Introductory Analysis, Design, and Justification

Ruben Acuña

Spring 2020

Learning Objectives

The final goal:

- ADJ-CO1: Understand problem solving as a learning and growth process.
- ADJ-CO2: Demonstrate an ability to construct a sound and evidence-based solution to a problem.
- ADJ-CO3: Demonstrate the ability to produce solutions to open-ended problems without a specific correct solution.
- ADJ-CO4: Demonstrate the ability to produce solutions to ill-defined problems with problems statements which interfere with problem solution.
- ADJ-CO5: Demonstrate the ability to construct solutions based on contextual needs of a problem, team, or customer.

A Typology of Problems

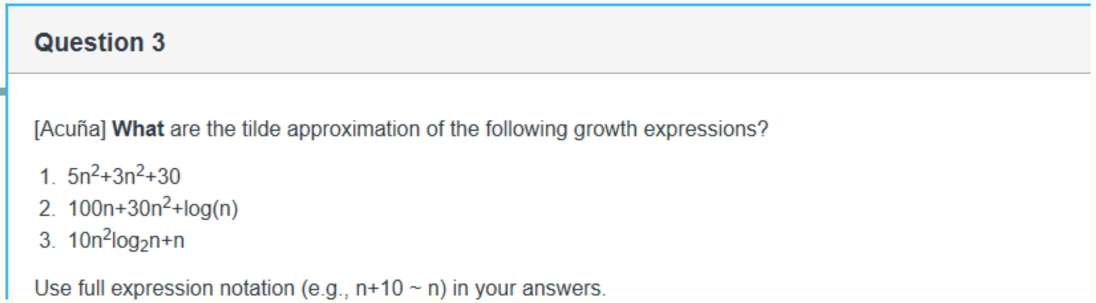
Real Problems

So here’s a thought: how are real problems different than what we see in a class at school?

Educational Problems

- Here are a couple of answers of examples from SER222 problems: one short answer, and one programming.

- Questions:
  - What do you know immediately about these problems?
  - What the steps in solving it?
  - If someone handed you a "solution", could you check it and make sure they are right?

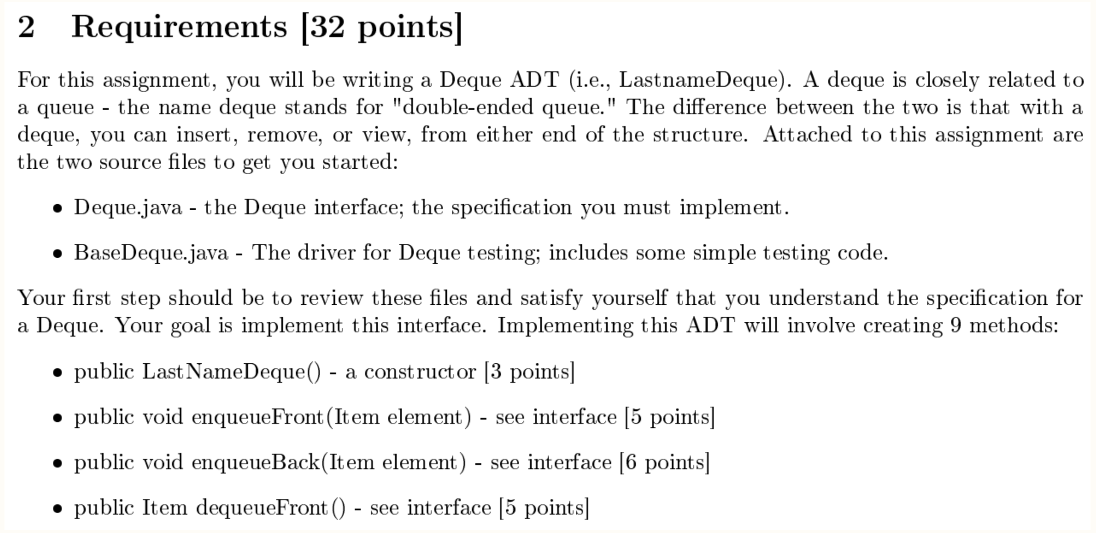


Question 3

[Acuña] **What** are the tilde approximation of the following growth expressions?

1. $5n^2+3n^2+30$
2. $100n+30n^2+\log(n)$
3. $10n^2\log_2 n+n$

Use full expression notation (e.g., n+10 ~ n) in your answers.



2 Requirements [32 points]

For this assignment, you will be writing a Deque ADT (i.e., LastnameDeque). A deque is closely related to a queue - the name deque stands for "double-ended queue." The difference between the two is that with a deque, you can insert, remove, or view, from either end of the structure. Attached to this assignment are the two source files to get you started:

- Deque.java - the Deque interface; the specification you must implement.
- BaseDeque.java - The driver for Deque testing; includes some simple testing code.

Your first step should be to review these files and satisfy yourself that you understand the specification for a Deque. Your goal is implement this interface. Implementing this ADT will involve creating 9 methods:

- public LastNameDeque() - a constructor [3 points]
- public void enqueueFront(Item element) - see interface [5 points]
- public void enqueueBack(Item element) - see interface [6 points]
- public Item dequeueFront() - see interface [5 points]

Real-world Problems

“Just meet the course outcomes. Teach it like CSE310.”

A *real world problem* is typically characterized by being *open-ended* and *ill-defined*.

Same Questions:

- What do you know immediately about this problem?
- What are the steps in solving it?
- If someone handed you a “solution”, could you check it and make sure they are right?

Problem Descriptions

In general problem descriptions fall into one of three categories

- ***Structured:*** the structure behind the elements of the problem, and the solution is provided.
- ***Semi-structured:*** there is a structure behind the problem, but it is not made explicitly clear.
- ***Unstructured:*** there is little or no structure behind the problem, and there is no clear way to distinguish a “solution”.

- Which of these is the easiest to solve? Be careful!
- What are the specific differences in *information* between these?

What is the Tilde approximation of 5n^2+1?

What is the Tilde approximation of the following snippet?

```
void test() {
  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++) {
      System.out.print("hello");
    }
  System.out.print("hello");
}
```

How much time does it take for a program to print "hellohello", "hellohellohellohellohello ", "hellohellohellohellohellohellohellohellohellohello“, etc?

Open Ended

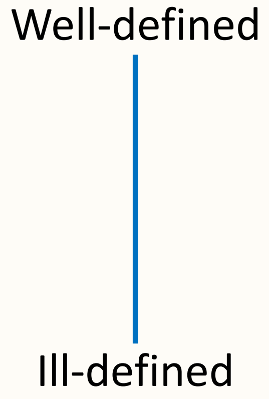
Close-ended —— Open-ended

- **Closed-ended Problem:** An answer must look a very specific way (e.g., only one answer), and/or be achieved in a limited number of ways.
- **Opened-ended Problem:** An answer has no specific form that limits that it, rather there are some constrains that define what it means to be an answer.

- Example 1: What is the Tilde approximation of 5n^2+1?
- Example 2: Write a program to take the sum of the first n positive integers.
- Example 3: Consider designing a program where you need to store a fixed size collection of items. Would you use an array or linked list?

Open Ended: Design Problems

- Unfortunately for us, the types of problems that engineers are asked to solve are often design problems.
- A ***design problem*** has the flavor of picking one solution among many options, and where the “best” solution may not be immediately apparent.
  - An example of a design problem: converting the idea for this sentence into a written sentence.
- In this case, the open-endedness comes from the fact that some attributes of the solution are not predetermined. For example: the programming language you use, or even the statements on a particular line.
- It may also be the case that you’re allowed to solve a problem differently: you can write code or just find an existing library that does it for you.



Ill-defined

– ***Well-defined:*** We know what we start with, we have options/ideas for what can be done, and we'll know the solution when we see it. (We say a solution is *verifiable*, perhaps with some additional information call a certificate.)
– ***Ill-defined:*** The basic elements of the problem, or the actions that can be taken, or what a solution looks like may be unclear.

