n,m-1])
            m = m-1
        else
            n = n-1
}
return L
```

a b c d e
a c e

dp[1,1] = 1
dp[1,2] = 2
dp[1,3] = ?



Redundant Computation

- Matrix Multiplication

A1 × A2 × ···· × AN

A1 × A2 = Po[ ] × P1[ ] = Po[ ]
Po×P1 P1×P2 P1 P2 P2

each takes O(p1) to compute

O(po × p2 × p1)

# of numbers we need to compute

(A1 × A2) × A3
100×10 10×100 100×5

#1 100×10×100 = 100,000
100×100×5 = 50,000
cost = 150,000

A1 × (A2 × A3)
100×10 10×100 100×5

#2 10×100×5 = 5,000
100×10×5 = 5,000
cost = 10,000

((A1 × A2) × A3) × A4
(A1 × (A2 × A3)) × A4
A1 × ((A2 × A3) × A4)
A1 × (A2 × (A3 × A4))

- Po, P1, P2, ···· Pn f(i) := min cost to multiply A1 × A2 × ···· × Ai

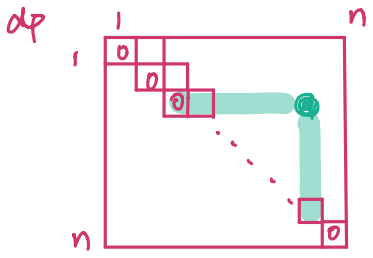
A1 Po×P1
A2 P1×P2
⋮
An Pn-1×Pn

Last multiplication

A1 × [A2 × ··· × An]
=
Po×P1 P1×Pn Po×P1×Pn + f(2,n)

lost of last op

$$f(i,j) = \min \begin{cases} f(2,n) + P_0 \times P_1 \times P_n & P_{i-1} \times P_i \times P_j \\ f(1,2) + f(3,n) + P_0 \times P_2 \times P_n & P_{i-1} \times P_{i+1} \times P_j \\ \vdots & P_{i-1} \times P_{j-1} \times P_j \\ f(1,n-1) + P_0 \times P_{n-1} \times P_n \end{cases}$$




```
P[0 . . . . n]
    for (i=1 . . . . n-1) dp[i, i+1] = p[i-1] p[i] p[i+1]
    for (j=3 . . . . n) {
        for (i=j-2 . . . . 1){
            for(k=i . . . . j-1){
                dp[i,j] = min{dp[i,k] + dp[k+1, j] + p[i-1] p[k] p[j]
                              dp[i,j]
            }
        }
    }
```

$O(n^3)$

Lecture 6

① OPT / counting / feasibility (LCS)
② Design recursions
③ naive implementation (running time)
④ { Bottom up
  { Top down (memoization)

Use which one? Consider { ① Easy to implement
② Space complexity (bottom up is easy to optimize space)
③ Sequence from small to large (top-down)

Palindrome

- Given a string, verify if a string is palindrome :  O(n)

- Split to min # of palindromes

  e.g. str = banana

  ① [b ana n a] n = 4

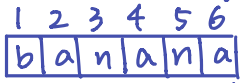
  ② [b a n a n a] n = 6

  ★ ③ [b anana] n = 2

size of problem

f(i) := min # of palindrome if length of given string is i

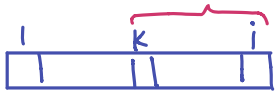
⇒

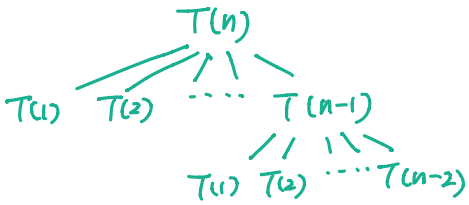


$f(6) = \begin{cases} f(5)+1 \\ f(3)+1 \\ f(1)+1 \end{cases}$

f(1)
f(2)
⋮
f(i-1)

Substr(a,b) (inclusive)





$f(i) = \min\{f(k-1)+1\}$ $k = 1 \cdots i$ if substr(k,i) is palindrome

$f(1) = 1$

$f(0) = 0$ (Empty String)

f(i) A 

str 


```
min_palindrome(str){
    A[0] = 0;
    for (i = 1 . . . . n){
        A[i] = i;          # initialize the trivial cases (each character is a palindrome)
        for (k = 1 . . . . i){          → O(n) ⇒ can preprocess is palindrome and reduce it to O(1)
            if (substr(k,i) is palindrome) {
                A[i] = min(A[k-1] + 1, A[i])
```

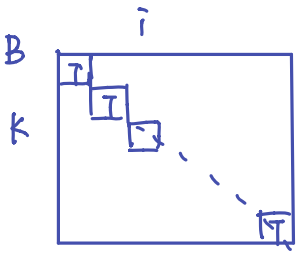
$O(n^3)$

```
    }
  }
}
return A[n];
}
```

isPalindrome (k, i)

$g(k,i) = \begin{cases} T & \text{if substr}(k,i) \text{ is palin} \\ F & \text{o/w} \end{cases}$

$\rightarrow g(k+1, i-1)$ && str[k] == str[i]



B[i,i] = T for ∀i

B[i, i+1] = (str[i] == str[i+1])

B[k] == B[i]

e.g. string = banana

B

k \ i	1	2	3	4	5	6
1	T	F	F	F	F	F
2		T	F	T	F	T
3			T	F	T	F
4				T	F	T
5					T	F
6						T

$O(n^2)$

B[k,i] = T iff substr(k,i) is palindrome

len 1 B[k,k] = T

len 2 B[k, k+1] = (str[k] == str[k+1])

len 3 B[1,3] = B[2,2] && (str[1] == str[3])

B[1,4] = B[2,3] && (str[1] == str[4])

B[2,4] = B[3,3] && (str[2] == str[4])

str = banana

i = 1	A[1] = 1	b	f(0)+1
i = 2	A[2] = 2	b a	f(1)+1 = 2
i = 3	A[3] = 3	b a n	f(2)+1 = 3
i = 4	A[4] = ~~3~~ 2	b a n a	f(1)+1 = 2 f(3)+1 = 4
i = 5	A[5] = ~~4~~ 3	b a n a n	f(2)+1 = 3 f(4)+1 = 3
i = 6	A[6] = ~~5~~ 2	b a n a n a	f(1)+1 = 2 f(3)+1 = 4 f(5)+1 = 4

Longest increasing path in Matrix # 1

A

3	4	1	7
.	.	.	.
.	.	.	.
.	.	.	.

positive ints

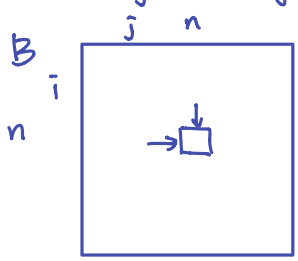
path
- start anywhere
- all #'s need to be increasing
- can either go right or downward by 1 step

Q: what's length of longest path?

f(i,j) := length of longest path to position (i,j)

$f(i,j) = \max \begin{cases} f(i-1,j)+1 & \text{if } A[i-1,j] < A[i,j] \\ f(i,j-1)+1 & \text{if } A[i,j-1] < A[i,j] \\ 1 & \text{(itself)} \end{cases}$

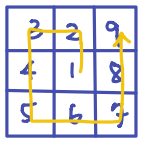
B




