Algorithm Analysis: Recurrences

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- EX02 out
  - If you haven't already, read the instructions and get the introductions done (due **tonight**)
- EX03 out
  - Widely considered the hardest exercise, but you have 2 weeks
  - *Start early*
- Project 1 checkpoint due tomorrow night
  - You won't be graded down for not finishing, but know that the second half is much more involved.
- Readings on website :D

Lecture Outline

- **Thinking about Runtime: Tries**
- Tree Review!!
- Recurrences
  - Analyzing Recursive Code
  - Expansion Method
    - Binary Search example
  - Tree Method
    - Binary Linear Sum example

gradescope

gradescope.com/courses/275833

Question 1 (follow along!):

What is the runtime for the following methods in a Trie?

- `insert`
- `find`
- `delete`

Tries again: Runtime?

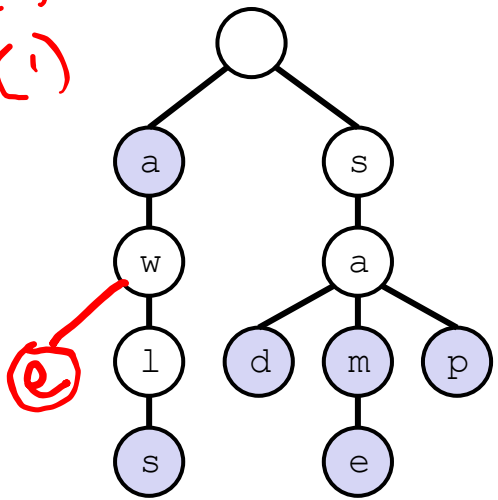
- `insert`(k)
- `find`(k)
- `delete`(k)

insert("awe")

O(k) ~ ϵ O(1)

O(1)

O(1)

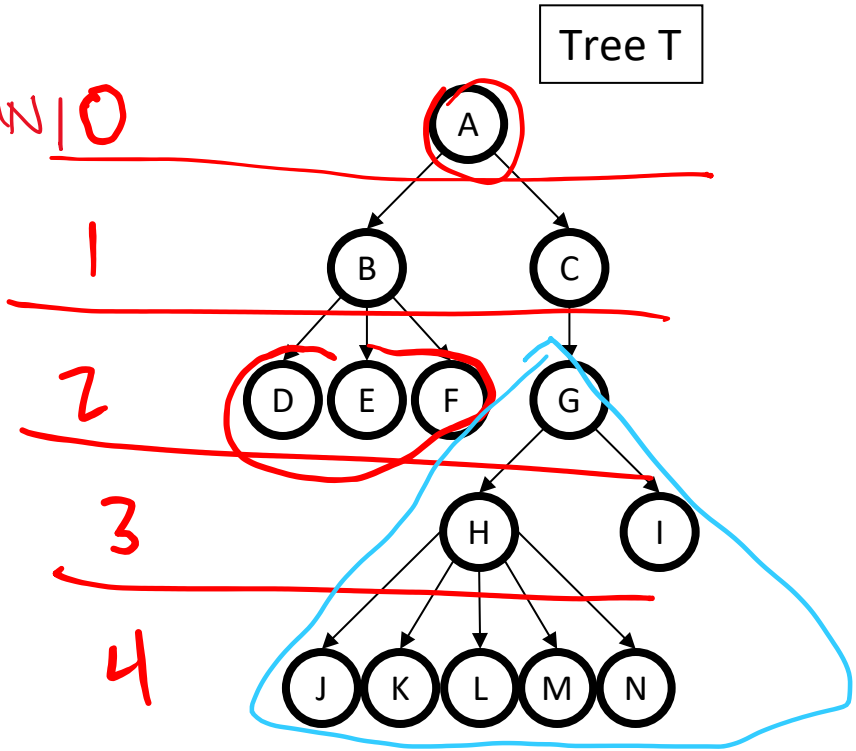


Lecture Outline

- Thinking about Runtime: Tries
- **Tree Review!!**
- Recurrences
  - Analyzing Recursive Code
  - Expansion Method
    - Binary Search example
  - Tree Method
    - Binary Linear Sum example

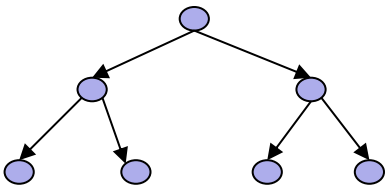
Review: Tree Terminology

- root(T): A
- leaves(T): D E F J K L M N
- children(B): D, E, F
- parent(H): G
- siblings(E): D, F
- ancestors(F): A, B
- descendants(G): H - N
- subtree(G): in blue
- height(G): 2
- height(T): 4
- degree(B): 3

