Discussion 13
Greedy Algorithm & Homework Review

Dec.2nd 2019

Overview

- Greedy Algorithm
  - Scheduling to Minimizing Lateness
  - Huffman coding
- Homework Review
  - Problem 2
  - Problem 3
  - Problem 5

Greedy Algorithm

Greedy algorithms

- Make the best choice at the moment.
  - No planning ahead. “Short-sighted”.
- Once choice made, it’s fixed.
  - No take-backs.
- Cons Doesn’t always find optimal answer.
- Pros Simple and fast. Sometimes optimal.

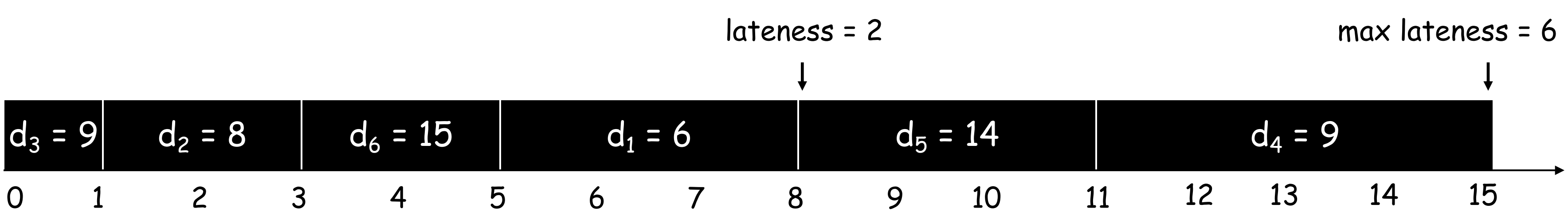
Scheduling to Minimizing Lateness

Scheduling to Minimizing Lateness

- Minimizing lateness problem.
  - Single resource processes one job at a time.
  - Job j requires t_j units of processing time and is due at time d_j.
  - If j starts at time s_j, it finishes at time $f_j = s_j + t_j$.
  - Lateness: $\ell_j = \max\{0, f_j - d_j\}$.
  - Goal: schedule all jobs to minimize maximum lateness $L = \max \ell_j$.

- Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time t_j.
  - [Earliest deadline first] Consider jobs in ascending order of deadline d_j.
  - [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

Minimizing Lateness: Greedy Algorithms

- Greedy template. Consider jobs in some order.
  - [Shortest processing time first] Consider jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

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

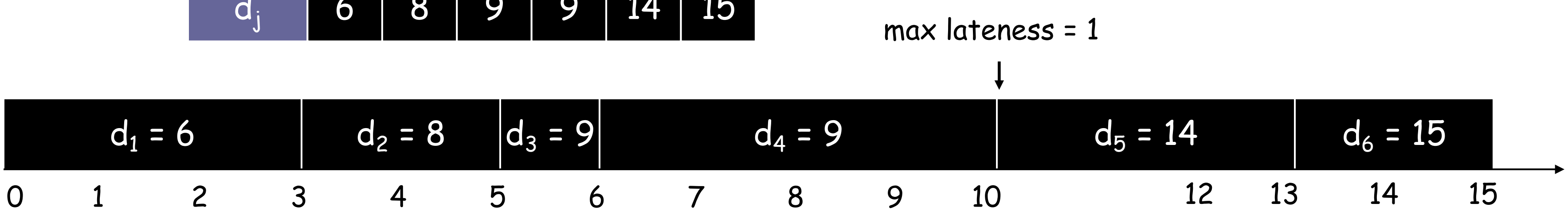
	1	2
t_j	1	10
d_j	2	10

Minimizing Lateness: Greedy Algorithm

- Greedy algorithm. Earliest deadline first.

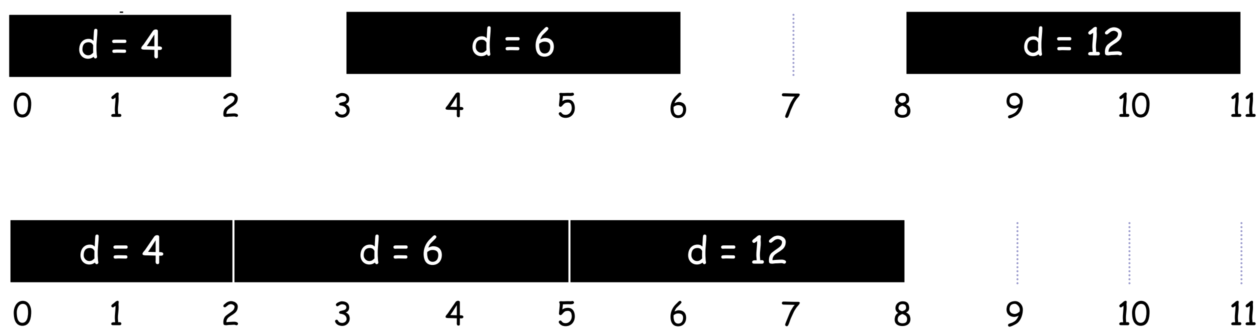
```
Sort n jobs by deadline so that d1 ≤ d2 ≤ … ≤ dn

t ← 0
for j = 1 to n
   Assign job j to interval [t, t + tj]
   sj ← t, fj ← t + tj
   t ← t + tj
output intervals [sj, fj]
```

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness: No Idle Time

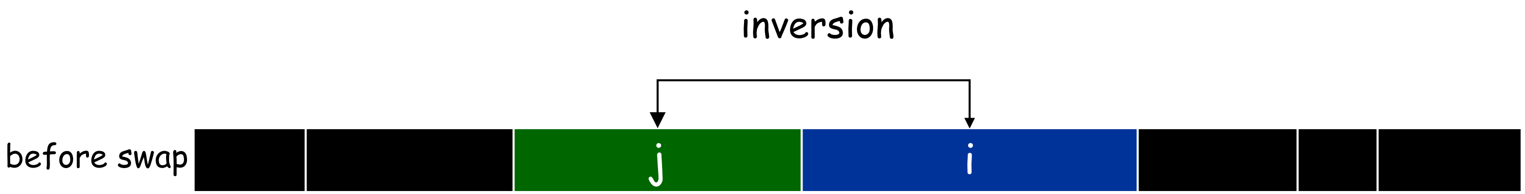
- Observation. There exists an optimal schedule with no idle time.



- Observation. The greedy schedule has no idle time.

Minimizing Lateness: Inversions

- Def. An inversion in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i.

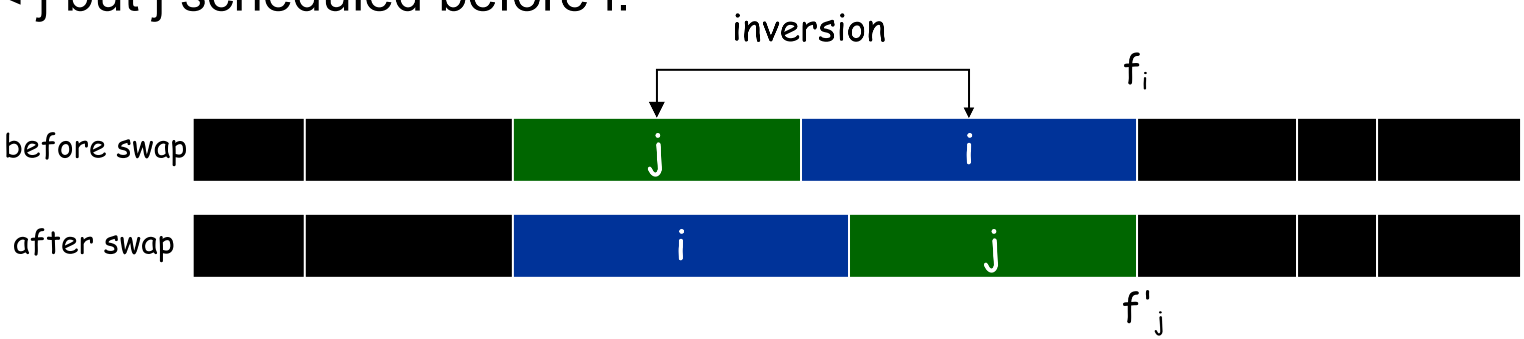


- Observation. Greedy schedule has no inversions.

- Observation. If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.

Minimizing Lateness: Inversions

- Def. An inversion in schedule S is a pair of jobs i and j such that: $i < j$ but j scheduled before i.



