COMP 3011
DESIGN AND ANALYSIS OF ALGORITHMS
FALL 2024

Dynamic Programming

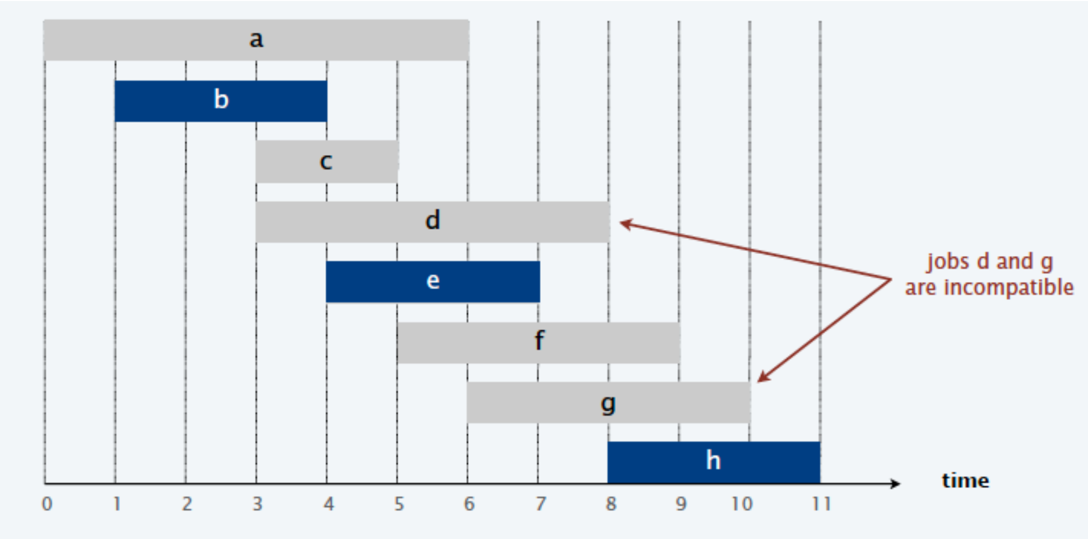
LI Bo
Department of Computing
The Hong Kong Polytechnic University

DYNAMIC PROGRAMMING

Algorithmic Paradigms

fancy name for caching intermediate results in a table for later reuse

- ***Greedy***. Process the input in some order, myopically making irrevocable decisions.
- ***Divide-and-conquer***. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.
- ***Dynamic programming***. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.



Interval Scheduling Problem

Given a set of jobs $J = \{1, 2, \ldots, n\}$

- Job j starts at s_j and finishes at $f_j \geq s_j$.
- Two jobs are compatible if they don't overlap.

Goal: find maximum subset of mutually compatible jobs.

What if different jobs bring different values?

Weighted Interval Scheduling Problem

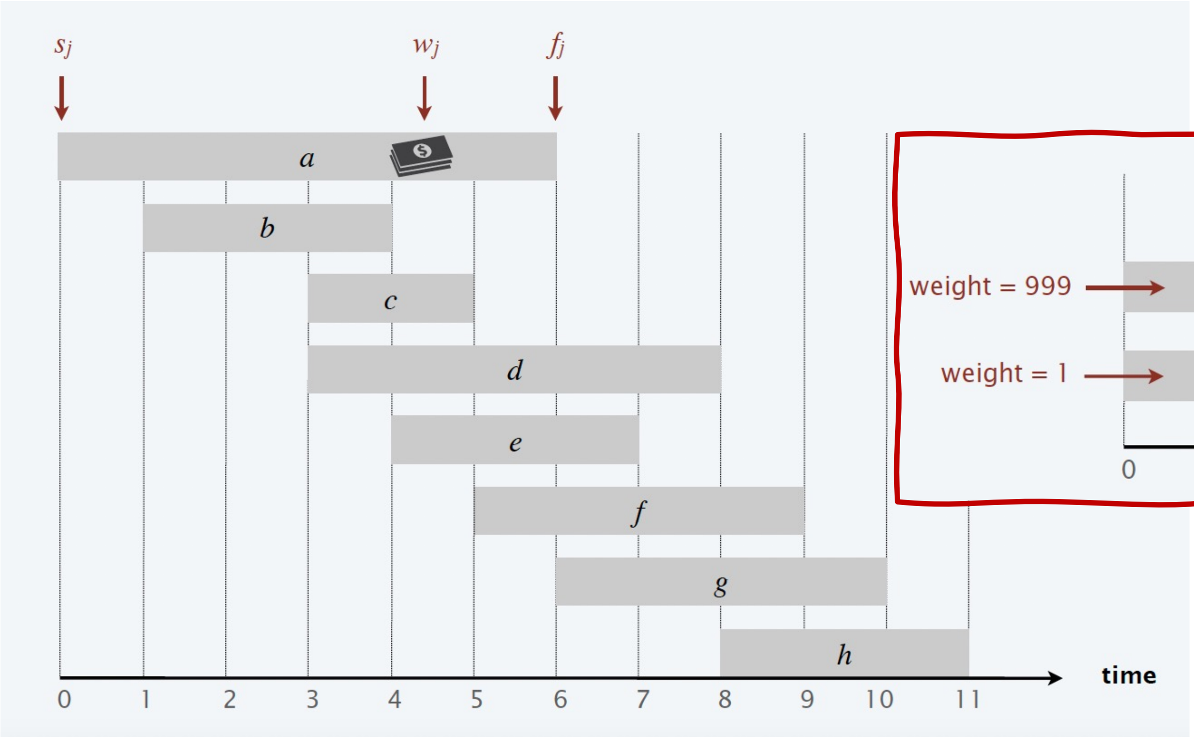
Given a set of jobs $J = \{1, 2, \ldots, n\}$

