

COMP3711: Design and Analysis of Algorithms

Tutorial 5
Revision of March 11, 2019

HKUST

Question 1

Input is k sorted lists that need to be merged into one sorted list.

The “obvious” method is to modify the merging procedure from mergesort; at every step, compare the smallest items from each list and move the minimum one among them to the sorted list.

Finding the minimum value of k items requires $O(k)$ time so, if the lists contain n items in *total* the full k -way merge would take $O(nk)$ time.

This can be improved.

Design an $O(n \log k)$ -time algorithm to merge k sorted lists into one sorted list by making use of priority queues.

Note that each sorted list may contain a different number of elements.

Solution 1

The idea is to modify the obvious algorithm;
find the minimum at each step using **using a priority queue**.

Walk through the k lists, maintaining a pointer to the first (*smallest*) unmerged element in each list. At the start, the pointers point to the first element of each list. We'll call these items the *heads* of the lists.

1. Label each element to indicate to which sorted list it belongs. $//O(n)$
2. Build a min-heap of k list-heads. $//O(k \log k)$
3. Extract-Min from the heap and output item. $//O(\log k)$
4. If extracted element belongs to i -th sorted list, move the i -th pointer by one and insert the new head of the i -th list into the min-heap. $//O(\log k)$
5. Repeat 3 and 4 until all n elements have been outputted.

Step 3 and 4 will be repeated at most n times.

=> Algorithm's running time is $n + k \log k + 2n \log k = O(n \log k)$.

Question 2

The problem input is n/k lists:

- (i) Each list has size k , and
- (ii) for $i = 1$ to n/k , the elements in list $i - 1$ are all less than all the elements in list i

The obvious algorithm to fully sort these items is to sort each list separately and then concatenate the sorted lists together. This uses $\frac{n}{k} O(k \log k) = O(n \log k)$ comparisons.

Show that this is the best possible. That is, any comparison-based sorting algorithm to sort the n/k lists into one sorted list with n elements will need to make at least $\Omega(n \log k)$ comparisons.

Note that you can not derive this lower bound by simply combining the lower bounds for the individual lists.

Solution 2

First calculate the number of different possible output (orderings).

Let L_i be list i , for $i = 1, \dots, \frac{n}{k}$:

The final list is the concatenation $L = L_1 L_2 \cdots L_{n/k}$.

Each list L_i has $k!$ possible orderings.

Since elements in L_{i-1} are less than the elements in L_i , a final ordering can be any ordering of L_1 , followed by any ordering of L_2 , etc..

Then the *total* number of possible output orderings of L is

$$(k!)^{n/k}$$

Then, the decision tree for sorting these n elements contains at least $(k!)^{n/k}$ leaves.

Solution 2

Recall that a binary tree of height h has at most 2^h leaves.

Thus, in our problem, height h of the comparison tree must satisfy

$$2^h \geq (k!)^{n/k} \Rightarrow h \geq \log((k!)^{n/k})$$


Note that

$$\log((k!)^{n/k}) = \frac{n}{k} \cdot \log(k!)$$

$$= \frac{n}{k} \Theta(k \log k)$$

$$= \Theta(n \log k)$$

Stirling's
Approximation



Therefore, any comparison-based sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst-case for solving this problem.

So our simple algorithm was optimal.

Question 3

In this problem, you are given n integers to sort.

(a) You are told they are all in the range $[0, n^2 - 1]$.
How fast can you sort them?

(b) Now you are told they are all in the range $[0, n^t - 1]$ for some fixed t . How fast can you sort them?

Solution 3

(a) Recall that a number $m \in [0, n^2 - 1]$ can be uniquely represented in base n as a pair (m_1, m_0) where