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

- Pf. Let ℓ be the lateness before the swap, and let ℓ' be it afterwards.
  - $\ell'_k = \ell_k$ for all $k \neq i, j$
  - $\ell'_i \leq \ell_i$
  - If job j is late:

$$\begin{aligned} \ell'_j &= f'_j - d_j && \text{(definition)} \\ &= f_i - d_j && (j \text{ finishes at time } f_i) \\ &\leq f_i - d_i && (i < j) \\ &\leq \ell_i && \text{(definition)} \end{aligned}$$

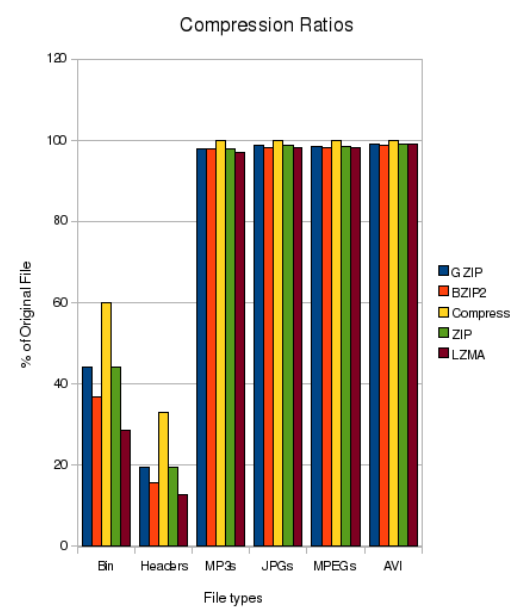
Minimizing Lateness: Analysis of Greedy Algorithm

- Theorem. Greedy schedule S is optimal.
- Pf. Define S* to be an optimal schedule that has the fewest number of inversions, and let's see what happens.
  - Can assume S* has no idle time.
  - If S* has no inversions, then S = S*.
  - If S* has an inversion, let i-j be an adjacent inversion.
    - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions
    - this contradicts definition of S*

Huffman Coding

Compression

- Storing and transmitting data is expensive. Compression represents data more compactly.
- ASCII has 256 characters, so we use $\log_2(256)=8$ bits to represent each character.
- But typically some characters appear more often than others. So we shouldn't use same number of bits for all letters.
- Basic idea for compression is to use different length bitstrings.
  - Use short bitstrings to represent common characters.
  - Use long bitstrings to represent uncommon characters.
  - We save space on average.



Lossless vs. lossy compression

- Different algorithms for different applications.
- Lossless compression used in settings where losing even one bit can make data useless.
  - Ex Computer code, financial document.
  - Typical compression ratio is 2:1.
  - Huffman encoding is loseless.
- Lossy compression used when data still useful after losing some information.
  - Ex Audio and video, MP3 and MPEG.
  - We can’t hear high frequencies or see fast movement, so we can discard this info.
  - Typical compression ratio is 5:1-50:1.

Variable length encoding

- Let's compress "lollapalooza".
- There are 5 different letters. If we use the same length bitstring to represent each letter, we need 3 bits per letter.
  - We use 36 bits total.
- To use different length bitstrings, first count how many times each letter appears.
  - 4 l's, 3 a's, 3 o's, 1 p, 1z.
- Use shorter bitstrings for more frequent letters.
- Use the encoding l=00, a=01, o=10, p=110, z=111.
  - Encoding of "lollapalooza" is 00100000011100100101011101, formed by replacing each letter by its encoding.
  - We use 26 bits, for a 28% savings.

Ambiguity

- We want the codewords to be short, but we also need them to be unambiguously decodable.
- Ex If we use l=0, a=01, o=10, p=110, z=111, then “lollapalooza” is only 22 bits.
  - But we can’t decode this encoding!
  - If we see 00010, we can’t tell whether this is encoding llal=[0,0,01,0], or lllo=[0,0,0,10].
- We could use a separator, 0#0#01#0 vs 0#0#0#10. But that’s wasteful.
- Instead, we use prefix-free codes, which are unambiguously decodable.

Prefix-free codes

- Let W be the set of codewords we use. Then W is prefix-free if no codeword in W is a prefix of another codeword.
  - 00,10,001,100 is not prefix-free.
    - 00 is a prefix of 001, and 10 is a prefix of 100.
  - 00,01,10,110,111 is prefix-free.
- Prefix-free codes allow unique decoding.
  - Given the encoded string, just keep reading until you've read a complete codeword.
  - This codeword can't be part of a longer codeword, because the code is prefix-free.

Decoding prefix-free codes

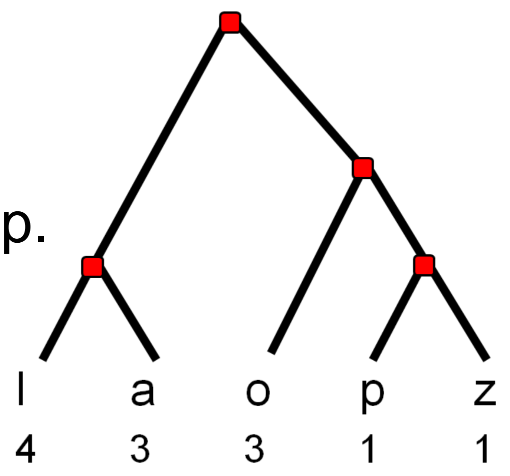
- Let S=00100000011100100101011101, W={00,01,10,110,111 }, representing l,a,o,p,z.

00100000011100100101011101	00→l
00100000011100100101011101	10→o
00100000011100100101011101	00→l
00100000011100100101011101	00→l
00100000011100100101011101	00→a
00100000011100100101011101	110→p

…

Huffman coding

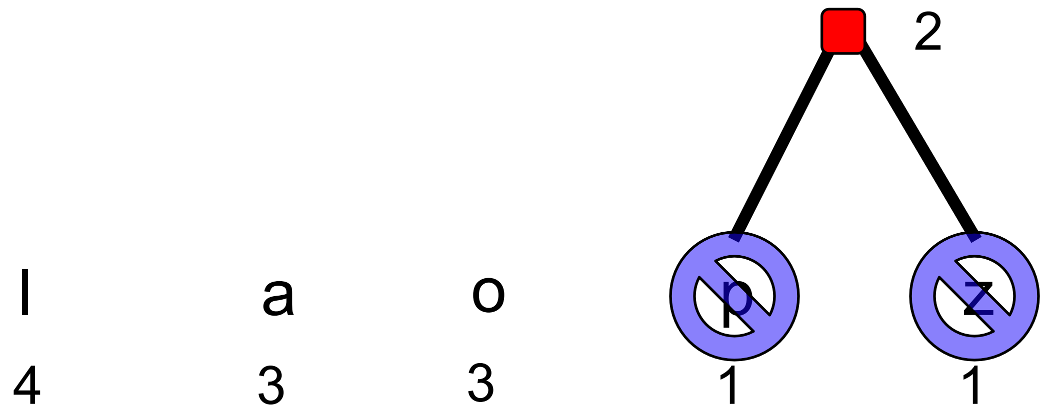
- Huffman encoding is an optimal prefix-free code, invented in 1951.
- First, find the frequencies of the letters in your text.
  - For “loolapalooza”, it’s [l,a,o,p,z] $\rightarrow$[4,3,3,1,1].
- Now, build a binary tree on the letters bottom up.
  - Make each letter a leaf, and set its weight to its frequency.
  - Take the two lowest weight nodes
    - Make them the children of a parent node.
    - Set the weight of the parent node equal to the sum of the weights of the two children.
    - Remove the two nodes.
    - Notice this is a greedy step.
  - Repeat till all nodes part of one tree.
- Represent left by 0, right by 1.
  - A letter’s encoding is represented by its path from the root.



