CSE 332 Winter 2026
Lecture 2: Algorithm Analysis pt.1

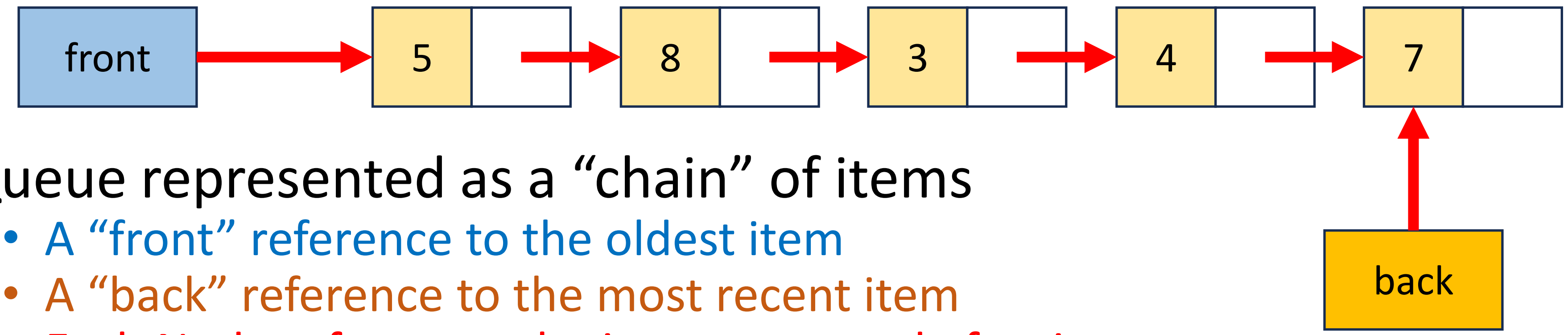
Nathan Brunelle

http://www.cs.uw.edu/332

Announcements

- Exercise 0 released
  - Due Wednesday 1/14
  - There are 2 separate gradescope submissions
- Concept check 0 released
  - Helps us to get more familiar with each other!
  - Gradescope submission due 1/15
  - Meet the staff activity due by 1/30
    - Come to any office hours, chat with us, ask us to mark you off for this

Linked Queue Data Structure (Algorithms)



- Queue represented as a "chain" of items
  - A "front" reference to the oldest item
  - A "back" reference to the most recent item
  - Each Node references the item enqueued after it
- enqueue Procedure:

```
enqueue(x){
    last = new ListNode(x);
    back.next = last;
    back = last;
}
```

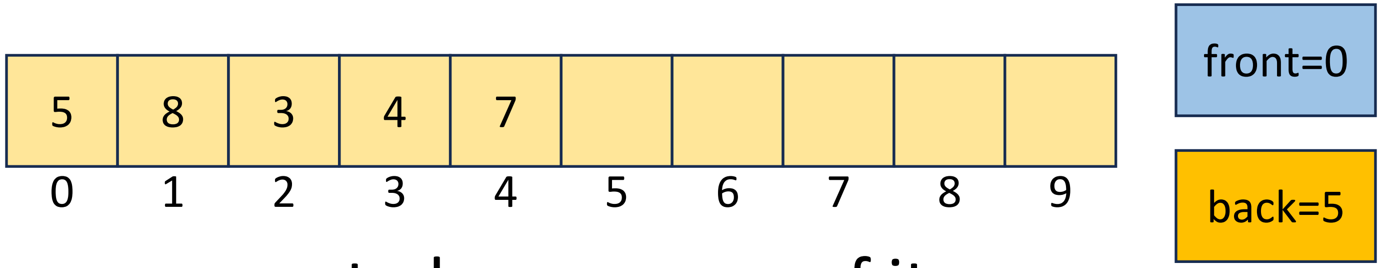
- dequeue Procedure:

```
dequeue(){
    first = front.value;
    front = front.next;
    if (front == null) {back = null;}
    return first
}
```

