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
Running Time Analysis

• Units of “time”
• How do we express running time?
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 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 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");
        }
    }
}

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
Searching in a Sorted List

boolean linearSearch(array a, integer k){
    for(i=0; i< a.length; i++){
        if (a[i] == k){
            return true;
        }
    }
    return false;
}
Faster way?

Can you think of a faster algorithm to solve this problem?
Comparing
Comparing Running Times

• Suppose I have these algorithms, all of which have the same input/output behavior:
  • Algorithm A’s worst case running time is $10n + 900$
  • Algorithm B’s worst case running time is $100n - 50$
  • Algorithm C’s worst case running time is $\frac{n^2}{100}$

• Which algorithm is best?
\[ f(n) = O(g(n)) \]
\[ f(n) = \Theta(g(n)) \]
\[ f(n) = \Omega(g(n)) \]
Asymptotic Notation

- $O(g(n))$
  - The set of functions with asymptotic behavior less than or equal to $g(n)$
  - Upper-bounded by a constant times $g$ for large enough values $n$
  - $f \in O(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \leq c \cdot g(n)$

- $\Omega(g(n))$
  - the set of functions with asymptotic behavior greater than or equal to $g(n)$
  - Lower-bounded by a constant times $g$ for large enough values $n$
  - $f \in \Omega(g(n)) \equiv \exists c > 0. \exists n_0 > 0. \forall n \geq n_0. f(n) \geq c \cdot g(n)$

- $\Theta(g(n))$
  - "Tightly" within constant of $g$ for large $n$
  - $\Omega(g(n)) \cap O(g(n))$
Asymptotic Notation Example

• Show: \(10n + 100 \in O(n^2)\)
  • **Technique:** find values \(c > 0\) and \(n_0 > 0\) such that \(\forall n > n_0. 10n + 100 \leq c \cdot n^2\)
  • **Proof:**
Asymptotic Notation Example

• Show: $10n + 100 \in O(n^2)$
  • Technique: find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 10n + 100 \leq c \cdot n^2$
  • Proof: Let $c = 10$ and $n_0 = 6$. Show $\forall n \geq 6.10n + 100 \leq 10n^2$
    $10n + 100 \leq 10n^2$
    $\equiv n + 10 \leq n^2$
    $\equiv 10 \leq n^2 - n$
    $\equiv 10 \leq n(n - 1)$
    This is True because $n(n - 1)$ is strictly increasing and $6(6 - 1) > 10$
Asymptotic Notation Example

• Show: $13n^2 - 50n \in \Omega(n^2)$
  • **Technique:** find values $c > 0$ and $n_0 > 0$ such that $\forall n \geq n_0. 13n^2 - 50n \geq c \cdot n^2$
  • **Proof:**
Asymptotic Notation Example

• Show: \(13n^2 - 50n \in \Omega(n^2)\)
  
  • **Technique:** find values \(c > 0\) and \(n_0 > 0\) such that \(\forall n \geq n_0. 13n^2 - 50n \geq c \cdot n^2\)
  
  • **Proof:** let \(c = 12\) and \(n_0 = 50\). Show \(\forall n \geq 50. 13n^2 - 50n \geq 12n^2\)
    
    \[
    13n^2 - 50n \geq 12n^2
    \]
    
    \[
    \equiv n^2 - 50n \geq 0
    \]
    
    \[
    \equiv n^2 \geq 50n
    \]
    
    \[
    \equiv n \geq 50
    \]
    
    This is certainly true \(\forall n \geq 50\).
Asymptotic Notation Example

• Show: $n^2 \notin O(n)$
Asymptotic Notation Example

• To Show: \( n^2 \notin O(n) \)
  
  • **Technique:** Contradiction
  
  • **Proof:** Assume \( n^2 \in O(n) \). Then \( \exists c, n_0 > 0 \) s. t. \( \forall n > n_0, n^2 \leq cn \)
  
  Let us derive constant \( c \). For all \( n > n_0 > 0 \), we know:
  
  \[
  cn \geq n^2, \\
  c \geq n.
  \]

  Since \( c \) is lower bounded by \( n \), \( c \) cannot be a constant and make this True.
  
  Contradiction. Therefore \( n^2 \notin O(n) \).
Gaining Intuition

• When doing asymptotic analysis of functions:
  • If multiple expressions are added together, ignore all but the “biggest”
    • If $f(n)$ grows asymptotically faster than $g(n)$, then $f(n) + g(n) \in \Theta(f(n))$
  • Ignore all multiplicative constants
    • $f(n) + c \in \Theta(f(n))$ for any constant $c \in \mathbb{R}$
  • Ignore bases of logarithms
  • Do NOT ignore:
    • Non-multiplicative and non-additive constants (e.g. in exponents, bases of exponents)
    • Logarithms themselves

• Examples:
  • $4n + 5$
  • $0.5n \log n + 2n + 7$
  • $n^3 + 2^n + 3n$
  • $n \log(10n^2)$
More Examples

• Is each of the following True or False?
  • $4 + 3n \in O(n)$
  • $n + 2 \log n \in O(\log n)$
  • $\log n + 2 \in O(1)$
  • $n^{50} \in O(1.1^n)$
  • $3^n \in \Theta(2^n)$
Common Categories

- $O(1)$ “constant”
- $O(\log n)$ “logarithmic”
- $O(n)$ “linear”
- $O(n \log n)$ “log-linear”
- $O(n^2)$ “quadratic”
- $O(n^3)$ “cubic”
- $O(n^k)$ “polynomial”
- $O(k^n)$ “exponential”
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).