```
B[i,j] = f(i,j)
for(i=1 .... n){
    for(j=1 .... n){
        B[i,j] = 1
        if(i>1 && A[i-1,j] < A[i,j]) { B[i,j] = max(B[i,j], B[i-1,j]+1) }
        if(j>1 && A[i,j-1] < A[i,j]) { B[i,j] = max(B[i,j], B[i,j-1]+1) }
    }
}
return max(B)
```

Longest increasing path in Matrix #2

e.g.

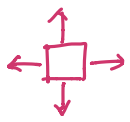


path { Start anywhere
increasing
move in any direction

$f(i,j)$:= length of longest path to position (i,j)

$$f(i,j) = \max \begin{cases} f(i-1,j)+1 & \text{if } A[i-1,j] < A[i,j] \\ f(i,j-1)+1 & \text{if } A[i,j-1] < A[i,j] \\ f(i+1,j)+1 & \text{if } A[i+1,j] < A[i,j] \\ f(i,j+1)+1 & \text{if } A[i,j+1] < A[i,j] \\ 1 \end{cases}$$

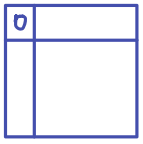
(?)



One value depends on 4 other values
What's the sequence to fill matrix?
→ the smallest value doesn't have any dependency

Bottom - Up

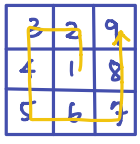
A n × n



n^2 integers
sort??

e.g. Quicksort

e.g. A 3 × 3



1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9
(2,2) (1,2) (1,1) (2,1) (3,1) (3,2) (3,3) (2,3) (1,3)

Start from smallest → largest

key = num, value = (i,j)

3:(1,1)	2:(1,2)	. . .	7:(3,3)

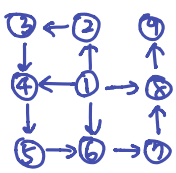
① (I) Sort : $O(n^2 \log(n^2)) = O(n^2 \log n)$
(II) Compute B : $O(n^2)$

② Using min heap: - make_heap $O(n^2)$
- n^2 { delete $O(\lg n^2)$ } $n^2 \log n$
$O(1)$

while (heap is not empty)
- take min
- compute B at (i,j)

③

Topological Sort



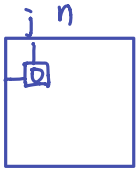
DAG $O(k^2)$

$O(n^2 \lg n) \to O(n^2)$

Top - down

input A n × n

memorization



f(i,j)

Doesn't need sorting
Recursion will sort out the dependency for us

```
recursion(i,j){
 * if (B[i,j] > 0) return B[i,j]; // return value if it exists
   val = 1;
   if (i > 1 && A[i-1,j] < A[i,j]){ val = max(val, recursion(i-1,j));}
   if (j > 1 && A[i,j-1] < A[i,j]){ val = max(val, recursion(i,j-1)); }
   if (i <= n && A[i+1,j] < A[i,j]){ val = max(val, recursion(i+1,j));}
   if (j <= n && A[i,j+1] < A[i,j]){ val = max(val, recursion(i,j+1));}
 * B[i,j] = val; // save the value before return
   return val;
}
```

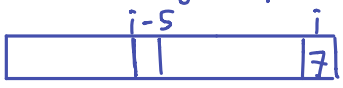
$O(n^2)$

Every value in the matrix is computed once → $O(n^2)$

Longest Increasing Subsequence (LIS)

- A [3|5|1|7|2] (3, 7, 2) ✗ (1, 2) ✓ (3, 5, 7) ✓

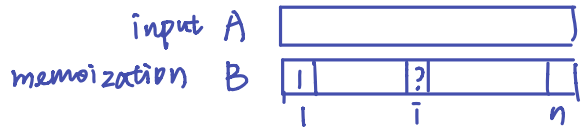
f(i) := length of LIS if given A[1 ··· i] that ends with A[i]



f(i-5) = 3

$$f(i) = \max \begin{cases} f(1)+1 & \text{if } A[i] > A[1] \\ f(2)+1 & \text{if } A[i] > A[2] \\ \vdots & \\ f(i-1)+1 & \text{if } A[i] > A[i-1] \\ 1 \end{cases} = \max \left\{ \max_{k=1 \cdots i-1} \{ f(k)+1 \text{ if } A[k] < A[i] \}, 1 \right\}$$

Bottom-Up



B[i] = 1 initialize

```
for (i = 2 ---- n) {
   for (k = 1 ···· i-1) {
      if (A[k] < A[i]) { B[i] = max (B[i], B[k]+1); }
   }
}
return max{B}
```

How to solve LIS with LCS?

A ⇒ A' } $O(n^2)$ LCS **Reduction**

$O(n \lg n)$ Sort(A) ⇒ B }

Edit Distance

input: A = "car", B = "cat"

allowed ops { insert char, delete char, replace char

e.g. car $\xrightarrow{\text{ins}}$ char, car $\xrightarrow{\text{delete}}$ cr, car $\xrightarrow{\text{replace}}$ cas

A ⇒ - - - - ⇒ B

car ⇒ ar ⇒ r ⇒ o ⇒ c ⇒ ca ⇒ cat

f(i,j) = min # of ops A[1...i] ⇒ B[1...j]

① A[1 ··· i] ⇒ B[1 ··· j-1] insert B[j] at the end : f(i, j-1)+1

② A[1 ··· i-1] ⇒ B[1 ··· j] delete A[i] at the end : f(i-1, j)+1

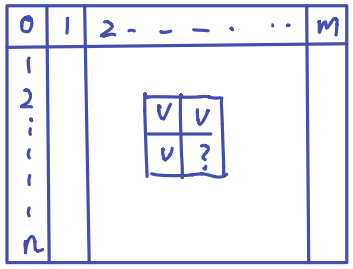
③ A[1 ··· i-1] ⇒ B[1 ··· j-1] replace A[i] with B[j] : f(i-1, j-1)+(1 or 0)

$$f(i,j) = \min \begin{cases} f(i-1,j)+1 & \text{delete} \\ f(i,j-1)+1 & \text{insert} \\ f(i-1,j-1)+1 \text{ or } 0 & \text{replace} \end{cases}$$

Base case : f(i, 0) = i delete ops

f(0, j) = j insert ops

dp




```
for (i = 1 ---- n) {
   for (j = 1 ---- m) {
      dp[i,j] = min (dp[i-1,j], dp[i,j-1]) + 1
      if (A[i] == B[j]) { dp[i,j] = min( dp[i,j], dp[i-1,j-1] )
      else { dp[i,j] = min (dp[i,j], dp[i-1,j-1] + 1)
   }
}
```

One more knapsack → knapsack w/ multiple constraints

- 0-1 knapsack
- complete knapsack (w/ repetition)

- n items
- each item i, there is only 1 copy
- each item i, { value[i] $ value; weight[i]; volume[i] }

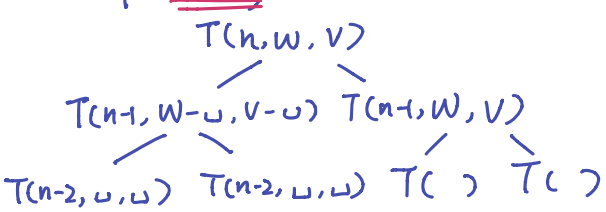
- knapsack { W capacity on weights; V capacity on volumes }
- Q: max value can get?

i: # items
j: weight cap of knapsack
k: volume cap of knapsack
f(i,j,k) = max value given { - first i items; - cap of { j; k } }

$f(i,j,k) = \max \begin{cases} value[i] + f(i-1, j-weight[i], k-volume[i]) \\ f(i-1,j,k) \end{cases}$

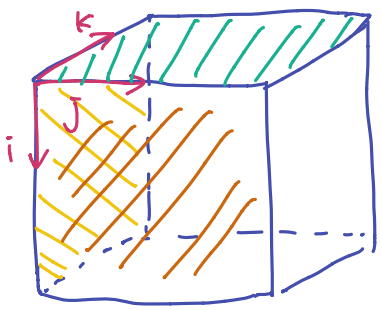
Base Case:
$f(0,i,j) = f(i,0,k) = f(i,j,0) = 0$
(if one of the dimension reaches 0, the value = 0)

① if impl naively, what would be the running time?? $O(2^n)$ → Optimize → $O(nWV)$



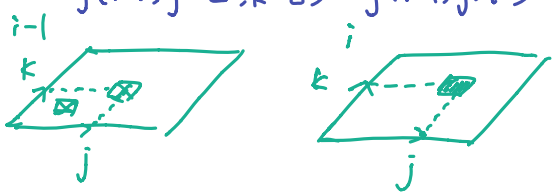
② space optimization f(i,j,k) space O(nWV)

③ easy to define sequence of instances from small to large? Yes, bottom up ✓ (Depends on smaller instances) No, top down



Base case: 3 planes (i,j), (i,k), (j,k)

f(i,j,k)
f(i-1,j-□,k-□) f(i-1,j,k)



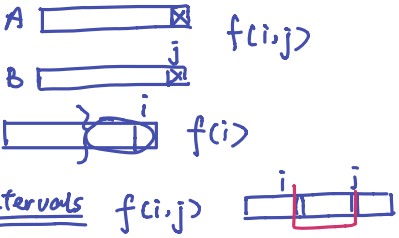

