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

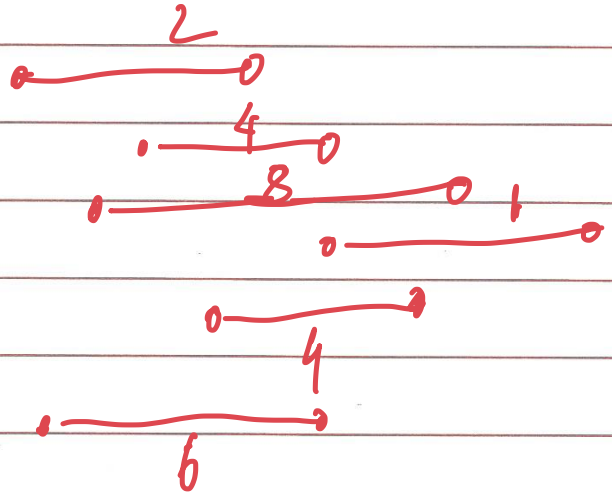
3. Compute the value of an opt. solution in a bottom up fashion

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 $\sum_{i \in S} w_i$



req = [ ]

req = sort(req)

$O(n \log n)$

fnc(req):

ans = max(fnc(1:req), w[0] + fnc(1:req))

↑ remove all non compatible reqs

my solution

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

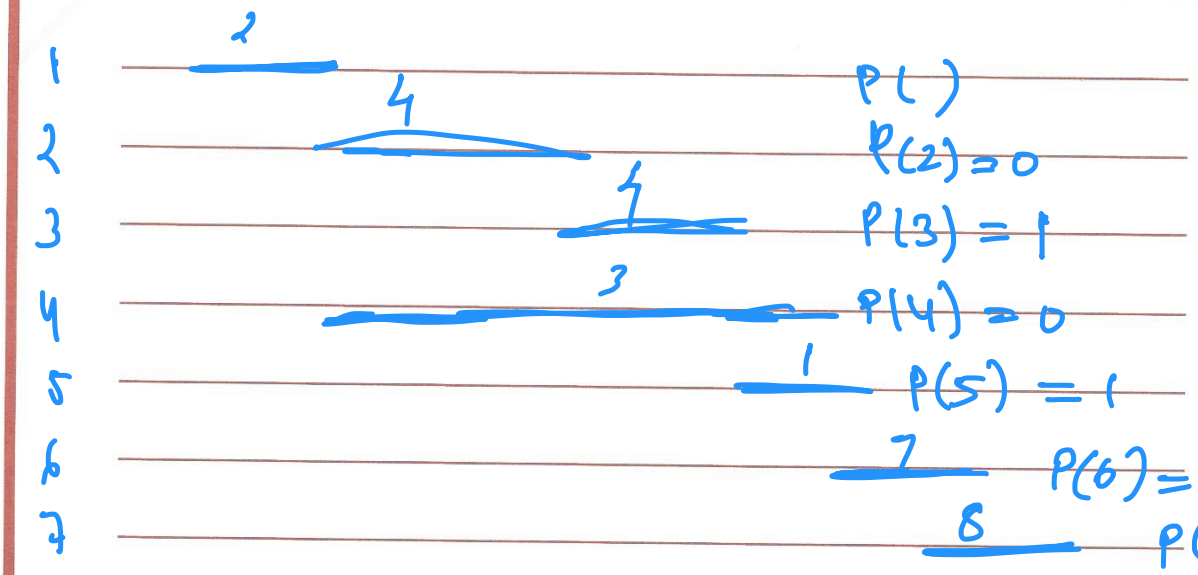
Case 2 - if it isn't, value of the opt. sol. = value of the opt. sol. without job i

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()



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

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

(compatible req with j)

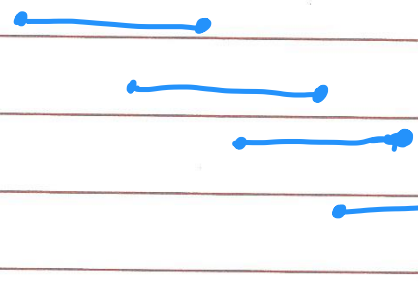
CASE 1: $j \in O_j \Rightarrow OPT(j) = w_j + OPT(P(j))$

