Lecture 19

Recap part A

CS 161 Design and Analysis of Algorithms

Ioannis Panageas

Divide and conquer method

- Steps of method:
  - **Divide** input into parts (smaller problems)
  - **Conquer** (solve) each part recursively
  - **Combine** results to obtain solution of original

$$\begin{aligned} T(n) &= \text{divide time} \\ &+ T(n_1) + T(n_2) + \ldots + T(n_k) \\ &+ \text{combine time} \end{aligned}$$

Mergesort - A fast sorting recursive Algorithm

- Key idea:

  Divide input into two parts of equal size

  Sort each part recursively

  Merge the two sorted parts to obtain the solution.

Example: Sort the following 11 numbers

9 3 4 220 1 3 10 5 8 7 2

Divide

Recursion

Merge

Mergesort - A fast sorting recursive Algorithm

- Key idea:

  Divide input into two parts of equal size
  Sort each part recursively
  Merge the two sorted parts to obtain the solution.

Example: Sort the following 11 numbers

9 3 4 220 1 3 10 5 8 7 2

9 3 4 220 1   3 10 5 8 7 2

Divide

Recursion

Merge

Mergesort - A fast sorting recursive Algorithm

- Key idea:

  Divide input into two parts of equal size

  Sort each part recursively

  Merge the two sorted parts to obtain the solution.

Example: Sort the following 11 numbers

9 3 4 220 1 3 10 5 8 7 2

9 3 4 220 1 3 10 5 8 7 2

Divide

1 3 4 9 220 2 3 5 7 8 10

Recursion

Merge

Mergesort - A fast sorting recursive Algorithm

- Key idea:

  Divide input into two parts of equal size

  Sort each part recursively

  Merge the two sorted parts to obtain the solution.

Example: Sort the following 11 numbers

9 3 4 220 1 3 10 5 8 7 2

9 3 4 220 1 3 10 5 8 7 2 **Divide**

1 3 4 9 220 2 3 5 7 8 10 **Recursion**

1 2 3 3 4 5 7 8 9 10 220 **Merge**

Mergesort - A fast sorting recursive Algorithm

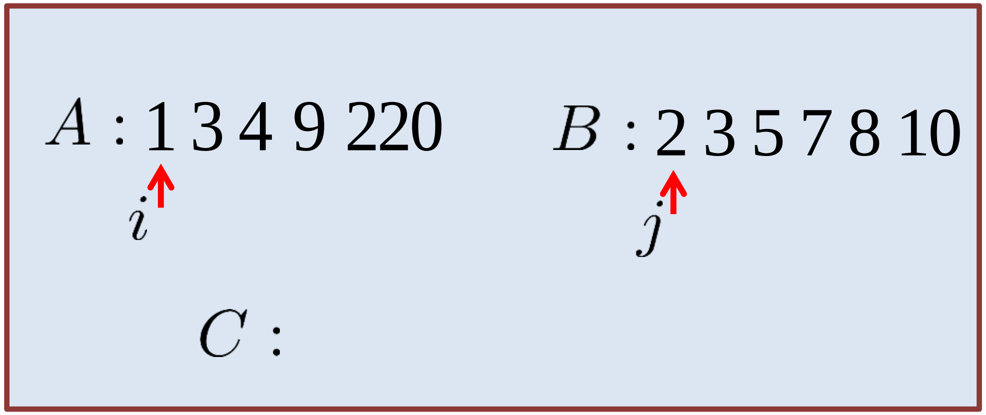
- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A: 1\ 3\ 4\ 9\ 220$

