

# Design and Analysis of Algorithms

## Greedy Algorithms (I)

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

# Outline

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

## Motivation

A game like chess can be won only by *thinking ahead*

- a player who is focused entirely on immediate advantages is easy to defeat.

But in many other games, such as Scrabble

- it's fine to make whichever move seems best at the moment and not worrying too much about future consequences.





## Greedy Algorithm

Greedy algorithm works: proof of correctness

- Interval scheduling: induction on step
- Optimal loading: induction on input size
- Scheduling to minimum lateness: exchange argument

Greedy algorithm does not work: find a counter-example

- Coin changing problem

# Outline

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling**
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

## Interval Scheduling

**Input.**  $S = \{1, 2, \dots, n\}$  is a set of  $n$  jobs, job  $i$  starts at  $s_i$  and finishes at  $f_i$ .

- Two jobs  $i$  and  $j$  are **compatible** if they don't overlap:  
 $s_i \geq f_j$  or  $s_j \geq f_i$

**Goal:** find maximum subset of mutually compatible jobs.

## Interval Scheduling

**Input.**  $S = \{1, 2, \dots, n\}$  is a set of  $n$  jobs, job  $i$  starts at  $s_i$  and finishes at  $f_i$ .

- Two jobs  $i$  and  $j$  are **compatible** if they don't overlap:  
 $s_i \geq f_j$  or  $s_j \geq f_i$

**Goal:** find maximum subset of mutually compatible jobs.

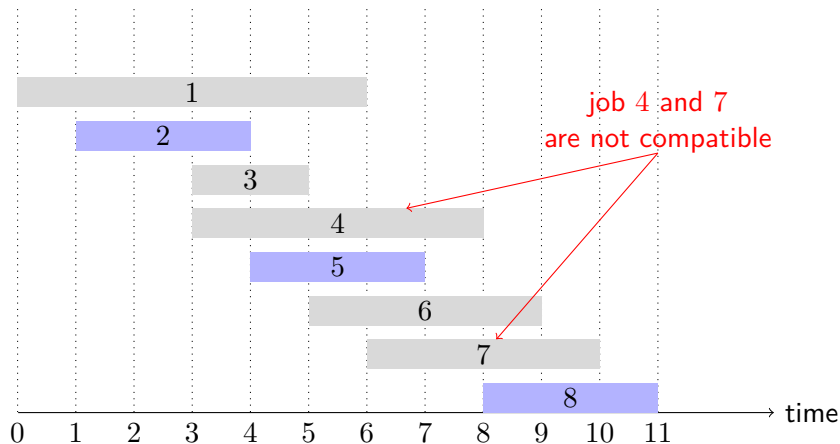
**Instance**

$i$	1	2	3	4	5	6	7	8
$s_i$	0	1	3	3	4	5	6	8
$f_i$	6	4	5	8	7	9	10	11

**Solution.**  $\{2, 5, 8\}$



## Example



# Interval Scheduling: Greedy Algorithm

## Greedy template

- Consider jobs in some **natural order**, then take each job provided it's compatible with the ones already taken.
- Selection strategy is short-sighted  $\leadsto$  the order might not be optimal

# Interval Scheduling: Greedy Algorithm

## Greedy template

- Consider jobs in some **natural order**, then take each job provided it's compatible with the ones already taken.
- Selection strategy is short-sighted  $\leadsto$  the order might not be optimal

## Candidate selection strategies

- [Earliest start time] Consider jobs in ascending order of  $s_i$
- [Earliest finish time] Consider jobs in ascending order of  $f_i$
- [Shortest interval] Consider jobs in ascending order of  $f_i - s_i$
- [Fewest conflicts] For each job  $j$ , count the number of conflicting jobs  $c_j$ . Schedule in ascending order of  $c_j$ .

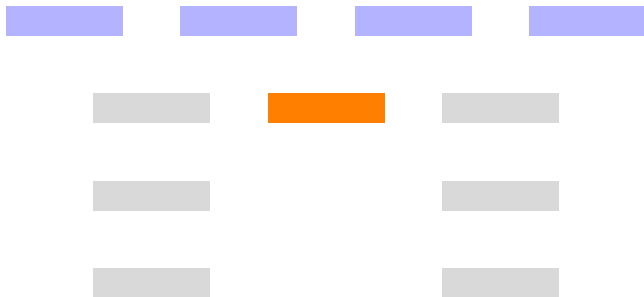
## Counterexample for Earliest Start Time



