Algorithm Analysis III: Recurrences

CSE 332 Spring 2021

Instructor: Hannah C. Tang
Guest Lecturer: Kris Wong

Teaching Assistants:

Aayushi Modi
Aashna Sheth
Frederick Huyan
Hamsa Shankar
Khushi Chaudhari
Kris Wong
Logan Milandin
Nachiket Karmarkar
Patrick Murphy
Richard Jiang
Winston Jodjana

Announcements

- Project 1 Checkpoint tomorrow
  - Will release on Gradescope at midnight
  - No penalty if you haven't met the checkpoint

- Quiz 1 released next Tuesday!
  - We will be posting Quiz 1 from Autumn on the website this afternoon
  - Recordings of TA's walking through the problems from last quarter will be posted to Panopto

- Quiz 1 topics list
  - ADT vs Data Structure
  - Lists, Stacks, Queues
  - Sets, Dictionaries, Tries
  - Asymptotic Analysis
    - Big Oh, Theta, Omega
    - Formal Definitions
    - Amortization
    - Recurrences (Today!)
  - Priority Queues, Heaps (Friday / Monday)

gradescope.com/courses/256241

- Recall our `find()` method from several lectures back:

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

- Reimplement this method using recursion
  - Hint: you may need a helper function
- What is the base case for your recursive method?

Learning Objectives

- Understand when asymptotic analysis is useful and when it is not
- Be able to use both the expansion method and the tree method, to find the closed-form of a recurrence relation

Lecture Outline

- Algorithm Analysis III
  - **Closing thoughts on Big Oh**
  - Analyzing Recursive Code
    - Linear Search example
    - Binary Search example
    - Binary Linear Sum example

Closing Thoughts: Multivariable

- big-Oh can also use more than one variable
  - Example: can sum all elements of an n-by-m matrix in $O(nm)$

Closing Thoughts: When NOT to Use Big-Oh

- Asymptotic complexity (Big-Oh) describes behavior for **large *n*** and is independent of any computer / coding trick

- Asymptotic complexity for ***small n*** can be misleading
  - Example: $n^{1/10}$ vs. `log` n
    - Asymptotically, $n^{1/10}$ grows more quickly
    - But the "cross-over" point (n_0) is around $5*10^{17} \approx 2^{58}$; you might prefer $n^{1/10}$
  - Example: QuickSort vs InsertionSort
    - *Expected runtimes:* Quicksort is O(n log n) vs InsertionSort $O(n^2)$
    - In reality, InsertionSort is faster for small n's
    - (we'll learn about these sorts later)

Closing Thoughts: Timing vs. Big-Oh?

- *Evaluating an algorithm?* Use asymptotic analysis
- *Evaluating an implementation?* Timing can be useful
  - Either a hardware or a software implementation

- At the core of CS is a backbone of theory & mathematics
  - We've spent 2 ½ lectures talking about how to analyze the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of n

- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards

Algorithm Analysis Summary (1 of 2)

- **What are we analyzing**: Problem or the algorithm
- **Metric**: Time or space
    - Or power, or dollars, or …
- **Complexity Bounds**:
    - Describing curve shapes "at infinity"
        - 'c' allows us to ignore effect of multiplicative constants on curve shape
        - 'n_0' allows us to ignore effect of low-order terms on curve shape
    - *Upper bound*: big-O or little-o
    - *Lower bound*: big-Ω or little-ω
    - *Tight bound*: Θ

Algorithm Analysis Summary (2 of 2)

- **Complexity Cases**: two different dimensions:
  - The specific path through an algorithm for input of size N
    - ***Worst-case***: max # steps on "most challenging" input
    - ***Best-case***: min # steps on "easiest" input
    - *Average-case: varying definitions, typically not used in 332*
  - Number of executions considered
    - ***Single-execution***
    - ***Multiple-execution***: *amortized case* is only one of several techniques for combining executions

- Usually:
  - We analyze the algorithm's time complexity to understand its upper or tight bound for a single-execution's worst-case

Lecture Outline

- Algorithm Analysis III
  - Closing thoughts on Big Oh
  - Analyzing Recursive Code
    - **Linear Search example**
    - Binary Search example
    - Binary Linear Sum example

Analyzing Code

- Basic ***operations*** take "some amount of" ***constant time***
  - Arithmetic
  - Assignment
  - Access one Java field **or array index**
  - Etc.
  - (Again, this is an *approximation of reality*)

