CSCI 103
More Recursion & Generating All Combinations

Mark Redekopp

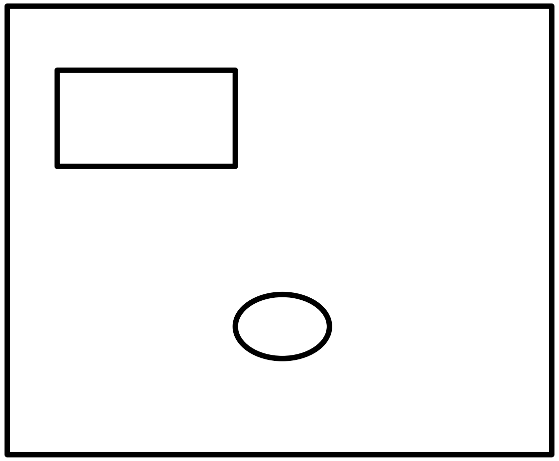
Get Some Code

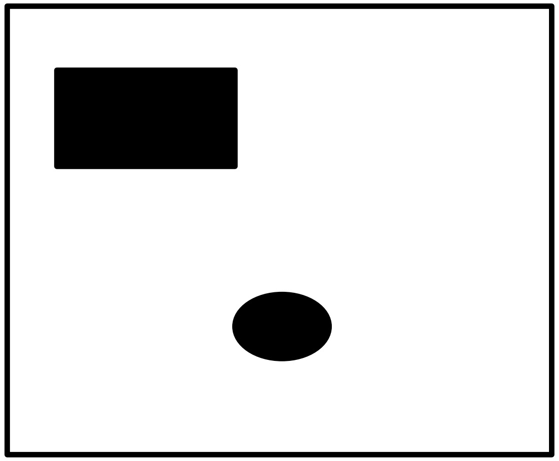
- In your VM create a new folder and cd into it
  - mkdir ffill
  - cd ffill
  - wget http://ee.usc.edu/~redekopp/cs103/floodfill.tar
  - tar xvf floodfill.tar

TRACING RECURSIVE ALGORITHMS

Flood Fill

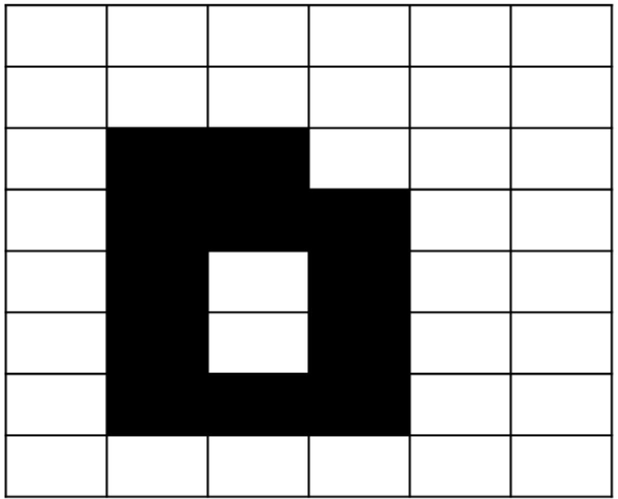
- Imagine you are given an image with outlines of shapes (boxes and circles) and you had to write a program to shade (make black) the inside of one of the shapes. How would you do it?
- Flood fill is a recursive approach
- Given a pixel
  - Base case: If it is black already, stop!
  - Recursive case: Call floodfill on each neighbor pixel
  - Hidden base case: If pixel out of bounds, stop!





