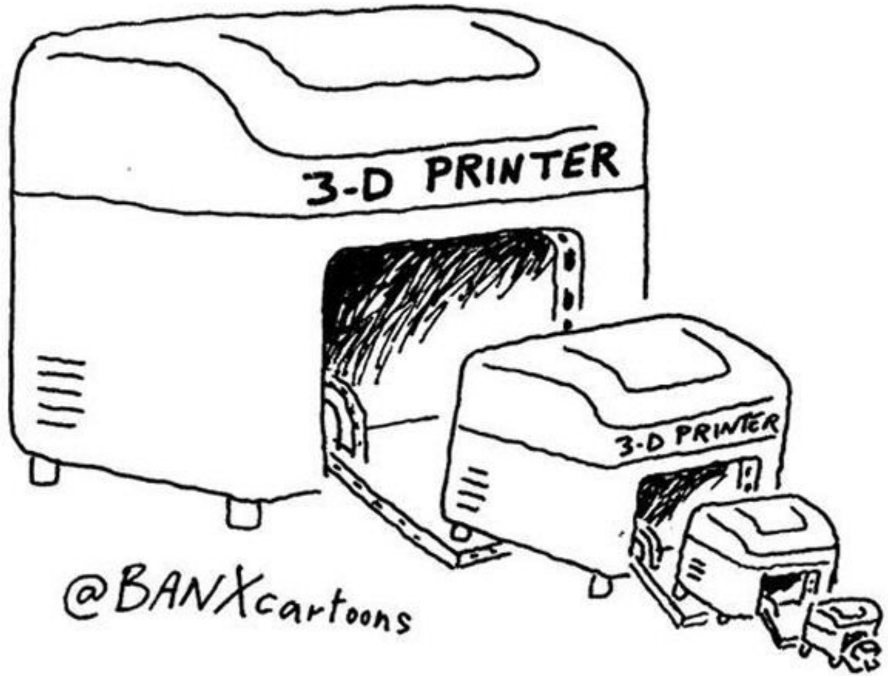
Lecture 33-34: Recursion

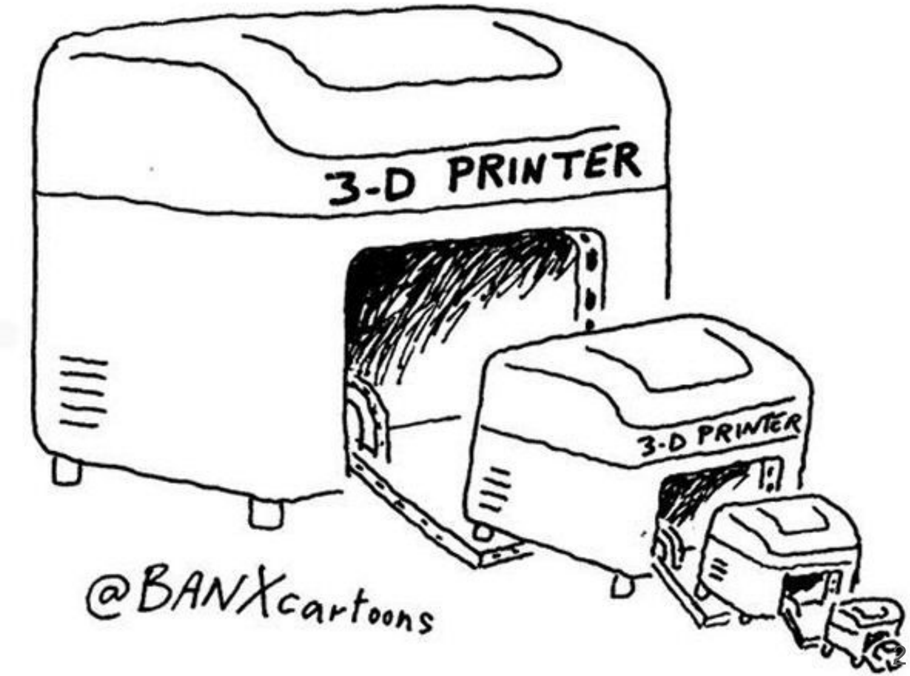


Announcements and reminders

- Project 3 posted
  - By **Wednesday** -- TA/CA design meeting
  - … & submit classes and code skeleton