```
knapsack(value[1···n], weight[1···n], vol[1···n], W, V){
    for(i=1···n){
        for(j=1···W){
            for(k=1···V){
                A[i,j,k] = A[i-1,j,k];
                if(weight[i] ≤ j && vol[i] ≤ k){
                    A[i,j,k] = max(A[i,j,k], A[i-1, j-weight[i], k-volume[i]] + value[i])
                }
            }
        }
    }
    return A[n,W,V];
}
```

Time $O(nWV)$

Space $O(nWV)$ $\xrightarrow{i\ \%\ 2}$ $O(WV)$

Recursion Design

- knapsack { 0-1; complete; multiple constraint }
- LCS, Edit Distance, etc. {given two sequences} — A, B — f[i,j]
- Palindrome, LIS, etc — f(i)
- Matrix Multiplication — intervals — f[i,j]



Bottom Up VS. Top Down

- space (can recursion reduce space complexity?)
- sequence (can do from small to large?)

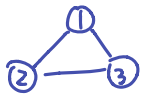
	Pros	cons
Bottom Up	Space, Runtime?	Seq?
Top Down	Seq	Space?

Graph

directed graph

$V = \{1, 2, 3\}$
$E = \{(1,2), (2,3), (1,3)\}$

undirected graph

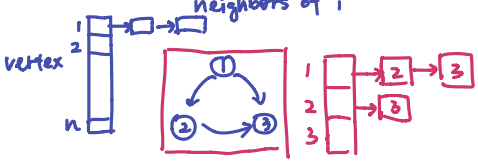


$V = \{1, 2, 3\}$
$E = \{(1,2), (2,1), (2,3), (3,2), (1,3), (3,1)\}$

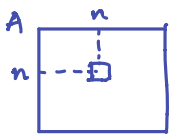
V = set of vertices $|V|$ = size of set $V = n$
E = set of edges $|E|$ = size of set $E = m$
$G = (V, E)$,

Graph Representation

Adjacency List: an array of list



Adjacency Matrix: 2D Array



A[i,j] = 1 iff $(i,j) \in E$

	1	2	3
1	0	1	1
2	0	0	1
3	0	0	0

→ usually no self loops, can put whatever on the diagnal

Space adj. list $O(n+m)$
adj. matrix $O(n^2)$

$m = [0, n^2]$

$m = O(n)$ sparse graph (adj. list better)
$m = \Omega(n^2)$ dense graph (adj. list/matrix the same)

Operation Run Time

adj list

- traverse neighbors given vertex u. I want to iterate all its neighbors | O(# neighbors)
  $O(n)$ for adj. matrix

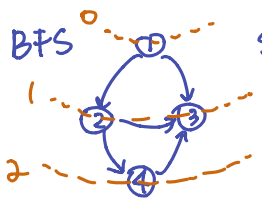
adj matrix

- $(u, v) \in E$ $O(1)$

Graph Traversal

- BFS Breadth First Search
- DFS Depth First Search

BFS



Starting point

[1], [3, 2], [4]

distance to neighbors

G, queue, visited[1....n]
u is starting point

Run time = $O(m)$ {
O(# neighbors of ①) +
O(# neighbors of ②) +
⋮
O(# neighbors of n)
} = # of edges

$O(m)$

```
BFS (G, u) {
    queue q;
    visited[1 ····n]; // initialize to false
    q.enqueue(u);
    visited[u] = T;
    while (q.empty() == F) {
        v = q.dequeue();
        process(v);
        for (w in G.neighbor(v)) {
            if (visited[w] == F) {
                q.enqueue(w);
                visited[w] = T;
            }
        }
    }
}
```

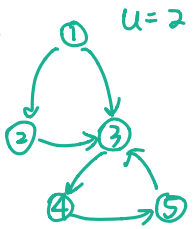
Distance $u \rightsquigarrow$ everyone → Shortest Distance
→ Shortest Paths

dist $u \rightsquigarrow i$ — dist[i] = $u \rightsquigarrow i$

π (parents) — $\pi[i]$ = second last vertex on shortest path

```
BFS (G, u) {
    queue q;
    visited[1 ···· n]; // initialize to false
    q.enqueue(u); dist[u] = 0
    visited[u] = T;
    while (q.empty() == F) {
        v = q.dequeue();
        process(v);
        for (w in G.neighbor(v)) {
            if (visited[w] == F) {
                q.enqueue(w); dist[w] = dist[v] + 1;
                visited[w] = T; π[w] = v;
            }
        }
    }
}
```

Example



② ⇝ ⑤ shortest distance : 3

shortest path: ② → ③ → ④ → ⑤

```
BFS_ALL(G) {
    visited[1 ···· n]
    for (u = 1 ···· n) {        → O(n)
        if (visited[u] == F) {
            BFS (G, u, visited[]);   → O(m)
        }
    }
}
```

O(n+m)

DFS — O(m) same as BFS

```
DFS (G, u, visited[]) {
    visited[u] = True
    process(u); // print(u);
    for (v in G.neighbors(u)) {
        if (visited[v] == False) {
            dfs (G, v, visited[]);
        }
    }
}
```

can't find shortest distance/path

```
DFS_ALL(G) {
    visited[1 ···· n]
    for (u = 1 ···· n) {
        if (visited[u] == False) {
            DFS (G, u, visited[]);
        }
    }
}
```

O(n+m)

	Run time	Space
BFS (G, u)	O(m)	O(n) queue (worst case)
BFS_ALL(G)	O(n+m)	
DFS (G, u)	O(m)	O(n) visited (worst case)
DFS_ALL(G)	O(n+m)	

Cycle Detection — DAG Directed Acyclic Graph

e.g. G

visited { path not finished / path finished

if we only check if visited[u] == True, the first case would work, second case won't (③ is visited when ② reaches 3)

color[1 ···· n] { WHITE not visited / GREY visited, not finished / BLACK visited, finished

G1 G2

color: W W W → G1 → G1 → G1

color: W W W → G1 G2 B → B B

```
DFS (G, u, color[]) {
    color[u] = GREY; // has_cycle = False
    for (v in G.neighbors(u)) {
        if (color[v] == WHITE) { has_cycle || DFS(G, v, color[]) }
        else if (color[v] == GREY) { cycle detected; return True } // has_cycle = True
        else if (color[v] == BLACK) { do nothing; }
    }
    color[u] = BLACK;
    return has_cycle
}
```

Connected Components in undirected graph → # of islands


```
BFS(G, u, cc_num){
    queue q;
    q.en(u); visited[u] = T; cc[u] = cc_num;
    while (q.empty() == F){
        v = q.de();
        for(w in G.neighbors(v)){
            if (visited[w] == F){
                q.en(w);
                visited[w] = T;
                CC[w] = cc_num;
            }
        }
    }
}
```

