Lecture 24: Weighted Knapsack

Dall·E

COSC 311 *Algorithms*, Fall 2022

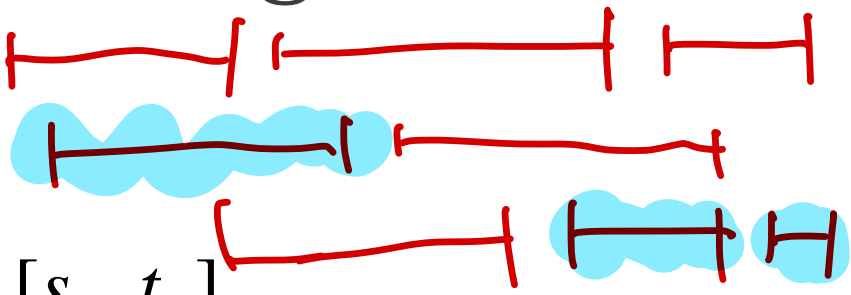
Upcoming

- Week 09: finish dynamic programming
- Week 10: network flow & applications
- Week 11: midterm 2
- Weeks 12, 13: NP completeness

Overview

1. Finishing Weighted Interval Scheduling
2. The Knapsack Problem

Weighted Interval Scheduling



Input:

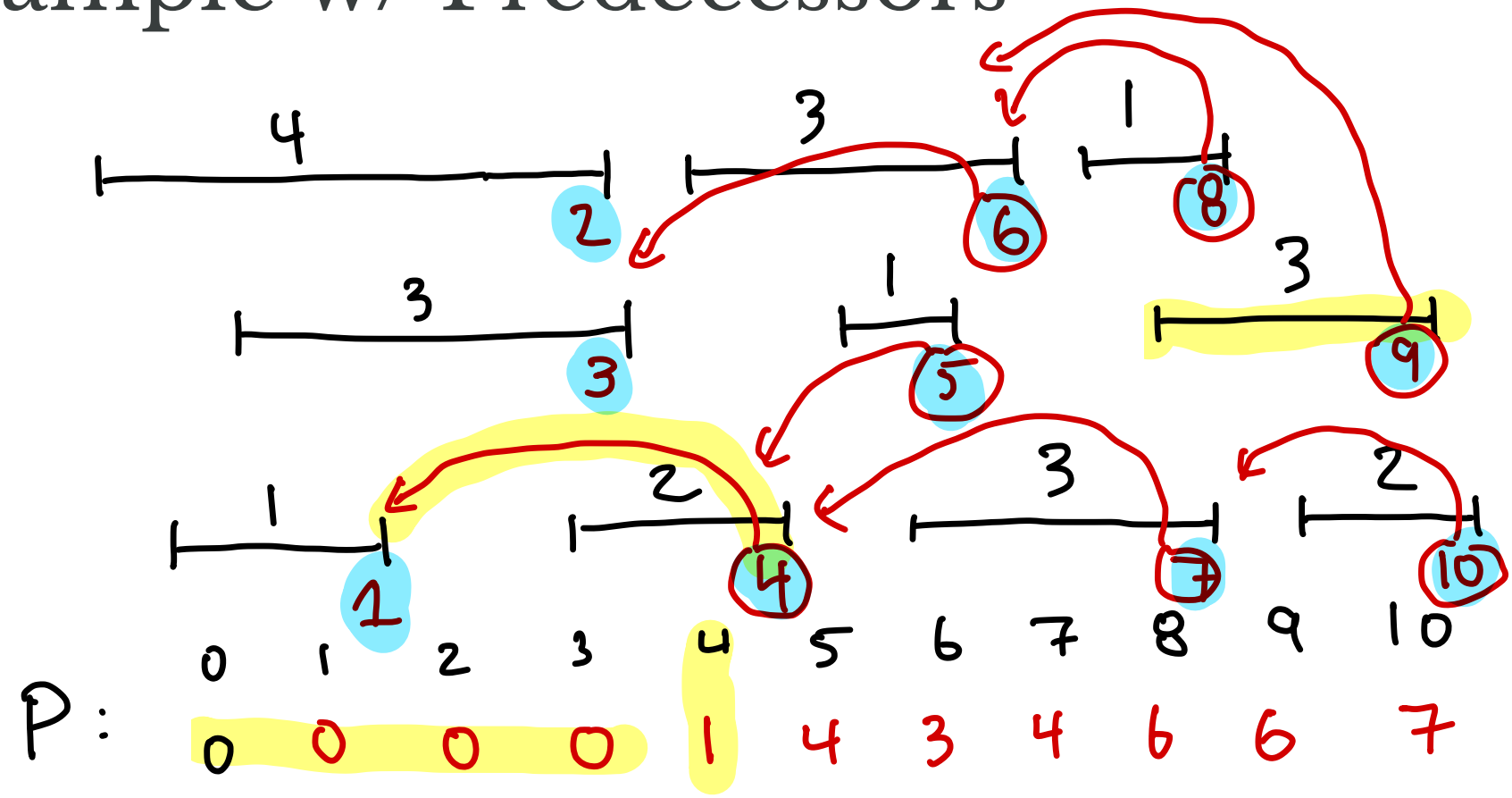
1. A set R of n intervals
   $r_1 = [s_1, t_1], r_2 = [s_2, t_2], \ldots, r_n = [s_n, t_n]$
2. For each interval $r \in R$, a **weight** $w(r) > 0$
   - e.g., weight = profit from serving request r

