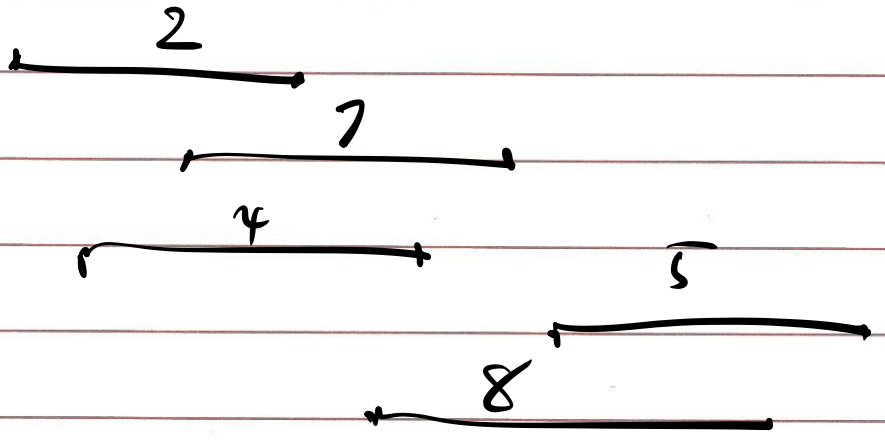
General Approach to Solving optimization problems using Dynamic Programming

1. Characterize the structure of an opt. solution
2. Recursively define the value of an opt. solution
3. Compute the value of an opt. solution in a bottom up fashion
4. Construct an opt. sol. from computed information

Problem Statement

- We have 1 resource
- " " n requests labeled 1 to n
- Each request has start time s_i, finish time f_i, and weight w_i

Goal: Select a subset $S \subseteq \{1..n\}$ of mutually compatible intervals so as to Maximize $\sum_{i \in S} w_i$



Observation: Either job i is part of the opt sol. or it isn't

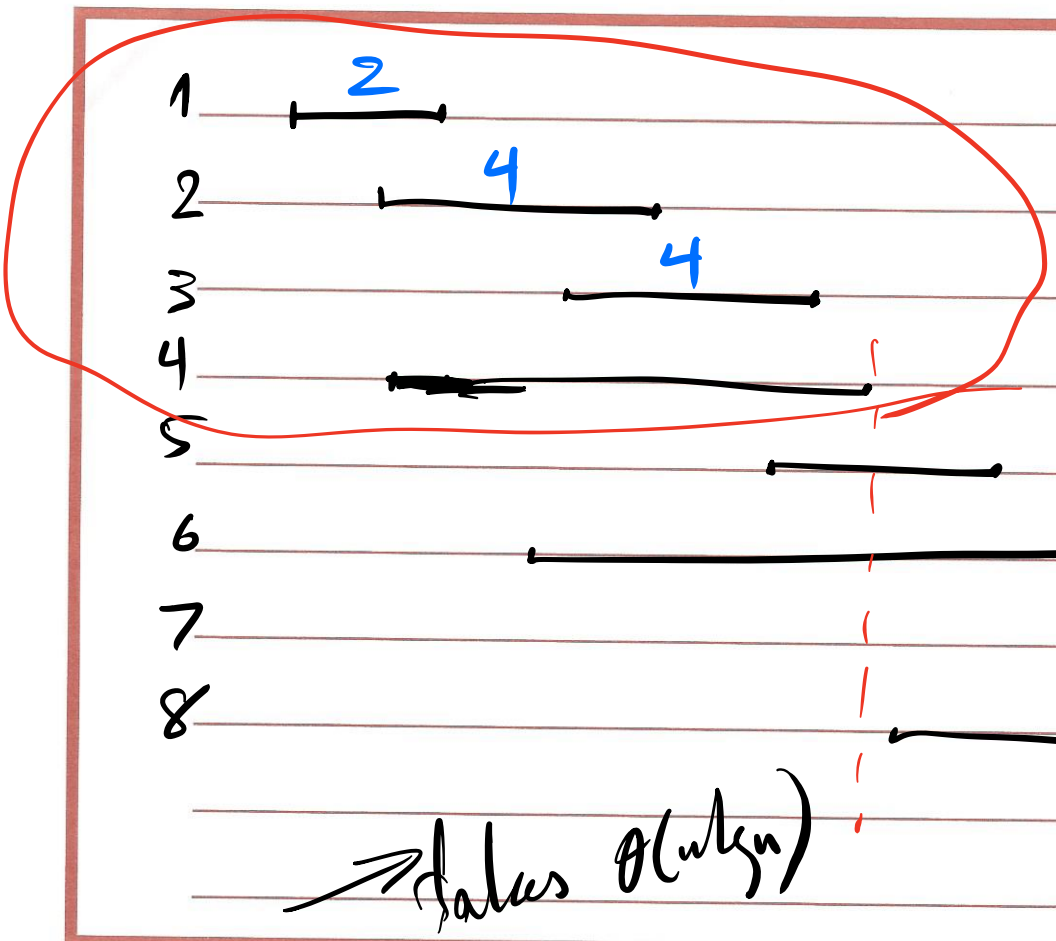
Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

Case 2 - if it isn't, value of the opt. sol. = value of the opt. sol. without job i

Sort requests in order of non-decreasing finish time.

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

Define $P(j)$ for an interval j to be the largest index $i < j$ such that interval i & j are disjoint.



P( )

P(1) = 0

P(2) = 0

P(3) = 1

P(4) = 0

P(5) = 2

P(6) = 1

P(7) = 5

P(8) = 4

→ Solves $\theta(n \log n)$

Def Let O_j denote the opt. solutions to the problem consisting of requests $\{1..j\}$