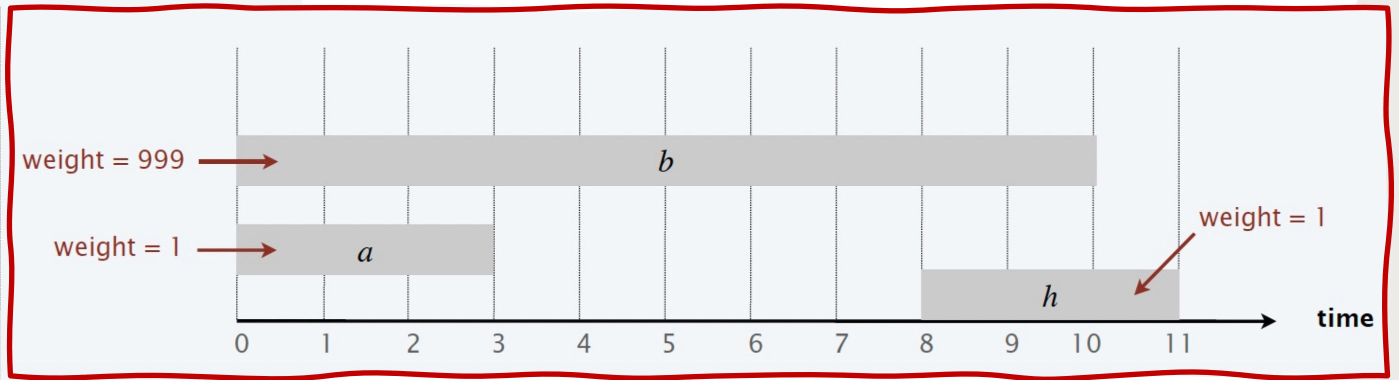
- Job j starts at s_j, finishes at f_j, and has weight $w_j > 0$.
- Two jobs are compatible if they don't overlap.

Goal: find max-weight subset of mutually compatible jobs.

Earliest finish-time first

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





Weighted Interval Scheduling Problem

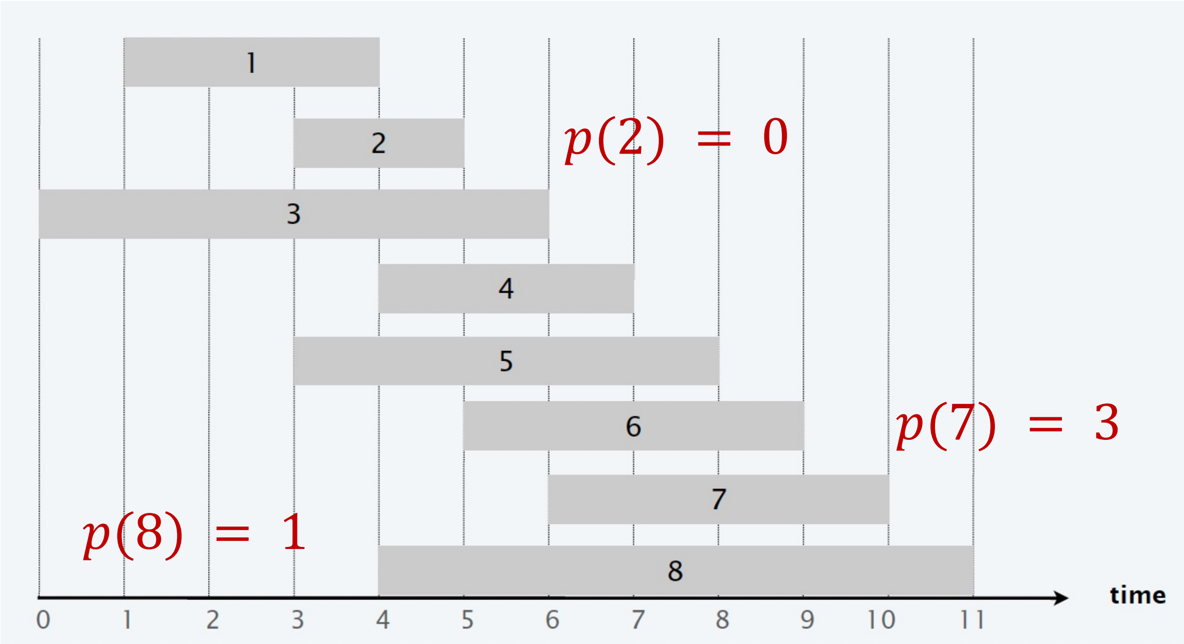
i is rightmost interval that ends before j begins

Jobs are in ascending order of finish time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

***Definition*.**

- $p(j)$ = largest index $i < j$ such that job i is compatible with j.
- $OPT(j)$ = max weight of any subset of mutually compatible jobs for subproblem consisting only of jobs $1, 2, \ldots, j$.