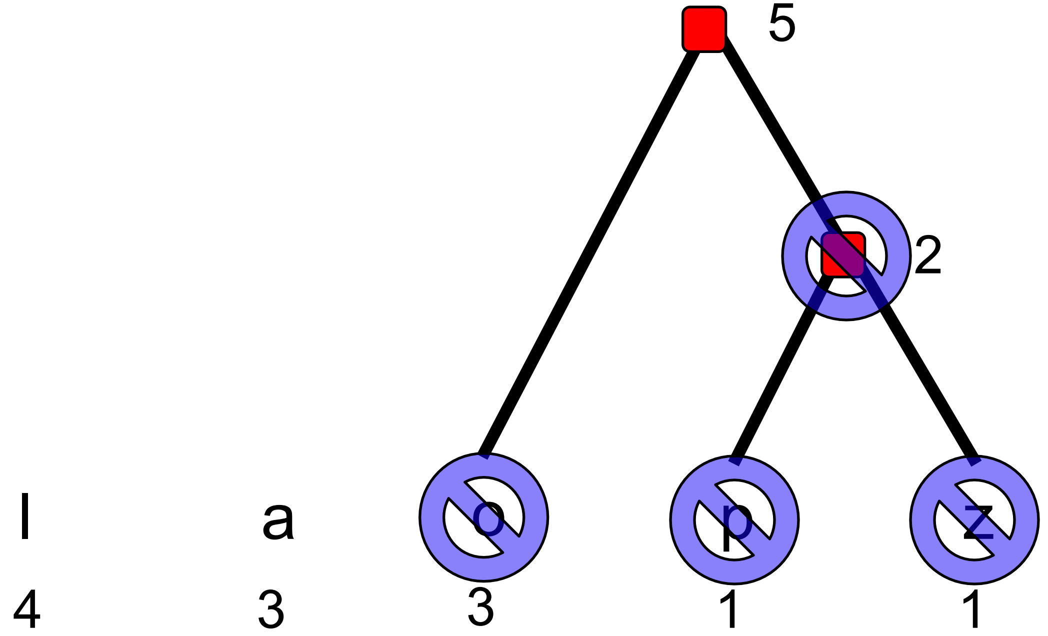
Huffman coding example

l	a	o	p	z
4	3	3	1	1

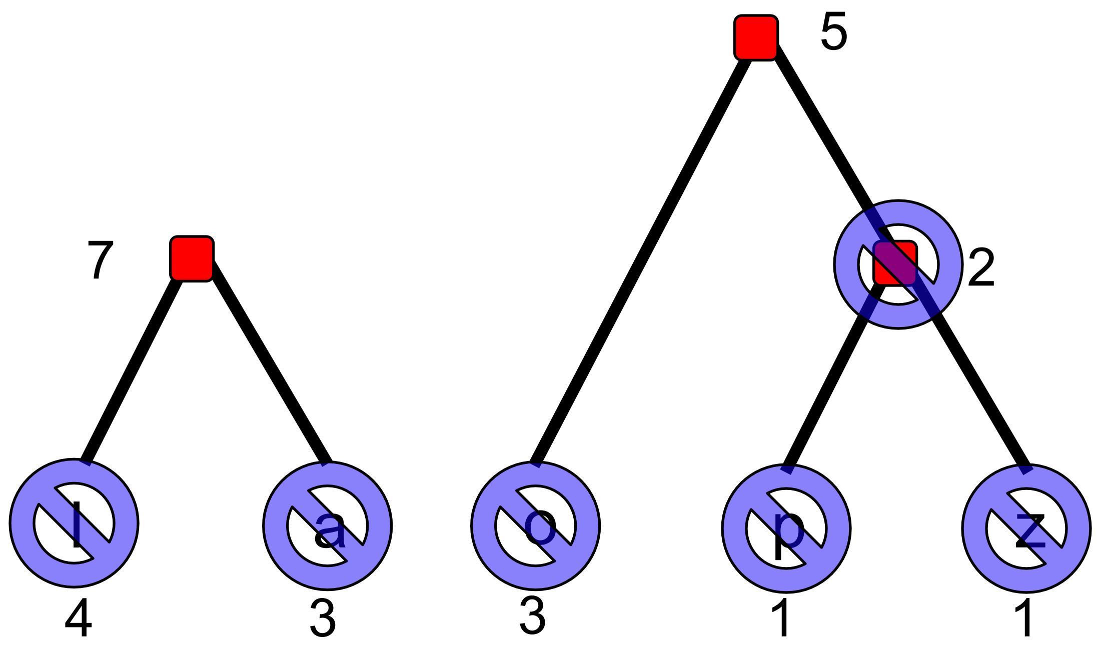
Huffman coding example



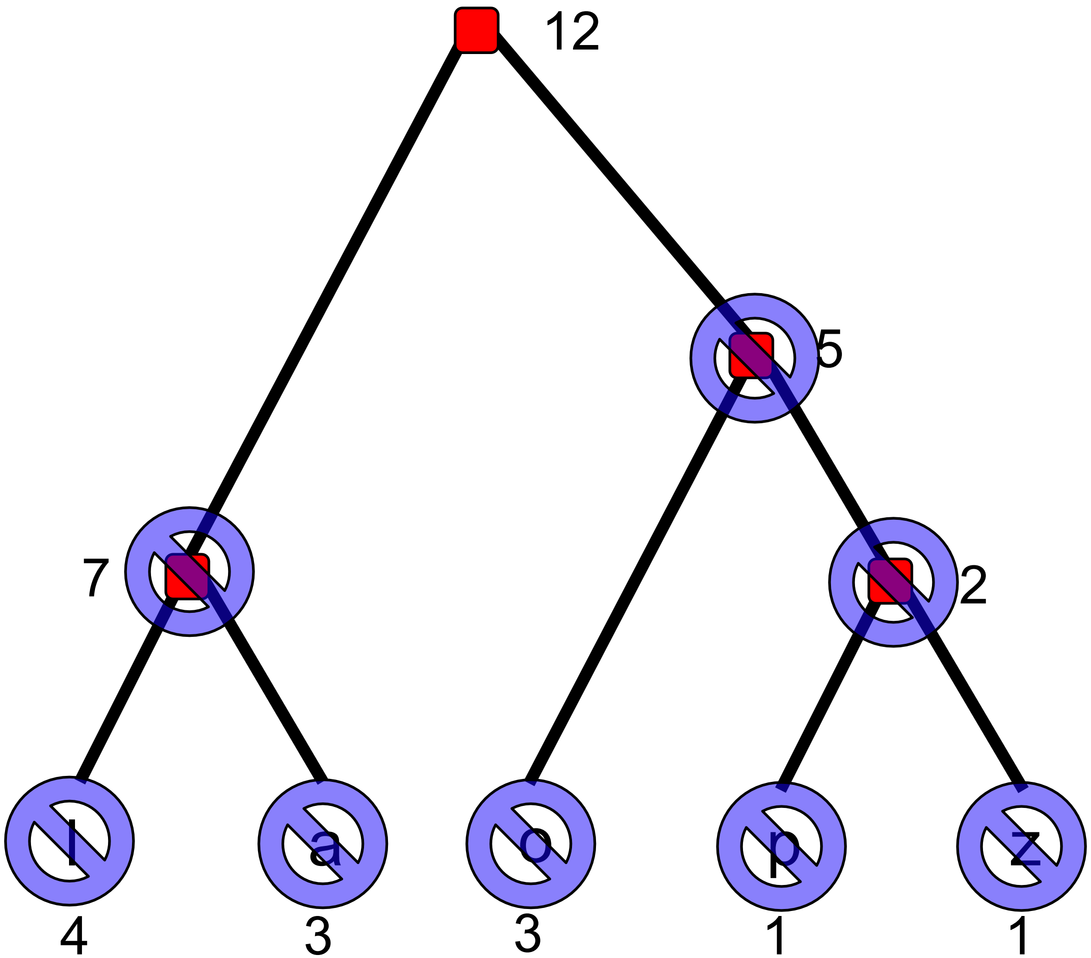
Huffman coding example



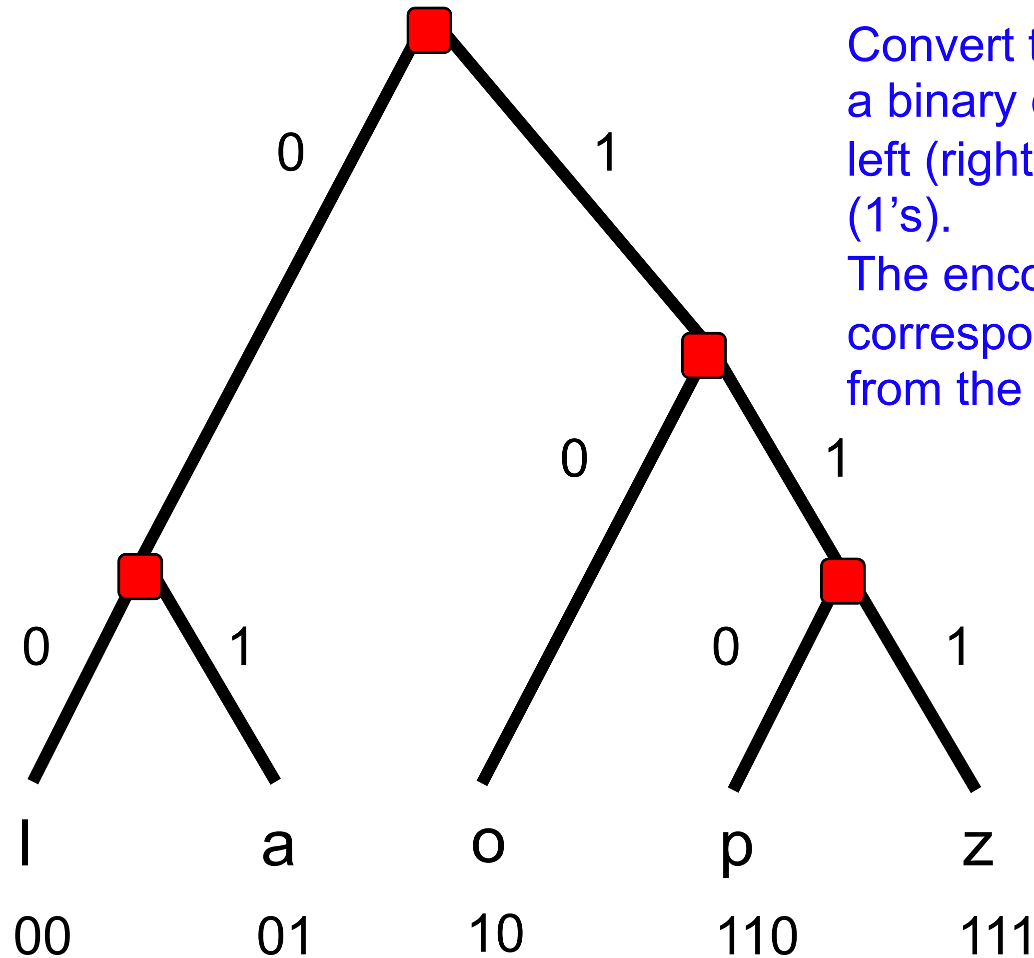
Huffman coding example



Huffman coding example



Huffman coding example



Convert the Huffman tree into a binary encoding, by treating left (right) branches as 0's (1's).

The encoding of the letters corresponds to their path from the root.

Huffman implementation

Let $S=s_1s_2\ldots s_n$ be a string. Let f(s) be the number of occurrences of char s in S.

for i=1 to n

add (s_i, f(s_i)) to a min-heap H

for i=1 to n-1

left $\leftarrow$ removeMin(H)

right $\leftarrow$ removeMin(H)

make a new node parent

set parent's left and right children to left, right

add(parent, f(left) + f(right)) to H

a letter's encoding is represented by its path

Huffman's complexity

Total time is O(n log n).