Let $OPT(j)$ denote the value of O_j

$O_3 = \{1, 3\}$ $OPT(3) = 6$

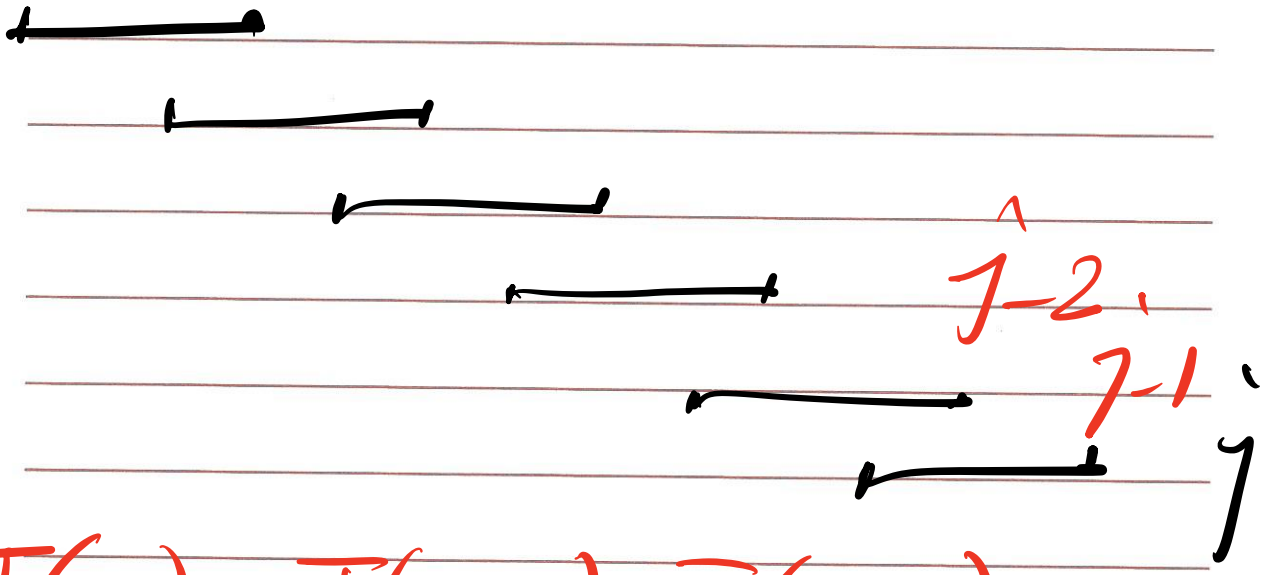
Recurrence formula

Case 1: $j \in O_j \Rightarrow OPT(j) = w_j + OPT(P(j))$

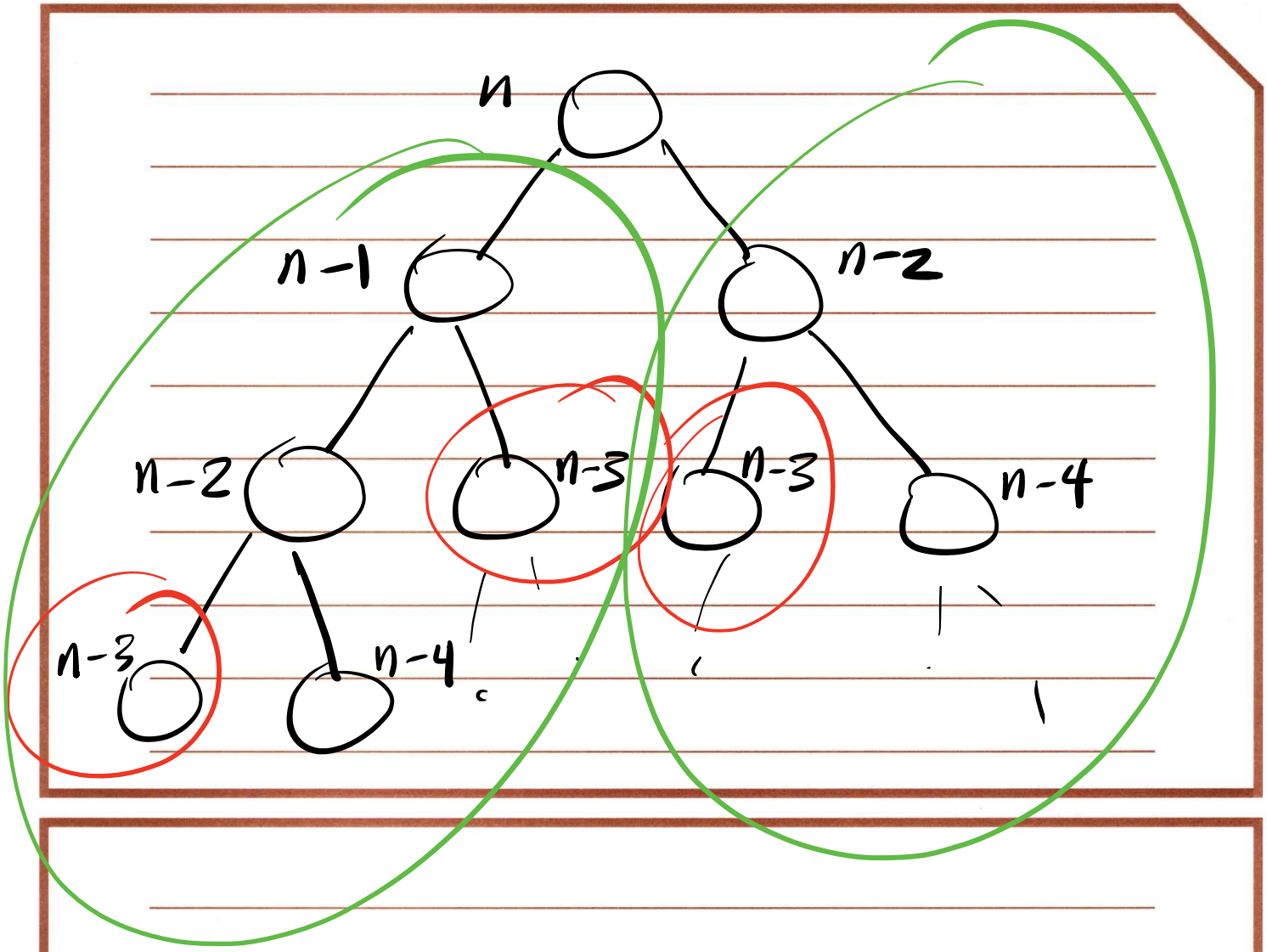
Case 2: $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

Solution:

```
Compute-opt(j)
    if j = 0 then
        return 0
    else
        return Max(
            wj + Compute-opt(P(j)),
            Compute-opt(j-1))
    endif
```



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


n
n-1
n-2
n-2
n-3
n-3
n-4
n-3
n-4

Memoization

Store the value of Compute-opt. in a globally accessible place the first time we compute it. Then simply use this precomputed value in place of all future recursive calls.

```
M-Compute-opt(j)
    if j=0 then
        return 0
    else if M[j] is not empty then
        return M[j]
    else define M[j] = Max(wj + M-Compute-opt(p(j)), M-Compute-opt(j-1))
        return M[j]
    endif
```