- isEmpty Procedure:

```
isEmpty(){
    return front == null;
}
```

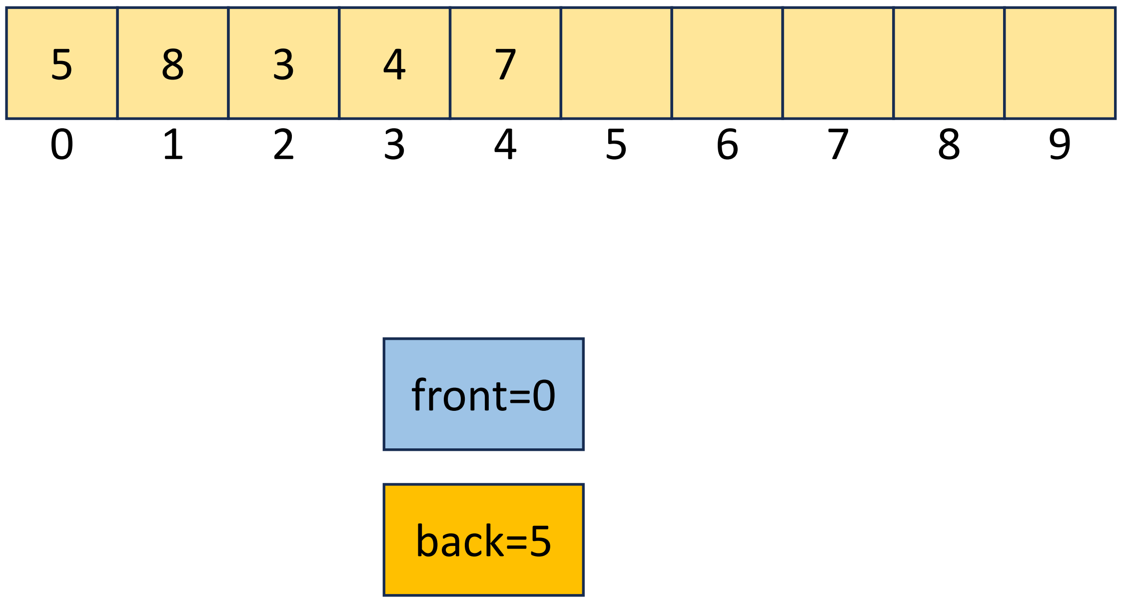
"Circular" Array Queue (Idea)

