Chapter 4- Recursion

4.1 Recursive vs Non-recursive Definition

4.2 User stack for Factorial

4.3 Finding the Smallest in an array with Recursion

4.4 Fibonacci numbers with Recursion

4.5 Backtracking

4.6 Variations in recursion

4.7 Comparison Based Sorting

4.8 Exercises

4.1 Recursive versus Non-recursive Definitions

A recursive definition of a "concept" is one that uses the "concept" in the definition itself!

For example, if we say is "Saturday is the day that comes seven days after the previous Saturday"! We are defining Saturday recursively.

Or if we describe the process of finding sum as follows

"To find the sum of first 100 numbers, first find the sum of first 99 numbers and then add a 100"

Note that to find the sum of first 99 numbers, we need to first find the sum of first 98 numbers and add 99. etc.

4.1 Non Recursive factorial

To define factorial of a number n, we could use

```
factorial(n) = 1*2*3* ... *(n-1) *n, n >0
```

Note that the above definition is non-recursive (also called iterative)

To find the factorial(4), using the non-recursive definition, we would find

```
factorial(4) = 1*2*3*4 = 24
```


4.1 Recursive factorial

But here is a recursive version

```
factorial(n) = n*factorial(n-1);  n > 0
```

To find the factorial(4), using the recursive definition, we would find

```
factorial(4) = 4*factorial(3); (substituting
n=4 in the definition above)
factorial(3) = 3*factorial(2); (substituting
n=3 in the definition above)
factorial(2) = 2*factorial(1); (substituting
n=2 in the definition above)
factorial(1) = 1*factorial(0); (substituting
n=1 in the definition above)

WHEN DO WE STOP?
```


4.1 Recursive factorial

So we need a stopping condition (also called *starting condition* or *anchor case* or *base case* or *ground case*)

```
factorial(n) = n*factorial(n-1);  n >0
factorial(0) = 1;  //stopping condition
```

To find the factorial(4), using the recursive definition, we would find

```
factorial(4) = 4*factorial(3);
factorial(3) = 3*factorial(2);
factorial(2) = 2*factorial(1);
factorial(1) = 1*factorial(0);
We stop here, since stopping condition has
reached. Now we start computing values
factorial(1) = 1*factorial(0) = 1* 1 = 1
factorial(2) = 2*factorial(1) = 2 * 1 = 2;
factorial(3) = 3*factorial(2) = 3 * 2 = 6;
factorial(4) = 4*factorial(3) = 4 * 6 = 24;
```


4.1 Recursive Code for Factorial

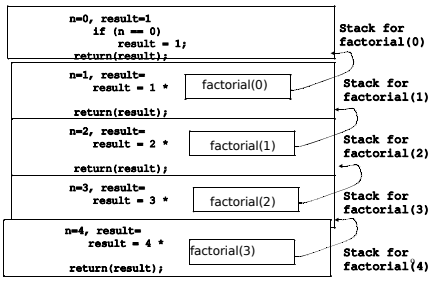
```
public int factorial(int n){
  if (n == 0)
    return(1);
  else
    return(n * factorial(n-1));
}
```

4.1 Recursive Code of Factorial

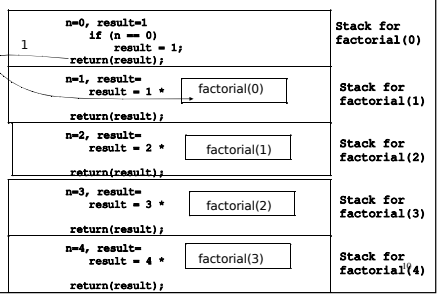
```
public int factorial(int n){
  int result;
  if (n == 0)
    result = 1;
  else
    result = n * factorial(n-1);

    return(result);
}
```


4.2 User Stack for recursive factorial

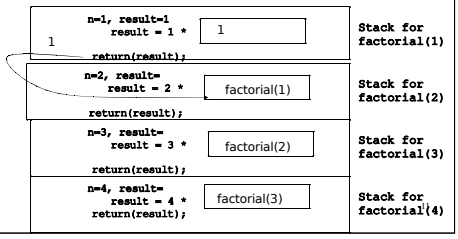
4.2 Stack for Factorial of 4.



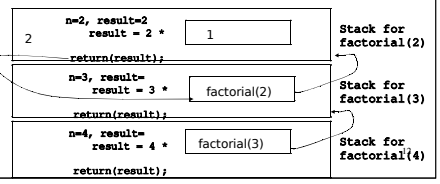
4.2 Stack for Factorial of 4.



