CS 341: ALGORITHMS

Lecture 8: dynamic programming II

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

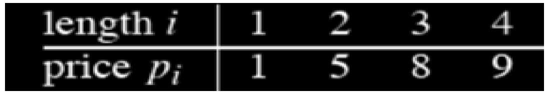
trevor.brown@uwaterloo.ca

ROD CUTTING

A "REAL" DYNAMIC PROGRAMMING EXAMPLE

- Input:
  - n: length of rod
  - $p_1, \dots, p_n$: p_i = price of a rod of length i
- Output:
  - Max **income** possible by cutting the rod of length n into any number of **integer** pieces (maybe **no** cuts)

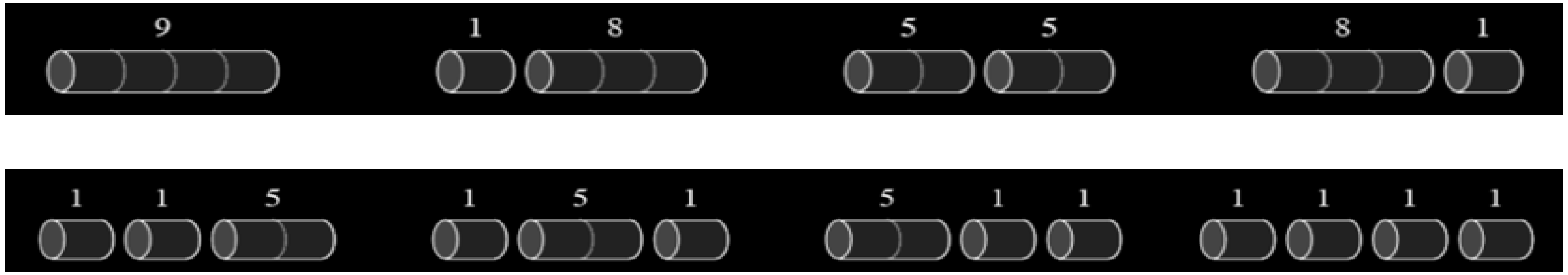
$n = 4$



length i	1	2	3	4
price p_i	1	5	8	9

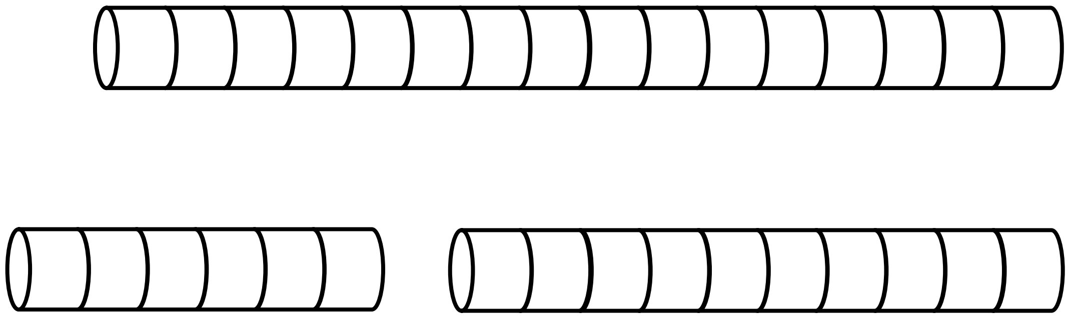
All ways of cutting a rod of length 4

Example output: 10



DYNAMIC PROGRAMMING APPROACH

- High level idea **(can just think recursively to start)**
  - Given a rod of length n
  - Either make no cuts,
    or make a cut and **recurse** on the remaining parts



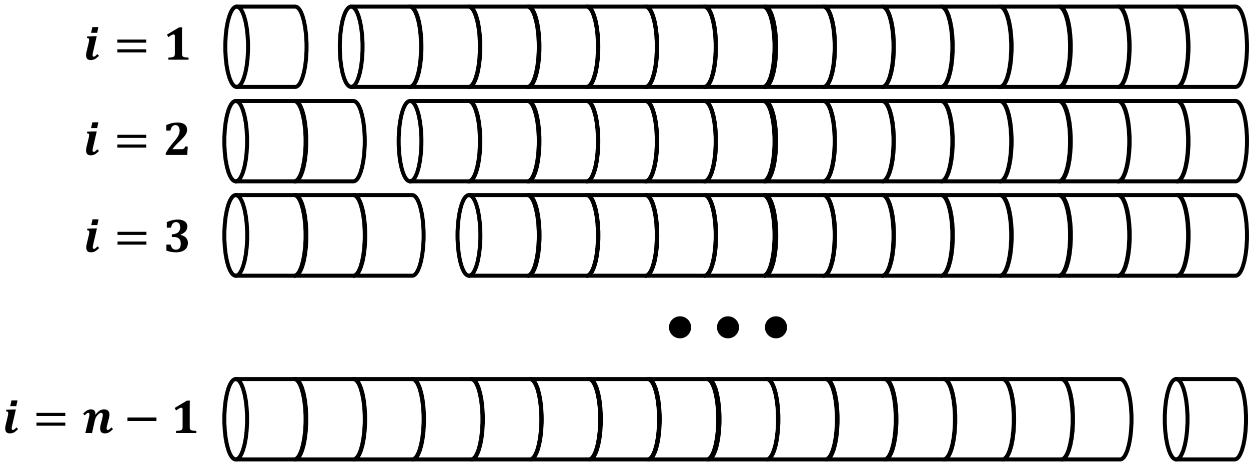
Income p_n

$\text{Income(Left)} + \text{Income(Right)}$

  - **Where** should we cut?

DYNAMIC PROGRAMMING APPROACH

- Try **all ways** of making that cut
  - I.e., try a cut at positions $1, 2, \ldots, n-1$
  - In each case, recurse on two rods $[0, i]$ and $[i, n]$
- Take the max income over **all possibilities** (each i / no cut)



Optimal substructure: Max income from two rods w/sizes i and $n - i$

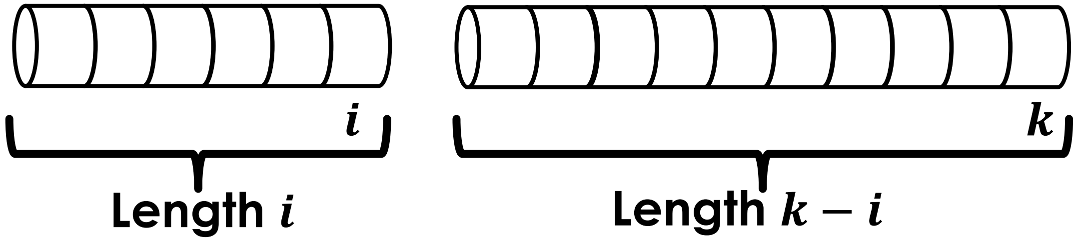
… is max income we can get from the rod size i

+ max income we can get from the rod size $n - i$

RECURRENCE RELATION

Critical step! Must define what M(k) means, semantically!

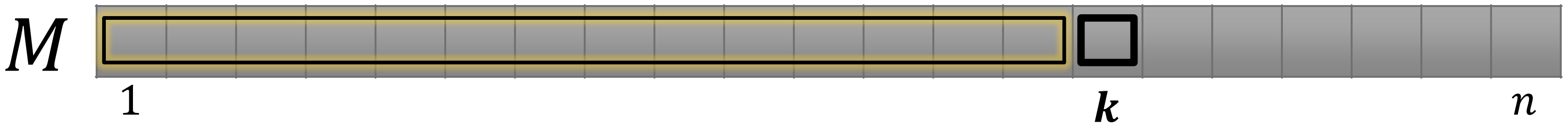
- Define $M(k)$ = maximum income for rod of length k
- If we do **not** cut the rod, max income is $\boldsymbol{p_k}$
- If we **do** cut a rod **at** $\boldsymbol{i}$



  - max income is $M(i) + M(k-i)$
  - Want to maximize this **over all** $\boldsymbol{i}$
    - $\boldsymbol{max_i\{M(i) + M(k-i)\}}$ (for $0 < i < k$)
- $\boldsymbol{M(k) = \max\left\{p_k, \max_{1 \le i \le k-1}\{M(i) + M(k-i)\}\right\}}$

COMPUTING SOLUTIONS BOTTOM-UP

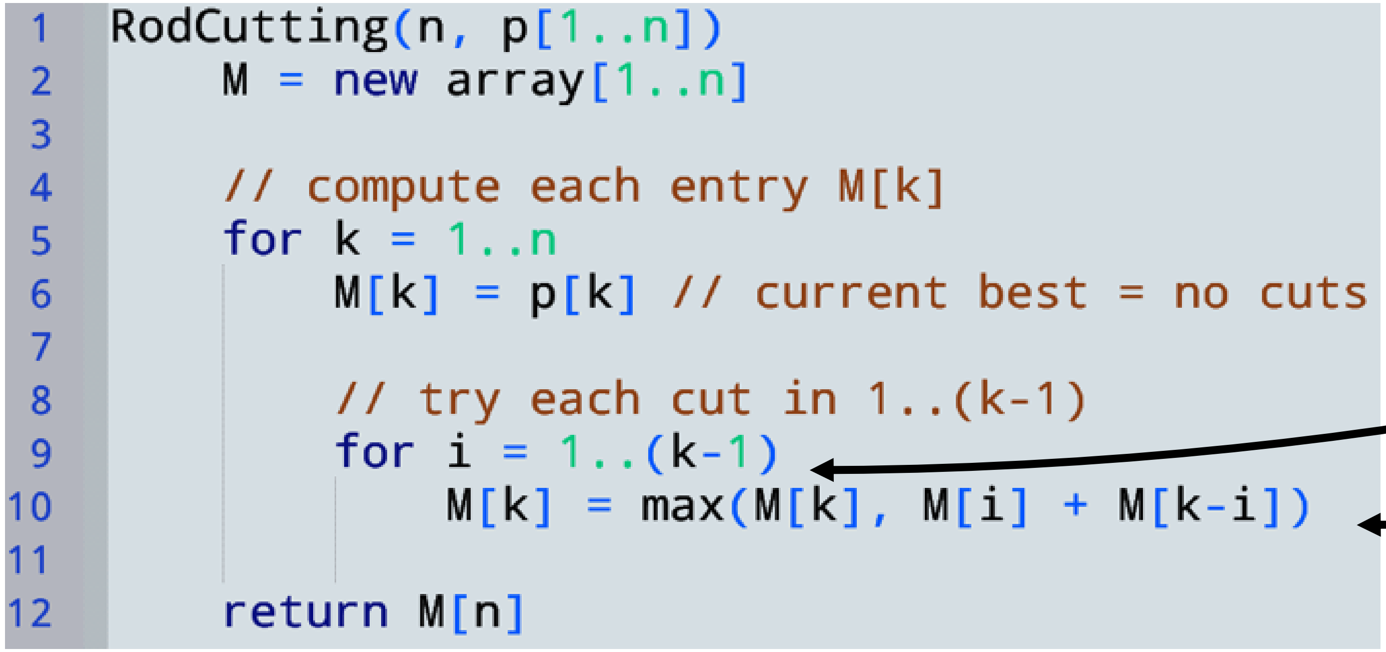
- **Recurrence:** $\boldsymbol{M(k) = max\{p_k, \max_{1\le i\le k-1}\{M(i) + M(k-i)\}\}}$
- Compute **table** of solutions: $\mathrm{M}[1..n]$



- Dependencies: **entry $\boldsymbol{k}$** depends on
    - $M[i]$ → $M[\mathbf{1}..(\boldsymbol{k}-\mathbf{1})]$
    - $M[k-i]$ → $M[\mathbf{1}..(\boldsymbol{k}-\mathbf{1})]$
  - All of these dependencies are $< k$
- So we can fill in the table entries in order $1..n$

Recall, semantically, $M(k)$ = maximum income for rod of length k

Recurrence: $\boldsymbol{M(k) = max\{p_k, \max_{1\le i\le k-1}\{M(i) + M(k-i)\}\}}$



```
RodCutting(n, p[1..n])
    M = new array[1..n]

    // compute each entry M[k]
    for k = 1..n
        M[k] = p[k] // current best = no cuts

        // try each cut in 1..(k-1)
        for i = 1..(k-1)
            M[k] = max(M[k], M[i] + M[k-i])

    return M[n]
```

Time complexity (unit cost)?

$\Theta(n^2)$

MISCELLANEOUS TIPS

- Building a table of results bottom-up
  is what makes an algorithm DP
- There is a similar concept called **memoization**
  - But, for the purposes of this course,
    we want to see bottom-up table filling!
- Base cases are **critical**
  - They often completely
    determine the answer
  - Try setting f[0]=f[1]=0 in FibDP…

DP SOLUTION TO **0-1 KNAPSACK**

Problem: output maximum value one can get from taking **≤ 7kg**, out of these **four items**.

Suppose the optimal solution **O does not include this**

Then with the **O** must achieve the best possible value **using only items 1-3**.