***Goal*.** $OPT(n)$ = max weight of any subset of mutually compatible jobs.



***Case 1*.** $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j-1$.

***Case 2*.** $OPT(j)$ selects job j.

- Collect profit w_j.
- Can't use jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

Weighted Interval Scheduling Problem

Case 1. $OPT(j)$ does not select job j.

- Must be an optimal solution to problem consisting of remaining jobs $1, 2, \ldots, j - 1$.

Case 2. $OPT(j)$ selects job j.

- Collect profit w_j.
- Can't use jobs $\{p(j) + 1, p(j) + 2, \ldots, j - 1\}$.
- Must include optimal solution to problem consisting of remaining compatible jobs $1, 2, \ldots, p(j)$.

optimal substructure property

Bellman equation

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{OPT(j-1),\ w_j + OPT(p(j))\} & \text{if } j > 0 \end{cases}$$

Exponential running time!

Recursive algorithm
to compute $OPT(j)$:

```
COMPUTE-OPT(j)
IF (j = 0)
    RETURN 0.
ELSE
    RETURN max {COMPUTE-OPT(j – 1), w_j + COMPUTE-OPT(p[j]) }.
```

Case 1

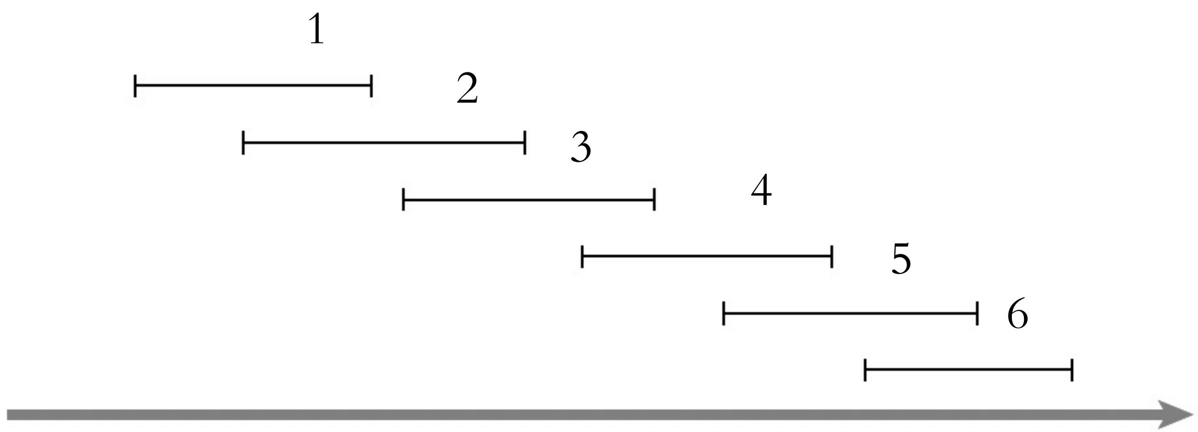
Case 2

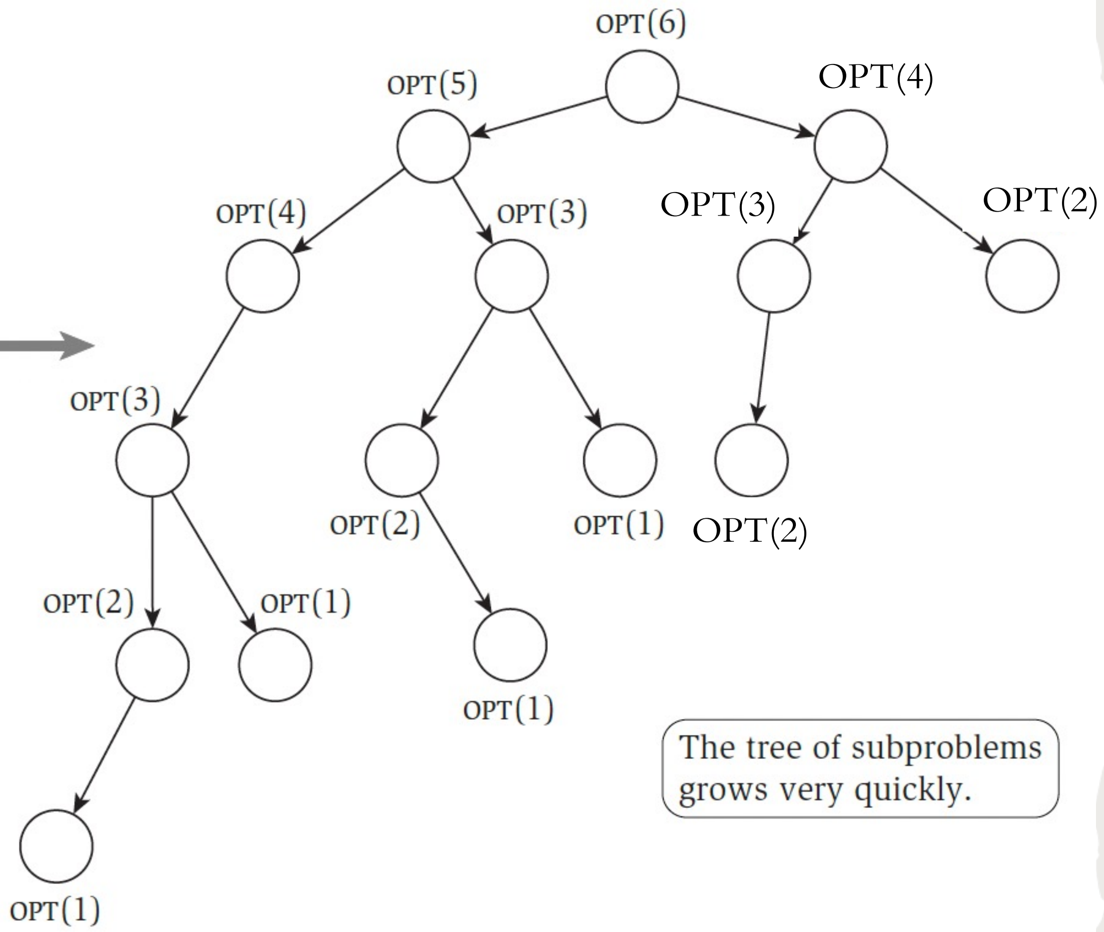
How to compute $p(j)$?

binary search!

Weighted Interval Scheduling Problem

Example:





$$p(1) = p(2) = 0, p(j) = j - 2 \text{ for } j \geq 3$$

Grows like Fibonacci sequence!

Memoizing the Recursion

Observation: the recursive algorithm Compute-Opt is really only solving $n+1$ different subproblems: Compute-Opt(0), Compute-Opt(1), . . . , Compute-Opt(n).

Technique: store the value of Compute-Opt the first time we compute it and then simply use this precomputed value in place of all future recursive calls.

Memoization

TOP-DOWN($n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$)

Sort jobs by finish time and renumber so that $f_1 \le f_2 \le \ldots \le f_n$.

Compute $p[1], p[2], \ldots, p[n]$ via binary search.

$M[0] \leftarrow 0$. global array

RETURN M-COMPUTE-OPT(n).

M-COMPUTE-OPT(j)

IF ($M[j]$ is uninitialized)

