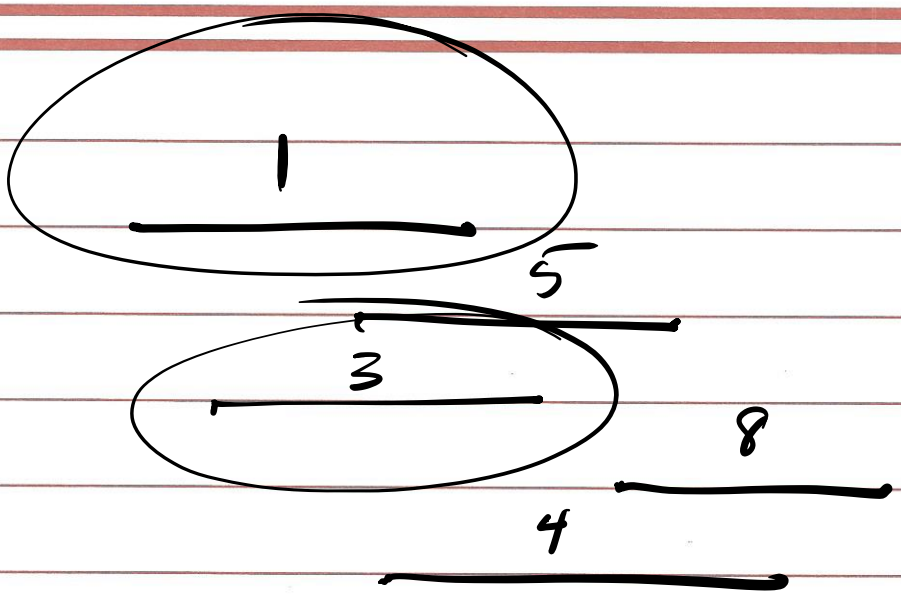
General Approach to Solving optimization problems using Dynamic Programming

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: The opt-sol either contains job i in it or it doesn't

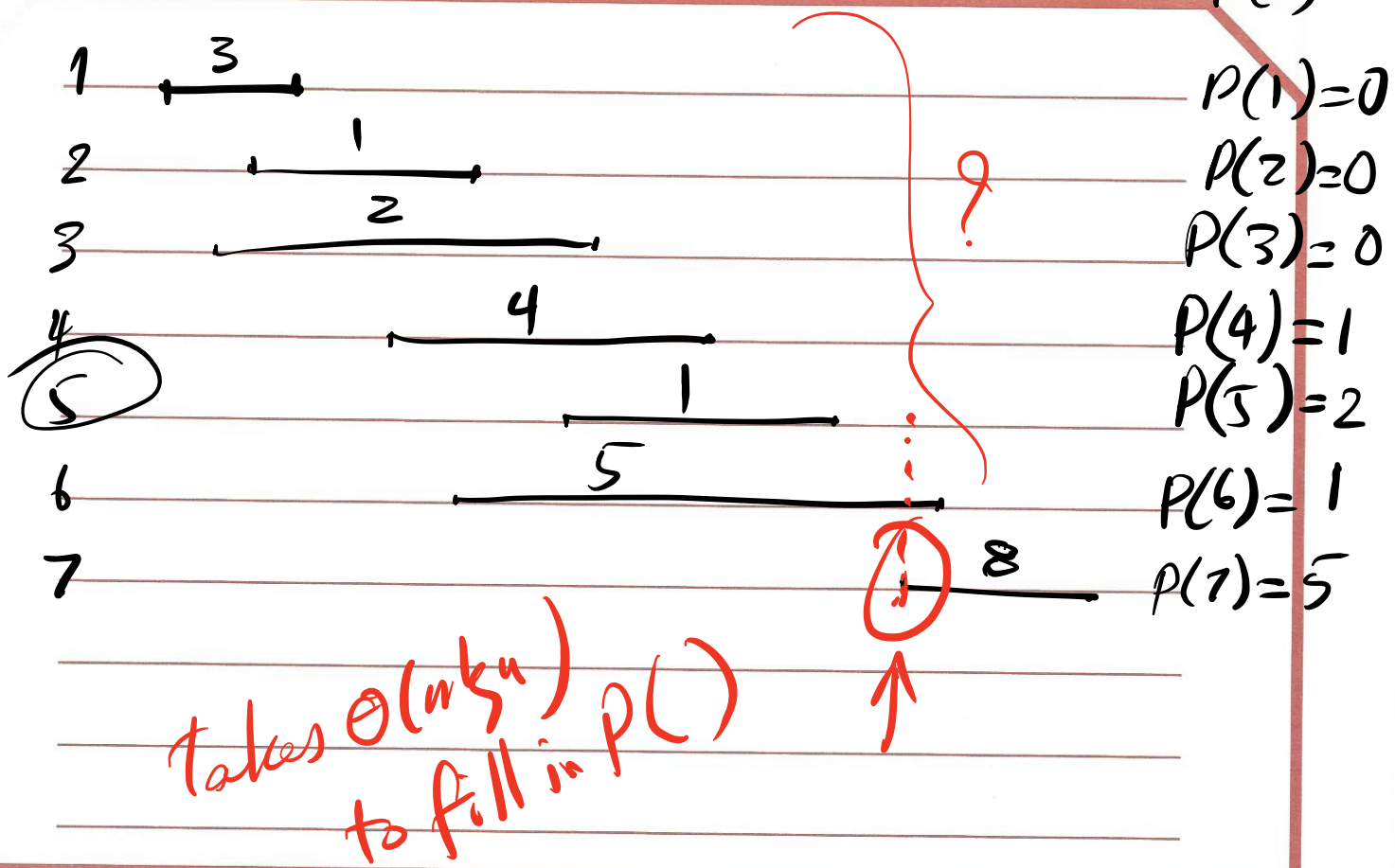
Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

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.



