Jun 17

Dynamic Programming

① Question type: Optimization / Counting / Feasibility
② Design Recursion
③ Naive recursive implement yields bad running time (usually exponentially) due to recompute the same thing
④ Bottom up [small → large]
Top Down (with memorization)

- LCS (Longest Common Subsequence)

Input A[1, ..., n]
B[1, ..., m]

eg A = [1,5,2,3,4]
B = [7,2,5,4,3]
CS: [2,3], [2,4] [5,4] [2] ...

$f(i,j)$: length of the LCS of given A[1:i] and B[1:j]

case 1: $A[i] = B[j]$
$f(i,j) = f(i-1, j-1) + 1$

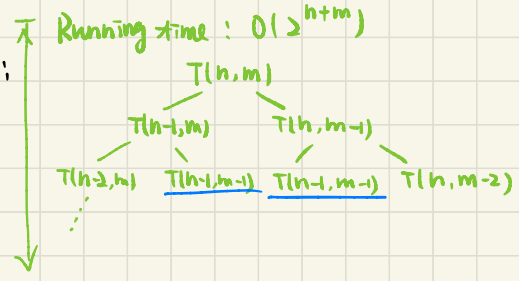
case 2: $A[i] \neq B[j]$
$f(i,j) = \max\{f(i-1,j), f(i,j-1)\}$

Base Case: $f(i,0) = f(0,j) = 0$

A [7,1,3] B [1,2,3]
+1
[7,1] [1,2]
[7] [1,2] [7,1] [1]
×
+1
[7] [ ]

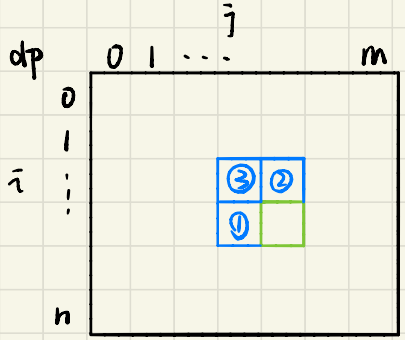
```
LCS_Slow(A[], B[], i, j){
    if(i==0 || j==0) return 0; // Base case
    if(A[i] == B[j])
        return LCS_Slow(A, B, i-1, j-1) + 1;
    return max{LCS_Slow(A, B, i-1, j), LCS_Slow(A, B, i, j-1)};
```

Driver Method
LCS_Slow(A[], B[], A.length, B.length);



Bottom Up

```
LCS(A[1..n], B[1,...m]){
    for(i=0,...,n) dp[i,0] = 0
    for(j=0,...m) dp[0,j]=0
    for(i=1,...n){
        for(j=1,...,m){
            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];
```



$$f[i,j] = \begin{cases} f[i-1,j] & ① \\ f[i,j-1] & ② \\ f[i-1,j-1] & ③ \end{cases}$$

TC: O(n·m)
SC: O(n·m)
can be reduced to O(n)
See next page

Top Down

```
LCS(A[], B[], i, j, dp[][]){   dp filled with -1
    if(i==0 || j==0) return 0;
    if(dp[i,j] != -1) return dp[i,j];
    if(A[i]==B[j])
        dp[i,j] = 1+LCS(A,B,i-1,j-1,dp);
    else
        dp[i,j] = max{LCS(A,B,i-1,j,dp), LCS(A,B,i,j-1,dp)}
    return dp[i,j];
}
```

TC: O(nm)
For each grid in the dp array, we only compute

i: A=[7,1,3] j: B[1,2,3]

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

? Given the dp[i][j], construct the actual sub sequence

#1092

```
construct(A[], B[], dp[][]){
    res[]; i=n, j=m;
    while(i>0 && j>0)
        if(A[i]==B[j])
            res.append(A[i]); i--; j--;
        elif(dp[i-1,j] < dp[i,j-1])
            j--;
        else
            i--;
    return res.reverse();
}
```

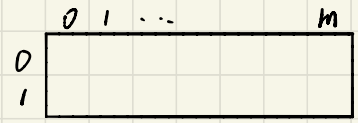
O(n+m)

eg

l=[3,1]
i=3, j=3 ✓ append ③
i=2, j=2 ×
i=2, j=1 ✓ append ①
i=2, j=0 × terminate

Reduce Space Complexity

```
for(i=1,...,n)
  for(j=1,...,m)
    if(A[i]==B[j])
        dp[i%2,j] = 1+dp[(i-1)%2,j-1]
    else
        dp[i%2,j] = max{ dp[(i-1)%2,j],
                         dp[i%2,j-1] }
return dp[n%2,m]
```



Space Complexity Stack + Heap

- Knapsack Problem (0-1 version, either take or not) (Other version may allow take a fraction of the item) ⇒ Greedy

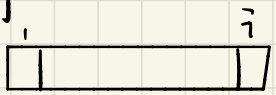
Given n objects and a knapsack of capacity W. Item i has weight w_i, and value v_i.