$M[j] \leftarrow \max \{$ M-COMPUTE-OPT $(j-1)$, w_j + M-COMPUTE-OPT($p[j]$) $\}$.

RETURN $M[j]$.

Memoizing the Recursion

Theorem. Memoized version of algorithm takes $O(n \log n)$ time.

Proof.

- Sort by finish time: $O(n \log n)$ via Mergesort.
- Compute $p[j]$ for each j: $O(n \log n)$ via binary search.

- M-COMPUTE-OPT(j): each invocation takes $O(1)$ time and either
  - (1) returns an initialized value $M[j]$.
  - (2) initializes $M[j]$ and makes two recursive calls
- How many M-Compute-Opt calls?
  - M has $n+1$ entries, which are initially empty.
  - Each time the procedure invokes the recurrence, issuing two recursive calls to M-Compute-Opt, it fills in a new entry, and hence increases the number of filled-in entries by 1.
  - There can be at most $O(n)$ calls to M-Compute-Opt.
- Overall running time of M-COMPUTE-OPT(n) is $O(n)$.

Question:
DP algorithm computes optimal value.
How to find optimal solution?

Keep track of an optimal solution
in addition to its value

E.g. maintain an additional array S so that $S[i]$ contains an optimal set of intervals among $\{1, 2, \ldots, i\}$.

Blow up the running time by an additional factor of $O(n)$:

- While a position in the M array can be updated in $O(1)$ time, writing down a set in the S array takes $O(n)$ time.

Memoizing the Recursion

To avoid this $O(n)$ blow-up, we do not explicitly maintain S, but rather recover the optimal solution from values saved in the array M after the optimum value has been computed.

Job j belongs to an optimal solution for the set of intervals $\{1, \ldots, j\}$ if and only if

$$v_j + OPT(p(j)) \geq OPT(j-1).$$

```
Find-Solution(j)
  If j = 0 then
    Output nothing
  Else
    If v_j + M[p(j)] ≥ M[j − 1] then
      Output j together with the result of Find-Solution(p(j))
    Else
      Output the result of Find-Solution(j − 1)
    Endif
  Endif
```

Theorem. Given the array M of the optimal values of the sub-problems, Find-Solution returns an optimal solution in $O(n)$ time.