## Counterexample for Shortest Interval



## Counterexample for Fewest Conflicts



## Greedy Algorithm: Earliest-Finish-Time-First

---

**Algorithm 1:** GreedySelect( $S, s_i, f_i, i \in [n]$ )

---

**Output:** maximum compatible subset  $A \subseteq S$

- 1: Sort jobs by finish time so that  $f_1 \leq \dots \leq f_n$ ;
  - 2:  $n \leftarrow |S|$ ;
  - 3:  $A \leftarrow \emptyset$ ;
  - 4: **for**  $i \leq 1$  **to**  $n$  **do**
  - 5:     **if** *job  $i$  is compatible with  $A$*  **then**  $A \leftarrow A \cup \{i\}$ ;
  - 6: **end**
  - 7: **return**  $A$ ;
- 

**Q.** How to decide job  $i$  is compatible with  $A$ ?

**A.** Keep track of job  $j^*$  that was last added to  $A$ . Job  $i$  is compatible with  $A$  iff  $s_i \geq f_{j^*}$  holds.

## Demo of Earliest Finish Time First

Input.  $S = \{1, 2, \dots, 8\}$

$i$	1	2	3	4	5	6	7	8
$s_i$	0	1	3	3	4	5	6	8
$f_i$	6	4	5	8	7	9	10	11

Solut.  $A = \{2, 4, 8\}$

Complexity. overall  $\Theta(n \log n)$

- Sorting by finish time:  $\Theta(n \log n)$
- Compare to check compatible:  $O(n)$

**Lemma.** Earliest-finish-time-first algorithm always give the correct solution.

*How to prove it?*



# Mathematic Induction for Greedy Algorithm

## Proof template for greedy algorithm

- ① Describe the correctness as a proposition about natural number  $n$ , which claims greedy algorithm yields correct solution.
  - Here,  $n$  could be the algorithm steps or input size.
- ② Prove the proposition is true for all natural number.
  - Induction basis: from the smallest instance
  - Induction steps: type 1 or type 2 induction

## Proposition for Earliest-Finish-Time-First

Let  $S$  be the job set of size  $n$ ,  $s_i$  and  $f_i$  are the start time and finish time,  $A$  be a maximum compatible subset of  $S$ .

**Proposition.** When algorithm GreedySelect carries on the  $k$ -th step, it choose  $k$  jobs  $(i_1 = 1, i_2, \dots, i_k)$ , which is exactly the first  $k$  jobs of  $A$ .

According the above proposition,  $\forall k$ , the first  $k$ -step choice is exactly the first  $k$ -jobs of some maximum compatible subset  $A$ , and will yield  $A$  in at most  $n$  steps.

## Mathematic Induction: Induction Basis

Let  $S = \{1, 2, \dots, n\}$  be the sorted job set:  $f_1 \leq \dots \leq f_n$

## Mathematic Induction: Induction Basis

Let  $S = \{1, 2, \dots, n\}$  be the sorted job set:  $f_1 \leq \dots \leq f_n$

Induction basis.  $k = 1$ , prove  $A$  includes job 1

## Mathematic Induction: Induction Basis

Let  $S = \{1, 2, \dots, n\}$  be the sorted job set:  $f_1 \leq \dots \leq f_n$

**Induction basis.**  $k = 1$ , prove  $A$  includes job 1

For an arbitrary maximum compatible subset  $A$ , sort jobs in  $A$  in ascending order according to the finish time.

If the first job in  $A$  is  $j$  and  $j \neq 1$ , then replace job  $j$  with job 1, yielding  $A'$ :

$$A' = (A - \{j\}) \cup \{1\}$$

- 1 won't appear in  $(A - \{j\}) \Rightarrow |A| = |A'|$
- $f_1 \leq f_j \Rightarrow$  replacement does not affect compatibility  $\Rightarrow A'$  is also one of the maximum compatible subset of  $A$  and includes job 1.



## Mathematic Induction: Induction Step (1/2)

Assume Proposition is true for  $k$ , prove it is also true for  $k + 1$

- $(k + 1)$ -step choice job  $i_{k+1}$  and  $(i_1, \dots, i_k)$  forms the first  $k + 1$  jobs of some  $A$  for  $S$ .

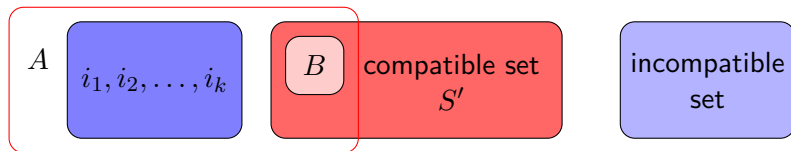
**Proof.** After  $k$  steps, algorithm chooses  $i_1 = 1, i_2, \dots, i_k$ .

Premise  $\Rightarrow \exists$  a maximum compatible  $A$  that contains  $i_1, i_2, \dots, i_k$ .

