CSE 431/531: Algorithm Analysis and Design (Spring 2022)
Greedy Algorithms

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo

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

Analysis of Greedy Algorithm

- 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

Box Packing

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

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

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: $45 \rightarrow 60, 20 \rightarrow 40, 19 \rightarrow 25$

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.

Analysis of Greedy Algorithm

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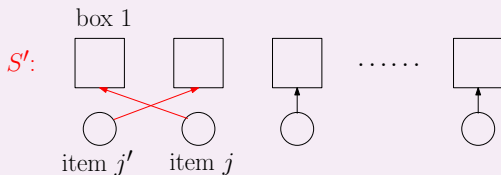
- $s_{j'} \leq s_j$, and swapping gives another solution S'
- S' is also an optimum solution. In S', j is put into Box 1. $\square$

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

Analysis of Greedy Algorithm

- 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: $T \leftarrow \{1, 2, 3, \cdots, m\}$
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:        $T \leftarrow T \setminus \{j\}$

Generic Greedy Algorithm

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

Lemma Generic algorithm is correct if and only if the greedy strategy is safe.

- Greedy strategy is safe: we will not miss the optimum solution
- Greedy stretegy is not safe: we will miss the optimum solution for some instance, since the choices we made are irrevocable.

Greedy Algorithm

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

Analysis of Greedy Algorithm

- 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

Def. A strategy is "safe" if there is always an optimum solution that is "consistent" with the decision made according to the strategy.

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

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
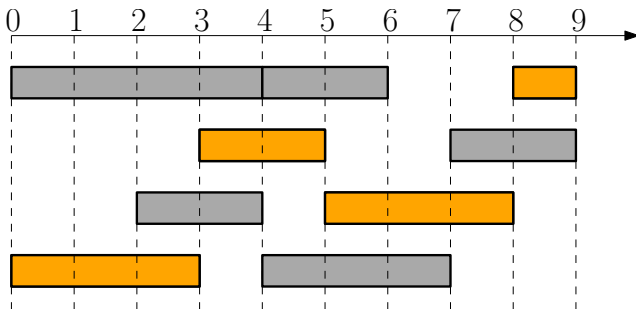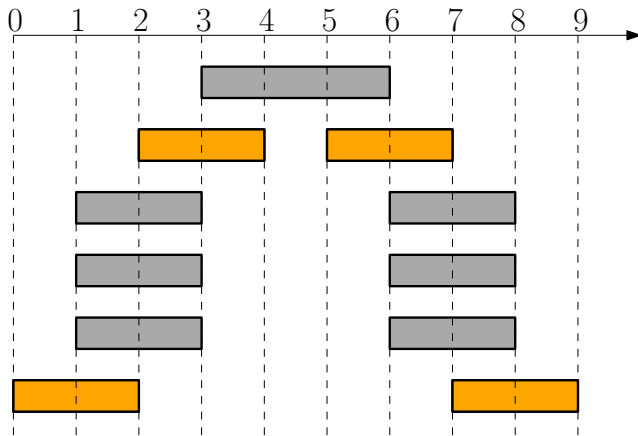
Greedy Algorithm for Interval Scheduling

- Which of the following strategies are safe?
- Schedule the job with the smallest size? No!
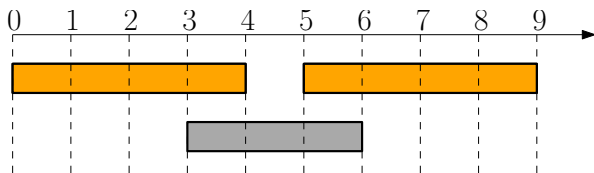
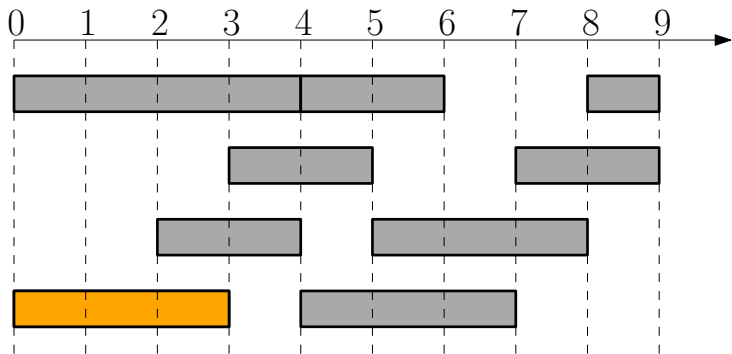# Greedy Algorithm for Interval Scheduling