Recursive Flood Fill

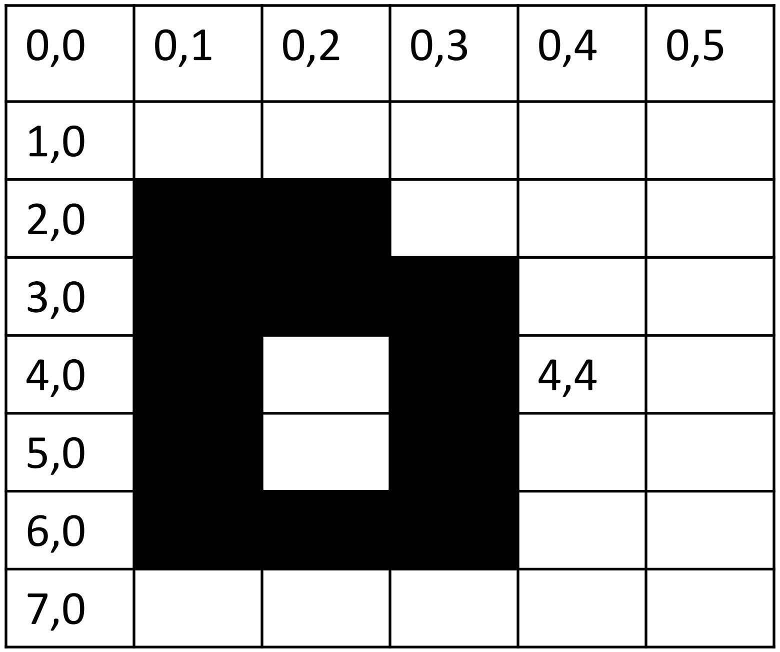
- Recall the recursive algorithm for flood fill?
  - Base case: black pixel, out-of-bounds
  - Recursive case: Mark current pixel black and then recurse on your neighbors



```
void flood_fill(int r, int c)
{
  if(r < 0 || r > 255 )
    return;
  else if ( c < 0 || c > 255){
    return;
  }
  else if(image[r][c] == 0){
    return;
  }
  else {
    // set to black
    image[r][c] = 0;
    flood_fill(r-1,c);  // north
    flood_fill(r,c-1);  // west
    flood_fill(r+1,c);  // south
    flood_fill(r,c+1);  // east

  }
}
```

Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?



Recursive Ordering

- Give the recursive ordering of all calls for recursive flood fill assuming N, W, S, E exploration order starting at 4,4
  - From what square will you first explore to the west?
  - From what square will you first explore south?
  - From what square will you first explore east?
  - What is the maximum number of recursive calls that will be alive at any point in time?
  - Notice recursive flood fill goes deep before it goes broad
  - Also notice that each call that is not a base case will make 4 other recursive calls

0,0	0,1	0,2	0,3	0,4	0,5
1,0					
2,0	■	■			
3,0	■	■	■		
4,0	■		■	4,4	
5,0	■		■		
6,0	■	■	■		
7,0					

Tracing Recommendations

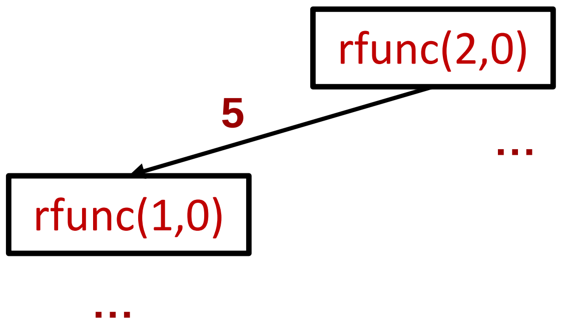
- Show the call tree
  - Draw each instance of a recursive function as a box and list the inputs passed to it
  - When you hit a recursive call draw a new box with an arrow to it and label the arrow with the line number of where you left off in the caller

Analyze These!

- What does this function print? Show the call tree?

- What is the runtime in terms of n?

```
00: void rfunc(int n, int t) {
01:    if (n == 0) {
02:      cout << t << " ";
03:      return;
04:   }
05:   rfunc(n-1, 3*t);
06:   rfunc(n-1, 3*t+2);
07:   rfunc(n-1, 3*t+1);
08: }
09: int main() {
10:    rfunc(2, 0);
11: }
```



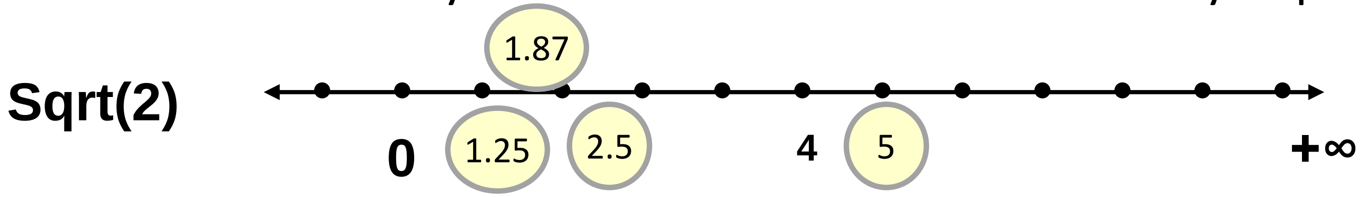
Analyze These!

