CSE 421 Section 3

Greedy Algorithms

Administrivia

Announcements & Reminders

- HW1
  - If you feel something was graded incorrectly, submit a regrade request!
- HW2
  - Was due yesterday, 1/17
  - 5 late days total
- HW3
  - Due Wednesday, 1/24

How to Approach Writing an Algorithm

Writing an Algorithm

- Throughout this course, you will learn how to write several different kinds of algorithms.
  (Ex: greedy algorithms, more kinds to come)

- It can be difficult to understand what a problem actually needs you to do, which makes picking what kind of algorithm might solve the problem challenging as well!

- Today, we will work through a problem using an algorithm-writing strategy that you can apply to all the problems throughout this course and beyond

The Strategy

1. Read and Understand the Problem
2. Generate Examples
3. Come Up with a Baseline
4. Brainstorm and Analyze Possible Algorithms
5. Write an Algorithm
6. Show Your Algorithm is Correct
7. Optimize and Analyze the Run Time

1.1. Read and Understand the Problem

No really, we mean it!

- You can’t solve a problem if you don’t know what you’re supposed to do!
- Remember that problems are written in **mathematical English,** which is likely to be **much more dense** than, say, a paragraph from a novel. You’ll have to read more slowly (this advice still applies for our ridiculous word problems).
- As you’re reading, **underline anything you don’t understand**.
- **Rereading the problem** a few times can often help (it’s easier to understand details once you have the big picture in your brain).

Ask Yourself some Questions:

- Are there any **technical terms** in the problem you don't understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)
- Are there any words that look like normal words, but are **secretly technical terms** (like "subsequence" or "list")? These words sometimes subtly add restrictions to the problem and can be easily missed.

Problem 1 – Line Covering

Your new tow-truck company wants to be prepared to help along the highway during the next snowstorm. You have a list of n `doubles`, representing mile markers on the highway where you think it is likely someone will need a tow (entrances/exits, merges, rest stops, etc.). To ensure you can help quickly, you want to place your tow-trucks so one is at most 3 miles from **every** marked location. Find the locations which will allow you to place the minimal number of trucks while covering every marked location.

More formally, you will be given an array $A[]$, containing n `doubles` (in increasing order), representing the locations to cover.

Your task is to produce a list `sites` containing as few `doubles` as possible, such that for all i from 1 to n there is a j such that $|A[i] - \texttt{sites}[j]| \leq 3$.

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don’t understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)
- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

Work reading through this problem with the people around you and answering the four questions, and then we’ll go over it together!

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don't understand?

  "cover" means "place a truck at most 3 miles from"

- What is the **input type**? (Array? Graph? Integer? Something else?)

- What is your **return type**? (Integer? List?)

- Are there any words that look like normal words, but are **secretly technical terms** (like "subsequence" or "list")? These words sometimes subtly add restrictions to the problem and can be easily missed.

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don't understand?

  “cover” means “place a truck at most 3 miles from”

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `double[]`

- What is your **return type**? (Integer? List?)

- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don't understand?

  “cover” means “place a truck at most 3 miles from”

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `double[]`

- What is your **return type**? (Integer? List?)

  `double[]`

- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

Problem 1.1 – Line Covering

- Are there any **technical terms** in the problem you don’t understand?

  “cover” means “place a truck at most 3 miles from”

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `double[]`

- What is your **return type**? (Integer? List?)

  `double[]`

- Are there any words that look like normal words, but are **secretly technical terms** (like “subsequence” or “list”)? These words sometimes subtly add restrictions to the problem and can be easily missed.

  list

1.2. Generate Examples

Examples Help us Understand!

- You should generate two or three sample instances and the correct associated outputs.

- If you're working with others, these instances help make sure you've all interpreted the problem the same way.

- Your second goal is to get better intuition on the problem. You'll have to find the right answer to these instances. In doing so you might notice some patterns that will help you later

- *Note*: You should not think of these examples as debugging examples – null or the empty list is not a good example for this step. You can worry about edge cases at the end, once you have the main algorithm idea. You should be focused on the "normal" (not edge) case.

Problem 1.2 – Line Covering

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

Work through reading this problem with the people around you and answering the four questions, and then we'll go over it together!

Problem 1.2 – Line Covering

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

[0,1,2,3,4,5,6,7] is covered by [3,7]

[0, 6,7,8,9,13] is covered by [3,10]

[0,1,2,3, 100,101,102,103] is covered by [0,100]

1.3. Come Up with a Baseline

Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow**.
- We’re going to skip this step in this question, but you’ll see it in future examples and in lectures.