$$m = m_1n + m_0 \quad \text{and} \quad 0 \leq m_0, m_1 \leq n - 1$$

This means that we can apply radix sort with

$$d = \text{\#digits} = 2$$

and

$$k = n \text{ where digits range from } 0 \text{ to } k - 1.$$

Radix sort will run in time

$$d(n + k) = O(n)$$

Solution 3: Example

$n = 9 \implies$ all numbers in $[0, n^2 - 1] = [0, 80]$

Example below illustrates radix sorting 9 #s in $[0, 80]$

$19 = 2 \cdot 9 + 1 = [2, 1]$	$[4, 0]$	$[0, 3]$	$= 3$
$7 = 0 \cdot 9 + 7 = [0, 7]$	$[2, 1]$	$[0, 7]$	$= 7$
$3 = 0 \cdot 9 + 3 = [0, 3]$	$[0, 3]$	$[2, 1]$	$= 19$
$69 = 7 \cdot 9 + 4 = [7, 4]$	$[6, 3]$	$[2, 3]$	$= 21$
$57 = 6 \cdot 9 + 3 = [6, 3]$	$[2, 3]$	$[3, 8]$	$= 35$
$35 = 3 \cdot 9 + 8 = [3, 8]$	$[7, 4]$	$[4, 0]$	$= 36$
$36 = 4 \cdot 9 + 0 = [4, 0]$	$[8, 6]$	$[6, 3]$	$= 57$
$78 = 8 \cdot 9 + 6 = [8, 6]$	$[0, 7]$	$[7, 4]$	$= 69$
$21 = 2 \cdot 9 + 3 = [2, 3]$	$[3, 8]$	$[8, 6]$	$= 78$

Solution 3

(b) If the numbers are in $[0, n^t - 1]$, solution is almost the same.

Note that if a number $m \in [0, n^t - 1]$, it can be uniquely represented in base n as a t -tuple $(m_{t-1}, \dots, m_1, m_0)$ where

$$m = \sum_{i=0}^{t-1} m_i n^i \quad \text{and} \quad \forall i, 0 \leq m_i \leq n - 1$$

This means that we can apply radix sort with

$$d = \# \text{ digits} = t$$

and

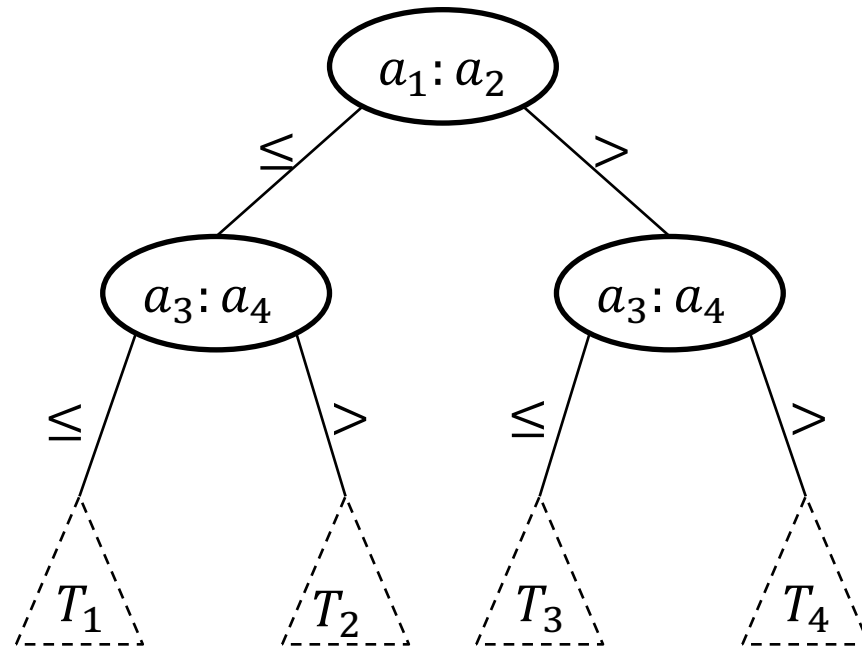
$$k = n \quad \text{where digits range from } 0 \text{ to } k - 1.$$

For *fixed* t , Radix sort will run in time

$$t(n + k) = O(n).$$

Question 4

The figure below shows part of the decision tree for mergesort operating on a list of 4 numbers, a_1, a_2, a_3, a_4 . Please expand subtree T_3 , i.e., show all the internal (comparison) nodes and leaves in subtree T_3 .