- What does this function return for g(3122013)

```
int g(int n) {
   if (n % 2 == 0)
      return n/10;
   return g(g(n/10));
}
```

Recursive Helper Functions

- Sometimes we want to provide a user with a simple interface (arguments, etc.), but to implement it recursively we need additional arguments to our function
- In that case, we often let the top-level, simple function call a recursive "**helper**" function that provides the additional arguments needed to do the work
  - double sqrt(double x); // Simplified user interface
  - double sqrt(double x, double lo, double hi); // Helper function
- In-class-exercises
  - Find the square root of, x, without using sqrt function…
  - Pick a number, square it and see if it is equal to x
  - Use a binary search to narrow down the value you pick to square



GENERATING ALL COMBINATIONS

Recursion's Power

- The power of recursion often comes when each function instance makes ***multiple*** recursive calls
- As you will see this often leads to exponential number of "combinations" being generated/explored in an easy fashion

Binary Combinations

- If you are given the value, n, and a string with n characters could you generate all the combinations of n-bit binary?
- Do so recursively!

```
0
1
```

1-bit Bin.

```
00
01
10
11
```

2-bit Bin.

```
000
001
010
011
100
101
110
111
```

3-bit Bin.

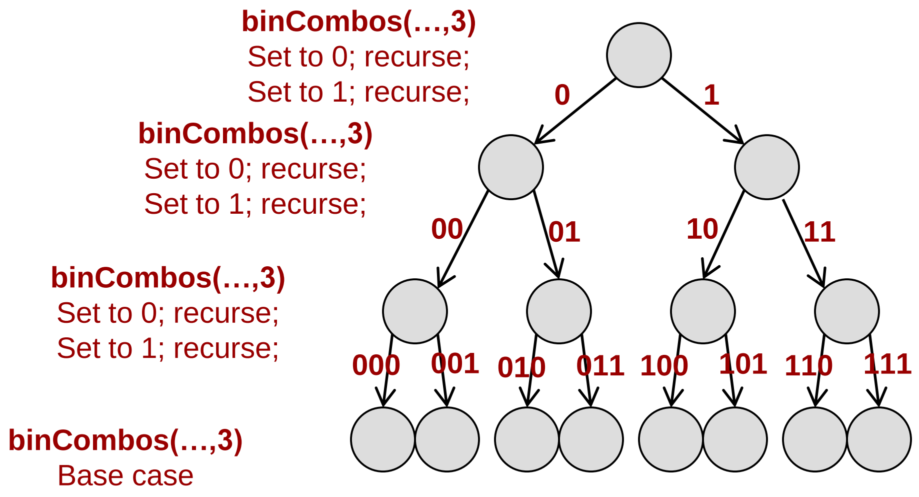
```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

4-bit Bin.

Exercise: bin_combo_str

Recursion and DFS

- Recursion forms a kind of Depth-First Search



```
// user interface
void binCombos(int len)
{
  binCombos("", len);
}
// helper-function
void binCombos(string prefix,
               int len)
{
  if(prefix.length() == len )
    cout << prefix << endl;
  else {
    // recurse
    binCombos(data+"0", len);
    // recurse
    binCombos(data+"1", len);
  }
}
```

Recursion and DFS (w/ C-Strings)

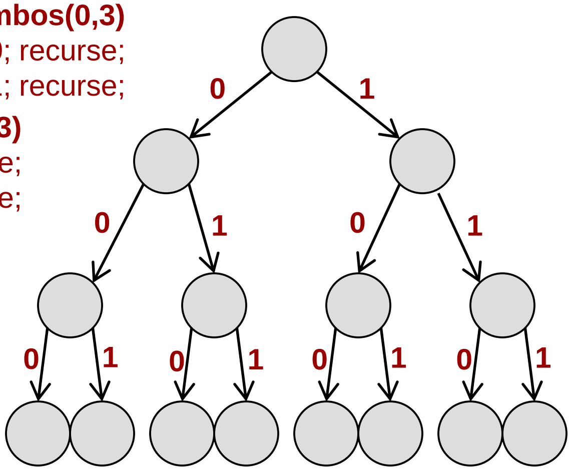
- Recursion forms a kind of Depth-First Search

binCombos(0,3)
Set to 0; recurse;
Set to 1; recurse;

binCombos(1,3)
Set to 0; recurse;
Set to 1; recurse;

binCombos(2,3)
Set to 0; recurse;
Set to 1; recurse;

binCombos(3,3)
Base case