Goal: fill knapsack so as to maximize total value. max$

Greedy solution:

Repeatedly add item with maximum $\frac{v_i}{w_i}$ ratio. NOT OPTIMAL

$f(i,j)$ = max $ given $A[1,\cdots,i]$, capacity is j



If you take ith item:

value: $v[i] + f(i-1, j-w[i])$ ①

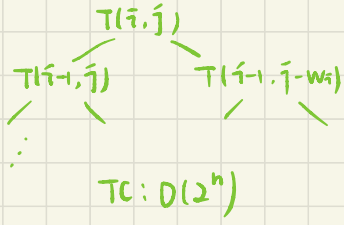
NOT take

value: $f(i-1, j)$ ②

Recursion: Choose the max of ① ②

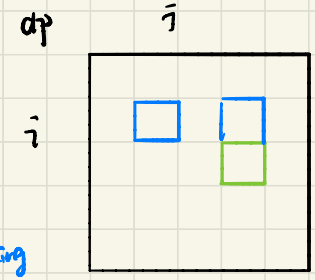
Base case: $f(0,j) = f(i,0) = 0$

no item to choose no capacity left



TC: $O(2^n)$

```
knapSack(v[], w[], C){
    for(i=1, ..., n)
        for(j=1, ..., C)
            if(w[i] ≤ j)
                dp[i,j] = max{dp[i-1, j],        Not taking
                             v[i] + dp[i-1, j-w[i]]}   taking
            else
                dp[i,j] = dp[i-1, j]
    return dp[n, C]
```



TC: O(n·C)

SC: O(n·C)

Reduce space complexity to O(C)

```
backpack(v[], w[], C){
    dp[C+1]
    for(i=1, ..., w.len)
        for(j=C ... w[i]) // dp array must be filled from backward, otherwise we may pick one item twice.
            dp[j] = max{dp[j], dp[j-w[i]] + v[i]};   // Not taking; Take the item
    return dp[C];
}
```

dp[3] = dp[0] + v_i, dp[6] = dp[3] + v_i

- Mining problem
  - n people
  - m mining opportunities. (v[i], # people) — each oppor requires certain people and worth v[i]

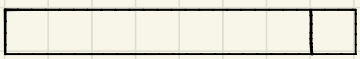
  Goal: maximize the profit

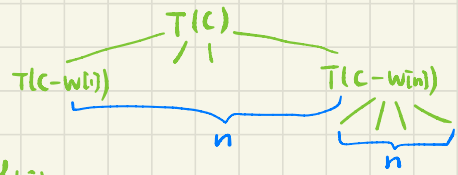
- Unbounded knapsack

$ weight

- Input: n types of items [for each type, there is inf copy], each item of type i $v[i]$, $w[i]$
  capacity of knapsack C

The only changing variable on the fly is the current capacity of knapsack
$f(i)$ = max $ give knapsack capacity i.



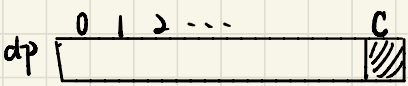


If you take the first item $v[1] + f(i - w[1])$
Second $v[2] + f(i - w[2])$
⋮ ⋮
nth $v[n] + f(i - w[n])$

$\max = f(i)$

TC: $O(n^n)$

Optimization



Pseudocode:

```
knapsack(v[1....n], w[1,....n], C){
    dp[0, ....C]
    for(i=1.... C)
        for(k=1, ...., n){
            if(w[k] < i){
f(i)        dp[i] = max{dp[i], v[k] + dp[i - w[k]]};
            }
        }
    }
    return dp[C];
}
```

max of n options

TC: $O(nC)$
SC: $O(C)$

Example

C=12. ($5, 2lb) ($1, 10lb) ($7, 3lb), ($4, 5lb)