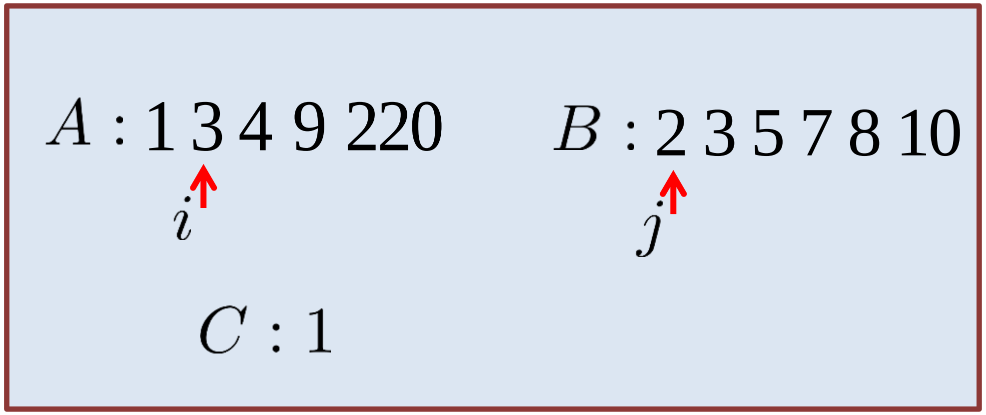
$B: 2\ 3\ 5\ 7\ 8\ 10$

$C:$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A : 1\ 3\ 4\ 9\ 220$

$B : 2\ 3\ 5\ 7\ 8\ 10$

i

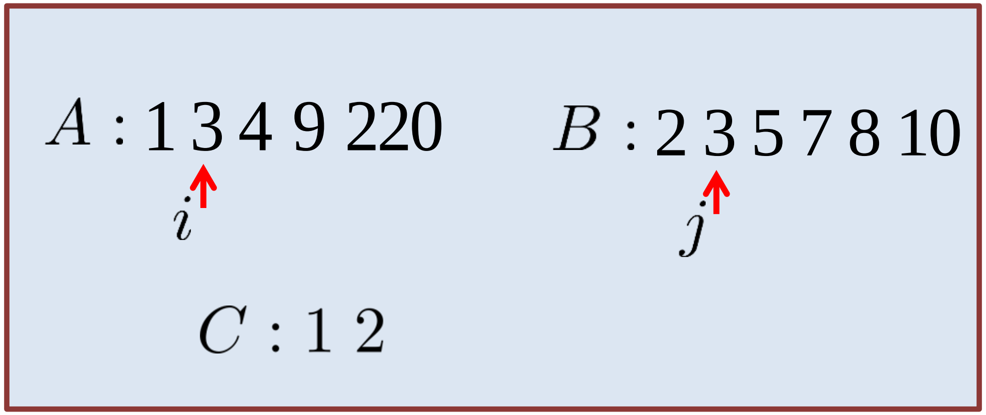
j

$C : 1$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A: 1\ 3\ 4\ 9\ 220$

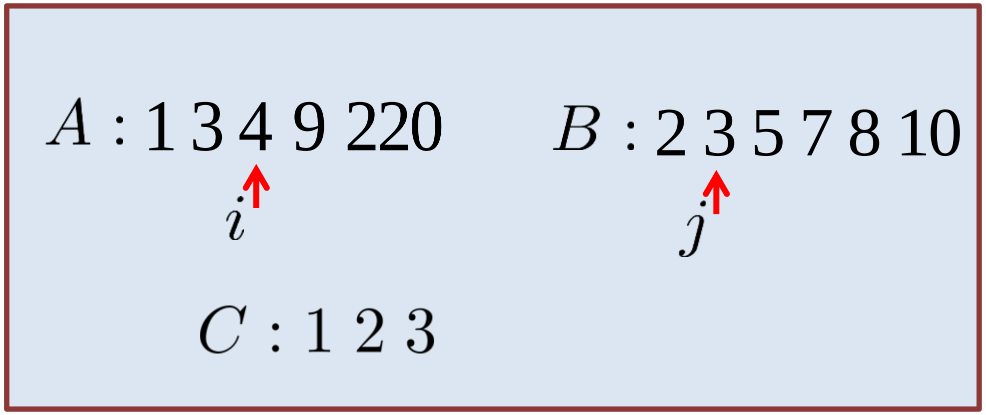
$B: 2\ 3\ 5\ 7\ 8\ 10$

$C: 1\ 2$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A : 1\ 3\ 4\ 9\ 220$

i

$B : 2\ 3\ 5\ 7\ 8\ 10$

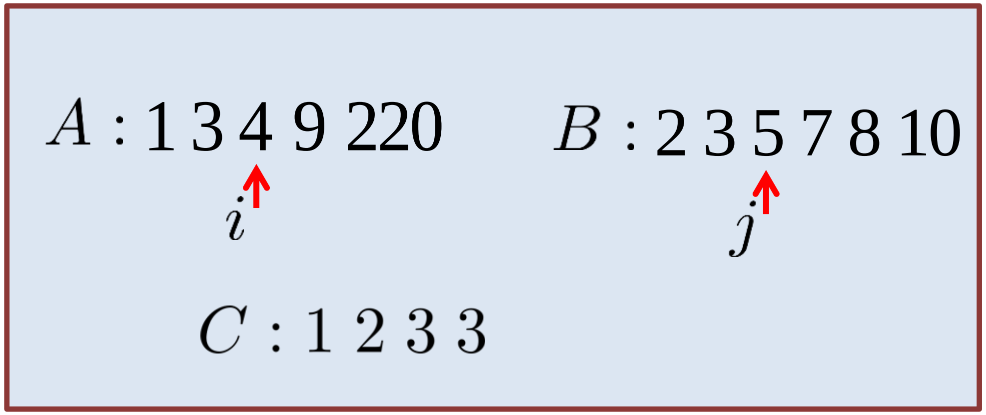
j

$C : 1\ 2\ 3$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A: 1\ 3\ 4\ 9\ 220$

$B: 2\ 3\ 5\ 7\ 8\ 10$

i

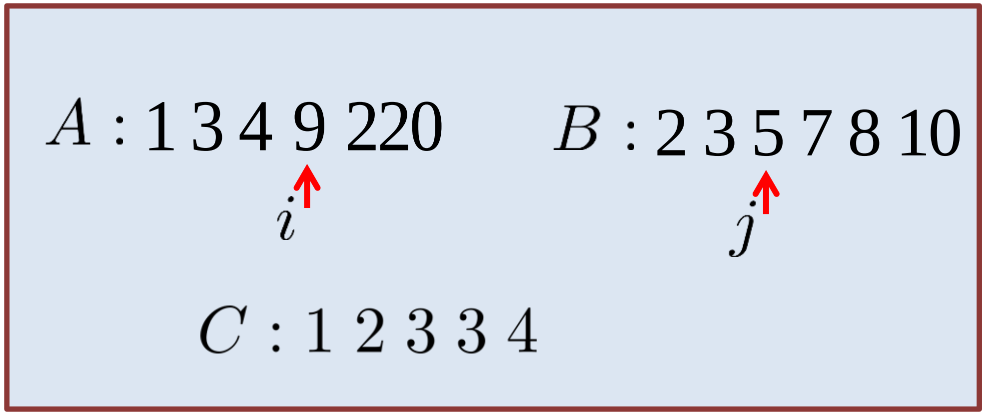
j

$C: 1\ 2\ 3\ 3$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A : 1\ 3\ 4\ 9\ 220$