- 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.
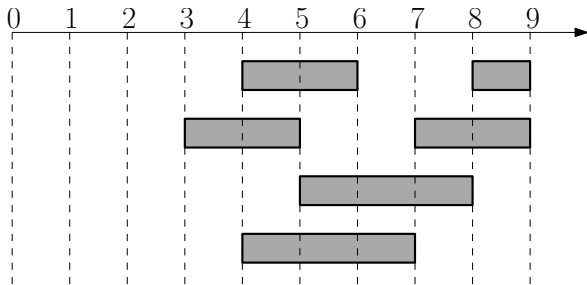
Proof.

- 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'. $\square$

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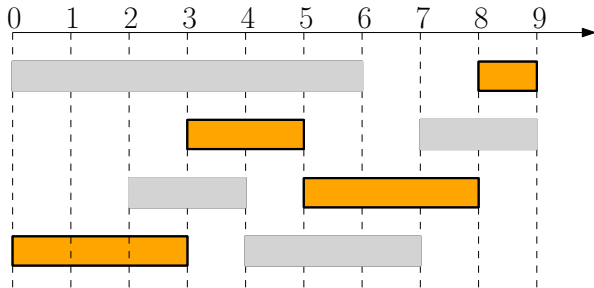
- What is the remaining task after we decided to schedule j?
- Is it another instance of interval scheduling problem? Yes!

Schedule(s, f, n)

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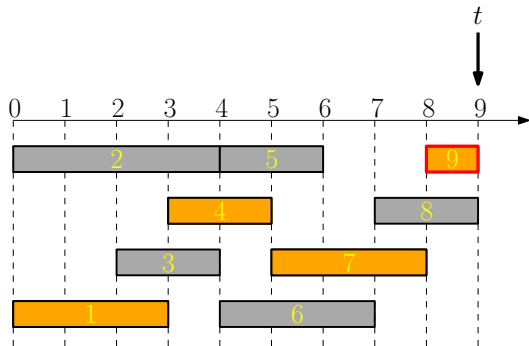5: **return** S

Running time of algorithm?

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

Clever Implementation of Greedy Algorithm

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:         $S \leftarrow S \cup \{j\}$
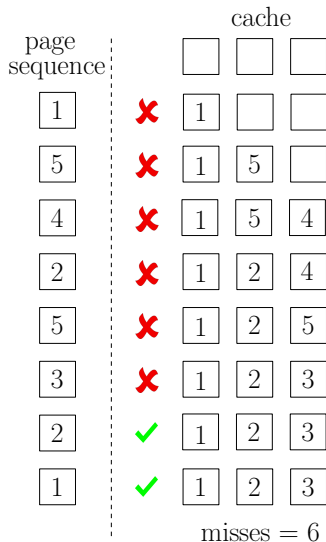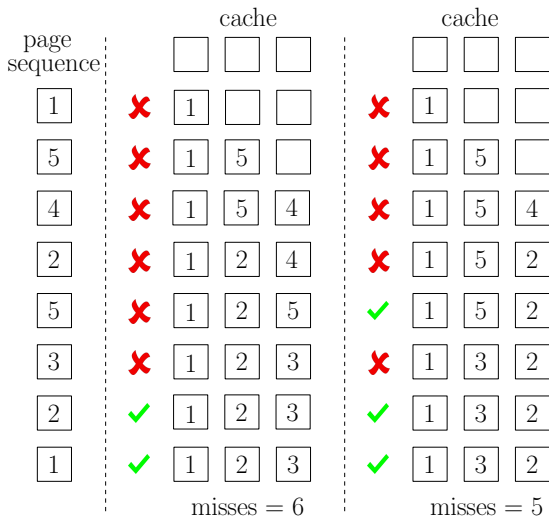6: $t \leftarrow f_j$
7: **return** S