1.4. Brainstorm and Analyze Possible Algorithms

Think about Algorithm Possibilities

- It sometimes helps to ask, **“what kind of algorithm could I design?”**
(This week the answer is going to be “a greedy algorithm” because we’re learning about greedy algorithms)

- By the end of the quarter, you’ll have a list of possible techniques.

- Questions to help you pick:
  - Does this problem remind me of any algorithms from class? What technique did we use there?
  - Do I see a modeling opportunity (say a graph we could run an algorithm on, or a stable matching instance we could write)?
  - Is there a way to improve the baseline algorithm (if you have one) to something faster?

Remember: Your First Guess Might Not Be Right!

- You may want to **try multiple different algorithm paradigms** if you're not sure what might work best / be fastest

- Even if you're pretty sure of the algorithm type you want to use, it helps to **brainstorm multiple different possibilities**

- Then, we can test these possibilities, using the examples we came up with in part 2, to see which choice gives the correct output

Problem 1.4 – Line Covering

For this problem today, you should use a greedy algorithm.

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

Problem 1.4 – Line Covering

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Here are some ideas we thought of (you might have thought of others)

- Choose some `double` that will cover the most remaining elements of A.
- For $A[i]$ being the leftmost (remaining, uncovered) element, add $A[i] + 3$ to `sites` (the right-most spot which will cover the left-most element).
- Start with $A[1] + 3$, and thereafter if your previous point was p, choose $p + 6$
- Pick the left-most (remaining, uncovered) $A[i]$

Try your Algorithm Ideas on your Examples

- Now that you have some (hopefully) good ideas, you want to test them out on actual example inputs and see if you get the expected output

- Generate a few more instances and narrow down to just one possibility:
  - If you eliminate all of your ideas, go back to the last step and generate a new rule
  - If you can't eliminate down to one choice after multiple examples, try specifically to create an instance where they should behave differently
  - If you still can't make them behave differently, just pick one to try

- This will help you determine which algorithm to pursue!

Problem 1.4 – Line Covering

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Here are some ideas we thought of (you might have thought of others)

- Choose some `double` that will cover the most remaining elements of A.
  [0,6,7,8,9,13] eliminates this, this rule gives [7.5, 0, 13] (It has a [3,10] solution)
- For $A[i]$ being the leftmost (remaining, uncovered) element, add $A[i] + 3$ to `sites` (the right-most spot which will cover the left-most element).
  This one passes all our examples! That's a good indication this could be the right rule
- Start with $A[1] + 3$, and thereafter if your previous point was p, choose $p + 6$
  [0,1,2,3, 100,101,102,103] eliminates this, this rule gives [3, 9, 15, ..., 99, 105]
- Pick the left-most (remaining, uncovered) $A[i]$
  [0,6,7,8,9,13] eliminates this, this rule gives [0, 6, 13]

1.5. Write an Algorithm

Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode

- Some pseudocode tips:
  - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
  - You can have English phrases and sentences in your pseudocode!
  - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
  - You don't need to be very specific about data structures
  - You should (sparingly) add comments to help explain anything that may be confusing

Problem 1.5 – Line Covering

Greedy Rule: For $A[i]$ being the leftmost (remaining, uncovered) element, add $A[i] + 3$ to `sites` (the right-most spot which will cover the left-most element)

```
function Placements(A[1..n])
   uncoveredIndex ← 1
   sites ← empty list
   while uncoveredIndex ≤ n do
       newSite ← A[uncoveredIndex] + 3
       append newSite to sites.
       while uncoveredIndex ≤ n and |A[uncoveredIndex] - newSite| ≤ 3 do
           uncoveredIndex++
   return sites
```

1.6. Show Your Algorithm is Correct

Writing the Algo isn't Enough…
We Need to Prove that it Works!

- In general, you'll be often writing some kind of induction proof, or proving some implications to show that your algorithm is correct

- For greedy algorithms specifically, we have three common proof strategies:
  - greedy stays ahead
  - exchange arguments
  - structural arguments

Problem 1.6 – Line Covering

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we’ll go over it together!

Problem 1.6 – Line Covering

We prove the claim using the "greedy stays ahead" format.

Let OPT be an optimal solution (sorted in increasing order), and let ALG be the solution generated by our algorithm (similarly sorted).

We will show that for all i, the first i elements of ALG cover at least as many elements of A as the first i elements of OPT

We proceed by induction on i.