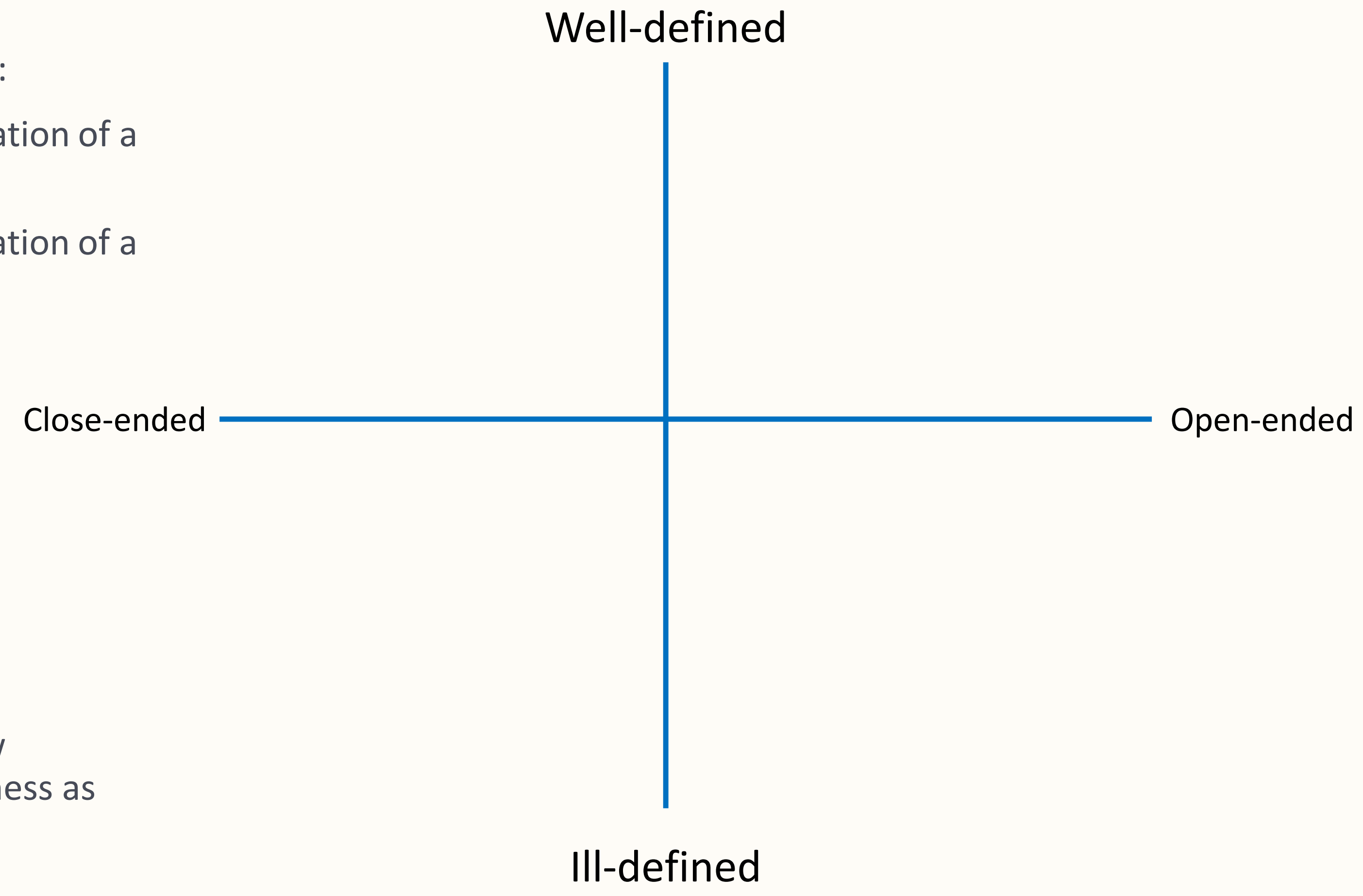
Related ideas: *underspecified problems*, *ill-structured problems* and *wicked problems*.

Practicing Classification

Let’s try plotting a few of these:

- Finding the tilde approximation of a function?
- Finding the tilde approximation of a method?
- You choose!
- You choose!
- Curing COVID-19?

- Question: is correct to view “defined”ness and “open”ness as orthogonal?



Ill-defined: Ambiguity

- Oftentimes parts of problems are ambiguous – both in the sense of the description and the problem itself.
- This could mean:
  - Something is unspecified/unknown.
  - Something is underspecified.
  - Something cannot be known.
- What are examples of all of these?
- This gives us all sorts of trouble – how can we have any confidence in our solution if it is not *well founded*?

Ill-defined: Faulty

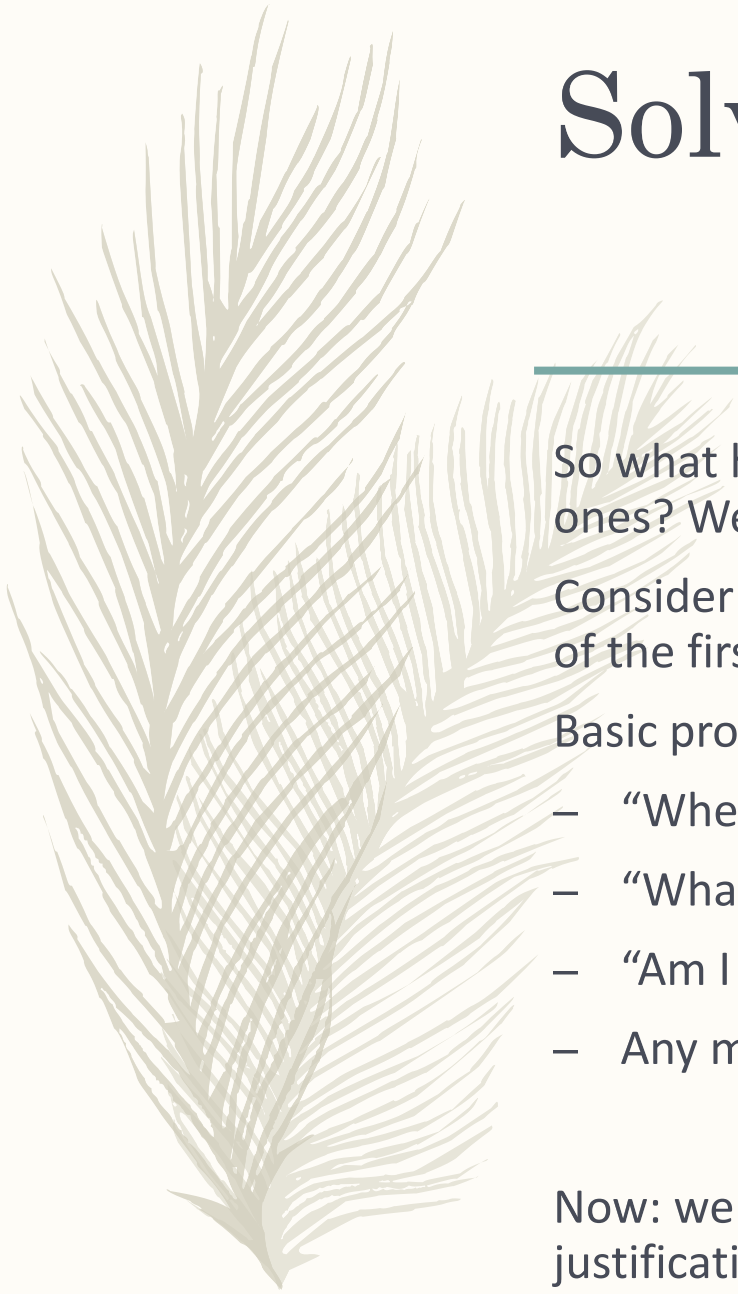
A **faulty** problem is one where the prompt is misleading, contradictory, or simply incorrect.

For the following examples, are they fine/ambiguous/faulty?

- Example 1: Store a list of contacts such that we can quickly access them by name as well as ID number.
- Example 2 : Consider designing a program where you need to store a fixed size collection of items. What data structure would you use?
- Example 3 : “Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.” (from Wikipedia)
- Example 4: Efficiently check if an image is black and white using a neural network.

Solving Real-World Problems

- Thinking about homework in your SER classes for a second, what do your answers look like?
- To contrast: how are real worlds solutions different?
  - Further: are real world problems graded…?
- To be a little more specific: in homework we often jump straight to a ***design***: that is, a proposed solution that we have generated.
- Really, we are missing two extra steps: analysis, and justification.
  - ***Analysis***: sitting down and explicitly dealing with all the muddling of the problem and its description.
  - ***Justification***: arguing and supporting our proposed solution against alternatives.
    - A thought: <u>we want to be right</u>. All the time if possible, and before anyone else.



