Lecture 4
Recursion

EECS 281: Data Structures & Algorithms

What Counts as One Step in a Program ?

Primitive operations

- **a)** Variable assignment
- **b)** Arithmetic operation
- **c)** Comparison
- **d)** Array indexing or pointer reference

  In reality: a[i] is the same as *(a + i)
- **e)** Function call (not counting the data)
- **f)** Function return (not counting the data)

Runtime of 1 step is independent of input

Counting Steps

```
int  myFunc(
                char*   p,
                int  aN,
                int  ar[])  {
…
        return
                zzz;
}
```

- 1 step (myFunc as a callee)
- 1 step
- 1 step
- 1 step
- 1 steps
- 1 steps

Passing every datum to/from a function takes time. Passing larger objects and containers *by value* takes longer

When passing objects, count copy-constructors

```
char  *course  =  "EECS281";
int  HWKs[4]  =  {100,110,120,140};
int  retCode  =
            myFunc(
                    course,  4,  HWKs);
```

- 8 steps (copy chars at run time)
- 4 steps (copy at run time)
- 1 step
- 2 steps (call and return)
- 4 steps (one for each parameter and return value)

The Program Stack (1)

- When a function call is made

  1a. All local variables are saved in a special storage called *the program stack*

  2a. Then argument values are pushed onto *the program stack*

- When a function call is received

  2b. Function arguments are popped off the stack

- When return is issued within a function

  3a. The return value is pushed onto *the program stack*

- When return is received at the call site

  3b. The return value is popped off the *the program stack*
  1b. Saved local variables are restored

The Program Stack (2)

- Program stack supports nested function calls
  - Five nested calls would save five sets of local variables and five sets of arguments on the P.S.
- There is only one program stack (per thread)
  - Different from *the program heap*, where dynamic memory is allocated
- *Program stack* size is limited in practice
- The number of nested function calls is limited
- Example: a bottomless (buggy) recursion function will exhaust program stack very quickly

Important Practical Considerations

- Program stack is very limited in size
- For a large data set
  - "Plain" recursion over every element is a bad idea
  - Use tail recursion or iterative algorithms instead
- Problems solvable with $O(1)$ additional memory do not favor "plain" recursive algorithms

Step-Counting For Recursion

```
int factorial(int n) {
    return (n ? n * factorial(n- 1) : 1);
}
```

```
int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

2 steps
1 step
2 step
6 steps

Total steps: #calls * steps in 1 function call

Challenge: what if the number of steps per call depends on the argument values?

Recurrence Equations

- A *recurrence equation* describes the overall running time on a problem of size *n* in terms of the running time on smaller inputs. [CLRS]

Recurrence Equation Example

```
int factorial (int n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}
```

$$T(n) = \begin{cases} c_0 & n == 0 \\ T(n-1) + c_1 & n > 0 \end{cases}$$

- $T(n)$ is the running time of `factorial()` with input size n
- $T(n)$ is expressed in terms of the running time of `factorial()` with input size n - 1
- c_0 and c_1 are constants

Solving Recurrences

- Recursion tree method [CLRS]
- AKA Telescoping method

1. Write out $T(n)$, $T(n - 1)$, $T(n - 2)$
2. Substitute $T(n - 1)$ and $T(n - 2)$ into $T(n)$
3. Look for a pattern
4. Use a summation formula

Exercise

```
int power(int x, unsigned y);
// returns x^y
```

Write two versions:

1. With recursion (or tail recursion) and $O(n)$ complexity
   - Write the recurrence equation
2. With a loop and $O(\log n)$ complexity
   - Hint: $2^8 = ((2^2)^2)^2$

Does it work for 2^0? 0^2? 0^0?

Another solution

Write the recurrence equation:

```
int power(int x, unsigned y, int result = 1) {
  if (y == 0)
    return result;
  else if (y % 2)
    return power(x * x, y / 2, result * x);
  else
    return power(x * x, y / 2, result);
}
```

Common Recurrence Equations

Recurrence	Example	Big-O Solution
$T(n) = T(n / 2) + c$	Binary Search	$O(\log n)$
$T(n) = T(n - 1) + c$	Sequential Search	$O(n)$
$T(n) = 2T(n / 2) + c$	Tree Traversal	$O(n)$
$T(n) = T(n - 1) + c_1 * n + c_2$	Selection/etc. Sorts	$O(n^2)$
$T(n) = 2T(n / 2) + c_1 * n + c_2$	Merge/Quick Sorts	$O(n \log n)$

Solving Recurrences

- We have used the recursion tree (AKA telescoping) method to solve recurrence equations
- Another way to solve recurrence equations is the Master Method (AKA Master Theorem)

Master Theorem Basis

Let $T(n)$ be a monotonically increasing function that satisfies:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(1) = 1$$

Where $a \geq 1$, $b \geq 2$. If $f(n) \in \Theta(n^c)$, then:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c \\ \Theta(n^c \log_2 n) & \text{if } a = b^c \\ \Theta(n^c) & \text{if } a < b^c \end{cases}$$

When Not to Use

- You cannot use the Master Theorem if:
  - $T(n)$ is not monotonic, such as $T(n) = \sin(n)$
  - $f(n)$ is not a polynomial, i.e. $f(n)=2^n$
  - b cannot be expressed as a constant, i.e.

$$T(n) = \sqrt{n}$$

- There is also a special fourth condition if $f(n)$ is not a polynomial; see later in slides

When Not to Use

Example:

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

Master Theorem not applicable

$$T(n) \neq aT(n / b) + f(n)$$

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

a =

b =

c =

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

$a = 3$

$b =$

$c =$

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

$a = 3$

$b = 2$

$c =$

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

$a = 3$

$b = 2$

$c = 1$

Therefore which condition?

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

$a = 3$

$b = 2$

$c = 1$

Therefore which condition? Since $3 > 2^1$, case 1

Example

$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$ What are the parameters?

$a = 3$

$b = 2$

$c = 1$

Therefore which condition? Since $3 > 2^1$, case 1

Thus we conclude that:

$$T(n) \in \Theta\left(n^{\log_b a}\right) = \Theta(n^{\log_2 3})$$

Exercise

$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} + 7$ What are the parameters?

$a =$

$b =$

$c =$

Solve the recurrence equation

Exercise

$T(n) = T\left(\frac{n}{2}\right) + \frac{1}{2}n^2 + n$ What are the parameters?

a =

b =

c =

Solve the recurrence equation

Fourth Condition

- There is a 4th condition that allows polylogarithmic functions

$$\text{If } f(n) \in \Theta(n^{\log_b a} \log^k n) \text{ for some } k \geq 0,$$

$$\text{Then } T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

- This condition is fairly limited, and not one you need to memorize/write down

Fourth Condition Example

- Say that we have the following recurrence:

$$T(n) = 2T\left(\frac{n}{2}\right) + n\log n$$

- Clearly a=2, b=2, but $f(n)$ is not polynomial
- However: $f(n) \in \Theta(n\log n)$ and k = 1

$$T(n) = \Theta(n\log^2 n)$$

Job Interview Question

Write an efficient algorithm that searches for a value in an n x m table (two-dim array). This table is sorted along the rows and columns — that is,

$$\text{table}[i][j] \le \text{table}[i][j+1],$$
$$\text{table}[i][j] \le \text{table}[i+1][j]$$

- Obvious ideas: linear or binary search in every row
  - nm or $n \log m$ … too slow

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Solution #1: Quad Partition

Split the region into four quadrants – one can be eliminated.
Then recurse

$$T(n) = 3T(n/2) + c$$

By the Master Theorem (or telescoping),

$$T(n) = \Theta(n^{\log 2(3)}) \approx \Theta(n^{1.58})$$

Not competitive enough!

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

Solution #2: Binary Partition

Split the region into four quadrants:

- scan a middle row/column/diagonal for the target element
- if not found, split where it would have been
- eliminate 2 of 4 sub-regions

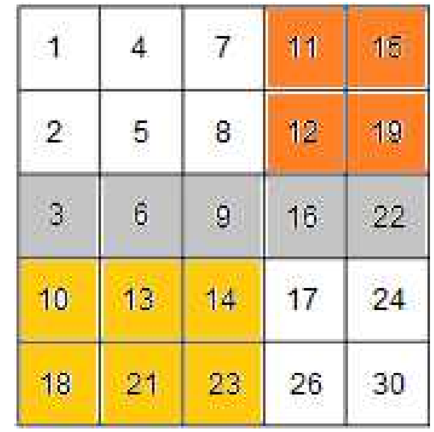
Then recurse:

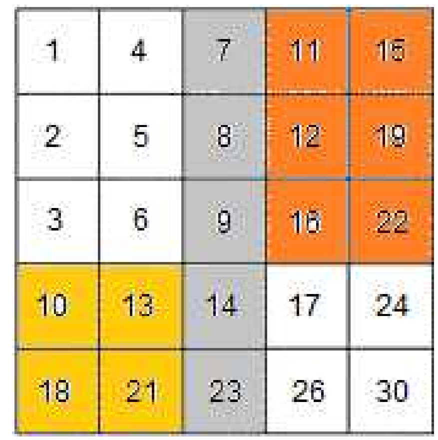
$T(n) = 2T(n/2) + cn$ <u>or</u> $T(n) = 2T(n/2) + \log n$

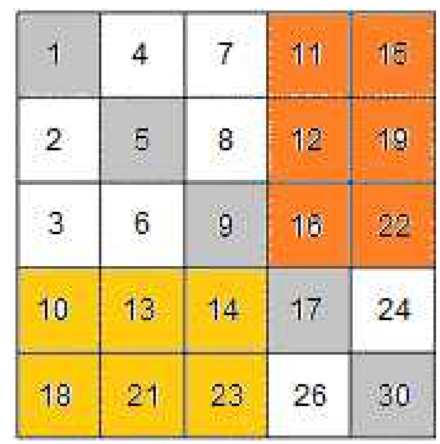
By the Master Theorem (or by telescoping),

$T(n)=\Theta(n \log n)$ <u>or</u> $T(n)=\Theta(n)$

Not entirely rigorous because sub-arrays may differ in size. What happens to complexity then?

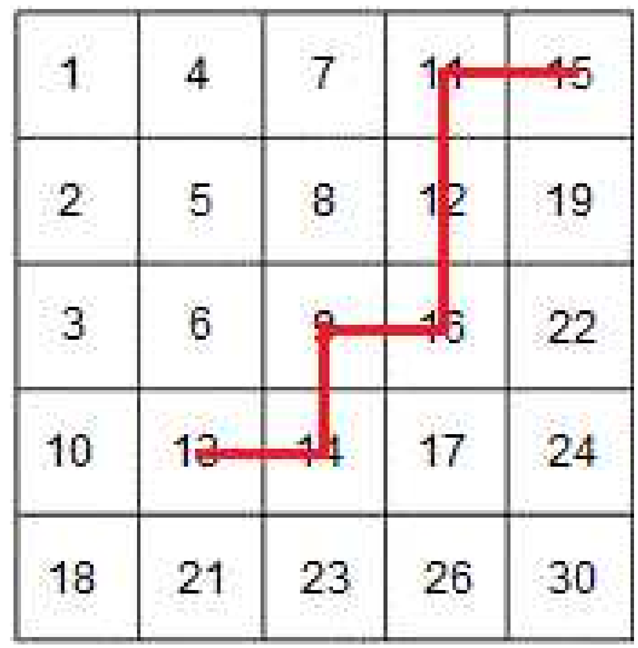






Solution #3:
Stepwise Linear Search