Base Case: $i = 1$
The left-most chosen site must cover the leftmost element of A. ALG chooses the rightmost point that still allows for $A[1]$ to be covered, so it must cover every element that OPT's first site does.

IH: Suppose that the claim holds for $i = 1, \ldots, k$.

Problem 1.6 – Line Covering

IS: Want to prove for $i = k + 1$:
Let $A[\ell]$ be the left-most element not covered by the first k entries of ALG.
By IH, $A[\ell]$ is also not covered by the first k entries of OPT.

Note that `uncoveredIndex` will be ℓ at this point of the algorithm: it begins at index 1 and increases to the left-most-uncovered index with the inner-most while-loop at each step.

Thus the next site we choose is $A[\ell] + 3$. This is the rightmost point that can still cover $A[\ell]$ (any further right point is too far away to cover it)

Since $A[\ell]$ is uncovered by OPT also, the next element of OPT cannot be greater than $A[\ell] + 3$ (if it were, by the sorted ordering, $A[\ell]$ would remain uncovered).

So, the first $k + 1$ elements of ALG also cover at least as many elements as the first $k + 1$ elements of OPT.

1.7. Optimize and Analyze the Run Time

Just Like Back in 332...

- Make your algorithm as efficient as possible.

- Flesh out any pseudocode you've written with enough detail to analyze the running time (do you need particular data structures?).

- Write and justify the big-O running time.
    - Can you make your code more efficient?
    - Can you give a reason why you shouldn't expect the code to be any faster?

Problem 1.7 – Line Covering

Write the big-O of your code and justify the running time with a few sentences.

Problem 1.7 – Line Covering

Write the big-O of your code and justify the running time with a few sentences.

The loop runs in $O(n)$ time.

It might look worse at first glance, but it's not! (Think of two markers moving down the highway.)

Every iteration of the inner loop increases `uncoveredIndex`, so those lines run at most n times total (across all iterations of the outer-loop).

Similarly, every line in the outer-loop executes at most n times, since every new site covers at least one element of A and therefore increases `uncoveredIndex`.

Each step can be implemented in $O(1)$ time, so the total time is $O(n)$.

2.1. Read and Understand the Problem

Problem 2 – Another Greedy Algorithm

You have a set, A , of (possibly overlapping) intervals. You wish to choose a subset B of the intervals to cover the full set. Here, cover means every interval X in A is contained in some interval Y in B.

Describe (and prove correct) an algorithm which gives you a cover with the fewest intervals.

Problem 2.1 – Another Greedy Algorithm

- Are there any **technical terms** in the problem you don’t understand?

- What is the **input type**? (Array? Graph? Integer? Something else?)

- What is your **return type**? (Integer? List?)

Work reading through this problem with the people around you and answering the questions, and then we’ll go over it together!

Problem 2.1 – Another Greedy Algorithm

- Are there any **technical terms** in the problem you don't understand?

  “cover” means that every interval in X is contained in some interval in B

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `A list of intervals`

- What is your **return type**? (Integer? List?)

  `A list of intervals`

2.2. Generate Examples

Problem 2.2 – Another Greedy Algorithm

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

{[0,1], [4,5], [7,8]} is covered by {[0,1], [4,5], [7,8]}

{[1,2], [3,4], [0,5]}is covered by [0,5]

{[1,7], [3,13], [2,12]} is covered by {[1,7], [3,13]}

2.3. Come Up with a Baseline

Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow**.
- What would be a baseline for this question?

Problem 2.3 – Another Greedy Algorithm

Baseline Algorithm (Exponential Time):

Iterate through all subsets of B and check if each subset covers the whole set. Keep track of the smallest such subset, and return it.

2.4. Brainstorm and Analyze Possible Algorithms

Problem 2.4 – Another Greedy Algorithm

For this problem today, you should use a greedy algorithm.

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

Problem 2.4 – Another Greedy Algorithm

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Here are some ideas we thought of (you might have thought of others)

- Choose the longest remaining interval.
- Choose the remaining interval that covers the most other intervals
- Choose the remaining interval with the smallest left endpoint, breaking ties by choosing the interval with the largest right endpoint.
- Pick the remaining interval intersecting the last chosen interval with largest right endpoint. If no intervals intersect the last chosen one (or this is the first chosen interval), instead pick amongst the intervals with smallest left endpoint.