Solving Real-World Problems

So what happens when people go from solving educational problems to real-world ones? Well, it’s a lot of trouble.

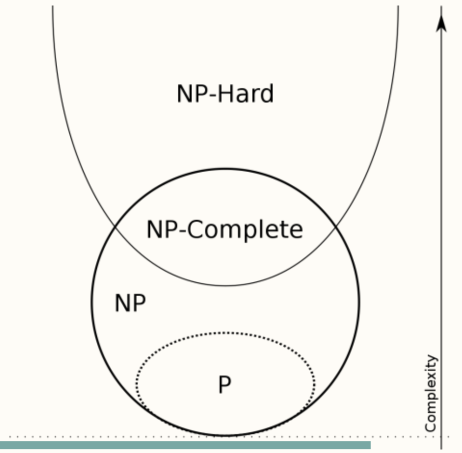
Consider for second: if you ask someone about the transition from school to work, one of the first things you’ll here is: “I learned a lot!”

Basic problems:

- “Where do I start?”
- “What do I do next?”
- “Am I done?”
- Any more? Perhaps in a wider sense?

Now: we can’t solve this all at once, but paying more attention to analysis and justification will help us greatly.

Problem Hardness



- Before we go farther, we should take a minute to talk about problem hardness.
- In our coverage of analysis of algorithms and Big-Oh, we talked about algorithm run-time: some are slow, some are fast.
- In fact there is a sense that some are so slow (and cannot be made faster) that a problem is hard or impossible.
  - Recall: people tend to give up if a problem takes an exponential amount of time in best case because no computer will give a result in a reasonable amount of time.
  - The common language here is to talk about problem *hardness*.
- How exactly does this relate our ideas here on a difficult problem typically being open-ended and ill-defined?

Image from https://en.wikipedia.org/wiki/NP-completeness

Introducing ADJ: Analysis, Design, and Justification

ADJ

We now define the ADJ process as following these three stages in sequence:

- **Analysis (A)**: a translation of the original ill-defined problem into concrete requirements that can be evaluated.
- **Design (D)**: a solution to the problem being solved.
- **Justification (J)**: an argument which supports your design as being superior to other potential designs which also satisfy the analysis requirements.

- We wish to show that a problem under analysis (A), soundly yields a design (D), such that the design is the/an optimal solution to the problem. The justification (J) is a sound argument that the design is the optimal solution for the problem given the analysis.
- The plan: we'll take a look at a simple problem and then step through each of the stages. For each stage, we'll talk about some useful (but minimal) techniques.

Example

- As the simplest possible example, let’s take the following problem: “Consider designing a program where you need to store a fixed size collection of items and display them. Would you use an array or linked list? Justify your answer.”
  - More educationally, we might attach “Analyze the problem, design a choice, and justify the choice.“ to the end of it.
- For now, we’ll assume that the answer would be given as a write up.
- Here’s a quick answer for it: “Linked list, because it will allow you to add or remove elements quickly.” Any thoughts on this answer as good/bad?

Soundness

Proof?

- Soundness means that the parts of our argument build upon each logically.
  - Every piece of text must be linked conceptual to the text, otherwise it will introduce inconsistencies and lose soundness.
- The follow must be emphasized: **Answers must follow from problem statement.**
- *Consider designing a program where you need to store a fixed size collection of items and displaying them. Would you use an array or linked list? Justify your answer.* Consider some more sample answers:
  - “Linked list, because it will allow you to add or remove elements quickly.” Is this sound?
  - “Array, because it will allow you to quickly access each of the elements.” Is this sound?
  - “Linked list, because it will allow you to quickly access each of the elements.” Is this sound?
  - “Array, because it will allow you to quickly display each entry.” Is this sound?
  - “Linked list, because it will allow you to quickly display each entry.” Is this sound?
- Random thought: should an answer have more or less raw text in it?

Process & Patterns

- When doing engineering, we would expect to see some general flavor of process. Often people indicate process to explain how SE is different than CS.
  - There is another aspect as well: control. People don't like using this word (it sounds harsh), but it think it captures a difference between science and engineering.

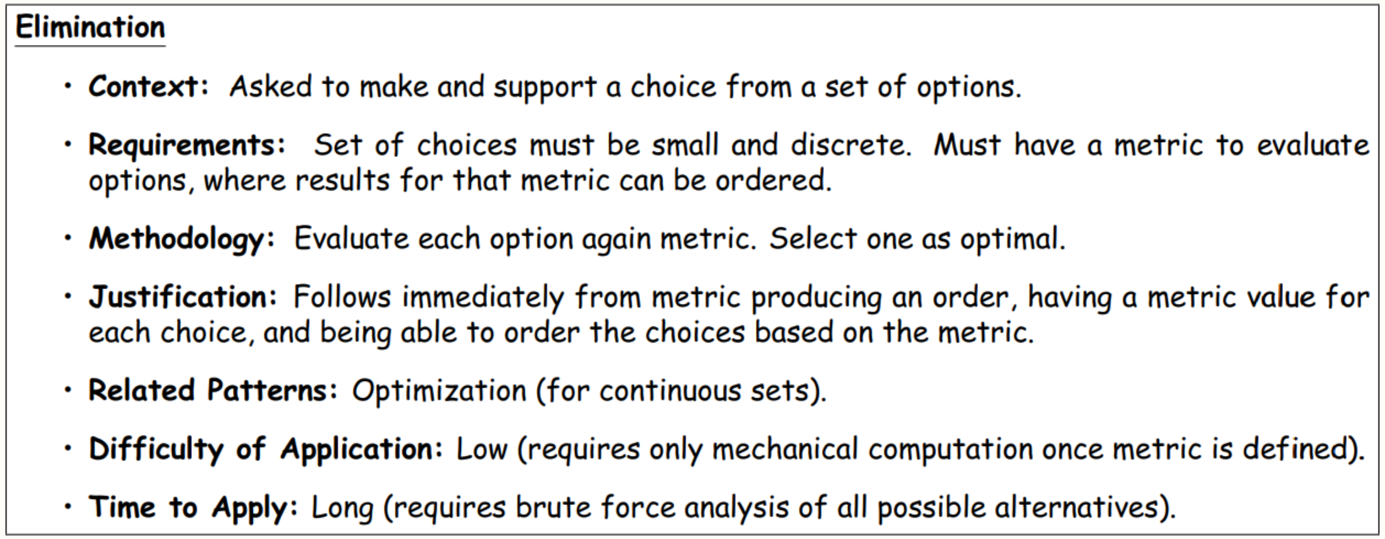
  - Are we using a process here?
  - Why is using process good?
  - Why is using process bad?
  - What would justify the use of process for problem solving?

Process & Patterns

– Another central idea in engineering is the use of *patterns*.

– Most likely, you have already heard the term design pattern. This refers to a reusable software solution to a common problem. The goal is both save time and increase quality.

– Although we typically don't talk about patterns in problem solving, they are certain there. The following is a (simple) technique:



Elimination

- **Context:** Asked to make and support a choice from a set of options.
- **Requirements:** Set of choices must be small and discrete. Must have a metric to evaluate options, where results for that metric can be ordered.
- **Methodology:** Evaluate each option again metric. Select one as optimal.
- **Justification:** Follows immediately from metric producing an order, having a metric value for each choice, and being able to order the choices based on the metric.
- **Related Patterns:** Optimization (for continuous sets).
- **Difficulty of Application:** Low (requires only mechanical computation once metric is defined).
- **Time to Apply:** Long (requires brute force analysis of all possible alternatives).

Analysis

Overview

Spoiler: this will be the most complicated section of ADJ that we talk about.

- Why?

- In Analysis, you should define your immediate problem, and any useful corollaries.
- The answer should be in terms of the problem itself, and not be biased in any towards a possible solution.
- **It is possible that some portions of your analysis will not be needed later** - this is expected since otherwise we do not have multiple inputs to a design process, meaning there would be only one choice.

Rough Steps:

1. Identify and address ambiguities in problem statement.
2. Generate definitions for ill-specified parts of the problem, along with useful corollaries.
3. Using the definitions propose a more well-specified view of the problem.
4. Define metrics (more to come!)

Definitions

- Almost always there are ambiguities in your problems – fix them.
- Treat definitions as your guide posts. They tell you what is real.
  - If you find yourself asking: “Am I allow to do that?”, then look to the definitions to guide you.