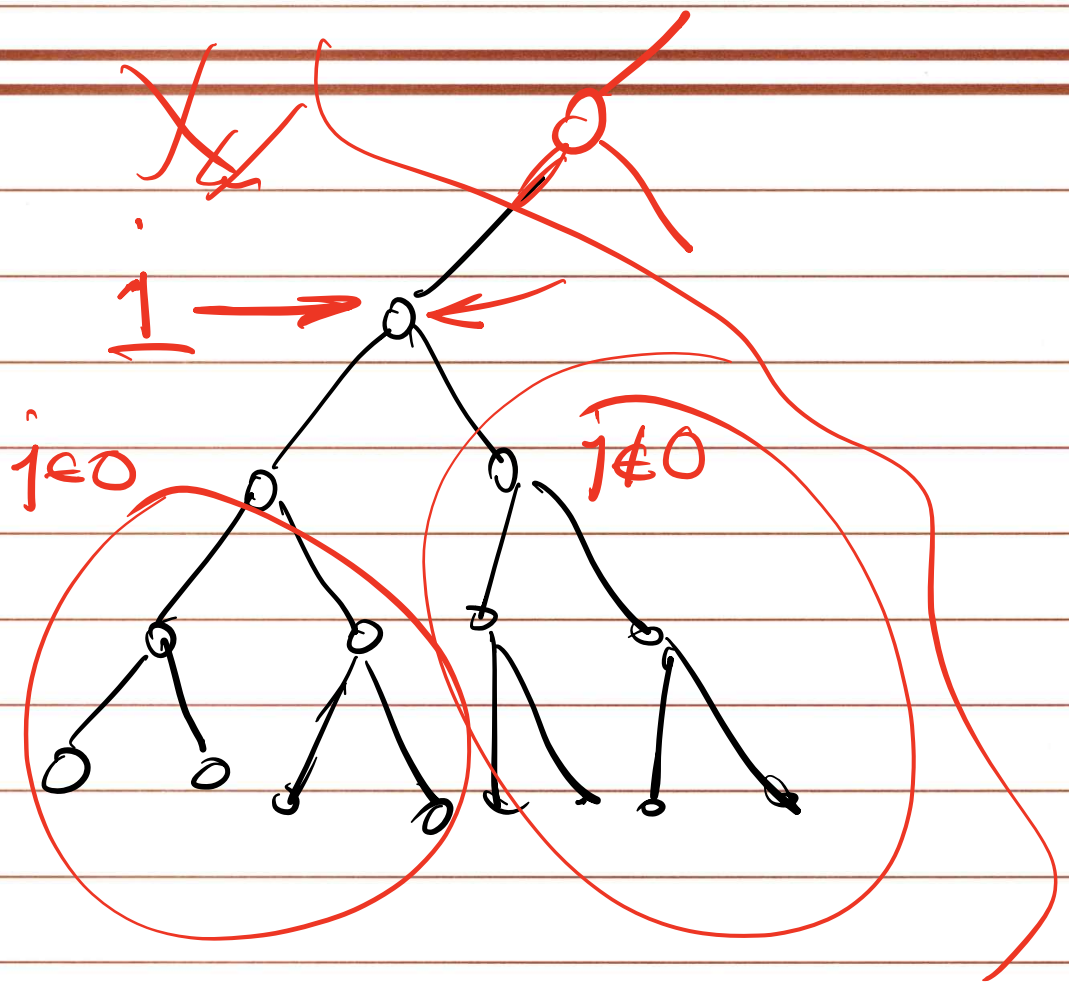
O(n)

initial Sorting : $O(n \log n)$

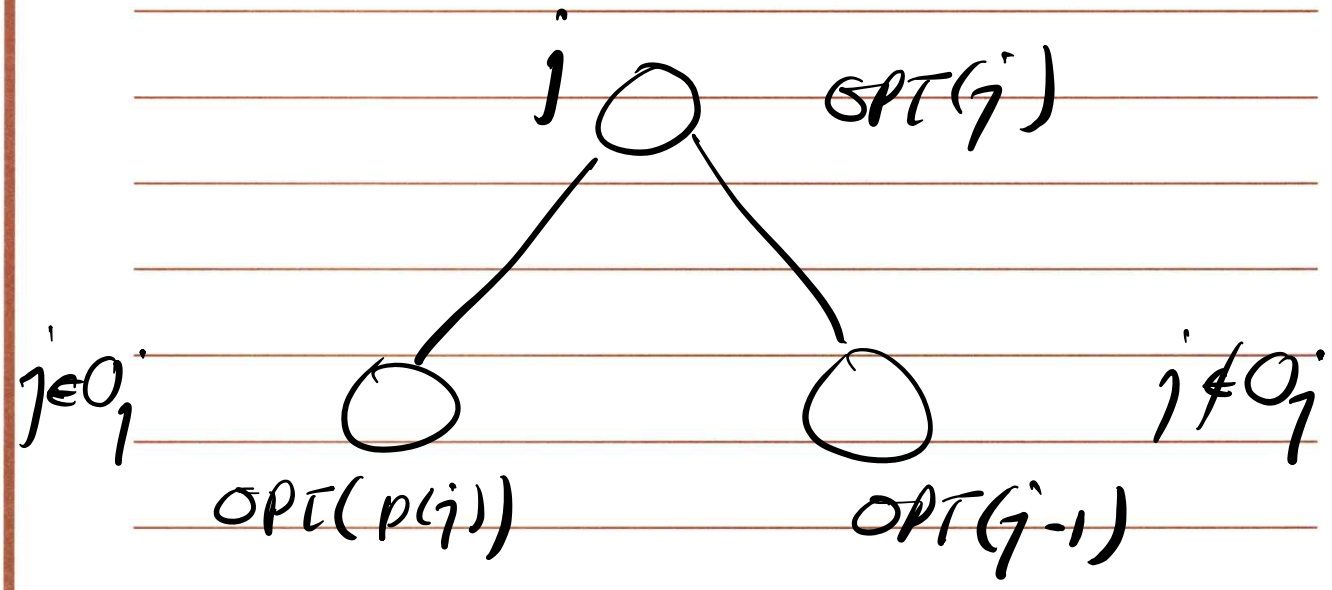
Build P() : $O(n \log n)$

M_Compute_opt : $O(n)$

overall Complexity = $O(n \log n)$



Compute an opt. Sol.



j belongs to O_j iff

$$w_j + OPT(p(j)) \geq OPT(j-1)$$

Find_Solution

if $j > 0$ then

if $w_j + M[p(j)] \geq M[j-1]$ then

output j together w/ the results of Find_Solution ($p(j)$)