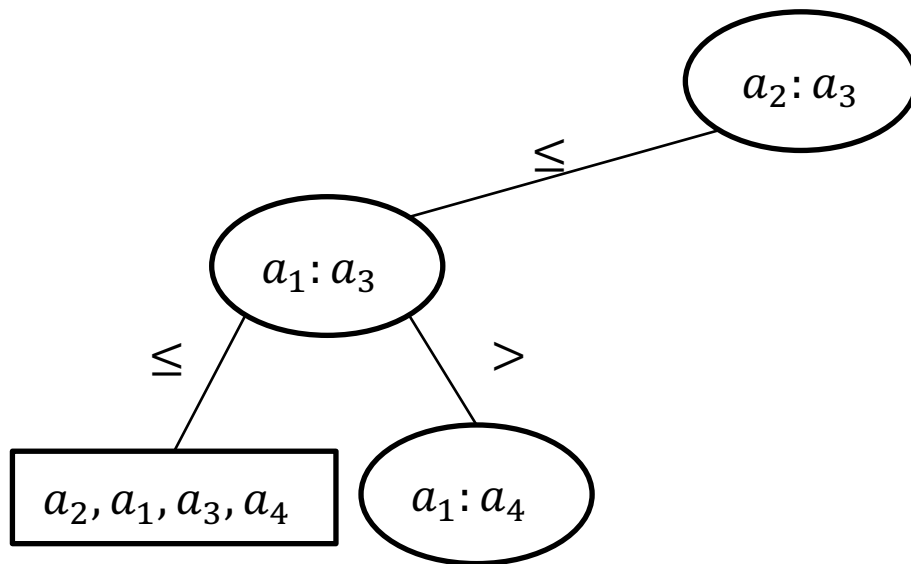
Solution 4

Mergesort on a_1, a_2, a_3, a_4 first starts by comparing a_1, a_2 and then comparing a_3, a_4 to get two sorted lists of size 2. It then merges those lists.

On previous page you were told that $a_2 < a_1$ and $a_3 \leq a_4$.

Algorithm now has two sorted lists a_2, a_1 and a_3, a_4 .

Tree T_3 is the decision tree that merges those two sorted lists.



If $a_2 \leq a_3$ then a_2 is smallest item and removed from list and a_1 is compared to a_3 .

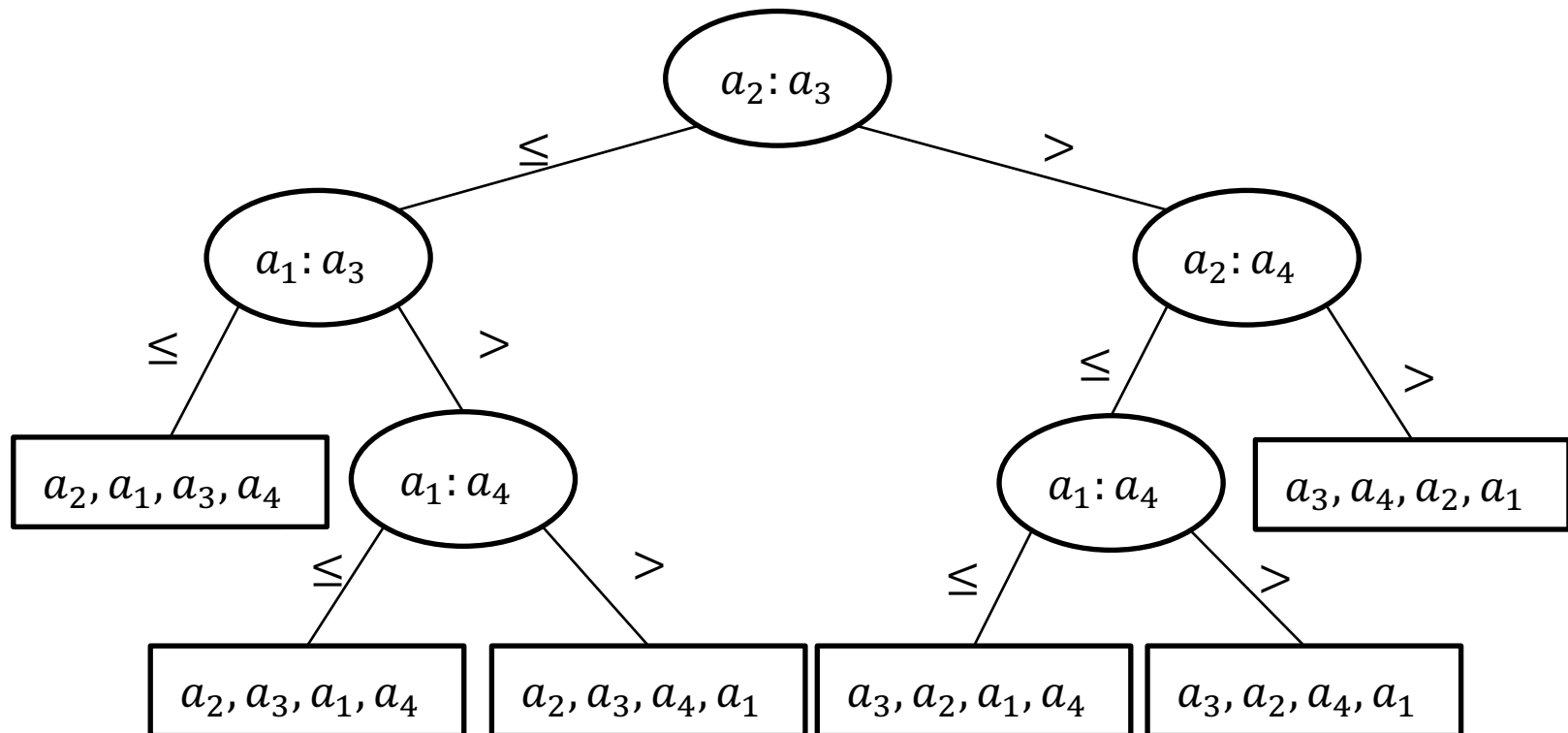
If $a_1 \leq a_3$ then a_1 is next smallest item and removed. Only a_3, a_4 remain and they are in same list.