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) )
```

WORST CASE

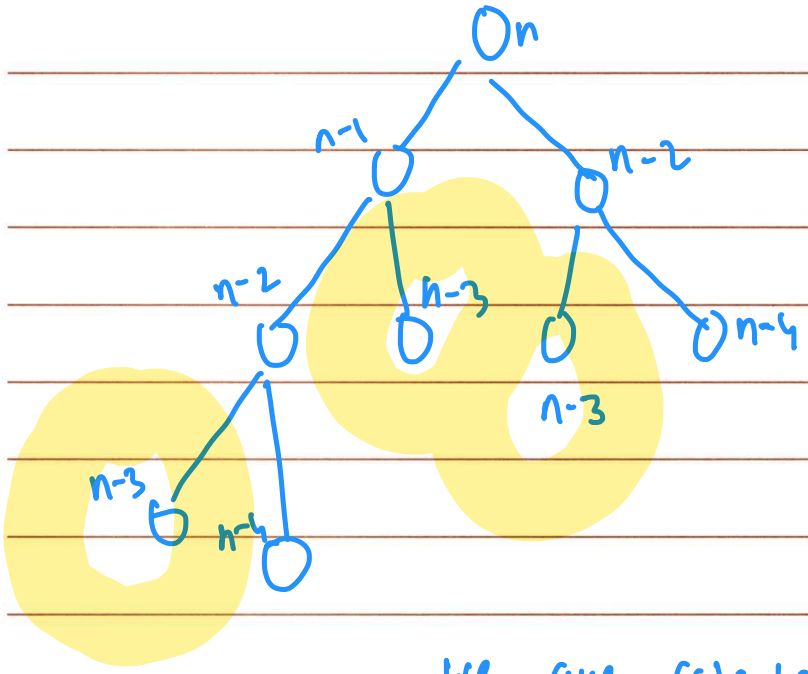


each req only has
- 2 overlap
- 1 before
- 1 after

$T(n) = T(n-1) + T(n-2)$ → w_j + compute-opt(P(j)) → j-2

$T(n-1)$ → compute-opt(j-1)

{exponential time complexity}



We are calculating same thing again and again

Hence we can store it once it is found and reuse it when called again

We use memoization for the issue

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:
    M[j] = max( M_compute_opt(j) ,
                w(j) + M_compute_opt(p(j)) )

    return M[j]
```

O(n)

As every subproblem solved only once and it is reused when called again

2:1 → 4
3:1 → 5
4:1 → 6
5:1 → 7

P( )
P(1)
P(2)
P(3)

f = [4,5,6,7]

Initial sorting — O(n log n)

making p[ ] — O(n log n)

- Using binary search we find the point from which all further points will be compatible

M-compute-opt — O(n)

P(3) ⇒ largest index from [4,5,6,7] < 5:1 → do binary search

O(n log n)

[4,5,6,7]

5 < 6 go left

[4,5]

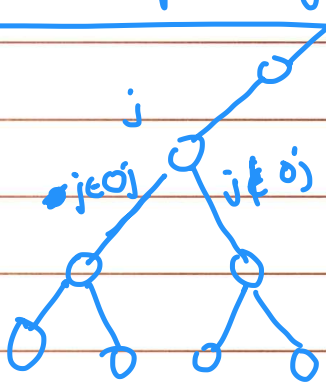
5 < 5:1 go right and check if bigger number is there

[4,5,6,7]

6 > 5:1

return 5

Overall complexity — O(n log n)

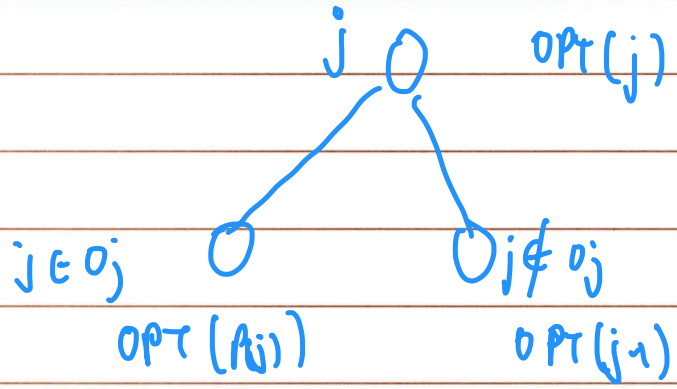


We

Do bottom up to find optimal value

Then Do Top Down to trace the path.

