CSE 332
Data Structures & Parallelism

Recurrence Relations

Melissa Winstanley

Spring 2024

Today

- Analyzing Recursive Code
- Solving Recurrences

Analyzing code

Basic operations take "some amount of" constant time

- Arithmetic
- Assignment
- Access one Java field or array index
- Etc.

(This is an *approximation of reality*: a very useful "lie".)

Consecutive statements	Sum of time of each statement
Loops	Num iterations * time for loop body
Conditionals	Time of condition plus time of slower branch
Function Calls	Time of function's body
Recursion	Solve recurrence equation

Linear search

5	8	13	42	75	79	88	90	95	99
0	1	2	3	4	5	6	7	8	9

```
// 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 steps * (arr.length) = O(n)

Analyzing Recursive Code

- Computing run-times gets interesting with recursion
- Say we want to perform some computation 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
- So, if we do w(n) work per step, and reduce the problem size in the next recursive call by 1, we do total work:

  $$\mathbf{T(n)=w(n)+T(n-1)}$$

- With some base case, like T(1)=5=O(1)

Example Recursive code: sum array

Recursive:

- Recurrence is some constant amount of work O(1) done **n** times

```
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);
}
```

Each time help is called, it does that O(1) amount of work, and then calls help again on a problem one less than previous problem size

Recurrence Relation: $T(n) = c_1 + T(n-1)$
Base case? $T(0) =$

Today

- Analyzing Recursive Code
- Solving Recurrences

Solving Recurrence Relations

- Say we have the following recurrence relation:

```
T(n) = 6 “ish” + T(n-1)
T(1) = 9 “ish” base case
```

Solving Recurrence Relations

- Say we have the following recurrence relation:

```
T(n) = 6 “ish” + T(n-1)
T(1) = 9 “ish” base case
```

- Now we just need to solve it; that is, reduce it to a closed form.
- Start by writing it out:

```
T(n) = 6 + T(n-1)
     = 6 + 6 + T(n-2)
     = 6 + 6 + 6 + T(n-3)
     = 6 + 6 + 6 +…+ 6 + T(1) = 6 + 6 + 6 +…+ 6 + 9
     = 6k + T(n-k)
```

Solving Recurrence Relations

```
T(n) = 6k + T(n-k)
```

- We set k equal to n-1, because in order to reach the base case (1):

```
n - k = 1
n     = 1 + k
n - 1 = k
```

- We expanded it out n-1 times, so

```
T(n) = 6k + T(n-k)

     = 6(n-1) + T(1) = 6(n-1) + 9

     = 6n + 3 = O(n)
```

We'll usually just use a constant (eg c_1 or c_2) instead of the literal "6" and "9"

Solving Recurrence Relations: Unrolling Method

1. Write out your recurrence relation
2. Unroll it several times
3. Write the unrolled function in terms of some variable *k* (or *i*, whatever you like)
4. Figure out what *k* has to equal when you hit the base case (for instance, when you reach T(1)).
5. Solve!

Binary Search

```
// 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);
}
```

Recurrence Relation:

Base Case:

Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?

   $T(n) = c_2 + T(n/2)$ $\qquad$ $T(1) = c_1$

2. "Expand" the original relation to find an equivalent general expression in terms of the number of expansions.

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

Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?

   $T(n) = c_2 + T(n/2)$ $\qquad$ $T(1) = c_1$

2. "Expand" the original relation to find an equivalent general expression in terms of the number of expansions.

   $T(n) = c_2 + c_2 + T(n/4)$
   $T(n) = c_2 + c_2 + c_2 + T(n/8)$
   $T(n) = c_2k + T(n/(2^k))$ $\qquad$ (where k is the number of expansions)

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

   $n/(2^k) = 1$ means $n = 2^k$ means $k = \log n$
   $T(n) = c_2 \log n + c_1$ (get to base case and do it)
   $T(n)$ is $O(\log n)$

Another example: a binary version of sum