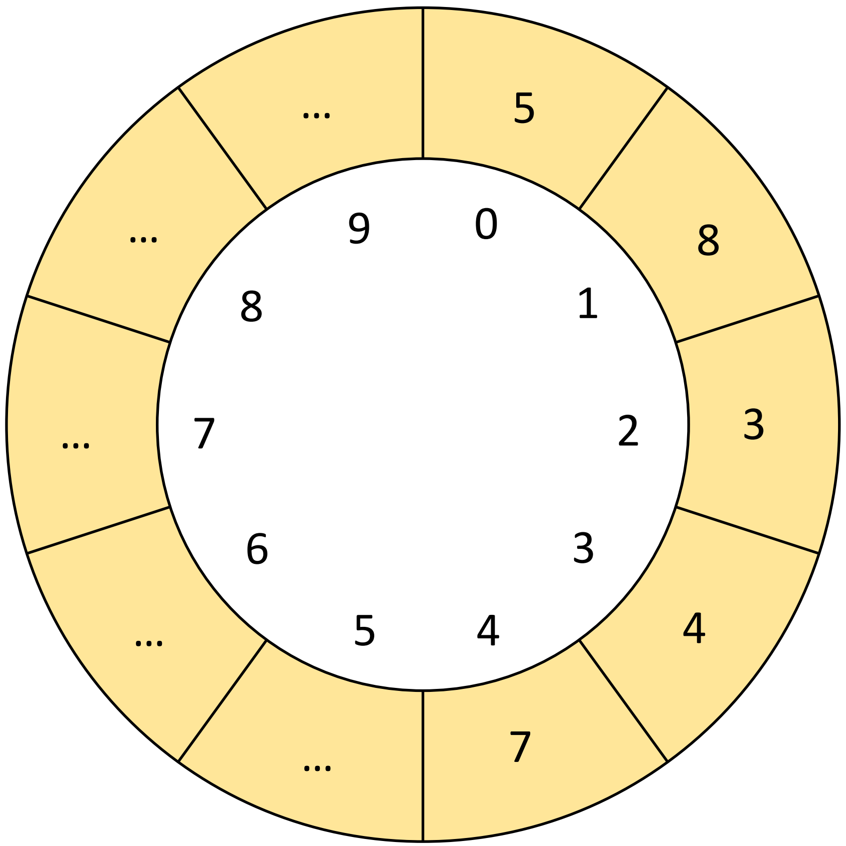


- Queue represented as an array of items
  - A "front" index to indicate the oldest item in the queue
  - A "back" index to indicate the most recent item in the queue
    - Actually, the first "open" slot in the array
- enqueue Procedure:
- dequeue Procedure:
- isEmpty Procedure:

“Circular” Array

- Intuitively, An array of values arranged in a “circle” rather than a line
  - If you go beyond the last index, to wrap back around to 0





"Circular" Array Queue (Algorithms)

5	8	3	4	7					
0	1	2	3	4	5	6	7	8	9

front=0

back=5

- Queue represented as an array of items
  - A "front" index to indicate the oldest item in the queue
  - A "back" index to indicate the most recent item in the queue
- enqueue Procedure:

```
enqueue(x){
    queue[back] = x;
    back = (back + 1) % queue.length;
    size++;
}
```

- dequeue Procedure:

```
dequeue(){
    // Assumes queue is not empty
    first = queue[front];
    front = (front + 1) % queue.length;
    size--;
    return first;
}
```

- isEmpty Procedure:

```
isEmpty(){
    return size == 0;
}
```

What if we run out of space?!

Resizing “Circular” Array Queue

5	8	3	4	7					
0	1	2	3	4	5	6	7	8	9

front=0

back=5

How do you resize?

That’s for Exercise 0!

- Queue represented as an array of items
  - A “front” index to indicate the oldest item in the queue
  - A “back” index to indicate the most recent item in the queue
- enqueue Procedure:

```
enqueue(x){
    if (size == queue.length-1) {resize();}
    queue[back] = x;
    back = (back + 1) % queue.length;
    size++;
}
```

- dequeue Procedure:

```
dequeue(){
    // Assumes queue is not empty
    first = queue[front];
    front = (front + 1) % queue.length;
    size--;
    return first;
}
```

