

算法设计与分析(2026年春季学期)

# Greedy Algorithms

授课老师: 栗师  
南京大学计算机学院

**Def.** In an **optimization problem**, our goal of is to find a valid solution with the minimum cost (or maximum value).

## Trivial Algorithm for an Optimization Problem

Enumerate all valid solutions, compare them and output the best one.

- However, trivial algorithm often runs in **exponential** time, as the number of potential solutions is often exponentially large.
- $f(n)$  is a **polynomial** if  $f(n) = O(n^k)$  for some **constant**  $k > 0$ .
- convention: polynomial time = **efficient**

## Goals of algorithm design

- ① Design efficient algorithms to solve problems
- ② Design more efficient algorithms to solve problems

# Common Paradigms for Algorithm Design

- Greedy Algorithms
- Divide and Conquer
- Dynamic Programming
- Greedy algorithms are often for optimization problems.
- They often run in polynomial time due to their simplicity.

## Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (**key**)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (**usually easy**)

**Def.** A strategy is safe: there is always an optimum solution that agrees with the decision made according to the strategy.

# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

## Box Packing

**Input:**  $n$  boxes of capacities  $c_1, c_2, \dots, c_n$

$m$  items of sizes  $s_1, s_2, \dots, s_m$

Can put **at most 1** item in a box

Item  $j$  can be put into box  $i$  if  $s_j \leq c_i$

**Output:** A way to put as many items as possible in the boxes.

### Example:

- Box capacities: 60, 40, 25, 15, 12
- Item sizes: 45, 42, 20, 19, 16
- Can put 3 items in boxes: 45 → 60, 20 → 40, 19 → 25

## Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

## Designing a Reasonable Strategy for Box Packing

- Q: Take box 1. Which item should we put in box 1?
- A: The item of the largest size that can be put into the box.

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem

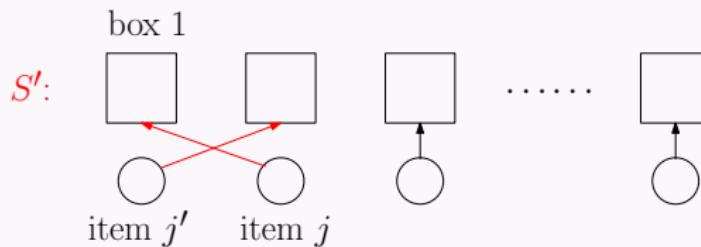
**Lemma** The strategy that put into box 1 the largest item it can hold is “safe”: There is an optimum solution in which box 1 contains the largest item it can hold.

- Intuition: putting the item gives us the easiest residual problem.
- formal proof via **exchanging argument**:

**Lemma** There is an optimum solution in which box 1 contains the largest item it can hold.

## Proof.

- Let  $j = \text{largest item that box 1 can hold.}$
- Take any optimum solution  $S$ . If  $j$  is put into Box 1 in  $S$ , done.
- Otherwise, assume this is what happens in  $S$ :



- $s_{j'} \leq s_j$ , and swapping gives another solution  $S'$
- $S'$  is also an optimum solution. In  $S'$ ,  $j$  is put into Box 1. □

- Notice that the exchanging operation is only for the sake of analysis; it is not a part of the algorithm.

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe”
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem
- Trivial: we decided to put Item  $j$  into Box 1, and the remaining instance is obtained by removing Item  $j$  and Box 1.

## Generic Greedy Algorithm

- 1: **while** the instance is non-trivial **do**
- 2:     make the choice using the greedy strategy
- 3:     reduce the instance

## Greedy Algorithm for Box Packing

- 1:  $T \leftarrow \{1, 2, 3, \dots, m\}$
- 2: **for**  $i \leftarrow 1$  to  $n$  **do**
- 3:     **if** some item in  $T$  can be put into box  $i$  **then**
- 4:          $j \leftarrow$  the largest item in  $T$  that can be put into box  $i$
- 5:         print("put item  $j$  in box  $i$ ")
- 6:          $T \leftarrow T \setminus \{j\}$

## Exchange argument: Proof of Safety of a Strategy

- let  $S$  be an arbitrary optimum solution.
- if  $S$  is consistent with the greedy choice, done.
- otherwise, show that it can be modified to another optimum solution  $S'$  that is consistent with the choice.

---

- The procedure is not a part of the algorithm.

# Outline

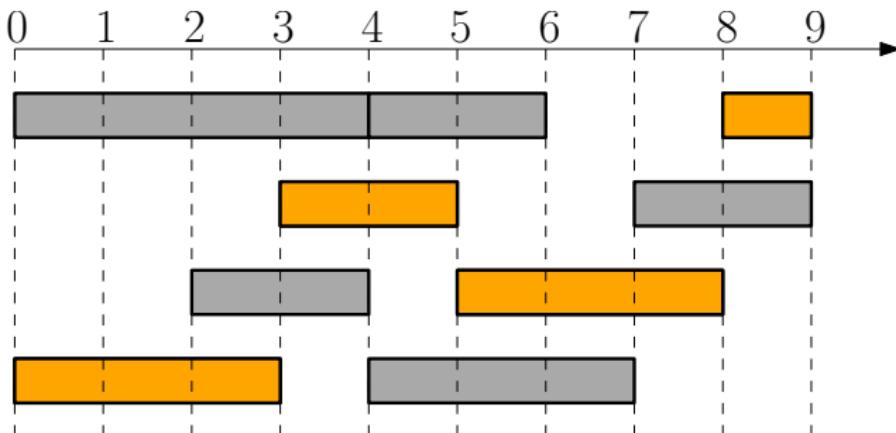
- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

## Interval Scheduling

**Input:**  $n$  jobs, job  $i$  with start time  $s_i$  and finish time  $f_i$

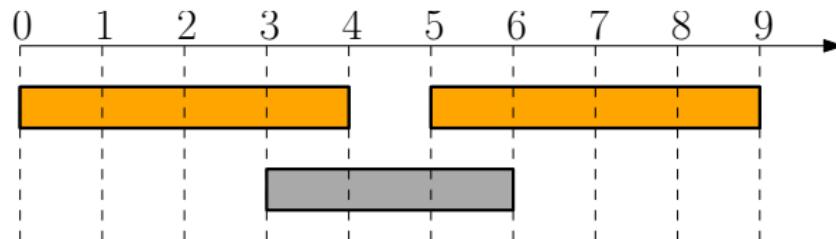
$i$  and  $j$  are **compatible** if  $[s_i, f_i)$  and  $[s_j, f_j)$  are disjoint