Each add takes O(log n) time.

for i=1 to n

add (s_i, $f(s_i)$) to a min-heap H

Each remove takes O(log n) time.

for i=1 to n-1

left $\leftarrow$ removeMin(H)

Add takes O(log n) time.

right $\leftarrow$ removeMin(H)

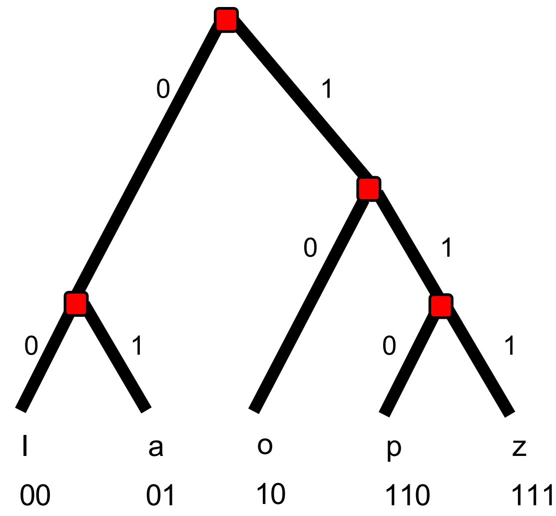
make a new node parent

set parent's left and right children to left, right

add(parent, f(left) + f(right)) to H

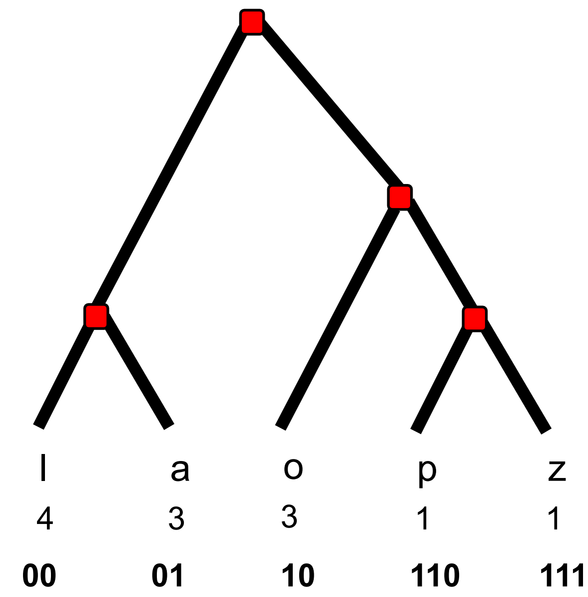
Huffman code is prefix-free

- Any two codewords correspond to paths from the root to leaves.
  - The 2 paths split from each other somewhere.
  - After the split, neither codeword is a prefix of the other.
- Huffman codes are uniquely decodable.



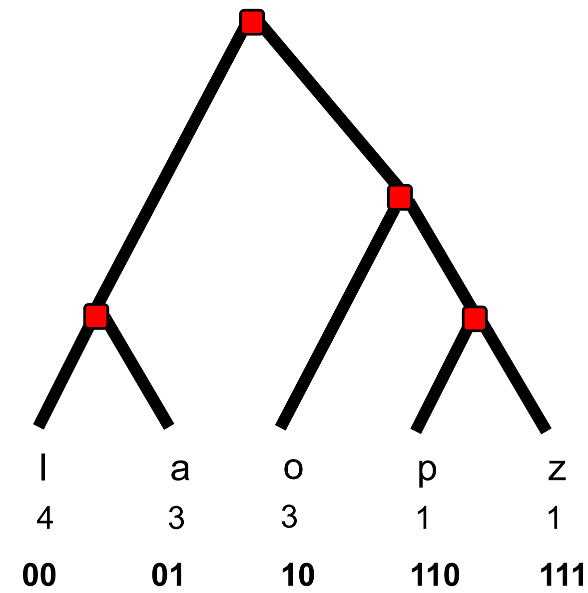
Huffman code is optimal

- Huffman encoding gives the shortest uniform encoding of strings.
  - Uniform basically means you can't change your encoding method for different strings.
- Call an encoding in which codewords are derived from paths in trees a tree code.
- Fact There exists tree codes that are optimal.
- Since the Huffman code is a tree code, to prove Huffman is optimal, we just need to prove it's an optimal tree code.



Huffman code is optimal

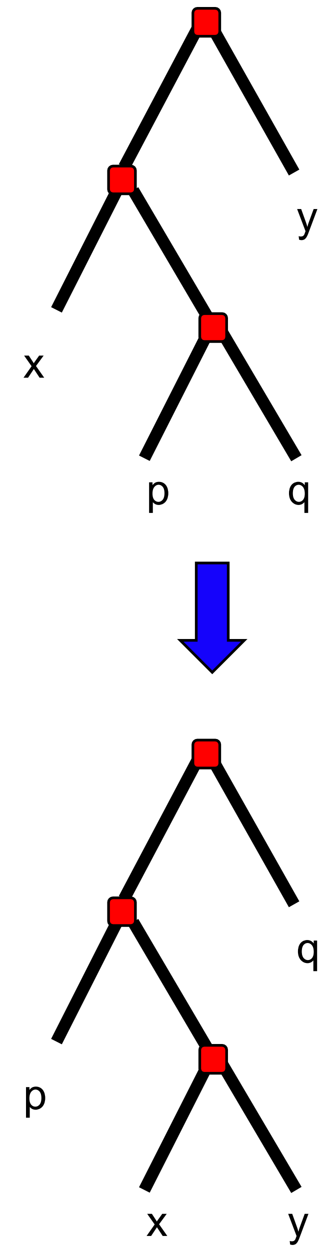
- Def The cost of a tree code is $cost(T) = \Sigma_{v \in T} d(v) \cdot f(v)$, where $d(v)$ denotes the depth of letter v, and $f(v)$ denotes its frequency.
  - $cost(T)$ = number of bits to represent original string.
- Claim 1 In an optimal tree code, every leaf has a sibling.
- Proof Otherwise, replace the lone leaf by its parent to get a tree code with lower cost.



cost(T) = 4*2+3*2+3*2+
3*1+3*1=26

Huffman code is optimal

- Claim 2 Consider the two least frequent letters x and y. In the an optimal tree code T, x and y are siblings of each other at the max depth of the tree.
- Proof Suppose not, and let p be a node at the max depth.
  - p has a sibling q by Claim 1.
  - p and q have higher frequency than x and y, resp.
  - Create a new tree where we swap p and x, and q and y.
  - The new tree has strictly lower cost than T, since p, q have higher frequency than x, y. Contradiction.



Huffman code is optimal

- Thm Huffman code is optimal.
- Proof Use induction on number of letters in the code. Suppose it's true up to $n-1$.
  - Consider a code with n letters. Let x,y be the letters with the lowest frequency.
  - Let T be the Huffman code tree on the n letters.
    - Create a new node z with frequency $f(z) = f(x) + f(y)$.
    - Let S be a tree formed from T by removing x and y, and replacing their parent by z.
  - S is the Huffman code on the $n-1$ letters.
    - Because of the recursive way Huffman encoding works.
  - S is an optimal tree code on the $n-1$ letters, by induction.
  - $cost(T) = cost(S) + f(x) + f(y)$.
    - All the nodes in S and T are the same, except x,y,z.
    - $cost(z) = d(z) \cdot f(z) = d(z) \cdot (f(x) + f(y))$.
    - $d(z) = d(x) - 1 = d(y) - 1$.
    - $cost(x) + cost(y) = cost(z) + f(x) + f(y)$.

Huffman code is optimal

- Proof (continued)
  - Let T' be an optimal tree code on the n letters.
    - By Claim 2, x and y are siblings in T'.
    - Merge them into a node z', with $f(z') = f(x) + f(y)$. Form a tree S' by removing x and y from T', and replacing their parent by z'.
    - S' is a tree code on $n - 1$ letters.
  - $cost(T') = \ cost(S') + f(x) + f(y) \geq cost(S) + f(x) + f(y) \ = \ cost(T)$.
    - First equality because x,y at depth one greater than z.
    - First inequality because S is opt tree code on $n - 1$ letters.
  - So, the tree T produced by Huffman encoding is optimal.

Homework Review

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in O(n log n) time.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in O(n log n) time.

- Observation
  - If there is a Majority Element, then this element will appear in at least one sub-array after splitting the original array into halves with almost same size.
- Main Idea
  - Try to determine whether the majority of either subset is the majority of the whole array.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in O(n log n) time.

- Step 1: Divide array into two sub-arrays
- Step 2: Find majority element in each sub array
- Step 3: Compare majority element (***left***, ***right***)
  - ***left*** &/or ***right***: Scan whole array to determine which is majority
  - Neither exist: Return ‘No Majority Element’

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(a) Show how to solve this problem in O(n log n) time.

T(n/2)

- Step 1: Divide array into two sub-arrays
- Step 2: Find majority element in each sub array
- Step 3: Compare majority element (***left***, ***right***)
  - ***left*** &/or ***right***: Scan whole array to determine which is majority
  - Neither exist: Return 'No Majority Element'

O(n)

Time Complexity:
$T(n) = 2T(n/2)+O(n)$
$T(n) = n\log n$

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Main Idea: Firstly, we can find an element may be the majority. We scan the array get if the element is same as the majority then count adds 1, else count subs 1. And if count equals 0, we just select majority as the next element. And then, after scan the last element, we check whether this majority shows more than n/2 times.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Correctness: (1) the majority of the array occurs more than n/2 times. (2) If Majority Element exists, it must be selected after first traverse.
So if the count1 is greater than 1, it has the probability of being majority, since it may occurs more than any other elements. And then we can scan the array to get whether it's the majority. If the count2 is larger than n/2, it's majority. Otherwise, this array does not have majority.

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 0
Count2 = 0
Majority = None

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 2
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 0
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 3

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 0
Count2 = 0
Majority = 3

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 2
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 2
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?” in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 0
Majority = 7

Problem 2: An array A=[1…n] is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is A[i] > A[j]?". (For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is A[i] = A[j]?" in constant time. Four part solutions are required for each part below.