If $a_1 > a_3$ then a_3 is next smallest item and removed. Only a_1, a_4 remain.

Solution 4

Mergesort on a_1, a_2, a_3, a_4 first starts by comparing a_1, a_2 and then comparing a_3, a_4 to get two sorted lists of size 2. It then merges those lists.

Continue this process to build Tree T_3 , the decision tree that merges the two sorted lists a_2, a_1 and a_3, a_4 .



Question 5

The goal of the interval partitioning problem (i.e., the classroom assignment problem) taught in lecture was to open as few classrooms as possible to accommodate all of the classes.

The greedy algorithm taught, sorted the classes by ***starting time***. It then ran through the classes one at a time; at each step it first checked if there was an available empty classroom. Only if there was no such classroom would it open a new classroom.

The algorithm looks simple but the proof is actually very dependent upon the classrooms being sorted by ***starting time***.

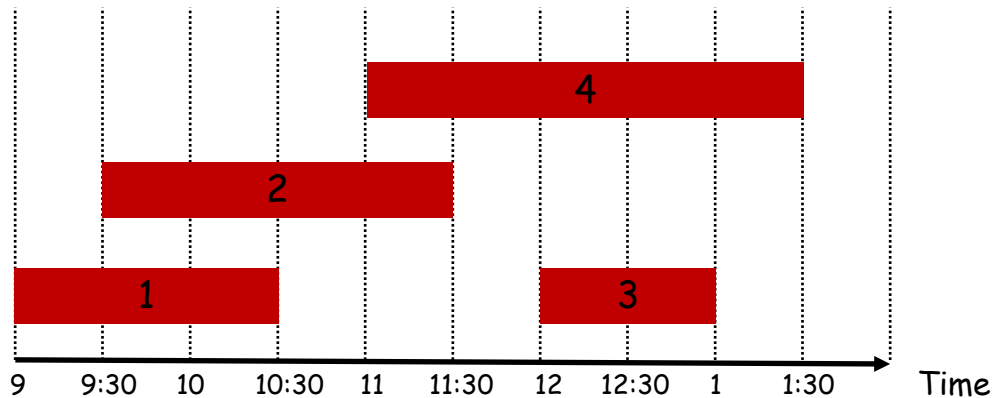
Prove that if the classes are sorted by ***finishing time*** the algorithm might not give a correct answer.

Note: Proving that something does not work usually means to find a counter-example.

Solution 5

```
Sort intervals by startingfinishing time so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .  
 $d \leftarrow 0$  // # classrooms used so far  
for  $j \leftarrow 1$  to  $n$   
    if lecture  $j$  is compatible with some classroom  $k$  then  
        schedule lecture  $j$  in classroom  $k$   
    else  
        allocate a new classroom  $d + 1$   
        schedule lecture  $j$  in classroom  $d + 1$   
         $d \leftarrow d + 1$ 
```

Algorithm
was modified
to use finish
time rather
than starting
time



The 4 intervals provide a counterexample.

