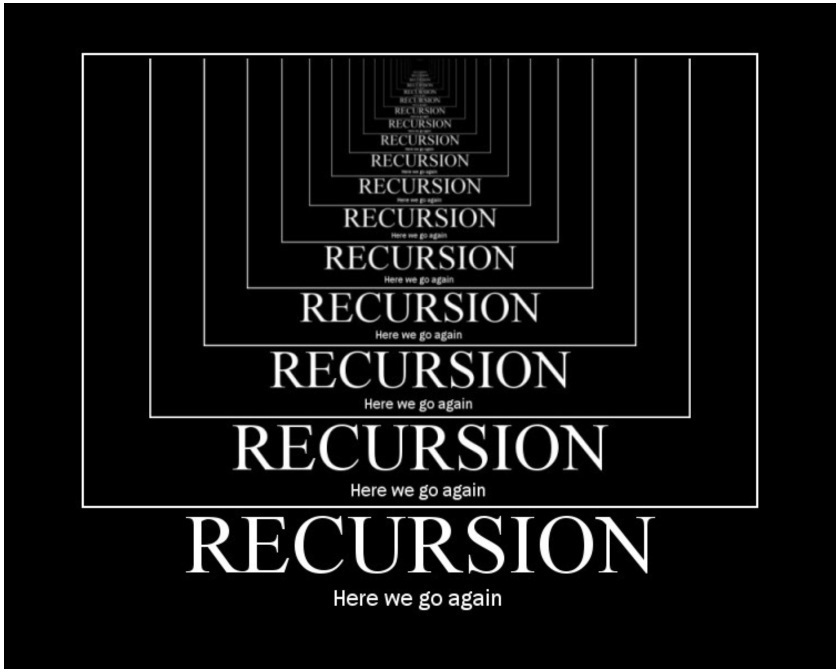
Lecture 5
Recursion



EECS 281: Data Structures & Algorithms

Introduction to Recursion

Data Structures & Algorithms

Recursion Basics

- A recursive function calls itself to accomplish its task
- At least one base case is required
- At least one recursive case is required
- Each subsequent call has simpler (smaller) input
- Single recursion can be used on lists
- Multiple recursion can be used on trees

Job Interview Question

- Implement this function

```
// calculate x^n
int power(int x, uint32_t n);
```

- The obvious solution uses n - 1 multiplications
  - $2^8 = 2*2* \ldots *2$
- Less obvious: ***O*(log *n*)** multiplications
  - Hint: $2^8 = ((2^2)^2)^2$
  - How does it work for 2^7 ?
- Write both solutions iteratively and recursively

Ideas

- Obvious approach uses a subproblem "one step smaller"

$$x^n = \begin{cases} 1 & n == 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

- Less obvious approach splits the problem into two halves

$$x^n = \begin{cases} 1 & n == 0 \\ x^{n/2} * x^{n/2} & n > 0, \text{even} \\ x * x^{\lfloor n/2 \rfloor} * x^{\lfloor n/2 \rfloor} & n > 0, \text{odd} \end{cases}$$

Two Recursive Solutions

Solution #1

```
int power(int x, uint32_t n) {
  if (n == 0)
    return 1;

  return x * power(x, n - 1);
}  // power()
```

Recurrence: $T(n) = T(n-1) + c$
Complexity: $\boldsymbol{\Theta(n)}$

Solution #2

```
int power(int x, uint32_t n) {
  if (n == 0)
    return 1;

  int result = power(x, n / 2);
  result *= result;
  if (n % 2 != 0)  // n is odd
    result *= x;

  return result;
}  // power()
```

