Review and Warmup: Recursion
CISC4080
CIS, Fordham Univ.

Instructor: X. Zhang

Goal

- Understand basic components of recursive algorithms: one or more base cases, and one or more general cases
- Trace execution of recursive algorithms
  - Call stack (detailed tracing)
  - Recursion trees
- Understand three questions rule for checking correctness of recursive algorithms

- Example of recursion algorithms studied:
  - Fibonacci sequence calculation
  - bubble sort, selection sort recursively, recursive thinking

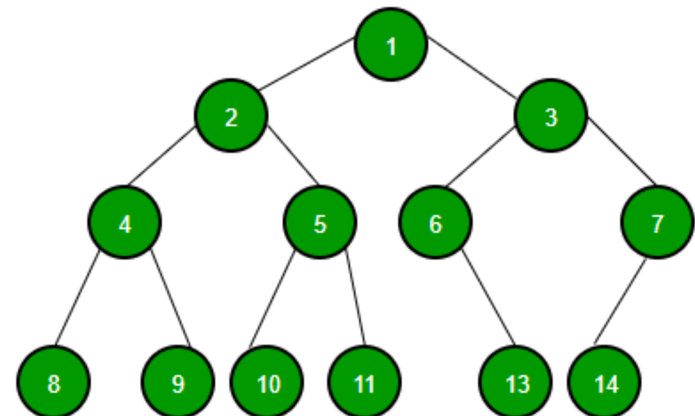
Recursion

- Recursion is a distinct algorithmic problem-solving technique. Essentially this technique boils down to creating smaller and smaller versions of the same problem until the smallest version is easily solved and then going in reverse to solve the larger problem piece by piece.

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Recursive View of Data Structure

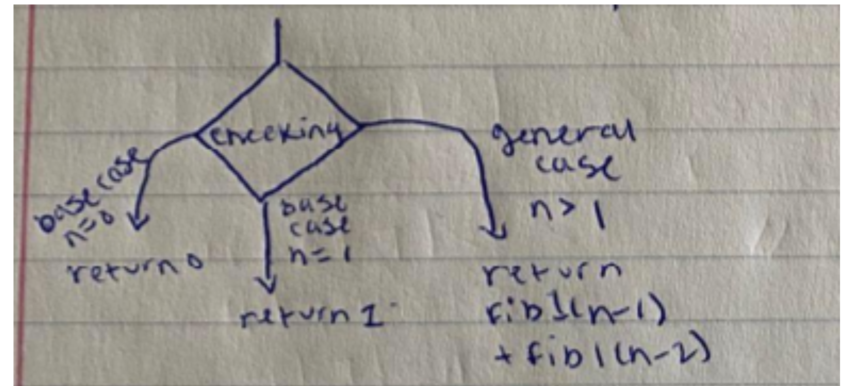
- Sublist in a list a[first….last]
  - Made up of a Left half and Right half, both sublists
  - Make up a[first] and the rest, a[first+1…last]
  - …
- Subtrees in a tree
  - Left subtree
  - Right subtree
- …
- Lead naturally to recursive al structures



Fib1

- Recursive function: a function that calls itself.
- All valid recursive function starts by checking if the problem instant belongs to which cases below
    - Base case(s): smallest problem instance that can be solved directly (i.e., without recursion)
    - General case(s): larger problem instance that is solved by reducing to smaller problems…

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```



Call stack: detailed on recursion

- a **call stack** is a stack data structure that stores information about active subroutines (i.e., functions) of a computer program.
  - An active subroutine is one that has been called, but is yet to complete execution.
-

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Demo: Let's look at what happened during fib1(3)

Whiteboard demo and gdb demo…

Call stack: detailed on recursion

- For each function call (i.e. active subroutine): its own local variables, parameters; and return address is stored in a **call stack frame**
  - Return address: the point to which the function call should return control when it finishes executing.