```
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);
}
```

Recurrence?

Another example: a binary version of sum

```
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);
}
```

Recurrence:

$T(n) = c_1$ for n=0 and n=1

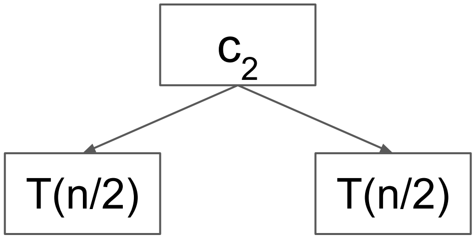
$T(n) = c_2 + 2T(n/2)$

Another example: a binary version of sum: tree method

Recurrence:

$T(n) = c_1$ for n=0 and n=1

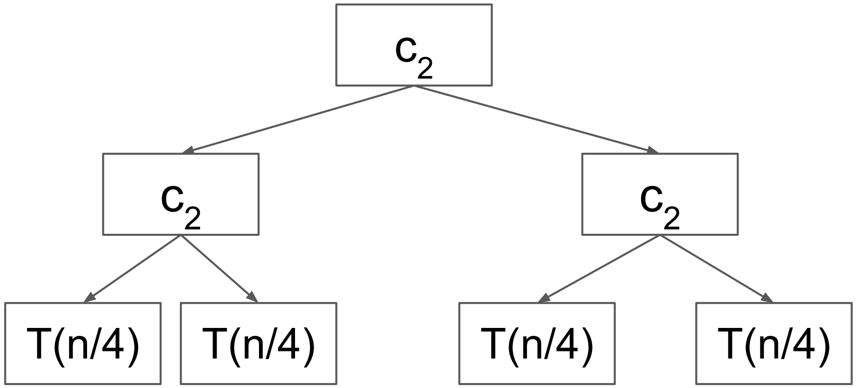
$T(n) = c_2 + 2T(n/2)$



Another example: a binary version of sum

Recurrence:

$T(n) = c_1$ for n=0 and n=1
$T(n) = c_2 + 2T(n/2)$

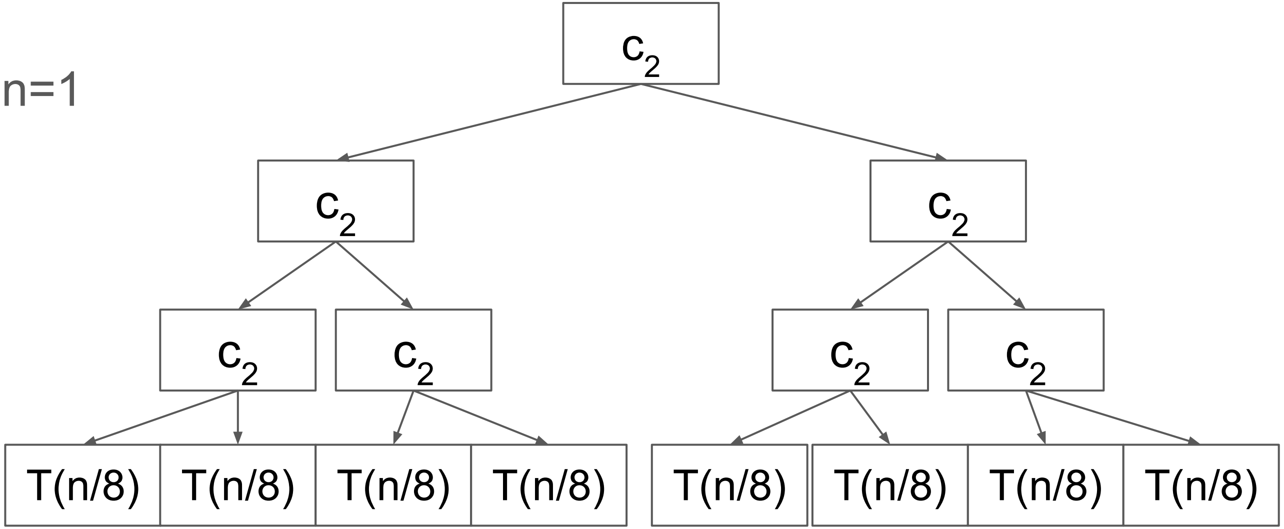


Another example: a binary version of sum

Recurrence:

$T(n) = c_1$ for n=0 and n=1

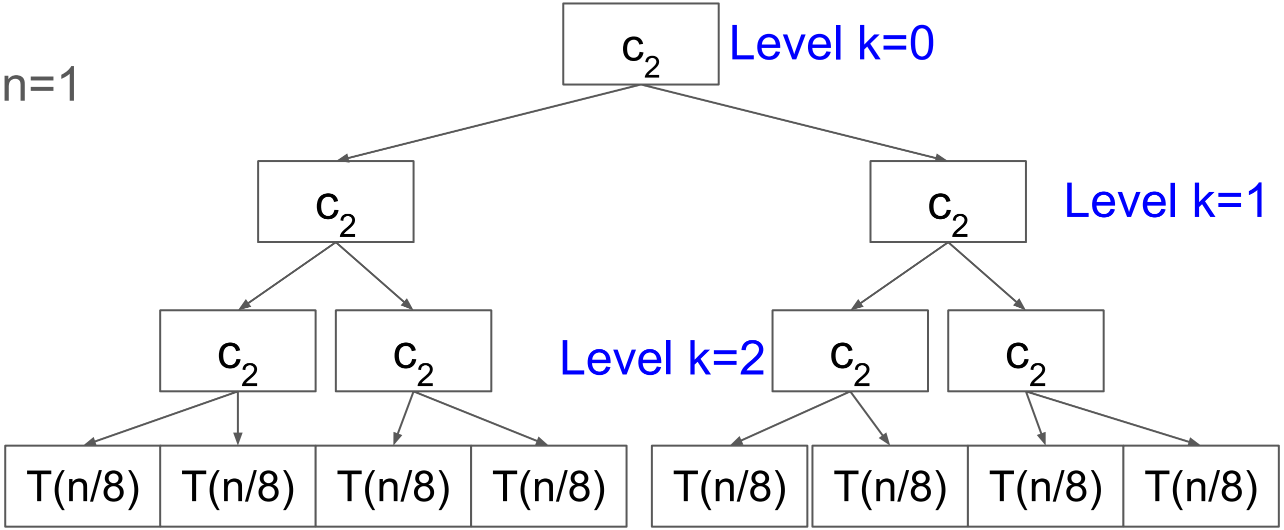
$T(n) = c_2 + 2T(n/2)$



Another example: a binary version of sum

Recurrence:

$T(n) = c_1$ for n=0 and n=1
$T(n) = c_2 + 2T(n/2)$



At each level, we have $2^k * c_2$ work, where k is the depth of the level (plus base case)

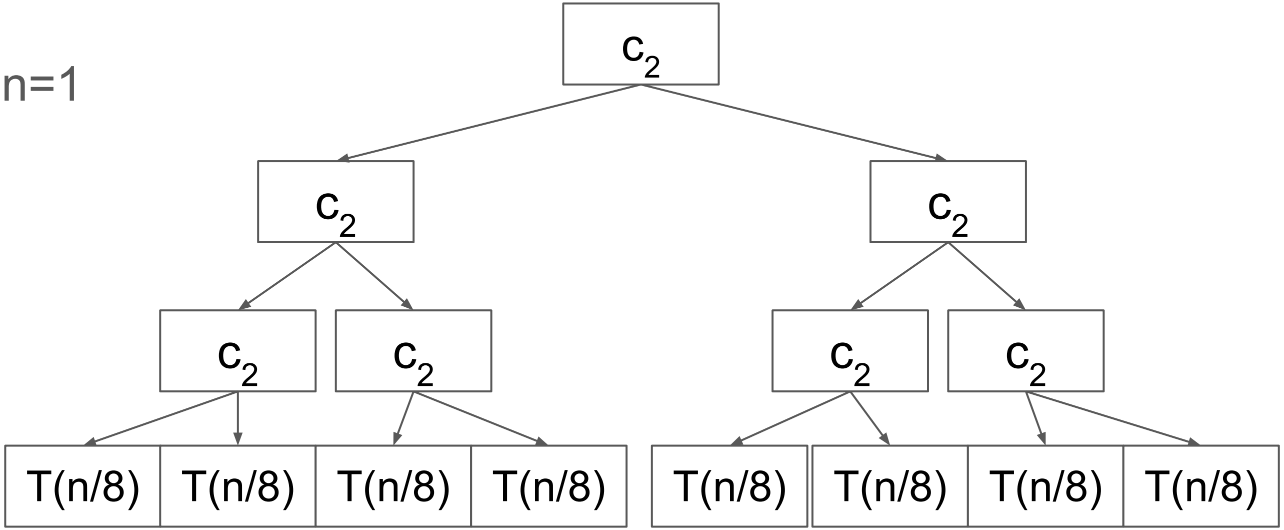
$c_2 * (1 + 2 + 4 + 8 + ...)$

$T(n) = c_2 * (\text{sum to k of } 2^k) + \text{base case}$

Another example: a binary version of sum

Recurrence:

$T(n) = c_1$ for n=0 and n=1
$T(n) = c_2 + 2T(n/2)$



What is the maximum k in terms of n? When does our recurred case hit T(1)?

$n / 2^k = 1$ $2^k = n$ $k = \log n$

Another example: a binary version of sum

Recurrence:

$T(n) = c_1$ for n=0 and n=1
$T(n) = c_2 + 2T(n/2)$

c_2 * (1 + 2 + 4 + 8 + ...) for log n times (plus the base case)