- Let  $B$  the set of other elements in  $A$  (already sorted and not empty), and  $S'$  be the set of compatible elements w.r.t.  $\{i_1, i_2, \dots, i_k\}$ .

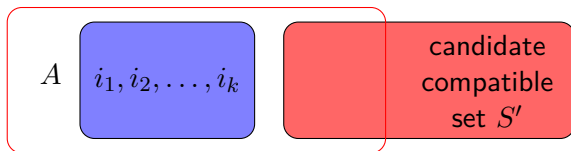
$$A = \{i_1, i_2, \dots, i_k\} \cup B$$

$$S' = \{i \mid i \in S, s_i \geq f_k\}$$



## Mathematic Induction: Induction Step (2/2)

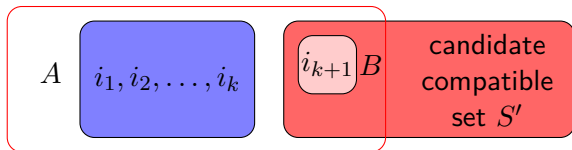
Consider two cases according to if job  $i_{k+1}$  is the 1st job in  $B$ .



## Mathematic Induction: Induction Step (2/2)

Consider two cases according to if job  $i_{k+1}$  is the 1st job in  $B$ .

- If  $i_{k+1}$  happens to be the first job in  $B$ , then the desired result immediately follows,  $(k + 1)$ -step choice still yields the partial solution of  $A$ .





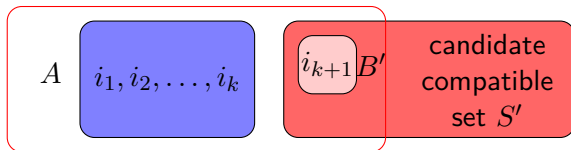
## Mathematic Induction: Induction Step (2/2)

Consider two cases according to if job  $i_{k+1}$  is the 1st job in  $B$ .

- If  $i_{k+1}$  happens to be the first job in  $B$ , then the desired result immediately follows,  $(k+1)$ -step choice still yields the partial solution of  $A$ .
- If  $i_{k+1}$  is not the first job in  $B$ , then we must have  $i_{k+1} \notin B$ 
  - the strategy choice of the greedy algorithm  $\Rightarrow$  the finish time of  $i_{k+1}$  must be earlier than the first job in  $B$
  - At this point, we can replace the first job in  $B$  with job  $i_{k+1}$ , yielding  $B'$ . Obviously,  $|B'| = |B|$ .

$$\{i_1, i_2, \dots, i_k\} \cup B' = A'$$

Note that  $|A| = |A'| \Rightarrow A'$  is still a maximum compatible set of  $S$ . This proves the induction step.



# Outline

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading**
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

# Optimal Loading Problem

**Problem.** Given  $n$  containers with weight  $w_i$  and a boat with maximum weight capacity  $W$  (no volume limit).

**Goal.** A loading plan that maximizes the number of containers on the ship.

**Analysis.** This problem is a special case of 0-1 knapsack problem.

- item: container
- boat: knapsack
- all  $v_i = 1$

## Modeling

Let  $(x_1, x_2, \dots, x_n)$  be the solution vector,  $x_i \in \{0, 1\}$ .

- $x_i = 1$  iff  $i$ -th container is on the boat

Goal function:

$$\max \sum_{i=1}^n x_i$$

Constraint:

$$\sum_{i=1}^n w_i x_i \leq W, x_i = \{0, 1\}, i \in [n]$$

# Algorithm Design

Greedy strategy. lightest first

## Algorithm steps

- sorting container according to weight in ascending order, to ensure  $w_1 \leq w_2 \leq \dots \leq w_n$
- loading the container from the smallest label, and stop until loading next container will exceed the limit

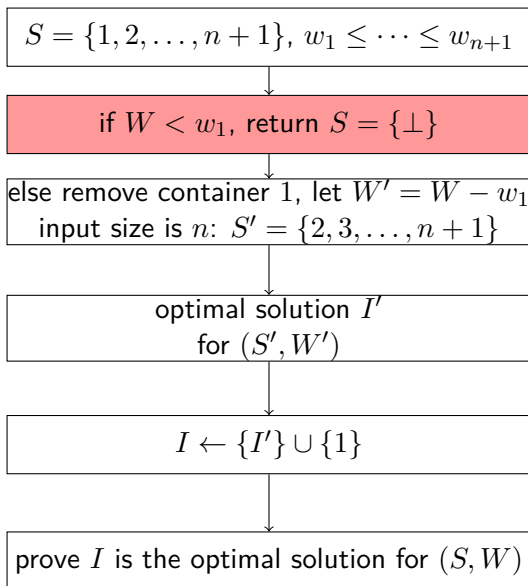
## Proof of Correctness (Induction on Input Size)

**Lemma.**  $\forall$  input size  $n$ , the algorithm yields the correct solution.

Let  $S = \{1, 2, \dots, n\}$  be the set of containers that has been sorted in ascending order, and  $w_1 \leq w_2 \leq \dots \leq w_n$ .