- Calling a function => push a (call stack) frame to call stack
- Returning from a function => pop the top frame from call stack
  - Control passed back to the saved return address
  - Resume execution from caller at the address

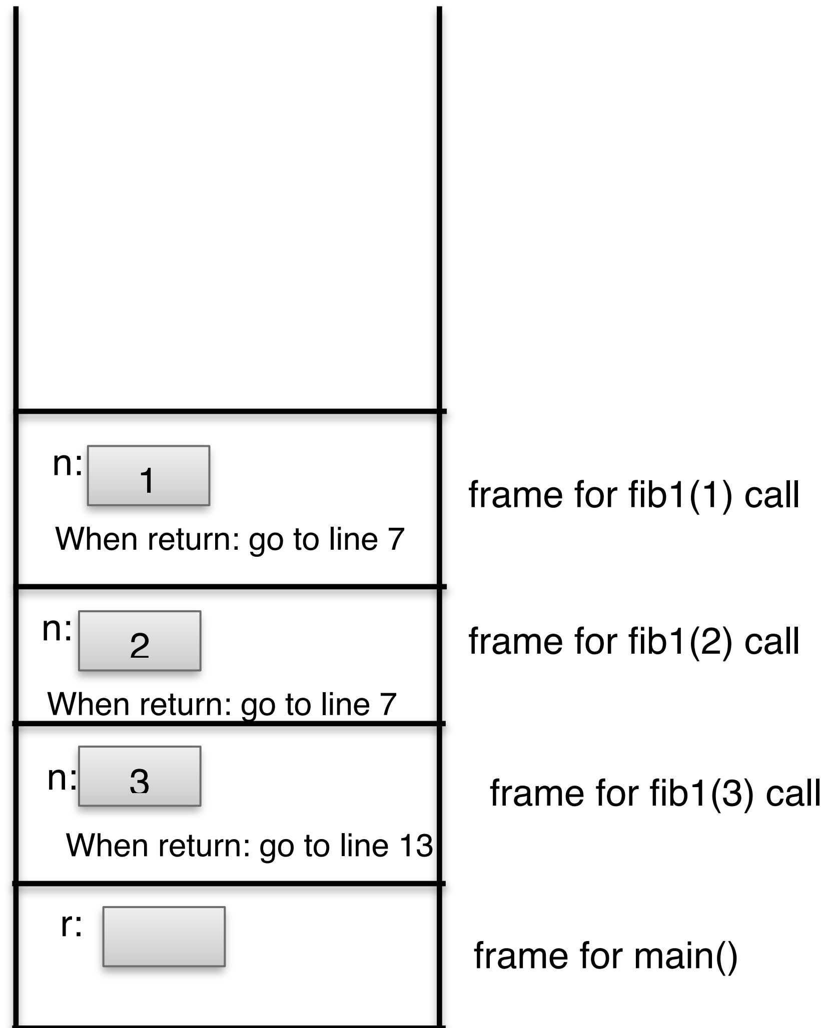
```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

Call stack: detailed on recursion

```
1. int fib1(int n)
2. {
3.     if (n==0)
4.         return 0;
5.     if (n==1)
6.         return 1;
7.     return  fib1(n-1) +
8.                fib1(n-2);
9. }

11.int main()
12.{
13.     int r= fib1(3);
14.     cout<<r<<endl;
15.}
```

Question: what's current function call?
What happens when this function returns?



Exercise

- What's the base cases of the following function? General cases?
- Trace the execution of following function call Exp (2,5).

```
1. int Exp (int a, int n)
2. {
3.      if (n==0)
4.          return 1;
5.      else if (n==1)
6.          return a;
7.      int result = Exp (a, n/2);
8.
9.      if (n%2==0)
10.             return result*result;
11.      else
12.             return result*result*a;
13.}
```

Tracing recursion: Recursion trees

- Trace recursive function call by drawing call stack is tedious
- We can just draw recursion tree:
  - Node: recursive call
  - Edges: calling
  - Label parameters and returning values

```
1. int Exp (int a, int n)
2. {
3.      if (n==0)
4.          return 1;
5.      else if (n==1)
6.          return a;
7.      int result = Exp (a, n/2);
8.
9.      if (n%2==0)
10.             return result*result;
11.      else
12.             return result*result*a;
13.}
```

Why this recursive function is correct?

```
function fib1(n)
if n = 0:  return 0
if n = 1:  return 1
return fib1(n - 1) + fib1(n - 2)
```

- Does it return the right value when n=0?
- How about for input instance n=1?
- How about n=2?
- How about n=5?
- How about n=32?
- Any n>=0 (assuming there is no integer overload problem)…

Why this recursive function is correct?