Output. A collection of intervals from R that is

1. *feasible* no two intervals overlap
2. *maximum weight* choice maximizes sum of $w(r)$ for chosen r

Note: equivalent to (unweighted) interval scheduling when all weights are the same

Example w/ Predecessors



→ running time

$\Omega(2^n)$

weights

predecessors. p[i] = j means j is i's pred.

last interval considered

A Recursive Solution

```
MaxWeightSchedule(w, p, n):
  if n = 0 then return 0
  opt-n <- w[n] + MaxWeightSchedule(w, p, p[n])
  opt-no-n <- MaxWeightSchedule(w, p, n-1)
  return Max(opt-n, opt-no-n)
```

base case

if opt sol'n does not contain n
then opt value is opt val for 1...n-1

if opt sol'n contains n, then value
is w[n] + opt value for 1..p[n]

Recursion to Iteration

Idea. Store array `max`:

- `max[i]` is maximum weight of schedule consisting of intervals $r_1, r_2, \ldots, r_i$

Question. How to initialize/update `max` values?

max[0] ← 0 (no requests served)

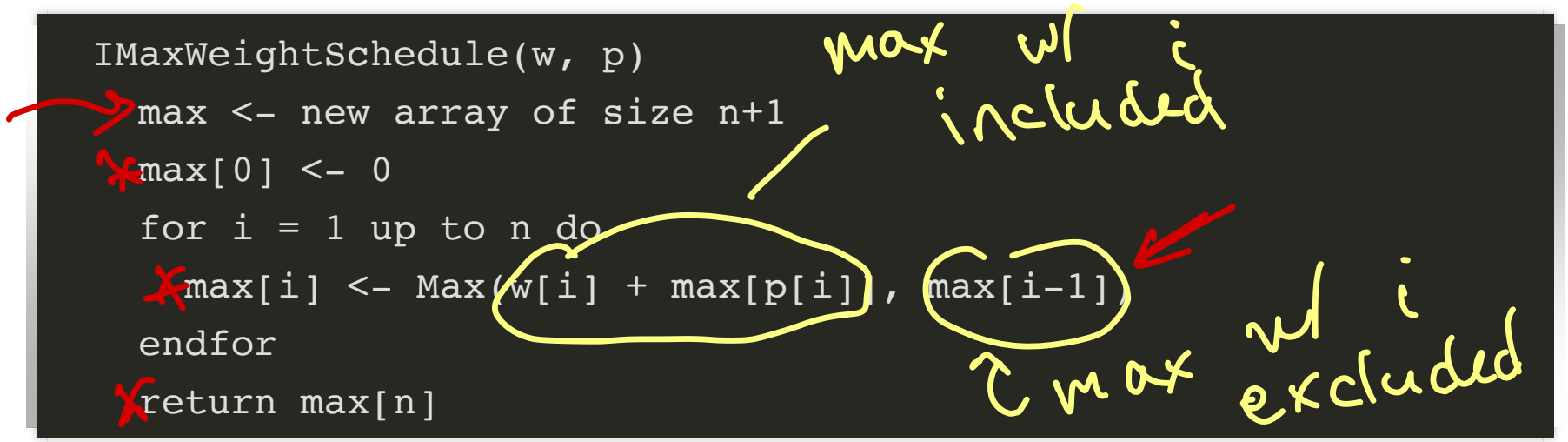
computed max[0..i-1]

max[i] ← Max(max[i-1], w(i) + max[p[i]])

i's predecessor

Iterative Solution

```
IMaxWeightSchedule(w, p)
  max <- new array of size n+1
  max[0] <- 0
  for i = 1 up to n do
    max[i] <- Max(w[i] + max[p[i]], max[i-1])
  endfor
  return max[n]
```



O(n).

Iterative Solution

```
IMaxWeightSchedule(w, p)
  max <- new array of size n+1
  max[0] <- 0
  for i = 1 up to n do
    max[i] <- Max(w[i] + max[p[i]], max[i-1])
  endfor
  return max[n]
```

Correctness:

- same argument as recursive solution

Running Time?

Overall Running Time?

