Dynamic Programming (1)

Desain & Analisis Algoritma

Fakultas Ilmu Komputer
Universitas Indonesia

Compiled by **Alfan F. Wicaksono** from multiple sources

Credits

- Introduction to Algorithms, **Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein**
- Dynamic Programming: Weighted Interval Scheduling, CMSC 451: Lecture 10, by **Dave Mount**

Review: Divide-and-Conquer Strategy

Remember that the **Divide-and-Conquer** algorithms

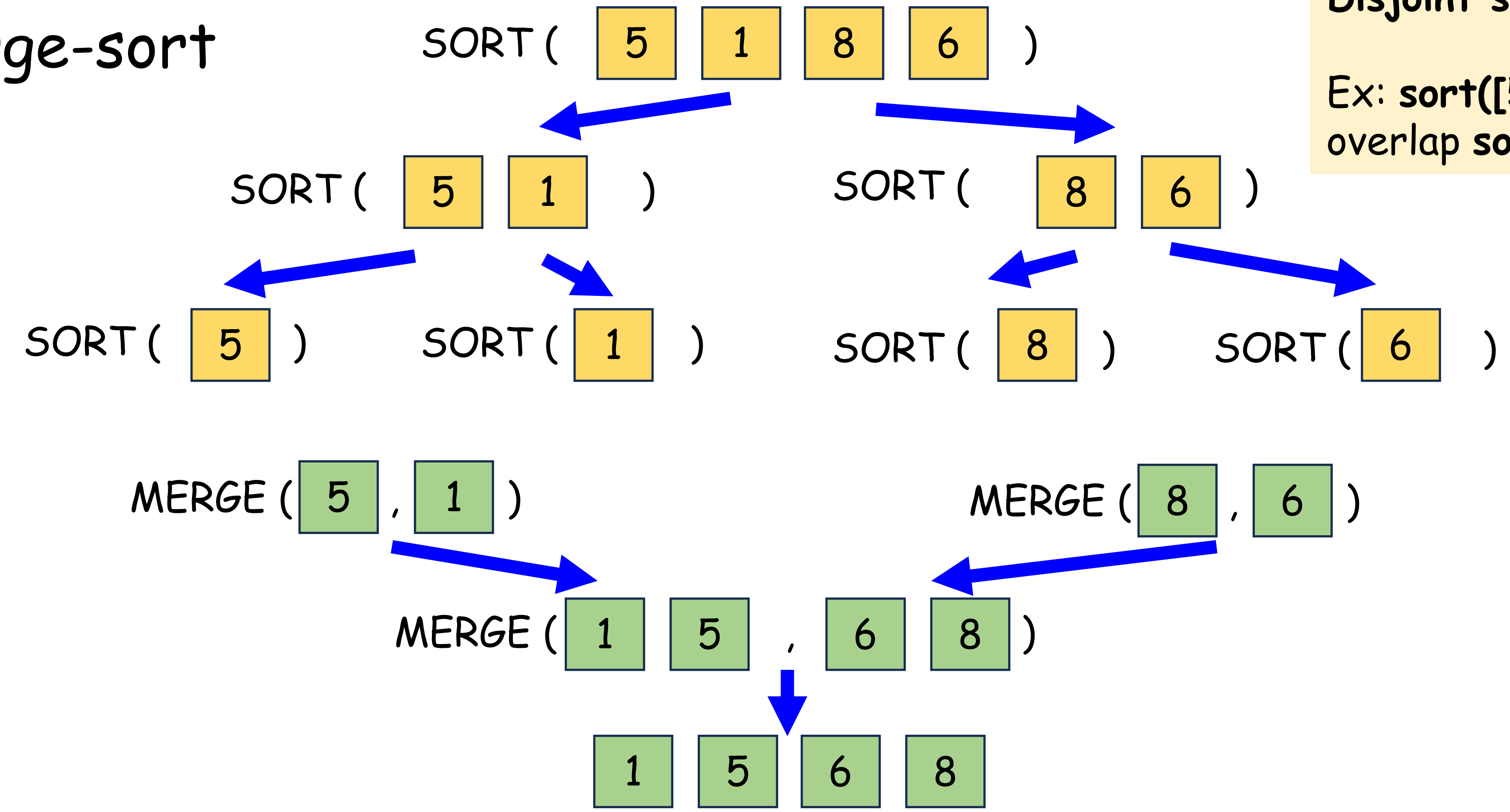
(1) partition the problem into **disjoint or non-overlapping subproblems**,

(2) solve the subproblems recursively, and then

(3) combine their solutions to solve the original problem.

Review: Divide-and-Conquer Strategy

Merge-sort



Disjoint subproblems!