Compute an optimal sol



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

j is in optimal solution if & only if

```
Find_Solution
    if j > 0 then
        if w_j + M[p(j)] ≥ 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)

worst case when subproblem size does not reduce much

i.e compatible set reduces very less

$j \rightarrow j-1 \rightarrow j-1$

{size of subproblem only reduces by 1}

Memoization array

M

Start solving left → right

Trying to make memoization array using iteration

if we get m[] we found the ans

m[0]

O(n)

for i=1 to n

$m[i] = \max\left(m[i-1], w_i + m[p(i)]\right)$

Top Down

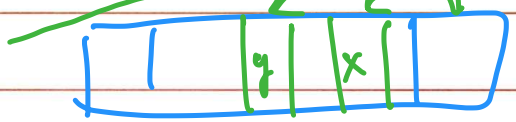
M

Value of optimal solution

Bottom up

Top down having optimal value we are finding the path taken.

$j \in O_j$ $j \notin O_j$



just choose which is greater and take that path

if $x > y$: $j \notin O_j$

if $y > x$: $j \in O_j$

Divide & Conquer

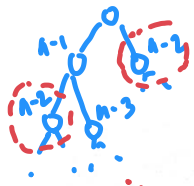
↓

- independent subproblems
- we find optimal soln for each subproblem and combine

Dynamic

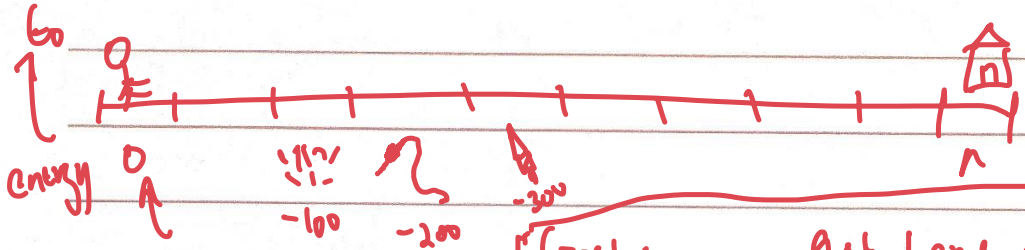
↓

- dependent subproblems {reduce overlapping subproblems}
- we find optimal value then from that optimal solution



if we consider them independent subproblems we lose the whole advantage of Dynamic programming

Videogame Problem



Goal: get home at level 'n'

E_0 ← initial energy

Each stage reduces some energy

In general, we lose e_i units of energy when we land in stage i

		Cost
(choices) 1.	walk into stage	50 units
2.	Jump over a stage	150
3.	Jump over 2 stages	35

Go home such that you loose minimum energy

