Lecture 6:
Dynamic Programming I

THE UNIVERSITY OF SYDNEY

General techniques in this course

- Greedy algorithms [Lecture 3]
- Divide & Conquer algorithms [Lecture 4]
- Sweepline algorithms [Lecture 5]
- Dynamic programming algorithms [today and 5 Sep]
- Network flow algorithms [12 and 19 Sep]

Algorithmic Paradigms

- Greed. Build up a solution incrementally, myopically optimizing some local criterion.

- Divide-and-conquer. Break up a problem into two sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

- Dynamic programming. Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming Applications

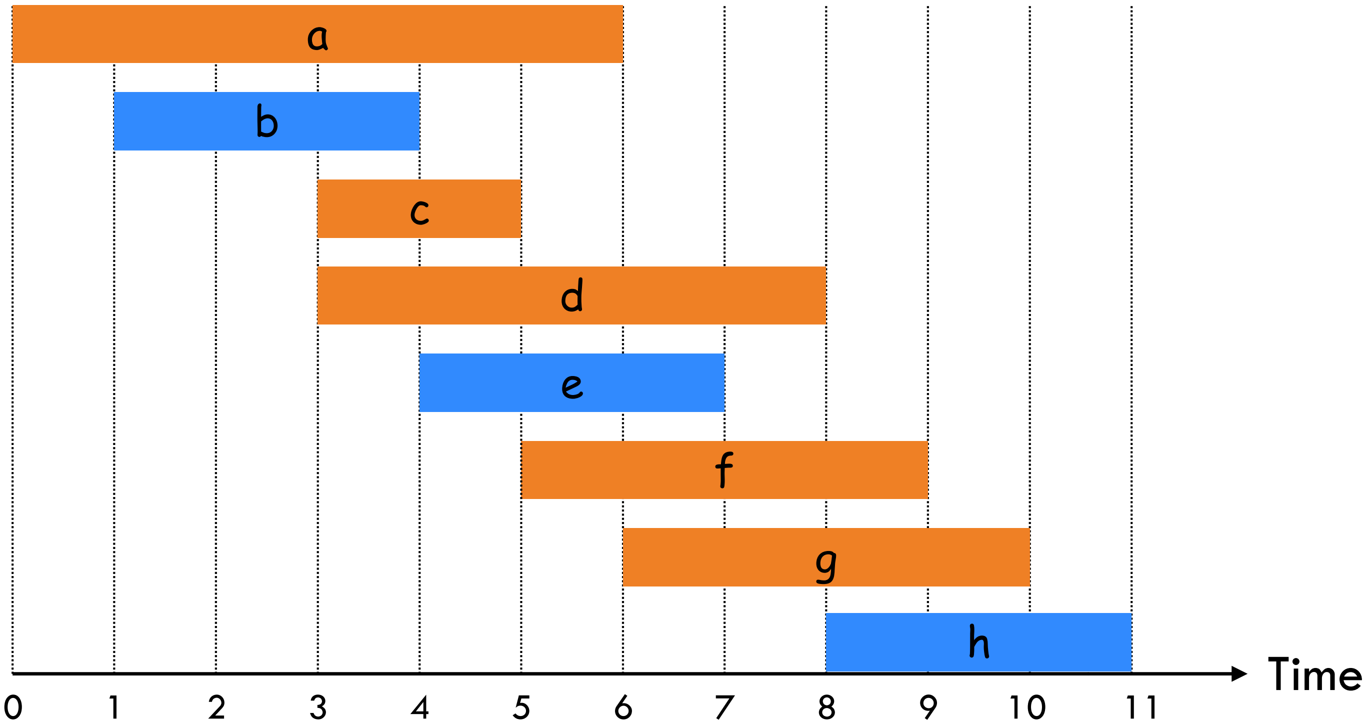
- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, systems, ….

- Some famous dynamic programming algorithms.
  - Viterbi for hidden Markov models.
  - Unix diff for comparing two files.
  - Smith-Waterman for sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.