**Output:** A maximum-size subset of mutually compatible jobs



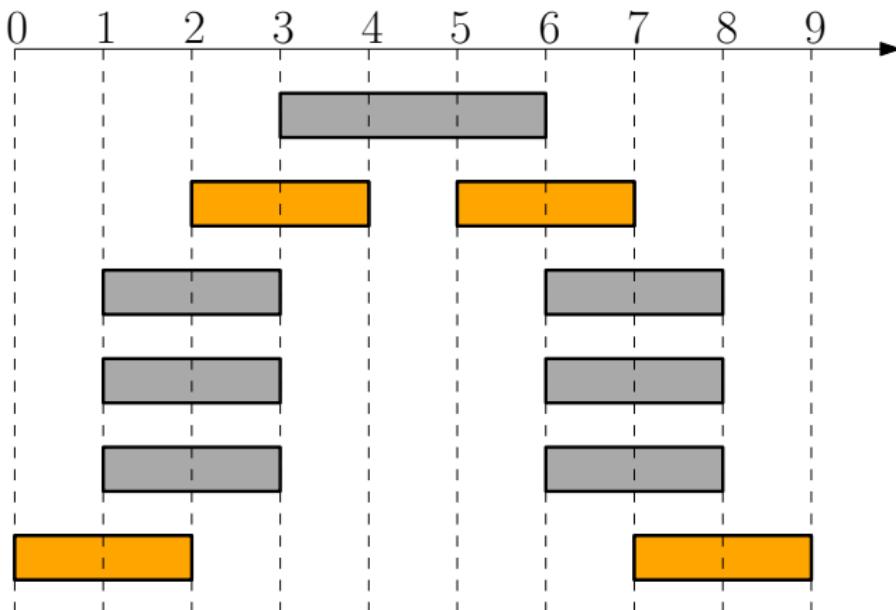
# Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? **No!**



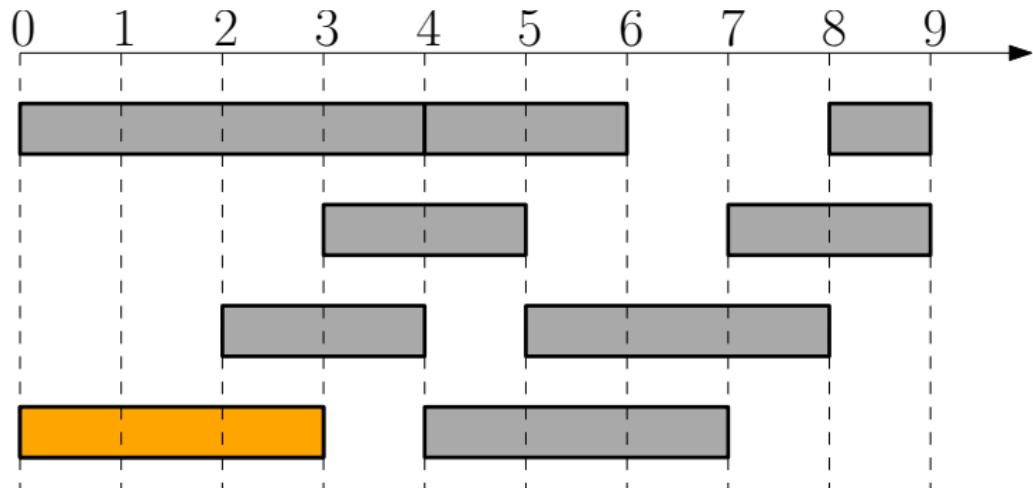
# Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs?  
**No!**



# Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
- Schedule the job conflicting with smallest number of other jobs? No!
- Schedule the job with the earliest finish time? Yes!



# Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job  $j$  with the earliest finish time: There is an optimum solution where the job  $j$  with the earliest finish time is scheduled.

## Proof.

- Take an arbitrary optimum solution  $S$
- If it contains  $j$ , done
- Otherwise, replace the first job in  $S$  with  $j$  to obtain another optimum schedule  $S'$ .

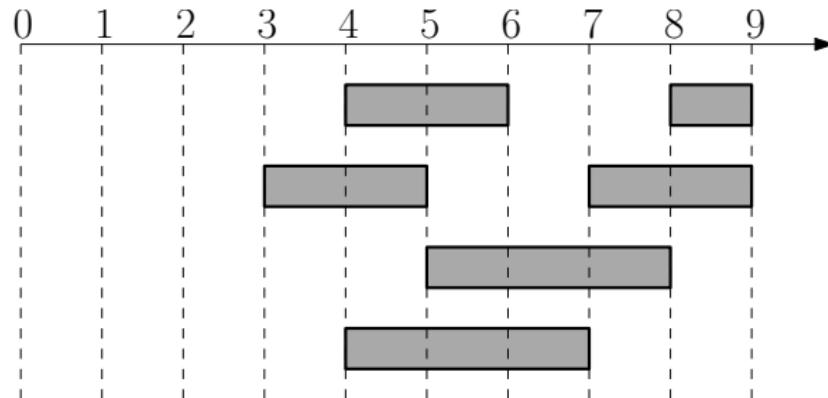
□



# Greedy Algorithm for Interval Scheduling

**Lemma** It is safe to schedule the job  $j$  with the earliest finish time: There is an optimum solution where the job  $j$  with the earliest finish time is scheduled.

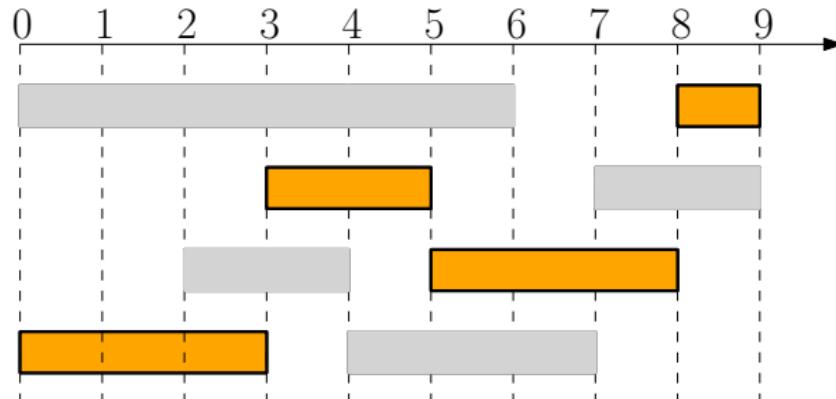
- What is the remaining task after we decided to schedule  $j$ ?
- Is it another instance of interval scheduling problem? Yes!