Outline

- 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	✘	1		
5	✘	1	5	
4	✘	1	5	4
2	✘	1	2	4
5	✘	1	2	5
3	✘	1	2	3
2	✔	1	2	3
1	✔	1	2	3

misses $= 6$

Offline Caching Problem

Input: k : the size of cache

n : number of pages

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

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

Output: $i_1, i_2, i_3, \cdots, i_T \in \{\text{hit}, \text{empty}\} \cup [n]$: 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.
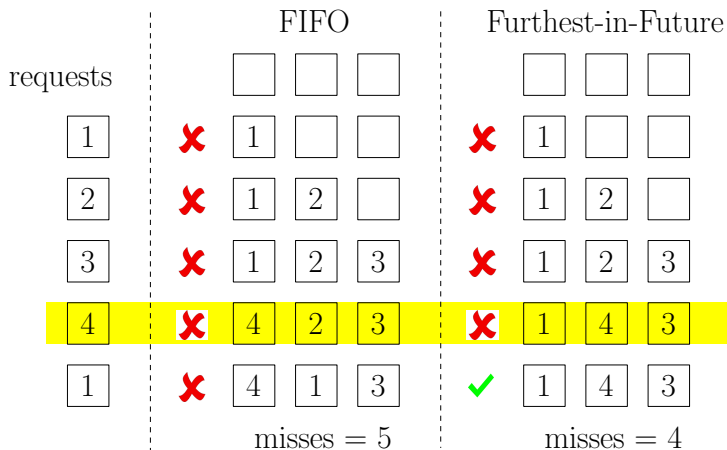
FIFO is not optimum

	FIFO				Furthest-in-Future		
requests		☐	☐	☐		☐	☐	☐
1	✘	1			✘	1		
2	✘	1	2		✘	1	2	
3	✘	1	2	3	✘	1	2	3
4	✘	4	2	3	✘	1	4	3
1	✘	4	1	3	✔	1	4	3
	misses = 5				misses = 4			

Optimum Offline Caching

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.

Example

requests

| 1 | 5 | 4 | 2 | 5 | 3 | 2 | 4 | 3 | 1 | 5 | 3 |

✘ ✘ ✘ ✘ ✔ ✘ ✔ ✔ ✔ ✘ ✘ ✔

	1	1	1	2	2	2	2	2	2	1	5	5
		5	5	5	5	3	3	3	3	3	3	3
			4	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

Analysis of Greedy Algorithm

- 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

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

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

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

- empty stands for an empty page
- "hit" means evicting no pages

Analysis of Greedy Algorithm

- 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.
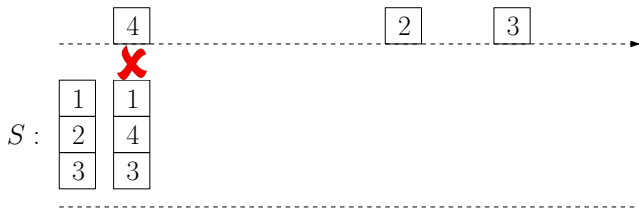
Proof.

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, $p' = 2$.
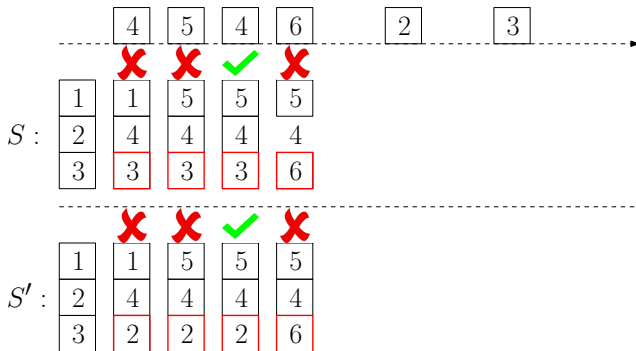
Proof.

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.
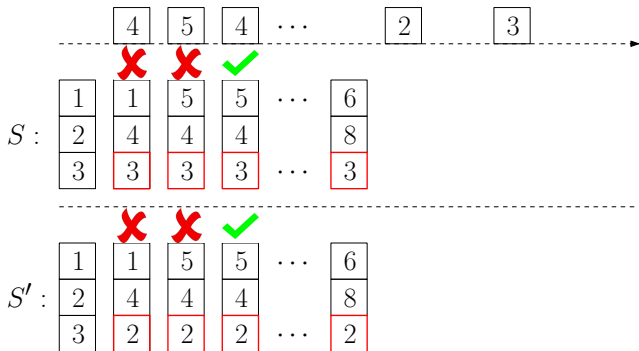
Proof.