Steps: (assume n intervals)

1. Sort intervals by end time
   - $O(n \log n)$ (merge-sort)
2. Compute array p
   Since array sorted by end time
   - use binary search for each interval
     s = start time
     find largest i w/ $t_i < s$
     $O(n \log n)$
3. Run `IMaxWeightSchedule(w, p)`
   - $O(n)$

Total?

$O(n \log n)$.

Exercise

Update `IMaxWeightSchedule` to return the actual schedule of maximum weight, not just the weight itself.

The Knapsack Problem

Knapsack Motivation

In *weighted interval scheduling* each request had:

- start time
- end time
- value

Goal: to service set of non-overlapping requests to maximize total value

Knapsack Motivation

In *weighted interval scheduling* each request had:

- start time
- end time
- value

Goal: to service set of non-overlapping requests to maximize total value

Relaxation. Requests have

- duration
- value

Each request can be scheduled at any time $\leq B$

Knapsack Problem

Input:

1. A set R of n requests, each having
   - duration (weight) b_r
   - value v_r
2. Total time (weight) budget B

Output: A set S of requests to service with

1. sum of durations of requests in S is at most B (feasibility)
2. sum of values of requests is maximized (optimality)

Knapsack Problem

Input:

1. A set R of n requests, each having
   - duration (weight) b_r
   - value v_r
2. Total time (weight) budget B

Output: A set S of requests to service with

1. sum of durations of requests in S is at most B
2. sum of values of requests is maximized