```
void binCombos(char* data,
               int curr,
               int len)
{
  if(curr == len )
    data[curr] = '\0';
  else {
    // set to 0
    data[curr] = '0';
    // recurse
    binCombos(data, curr+1, len);
    // set to 1
    data[curr] = '1';
    // recurse
    binCombos(data, curr+1, len);
  }
}
```

Recursion and Combinations

- Consider the problem of generating all 2-length combinations of a set of values, S.
  - Ex. Generate all length-n combinations of the letters in the set S={'U','S','C'} (i.e. UU, USS, UC, SU, SS, SC, CU, CS, CC)
  - How could you do it with loops (how many would you need)?
- Consider the problem of generating all 3-length combinations of a set of values, S.
  - Ex. Generate all length-n combinations of the letters in the set S={'U','S','C'} (i.e. UUU, UUS, UUC, USU, USS, USC, etc.)
  - How could you do it with loops (how many would you need)?

```
void usccombos2()
{
  char str[3] = "--";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
       str[1] = vals[j];
       cout << str << endl;
    }
  }
}
```

```
void usccombos3()
{
  char str[3] = "--";
  char vals[3] = {'U','S','C'};
  for(int i=0; i != 3; i++){
    str[0] = vals[i];
    for(int j=0; j != 3; j++){
       ...

    }
  }
}
```

Recursion and Combinations

- Recursion provides an elegant way of generating all **n**-length combinations of a set of values, S.
  - Ex. Generate all length-**n** combinations of the letters in the set **S**={'U','S','C'}
  - You would need **n** loops. But we don't have a way of executing a "variable" number of loops…Oh wait! We can use recursion!
- General approach:
  - Need some kind of **array/vector/string** to store partial answer as it is being built
  - Each recursive call is only responsible for one of the **n** "places" (say location, **i**)
  - The function will iteratively (loop) try each option in **S** by setting location i to the current option, then recurse to handle all remaining locations (i+1 to n)
    - Remember you are responsible for only one location
  - Upon return, try another option value and recurse again
  - Base case can stop when all n locations are set (i.e. recurse off the end)
  - Recursive case returns after trying all options

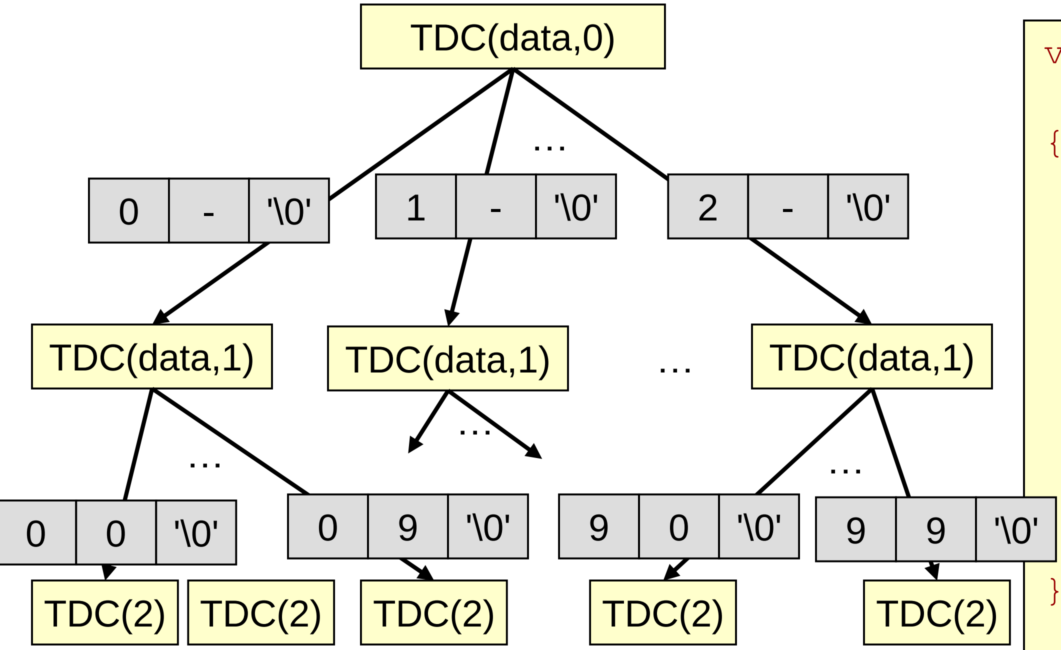
Recursion Analysis

- What would this code print for
  - X=3, y=2
  - X=10, y=1
  - X=2, y=3