Consecutive statements	*Sum of time of each statement*
Loops	*Num iterations * time for loop body*
Recurrence	Solve recurrence equation
Function Calls	*Time of function's body*
Conditionals	*Time of condition + time of {slower/faster} branch*

Analyzing Iterative Code: Linear Search

Find an integer in a *sorted* array

2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k) {
  for(int i=0; i < arr.length; ++i)
    if(arr[i] == k)
      return true;
  return false;
}
```

Best case: 6 “ish” steps = $O(1)$

Worst case: 5 “ish” * (arr.length) + 1
= O(arr.length)

Runtime expression:
$T(n) = 1 + 5n$

Analyzing Recursive Code

- Computing runtimes gets interesting with recursion
- *Example*: compute something recursively on a list of size n. Conceptually, in each recursive call we:
  - Perform some amount of work; call it **w(n)**
  - Call the function recursively with a smaller portion of the list
- If reduce the problem size by 1 during each recursive call, the runtime expression is:
  - *Recursive case*: T(n) = w(n) + T(n-1)
  - *Base case*: T(1) = 5 = O(1)

$$T(n) = \begin{cases} 5 & \text{if } n=1 \\ w(n) + T(n-1) & \text{otherwise} \end{cases}$$

- Recursive part of the expression is the “recurrence relation”

Example Recursive Code: Summing an Array

- We can ignore **`sum`**'s contribution to the runtime since it's called once and does a constant amount of work

- Each time **`help`** is called, it does that a constant amount of work, and then calls **`help`** again on a problem one less than previous problem size

- Runtime Relation:

$$T(n) = \begin{cases} \sim 3 & \text{if } n=0 \\ \sim 5 & \text{otherwise} \end{cases}$$

```
int sum(int[] arr) {
  return help(arr, 0);
}

int help(int[]arr,int i) {
  if(i == arr.length)
    return 0;
  return arr[i] + help(arr, i+1);
}
```

Solving Recurrence Relations: Expansion (1 of 2)

- Now we just need to solve our recurrence relation
  - ie, reduce it to a closed form

- Use Technique #1: Expansion
  - Also known as "unrolling"

- Basically, we write it out to find the general-form expansion

$$\begin{aligned} T(n) &= 5 + (T(n-1)) && \text{expansion 1} \\ &= 5 + (5 + (T(n-2))) && \text{expn 2} \\ &= 5 + (5 + (5 + T(n-3))) && \text{expn 3} \\ &= \dots \quad \vdots && \vdots \\ &= 5k + T(n-k) && \text{expn k} \end{aligned}$$

Solving Recurrence Relations: Expansion (2 of 2)

- We have a general-form expansion:

  $T(n) = 5k + T(\underline{n-k})$

recursion stops when $n-k=0$

$n=k$

- And a base case:

  $T(0) = 3$

- When do we hit the base case?
  - When n-k = 0!

$T(n) = 5n + T(n-n)$

$= 5n + \underline{T(0)}$

$= 5n+3$

$T(n) \in O(n)$

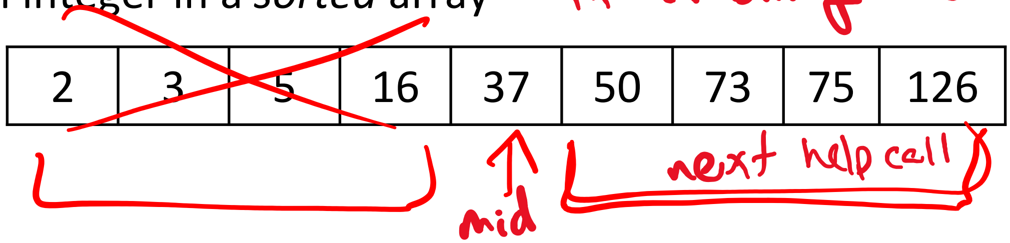
Lecture Outline

- Algorithm Analysis III
    - Closing thoughts on Big Oh
    - Analyzing Recursive Code
        - Linear Search example
        - **Binary Search example**
        - Binary Linear Sum example

Example Recursive Code: Binary Search

Find an integer in a *sorted* array

if finding 73



2	3	5	16	37	50	73	75	126

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k) {
  return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
  int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
  ~~if(lo==hi)          return false;~~
  if(arr[mid] == k)  return true;
  if(arr[mid] < k)   return help(arr, k, mid+1, hi);
  else               return help(arr, k, lo, mid);
}
```

ignore for simplicity

Example Recursive Code: Binary Search

$$T(n) = \begin{cases} 5 \text{ } \underline{\text{ish}} & \text{if } n=1 \\ 7\underline{\text{ish}} + T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ c_2 + T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

