# CSCI 2011: Recurrence Relations

Chris Kauffman

*Last Updated:*
*Wed Jul 25 16:27:35 CDT 2018*

# Logistics

## Reading: Rosen

- Now: 7.1 - 7.4
- Now/Next: 8.1 - 8.3

## Assignments

- A07: Post Later today
- Due Tuesday

# Basics

- A **Recurrence Relation** defines a relationship between elements of a sequence
- Ex: Fibonacci sequence *satisfies* the recurrence relationship

$$f_n = f_{n-1} + f_{n-2}$$

- Note: by some accounts, this is relationship is also the *definition* of Fibonacci but originally relationship was discovered to relate the growth of an idealized **rabbit population**
- Recurrence relations can be expressed in a variety of notations such as

$$f(n) = f(n-1) + f(n-2)$$
$$g(n) = 3g(n-1) + 4g(n/5)$$

# Algorithm Runtime and Recurrence Relations

- Will survey several algorithms and show that their runtimes can be represented as recurrence relations
- Motivates one to look at how to characterize recurrence relations so algorithm runtimes can be estimated

# Binary Search

- Array input of size `n`
- At iteration 0 do 8-10 ops to half input size to `n/2`
- At iteration 1 do 8-10 ops to half input size to `n/4`
- etc.
- In the worst case, `key` not in array so reduce to 0 elements
- Total Ops in worst case is described in recurrence

$$f(n) = f(n/2) + c$$

where $c$ is a constant

```
int binary_search(int a[], int key){
  int left=0, right=a.length-1;
  int mid = 0;
  while(left <= right){
    mid = (left+right)/2;
    if(key == a[mid]){
      return mid;
    }else if(key < a[mid]){
      right = mid-1;
    }
    else{
      left = mid+1;
    }
  }
  return -1;
}
```

# Exercise: Midpoint Search Recurrence

- Recursive search for an element in an unsorted array
- How is this different from binary search?
- Develop a recurrence relation for the number of ops used for an array of size $n$
- What do you expect the runtime of this algorithm to be?

```
// Determine if key is present in
// UNSORTED array a[] by repeated
// bisection search
boolean midpoint_search(int a[], int key){
  int left=0, right=length(a)-1;
  return helper(a,key,left,right);
}
boolean helper(int a[], int key,
               int left, int right)
{
  if(left > right){
    return false;
  }
  int mid = (left+right)/2;
  if(key == a[mid]){
    return true;
  }
  boolean foundL,foundR;
  foundL = helper(a,key,left,mid-1);
  foundR = helper(a,key,mid+1,right);
  reutrn foundL OR foundR;
}
```

# **Answers:** Midpoint Search Recurrence

- How is this different from binary search?
  - Midpoint search Goes BOTH left AND right
  - Binary search goes ONLY left OR right
- Develop a recurrence relation for the number of ops used
  - Halves array but goes both left/right
  - Uses a constant number of ops to halve
  - $f(n) = 2f(n/2) + c$
- What do you expect the runtime of this algorithm to be?
  - Visits every element of the array once so worst case linear $O(N)$

```
// Determine if key is present in
// UNSORTED array a[] by repeated
// bisection search
boolean midpoint_search(int a[], int key){
  int left=0, right=length(a)-1;
  return helper(a,key,left,right);
}
boolean helper(int a[], int key,
               int left, int right)
{
  if(left > right){
    return false;
  }
  int mid = (left+right)/2;
  if(key == a[mid]){
    return true;
  }
  boolean foundL,foundR;
  foundL = helper(a,key,left,mid-1);
  foundR = helper(a,key,mid+1,right);
  reutrn foundL OR foundR;
}
```

# Merge Sort

- Involves two phases
  - **Downward splitting** of an array into two halves, stops on reaching arrays of size 1
  - **Upward merging** of two sorted arrays into a larger array
- Will look at both briefly to establish it for analysis