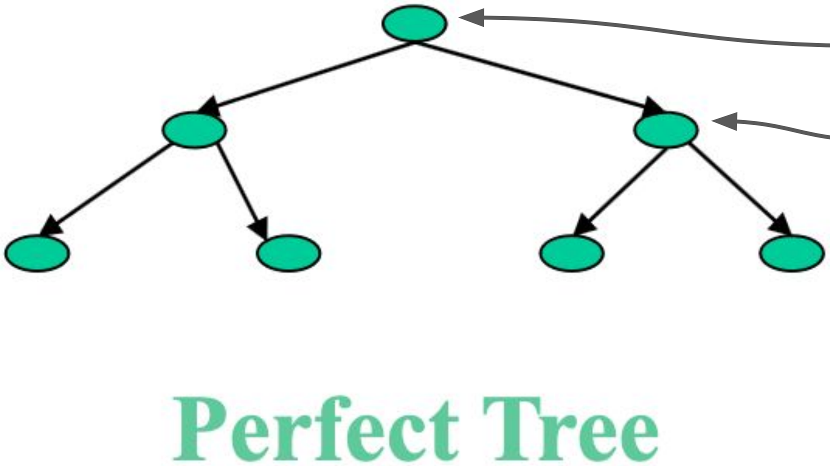
$$T(n) = c_1 n + \sum_{k=0}^{log(n)-1} c_2 2^k = c_1 n + c_2 \sum_{k=0}^{log(n)-1} 2^k$$

More on Perfect Trees

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

See Weiss 1.2.3 (p4)

Perfect tree: Every row is completely full



Height (h)	# nodes (n)	# leaves
0	1	1
1	3	2
2	7	4
3	15	8
h	$2^{h+1} - 1$	2^h

Another example: a <u>binary</u> version of sum

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

$$= c_1 n + c_2 \sum_{k=0}^{log(n)-1} 2^k$$

$$= c_1 n + c_2 (2^{log(n)-1+1} - 1)$$

$$= c_1 n + c_2 (2^{log(n)} - 1)$$

$$= c_1 n + c_2 (n - 1) \qquad = c_1 n + c_2 n - c_2$$

O(n)!!!

Solving Recurrence Relations: Tree Method

1. Write out your recurrence relation
2. Diagram it out as a *tree* several times
3. Write the unrolled function in terms of some variable *k* (or *i*, whatever you like)
4. Figure out what *k* has to equal when you hit the base case (for instance, when you reach T(1)).
5. Solve! Often using math.

SECTION!!!!!

Magic (i.e. log rules):

$$a^{\log_b c} = c^{\log_b a}$$

Finite Geometric Series:

$$\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$$

The Master Theorem

$$F(n) = \begin{cases} 2 & if\ n < 3 \\ 2\,F\left(\frac{n}{3}\right) + n + 2 & otherwise \end{cases}$$

It's still really hard to tell what the big-O is just by looking at it.

But fancy mathematicians have a formula for us to use!

Master Theorem