- Find-Solution calls itself recursively only on strictly smaller values, it makes a total of $O(n)$ recursive calls;
- It spends constant time per call.

Bottom-up Dynamic Programming

Theorem. The bottom-up version takes $O(n \log n)$ time.

Iterative algorithm
to compute $OPT(j)$:

BOTTOM-UP$(n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n)$

Sort jobs by finish time and renumber so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

Compute $p[1], p[2], \ldots, p[n]$.

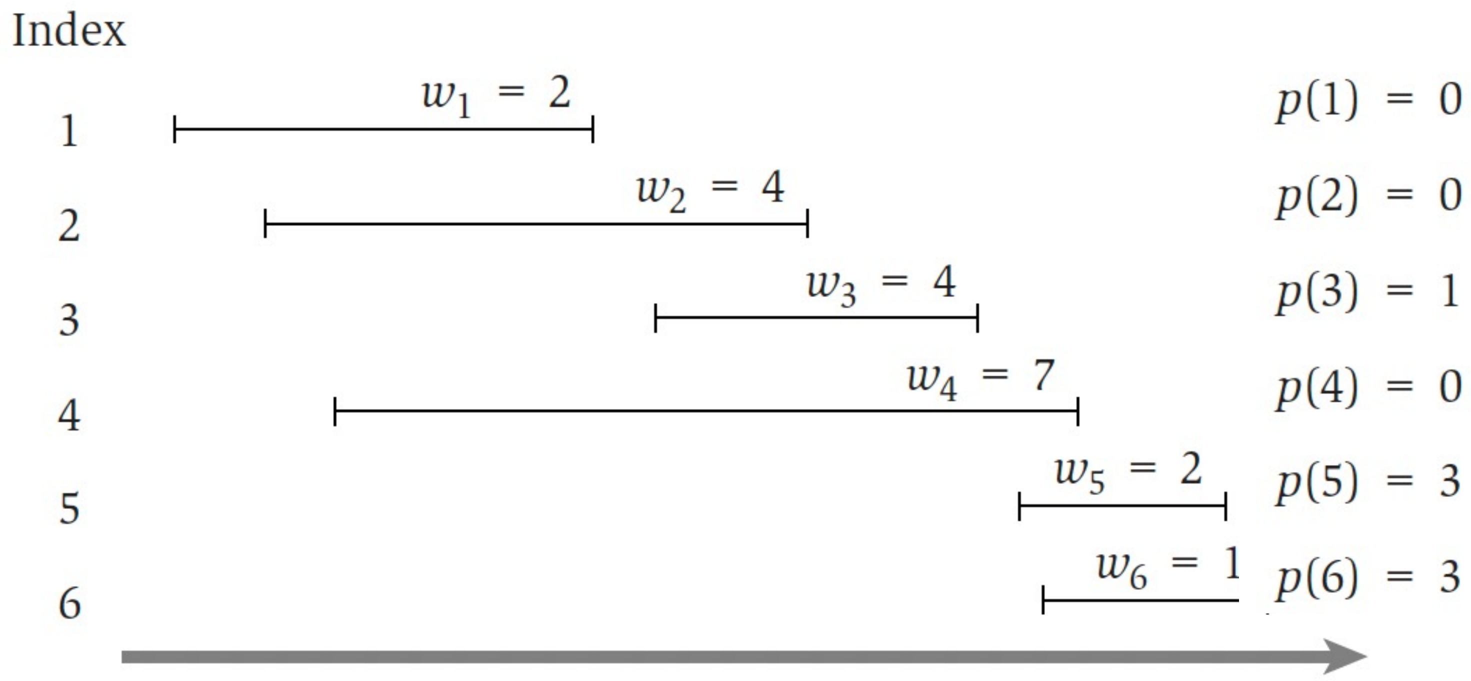
$M[0] \leftarrow 0.$ previously computed values

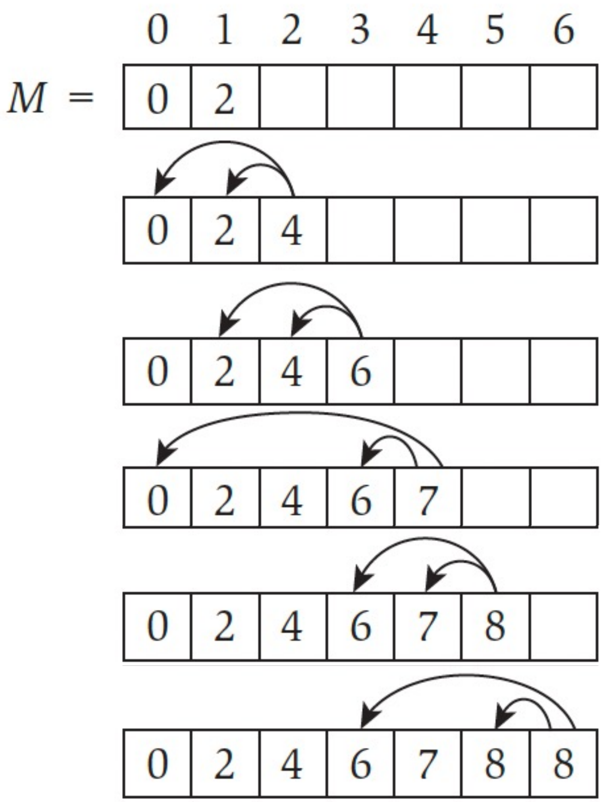
FOR $j = 1$ TO n

$\quad M[j] \leftarrow \max \{ M[j-1],\ w_j + M[p[j]] \}.$

Bottom-up Dynamic Programming

Example:





Algorithmic Paradigms

fancy name for caching intermediate results in a table for later reuse

- ***Greedy***. Process the input in some order, myopically making irrevocable decisions.
- ***Divide-and-conquer***. Break up a problem into independent subproblems; solve each subproblem; combine solutions to subproblems to form solution to original problem.
- ***Dynamic programming***. Break up a problem into a series of overlapping subproblems; combine solutions to smaller subproblems to form solution to large subproblem.

Two More Examples

Max Contiguous Subarray

5	–1	31	–61	59	26	–53	58	97	–93	–23	84

187

Given an array x of n integer (positive or negative), find a contiguous subarray whose sum is maximum.

***Approach 1. Brute-force algorithm*.**

- For each i and j: compute
$$a[i] + a[i+1] + \cdots + a[j]$$
- Takes $\Theta(n^3)$ time.

***Approach 2. Apply "cumulative sum" trick*.**

- Precompute cumulative sums:
$$S[i] = a[0] + a[1] + \cdots + a[i].$$
- Now $a[i] + a[i+1] + \cdots + a[j] = S[j] - S[i-1]$.
- Improves running time $\Theta(n^2)$.

Let $OPT(i)$ = max sum of any subarray of x whose rightmost index is i.

Goal: $\max_i OPT(i)$

take only element i

take element i together with best subarray ending at index $i-1$

$$OPT(i) = \begin{cases} x_1, & \text{if } i = 1 \\ \max\{x_i, OPT(i-1) + x_i\}, & \text{if } i > 1 \end{cases}$$

Bellman equation

Running time: $O(n)$.

Monotonically Increasing Subsequence

- Give an efficient algorithm to find the longest monotonically increasing sequence in a sequence of n numbers.
- Example: given (5,2,8,7,3,1,6,4)

  (2,6),

  (1,4),

  (2,3,6)

$(s_1, s_2, \cdots, s_{n-1}, s_n)$

What do we want to know about the first $n-1$ numbers?

What are the subproblems?

Monotonically Increasing Subsequence

- Give an efficient algorithm to find the longest monotonically increasing sequence in a sequence of n numbers.
- Example: given (5,2,8,7,3,1,6,4)

  (2,6), (1,4), (2,3,6)

$(s_1, s_2, \cdots, s_{n-1}, s_n)$

What do we want to know about the first $n-1$ numbers?

Let $OPT(i)$ = the length of the longest sequence ending with the ith character.

Goal: $\max_i OPT(i)$

$$OPT(i) = \begin{cases} 0, & if\ i = 0 \\ \max_{0 \le j < i : s[j] < s[i]} \{OPT(j) + 1\}, & if\ i > 1 \end{cases}$$

Running time: $O(n^2)$.

Subset Sum Problem
and
Knapsack Problem

Subset Sum Problem

Given n items $\{1, \dots, n\}$, and each has a given nonnegative integral weight w_i (for $i = 1, \dots, n$). We are also given an integral bound W. We would like to select a subset S of the items so that $\sum_{i \in S} w_i \leq W$ and, subject to this restriction, $\sum_{i \in S} w_i$ is as large as possible.

For Interval Scheduling, we could simply delete each request that conflicted with request i.

Greedy Approaches:

Start selecting items as long as the total weight remains below W.

- sort the items by decreasing weight

$$\{W/2 + 1, W/2, W/2\}$$

- sort by increasing weight

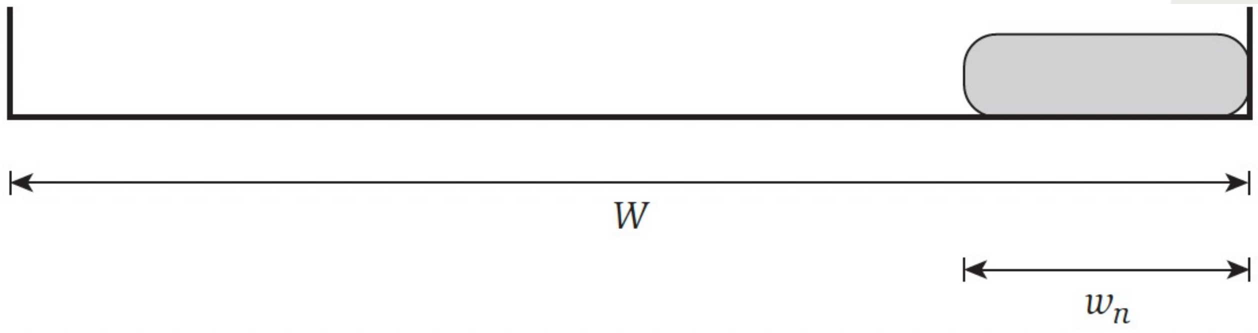
$$\{1, W/2, W/2\}$$

Motivated by ***Weighted Interval Scheduling***:

- consider subproblems involving only the first i items.
- $OPT(i)$ = the best possible solution using a subset of the items $\{1, \dots, i\}$.
- Case 1. $i \notin OPT$, then

$$OPT(i) = OPT(i-1).$$

- Case 2. $i \in OPT$:
  - for the subset of requests $S \subseteq \{1, \dots, i-1\}$ that we will accept, we only have $W - w_i$ weight left.
  - But $OPT(i)$ does not reflect the remaining weight!



Subset Sum Problem: two variables

For for each $i = 0, 1, \ldots, n$ and each integer $0 \le w \le W$:

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

$$OPT(i, w) = \max_{S \subseteq \{1,\cdots,i\}:\ \sum_{l \in S} w_l \le w} \sum_{j \in S} w_j$$

- $OPT(n, W)$ is the quantity we're looking for.

optimal substructure property

***Case 1**. $OPT(i, w)$ does not select item i.*

- $OPT(i, w)$ selects best of $\{1, 2, \ldots, i-1\}$ subject to weight limit w.

***Case 2**. $OPT(i, w)$ selects item i.*

- Collect w_i.
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \ldots, i-1\}$ subject to new weight limit.

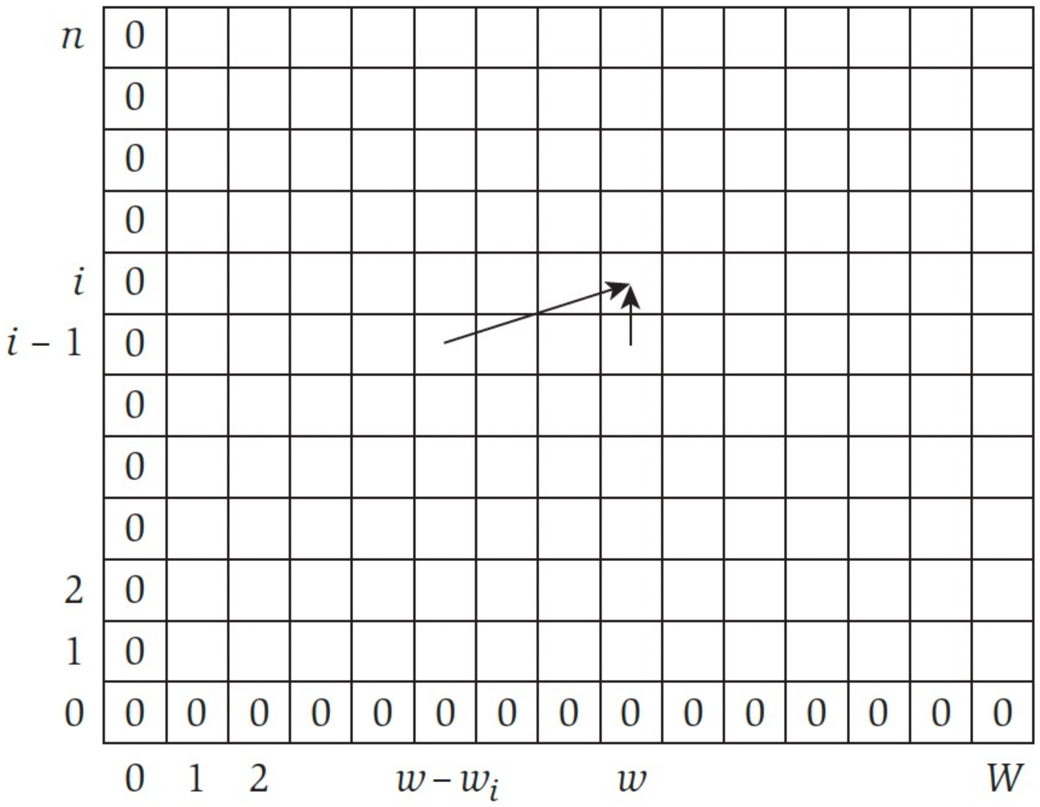
$$OPT(i, w) = \begin{cases} OPT(i-1, w), & \text{if } w < w_i \\ \max\big(OPT(i-1, w), w_i + OPT(i-1, w - w_i)\big), & \textit{otherwise} \end{cases}$$

Subset Sum Problem

$$OPT(i,w) = \begin{cases} OPT(i-1,w), & \text{if } w < w_i \\ \max\big(OPT(i-1,w), w_i + OPT(i-1,w-w_i)\big), & otherwise \end{cases}$$

Subset-Sum(n, W)
  Array $M[0 \ldots n, 0 \ldots W]$
  Initialize $M[0, w] = 0$ for each $w = 0, 1, \ldots, W$
  For $i = 1, 2, \ldots, n$
    For $w = 0, \ldots, W$
      Use the above recurrence to compute $M[i, w]$
    Endfor
  Endfor
  Return $M[n, W]$

- The leftmost column and bottom row are always 0.
- The entry for $OPT(i,w)$ is computed from the two other entries $OPT(i-1,w)$ and $OPT(i-1,w-w_i)$.