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 $p'(=2)$.
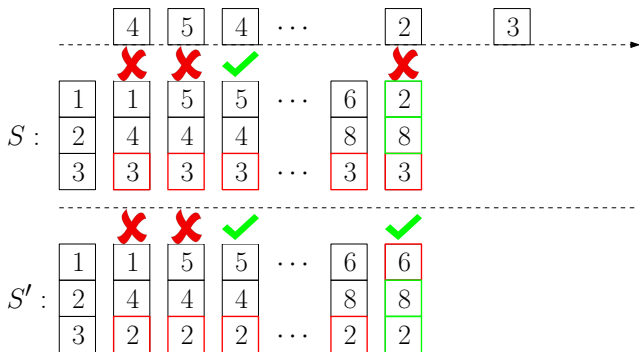
Proof.

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.
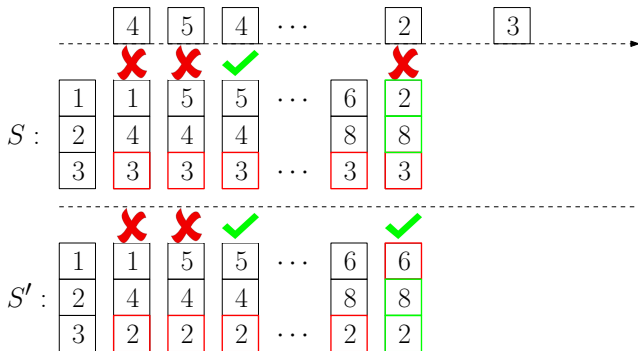
Proof.

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 ← 1 to T do
2:     if ρ_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 ρ_t in cache
5:     else
6:         p* ← page in cache that is not used furthest in the future
7:         evict p* and load ρ_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
		✗	✗	✗	✗	✓	✗	✓	✓	✓	✗	✗	✓
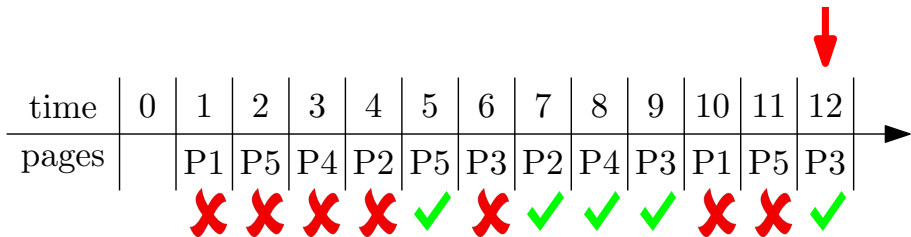
P1: | 1 | 10 | |

P2: | 4 | 7 | |

P3: | 6 | 9 | 12 | |

P4: | 3 | 8 | |

P5: | 2 | 5 | 11 | |

priority queue

pages	priority values
P5	∞
P3	∞
P4	∞

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

Outline

- Let V be a ground set of size n.

Def. A priority queue is an abstract data structure that maintains a set $U \subseteq V$ of elements, each with an associated key value, and supports the following operations:

- insert(v, key_value): insert an element $v \in V \setminus U$, with associated key value key_value.
- decrease_key(v, new_key_value): decrease the key value of an element $v \in U$ to new_key_value
- extract_min(): return and remove the element in U with the smallest key value
- $\cdots$

Simple Implementations for Priority Queue

- $n =$ size of ground set V

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$
heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

The elements in a heap is organized using a complete binary tree:
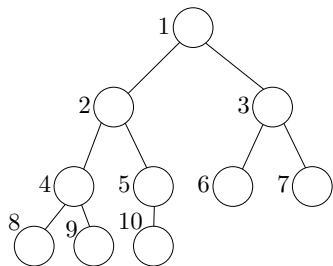
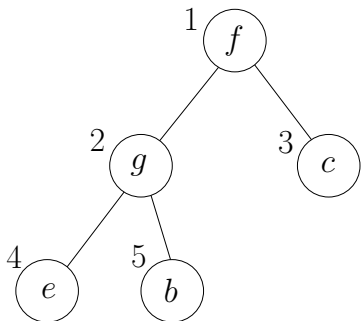- Nodes are indexed as $\{1, 2, 3, \cdots, s\}$
- Parent of node i: $\lfloor i/2 \rfloor$
- Left child of node i: $2i$
- Right child of node i: $2i + 1$