Subproblem: output max value for **≤ 7kg** out of these **three items**

This is a **smaller subproblem:** reduced # of items

Goal: create **recurrence relation** to describe optimal solution in terms of subproblems

Let P[i, m] = maximum profit using any subset of the items **1..i**, with weight limit **m**

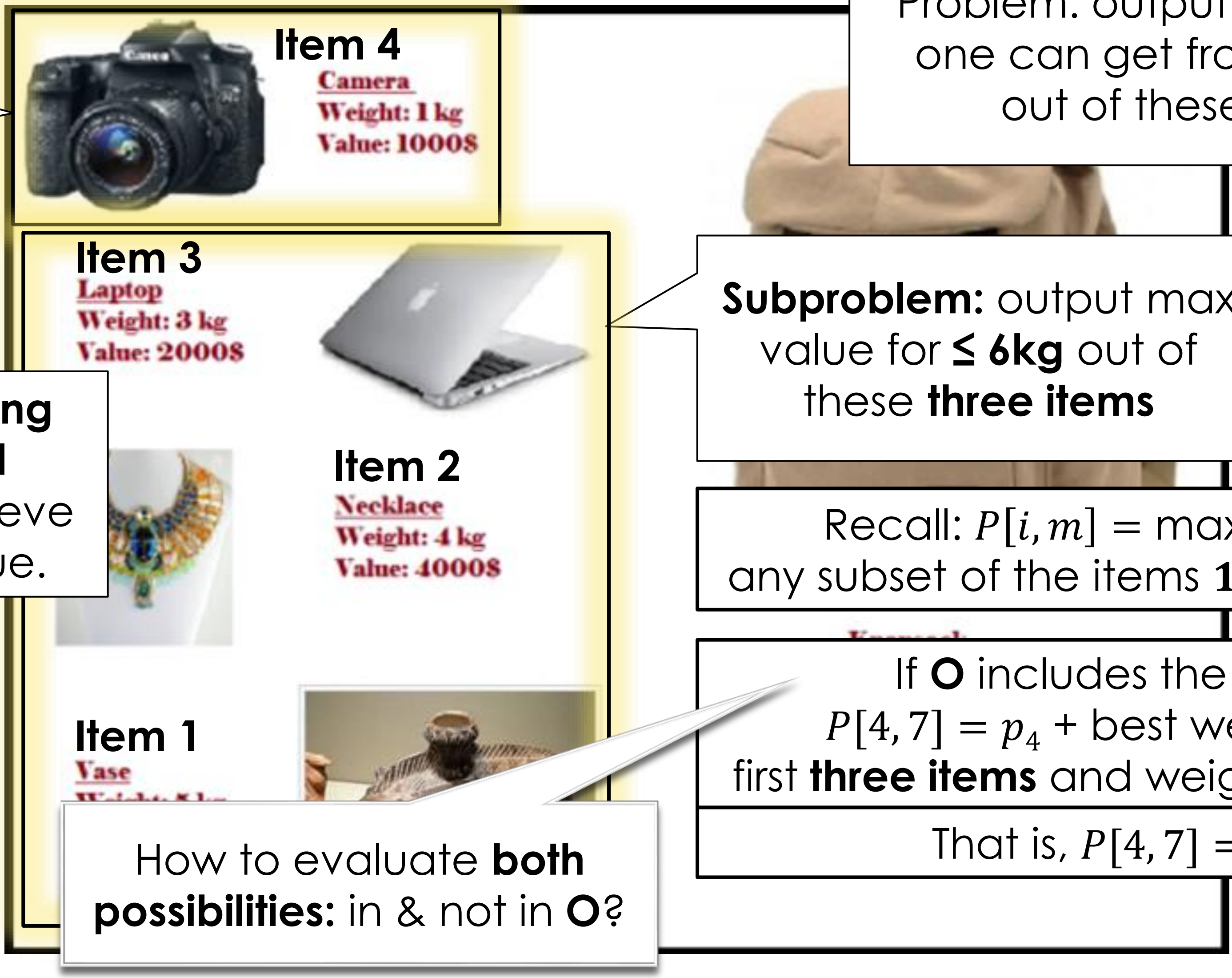
Note: P[n, M] (= P[4, 7]) is the **optimal profit**

If **O does not include** the camera, then P[4, 7] = best we can do with the **first three** items and weight limit **7kg**

That is, P[4, 7] = P[3, 7]

What if the camera <u>IS</u> included in O?

Problem: output maximum value one can get from taking **≤ 7kg**, out of these **four items**.



Suppose the optimal solution **O includes this**

Then with the **remaining 7kg – w$_4$ = 6kg, and items 1-3, O** must achieve the best possible value.

Subproblem: output max value for **≤ 6kg** out of these **three items**

This is a **smaller subproblem:** reduced weight **and** # of items

Recall: $P[i,m]$ = maximum profit using any subset of the items $\mathbf{1..}\boldsymbol{i}$, with weight limit $\boldsymbol{m}$

If **O** includes the camera, then $P[4,7] = p_4$ + best we can do with the first **three items** and weight limit **7kg – w$_4$** = 6kg

That is, $P[4,7] = p_4 + P[3,6]$

How to evaluate **both possibilities:** in & not in **O**?

Recall: $P[i, m]$ = maximum profit using any subset of the items $\mathbf{1..}\boldsymbol{i}$, with weight limit $\boldsymbol{m}$

		In general:
If **O does not include** the camera, then $P[4,7]$ = best we can do with the **first three** items and weight limit **7kg**	$P[4,7] = P[3,7]$	$P[i,m] = P[i-1,m]$
If **O includes** the camera, then $P[4,7] = \boldsymbol{p_4}$ + best we can do with the **first three** items and weight limit **7kg – $\mathbf{w_4}$** = 6kg	$P[4,7] = p_4 + P[3, 7 - w_4]$	$P[i,m] = p_{\boldsymbol{i}} + P[i-1, m - w_{\boldsymbol{i}}]$
Try both and take the better result! **(How?)**	$P[4,7] = \mathbf{max}\{$ $P[3,7],$ $p_4 + P[3, 7 - w_4]\}$	$P[i,m] = \mathbf{max}\{$ $P[i-1,m],$ $p_{\boldsymbol{i}} + P[i-1, m - w_{\boldsymbol{i}}]\}$

Note that $\max\{P[i-1,m], p_i + P[i-1, m - w_i]\}$ **is only valid if** $i \geq 2$ and $m \geq w_i$

What to do when $i = 1$ or $m < w_i$? These are **special cases**.

General case: $i \geq 2$ and $m \geq w_i$

Since $m \geq w_i$, we **can carry item i.**
$P[i, m] = \max\{P[i-1, m], p_i + P[i-1, m-w_i]\}$

Special case 1: $i \geq 2$ and $m < w_i$

Since $m < w_i$, we **cannot carry item i.**
So, $P[i, m] = P[i-1, m]$.

Special case 2: $i = 1$ and $m \geq w_i$

Since $i \leq 1$, we **can only use item 1.**
Since $m \geq w_i$, we **can carry item 1.**
So, $P[i, m] = p_i$.

Special case 3: $i = 1$ and $m < w_i$

Since $i \leq 1$, we **can only use item 1.**
Since $m < w_i$, we **cannot carry item 1.**
So, $P[i, m] = 0$.

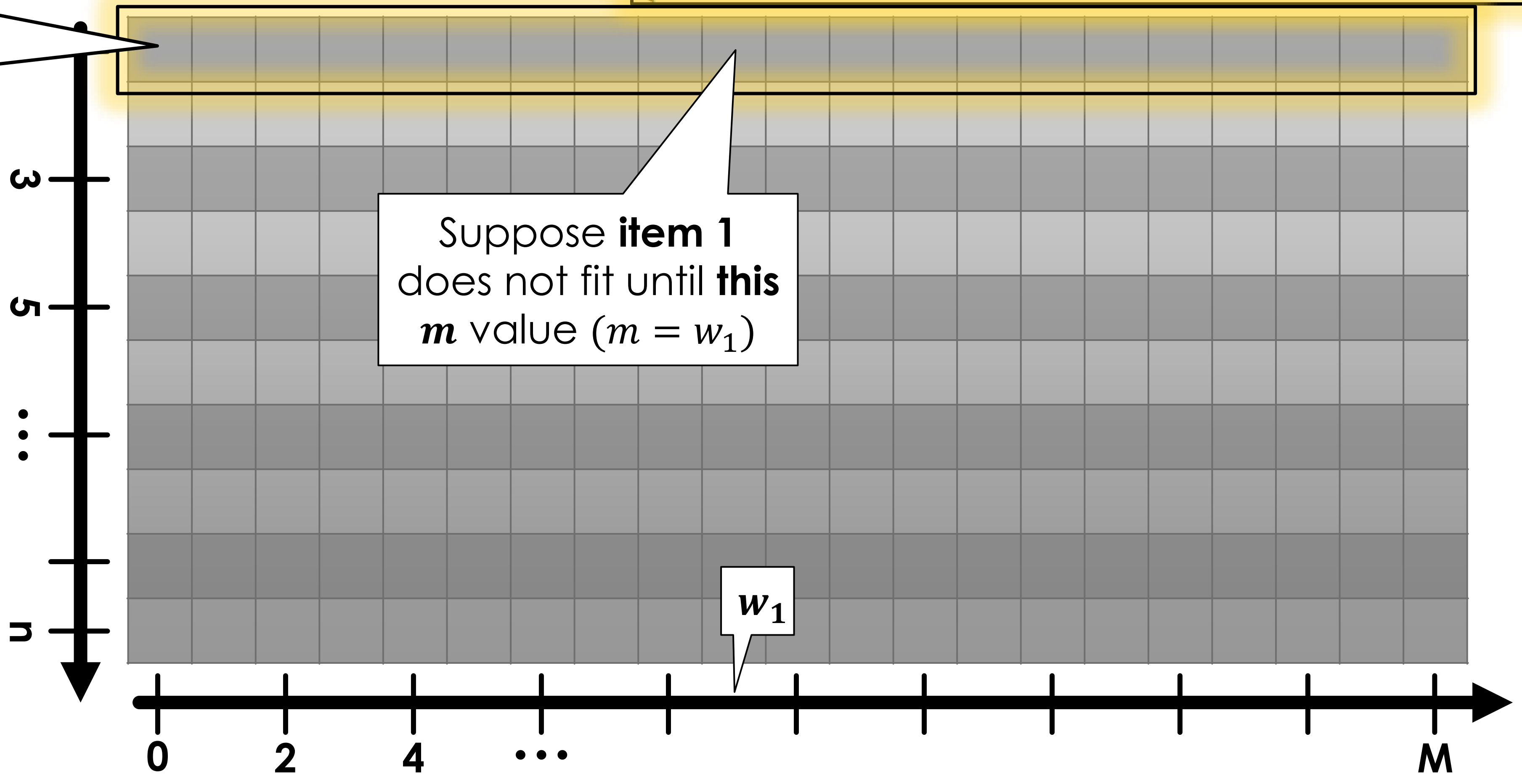
Recurrence Relation:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1, m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$

FILLING THE ARRAY:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$

No data dependencies on any other array cells.



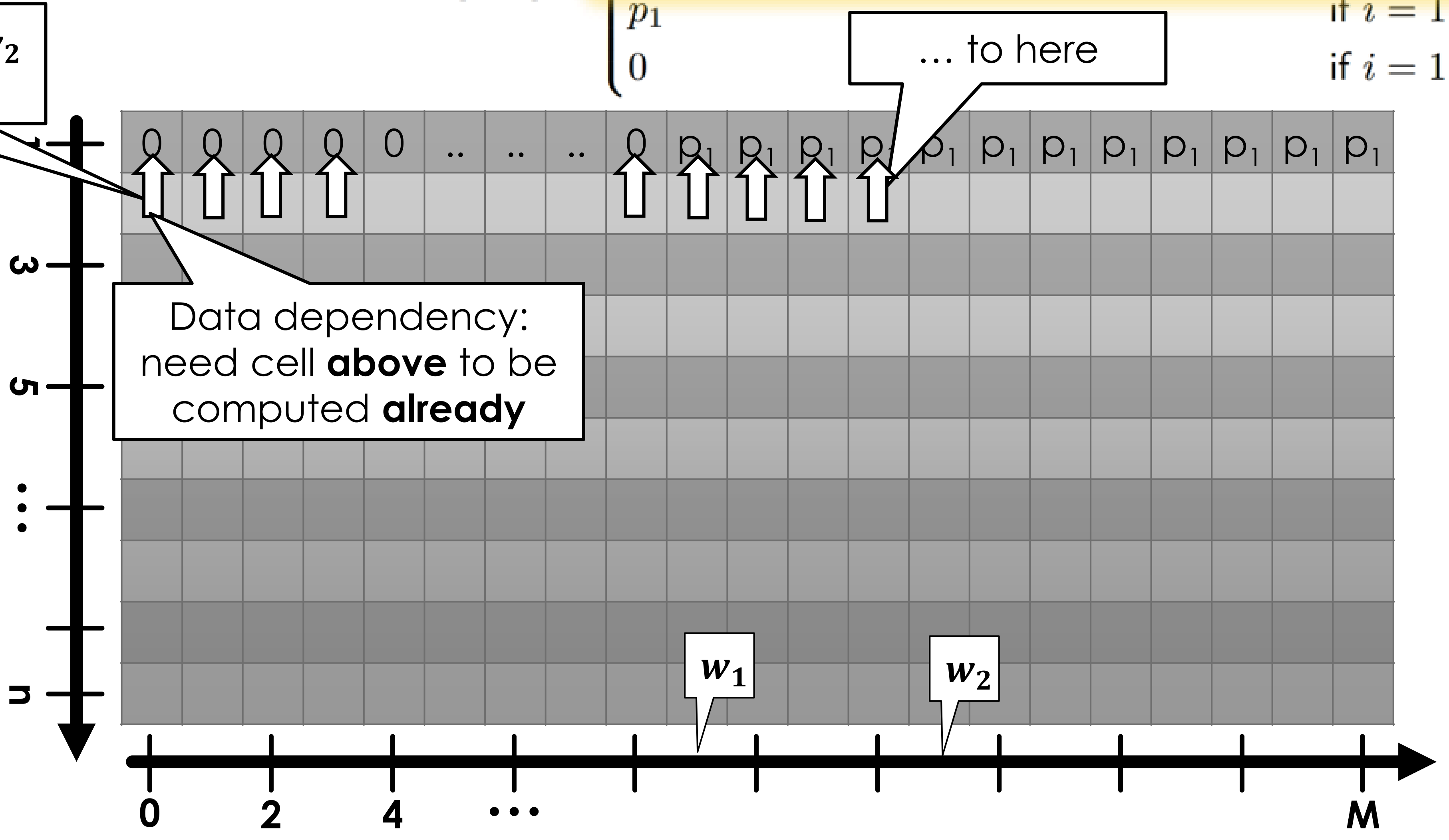
Suppose **item 1** does not fit until **this** $\boldsymbol{m}$ value ($m = w_1$)