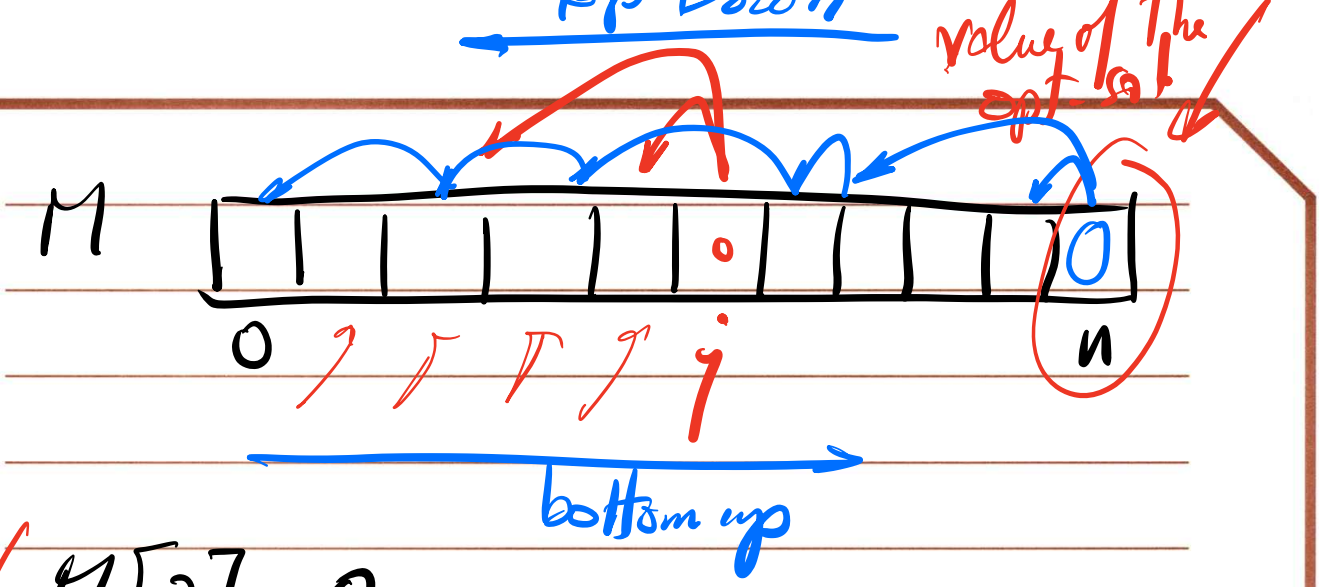
else

output the results of Find_solution ($j-1$)