Base case:

5 ish ops

Recursive case:

7 ish ops + operations in recursive call on ½ input

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k) {
  return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
  int mid = (hi+lo)/2; // i.e., lo+(hi-lo)/2
  if(lo==hi)          return false;
  if(arr[mid] == k)   return true;
  if(arr[mid] < k)    return help(arr, k, mid+1, hi);
  else                return help(arr, k, lo, mid);
}
```

Technique #1: Expansion

1. Determine the recurrence relation and base case

$$T(n) = \begin{cases} c_1 & \text{if } n=1 \\ c_2 + T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

2. "Expand" the original relation to find the general-form expression *in terms of the number of expansions*

$$T(n) = \underbrace{c_2}_{1} + T(\tfrac{n}{2 \to 2^1}) \quad \text{expn 1}$$

$$= \underbrace{c_2 + (c_2}_{2} + T(\tfrac{n}{4 \to 2^2})) \quad \text{expn 2}$$

$$= \underbrace{c_2 + (c_2 + (c_2}_{3} + T(\tfrac{n}{8 \to 2^3}))) \quad \text{expn 3}$$

$$\vdots \qquad\qquad \vdots$$

$$= k \cdot c_2 + T(\tfrac{n}{2^k}) \quad \text{expn } k$$

3. Find the closed-form expression by setting *the number of expansions* to a value which reduces to a base case

Base Case: $\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = k$

$$T(n) = \log_2 n \cdot c_2 + T\left(\frac{n}{2^{\log_2 n}}\right)$$

$$= \log_2 n \cdot c_2 + T(1) = c_2 \log n + c_1$$

$$\in O(n)$$

Lecture Outline

- Algorithm Analysis III
  - Closing thoughts on Big Oh
  - Analyzing Recursive Code
    - Linear Search example
    - Binary Search example
    - **Binary Linear Sum example**

Summing an Array, Again (1 of 5)

Two “obviously” linear algorithms:

Iterative:

$O(n)$

```
int sum(int[] arr) {
  int ans = 0;
  for (int i=0; i < arr.length; ++i)
    ans += arr[i];
  return ans;
}
```

Recursive:

$O(n)$

```
int sum(int[] arr) {
  return help(arr,0);
}
int help(int[]arr,int i) {
  if (i == arr.length)
    return 0;
  return arr[i] + help(arr, i+1);
}
```

Summing an Array, Again (2 of 5)

- What about a <u>binary</u> version of **`sum`**?
  - Can we get a BinarySearch-like runtime?