The optimal solution is TWO but the greedy algorithm that sorts them by finishing time needs to use THREE classrooms.

Note that if the intervals were sorted by starting time, the algorithm WOULD use only two classrooms.

Question 6

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer and the coins have values $c_1 < c_2 < \dots < c_k$.

The *Greedy Algorithm* for change making is to use as many coins as possible of size c_k then as many as possible of size c_{k-1} etc.

The setup assumes that $c_1 = 1$, to ensure that change can always be made.

As an example, suppose that $(c_1, c_2, c_3, c_4) = (1, 5, 10, 25)$ and $n = 94$. Then the algorithm would find

3 coins of size 25

1 coin of size 10

1 coin of size 5

4 coins of size 1

Question 6 (continued)

- (a) Prove that the greedy algorithm for change making always uses the smallest number of possible coins if the coin set consists of quarters (25 cents) , dimes (10) , nickels (5), and pennies (1).
- (b) Suppose that the available coins are in denominations that are powers of c . i.e. the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
- (c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. As noted, your set should include a penny so that there is a solution for every value of n .

Solution 6a

(a) Prove that the greedy algorithm for change making always uses the smallest number of possible coins if the coin set consists of quarters (25 cents) , dimes (10) , nickels (5), and pennies (1).

Let $X = (x_1, x_5, x_{10}, x_{25})$ be an optimal change making solution and $X' = (x'_1, x'_5, x'_{10}, x'_{25})$ be the greedy solution. Then

$$x_1 + 5x_5 + 10x_{10} + 25x_{25} = n = x'_1 + 5x'_5 + 10x'_{10} + 25x'_{25}$$

Because X is optimal, $x_1 + x_5 + x_{10} + x_{25}$ is the smallest possible # of coins that can be used. We want to show that

$$x_1 + x_5 + x_{10} + x_{25} = x'_1 + x'_5 + x'_{10} + x'_{25}.$$

We will actually show the stronger fact that

$$\begin{aligned} x_1 &= x'_1, x_5 = x'_5 \\ x_{10} &= x'_{10}, x_{25} = x'_{25} \end{aligned}$$

Solution 6a (cont)

$$x_1 + 5x_5 + 10x_{10} + 25x_{25} = n = x'_1 + 5x'_5 + 10x'_{10} + 25x'_{25}$$

X minimizes number of coins used.

The first thing to notice is that $x_1 = (n \bmod 5)$.

- $x_1 < 5$ because if $x_1 > 5$ then replacing 5 pennies with a nickel uses a smaller number of coins, contradicting the optimality of X .
- $n - x_1 = 5x_5 + 10x_{10} + 25x_{25}$. Thus $n = x_1 \bmod 5$. Since $x_1 < 5$ this implies $x_1 = (n \bmod 5)$.

$x'_1 = (n \bmod 5)$ as well since the greedy algorithm will use as many nickels as possible before using pennies.

$$\Rightarrow x_1 = x'_1 = (n \bmod 5).$$

We can therefore rewrite the setup as

$$5x_5 + 10x_{10} + 25x_{25} = n - x_1 = 5x'_5 + 10x'_{10} + 25x'_{25}$$

and X minimizes the # of coins used to make change for $n - x_1$ where $x_1 = (n \bmod 5)$.

Solution 6a (cont)

$$5x_5 + 10x_{10} + 25x_{25} = n' = 5x'_5 + 10x'_{10} + 25x'_{25}$$

X minimizes the number of coins used to make change for $n' = n - x_1$ where $x_1 = (n \bmod 5)$. Next note

• $x_5 \leq 1$.

If $x_5 > 1$: replace two nickels with one dime by setting

$$x_5 = x_5 - 2 \text{ and } x_{10} = x_{10} + 1.$$

This uses fewer coins, contradicting minimality.

Solution 6a (cont)

$$5x_5 + 10x_{10} + 25x_{25} = n' = 5x'_5 + 10x'_{10} + 25x'_{25}$$

X minimizes the number of coins used to make change for $n' = n - x_1$ where $x_1 = (n \bmod 5)$. Next note

- $x_5 \leq 1$.