$\boldsymbol{i}$-axis (can use items in $1..i$)

$\boldsymbol{m}$-axis (remaining weight limit)

FILLING THE ARRAY:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ \boxed{P[i-1,m] \qquad \text{if } i \geq 2,\ m < w_i} & \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$



Suppose $m < w_2$ from here

… to here

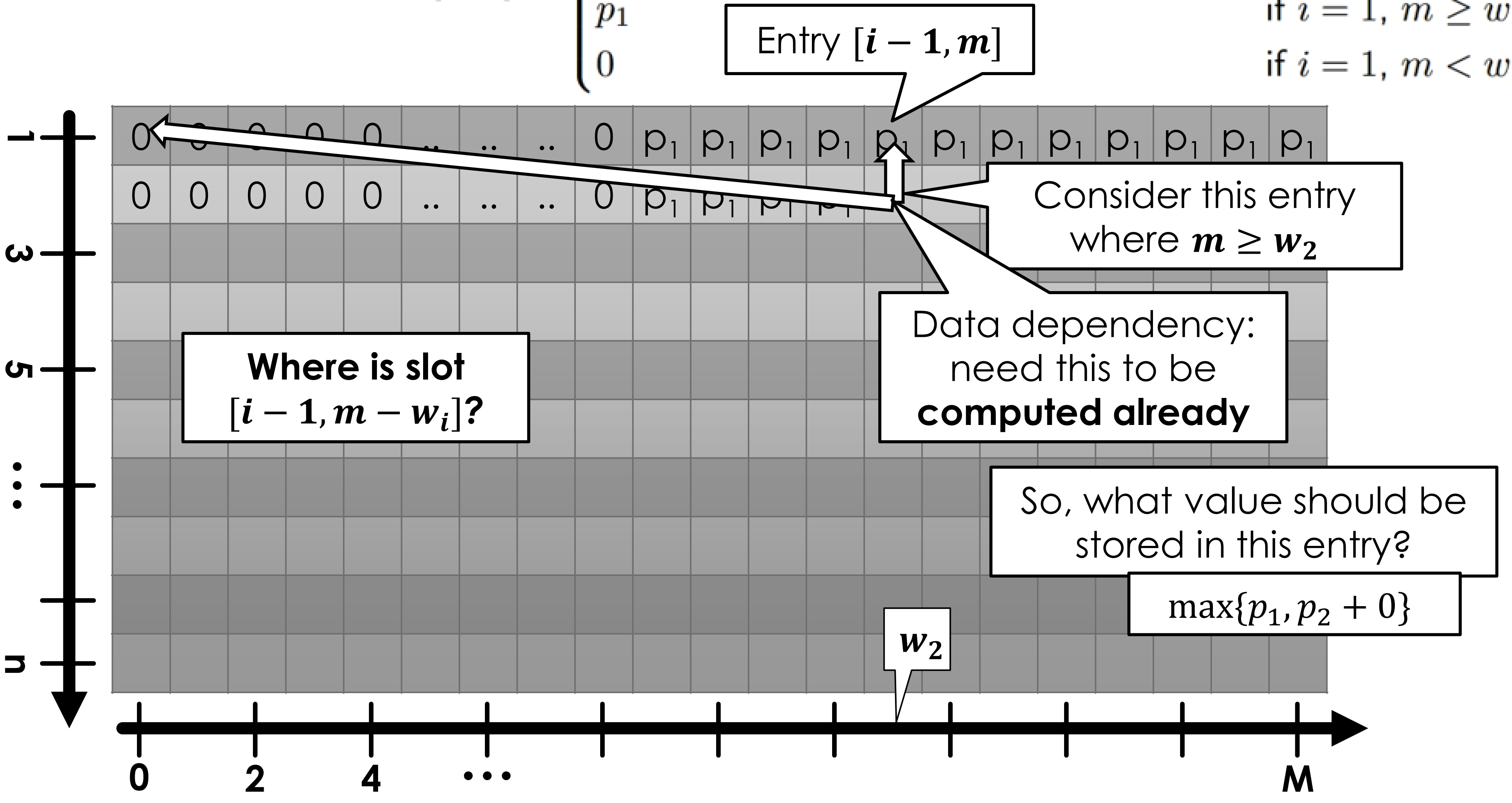
Data dependency: need cell **above** to be computed **already**

i-axis (can use items in $1..i$)

m-axis (remaining weight limit)

FILLING THE ARRAY:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$



Entry $[i-1,m]$

Consider this entry where $m \geq w_2$

Data dependency: need this to be **computed already**

Where is slot $[i-1, m-w_i]$?

So, what value should be stored in this entry?

$\max\{p_1, p_2 + 0\}$

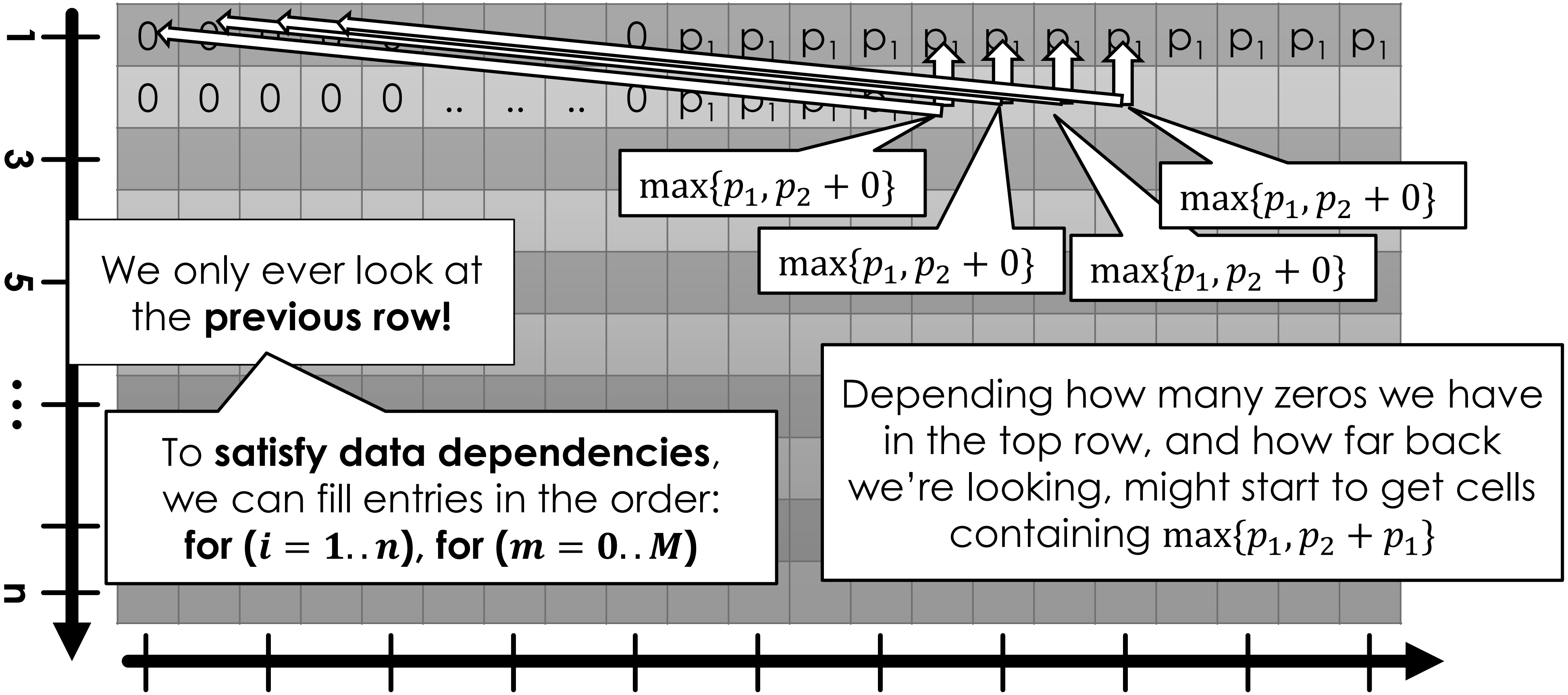
w_2

i-axis (can use items in $1..i$)

m-axis (remaining weight limit)

FILLING THE ARRAY:

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$



$\max\{p_1, p_2 + 0\}$

$\max\{p_1, p_2 + 0\}$

$\max\{p_1, p_2 + 0\}$

$\max\{p_1, p_2 + 0\}$

We only ever look at the **previous row!**

To **satisfy data dependencies**, we can fill entries in the order: **for ($i = 1..n$), for ($m = 0..M$)**

Depending how many zeros we have in the top row, and how far back we're looking, might start to get cells containing $\max\{p_1, p_2 + p_1\}$

i-axis (can use items in $1..i$)

m-axis (remaining weight limit)

Would the following fill-order work?
for ($i = 1..n$), for ($m = M..0$)

EXERCISE

$$\begin{array}{ll} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \end{array}$$

Suppose we have profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30.

The following table is computed:

$\boldsymbol{m}$-axis
(weight)

$\boldsymbol{i}$-axis
(items)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1														
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3														
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	?														
4																															
5																															
6																															

$P[3, 16] =$ **? What do you think ?**

EXERCISE

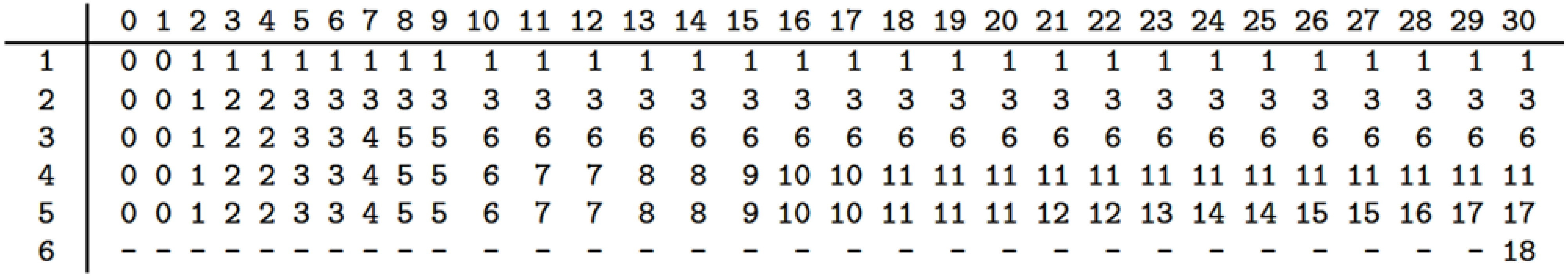
$$\begin{matrix} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \end{matrix}$$

Suppose we have profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30.