```
#include <iostream>
#include <string>
using namespace std;

void mystery(int r, string pre, int n) {
   if(pre.length() == n){
      cout << pre << endl;
   }
   else {
      for(int i=0; i < r; i++){
         char c = static_cast<char>('0'+i);
         mystery(r, pre + c, n);
      }
   }
}

int main() {
   int x, y;
   cin >> x >> y;

   string pre;

   mystery(x, pre, y);

   return 0;
}
```

Generating All Combinations

- Recursion offers a simple way to generate all combinations of N items from a set of options, S
  - Example: Generate all 2-digit decimal numbers (N=2, S={0,1,…,9})



```
void TwoDigCombos(char data[3],
                  int curr)
{
  if(curr == 2 )
    cout << data;
  else {
    for(int i=0; i < 10; i++){
      // set to i
      data[curr] = '0'+i;
      // recurse
      TwoDigCombos(data, curr+1);
  }
}
```

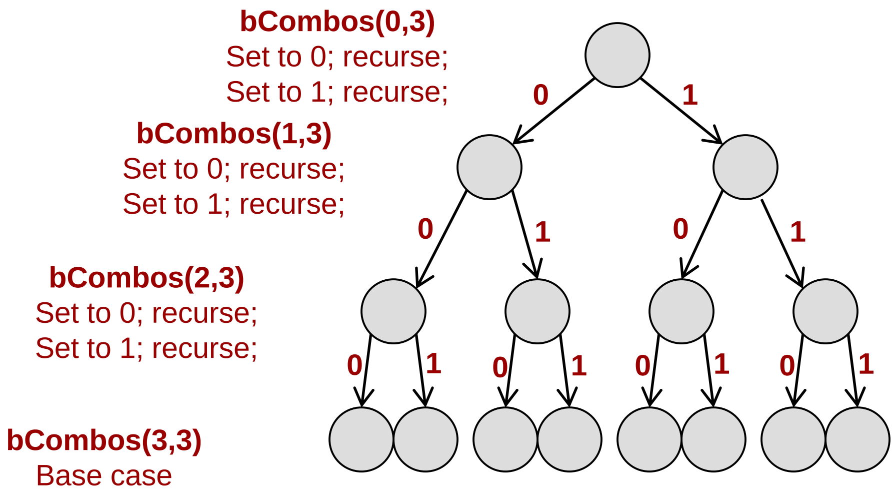
Exercises

- bin_combos_str
- basen_combos
- Zero_sum
- Prime_products_print
- Prime_products
- all_letter_combos

Recursion and DFS (w/ C-Strings)

- Answer: All combinations of base x with y digits

```
#include <iostream>
#include <string>
using namespace std;

void basen_combos(int r, string pre, int n)  {
   if(prefix.length() == n){
      cout << pre << endl;
   }
   else {
      for(int i=0; i < r; i++){
         char c = static_cast<char>('0'+i);
         basen_combos(r, prefix + c, n);
      }
   }
}

int main()  {
   int base, numDigits;
   cin >> x >> y;

   string pre;

   basen_combos(x, pre, y);

   return 0;
}
```



Another Exercise

- Generate all string combinations of length n from a given list (vector) of characters

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

void all_combos(vector<char>& letters, int n) {

}

int main() {
   vector<char> letters;
   letters.push_back('U');
   letters.push_back('S');
   letters.push_back('C');

   all_combos(letters, 2);

   all_combos(letters, 4);

   return 0;
}
```

Knapsack Problem

- Knapsack problem
  - You are a traveling salesperson. You have a set of objects with given weights and values. Suppose you have a knapsack that can hold N pounds, which subset of objects can you pack that maximizes the value.
  - Example:
    - Knapsack can hold 35 pounds
    - Object A: 7 pounds, $12.50 ea. Object B: 10 pounds, $18 ea.
    - Object C: 4 pounds, $7 ea. Object D: 2.4 pounds, $4 ea.
- Get the code:
  - $ wget http://ee.usc.edu/~redekopp/cs103/knapsack.cpp

MERGESORT

Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Bubble Sort (simple though not terribly efficient)
  - On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)