# Greedy Algorithm for Interval Scheduling

**Schedule( $s, f, n$ )**

```
1:  $A \leftarrow \{1, 2, \dots, n\}$ ,  $S \leftarrow \emptyset$ 
2: while  $A \neq \emptyset$  do
3:    $j \leftarrow \arg \min_{j' \in A} f_{j'}$ 
4:    $S \leftarrow S \cup \{j\}$ ;  $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$ 
5: return  $S$ 
```



# Greedy Algorithm for Interval Scheduling

## Schedule( $s, f, n$ )

```
1:  $A \leftarrow \{1, 2, \dots, n\}$ ,  $S \leftarrow \emptyset$ 
2: while  $A \neq \emptyset$  do
3:    $j \leftarrow \arg \min_{j' \in A} f_{j'}$ 
4:    $S \leftarrow S \cup \{j\}$ ;  $A \leftarrow \{j' \in A : s_{j'} \geq f_j\}$ 
5: return  $S$ 
```

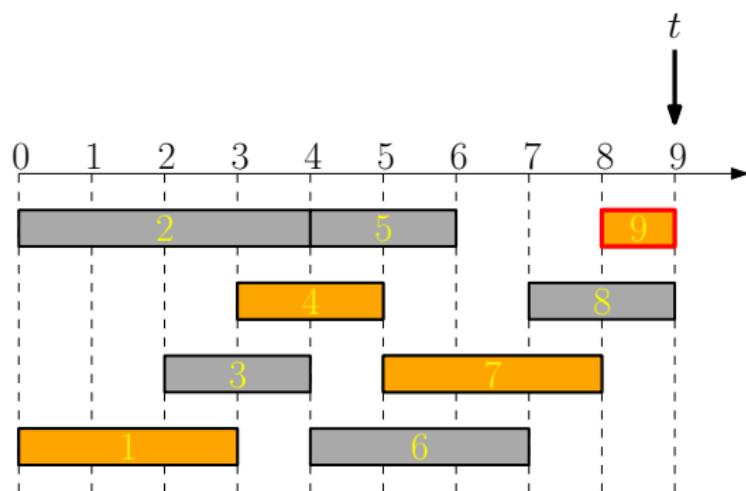
Running time of algorithm?

- Naive implementation:  $O(n^2)$  time
- Clever implementation:  $O(n \lg n)$  time

# Clever Implementation of Greedy Algorithm

## $\text{Schedule}(s, f, n)$

```
1: sort jobs according to  $f$  values
2:  $t \leftarrow 0, S \leftarrow \emptyset$ 
3: for every  $j \in [n]$  according to non-decreasing order of  $f_j$  do
4:   if  $s_j \geq t$  then
5:      $S \leftarrow S \cup \{j\}$ 
6:      $t \leftarrow f_j$ 
7: return  $S$ 
```



# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

## Scheduling to minimize lateness

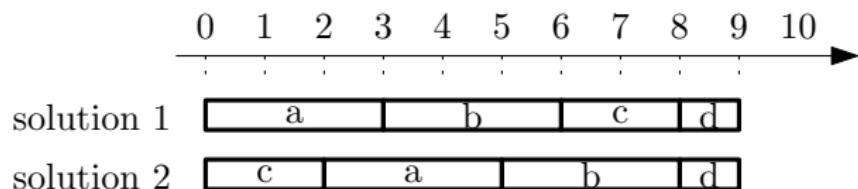
**Input:**  $n$  jobs, each job  $j \in [n]$  with a processing time  $p_j$  and deadline  $d_j$

**Output:** schedule jobs on 1 machine, to minimize the max. lateness

$C_j$ : completion time of  $j$     lateness  $l_j := \max\{C_j - d_j, 0\}$

- Example input:

$j$	a	b	c	d
$p_j$	3	3	2	1
$d_j$	5	7	4	8



- solution 1: max lateness =  $\max\{0, 3 - 5, 6 - 7, 8 - 4, 9 - 8\} = 4$
- solution 2: max lateness =  $\max\{0, 2 - 4, 5 - 5, 8 - 7, 9 - 8\} = 1$
- solution 2 is better

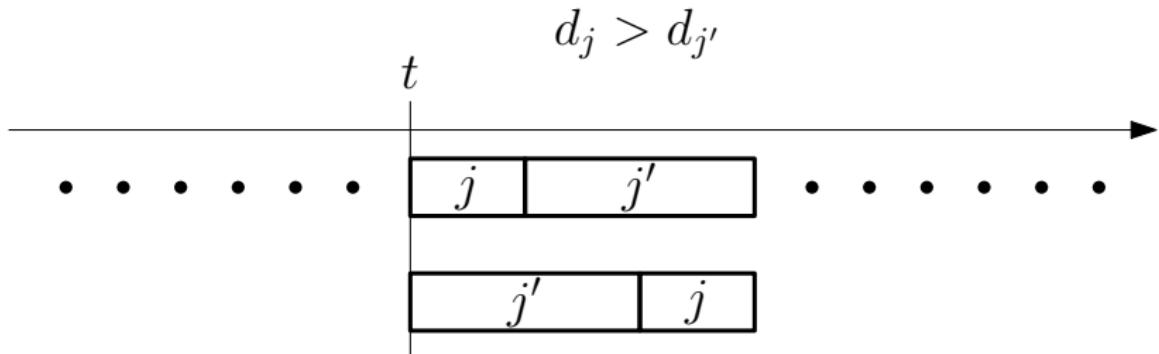
## Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

- A Ascending order of processing times  $p_j$
- B Ascending order of slackness  $d_j - p_j$
- C Ascending order of deadline  $d_j$ .

**Lemma** The ascending order of deadlines  $d_j$  (the Earliest Deadline First order or the EDF order) is the optimum schedule.

- maximum lateness =  $\max \left\{ 0, \max_{j \in [n]} \{C_j - d_j\} \right\}.$



- before:  $\max\{t + p_j - d_j, t + p_j + p_{j'} - d_{j'}\} = t + p_j + p_{j'} - d_{j'}$
- after:  $\max\{t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j\}$
- $p_{j'} - d_{j'} < p_j + p_{j'} - d_{j'}$  and  $p_j + p_{j'} - d_j < p_j + p_{j'} - d_{j'}$
- $\max\{t + p_{j'} - d_{j'}, t + p_j + p_{j'} - d_j\} < t + p_j + p_{j'} - d_{j'}$
- after swapping, the maximum of the two terms strictly decreases

## Repeated Swapping (for Analysis Only)

- 1: let  $S$  be any schedule (i.e, a permutation of  $[n]$ )
- 2: **while** there are two adjacent jobs  $j$  and  $j'$  in  $S$ , with  $j$  before  $j'$  and  $d_j > d_{j'}$  **do**
- 3:     swap  $j$  and  $j'$  in  $S$

**Q:** Does the algorithm terminate?

**A:** Yes. Number of inversions go down!

- $(j, j')$  is an inversion in  $S$  if  $j$  appears before  $j'$  and  $d_j > d_{j'}$ .
- So the algorithm converges to an EDF order.