```
BFS_ALL(G){
    cc_num = 1;
    for(u=1 ... n){
        if(visited[u] == F){
            BFS(G, u, visited, cc_num);
            CC_num += 1
        }
    }
}
```

Summary

- Graph Traversal
  - BFS
    - iterative
      - BFS(G, s)
      - BFS_ALL(G)
    - shortest path
  - DFS
    - recursive
      - DFS(G, s)
      - DFS_ALL(G, s)
    - can find a path (not shortest)

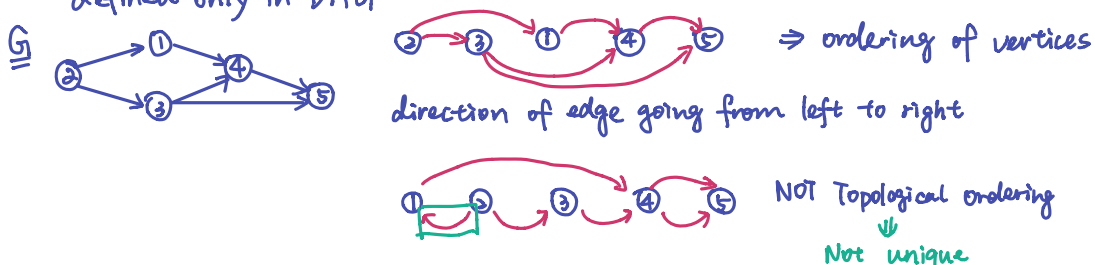
iterate through neighbors
adj. list

Application

① Cycle detection [using DFS_ALL(G)]
DAG: directed acyclic graph
② Connected Component [undirected graph] [Either Bfs or DFS]

Topological sort

- defined only in DAG



DAG ⇒ a topological ordering

- PDP → Algo → System
- make / cmake / other build sys

Two Algo
- #1 DFS based
  - hard to understand
  - easy to write
- #2 BFS based
  - "more" to write
  - easy to understand

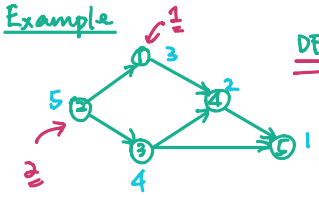
DFS based

```
DFS(G, S){
    visited[S] = T;
    for(u in G.neighbors(S)){
        if(visited[u] == F){ DFS(G, u); }
    }
    f[S] = time; time += 1;
}
```

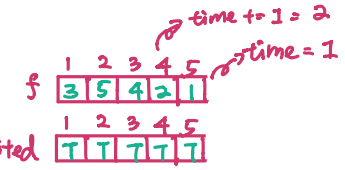
global time stamp that keeps track of the finishing time of each vertex

finish time f[i]

Example



DFS-ALL



Order vertices by finish time in descending order would give us topological ordering of G.

Q1 All # of topological orderings — Exponential time, and often not necessary

Q2 Make sure G is a DAG first? Yes

Running time O(n+m)

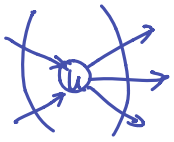
```
DFS(G, S, visited[], soln){
    visited[S] = T;
    for(u in G.neighbors(S)){
        if(visited[u] == F){ DFS(G, u, visited, soln); }
        soln.append(S);
    }
}
topological_sort(G){
    soln = []
    for(S in V){
        if(visited[S] == F){ DFS(G, S, visited, soln); }
    }
    reverse(soln);
}
```

BFS based

Can also detect cycles

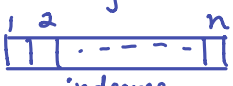
- iterative
  - select indegree-0 vertices
  - remove them from G

indegree should be 0



in degree out degree

BUT
- remove vertex from G is expensive
- adj. list represents out-degrees only (we don't actually remove the edge, simply simulate it)

- G is adj. list
- 

- 

① compute initial indegree:

```
for(u in V){
    for(w in G.neighbors(w)){
        indegree[w] += 1
    }
}
```

O(n+m)
O(n²) - matrix

② select vertices with indegree == 0:

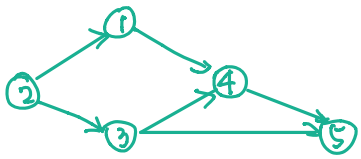
```
for(u in V){
    if(indegree[u] == 0){ q.push(u); }
}
```

O(n)
O(n²) - matrix

③