- Project 2 interview grading

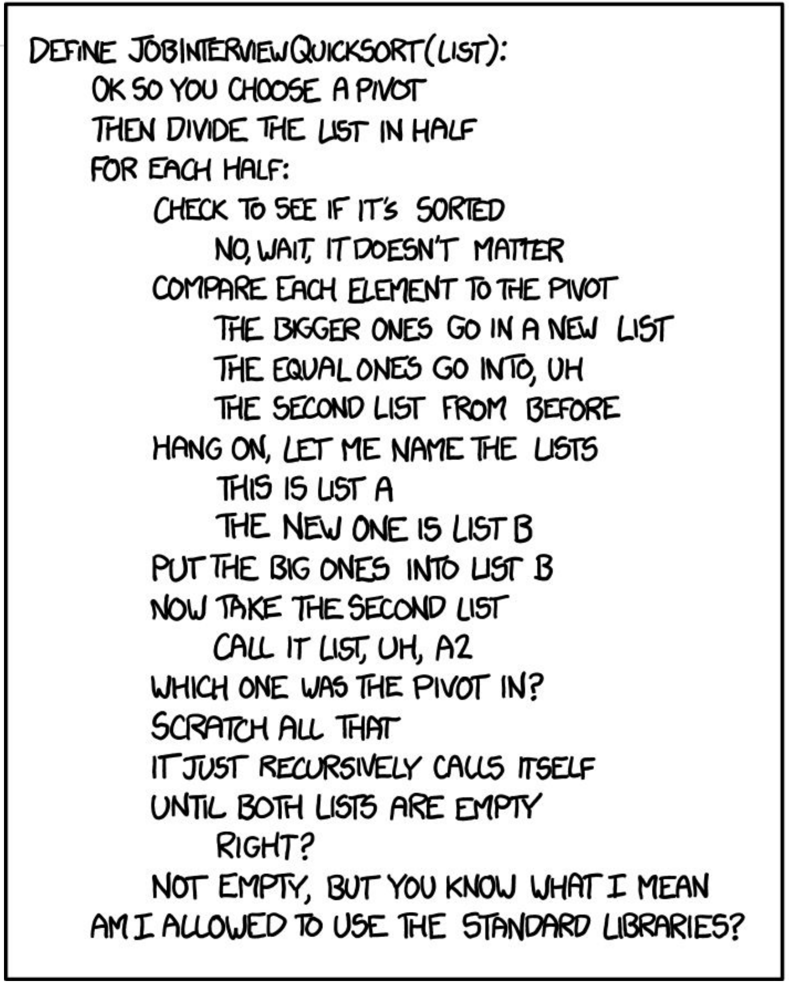
- Homework 9
  Due Wednesday April 17 by 11 PM



Last time on *Intro Computing…*

We **sorted!**

- Bubble sort
- Cocktail sort
- Merge sort
- Quick sort



Quick sort

https://www.geeksforgeeks.org/quick-sort/

Like **merge sort**, this is a **divide and conquer algorithm**

- Pick one of the elements of the list to sort as the **pivot**
- ***Divides*** the original list into parts <= pivot, and > pivot…
- … and calls itself on those smaller parts to sort them (***conquers***)

Example: S'pose we want to sort the list {27, 10, 43, 3, 9, 82, 38}.

- Let's arbitrarily pick the **last element** as the pivot, always
  (different versions do different things)

Quick sort

27	10	43	3	9	82	**38**

Quick sort

27	10	43	3	9	82	**38**

Recursion

Recursion is a computational technique to break complex problems up into **smaller versions** of the **larger problem**, and solving the smaller, simpler problems.

Recursion is often the most natural way to think about a problem,
and there are some calculations that are difficult to perform without recursion.

Definition: A recursive function is a function that calls itself, reducing the “size” of the input with each call.

Example: the factorial function, $f(n) = n*(n-1)*\ldots*3*2*1$

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

How can start to think recursively?