**Q:** What if there are multiple EDF orders, i.e., some jobs have the same deadline?

**A:** All EDF orders have the same maximum lateness.

# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

## Scheduling to Minimize Weighted Completion Time

**Input:** A set of  $n$  jobs  $[n] := \{1, 2, 3, \dots, n\}$

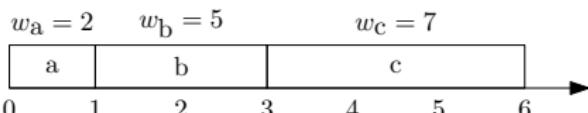
each job  $j$  has a **weight**  $w_j$  and **processing time**  $p_j$

**Output:** an ordering of jobs so as to minimize the **total weighted completion time** of jobs

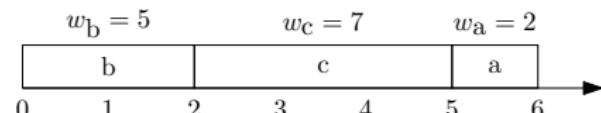
$$\begin{array}{c} p_a = 1 \\ \boxed{a} \\ w_1 = 2 \end{array}$$

$$\begin{array}{c} p_b = 2 \\ \boxed{b} \\ w_2 = 5 \end{array}$$

$$\begin{array}{c} p_c = 3 \\ \boxed{c} \\ w_3 = 7 \end{array}$$



$$\text{cost} = 2 \times 1 + 5 \times 3 + 7 \times 6 = 59$$



$$\text{cost} = 5 \times 2 + 7 \times 5 + 2 \times 6 = 57$$

## Candidate algorithms

Schedule the jobs in some natural order. Which order should we choose?

- A Ascending order of processing times  $p_j$
- B Descending order of slackness  $w_j$
- C Ascending order of  $p_j - w_j$
- D Ascending order of  $p_j/w_j$

**Def.** The Smith ratio of a job is  $w_j/p_j$ .

**Lemma** The descending order of Smith ratios (the Smith rule) is optimum.

- A schedule  $S$ ,  $j$  is right before  $j'$ .

**Q:** How does the total weighted completion time change if we swap  $j$  and  $j'$ ?  $(\dots, j, j', \dots) \Rightarrow (\dots, j', j, \dots)$

**A:**  $w_{j'}p_j \implies w_jp_{j'}$

- Therefore, swapping decrease the weighted completion time if  $\frac{p_{j'}}{w_{j'}} < \frac{p_j}{w_j}$ .
- Using the same argument as for the maximum lateness problem: ascending order of  $p_j/w_j$  is optimum.
- Indeed, optimum weighted completion time is

$$\sum_{j \in [n]} w_j p_j + \sum_{1 \leq j < j' \leq n} \min\{w_j p_{j'}, w_{j'} p_j\}.$$

# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

# Offline Caching

- Cache that can store  $k$  pages
- Sequence of page requests
- Cache miss happens if requested page not in cache. We need bring the page into cache, and evict some existing page if necessary.
- Cache hit happens if requested page already in cache.
- Goal: minimize the number of cache misses.

page sequence	cache
1	
5	✗ 1
4	✗ 1 5
2	✗ 1 5 4
5	✗ 1 2 4
3	✗ 1 2 5
2	✗ 1 2 3
1	✓ 1 2 3
1	✓ 1 2 3

misses = 6

## A Better Solution for Example

page sequence	cache				cache			
	1	2	3	4	1	2	3	4
1	✗	1			✗	1		
5	✗	1	5		✗	1	5	
4	✗	1	5	4	✗	1	5	4
2	✗	1	2	4	✗	1	5	2
5	✗	1	2	5	✓	1	5	2
3	✗	1	2	3	✗	1	3	2
2	✓	1	2	3	✓	1	3	2
1	✓	1	2	3	✓	1	3	2

## Offline Caching Problem

**Input:**  $k$  : the size of cache

$n$  : number of pages

We use  $[n]$  for  $\{1, 2, 3, \dots, n\}$ .

$\rho_1, \rho_2, \rho_3, \dots, \rho_T \in [n]$ : sequence of requests

**Output:**  $i_1, i_2, i_3, \dots, i_T \in \{\text{hit, empty}\} \cup [n]$ : indices of pages to evict ("hit" means evicting no page, "empty" means evicting empty page)

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

**Q:** Which one is more realistic?

**A:** Online caching

- Offline Caching: we know the whole sequence ahead of time.
- Online Caching: we have to make decisions on the fly, before seeing future requests.

**Q:** Which one is more realistic?

**A:** Online caching

**Q:** Why do we study the offline caching problem?

**A:** Use the offline solution as a benchmark to measure the “competitive ratio” of online algorithms

# Offline Caching: Potential Greedy Algorithms

- FIFO(First-In-First-Out): always evict the first page in cache
- LRU(Least-Recently-Used): Evict page whose most recent access was earliest
- LFU(Least-Frequently-Used): Evict page that was least frequently requested
- All the above algorithms are not optimum!
- Indeed all the algorithms are “online”, i.e, the decisions can be made without knowing future requests. Online algorithms can not be optimum.

# FIFO is not optimum

	FIFO			Furthest-in-Future		
requests						
1	<b>X</b>	1				
2	<b>X</b>	1	2			
3	<b>X</b>	1	2	3		
4	<b>X</b>	4	2	3	<b>X</b>	1
1	<b>X</b>	4	1	3	<b>✓</b>	1

misses = 5      misses = 4

## Furthest-in-Future (FF)

- Algorithm: every time, evict the page that is not requested until furthest in the future, if we need to evict one.
- The algorithm is **not** an online algorithm, since the decision at a step depends on the request sequence in the future.