end if

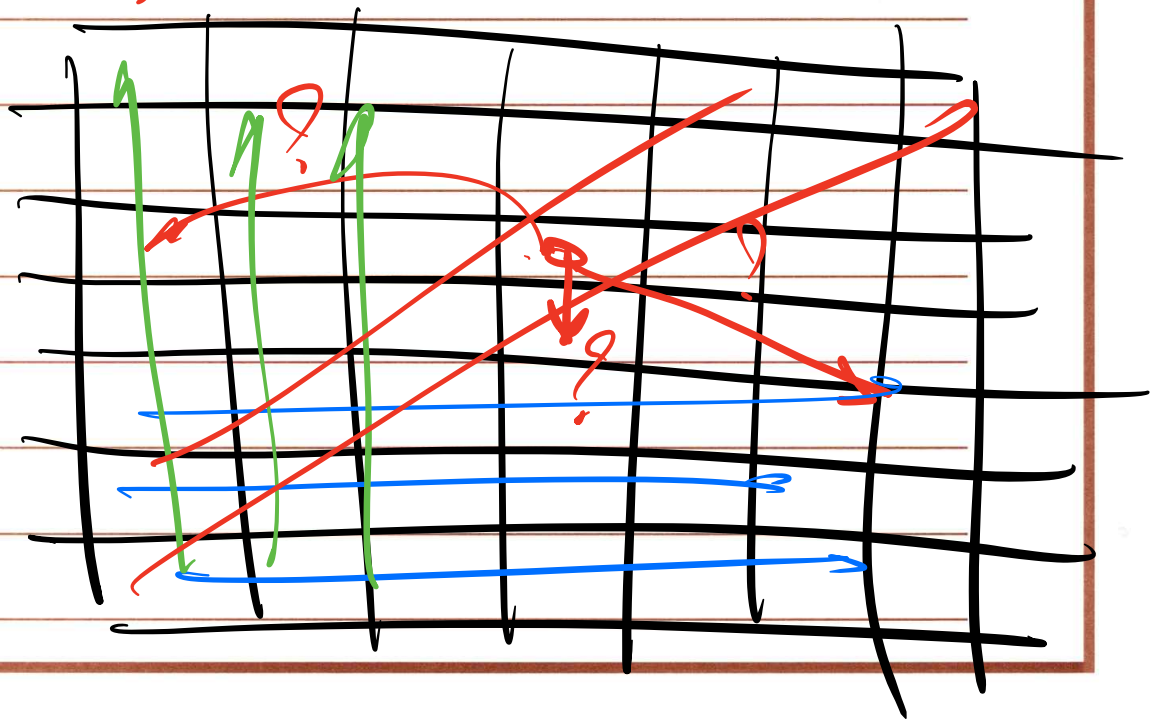
end if

$O(n)$

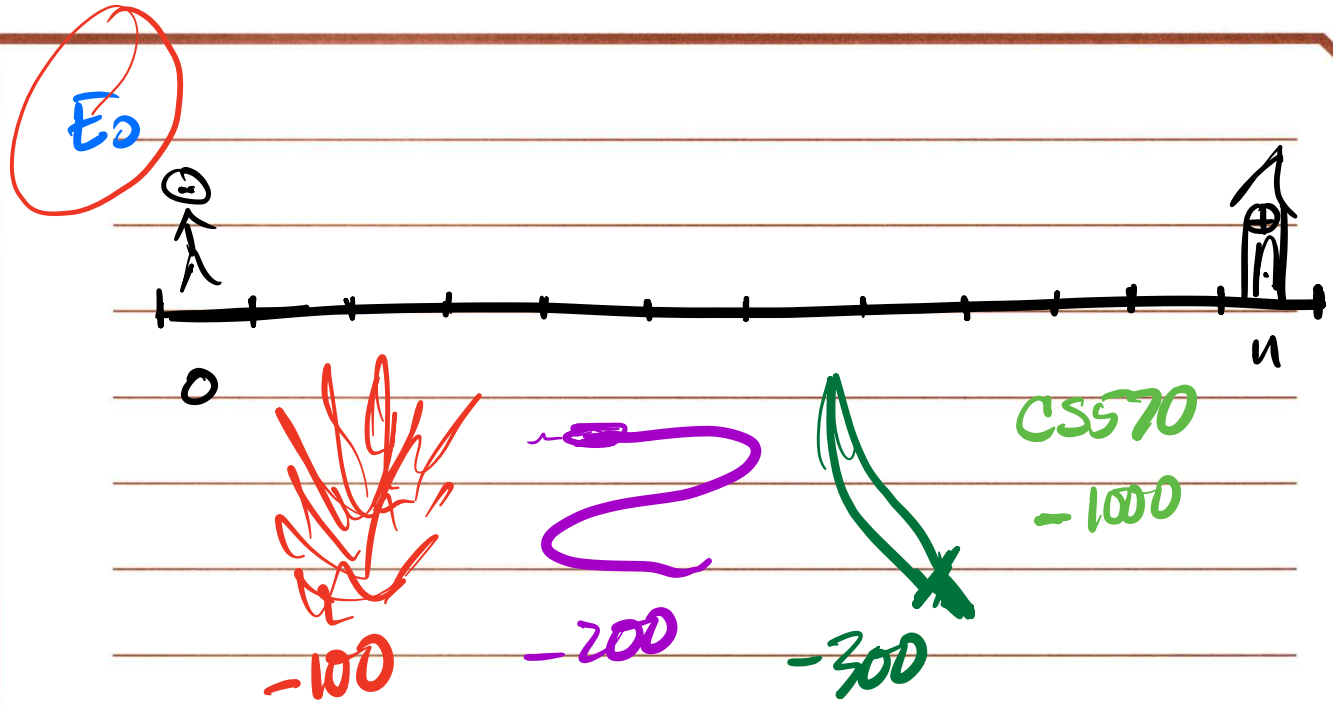



```
M[0] = 0
for i = 1 to n
    M[i] = Max (M[i-1],
                w_i + M[p(i)])
endfor
```

$\Theta(n)$



Videogame Problem



in general, we lose e_i units of energy when landing on stage i

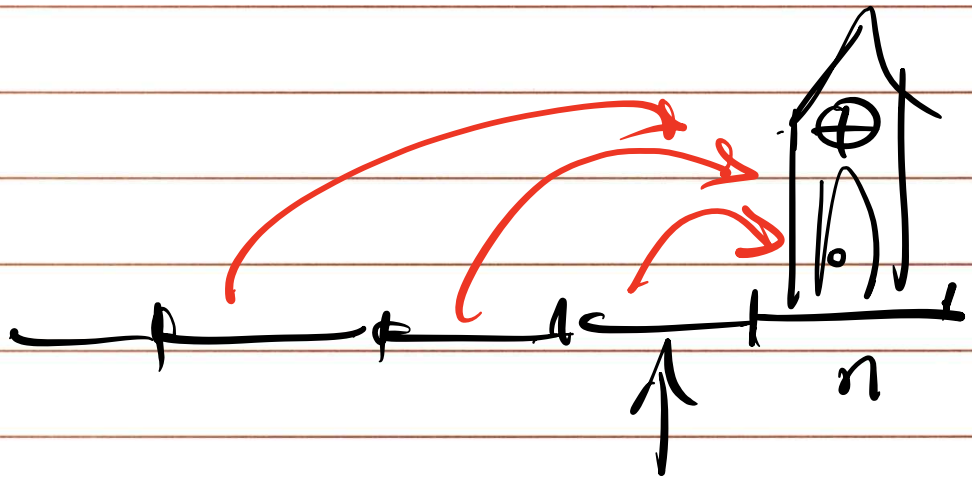
Choice: 1- walk into next stage costs 50 units

2- jump over one stage " 150

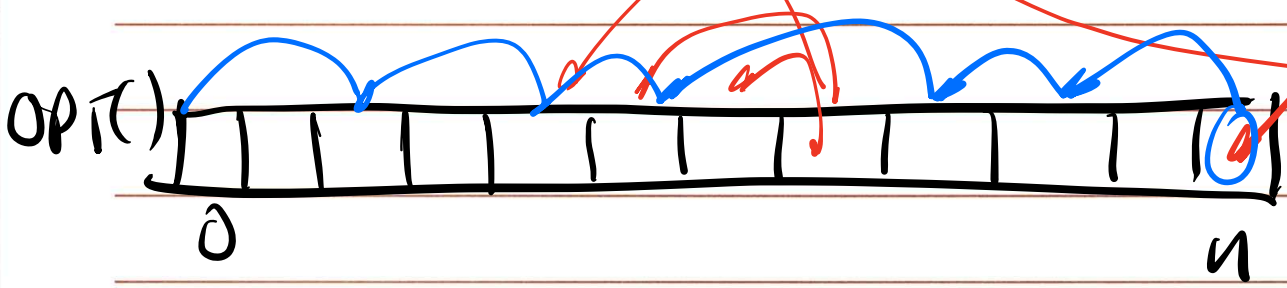
3- " " two stages 350

Question: How do you go home such that you lose as little energy as possible?

OPT(i) = opt. level of energy when we reach stage 'i'.



$$OPT(n) = Max(OPT(n-1) - 50 - e_n,$$
$$OPT(n-2) - 150 - e_n,$$
$$OPT(n-3) - 350 - e_n)$$



Value of opt. sol.

Top down

$OPT(2) = Max(\ldots)$

$OPT(0) = E_0$, $OPT(1) = E_0 - 50 \cdot e_1$

for $i = 3$ to n

end for

$\rightarrow$ takes $\Theta(n)$

Coin Problems

Austrian Schillings denominations.

1
5
10
20
25

Before Euro currency

How to pay for n Schillings w/ the min. # of coins?

$OPT(i)$ = Min # of coins to pay i Schillings w/.

$$OPT(i) = \text{Min}\,(OPT(i-1)+1,\ OPT(i-5)+1,\ OPT(i-10)+1,\ OPT(i-20)+1,\ OPT(i-25)+1)$$

$OPT(0) = 0$, $OPT(1) = \cdots$, $OPT(24) = \cdots$

for $i = 25$ to n

end for

↖ takes $\Theta(n)$

0-1 knapsack &

Subset Sum

Problem Statement

- A single resource
- Requests $\{1..n\}$ each take time w_i to process
- Can schedule jobs at any time between 0 to W

Objective: To schedule jobs such that we maximize the machine's utilization

---

$OPT(i)$ = value of the opt. sol. for requests $1-i$.

$$\begin{cases} \text{if } n \notin O, \text{ then } OPT(n) = OPT(n-1) \\ \text{if } n \in O, \text{ then } OPT(n) = w_n + OPT(n-1) \end{cases}$$

?