The big question: How can we break the problem down into a smaller version of itself?

Example: Walk from Point A to Point Z.

How can start to think recursively?

The big question: How can we break the problem down into a smaller version of itself?

Example: Walk from Point A to Point Z.

Solution: S’pose our function to give us a route from A to Z is `route(A, Z)`.

- Answer: Walk A → B, then `route(B, Z)`. Done!

How can start to think recursively?

The big question: How can we break the problem down into a smaller version of itself?

Example: Walk from Point A to Point Z.

Solution: S'pose our function to give us a route from A to Z is `route(A, Z)`.

- Answer: Walk A → B, then `route(B, Z)`. Done!
- Behind the curtains: `route(B, Z)` = walk B → C, then `route(C, Z)`

  … and `route(C, Z)` = walk C → D, then `route(D, Z)`

  … and `route(D, Z)` = walk D → E, then `route(E, Z)`

Triangle of boxes -- recursive

Example: Write code to print to the screen a triangle of boxes with an input parameter side length.

```
[]
[][]
[][][]
[][][][]

void printTriangle(int side_length);
```

Triangle of boxes -- recursive

Example: Write code to print to the screen a triangle of boxes with an input parameter side length.

```
[]
[][]
[][][]
[][][][]
```

```
void printTriangle(int side_length);
```

Strategy: Think: what is a **smaller version** of this problem?

Triangle of boxes -- recursive

Example: Write code to print to the screen a triangle of boxes with an input parameter side length.

```
void printTriangle(int side_length) {
    if (side_length < 1) {
        return;
    }
    printTriangle(side_length - 1);
    for (int i=0; i < side_length; i++) {
        cout << “[]”;
    }
    cout << endl;
}
```

Thinking recursively

There are two key requirements for successful recursive functions:

1) Every recursive function call must simplify the task in some way
2) There must be special case(s) to handle the simplest forms, so the function will eventually stop calling itself.

```
void printTriangle(int side_length) {
    if (side_length < 1) {
        return;
    }
    printTriangle(side_length - 1);
    for (int i=0; i < side_length; i++) {
        cout << "[]";
    }
    cout << endl;
}
```

Common error -- infinite recursion

Infinite recursion occurs when we either

1) forget to write the end test, or
2) the test to end the recursion never becomes true

Example: Maybe we screwed up our `printTriangle` function as follows:

```
void printTriangle(int side_length) {
    printTriangle(side_length - 1);
    for (int i=0; i < side_length; i++) {
        cout << “[]”;
    }
    cout << endl;
}
```

Consider: What would happen? Say, we call `printTriangle(3)`.

Common error -- infinite recursion

Example: Maybe we screwed up our `printTriangle` function as follows:

```
void printTriangle(int side_length) {
    printTriangle(side_length - 1);
    for (int i=0; i < side_length; i++) {
        cout << "[]";
    }
    cout << endl;
}
```

Consider: What would happen? Say, we call `printTriangle(3)`.

Common error -- infinite recursion

Example: Maybe we screwed up our `printTriangle` function as follows:

```
void printTriangle(int side_length) {
    printTriangle(side_length - 1);
    for (int i=0; i < side_length; i++) {
        cout << "[]";
    }
    cout << endl;
}
```

Consider: What would happen? Say, we call `printTriangle(3)`.

- Inside the function for the **first** time, we call `printTriangle(2)`
- … the **second** time, we call `printTriangle(1)`
- … the **third** time, we call `printTriangle(0)`
- … the **fourth** time, we call `printTriangle(-1)`
- … and so on… until our laptop battery dies.

Thinking recursively -- palindromes

Definition: a **palindrome** is a string that is equal to itself if you reverse the order of all its characters

Examples:

- kayak
- racecar
- 1101011

Note: Sometimes, all capitalization, punctuation and spaces are removed, so we can have more fun:

- No ‘x’ in Nixon

Thinking recursively -- palindromes

Example: Write a function to test if a string is a palindrome.

For example, `is_palindrome("rotor") = true` and
`is_palindrome("elvis") = false`

Step 1: Break the problem into smaller parts that can themselves be inputs to the problem

→ How can we simplify the input?

