CSE 332 Autumn 2023
Lecture 2: Algorithm Analysis pt.1

Nathan Brunelle

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

• Abstract Data Type (ADT)
  • Mathematical description of a “thing” with set of operations on that “thing”

• Algorithm
  • A high level, language-independent description of a step-by-step process

• Data structure
  • A specific organization of data and family of algorithms for implementing an ADT

• Implementation of a data structure
  • A specific implementation in a specific language
ADT: Queue

• What is it?
  • A “First In First Out” (FIFO) collection of items

• What Operations do we need?
  • Enqueue
    • Add a new item to the queue
  • Dequeue
    • Remove the “oldest” item from the queue
  • Is_empty
    • Indicate whether or not there are items still on the queue
Linked List – Queue Data Structure

- Queue represented as a “chain” of items
  - A “front” variable referencing the oldest item
  - A “back” variable referencing the most recent item
  - Each item points to the item enqueued after it

- Enqueue Procedure:

- Dequeue Procedure:

- Is_empty Procedure:
Linked List – Queue Data Structure

- Queue represented as a “chain” of items
  - A “front” variable referencing the oldest item
  - A “back” variable referencing the most recent item
  - Each item points to the item enqueued after it

Queue Representation:
- Front: 5 → 8 → 3 → 4 → 7 → Back

Enqueue Procedure:
```java
enqueue(x){
    last = new Node(x)
    back.next = last
    back = last
}
```

Dequeue Procedure:
```java
dequeue(){
    first = front.item
    front = front.next
    return first
}
```

Is_empty Procedure:
```java
is_empty(){
    return front.equals(Null)
}
```
Circular Array – Queue Data Structure

- Queue represented as a “chain” of items
  - A “front” variable referencing the oldest item
  - A “back” variable referencing the most recent item
  - Each item points to the item enqueued after it

- Enqueue Procedure:

- Dequeue Procedure:

- Is_empty Procedure:
Circular Array – Queue Data Structure

• 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:
• Dequeue Procedure:
• Is_empty Procedure:
Circular Array – Queue Data Structure

• 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:

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

• Dequeue Procedure:

```java
dequeue(){
  first = queue[front]
  front = (front + 1) % queue.length
}
```

• Is_empty Procedure:

```java
is_empty(){
  return front == back
}
```
Linked List vs. Circular Array

• If you know the max size of the queue, maybe consider array version
  • Want to enforce a max size
• Want to peek at a specific index
• Circular array might use memory more efficiently
• One might be better for some implementations (e.g. different types of objects)
• Concurrency...?
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)

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
  • 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
Goals for Algorithm Analysis

• Identify a function which maps the algorithm’s input size to a measure of resources used
  • Domain 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
  • Codomain 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 domain and codomain!
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”
  • I.e. $f(n)$ gives the maximum count of resources 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”
  • I.e. $f(n)$ gives the maximum operation count 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”
  • I.e. $f(n)$ gives the maximum memory unit count 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?
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 \([\text{time per loop} \times \text{number of iterations}]\)
• 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