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

$$O_4 = \{1,4\} \qquad OPT(4) = 7$$

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

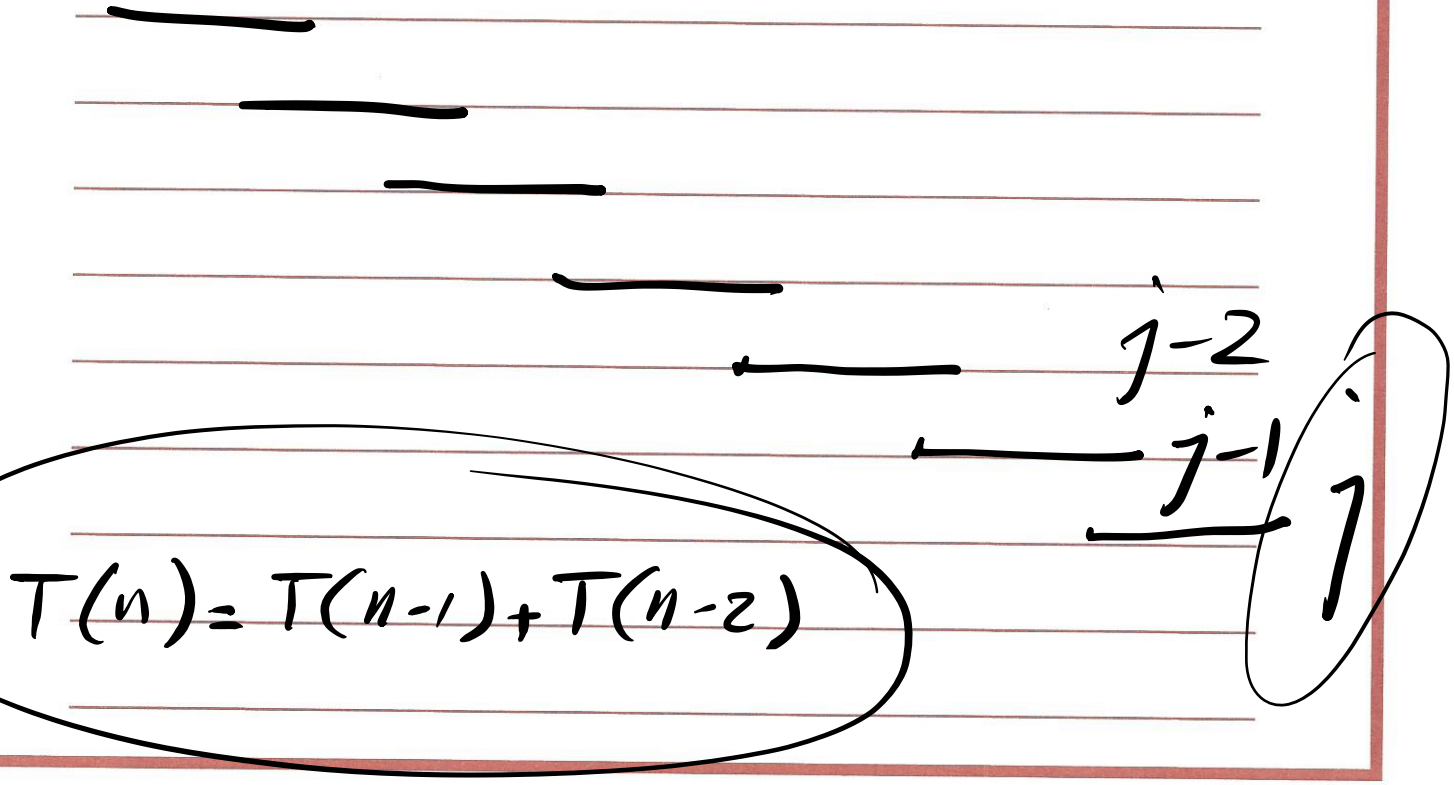
Case 2: $j \notin O_j \Rightarrow OPT(j) = OPT(j-1)$

Solution:

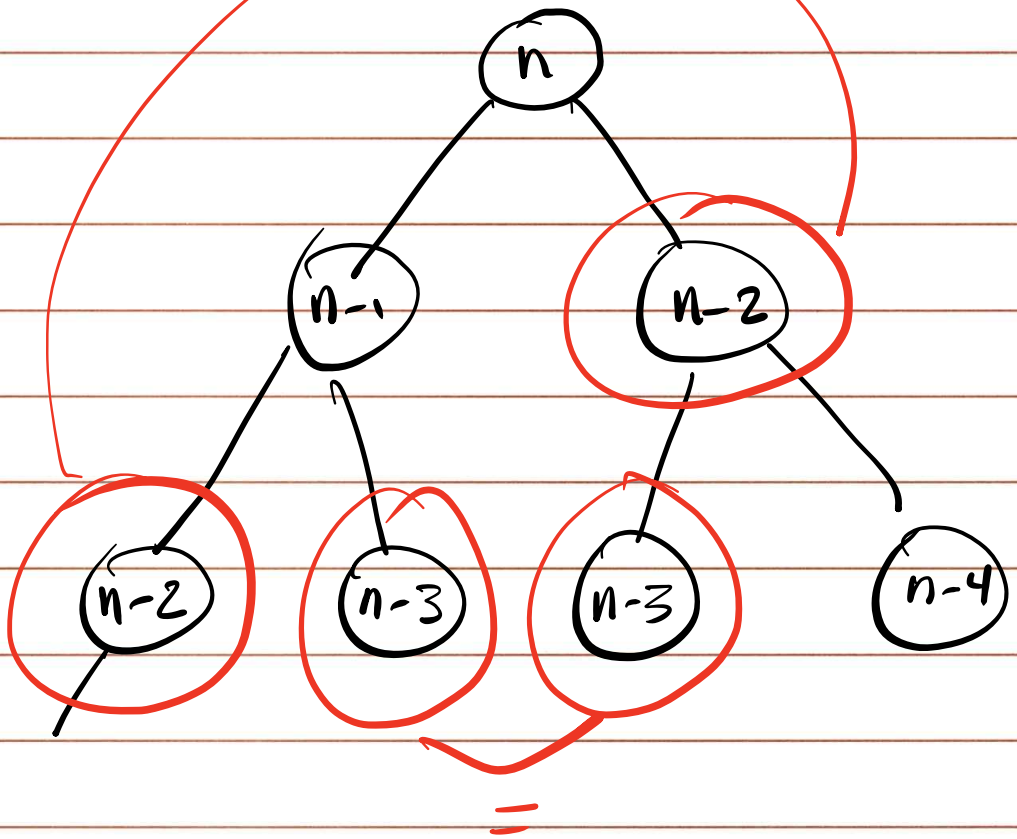
```
Compute-opt(j)
    if j = 0 then
        return 0
    else
        return Max(
            wj + Compute-opt(p(j)),
            Compute-opt(j-1)
    end if
```