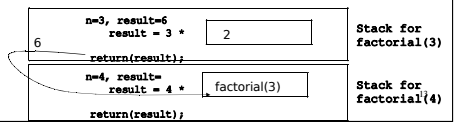
4.2 Stack for Factorial of 4.



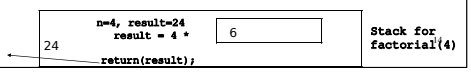
4.2 Stack for Factorial of 4.



4.2 Stack for Factorial of 4.



4.2 Stack for Factorial of 4.



4.3 Finding Smallest in an array with recursion

4.3 Recursive Code for Finding Smallest

Smallest element in an array can be found recursively too!

findSmallest(arr, 0) = Step1: smallI = findSmallest(arr, 1);
Step 2: find minimum(arr[0], arr[smallI]);
findSmallest(arr, 1) = Step1: smallI = findSmallest(arr, 2)
Step 2: find minimum(arr[1], arr[smallI]);
etc

findSmallest(arr, n-1) = Step1: smallI = findSmallest(arr, n-2)
Step 2: minimum(arr[n-1], arr[smallI]);

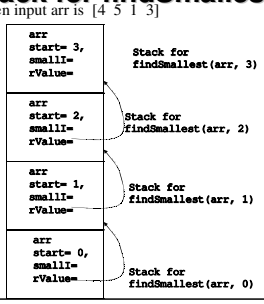
WHAT IS THE STOPPING CONDITION?

4.3 Recursive Code for Finding Smallest

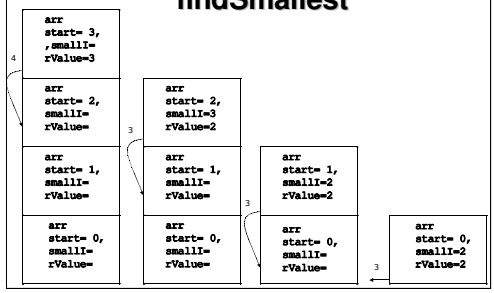
```
public int findSmallestIndex(T[] arr){
        findSmallest(arr, 0);
}

public int findSmallest(T[] arr, int start){
    int smallI, rValue;
   if ( start == arr.length-1)
     rValue = start;
   else{
     smallI findSmallest(arr, start+1);
     if (arr[smallI]) < arr[start])
        rVlaue = smallI;
     else
        rValue = start;
   }
   return(rValue);
}
```


4.3 Stack for findSmallest

Stack when input arr is [4 5 1 3]



4.3 Return Stack for findSmallest



4.4 Fibonacci Numbers with Recursion

4.4 Fibonacci Numbers

Fibonacci numbers are defined below

fib(n) = fib(n-1) + fib(n-2); n > 1

fib(0) = fib(1) = 1.

fib(4) = fib(3) + fib(2) fib(3) = fib(2) + fib(1)

fib(2) = fib(1) + fib(0)

fib(2) = 1 + 1 = 2 fib(3) = 2 + 1 = 3

fib(4) = 3+2 = 5

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


4.4 In Class Activity

Show the stack for fib, when n = 4.

4.5 Backtracking

4.5 Backtracking

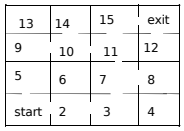
When searching for an item, the technique of systematically searching all possible routes is called backtracking.

Recursion allows backtracking naturally, since you come back to the previous stack when one execution is complete. Now you can search other paths.

For example, when you are travelling a city, searching for a treasure, how do we go about trying to search for a treasure systematically?

4.5 Travelling a maze

Let us we want to find the path from *start* to *exit* in the maze below. Basically we are searching for square marked *exit*



mark the square. If square is *exit*, return *success*
If not, search left square , (if no wall and unmarked)
If failed, search right, (if no wall and unmarked)
If failed, search up, (if no wall and unmarked)
If failed search down, (if no wall and unmarked)

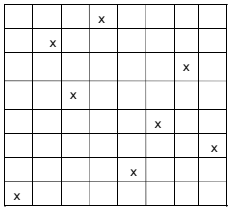
Travelling a maze-in class activity

Show the stack for travelling the maze above.