We have 'n' unique subproblems

stage $0 \to 1$

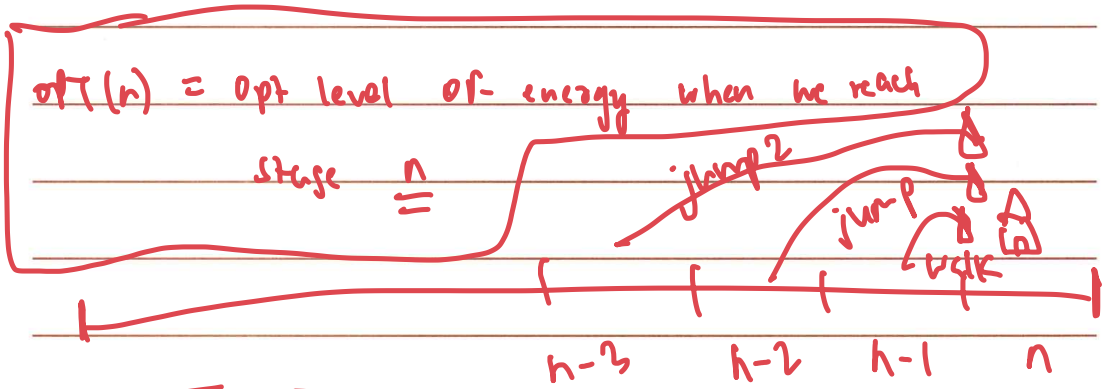
$0 \to 2$

$0 \to 3$

$\vdots$

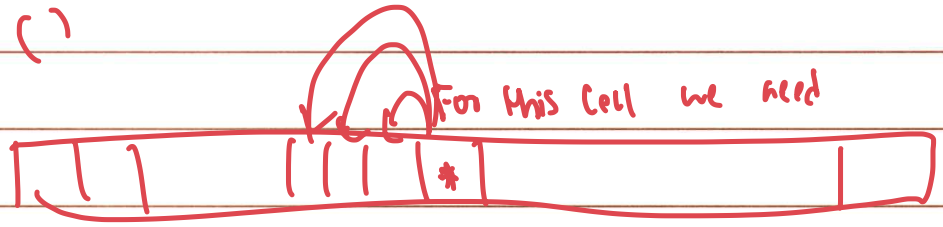
$0 \to n$

OPT(n) = opt level of energy when we reach stage $= n$



$$OPT(n) = \max \begin{cases} OPT(n-1) - 50 - e_n, \\ OPT(n-2) - 150 - e_n, \\ OPT(n-3) - 350 - e_n \end{cases}$$ Formula (1)

OPT( )



OPT(0) = E_0 OPT(1) = $E_0 - 50 - e_1$ OPT(2) = max(. . .)

For i = 3 to n

Use rel formula (1)

End For

Austrian Coin Denomination

Goal: Gives less no. of coins

1

5

10

20

25

return 40

greedy

25, 10, 5

optimal

20, 20

OPT(n) = min no. of coins to pay for n Schillings

$OPT(n) = \min \begin{cases} 1 + opt(n-1) \\ 1 + opt(n-5) \\ 1 + opt(n-10) \\ 1 + opt(n-20) \\ 1 + opt(n-25) \end{cases}$

Solving

right to left

left → right

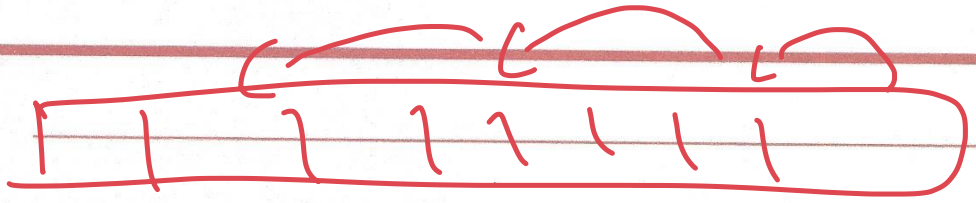
initialize opt(1 ... 25)

Solve

25

For i: 25 to n

do



Find opt then top down to find number of each coins included.

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 $\underline{0}$ to $\overline{W}$

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

---

OPT(i) = value of the optimal soln for job 1 ... n

if $n \neq 0 \Rightarrow$ OPT(n) = OPT(n-1)

if $n \in 0 \Rightarrow$ OPT(n) = w_n + OPT(n-1)

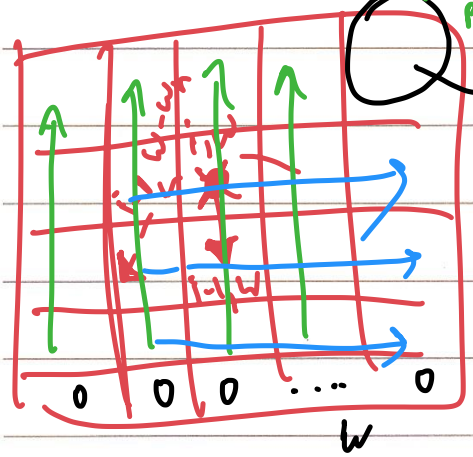
$OPT(i, w)$ = value of the opt. solution using a subset of the items $\{1 \ldots i\}$ with Max. allowed weight w.

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 (for 0 1 Knapsack problem) $+ OPT(i-1, w - w_i))$



$OPT(n, W)$ value of optimal solution

RHS from ②

We can fill array ↑ way

and → way

When $i = 0$ i.e no row all 0