- In general, this means we need to find the ambiguities, and then define them. For example:
  - if a problem says “display”, we might want to define more specifically “print each element in the collection exactly once in any order”.
  - If a problem says “efficiently”, we might want to define it as “run in a Big-Oh polynomial time on the number of operations”.

Fixing Ill-defined Problems

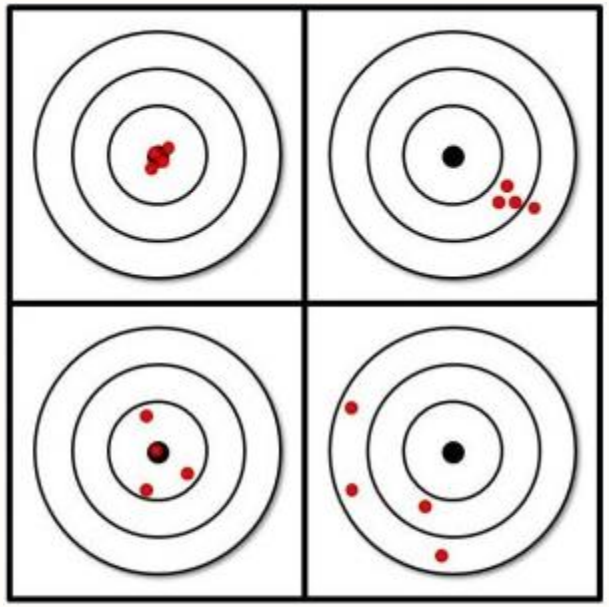
When faced with an ill-defined problem, there are three things we can think about:

- Make **reasonable** assumptions.
- Ask questions!
- **<u>Experiment:</u>** experimentation is a key tool with dealing with the unknown.
  - We want to take the scientific approach of evidence based discovery.
  - Have we learned any experimental methods in this class?
    - *Are there any others you have seen that might be useful?*

Take the knowledge you gain from these, and codify them in definitions which you can then build on.

Metrics and Measurements

- Before we attempt to do any design work, we need to know what makes a solution good – we need to define metrics.
  - A metric will be a way for us to measure different solutions to the problem and compare them.
- Ideally, we would want:
  - Metrics that are useful relative to our problem statement.
  - Metrics that are reliable: high in accuracy and precision.
  - More practically: efficient metrics
- Have we learned any metrics in this class?
  - Are there any others you have seen that might be useful?



http://glsi.agron.iastate.edu/2015/01/18/accuracy-vs-precision/

Design

Overview

- This section should build on the results of the analysis approach to construct a solution which meets a metric (defined in analysis) by which a design may be judged to provide a good answer.
  - Design is subjective - in and of itself, we cannot state that a particular design is best without a metric.

- The design can look many different ways
  - For the simplest problems, it might be just be a selection: “array” or “list”.
    - For our sample problem, a design looks like “array”.
  - For some problems, design will take the form of designing an explanation (later backed with an argument which states a position on the validity of the “claim”).
  - In others, it will literally mean designing an algorithm.

Justification

Overview

- Given that we have a design, we want to argue that this design is:
  a) well-founded, sound, and solves the problem, and
  b) is the optimal design for the problem.
- A justification is an easily verifiable piece of information (here: readable by a 3rd party) that shows that the claim for the design is correct.

- Rough Steps:
  1. Evaluate your metrics, and show that they either are optimal or hold.
     - *Sometimes they need to be evaluated only for your solution and sometimes for alternatives as well.*
  2. Done!*

* Provided that your analysis itself is sound.

Algorithmic Analysis, Design, and Justification

ADJ for Algorithms

- For our next topic, we'll look at how to solve algorithm design problems within the ADJ framework.
- Fundamentally, the difference in using ADJ for algorithms is that the result of ADJ will include pseudocode in the design portion. This is huge!
- It's a big deal because it means we have a large "search space" of possible answers we could invent. Maybe that makes our life easier – we're not looking for a needle in a haystack, but it also makes life harder: when justifying our answer, we're competing with an infinite number of possible choices.

A few notes about the arrangement of these slides:

- Each slide is labeled in the top right with its the corresponding ADJ stage.
- Specific techniques for algorithm analysis, design, and justification will be taught JIT style. JIT stands for Just-In-Time style, meaning we'll only bring something up when it's important.

Justification for Algorithms

- Although we’re not at the step of justifying our solution, we should start thinking about justification since it takes a long time and will have a different flavor than previous examples.
- In previous problems, we had statements like “select the data structure”. This meant the result was either “list” or “array”. Showing one was better only required looking at one alternative.
- In algorithms we have an infinite number of competing solutions. This is addressed by that rather than being *the best*, a solution is *at least as good as the best solution*. The view is that we are defining solution by the constraints it must meet, rather than picking a specific one.
  - For functionality, we say that a solution should have feature X, therefore any algorithm that supports X is at least as good as the best option.
  - For performance, things are trickier. An efficient algorithm is one that runs in polynomial time but often it is implied that an algorithm be as fast as possible. That requires writing a lower bound proof.

Updating a PQ: Analysis

ADJ Outline

So what’s our plan?

- Analysis:
  - Need to define the problem:
    - *What is meant by updating a PQ? (and heap – how is data represented).*
    - *Are any (reasonable) assumptions needed?*
  - Need to define metrics to define what makes a good solution.
- Design:
  - Need to create a pseudocode algorithm for the problem.
- Justification:
  - Need to evaluate our metrics from analysis on the design that is produced.

Problem Statement

To get started, let’s work a sample design problem:

> *Design an efficient algorithm to update the priority of an entry inside of a priority queue. Analyze the problem, design an algorithm for updating the priority queue, and justify the algorithm's optimality.*

Thoughts…

- How might ADJ be different for an algorithm than other examples we’ve seen?
  - What will our result for Design look like?
- Is this problem possible? (Or does it appear to be in anyway malformed…?)
- Can this problem (or a portion of it) be solved by something we know about already?
- Are any aspects of this prompt underdefined?
- What should be our first step?

Understanding the Problem

– *Design an efficient algorithm to update the priority of an entry inside of a priority queue. Analyze the problem, design an algorithm for updating the priority queue, and justify the algorithm's optimality.*

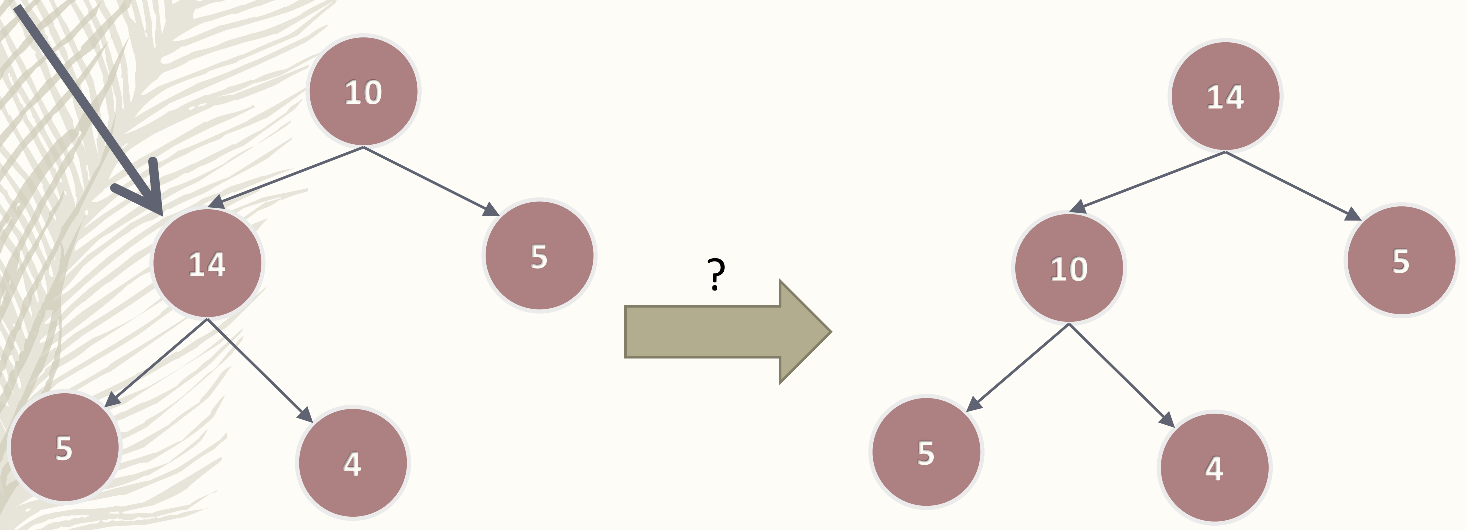
Underspecified *aspects:*

– *Is this a max or min PQ? A: This is a max PQ.*

– *What implementation do we need to support? A: Sedgewick's.*

Doing Examples by Hand

Although it might be fun to jump immediately in writing some code, let’s not do that. Let’s be sure we know exactly the problem we’re trying to solve. Afterall, won’t that save us from potentially wasting our time?