4.5 Eight Queens Problem

```
boolean putQueen(int row){
    if (row = 9)  return(true);
    else{
        triedAll and done are set to false;
        while (not done and not triedAll){
            search from left to right for a col on the same row that
            does not conflict; place the next queen in col;
            done = putQueen(row+1));
            if nothing is available, set triedAll=true;
        }
        if (triedAll) { remove queen from col; return(false);}
        else { return(done)};
}
```


4.5 Solution to Eight Queens Problem



4.6 Variations in Recursion

4.6 Variations in Recursion

Tail Recursion is one which there is only one recursive call in the end of the code.
factorial (n) = n*factorial(n-1);

Non-tail recursion has more recursive call than one
fib(n) = fib(n-1) + fib(n-2);

Indirect Recursion is one where
A calls B, B calls C, C calls D, D calls A.

4.7 Sorting Algorithms

Comparison Based Sorting

1. Selection Sort
2. Insertion Sort
3. MergeSort

Selection Sort: Introduction

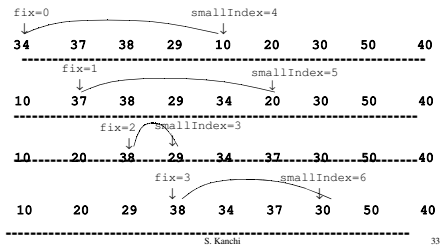
The general idea is that smallest element is swapped with the element at the first position, the next smallest is swapped with the element at the second position etc. This is a *brute force* technique

Initially, both *fix* and *smallIndex* point at the first number. *selectionSort* then checks which numbers from A[0] to the end is the smallest and returns the *smallIndex*. Then, the numbers in *fix* and *smallIndex* are swapped.

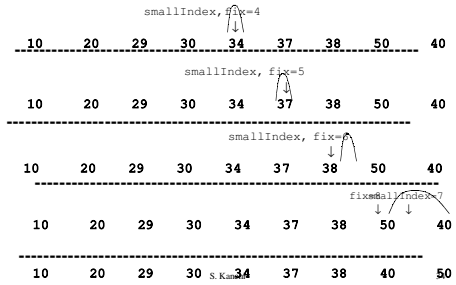
After that *fix* points to the next number, A[1], and then *selectionSort* looks for the smallest number from A[1] to the end. Then *selectionSort* swaps *smallIndex* and *fix*.

The whole search and swap process continues until *fix* reaches (n-2)nd position.

Selection Sort : Demonstration-1



Selection Sort : Demonstration-2



Selection Sort : Algorithm

```
public void selectionSort (T[] data,int min, int max){
     int fix, sI;
     T temp;

     for (fix = min, fix < max-1; fix++){
       sI = findSmallestIndex(data,fix,max);
       temp = data[sI];
       data[sI] = data[fix];
       data[fix] = temp;
     }
}
```


Selection Sort: Algorithm

```
private int findSmallestIndex(T[] data,int low,int high){
{
     int i, smallIndex;
      smallIndex = low;
      for (i=low + 1; i <= high; i++) {
          if (data[i].compareTo(data[smallIndex]) < 0)
       smallIndex = i;
      }
      return(smallIndex);
}
```

Insertion Sort: Introduction

Insertion sort(bubble sort) works by inserting every element by comparing it with previous elements. First, second number is compared with first and if needed it is swapped with the first. So, the first number and second number are in the right place relative to each other. Now, the third number compared to second number and then the first number and inserted in position one, two or three. This process continues until all numbers are put in the right place.

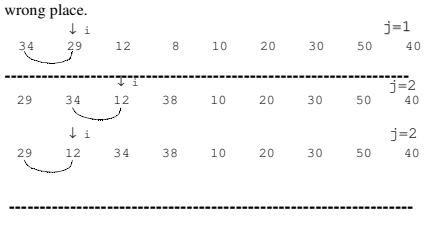
12 29 34 38 13 20 30 50 40

13 is put in position number 2, by moving 38, 34 and 29 to the right

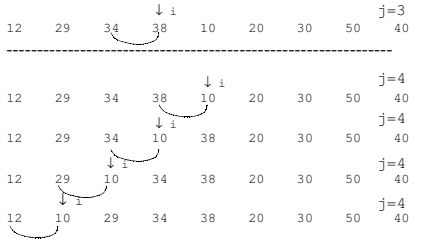
12 13 29 34 38 20 30 50 40

Insertion Sort :Demonstration-1

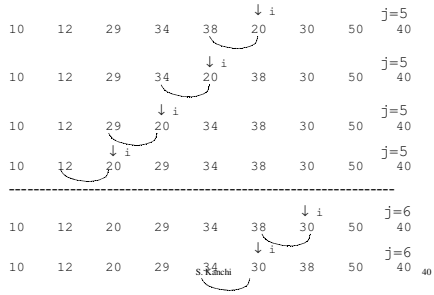
The idea of insertion sort is to insert elements if they are in the wrong place.