```
/* Calculate a^n, for n>=0 */
1. int Exp (int a, int n)
2. {
3.      if (n==0)
4.          return 1;
5.      else if (n==1)
6.          return a;
7.      int result = Exp (a, n/2);
8.
9.      if (n%2==0)
10.             return result*result;
11.      else
12.             return result*result*a;
13.}
```

Three questions rule

```
/* Calculate a^n, for n>=0 */
1. int Exp (int a, int n)
2. {
3.       if (n==0)
4.          return 1;
5.       else if (n==1)
6.          return a;
7.       int result = Exp (a, n/2);
8.
9.       if (n%2==0)
10.            return result*result;
11.      else
12.            return result*result*a;
13.}
```

To find an algorithm to solve your problem and to verify a recursive solution works, we must be able to answer "yes" to all three of these questions.

- Base Case Question: Is there a nonrecursive way out of the algorithm, and does the algorithm work correctly for this base case?

Three questions rule

```
/* Calculate a^n, for n>=0 */
1. int Exp (int a, int n)
2. {
3.      if (n==0)
4.          return 1;
5.      else if (n==1)
6.          return a;
7.      int result = Exp (a, n/2);
8.
9.      if (n%2==0)
10.             return result*result;
11.      else
12.             return result*result*a;
13.}
```

- **Smaller Caller Question:** Does each recursive call to the algorithm involve a smaller case of the original problem, leading **inescapably** to the base case?

Three questions rule

```
/* Calculate a^n, for n>=0 */
1. int Exp (int a, int n)
2. {
3.      if (n==0)
4.          return 1;
5.      else if (n==1)
6.          return a;
7.      int result = Exp (a, n/2);
8.
9.      if (n%2==0)
10.             return result*result;
11.      else
12.             return result*result*a;
13.}
```

- **General Case Question:** Assuming the recursive call(s) to the smaller case(s) work correctly, does the algorithm work correctly for the general case?

Practice Recursive Thinking

```
/* Finding Largest value in a[first…last]
    @param a: the list
    @param first, last: specify the range of the sublist
    @return largest value in a[first…last]*/
int Largest (a, first, last)
{


}
```

one bubbling round?

