Algorithm Analysis III: Recursive
CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Aaron Johnston    Ethan Knutson    Nathan Lipiarski
Amanda Park      Farrell Fileas    Sam Long
Anish Velagapudi Howard Xiao    Yifan Bai
Brian Chan       Jade Watkins    Yuma Tou
Elena Spasova    Lea Quan
Announcements

❖ No lecture on Monday; MLK Day of Service
  ▪ Please go out there and volunteer to improve your communities
  ▪ DITs cancelled for Monday

❖ HW1 feedback form emailed to your @uw
  ▪ Help us make your homeworks better!

❖ Homeworks:
  ▪ The score you get in Gradescope is your “final” score. No style grading; late deductions already factored in.
  ▪ HW2 due Tuesday; check your Gradle now while the staff is still around!
Questions from Reading Quiz

❖ What’s the base for log N?

❖ What do you mean by “unrolling the recurrence”? How do you get \( T(N) = T(N / 2) + c \)?

❖ What is \( T(N) \)?
Lecture Outline

❖ Recursion

❖ Pattern #1: (Almost) Doubling the Input

❖ Pattern #2: Halving the Input

❖ Pattern #3: Constant-size Input and Doing Work
Recursion

Recursion

- An algorithm or a data structure that is defined in terms of itself
- Usually has a base case that doesn’t use recursion to terminate

Examples:

- Fibonacci:
  - $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
  - $\text{fib}(1) = 1$

- An ancestor is a person who is:
  - Your parent
  - Your parent’s ancestor

- Sourdough starter
  - A colony of microorganisms
  - Your last loaf’s starter
Lecture Outline

❖ Recursion

❖ Pattern #1: (Almost) Doubling the Input

❖ Pattern #2: Halving the Input

❖ Pattern #3: Constant-size Input and Doing Work
Asymptotic Analysis for Iterative Recursive Problems

❖ Case Analysis != Asymptotic Analysis

❖ Memorize these summations since they’re common:

\[
1 + 2 + 3 + 4 + \ldots + (N-1) = \frac{N(N-1)}{2} \in \Theta(N^2)
\]

\[
1 + 2 + 4 + 8 + \ldots + 2^{\text{floor}(\log_2 N)} = 2N - 1 \in \Theta(N)
\]

❖ Strategies for finding an asymptotic bound:

- Use a geometric argument / visualizations
- Find an expression for the exact step count
- Write out examples
Find a tight bound for f’s runtime

A. 1  
B. \log N  
C. N  
D. N^2  
E. 2^N

```c
int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-1);
}
```
(Almost) Doubling the Input: f

```c
int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-1);
}
```

- \( T(0) = d \)
- \( T(1) = d \)
- \( T(2d) = T(d) + T(d) + c \)
- \( T(N) = 2\cdot T(N-1) + c \)

Runtime is recursively defined!
(Almost) Doubling the Input, Geometrically

- Draw one node for each function call
  - Why are we counting function calls and not operations?

```c
int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-1);
}
```

\[ f(4) = \]
(Almost) Doubling the Input, Examples

<table>
<thead>
<tr>
<th>N=</th>
<th>Total Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1 + 2</td>
</tr>
<tr>
<td>3</td>
<td>1 + 2 + 4</td>
</tr>
<tr>
<td>4</td>
<td>1 + 2 + 4 + 8</td>
</tr>
</tbody>
</table>

```
int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-1);
}
```
(Almost) Doubling the Input, Counting

- How many “levels” are in the function call tree? $N$

- How much work occurs at each level $L$? $2^{L-1}$

```c
int f(int n) {
    if (n <= 1)
        return 1;
    return f(n-1) + f(n-1);
}
```
(Almost) Doubling the Input, Counting

We know that f’s runtime looks like:

\[ T(4) = 1 + 2 + 4 + 8 \]
\[ T(N) = 1 + 2 + 4 + 8 + \ldots + 2^{N-1} \]

And we’ve previously seen:

\[ 1 + 2 + 4 + 8 + \ldots + 2^{\lfloor \log_2 Q \rfloor} = 2Q - 1 \]

If we let \( Q = 2^{N-1} \)

\[ T(N) = 1 + 2 + 4 + 8 + \ldots + 2^{N-1} = 2 \cdot 2^{N-1} - 1 \]
\[ T(N) = 2^N - 1 \in \Theta(2^N) \]
Out-of-Scope: Recurrence Solution

\[
C(1) = 1 \\
C(N) = 2C(N - 1) + 1 \\
= 2(2C(N - 2) + 1) + 1 \\
= 2(2(2C(N - 2) + 1) + 1) + 1 \\
= 2(\cdots 2 \cdot 1 + 1) + \cdots 1 \\
= 2(\cdots 2 \cdot 1 + 1) + \cdots 1 \\
= \underbrace{2^{N-1} + 2^{N-2} + \cdots + 1}_{N-1} = 2^N - 1 \in \Theta(2^N)
\]
Lecture Outline