- **Induction basis.** Prove when the input size  $n = 1$  (there is only one container), the greedy algorithm will yield the correct solution. Obviously hold.
- **Induction steps.** Prove if the greedy algorithm yield optimal solution for input size  $n$ , it will also yield optimal solution for input size  $n + 1$ .

## Analysis of Greedy Algorithm: Interpretation



## Correctness Proof (1/2)

**Premise of induction:** greedy strategy will yield optimal solution for input size  $n$ , consider input size  $n + 1$

$$S = \{1, 2, \dots, n + 1\}, w_1 \leq w_2 \leq \dots \leq w_{n+1}$$

Premise of induction  $\Rightarrow$  for input size  $n$

$$S' = \{2, \dots, n + 1\}, W' = W - w_1$$

Greedy strategy yields optimal solution  $I'$  for  $(S', W')$ .

Let  $I = I' \cup \{1\}$ .



## Correctness Proof (2/2)

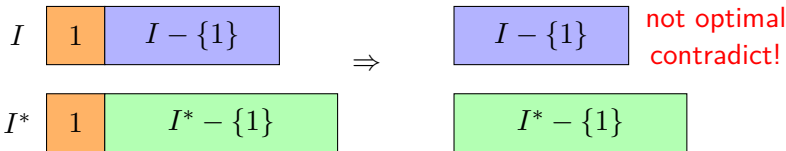
**Claim.**  $I$  is the optimal solution for  $(S, W)$ .

**Proof by contradiction.** If not, suppose there exists an optimal solution  $I^*$  for  $(S, W)$  and  $|I^*| > |I|$ .

- Assume w.l.o.g.  $1 \in I^*$ , since otherwise we can replace 1 with the first container in  $I^*$ , also yield the optimal solution.
- $I^* - \{1\}$  forms a solution for  $(S', W')$  and

$$|I^* - \{1\}| > |I - \{1\}| = |I'|$$

The existence of  $I^*$  contradicts to the premise that  $I'$  is the optimal solution for  $(S', W')$ .



## Summary

0-1 knapsack is an  $\mathcal{NP}$ -hard problem

- optimal loading is a variant of 0-1 knapsack problem, and can be solved using greedy algorithm efficiently

Correctness proof. Induction on input size

# Outline

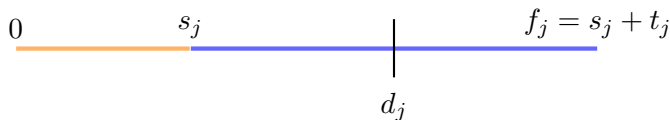
- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness**
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

# Scheduling to Minimizing Lateness

## Minimizing lateness problem (最小延迟调度)

- A job set  $A$ , single resource processes one job at a time, all jobs come in at time 0
- Job  $j$  requires  $t_j$  units of processing time and is due at time  $d_j$  (ddl). Clearly,  $t_j \leq d_j$ .
- If job  $j$  starts at time  $s_j$ , it finishes at time  $f_j = s_j + t_j$ .
- **Scheduling:**  $S : A \rightarrow \mathbb{N}$ ,  $S(j) = s_j$  is the start time of job  $j$ .
- **Lateness:** Lateness function computes the lateness of job:

$$L(j) = \ell_j = \max\{0, f_j - d_j\} = \max\{0, s_j + t_j - d_j\}$$



Goal. Schedule all jobs to minimize **max** lateness

$$\min\{\max_{j \in A} \ell_j\} = \min\{\max_{j \in A} \{\max\{0, s_j + t_j - d_j\}\}\}$$



Constraint. No overlap

$$\forall i, j \in A, i \neq j$$

$$s_i + t_i \leq s_j \vee s_j + t_j \leq s_i$$

## Example 1

<i>A</i>	1	2	3	4	5
<i>S</i>	0	5	13	17	27
<i>T</i>	5	8	4	10	3
<i>D</i>	10	12	15	11	20
<i>L</i>	0	1	2	16	10

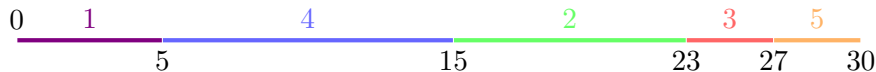
Table: Sequential scheduling



## Example 2

<i>A</i>	1	4	2	3	5
<i>S</i>	0	5	15	23	27
<i>T</i>	5	10	8	4	3
<i>D</i>	10	11	12	15	20
<i>L</i>	0	4	11	12	10

Table: Earliest-deadline first



## Minimizing Lateness: Greedy Algorithms

**Greedy template.** Schedule jobs according to some natural order.

- [Shortest processing time first] Schedule jobs in ascending order of processing time  $t_j$ .