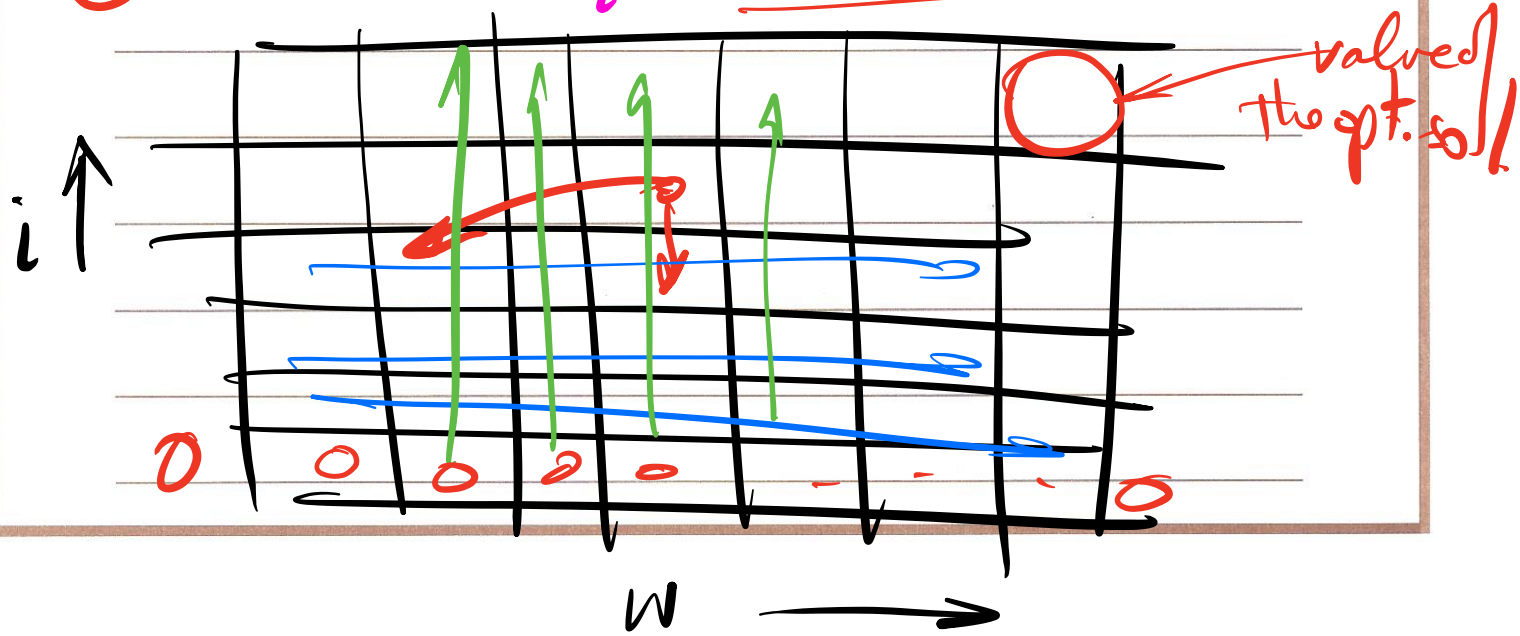
$OPT(i,w)$ = value of the opt. solution using a subset of the items $\{1..i\}$ with Max. allowed weight w.

if $n \notin O$, Then $OPT(n,w) = OPT(n-1, w)$

if $n \in O$, Then $OPT(n,w) = w_n + OPT(n-1, w-w_n)$

If $w < w_i$, Then $OPT(i,w) = OPT(i-1, w)$

else, $OPT(i,w) = Max(OPT(i-1, w), v_i + OPT(i-1, w-w_i))$



```
Subset-sum (n, W)
  array M[0,w] = 0  for each w = 0 to W
  for i = 1 to n
    for w = 0 to W
      use recurrence formula (1)
        to compute M[i, w]
    end for
  end for
  Return M[n, W]
```

n, W

Complexity = $O(nW)$

input:

w_1 | w_2 | -1 | -1 | 1 | 1 | ... | run |

n

W

01010111011

$\log_2 W$ bits

pseudo polynomial run time.

$n_W = n_Z$

$\log_2 W$

Pseudo-polynomial time

An algorithm runs in pseudo-polynomial time if its running time is a polynomial in the numeric value of the input

Polynomial time

An algorithm runs in polynomial time if its running time is a polynomial in the length of the input (or output).

Discussion 6

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order:$1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

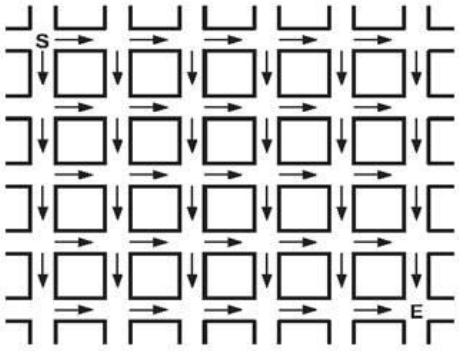


Figure A.

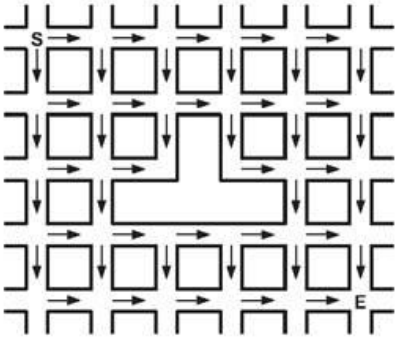
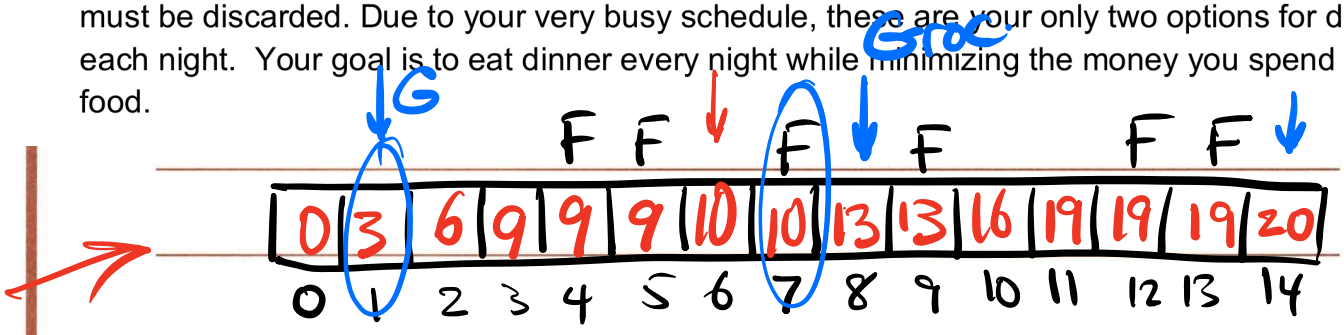


Figure B.

b) Repeat this process with Figure B; be wary of dead ends.