(b) Show how to solve this problem in O(n) time.

Example:

7	7	1	2	3	7	7	7	4	7	3

Count1 = 1
Count2 = 6 > 11/2
Majority = 7

7 is Majority Element

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Correctness: Removing a number, m, creates an imbalance of 0s and 1s. If N was odd, the number of 0s in least significant bit position should equal the number of 1s. If N was even, then number of 0s should be 1 more than the number of 1s. Thus if all numbers are present, count(0s) ≥ count(1s). We have four cases:

- If LSB of m is 0, removing m removes a 0
  - If N is even, count(0s) = count(1s)
  - If N is odd, count(0s) < count(1s)

- If LSB of m is 1, removing m removes a 1
  - If N is even, count(0s) > count(1s)
  - If N is odd, count(0s) > count(1s)

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can’t even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Correctness:

- If LSB of m is 0, removing m removes a 0
  - If N is even, count(0s) = count(1s)
  - If N is odd, count(0s) < count(1s)
- If LSB of m is 1, removing m removes a 1
  - If N is even, count(0s) > count(1s)
  - If N is odd, count(0s) > count(1s)

Notice that if count(0s) ≤ count(1s) m’s least significant bit is 0. We can discard all numbers with LSB of 1 because removing m does not the count of 1s. If count(0s) > count(1s), m’s least significant bit is 1. Likewise we can discard all numbers with LSB of 0. Assume that this condition applies for all bit positions up to k. If we look at the k+1)th bit position, the condition above holds true. The elements at the (k+1)th bit position have the same bit at the k^{th} position as m and thus are the only elements we are interested in at the (k+1) position.

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can’t even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Time Complexity:

$T(n) = T(n/2) + O(n)$

$T(n) = O(n)$

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can’t even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A

0 → 0000
1 → 0001
2 → 0010
3 → 0011
4 → 0100
5 → 0101
6 → 0110
8 → 1000
9 → 1001
10 → 1010

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A

0 → 0000
1 → 0001
2 → 0010
3 → 0011
4 → 0100
5 → 0101
6 → 0110
8 → 1000
9 → 1001
10 → 1010

Count(0) = 6 < 4 = Count(1)

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the j^{th} bit of A[i]". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A

~~0 → 0000~~

1 → 0001

~~2 → 0010~~

3 → 0011

~~4 → 0100~~

5 → 0101

~~6 → 0110~~

~~8 → 1000~~

9 → 1001

~~10 → 1010~~

Missing element = ___1

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the j^{th} bit of A[i]". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A

1 → 0001

3 → 0011

5 → 0101

9 → 1001

Count(0) = 3 > 1 = Count(1)

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the j^{th} bit of A[i]". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A ~~1 → 0001~~

3 → 0011

~~5 → 0101~~

~~9 → 1001~~

Missing element = __11

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the j^{th} bit of A[i]". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A 3 → 0011

Count(0) = 1 > 0 = Count(1)

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the j^{th} bit of A[i]". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A ~~3 → 0011~~

Missing element = _111

Problem 3: An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is ”fetch the j^{th} bit of A[i]”. Using only this operation to access A, give an algorithm that determines the missing integer by looking at only O(N) bits. (Note that there are O(NlogN) bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s.

Example: N = 10 A = [0, 1, 2, 3, 4, 5, 6, 8, 9, 10]

A Empty

Missing element = 0111 = 7

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than $O(n)$.

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

Main Idea: In this problem, design a algorithm which get the s^{th} element in the k sorted array. Using the search method like binary, every time erase almost half of the elements.

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

Algorithm 6 $sth(s, a_1[1, \cdots, n_1], a_2[1, \cdots, n_2], \cdots, a_k[1, \cdots, n_k])$

if $n_1 \leq 1$ and $n_2 \leq 1$ and $\cdots$ and $n_k \leq 1$ **then**
  $a \leftarrow Sorted([a_1[1], a_2[1], \cdots, a_k[1]])$
  return $a[s]$
end if
for $i = 1$ to k **do**
  if $n_i \geq 1$ **then**
    $b_i = Median(a_i)$
  end if
end for
$median \leftarrow Median([b_1, b_2, \cdots, b_k])$
for $i = 1$ to k **do**
  $c_i = BinarySearch(median, a_i)$
end for
if $Sum(c_i) < s$ **then**
  return $sth(s - Sum(c_i), a_1[c_1, \cdots, n_1], a_2[c_2, \cdots, n_2], \cdots, a_k[c_k, \cdots, n_k])$
else
  return $sth(s, a_1[1, \cdots, c_1], a_2[1, \cdots, c_2], \cdots, a_k[1, \cdots, c_k])$
end if

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

Algorithm 6 $sth(s, a_1[1, \cdots, n_1], a_2[1, \cdots, n_2], \cdots, a_k[1, \cdots, n_k])$

if $n_1 \leq 1$ and $n_2 \leq 1$ and $\cdots$ and $n_k \leq 1$ **then**
  $a \leftarrow Sorted([a_1[1], a_2[1], \cdots, a_k[1]])$
  return $a[s]$
end if
for $i = 1$ to k **do**
  if $n_i \geq 1$ **then**
    $b_i = Median(a_i)$ K
  end if
end for
$median \leftarrow Weighted\_Median([b_1, b_2, \cdots, b_k, n_1, n_2, \cdots n_k])$ K //Weighted median is the function that sort the median first and add up the size of the previous lists until they add up to half.
for $i = 1$ to k **do**
  $c_i = BinarySearch(median, a_i)$ KlogL
end for
if $Sum(c_i) < s$ **then**
  return $sth(s - Sum(c_i), a_1[c_1, \cdots, n_1], a_2[c_2, \cdots, n_2], \cdots, a_k[c_k, \cdots, n_k])$
else
  return $sth(s, a_1[1, \cdots, c_1], a_2[1, \cdots, c_2], \cdots, a_k[1, \cdots, c_k])$
end if

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

Time Complexity:

- Median is within [n/4, 3n/4]
- The iteration ends by O(logL) times
- $T(n) <= O(K\log L)+O(K)+T(3n/4)$
- $T(n) <= O(K\log L)O(\log L) = O(K\log^2 L) < O(KL) = O(n)$

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

L=7

K=4

1	2	9	13	19	21	30
3	15	17	22	29	33	36
5	7	10	14	16	27	35
4	6	12	20	24	31	34

Find median for each list
(Suppose that if num is even, we choose num/2+1 as median)

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

L=7

K=4

1	2	9	13	19	21	30
3	15	17	22	29	33	36
5	7	10	14	16	27	35
4	6	12	20	24	31	34

Find median of medians

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

L=7

K=4

1	2	9	13	19	21	30
3	15	17	22	29	33	36
5	7	10	14	16	27	35
4	6	12	20	24	31	34

Use Binary Search to find elements smaller than current median

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).