# Exercise: Merge Operation

- Merges two sorted arrays into a combined sorted array
- Show how it works on `a[]={1,3,5,9};` `b[]={2,3,6}`
- What is the Runtime complexity of `merge()`?

```
// Merge sorted arrays a[] and b[] int res[]
// which is also sorted
void merge(int[] res, int[] a, int[] b){
  int ai=0, bi=0;
  for(int ri=0; ri<length(res); ri++){
    if(ai >= length(a)){     // a[] gone
      res[ri] = b[bi];
      bi++;
    }
    else if(bi >= length(b)){// b[] gone
      res[ri] = a[ai];
      ai++;
    }
    else if(a[ai]<=b[bi]){   // a[] smaller
      res[ri] = a[ai];
      ai++;
    }
    else{                   // b[] smaller
      res[ri] = b[bi];
      bi++;
    }
  }
}
```

# **Answers**: Merge Operation

- Merges two sorted arrays into a combined sorted array
- Show how it works on `a[]={1,3,5,9}; b[]={2,3,6}`
- What is the Runtime complexity of `merge()`?
  - Linear time in size of `res[]` array which is sum of lengths of `a[]` and `b[]`

```
// Merge sorted arrays a[] and b[] int res[]
// which is also sorted
void merge(int[] res, int[] a, int[] b){
  int ai=0, bi=0;
  for(int ri=0; ri<length(res); ri++){
    if(ai >= length(a)){     // a[] gone
      res[ri] = b[bi];
      bi++;
    }
    else if(bi >= length(b)){// b[] gone
      res[ri] = a[ai];
      ai++;
    }
    else if(a[ai]<=b[bi]){   // a[] smaller
      res[ri] = a[ai];
      ai++;
    }
    else{                   // b[] smaller
      res[ri] = b[bi];
      bi++;
    }
  }
}
```

# Exercise: Merge Sort, Split Down, Merge Up

- Merge sort works by recursing down halving arrays
- On reaching an array of size 0 or 1 recursion stops: these arrays are "sorted"
- Merge arrays on the way back up the recursion

```
void merge_sort(int[] a) {
  if (length(a) <= 1) {
    return;
  }
  int len = length(a);
  int[] left  = array_copy(a, 0,     len/2);
  int[] right = array_copy(a, len/2, len);

  mergeSort(left);
  mergeSort(right);

  merge(a, left, right);
}
```

## Questions

- What is the complexity of `array_copy()`?
- What is the complexity of `merge()`?
- Give a recurrence relation for the total operations done by `merge_sort()`

# **Answers:** Merge Sort, Split Down, Merge Up

- What is the complexity of `array_copy()`?
  - Linear $O(N)$
- What is the complexity of `merge()`?
  - From last exercise was $O(N)$
- Give a recurrence relation for the total operations done by `merge_sort()`
  - Recurse on half: $f(N/2)$
  - Recurse on both sides: $2f(N/2)$
  - Doing linear work at each step for copy/merge

$$f(N) = 2f(N/2) + a \cdot N + b$$

```
void merge_sort(int[] a) {
  if (length(a) <= 1) {
    return;
  }
  int len = length(a);
  int[] left  = array_copy(a, 0,     len/2);
  int[] right = array_copy(a, len/2, len);

  mergeSort(left);
  mergeSort(right);

  merge(a, left, right);
}
```

## The Master Theorem

Let $f$ be an increasing function that satisfies the recurrence relation

$$f(N) = af(N/b) + cN^d$$

- whenever $n = b^k$, with $k$ as a positive integer
- $a \geq 1$
- $b \geq 1$ and an integer
- $c > 0$ and $d \geq 0$ real numbers

Then $f(n)$ falls into one of the following complexity classes

$$\begin{array}{lll} \text{(Case 1)} & O(N^d) & \text{for } a < b^d \\ \text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\ \text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d \end{array}$$

- Proof is given as exercises in the text and we won't dwell on it
- Practical matter is that it allows MUCH easier analysis of recursive / divide-conquer algorithms

# Exercise: Analysis of Algorithms

## Master Theorem

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} \text{(Case 1)} & O(N^d) & \text{for } a < b^d \\ \text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\ \text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d \end{array}$$

## Binary Search

Total Ops in worst case is described in recurrence $f(N) = f(N/2) + q$ with $q$ a constant

- $a = 1, b = 2, d = 0$
- By master theorem, Case 2
- $O(N^0 \log N) = O(\log N)$