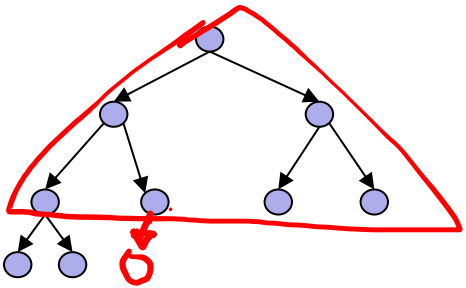


Types of Trees

Binary tree	Every node has ≤ 2 children
N-ary tree	Every node has ≤ n children
Perfect tree	Every row is completely full
Complete tree	All rows except possibly the bottom are completely full. The bottom row is filled from left to right



Perfect Tree



Complete Tree

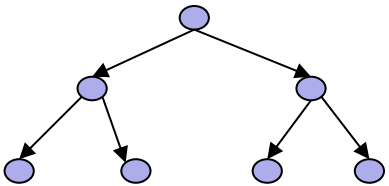
Perfect Binary Tree Properties

height: $N = 2^{h+1} - 1$

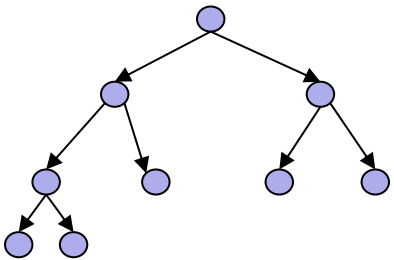
$N = 2^h$

$\lg N = h$

Height	Number of Nodes	Number of Leaves
1	3	2
2	7	4
3	15	8
4	31	16
h	$2^{h+1} - 1$	2^h



Perfect Tree



Complete Tree

Lecture Outline

- Thinking about Runtime: Tries
- Tree Review!!
- Recurrences
  - **Analyzing Recursive Code**
  - Expansion Method
    - Binary Search example
  - Tree Method
    - Binary Linear Sum example

Analyzing Code (Review)

- Basic ***operations*** take "some amount of" ***constant time***
  - Arithmetic (+,-,*,/, %)
  - Assignment (=)
  - Access one Java field or array index (Object.field, arr[i])
  - Etc.
  - (Again, this is an *approximation of reality*)

Consecutive statements	*Sum of time of each statement*
Loops	*Num iterations * time for loop body*
Recurrence (Recursive code?)	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;
    else if(arr[i] > k)
      return false;
  return false;
}
```

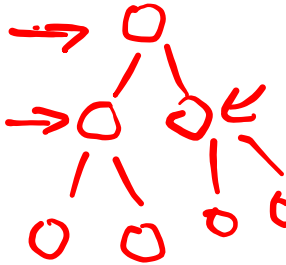
Best case: 8 "ish" steps ∈ O(1)

Worst case: 8 "ish" * (arr.length) + 1
∈ O(arr.length)

Runtime expression:

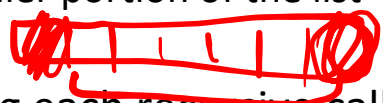
$T(n) = 8 \cdot n + 1$

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(1) = c_1
  T(n) = $c_2 + T(n-1)$

```
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);
}
```

gradescope

gradescope.com/courses/275833

Question 2: Survey

- How comfortable are you with Recursion right now?
- What's your best guess for how we'll find a Big O bound from our piecewise function?

Solving Recurrence Relations: Expansion (1 of 2)

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

when is the base case?

$n-k=1$

$k=n-1$

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

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

expn #

$$T(n) = 5 + T(n-1)$$
$$= 5 + 5 + T(n-2)$$
$$= 5 + 5 + 5 + T(n-3)$$
$$= \ldots$$
$$= 5k + T(n-k)$$

1
2
3
⋮
k

$\Theta(n)$

$T(n) = 5n - 5 + 3$

$n-k=1$ → $5(n-1) + T(n-(n-1))$ → $T(1)$

Solving Recurrence Relations: Expansion (2 of 2)

- We have a general-form expansion:

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

- And a base case:

  $$T(0) = 3$$

- When do we hit the base case?
  - When n-k = 1

Lecture Outline

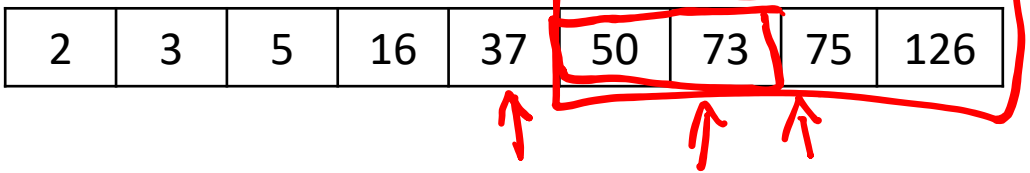
- Thinking about Runtime: Tries
- Tree Review!!
- Recurrences
  - Analyzing Recursive Code
  - **Expansion Method**
    - Binary Search example
  - Tree Method
    - Binary Linear Sum example

Example Recursive Code: Binary Search

O(log n)

Find an integer in a *sorted* array

k=73