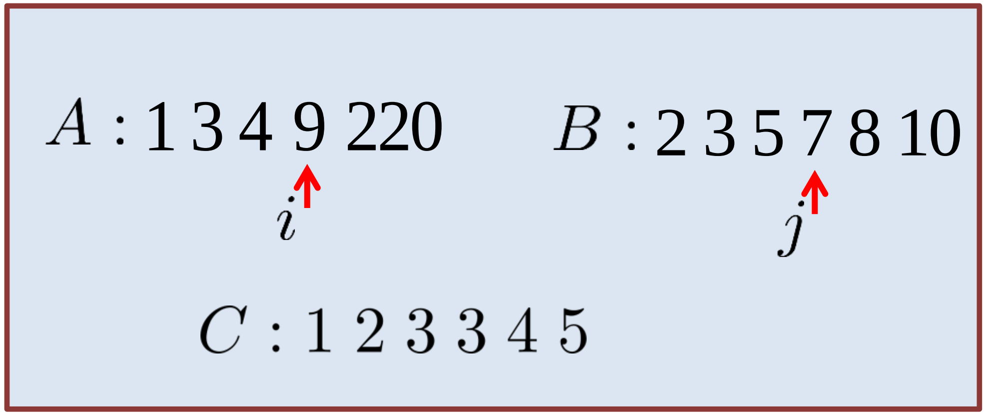
$B : 2\ 3\ 5\ 7\ 8\ 10$

$C : 1\ 2\ 3\ 3\ 4$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



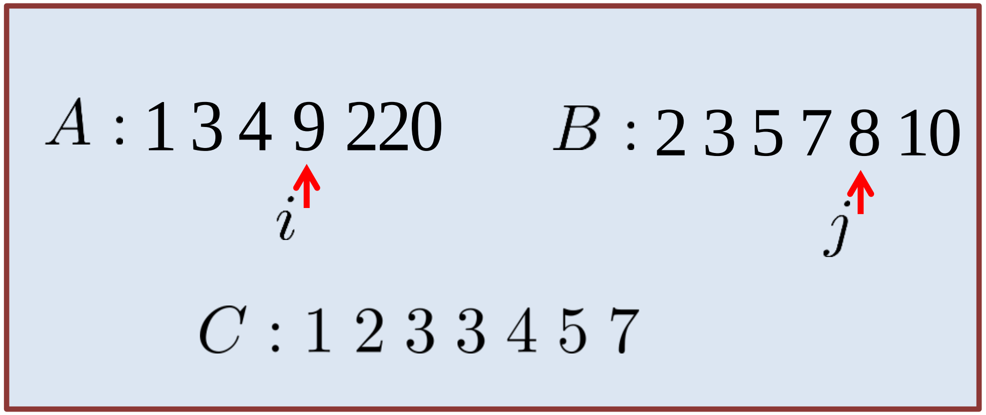
$A : 1\ 3\ 4\ 9\ 220$ $\quad B : 2\ 3\ 5\ 7\ 8\ 10$

$C : 1\ 2\ 3\ 3\ 4\ 5$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



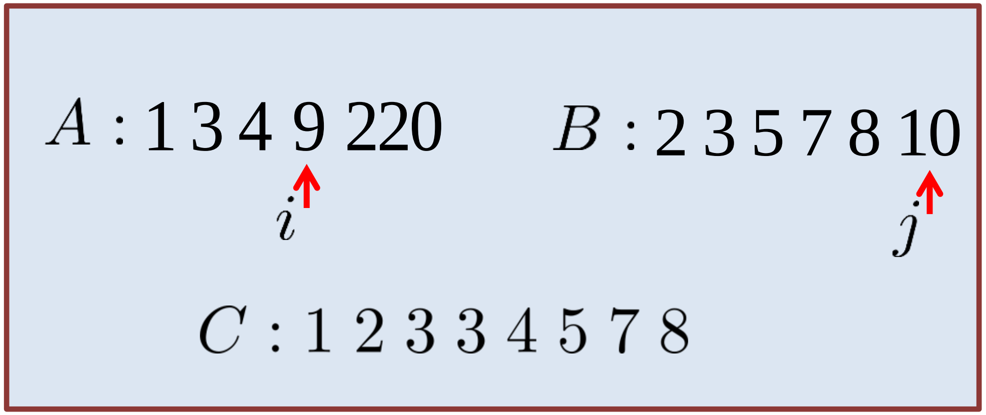
$$A : 1\ 3\ 4\ 9\ 220 \qquad B : 2\ 3\ 5\ 7\ 8\ 10$$

$$C : 1\ 2\ 3\ 3\ 4\ 5\ 7$$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A: 1\ 3\ 4\ 9\ 220$

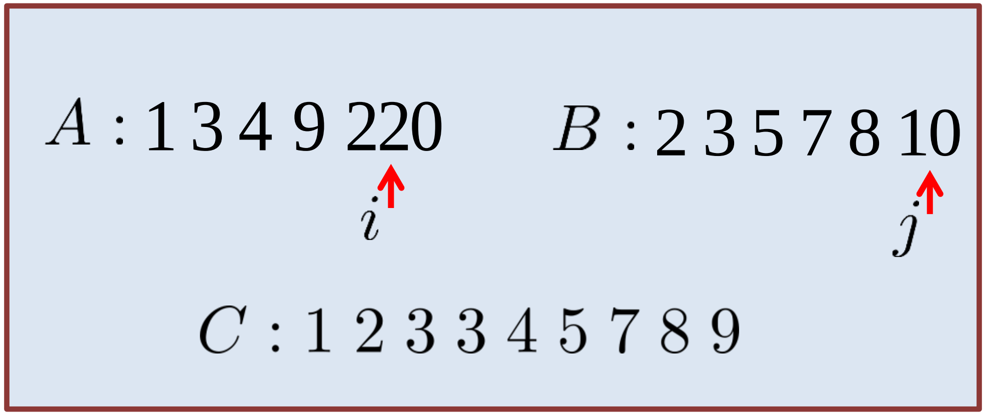
$B: 2\ 3\ 5\ 7\ 8\ 10$

$C: 1\ 2\ 3\ 3\ 4\ 5\ 7\ 8$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A : 1\ 3\ 4\ 9\ 220$ $\qquad$ $B : 2\ 3\ 5\ 7\ 8\ 10$