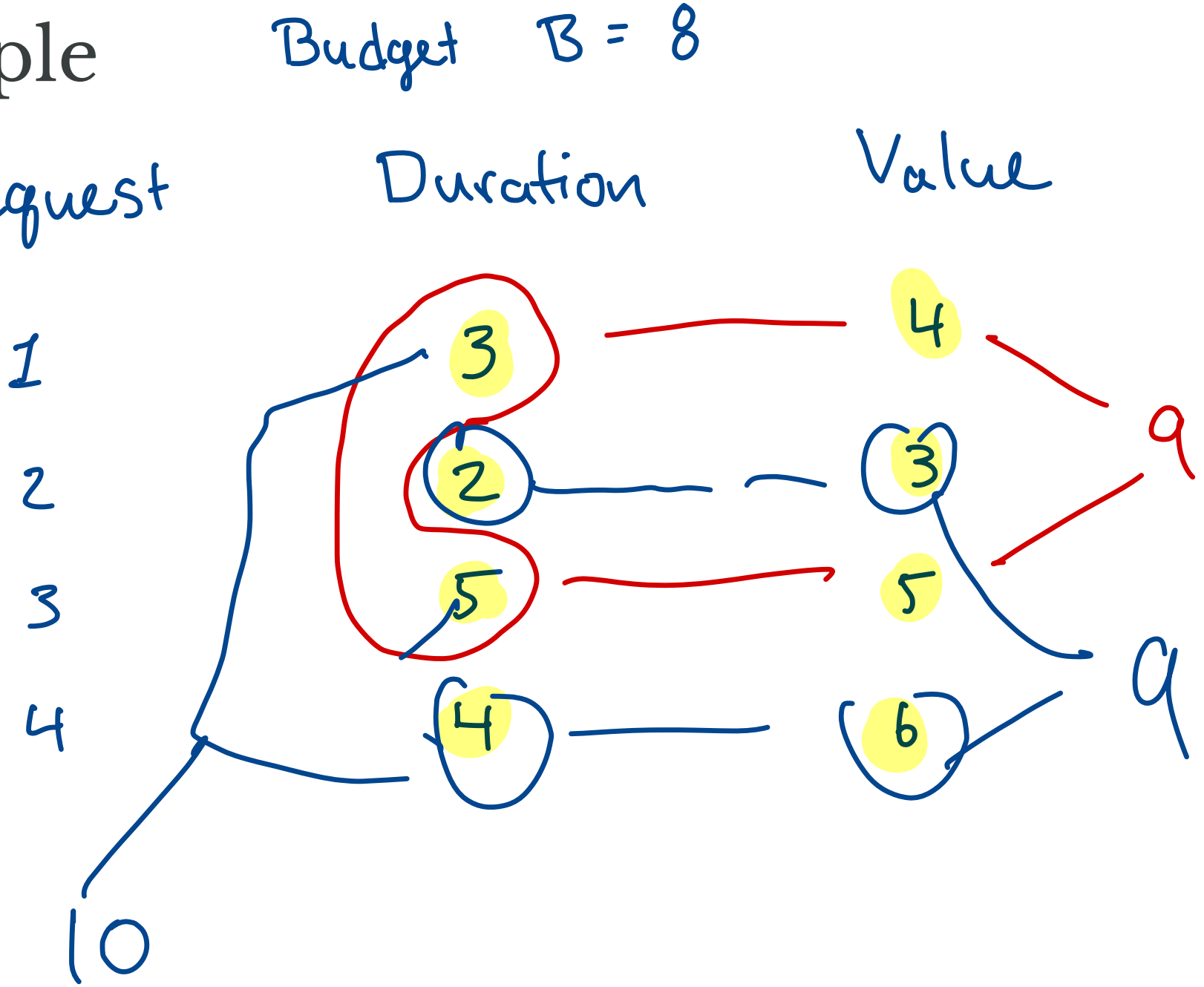
Constrained Optimization Problem

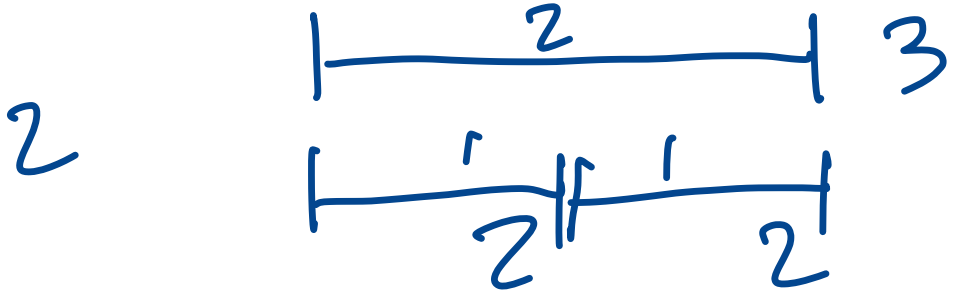
- maximize $\sum_{r \in S} v_r$ subject to $\sum_{r \in S} b_r \leq B$

objective

constraint

Example





B total budget

Recurrence Relation?

Previous technique. Express relationship between optimal solutions that

1. service the final request r_n — B − b_n remaining time budget
2. do not service final request r_n — B remaining budget

Question. How can we express this relationship for the knapsack problem?

A Recurrence Relation

Subtlety. Must keep track of *remaining budget*

- if r_n is *not* serviced, remaining budget is B
- if r_n is serviced, remaining budget is $B - b_n$

A Recurrence Relation

request

remaining budget

Subtlety. Must keep track of *remaining budget*

- if r_n is *not* serviced, remaining budget is B
- if r_n is serviced, remaining budget is $B - b_n$

Definition. For $j = 0, 1, \ldots, n$, $\text{opt}(j, C)$ is optimal value of set of requests from $1, 2, \ldots, j$ with budget C.

Question. Recursion relation for $\text{opt}(n, B)$ depending on whether or not we include request r_n?

$$\text{opt}(n,B) \leftarrow \text{Max}\left(\text{opt}(n-1, B),\ v_n + \text{opt}(n-1, B-b_n)\right)$$

Computing Optimal Values

Assume. All durations b_i are integers at most B.

- we will revisit this assumption later

Compute. To compute opt(n, B):

- Generate a *two dimensional array* `max` where `max[j, C]` stores the value opt(j, C)

Questions.

1. How to initialize `max`?
2. How to update `max`?

req index

budget

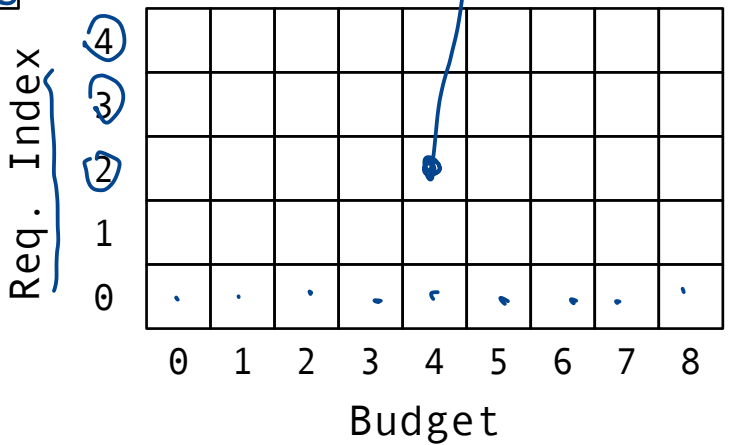
Example

Requests:

B=8

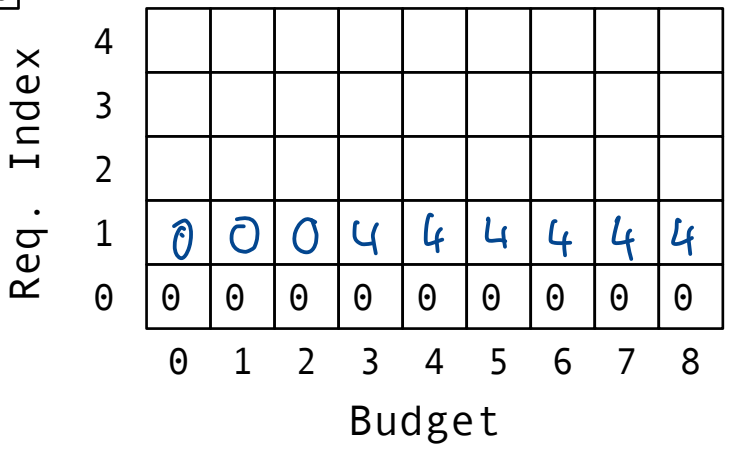
i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6

max value for servicing requests up to index 2 w/ time budget of 4



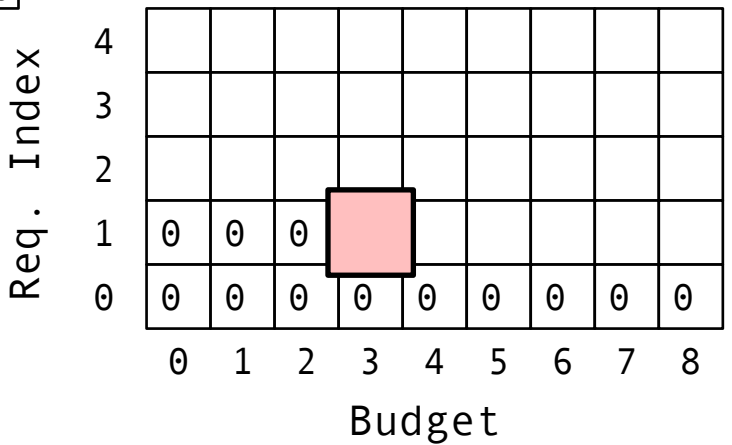
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



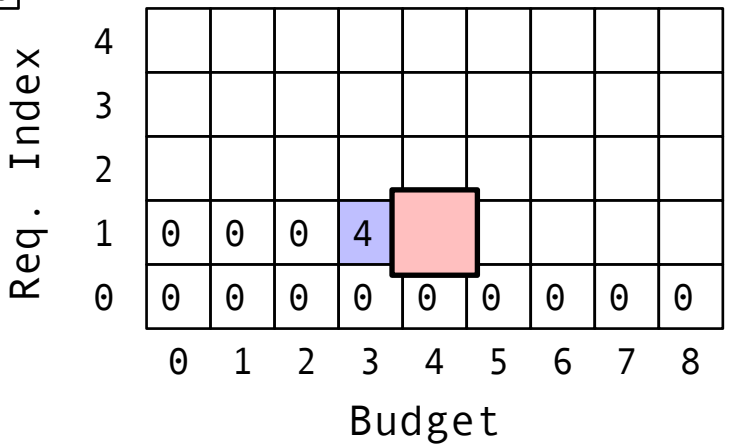
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



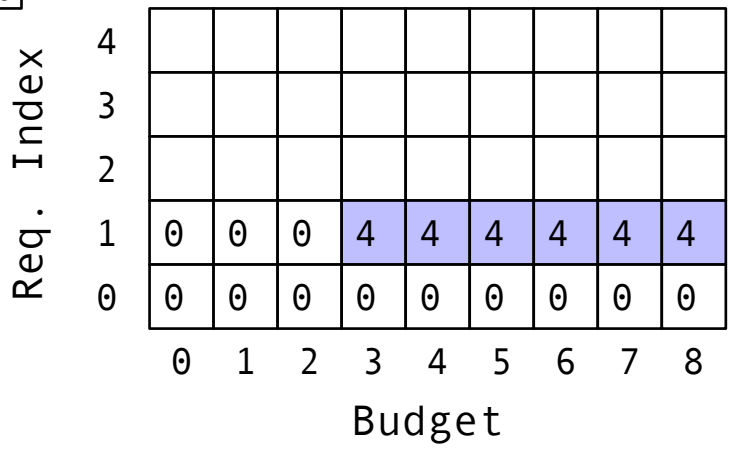
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



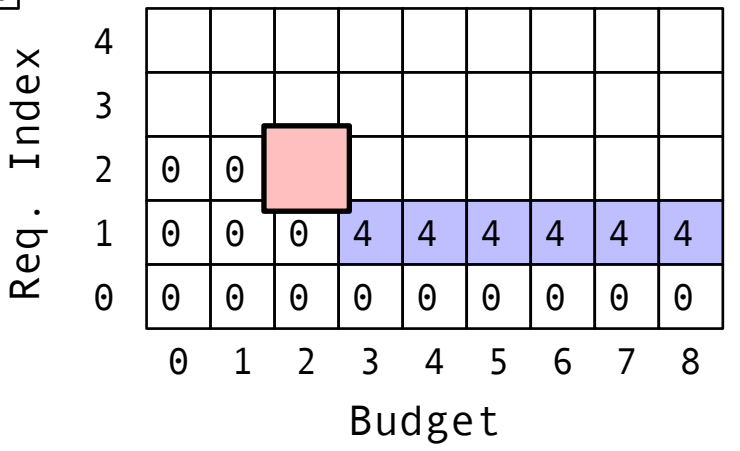
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