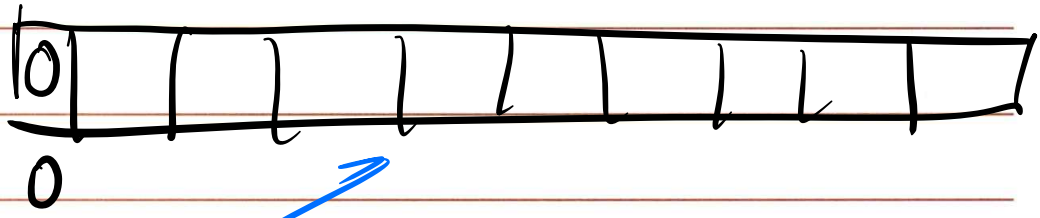
programming problem.

2. Graduate students get a lot of free food at various events. Suppose you have a schedule of the next *n* days marked with those days when you get a free dinner, and those days on which you must acquire dinner on your own. On any given day you can buy dinner at the cafeteria for $3. Alternatively, you can purchase one week's groceries for $10, which will provide dinner for each day that week (that day and the six that follow). However, because you don't have a fridge, the groceries will go bad after seven days (including the day of purchase) and any leftovers must be discarded. Due to your very busy schedule, these are your only two options for dinner each night. Your goal is to eat dinner every night while minimizing the money you spend on food.



$OPT(i)$ = Min Cost to get dinner for days $1..i$

$$OPT(i) = \begin{cases} OPT(i-1) \text{ if we have free food on } i \\ \text{otherwise, } Min(OPT(i-1)+3, \\ \qquad OPT(i-7)+10) \end{cases}$$



Filling this out takes $O(n)$

3. You are in Downtown of a city and all the streets are one-way streets. You can only go east (right) on the east-west (left-right) streets, and you can only go south (down) on the north-south (up-down) streets. This is called a Manhattan walk.

a) In Figure A below, how many unique ways are there to go from the intersection marked S (coordinate (0,0)) to the intersection marked E (coordinate (n,m))?
Formulate the solution to this problem as a dynamic programming problem. Please make sure that you include all the boundary conditions and clearly define your notations you use.

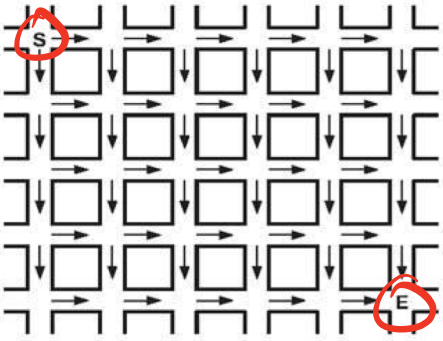


Figure A.

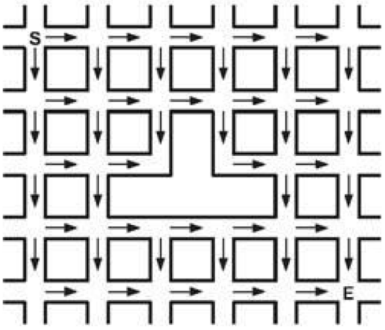


Figure B.

b) Repeat this process with Figure B; be wary of dead ends.

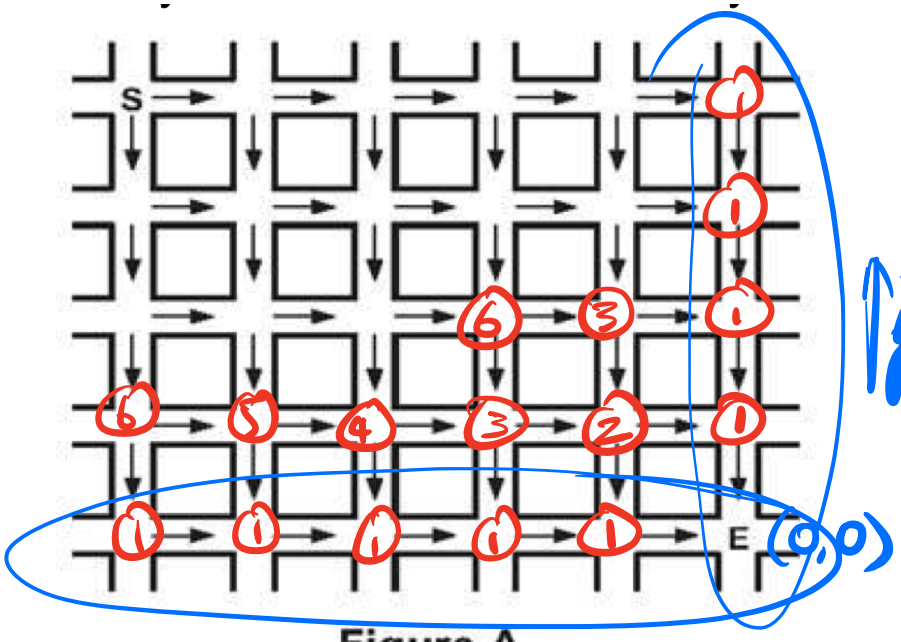


Figure A.

$OPT(i,j)$ = # of ways to go from (i,j) to E.

$OPT(i,j) = OPT(i-1,j) + OPT(i,j-1)$

This takes $O(nm)$

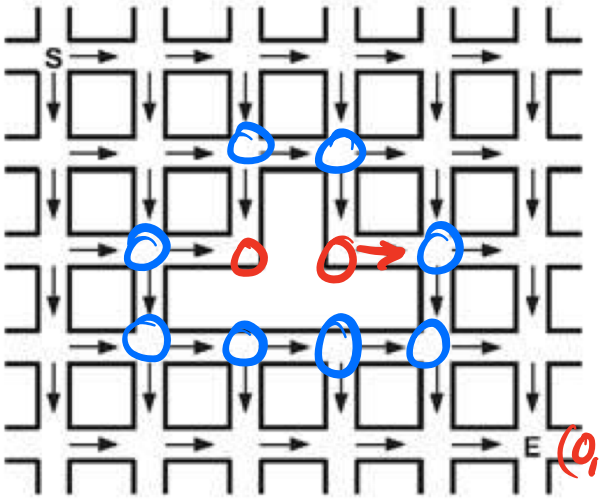


Figure B.