$A$	1	2
$T$	1	10
$D$	100	10

- $\ell_1 = 0, \ell_2 = 11 - 10 = 1$

- $\ell_2 = 0, \ell_1 = 0$  (better)

- [Smallest slack] Schedule jobs in ascending order of slack  $d_j - t_j$ .

$A$	1	2
$T$	1	10
$D$	2	10

- $\ell_2 = 10 - 10 = 0, \ell_1 = 11 - 2 = 9$

- $\ell_1 = 0, \ell_2 = 10 + 1 - 10 = 1$  (better)



## Minimizing Lateness: Earliest Deadline First

---

**Algorithm 2:** Schedule( $A, T, D$ )

---

```
1: sort  $n$  jobs in  $A$  so that  $d_1 \leq d_2 \leq \dots \leq d_n$ ;  
2:  $t \leftarrow 0$  //from time 0;  
3: for  $j = 1$  to  $n$  do  
4:   assign job  $j$  to interval  $[t, t + t_j]$ ;  
5:    $s_j \leftarrow t$ ;  
6:    $f_j \leftarrow t + t_j$ ;  
7:    $t \leftarrow t + t_j$   
8: end  
9: return intervals  $[s_1, f_1], \dots, [s_n, f_n]$ 
```

---

### Main idea

- earliest deadline first
- assign jobs one after another, no idle time

## Correctness Proof: Exchange Argument

### Proof sketch

- Analyze the difference between **optimal solution** and **algorithm solution** (e.g. different order)
  - Design a transform operation (e.g. swap), thus we can gradually convert an optimal solution to algorithm solution in finite steps.
  - The transformation does not affect optimality of solution, since every step preserving optimality.
- 

In this case, two properties of greedy algorithm solution:

- No idle time: every time there is a job being processed
- No inversion. We say  $(i, j)$  forms an inversion if  $d_i > d_j$  but  $s_i < s_j$

## Key Lemma about Algorithm Solution

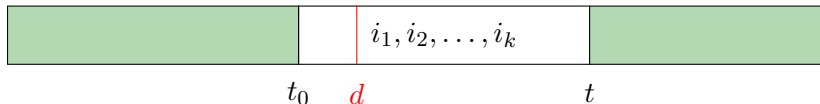
**Lemma.** All schedulings with no inversion and idle time have the same minimal **max** lateness time.

**Proof.** No inversion  $\Rightarrow$  tasks are sorted in ascending order of  $d_i$ .

It is possible that several jobs has the same deadline. Jobs  $i_1, i_2, \dots, i_k$  with the same deadline  $d$  are assigned arbitrarily.  
(green parts are identical)

- The start time is  $t_0$ , the finish time for one of these jobs is  $t$ , among this jobs, the max lateness is  $\max\{0, t - d\} \Leftarrow$  irrelevant to the order of  $i_1, i_2, \dots, i_k$ .

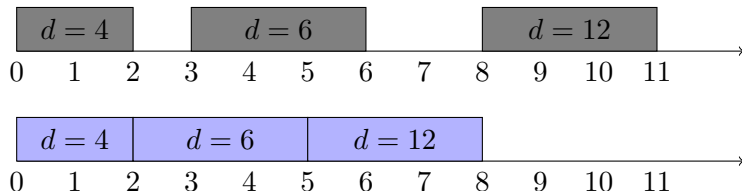
$$t = t_0 + (t_{i_1} + t_{i_2} + \dots + t_{i_k})$$



**Corollary.** All possible algorithm solutions have the same minimal **max** lateness time.

## Examine the Optimal Solution

**Observation.** There always exists an optimal schedule with **no idle time**.

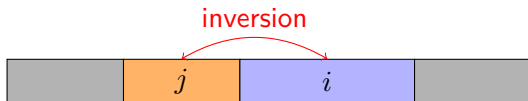


Algorithm solution: the earliest-deadline-first schedule has no idle time.

- We have eliminated one difference between optimal solution and algorithm solution.
- There is another one: inversion

## Minimizing Lateness: Inversions

**Inversion.** Given a schedule  $S$ , an **inversion** is a pair of jobs  $i$  and  $j$  such that  $d_i < d_j$  but  $j$  scheduled before  $i$ , i.e.,  $s_j < s_i$ .