6.1 Weighted Interval Scheduling

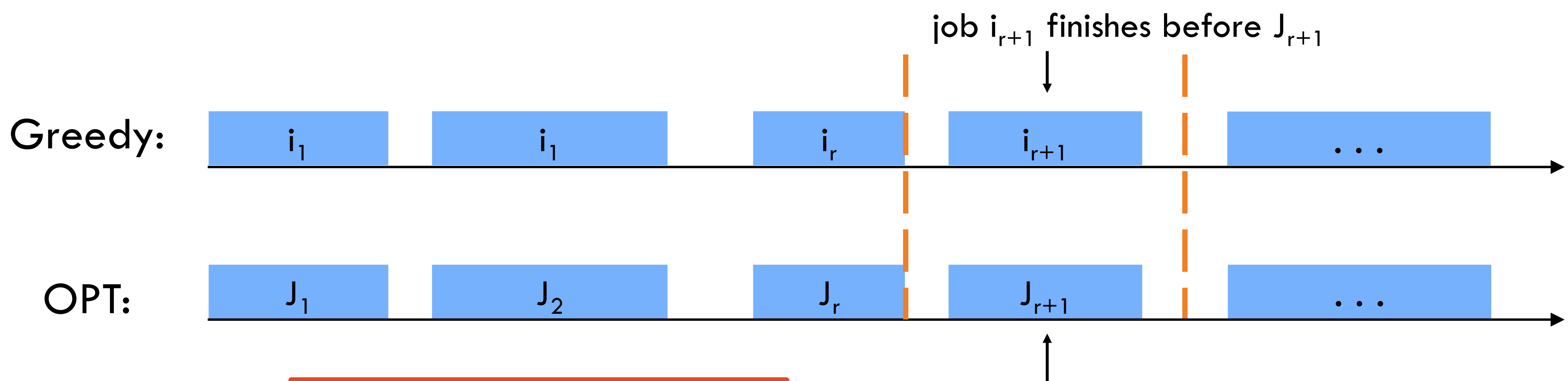
Recall Interval Scheduling (Lecture 3)

- Interval scheduling.
  - **Input:** Set of n jobs. Each job i starts at time s_i and finishes at time f_i.
  - Two jobs are compatible if they don't overlap in time.
  - **Goal:** find maximum subset of mutually compatible jobs.



Recall Interval Scheduling (Lecture 3)

- **Theorem:** Greedy algorithm [Earliest finish time] is optimal.
- **Proof:** (by contradiction)
  - Assume greedy is not optimal, and let's see what happens.
  - Let $i_1, i_2, \ldots i_k$ denote the set of jobs selected by greedy.
  - Let $J_1, J_2, \ldots J_m$ denote the set of jobs in an optimal solution with $i_1 = J_1, i_2 = J_2, \ldots, i_r = J_r$ for the largest possible value of r.



Exchange argument!