Ex: **sort([5,1])** does not overlap **sort([8,6])**

Review: Divide-and-Conquer Strategy

However, when the **subproblems overlap** (that is, when subproblems share subsubproblems), a divide-and-conquer algorithm does more work than necessary, **repeatedly solving the common subsubproblems**.

Review: Divide-and-Conquer Strategy

Example: Subset Sum Problem (SSP)

Given **N** non-negative integers and a value **sum**, check whether there is a subset of these integers whose sum is equal to the given **sum**.

Input: {3, 73, 4, 26, 6, 9}   sum = 10

Output: **True** , a subset {4, 6}

Review: Divide-and-Conquer Strategy

Example: Subset Sum Problem (SSP)

```
SSP(array, n, sum):     // the array index is one-based
    if sum == 0 then
        return True
                                    Base Cases!
    if n == 0 then
        return False

    if array[n] > sum then
        return SSP(array, n-1, sum)    // just skip, last element > sum
    else                                                          Recurrence
        return SSP(array, n-1, sum)   OR   SSP(array, n-1, sum - array[n])
```

$O(2^n)$ … **prove it!**

Review: Divide-and-Conquer Strategy

Input: array = {9, 4, 7, 1, 6, 8, 2, 5, 3} , sum = 45

$O(2^n)$ … prove it!

SSP(array, 9, 45)

SSP(array, 8, 45) **OR** SSP(array, 8, 42)

SSP(array, 7, 45) **OR** SSP(array, 7, 40) SSP(array, 7, 42) **OR** SSP(array, 7, 37)

… **SSP(array, 6, 40)** **OR** SSP(array, 6, 38) SSP(array, 6, 42) **OR** **SSP(array, 6, 40)** …

… … … … … … … …

Overlapping Subproblems!

To the Rescue: Dynamic Programming

- Like the divide-and-conquer method, Dynamic programming solves problems by combining the solutions to subproblems.
- In contrast, dynamic programming applies **when the subproblems overlap**.
- A dynamic programming algorithm solves each subsubproblem just once and then **saves its answer in a table**, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem.

- Dynamic programming (**DP**) typically applies to **optimization problems**. Such problems can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value.
- We call such a solution **an** **optimal** solution to the problem, as opposed to **the** **optimal** solution, since there may be several solutions that achieve the optimal value.

- **Four steps** to develop a DP solution:
  - Characterize the structure of an optimal solution;
  - Recursively define the value of an optimal solution;
  - Compute the value of an optimal solution, typically in a bottom-up fashion;
  - Construct an optimal solution from computed information

Dynamic Programming Solutions

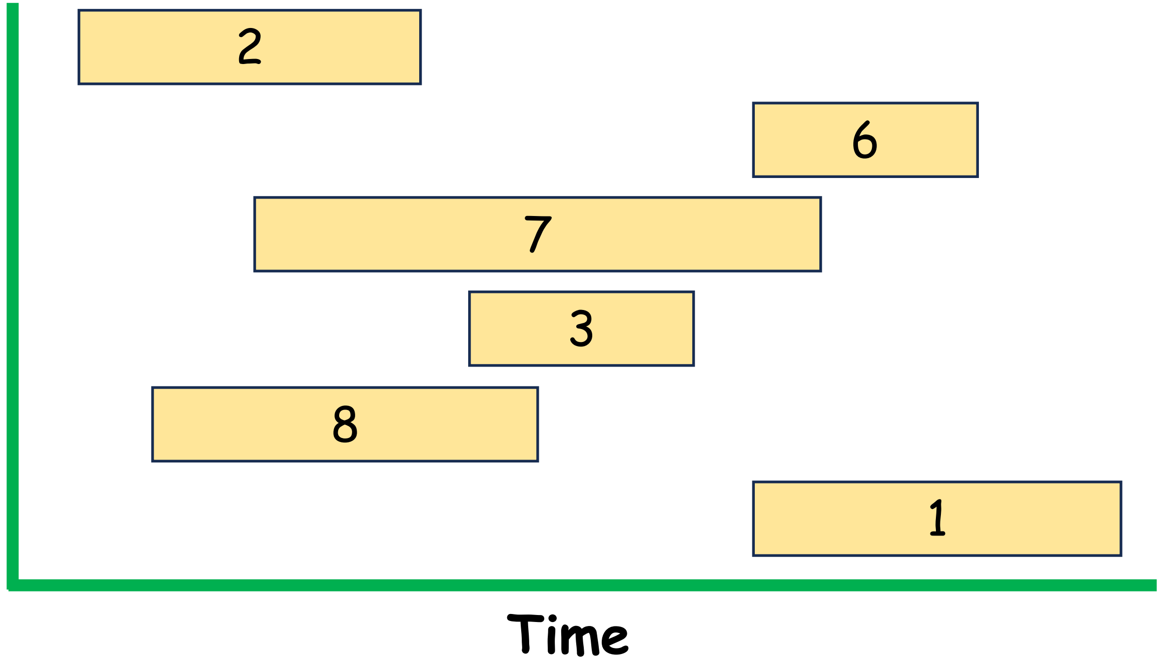
From now, we will use the dynamic programming method to solve some optimization problems:

- Weighted Interval Scheduling
- Knapsack Problem
- Longest Common Subsequence
- Rod Cutting
- Segmented Least Squares --> popular in "Data Science" area

Weighted Interval Scheduling

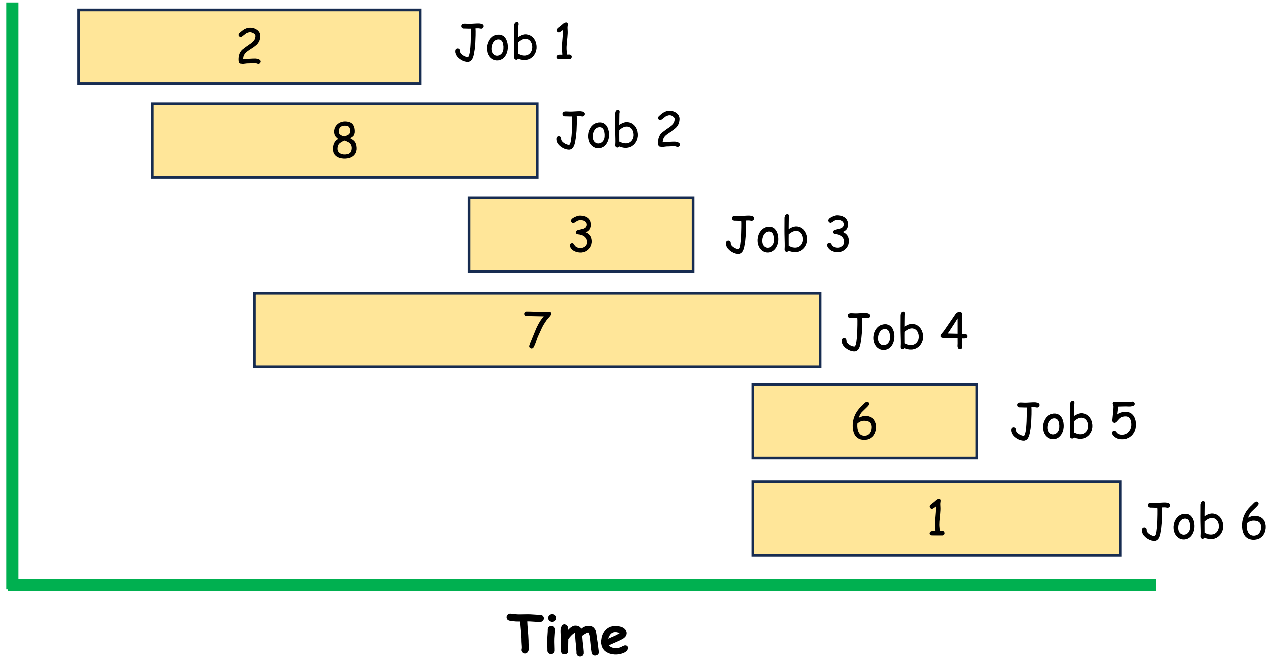
Weighted Interval Scheduling

- We are given a set of jobs;
- A job **j** is associated with a **start** time s_j, **finish** time f_j, and a **weight** w_j (or importance);
- Two jobs are **compatible** if they don't overlap;
- **Goal**: Find a set of compatible jobs such that sum of weights is maximum.



Weighted Interval Scheduling

- It is convenient to sort the jobs in **non-decreasing order** of finish time f_j; so that $f_1 \leq f_2 \leq \cdots \leq f_n$;
- We define $p(j)$: largest index $i < j$ such that job i is compatible with j.



