What we learned so far:

1. Euler Circuit Problem. Theorem and algorithm
2. Greedy Algorithms
   1. Interval scheduling and interval partitioning problems. Algorithms, correctness, running time.
   2. MST: Kruskal and Prim Algorithms, theorem, running times with different data structures.
   3. Single-source shortest path: Dijkstra algorithm: algorithm, running time and correctness.
3. Divide and Conquer: divide, recur, and conquer
   1. Merge sort: algorithm and running time
   2. Counting Inversions: algorithms and running: two jobs are easier than one.

Mid-term covers week 1 to week 6. (not including dynamic programming algorithms)

How to prepare mid-term exam: go through notes/tutorials/in-class exercises. If you want to get A+, read related chapters in the text book.

Dynamic Programming

Basic Idea

* Break problems into subproblems and combine their solutions into solutions to larger problems.

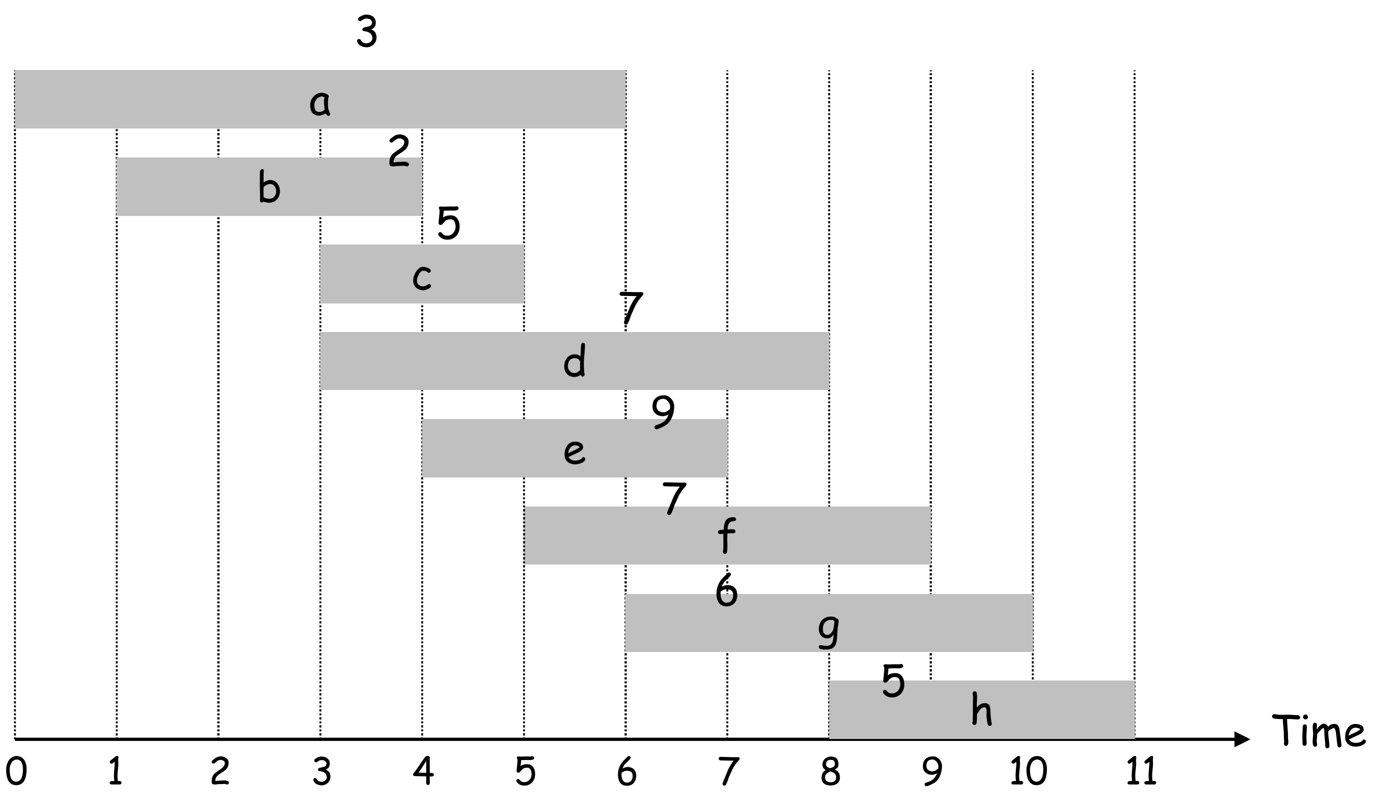
In contrast to divide-and-conquer, dynamic programming uses memorization: each sub-problem is solved only once and the result of each sub-problem is stored in a table

1. Weighted Interval Scheduling

Weighted Interval Scheduling

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find a subset of pairwise compatible (nonoverlapping) jobs with maximal total value.



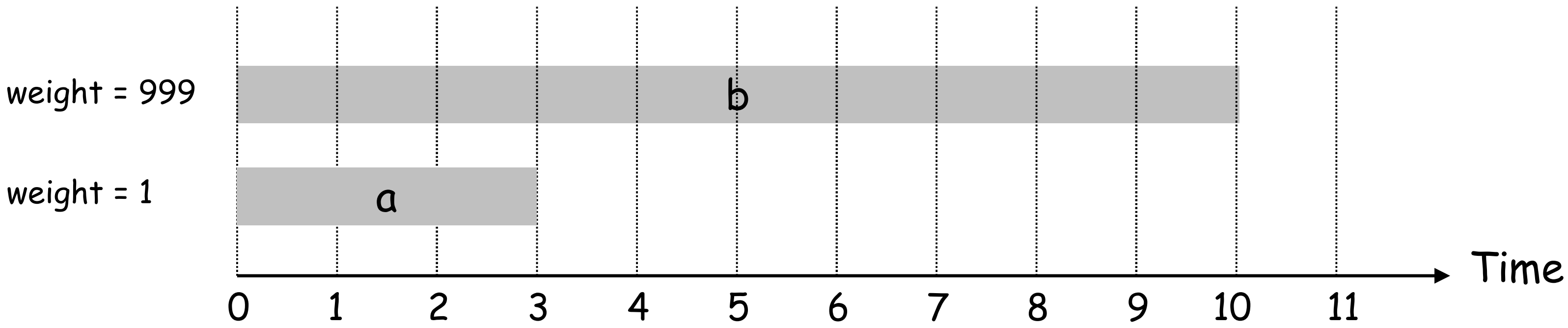
Unweighted Interval Scheduling Review

Unweighted Interval scheduling (all weights=1) => select as many compatible jobs as possible.

Recall. Greedy algorithm works if all weights are 1 .

- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously selected jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.