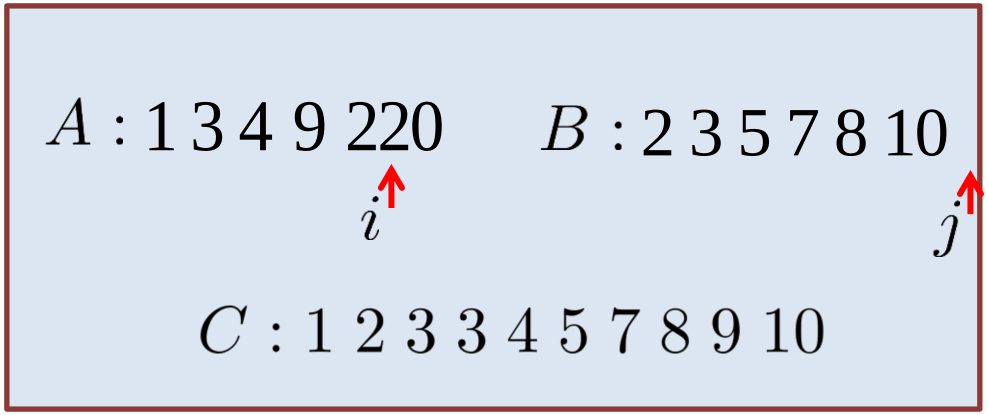
i $\qquad$ j

$C : 1\ 2\ 3\ 3\ 4\ 5\ 7\ 8\ 9$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.



$A : 1\,3\,4\,9\,220$ $\quad B : 2\,3\,5\,7\,8\,10$

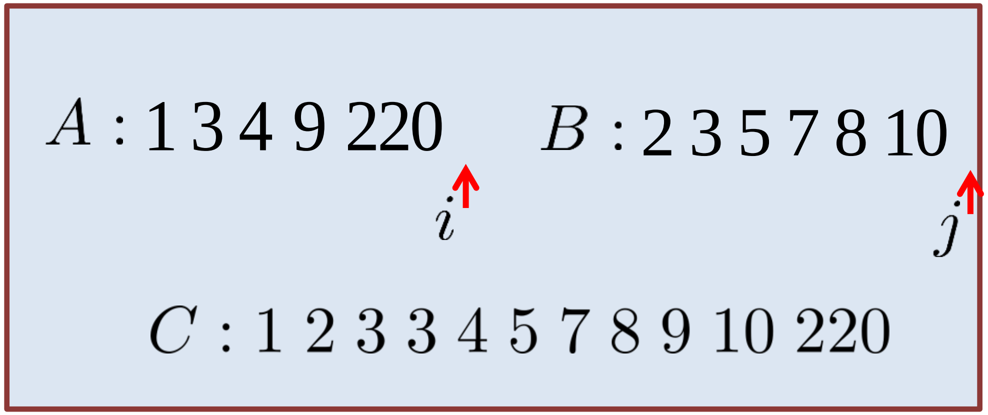
$C : 1\;2\;3\;3\;4\;5\;7\;8\;9\;10$

Mergesort - A fast sorting recursive Algorithm

- Tricky part: Merge

Problem: Given two sorted arrays A, B, merge them to a sorted array C.

Running time: $\Theta(n)$



$A : 1\ 3\ 4\ 9\ 220$

$B : 2\ 3\ 5\ 7\ 8\ 10$

$C : 1\ 2\ 3\ 3\ 4\ 5\ 7\ 8\ 9\ 10\ 220$

Mergesort

- Pseudocode:

```
Mergesort($A[1:n]$)
    If $n == 1$ then
        return $A$
    Mergesort $(A[1:\frac{n}{2}])$
    Mergesort $(A[\frac{n}{2}+1:n])$
    $C \leftarrow$ Merge$(A[1:\frac{n}{2}], A[\frac{n}{2}+1:n])$
    return $C$
```

- Running time:

$$T(n) = T(n/2) + T(n/2) + \Theta(n) + \Theta(1)$$

$$= 2T(n/2) + \Theta(n)$$

How to analyze?

Master theorem

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ aT(n/b) + f(n) \end{cases}$$

- The Master Theorem can find the order of $T(n)$ which is defined recursively.

Master theorem

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ aT(n/b) + f(n) \end{cases}$$

- The Master Theorem can find the order of $T(n)$ which is defined recursively.

- Key idea: The answer depends on the comparison between $f(n)$ and $n^{\log_b a}$. So, there are **3** cases!

Master theorem

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ aT(n/b) + f(n) \end{cases}$$

1. **If $f(n)$ is $O(n^{\log_b a-\epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$**
2. If $f(n)$ is $\Theta(n^{\log_b a}\log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a}\log^{k+1} n)$
3. If $f(n)$ is $\Omega(n^{\log_b a+\epsilon})$, then $T(n)$ is $\Theta(f(n))$,
   need to check $af(n/b) < f(n)$.

Case 1: $n^{\log_b a}$ **dominates** $f(n)$

Master theorem

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ aT(n/b) + f(n) \end{cases}$$

1. If $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. **If** $f(n)$ **is** $\Theta(n^{\log_b a} \log^k n)$**, then** $T(n)$ **is** $\Theta(n^{\log_b a} \log^{k+1} n)$
3. If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, then $T(n)$ is $\Theta(f(n))$, need to check $af(n/b) < f(n)$.