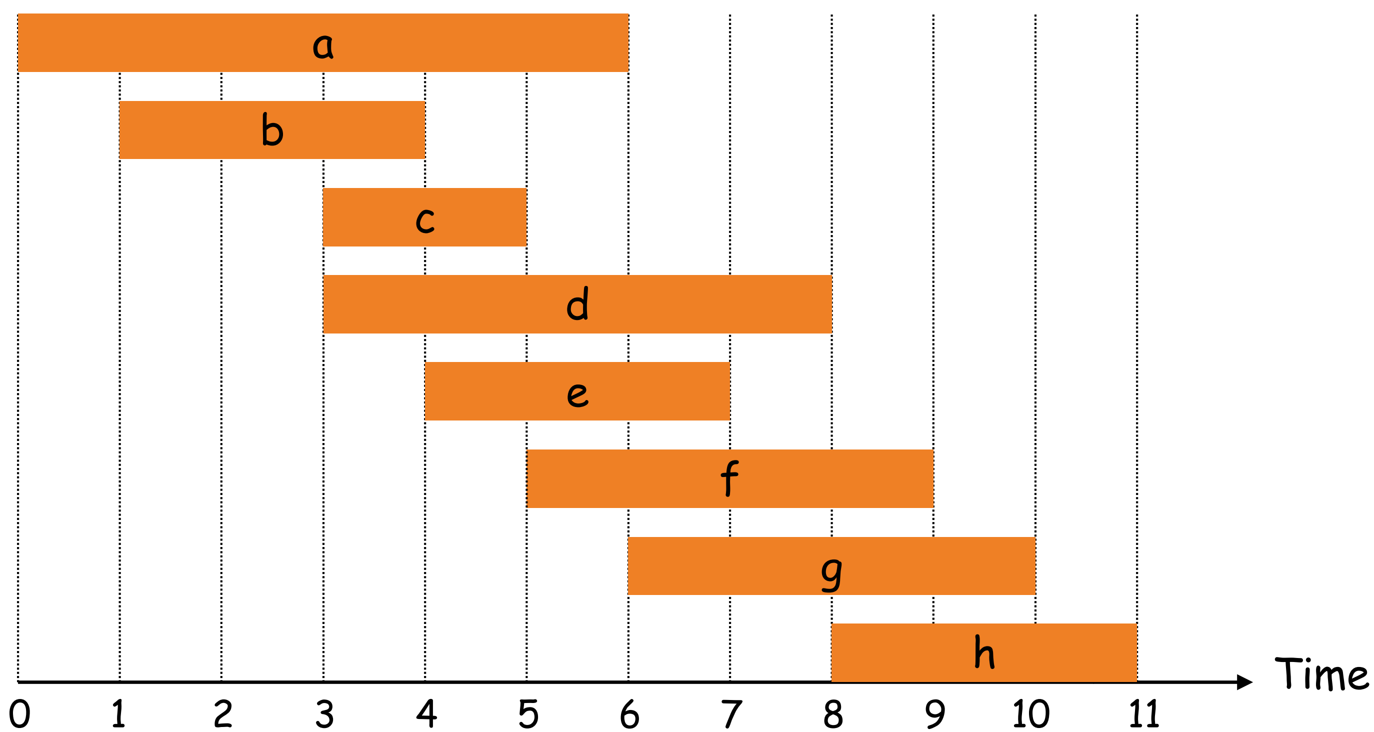
solution still feasible and optimal,
but contradicts maximality of r.

Recall Interval Scheduling (Lecture 3)

There exists a greedy algorithm [Earliest finish time] that computes the optimal solution in O(n log n) time.

Weighted Interval Scheduling

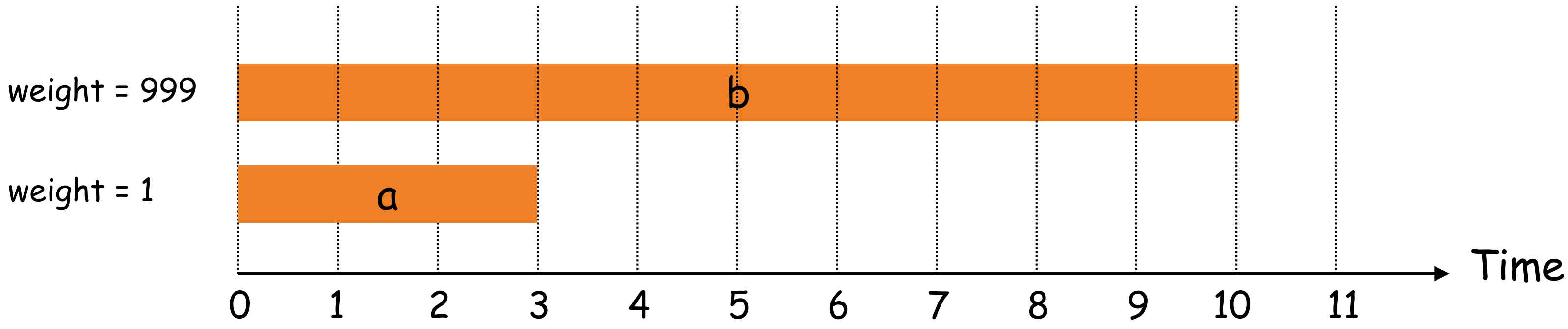
- Weighted interval scheduling problem.
  - Job j starts at s_j, finishes at f_j, and has weight or value v_j .
  - Two jobs compatible if they don't overlap.
  - Goal: find maximum weight subset of mutually compatible jobs.



Unweighted Interval Scheduling Review

- 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 chosen jobs.