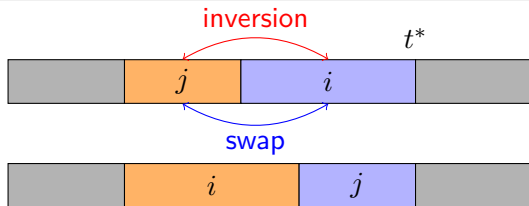


**Figure:** As before, jobs are numbered so that  $d_1 \leq d_2 \leq \dots \leq d_n$

**Fact.** If a schedule (with no idle time) has an inversion, it has at least one pair of inverted jobs scheduled consecutively. (according to definition)

## Minimizing Max Lateness: Inversions

**Claim.** Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.



**Proof.** Let  $\ell$  be the lateness before the swap,  $\ell'$  be it afterwards.

- $i \leftrightarrow j$  does not affect the latest time of other jobs:  $\ell'_k = \ell_k$  for all  $k \neq i, j$
- $\ell'_i \leq \ell_i$  (because job  $i$  has been moved forwards)
- $\ell'_j = \max\{0, t^* - d_j\}$  (definition),  $i$  and  $j$  are inverted  $\Rightarrow d_i < d_j$ , thus  $\ell'_j \leq \max\{0, t^* - d_i\} = \ell_i$   
 $\Rightarrow \max\{\ell_i, \ell_j\} \geq \max\{\ell'_i, \ell'_j\}$

## Putting All the Above Together

**Theorem.** The earliest-deadline-first schedule  $S$  is optimal.

**Proof.** Define  $S^*$  to be an optimal schedule. Let's see what happens.

- Can always assume  $S^*$  has no idle time.
- If  $S^*$  has no inversions, then key lemma  $S \sim S^*$ , stop here.
- If  $S^*$  has an inversion, let  $i \leftrightarrow j$  be an adjacent inversion. Swapping  $i$  and  $j$ :
  - does not increase the max lateness
  - strictly decreases the number of inversions
- Continue the above process until there is no inversion, we can also conclude that  $S \sim S^*$ .

Max number of inversion is  $n(n-1)/2$  (completely inverted), thus the transformation will stop in finite steps.

## Summary of Greedy Analysis Trick

**Analysis.** Find the difference between optimal solution and algorithm solution.

**Exchange argument.** Gradually transform an optimal solution to the one found by the greedy algorithm.

- at most require finite steps (seems unnecessary)
- each step of transformation does not hurt its quality



# Outline

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem**
- 6 Greedy Algorithm Does Not Work (not teach in class)

## Fractional Knapsack Problem

**Input.** Given  $n$  items with weight vector  $(w_1, \dots, w_n)$  and value vector  $(v_1, \dots, v_n)$ , and weight limit  $W > 0$ .

**Goal.** Find  $x = (p_1, \dots, p_n) \in [0, 1]^n$  (choose some fractions of  $n$  items) to satisfy:

- **Optimized goal:** maximizes  $\sum_{i=1}^n p_i v_i$
- **Constraint:**  $\sum_{i=1}^n p_i w_i \leq W$

The difference is that now the items are infinitely divisible.

# Greedy Algorithm

Greedy strategy. greatest value-per-weight ratio first

## Algorithm

- Sort  $n$  items according to the decending order of value-per-weight ratio  $\alpha_i = v_i/w_i$ .
- iteratively picks the item with the greatest value-per-weight ratio
- if, at some step, the knapsack cannot fit the entire last item with current greatest value-per-weight ratio items, we will take a fraction of it to fill the knapsack.

## Correctness Proof (1/3)

**Lemma.**  $\forall$  input size  $n$ , the algorithm yields the optimal solution.

**Proof idea.** Mathematical reduction on input size.

**Induction basis.** When  $n = 1$ , the greedy algorithm is obviously the optimal solution.

**Induction step.** Suppose the algorithm is optimal for  $n = k$ , then it is also optimal for  $n = k + 1$ .

- Let  $p_1$  be the algorithm's output for the first item,  $I' = (p_2, \dots, p_{k+1})$  be the output on instance  $(w_2, \dots, w_{k+1})$ ,  $(v_2, \dots, v_{k+1})$ , and  $W - p_1 w_1$ .
- According to the induction premise,  $I'$  is the optimal solution of the above sub-instance of size  $n = k$ . Let  $I = p_1 \cup I'$ .