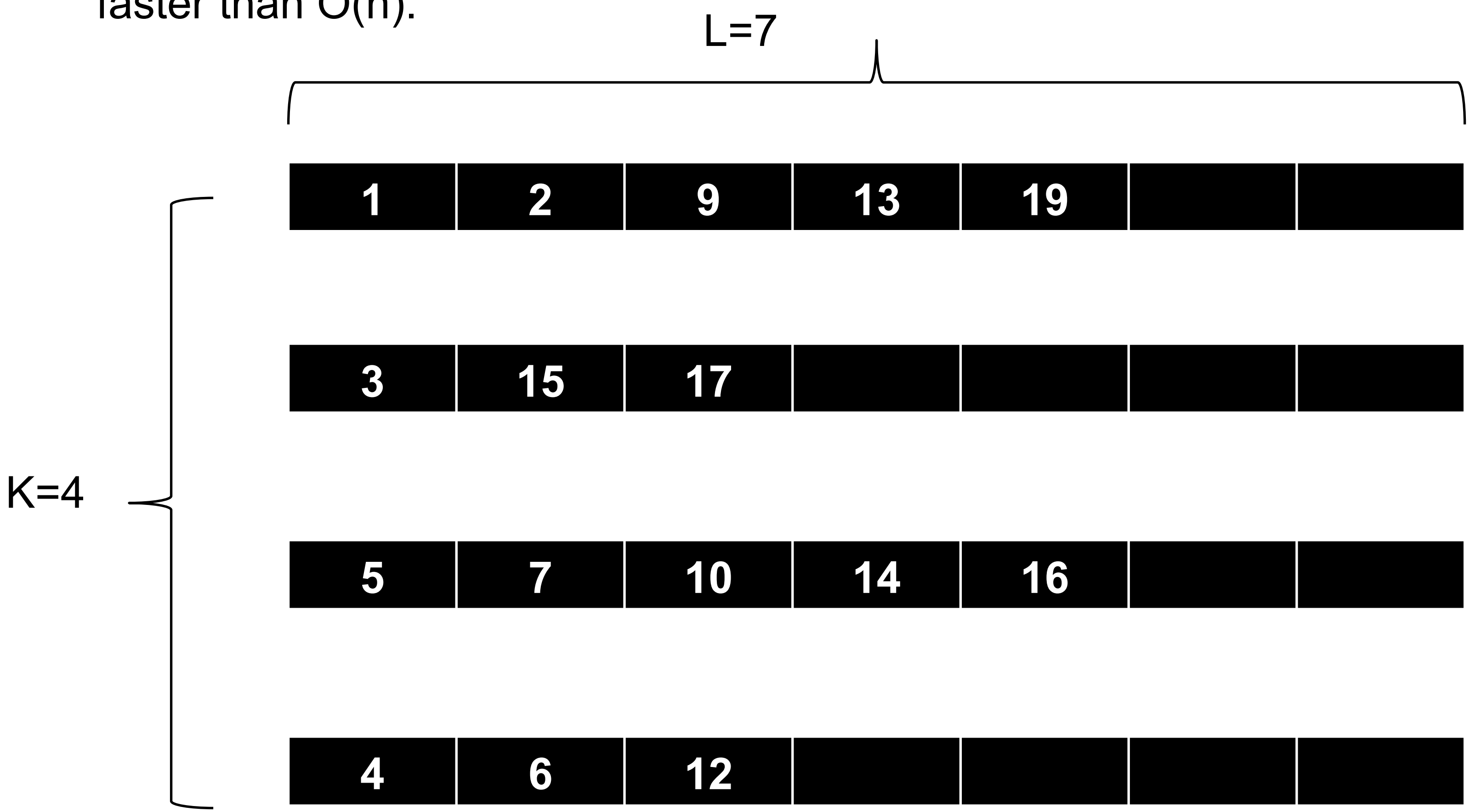
L=7

K=4

1	2	9	13	19	21	30
3	15	17	22	29	33	36
5	7	10	14	16	27	35
4	6	12	20	24	31	34

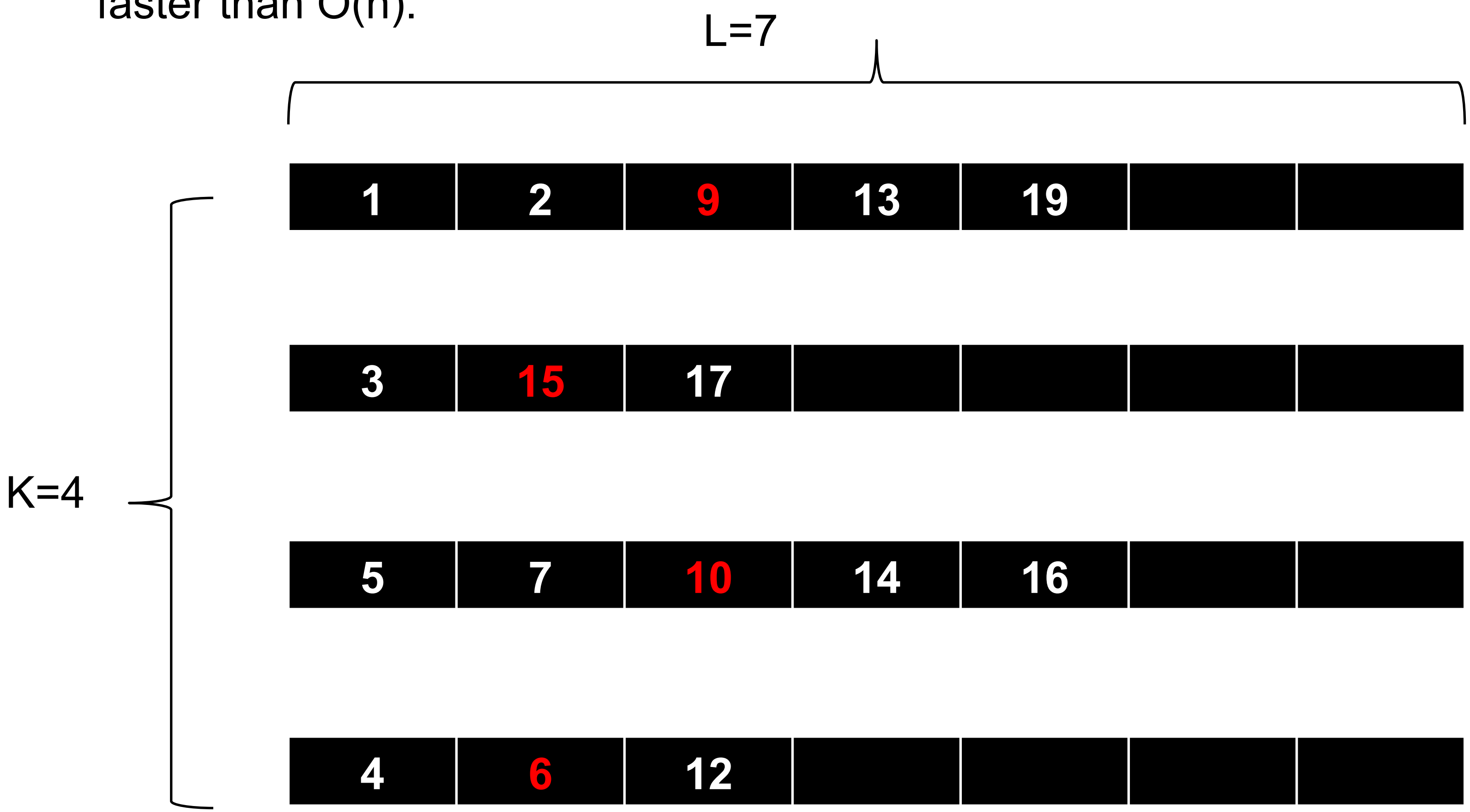
Count(Smaller) = 16 > 15 = 4*7/2+1 = Median Index
Desired Median is in Smaller sub-arrays