Thinking recursively -- palindromes

Example: Write a function to test if a string is a palindrome.

For example, `is_palindrome(“rotor”) = true` and
`is_palindrome(“elvis”) = false`

Step 1: Break the problem into smaller parts that can themselves be inputs to the problem

→ How can we simplify the input?

→ remove the first character?

→ remove the last character?

→ remove *both* the first and last characters?

→ remove a character from the middle?

→ cut the string into two halves?

Thinking recursively -- palindromes

Step 1: Break the problem into smaller parts that can themselves be inputs to the problem

Every palindrome's first and second halves are the same, so cutting the string into two halves ***seems*** like a good idea, but…

… how do we chop up "rotor" ?

Thinking recursively -- palindromes

Step 1: Break the problem into smaller parts that can themselves be inputs to the problem

Every palindrome's first and second halves are the same, so cutting the string into two halves ***seems*** like a good idea, but…

… how do we chop up "rotor" ?

And since the palindrome needs to be a ***mirror*** of itself, removing a **single** character at a time also isn't a good idea.

Instead, what about comparing **two characters** at a time?

Thinking recursively -- palindromes

Step 1: Break the problem into smaller parts that can themselves be inputs to the problem

Instead, what about comparing **two characters** at a time?

`“rotor”`

(chop) (chop)

`“r”` `“oto”` `“r”`

Now our problem is reduced to the **middle** of the original string (“oto”), and comparing the two characters at the ends:

Thinking recursively -- palindromes

Step 2: Combine solutions with simpler inputs to create a solution to the original problem

→ **reduction step** (*reduce* to a smaller problem)

`"rotor"`

(chop) (chop)

`"r"` `"oto"` `"r"`

Now our problem is reduced to the **middle** of the original string ("oto"), and comparing the two characters at the ends:

If (*the end letters are the same* AND

is_palindrome(*the middle string*)) then

the original string is a palindrome!

Thinking recursively -- palindromes

Step 3: Find solutions to the simplest inputs

A recursive computation keeps simplifying its inputs.

Eventually, it arrives at the simplest reasonable inputs. To make sure that the recursion comes to a stop, deal with the simplest inputs separately.

Question: What are the simplest possible palindrome situations?

Thinking recursively -- palindromes

Step 3: Find solutions to the simplest inputs

A recursive computation keeps simplifying its inputs.

Eventually, it arrives at the simplest reasonable inputs. To make sure that the recursion comes to a stop, deal with the simplest inputs separately.

Question: What are the simplest possible palindrome situations?

→ strings of length 0 or 1 are **always** palindromes!

→ we have this stopping condition: `if (str.length() <= 1) { return true; }`

Thinking recursively -- palindromes

Step 4: Implement the solution by combining the simplest cases and the reduction step

First, pseudocode from our previous slides:

```
bool is_palindrome(string str) {
    // simplest cases
    if (str.length() <= 1) { return true; }
    // reduction step
    If ( the end letters are the same   AND
        is_palindrome( the middle string ) ) then
    the original string is a palindrome!
}
```

Now we can tidy it up...

Thinking recursively -- palindromes

```
bool is_palindrome(string str) {

    // simplest cases
    if (str.length() <= 1) {
        return true;
    }

    // reduction step
    char first = tolower(str[0]);
    char last  = tolower(str[str.length()-1]);
    if (first==last) {
        string shorter = str.substr(1, str.length()-2);
        return is_palindrome(shorter);
    } else {
        return false;
    }
}
```

Efficiency -- recursion versus iteration

Example: The **Fibonacci sequence** is a sequence of integers defined by:

initial conditions: $f_0 = 0$ and $f_1 = 1$, and

recursion equation: $f_n = f_{n-1} + f_{n-2}$

So the first 10 terms in the sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

The task: Write both a **recursive** and an **iterative** (i.e., with a **for** loop) functions to get the n^{th} Fibonacci number.