```
while(q.empty() == F){
    u = q.pop();
    soln.append(u);
    for(w in G.neighbors(w)){
        indegree[w] -= 1;
        if(indegree[w] == 0){ q.push(w); }
    }
}
```

O(n+m)
O(n²) - matrix



1	2	3	4	5
0	0	0	2	2

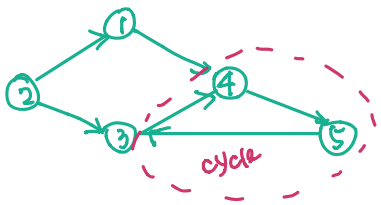
0 0

indegree

queue: ~~2~~ 1 ~~3~~ ~~4~~ 5

soln = [2, 1, 3, 4, 5]

if len(soln) < n, means not all vertices can be added in solution, i.e. there is a cycle

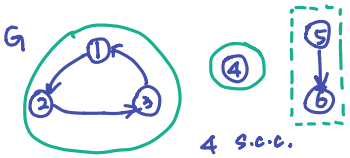


1	2	3	4	5
0	0	1	1	1

indegree → end result

queue: 2 1

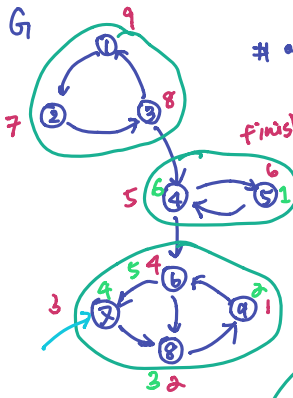
Strongly Connected Components (S.C.C) -- Informational



4 S.C.C.
3 W.C.C

u, v in the same **strongly** connected components iff u ⇝ v **and** v ⇝ u

u, v in the same **weakly** connected components iff u ⇝ v **or** v ⇝ u

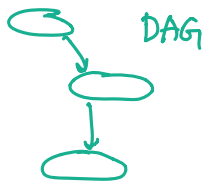


# of S.C.C?

① DFS(G, 7) start from — can discover [6, 7, 8, 9]

② DFS(G, 5) [4, 5, ~~6, 7, 8, 9~~] marked as visited, can't be reached

③ DFS(G, 2) [1, 2, 3, ~~4, 5, 6, 7, 8, 9~~] marked as visited, can't be reached

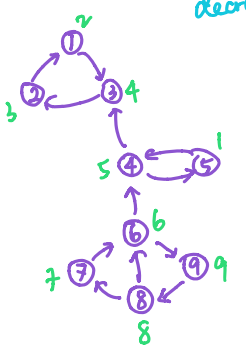


S.C.C. form a DAG

- DFS_ALL w/ finish time on G → G^T (inverse graph)
- DFS_ALL in G using increasing order of finish time (decreasing)

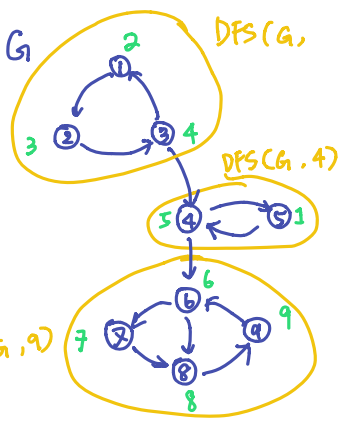
① DFS(G, 5)
DFS(G, 3)
} DFS_ALL

② DFS(G, 9) → finish time = 1
[6, 7, 8, 9]
DFS(G, 4) → finish time = 5
[4, 5]
DFS(G, 2) → finish time = 7
[1, 2, 3]



① DFS_ALL(G^T, 4)
DFS_ALL(G^T, 9)

② DFS(G, 9)
[6, 7, 8, 9]
DFS(G, 4)
[4, 5]
DFS(G, 3)
[2, 3, 4]



Not perfect, there's flaw as below:

① DFS(G, 4)

② DFS(G, 5)
[4, 5, 6, 7, 8, 9] ⇝ Should be 2 components instead of 1

Word Ladders

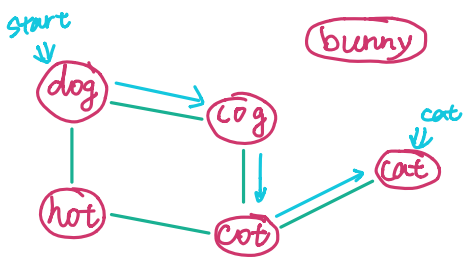
- dictionary (dog, cat, bunny, ···)
- start word "dog"
- destination word "cat"

dog ⇒ cog ⇒ cot ⇒ cat

① is it possible?

② can you find a soln?

③ can you find the "best" soln?



G undirected graph

puzzle-8

1	3	
7	2	5
8	6	4

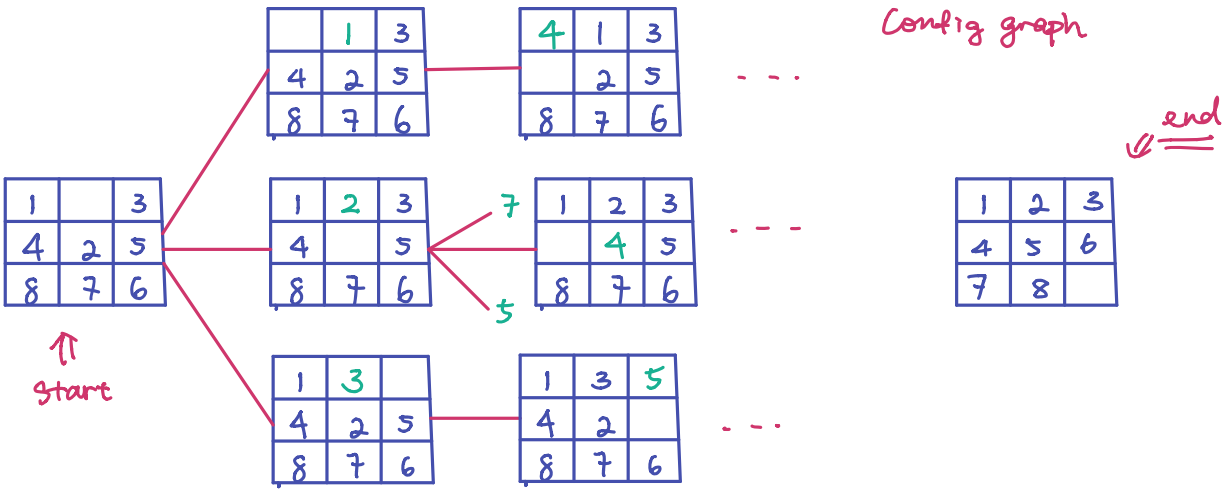
⇒

1	2	3
4	5	6
7	8	

allowed moves

- move number into adj. space

Q: ① Is it possible?
② can you find a seq of move that's possible?
③ shortest move?



n = # vertices = 9! << 1 million
m = # edges ~ 3×n

How to represent a puzzle?

`int [][]` ⇒

`visited` Hashtable

`103425876` ⇒ a string

1		3
4	2	5
8	7	6

Single Source Shortest Path (SSSP)

- input G = (V, E)
  ∀e ∈ E, there is a cost/distance cost[e]
  Source vertex s
- output shortest distance from s to everyone else

dist [1 ... n] |V| = n

dist[u] is shortest distance from s ⇝ u

SSSP 3 Algorithms
① Is G a DAG?
② Is there negative weighted edge?
∃e s.t. cost[e] < 0

Decision Matrix

Question 2 \ Question 1	Yes	No
Yes	Shortest path on DAG O(n+m)	Bellman Ford O(nm)
No		Dijkstra's Algo O(n²) / O(mlgn)

#1 Algo Shortest Path on DAG

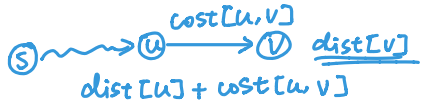
dist [1 ... n]

Topological_sort O(n+m)

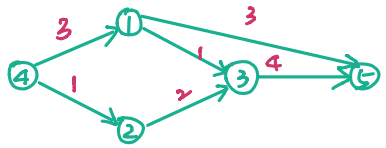
```
for(u in topological_sort(G)){
    for(v in G.neighbors(u)){
        relax(u, v);
    }
}
```

helper function: relax(u, v) { (u, v) ∈ E

```
if (dist[u] + cost[u, v] < dist[v]){
    dist[v] = dist[u] + cost[u, v];
}
}
```



e.g.



source is ④

topological sort [4, 2, 1, 3, 5]

dist | ∞ | ∞ | ∞ | 0 | ∞ | initial
1 2 3 4 5

dist | 3 | 1 | ∞ | 0 | ∞ | u = 4
1 2 3 4 5

④ 3→ ①
⑤ ⇝ ④ dist[1]
0 + 3 < ∞

④ 1→ ②
⑤ ⇝ ④ dist[2]
0 + 1 < ∞

dist | 3 | 1 | 3 | 0 | ∞ | u = 2
1 2 3 4 5

② 2→ ③
⑤ ⇝ ② dist[3]
1 + 2 < ∞

dist | 3 | 1 | 3 | 0 | 6 | u = 1
1 2 3 4 5

① 1→ ③
⑤ ⇝ ① dist[3]
3 + 1 > 3 don't update

① 3→ ⑤
⑤ ⇝ ① dist[5]
3 + 3 < ∞

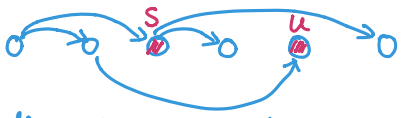
dist | 3 | 1 | 3 | 0 | 6 | u = 3
1 2 3 4 5

③ 4→ ⑤
⑤ ⇝ ③ dist[5]
3 + 4 > 6 don't update

dist | 3 | 1 | 3 | 0 | 6 | u = 5
1 2 3 4 5

Intuition

G is a DAG



dist[u] only depends on vertices to the left of u

invariance dist[u] is u's true shortest distance

dist[v] = dist[u] + cost[u, v]
⑤ ⇝ ⓤ ⓤ → ⓥ

Dijkstra's Algo
- G = (V, E) may have cycles
- weights of edges are positive

Iterative Algo

	1		s		n
dist	∞	····	0		∞

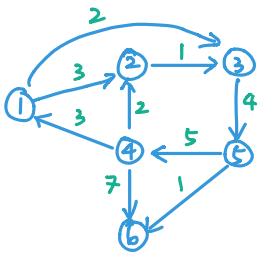
N = { } set of vertices we pick over time

while (N ≠ V) {
① pick u s.t. u has smallest dist[·] value among vertices in V \ N O(n)
② N = N ∪ {u} N[u] = T O(1) ≡ delete_min O(lgn)
③ for (v in G.neighbors(u)) { relax(u, v); } O(outdegree) ⇝ dist[v] ↓ bubble up O(lgn) × outdegree
O(lgn × outdegree)

heap → O(1)
O(n) } O(n²)
for all vertices O(mlgn)

? How to find v in pq?
map id ⇒ pos in array
v v in physical view

G



source is ④

dist | 1 | 2 | 3 | 4 | 5 | 6 |
| ∞ | ∞ | ∞ | 0 | ∞ | ∞ |

dist | 1 | 2 | 3 | x 4 | 5 | 6 |
| 3 | 2 | ∞ | 0 | ∞ | 7 |
N = {4}
u = 4
④ 3→ ①
④ 2→ ②
④ 7→ ⑥

dist | 1 | 2 | 3 | x 4 | 5 | 6 |
| 3 | 2 | 3 | 0 | ∞ | 7 |
N = {4, 2}
u = 2
② 1→ ③
③ ⇝ ② + cost[2, 3]
2 + 1 < ∞

dist | 1 | x 2 | 3 | x 4 | 5 | 6 |
| 3 | 2 | 3 | 0 | ∞ | 7 |
N = {4, 2, 1}
u = 1
① 3→ ② 3 + 3 > 2
① 2→ ③ 3 + 2 > 3 don't update

	1	2	3	4	5	6
dist	3	2	3	0	7	7

$N = \{4, 2, 1, 3\}$, $u = 3$

③ $\xrightarrow{4}$ ⑤ : $3 + 4 < \infty$

	1	2	3	4	5	6
dist	3	2	3	0	7	7

$N = \{4, 2, 1, 3, 5\}$, $u = 5$

⑤ $\xrightarrow{3}$ ④ : $3 > 0$

⑤ $\xrightarrow{1}$ ⑥ : $7 + 1 > 7$ don't update

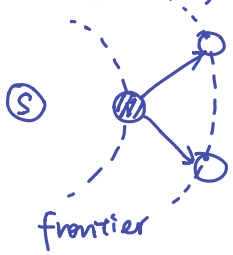
	1	2	3	4	5	6
dist	3	2	3	0	7	7

$N = \{4, 2, 1, 3, 5, 6\}$, $u = 6$

BFS: all edges have the same weight

Dijkstra: $N = \{ , \dots \}$

$N = N \cup \{u\}$ when pick vertex u, we know the true shortest dist[u]



Bellman-Ford

- most general SSSP Algo
- cycles, negatively weighted edges