worst case

Runs in Exp. time



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


n
n-1
n-2
n-2
n-3
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
```

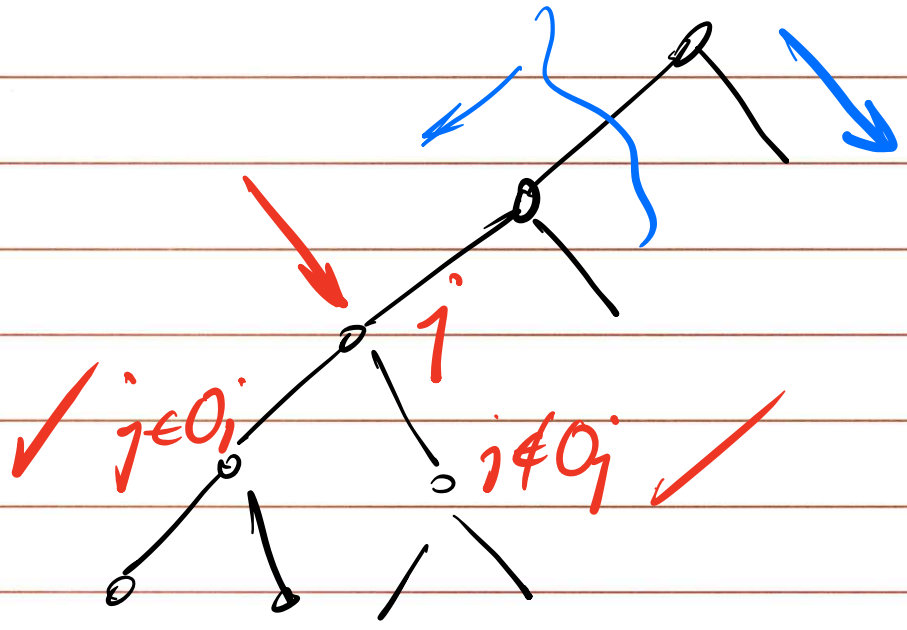
O(n)

Initial Sorting $\rightarrow \Theta(n \lg n)$

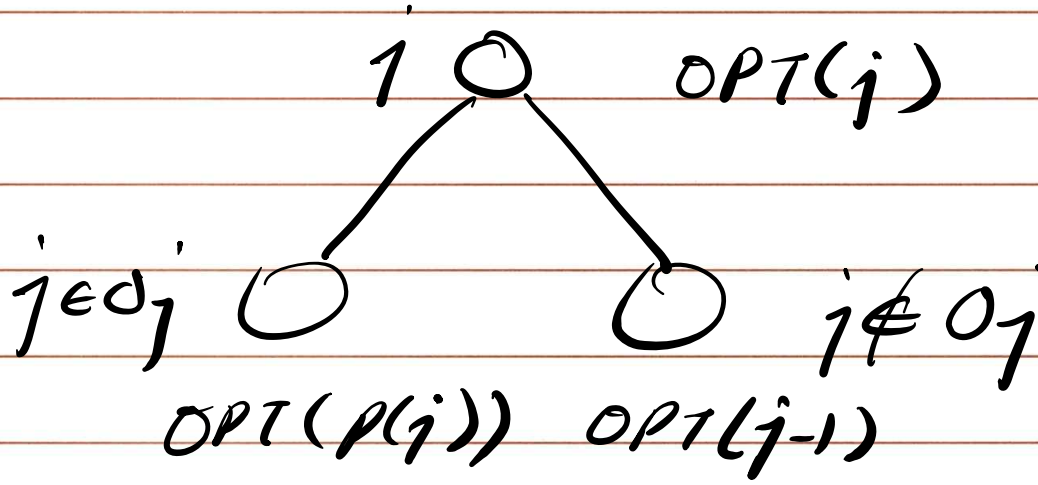
Build $P() \rightarrow \Theta(n \lg n)$

M-Compute-opt $\rightarrow \Theta(n)$

overall complexity $= \Theta(n \lg n)$



Compute an opt. Sol!

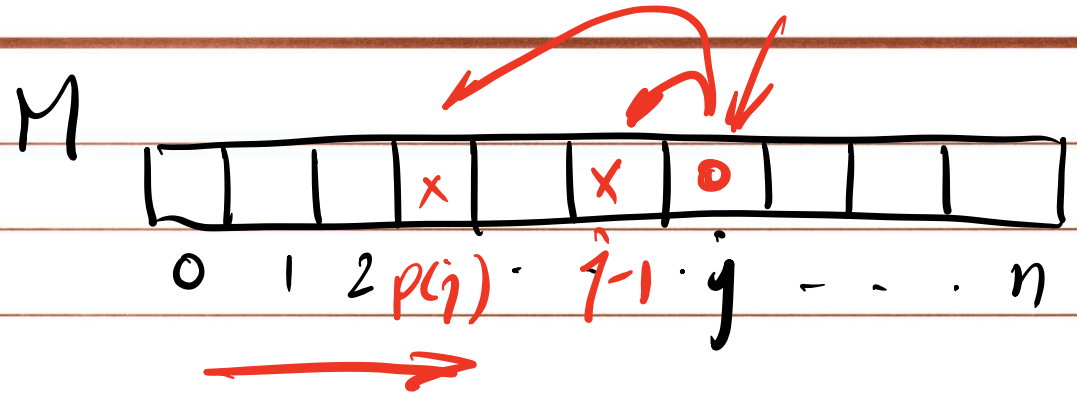


