General Approach to Solving optimization problems using Dynamic Programming

1. characterize the structure of an opt. solution

2. Recursively define the <u>value</u> of an opt. solution

3. Compute the <u>value</u> of an opt. solution in a bottom up fashion

4. Construct an opt. sol. from computed information

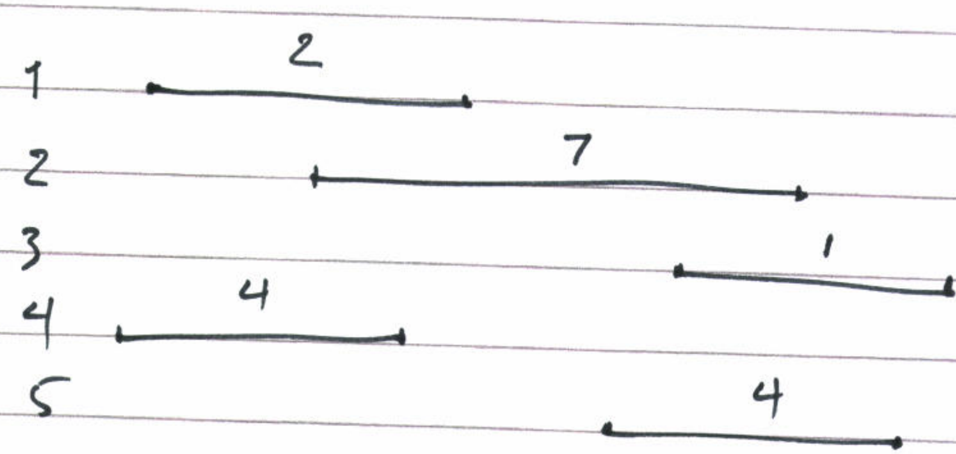
Problem Statement

- 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 $\sum_{i \in S} w_i$

$O(n)$ contains a ~~set~~ subset of jobs $1..n$ with maximum total weight

In general, $O(j)$ contains a subset of jobs $1..j$ with maximum total weight



either request i is part of an opt. sol. or it isn't.

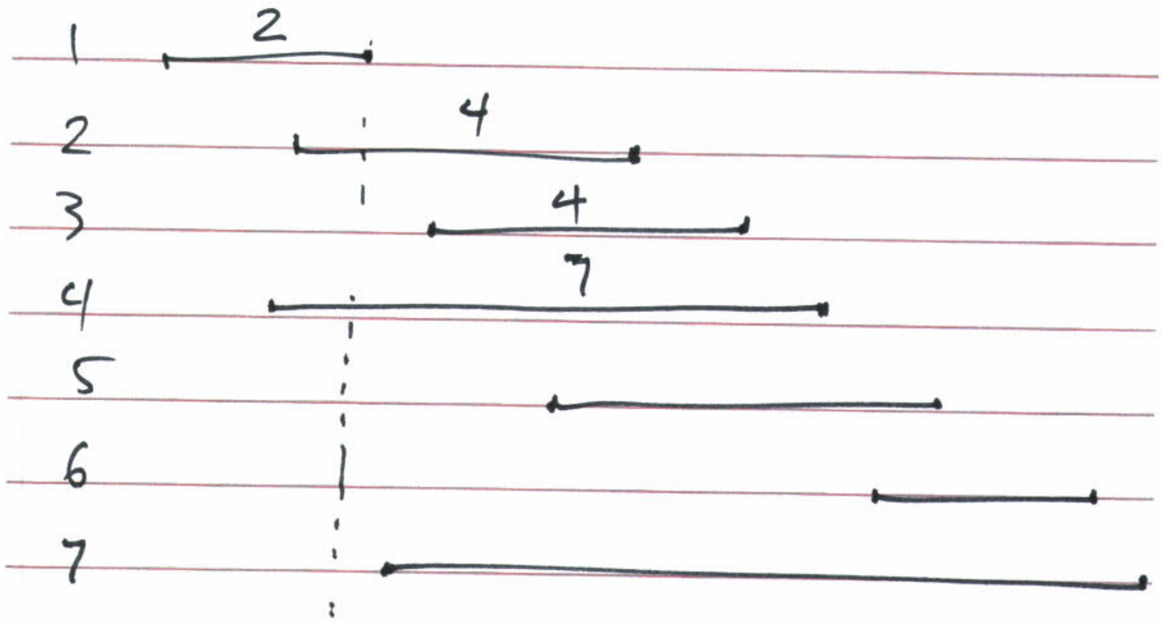
Case 1 - if it is, value of the opt. sol. = w_i + value of the opt. sol. for the subproblem that consists only of compatible requests with i

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

$p(1) = 0$
$p(2) = 0$
$p(3) = 1$
$p(4) = 0$
$p(5) = 1$
$p(6) = 4$
$p(7) = 1$

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.