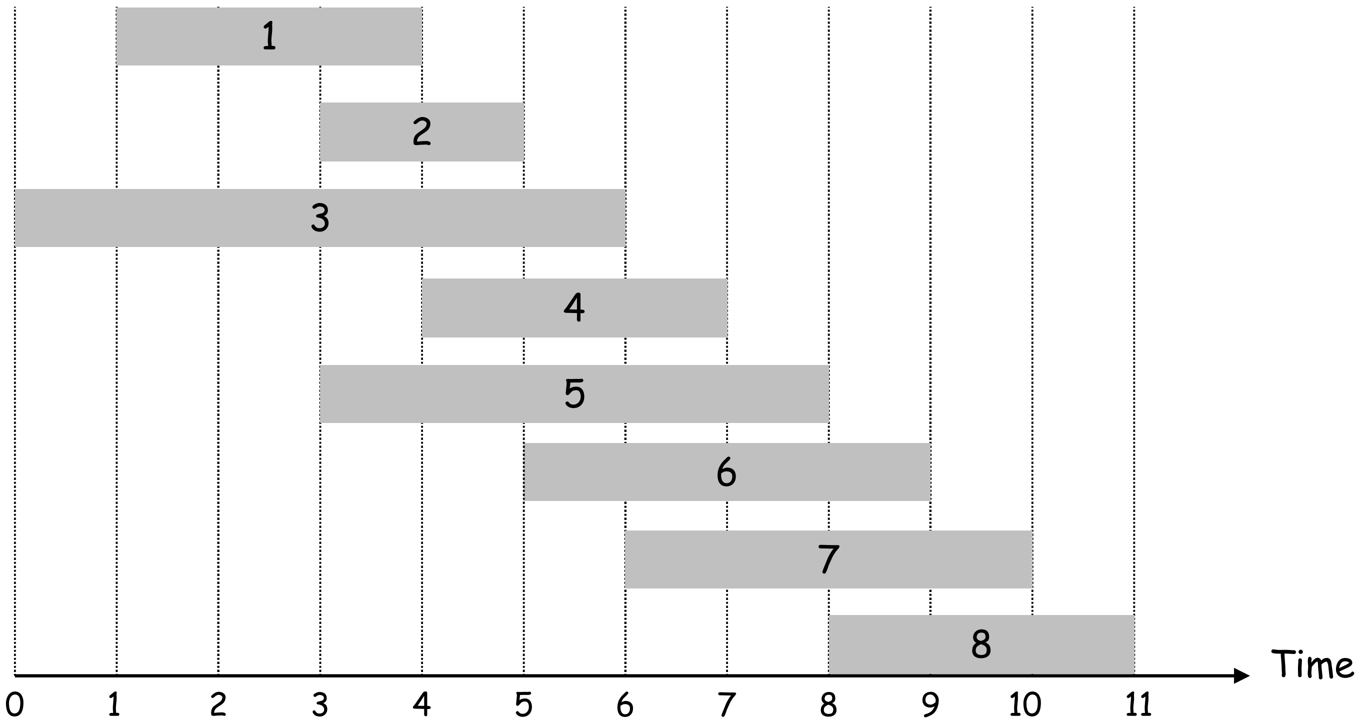
Weighted Interval Scheduling

Notation. Label jobs by finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. $p(j)$ = largest index $i < j$ such that job i is compatible with j.

Ex: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.

Observation: all jobs $p(j)+1, \ldots, j-1$ are incompatible with j; and all jobs $1, \ldots, p(j)$ are compatible with j.



Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max \left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Recursive Algorithm:

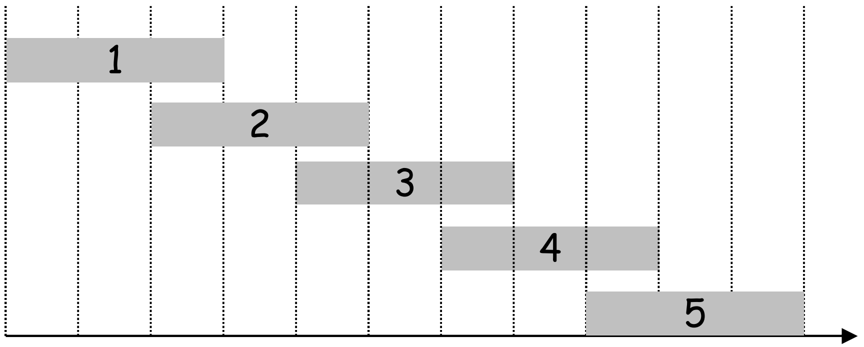
```
Compute-Opt(n)
    if n=0 then
      return 0
    else
        return max {v_n+Compute-Opt(p(n)), Compute-Opt(n-1)}
```

Running time: $>2^{n/2}$.

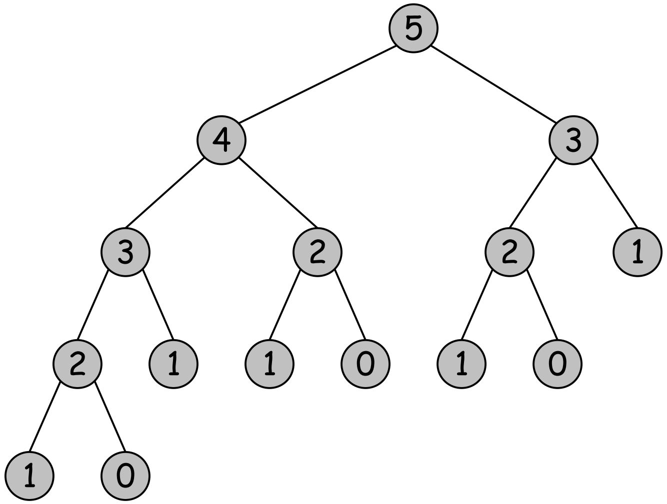
(not required)

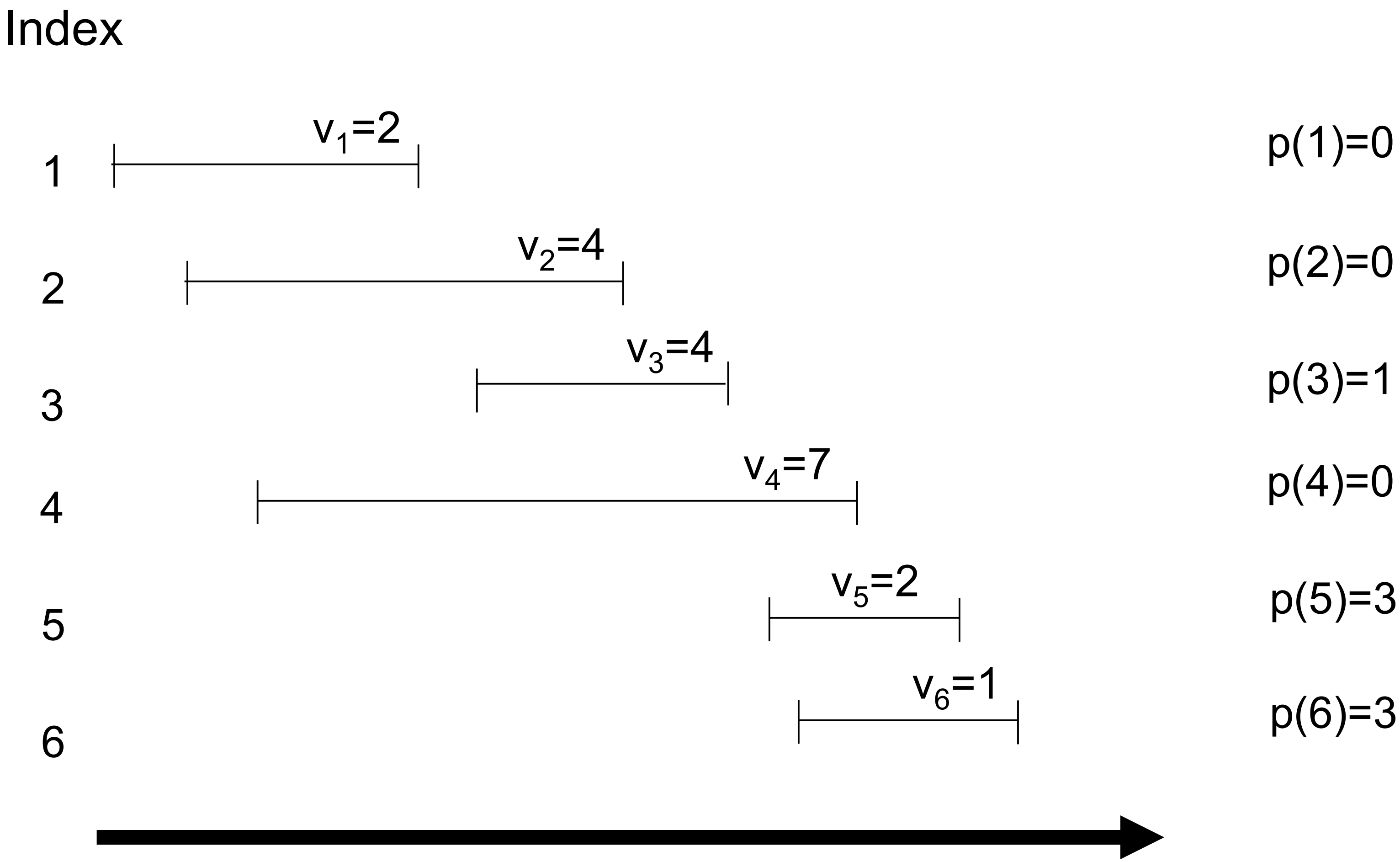
Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2