Then, compare how long each takes to compute f_{43}

Efficiency -- recursion versus iteration

Example: The **Fibonacci sequence** is a sequence of integers defined by:

initial conditions: $f_0 = 0$ and $f_1 = 1$, and

recursion equation: $f_n = f_{n-1} + f_{n-2}$

So the first 10 terms in the sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

The task: Write both a **recursive** and an **iterative** (i.e., with a **for** loop) functions to get the n^{th} Fibonacci number.

Then, compare how long each takes to compute f_{43}

Code in the Moodle materials, and at end of these slides

Efficiency -- recursion versus iteration

Okay! So, we should have just seen that the recursive solution takes **A LOT** longer to compute f_{43} than the iterative solution.

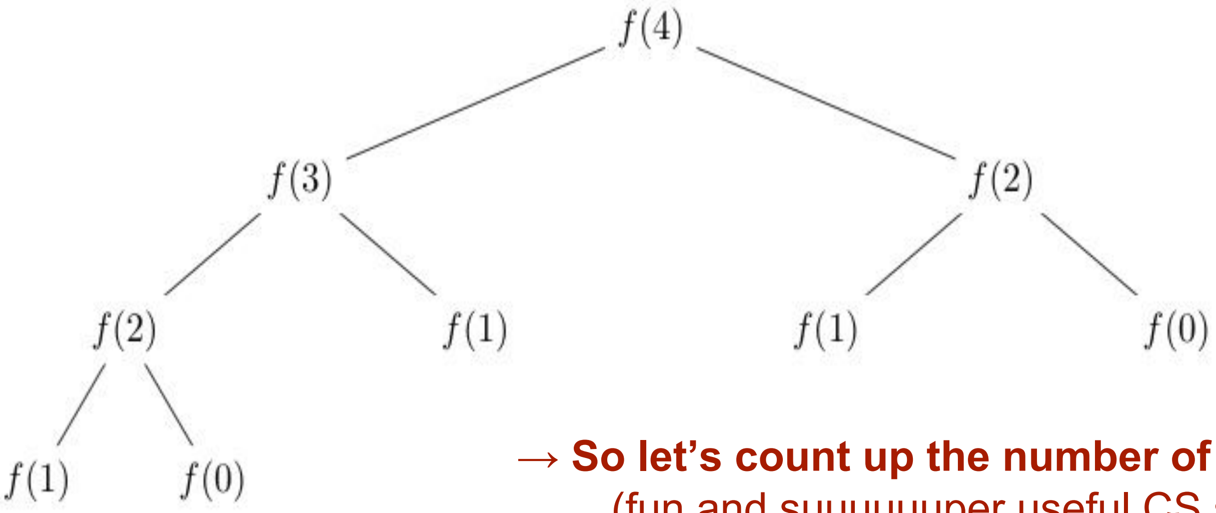
… why??

Efficiency -- recursion versus iteration

Okay! So, we should have just seen that the recursive solution takes **A LOT** longer to compute f_{43} than the iterative solution.

… why??

Turns out, **recursion** ends up computing a lot of the values **multiple times**



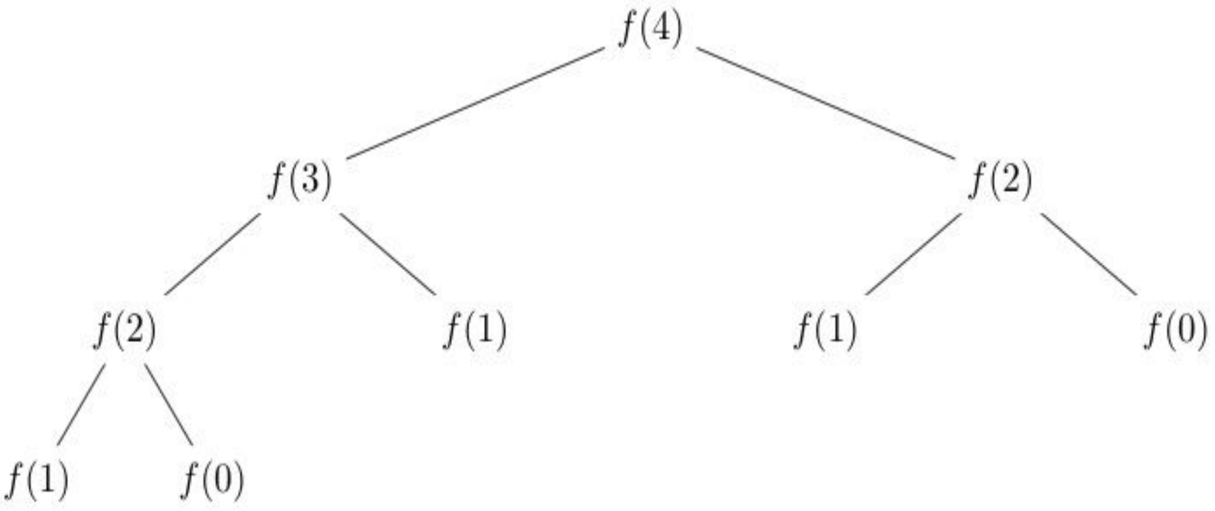
→ **So let's count up the number of function evaluations!**
(fun and suuuuuuper useful CS skill!)