Recurrence: $T(n) = T(n / 2) + c$
Complexity: $\boldsymbol{\Theta(\log n)}$

Introduction to Recursion

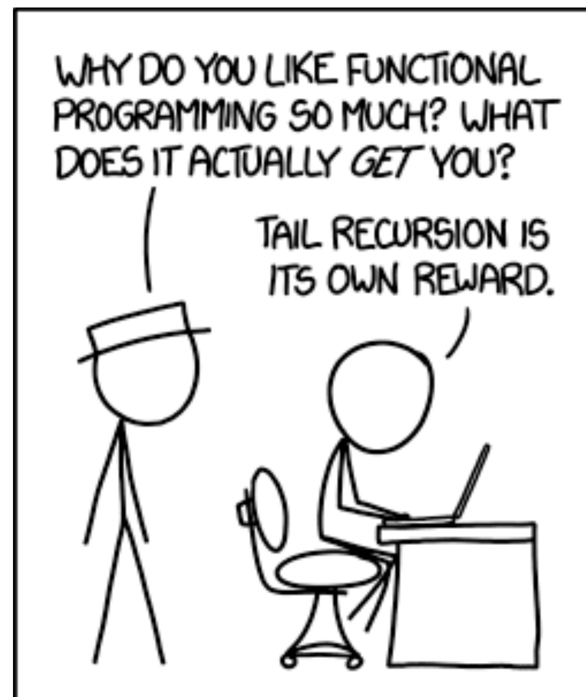
Data Structures & Algorithms

Tail Recursion

Data Structures & Algorithms

Tail Recursion

- When a function is called, it gets a *stack frame*, which stores the local variables
- A simply recursive function generates a stack frame for each recursive call
- A function is *tail recursive* if there is no pending computation at each recursive step
  - "Reuse" the stack frame rather than create a new one
- Tail recursion and iteration are equivalent



http://xkcd.com/1270

Recursion and the Stack

```
uint32_t factorial(uint32_t n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}  // factorial()

int main() {
  factorial(5);
  return 0;
}  // main()
```

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
  - Six nested calls = six sets of local variables
- There is only one program stack (per thread)
  - NOT *the program heap*, (where dynamic memory is allocated)
- *Program stack* size is limited
  - The number of nested function calls is limited
- Example: a bottomless (buggy) recursion function will exhaust program stack very quickly

Recursion vs. Tail Recursion

```
uint32_t factorial(uint32_t n) {
  if (n == 0)
    return 1;
  return n * factorial(n - 1);
}  // factorial()
```

Recursive

Θ(n) time complexity
Θ(n) space complexity
(uses *n* stack frames)

```
uint32_t factorial(uint32_t n, uint32_t res = 1) {
  if (n == 0)
    return res;
  return factorial(n - 1, res * n);
}  // factorial()
```

Tail recursive*

Θ(n) time complexity
Θ(1) space complexity
(reuses 1 stack frame)

*The default argument is used to seed the `res` parameter. Alternatively, the "helper function" pattern could be used.

Logarithmic Tail Recursive power()

```
int power(int x, uint32_t n, int result = 1) {
  if (n == 0)
    return result;
  else if (n % 2 == 0)  // even
    return power(x * x, n / 2, result);
  else  // odd
    return power(x * x, n / 2, result * x);
}  // power()
```

***Θ*(log *n*)** time complexity
***Θ*(1)** space complexity

Practical Considerations

- Program stack is limited in size
  - It's actually pretty easy to exhaust this!
    e.g. Computing the length of a very long vector using a "linear recursive" function with $\boldsymbol{\Theta(n)}$ space complexity

- For a large data set
  - ”Simple" recursion is a bad idea
  - Use tail recursion or iterative algorithms instead

- This doesn't mean everything should be tail recursive
  - Some problems can't be solved in $\boldsymbol{\Theta(1)}$ space!