Index
1
$v_1=2$
p(1)=0
2
$v_2=4$
p(2)=0
3
$v_3=4$
p(3)=1
4
$v_4=7$
p(4)=0
5
$v_5=2$
p(5)=3
6
$v_6=1$
p(6)=3

Weighted Interval Scheduling: Bottom-Up

Input: $n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n, v_1, v_2, \ldots, v_n$

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p(1), p(2), \ldots, p(n)$

M[0]=0; */* Memoization*

for j = 1 to n ***do***

M[j] = max $\{ v_j + M[p(j)], M[j-1] \}$

if *(M[j] == M[j-1])* ***then*** *B[j]=0* ***else*** *B[j]=1 /*for backtracking*

*m=n; /*** Backtracking*

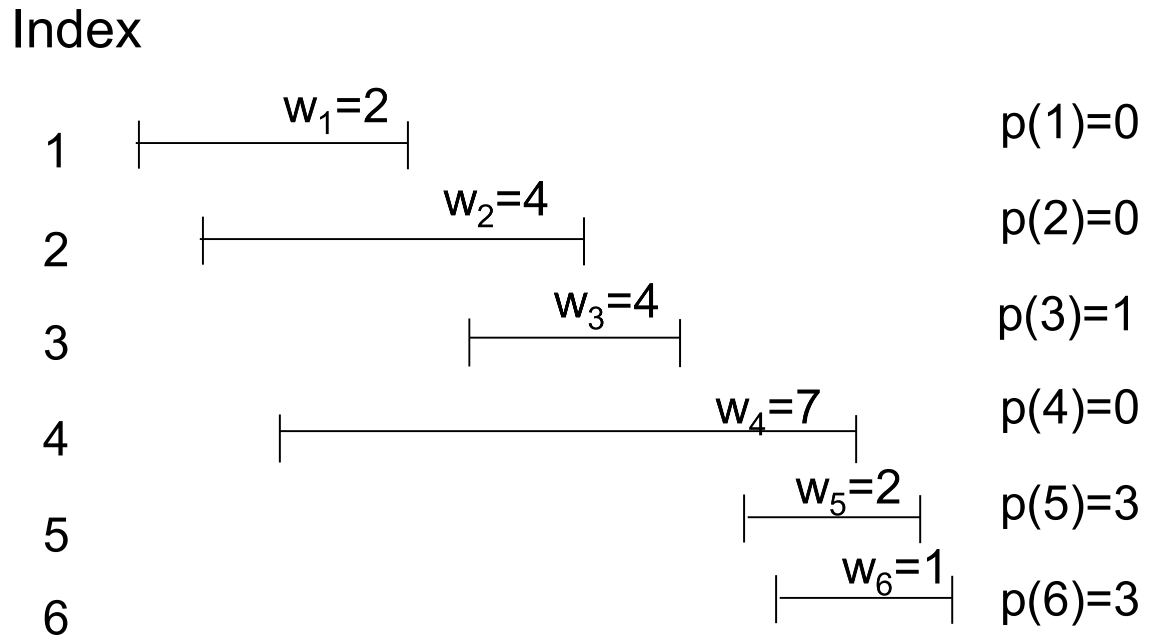
B[j]=0 indicating job j is not selected.
B[j]=1 indicating job j is selected.

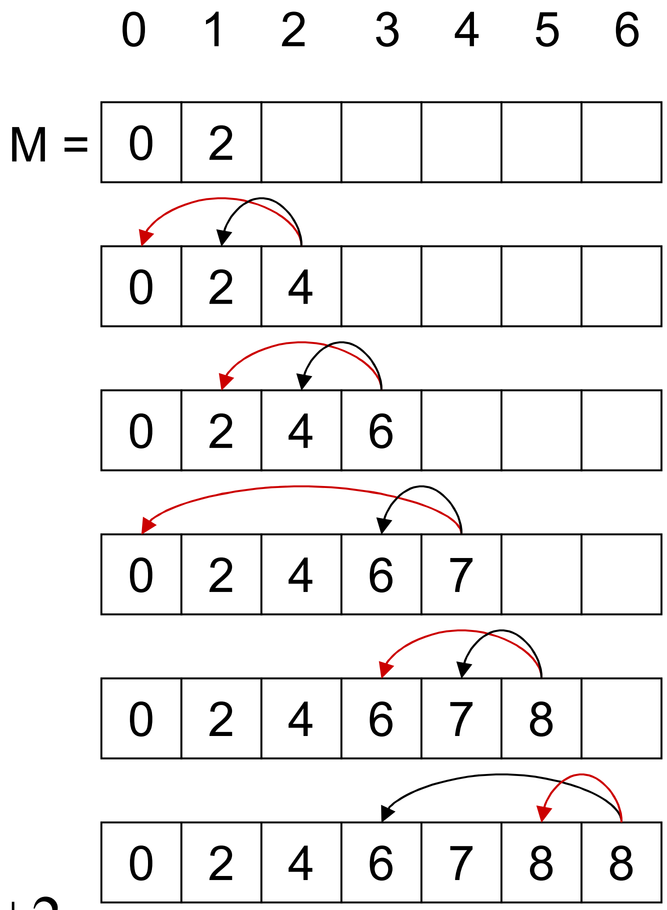
while *(m ≠0) {* ***if*** *(B[m]==1)* ***then***

print job m; m=p(m)

else

m=m-1 }





$M[j] = \max \{ v_j+M[p(j)], M[j-1]\}$

M[2]=w2+M[0]=4+0; M[3]=w3+M[1]=4+2;

M[4]=W4+M[0]=7+0; M[5]=W5+M[3]=2+6;

M[6]=w6+M[3]=1+6<8;

j:	0	1	2	3	4	5	6
B:	0	1	1	1	1	1	0

Backtracking: job1, job 3, job 5

Computing p()'s in O(n) time

p()'s can be computed in O(n) time using two sorted lists, one sorted by finish time and the other sorted by start time.

