Algorithm Analysis III: Recurrences

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- No quiz this week, but Q1 grades out shortly
- Exs 2-4 due tonight
- Ex 5 out and widely considered to be the hardest one!
- Project 1 due tomorrow night
  - Please request late days via tokens
  - Don't forget your Gradescope writeup
  - And please tag your questions in the writeup!

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
  - **Where We've Come / Where We're Going**
  - 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)

- Asymptotic complexity for ***specific implementations*** can also be misleading …

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
  - Where We've Come / Where We're Going
  - 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:

$T(n) = 5n + 1 \in O(n)$

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)

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

- 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:

  $T(0) = 3\text{(ish)}$ c_1

  $T(n) = 5\text{(ish)} + T(n-1)$ $c_2 + T(n-1)$

```
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);
}
```

base case is when we've fully processed the array (ie, remaining size is 0)

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

$$T(n) = 5 + T(n-1) \quad \text{1 expansion}$$
$$= 5 + 5 + T(n-2) \quad \text{2 expns}$$
$$= 5 + 5 + 5 + T(n-3) \quad \text{3 expns}$$
$$= \ldots$$
$$= 5k + T(n-k) \quad \text{k expns}$$

Solving Recurrence Relations: Expansion (2 of 2)

- We have a general-form expansion:

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

- And a base case:

  $T(0) = 3$

recursion stops when $n-k=0$

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

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

$= 5n + T(0)$

$= 5n + 3$

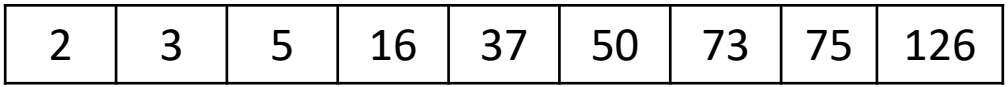
$T(n) \in O(n)$

Lecture Outline

- Algorithm Analysis III
  - Where We've Come / Where We're Going
  - 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



(assume elt is in array, so lo==hi case is never called)

```
// 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

Example Recursive Code: Binary Search

$T(1) = 7ish = c_1$

$T(n) = 9ish + T(\frac{n}{2})$

$= c_2 + T(\frac{n}{2})$

Base case: 7ish operations

Recursive case: 9ish ops + remaining half of array

```
// 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

$T(1) = 7$
$T(n) = 9 + T(\frac{n}{2})$

OR

$T(1) = c_1$
$T(n) = c_2 + T(\frac{n}{2})$

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

$T(n) = c_2 + T(\frac{n}{2})$ — 1st expn
$= c_2 + (c_2 + T(\frac{n}{4}))$ — 2nd expn
$= c_2 + c_2 + (c_2 + T(\frac{n}{8}))$ — 3rd expn
$= c_2 k + T(\frac{n}{2^k})$ — k^{th} expn

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

Base case when $\frac{n}{2^k} = 1$
$n = 2^k$
$\log_2 n = \log_2 2^k$
$\log n = k$

Therefore $T(n) = c_2 \log n + T(\frac{n}{2^{\log n}})$
$= c_2 \log n + T(1)$
$= c_2 \log n + c_1$
$\in O(\log n)$

Lecture Outline

- Algorithm Analysis III
  - Where We've Come / Where We're Going
  - 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 binary 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) → `if(lo == hi) return 0;`

T(1) → `if(lo == hi-1) return arr[lo];`

T(n) → `int mid = (hi+lo)/2;` `return help(arr, lo, mid) + help(arr, mid, hi);`

Summing an Array, Again (3 of 5)

Expansion Method

(1) Base case + Recurrence Relation

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

(2) Expansion + general form

$$T(n) = c_3 + 2T(\tfrac{n}{2})$$
$$= c_3 + (c_3 + c_3 + 4T(\tfrac{n}{4}))$$
$$= c_3 + c_3 + c_3 + (c_3 + c_3 + c_3 + c_3 + 8T(\tfrac{n}{8}))$$
$$\vdots$$
$$= \sum_{i=0}^{k-1} 2^i c_3 + 2^k T(\tfrac{n}{2^k})$$