## Furthest-in-Future (FF)

	FIFO			Furthest-in-Future		
requests						
1						
2	X	1				
3	X	1	2			
4	X	1	2	3		
	4	2	3			
1	X	4	1	3		
misses = 5			misses = 4			

# Example

requests



✗ ✗ ✗ ✗ ✓ ✗ ✓ ✓ ✓ ✗ ✗ ✓

□	1	1	1	2	2	2	2	2	1	5	5
---	---	---	---	---	---	---	---	---	---	---	---

□	□	5	5	5	5	3	3	3	3	3	3
---	---	---	---	---	---	---	---	---	---	---	---

□	□	□	4	4	4	4	4	4	4	4	4
---	---	---	---	---	---	---	---	---	---	---	---

# Recall: Designing and Analyzing Greedy Algorithms

## Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

## Offline Caching Problem

**Input:**  $k$  : the size of cache

$n$  : number of pages

$\rho_1, \rho_2, \rho_3, \dots, \rho_T \in [n]$ : sequence of requests

$p_1, p_2, \dots, p_k \in \{\text{empty}\} \cup [n]$ : initial set of pages in cache

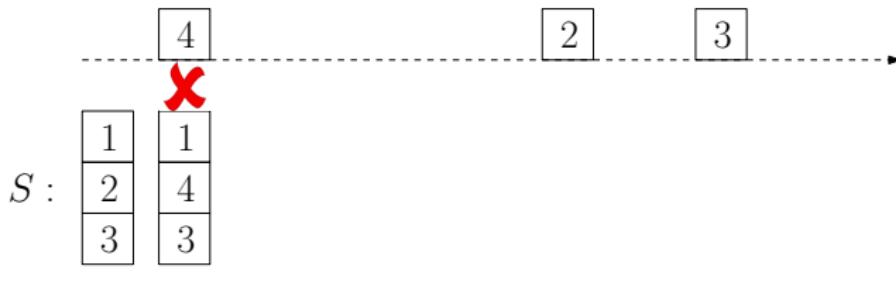
**Output:**  $i_1, i_2, i_3, \dots, i_t \in \{\text{hit, empty}\} \cup [n]$

- empty stands for an empty page
- “hit” means evicting no pages

## A Common Way to Analyze Greedy Algorithms

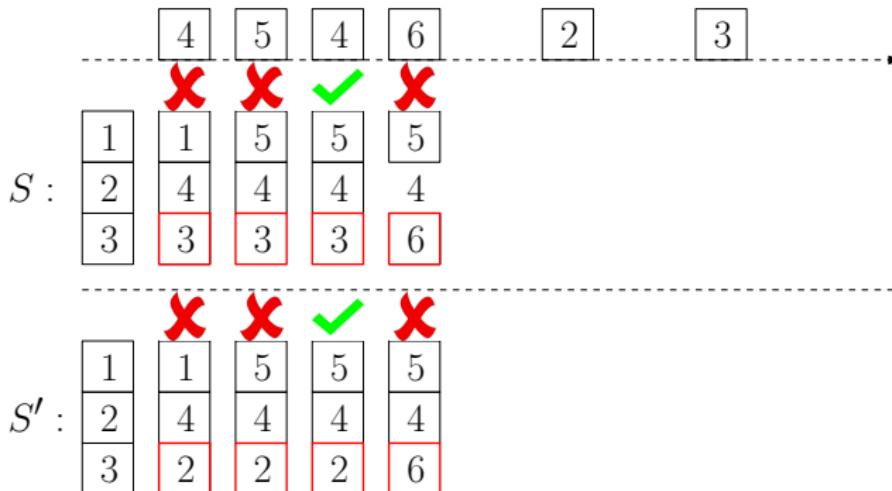
- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Lemma** Assume at time 1 a page fault happens and there are no empty pages in the cache. Let  $p^*$  be the page in cache that is not requested until furthest in the future. **There is an optimum solution in which  $p^*$  is evicted at time 1.**



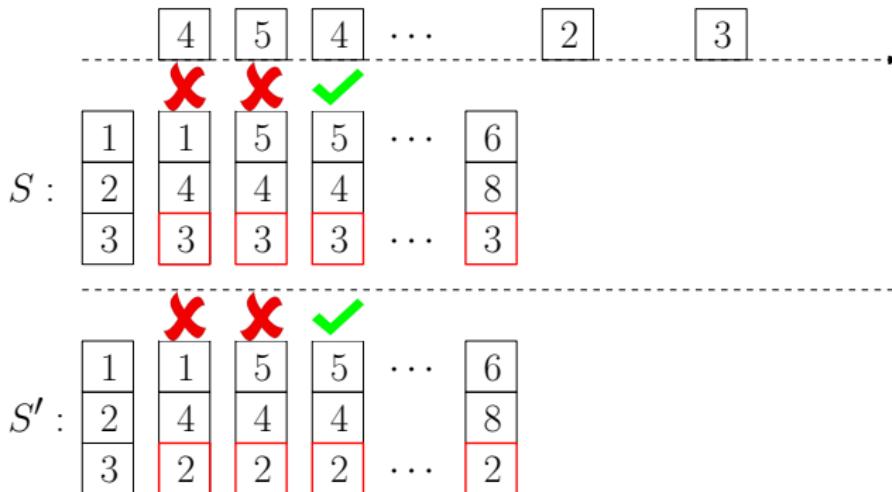
## Proof.

- ①  $S$ : any optimum solution
- ②  $p^*$ : page in cache not requested until furthest in the future.
  - In the example,  $p^* = 3$ .
- ③ Assume  $S$  evicts some  $p' \neq p^*$  at time 1; otherwise done.
  - In the example,  $p' = 2$ .



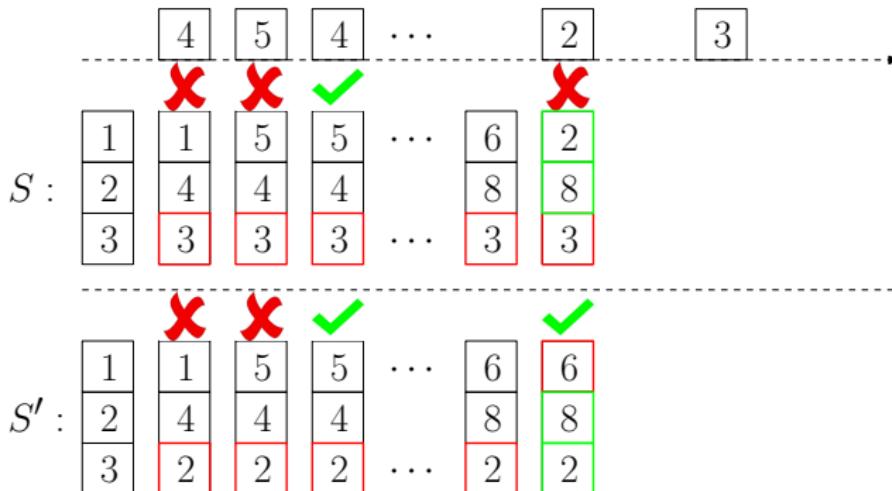
## Proof.

- ④ Create  $S'$ .  $S'$  evicts  $p^*(=3)$  instead of  $p' (=2)$  at time 1.
- ⑤ After time 1, cache status of  $S$  and that of  $S'$  differ by only 1 page.  $S'$  contains  $p' (=2)$  and  $S$  contains  $p^*(=3)$ .
- ⑥ From now on,  $S'$  will “copy”  $S$ .