The following table is computed:

$\boldsymbol{m}$-axis
(weight)

$\boldsymbol{i}$-axis
(items)

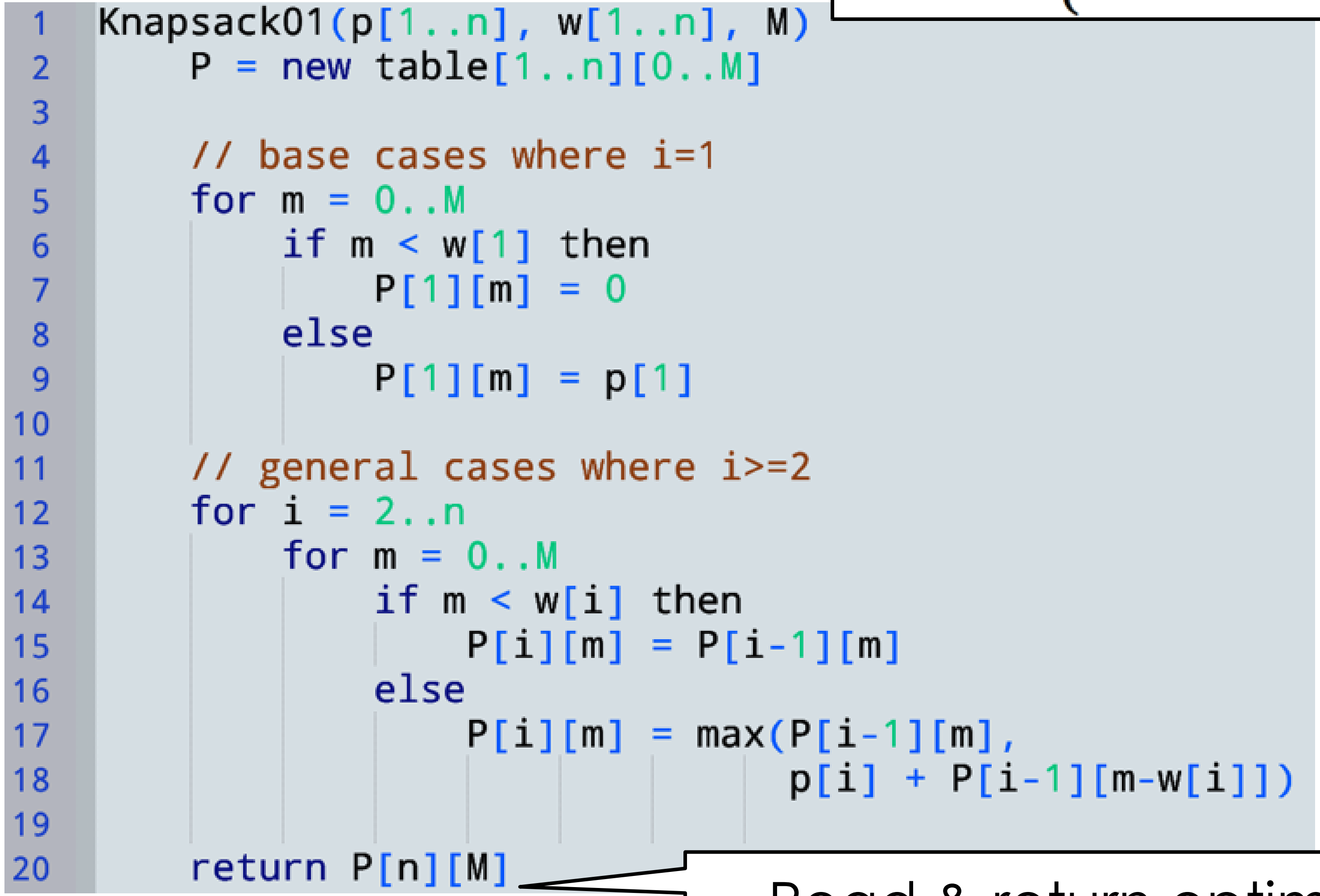


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	18

$$P[3,16] = \max\{P[2,16], P[2,11] + 3\} = \max\{3, 3+3\} = 6.$$

Recall: To satisfy data dependencies, we can fill entries in the order: **for ($i = 1..n$), for ($m = 0..M$)**

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$



```
Knapsack01(p[1..n], w[1..n], M)
    P = new table[1..n][0..M]

    // base cases where i=1
    for m = 0..M
        if m < w[1] then
            P[1][m] = 0
        else
            P[1][m] = p[1]

    // general cases where i>=2
    for i = 2..n
        for m = 0..M
            if m < w[i] then
                P[i][m] = P[i-1][m]
            else
                P[i][m] = max(P[i-1][m],
                              p[i] + P[i-1][m-w[i]])

    return P[n][M]
```

Read & return optimal **profit**

How about the optimal **items?**

OUTPUTTING **CONTENTS OF** THE OPTIMAL KNAPSACK **O**

The optimal solution is computed by tracing back through the table.

For the previous example, consisting of profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30, the optimal solution is **? ? ?**

weight limit remaining

Items you can take

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	18

8 > 6 so O **must** take item 4

Same profit using items 1..4 or 1..5. So, **there exists** an optimal solution O that does **not** use **item 5!** Consider O.

Best profit for remaining items + weight

18 > 17, so **any** optimal solution **must** take item 6

remaining weight = 14

Start at optimal profit

Exercise: continue, and determine which other items **are in O**

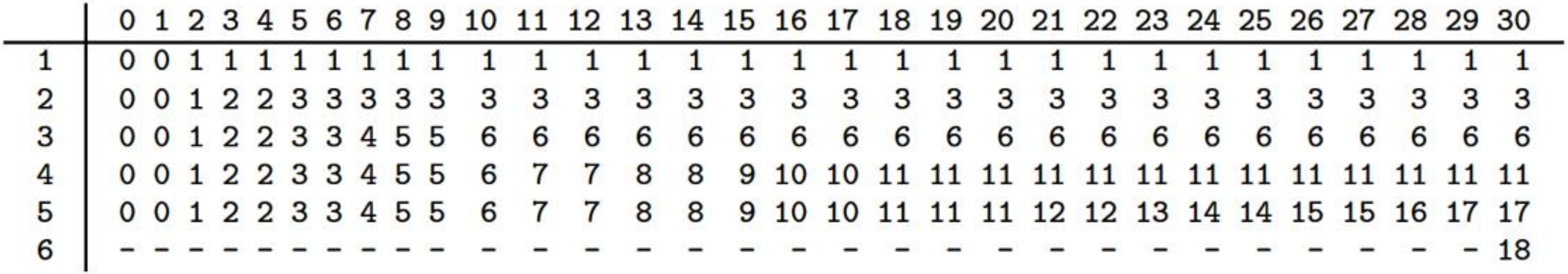
OUTPUTTING **CONTENTS OF** THE OPTIMAL KNAPSACK **O**

The optimal solution is computed by tracing back through the table.

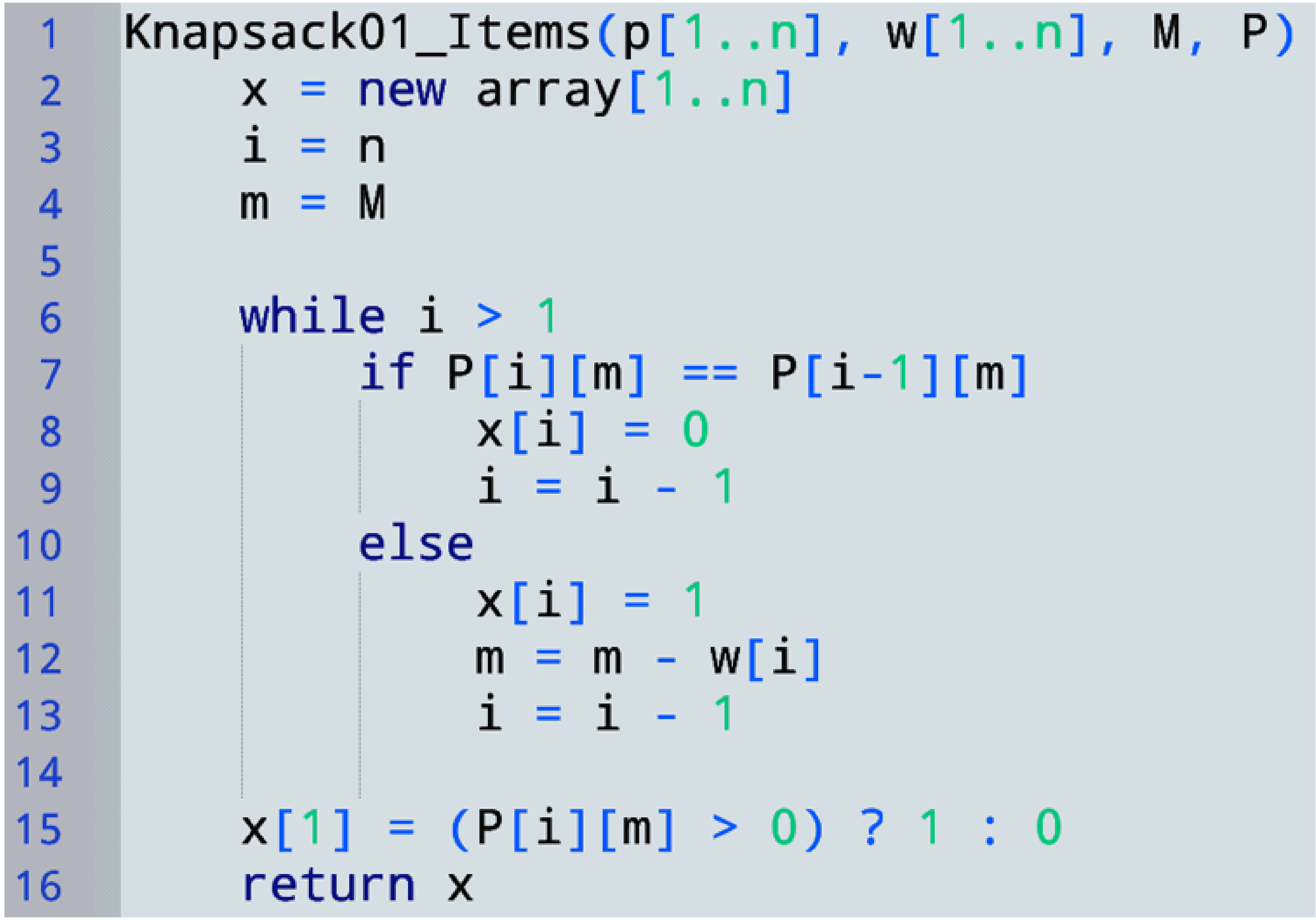
For the previous example, consisting of profits $1, 2, 3, 5, 7, 10$, weights $2, 3, 5, 8, 13, 16$, and capacity 30, the optimal solution is $[1, 1, 0, 1, 0, 1]$.

weight limit remaining

Items you can take



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
6	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	18



```
Knapsack01_Items(p[1..n], w[1..n], M, P)
    x = new array[1..n]
    i = n
    m = M

    while i > 1
        if P[i][m] == P[i-1][m]
            x[i] = 0
            i = i - 1
        else
            x[i] = 1
            m = m - w[i]
            i = i - 1

    x[1] = (P[i][m] > 0) ? 1 : 0
    return x
```

Runtime given P?

$\Theta(n)$

Is this linear time?

More on this soon…

Complexity of the Algorithm

Suppose we assume the unit cost model, so additions / subtractions take time $O(1)$.

The complexity to construct the table is $\Theta(nM)$

Is this a polynomial-time algorithm, as a function of the size of the problem instance?

We have

$$\text{size}(I) = \log_2 M + \sum_{i=1}^{n} \log_2 w_i + \sum_{i=1}^{n} \log_2 p_i.$$

Note in particular that M is exponentially large compared to $\log_2 M$. So constructing the table is not a polynomial-time algorithm, even in the unit cost model.

What would the complexity of a recursive algorithm be?