```
bool stepWise(int mat[][N_MAX], int N, int target, int &row, int &col) {
  if (target< mat[0][0] || target > mat[N - 1][N - 1])
    return false;
  row = 0; col = N - 1;
  while (row <= N - 1 && col >= 0)
  {
    if (mat[row][col] < target)
      row++;
    else if (mat[row][col] > target)
      col--;
    else
      return true;
  }
  return false;
}
```



Runtime Comparisons

- Source code and data (M = N = 100) available at

http://www.leetcode.com/2010/10/searching-2d-sorted-matrix.html

http://www.leetcode.com/2010/10/searching-2d-sorted-matrix-part-ii.html

http://www.leetcode.com/2010/10/searching-2d-sorted-matrix-part-iii.html

- Runtime for 1,000,000 searches

Algorithm	Runtime
Binary search	**31.62s**
Diagonal Binary Search	**32.46s**
Step-wise Linear Search	**10.71s**
Quad Partition	**17.33s**
Binary Partition	**10.93s**
Improved Binary Partition	**6.56s**

Linear Recurrences

- Fibonacci sequence: 0 1 1 2 3 5 8 13 …
  - $F_0 = 0$, $F_1 = 1$
  - $F = F_{n-1} + F_{n-2}$, where ≥ 2
- Appears frequently in many contexts
  - Illustrates several types of algorithms
  - Stock-trading strategies
  - Nice-looking architectural proportions
- Often used in interview questions

Linear Recurrences

- Fibonacci sequence: 0 1 1 2 3 5 8 13 …
  - $F_0 = 0$, $F_1 = 1$
  - $F = F_{n-1} + F_{n-2}$, where ≥ 2
- Closed-form solution:

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

- Can be computed in $O(\log n)$ time

Questions for Self-Study

- Consider a recursive function that only calls itself. Explain how one can replace recursion by a loop and an additional stack.
- Go over the Master Theorem in the CLRS textbook
- Which cases of the Master Theorem were exercised for different solutions in the 2D-sorted-matrix problem ?
- Solve the same recurrences by telescoping w/o the Master Theorem
- Write (and test) programs for each solution idea, time them on your data