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



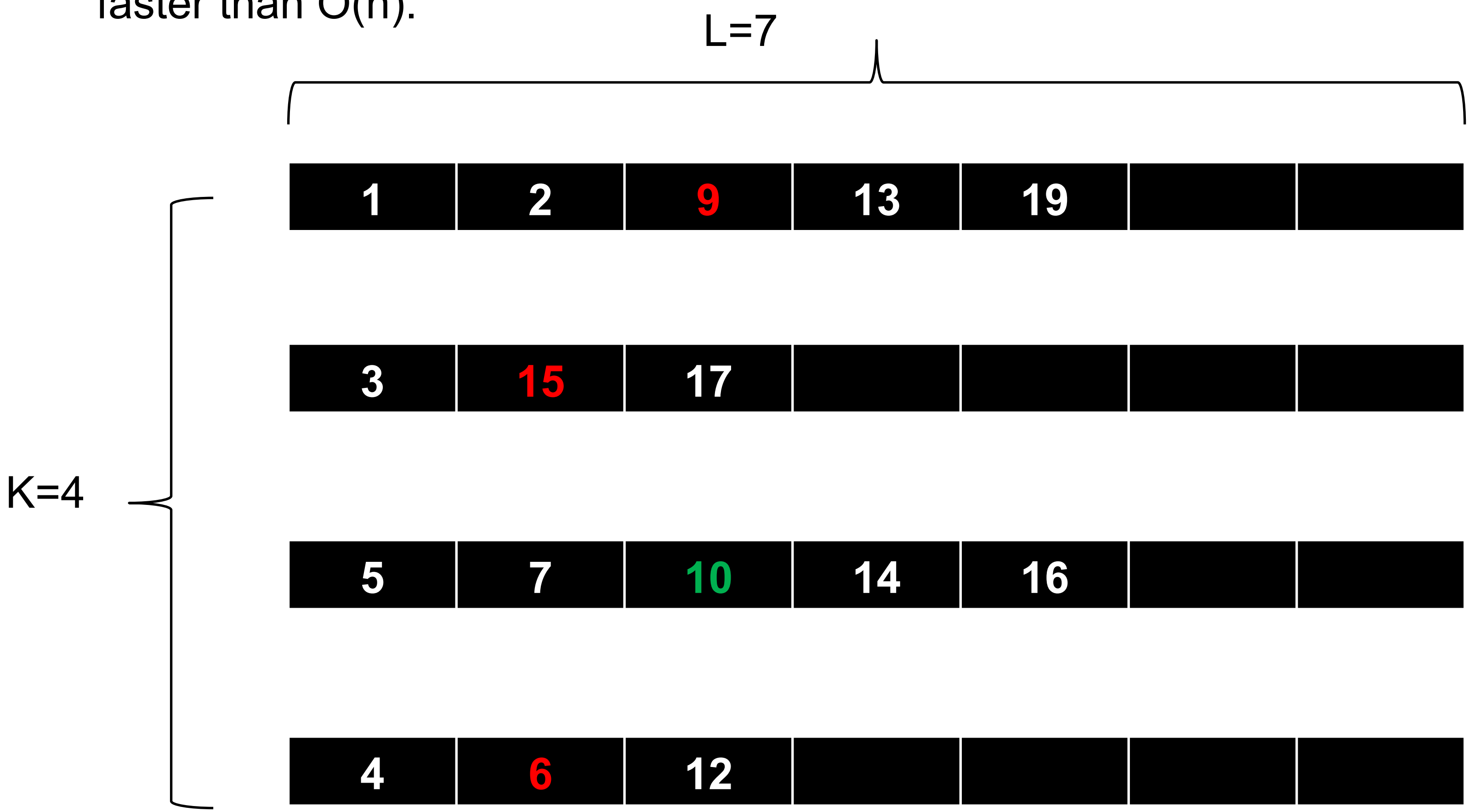
Delete larger numbers

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



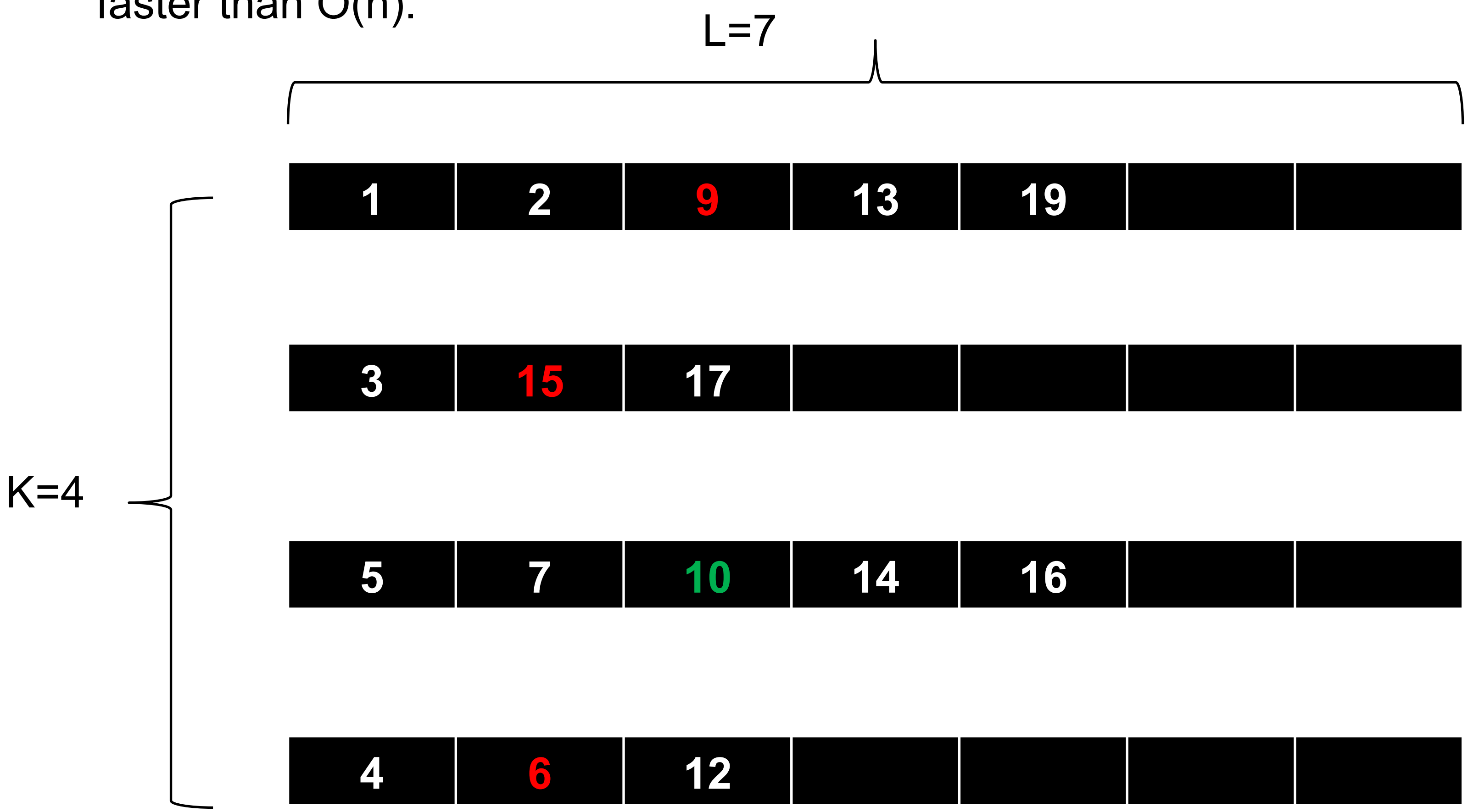
Continue to find median in each sub-array

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



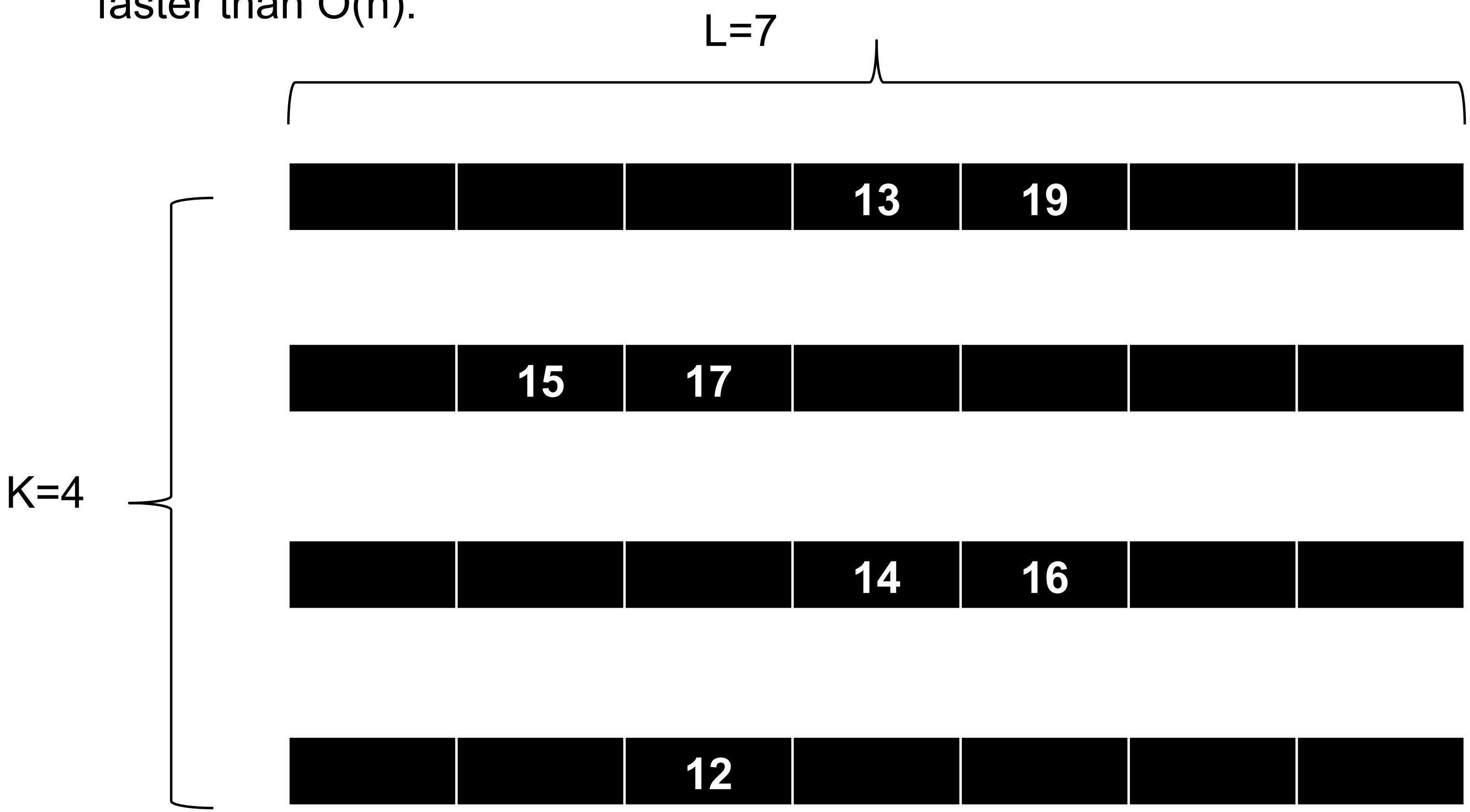
Find median of medians

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



Count(Smaller) = 8 < 15 = Median Index
Desired median is in larger part

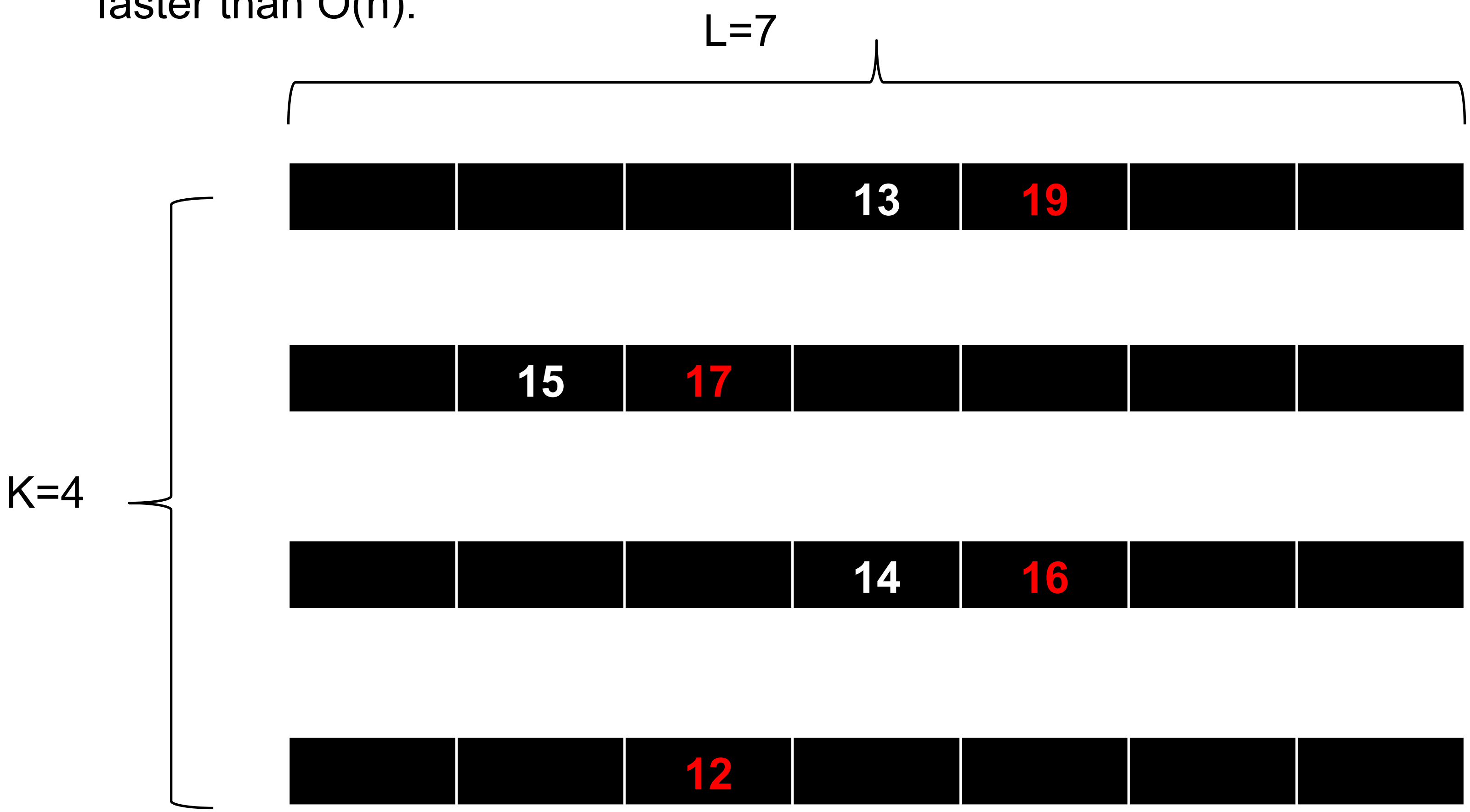
Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