Start time: b(0, 5), a(1, 3), e(3, 8), c(5, 6), d(6, 8)

Finish time: a(1, 3), b(0,5), c(5,6), d(6,8), e(3,8)

p(d)=c, p(c)=b, p(e)= a, p(a)=0, p(b)=0. *(See demo)*

Time complexity

- Sorting the jobs: O(n log n)
- Computing p(): O(n) time after sorting all the jobs based on the starting times (O (n log n)) .
- The whole loop O(n) (each pass: O(1))
- The backtracking O(n)

•Time complexity: *O(n log n)* including sorting.

Example 2:

v(a)=2, v(b)=3, v(c)=5, v(d) =6, v(e)=8.8

Start time: b(0, 5), a(1, 3), e(3, 8), c(5, 6), d(6, 8)

Finish time a(1, 3), b(0,5), c(5,6), d(6,8), e(3,8)

p(d)=c, p(c)=b, p(e)= a, p(a)=0, p(b)=0.

Solution: M[0]=0, M[a]=2. M[b]=max{2, 3+M[p(b)]}=3.

M[c]=max{3, 5+M[p(c)]}=5+M[b]=8.

M[d]=max{8, 6+M[p(d)]}=6+M[c]=6+8=14.

M[e]=max{14, 8.8+M[p(e)]}=max{14, 8.8+M[a]}=max {14, 10.8}=14.

Backtracking: b, c, d.

Job:	a	b	c	d	e
B:	1	1	1	1	0

Chanllenge Q1: Billboard placement (Not Required)

You are trying to decide where to place billboards on a highway that goes East-West for M miles. The possible sites for billboards are given by numbers $x_1, \ldots, x_n$, each in the interval [0, M]. If you place a billboard at location x_i, you get a revenue r_i.

You have to follow a regulation: no two of the billboards can be within less than or equal to 5 miles of each other.

You want to place billboards at a subset of the sites so that you maximize your revenue subject to this constraint.

How?

2. Manhattan Tourist Problem

Manhattan Tourist Problem: Formulation

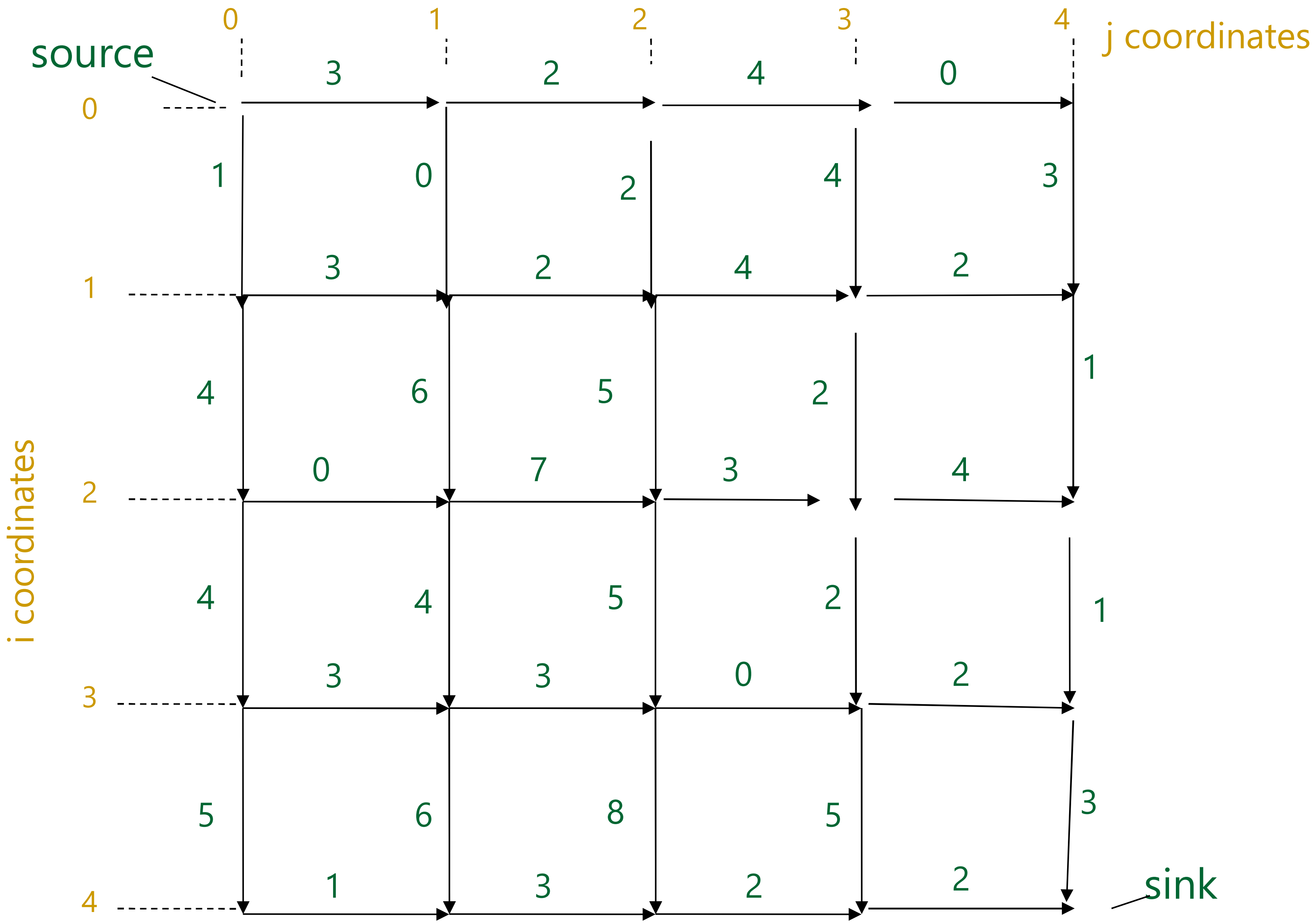
Goal: Find the longest path in a weighted grid

Input: A weighted grid **G** with two distinct vertices, one labeled “source” and the other labeled “sink”.