Case 2: $n^{\log_b a}$ **have same order as** $f(n)$ (up to $\log^k n$)

Master theorem

$$T(n) = \begin{cases} T(1) = \Theta(1) \\ aT(n/b) + f(n) \end{cases}$$

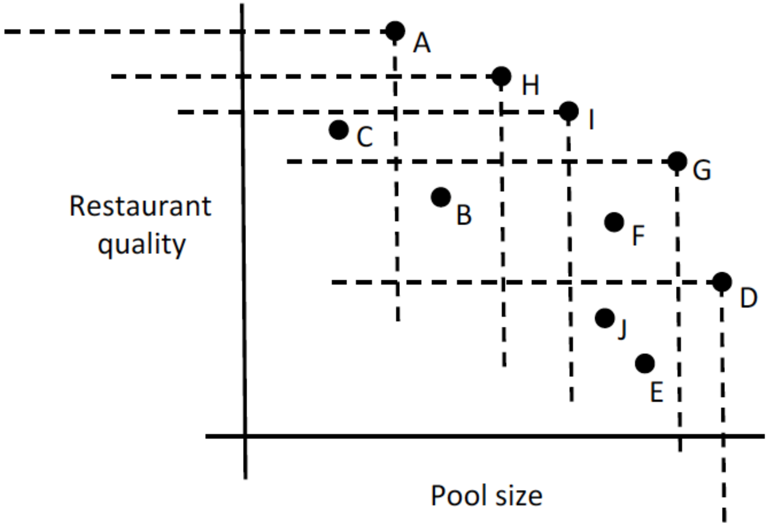
1. If $f(n)$ is $O(n^{\log_b a - \epsilon})$, then $T(n)$ is $\Theta(n^{\log_b a})$
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then $T(n)$ is $\Theta(n^{\log_b a} \log^{k+1} n)$
3. **If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, then $T(n)$ is $\Theta(f(n))$, need to check $af(n/b) < f(n)$.**

Case 3: $n^{\log_b a}$ **is dominated by $f(n)$** (+ another condition)

Case study: Maxima Set

Problem: We are given n points $(x_1, y_1), \dots, (x_n, y_n)$ on the plane. A point (x_i, y_i) is called a maximum point if there is no other point (x_j, y_j) that $x_i \leq x_j$ and $y_i \leq y_j$.

Example: x captures pool size and y restaurant quality. 10 hotels

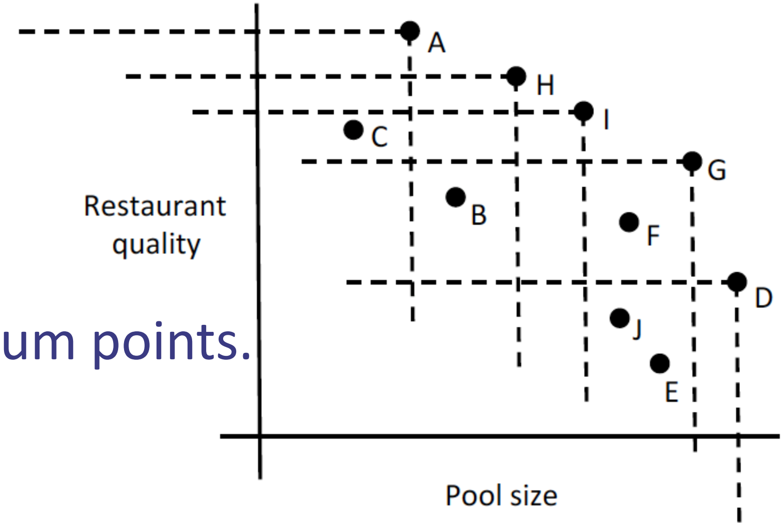


Case study: Maxima Set

Problem: We are given n points $(x_1, y_1), \ldots, (x_n, y_n)$ on the plane. A point (x_i, y_i) is called a **maximum point** if there is no other point (x_j, y_j) that $x_i \leq x_j$ and $y_i \leq y_j$.

Example: x captures pool size and y restaurant quality. 10 hotels

Explanation:



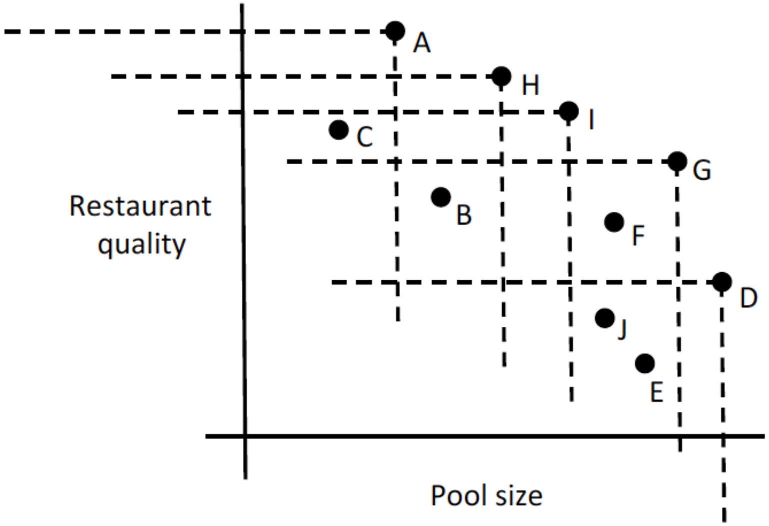
A, H, I, G, D are maximum points.

C, B, F, J, E are not.

Case study: Maxima Set

Problem: We are given n points $(x_1, y_1), \ldots, (x_n, y_n)$ on the plane. A point (x_i, y_i) is called a maximum point if there is no other point (x_j, y_j) that $x_i \leq x_j$ and $y_i \leq y_j$.

Idea: Divide and conquer. Divide step and Combine step is challenging.