$$F(n) = \begin{cases} d & if\ n\ is\ at\ most\ some\ constant \\ \boldsymbol{a}\,F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^c) & otherwise \end{cases}$$

If	$\log_b a < c$	then	$T(n) \in \Theta(n^c)$
If	$\log_b a = c$	then	$T(n) \in \Theta(n^c \log n)$
If	$\log_b a > c$	then	$T(n) \in \Theta\left(n^{\log_b a}\right)$

a=2 b=3 and c=1

$y = \log_b x$ *is equal to* $b^y = x$

$\log_3 2 = x \Rightarrow 3^x = 2 \Rightarrow x \cong 0.63$

$\log_3 2 < 1$

We're in case 1

$T(n) \in \Theta(n)$

Slide thanks to Kasey Champion!

Understanding The Master Theorem

Master Theorem

$$F(n) = \begin{cases} d & \textit{if } n \textit{ is at most some constant} \\ \boldsymbol{a}\, F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \textit{otherwise} \end{cases}$$

If	$\log_b a < c$	then	$T(n) \in \Theta(n^c)$
If	$\log_b a = c$	then	$T(n) \in \Theta(n^c \log n)$
If	$\log_b a > c$	then	$T(n) \in \Theta\left(n^{\log_b a}\right)$

- **A** measures how many recursive calls are triggered by each method instance
- **B** measures the rate of change for input
- **C** measures the dominating term of the non recursive work within the recursive method
- **D** measures the work done in the base case

Understanding The Master Theorem

Master Theorem

$$F(n) = \begin{cases} d & \textit{if } n \textit{ is at most some constant} \\ \boldsymbol{a}\, F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \textit{otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

The log of a < c case