```
// 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);
}
```

Example Recursive Code: Binary Search

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

Base case:

$$T(1) = c_1$$

Recursive case:

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

```
// 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*

$$T(n) = c_2 + T(\tfrac{n}{2})$$
$$= c_2 + (c_2 + T(\tfrac{n}{4}))$$
$$= c_2 + (c_2 + (c_2 + T(\tfrac{n}{8})))$$
$$\vdots$$
$$= k \cdot c_2 + T(\tfrac{n}{2^k})$$

expn #
1
2
3
⋮
k

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

Base Case?
when $\frac{n}{2^k} = 1$
↳ $k = \log n$

$$T(n) = k \cdot c_2 + T(\tfrac{n}{2^k})$$
$$= \log n \cdot c_2 + T(\tfrac{n}{2^{\log n}}) \rightarrow T(1)$$
$$= \log n \cdot c_2 + c_1$$
$$\in \Theta(\log n)$$

Lecture Outline

- Thinking about Runtime: Tries
- Tree Review!!
- Recurrences
  - Analyzing Recursive Code
  - Expansion Method
    - Binary Search example
  - **Tree Method**
    - Binary Linear Sum example

Summing an Array, Again

Two "obviously" linear algorithms:

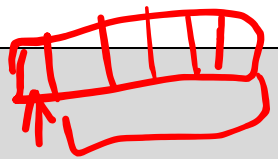
Iterative:

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)



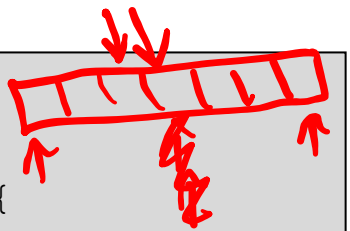
```
int sum(int[] arr) {
  return help(arr,0);
}
int help(int[]arr,int i) {
  if (i == arr.length)
    return 0;
  return arr[i] + help(arr, i+1);
}
```

Summing an Array, Again

- What about a binary version of **sum**?
  - Can we get a BinarySearch-like runtime?

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

```
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);
}
```



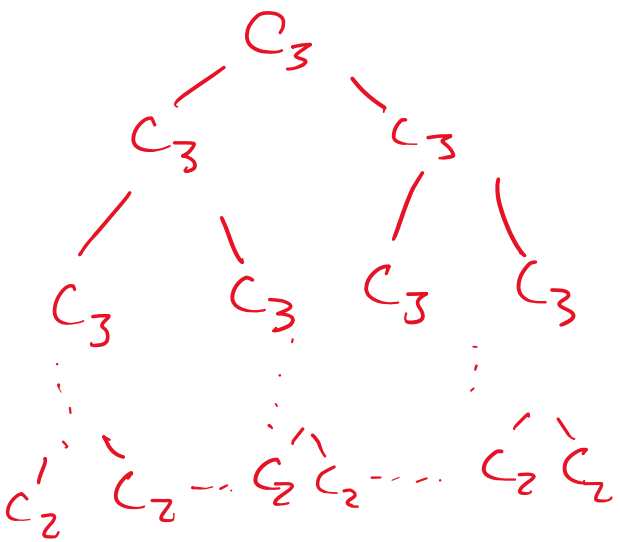
Technique #2: Tree Method

- 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)

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



lvl	# nodes	work
0	1	c_3
1	2	$2 \cdot c_3$
2	4	$4 \cdot c_3$
i	2^i	$2^i c_3$

levels of c_3's: $\log n - 1$

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

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

$= c_3 \cdot 2^{\log n - 1 + 1} - 1$

$= c_3(n-1)$

$T(n) = n \cdot c_2 + c_3(n-1)$

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

Summing an Array, Again

- Runtime is:

$\Theta(n)$

- 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