j belongs to Oj iff

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

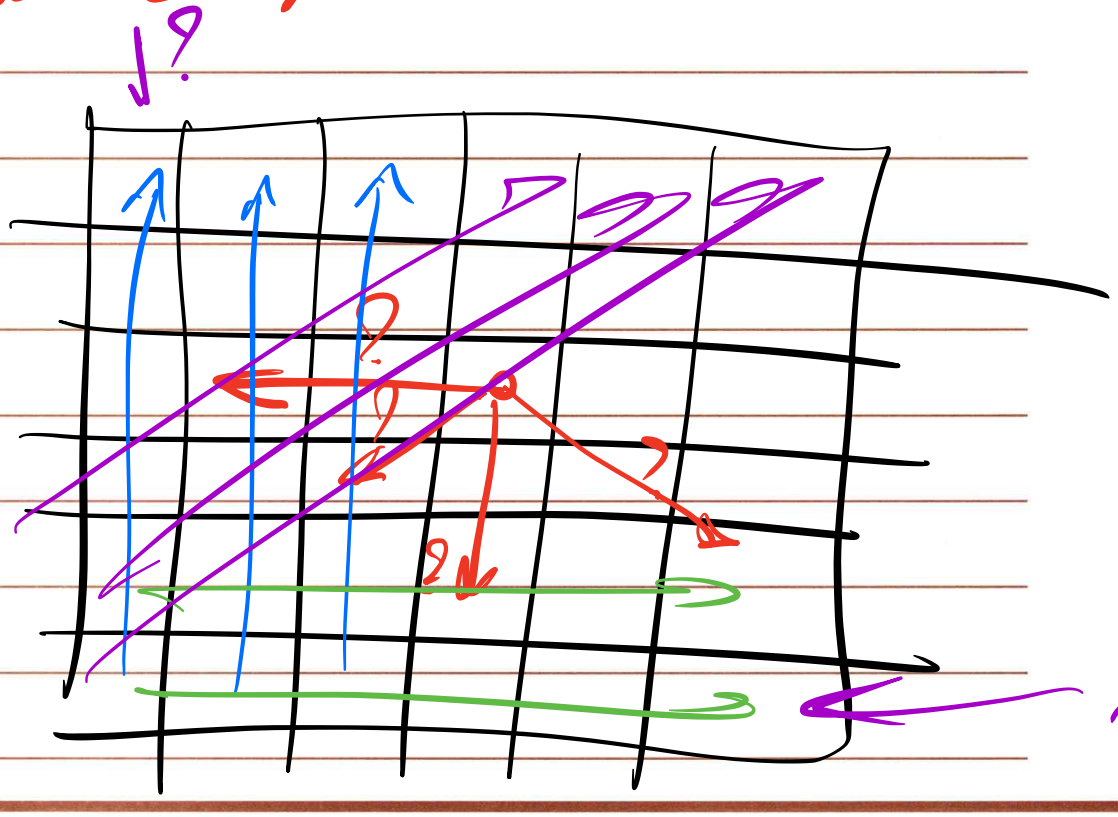
```
Find_Solution
    if j > 0 then
        if w_j + M[p(j)] ≥ M[j-1] then
            output j together w/ the results
              of Find_Solution ( p(j) )
        else
            output the results of
              Find_solution (j-1)
        end if
    end if
```

Takes $O(n)$

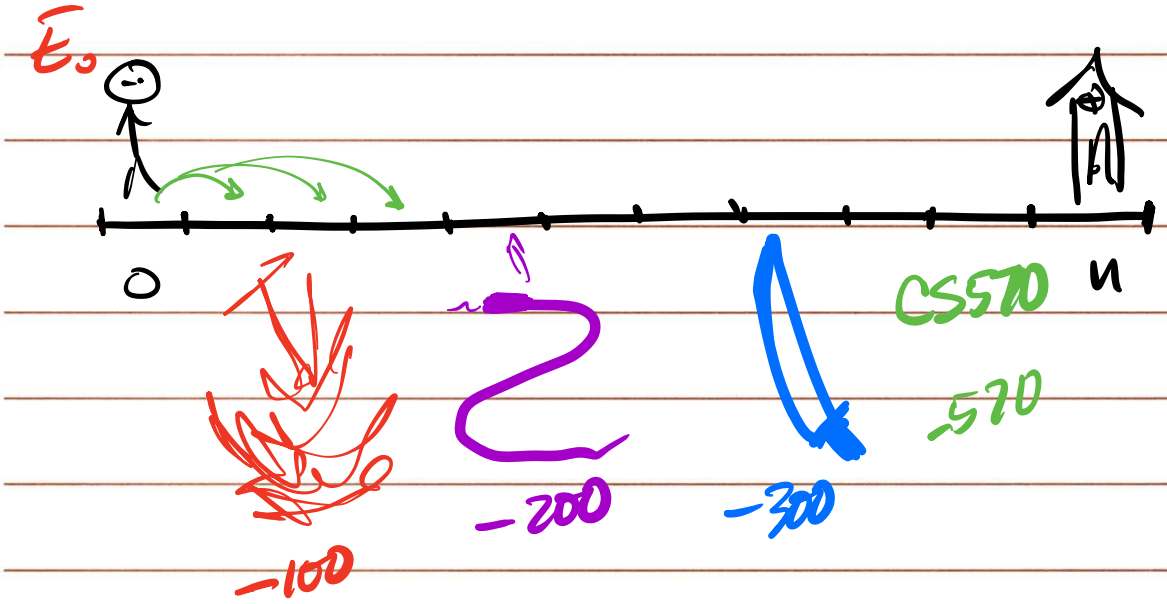