**Claim.** Then, we claim  $I$  is the optimal solution for  $n = k + 1$ .

## Correctness Proof (2/3)

**Proof by contradiction.** If not, suppose there exists a more optimal solution  $I^*$  with maximal value  $V^*$ .

Prove the first element  $p_1^*$  of  $I^*$  must be equal to  $p_1$  of  $I$ .

- ①  $p_1^* = p_1$ : we have nothing to prove.
- ②  $p_1^* > p_1$  is impossible, because the greedy strategy guarantees that  $p_1$  of  $I$  is as large as possible.
- ③ If  $p_1^* < p_1$ , we can always increase it to  $p_1$  by decreasing total weight of its remaining  $k$  items by  $\Delta = (p_1 - p_1^*)w_1$ . Note that such adjustment makes sense since the total weight of the remaining  $k$  items must be larger than  $\Delta$ . Otherwise, we must have  $V^* < V$ , which is not possible by premise. We then consider two sub-cases after adjustment:
  - The total value is unchanged. This is only possible when there exists at least one more item  $j$  such that  $\alpha_j = \alpha_1$ .
  - The total value is higher. However, this case will never occur since it goes against the assumed optimality of  $I^*$ .

## Correctness Proof (3/3)

We conclude that either  $p_1^* = p_1$  or we can adjust it to this case without compromising optimality.

$I^* - \{p_1\}$  forms a solution for  $W - p_1^*w_1 = W - p_1w_1$  with items  $(2, \dots, n+1)$  with total value  $V^* - \alpha_1p_1 > V - \alpha_1p_1$   
 $\leadsto$  contradicts the optimality of  $I'$

This proves  $I$  is the optimal solution for input size  $n = k + 1$ .

# Outline

- 1 Introduction of Greedy Algorithm
- 2 Interval Scheduling
- 3 Optimal Loading
- 4 Scheduling to Minimizing Lateness
- 5 Fractional Knapsack Problem
- 6 Greedy Algorithm Does Not Work (not teach in class)

# What if Greedy Algorithm Does not Work

## Input analysis

- Determine the range of input that greedy strategy works.

## Error analysis

- Greedy algorithm is the approximation algorithm of the problem: estimate the distance between greedy solution and optimal solution (the upper bound over all inputs)



## Coin Changing Problem

**Coin changing.** Given  $n$  currency denominations

- $v_1 = 1, v_2, \dots, v_n, v_1 < v_2 < \dots < v_n,$
- weight  $w_1, w_2, \dots, w_n.$

**Goal.** Devise a method to pay amount  $y$  using coins with lightest weight.

---

**Example.**  $v_1 = 1, v_2 = 5, v_3 = 14, v_4 = 18, w_i = 1, i \in [n],$   
 $y = 28.$  In this case, the problem is equivalent to **using fewest number of coins.**

**Solution.:**  $x_3 = 2, x_1 = x_2 = x_4 = 0,$  total weight is 2.

## Modeling

Let  $x_i$  be the number of coin  $i$ ,  $i \in [n]$

Goal function.

$$\min \left\{ \sum_{i=1}^n w_i x_i \right\}$$

Constraint.

$$\sum_{i=1}^n v_i x_i = y, x_i \in \mathbb{N}, i \in [n]$$

Next, we consider a special case:  $w_i = 1$  for all  $i \in [n]$ .

## Dynamic Programming

$F_k(y)$ : the lightest weight using first  $k$  types of coins to pay amount  $y$

The iteration equation

$$\begin{cases} F_k(y) = \min_{0 \leq x_k \leq \lfloor \frac{y}{v_k} \rfloor} \{F_{k-1}(y - v_k x_k) + 1 \cdot x_k\} \\ F_1(y) = \frac{y}{v_1} = y \end{cases}$$

- Dynamic programming requires the domination of the first coin is 1 to ensure the constraint can always be met.
- Dynamic programming always give the optimal solution.

## Greedy Algorithm

**Strategy.** Smallest  $w_i/v_i$  coin first. Since all  $w_i = 1$ , this means largest denomination coin first and  $v_1 = 1$ .

$$\frac{1}{v_1} > \frac{1}{v_2} > \dots > \boxed{\frac{1}{v_n}}$$

$G_k(y)$ : greedy solution of using first  $k$  types coins to pay  $y$

$$\begin{cases} G_k(y) = \left\lfloor \frac{y}{v_k} \right\rfloor + G_{k-1}(y \bmod v_k), k > 1 \\ G_1(y) = \frac{y}{v_1} = y \end{cases}$$

**Thinking.** Why we require all  $w_i = 1$ ? Otherwise, we cannot guarantee  $v_1 = 1$  appears at first place in line with greedy algorithm's input order. Looking ahead, we will use dynamic programming as a reference.