```
BF(G){
  for(i = 1 ..... n-1){
    for(u = 1 ..... n){
      for(v = G.neighbors(u)){
        relax(u, v);
      }
    }
  }
}
```

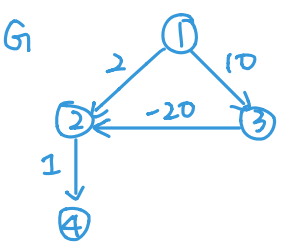
O(n+m) inside, O(nm) overall

⑤ ⇝ ⓤ: length path

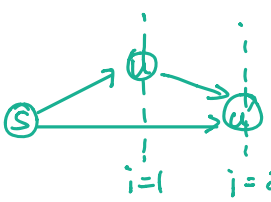
edges

i = k: ⑤ ⇝ ⓤ # edges ≤ k

after k^{th}, dist[u] has the true shortest distance



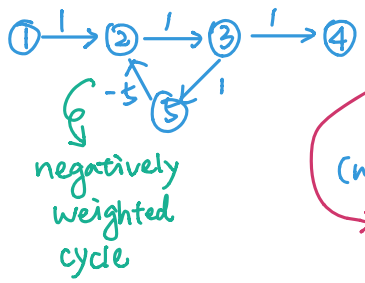
	1	2	3	4	
dist	0	-10	10	3	i = 1 [1, 2, 3, 4]
dist	0	-10	10	-9	i = 2 [1, 2, 3, 4]
dist	0	-10	10	-9	i = 3 [1, 2, 3, 4]



max # of edges = n-1

① → ② → ③ → ... → ⓝ

n-1 edges



① ⇝ ④ shortest dist = ?

has cycle: ① → ② → ③ → ⑤ → ② → ③ → ④ dist[4] = 0

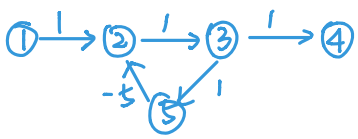
simple path: ① → ② → ③ → ④ dist[4] = 3

(no cycle allowed)

(if limit shortest paths to simple path, it's a NP problem) while having negatively weighted cycles

but it would 6 edges to go from ① → ④, while our algorithm only goes at most n-1 = 4 edges.

So how to detect negatively weighted cycles?



source is ①
Bellman - Ford
$i = 1 \cdots 4$

```
for (i=1 ---- n-1) {
    for (u=1 ---- n) {
        for (v in G.neighbors(u)) {
            relax(u, v);
        }
    }
}
for ((u,v) ∈ E) {
    relax(u,v)
}
```

equivalent to

```
for ((u,v) ∈ E) {
    relax(u,v)
}
```

Do one more round to detect negative cycles

① "good" graph, no update to dist[]

② "bad" graph, there is update to dist[]

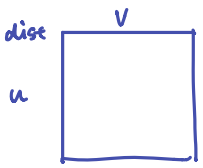
- SSSP [review]

DAG	V	N
neg weight Y	SP on DAG	Bellman-Ford $O(nm)$ detect → negatively weighted cycle
N		Pijkstra's Algo $O(n^2)$ --- dense graph $O(m \lg n)$ --- sparse graph uses heap

- APSP [All-pair shortest path]

(1) $G = (V, E)$ $e \in E$ cost[e]

(2) Output

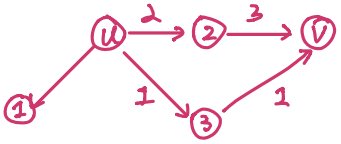


dist[u, v] = ⓤ ⇝ ⓥ

① Run SSSP n time (Except Bellman-Ford, which would yield $O(n^2 m)$) ↘ sparse : $O(n^3)$; Dense : $O(n^4)$

② Floyd-Warshall $G = (V, E)$ $V = \{1, 2, 3, \cdots, n\}$

$f^k(u, v)$ = shortest distance from u to v [only allow to use $\{1, 2, \cdots k\}$ on any path]

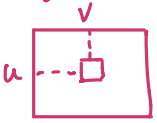


$f^2(u,v) = 5$ can only use $\{1, 2\}$

$f^3(u,v) = 2$ can only use $\{1, 2, 3\}$

$f^0(u,v) = \begin{cases} cost[u,v] & \text{if } (u,v) \in E \\ \infty & \text{o/w} \end{cases}$

→ adj matrix



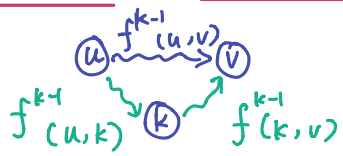
$f^0(u,v) \to f^1(u,v) \to f^2(u,v) \to \cdots \to f^n(u,v)$

$f^0(u,v)$: adj matrix; $f^n(u,v)$: true short distance

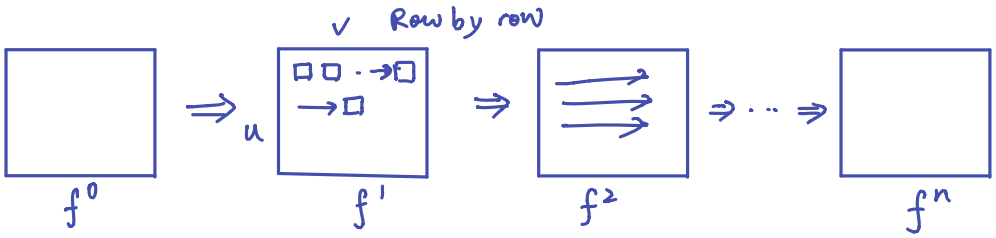
$f^0(u,v) \cdots f^{k-1}(u,v)$ $\quad$ $f^k(u,v)$

① Don't use k

② Use k

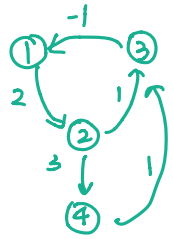


$$f^k(u,v) = \min\{f^{k-1}(u,v),\ f^{k-1}(u,k) + f^{k-1}(k,v)\}$$