	0	1	2	3	4	5	6	7	8	9	10	11
dp	0	0	5	5								

$i=3$, 1st item $5 fits. $5 + dp[3-2] = $5
2nd item $1 not fits
3rd $7 fits $7 + dp[3-3] = $7 ✓ > $5 ✗

Outer loop: loop through all possible capacity
Inner loop: loop through all the items

Coin Change

Dinomination $d_1, d_2, \cdots, d_k$

Goal: minimize the # coins used

$f(i)$ = amount of coins want to make changes

Set related to goal

$$f(i) = \min \begin{cases} f(i-d[k]) + 1 & \text{// try the last coin} \\ f(i-d[k-1]) + 1 \\ \vdots \\ f(i-d[1]) + 1 & \text{// try the first coin} \end{cases}$$

```
coinChange(d[1,...,k], n])          O(nk)   n: amount of coins
                                            k: number of domination
    dp[0,...,n]
    for(i=1,...,n)
        dp[i] = i; // worst case
        for(j=1,...,k){
            if(d[j] ≤ i){
                dp[i] = min{dp[i], 1+dp[i-d[j]]}
            }
        }
    return dp[n];
```

Knapsack with Constraints

- n items v[i], w[i], u[i] // volume of item Each item has only one copy
- W: weight capacity U: volume capacity
- Goal: maximize $

Different from Knapsack 1, this problem has three dimensions.

f(i, j, k) i: # of items first i items
j: remaining weight capacity
k: remaining volume capacity

For ith item:

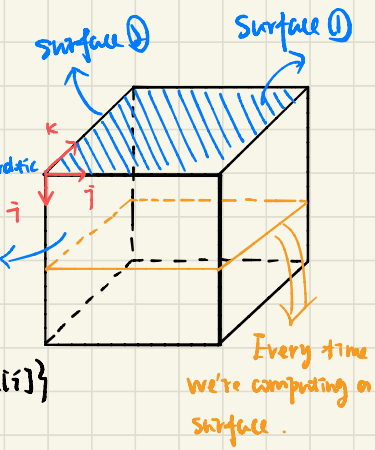
$$f(i,j,k) = \max \begin{cases} f(i-1, j, k) \\ f(i-1, j-w[i], k-u[i]) + v[i] \end{cases}$$

Base Case:

$f(0,j,k)$ ① $= f(i,0,k)$ ② $= f(i,j,0) = 0$

Base case: Three surface

```
for(i=1,...,n)   SC: O(nWU)   TC: O(nWU)
  for(j=1,...W)                can be reduced to quadratic
    for(k=1,...,U)
        dp[i,j,k] = dp[i-1,j,k];
        if(w[i]<=j && u[i]<=k)
            dp[i,j,k] = max{dp[i,j,k],
                            v[i]+dp[i-1,j-w[i],k-u[i]]}
return dp[n,W,U];
```



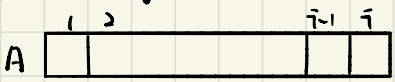
- Palindrome Problem

Given a string, return the least length of array of substring (palindrome)

banana : $b_1 = [b, ana, n, a]$, $|b_1| = 4$

$b_2 = [b, anana]$, $|b_2| = 2$

$f(i)$ = min # of palindromes given $A[1 \cdots i]$



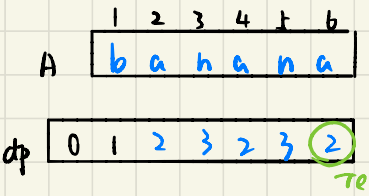
$f(i) =$

$f($"banana"$)$: remove the last palindrome substring $O(n^n)$

$$f(\text{"banana"}) = \min \begin{cases} f(\text{"bnanan"}) + 1^a \\ f(\text{"ban"}) + 1 \text{ ana} \\ f(\text{"b"}) + 1 \text{ anana} \end{cases} \xrightarrow{\text{Generative}} \min \begin{cases} f(i-1) + 1 & \text{if } A[i] \text{ is palind} \\ f(i-2) + 1 & \text{if } A[i-1:i] \text{ is } \cdots \\ \vdots \\ f(0) + 1 & \text{if } A[1:i] \text{ is palind} \end{cases}$$

Bottom up

```
minPalindrome(A[1,...n]){ O(n^3)
    dp[0,...,n];
    for(i=1,...,n){
        dp[i] = i;
        for(j=1,...,i)
            if(isPalidrome(A[j:i])) => O(n)
                dp[i] = min{dp[i], dp[j-1]+1}
    return dp[n];
```



$i=1$, "b" . $1 + dp[0]$

$i=2$, "ba" { "ba" ✗ ; "a" ✓ $1 + dp[1] = 2$.