- $x_{10} \leq 2$.

If $x_{10} > 2$: replace three dimes with one quarter and one nickel by setting

$$x_{10} = x_{10} - 3, \quad x_5 = x_5 + 1, \quad x_{25} = x_{25} + 1,$$

again contradicting minimality

(because we have replaced 3 coins with 2 coins).

Solution 6a (cont)

$$5x_5 + 10x_{10} + 25x_{25} = n' = 5x'_5 + 10x'_{10} + 25x'_{25}$$

X minimizes the number of coins used to make change for $n' = n - x_1$ where $x_1 = (n \bmod 5)$. Next note

- $x_5 \leq 1$.
- $x_{10} \leq 2$.
- **If $x_{10} = 2$ then $x_5 = 0$.**

Otherwise, replace two dimes and one nickel by one quarter by setting

$x_{10} = x_{10} - 2, \quad x_5 = x_5 - 1, \quad x_{25} = x_{25} + 1,$
again contradicting minimality

(because we have replaced 3 coins with 1 coin).

Solution 6a (cont)

$$5x_5 + 10x_{10} + 25x_{25} = n' = 5x'_5 + 10x'_{10} + 25x'_{25}$$

where $n' = n - x_1$ where $x_1 = (n \bmod 5)$ and

$$x_5 \leq 1, \quad x_{10} \leq 2, \quad \text{and} \quad \text{if } x_{10} = 2 \text{ then } x_5 = 0$$

This only permits five options for (x_5, x_{10}, x_{25}) and we can check each of them to see that Greedy gives the same solution.

- $(x_5, x_{10}, x_{25}) = (0, 0, x_{25})$,
Greedy also sets $(x'_5, x'_{10}, x'_{25}) = (0, 0, x_{25}) \Rightarrow$ Greedy is also optimal
- $(x_5, x_{10}, x_{25}) = (1, 0, x_{25})$,
Greedy also sets $(x'_5, x'_{10}, x'_{25}) = (1, 0, x_{25}) \Rightarrow$ Greedy is also optimal
- $(x_5, x_{10}, x_{25}) = (0, 1, x_{25})$,
Greedy also sets $(x'_5, x'_{10}, x'_{25}) = (0, 1, x_{25}) \Rightarrow$ Greedy is also optimal
- $(x_5, x_{10}, x_{25}) = (0, 2, x_{25})$,
Greedy also sets $(x'_5, x'_{10}, x'_{25}) = (0, 2, x_{25}) \Rightarrow$ Greedy is also optimal
- $(x_5, x_{10}, x_{25}) = (1, 1, x_{25})$,
Greedy also sets $(x'_5, x'_{10}, x'_{25}) = (1, 1, x_{25}) \Rightarrow$ Greedy is also optimal

In all cases Greedy gives the same solution as optimal \Rightarrow Greedy is optimal.

Solution 6b

(b) Let a_i denote the number of c^i coins used in a solution for make change of n cents.

Recall that there is a unique way to write n in base c , i.e.,

$$n = \sum_i a_i c^i \text{ such that } \forall i, 0 \leq a_i < c.$$

In this notation, for every i , $\sum_{j < i} a_j c^j < c^i$.

\Rightarrow the greedy algorithm will choose EXACTLY a_i coins of size c^i .

Suppose there exists some n for which greedy is not optimal.

Let O be optimal solution which is not same as Greedy.

Let o_i be the number of coins of size c^i used by O .

Let j be the largest index such that $a_j \neq o_j$. Because the two solutions agree on the larger indices, $n' = \sum_{i \leq j} a_i c^i = \sum_{i \leq j} o_i c^i$.

Solution 6b (cont)

$$n' = \sum_{i \leq j} a_i c^i = \sum_{i \leq j} o_i c^i$$

where, (from prev page,) $a_i < c$.
Furthermore, $a_j \neq o_j$.

If $o_j > a_j$ then

$$\sum_{i < j} a_i c^i = n' - a_j c^j = (o_j - a_j) c^j + \sum_{i < j} o_i c^i \geq c^j$$

Then Greedy would then have taken more than a_j coins of size c^j , contradicting definition of a_j .

Thus $o_j \leq a_j$.

Solution 6b (cont)

$$n' = \sum_{i \leq j} a_i c^i = \sum_{i \leq j} o_i c^i \quad \text{where, (from prev page,) } a_i < c.$$

Furthermore, $a_j \neq o_j$.