```
M[0] = 0
for i = 1 to n
    M[i] = Max( M[i-1],
                wi + M[P(i)] )
endfor
```

takes $\Theta(n)$



Videogame Problem

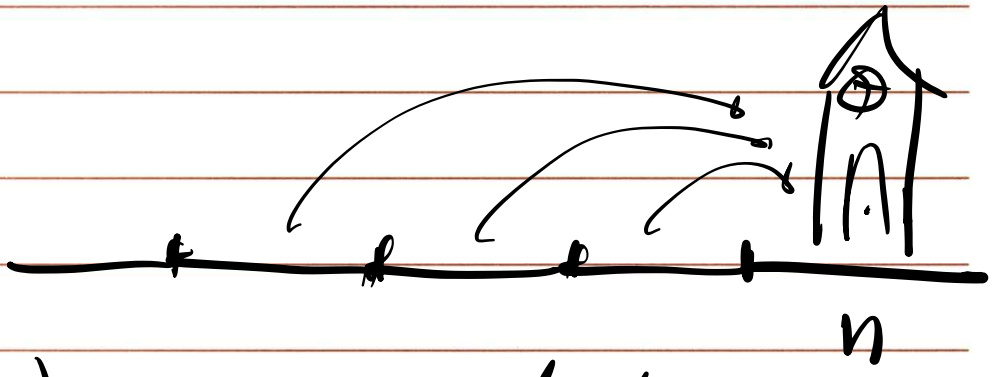


Choices: 1- walk into next stage → costs 50 units
2- jump over one stage 150 "

3- " " two stages 350 "

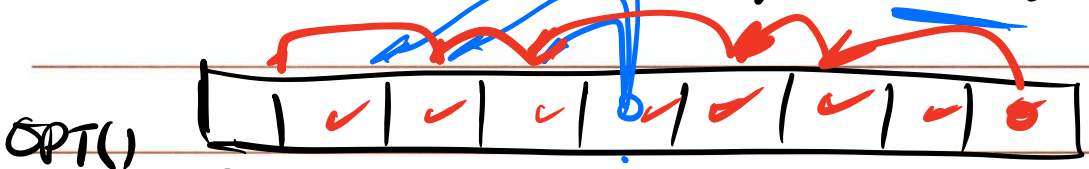
in general. we lose e_i units of energy when landing on stage i.

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



OPT(i) = Max amount of energy we could have when landing on stage i.

$$OPT(i) = \max\begin{cases} OPT(i-1) - 50 - e_i \\ OPT(i-2) - 150 - e_i \\ OPT(i+3) - 350 - e_i \end{cases}$$



bottom up

Max(..., ...)

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

for i = 3 to n

end for

takes $\Theta(n)$ bottom up

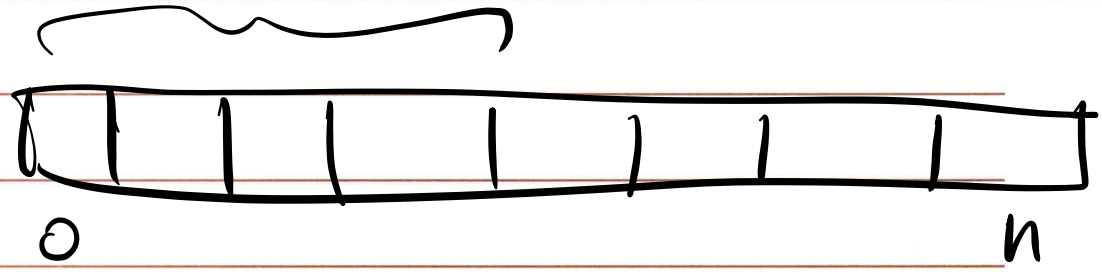
" " for top down

Coin Problem