$i=3$. "ban" { "ban" ✗ ; "an" ✗ ; "n" ✓ $1 + dp[2] = 3$.

$i=4$. "bana" { "bana" ✗ ; "ana" ✓ $1 + dp[1] = 2$ ✓ ; "na" ✗ ; "a" ✓ $1 + dp[3] = 4$

$i=5$ "banan" { "banan" ; "anan" ; "nan" ✓ $1 + dp[2] = 3$. ; "an" ; "n" ✓ $1 + dp[4] = 3$

$i=6$ "banana" { "anana" ✓ $1 + dp[1] = 2$ ✓ ; "ana" ✓ $1 + dp[3] = 4$; "a" ✓ $1 + dp[5] = 4$

Optimization : build dp array on the fly ,
use dp array to check if substring $A[i:j]$ is palindrome . (constant)
Reduce the total running time to $O(n^2)$

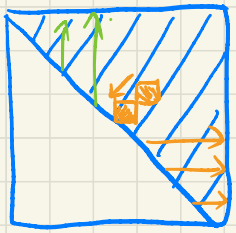
Improve to $O(n^2)$: reduce isPalindrome to $O(1)$

$g(i,j)$ = if $A[i:j]$ is palindrome

$g(i,j) = A[i] == A[j]$ && $g(i+1, j-1)$

Base case :

$g(i,i) = T$, $g(i,i+1) = A[i] == A[i+1]$



Pseudocode

```
for(i=1,...,n)  P[i,i] = T;
for(i=1,...n-1)  P[i,i+1] = A[i] == A[i+1];
for(col=3,...., n)
    for(row=1, col-2)
        P[row, col] = (A[row] == A[col])
                && P[row+1, col-1];
```

$O(n^2)$.

↙ dependency $(i+1, j-1)$

populate from last row or column by column ↓↓↓

#2

```
String longestPalindrome(String s){
    dp[s.len][s.len];
    res = "";
    for(j=1,...., len){
        for(i=1,...., i){
            dp[i][j] = S[i] == S[j] &&
                    (j-i<=2 || dp[i+1][j-1])
            if(dp[i][j] && res.len < j-i+1)
                res = S.substring(i, j+1);
        }
    }
    return res;
```

	1	2	3	4	5	6
A	b	a	n	a	n	a

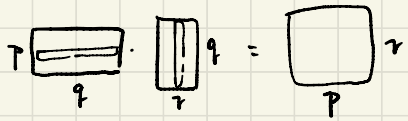
p (i ↓, j →)

i \ j	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

- Matrix Multiplication

$A_1 \times A_2 \times A_3 \cdots A_n$

$A_{p\times q} \cdot B_{q\times r}$ $O(p\cdot q\cdot r)$



p·r number each takes q

$A_1 \times A_2 \times A_3$
$2\times 100 \quad 100\times 5 \quad 5\times 3$

① $(A_1 \times A_2) \times A_3$ $2\times 100\times 5 = 1000$ $2\times 5\times 3 = 30$
$O(1030)$

② $A_1 \times (A_2 \times A_3)$ $2\times 100\times 3 = 600$ $100\times 5\times 3 = 1500$
$O(2100)$

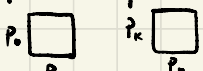
Calculate the min cost to do multiplication ?

$(A_1 \times A_2 \times A_3 \times \cdots A_i) \times (A_{i+1} \cdots A_n)$ — ith multiply

Subproblem | Subproblem

$f(\hat{i}, \hat{j})$ minimum cost to multiply $A_{\hat{i}} \times A_{\hat{i}+1} \times \cdots A_{\hat{j}}$

$f(1,n) = f(1,k) + f(k+1,n) + P_0 P_k P_n$



$$f(\bar{i},\bar{j}) = \min \begin{cases} f(\bar{i}+1,\bar{j}) + P_{\bar{i}-1} P_{\bar{i}} P_{\bar{j}} \\ f(\bar{i},\bar{i}+1) + f(\bar{i}+2,\bar{j}) + P_{\bar{i}-1} P_{\bar{i}+1} P_{\bar{j}} \\ \vdots \\ f(\bar{i},\bar{j}-1) + P_{\bar{i}-1} P_{\bar{j}-1} P_{\bar{j}} \end{cases}$$

Base case: $f(\bar{i},\bar{i}) = 0$

Example.