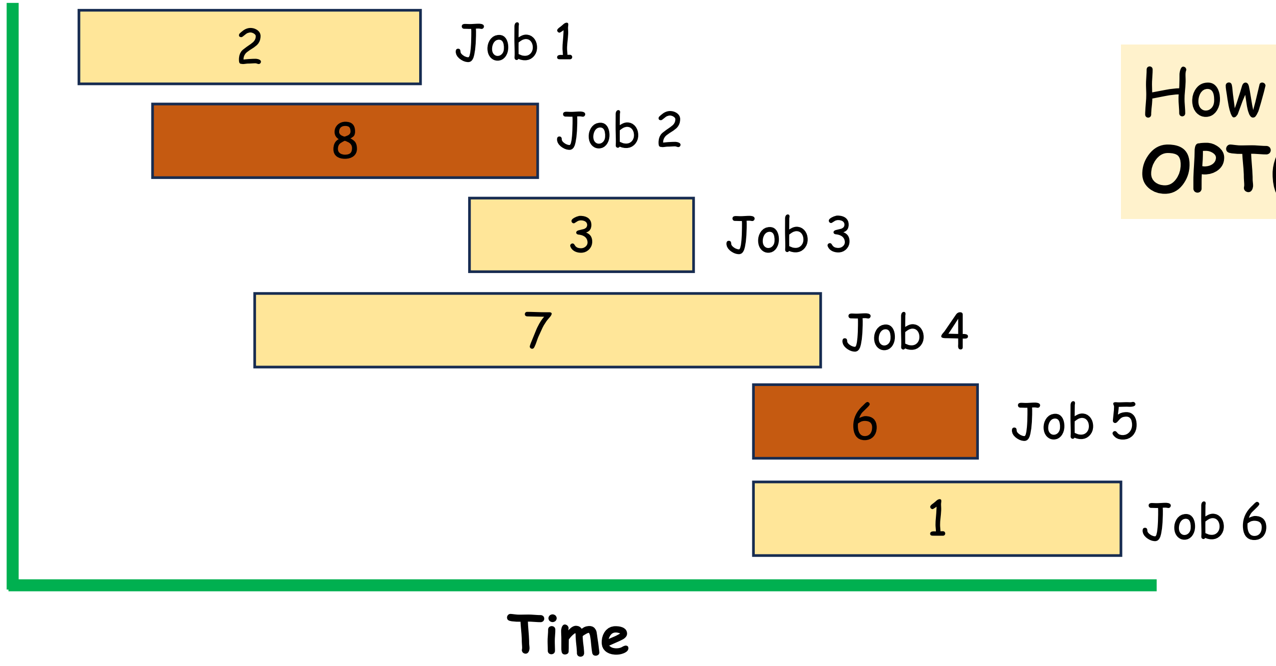
j	$p(j)$
0	-
1	0
2	0
3	1
4	0
5	3
6	3

Weighted Interval Scheduling

Notation:

- **OPT(j)** : value of optimal solution with respect to job requests **1, 2, …, j**

- In the following example, an optimal solution **OPT(6)** is **14**



How to compute **OPT(j)** using DP?

Dynamic Programming

Two important structural qualities

(1) Optimal Substructure

This property states that for the global problem to be solved optimally, **each subproblem** should be solved optimally.

Consequently, you must take care to ensure that the range of subproblems you consider includes those used in an optimal solution.

Dynamic Programming

Two important structural qualities

(2) Overlapping Subproblems

While it may be possible subdivide a problem into subproblems in **exponentially many different ways**, these subproblems overlap each other in such a way that the number of **distinct subproblems** is reasonably **small**, ideally **polynomial** in the input size.

> Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

Weighted Interval Scheduling

Step 1. Characterize the structure of an optimal solution OPT(j)

Case 1: Optimum selects job j

- In this case, we can't use jobs {p(j)+1, p(j)+2, … , j-1}. They are incompatible!
- We must include optimal solution to subproblem consisting of compatible jobs {1, 2, 3, …, p(j)}

Optimal Substructure

Case 2: Optimum does not select job j

- We must include optimal solution to subproblem consisting of compatible jobs {1, 2, 3, …, j-1}

Weighted Interval Scheduling

Step 2. Define a recursive solution to OPT(j)

Job **j** is in opt

Job **j** is not in opt

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

OPT(j):

```
  if j == 0 then
      return 0
  else
      return max(  wj + OPT(p(j)),  OPT(j - 1)  )

WIS (n):
  sort jobs by finish times so that f1 ≤ f2 ≤ ··· ≤ fn
  compute p(1), p(2), …, p(n)
  return OPT(n)
```

Time Complexity?

Weighted Interval Scheduling

Step 2. Define a recursive solution to OPT(j)