```
/*Bubble largest element to right as in bubble sort
   @param a: the list
   @param n: length of a
*/
bubbleRound (a, n)
{
       //scan list from left to right, compare each
adjacent pair of elements, swap them if they are in
wrong order
}
```

one bubbling round?

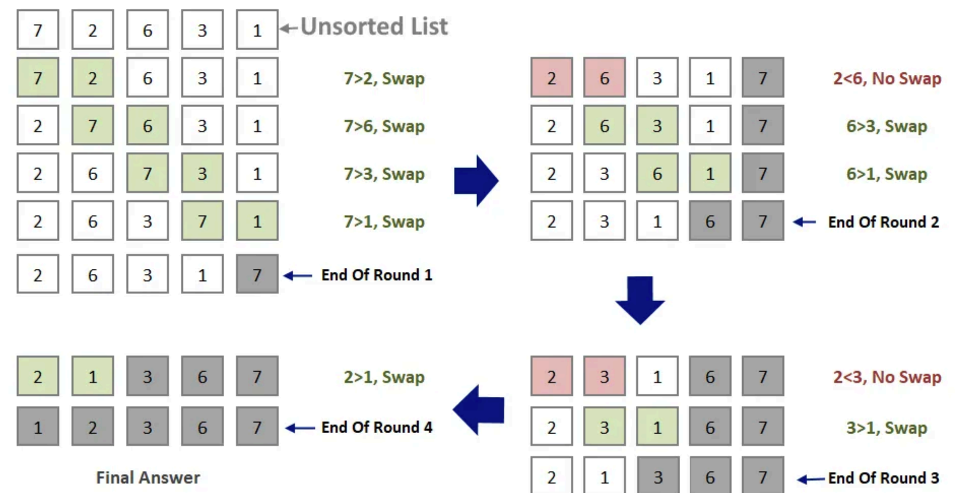
```
/*Bubble largest element to right as in bubble sort
   @param a: the list
   @param n: length of a
*/
bubbleRound (a, n)
{
        //scan list from left to right, compare each adjacent pair of elements, swap them if they are in wrong order
       for (int i=0; i<=n-1;i++)
           If (a[i] > a[i+1])
               swap (a[i], a[i+1])
}
```

Check boundary condition:
Look at boundary value for I, and see what's happens at these boundary condition:
i=0 => compare a[0] with a[1]
i=n-1 => compare a[n-1] with a[n-1+1]

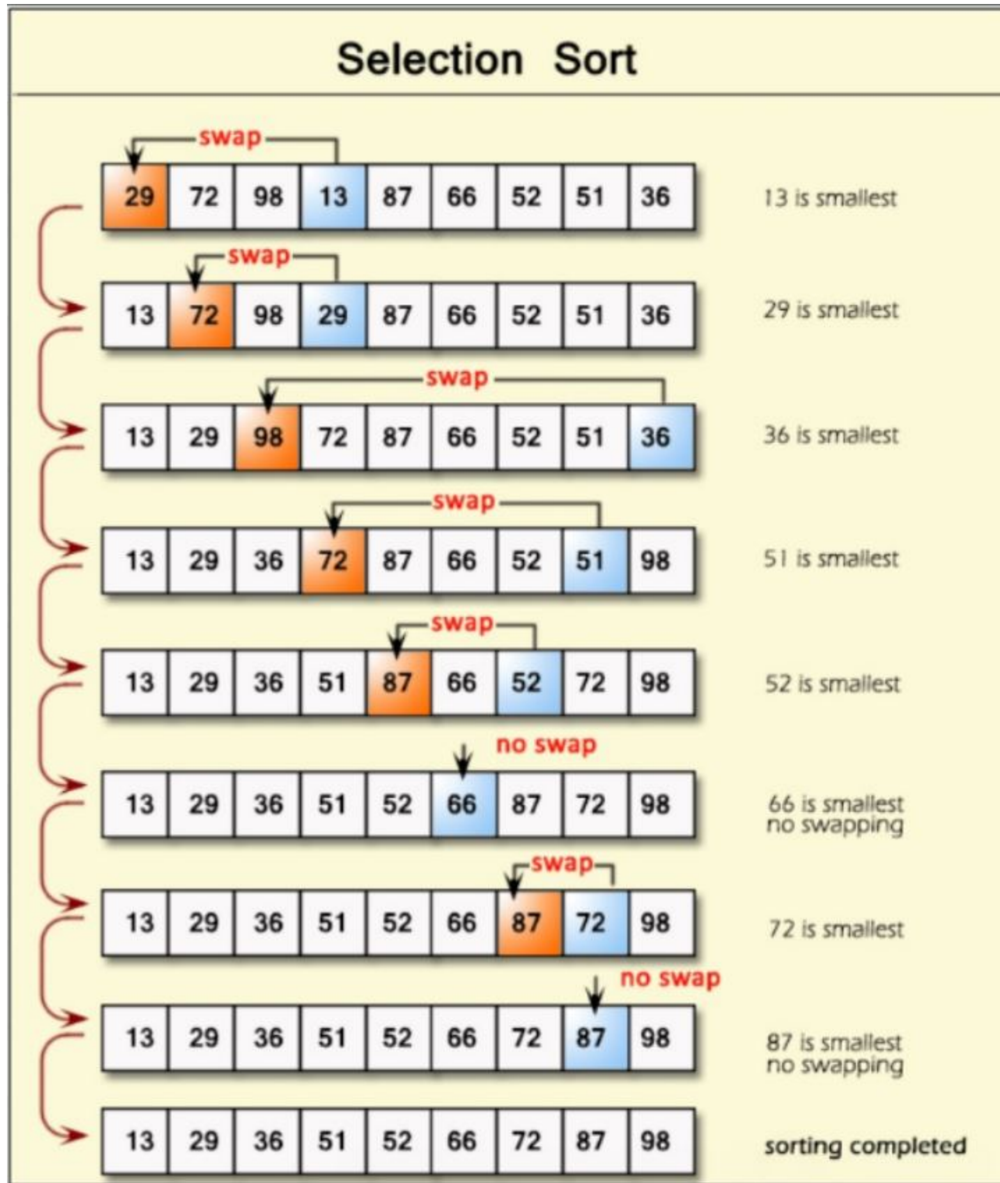
How to fix?

bubble sort recursively…

- In iterative implementation: we repeat n-1 rounds to sort whole list
  - or repeat until there is no swap in prev round
- Recursive thinking?
  - After round 1, we have a smaller problem in our hand



Selection sort



- First round: find location of smallest element, swap it with the front element
- Iteratively, we can repeat the above for n-2 times
- Recursively?