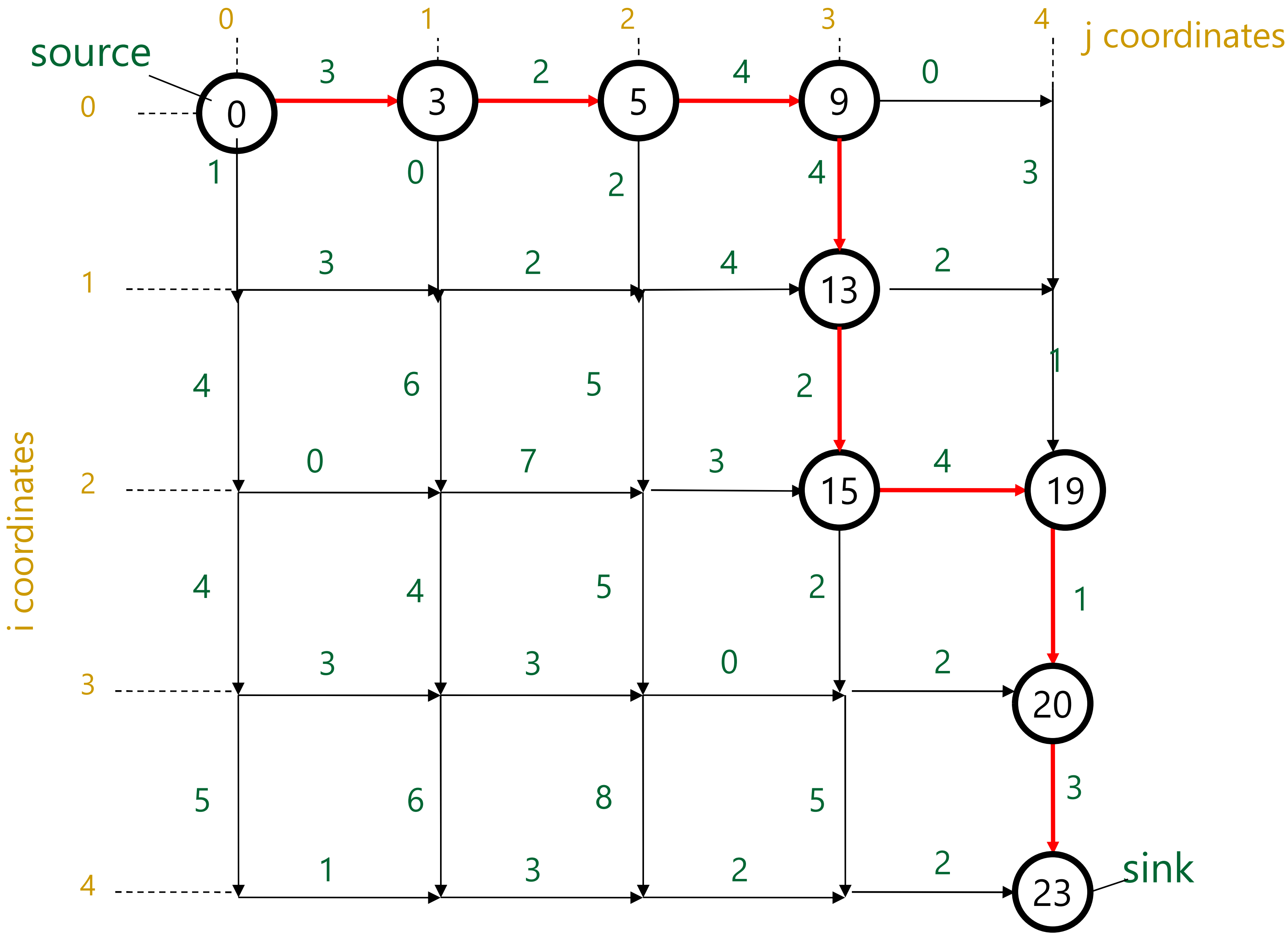
(We can only go from left to right and from top to bottom.)

Output: A longest path in **G** from “source” to “sink”