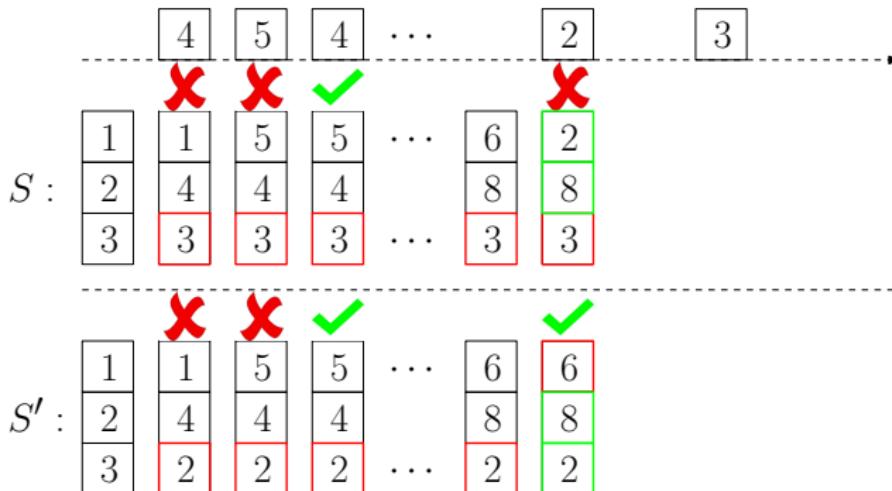
## Proof.

- ⑦ If  $S$  evicted the page  $p^*$ ,  $S'$  will evict the page  $p'$ . Then, the cache status of  $S$  and that of  $S'$  will be the same.  $S$  and  $S'$  will be exactly the same from now on.
- ⑧ Assume  $S$  did not evict  $p^*(=3)$  before we see  $p'(=2)$ .



## Proof.

- ⑨ If  $S$  evicts  $p^*(=3)$  for  $p' (=2)$ , then  $S$  won't be optimum. Assume otherwise.
- ⑩ So far,  $S'$  has 1 less page-miss than  $S$  does.
- ⑪ The status of  $S'$  and that of  $S$  only differ by 1 page.



## Proof.

- ⑫ We can then guarantee that  $S'$  make at most the same number of page-misses as  $S$  does.
  - Idea: if  $S$  has a page-hit and  $S'$  has a page-miss, we use the opportunity to make the status of  $S'$  the same as that of  $S$ . □

- Thus, we have shown how to create another solution  $S'$  with the same number of page-misses as that of the optimum solution  $S$ . Thus, we proved

**Lemma** Assume at time 1 a page fault happens and there are no empty pages in the cache. Let  $p^*$  be the page in cache that is not requested until furthest in the future. **It is safe to evict  $p^*$  at time 1.**

**Theorem** The furthest-in-future strategy is optimum.

```
1: for  $t \leftarrow 1$  to  $T$  do
2:   if  $\rho_t$  is in cache then do nothing
3:   else if there is an empty page in cache then
4:     evict the empty page and load  $\rho_t$  in cache
5:   else
6:      $p^* \leftarrow$  page in cache that is not used furthest in the future
7:     evict  $p^*$  and load  $\rho_t$  in cache
```

**Q:** How can we make the algorithm as fast as possible?

**A:**

- The running time can be made to be  $O(n + T \log k)$ .
- For each page  $p$ , use a linked list (or an array with dynamic size) to store the time steps in which  $p$  is requested.
  - We can find the next time a page is requested easily.
- Use a priority queue data structure to hold all the pages in cache, so that we can easily find the page that is requested furthest in the future.

time	0	1	2	3	4	5	6	7	8	9	10	11	12
pages		P1	P5	P4	P2	P5	P3	P2	P4	P3	P1	P5	P3
	✗	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	✗	✓

P1: 

1	10	
---	----	--

P2: 

4	7	
---	---	--

P3: 

6	9	12	
---	---	----	--

P4: 

3	8	
---	---	--

P5: 

2	5	11	
---	---	----	--

priority queue

pages	priority values
P5	$\infty$
P3	$\infty$
P4	$\infty$

```

1: for every  $p \leftarrow 1$  to  $n$  do
2:    $times[p] \leftarrow$  array of times in which  $p$  is requested, in
   increasing order                                 $\triangleright$  put  $\infty$  at the end of array
3:    $pointer[p] \leftarrow 1$ 
4:    $Q \leftarrow$  empty priority queue
5: for every  $t \leftarrow 1$  to  $T$  do
6:    $pointer[\rho_t] \leftarrow pointer[\rho_t] + 1$ 
7:   if  $\rho_t \in Q$  then
8:      $Q.\text{increase-key}(\rho_t, times[\rho_t, pointer[\rho_t]])$ , print "hit",
   continue
9:   if  $Q.\text{size}() < k$  then
10:     print "load  $\rho_t$  to an empty page"
11:   else
12:      $p \leftarrow Q.\text{extract-max}()$ , print "evict  $p$  and load  $\rho_t$ "
13:      $Q.\text{insert}(\rho_t, times[\rho_t, pointer[\rho_t]])$        $\triangleright$  add  $\rho_t$  to  $Q$  with key
        value  $times[\rho_t, pointer[\rho_t]]$ 

```

# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

# Encoding Letters Using Bits

- 8 letters  $a, b, c, d, e, f, g, h$  in a language
- need to encode a message using bits
- idea: use 3 bits per letter

$a$	$b$	$c$	$d$	$e$	$f$	$g$	$h$
000	001	010	011	100	101	110	111

$deacfg \rightarrow 011100000010101110$

**Q:** Can we have a better encoding scheme?

- Seems unlikely: must use 3 bits per letter

**Q:** What if some letters appear more frequently than the others?

**Q:** If some letters appear more frequently than the others, can we have a better encoding scheme?

**A:** Using **variable-length encoding scheme** might be more efficient.

### Idea

- using fewer bits for letters that are more frequently used, and more bits for letters that are less frequently used.

**Q:** What is the issue with the following encoding scheme?

- a: 0      b: 1      c: 00

**A:** Can not guarantee a unique decoding. For example, 00 can be decoded to *aa* or *c*.

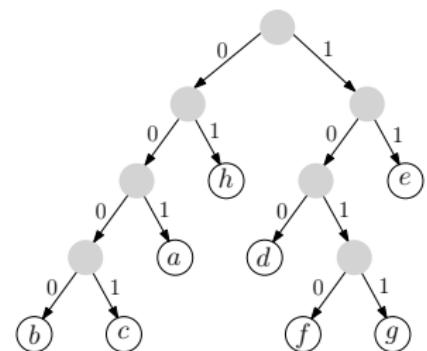
### Solution

Use **prefix codes** to guarantee a unique decoding.