$\cancel{O(3)}$ $O_3 = \{1, 3\}$ ← opt. sol.

$OPT(3) = \underline{6}$ ← value of opt. sol.

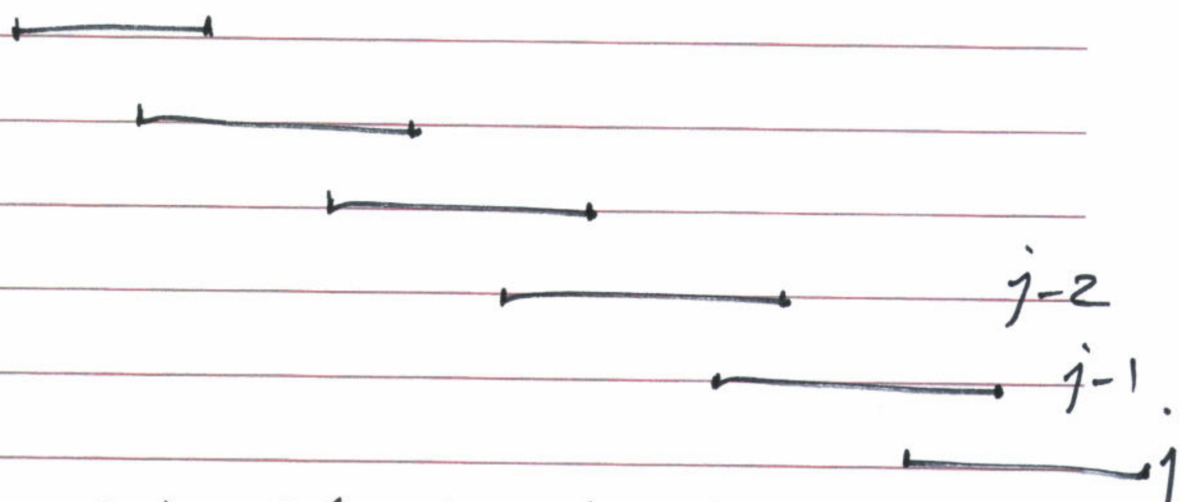
Case 1: $j \in O_j \Rightarrow OPT(j) = w_j + OPT(p(j))$

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

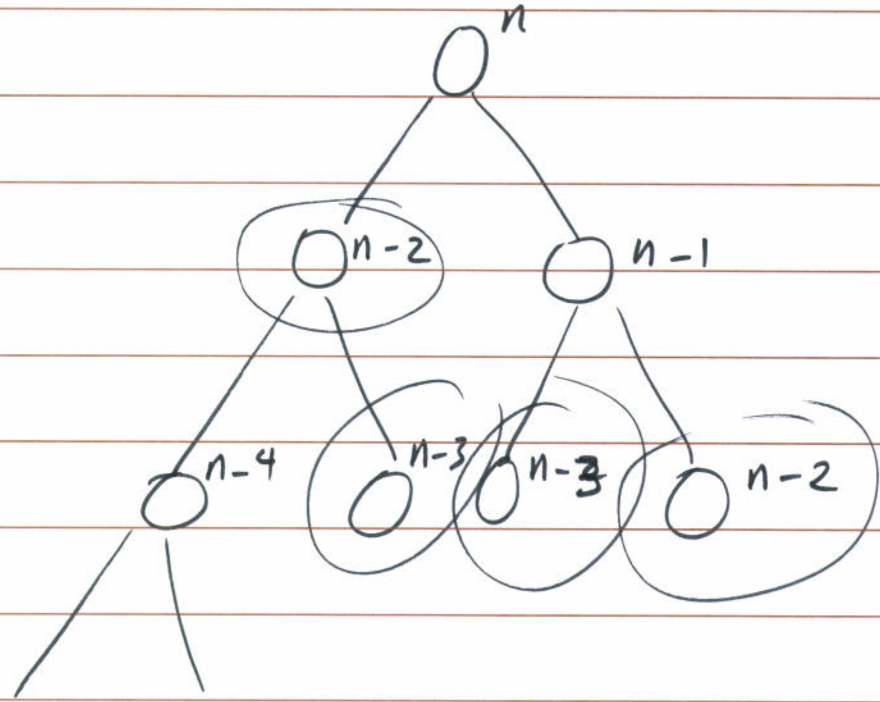
Solution:

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

exponets
exponential!
complexity.



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


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

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
```

takes O(n)!

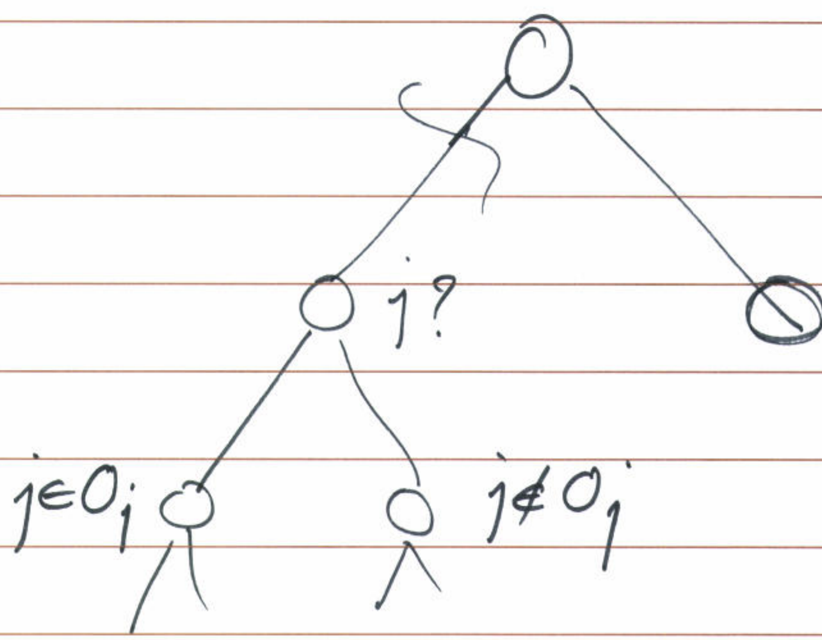
weighted interval scheduling

- Sorting $O(n \lg n)$

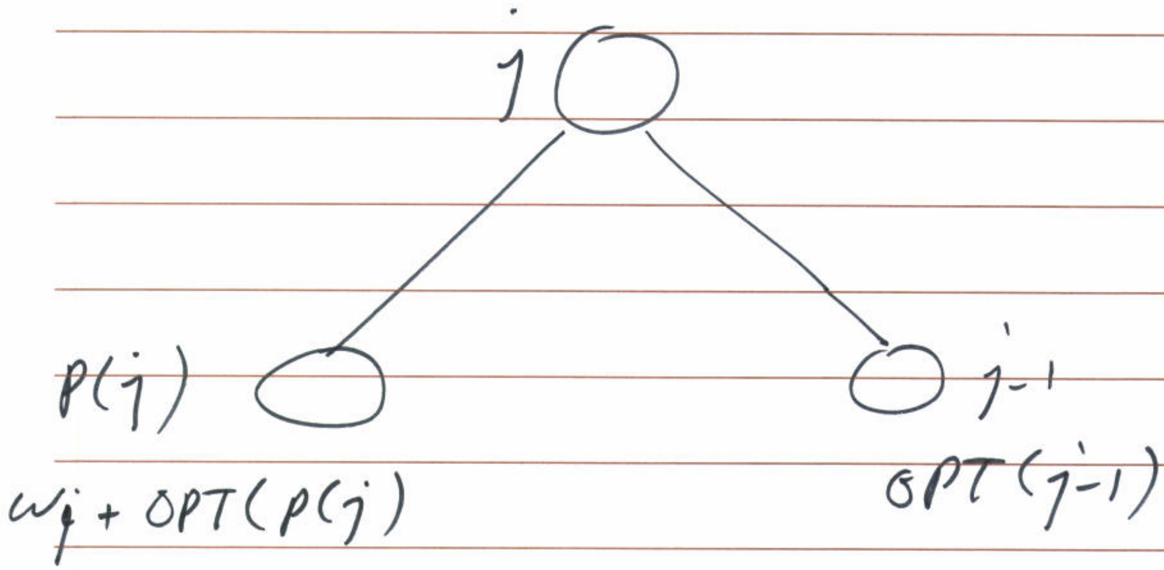
- Build P() $O(n \lg n)$

- M-compute-opt : $O(n)$

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



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 ($p(j)$)

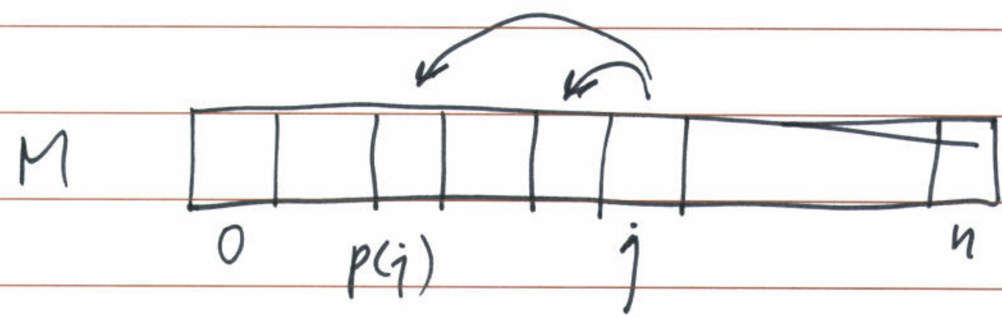
else

output the results of Find-solution ($j-1$)

endif

endif

Takes $O(n)$




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

$O(n)$

Schillings