List	7	3	8	6	5	1
index	0	1	2	3	4	5

Original

List	3	7	6	5	1	8
index	0	1	2	3	4	5

After Pass 1

List	3	6	5	1	7	8
index	0	1	2	3	4	5

After Pass 2

List	3	5	1	6	7	8
index	0	1	2	3	4	5

After Pass 3

List	3	1	5	6	7	8
index	0	1	2	3	4	5

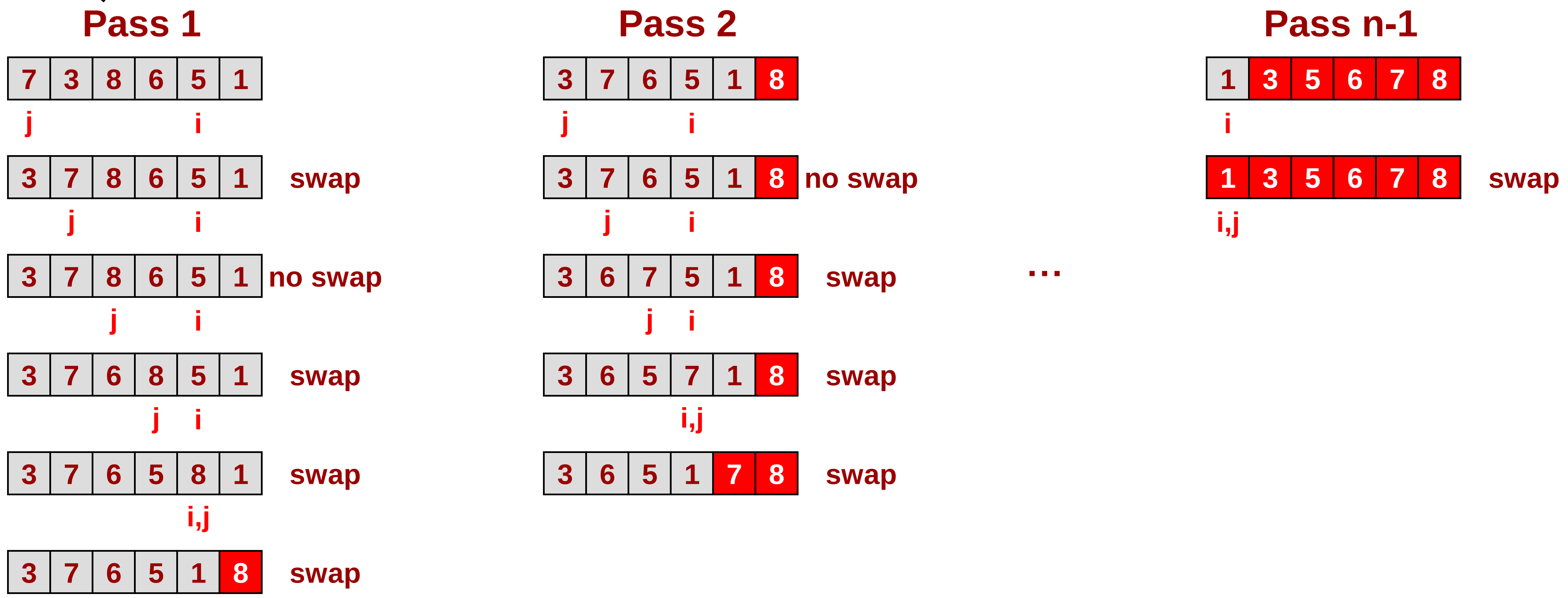
After Pass 4

List	1	3	5	6	7	8