Case study: Maxima Set

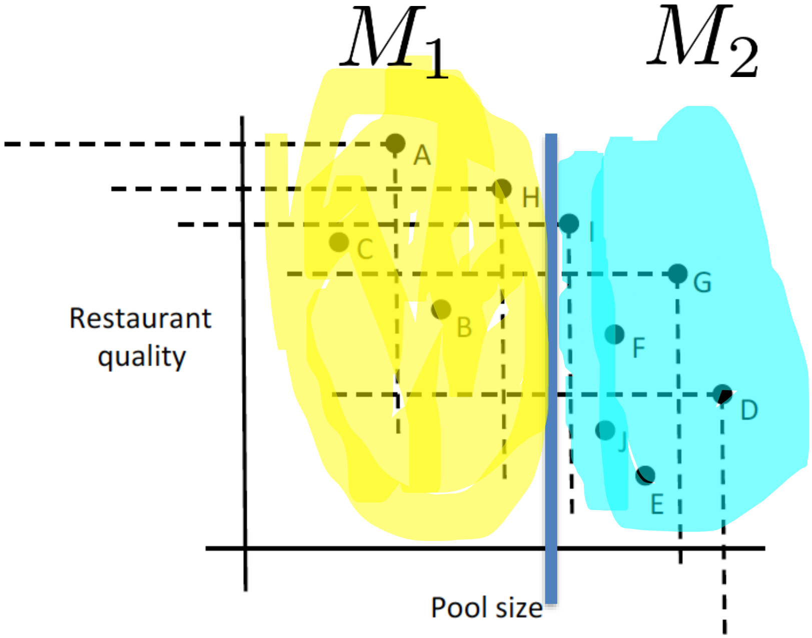
Divide step: It should split the points in two parts of equal size.

How?

Case study: Maxima Set

Divide step: It should split the points in two parts of equal size.

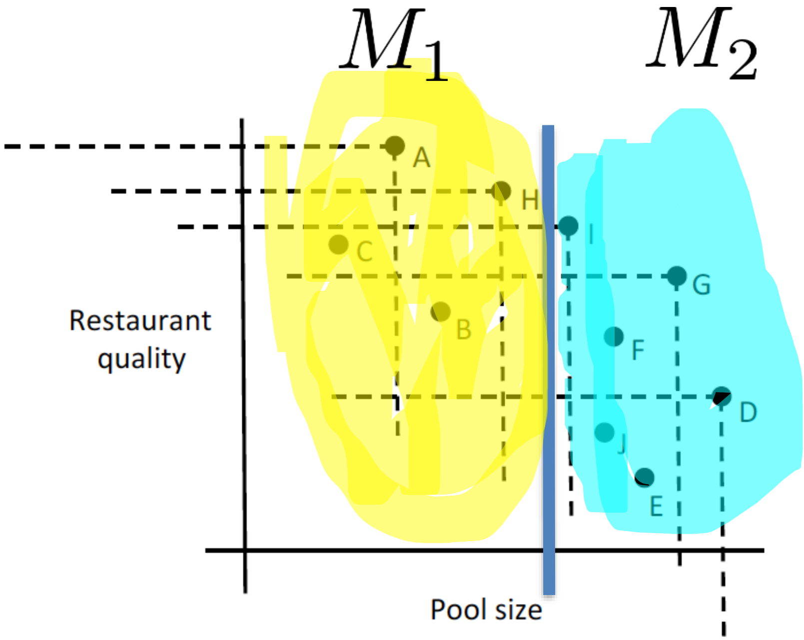
How? Choose the middle (median) point with respect to x coordinates.



Case study: Maxima Set

Divide step: It should split the points in two parts of equal size.

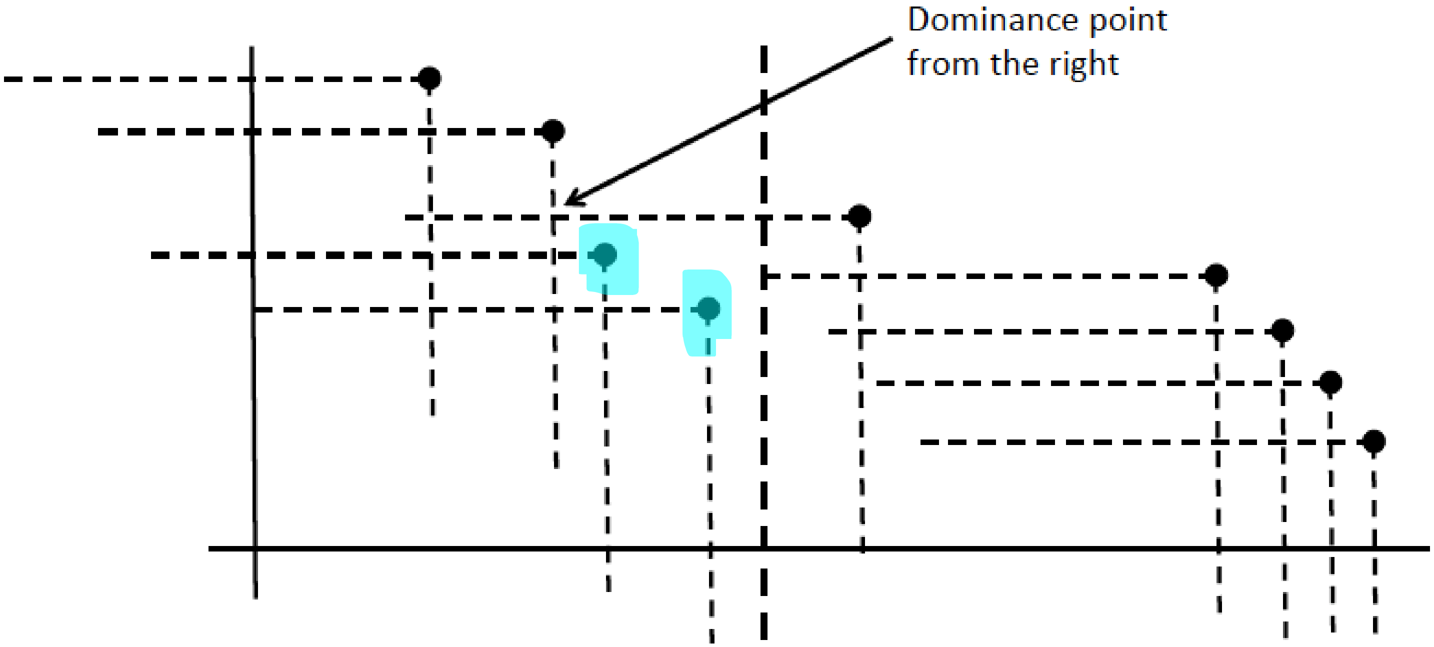
How? Choose the middle (median) point with respect to x coordinates.