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

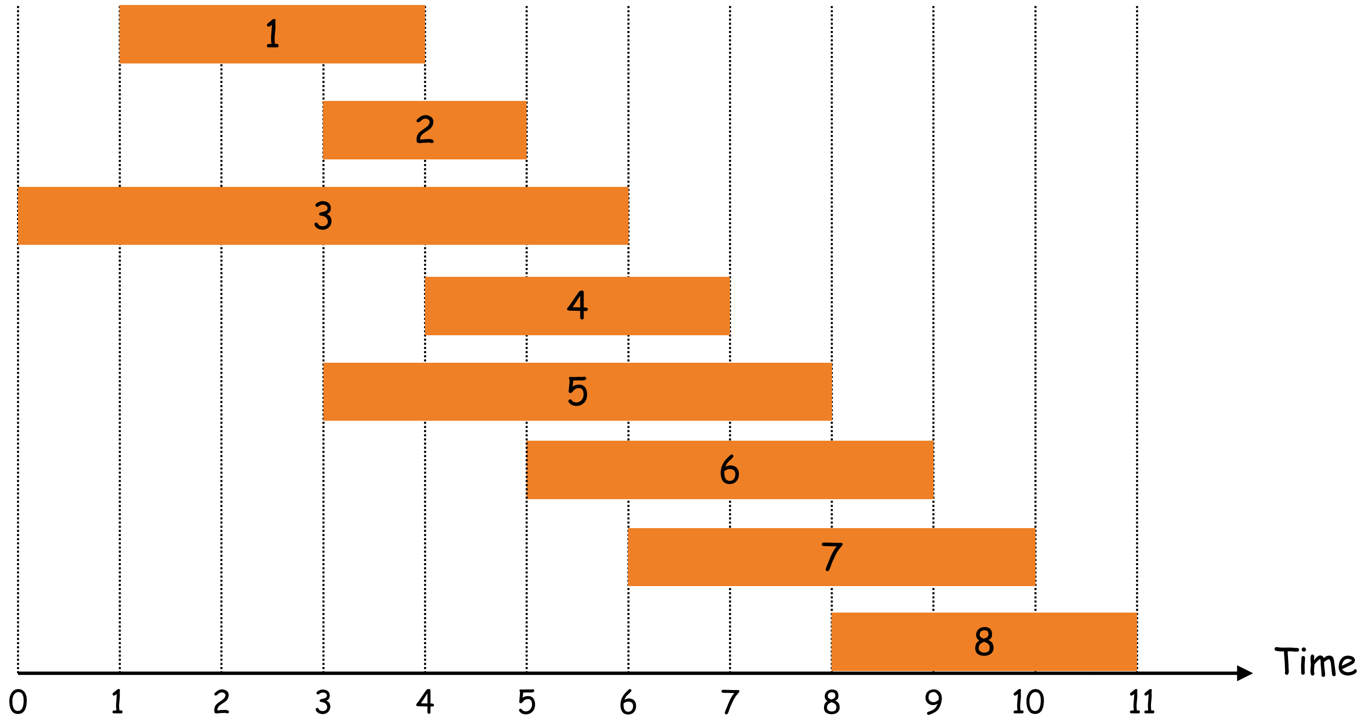


Weighted Interval Scheduling

Notation. Label jobs by finishing 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: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Key steps: Dynamic programming

1. Define subproblems
2. Find recurrences
3. Solve the base cases
4. Transform recurrence into an efficient algorithm

Dynamic Programming: Weighted Interval Scheduling

Step 1: Define subproblems

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

Dynamic Programming: Weighted Interval Scheduling

Step 2: Find recurrences

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

$$\boxed{OPT(j) = \max\{\underbrace{v_j + OPT(p(j))}_{\text{Case 1}}, \underbrace{OPT(j-1)}_{\text{Case 2}}\}}$$

Dynamic Programming: Weighted Interval Scheduling

Step 3: Solve the base cases

$$OPT(0) = 0$$

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

Done…more or less

Weighted Interval Scheduling: Brute Force

– Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

This is for Base Case

Weighted Interval Scheduling: Correctness

Theorem: `Compute-Opt(j)` correctly computes OPT(j)

Proof: Proof by induction.

- Base case: OPT(0) = 0
- Induction hypothesis (i<j):

  For any i < j `Compute-Opt(i)` correctly computes OPT(i).
- Induction step (j):

  Since p(j)< j we have `Compute-Opt(p(j))`=OPT(p(j))

  and `Compute-Opt(j-1)`=OPT(j-1).