So the DP alg is faster when there are **many item types**, but **small weight limit**

Huge **n** is fine, but **M** should be in **poly(n)** to get an asymptotic improvement

DP takes $\boldsymbol{\Theta(nM)}$ **time**, which *could be* $\boldsymbol{\Theta(n2^n)}$ **for huge M**

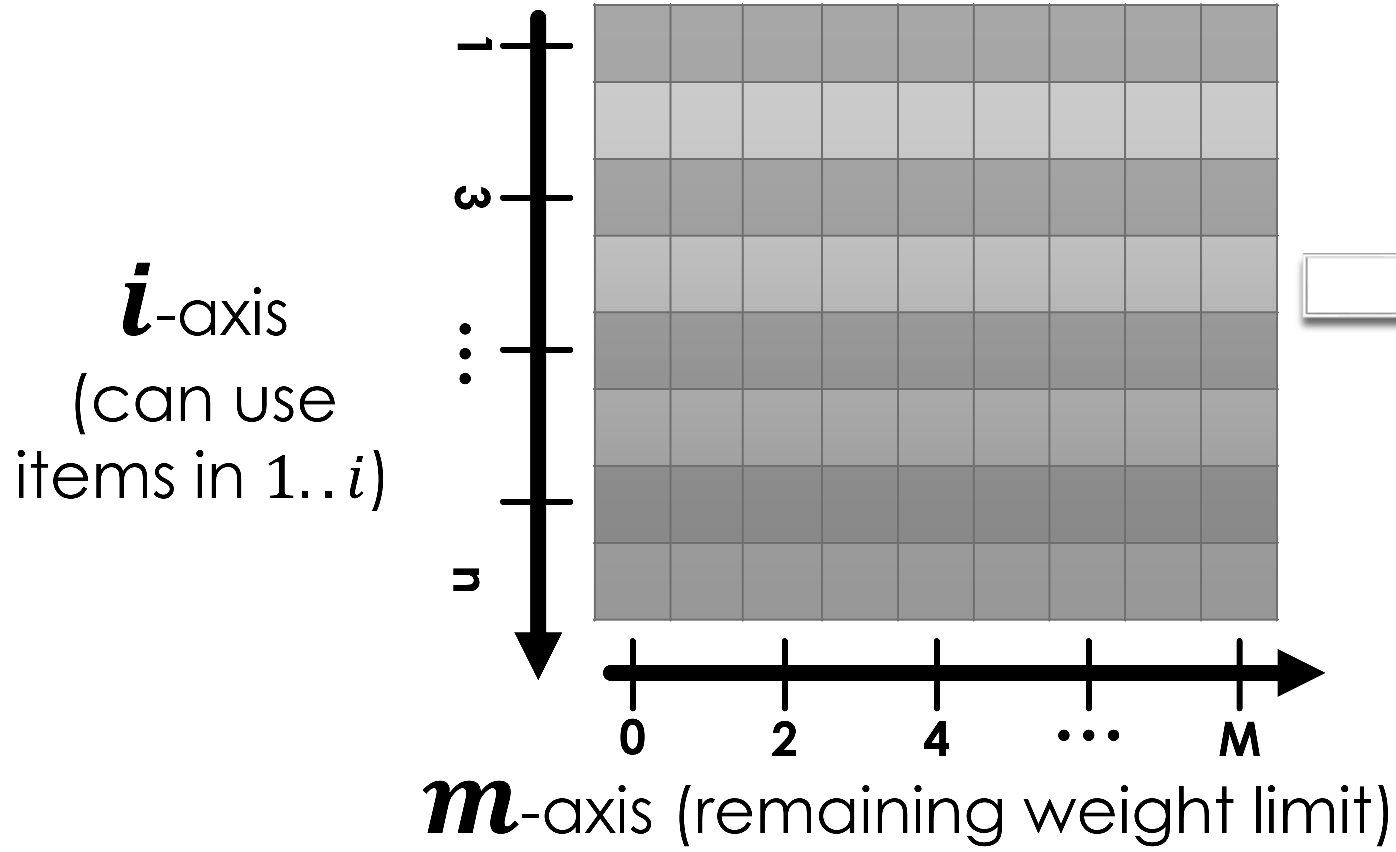
n must be **very small**

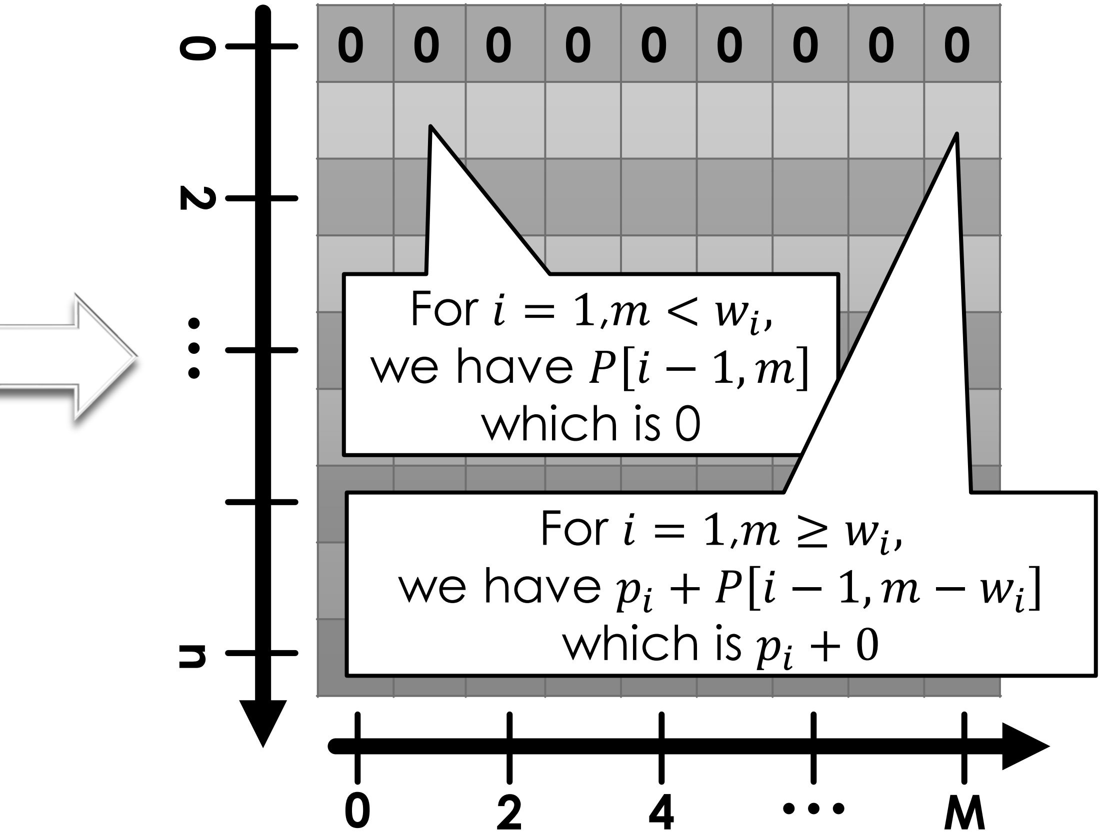
A recursive algorithm would take $\sim\boldsymbol{\Theta(2^n)}$ **time**

SIMPLIFYING BASE CASES

$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 1, m \geq w_i \\ P[i-1,m] & \text{if } i \geq 1, m < w_i \\ 0 & \text{if } i = 0 \end{cases}$$

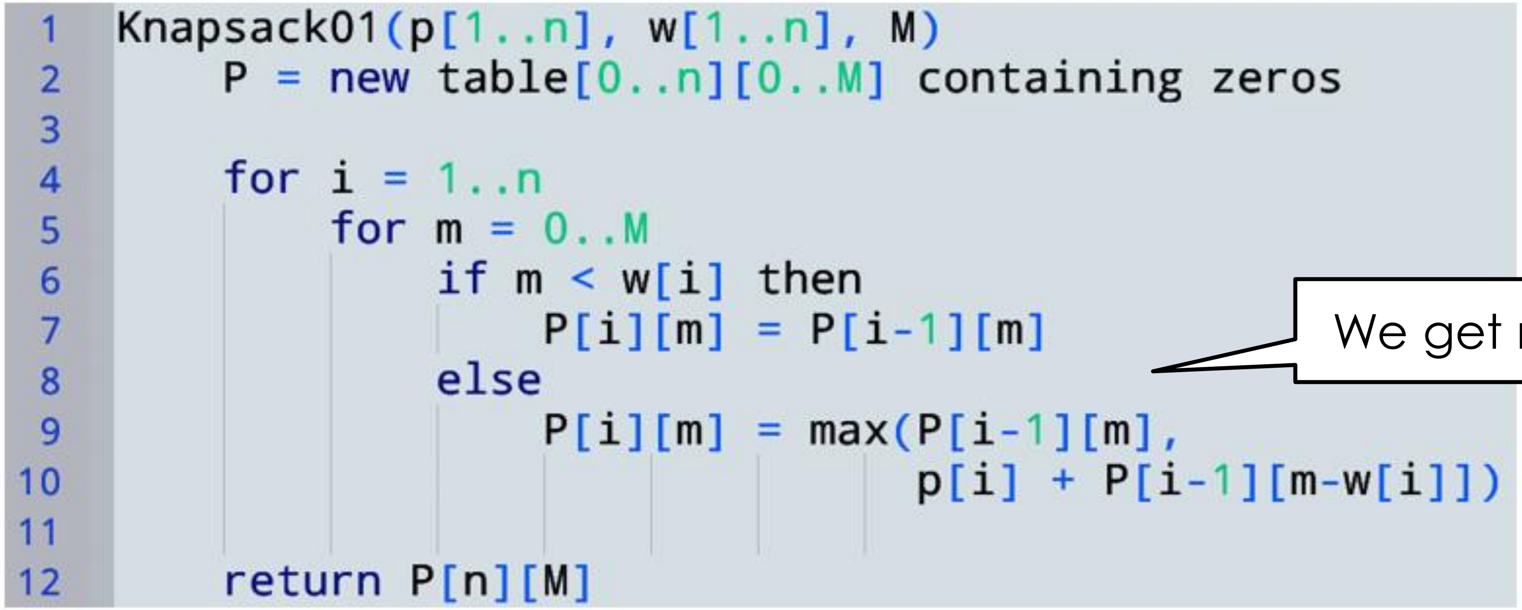
$$P[i,m] = \begin{cases} \max\{P[i-1,m], p_i + P[i-1,m-w_i]\} & \text{if } i \geq 2,\ m \geq w_i \\ P[i-1,m] & \text{if } i \geq 2,\ m < w_i \\ p_1 & \text{if } i = 1,\ m \geq w_1 \\ 0 & \text{if } i = 1,\ m < w_1. \end{cases}$$





For $i = 1, m < w_i$,
we have $P[i-1,m]$
which is 0

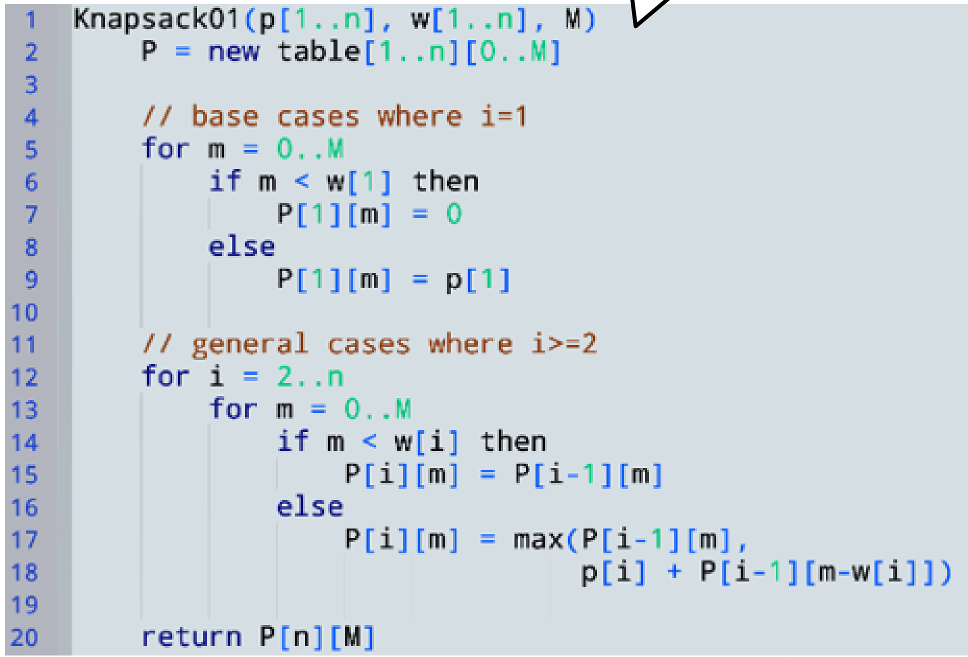
For $i = 1, m \geq w_i$,
we have $p_i + P[i-1,m-w_i]$
which is $p_i + 0$



```
Knapsack01(p[1..n], w[1..n], M)
    P = new table[0..n][0..M] containing zeros

    for i = 1..n
        for m = 0..M
            if m < w[i] then
                P[i][m] = P[i-1][m]
            else
                P[i][m] = max(P[i-1][m],
                              p[i] + P[i-1][m-w[i]])

    return P[n][M]
```

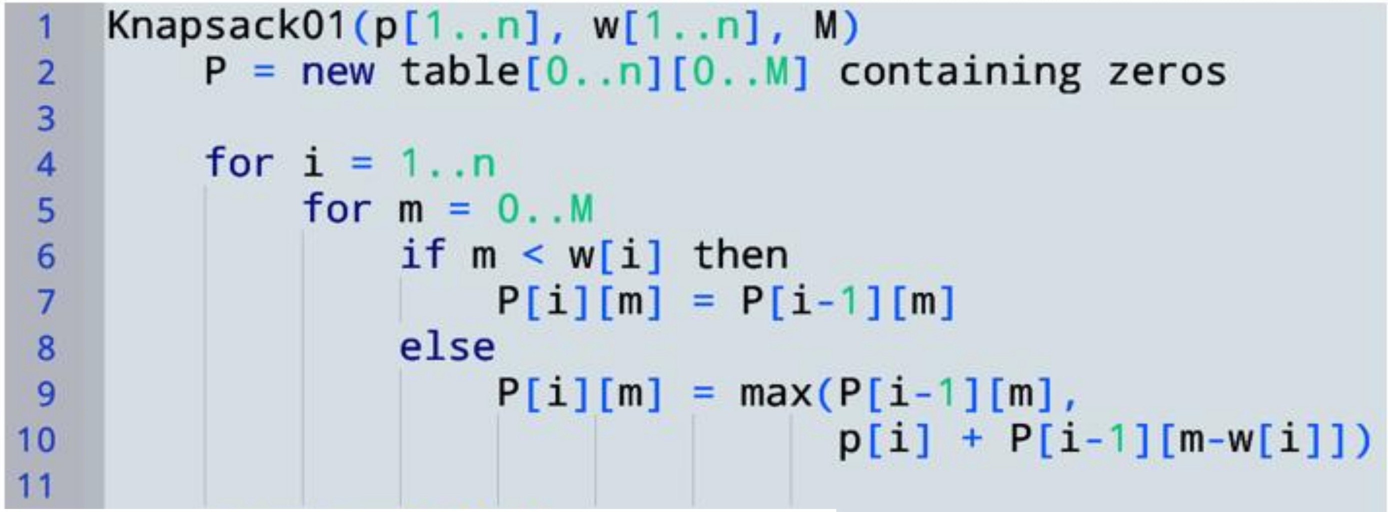
We get much simpler code!