Austrian Schillings $\left\{ \begin{array}{l} 1 \\ 5 \\ 10 \\ 20 \\ 25 \end{array} \right.$

$OPT(n)$ = Min. no. of coins needed to pay for n schillings

$$OPT(i) = \min(OPT(i-1)+1,\ OPT(i-5)+1,\ OPT(i-10)+1,\ OPT(i-20)+1,\ OPT(i-25)+1)$$



initialize $OPT(0 \ldots 24)$

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 $1..i$

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

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

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

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

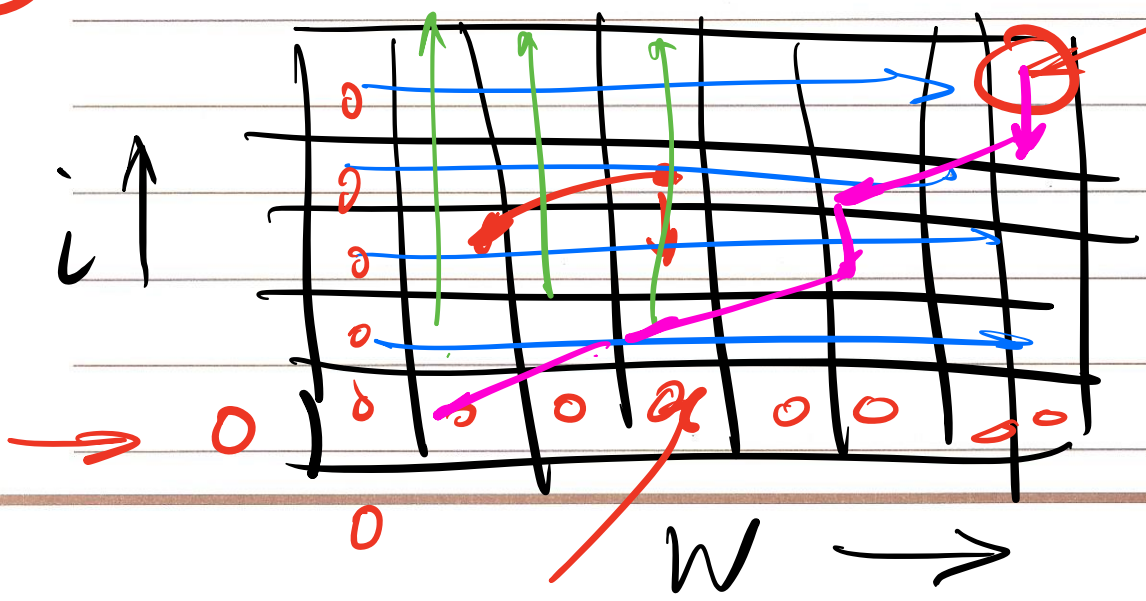
if $n \in O$, Then $OPT(n, w) = w_n +$

$OPT(n-1, w-w_n)$

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

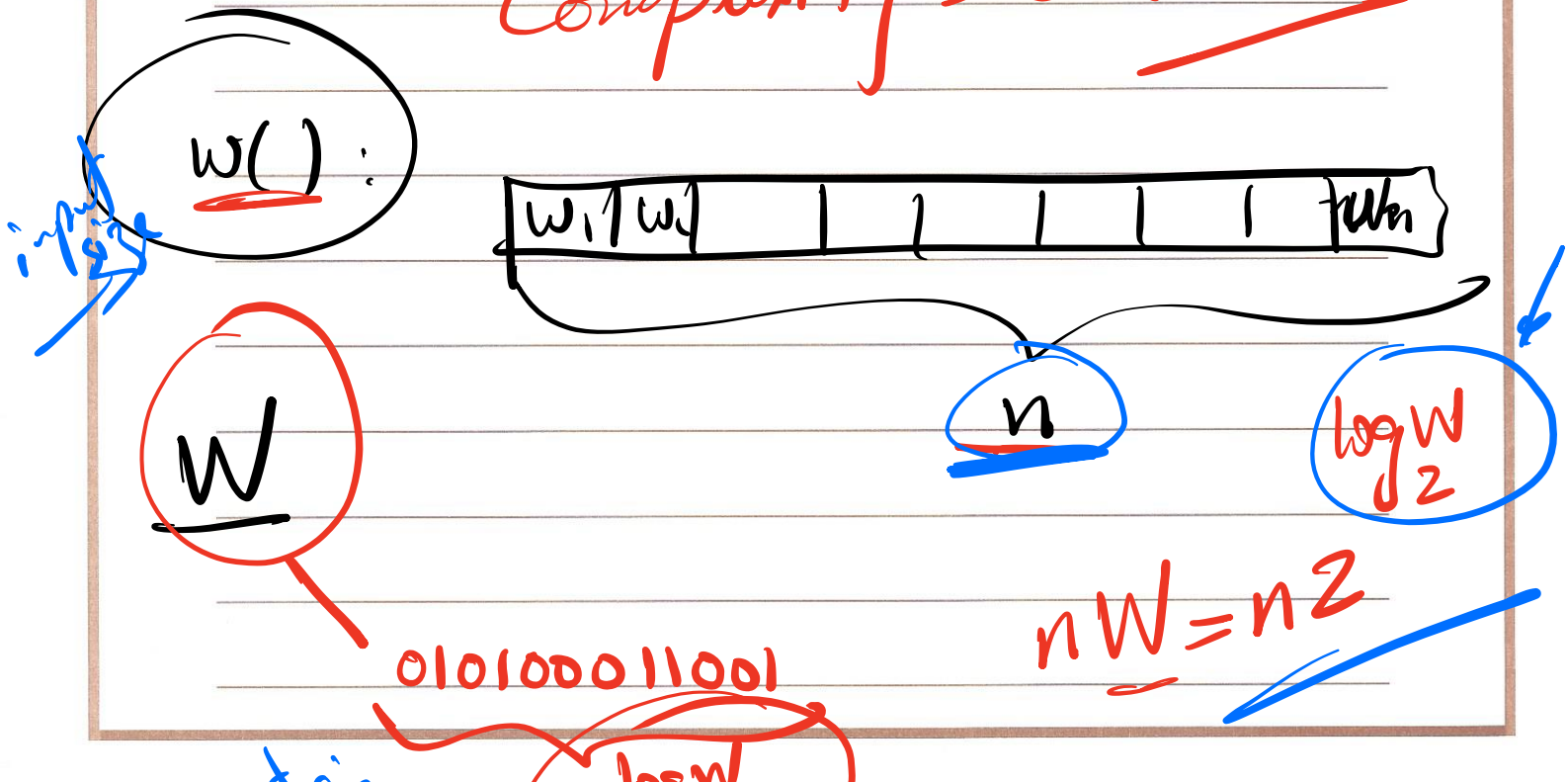
else, $OPT(i, w) = Max($ $OPT(i-1, w)$

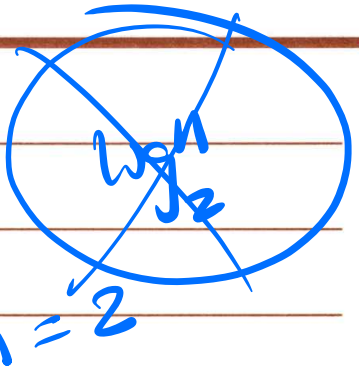
$w_i + OPT(i-1, w-w_i))$

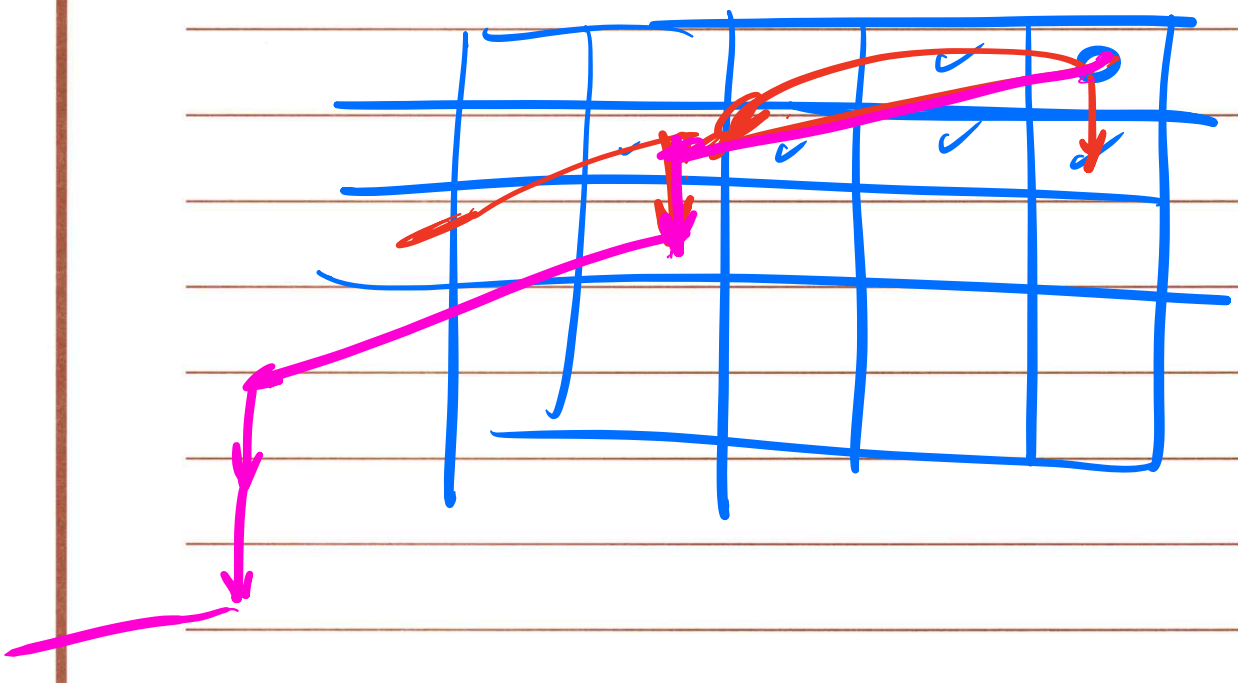