Definitions

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Any ambiguity here?

If there is: definitions are the way we fight it.

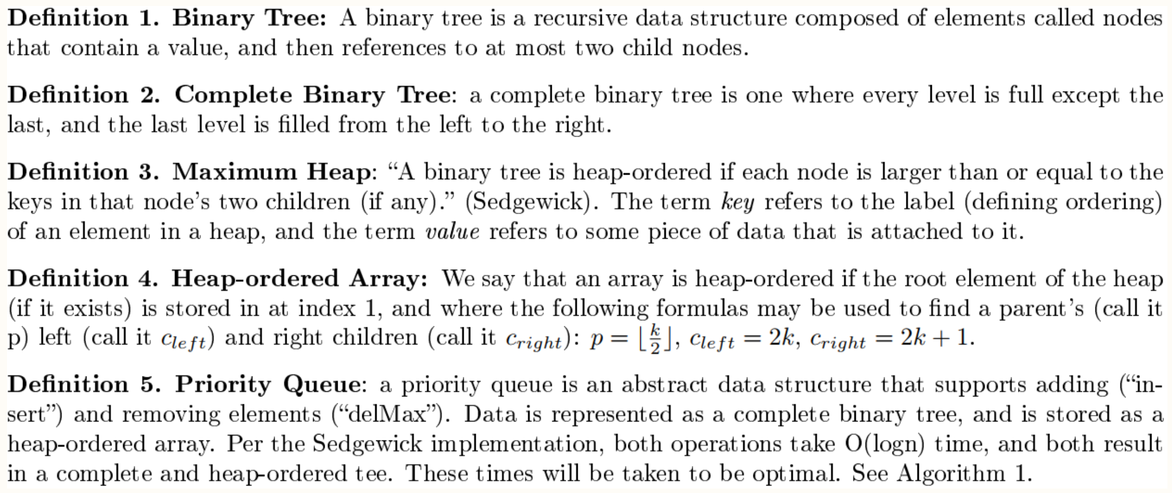
Definitions

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Assumptions:

- The priority queue (PQ) is a maximum PQ.
- The specific implementation is Sedgewick's.
- The algorithm will assume that it operates over a valid PQ, and produces a valid PQ.

Hopefully these are all reasonable…?



Definition 1. Binary Tree: A binary tree is a recursive data structure composed of elements called nodes that contain a value, and then references to at most two child nodes.

Definition 2. Complete Binary Tree: a complete binary tree is one where every level is full except the last, and the last level is filled from the left to the right.

Definition 3. Maximum Heap: "A binary tree is heap-ordered if each node is larger than or equal to the keys in that node's two children (if any)." (Sedgewick). The term *key* refers to the label (defining ordering) of an element in a heap, and the term *value* refers to some piece of data that is attached to it.

Definition 4. Heap-ordered Array: We say that an array is heap-ordered if the root element of the heap (if it exists) is stored in at index 1, and where the following formulas may be used to find a parent's (call it p) left (call it c_{left}) and right children (call it c_{right}): $p = \lfloor \frac{k}{2} \rfloor$, $c_{left} = 2k$, $c_{right} = 2k + 1$.

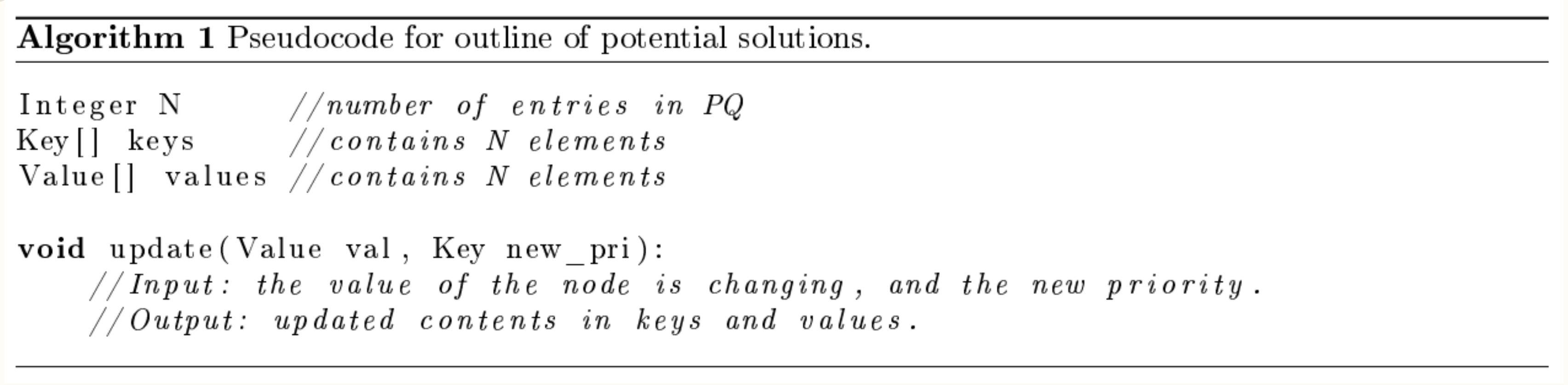
Definition 5. Priority Queue: a priority queue is an abstract data structure that supports adding ("insert") and removing elements ("delMax"). Data is represented as a complete binary tree, and is stored as a heap-ordered array. Per the Sedgewick implementation, both operations take O(logn) time, and both result in a complete and heap-ordered tee. These times will be taken to be optimal. See Algorithm 1.

Definitions

Let’s go a step further: let’s define what an answer would look like:

- We that it will look like pseudocode for an algorithm.
- We know the input and the output.

Write it up:



Algorithm 1 Pseudocode for outline of potential solutions.

```
Integer N        //number of entries in PQ
Key[] keys       //contains N elements
Value[] values   //contains N elements

void update(Value val, Key new_pri):
    //Input: the value of the node is changing, and the new priority.
    //Output: updated contents in keys and values.
```

Metrics

Design an efficient algorithm to update the priority of an entry inside of a priority queue.

Need to find what is important, distill it, and define it:

- M1: The ability of the algorithm to process a heap sorted array with at most one node which violates the heap rule and produce a heap sorted array with no violations of the maximum heap rule. The array must contain a complete tree at all times.
- M2: The efficiency of the algorithm. For a cost metric, we will use the number of lines run as a measure of computational time needed for a particular design.

Complete Analysis

What we've done:

– We understand the problem.

– We have a sound technical context for the problem (definitions, corollaries).

– We know what makes a good solution (metrics).

Now we can design an answer.

Remember: the analysis should not be biased in any towards a possible solution, and may contain work we ultimately don't need.

Updating a PQ: Design

JIT: Pseudocode

- The focus of pseudocode is to clearly express an algorithm.
  - The value is logic – not making the compiler happy.

- Important Note: do not use pseudocode to hide important details. For example: it's only fair to reference an undefined function when it doesn't add ambiguity like calling "distance(Integer x, Integer y)". In contrast, something like "has_deviation(Image original, Image new)" that is suppose to compare images to find a deviation isn't okay since we don't know what makes two images different.

Treat as a generic programming language.

Some Rules:

- Use Python style block rules (use colon to start intent, indents define where code belongs, not bracers).
- No semicolons.
- Use words instead of &|! operators.
- For loops should have the form FOR start TO end.
- Use complete type names (e.g., Integer instead of int).
- Avoid ++ or --.