$A_1 \times A_2 \times A_3 \times A_4$
$P_0\cdot P_1 \quad P_1\cdot P_2 \quad P_2\cdot P_3 \quad P_3\cdot P_4$

choose the minimum

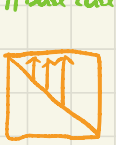
opt1. $A_1 \times (A_2 \times A_3 \times A_4)$
$P_0\cdot P_1 \quad f(2,4) \quad P_1\cdot P_4$
cost = $f(2,4) + P_0 P_1 P_4$

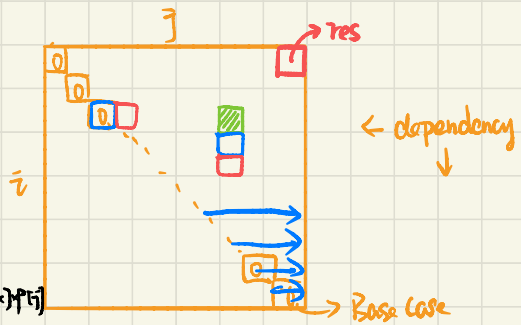
opt2 $(A_1 \times A_2) \times (A_3 \times A_4)$
$P_0\cdot P_2 \quad P_2\cdot P_4$
cost = $f(1,2) + f(3,4) + P_0 P_2 P_4$

opt3 $(A_1 \times A_2 \times A_3) \times A_4$

Pseudocode

```
M&M(P[0,...,n]){                                   O(n^3)
    for(i=1,...,n) dp[i,i] = 0;  // Base case
    for(j=2,...,n){
        for(i=j-1,...,1){
            dp[i][j] = max_value;
            for(k=i,...,j-1){
                cost = dp[i,k] + dp[k,j] + P[i-1]*P[k]*P[j]
                if(cost < dp[i,j])
                    dp[i,j] = cost;
            }
        }
    }
    return dp[1,n]
```





In order to compute ▨, we need to know all the element on the left of ▨, and all the element below ▨

Street of n house

Paint the house with Green, Red, Yellow.

No adjacent house can be painted with same color.

G[1···n]
R[1···n]
Y[1···n]

g(i) to color 1···i with ith colored as Green
r(i) to color 1···i · - · · - - Red
y(i) · · · Yellow

g(i) = G(i) + max{ r(i-1), y(i-1) };
r(i) = R(i) + max{ g(i-1), y(i-1) };
y(i) = Y(i) + max{ g(i-1), r(i-1) };

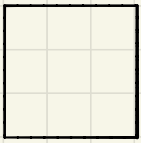
```
max$ (R[1···n], G[1···n], Y[1···n]){
    g[n], r[n], y[n];
    r[1] = R[1]; g[1] = G[1]; y[1] = Y[1];
    for(i = 2···n){
        g[i] = G[i] + max{ r[i-1], y[i-1] };
        r[i] = R[i] + max{ g[i-1], y[i-1] };
        y[i] = Y[i] + max{ g[i-1], r[i-1] };
    }
    return max{ g[n], r[n], y[n] };
}
```

Question:

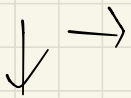
Given nxn grid filled with positive integers, find longest increasing path.

f[i,j] denotes the max path length to position [i,j]

$$f[i,j] = \max \begin{cases} 1 + f[i-1,j] & \text{if } (A[i][j] > A[i-1,j]) \\ 1 + f[i,j-1] & \text{if } (A[i][j] > A[i,j-1]) \\ 1 & \text{// Neither of above cases apply} \end{cases}$$



Pseudocode



```
dp[1,1] = 1 ; max_len = 0;
for( i = 2, ..., n )  // Filling first column
    if ( A[i,1] > A[i-1,1] ) dp[i,1] = dp[i-1,1] + 1
    else dp[i,1] = 1
for( j = 2, ..., m)  // Filling first row
    if ( A[1,j] > A[1,j-1] ) dp[1,j] = dp[1,j-1] + 1
    else dp[1,j] = 1
for( i = 2, ..., n ) {  // Filling the rest
    for( j = 2, ..., m) {
        if ( A[i,j] > A[i-1,j] )
            dp[i,j] = max{ dp[i,j], dp[i-1,j] + 1}
        if ( A[i,j] > A[i,j-1] )
            dp[i,j] = max{ dp[i,j], dp[i,j-1] + 1}
        max_len = max{ max_len, dp[i,j] };
    }
}
return max_len;
```

Reconstruct the sequence:

Starting from the dp[i,j] (res), decrement by 1 to get the previous element, also make sure that dp[i,j] > dp[i-1,j] or > dp[i,j-1].