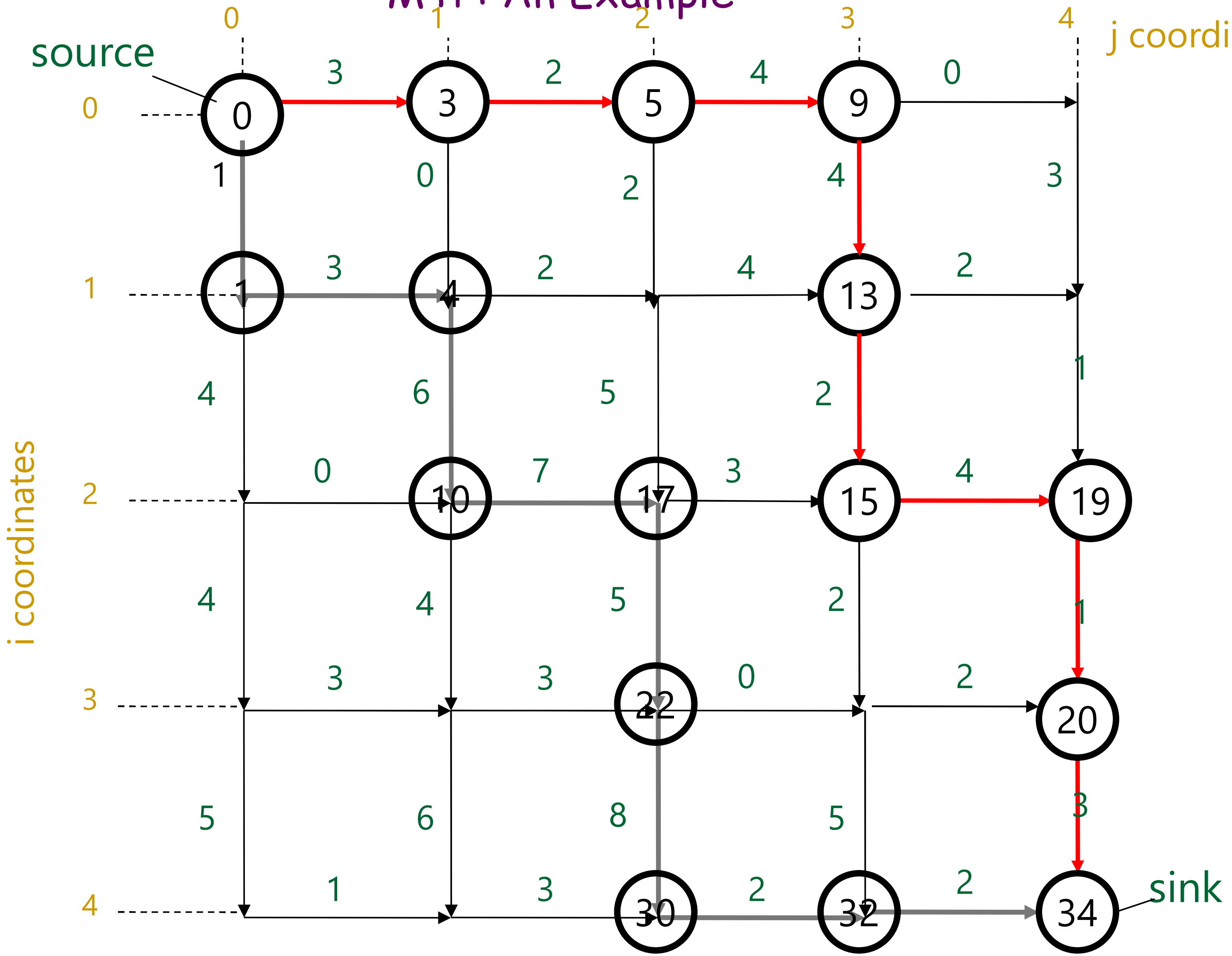
MTP: An Example



MTP: An Example



MTP: An Example



MTP: Recurrence

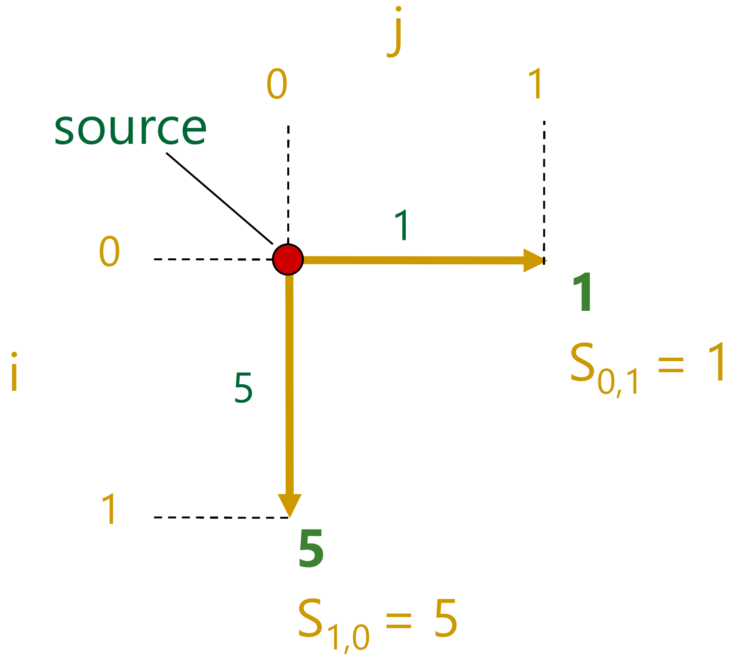
Computing the score for a point (i,j) by the recurrence relation:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} + \text{weight of the edge between } (i-1, j) \text{ and } (i, j) \\ s_{i,j-1} + \text{weight of the edge between } (i, j-1) \text{ and } (i, j) \end{cases}$$

the running time is **n x m** for a **n** by **m** grid

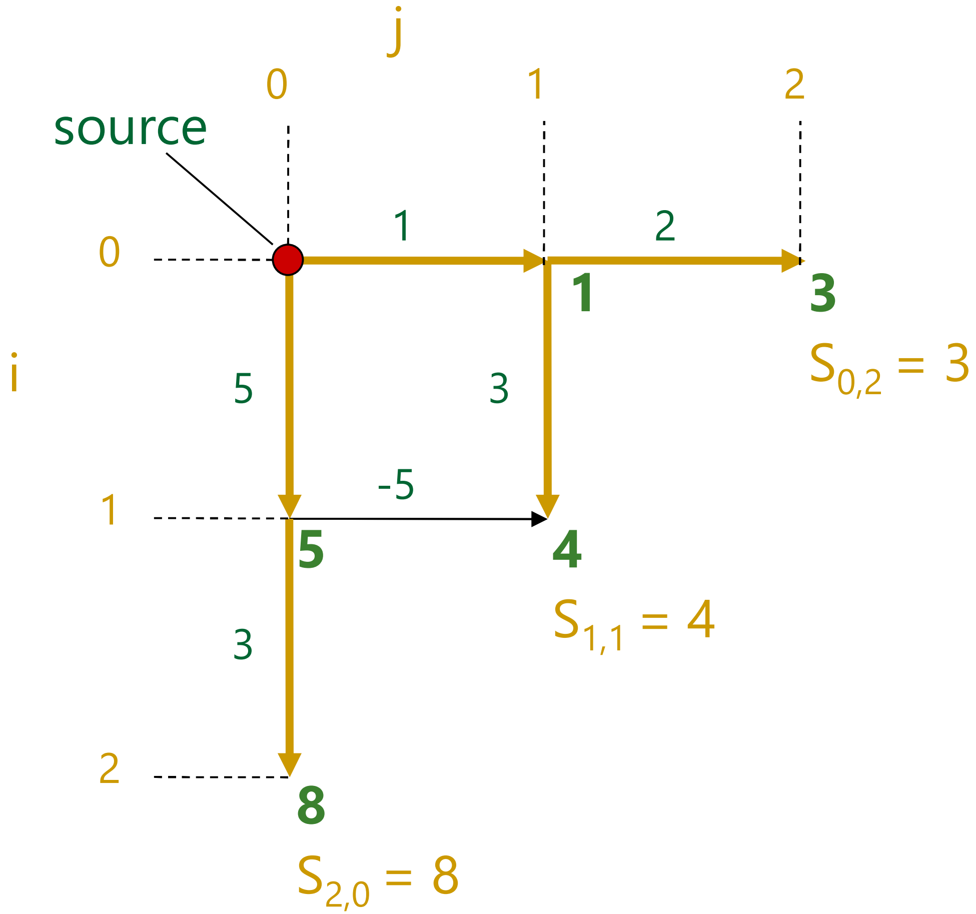
(**n** = # of rows, **m** = # of columns)