```
Subset-sum (n, w)
→ array M[0,w] = 0 for each w = 0 to W
  for i = 1 to n                          O(n)
    for w = 0 to W                        O(W)
      use recurrence formula (1)
      to compute M[i,w]
    end for
  end for
  Return M[n,w]
```

Complexity = $O(nW)$




n = 2



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).

For 0-1 knapsack:

if $i \notin O \rightarrow OPT(i, w) = OPT(i-1, w)$

if $i \in O \rightarrow OPT(i, w) = v_i + OPT(i-1, w - w_n)$

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.

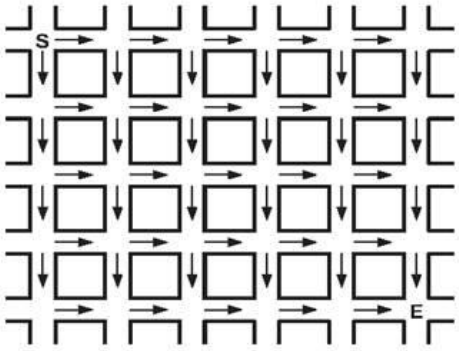


Figure A.

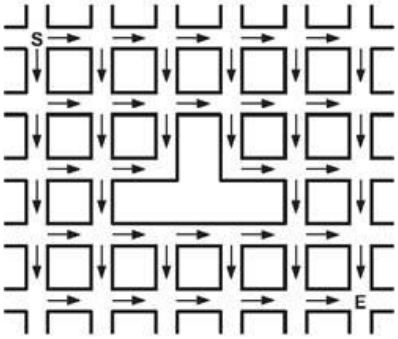
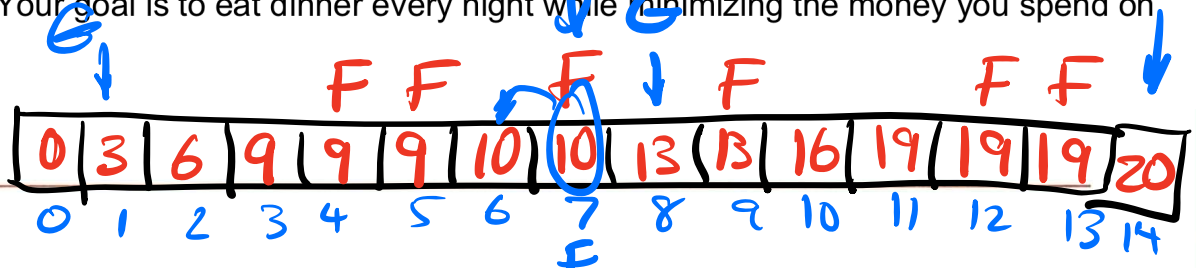


Figure B.

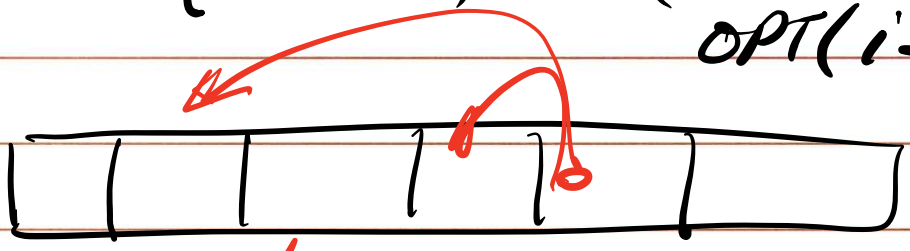
b) Repeat this process with Figure B; be wary of dead ends.

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 of dinner for days $1 \ldots i$