```
FloydWarshall(G){
    for(k=1.....n){
        for(u=1.....n){
            for(v=1.....n){
                A[u,v] = min(A[u,v], A[u,k] + A[k,v])
            }
        }
    }
    return A;
}
```

e.g.



Allowed to use ① Allowed to use ②

	1	2	3	4
1	0	2	∞	∞
2	∞	0	1	3
3	-1	∞	0	∞
4	∞	∞	1	0

f^0 ⇒

	1	2	3	4
1	0	2	∞	∞
2	∞	0	1	3
3	-1	1	0	∞
4	∞	∞	1	0

f^1 ⇒

	1	2	3	4
1	0	2	3	5
2	∞	0	1	3
3	-1	1	0	∞
4	∞	∞	1	0

f^2

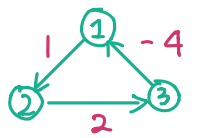
$f^1(3,2) = f^0(3,1) + f^0(1,2)$

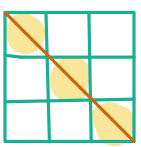
$$f^k(u,v) = \min\{ f^{k-1}(u,v),\ f^{k-1}(u,k) + f^{k-1}(k,v) \}$$

$f^{k-1}(u,k) = f^k(u,k)$

Can F-W detect negative cycle?

Yes





	1	2	3
1	0	1	∞
2	∞	0	2
3	-4	∞	0

f^0 ⇒

	1	2	3
1	0	1	∞
2	∞	0	2
3	-4	-3	0

f^1 ⇒

	1	2	3
1	0	1	3
2	∞	0	2
3	-4	-3	-1

f^2 ⇒

	1	2	3
1	-1	0	2
2	-2	-1	1
3	-4	-3	-2

f^3

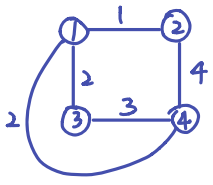
If there is negative cycle,
the value on the diagnal will be negative

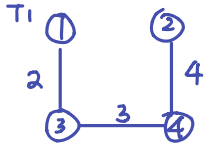
MST : Minimum Spanning Tree

connected undirected graph

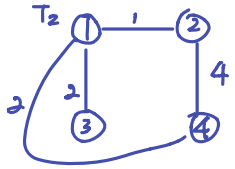
$G_1 = (V, E)$

Spanning Tree $T = (V, E')$ $E' \subseteq E$





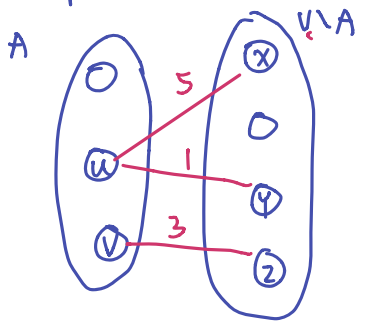
cost(T_1) = 9



cost(T_2) = 5

- Prim's algo [very similar to Dijkstra]
- Kruskal's algo [Disjoint Set, Union & Find]

cut partition of vertice $G = (V, E)$



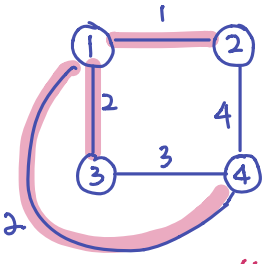
e with the cut cross the cut

3 edges crossing this cut

Observation: $(u,v), (u,y), (v,z)$ at least one of them belongs to MST

claim: the one with minimum weight is safe to be picked as part of MST

Prim's



A

$A = \{2\}$ $V\setminus A = \{1,3,4\}$

$A = \{1,2\}$ $V\setminus A = \{3,4\}$

$A = \{1,2,3\}$ $V\setminus A = \{4\}$

$A = \{1,2,3,4\}$ $V\setminus A = \emptyset$

min

$(1,2)=1$, $(2,4)=4$

$(1,3)=2$, $(1,4)=2$, $(2,4)=4$

$(1,4)=2$, $(3,4)=3$ $(2,4)=4$

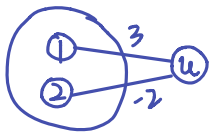
V/A

$A = \{1\}$

dist

distance to set A

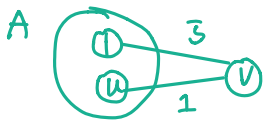
A



dist[u] = -2

```
while(A ≠ V){
    ① pick u ∈ V\A s.t. u has smallest dist[] value
    ② A = A ∪ {u}
    ③ for(v in G.neighbors(u)){
        dist[v] = min(dist[v], cost[u,v])
    }
}
```

A

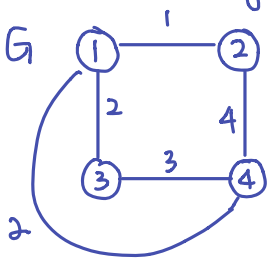


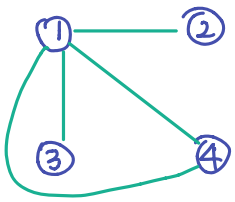
$O(n^2)$ → for loop

$O(m \lg n)$ → heap

Kruskal's Algo

G



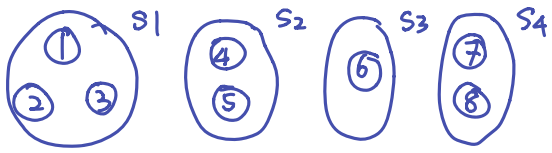


sort $(1,2)^{\checkmark}$ $(1,3)^{\checkmark}$ $(1,4)^{\checkmark}$ $(3,4)^{\times}$ $(2,4)^{\times}$

1 2 2 3 4

has cycle has cycle

Disjoint Sets n items



Find item

Union(S1, S3)

```
T = { }
for(e is sort(E)){
    if(T ∪ {e} doesn't introduce cycle)   → O(m) or O(n)
        T = T ∪ {e}
}
return T;
```

m

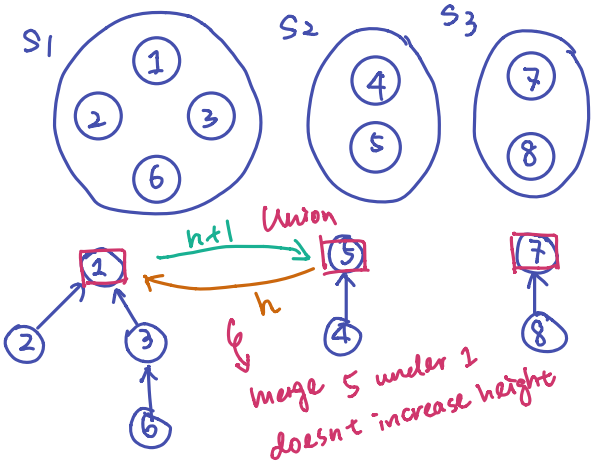
$O(m^2)$ or $O(mn)$

Worse than Prim's

```
for((u,v) is sort(E)){
    Su = Find(u);          O(lg n)
    Sv = Find(v);
    if(Su ≠ Sv){           O(1)   → If u and v are not in the same set
        T = T ∪ {(u,v)}; Union(Su, Sv);
    }                      every set is C.C in spanning tree
}
return T;
```

$O(m \lg n)$

Logical View



Physical View

If has no parent
parent = itself

	1	2	3	4	5	6	7	8
parents	1	1	1	5	5	3	7	7