## $n = 1, 2$ : Greedy Strategy Yield Optimal Solution

$n = 1$ : only one type of coin and we must have  $v_1 = 1$ .

- In this case,  $F_1(y) = G_1(y) = w_1 y$

$n = 2$ : for dynamic programming algorithm, the larger is  $x_2$ , the better is the solution

$$F_2(y) = \min_{0 \leq x_2 \leq \lfloor y/v_2 \rfloor} \{F_1(y - v_2 x_2) + x_2\}$$

**Goal:** prove  $F_2(y) = G_2(y)$

**Technique:** decide the monotonicity of function  $F_1(y - v_2 x_2) + x_2$  about  $x_2$

$$\begin{aligned} & [F_1(y - v_2(x_2 + \delta)) + (x_2 + \delta)] - [F_1(y - v_2 x_2) + x_2] \\ &= [(y - v_2 x_2 - v_2 \delta) + x_2 + \delta] - [(y - v_2 x_2) + x_2] \\ &= -v_2 \delta + \delta = \delta(1 - v_2) < 0 \end{aligned}$$

This proves the greedy that choice is optimal for  $n = 2$ .

**Theorem.** Let  $n_0$  be an integer. Suppose  $\forall k \leq n_0, G_k(y) = F_k(y)$  for all  $y \in \mathbb{N}$ . Let  $(p, \delta)$  be the tuple such that  $v_{k+1} = pv_k - \delta$ , where  $0 \leq \delta < v_k$ ,  $v_k < v_{k+1}$ ,  $p \in \mathbb{Z}^+$ .

The following propositions are equivalent:

- ❶  $G_{k+1}(y) = F_{k+1}(y)$  for all  $y \in \mathbb{Z}^+$ ;
- ❷  $G_{k+1}(pv_k) = F_{k+1}(pv_k)$  (can be used to give counterexample)
- ❸  $1 + G_k(\delta) \leq p$  (can be used to decide if the first statement holds)

The uniqueness of  $(p, \delta)$ :

- Since  $v_{k+1} > v_k$ ,  $v_{k+1}$  can be uniquely expressed as  $p'v_k + \eta$ , where  $0 \leq \eta < v_k$ .
- $p'v_k + \eta = (p' + 1)v_k - (v_k - \eta)$ . Set  $p' + 1 = p$ ,  $v_k - \eta = \delta$ . The uniqueness of  $(p', \eta)$  implies the uniqueness of  $(p, \delta)$ .

## Some Remarks

By the equivalence of (1) and (3), we can decide if greedy algorithm gives the optimal solution for  $k \geq 3$ .

---

Verifying the truth of statement (3) requiring  $O(k)$  complexity.

---

Statement (2) is a special case of proposition (1) when  $y = pv_k$ .

Statement (1) is true  $\Rightarrow$  Statement (2) is true

Statement (2) is false  $\Rightarrow$  Statement (1) is false

The amount  $y = pv_k$  provide a counterexample for the correctness of greedy algorithm.

**Demo:**  $n = 3$

$$v_{k+1} = pv_k - \delta, 0 \leq \delta < v_k, p \in \mathbb{Z}^+$$

$$\text{proposition (3) : } 1 + G_k(\delta) \leq p$$

**Example.**  $v_1 = 1, v_2 = 5, v_3 = 14, v_4 = 18$ .

$$\forall y: G_1(y) = F_1(y), G_2(y) = F_2(y)$$

Decide if  $G_3(y) = F_3(y)$

To utilize proposition (3), we first compute tuple  $(p, \delta)$ :

$$v_3 = pv_2 - \delta \Rightarrow p = 3, \delta = 1$$

$$1 + G_2(\delta) = 1 + 1 = 2 \leq 3 = p$$

**Conclusion:** proposition (3) is true thus greedy algorithm still works for  $n = 3$ .



## Demo: $n = 4$

**Example.**  $v_1 = 1, v_2 = 5, v_3 = 14, v_4 = 18$ .

$\forall y$  we have:  $G_1(y) = F_1(y), G_2(y) = F_2(y), G_3(y) = F_3(y)$

Decide if  $G_4(y) = F_4(y)$

To utilize proposition (3), we first compute tuple  $(p, \delta)$ :

$$v_4 = pv_3 - \delta \Rightarrow p = 2, \delta = 10$$

$$1 + G_3(\delta) = 1 + 2 > p = 2$$

**Conclusion:** proposition (3) is false thus greedy algorithm does not work for  $n = 3$ .

**Counterexample** is give by proposition (2), i.e.  $n = 4$ ,

$$y = pv_3 = 28$$

Optimal solution  $x_3 = 2$  vs. Greedy solution  $(x_4 = 1, x_2 = 2)$