```
int sum(int[] arr) {
   return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo == hi)    return 0;
   if(lo == hi-1)  return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr, lo, mid) + help(arr, mid, hi);
}
```

T(0) →

T(1) →

T(n) →

This happens TWICE!

Summing an Array, Again (3 of 5)

$$T(n) = \begin{cases} c_1 & \text{if } n = 0 \\ c_2 & \text{if } n = 1 \\ c_3 + \underline{\underline{2}}\,T(\frac{n}{2}) & \text{otherwise} \end{cases}$$

Expansion:

$$T(n) = c_3 + 2T(\tfrac{n}{2})$$

$$= c_3 + (2c_3 + 4T(\tfrac{n}{4}))$$

$$= c_3 + (2c_3 + (4c_3 + 8T(\tfrac{n}{8})))$$

$$= c_3 + (2c_3 + (4c_3 + (8c_3 + 16T(\tfrac{n}{16}))))$$

$$\vdots$$

$$= \sum_{i=0}^{k-1} 2^i \cdot c_3 + 2^k \cdot T(\tfrac{n}{2^k})$$

	# c_3's
Expn 1	$1 = 2^0$
Expn 2	$1+2 = 2^0+2^1$
Expn 3	$1+2+4 = 2^0+2^1+2^2$
Expn 4	$1+2+4+8 =$ $2^0+2^1+2^2+2^3$
Expn k	

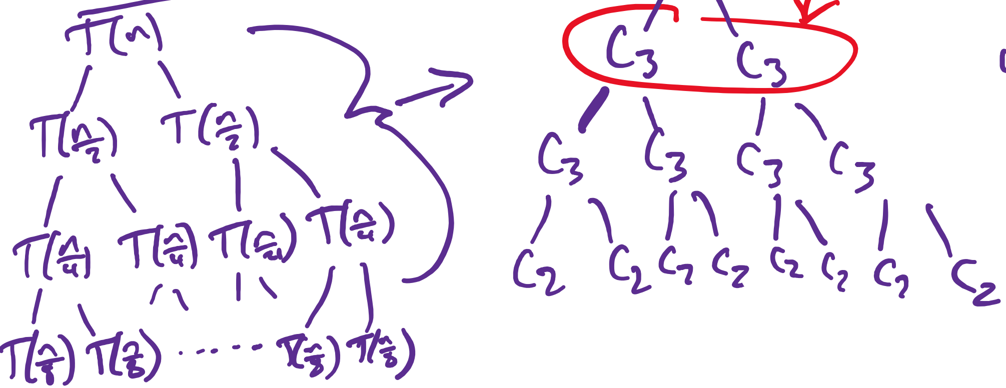
Technique #2: Tree Method

- Idea: We'll do the same reasoning, but give ourselves a visual to make the organization easier

$$T(n) \rightarrow \underline{C_3} + \underline{2T(\tfrac{n}{2})}$$

- We'll make a **tree**
  - Each node of the tree represents one recursive call
  - The children of that node are the new recursive calls made

Assume $n=8$

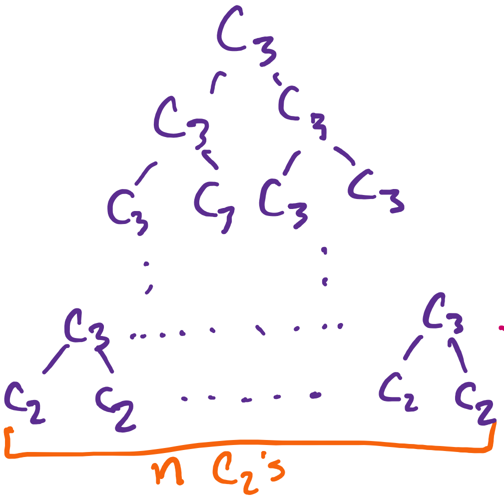


Work for all interior nodes: C_3

work for all $T(\frac{n}{8})$: C_2

$\leftarrow 8 \cdot C_2 \rightsquigarrow n \cdot C_2$

Summing an Array, Again (4 of 5)



Level (in tree)
0
1
2
⋮
i-1
i

nodes
$1 = 2^0$
$2 = 2^1$
$4 = 2^2$
⋮
2^{k-1}
2^k (k = height)

How many C_3's?

$k = \log_2 n$ (true for perfect binary trees like this)

$T(n) = C_2 \cdot n + \underline{\ ?\ }$

$T(n) = C_2 \cdot n + C_3 \cdot \sum_{i=0}^{\log n} 2^i$

$\# C_3\text{'s} = \sum_{i=0}^{\log n - 1} 2^i$

Closed Summation:

$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

$\sum_{i=0}^{\log n - 1} 2^i = 2^{\log n} - 1 = n - 1$

$T(n) = C_2 \cdot n + C_3(n-1) = (C_2 + C_3)n + C_3 \in O(n)$

Summing an Array, Again (5 of 5)

- Runtime is: O(n) ← makes sense, Can't sum all elements w/o looking at all elements
- Observation: it adds each number once while doing little else
  - Can't do better than O(n); have to read whole array!

```
int sum(int[] arr) {
   return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo == hi)   return 0;
   if(lo == hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Parallelism Teaser

- But suppose we could do two recursive calls *at the same time*

- If you have as much parallelism as needed, the recurrence becomes
  - $T(n) = O(1) + 1\ T(n/2)$ ← Same as Binary Search: $O(\log n)$

```
int sum(int[] arr) {
   return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
   if(lo == hi)   return 0;
   if(lo == hi-1) return arr[lo];
   int mid = (hi+lo)/2;
   return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Really Common Recurrences

Recurrence Relation	*Closed Form*	*Name*	*Example*
T(n) = O(1) + T(n/2)	O(log n)	Logarithmic	Binary Search
T(n) = O(1) + T(n-1)	O(n)	Linear	Sum (v1: “Recursive Sum”)
T(n) = O(1) + 2T(n/2)	O(n)	Linear	Sum (v2: “Recursive Binary Sum”)
T(n) = O(n) + T(n/2)	O(n)	Linear	
T(n) = O(n) + 2T(n/2)	O(n log n)	Loglinear	MergeSort
T(n) = O(n) + T(n-1)	$O(n^2)$	Quadratic	
T(n) = O(1) + 2T(n-1)	$O(2^n)$	Exponential	Fibonacci