Combine step: Return $M_1 \cup M_2$?

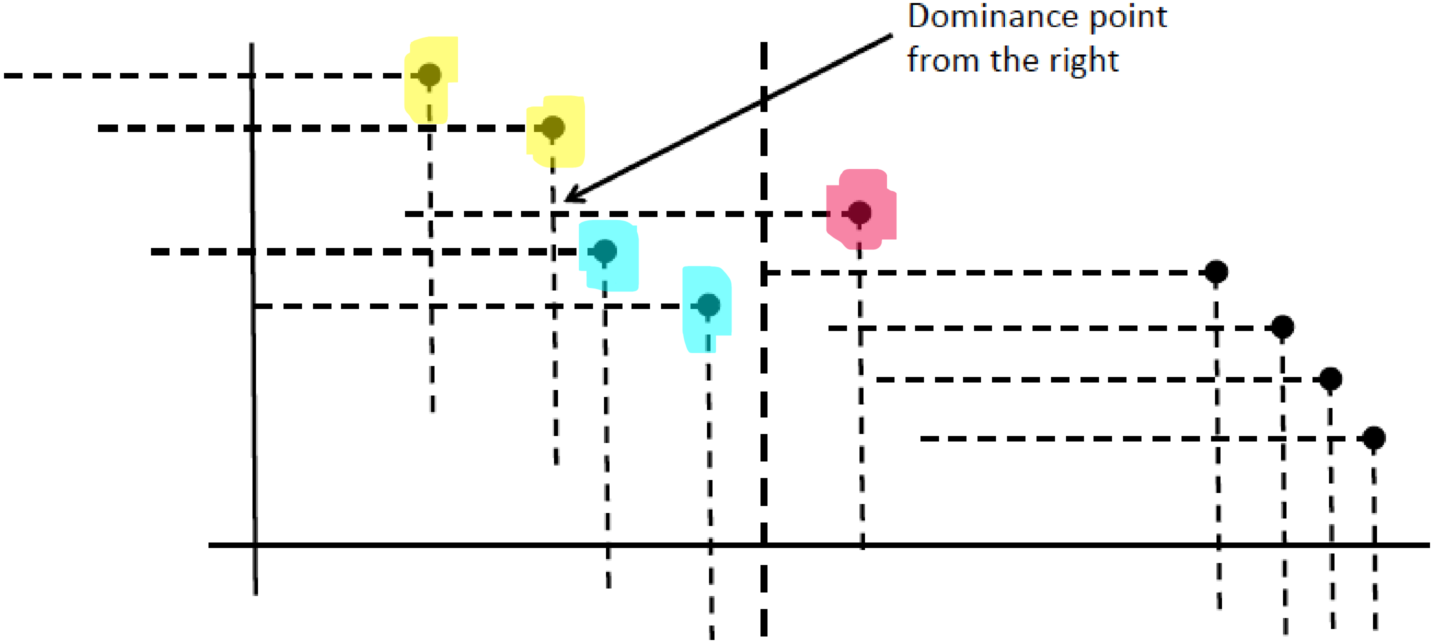
Case study: Maxima Set

Combine step: Return $M_1 \cup M_2$? Wrong: blue points below of M_1 are not part of the solution



Case study: Maxima Set

Combine step idea: M_2 points should part of the solution. From M_1, the points that are maximum should not be dominated by smallest with respect to x coordinates in M_2



Case study: Maxima Set

Pseudocode:

```
MaximaSet($S$,$n$):
    if $n = 1$ then
        return $S$
    Let $p$ be the median point in $S$, by $x$ -coordinates
    Let $L$ be the set of points less than $p$ in $S$ by $x$ -coordinates
    Let $G$ be the set of points greater than or equal to $p$ in $S$ by $x$ -coordinates
    $M_1 \leftarrow$ MaximaSet($L$)
    $M_2 \leftarrow$ MaximaSet($G$)
    Let $q$ be the smallest point in $M_2$
    for each point, $r$, in $M_1$ do by $x$ -coordinates
        if $x(r) \leq x(q)$ and $y(r) \leq y(q)$ then
            Remove $r$ from $M_1$
    return $M_1 \cup M_2$
```

Running time??

$$\text{Running time is } T(n) = 2T(n/2) + T_{\text{media}}(n) + T_{\text{min}}(n) + \Theta(n)$$

$$= 2T(n/2) + T_{\text{media}}(n) + \Theta(n)$$

Dynamic Programming

Technique for solving optimization problems.

Solve problem by solving **sub**-problems and combine:

This is called Optimal substructure property.

Dynamic Programming

Technique for solving optimization problems.

Solve problem by solving **sub**-problems and combine:

This is called Optimal substructure property.

- Similar to divide-and-conquer: recursion (for solving sub-problems)
- Sub-problems overlap: solve them only once!