Hence,

$$OPT(j) = \max\{v_j + \texttt{Compute-Opt(p(j))}, \texttt{Compute-Opt(j-1)}$$

∎

Weighted Interval Scheduling: Brute Force

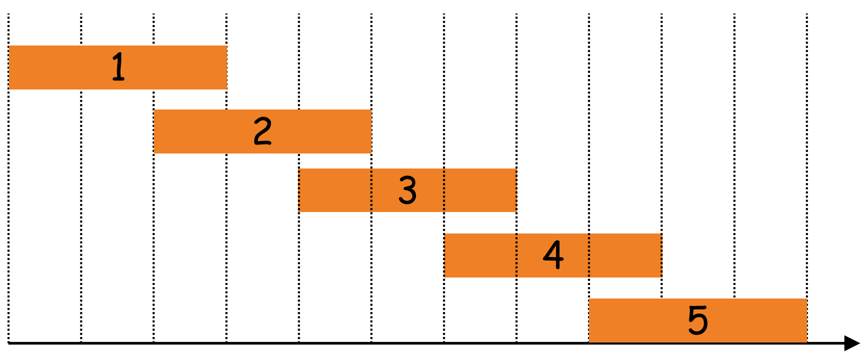
– Brute force algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(v_j + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

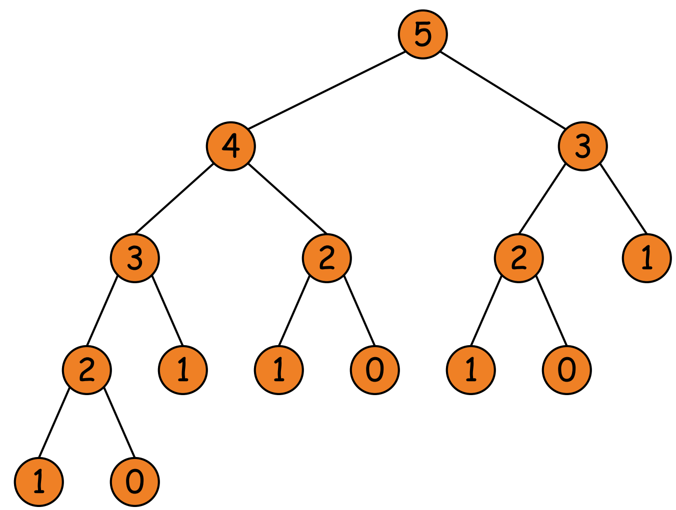
Weighted Interval Scheduling: Brute Force

Observation. Recursive algorithm is slow because of redundant sub-problems $\Rightarrow$ exponential algorithms.

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



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



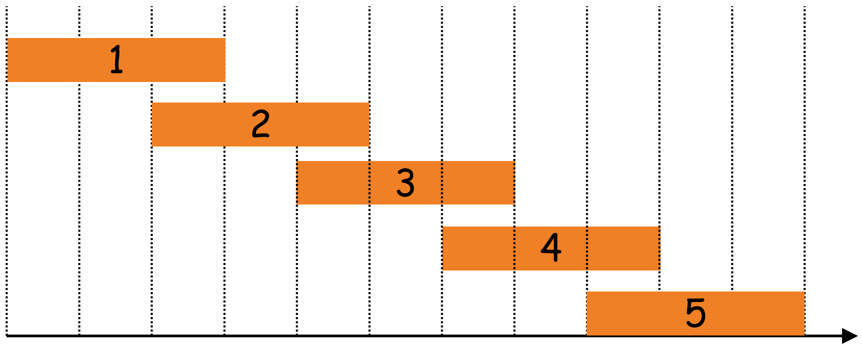
Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem; lookup when needed.