1

5

10

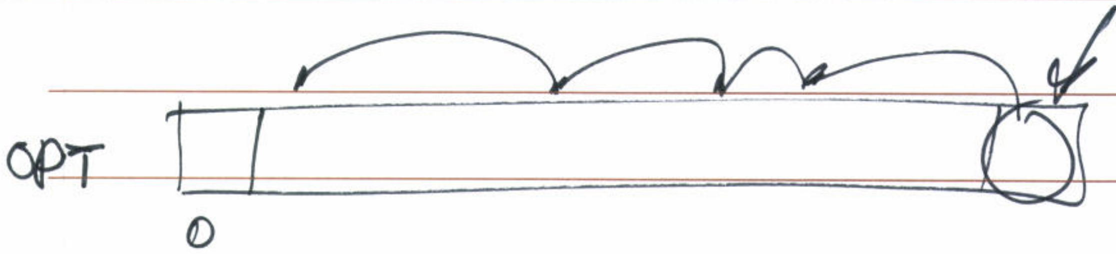
20

25

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

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

1

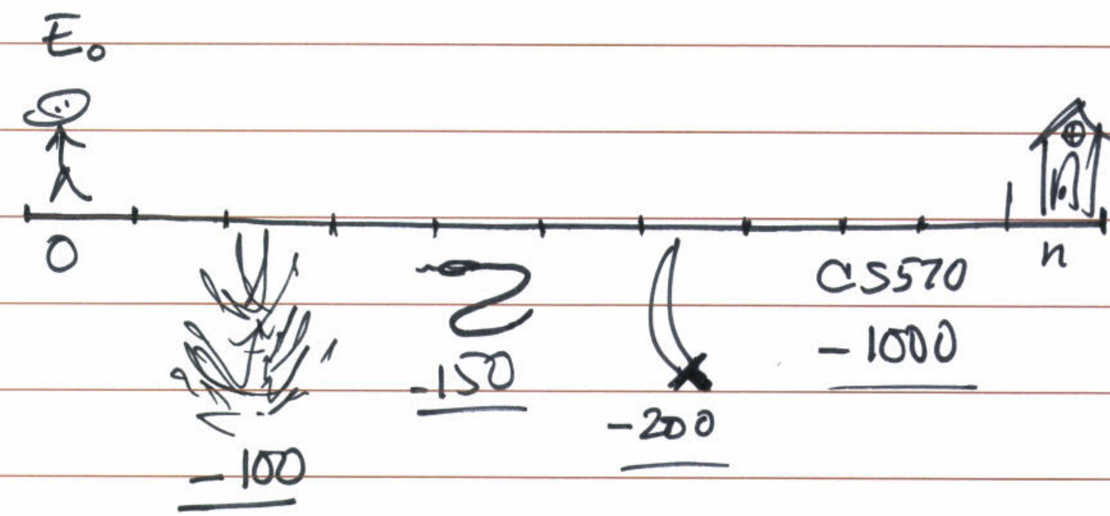


$OPT(0) = 0$, $OPT(1 .. 24)$

for $i = 25$ to n

use rec. formula (1)

endfor



Choice : 1- walk into the next square 50 units
2- jump over one square 150 "
3- " " two " (s) 350 "

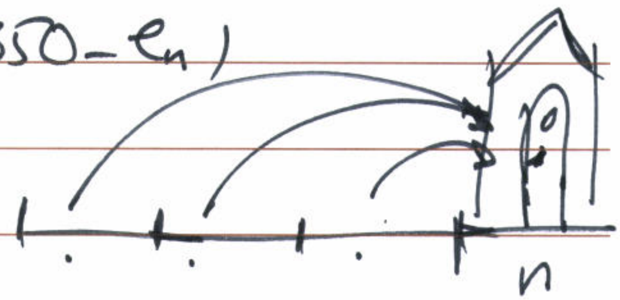
in general, we lose e_i units of energy when we land on square i

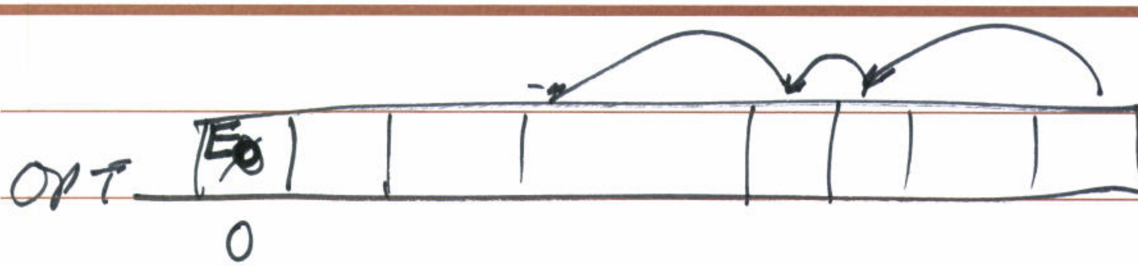
Question: How do you go home such that you lose the Min amount of energy?

Max.

OPT(n) = opt. level of energy when we reach square n.

$$OPT(n) = \max\big(OPT(n-1) - 50 - e_n,\; OPT(n-2) - 150 - e_n,\; OPT(n-3) - 350 - e_n\big)$$