## Midpoint search

- $f(N) = 2f(N/2) + q$
- Analyze and determine Big-O op count

## Merge Sort

- $f(N) = 2f(N/2) + q \cdot N + w$
- Analyze and determine Big-O op count

# **Answers:** Analysis of Algorithms

## Master Theorem

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll}
\text{(Case 1)} & O(N^d) & \text{for } a < b^d \\
\text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\
\text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d
\end{array}$$

## Binary Search

Total Ops in worst case is described in recurrence $f(N) = f(N/2) + q$ with $q$ a constant

- $a = 1, b = 2, d = 0$
- By master theorem, Case 2
- $O(N^0 \log N) = O(\log N)$

## Midpoint search

- $f(N) = 2f(N/2) + q$
- $a = 2, b = 2, d = 0$
- Master Theorem Case 3

$$\begin{aligned} O(N^{\log_b(d)}) &= O(N^{\log_2(2)}) \\ &= O(N) \end{aligned}$$

## Merge Sort

- $f(N) = 2f(N/2) + q \cdot N + w$
- $a = 2, b = 2, d = 1$
- Master Theorem Case 2

$$O(N^d \log N) = O(N^1 \log N)$$

# Exercise: Other Kinds of Recurrence Relations

**Master Theorem**

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} \text{(Case 1)} & O(N^d) & \text{for } a < b^d \\ \text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\ \text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d \end{array}$$

Which of the following recurrence relations does the master theorem apply to and which does it not?

1. $f(n) = f(n-1) + 7$
2. $f(n) = 3 \cdot f(n-1)$
3. $f(n) = 4 \cdot (f(n/2)$
4. $f(n) = f(n-1) + f(n-2)$

# Exercise: Other Kinds of Recurrence Relations

**Master Theorem**

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} \text{(Case 1)} & O(N^d) & \text{for } a < b^d \\ \text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\ \text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d \end{array}$$

Which of the following recurrence relations does the master theorem apply to and which does it not?

1. $f(n) = f(n-1) + 7$
2. $f(n) = 3 \cdot f(n-1)$
3. $f(n) = 4 \cdot (f(n/2)$
4. $f(n) = f(n-1) + f(n-2)$

Why does the master theorem apply to some and not others?

# **Answers:** Other Kinds of Recurrence Relations

**Master Theorem**

$$f(n) = af(n/b) + cn^d$$

$$\begin{array}{lll} \text{(Case 1)} & O(N^d) & \text{for } a < b^d \\ \text{(Case 2)} & O(N^d \log N) & \text{for } a = b^d \\ \text{(Case 3)} & O(N^{\log_b a}) & \text{for } a > b^d \end{array}$$

Which of the following recurrence relations does the master theorem apply to and which does it not?

| | | |
|---|---|---|
| 1 | $f(n) = f(n-1) + 7$ | Nope, linear RR, degree 1 |
| 2 | $f(n) = 3 \cdot f(n-1)$ | Nope, linear RR, degree 1 |
| 3 | $f(n) = 4 \cdot (f(n/2)$ | Yep, divide/conquer RR, $O(N)$ |
| 4 | $f(n) = f(n-1) + f(n-2)$ | Nope, linear RR, degree 2 |

- The Master Theorem applies to **Divide and Conquer** algorithms and their associated recurrence relations
- Requires a recurrence involving **division**
- 1,2,4 are **linear recurrence relations** and are worth a few words

# Exercise: Propose a Solution

For the following recurrence relations

- Compute $f(5)$
- Give a closed form solution for the Recurrence Relation
  - One that doesn't involve a recurrence

## Recurrence Relation 1

**Recurrence** $f(n) = f(n-1) + 7$
**Base Case** $f(0) = 0$

- $f(5) = f(4) + 7 = \dots$

## Recurrence Relation 2

**Recurrence** $f(n) = 3 \cdot f(n-1)$
**Base Case** $f(0) = 1$

- $f(5) = 3 \cdot f(4) = \dots$

# **Answers:** Propose a Solution

For the following recurrence relations

- Compute $f(5)$
- Give a closed form solution for the Recurrence Relation

## Recurrence Relation 1

**Recurrence** $f(n) = f(n-1) + 7$
**Base Case** $f(0) = 0$

$$\begin{aligned} f(5) &= 7 + f(4) \\ &= 7 + 7 + f(3) \\ &= 7 + 7 + 7 + f(2) \\ &= 7 + 7 + 7 + 7 + f(1) \\ &= 7 + 7 + 7 + 7 + 7 + f(0) \\ &= 7 + 7 + 7 + 7 + 7 + 0 \\ &= 7 \cdot 5 \\ f(n) &= 7 \cdot n \end{aligned}$$