Follow up.

Now you can go 4 directions in matrix, find the longest increasing path.

#1 $O(n^2)$

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

#2. Bottom up $O(n^2 \log n) \leq O(n^2 \log n + n^2) \leq O(n^2 \log n^2 + n^2)$ (sorting + filling dp)

3	5	1
2	1	6
2	4	7

1, 1, 2, 2, 3, 4, 5, 6, 7

(1,3) (2,3) (2,1) (3,1) ...

$$dp[2,1] = \begin{cases} 1 \\ 1 + dp[2,2] \end{cases}$$

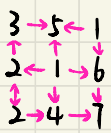
$$dp[1,1] = \begin{cases} 1 \\ 1 + dp[2,1] \end{cases}$$

$$dp[3,2] = \begin{cases} 1 \\ 1 + dp[2,2] \end{cases}$$

dp

3		1
2	1	
1	2	

For better performance, we can convert the the matrix into DAG and use topological to get $O(n^2)$



#3. Top Down $O(n^2)$

DFS.

```
sp(A[]){
    dp[]  // initialize to -1
    for(i=1,...,n){
        for(j=1,...,n){
            compute(A[], dp[], i, j)
        }
    }
    return max{dp[]};
}

compute(A[], dp[], i, j){
    if(dp[i,j] != -1) return dp[i,j];
    dp[i,j] = 1;
    if(i>1 && A[i,j] > A[i-1,j]) dp[i,j] = max{dp[i,j], 1+ compute(A[], dp[], i-1, j)};
    if(i<n && A[i,j] > A[i+1,j]) dp[i,j] = max{dp[i,j], 1+ compute(A[], dp[], i+1, j)};
    if(j>1 && A[i,j] > A[i,j-1]) dp[i,j] = max{dp[i,j], 1+ compute(A[], dp[], i, j-1)};
    if(j<m && A[i,j] > A[i,j+1]) dp[i,j] = max{dp[i,j], 1+ compute(A[], dp[], i, j+1)};
}
```

Longest Increasing Subsequence

$f(i)$ denotes the length of LIS given $A[1:i]$ subsequence end with $A[i]$

1	2	4	3

[1,2] ✓

[1,4,3] ×

[1,2,4] ✓

$$f(i) = \max \begin{cases} f(i-1)+1 & \text{if } A[i-1] < A[i] \\ f(i-2)+1 & \text{if } A[i-2] < A[i] \\ \vdots & \vdots \\ f(1)+1 & \text{if } A[1] < A[i] \\ 1 & \text{none of } A[1], \cdots A[i-1] < A[i] \end{cases}$$

```
LIS(A[1,···,n]){
  dp[] // init to 1
  for(i=2,···,n){
      for(j=1,···,j-1){
          if(A[i] >A[j]){
              dp[i] = max{dp[i], dp[j]+1};
          }
      }
  }
  return max{dp[]};
}
```

Edit Distance

- String 1,
- String 2

Using operation Insertion, Deletion, Replacement to reach String 2 from String 1 with least operations

$f[i,j]$ denotes edit distance given $A[1:i]$, $B[1:j]$

$A[i] \neq B[j]$ "apple" "banana"

If the last step is

1. Insert $f[i,j-1]+1$ "apple" → "banana" insert 'a' in the last step
2. Delete $f[i-1,j]+1$ "apple" → "banana" delete 'e' in the last step
3. Replace $f[i-1,j-1]+1$ "apple" → "banana" replace 'e' with 'a' in the last step

$A[i] == B[j]$ "apple", "battle"

1. Insert $f[i,j-1]+1$ "apple" → "battle" insert 'e' in the last step
2. Delete $f[i-1,j]+1$ "apple" → "battle" delete 'e' in the last step
3. Replace $f[i-1,j-1]+1$ Since $A[i] == B[j]$, replacement is not needed

$$f[i,j] = \min \begin{cases} f[i,j-1]+1 \\ f[i-1,j]+1 \\ f[i-1,j-1] + \begin{cases} 1 & \text{if } A[i] \neq B[j] \\ 0 & \text{if } A[i] == B[j] \end{cases} \end{cases}$$