- isEmpty Procedure:

```
isEmpty(){
    return size == 0;
}
```

Linked List vs. Circular Array

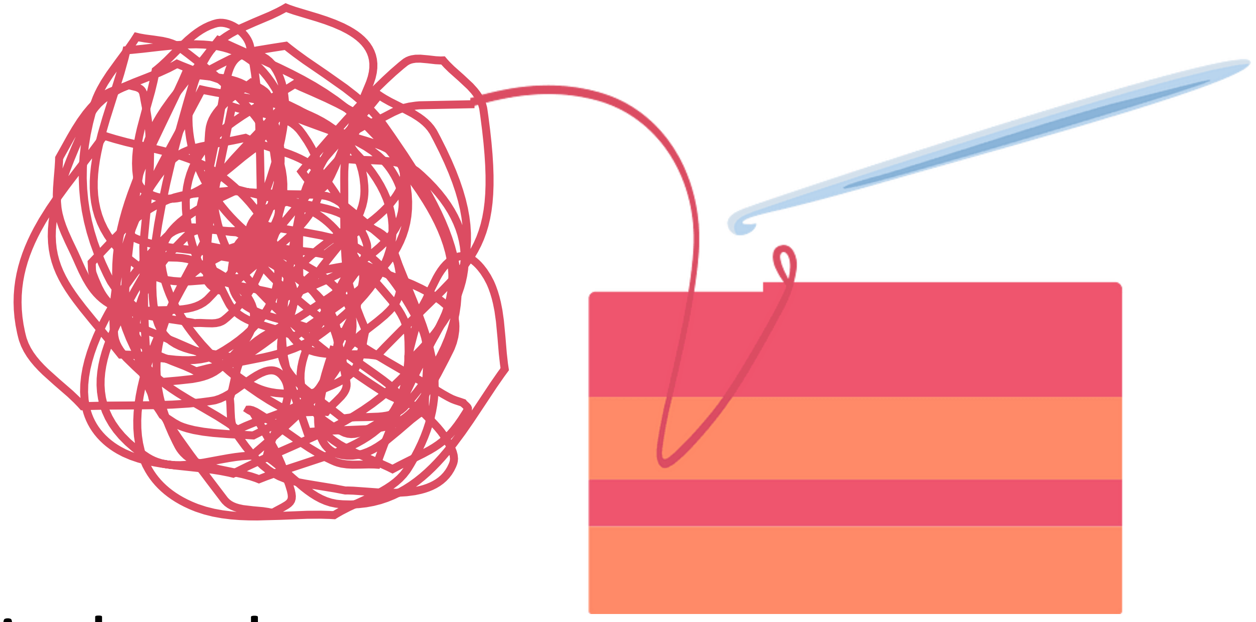
- Let’s Summarize the benefits and drawbacks of each

ADT: Stack

- What is it?
  - A “Last In First Out” (LIFO) collection of items (sometimes called FILO)
- What operations do we need?
  - push
    - Add a new item onto the stack
  - peek
    - Return the value of the most recently pushed item
  - pop
    - Return the value of the most recently pushed item and remove it from the stack
  - isEmpty
    - Indicate whether or not there are items still on the stack

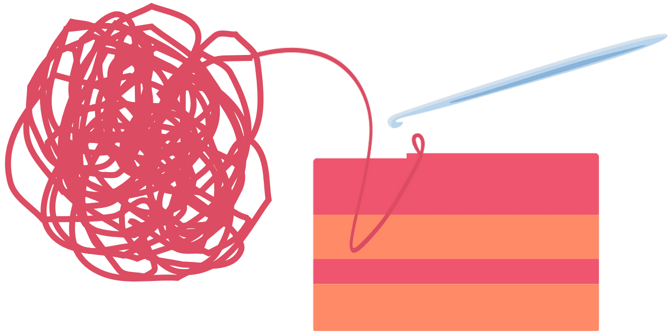
Motivating Example

Let's design an algorithm



- I have a pile of string
- I have one end of the string in-hand
- I need to find the other end in the pile
- How can I do this efficiently?