JIT: Pseudocode

Here are some examples (left is Java from Sedgewick's book): (colors optional)

```
public void insert(Key v) {
  pq[++N] = v;
  swim(N);
}

private void swim(int k) {
  while (k > 1 && less(k/2, k)) {
    exch(k, k/2);
    k = k/2;
  }
}
```

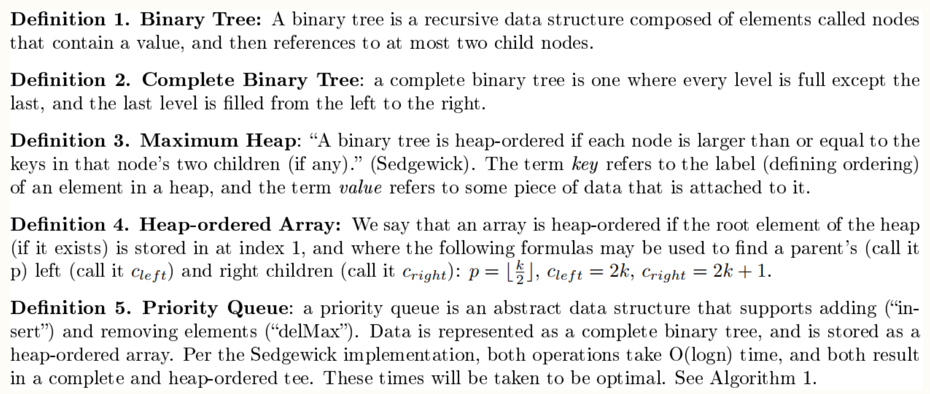
```
void insert(Key v):
  N = N + 1
  pq[N] = v
  swim(N)


void swim(integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

JIT: K

- K stands for Knowledge and is the symbol which we will use to refer the set of information which is known to be good and real (i.e., true).
  - In a class, K is typically provided as a working document...
- The purpose of K is to make sure that our design is *well-founded.*
- K can serve as a safe place to put generic definitions.

K =



Definition 1. Binary Tree: A binary tree is a recursive data structure composed of elements called nodes that contain a value, and then references to at most two child nodes.

Definition 2. Complete Binary Tree: a complete binary tree is one where every level is full except the last, and the last level is filled from the left to the right.

Definition 3. Maximum Heap: "A binary tree is heap-ordered if each node is larger than or equal to the keys in that node's two children (if any)." (Sedgewick). The term *key* refers to the label (defining ordering) of an element in a heap, and the term *value* refers to some piece of data that is attached to it.

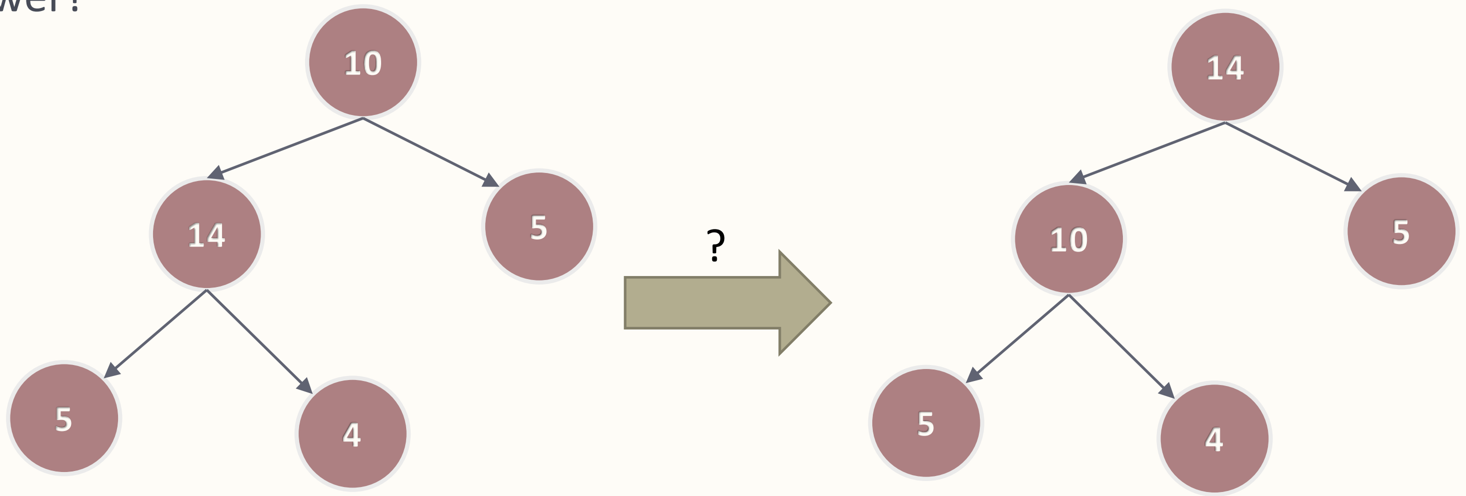
Definition 4. Heap-ordered Array: We say that an array is heap-ordered if the root element of the heap (if it exists) is stored in at index 1, and where the following formulas may be used to find a parent's (call it p) left (call it c_{left}) and right children (call it c_{right}): $p = \lfloor \frac{k}{2} \rfloor$, $c_{left} = 2k$, $c_{right} = 2k + 1$.

Definition 5. Priority Queue: a priority queue is an abstract data structure that supports adding ("insert") and removing elements ("delMax"). Data is represented as a complete binary tree, and is stored as a heap-ordered array. Per the Sedgewick implementation, both operations take O(logn) time, and both result in a complete and heap-ordered tee. These times will be taken to be optimal. See Algorithm 1.

Note: please refer to the assignment ADJ ruleset for information on the use of K.

Attempt

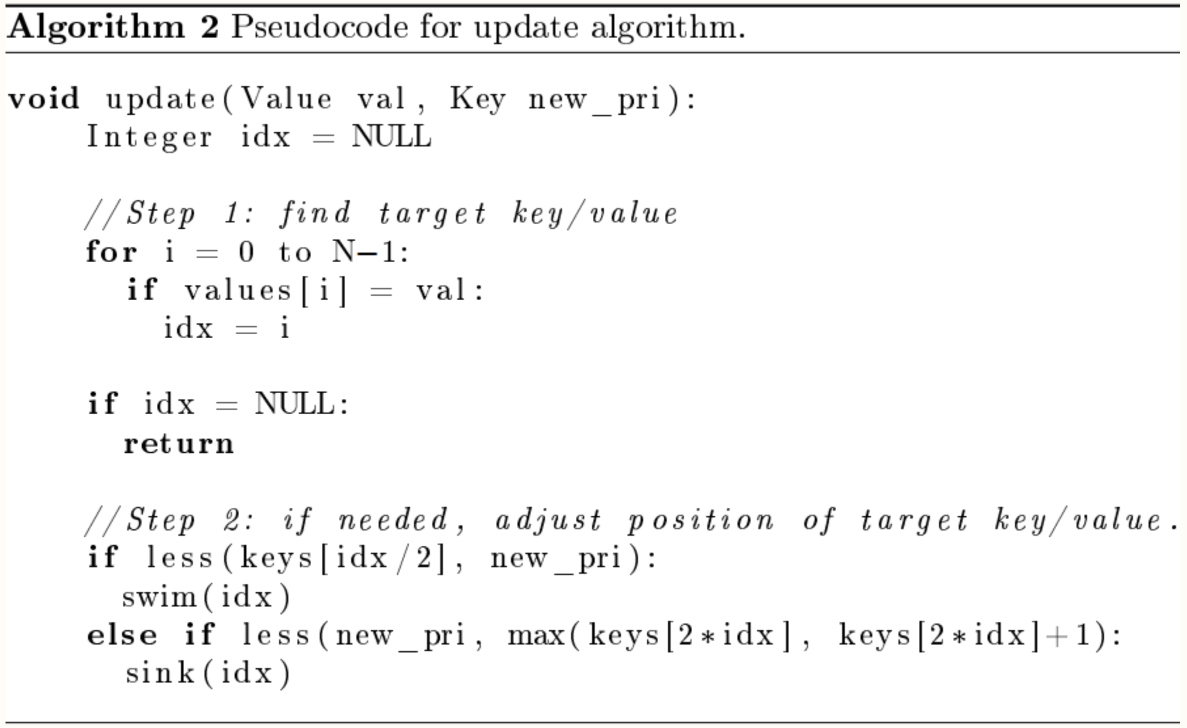
- We already have a good understanding of our problem: can we break it into pieces?
- This is called *decomposition* and is the standard way to deal with any problem whose solution isn't immediately obvious.
  - A thought: is it always safe to decompose a problem? Can you prove your answer?



Attempt

Here we go:

- Step 1: find target key/value.
- Step 2: if needed, adjust position of target key/value.



Algorithm 2 Pseudocode for update algorithm.

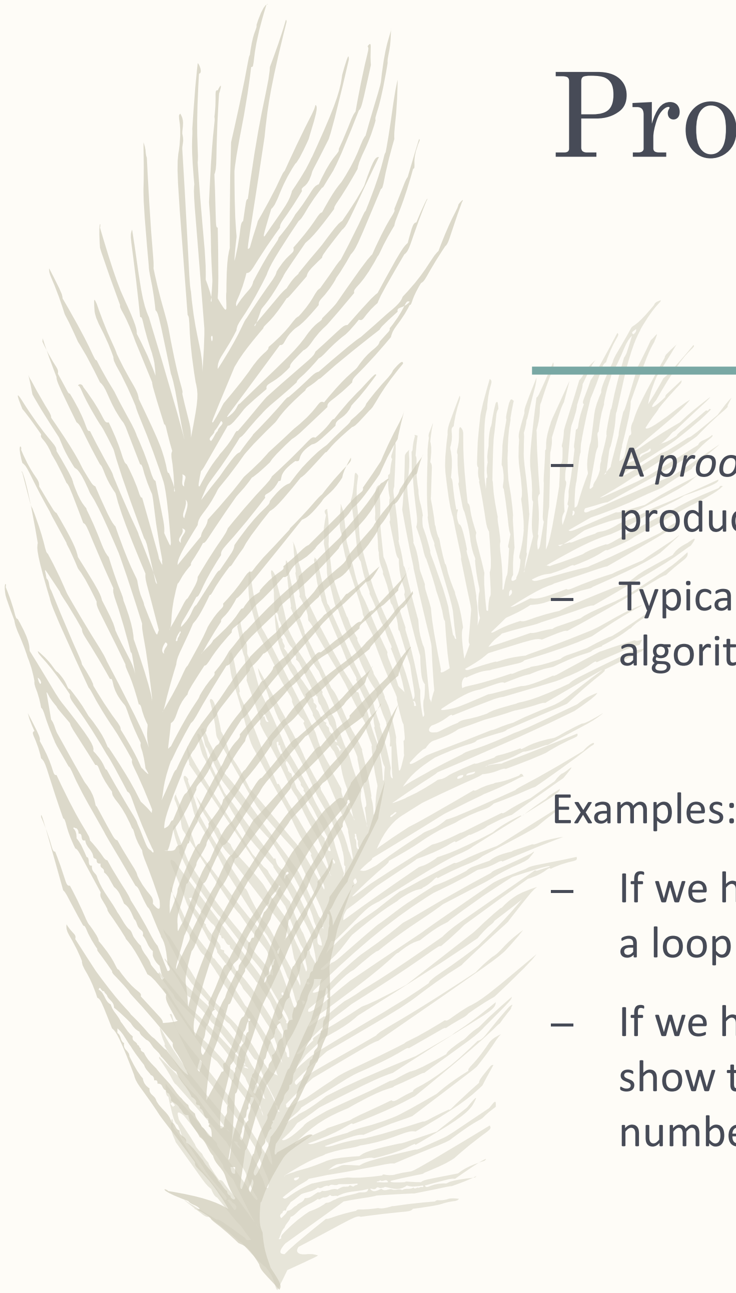
```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N-1:
      if values[i] = val:
        idx = i