## Recurrence Relation 2

**Recurrence** $f(n) = 3 \cdot f(n-1)$
**Base Case** $f(0) = 1$

$$\begin{aligned} f(5) &= 3 \cdot f(4) \\ &= 3 \cdot 3 \cdot f(3) \\ &= 3 \cdot 3 \cdot 3 \cdot f(2) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot f(1) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot f(0) \\ &= 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 1 \\ &= 3^5 \\ f(n) &= 3^n \end{aligned}$$

# Linear Recurrence Relations Have General Solutions

- Linear **homogenous** recurrence relations have the form and closed form solution for some constant $r$

$$\begin{aligned} f(n) &= a_1 f(n-1) + a_2 f(n-2) + \ldots + a_k f(n-k) \\ &= \alpha_1 r_1^n + \alpha_2 r_2^n + \ldots + \alpha_k r_k^n \end{aligned}$$

- Solution usually involves determining $r_1, r_2, \ldots, r_k$ as the roots of the the associated **characteristic equation**

$$r^k - a_1 r^{k-1} - a_2 r^{k-1} - \ldots - a_k r^{k-k}$$

- Once \$r$_1$,r$_2$,...\$ known, determine coefficients \$$\alpha_1$,$\alpha_2$,...\$ by solving initial conditions

# Example of Solving a Linear RR

Solve the following Linear RR

| | |
|---|---|
| GIVEN | |
| Recurrence: | $f(n) = f(n-1) + 2f(n-2)$ |
| | So degree 2 with $a_1 = 1, a_2 = 2$ |
| Base Cases: | $f(0) = 2,\ f(1) = 7$ |
| Gen Soln Form: | $f(n) = \alpha_1 r_1^n + \alpha_2 r_2^n$ |
| SOLVE $r_i$'s | |
| Char Eqn.: | $r^2 - a_1 r - a_2 = 0$ with $a_1 = 1, a_2 = 2$ |
| Roots of Char Eqn: | $r^2 - r - 2 = 0$ |
| | $(r+2)(r-1) = 0$ so roots 2,-1 |
| Sub in Gen Soln: | $f(n) = \alpha_1(2)^n + \alpha_2(-1)^n$ |
| SOLVE $\alpha_i$'s | |
| Use Init Conds to | $f(0) = \alpha_1 \cdot 2^0 + \alpha_2 \cdot (-1)^0 = \alpha_1 + \alpha_2 = 2$ |
| solve $\alpha_i$: | $f(1) = \alpha_1 \cdot 2^1 + \alpha_2 \cdot (-1)^1 = 2\alpha_1 - \alpha_2 = 7$ |
| | 2 linear equations with 2 unknowns, $\alpha_1, \alpha_2$ |
| | Solve to get $\alpha_1 = 3, \alpha_2 = -1$ |
| Final Solution: | $f(n) = 3 \cdot 2^n - (-1)^n$ |

# Beyond Linear Homogeneous Recurrence Relations

- Generally recurrence relations that are based on $k$ last terms are exponential
- Using framework it is possible to show that nth Fibonacci number is

$$fib(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

  which is exponential
- Possible to solve more complex **nonhomogenous** recurrence relations which include functions $g(n)$ that are non-recurrent

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \ldots + a_k f(n-k) + \mathbf{g(n)}$$

- Solving is trickier but extend techniques for linear recurrences

# Takehome on Recurrence Relations

- Can solve some kinds of recurrence relations exactly, in particular linear recurrence relations
- Recurrence relations model operation counts in divide/conquer algorithms
- Most interested in Big-O operation estimates for these which are given by the Master Theorem