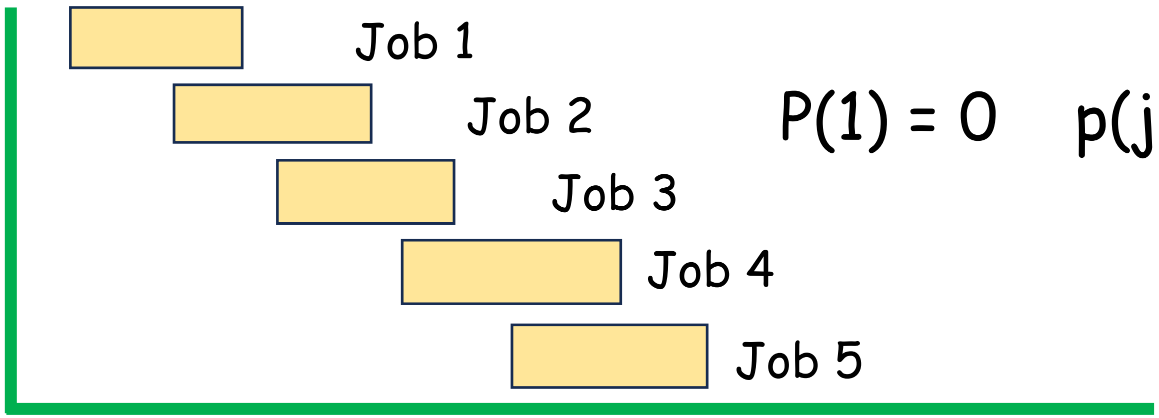
Job **j** is in opt

Job **j** is not in opt

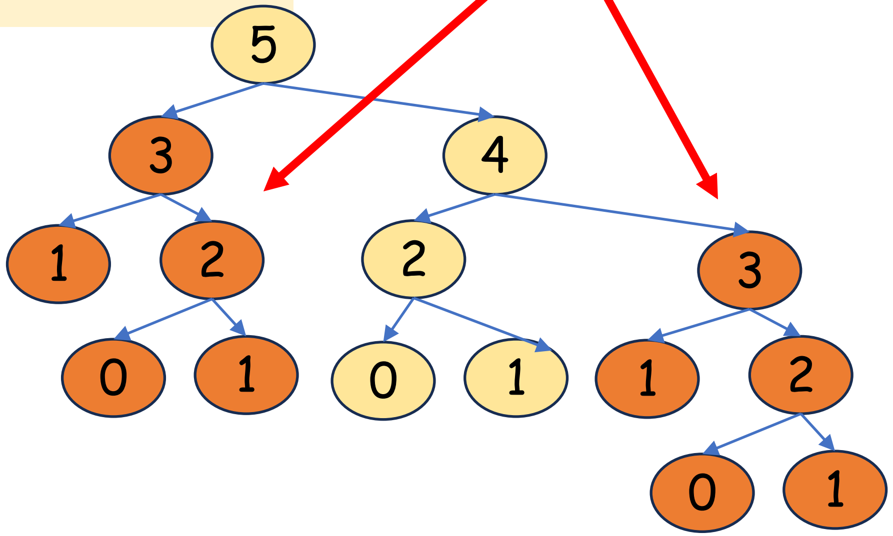
$$OPT(j) = \begin{cases} 0, & j = 0 \\ \max(\boldsymbol{w_j} + \boldsymbol{OPT(p(j))}, \quad \boldsymbol{OPT(j-1)}), & j > 0 \end{cases}$$

An example of overlapping subproblem

When the input jobs are "layered" (**worst case**), the number recursive calls grows like Fibonacci sequence! ➔ ***exponential algorithms!***



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



Dynamic Programming

Two approaches:

(1) Top-Down (Memoization)

- You write the procedure **recursively** in a natural manner, but modified to **save the result of each subproblem** (usually in an array or hash table)
- The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level.
- If not, the procedure computes the value in the usual manner but also saves it.

Dynamic Programming

Two approaches:

(2) Bottop-Up (Iterative)

- Solving any particular subproblem depends only on solving "smaller" subproblems.
- **Solve the subproblems in size order**, **smallest first**, storing the solution to each subproblem when it is first solved.
- You need to solve each subproblem only once, and when you first see it, you have already solved all of its prerequisite subproblems.

Weighted Interval Scheduling

Step 3. Compute OPT(j) using DP

Top-down (memoization) approach

```
OPT(j):
    if j == 0 then
        return 0
    else
        return max( w_j + OPT(p(j)), OPT(j - 1) )

WIS (n):
    sort jobs by finish times so that f_1 ≤ f_2 ≤ ⋯ ≤ f_n
    compute p(1), p(2), …, p(n)
    return OPT(n)
```

Recursive Version

```
MEMOIZED-OPT(OPT, j):

    if j == 0 then
        return 0
    else
        if j not in OPT then
            OPT[j] = max( w_j + MEMOIZED-OPT(OPT, p(j)), MEMOIZED-OPT(OPT, j - 1) )
        else
            return OPT[j]

WIS (n):
    sort jobs by finish times so that f_1 ≤ f_2 ≤ ⋯ ≤ f_n
    compute p(1), p(2), …, p(n)
    create a dictionary (hashtable) OPT
    return MEMOIZED-OPT(OPT, n)
```

OPT[j] is value of optimal solution for jobs **1** to **j**

Running Time?

How many recursive calls?