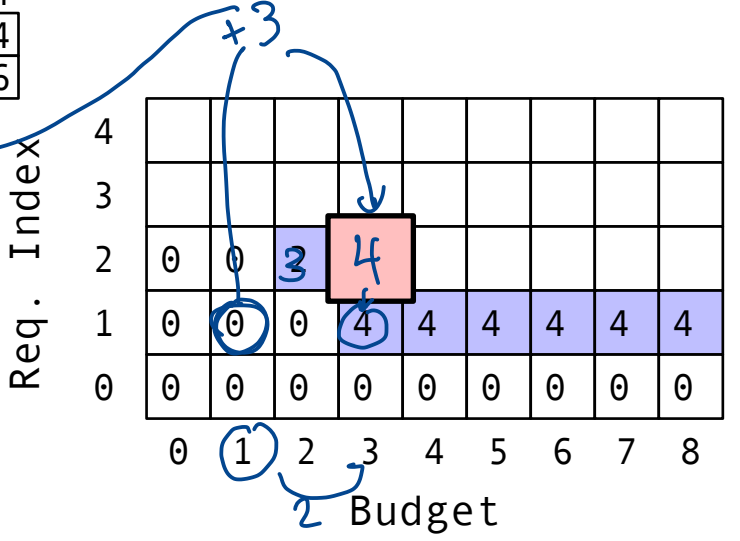
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



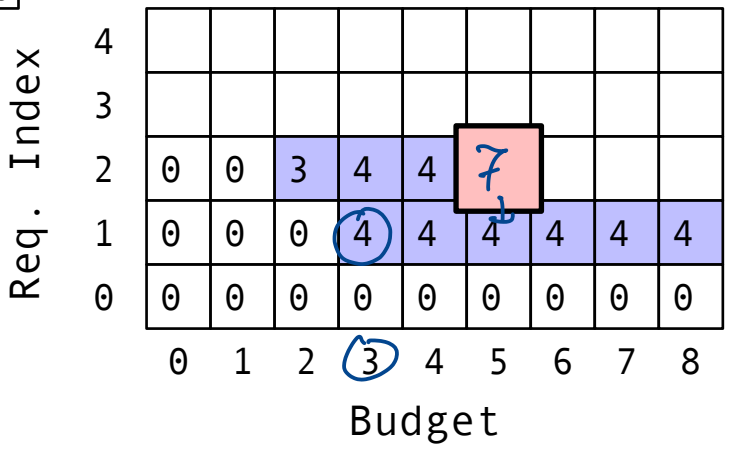
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



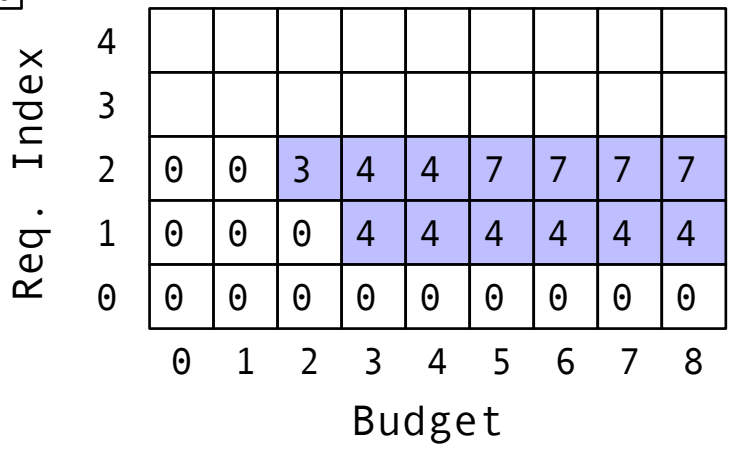
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



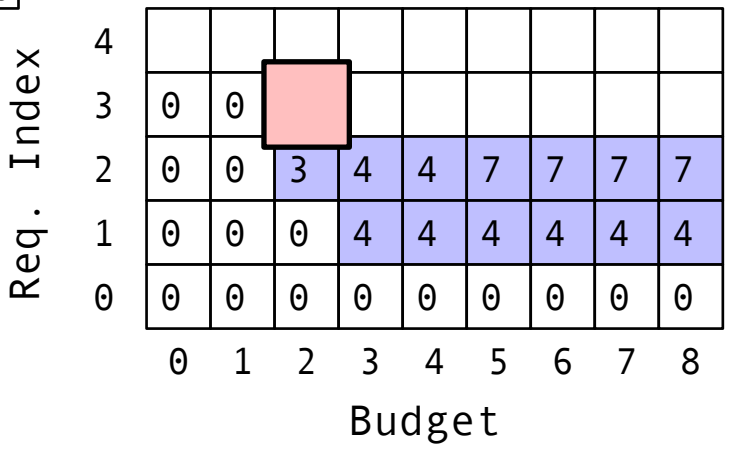
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