    if idx = NULL:
      return

    //Step 2: if needed, adjust position of target key/value.
    if less(keys[idx/2], new_pri):
      swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
      sink(idx)
```

Updating a PQ: Justification



Proof of Termination

- A *proof of termination* shows that an algorithm always terminates. It’s the first step towards producing something useful – otherwise your program might just be an infinite loop.
- Typically is performed by determining some *progress* metric, and then showing that the algorithm “moves along” that metric.

Examples:

- If we had an algorithm that displayed the contents of an array, we would want to show that is a loop that uses each index only once, always increments, and checks if it is at the end.
- If we had an algorithm to find a path from some place to some other place, we might need to show that it visits every single place once, that no place is unreachable, there is a finite number of places, and it checks if there is no other place left to visit.

Proof of Termination

What if this was Double?

What if this was k != 0?

Example: let's show swim terminates

Proof:

Suppose that k is some number – is a valid index into the array. It must be positive. The only way for this code to not terminate is if the while-loop cannot end. However, in each iteration, k is halved, meaning it is closer to K <= 1 than it was on the previous iteration. Therefore, we monotonically converge on an condition that can break out of the loop. ■

```
void swim(Integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

This has three simple lines, let's assume it terminates

What is the progress metric?

Justification

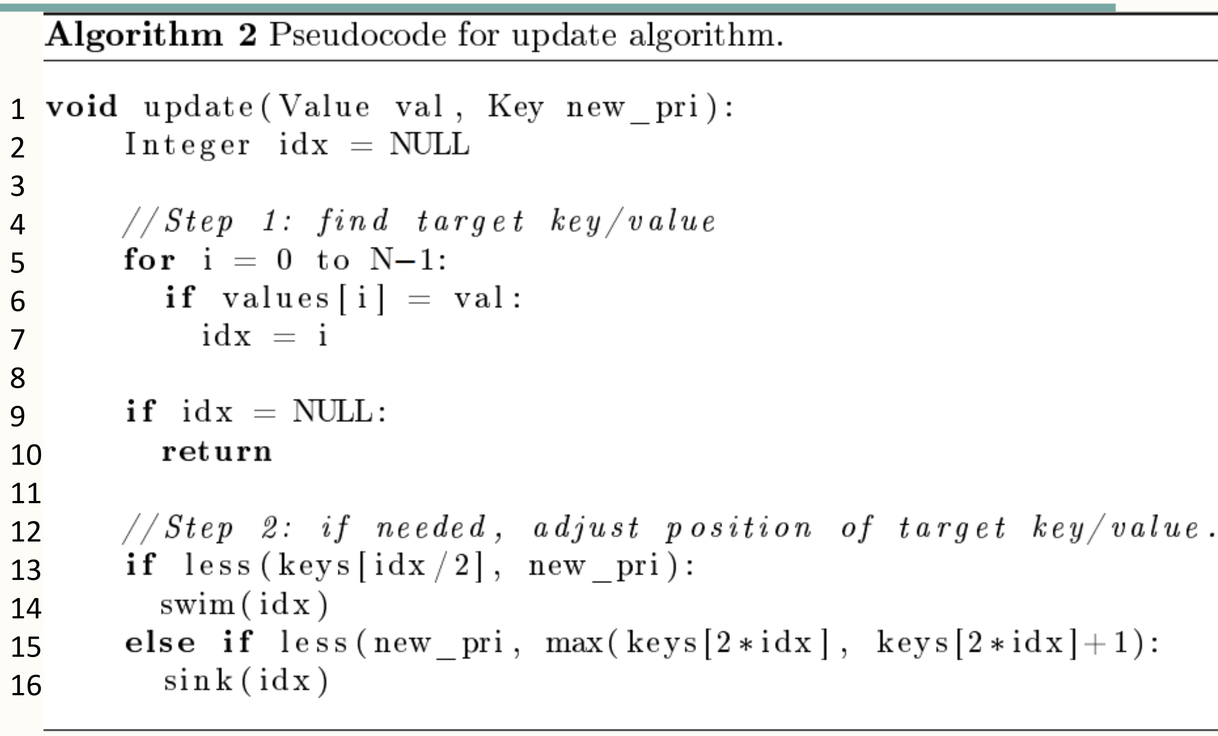
Proof of Termination

Note: the proof of termination won't occur in our final result for the update() problem. Instead, we will focus on correctness – why?

Proof:

This method will always terminate. Assume that that both less and sink will always terminate. Also note that do not use update. Initially line 2 runs. Then, there is a loop, which goes from 0 to N-1. Since N is the size of the tree, it will be a finite number, hence the loop will run a finite number of times. The code that follows contains no repetition constructs, therefore each line may be executed at most once. Therefore, this algorithm will terminate. ■

Not that reasonable.