Heap

A heap H contains the following fields

- s: size of U (number of elements in the heap)
- $A[i], 1 \leq i \leq s$: the element at node i of the tree
- $p[v], v \in U$: the index of node containing v
- $key[v], v \in U$: the key value of element v

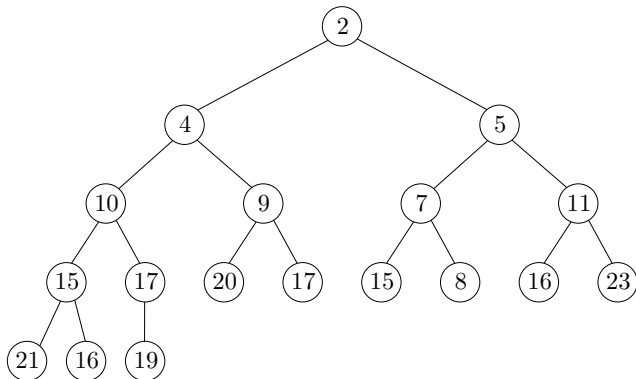- $s = 5$
- $A = (\text{'}f\text{'}, \text{'}g\text{'}, \text{'}c\text{'}, \text{'}e\text{'}, \text{'}b\text{'})$
- $p[\text{'}f\text{'}] = 1, p[\text{'}g\text{'}] = 2, p[\text{'}c\text{'}] = 3,$
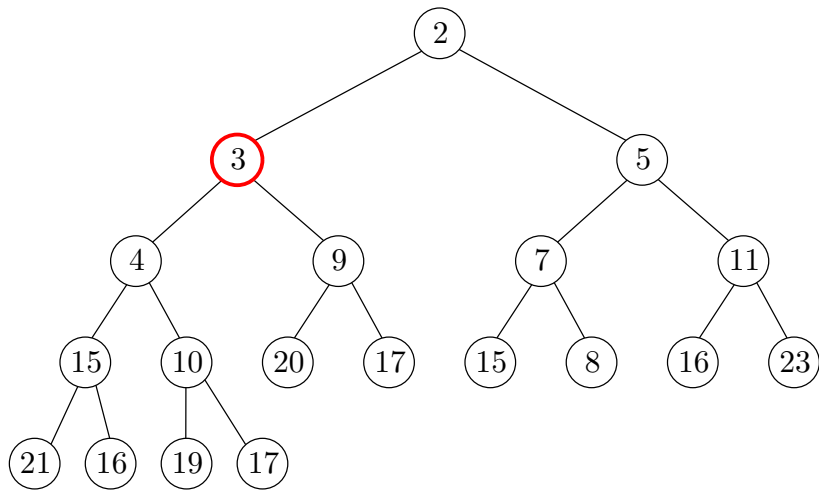 $p[\text{'}e\text{'}] = 4, p[\text{'}b\text{'}] = 5$

Heap

The following heap property is satisfied:

- for any two nodes i, j such that i is the parent of j, we have $key[A[i]] \leq key[A[j]]$.

A heap. Numbers in the circles denote key values of elements.

insert(v, key_value)