2.5. Write an Algorithm

Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode

- Some pseudocode tips:
    - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
    - You can have English phrases and sentences in your pseudocode!
    - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
    - You don't need to be very specific about data structures
    - You should (sparingly) add comments to help explain anything that may be confusing

Problem 2.5 – Another Greedy Algorithm

Greedy Rule: Pick the remaining interval intersecting the last chosen interval with largest right endpoint. If no intervals intersect the last chosen one (or this is the first chosen interval), instead pick amongst the intervals with smallest left endpoint.

```
function Covering(A[])
   sort A by increasing start, breaking ties by decreasing end
   Let y be the start time of the first element of A
   covering ← empty list
   while A is not empty do
       Let I = [s,e] be the element in A with latest end time among those
           starting y or earlier
       add I to covering
       remove all elements of A with end time e or earlier
       if any remaining elements of A start before e
           y ← e
       else
           y ← the earliest start time of all remaining elements in A
   return covering
```

2.6. Show Your Algorithm is Correct

Problem 2.6 – Another Greedy Algorithm

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we'll go over it together!

Problem 2.6 – Another Greedy Algorithm

Let ALG be the list of intervals found by the algorithm, and let OPT be the list of intervals in an optimal cover. In both cases, let these lists be sorted by increasing start time. We claim the following:

Lemma 1. for all i, END(ALG(i)) ≥ END(OPT(i)).

Base Case:

Let t be the left-most point of any interval in X . To be valid covers, both ALG and OPT must cover t. In the first while loop, END(ALG(1)) is chosen as the right-most point in any interval containing t. Since OPT also covers t, in sorted order OPT(1) must be an interval covering t, thus END(ALG(1)) ≥ END(OPT(1)).

IH: Suppose END(ALG(i)) ≥ END(OPT(i)).

Problem 2.6 – Another Greedy Algorithm

IS: Let y be the left-most point in some interval in A not covered by any of ALG[1],ALG[2],…,ALG[i].

By IH, OPT[i] also does not cover y. Since OPT is a valid cover and sorted, OPT[i+1] must start before y. Now, consider the execution of the algorithm: when it added ALG[i+1], it deleted all elements that would not cover a new point, thus it considered only intervals containing y and chose the one with the latest end time. Thus either OPT[i+1] contains y, was considered in this choice and thus ends before ALG[i+1], or OPT[i+1] ends before y and thus certainly must end before ALG[i+1]. This proves the statement by induction.

With the Lemma proven, we argue that ALG is a minimum-sized cover. By construction, ALG covers every point in X. Letting ALG[k] be the last interval in ALG, we have END(ALG[k]) > END(ALG[k-1]) ≥ END(OPT[k-1]). So the first k-1 intervals in OPT do not cover END(ALG[k]). Thus there are at least k intervals in OPT, so OPT has at least as many intervals as ALG, and thus ALG is optimal.

2.7. Optimize and Analyze the Run Time

Problem 2.7 – Another Greedy Algorithm

Write the big-O of your code and justify the running time with a few sentences.

Problem 2.7 – Another Greedy Algorithm

Write the big-O of your code and justify the running time with a few sentences.

Note that apart from sorting, the whole algorithm, including the deletion step, can be performed with a simple iteration — intervals in A will overlap with I if and only if their start time is before I's end-time.

After A is sorted by start time, every element is either added to the covering or is deleted in O(1) time by the deletion command, so apart from sorting, the function takes O(n) time. Thus with the sort, the function takes O(nlog(n)) time.

3.1. Read and Understand the Problem

Problem 3 – Art Comissions

You’ve just started a new one-person art company. You’ve convinced n of your friends to each put \$c of their current money to supporting your dreams by commissioning you to make art. It takes you one month to finish a commissioned piece (you are only working in your limited free-time). One of your friends will pay you at the beginning of the month to make their artwork. While waiting for you to start their commission, your friends are going to place their money into bank accounts, which earn small (and varying!) rates of interest.

Friend i earns interest at the rate of ri , compounding monthly. I.e., the amount in their bank account is ri times what it was at the start of the last month (until they withdraw their money; ri > 1). Your friends generously decide to pay you both the principal and the interest earned at the time you start their commission. Describe an ordering to take the commissions that will maximize the amount you are paid (you may assume you know the ri for each of your friends).

Problem 2.1 – Another Greedy Algorithm

- Are there any **technical terms** in the problem you don’t understand?
- What is the **input type**? (Array? Graph? Integer? Something else?)
- What is your **return type**? (Integer? List?)

Work reading through this problem with the people around you and answering the questions, and then we’ll go over it together!

Problem 3.1 – Art Comissions

- Are there any **technical terms** in the problem you don't understand?

  N/A

- What is the **input type**? (Array? Graph? Integer? Something else?)