We just saw that $o_j < a_j$ causes a contradiction. Now assume $o_j < a_j$.

$$\Rightarrow (*) \quad \sum_{i < j} o_i c^i = n' - o_j c^j = (a_j - o_j) c^j + \sum_{i < j} a_i c^i \geq c^j.$$

Recall that $\sum_{i < j} (c - 1) c^i = c^j - 1$.

Together with (*) this immediately implies there exists $i < j$ such that $o_i \geq c$.

\Rightarrow Take those o_i coins of size c^i ,
replace them with 1 coin of size c^{i+1} and $(o_i - c)$ coins of size c^i
This reduces the number of coins, also contradicting optimality of O .

Solution 6b (cont)

$$n' = \sum_{i \leq j} a_i c^i = \sum_{i \leq j} o_i c^i \quad \text{where, (from prev page,) } a_i < c.$$

Furthermore, $a_j \neq o_j$.

We just saw that $o_j < a_j$ causes a contradiction,
and that $o_j > a_j$ causes a contradiction.

Thus $o_j = a_j$.

But j was chosen to be the largest index such that $a_j \neq o_j$!

So, this contradicts the definition of j .

The only way this can happen is if no such j exists which means that

$$\forall i \quad o_i = a_i$$

\Rightarrow Greedy is optimal!

Solution 6c

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. As noted, your set should include a penny so that there is a solution for every value of n .

(c) Let 1, 4, 6 be the set of coin denominations.

Suppose we want to make change for $n = 8$ cents.

G) Greedy solution uses:

one 6-cent coin and two 1-cent coins, i.e. it uses **three coins**.

O) The optimal solution only uses **two 4-cent coins**.

So, Greedy is NOT always optimal

Solution 6

Comments on Change Making

The first two parts might have made you suspect that the greedy algorithm always works.

The last part showed you that it doesn't.

In fact, unless the coins

- i) have some very special general properties, such as in part (b), or
- ii) have very unique properties and you spend a lot of time proving special cases, as in part (a),

the greedy algorithm will usually *not* work.

There is a solution to the problem that does always work but it is not greedy.

It uses dynamic programming, which we will learn later in the class, and is much slower.

Problem 7

A *unit-length closed interval* on the real line is an interval $[x, x + 1]$.

Describe an $O(n)$ algorithm that, given input set

$$X = \{x_1, x_2, \dots, x_n\}$$

determines the smallest set of unit-length closed intervals that contains all of the given points.

Argue that your algorithm is correct. You should assume that

$$x_1 < x_2 < \dots < x_n$$



As an example the points above are given on a line and you are given the length of a 1-unit interval.

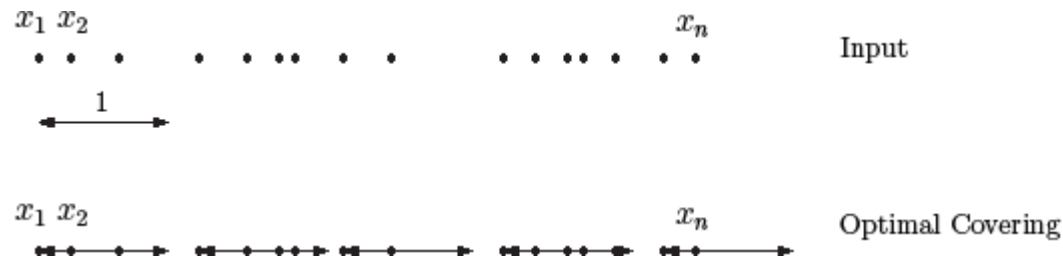
Show how to place a minimum number of such intervals to cover the points. 30

Solution 7

Keep the points in an array. Walk through the array as follows.

1. Set $x = x_1$
2. Walk through the points in increasing order until finding the first j such that $x_j > x + 1$
3. Output
4. If there was no such j in (2) then stop. Otherwise, set $x = x_j$
5. Go to Step (2)

Since each point is seen only once, this is an $O(n)$ algorithm.



The diagram above is the solution for the points on the previous page

Solution 7

Keep the points in an array. Walk through the array as follows.