i

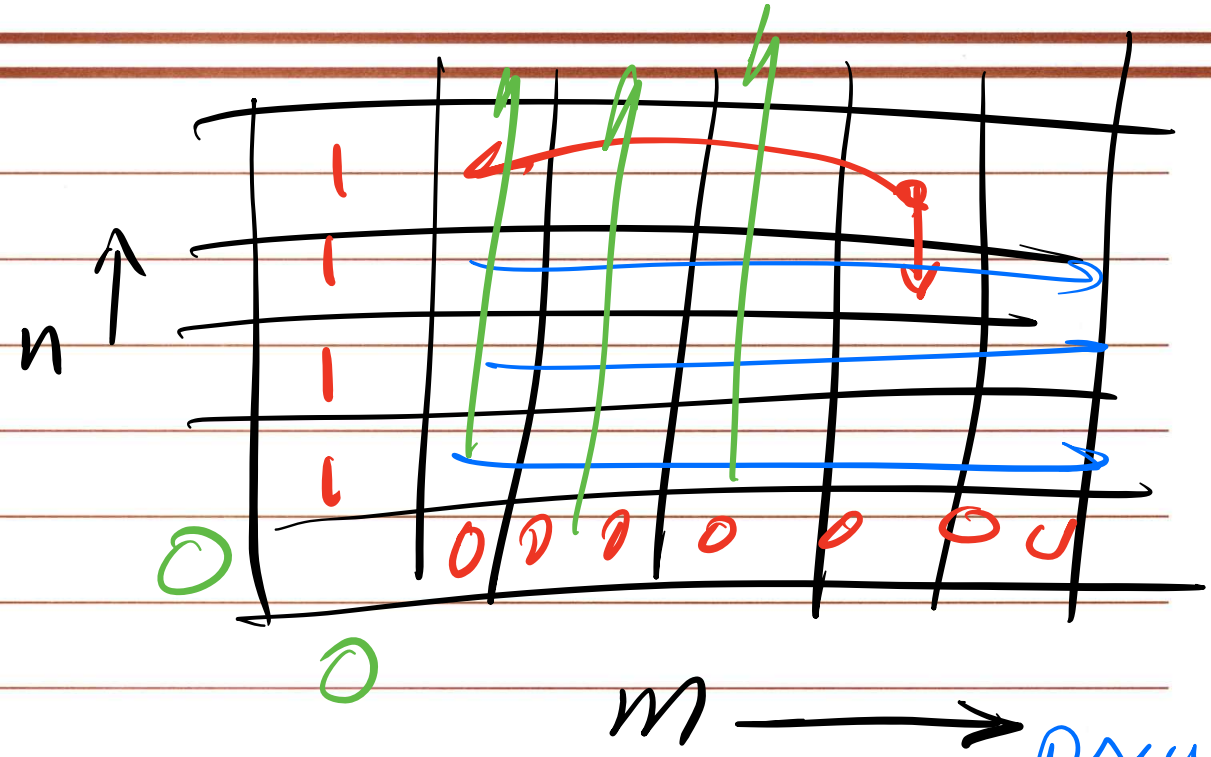
$(2,2) \rightarrow OPT(i,j) = OPT(i-1,j)$

$(3,2) \rightarrow OPT(i,j) = 0$

1. You are to compute the total number of ways to make a change for a given amount m. Assume that we have an unlimited supply of coins and all denominations are sorted in ascending order: $1 = d_1 < d_2 < \ldots < d_n$. Formulate the solution to this problem as a dynamic programming problem.

Count(n, m) = no. of ways to pay for amount m using coins 1..n.

$$Count(n, m) = Count(n-1, m) + Count(n, m - d_n)$$



takes $O(nm)$ ← pseudo polynomial!

Assume you want to ski down the mountain. You want the total length of your run to be as long as possible, but you can only go down, i.e. you can only ski from a higher position to a lower position. The height of the mountain is represented by an *n* X *n* matrix A. A[i][j] is the height of the mountain at position (i,j). At position (i,j), you can potentially ski to four adjacent positions (i-1,j) (i,j-1), (i,j+1), and (i+1,j) (only if the adjacent position is lower than current position). Movements in any of the four directions will add 1 unit to the length of your run. Provide a dynamic programming solution to find the longest possible downhill ski path starting at any location within the given *n* by *n* grid.

1200	1000	1200	1500	1700	1500	1000	1000
1100	1600	2000	1900	1800	1600	1200	1250
1200	1700	1900	2300	2400	2000	1900	1750
1000	1500	2000	2450	2600	2100	2000	1500
1100	1500	1800	2200	2300	2200	2100	1600
1100	1000	1500	1800	2100	1900	2000	1700
1000	1000	1200	1300	1700	1900	1900	1800
900	800	1000	1200	1500	1900	2000	2100

$OPT(i,j)$ = length of the longest downhill path starting from (i,j)

$$OPT(i,j) = \max(OPT(i',j')+1, \text{ where } (i',j') \text{ is a neighbor } \& A(i',j') < A(i,j))$$

(Sort points in increasing order of elevation

initialize all local minima to 0

loop over all subproblems in the order given by the sort.

Sort $O(n^2 \lg n^2) = O(n^2 \lg n)$

iteration $\longrightarrow O(n^2)$

overall complexity $= O(n^2 \lg n)$

Can be improved using topological ordering instead of sorting

topological ordering can be done in linear time w/ respect to the size of the graph which has $O(n^2)$ nodes & edges, so topological sort will take $O(n^2)$

Imagine starting with the given decimal number n, and repeatedly chopping off a digit from one end or the other (your choice), until only one digit is left. The square-depth SQD(n) of n is defined to be the maximum number of perfect squares you could observe among all such sequences. For example, SQD(32492) = 3 via the sequence

32492 → **3249** → **324** → 24 → **4**

since 3249, 324, and 4 are perfect squares, and no other sequence of chops gives more than 3 perfect squares. Note that such a sequence may not be unique, e.g.

32492 → **3249** → 249 → **49** → **9**

also gives you 3 perfect squares, viz. 3249, 49, and 9.

Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that runs in constant time and returns 1 if x is a perfect square and 0 otherwise.

$OPT(i,j)$ = Max. no of sq's from digit a_i to digit a_j

$n_{ij} = a_i \ldots a_j$

$OPT(i,j) = Max(OPT(i+1,j), OPT(i,j-1)) + IS\_Square(n_{ij})$

IS_Square() is a function that returns a 1 if the number passed to it is a perfect square and returns 0 otherwise.