MTP: Dynamic Programming

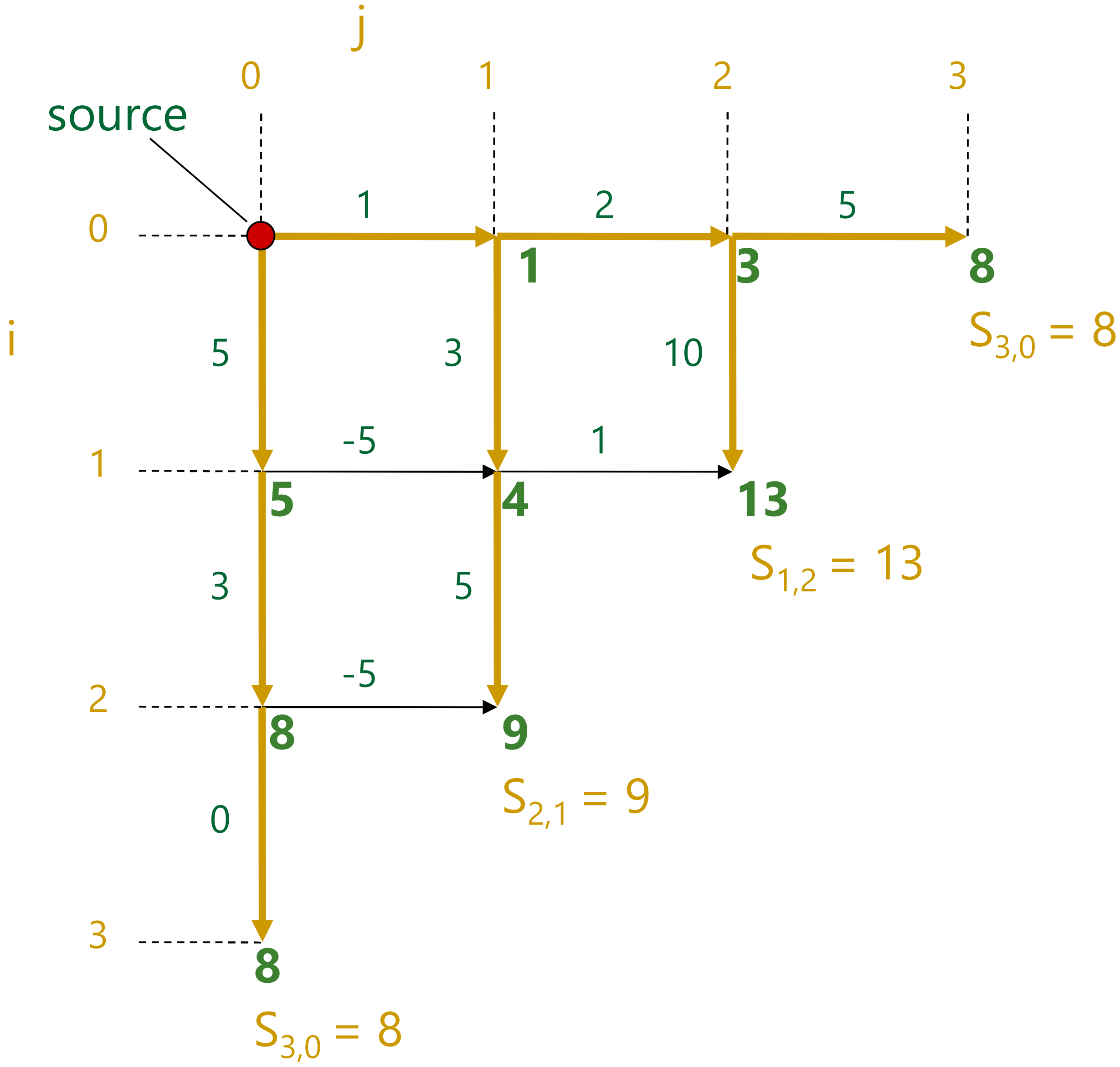


- Instead of recursion, store the result in an array **S**

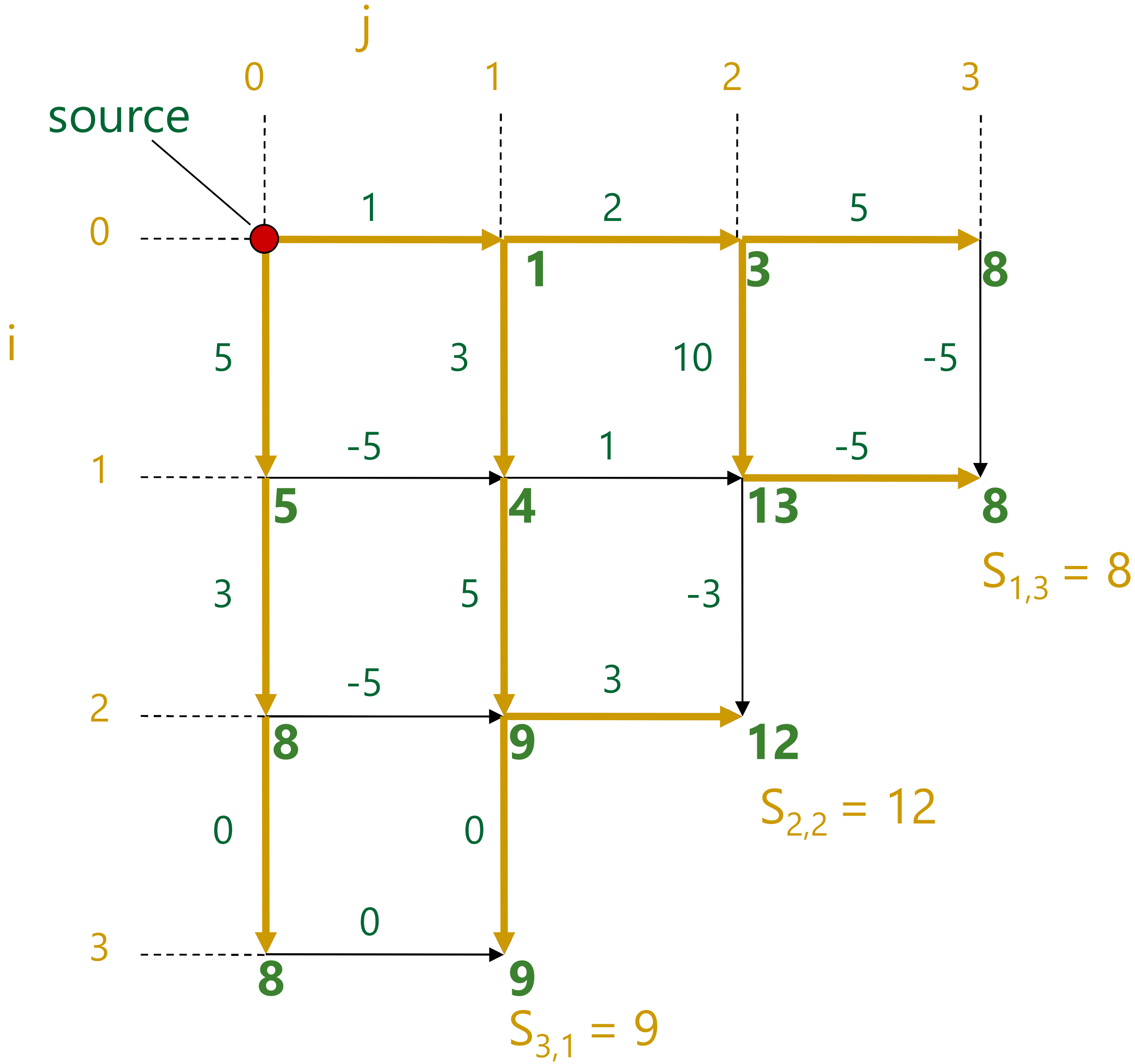
MTP: Dynamic Programming



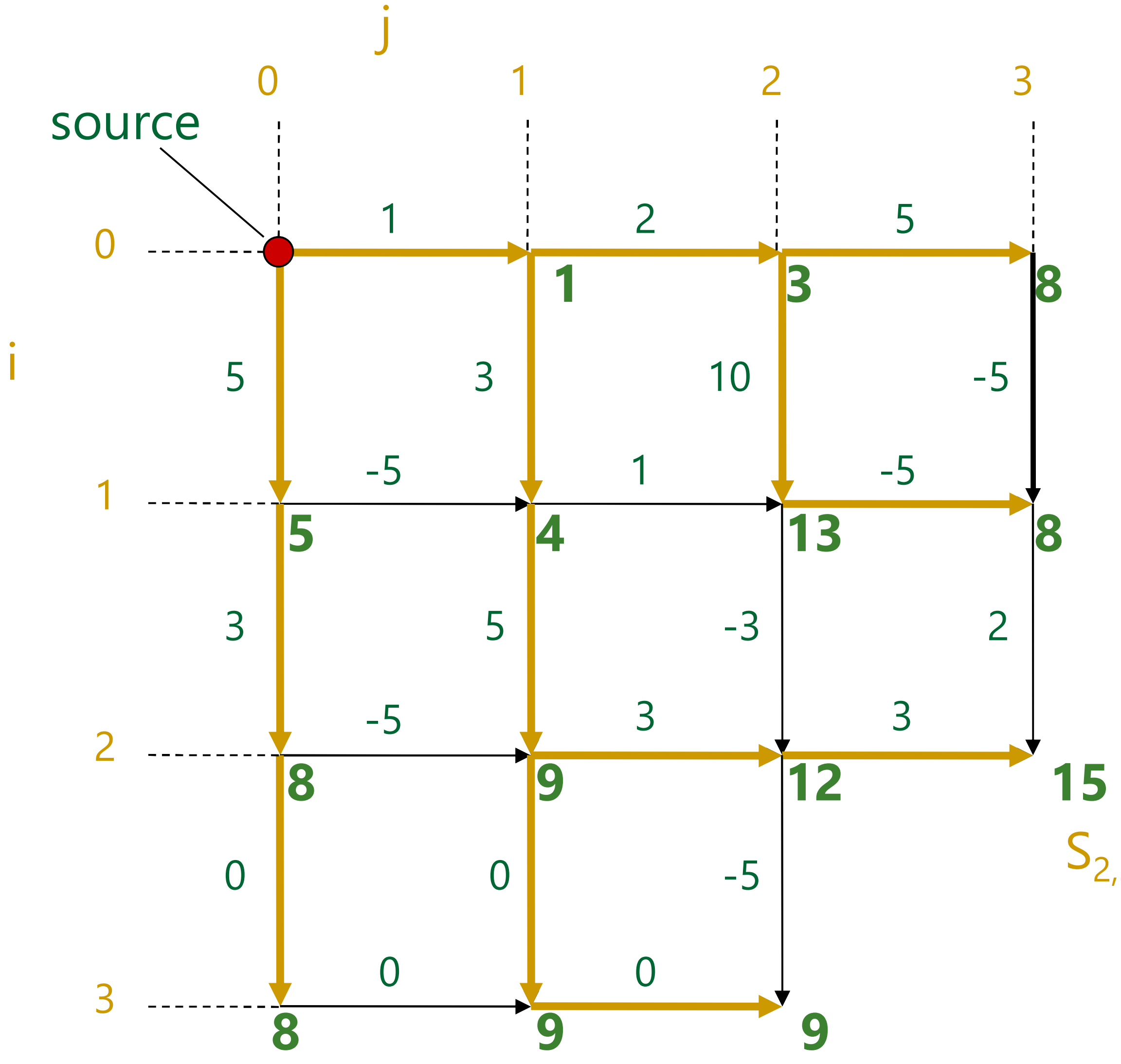
MTP: Dynamic Programming



MTP: Dynamic Programming



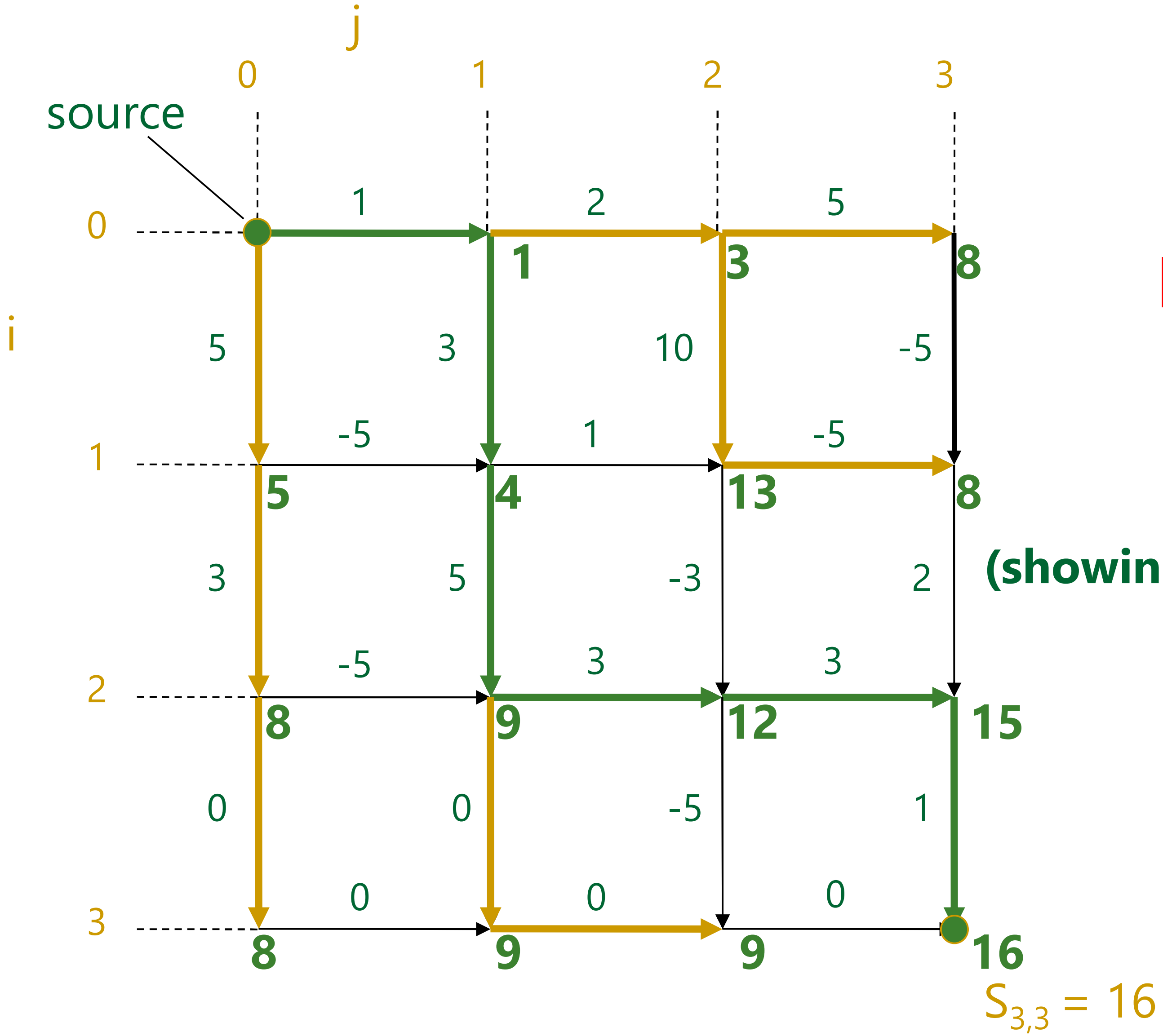
MTP: Dynamic Programming (cont'd)



$S_{2,3} = 15$

$S_{3,2} = 9$

MTP: Dynamic Programming (cont'd)



Done!

(showing all back-traces)

Exercises

Exercise 1: For the weighted interval scheduling problem, there are eight jobs with starting time and finish time as follows: j1=(0, 8), j2=(2, 3), j3=(3, 6), j4=(5, 9), j5=(8, 12), j6=(9, 11), j7=(10, 13) and j8=(11, 16). The weight for each job is as follows: v1=3.5, v2=2.0, v3=3.0, v4=3.0, v5=6.5, v6=2.5, v7=12.0, and v8=8.0.

Find a maximum weight subset of mutually compatible jobs. (Backtracking process is required.) (You have to compute p()'s. The process of computing p()'s is NOT required.)

Exercise: Write the pseudo codes for the Manhattan Tourist problem. Please pay attention to the backtracking process.