Weighted Interval Scheduling

How to compute "latest non-conflict job" p(j) ?

This is an exercise for you ☺

```
p(j):
    return Latest-Non-Conflict(s1, …, sj, f1, …, fj, j)


Latest-Non-Conflict(s1, …, sj, f1, …, fj, j):
    # pre-condition: jobs 1, 2, …, j were sorted by finish time


    # write your codes here …
```

What is the running time of your solution?

$\Theta(n)$ or $\Theta(\log n)$?

Dynamic Programming

The moral of the story so far …

Instead of solving the same subproblems repeatedly, as in the naive recursion solution, arrange for each subproblem to be **solved only once**.

There's actually an obvious way to do so: the first time you solve a subproblem, **save its solution**.

Dynamic Programming

The moral of the story so far …

Dynamic programming thus serves as an example of a **time-memory trade-off**. The savings may be dramatic.

A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and you can solve each such subproblem in polynomial time.

Weighted Interval Scheduling

Step 3. Compute OPT(j) using DP

Bottop-up (iterative) approach

BOTTOM-UP-OPT(n):
  create an array OPT of size n + 1
  initialize OPT with [0, 0, 0, …, 0]

  for j = 1 **to** n:
    OPT[j] = max(w_j + OPT[p(j)], OPT[j – 1])

  return OPT[n]

WIS (n):
  sort jobs by finish times so that $f_1 \le f_2 \le \cdots \le f_n$
  compute p(1), p(2), …, p(n)
  return BOTTOM-UP-OPT(n)

OPT(j):
  if j == 0 **then**
    return 0
  else
    return max(w_j + OPT(p(j)), OPT(j - 1))

Recursive Version

WIS (n):
  sort jobs by finish times so that $f_1 \le f_2 \le \cdots \le f_n$
  compute p(1), p(2), …, p(n)
  return OPT(n)

OPT[j] is value of optimal solution for jobs **1** to **j**

Running Time?
Sorting is $\Theta(n\,log(n))$ and the main loop is $\Theta(n)$

Weighted Interval Scheduling

Step 3. Compute OPT(j) using DP

Bottop-up (iterative) approach

BOTTOM-UP-OPT(n):
create an array OPT of size n + 1
initialize OPT with [0, 0, 0, …, 0]

for j = 1 **to** n:
OPT[j] = max(**w_j + OPT[p(j)]**, **OPT[j – 1]**)

return OPT[n]

WIS (n):
sort jobs by finish times so that $f_1 \le f_2 \le \cdots \le f_n$
compute p(1), p(2), …, p(n)
return BOTTOM-UP-OPT(n)

Recursive Version

OPT(j):
if j == 0 **then**
return 0
else
return max(**w_j + OPT(p(j))**, **OPT(j - 1)**)

WIS (n):
sort jobs by finish times so that $f_1 \le f_2 \le \cdots \le f_n$
compute p(1), p(2), …, p(n)
return OPT(n)

The bottop-up version requires $\Theta(n)$ **space**, since the size of the array **OPT** linearly grows with the size of input **n**.

Space Complexity: $\Theta(n)$

What we are still missing here …

- Remember that we still have step 4: **finding a solution**! (not just an optimal value)

- Given the array **OPT** of size **n**, such that **OPT[j]** is optimal value for jobs **1** to **j**, how can you produce a set of optimal jobs?

Weighted Interval Scheduling

Step 4. Reconstructing a solution

We can use the recurrence a second time to **backtrack** through **OPT** array.

```
SOLUTION-REC (OPT, j):
    if j == 0 then
        return []
    else if w_j + OPT[p(j)] > OPT[j – 1] then
        return [j] + SOLUTION-REC(OPT, p(j))     //case 1
    else
        return SOLUTION-REC(OPT, j - 1)          //case 2

// call SOLUTION-REC(OPT, n)
```

prev is an array to remind us of how we obtained the best choice for OPT[j]

Step 4. Reconstructing a solution

Or, we can also modify the bottom-up solution

```
EXTENDED-BOTTOM-UP-OPT(n):
  create an array OPT of size n + 1, initialized with [0, 0, …, 0]
  create an array prev of size n, initialized with [0, 0, …, 0]

  for j = 1 to n:
    take_job = w_j + OPT[p(j)]        //total weight if we take job j
    leave_job = OPT[j – 1]            //total weight if we leave job j
    if take_job > leave_job then
      OPT[j] = take_job
      prev[j] = p(j)                  //previous is p(j)    , we take job j
    else
      OPT[j] = leave_job
      prev[j] = j - 1                 //previous is j – 1    , we leave job j

  return prev, OPT[n]         //return both array 'prev' and the optimal total weights
```