```
Subset-sum (n, W)
  array M[0,w] = 0  for each w = 0 to W
  for i = 1 to n                          'n' times
    for w = 0 to W                        'w' times
      use (recurrence formula (2)) to compute M[i,w]   Constant Time
    endfor
  endfor
  Return M[n,W]        Takes O(n * w)
```

Takes $O(n * w)$ — size of array

Input : w_i — w of each i

1 . — — — — n

associated with numeric value of W

W ⇒ int (Capacity)

011 01001 — int is represent by 0 and 1 in PC

$\log_2 W$

$O(n * W)$ ⟹ pseudopolynomial complexity

Complexity depends on W

Complexity is in terms of size of input

$O(n * W)$ → knapsack capacity (int)

↑ size of req (no. of req)

$$n \cdot W = n \cdot 2^{\log_2 W}$$

∴ solution is running in exponential time wrt to input size

∴ It is not an efficient solution

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.

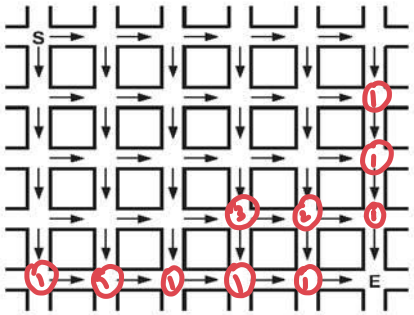


Figure A.

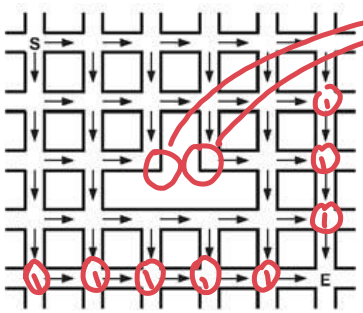
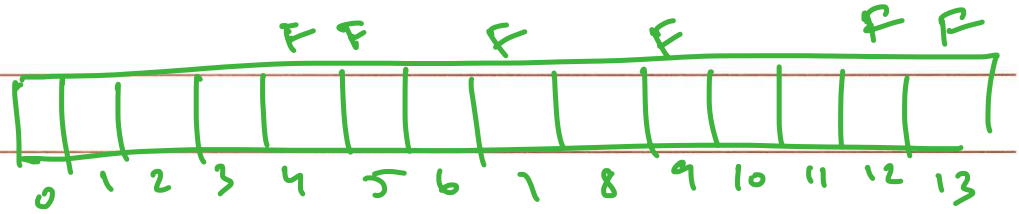


Figure B.

any this two points will have OPT( ) value affected

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

(2)

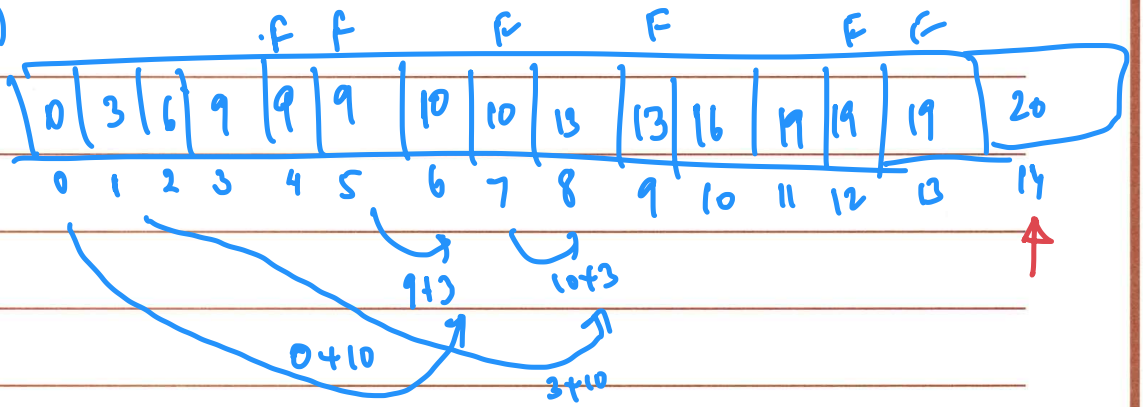


Subproblems $0 \to 1$, $0 \to 2$, ... $0 \to n$

OPT(i) = min cost of dinner for Days 1 ... i

$$OPT(i) = \begin{cases} OPT(i-1) & \text{if there is free food on day } i \\ \text{otherwise} \quad \min\left(OPT(i-1)+3,\ OPT(i-7)+10\right) \end{cases}$$

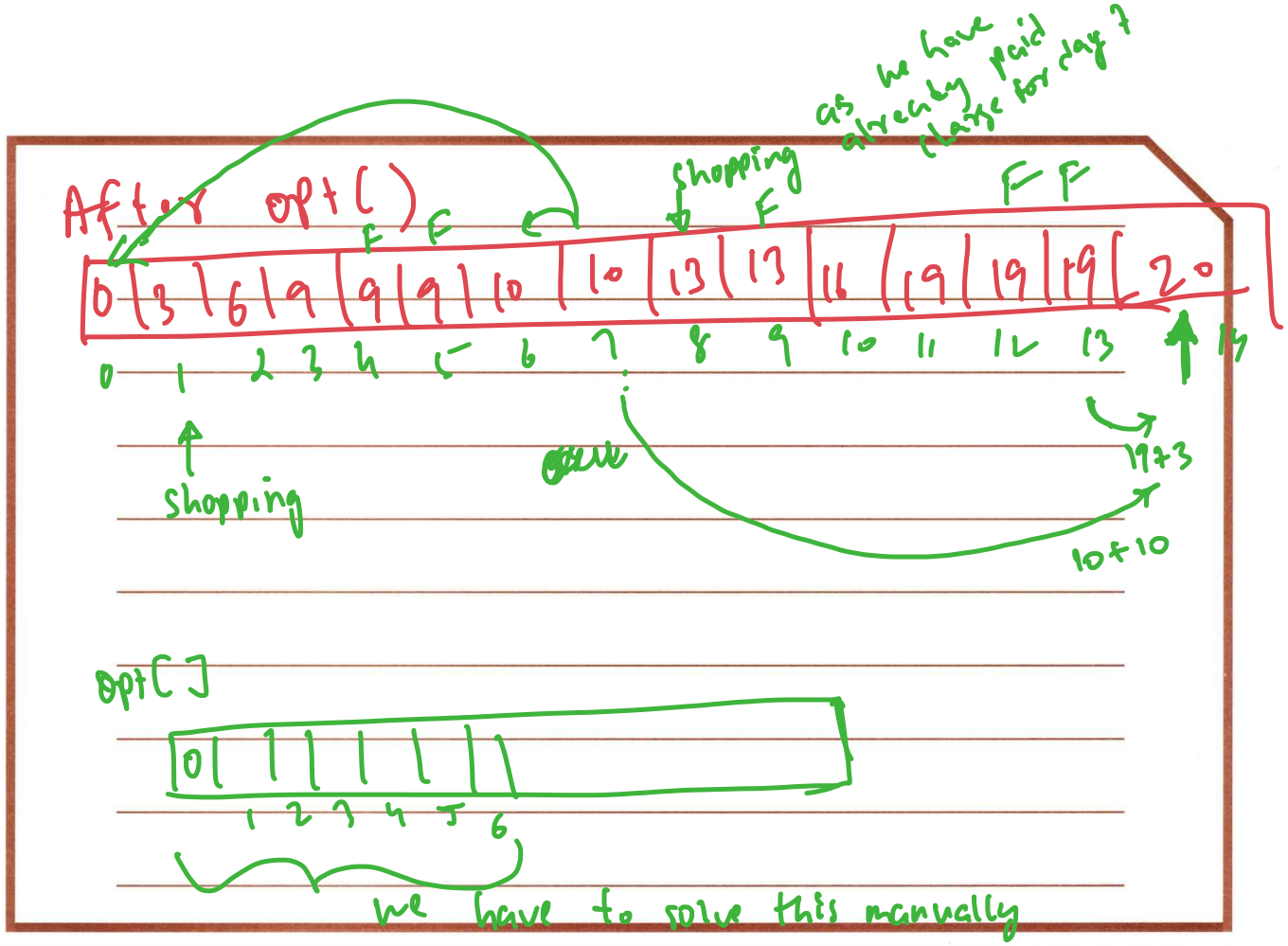
OPT()



Bottom Up

Bottom Up parse fills down this array

Top Down



for i=7 to n

Rec Formula.

(3) $(0,0) \rightarrow (n,m)$ { moves → ↓

3) (a)

OPT(i, j) = no. of ways to go from (i, j) to E

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