❖ Recursion

❖ Pattern #1: (Almost) Doubling the Input

❖ **Pattern #2: Halving the Input**

❖ Pattern #3: Halving the Input *and* Doing Work
Halving the Input: Binary Search

```java
public static boolean binarySearch(int[] sorted, int findMe) {
    if (sorted.length == 0)
        return false;

    int mid = sorted.length / 2;
    if (findMe < sorted[mid])
        int[] subrange = Arrays.copyOfRange(sorted, 0, mid);
        return binarySearch(subrange, findMe);
    else if (x > sorted[mid])
        int[] subrange = Arrays.copyOfRange(sorted, mid, sorted.length);
        return binarySearch(subrange, findMe);
    else
        return true;
}
```
Halving the Input, Geometrically

- Draw one node for each call to binarySearch. If there is more than one case, draw a tree for each case.
Halving the Input, Counting

- \( T(1) = d \)
  \[ T(2) = T(1) + c \]
  ...
  \[ T(N) = T(N/2) + c \]

- **Worst case**: keep halving the input size until you reach 0, which takes \( \log_2 N \) “halve” operations
  - In other words, there are \( \log_2 N \) function calls or \( \log_2 N \) layers in our function call tree

- Worst case \( T(N) \in \Theta(\log_2 N) \)
**Halving the Input, Counting**

- **Worst case**: keep halving the input size until you reach 0, which takes $\log_2 N$ “halve” operations

- Similar to $f$, there is a constant amount of work at each recursive step

<table>
<thead>
<tr>
<th>Case</th>
<th>Big-Theta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Worst</td>
<td>$\Theta(\log_2 N)$</td>
</tr>
<tr>
<td>Overall</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Lecture Outline

❖ Recursion

❖ Pattern #1: (Almost) Doubling the Input

❖ Pattern #2: Halving the Input

❖ Pattern #3: Constant-size Input and Doing Work
MergeSort: The Merge Operation

Given **two sorted arrays**, the merge operation combines them into a single sorted array by successively copying the smallest item from the two arrays into a target array.

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>6</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
What is the runtime of merge, specified in terms of N, the total number of items?

1. $\Theta(1)$
2. $\Theta(\log_2 N)$
3. $\Theta(N)$
4. $\Theta(N \log_2 N)$
5. $\Theta(N^2)$
MergeSort

- MergeSort algorithm is recursive
  - If array is of size 1, return
  - MergeSort the left half
  - MergeSort the right half
  - Merge the two sorted halves

```java
void mergeSort(int[] arr) {
    if (arr.length == 1)
        return;

    int mid = arr.length / 2;
    int[] left = Arrays.copyOfRange(arr, 0, mid);
    int[] right = Arrays.copyOfRange(arr, mid, unsorted.length);

    mergeSort(left);
    mergeSort(right);
    System.arraycopy(left, 0, arr, 0, left.length);
    System.arraycopy(right, 0, arr, left.length, right.length);
}
```
MergeSort

❖ Geometrically:
  ▪ How many *layers* are in our function call tree?
    • Compare with BinarySearch
      7 layers, or \( \log_2 N \)
  ▪ How much work is done at each layer?
    • Compare with \( f \)
      \( 64, \) or \( N \)

❖ Counting:
  ▪ \( T(1) = d \)
  ▪ \( T(2) = T(1) + T(1) + c \)
  ▪ ... 
  ▪ \( T(N) = 2T(N/2) + c \)
Counting Calls vs. Work-per-Layer

f: work for each call is constant, so can count the number of calls

\[ R(N) \in \Theta(2^N) \]

MergeSort: work for each call is \textit{variable}. However, the work \textit{per layer} is the same

\[ R(N) = \Theta(N \cdot \log N) \]
# Linear vs Linearithmic (N log N) vs Quadratic

**Table 2.1** The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th></th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>$&lt; 1$ sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

Asymptotic analysis deals with infinities, not specific inputs (e.g., 1, 2, 7, 4) or classes of input (e.g., best case).
tl;dr Asymptotic Analysis for Iterative Recursive Problems

❖ Case Analysis != Asymptotic Analysis

❖ Memorize these summations since they’re common:

\[
1 + 2 + 3 + 4 + \ldots + (N-1) = \frac{N(N-1)}{2} \in \Theta(N^2)
\]

\[
1 + 2 + 4 + 8 + \ldots + 2^{\text{floor}(\log_2 N)} = 2N - 1 \in \Theta(N)
\]

\[
1 + 2 + 4 + 8 + \ldots + 2^N = 2^{N+1} - 1 \in \Theta(2^N)
\]

❖ Strategies for finding an asymptotic bound:
   - Use a geometric argument / visualizations
   - Find an expression for the exact step count
   - Write out examples