Efficiency -- recursion versus iteration

Okay! So, we should have just seen that the recursive solution takes **A LOT** longer to compute f_{43} than the iterative solution.

→ f_{43} requires 1,402,817,465 function evaluations in the **recursive** solution!!

→ whereas the **iterative** solution computes each Fibonacci number **exactly one time**



Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Step 1) Break the input into parts that are smaller inputs to the problem.

→ Question: What's a smaller version of the problem?

→ Answer: A meeting with *n*-1 people

Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Step 2) Combine solutions with simpler inputs into a solution of the original problem.

`handshakes(n)` = [# *new handshakes from* n^{th} *person*] + `handshakes(n-1)`

Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Step 3) Find solutions to the simplest inputs.

Question: what are the simplest cases?

Answer: a meeting with only 1 person needs 0 handshakes

Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Step 4) Implement the solution by combining the simple cases and the reduction step.

```
int handshakes(int n) {
    if (n==1) {return 0;}
    return  [# new handshakes from nth person] + handshakes(n-1);
}
```

Recursion is *elegant*

Example: The **handshake problem**.

S'pose *n* people show up to a meeting. How many handshakes are needed for each person to have shaken every other person's hand exactly one time?

Step 4) Implement the solution by combining the simple cases and the reduction step.

```
int handshakes(int n) {
    if (n==1) {return 0;}
    return (n-1) + handshakes(n-1);
}
```

Last time: **selection sort** (Special Topic 6.2)

Input: **X** = [13, 3, 9, 5, 1]

Output: The **sorted** version of **X**, in increasing order: [1, 3, 5, 9, 13]

Step 1: Find the smallest element out of X[0 - end].
Swap X[0] and smallest element.

Step 2: Find the smallest element out of X[1 - end].
Swap X[1] and smallest element.

Step 3: Find the smallest element out of X[2 - end].
Swap X[2] and smallest element.

And so on…

Your mission is to ***rewrite a recursive version*** of the selection sort algorithm and ***compare the timing*** to the iterative version, for various sizes of input arrays/vectors.

What just happened?!

We just saw… **recursion!**

- A problem-solving approach in which we break the big problem down into smaller versions of that problem, and solve those.

1) Break the input into parts that are smaller inputs to the problem.

2) Combine solutions with simpler inputs into a solution of the original problem.

3) Find solutions to the simplest inputs.

4) Implement the solution by combining the simple cases and the reduction step.

Efficiency -- recursion versus iteration

Recursion:

```
int fib_r(int n) {
    if (n==0) {
        return 0;
    } else if (n==1) {
        return 1;
    } else {
        return fib_r(n-1) + fib_r(n-2);
    }
}
```

Efficiency -- recursion versus iteration

Iteration:

```
int fib_i(int n) {
    if (n==0) {
        return 0;
    } else if (n==1) {
        return 1;
    } else {
        int f[n+1];
        f[0] = 0;
        f[1] = 1;
        for (int i=2; i<=n; i++) {
            f[i] = f[i-1] + f[i-2];
        }
        return f[n];
    }
}
```