1 expn
2 expns
3 expns
⋮
k expns

(3) Closed form (see slide 19)

Let $k = \log n$

$$T(n) = (n-1)c_3 + 2^{\log n} c_2$$
$$= (n-1)c_3 + c_2 n \in O(n)$$

Technique #2: Tree Method

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

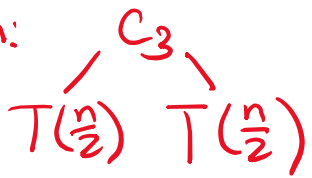
- We'll make a **tree**
  - Each node of the tree represents one recursive call
  - The children of that node are the new recursive calls made

Summing an Array, Again (4 of 5)

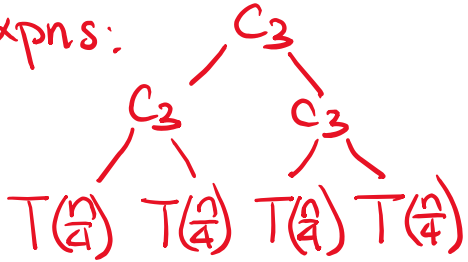
Tree Method

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

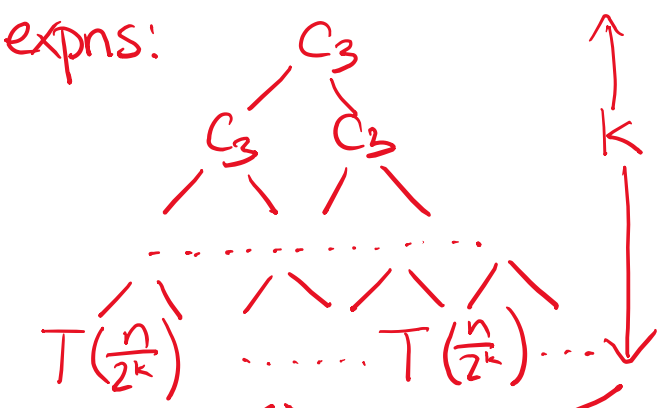
1 expn:



2 expns:



k expns:

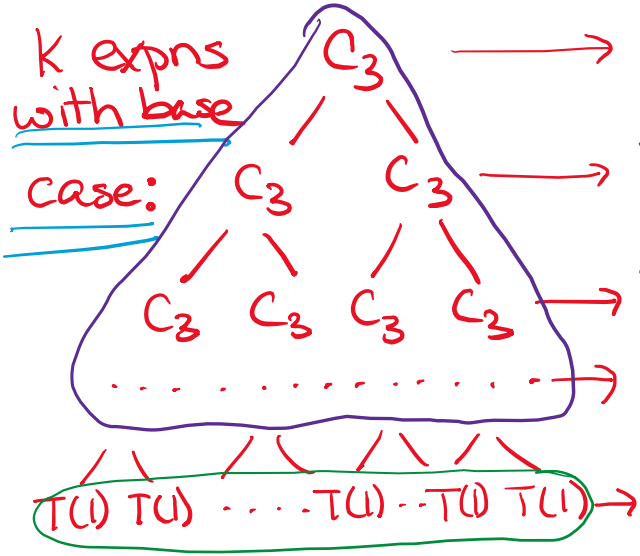


when does the recursion stop; i.e., when do these become leaves?

$T(0)$ or $T(1)$!

Tree Method (cont.)

k expns with base case:



level 0: $c_3 = 2^0 c_3$

level 1: $2c_3 = 2^1 c_3$

level 2: $4c_3 = 2^2 c_3$

level i: $2^i c_3$

level $\log n$: $2^{\log n} \cdot T(1) = n c_2$

★By inspection, we can see there are n leaves & $k = \log n$ levels★

So $T(n) = \underbrace{c_2 n}_{\text{leaves}} + \underbrace{??}_{\text{internal nodes}}$

$$= c_2 n + \sum_{i=0}^{\log n - 1} 2^i c_3$$

$$= c_2 n + (n-1) c_3 \in O(n)$$

Summing an Array, Again (5 of 5)

- Runtime is: O(n)
- 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)

```
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