The two-dimensional table of OPT values.

Subset Sum Problem

A sample example:

- $W = 6$, items $w_1 = 2, w_2 = 2, w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

3	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

trace back through the array

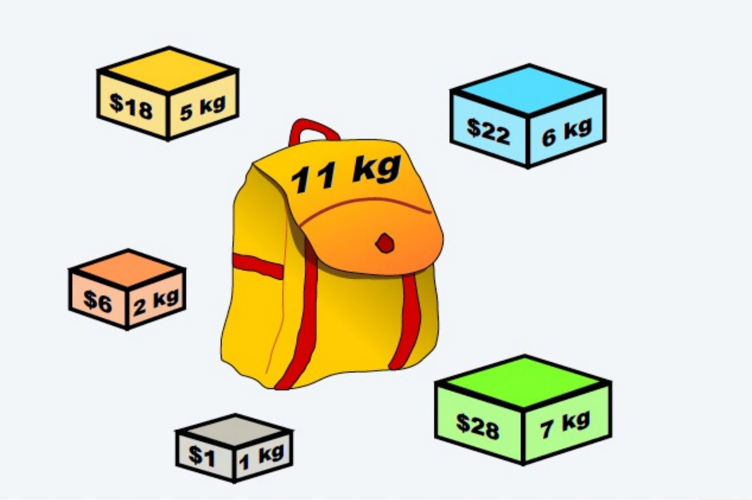
***Theorem*.**

- The DP Algorithm correctly computes the optimal value of the problem, and runs in $O(nW)$ time.
- Given a table M of the optimal values of the subproblems, the optimal set S can be found in $O(n)$ time.

Knapsack Problem

Pack knapsack so as to maximize total value of items taken.

- There are n items: item i provides value $v_i > 0$ and weighs $w_i > 0$.
- Value of a subset of items = sum of values of individual items.
- Knapsack has weight limit of W.
- All values and weights are integral.



Example: $W = 11$

- The subset { 1, 2, 5 } has value $35 (and weight 10).
- The subset { 3, 4 } has value $40 (and weight 11).

i	v_i	w_i
1	$1	1 kg
2	$6	2 kg
3	$18	5 kg
4	$22	6 kg
5	$28	7 kg

Greedy Algorithms:

- Greedy-by-value: repeatedly add item with maximum v_i.
- Greedy-by-weight: repeatedly add item with minimum w_i.
- Greedy-by-ratio: repeatedly add item with maximum ratio v_i / w_i.

$W = 100$
$v_1 = 51, w_1 = 100$
$v_2 = 50, w_2 = 50$
$v_3 = 50, w_3 = 50$

$W = 100$
$v_1 = w_1 = 1$
$v_2 = 99, w_2 = 100$

Dynamic programming: two variables

Let $OPT(i, w)$ = optimal value of knapsack problem with items $1, \dots, i$, subject to weight limit w.

Goal: Computing $OPT(n, W)$.

Case 1. $OPT(i, w)$ does not select item i.
- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to weight limit w.

Case 2. $OPT(i, w)$ selects item i.
- Collect value v_i.
- New weight limit = $w - w_i$.
- $OPT(i, w)$ selects best of $\{1, 2, \dots, i-1\}$ subject to new weight limit.

optimal substructure property

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{\, OPT(i-1, w),\ \ v_i + OPT(i-1, w - w_i)\,\} & \text{otherwise} \end{cases}$$

Dynamic programming: two variables

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

KNAPSACK($n, W, w_1, \ldots, w_n, v_1, \ldots, v_n$)

FOR $w = 0$ TO W

  $M[0, w] \leftarrow 0.$

previously computed values

FOR $i = 1$ TO n

  FOR $w = 0$ TO W

    IF $(w_i > w)$ $M[i, w] \leftarrow M[i-1, w].$

    ELSE $M[i, w] \leftarrow \max\{\, M[i-1, w],\ v_i + M[i-1, w-w_i]\,\}.$

RETURN $M[n, W].$

Dynamic programming

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ v_i + OPT(i-1, w-w_i)\,\} & \text{otherwise} \end{cases}$$

i	v_i	w_i
1	\$1	1 kg
2	\$6	2 kg
3	\$18	5 kg
4	\$22	6 kg
5	\$28	7 kg

$W = 11$

subset of items $1, \ldots, i$

	0	1	2	3	4	5	6	7	8	9	10	11
{ }	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

weight limit w

Theorem.

- The DP algorithm solves the knapsack problem with n items and maximum weight W in $\Theta(nW)$ time and $\Theta(nW)$ space.
- After computing optimal values, can trace back to find solution: $OPT(i,w)$ takes item i iff $M[i,w] > M[i-1,w]$.