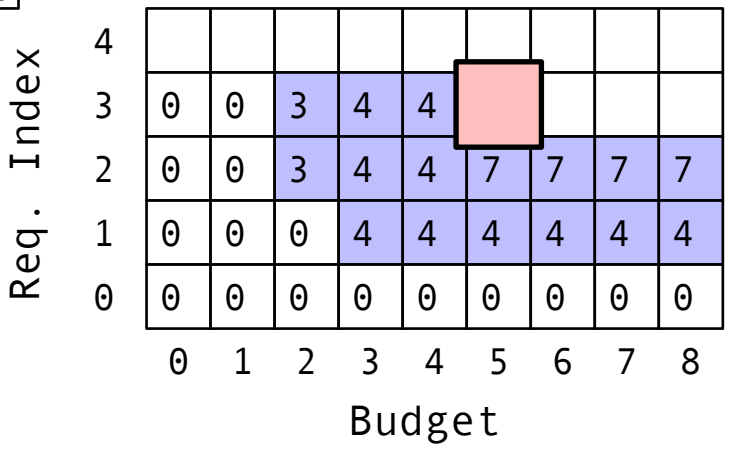
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



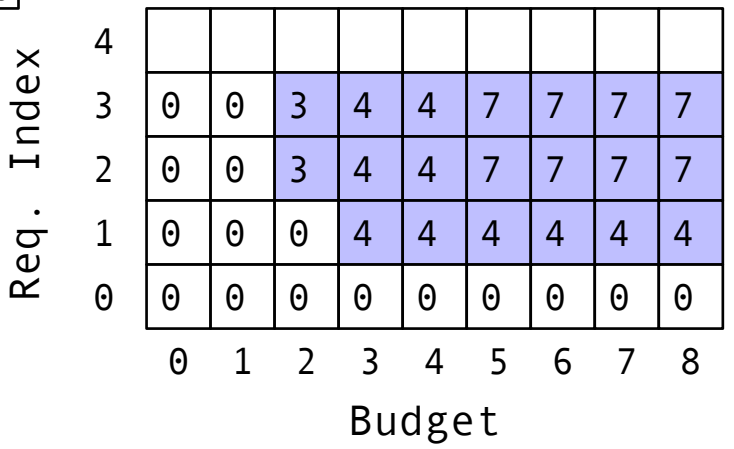
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



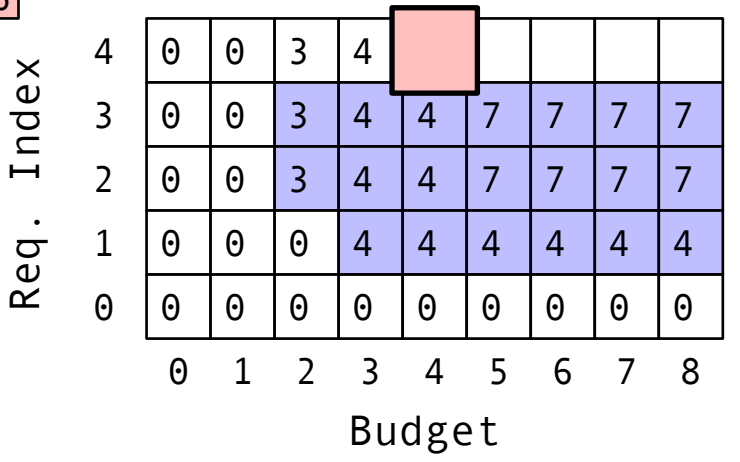
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



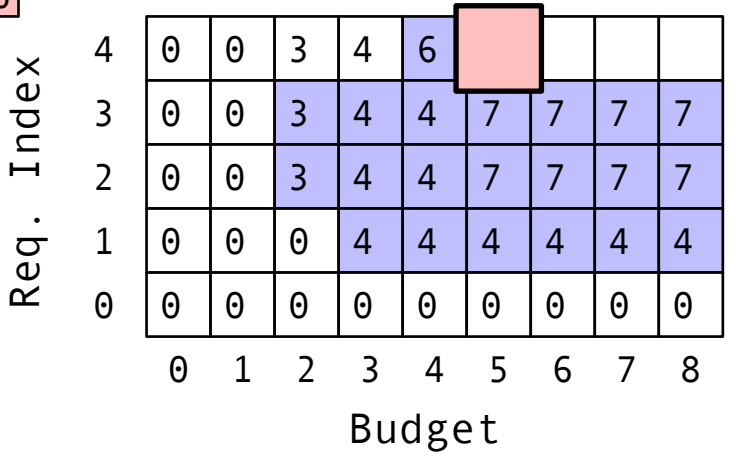
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