Compare:

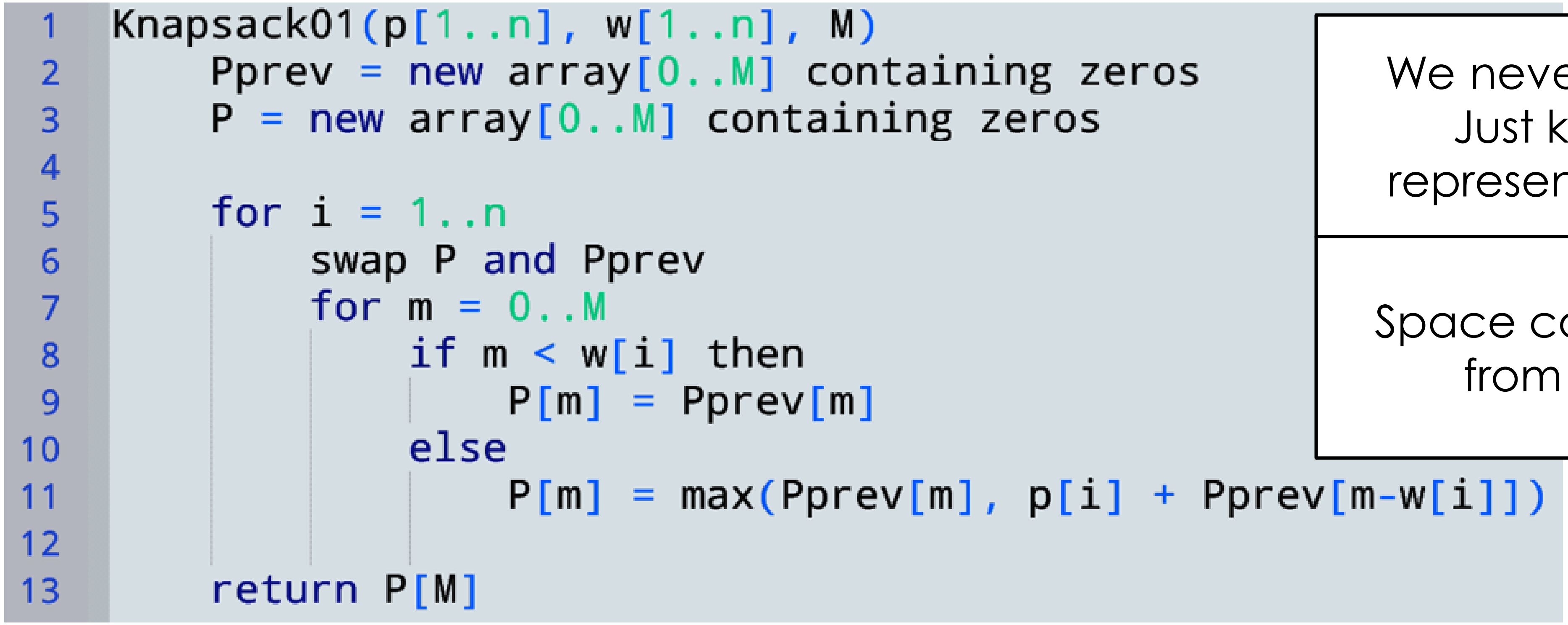


```
Knapsack01(p[1..n], w[1..n], M)
    P = new table[1..n][0..M]

    // base cases where i=1
    for m = 0..M
        if m < w[1] then
            P[1][m] = 0
        else
            P[1][m] = p[1]

    // general cases where i>=2
    for i = 2..n
        for m = 0..M
            if m < w[i] then
                P[i][m] = P[i-1][m]
            else
                P[i][m] = max(P[i-1][m],
                              p[i] + P[i-1][m-w[i]])

    return P[n][M]
```

SAVING SPACE



```
Knapsack01(p[1..n], w[1..n], M)
    P = new table[0..n][0..M] containing zeros

    for i = 1..n
        for m = 0..M
            if m < w[i] then
                P[i][m] = P[i-1][m]
            else
                P[i][m] = max(P[i-1][m],
                              p[i] + P[i-1][m-w[i]])

```



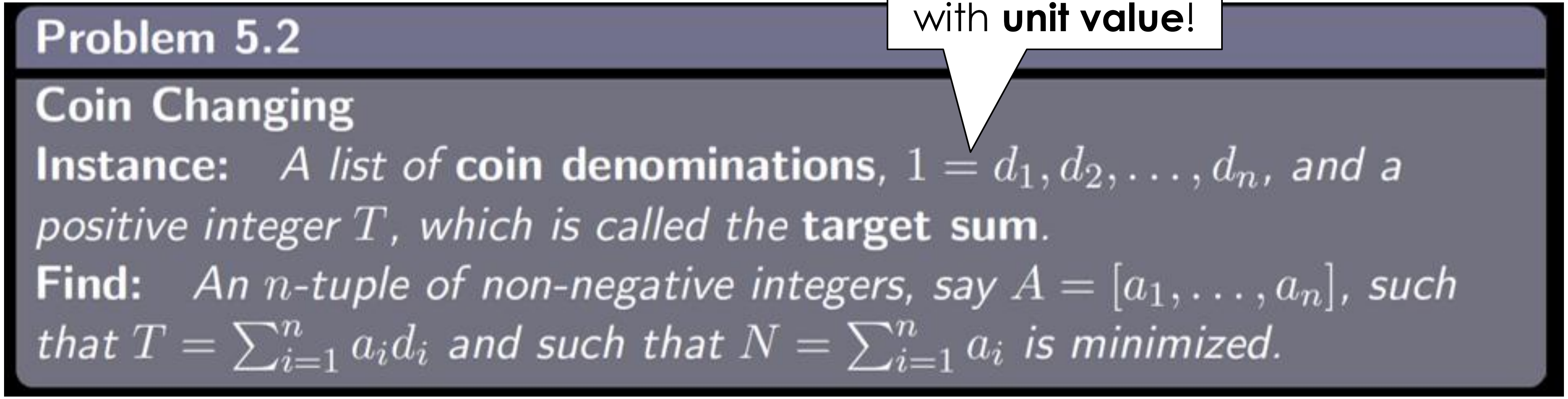
```
Knapsack01(p[1..n], w[1..n], M)
    Pprev = new array[0..M] containing zeros
    P = new array[0..M] containing zeros

    for i = 1..n
        swap P and Pprev
        for m = 0..M
            if m < w[i] then
                P[m] = Pprev[m]
            else
                P[m] = max(Pprev[m], p[i] + Pprev[m-w[i]])

    return P[M]
```

We never look at P[i-2][…]. Just keep two arrays representing P[i] and P[i-1]

Space complexity changes from $O(mn)$ to $O(m)$

COIN CHANGING

Coin Changing



Problem 5.2

Coin Changing

Instance: *A list of* **coin denominations**, $1 = d_1, d_2, \ldots, d_n$, *and a positive integer* T, *which is called the* **target sum**.

Find: *An* n*-tuple of non-negative integers, say* $A = [a_1, \ldots, a_n]$, *such that* $T = \sum_{i=1}^{n} a_i d_i$ *and such that* $N = \sum_{i=1}^{n} a_i$ *is minimized.*

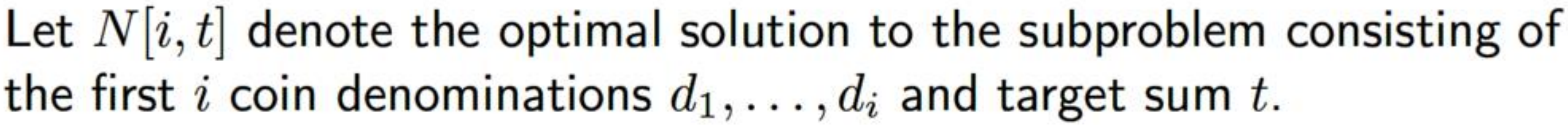
There **is** a denomination with **unit value**!

What subproblems should be considered?

What table of values should we fill in?

In 0-1 knapsack, we only considered **two subproblems** in our recurrence: **taking an item, or not.**

Here we can do **more than** *use a coin denomination or not*.



Let $N[i,t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Exploring: some sensible base case(s)?

General case:
What are the different ways we could use coin denomination $\boldsymbol{d_i}$?
What subproblems / solutions should we use?

Final recurrence relation

Let $N[i,t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Since $d_1 = 1$, we immediately have $N[1,t] = t$ for all t.

Also $\boldsymbol{N[i,0] = 0}$ for all i

General case:
What are the different ways we could use coin denomination $\boldsymbol{d_i}$?
What subproblems / solutions should we use?

Final recurrence relation

Let $N[i,t]$ denote the optimal solution to the subproblem consisting of the first i coin denominations $d_1, \ldots, d_i$ and target sum t.

Also $\boldsymbol{N[i,0] = 0}$ for all i

Since $d_1 = 1$, we immediately have $N[1,t] = t$ for all t.

For $i \geq 2$, the number of coins of denomination d_i is an integer j where $0 \leq j \leq \lfloor t/d_i \rfloor$.

If we use j coins of denomination d_i, then the target sum is reduced to $t - jd_i$, which we must achieve using the first $i-1$ coin denominations.

Thus we have the following recurrence relation:

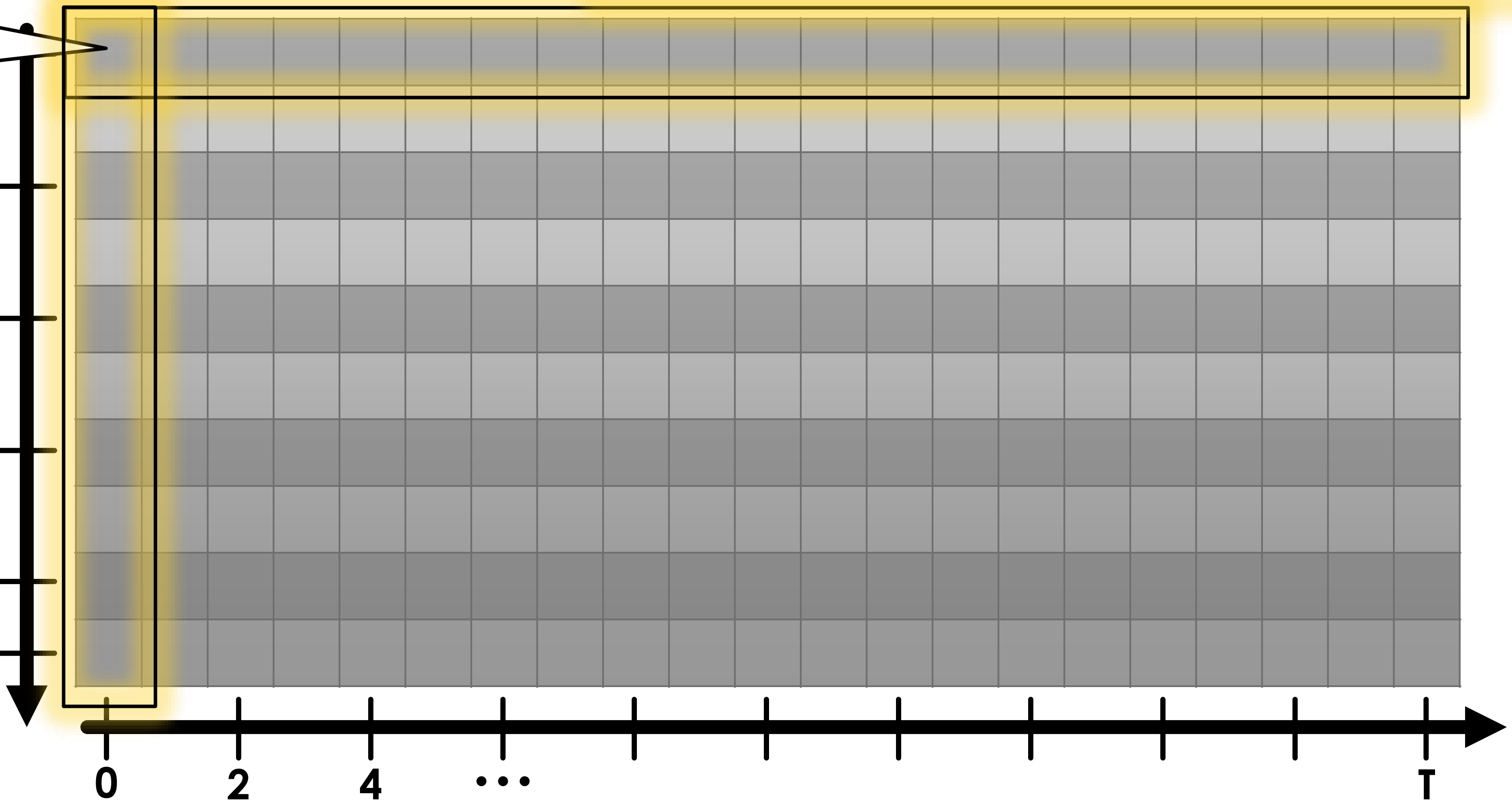
$$N[i,t] = \begin{cases} \min\{j + N[i-1, t-jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\ t & \text{if } i = 1 \text{ OR } t = 0 \end{cases}$$

FILLING THE ARRAY

$N[1 \ldots n, 0 \ldots T]$:

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \\ & \text{OR } t = 0 \end{cases}$$

No data dependencies on any other array cells.