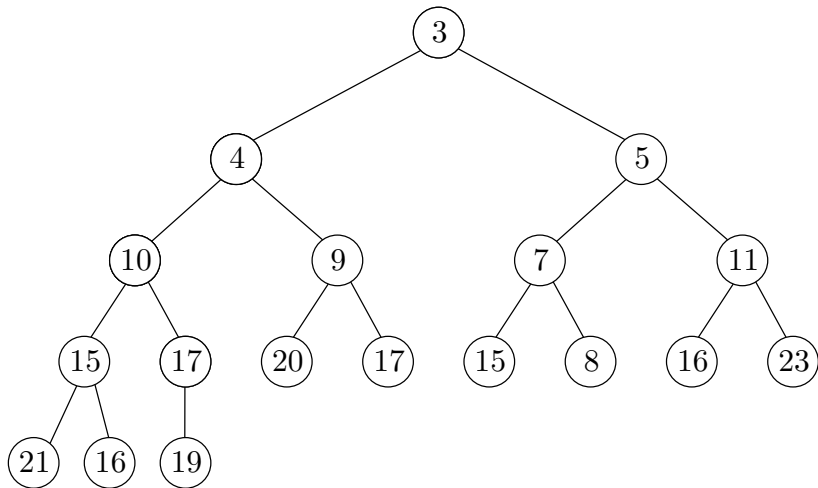1: $s \leftarrow s + 1$
2: $A[s] \leftarrow v$
3: $p[v] \leftarrow s$
4: $key[v] \leftarrow key\_value$
5: heapify_up(s)

heapify-up(i)

1: **while** $i > 1$ **do**
2:     $j \leftarrow \lfloor i/2 \rfloor$
3:     **if** $key[A[i]] < key[A[j]]$ **then**
4:        swap $A[i]$ and $A[j]$
5:        $p[A[i]] \leftarrow i, p[A[j]] \leftarrow j$
6:        $i \leftarrow j$
7:     **else** break

extract_min()

1: $ret \leftarrow A[1]$
2: $A[1] \leftarrow A[s]$
3: $p[A[1]] \leftarrow 1$
4: $s \leftarrow s - 1$
5: **if** $s \geq 1$ **then**
6:     heapify_down(1)
7: **return** ret

decrease_key$(v, key\_val$

1: $key[v] \leftarrow key\_value$
2: heapify-up$(p[v])$

heapify-down(i)

1: **while** $2i \leq s$ **do**
2:     **if** $2i = s$ or
    $key[A[2i]] \leq key[A[2i+1]]$ **then**
3:         $j \leftarrow 2i$
4:     **else**
5:         $j \leftarrow 2i + 1$
6:     **if** $key[A[j]] < key[A[i]]$ **then**
7:         swap $A[i]$ and $A[j]$
8:         $p[A[i]] \leftarrow i, p[A[j]] \leftarrow j$
9:         $i \leftarrow j$
10:     **else** break

- Running time of heapify_up and heapify_down: $O(\lg n)$
- Running time of insert, exact_min and decrease_key: $O(\lg n)$

data structures	insert	extract_min	decrease_key
array	$O(1)$	$O(n)$	$O(1)$
sorted array	$O(n)$	$O(1)$	$O(n)$
heap	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$

Two Definitions Needed to Prove that the Procedures Maintain Heap Property

Def. We say that H is almost a heap except that $key[A[i]]$ is too small if we can increase $key[A[i]]$ to make H a heap.

Def. We say that H is almost a heap except that $key[A[i]]$ is too big if we can decrease $key[A[i]]$ to make H a heap.

Outline

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

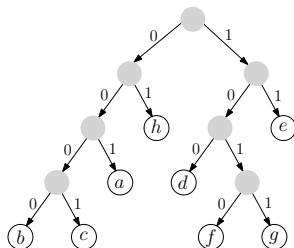**A:** Can not guarantee a unique decoding. For example, 00 can be decoded to aa or c.

Solution

Use prefix codes to guarantee a unique decoding.

Prefix Codes

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

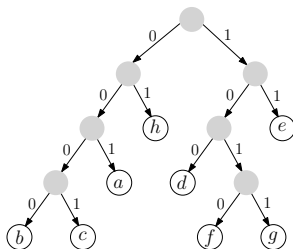| a	b	c	d
001	0000	0001	100

e	f	g	h
11	1010	1011	01

- 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/
- cadbhhefca
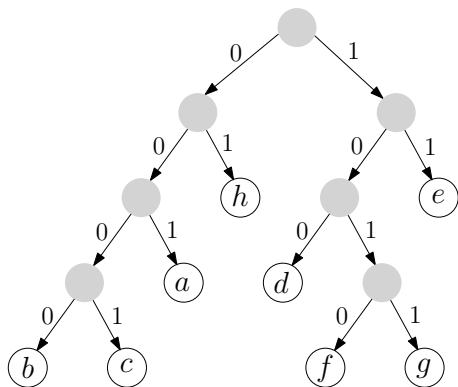
Properties of Encoding Tree