# Prefix Codes

**Def.** A prefix code for a set  $S$  of letters is a function  $\gamma : S \rightarrow \{0, 1\}^*$  such that for two distinct  $x, y \in S$ ,  $\gamma(x)$  is not a prefix of  $\gamma(y)$ .

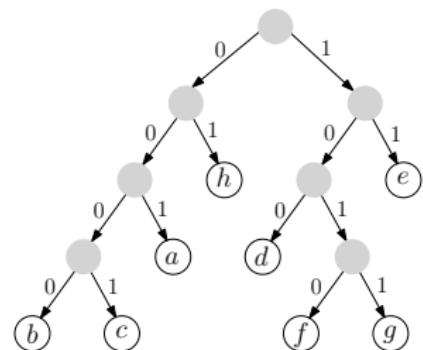
$a$	$b$	$c$	$d$
001	0000	0001	100
$e$	$f$	$g$	$h$
11	1010	1011	01



# Prefix Codes Guarantee Unique Decoding

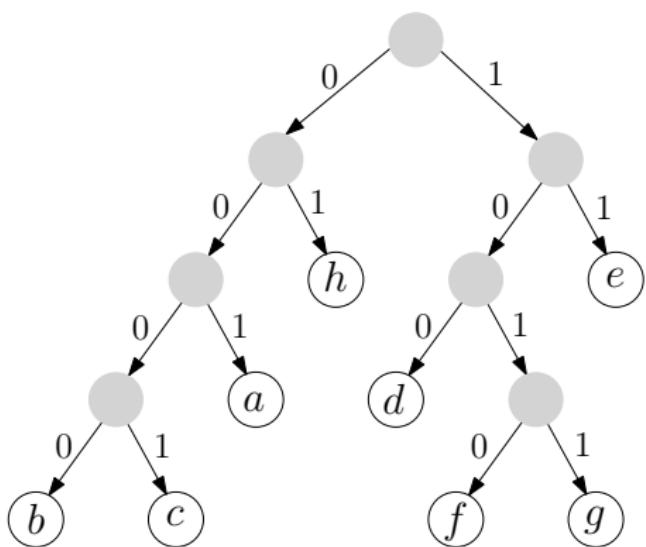
- Reason: there is only one way to cut the first code.

$a$	$b$	$c$	$d$
001	0000	0001	100
$e$	$f$	$g$	$h$
11	1010	1011	01



- $0001/001/100/0000/01/01/11/1010/0001/001/$
- $cadbhhefc$

## Properties of Encoding Tree



- Rooted binary tree
- Left edges labelled 0 and right edges labelled 1
- A leaf corresponds to a code for some letter
- If coding scheme is not wasteful: a non-leaf has exactly two children

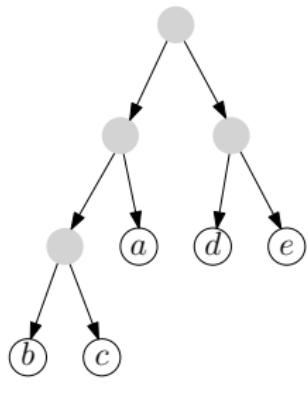
## Best Prefix Codes

**Input:** frequencies of letters in a message

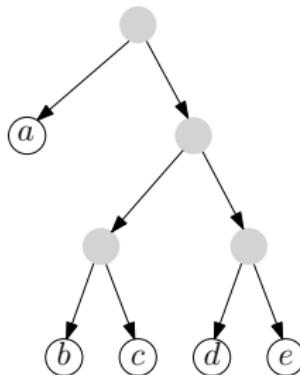
**Output:** prefix coding scheme with the shortest encoding for the message

## example

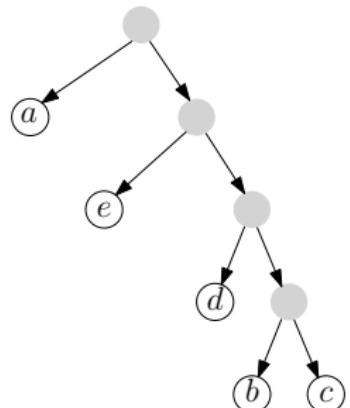
letters	a	b	c	d	e	
freqencies	18	3	4	6	10	
scheme 1 length	2	3	3	2	2	total = 89
scheme 2 length	1	3	3	3	3	total = 87
scheme 3 length	1	4	4	3	2	total = 84



scheme 1



scheme 2



scheme 3

- Example Input:  $(a: 18, b: 3, c: 4, d: 6, e: 10)$

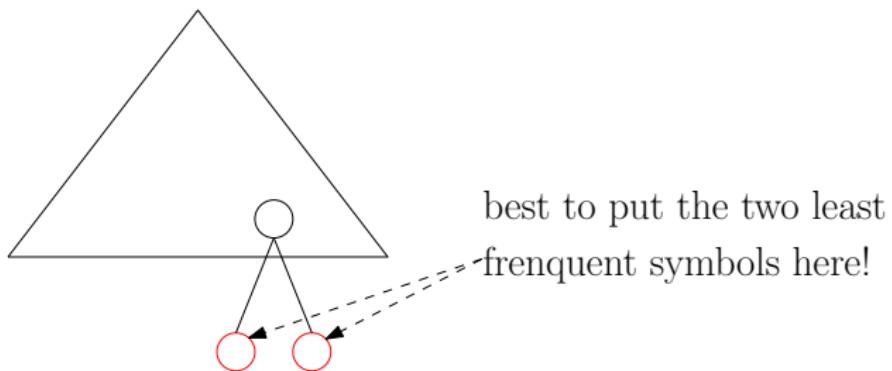
**Q:** What types of decisions should we make?

- Can we directly give a code for some letter?
- Hard to design a strategy; residual problem is complicated.
- Can we partition the letters into left and right sub-trees?
- Not clear how to design the greedy algorithm

**A:** We can choose two letters and make them brothers in the tree.

# Which Two Letters Can Be Safely Put Together As Brothers?

- Focus on the “structure” of the optimum encoding tree
- There are two deepest leaves that are brothers



**Lemma** It is safe to make the two least frequent letters brothers.

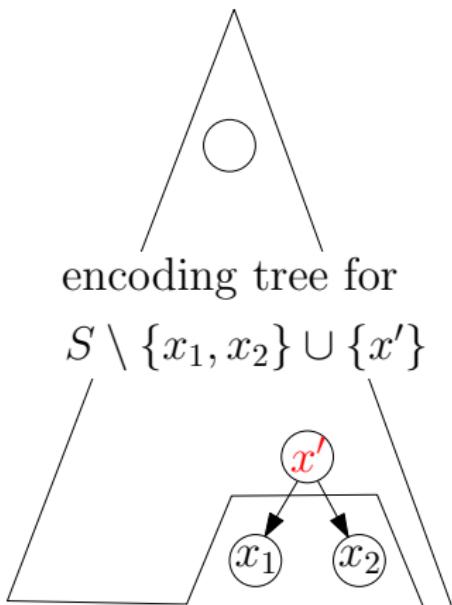
**Lemma** There is an optimum encoding tree, where the two least frequent letters are brothers.