***i*-axis (coin type)**

(recall: $N[i,t]$ uses coin types $1..i$)

***t*-axis (target sum remaining)**

FILLING THE ARRAY $N[1 \ldots n, 0 \ldots T]$:

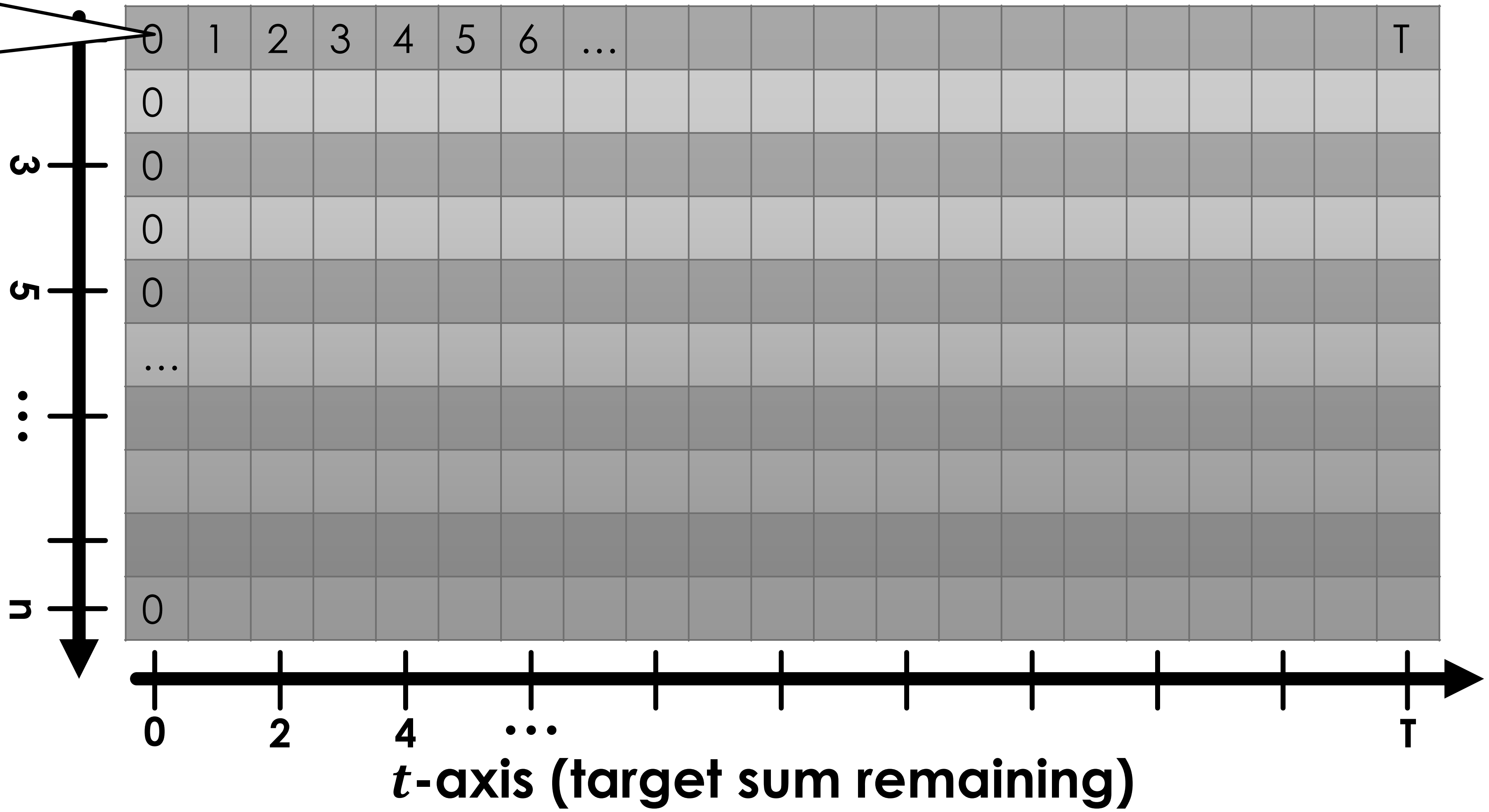
$$N[i,t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \end{cases}$$

OR $t = 0$

No data dependencies on any other array cells.

i-axis (coin type)

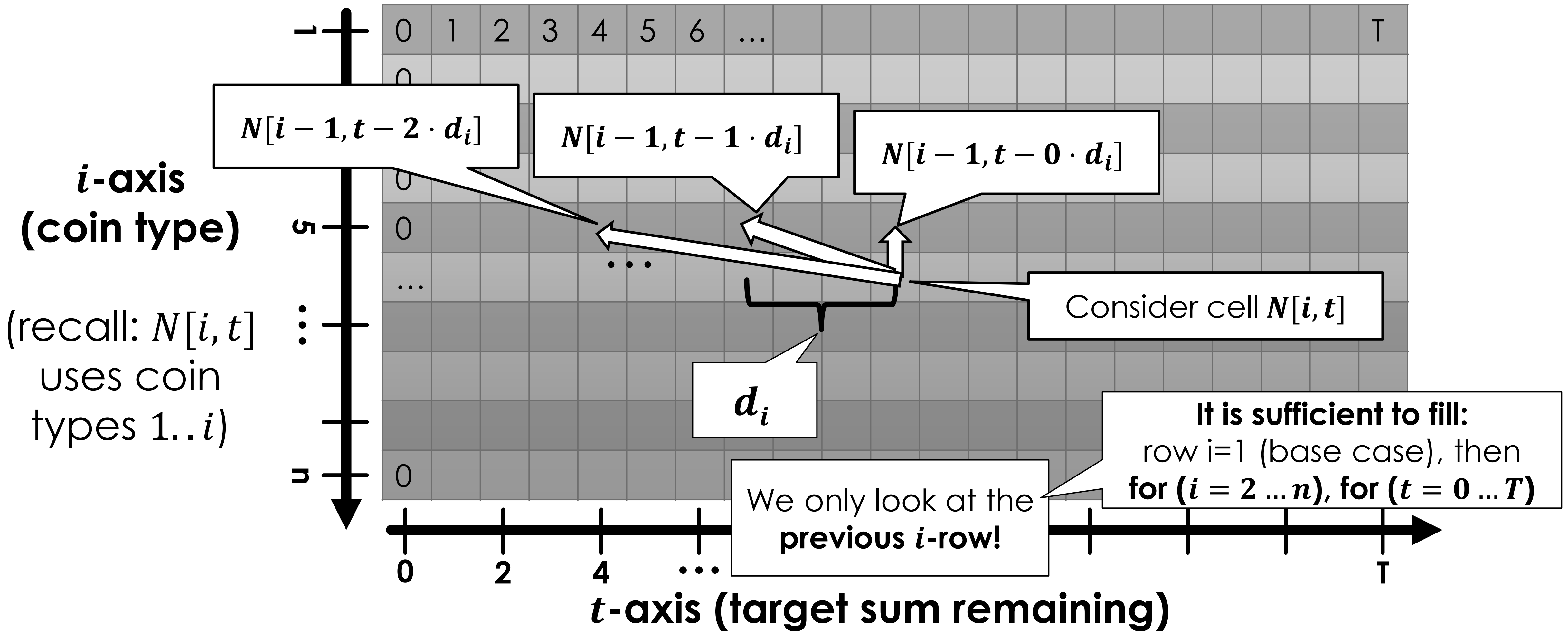
(recall: $N[i,t]$ uses coin types $1..i$)



FILLING THE ARRAY $N[1 \dots n, 0 \dots T]$:

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t-jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \end{cases}$$

OR $t = 0$



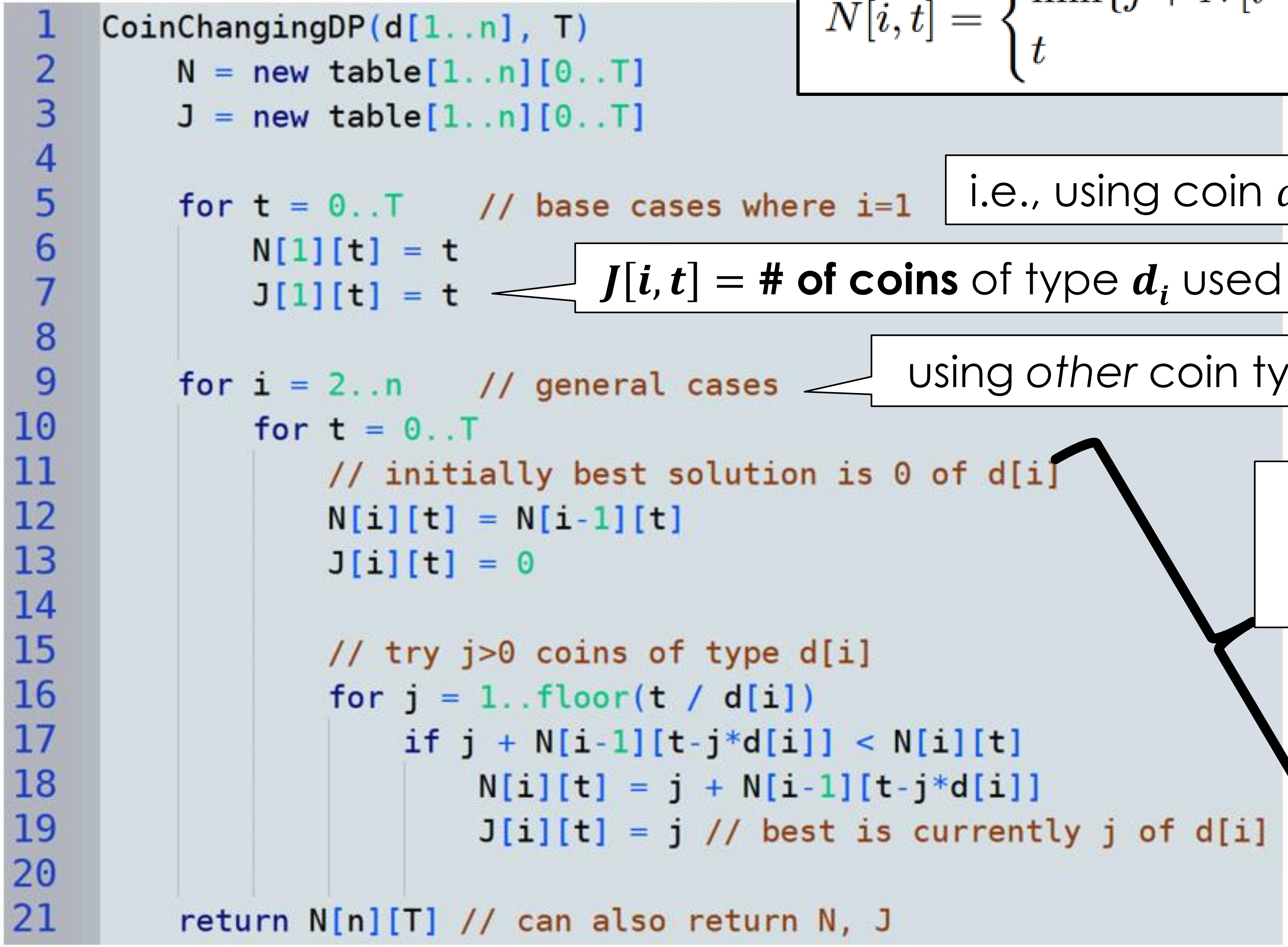
(recall: $N[i,t]$ uses coin types $1..i$)