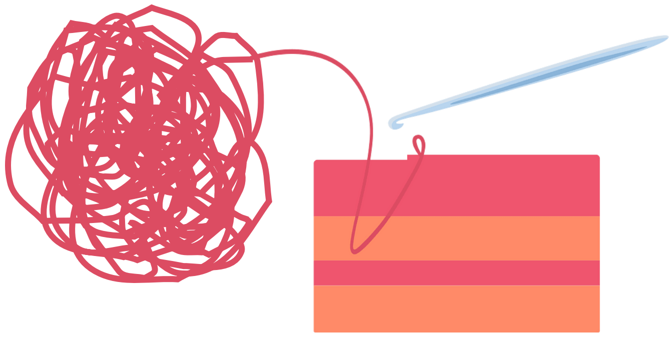
Algorithm Ideas



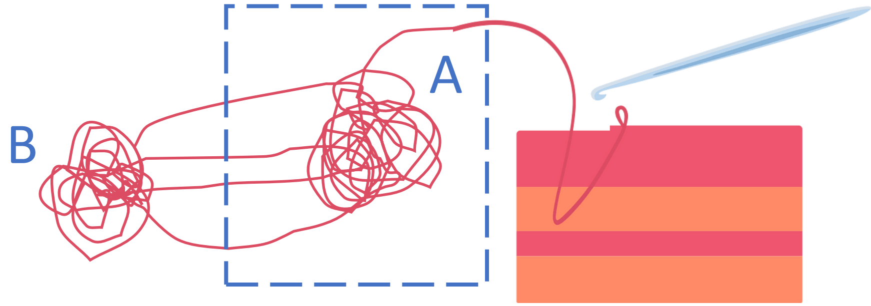
- Whatcha got?

My Approach

End-of-Yarn Finding Algorithm



Set aside the already-obtained beginning
Do the following until you find the end:
    Separate the pile of yarn into 2 piles
    Label A to be the pile that the beginning enters
    Label B to be the other pile
    Count the number of strands crossing the piles
    If count is even, set the pile to be A
    Otherwise set the pile to be B.



Resource Analysis

Why do resource analysis?

- Allows us to compare *algorithms,* not implementations
  - Using observations necessarily couples the algorithm with its implementation
  - If my implementation on my computer takes more time than your implementation on your computer, we cannot conclude your algorithm is better
- We can predict an algorithm's running time before implementing
- Understand where the bottlenecks are in our algorithm

Process for resource Analysis

- End Result: A *function* which maps the algorithm's input size to count of resources used
  - Input of the function: **sizes** of the input
    - Number of characters in a string, number of items in a list, number of pixels in an image
  - Output of the function: **counts** of resources used
    - Number of times the algorithm adds two numbers together, number times the algorithm does a > or < comparison, maximum number of bytes of memory the algorithm uses at any time
- Important note: Make sure you know the "units" of your input and output!

Resource Analysis – Worst Case Running Time

- If an algorithm has a worst case running time of $f(n)$
  - Among all possible size-n inputs, the “worst” one will do $f(n)$ “operations”
  - I.e. $f(n)$ gives the maximum operation count from among all inputs of size n

Resource Analysis – Best Case Running Time

- If an algorithm has a **best** case running time of $f(n)$
  - Among all possible size-n inputs, the "**best**" one will do $f(n)$ "operations"
  - I.e. $f(n)$ gives the **minimum** operation count from among all inputs of size n

Resource Analysis – Worst Case Space

- If an algorithm has a worst case space of $f(n)$
  - Among all possible size-n inputs, the "worst" one will use $f(n)$ bits of memory
  - I.e. $f(n)$ gives the maximum amount of memory required from among all inputs of size n

Analysis Process From 123/143

- Count the number of “primitive operations”
  - +, -, compare, arr[i], arr.length, etc
- Write that count as an expression using n (the input size)
- Put that expression into a “bucket” by ignoring constants and “non-dominant” terms, then put a $O(\ )$ around it.
  - $4n^2 + 8n - 10$ ends up as $O(n^2)$
  - $\frac{1}{2}n + 80$ ends up as $O(n)$
  - $n(n + 1)$ ends up as $O(n^2)$

Analysis Process For Us

- Count the number of *chosen* operation(s)
  - Factors to consider when choosing which operation(s) to count
    - **Necessity**: should be necessary for solving the problem
    - **Frequency**: should be the most frequently done (up to a constant factor)
    - **Magnitude**: should be expensive to perform each chosen operation
- Write that count as an expression using n (the input size)
- Put an asymptotic bound on it (one of O, Ω, Θ)
  - More on this next class

Worst Case Running Time - Example

```
myFunction(List n){
    b = 55 + 5;
    c = b / 3;
    b = c + 100;
    for (i = 0; i < n.size(); i++) {
        b++;
   }
    if (b % 2 == 0) {
        c++;
    }
    else {
        for (i = 0; i < n.size(); i++) {
            c++;
        }
    }
    return c;
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
    - How many times will it run?
    - How long does it take to run?
    - Does this change with different inputs?
- Answer:

Worst Case Running Time - Example