index	0	1	2	3	4	5

After Pass 5

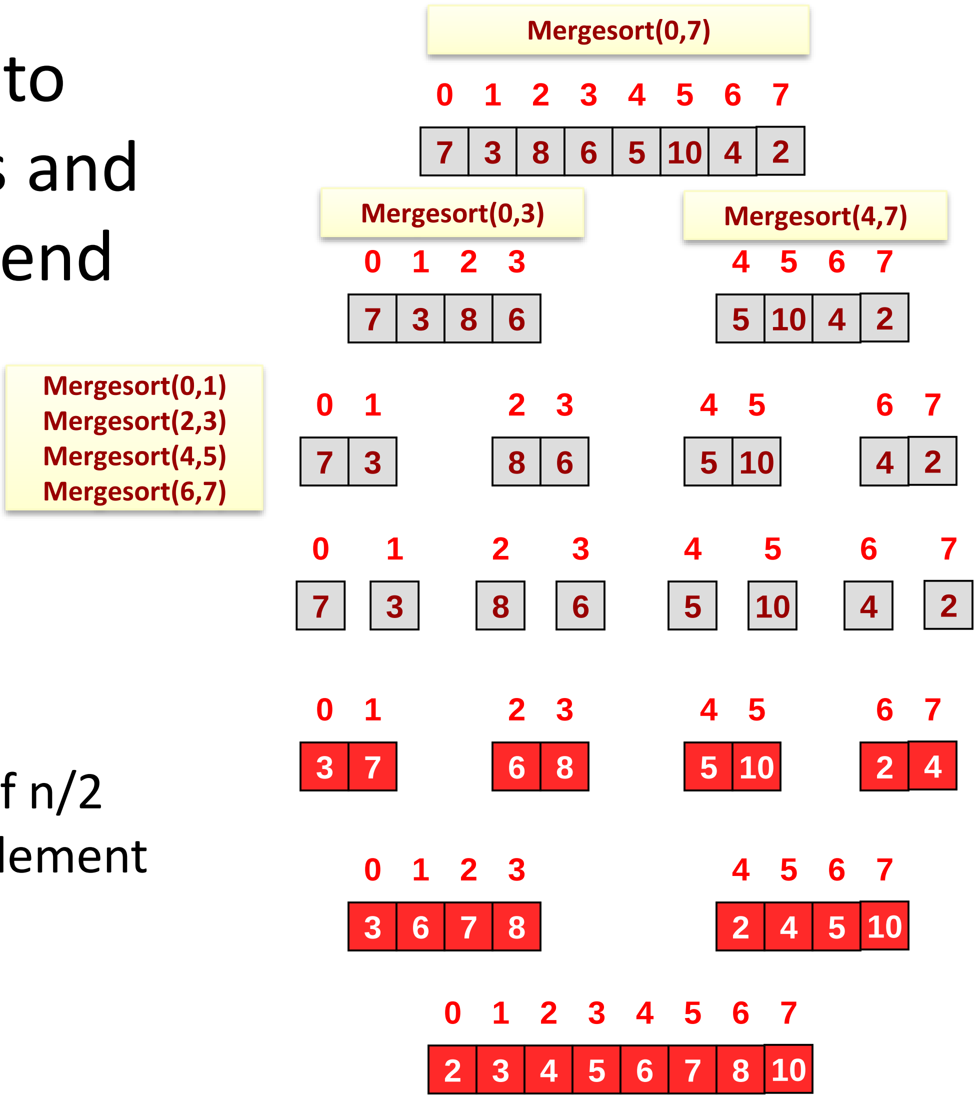
Bubble Sort Algorithm

```
n ← length(List);
for( i=n-2; i >= 1; i--)
  for( j =1; j <= i; j++)
    if ( List[j] > List[j+1] ) then
      swap List[j] and List[j+1]
```



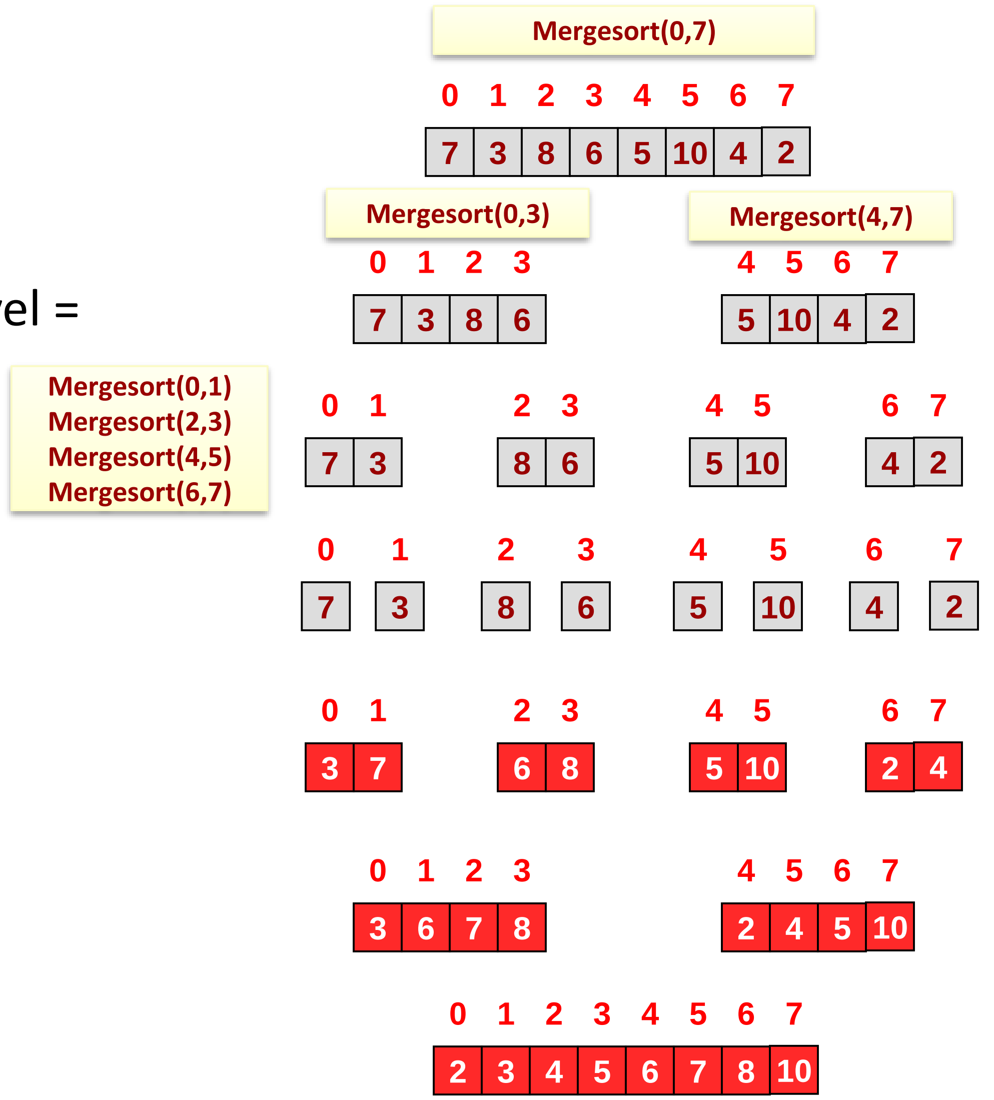
Recursive Sort (MergeSort)