Consider cell $N[i,t]$

We only look at the **previous i-row!**

It is sufficient to fill:
row i=1 (base case), then
for ($i = 2 \dots n$), for ($t = 0 \dots T$)

$$N[i,t] = \begin{cases} \min\{j + N[i-1, t-jd_i] : 0 \le j \le \lfloor t/d_i \rfloor\} & \text{if } i \ge 2 \\ t & \text{if } i = 1. \end{cases}$$



```
CoinChangingDP(d[1..n], T)
    N = new table[1..n][0..T]
    J = new table[1..n][0..T]

    for t = 0..T    // base cases where i=1
        N[1][t] = t
        J[1][t] = t

    for i = 2..n    // general cases
        for t = 0..T
            // initially best solution is 0 of d[i]
            N[i][t] = N[i-1][t]
            J[i][t] = 0

            // try j>0 coins of type d[i]
            for j = 1..floor(t / d[i])
                if j + N[i-1][t-j*d[i]] < N[i][t]
                    N[i][t] = j + N[i-1][t-j*d[i]]
                    J[i][t] = j // best is currently j of d[i]

    return N[n][T] // can also return N, J
```

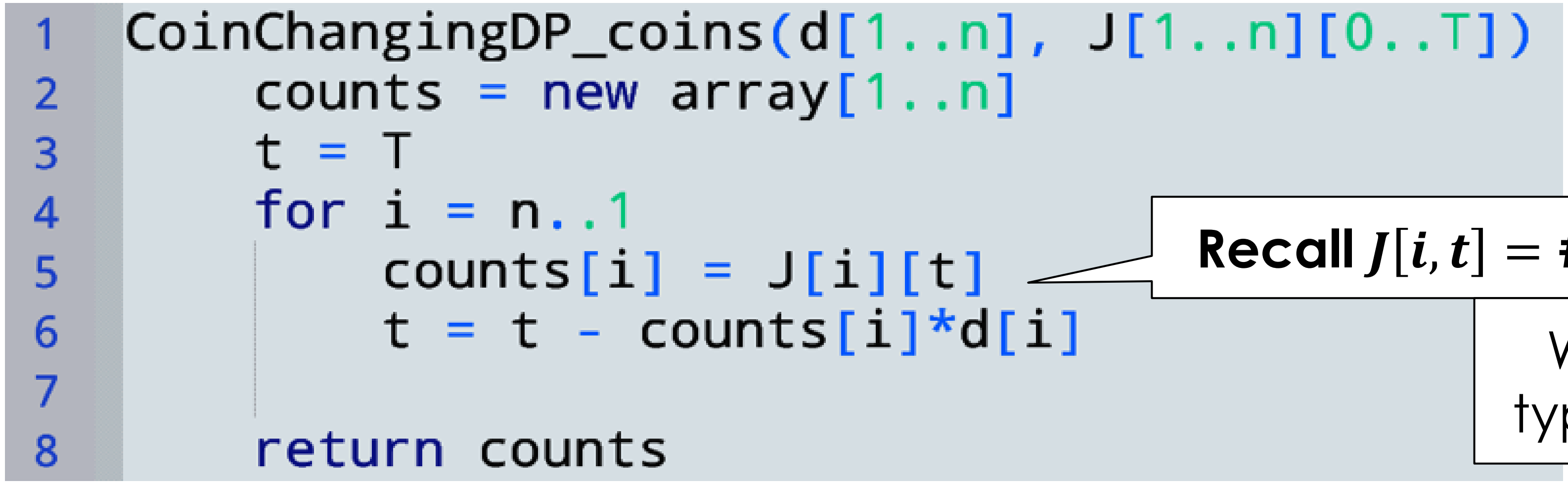
i.e., using coin $d_1 = 1$

$\boldsymbol{J[i,t]}$ = **# of coins** of type $\boldsymbol{d_i}$ used in $N[i,t]$

using *other* coin types

Compute min{…} over $\boldsymbol{j = 0 \ldots \lfloor t/d_i \rfloor}$

OUTPUTTING **OPTIMAL SET OF COINS**

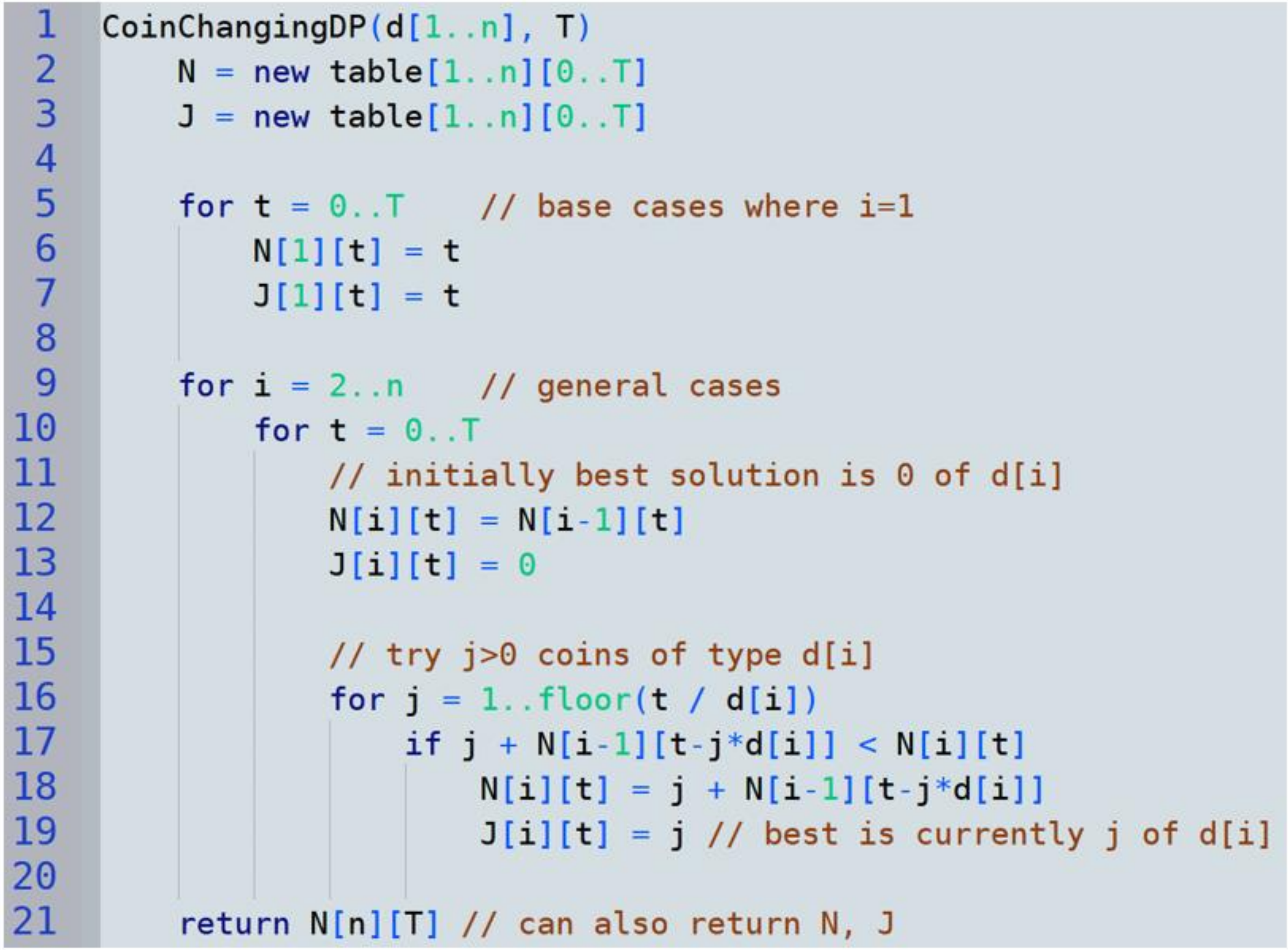


```
CoinChangingDP_coins(d[1..n], J[1..n][0..T])
    counts = new array[1..n]
    t = T
    for i = n..1
        counts[i] = J[i][t]
        t = t - counts[i]*d[i]

    return counts
```

Recall $\boldsymbol{J[i,t]}$ = # of coins of type $\boldsymbol{d_i}$ used in $N[i,t]$

We start at $J[n][T]$ = # of coins of type d_n used in the **optimal solution**

Exercise for later:
compute the correct output
without using $J[i,t]$
(i.e., using only N, d, T)



```
CoinChangingDP(d[1..n], T)
    N = new table[1..n][0..T]
    J = new table[1..n][0..T]

    for t = 0..T    // base cases where i=1
        N[1][t] = t
        J[1][t] = t

    for i = 2..n    // general cases
        for t = 0..T
            // initially best solution is 0 of d[i]
            N[i][t] = N[i-1][t]
            J[i][t] = 0

            // try j>0 coins of type d[i]
            for j = 1..floor(t / d[i])
                if j + N[i-1][t-j*d[i]] < N[i][t]
                    N[i][t] = j + N[i-1][t-j*d[i]]
                    J[i][t] = j // best is currently j of d[i]

    return N[n][T] // can also return N, J
```

Time complexity?

Unit cost computational model is reasonable here

Consider instance $I = (d, T)$

Runtime $\boldsymbol{R(I)} \in O\left(\sum_{i=2}^{n} \sum_{t=0}^{T} \left\lfloor \frac{t}{d_i} \right\rfloor\right)$

$$R(I) \in O\left(\sum_{i=2}^{n} \frac{1}{d_i} \sum_{t=0}^{T} t\right)$$

$$R(I) \in O\left(\sum_{i=2}^{n} \frac{1}{d_i}\left(\frac{T(T+1)}{2}\right)\right)$$

$\boldsymbol{R(I) \in O(DT^2)}$

where $D = \sum_{i=2}^{n} \frac{1}{d_i} < n$.

If T is small, this is much better than brute force

POLYNOMIAL TIME

- An algorithm runs in (worst case) **polynomial time** IFF its runtime $R(I)$ on every input is upper bounded by a polynomial in the input size S
- I.e., $R(I) \in O\left(c_0 + c_1 S + c_2 S^2 + c_3 S^3 + \cdots + c_k S^k\right)$
  for **constants $\boldsymbol{k}$ and $\boldsymbol{c_0, \dots, c_k}$**
- … so is $O(nT^2)$ polynomial in our input size S?

INPUT SIZE

- $S = bits(T) + bits(d_1) + \cdots + bits(d_n)$
- It takes $\lceil \mathbf{log_2} \boldsymbol{T} \rceil$ **bits** to store T
- It takes $\lceil \mathbf{log_2} \boldsymbol{d_i} \rceil$ **bits** to store **each** d_i
- **Assume $\boldsymbol{d_i \leq T}$ (otherwise $\boldsymbol{d_i}$ cannot be used at all, and should be omitted from the input)**
  - Then we have $\lceil \log_2 d_i \rceil \in O(\log T)$
  - So, $\boldsymbol{S} \in \boldsymbol{O(n \log T)}$

COMPARING $T(I)$ TO S

- Recall $\boldsymbol{R(I) \in O(nT^2)}$ and $\boldsymbol{S \in O(n \log T)}$
- As an example, if n is fixed at 10 and T is allowed to vary, then $\boldsymbol{S \in \mathrm{O}(\log T)}$ and $\boldsymbol{R(I) \in O(T^2)}$
  - In this case, $R(I)$ is **exponential in $\boldsymbol{S}$**
- However, if T is fixed at 10 and n is allowed to vary, then $\boldsymbol{S \in O(n)}$ and $\boldsymbol{R(I) \in O(n)}$
  - In this case, $R(I)$ is **linear in $\boldsymbol{S}$**
- So, **large $\boldsymbol{n}$** and **small $\boldsymbol{T}$** is where this DP solution shines!

A BIT MORE ANALYSIS

- Recall $\boldsymbol{R(I)} \in \boldsymbol{O(nT^2)}$ and $\boldsymbol{S} \in \boldsymbol{O(n \log T)}$
- **<u>If</u>** $\boldsymbol{T} \in \boldsymbol{O(n)}$, then $S \in O(n \log n)$ and $R(I) \in O(n^3)$
  - Note $O(n^3)$ is a **smaller** runtime than $O\left(\boldsymbol{S^3}\right) = O(n^3 \log n)$
  - And S^3 is polynomial in $\boldsymbol{S}$, so $O(n^3)$ is a **polynomial runtime**
- So, **for some inputs** with *relatively small* T,
  we can get polynomial runtimes!
  - In particular, for $\boldsymbol{T} \in \boldsymbol{O(n^k)}$ where $\boldsymbol{k}$ **is constant,**
    $R(I) \in O\left(n\left(n^k\right)^2\right) = O(n^{2k+1})$ and $S \in O\left(n \log n^k\right) = O(n \log n)$
  - And $R(I) \in O\left(n^{2k+1}\right) \subseteq O\left((n \log n)^{2k+1}\right) = O(S^{2k+1})$