```
for i in (m)
    for j in (n):
        formula
```

$O(m * n)$

IF input is array of size m*n
then soln is efficient

as n, m represent prob size

If input n=int m=int
not efficient

we do not know how large they are
((SAME AS IN ABOVE PAGE 20)

3(b) If (i,j) is $(2,2)$ then

$$opt(i,j) = opt(i-1,j)$$

If (i,j) is $(3,2)$

$$opt(i,j) = 0$$

Else

(SAME AS IN 3(A))

---

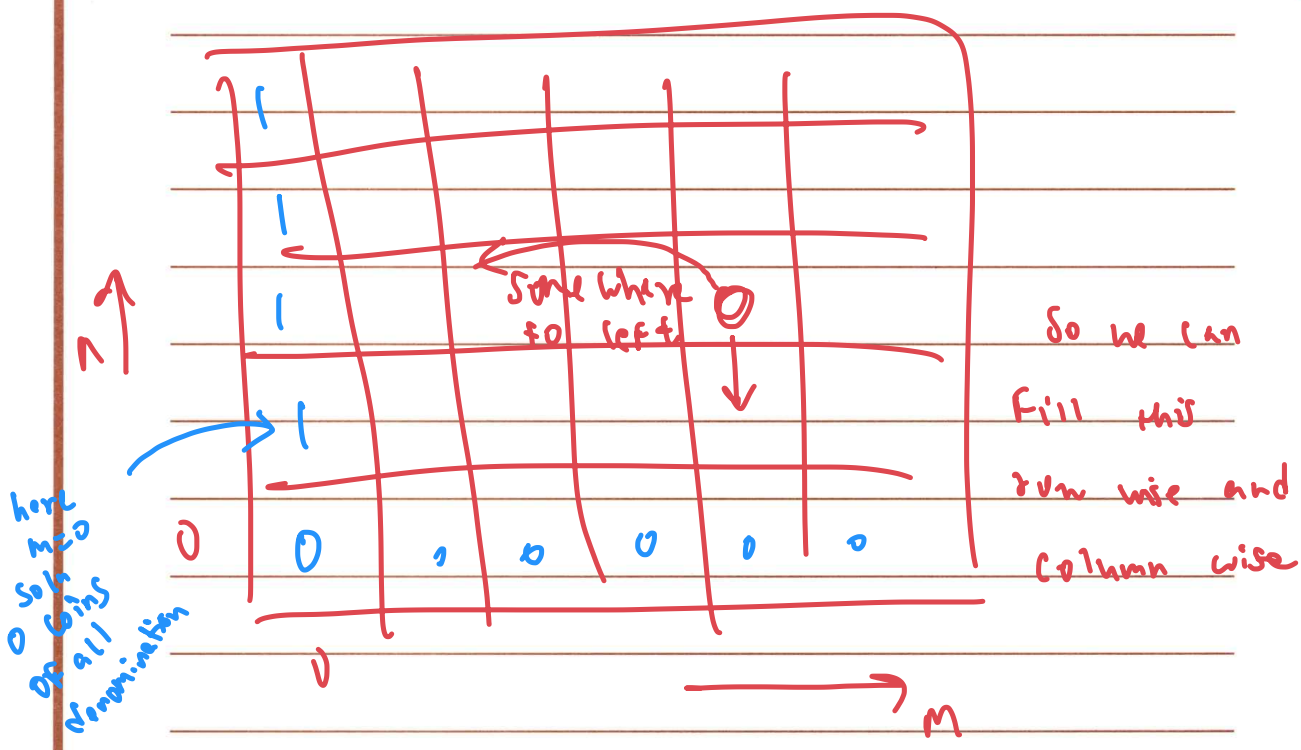
(1) $d_1 < d_2 < \ldots < d_n$ denominations

count (n,m) ⇒ no. of ways to pay for amount m using coin denom's $1 \ldots n$

denomination d_n used atleast once

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

denomination d_n not used



Complexity $\approx O(n * m)$

- n: len of denomination array
- m: int amount

Pseudo polynomial

Not an efficient solution

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

Sort this by elevation.

OPT(i,j) = length of longest path that ends (i,j)

OPT(i,j) = Max (OPT(i',j') + 1 such that (i',j') is a neighbour that is higher than (i,j))

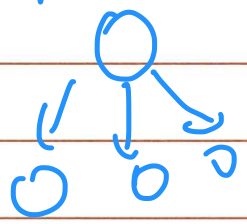