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



bottom up

takes $O(n)$

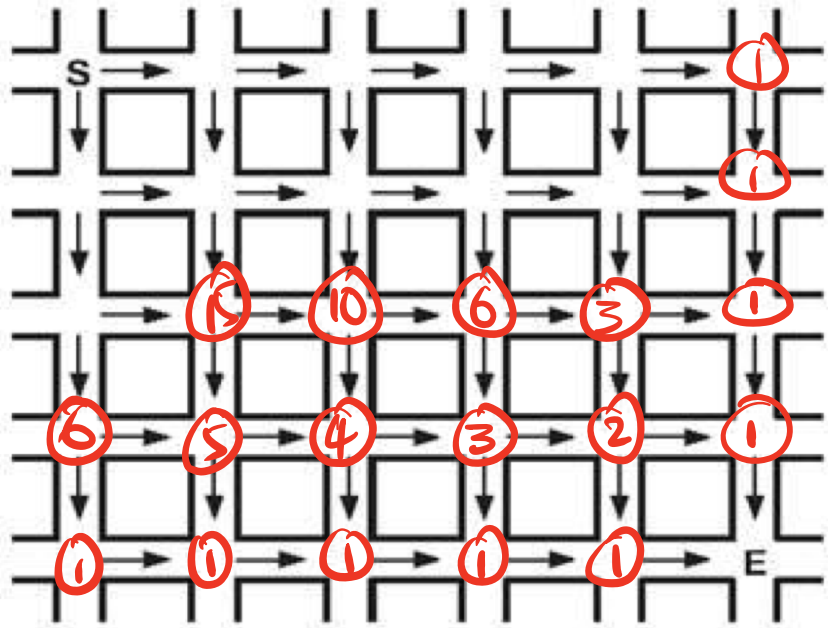


Figure A.

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

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

Takes $\Theta(nm)$

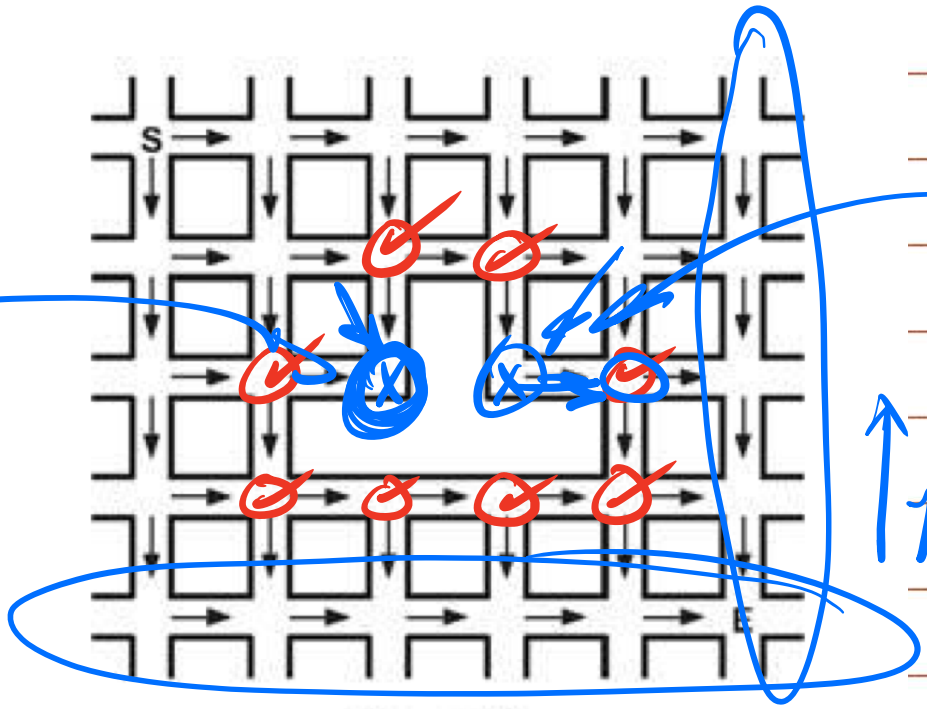


Figure B.

$\leftarrow i$

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

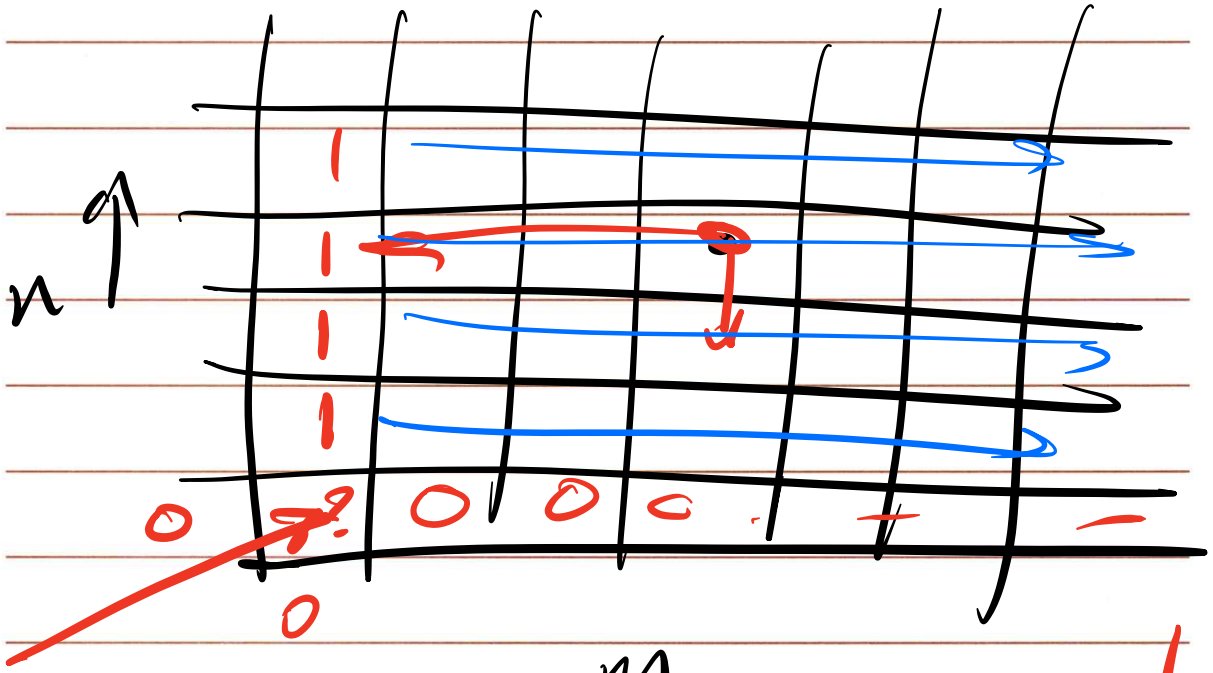
$i=3, j=2 \rightarrow OPT(i,j) = 0$

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.

$Count(m) = \sum_{i=1 \text{ to } n} Count(m - d_i)$

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

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



takes O(nm) ? not an eff. sol.

$a_1 = 1$

$a_2 = 2$

$n = 3$

{ 1, 1, 1
{ 1, 2

{ 1, 1, 1
{ 1, 2
{ 2, 1