Tail Recursion

Data Structures & Algorithms

Recurrence Relations

Data Structures & Algorithms

Recurrence Relations

- A *recurrence relation* describes the way a problem depends on a subproblem.
  - A recurrence can be written for a computation:

$$x^n = \begin{cases} 1 & n == 0 \\ x * x^{n-1} & n > 0 \end{cases}$$

  - A recurrence can be written for the time taken:

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

  - A recurrence can be written for the amount of memory used*:

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

*Non-tail recursive

A Logarithmic Recurrence Relation

$$T(n) = \begin{cases} c_0 & n == 0 \\ T\left(\frac{n}{2}\right) + c_1 & n > 0 \end{cases} \quad \rightarrow \quad \boldsymbol{\Theta(\log n)}$$

- Fits the logarithmic recursive implementation of `power()`
  - The power to be calculated is divided into two halves and combined with a single multiplication
- Also fits Binary Search
  - The search space is cut in half each time, and the function recurses into only one half

Common Recurrences

Recurrence	Example	Big-O Solution
$T(n) = T(n / 2) + c$	Binary Search	$O(\log n)$
$T(n) = T(n - 1) + c$	Linear 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

- Substitution method
  1. Write out $T(n)$, $T(n - 1)$, $T(n - 2)$
  2. Substitute $T(n - 1)$, $T(n - 2)$ into $T(n)$
  3. Look for a pattern
  4. Use a summation formula
- Another way to solve recurrence equations is the Master Method (AKA Master Theorem)

Recurrence Thought Exercises

- What if a recurrence cuts a problem into two subproblems, and both subproblems were recursively processed?
- What if a recurrence cuts a problem into three subproblems and…
  - Processes one piece
  - Processes two pieces
  - Processes three pieces
- What if there was additional, non-constant work after the recursion?

Recurrence Relations

Data Structures & Algorithms

The Master Theorem

Data Structures & Algorithms

Master Theorem

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

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

$$T(1) = c_0 \;\; or \;\; T(0) = c_0$$

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

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log n) & if\ a = b^c \\ \Theta(n^c) & if\ a < b^c \end{cases}$$

Exercise 1

$$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$$

What are the parameters?

a = 3

b = 2

c = 1

Which condition?

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log n) & if\ a = b^c \\ \Theta(n^c) & if\ a < b^c \end{cases}$$

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

$$f(n) \in \Theta(n^c)$$

Exercise 1

$$T(n) = 3T\left(\frac{n}{2}\right) + \frac{3}{4}n + 1$$

What are the parameters?

a = 3

b = 2

c = 1

Since 3 > 2^1, we conclude:

$$T(n) \in \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 3}\right) \approx \boldsymbol{\Theta}\left(\boldsymbol{n}^{\mathbf{1.58486\ldots}}\right)$$

Exercise 2

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} + 7$$

What are the parameters?

a = 2

b = 4

c = 1/2

Which condition?

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log n) & if\ a = b^c \\ \Theta(n^c) & if\ a < b^c \end{cases}$$

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

$$f(n) \in \Theta(n^c)$$

Exercise 2

$$T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n} + 7$$

What are the parameters?

a = 2

b = 4

c = 1/2

Since $2 = 4^{1/2}$, we conclude:

$$T(n) \in \Theta(n^c \log n) = \boldsymbol{\Theta}\left(\sqrt{\boldsymbol{n}}\,\mathbf{log}\,\boldsymbol{n}\right)$$

Exercise 3

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

What are the parameters?

a = 1

b = 2

c = 2

Which condition?

$$T(n) \in \begin{cases} \Theta\left(n^{\log_b a}\right) & if\ a > b^c \\ \Theta(n^c \log n) & if\ a = b^c \\ \Theta(n^c) & if\ a < b^c \end{cases}$$

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

$$f(n) \in \Theta(n^c)$$

Exercise 3

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

What are the parameters?

a = 1

b = 2

c = 2

Since $1 < 2^2$, we conclude:

$$T(n) \in \Theta(n^c) = \boldsymbol{\Theta\left(n^2\right)}$$

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.

$$T(n) = T\left(\sqrt{\sin n}\right)$$

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

When Not to Use

When the recursion does not use division:

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

the Master Theorem is not applicable.