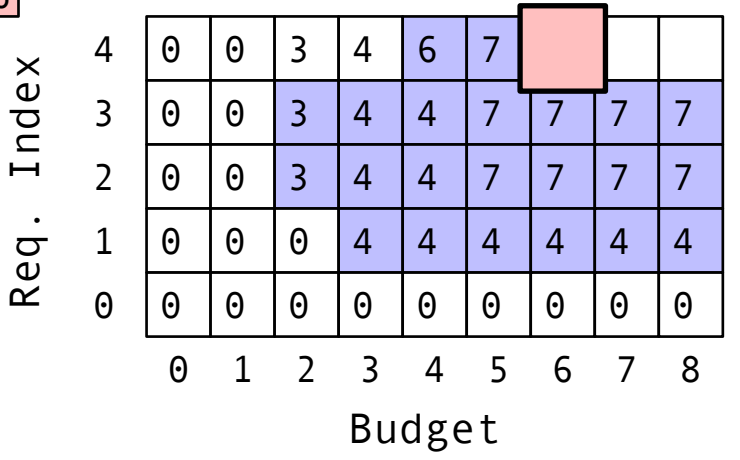
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



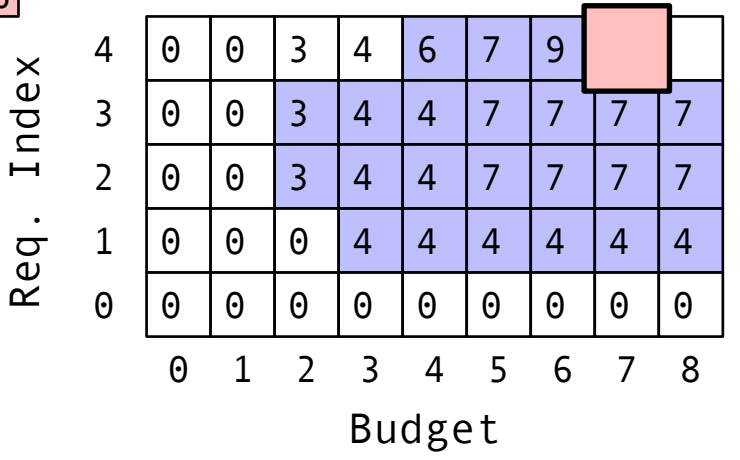
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



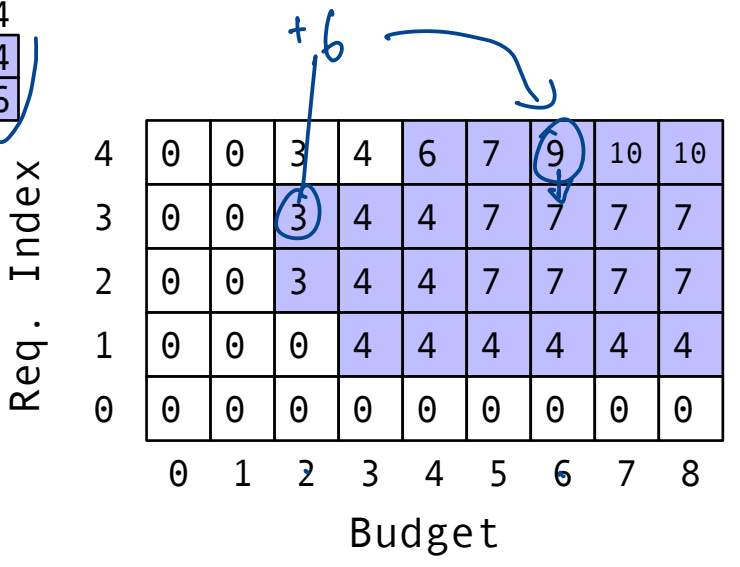
Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



Requests: B=8

i	1	2	3	4
b[i]	3	2	5	4
v[i]	4	3	5	6



Pseudocode

```
FindMax(R, n, B):
  max <- new 2d array of dimensions n+1, B+1
  set max[0, C] <- 0 for C = 0 to B
  for j from 1 to n
    (b, v) <- i-th request in R
    for C from 0 to B
      if b <= C then
        max[j, C] <- Max(v + max[j-1, C-b], max[j-1,C])
      else
        max[j, C] <- max[j-1, C]
  return max[n, B]
```