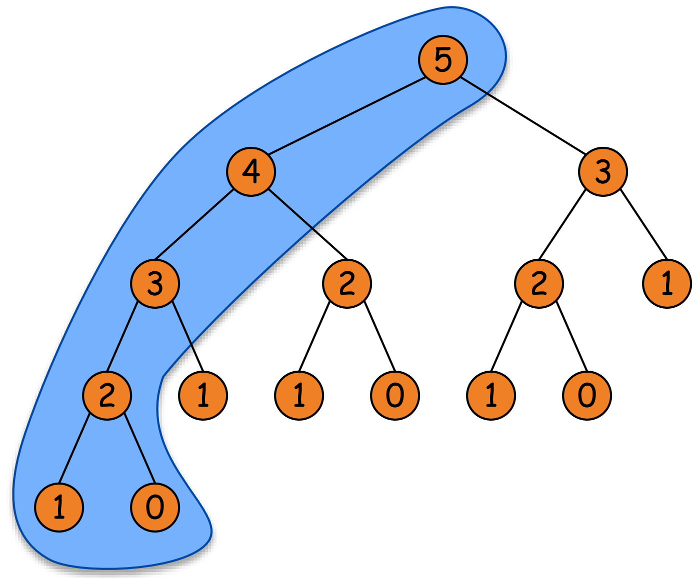
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty

M[0] = 0

Compute-Opt(j) {
   if (M[j] is empty)
      M[j] = max(wj + Compute-Opt(p(j)), Compute-Opt(j-1))
   return M[j]
}
```

Preprocessing

Weighted Interval Scheduling: Brute Force



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



Weighted Interval Scheduling: Running Time

Claim. Memoized version of algorithm takes O(n log n) time.

- Sort by finish time: O(n log n).
- Computing p(·) : O(n) after sorting by start time.

- `Compute-Opt(j)`: each call takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two new recursive calls

- Overall time is O(1) times the number of calls to `Compute-Opt(j)`.

- Progress measure K = # nonempty entries of `M[]`.
  - initially K = 0 and the number of empty entries is n.
  - Case (ii) increases K by 1 $\Rightarrow$ at most 2n recursive calls.

- Overall running time of `Compute-Opt(n)` is O(n). ▪

Remark: O(n) if jobs are pre-sorted by start and finish times.

Weighted Interval Scheduling: Finding a Solution

Question. Dynamic programming algorithm computes optimal value.

What if we want the solution itself?
Answer. Do some post-processing.

```
Run Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (vj + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

\# of recursive calls $\leq$ n $\Rightarrow$ O(n).