  `C, and a list R[1..n] of interest rates`

- What is your **return type**? (Integer? List?)

  `An ordering of the friends`

3.2. Generate Examples

Problem 3.2 – Art Comissions

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

Problem 3.2 – Art Comissions

Generate two examples with their associated outputs. Put some effort into these! The more different from each other they are, the more likely you are to catch mistakes later.

If the interest rates are [1.1,2.2,2.5], the friends should be in the order [3,2,1].

If the interest rates are [3,2,6], the friends should be in the order [3,1,2].

3.3. Come Up with a Baseline

Inefficient but Effective First Attempt

- In a time-constrained setting (like a **technical interview** or an **exam**) you often want a “baseline” algorithm.
- This should be an algorithm that you can implement and will give you the right answer, **even if it might be slow**.
- What would be a baseline for this question?

Problem 3.3 – Art Comissions

Baseline Algorithm (Exponential Time):

Try all orderings, compute how much each ordering earns, and return the one that earns the most.

3.4. Brainstorm and Analyze Possible Algorithms

Problem 3.4 – Art Comissions

For this problem today, you should use a greedy algorithm.

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Think of some greedy ideas with the people around you, and then we'll go over it together!

Problem 3.4 – Art Comissions

Come up with at least two greedy ideas (which may or may not work). Run each of your ideas through the examples you generated in part 2.

Here are some ideas we thought of (you might have thought of others)

- Largest interest rate first
- Smallest interest rate first

3.5. Write an Algorithm

Write That Pseudocode!

- Now that you've determined which of your ideas is (probably) correct, you need to formalize your algorithm into pseudocode

- Some pseudocode tips:
  - Pseudocode should be somewhere **between** a paragraph and actual (java/python/c/whatever) code
  - You can have English phrases and sentences in your pseudocode!
  - It can be helpful to name the variables you want to use in a paragraph before your actual pseudocode
  - You don't need to be very specific about data structures
  - You should (sparingly) add comments to help explain anything that may be confusing

Problem 3.5 – Art Comissions

```
function Commisions(c, R[1..n])
    ordering ← list of friends, represented as indices from 1 to n, sorted
        descending by interest rate
    return ordering
```

3.6. Show Your Algorithm is Correct

Problem 3.6 – Art Comissions

Write a proof of correctness.

Try to use one of those greedy proof techniques with the people around you, and then we'll go over it together!

Problem 3.6 – Art Comissions

Let OPT be an optimal solution, and ALG be the solution we have. Suppose, for the sake of contradiction, that OPT and ALG are different from each other.

Since ALG keeps elements in sorted order, it must be that OPT is not in sorted order. Then there will be a consecutive pair of elements in OPT; call them j, j + 1 where r_j > r_{j+1} . We will show that swapping these elements increases our earnings. In OPT, we get $cr_j^j + cr_{j+1}^{j+1}$ from j and j + 1. Now suppose we swap just j and j + 1. We will instead earn $cr_j^{j+1} + cr_{j+1}^j$. Observe that with this swap we have another ordering, and have not affected the amount earned for any other commission. Thus if $cr_j^j + cr_{j+1}^{j+1} > cr_j^{j+1} + cr_{j+1}^j$, we will have found a better option than OPT, giving our desired contradiction.

Problem 3.6 – Art Comissions

To show the desired inequality, we begin with the observation that:

$$r_j^j(r_j - 1) > r_{j+1}^j(r_{j+1} - 1)$$

which follows from $r_j > r_{j+1}$ (that this pair is not in sorted order) and that all terms in the expression are positive (since the rates themselves are greater than 1).
Distributing, we have

$$r_j^{j+1} - r_j^j > r_{j+1}^{j+1} - r_{j+1}^j$$

Rearranging, so everything is positive we have

$$r_j^{j+1} + r_{j+1}^j > r_{j+1}^{j+1} + r_j^j$$

Multiplying by c and reordering the terms, we have

$$cr_{j+1}^j + cr_j^{j+1} > cr_j^j + cr_{j+1}^{j+1}$$

as desired. We thus have that swapping will produce a better ordering than OPT, a contradiction

3.7. Optimize and Analyze the Run Time

Problem 3.7 – Art Comissions

Write the big-O of your code and justify the running time with a few sentences.

Problem 3.7 – Another Greedy Algorithm

Write the big-O of your code and justify the running time with a few sentences.

The entire algorithm is a sort, which runs in O(nlog(n)) time.

That’s All, Folks!

Thanks for coming to section this week!
Any questions?