Erase smaller elements

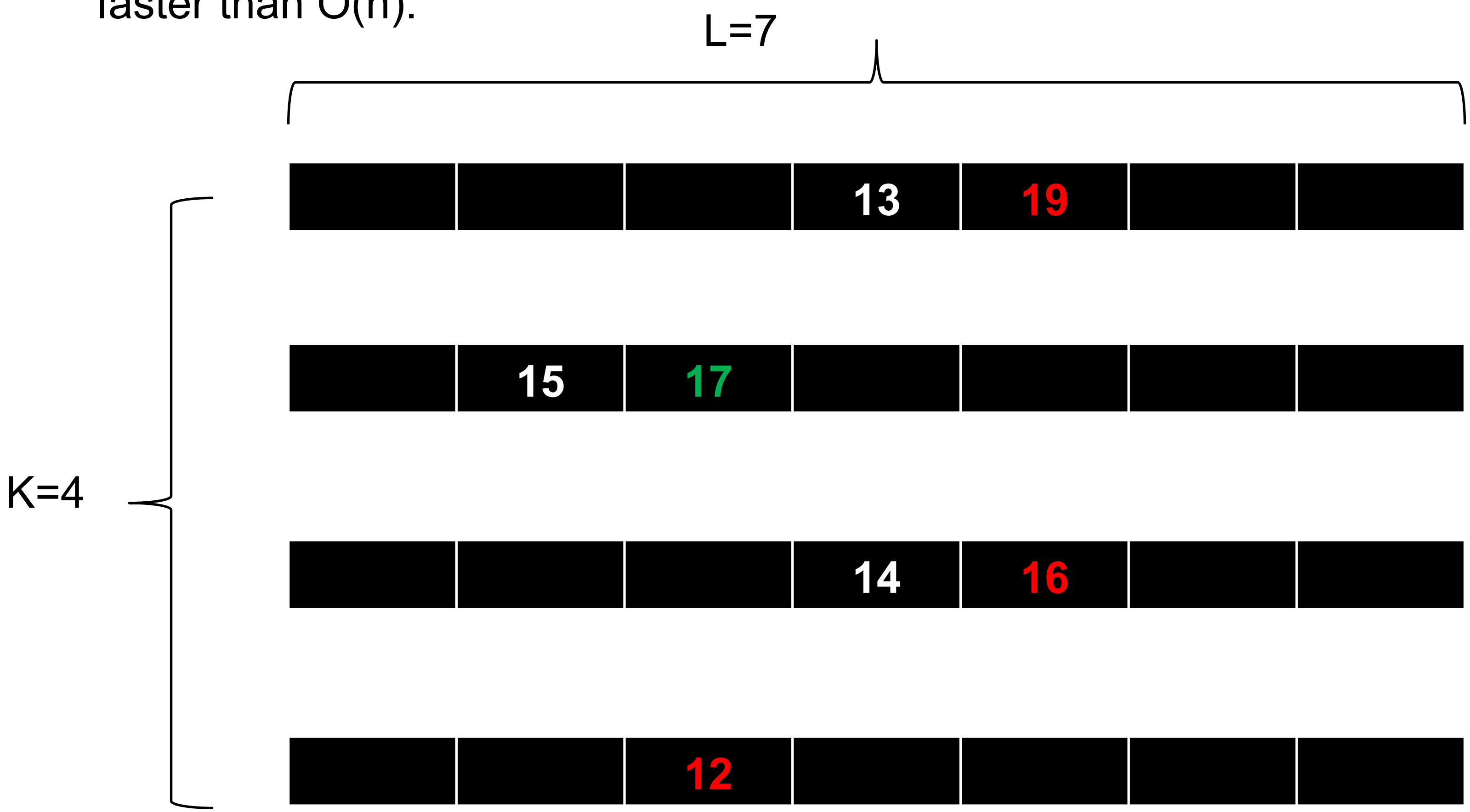
To find Index = Index – Count(Smaller) -1 = 6 in the rest elements.

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



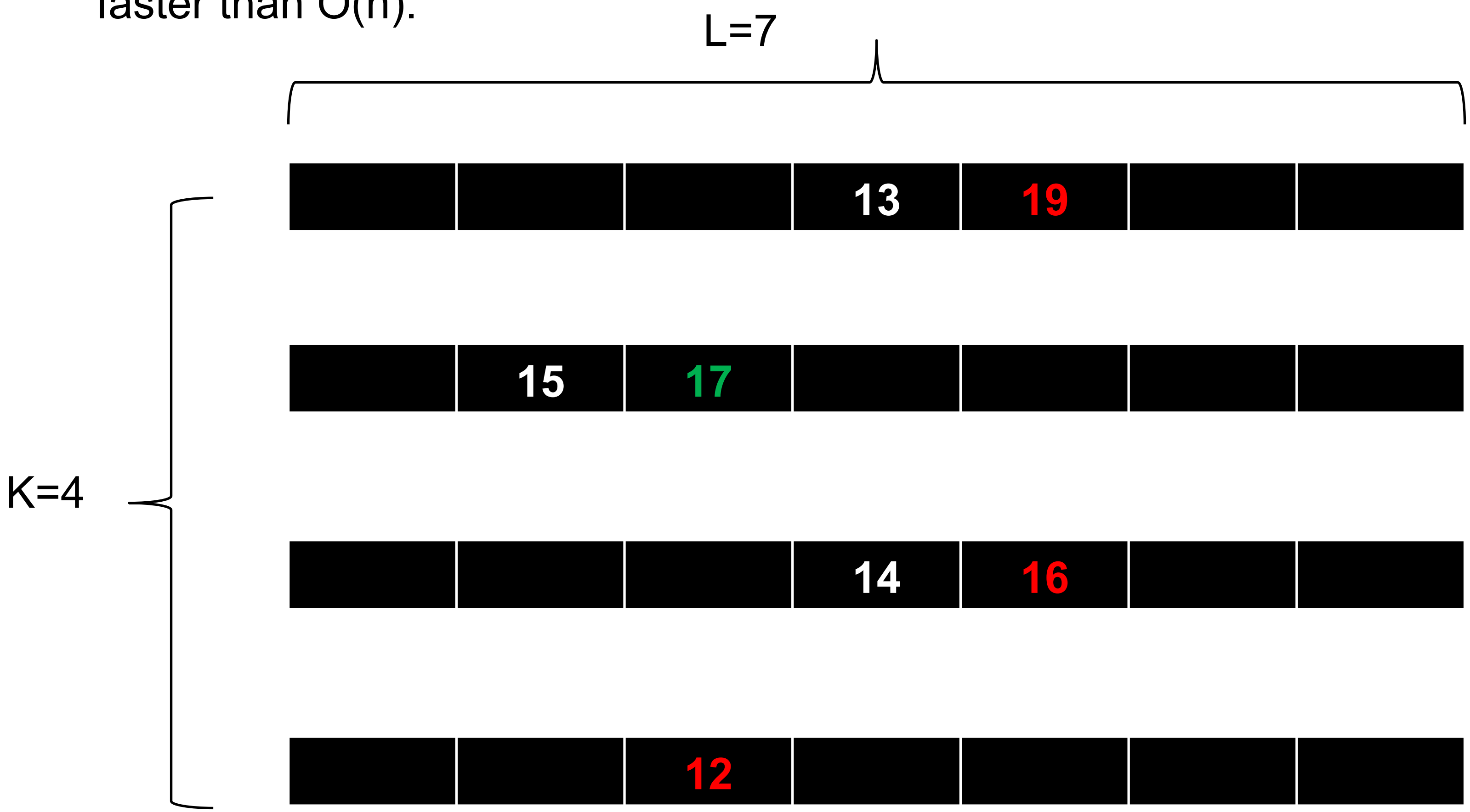
Find median for each

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



Find median of medians

Problem 5: Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the n = kl elements. Your algorithm should run asymptotically faster than O(n).



Count(Smaller) = 5 = Index-1
17 is desired median

Quiz