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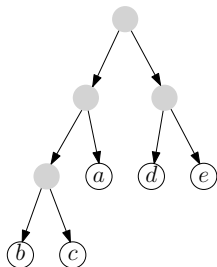- If coding scheme is not wasteful: a non-leaf has exactly two children

Best Prefix Codes
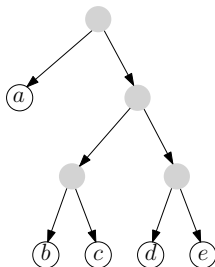
Input: frequencies of letters in a message

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

example

letters	a	b	c	d	e	
frequencies	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
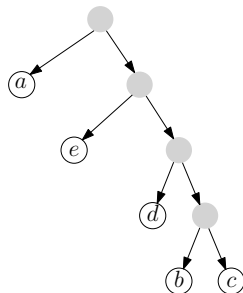
scheme 1   scheme 2   scheme 3

- 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

best to put the two least frenquent symbols here!
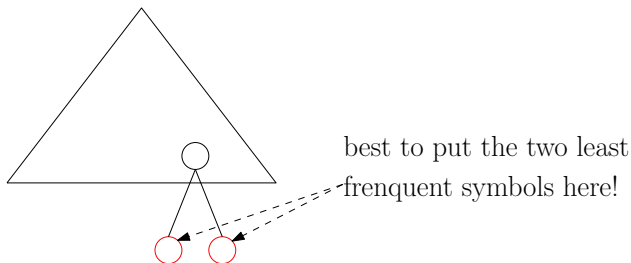
**Lemma** It is safe to make the two least frequent letters brothers.

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.
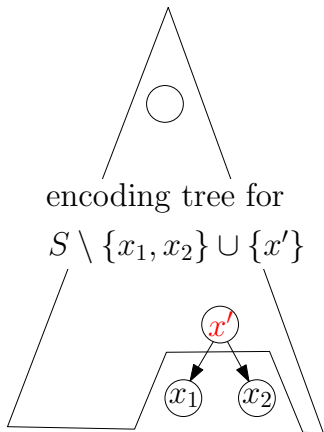
encoding tree for
$S \setminus \{x_1, x_2\} \cup \{x'\}$

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

$$
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + f_{x_1} d_{x_1} + f_{x_2} d_{x_2}
$$

$$
= \sum_{x \in S \setminus \{x_1, x_2\}} f_x d_x + (f_{x_1} + f_{x_2}) d_{x_1}
$$

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

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

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

In order to minimize
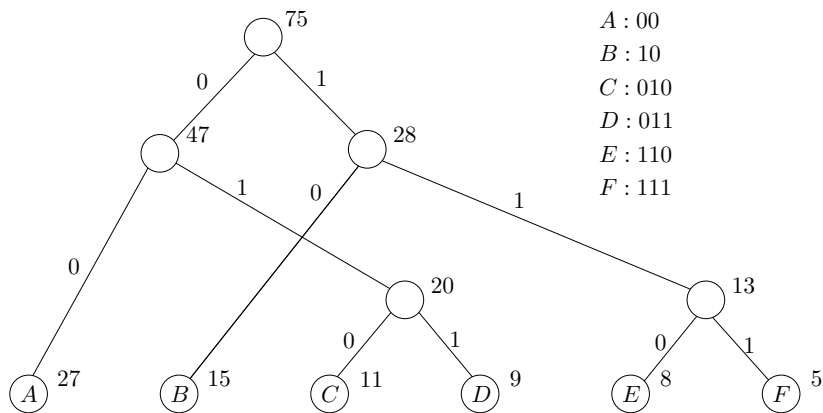
$$\sum_{x \in S} f_x d_x,$$

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!

$A : 00$
$B : 10$
$C : 010$
$D : 011$
$E : 110$
$F : 111$

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: $Q \leftarrow$ build-priority-queue(S)
2: **while** Q.size > 1 **do**
3:      $x_1 \leftarrow Q$.extract-min$()$
4:      $x_2 \leftarrow Q$.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.insert$(x', f_{x'})$
8: **return** the tree constructed

Outline

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

Analysis of Greedy Algorithm

- 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

Analysis of Greedy Algorithm

- 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