Step 4. Reconstructing a solution

Then we "backtrack" the solution via the array **prev**

```
SOLUTION(prev, n):
    j = n
    solution = []

    while j > 0:
        if prev[j] == p(j) then          // if prev[j] = p(j), job j is taken
            solution = solution + [j]
        j = prev[j]

    return solution
```

Knapsack Problem

Knapsack Problem

- You are given **n** items and a "knapsack"
- Item **i** has weight $w_i > 0$ and value $v_i > 0$
- The knapsack has capacity of **W** kilograms
- **Goal:** Fill knapsack so as to maximize total value

Example:

W = 11

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

Several instances and their values:

{1, 2, 5} ; weight = 10 ; value = 35
{2, 4} ; weight = 8 ; value = 28
{3, 4} ; weight = 11 ; value = 40
{4} ; weight = 6 ; value = 22
...

Optimal

Knapsack Problem

Can you find an **optimal substructure**?

Suppose **OPT(i)** denotes maximum profit for items **{1, 2, 3, …, i}**

- Case 1: **OPT(i)** does not include item **i**
  - In this case, **OPT(i)** selects best of **{1, 2, 3, …, i - 1}**
- Case 2: **OPT(i)** includes item **i**
  - If we accept item **i**, do we have to accept other items **{1, 2, …, i-1}** as well? How do you know that the "knapsack" still has some free space from them?
  - And, importantly, can we really accept item **i**? Are you sure that the knapsack has enough room for **i**?

Can you find an **optimal substructure**?

Suppose **OPT(i)** denotes maximum profit for items **{1, 2, 3, …, i}**

- Case 1: **OPT(i)** does not include item **i**
  - In this case, OPT(i) selects best of **{1, 2, 3, …, i - 1}**

- Case 2: **OPT(i)** includes item **i**
  - If we accept item i, do we have to accept other items **{1, 2, …, i-1}** as well? How do you know that the "knapsack" still has some free space from them?
  - And, importantly, can we really accept item i? Are you sure that the knapsack has enough room for i?

It seems like we are missing another variable: **Weight**!

Knapsack Problem

Yes, we just found the **optimal substructure**. (**Step 1**)

Suppose **OPT(i, w)** = max profit for items **{1, 2, 3, …, i}** with weight limit **w**

- Case 1: **OPT(i, w)** does not include item **i**
  - In this case, OPT(i, w) skips item **i**, **and** selects best of {1, 2, 3, …, i - 1}
  - New weight limit is still **w**

- Case 2: **OPT(i, w)** includes item **i**
  - New weight limit = **w** - w_i
  - OPT(i, w) then selects best of {1, 2, 3, …, i - 1} using the new weight limit

The recurrence relation computes which one of these two cases provides max profit!

Step 2. Define a recurrence relation

Suppose **OPT(i, w)** = max profit for items **{1, 2, 3, …, i}** with weight limit **w**

Special case when the weight of item **i** is greater than the knapsack capacity.

$$OPT(i,w) = \begin{cases} 0 & i = 0 \\ OPT(i-1,w) & w_i > w \\ \max\{\boldsymbol{OPT(i-1,w)},\quad \boldsymbol{v_i + OPT(i-1,w-w_i)}\} & w_i \le w \end{cases}$$

Case 1

Case 2

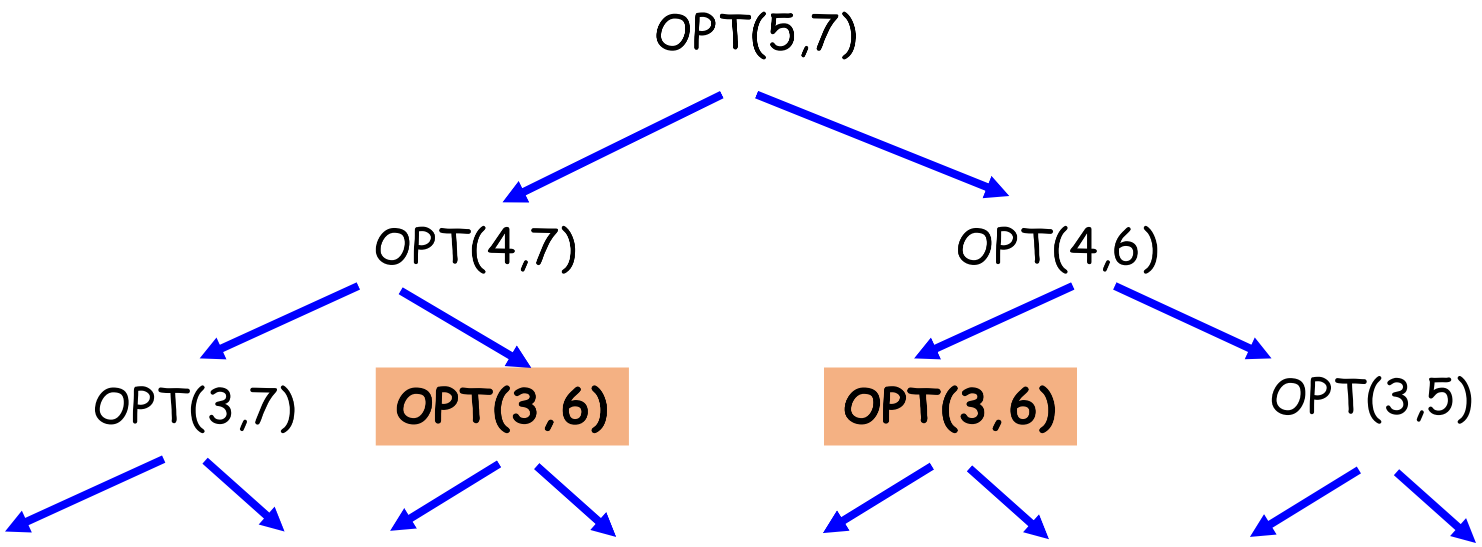
Worst case running time?

