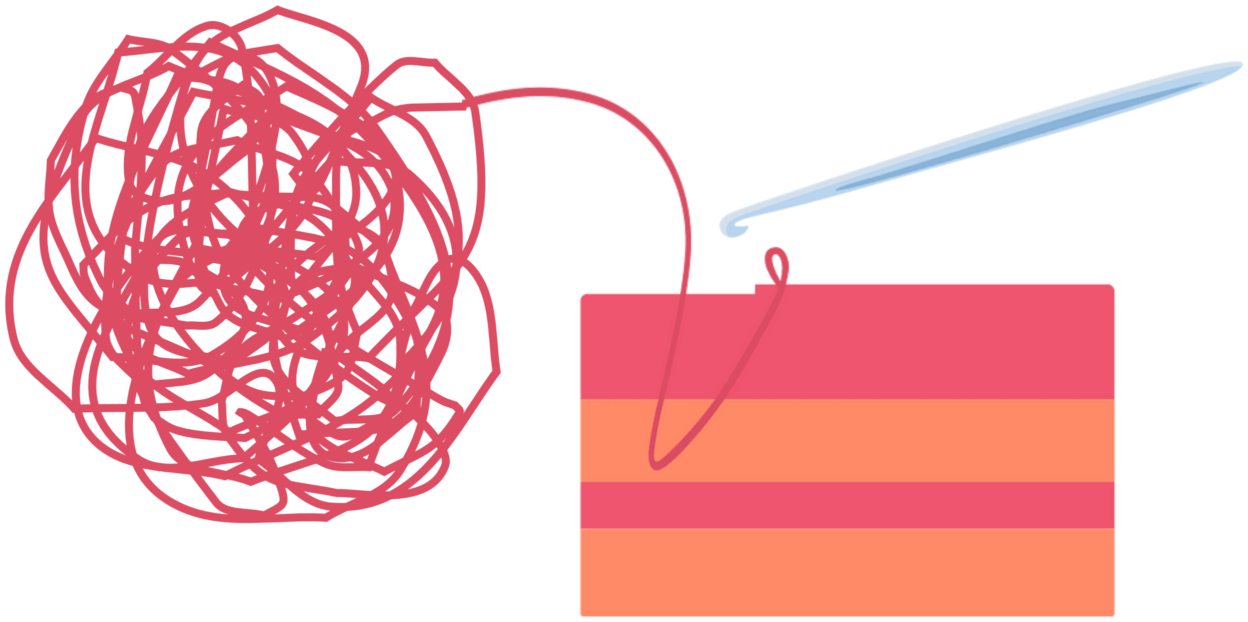
CSE 332 Summer 2024 Lecture 2: Algorithm Analysis pt.1

Nathan Brunelle

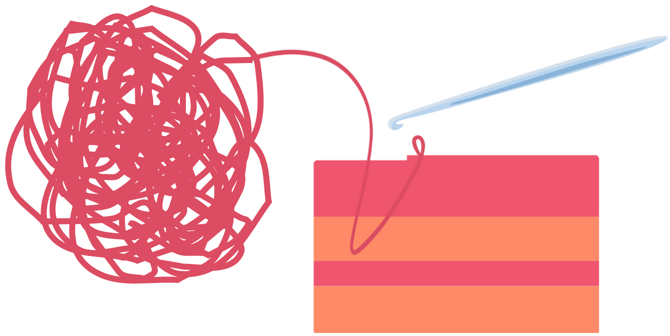
http://www.cs.uw.edu/332



Warm up:

- 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



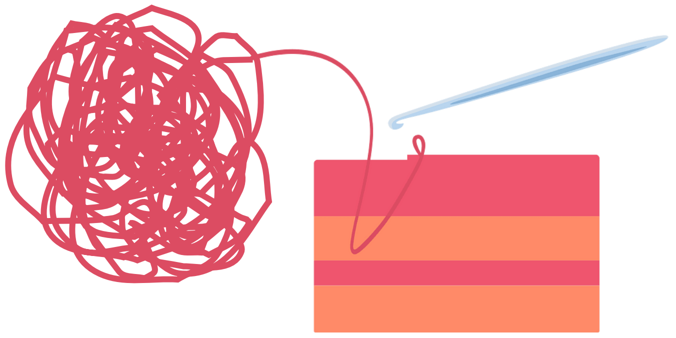
- Ideas:

Algorithm Running Times

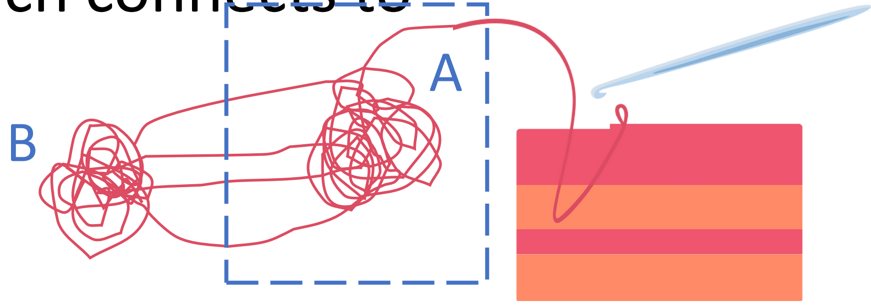
- How do we express running time?
- Units of “time”
- How to express efficiency?

My Approach

End-of-Yarn Finding



1. Set aside the already-obtained "beginning"
2. If you see the end of the yarn, you're done!
3. Separate the pile of yarn into 2 piles, note which connects to the beginning (call it pile A, the other pile B)



Repeat on pile with end

4. Count the number of strands crossing the piles
5. If the count is even, pile A contains the end, else pile B does

Why Do Resource Analysis?

- Allows us to compare *algorithms,* not implementations
  - Using observations *necessarily* couples the algorithm with its implementation
  - My implementation on my computer takes more time than your implementation on your computer. Do you have a better algorithm or computer?
- We can predict an algorithm's running time before implementing
- Understand where the bottlenecks are in our algorithm

Goals for Algorithm Analysis

- Identify a *function* which maps the algorithm's input size to a measure 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
      - does a > or < comparison
    - Number of bytes of memory needed
- Important note: Make sure you know the "units" of your domain (input size) and range (resource used)!

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 $O(n^2)$

Worst Case Analysis (in general)

- If an algorithm has a worst case resource complexity of $f(n)$
  - Among all possible size-n inputs, the “worst” one will use $f(n)$ “resources”
  - $f(n)$ gives the maximum count needed from among all inputs of size n

Worst Case Analysis (in general)

- If an algorithm has a worst case resource complexity of $f(n)$
  - Among all possible size-n inputs, the “worst” one will use $f(n)$ “resources”
  - $f(n)$ gives the maximum count needed from among all inputs of size n

Worst Case Running Time Analysis

- 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**”
  - $f(n)$ gives the maximum count of **operations** needed from among all inputs of size n

Worst Case Space Analysis

- If an algorithm has a worst case space complexity of $f(n)$
  - Among all possible size-n inputs, the "worst" one will need $f(n)$ "**memory units**" (usually bits)
  - $f(n)$ gives the maximum number of bits needed from among all inputs of size n

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 the input size?

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”);
        }
    }
    return;
}
```

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 – 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 (context-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(T value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    T[] oldData = data;
    data = (T[]) new Object[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:
  - Operations counted:
  - $O(?)$

Amortized Complexity Example - ArrayList

```
public void add(T value){
    if(data.length == size)
        resize();
    data[size] = value;
    size++;
}
private void resize(){
    T[] oldData = data;
    data = (T[]) new Object[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 guaranteed to not 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: $\frac{14c}{7c} = 2c$