```
myFunction(List n){
  b = 55 + 5; // 1
  c = b / 3;  // 1
  b = c + 100;  // 1
  for (i = 0; i < n.size(); i++) { // 1, n times
    b++; // 1
  }
  if (b % 2 == 0) { // 1
    c++; // 1
  }
  else {
    for (i = 0; i < n.size(); i++) { // 1, n times
      c++; // 1
    }
  }
  return c;
}
```

Questions to ask:

- What are the units of the input size?
  - # of items in the list
- What are the operations we're counting?
  - Arithmetic ops (+-*/)
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with different inputs?
- Answer:
  - $3 + 2n + 1 + 2n = 4n + 4$
  - $O(n)$

Worst Case Running Time – Example 2

```
beAnnoying(List n){
    List m = [];
    for (i=0; i < n.size(); i++){
        m.add(n[i]);
        for (j=0; j< n.size(); j++){
            print ("Hi, I'm annoying");
        }
    }
}
```

Questions to ask:

- What are the units of the input size?
- What are the operations we're counting?
- For each line:
    - How many times will it run?
    - How long does it take to run?
    - Does this change with the input size?

Worst Case Running Time – Example 2

```
beAnnoying(List n){
    List m = [];
    for (i=0; i < n.size(); i++){ // n times
        m.add(n[i]);
        for (j=0; j< n.size(); j++){ // n times
            print (“Hi, I’m annoying”); // 1
        }
    }
}
```

Questions to ask:

- What are the units of the input size?
  - # items
- What are the operations we’re counting?
  - Adding or printing
  - Printing: $O(n^2)$
- For each line:
  - How many times will it run?
  - How long does it take to run?
  - Does this change with the input size?

Worst Case Running Time – General Guide

- Add together the time of consecutive statements
- Loops: Sum up the time required through each iteration of the loop
  - If each takes the same time, then [time per loop × number of iterations]
- Conditionals: Sum together the time to check the condition and time of the slowest branch
- Function Calls: Time of the function's body
- Recursion: Solve a recurrence relation

Defining your running time function

- Worst-case complexity:
  - max number of steps algorithm takes on “most challenging” input
- Best-case complexity:
  - min number of steps algorithm takes on “easiest” input
- Average/expected complexity:
  - avg number of steps algorithm takes on random inputs (distribution-dependent)
- Amortized complexity:
  - max total number of steps algorithm takes on M “most challenging” consecutive inputs, divided by M (i.e., divide the max total sum by M).

Amortized Complexity Example - ArrayList

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    int[] oldData = data;
    data = new int[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

- What is the worst case running time of add?
    - Input size: size of “this”
    - Operations counted: indexing
    - $O(n)$

Amortized Complexity Example - ArrayList

```
public void add(int value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    int[] oldData = data;
    data = new int[data.length*2];
    for(int i = 0; i < oldData.length; i++)
        data[i] = oldData[i];
}
```

Every time we resize, we earn `data.length` more adds before the next resize!

- Amortized Analysis Idea:
  - Suppose we have a program that in total does n adds.
  - How much time was spent "on average" across all n?
- Let c be the initial size of `data`
  - The first c adds take: $c + c = 2c$
  - The next $2c$ adds: $2c + 2c = 4c$
  - The next $4c$ adds: $4c + 4c = 8c$
  - Overall: $\frac{14c}{7c} = 2c$

Amortized Analysis Analogy

- Suppose I’d like to park in a lot where they charge \$10 per day to park
- If you are caught in the lot without paying you are given a warning
- If you get 3 warnings, you are charged a \$25 fine, and your warnings reset.
- Should you actually pay to park?
  - If you pay every day then you pay an average of \$10 per day
  - If you do not pay then for every three days parking costs \$0+\$0+\$25, for an average of \$8.33 per day
    - This is an amortized analysis