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

Fourth Condition

- There is a 4th condition that allows polylogarithmic functions

$$\text{If } f(n) \in \Theta\left(n^{\log_b a} \log^k n\right) \text{ for some } k \geq 0,$$
$$\text{Then } T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$$

- This condition is fairly limited,
- No need to memorize/write down

Fourth Condition Example

- Given 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 $k = 1$

$$T(n) = \boldsymbol{\Theta}\left(\boldsymbol{n} \log^{2} \boldsymbol{n}\right)$$

If $f(n) \in \Theta\left(n^{\log_b a} \log^{k} n\right)$ for some $k \geq 0$,
Then $T(n) \in \Theta(n^{\log_b a} \log^{k+1} n)$

Bonus: Binary Max?

- A friend of yours claims to have discovered a (revolutionary!) new algorithm for finding the maximum element in an unsorted array:
  1. Split the array into two halves.
  2. Recursively find the maximum in each half.
  3. Whichever half-max is bigger is the overall max!

- Your friend says this algorithm leverages the power of "binary partitioning" to achieve better than linear time.
  – This sounds too good to be true. Give an intuitive argument why.
  – Use the master theorem to formally prove this algorithm is $\boldsymbol{\Theta(n)}$.

The Master Theorem

Data Structures & Algorithms

2D Table Search

Data Structures & Algorithms

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,

```
table[i][j] ≤ table[i][j + 1],
table[i][j] ≤ table[i + 1][j]
```

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

- Obvious ideas: linear or binary search in every row
  - ***O*(*nm*)** or ***O*(*n* log *m*)** … too slow

Potential Solution #1: Quad Partition

- Split the region into four quadrants, eliminate one.
- Then, recursively process the other 3 quadrants.
- Write a recurrence relation for the time complexity in terms of *n*, the length of one side:

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

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

n

Why *n/2* and not *n/4* if it's a quadrant?
Remember, *n* is the length of one side!

Potential Solution #2: Binary Partition

How can we improve this?

- Split the region into four quadrants.
- Scan down the middle column, until you find "where the value should be" if it were in that column.[1]
- This allows you to eliminate two quadrants. (Why? Which ones?)
- Recursively process the other two.
- Write a recurrence relation, again in terms of the side length *n*:

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

Use binary search!

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

n

1 Of course, you might get lucky and find the value here!

Potential Solution #3: Improved Binary Partition

- Split the region into four quadrants.
- Scan down the middle column, until you find "where the value should be" if it were in that column.[1]
- This allows you to eliminate two quadrants. (Why? Which ones?)
- Recursively process the other two.
- Write a recurrence relation, again in terms of the side length *n*:

$$T(n) = 2T(n/2) + c \log n$$

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

n

1 Of course, you might get lucky and find the value here!

Exercise: Use the Master Theorem

- Use the master theorem to find the complexity of each approach:
- Quad Partition:

$$T(n) = 3T(n/2) + c$$
$$T(n) = \Theta(n^{\log_2 3}) \approx \boldsymbol{\Theta(n^{1.58})}$$

- Binary Partition:

$$T(n) = 2T(n/2) + cn$$
$$T(n) = \boldsymbol{\Theta(n \log n)}$$

- Improved Binary Partition:

$$T(n) = 2T(n/2) + c \log n$$
$$T(n) = \boldsymbol{\Theta(n)}$$

Another Solution!
Stepwise Linear Search

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

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

m

n

Runtime Comparisons

- Further discussion and examples available at
  - https://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/
  - https://www.geeksforgeeks.org/search-in-a-row-wise-and-column-wise-sorted-2d-array-using-divide-and-conquer-algorithm/
- Runtime for 1,000,000 searches (M = N = 100)

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

2D Table Search

Data Structures & Algorithms

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.
- Which cases of the Master Theorem were exercised for different solutions in the 2D-sorted-matrix problem?
- Solve the same recurrences by substitution w/o the Master Theorem
- Write (and test) programs for each solution idea, time them on your data