Warm Up: Practice with $O$

Show $n^2 + 3n$ belongs to $O(4n^3)$
More Examples

• Is each of the following True or False?
  • $4 + 3n \in O(n)$
  • $n + 2 \log n \in O(\log n)$
  • $\log n + 2 \in O(1)$
  • $n^{50} \in O(1.1^n)$
  • $3^n \in \Theta(2^n)$
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:
  • \( 4n + 5 \)
  • \( 0.5n \log(n) + 2n + 7 \)
  • \( n^3 + 2^n + 3n \)
  • \( n \log(10n^2) \)
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).
Beware!

- Worst case, Best case, amortized are ways to select a function
- $O$, $\Omega$, $\Theta$ are ways to compare functions
- You can mix and match!
- The following statements totally make sense!
  - The worst case running time of my algorithm is $\Omega(n^3)$
  - The best case running time of my algorithm is $O(n)$
  - The best case running time of my algorithm is $\Theta(2^n)$
Recursive Binary Search

```java
class RecursiveBinarySearch {
    public static boolean binarySearch(List<Integer> lst, int k) {
        return binarySearch(lst, k, 0, lst.size());
    }

    private static boolean binarySearch(List<Integer> lst, int k, int start, int end) {
        if (start == end) {
            return false;
        }
        int mid = start + (end - start) / 2;
        if (lst.get(mid) == k) {
            return true;
        } else if (lst.get(mid) > k) {
            return binarySearch(lst, k, start, mid);
        } else {
            return binarySearch(lst, k, mid + 1, end);
        }
    }
}
```
Analysis of Recursive Algorithms

• Overall structure of recursion:
  • Do some non-recursive “work”
  • Do one or more recursive calls on some portion of your input
  • Do some more non-recursive “work”
  • Repeat until you reach a base case

• Running time: \( T(n) = T(p_1) + T(p_2) + \cdots + T(p_x) + f(n) \)
  • The time it takes to run the algorithm on an input of size \( n \) is:
  • The sum of how long it takes to run the same algorithm on each smaller input
  • Plus the total amount of non-recursive work done at that step

• Usually:
  • \( T(n) = a \cdot T \left( \frac{n}{b} \right) + f(n) \)
    • Called “divide and conquer”
  • \( T(n) = T(n - c) + f(n) \)
    • Called “chip and conquer”
How Efficient Is It?

- $T(n) = 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right)$
- Base case: $T(1) = 1$

$T(n) = "cost"$ of running the entire algorithm on an array of length $n$
Let’s Solve the Recurrence!

\[ T(1) = 1 \]
\[ T(n) = 1 + T\left(\frac{n}{2}\right) \]

Substitute until \( T(1) \)
So \( \log_2 n \) steps

\[ T(n) = \sum_{i=1}^{\log_2 n} 1 = \log_2 n \]
\[ T(n) \in \Theta(\log n) \]
Make our method “prettier”

- Draw a picture of the recursion
- Identify the work done per stack frame
- Add up all the work!
  - Sum is the answer!
  - In this case $\Theta(\log_2 n)$

The “Tree Method”

$$T(n) = T\left(\frac{n}{2}\right) + 1$$
Recursive Linear Search

```java
public static boolean linearSearch(List<Integer> lst, int k){
    return linearSearch(lst, k, 0, lst.size());
}

private static boolean linearSearch(List<Integer> lst, int k, int start, int end){
    if(start == end){
        return false;
    } else if(lst.get(start) == k){
        return true;
    } else{
        return linearSearch(lst, k, start+1, end);
    }
}
```
Make our method “prettier”

- Draw a picture of the recursion
- Identify the work done per stack frame
- Add up all the work!

Running time: $\Theta(n)$

$$T(n) = T(n - 1) + 1$$

Diagram:

```
 n
  ↓
 n-1
  ↓
 n-2
  ↓
 ...  
  ↓
  1
```
Recursive List Summation

```java
sum(list){
    return sum_helper(list, 0, list.size);
}
sum_helper(list, low, high){
    if (low == high){ return 0; }
    if (low == high - 1){ return list[low]; }
    middle = (high+low)/2;
    return sum_helper(list, low, middle) + sum_helper(list, middle, high);
}
```
Tree Method

\[ T(n) = 2T \left( \frac{n}{2} \right) + c \]

\[ \Rightarrow 2^i \cdot c \text{ work per level} \]

\[ \log_2 n \text{ levels of recursion} \]

\[ T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c \]

Red box represents a problem instance

Blue value represents time spent at that level of recursion
Recursive List Summation

\[ T(n) = \sum_{i=1}^{\log_2 n} 2^i \cdot c \]

\[ = c \cdot \sum_{i=1}^{\log_2 n} 2^i \]

\[ = c \left( \frac{1 - 2^{\log_2 n}}{1 - 2} \right) \]
Let’s do some more!

• For each, assume the base case is $n = 1$ and $T(1) = 1$
• $T(n) = 2T\left(\frac{n}{2}\right) + n$
• $T(n) = 2T\left(\frac{n}{2}\right) + n^2$
• $T(n) = 2T\left(\frac{n}{8}\right) + 1$
Tree Method

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

\( n \) work per level

\( \log_2 n \) levels of recursion

Red box represents a problem instance

Blue value represents time spent at that level of recursion

\[ T(n) = \sum_{i=1}^{\log_2 n} n \]
**Tree Method**

\[ T(n) = 2T\left(\frac{n}{2}\right) + n^2 \]

- Red box represents a problem instance
- Blue value represents time spent at that level of recursion

\[ \Rightarrow \text{?? work per level} \]

\[ \log_2 n \text{ levels of recursion} \]

\[ T(n) = \sum_{i=1}^{\log_2 n} \text{??} \]
Recursive List Summation

\[
T(n) = \sum_{i=1}^{\log_2 n} \frac{n^2}{2^i}
\]

\[
= n^2 \cdot \sum_{i=1}^{\log_2 n} \left(\frac{1}{2}\right)^i
\]
Tree Method

Red box represents a problem instance

Blue value represents time spent at that level of recursion

\[ T(n) = 2T \left( \frac{n}{8} \right) + 1 \]

\[ \Rightarrow 2^i \text{ work per level} \]

\[ \log_8 n \text{ levels of recursion} \]

\[ T(n) = \sum_{i=1}^{\log_8 n} 2^i \]
Recursive List Summation

\[ T(n) = \sum_{i=1}^{\log_8 n} 2^i \]

\[ = \left( \frac{1 - 2^{\log_8 n}}{1 - 2} \right) \]

\[ = 2^{\log_8 n} - 1 \]

\[ = n^{\log_8 2} = \frac{1}{n^3} \]
Finite Geometric Series

If $a > 1$

The series multiplied by $a$

$\sum_{i=0}^{L} a^i = (1 + a + a^2 + \cdots + a^L)\cdot a$

The series

$\sum_{i=0}^{L} a^i = (1 + a + a^2 + \cdots + a^L)$

The next term in the series

$\sum_{i=0}^{L} a^i = a^{L+1}$

The first term

$\sum_{i=0}^{L} a^i = 1$
Finite Geometric Series

If $a < 1$

$$\sum_{i=0}^{L} a^i$$

The series multiplied by $a$

$$(1 + a + a^2 + \cdots + a^L)a$$

The series

$$(1 + a + a^2 + \cdots + a^L)$$

The next term in the series $a^{L+1}$

The first term

1

Solve for the series