Knapsack Problem

Does the recurrence relation have overlapping subproblems? Let's look at the following instance:

W = 7

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



In a case where $\mathbf{w}_i = \mathbf{w}_{i-1}$, subproblems will overlap.

Knapsack Problem

Step 3. Define the solution using Dynamic Programming (Bottom-Up)

Time-Memory Trade-Off --> we need a **(n+1)-by-(W+1)** matrix (OPT)

```
BOTTOM-UP-KNAPSACK(n, W, [w_1, …, w_n], [v_1, …, v_n]):
    create a 2-dimensional array OPT with size (n+1) x (W+1)

    for w = 0 to W:
        OPT[0, w] = 0

    for i = 1 to n:
        for w = 1 to W:
            if w_i > w then
                OPT[i, w] = OPT[i – 1, w]
            else
                OPT[i, w] = max( OPT[i – 1, w] , v_i + OPT[i – 1, w – w_i] )

    return OPT[n, W]
```

Exercise:

Develop a top-down solution (memoization) !

Knapsack Problem

Step 3. Define the solution using Dynamic Programming (Bottom-Up)

<- W ->

n

OPT matrix	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	34	40

W = 11

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

Optimal Solution = {3, 4} with total value = 40

Space complexity = $\Theta(nW)$ since there are only $\Theta(nW)$ distinct subproblems.

Knapsack Problem

Running time is $\Theta(n\ W)$. Is this really a polynomial solution?

Indeed, that is a polynomial function of n and W. However, that is **not polynomial** in the **size of the input** – the number of bits required to represent the input.

Suppose **W = 10^{12}**, so we need **$\log_2 10^{12}$ = 40** bits to represent **W**, and so the input size is **40**. However, the running time uses the factor of **10^{12}**, which is **2^{40}**.

So, it's actually **exponential in input size**, with $\Theta\left(n\ 2^{bits(W)}\right)$

Step 4. Construct and optimal solution given the matrix ***OPT***

```
KNAPSACK-SOL(i, w, OPT, [w_1, ..., w_n], [v_1, ..., v_n]):
    if i == 0 then
        return []
    elif w_i > w then
        return KNAPSACK-SOL(i - 1, w, OPT, [w_1, ..., w_n], [v_1, ..., v_n])
    elif v_i + OPT[i - 1, w - w_i] > OPT[i - 1, w] then
        return [i] + KNAPSACK-SOL(i - 1, w - w_i, OPT, [w_1, ..., w_n], [v_1, ..., v_n])
    else
        return KNAPSACK-SOL(i - 1, w, OPT, [w_1, ..., w_n], [v_1, ..., v_n])
```

Call KNAPSACK-SOL(n, W, OPT, [w_1 …], [v_1 …])

Worst case running time is $\Theta(n)$

Knapsack Problem

When we are only interested in the optimal value, the bottom-up version can have a space-optimized solution with $\Theta(W)$ **space complexity**, since computing current row only requires previous row.

```
BOTTOM-UP-LINEAR-SPACE(n, W, [w_1, …, w_n], [v_1, …, v_n]):
    create an array OPT with size (W+1)

    for w = 0 to W:
        OPT[w] = 0

    for i = 1 to n:
        for w = W down to 1:
            if w_i <= w then
                OPT[w] = max( OPT[w] , v_i + OPT[w - w_i] )

    return OPT[W]
```

Exercise

- The **Subset Sum** problem introduced in the beginning of this slide is “similar” to the Knapsack Problem.

- Solve the **Subset Sum** problem using Dynamic Programming!