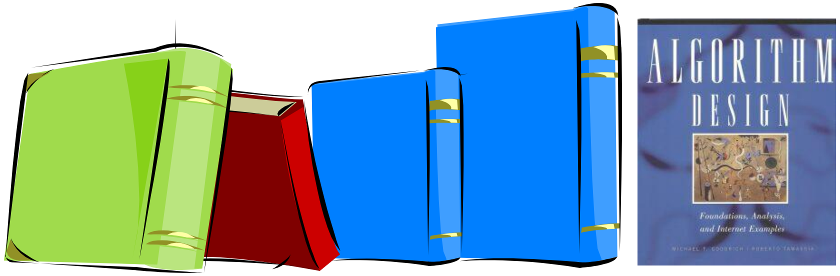
DP = recursion + re-use (Memoization)

Case study II: 0/1 Knapsack

Problem: A set of n items, with each item i having positive weight w_i and positive benefit v_i. You are asked to choose items with maximum total benefit so that the total weight is at most W

Example:

Items:



Weight:	4 lbs	2 lbs	2 lbs	6 lbs	2 lbs
Benefit:	$20	$3	$6	$25	$80

"knapsack" with 9 lbs capacity

Solution:

- item 5 ($80, 2 lbs)
- item 3 ($6, 2lbs)
- item 1 ($20, 4lbs)

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (first attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k]$ be the maximum value I can get from items $\{1, \ldots, k\}$ without exceeding W.

Step 2: Define the goal/output given Step 1.
It is $\boldsymbol{DP[n]}$.

Step 3: Define the base cases
It is $DP[0] = 0$.

Step 4: Define the recurrence

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (first attempt).

Step 4: Define the recurrence

Item k will be used or not.

$$DP[k] \ = \ \max(DP[k-1], DP[k-1] + v_k)$$

But how do we know that DP[k-1] does not exceed $W - w_k$ in weight so we can use k?

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the maximum value I can get from items $\{1, \ldots, k\}$ without exceeding j.

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the maximum value I can get from items $\{1, \ldots, k\}$ without exceeding j.

Step 2: Define the goal/output given Step 1.
It is $\boldsymbol{DP[n, W]}$.

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 1: Define the problem and subproblems.

Answer: Let $DP[k, j]$ be the maximum value I can get from items $\{1, \ldots, k\}$ without exceeding j.

Step 2: Define the goal/output given Step 1.

It is $\boldsymbol{DP[n, W]}$.

Step 3: Define the base cases

It is $DP[0, j] = 0$ for all j and $DP[i, 0] = 0$ for all i.

Step 4: Define the recurrence

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \max(\mathbf{DP[k-1][j-w_k]+v_k}, \mathbf{DP[k-1][j]})$$

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \max(\mathbf{DP[k-1][j-w_k]+v_k}, \mathbf{DP[k-1][j]})$$

Question: How do we know that item k does not have weight more than j?

Case study II: 0/1 Knapsack

Idea: Dynamic Programming (correct attempt).

Step 4: Define the recurrence

Item k will be **used** or **not**.

$$DP[k][j] = \begin{cases} \text{if } w_k \leq j & \max(\mathbf{DP[k-1][j-w_k]+v_k}, \mathbf{DP[k-1][j]}) \\ \text{If } w_k > j & \boldsymbol{DP[k-1][j]} \end{cases}$$

Answer: Add an if statement in the recurrence.

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$

$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

Initialization:

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0				
2	0				
3	0				

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0 $(j < w_1)$			
2	0	0 $(j < w_2)$			
3	0	0 $(j < w_3)$			

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	max(0,v_1+0)		
2	0	0			
3	0	0			

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1		
2	0	0	max(1,v_2+0)		
3	0	0			

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1		
2	0	0	1		
3	0	0	1 ($j < w_3$)		

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$

$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	max(0,v_1+0)	
2	0	0	1		
3	0	0	1		

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$

$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	
2	0	0	1	max(1,v_2+0)	
3	0	0	1		

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$

$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	
2	0	0	1	1	
3	0	0	1	max(1,v_3+0)	

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	max(0,v_1+0)
2	0	0	1	1	
3	0	0	1	5	

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	1
2	0	0	1	1	max(1,v_2+1)
3	0	0	1	5	

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$
$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	1
2	0	0	1	1	2
3	0	0	1	5	max(2,0+v_3)

Case study II: 0/1 Knapsack

Example: 3 items, $W = 4$

$w_1 = 2, v_1 = 1, w_2 = 2, v_2 = 1, w_3 = 3, v_3 = 5$

	j=0	1	2	3	4
i=0	0	0	0	0	0
1	0	0	1	1	1
2	0	0	1	1	2
3	0	0	1	5	**5**

Case study II: 0/1 Knapsack

Pseudocode:

```
Array DP[][]
For $i = 0$ to $n$ do
  $\mathrm{DP}[i, 0] \leftarrow 0$
For $j = 1$ to $W$ do
  $\mathrm{DP}[0, j] \leftarrow 0$
For $i = 1$ to $n$ do
  For $j = 1$ to $W$ do
    If $j < w_i$ then
      $\mathrm{DP}[i][j] \leftarrow \mathrm{DP}[i-1][j]$
    else $\mathrm{DP}[i][j] \leftarrow \max(\mathrm{DP}[i-1][j], \mathrm{DP}[i-1][j-w_i] + v_i$
return $\mathrm{DP}[n][W]$
```

Initialization

Bottom up filliing DP

Goal

Case study II: 0/1 Knapsack

Pseudocode:

```
Array DP[][]
For $i = 0$ to $n$ do
  $DP[i, 0] \leftarrow 0$
For $j = 1$ to $W$ do
  $DP[0, j] \leftarrow 0$
For $i = 1$ to $n$ do
  For $j = 1$ to $W$ do
    If $j < w_i$ then
      $DP[i][j] \leftarrow DP[i-1][j]$
    else $DP[i][j] \leftarrow \max(DP[i-1][j], DP[i-1][j-w_i] + v_i$
return $DP[n][W]$
```

Initialization

Bottom up filliing DP

Goal

Running time: $\Theta(nW)$