This takes $O(n^2 \log n^2)$ sorting

$\Downarrow$

$O(n^2 \log n)$

To reduce $O(n^2 \log n)$

we can use graph



points where we can go from a particular node

So it will be DAG

↳ we can only go from high to low

Thm on graph find topological ordering

$O(n^2)$

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.

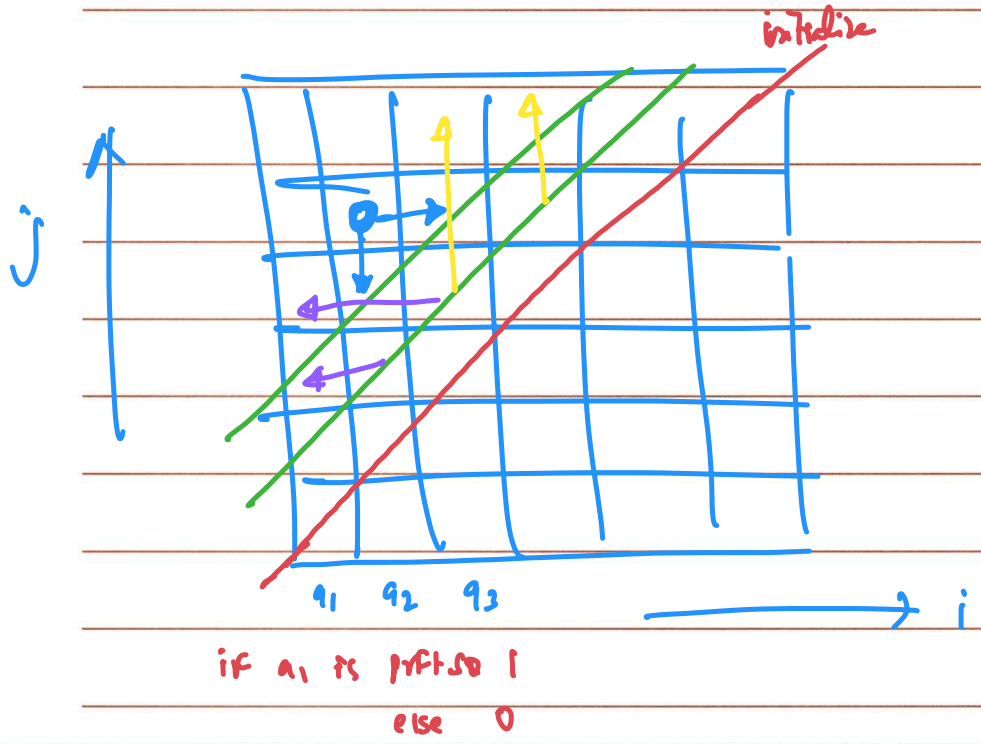
Describe an efficient algorithm to compute the square-depth SQD(n), of a given number n, written as a d-digit decimal number $a_1 a_2 \ldots a_d$. Analyze your algorithm's running time. Your algorithm should run in time polynomial in d. You may assume the availability of a function IS_SQUARE(x) that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

∴ To reach SQD the soln is not unique

$(a_1 a_2 \cdots a_n)$ ⇒ digits

$n_{ij} = a_i a_{i+1} \cdots a_j$

$$OPT(i,j) = \max \begin{pmatrix} OPT(i+1, j), \\ OPT(i, j-1) \end{pmatrix} + \text{IS\_SQUARE}(n_{ij})$$



if a_1 is prime 1
else 0

Takes $O(n^2)$ → no. of digits in the number

efficient array