- Break sorting problem into smaller sorting problems and merge the results at the end
- Mergesort(0..n-1)
  - If list is size 1, return
  - Else
    - Mergesort(0..n/2)
    - Mergesort(n/2+1 .. n-1)
    - Combine each sorted list of n/2 elements into a sorted n-element list



Recursive Sort (MergeSort)

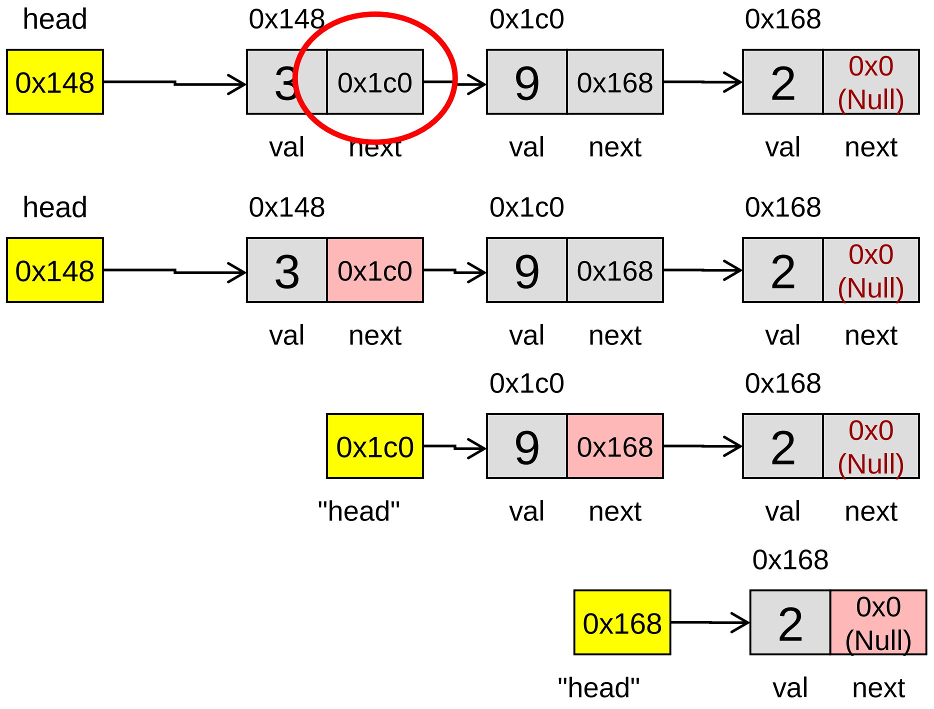
- Run-time analysis
  - # of recursion levels =
    - $\text{Log}_2(n)$
  - Total operations to merge each level =
    - n operations total to merge two lists over all recursive calls
- Mergesort = $O(n * \log_2(n))$



RECURSION & LINKED LISTS

Linked Lists and Recursion

- Consider a linked list with a head pointer
- If I gave you head->next, isn't that a "head" pointer to the n-1 other items in the list?



Monkey Around

- In-Class exercises
  - Monkey_recurse
  - Monkey_recback
  - List_max
  - Monkey_reverse