Base Case:

$f[i,0] = i$; $f[0,j] = j$

Pseudocode

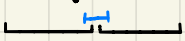
```
editDist(A[n], B[m]) {
    for(i=1,...,n) dp[i,0] = i
    for(j=1,...,m) dp[0,j] = j
    for(i=1,...,n) {
        for(j=1,...,m) {
            dp[i,j] = min{dp[i-1,j]+1, dp[i,j-1]+1};
            if(A[i] ≠ B[j])
                dp[i,j] = min{dp[i,j], dp[i-1,j-1]+1};
            else
                dp[i,j] = min{dp[i,j], dp[i-1,j-1]};
        }
    }
    return dp[n,m];
```

Greedy Algorithm.

Event Schedule

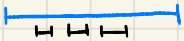
Schedule n events (s_i, e_i) in a room. How many events you can have at most?

#1. Prefer shorter event first

Counter example 

If you chose the shorter one, you can schedule one, but you could have schedule 2.

#2. Earliest start time (s_i)

Counter example 

#3. Earliest end time (e_i) Correct for this one.

#4. Prefer least number of conflict 

- Word Ladder

Given a dictionary, a starting word and a destination word, check if the starting word can be changed to dest word, one character at a time.

① Is it possible?

② # minimum changes

Data Structure

visited HashSet

dist HashMap<String, Integer>

parent(u) HashMap<String, String>

adj.list String ⇒ list of string

```
wordLadder(graph[], start, end, dict){
    q.enqueue(start);
    visited.insert(start);
    dist[start] = 0;
    while(!q.isEmpty()){
        u = q.deque();
        if(u == end) return dist[u];
        for(v in G.neighbor[u]){
            if(visited.contains(v) == False && dict.lookup(v)){
                q.enqueue(v);
                visited.insert(v);
                dist[v] = dist[u] + 1;
            }
        }
    }
    return -1;
}
```

- Counting Islands

Find the number of islands.

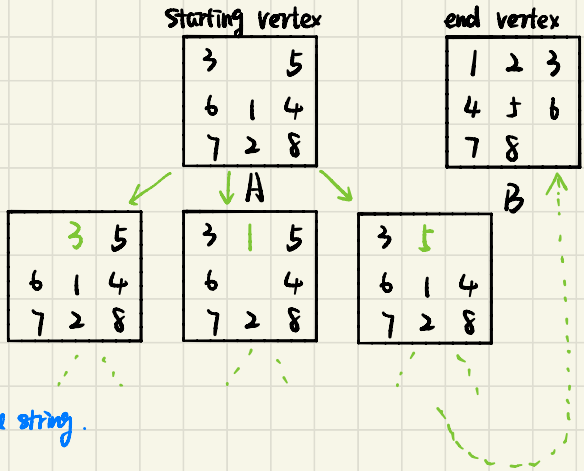
1	1	0	0
1	0	1	1
0	0	1	1
1	1	0	0

0 represents water

1 represents land

Puzzle-8

1. Can we solve the puzzle?
2. # minimum steps to reach B



Represent vertices

#1. encode matrix as string

f(A) = "3x5614728"

find neighbor

decode the string into matrix,

get next state and return the encode string.

#2. 3x3 matrix

design custom hash function for matrix

```
puzzle(graph[vertex => list(vertex)], vertex start, vertex end){

    for(v in G.neighbor(u))

}
```

```
getNeighbor(String state){
    state => matrix
    matrix => string
    return List<String>
}
```

- Find connected component

① # of connected components in G

② ∀ u∈V, which c.c it belongs to

For ②, keep a cc array, in the first BFS call, mark all visited vertices as 1, in the second BFS call, ...

```
BFS-ALL(G){          O(n+m)
    visited[1,...,n] = False;
    counter=0 ; cc[1,...,n]=0, cur_cc=1;
    for(s=1,...,n){
        if(!visited[s]){
            BFS(G, s, visited, cc, cur_cc);
        }
        counter++; ① cur_cc++;
    }
    return counter;
```

```
BFS(G, s, visited, cc, cur){
    q.enqueue(s); visited[s]=T; cc[s]=cur;
    while(!q.isEmpty){
```