Algorithm 2 Pseudocode for update algorithm.

```
1  void update(Value val, Key new_pri):
2      Integer idx = NULL
3
4      //Step 1: find target key/value
5      for i = 0 to N−1:
6        if values[i] = val:
7          idx = i
8
9      if idx = NULL:
10       return
11
12     //Step 2: if needed, adjust position of target key/value.
13     if less(keys[idx/2], new_pri):
14       swim(idx)
15     else if less(new_pri, max(keys[2∗idx], keys[2∗idx]+1):
16       sink(idx)
```

What is the progress metric?

Proof of Correctness

- A *proof of correctness* shows that an algorithm always performs in a certain way, or always acts according to certain constraints. If we have a proof of correctness, that asserts that our algorithm is “right”. The remaining is of course: is it fast enough?
- Typically is performed by looking at the way that algorithm works and transforms data to argue that the output will always have certain properties.

Examples:

- If we had an algorithm that displayed the contents of an array, we would want to show that is a loop that uses each index only once, always increments, and checks if it is at the end.
- If we had an algorithm to find a path from some place to some other place, we would need to show if a path existed in the input data, then then algorithm would find (and vice versa).

Proof of Correctness

Example: let's show swim is correct. We need to show that the result of running swim is a valid and complete heap, despite one node potentially being too large.

Notation: Let *parent(k)* indicate the parent k, *left(k)* the left child*, and right(k)* the right child. Since we are using a heap: parent(k) ≥ left(k) and parent(k) ≥ right(k).

Proof: We can trivially see that the tree will complete since the change that can be made is exchanges.

Let k be the node that will be swim'ed. If k is a node already in its proper position then less(parent(k), k)) will be false, and no code will run. Hence, the result is correct. If k=1, then we know that the heap is correctly ordered and the loop will terminate. In other cases: we swap k with parent(k). In this case, since k > parent(k), and parent(k) ≥ left(k) we see that both k > left(k) and k > parent(k) so moved k will be a valid root of a heap. It is moved. We then repeat again on parent(k), which then falls into one of these three categories repeatedly. ■

```
void swim(integer k)
  while (k > 1 AND less(k/2, k)):
    parent = k/2
    exch(k, parent)
    k = parent
```

```
void swim(integer k)
  while (k > 1 AND less(parent(k), k)):
    parent = p(k)
    exch(k, parent)
    k = parent
```

This whole thing falls under the technique of showing that running an algorithm preserves the existing properties of a data set.

Proof of Correctness

The metric we defined earlier is was:

- M1: The ability of the algorithm to process a heap sorted array with at most one node which violates the heap rule and produce a heap sorted array with no violations of the maximum heap rule. The array must contain a complete tree at all times.

Here we just need to show that fulfill (in a yes/no sense) this metric – not too bad.

Justification

Proof of Correctness: Metric 1

Would this be needed if we had an immutable data structure?

This can be shown by cases:

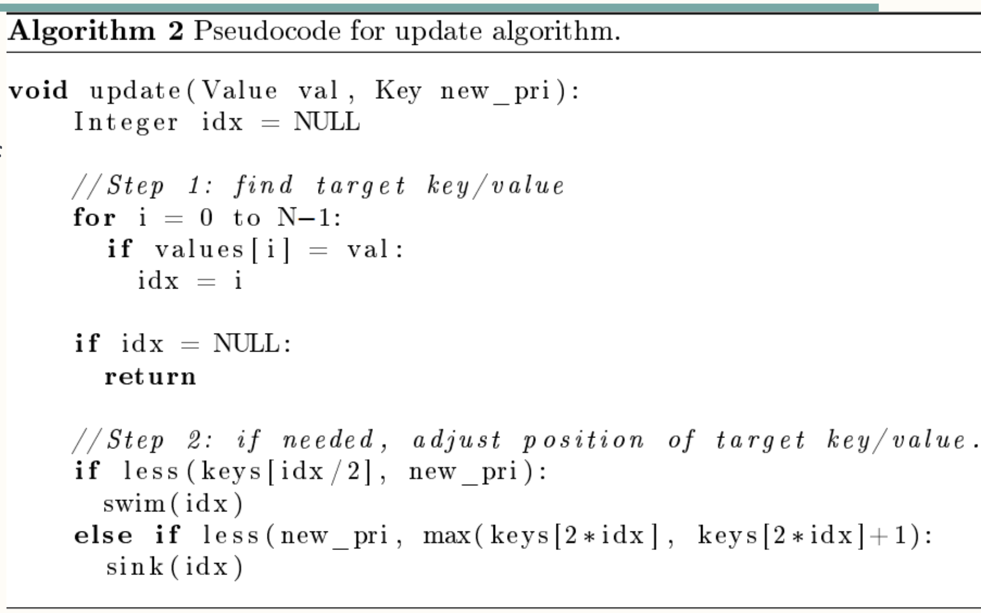
Case 1: tree does not contain node with value. In Step 1 of the algorithm, we loop over element and check its contents. No changes are made to N, keys, or value, so after that step, the data will still be heap ordered.

If there is no node to update, and the algorithm will terminate before step 2 (see if-conditional).

Case 2: tree contains node with value. Step will be fine. For Step 2, we defer to the mechanisms for swim and skin which are already known to result in a heap-ordered array that stores a complete tree.

Both functions will terminate if there is no work to do (thus we neglect the case of the node needing to move or not move).

The if-statements evaluates the case of being out of order with parent or children, and follows the formula as defined.



Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N-1:
      if values[i] = val:
        idx = i

    if idx = NULL:
      return

    //Step 2: if needed, adjust position of target key/value.
    if less(keys[idx/2], new_pri):
      swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
      sink(idx)
```

Proof of Efficiency

The metric we defined earlier was:

- M2: For a cost metric, we will use the number of lines run as a measure of computational time needed for a particular design.

Since we are working with an algorithm, we ideally want to show that our algorithm is at least as fast as any other.

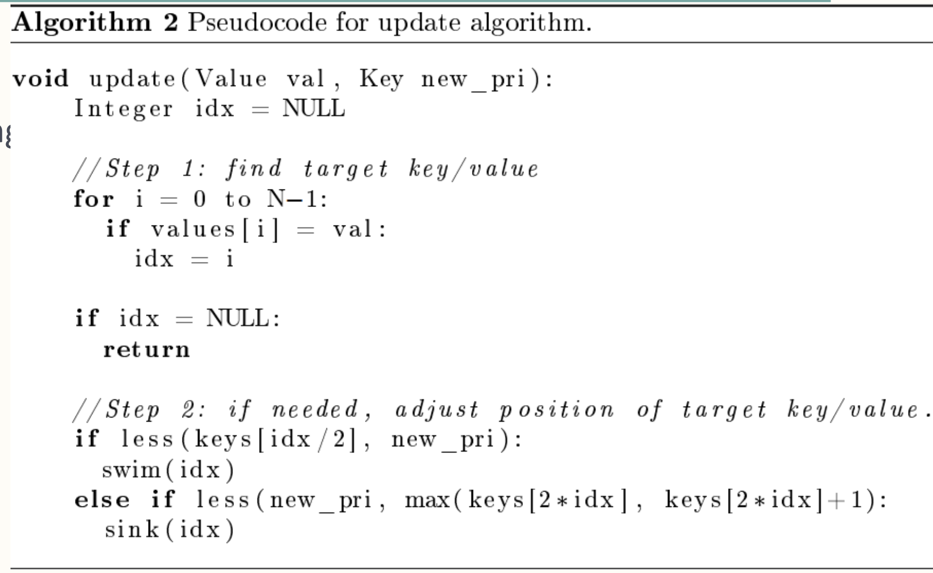
Justification

Proof of Efficiency: Metric 2

- Per K, insert takes O(logn). Thus, the best we can get is O(logn) or we would be able to implement insert() faster usin update().
- The code has a loop and a call to swim/sink: O(n)+O(logn)=O(n)
- We need to show there is no O(logn) performing algorithm.
- We propose that the linear time loop is a bottleneck.

Proof Sketch:

- We have to search for the index of the target value.
- Looking at one element, we have two children to consider. Moving downward means selecting a child, but since there is no way to tell which is likely to lead to the target, the choice must be arbitrary, and we probably need to background explore a different path.
- This means we cannot have a log time solution, and linear time is required. ■



Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
  Integer idx = NULL

  //Step 1: find target key/value
  for i = 0 to N-1:
    if values[i] = val:
      idx = i

  if idx = NULL:
    return

  //Step 2: if needed, adjust position of target key/value.
  if less(keys[idx/2], new_pri):
    swim(idx)
  else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
    sink(idx)
```