1. Set $x = x_1$
2. Walk through the points in increasing order until finding the first j such that $x_j > x + 1$
3. Output
4. If there was no such j in (2) then stop. Otherwise, set $x = x_j$
5. Go to Step (2)

We must now prove correctness.

Let *Greedy*(i, k) be the algorithm run on the Array $[i \dots k]$.

Note that this can be rewritten

1. Output $[x_i, x_i + 1]$
2. Find min j such that $x_j > x_i + 1$
3. If such a j does not exist, stop, else return *Greedy*(j, k)

We will prove correctness by **induction on number of points $|X|$** .

Solution 7

1. Output $[x_i, x_i + 1]$
2. Find min j such that $x_j > x_i + 1$
3. If such a j does not exist, stop, else return *Greedy(j,k)*

We will prove correctness by induction on number of points $|X|$.

(i) If $|X| = 1$ (only one point) then algorithm is obviously correct.

Otherwise, suppose $|X| = n$ and that we know (induction hypotheses) that the algorithm **is correct for all problems** with size $|X| < n$.

(ii) $|X| = n$ and we know that the algorithm **is correct for all problems** with $|X| < n$.

Solution 7

1. Output $[x_i, x_i + 1]$
2. Find min j such that $x_j > x_i + 1$
3. If such a j does not exist, stop, else return *Greedy(j,k)*

Proving correctness by induction on number of points $|X|$.

(ii) $|X| = n$ and we know that the algorithm is correct for **all** problems with $|X| < n$.

$O(i, j) = \text{minimum \# of intervals needed to cover } \{x_i, \dots, x_j\}$

$G(i, j) = \text{minimum \# of intervals Greedy uses to cover } \{x_i, \dots, x_j\}$

Assume $[x_1, x_1 + 1]$ does not cover all of X because, if it did,
Greedy would return that same one interval solution which is optimal.

Let j' be the smallest index such that $x_{j'} > x_1 + 1$

Greedy returns $[x_1, x_1 + 1]$ concatenated with the greedy solution for $\{x_{j'}, \dots, x_n\}$

\Rightarrow Greedy uses $1 + O(j', n)$ intervals

By induction, Greedy solution for $\{x_{j'}, \dots, x_n\}$ is optimal.

Solution 7

1. Output $[x_i, x_i + 1]$
2. Find min j such that $x_j > x_i + 1$
3. If such a j does not exist, stop, else return *Greedy(j,k)*

$O(i, j)$ = minimum # of intervals needed to cover $\{x_i, \dots, x_j\}$

$G(i, j)$ = minimum # of intervals Greedy uses to cover $\{x_i, \dots, x_j\}$

\Rightarrow Greedy uses $1 + O(j', n)$ intervals

Now, suppose that there is a solution to OPT different than the Greedy one.

Let $[x, x + 1]$ be interval on OPT with the **leftmost** starting point.

$x \leq x_1$ because otherwise x_1 would not be covered by any interval in OPT.

Let k be the minimum index such that $x_k > x + 1$

After removing $[x, x + 1]$, remaining intervals in OPT must form optimal solution for $\{x_k, \dots, x_n\}$, otherwise we could build better solution using fewer intervals.

\Rightarrow Optimal uses $1 + O(k, n)$ intervals

Solution 7

1. Output $[x_i, x_i + 1]$
2. Find min j such that $x_j > x_i + 1$
3. If such a j does not exist, stop, else return *Greedy(j,k)*

$O(i, j)$ = minimum # of intervals needed to cover $\{x_i, \dots, x_j\}$

$G(i, j)$ = minimum # of intervals Greedy uses to cover $\{x_i, \dots, x_j\}$

Greedy uses $1 + O(j', n)$ intervals

Optimal uses $1 + O(k, n)$ intervals

Because $x \leq x_1$, $k \leq j'$ (Why?)

=> the optimal solution for $\{x_k, \dots, x_n\}$ is SOME solution for $\{x_{j'}, \dots, x_n\}$

=> $O(j', n) \leq O(k, n)$ (WHY?)

$$\begin{aligned} \Rightarrow \quad G(1, n) &= 1 + G(j', n) \\ &= 1 + O(j', n) && \longleftarrow \text{Induction hypothesis!} \\ &\leq 1 + O(k, n) \\ &= O(1, n) && \longleftarrow \text{Final Result!} \end{aligned}$$