Weighted Interval Scheduling: Bottom-Up

– Bottom-up dynamic programming. Unwind recursion.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(v_j + M[p(j)], M[j-1])
}
```

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Maximum-sum contiguous subarray

Given an array A[ ] of n numbers, find the maximum sum found in any contiguous subarray

A zero length subarray has maximum 0

Example:

1	-2	7	5	6	-5	5	8	1	-6

Brute-force algorithm

- All possible contiguous subarrays
  - A[1..1], A[1..2], A[1..3], ..., A[1..(n-1)], A[1..n]
  - A[2..2], A[2..3], ..., A[2..(n-1)], A[2..n]
  - ...
  - A[(n-1)..(n-1)], A[(n-1)..n]
  - A[n..n]

- How many of them in total? $O(n^2)$

- **Algorithm:** For each subarray, $\overbrace{\text{compute the sum}}^{O(n)}$. Report the subarray that has the maximum sum.

Total time: $O(n^3)$

Divide-and-conquer algorithm

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

- Three cases:
  a) MCS in A[1..n/2]
  b) MCS in A[n/2+1..n]
  c) MCS that spans across A[n/2]

- (a) & (b) can be found **recursively**

- (c) can be found in two steps
  - Consider MCS in A[1..n/2] ending in A[n/2].
  - Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  - Sum these two maximum

Idea of divide-and-conquer

Example:	10	15	-3	-4 |	-2	-1	8	5
max on L (recursion)	10	15						
max on R (recursion)							8	5
mid extend to L	10	15	-3	-4				
mid extend to R					-2	-1	8	5

- Possible candidates:
  - 25, 13, 28 (=18+10)
  - overall maximum **28**.

Idea of divide-and-conquer

Example:	-2	5	-1	-5	2	-1	2
max on L (recursion)		5					
max on R (recursion)					2	-1	2
mid extend to L		5	-1				
mid extend to R				*not take any*			

- Possible candidates:
  - 5, 3, 4 (=4+0)
  - overall maximum **5**

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

– Three cases:
  a) MCS in A[1..n/2]
  b) MCS in A[n/2+1..n]
  c) MCS that spans across A[n/2]

$2 \cdot T(n/2)$

– (a) & (b) can be found **recursively**

– (c) can be found in two steps
  – Consider MCS in A[1..n/2] ending in A[n/2].
  – Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  – Sum these two maximum

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 ...									n/2	
1	-2	7	5	6	-5	5	8	1	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 ...									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
									-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 ...									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
								-5	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 ...									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
							3	-5	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 …									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
						8	3	-5	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 ...									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
					3	8	3	-5	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 …									n/2	
1	-2	7	5	6	-5	5	8	1	-6	
20	19	21	14	9	3	8	3	-5	-6	

Divide-and-conquer algorithm: Step c

MCS in A[1..n/2] ending in A[n/2]

1 …									n/2
1	-2	7	5	6	-5	5	8	1	-6
20	19	21	14	9	3	8	3	-5	-6

Time: O(n)

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

– MCS in A[1..n/2]
  a. MCS in A[n/2+1..n]
  b. MCS that spans across A[n/2]

2·T(n/2)

– (a) & (b) can be found **recursively**

– (c) can be found in two steps
  – Consider MCS in A[1..n/2] ending in A[n/2].
  – Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  – Sum these two maximum

O(n)

Divide-and-conquer algorithm

Maximum contiguous subarray (MCS) in A[1..n]

- MCS in A[1..n/2]
  a. MCS in A[n/2+1..n]
  b. MCS that spans across A[n/2]

$2 \cdot T(n/2)$

- (a) & (b) can be found **recursively**

- (c) can be found in two steps
  - Consider MCS in A[1..n/2] ending in A[n/2].
  - Consider MCS in A[n/2+1..n] starting at A[n/2+1].
  - Sum these two maximum

$O(n)$

Total time: $T(n) = 2 \cdot T(n/2) + O(n) = O(n \log n)$

Dynamic programming

Step 1: Define subproblems

OPT(i) = optimal solution ending at i.

Dynamic programming algorithm

Example 1:	6	-3	-2	3	-1	2
OPT[1] = 6	6					
OPT[2] = 3	6	-3				
OPT[3] = 1	6	-3	-2			
OPT[4] = 4	6	-3	-2	3		
OPT[5] = 3	6	-3	-2	3	-1	
OPT[6] = 5	6	-3	-2	3	-1	2

OPT[i] – optimal solution ending at i

Dynamic programming algorithm

Example 2:	2	-6	-1	3	-1	2	-2
OPT[1] = 2	2						
OPT[2] = 0	2	-6					
OPT[3] = 0	2	-6	-1				
OPT[4] = 3	2	-6	-1	3			
OPT[5] = 2	2	-6	-1	3	-1		
OPT[6] = 4	2	-6	-1	3	-1	2	
OPT[7] = 2	2	-6	-1	3	-1	2	-2

OPT[i] – optimal solution ending at i

Dynamic programming algorithm

Example 3:	-2	5	-1	-5	3	-1	2
OPT[1] = 0	-2						
OPT[2] = 5	-2	5					
OPT[3] = 4	-2	5	-1				
OPT[4] = 0	-2	5	-1	-5			
OPT[5] = 3	-2	5	-1	-5	3		
OPT[6] = 2	-2	5	-1	-5	3	-1	
OPT[7] = 4	-2	5	-1	-5	3	-1	2

OPT[i] – optimal solution ending at i

Dynamic programming algorithm

Example 3:	-2	5	-1	-5	3	-1	2
OPT[1] = 0	-2						
OPT[2] = 5	-2	5					
OPT[3] = 4	-2	5	-1				
OPT[4] = 0	-2	5	-1	-5			
OPT[5] = 3	-2	5	-1	-5	3		
OPT[6] = 2	-2	5	-1	-5	3	-1	
OPT[7] = 4	-2	5	-1	-5	3	-1	2

Step 2: Find recurrences

OPT[i] – optimal solution ending at i

OPT[i] = max{OPT[i-1]+A[i],0}

Dynamic programming algorithm

Step 3: Solve the base cases

$$OPT[1] = \max(A[1], 0)$$

$$OPT[i] = \begin{cases} \max(A[1], 0) & \text{if } i=1 \\ \max\{OPT[i-1]+A[i], 0\} & \text{if } i>1 \end{cases}$$

Pseudo Code

OPT[i] – optimal solution ending at i