- Recursive case does a lot of non recursive work in comparison to how quickly it divides the input size
- Most work happens in beginning of call stack
- Non recursive work in recursive case dominates growth, n^c term

Understanding The Master Theorem

Master Theorem

$$F(n) = \begin{cases} d & \mathit{if\ n\ is\ at\ most\ some\ constant} \\ \boldsymbol{a}\,F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \mathit{otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

The log of a = c

- Recursive case evenly splits work between non recursive work and passing along inputs to subsequent recursive calls
- Work is distributed across call stack

Understanding The Master Theorem

Master Theorem

$$F(n) = \begin{cases} d & \text{if } n \text{ is at most some constant} \\ \boldsymbol{a}\, F\left(\dfrac{n}{\boldsymbol{b}}\right) + \Theta(n^c) & \text{otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

The log of a > c case

- Recursive case breaks inputs apart quickly and doesn't do much non recursive work
- Most work happens near bottom of call stack

Understanding The Master Theorem

Master Theorem

$$F(n) = \begin{cases} d & \textit{if } n \textit{ is at most some constant} \\ \boldsymbol{a}\, F\left(\frac{n}{\boldsymbol{b}}\right) + \Theta(n^{\boldsymbol{c}}) & \textit{otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

NOT A SUBSTITUTE FOR KNOWING HOW TO UNROLL / TREE!

We may ask you that on exams, etc

But the Master Theorem is good for checking your work