# CSE 332: Data Structures & Parallelism

# Lecture 5: Algorithm Analysis II

Ruth Anderson

Autumn 2017

# Today

- Finish up Binary Heaps
- Analyzing Recursive Code
- Solving Recurrences

# Analyzing code ("worst case")

Basic operations take "some amount of" constant time
- Arithmetic (fixed-width)
- Assignment
- Access one Java field **or array index**
- Etc.

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

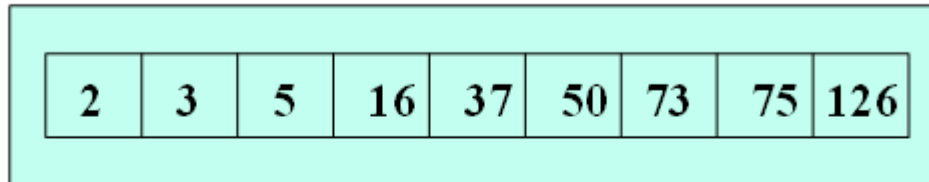| | |
|---|---|
| Consecutive statements | Sum of time of each statement |
| Conditionals | Time of condition plus time of slower branch |
| Loops | Num iterations * time for loop body |
| Function Calls | Time of function's body |
| Recursion | Solve *recurrence equation* |

## Linear search

| 2 | 3 | 5 | 16 | 37 | 50 | 73 | 75 | 126 |
|---|---|---|----|----|----|----|----|-----|

Find an integer in a *sorted* array

```java
// 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)
= $O(\text{arr.length})$

# 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:

$$T(n) = w(n) + T(n-1) \quad \longleftarrow \text{Recurrence Relation}$$

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

## Example Recursive code: sum array

$leng = 4$

$0, 1, 2, 3$

Recursive:
- Recurrence is
  some constant
  amount of work
  O(1) done $n$
  times

```java
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: $T(n) = O(1) + T(n-1)$

10/06/2017

6

Base Case: $T(0) = 3 = C = O(1)$

when problem size = 0

# Solving Recurrence Relations

- Say we have the following recurrence relation:

$$T(n) = 6 \text{ "ish"} + T(n-1)$$

$$T(1) = 9 \text{ "ish"} \qquad \leftarrow \text{base case}$$

- Now we just need to solve it; that is, reduce it to a closed form.

- Start by writing it out:

$$T(n) = 6 + T(n-1)$$
$$= 6 + 6 + T(n-2)$$
$$= 6 + 6 + 6 + T(n-3)$$
$$= 6 + 6 + 6 + \ldots + 6 + T(1) \;=\; 6 + 6 + 6 + \ldots + 6 + 9$$
$$= 6k + T(n-k)$$
$$= 6k + 9, \text{ where k is the \# of times we expanded } T()$$

- We expanded it out $n-1$ times, so

$$T(n) = 6k + T(n-k)$$
$$= 6(n-1) + T(1) = 6(n-1) + 9$$
$$= 6n + 3 = O(n)$$

Or When does $n-k=1$?
Answer: when $k=n-1$

## Binary search

| 2 | 3 | 5 | 16 | 37 | 50 | 73 | 75 | 126 |
|---|---|---|----|----|----|----|----|-----|

Find an integer in a *sorted* array

– Can also be done non-recursively but "doesn't matter" here

```java
// 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: $T(n) = C_2 + T\left(\frac{n}{2}\right)$

Base Case: $T(1) = C_1 = O(1)$

# Binary search

Best case: 9 "ish" steps = $O(1)$

Worst case: $T(n) = 10$ "ish" $+ T(n/2)$ where $n$ is `hi-lo`

- $O(\log n)$ where $n$ is `array.length`
- Solve *recurrence equation* to know that…

```java
// 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;
    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);
}
```

# Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$      $T(1) = 15$

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

$$T(n) = 10 + 10 + T\left(\frac{n}{4}\right)$$

$$T(n) = 10 + 10 + 10 + T\left(\frac{n}{8}\right) \qquad \frac{n}{2^k} = 1$$

$$T(n) = 10 \cdot K + T\left(\frac{n}{2^k}\right) \qquad 2^k = n$$

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

$$T(n) = 10 \cdot \log_2 n + 15 \qquad K = \log_2 n$$

$$= O(\log n)$$

## Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
   - $T(n) = 10 + T(n/2)$      $T(1) = 15$
2. "Expand" the original relation to find an equivalent general expression *in terms of the number of expansions*.
   - $T(n) = 10 + 10 + T(n/4)$
     $= 10 + 10 + 10 + T(n/8)$
     $= \ldots$
     $= 10k + T(n/(2^k))$    (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 $k = \log_2 n$
   - So $T(n) = 10 \log_2 n + 15$   (get to base case and do it)
   - So $T(n)$ is $O(\log n)$

# sum array again

Two "obviously" linear algorithms: $T(n) = O(1) + T(n-1)$

Iterative:

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

Recursive:
 – Recurrence is

$c + c + \ldots + c$

for $n$ times

```java
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);
}
```

## What about a *binary* version of sum?

```java
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);
}
```

## What about 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 is $T(n) = O(1) + 2T(n/2)$

- $1 + 2 + 4 + 8 + \ldots$ for $\log n$ times
- $2^{(\log n)} - 1$ which is proportional to $n$ (by definition of logarithm)

Easier explanation: it adds each number once while doing little else

"Obvious": You can't do better than $O(n)$ – have to read whole array

# Parallelism teaser

- But suppose we could do two recursive calls *at the same time*
  - *Like having a friend do half the work for you!*

```
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);
}
```

- If you have as many "friends of friends" as needed, the recurrence is now   $T(n) = O(1) + 1\,T(n/2)$
  - $O(\log n)$ : same recurrence as for `find`

# Really common recurrences

Should know how to solve recurrences but also recognize some
really common ones:

$T(n) = O(1) + T(n$-$1)$  linear
$T(n) = O(1) + 2T(n/2)$  linear
$T(n) = O(1) + T(n/2)$  logarithmic
$T(n) = O(1) + 2T(n$-$1)$  exponential
$T(n) = O(n) + T(n$-$1)$  quadratic
$T(n) = O(n) + T(n/2)$  linear
$T(n) = O(n) + 2T(n/2)$  $O(n \ \log n)$

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