- So we can irrevocably decide to make the two least frequent letters brothers.

**Q:** Is the residual problem another instance of the best prefix codes problem?

**A:** Yes, though it is not immediate to see why.

- $f_x$ : the frequency of the letter  $x$  in the support.
- $x_1$  and  $x_2$ : the two letters we decided to put together.
- $d_x$  the depth of letter  $x$  in our output encoding tree.



Def:  $f_{x'} = f_{x_1} + f_{x_2}$

$$\begin{aligned}
 & \sum_{x \in S} f_x d_x \\
 &= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2} \\
 &= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1} \\
 &= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x'} (d_{x'} + 1) \\
 &= \sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x + f_{x'}
 \end{aligned}$$

In order to minimize

$$\sum_{x \in S} f_x d_x,$$

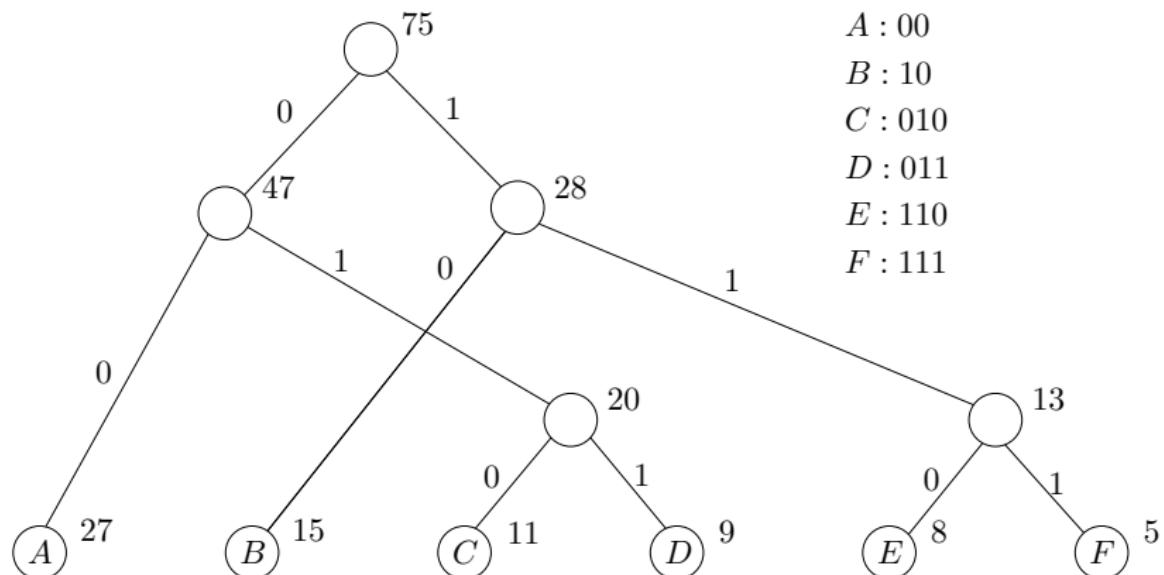
we need to minimize

$$\sum_{x \in S \setminus \{x_1, x_2\} \cup \{x'\}} f_x d_x,$$

subject to that  $d$  is the depth function for an encoding tree of  $S \setminus \{x_1, x_2\}$ .

- This is exactly the best prefix codes problem, with letters  $S \setminus \{x_1, x_2\} \cup \{x'\}$  and frequency vector  $f$ !

# Example



**Def.** The codes given the greedy algorithm is called the **Huffman codes**.

## Huffman( $S, f$ )

- 1: **while**  $|S| > 1$  **do**
- 2:   let  $x_1, x_2$  be the two letters with the smallest  $f$  values
- 3:   introduce a new letter  $x'$  and let  $f_{x'} = f_{x_1} + f_{x_2}$
- 4:   let  $x_1$  and  $x_2$  be the two children of  $x'$
- 5:    $S \leftarrow S \setminus \{x_1, x_2\} \cup \{x'\}$
- 6: **return** the tree constructed

# Algorithm using Priority Queue

## Huffman( $S, f$ )

```
1:  $Q \leftarrow \text{build-priority-queue}(S)$ 
2: while  $Q.\text{size} > 1$  do
3:    $x_1 \leftarrow Q.\text{extract-min}()$ 
4:    $x_2 \leftarrow Q.\text{extract-min}()$ 
5:   introduce a new letter  $x'$  and let  $f_{x'} = f_{x_1} + f_{x_2}$ 
6:   let  $x_1$  and  $x_2$  be the two children of  $x'$ 
7:    $Q.\text{insert}(x', f_{x'})$ 
8: return the tree constructed
```

# Outline

- 1 Toy Example: Box Packing
- 2 Interval Scheduling
- 3 Scheduling to Minimize Lateness
- 4 Weighted Completion Time Scheduling
- 5 Offline Caching
- 6 Data Compression and Huffman Code
- 7 Summary

# Summary for Greedy Algorithms

## Greedy Algorithm

- Build up the solutions in steps
- At each step, make an **irrevocable** decision using a “reasonable” strategy
- Interval scheduling problem: schedule the job  $j^*$  with the earliest deadline
- Offline Caching: evict the page that is used furthest in the future
- Huffman codes: make the two least frequent letters brothers

# Summary for Greedy Algorithms

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)

**Def.** A strategy is “safe” if there is always an optimum solution that “agrees with” the decision made according to the strategy.

# Proving a Strategy is Safe

- Take an arbitrary optimum solution  $S$
- If  $S$  agrees with the decision made according to the strategy, done
- So assume  $S$  does not agree with decision
- Change  $S$  slightly to another optimum solution  $S'$  that agrees with the decision
  - Interval scheduling problem: exchange  $j^*$  with the first job in an optimal solution
  - Offline caching: a complicated “copying” algorithm
  - Huffman codes: move the two least frequent letters to the deepest leaves.

# Summary for Greedy Algorithms

## A Common Way to Analyze Greedy Algorithms

- Prove that the reasonable strategy is “safe” (key)
- Show that the remaining task after applying the strategy is to solve a (many) smaller instance(s) of the same problem (usually easy)
- Interval scheduling problem: remove  $j^*$  and the jobs it conflicts with
- Offline caching: trivial
- Huffman codes: merge two letters into one
- Two problems that do not fall into the category: lateness, weighted completion time