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 $\underline{W}$

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 $\underline{w}$.

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

2

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 — W

    use recurrence formula (2)

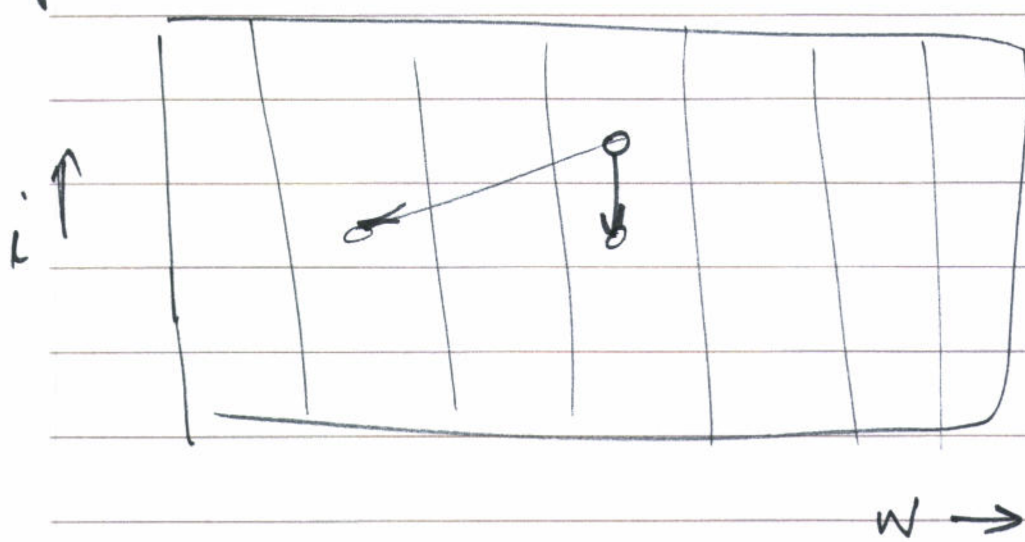
    to compute $M[i,w]$

  endfor

endfor

Return $M[n,W]$

pseudopolynomial run time → takes $O(\underline{n}\underline{W})$



Size of our input is : n & $\log W$

$O(n\,2^{\log W})$ ← exponential WRT size of input.

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).

0-1 knapsack

Recurrence formula for ~~Subset Sum~~

if $w < w_i$ then $OPT(i, w) = OPT(i-1, w)$

otherwise $OPT(i, w) = \mathrm{Max}(OPT(i-1, w),$

$\overset{v_i}{\cancel{w_i}} + OPT(i-1, w-w_i))$