```
Find (u) {
    if ( parents[u] == u){
        return u;
    }
    return Find (parents[u]);
}
```

O(h) → O(n)
height worst

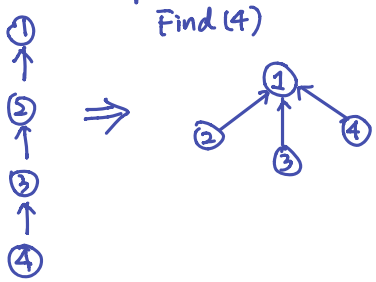
u and v are both roots

```
Union ( u, v){
    if ( r[u] < r[v]) {
        parents [u] = v ;
    } else if (r[u] > r[v]) {
        parents [v] = u ;
    } else {
        parents[v] = u;
        r[u] += 1;
    }
}
```

O(1)

Union by Rank
height = O(logn)
# item ≥ 2^{rank}

Path Compression




```
Find (u) {
    if ( parents[u] == u){
        return u;
    }
    parents[u] = Find (parents[u]);
    return parents[u];
}
```

Before the path is compressed, worth case O(lgn);
After the path is compressed, average case O(1)

Amortized analysis
O(log* n) ~ O(1)

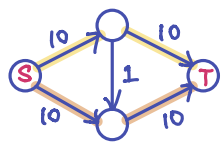
log* 2 = 1
log* 4 = 2
log* 16 = 3
log* 65536 = 4
log* 2^{65536} = 5

# of log ops performed until res = 1

# of Atoms in the Universe 10^{80}

Max Flow

- directed graph G = (V, E)
- e ∈ E, giving capacity Ce



oil pipe
- source s
- sink t
- Q: max oil flow s ⇝ t

fmax = 20

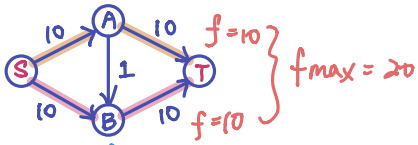
```
while (∃ path s⇝t) {
    - find path pi from s⇝t
    - figure how much flow fi can be sent on path pi
    - fmax += fi
}
return fmax;
```

→ Flawed
Might only find sub-optimal graph

← decrease cap by fi on path pi

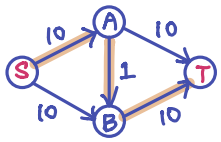
- Residual Network
- Augment Path

Integrated

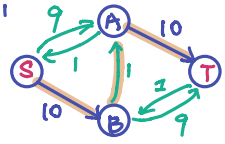


1) $S \to A \to B \to T$, $f_1 = 1$
2) $S \to B \to A \to T$, $f_2 = 1$
3) $S \to A \to T$, $f_3 = 9$
4) $S \to B \to T$, $f_4 = 9$
5) No path from $S \rightsquigarrow T$

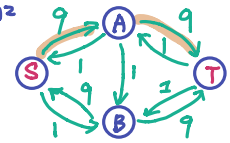
G_0



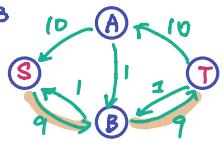
G_1



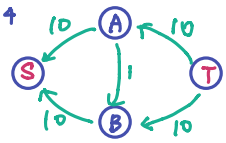
G_2



G_3



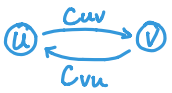
G_4



$G_1 \to G'$

pick augment path $S \rightsquigarrow t$

$(u, v) \in P$

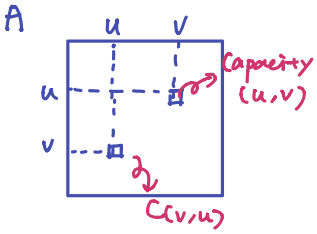


Cuv -= f
Cvu += f

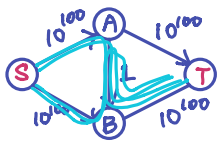
Fold - Fulkerson

while (∃ augmenting path $s \rightsquigarrow t$) {
① find path Pi
② figure out flow fi
③ update residual network
}
$f_{max} = \Sigma f_i$

$O(|f_{max}| m)$

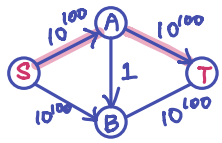


DFS vs BFS ☆ # of iteration



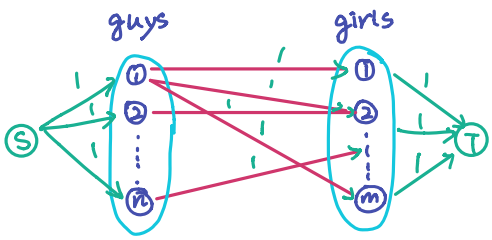
will run through the bad path 10^{100} times, bad!

$O(mn) \Longrightarrow O(m^2 n)$ overall



BFS would choose

Online dating application



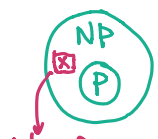
maximize # of matches

Maximal matching

Claim: max flow f_{max} = maximal matching

P vs NP

P is a set of problems can solved in polynormial time
NP is a set of problems whose solutions can be verified in polynomial time



identify one problem

$P \subseteq NP$ for sure

$P = NP$ (A) or $P \subset NP$ (B) ??

⇓ try to find hardest problem in NP

⇓ if you can solve x in polynomial time, then you can solve all NP problems in poly

~1970/1980 "hardest" problems in NP — NP-complete

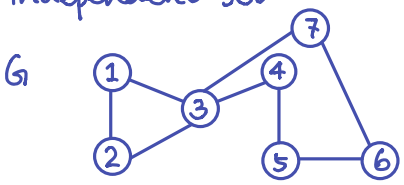
SAT
(Satisfiable Assignment Test)
boolean formula
variable $x_1, x_2, \cdots, x_3$
literals $x_1, \neg x_2$
clause $x_1 \vee \neg x_2 \vee x_4$

$F = (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4) \wedge (\cdots)$

3SAT

3D matching
NP-complete

independent set



{①, ④} independent

{⑤, ⑤} independent

{①, ④, ⑥} most vertices

How to find an independent set with most vertices? NP-complete

Efficient ?

$O(n \lg n)$ ✓
$O(n^2)$ ✓
$O(n^3)$ ✓
$O(n^4)$ ✓
⋮ ⋮
$O(n^{1.6})$ ✓

$O(2^n)$ ×
$O(n!)$ ×

A, B compare which is "harder" ?

$A \leq_p B$ reduction

⇑ polinomial time reduction

B Algo$_B$

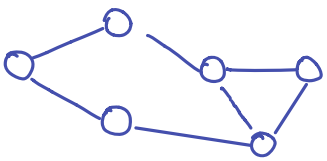
Input A → Input B → Algo$_B$ → Output$_B$ → Output A

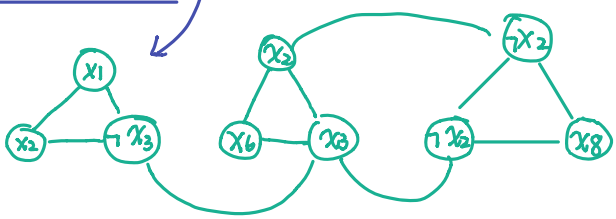
All NP problems $\leq_p$ SAT $\leq_p$ 3SAT $\leq_p$ Independent Set $\leq_p$ Travelling salesman

$(x_1 \vee x_2 \vee \neg x_3) \wedge (\cdots) \wedge \cdots \wedge (\quad)$ assume efficient algorithm

satisfiable assignment

$x_1 \quad x_2 \quad \cdots \quad x_n$





if ∃ independent set of size m, (m = # of clauses in the formula)
then formula is satisfiable

Independent set $\{\neg x_3, x_2, \cdots, x_8\}$

m vertices ⇒ assigned as true

knapsack
subset

$O(nC)$
$O(nT)$

pseudo polynomial

polynomial in terms of input size

NP complete