```
OPT[1] = max(A[1], 0)
for i = 2 to n do
    OPT[i] = max(OPT[i-1]+A[i], 0)
MaxSum = OPT[1]
for i = 2 to n do
    MaxSum = max(MaxSum, OPT[i])
```

Total time: O(n)

6.4 Knapsack

A 1998 study of the Stony Brook University Algorithm Repository showed that, out of 75 algorithmic problems, the knapsack problem was the 18th most popular and the 4th most needed after kd-trees, suffix trees, and the bin packing problem.

First mentioned by Mathews in 1897.
“Knapsack problem” by Dantzig in 1930.

Knapsack Problem 背包问题

- **Knapsack problem.**
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack so as to maximize total value.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem

- **Knapsack problem.**
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack so as to maximize total value.

- **Example:** { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem

- **Knapsack problem.**
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - **Goal:** fill knapsack so as to maximize total value.
- **Example:** { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- **Greedy:** repeatedly add item with maximum ratio v_i / w_i.
- Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

- **Definition.** OPT(i) = max profit subset of items 1, …, i.
  - **Case 1:** OPT does not select item i.
    - OPT selects best of { 1, 2, …, i-1 }
  - **Case 2:** OPT selects item i.
    - accepting item i does not immediately imply that we will have to reject other items
    - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion: Need more subproblems!

Dynamic Programming: Adding a New Variable

Step 1: Define subproblems

OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

Dynamic Programming: Adding a New Variable

Step 2: **Find recurrences**

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- **Case 2:** OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

Dynamic Programming: Adding a New Variable

Step 3: Solve the base cases

$$OPT[0] = 0$$

Dynamic Programming: Adding a New Variable

- **Base case:** OPT[0] = 0

- **Case 1:** OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w

- **Case 2:** OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i–1 } using this new weight limit

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

Knapsack Problem: Bottom-Up

– **Knapsack.** Fill up an (n+1)-by-(W+1) array.

```
Input: n, w1,…,wN, v1,…,vN

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}
return M[n, W]
```

Knapsack Algorithm

W + 1

n + 1

	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	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	34	40

OPT: { 4, 3 }
value = 22 + 18 = 40

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Knapsack Problem: Running Time

- **Running time:** $\Theta(nW)$.
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete. [Lecture 10]

- Knapsack approximation algorithm. There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum. [Lecture 10?]

Longest increasing subsequence

Given a sequence of numbers X[1..n] find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

5 2 8 6 3 6 9 7

Longest increasing subsequence

Given a sequence of numbers X[1..n] find the longest increasing *subsequence* $(i_1, i_2, \ldots, i_k)$, that is a subsequence where numbers in the sequence increase.

5 2 8 6 3 6 9 7

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Example:

L[1] = 1	L[4] = 2	L[7] = 4
L[2] = 1	L[5] = 2	
L[3] = 2	L[6] = 3	

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Dynamic programming formula:

$$L[i] = \max_{j<i} \{ L[j] + 1 \mid X[j] < X[i] \}$$

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Dynamic programming formula:

$$L[i] = \max_{j<i} \{ L[j] + 1 \mid X[j] < X[i] \}$$

- Running time: ?

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Dynamic programming formula:

$$\underset{\text{n times}}{L[i]} = \max_{j<i} \{\underset{n}{L[j]+1} \mid X[j] < X[i]\}$$

- Running time: ?

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Dynamic programming formula:

  $$L[i] = \max_{j<i} \{L[j] + 1 \mid X[j] < X[i]\}$$

  n times

  n

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

Can we do better?

Longest increasing subsequence

Define a vector L[]:

- L[i] = length of the longest increasing subsequence that ends at i.
- L[1] = 1

5 2 8 6 3 6 9 7

- Dynamic programming formula:

  $L[i] = \max \{ L[j] + 1 \mid X[j] < X[i] \}$

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

How can we compute the LIS in O(n log n) time?

Dynamic Programming Summary

- Recipe.
  - Characterize structure of problem.
  - Recursively define value of optimal solution.
  - Compute value of optimal solution.
  - Construct optimal solution from computed information.

- Dynamic programming techniques.
  - Binary choice: weighted interval scheduling. ← Viterbi algorithm for HMM also uses DP to optimize a maximum likelihood tradeoff between parsimony and accuracy
  - Adding a new variable: knapsack.

- Top-down vs. bottom-up: different people have different intuitions.