Imagine starting with a given decimal number n, we repeatedly chop off a digit from one end or the other until only one digit is left.

Def. The square-depth of n is the maximum no. of perfect squares you could observe among all such sequences

ex.

$32492 \to 3249 \to 324 \to 24 \to 4$

or, $32492 \to 3249 \to 249 \to 49 \to 9$

Problem statement

Describe an efficient algorithm to compute the square-depth of a given no.

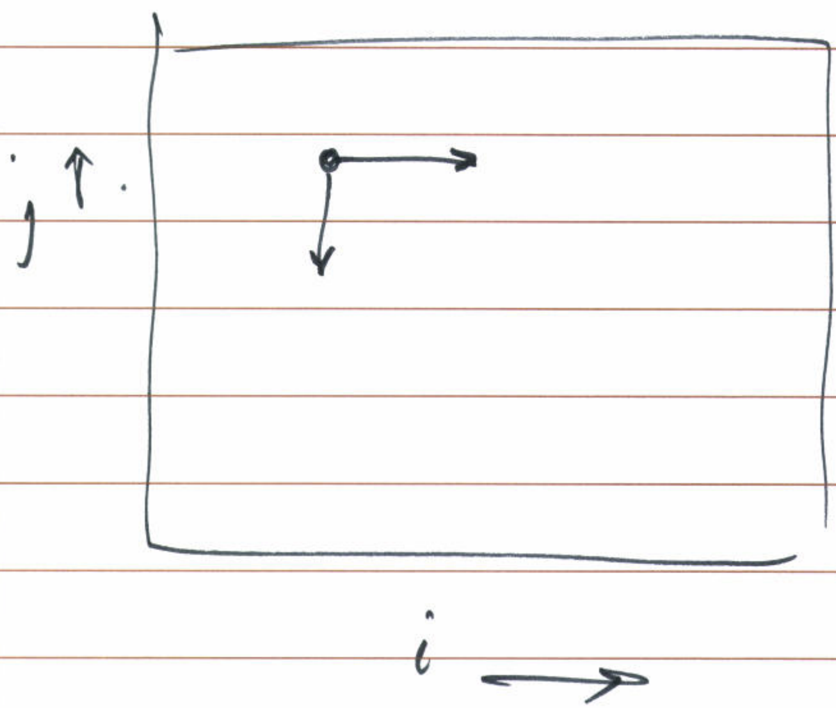
Say $n = d_1 \ldots d_m$

$$n_{ij} = d_i \ldots d_j$$

$SD[i,j]$ = Square depth of n_{ij}

$$SD[i,j] = \mathrm{Max}\left(SD[i+1,j],\ SD[i,j-1]\right) + \text{IS\_SQUARE}(n_{ij})$$

IS_SQUARE returns 1 if it is a square & $\underline{0}$ otherwise



How do we fill out the rest of the array.