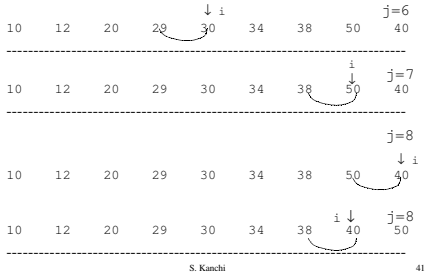
Insertion Sort :Demonstration-2



Insertion Sort :Demonstration-3



Insertion Sort :Demonstration-4



Insertion Sort: Algorithm

```
public void insertionSort(T[] arr, int min, int max){
  int i, j;
   for (j = min+1; j <= max; j++){
     i = j;
     while (i > min && (data[i].compareTo(data[i-1]) <
0 ){
         temp = data[i];
         data[i] = data[i-1];
         data[i-1] = temp;
         i = i-1;
     }
   }
}
```

Insertion Sort: Worst Case Analysis

Worst Case: Array is reverse sorted. So in each iteration to fix the j^{th} position there are j comparisons and swaps.

$$\sum_{j=1}^{j=n} j = \sum_{j=1}^{j=n} j - 1 = \frac{n(n+1)}{2} - 1 = O(n^2)$$

Merge Sort: Introduction

In merge sort, an array is divided equally into two parts. Each part is sorted and then the two sorted arrays are merged. This is a *divide-and-conquer* technique

For example,

34 43 12 23 67 22

Is divided into two arrays

34 43 12 and 23 67 22

Each is sorted. We get

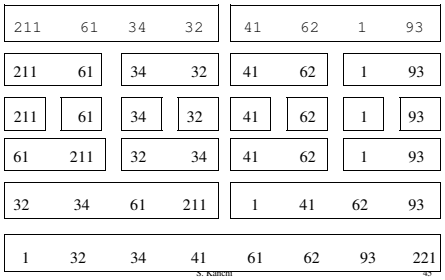
12 34 43 and 22 23 67.

Now the two arrays are merged into

12 22 23 34 43 67.

However, each array is sorted using recursive merge sort again.

Merge Sort :Demonstration



Merge Sort:Algorithm

```
pubic void mergeSort (T[] data, int min, int max){
  int midPoint;
  if (min < max){
    midPoint = (low + high)/2 ;
   a = mergeSort (low, midPoint);
   b = mergeSort (midPoint + 1, high);
   c = merge (a,b);
  }
}
```

NOTE : THE CODE ABOVE IS INCOMPLETE AND HAS INCORRECT SYNTAX.. YOU NEED TO REWRITE THE CODE.

Every time there is a merge, an array of size N is created.

Merge Sort: Analysis

There are logn merge steps. Each merge step takes O(n) time, the time complexity is O(nlogn).

4.8 Exercises

1. Write the merge code for mergesort. Analyze the time complexity of merge algorithm.
2. Convert each sorting algorithm to use linked list instead of an array. Analyze the new time and memory complexity.
3. Run the selction sort on the following array. Show the pointers smallestIndex and fix.
   56 45 67 78 90 34 45 8
4. Run the insertion sort on the following array. Show the i and j .
   56 45 67 78 90 34 45
5. Show the demonstration of mergesort on the following array.
   56 45 67 78 90 34 45 8

4.8 Exercises

6. Describe the worst case and best case time complexity scenarios of insertion sort and merge sort and state the best case and worst case time complexities.

7. A sorting method is "stable" if equal keys remain in the same relative order in the sorted list as they were in the original list. That is, let us say that L[i] = L[j] and i < j in the original list and after the sorting,
let us say L[i] was moved to L[k] and L[j] was moved to L[m], then sort is stable only if k < m. Which of the following sorting algorithms are stable? For the ones that are not stable give an example to indicate why it is not stable.

a. Insertion Sort
b. Selection sort
c. Mergesort

4.8 Exercises

8. If there are repeated elements in the array, which algorithm needs a change among the selection sort, insertion sort, and merge sort?

4.8 Exercises

9. Write a recursive code for binary search in a sorted array. Show the stack to perform binary search on 45 78 89 800 999 9992 67888 with key as 90

10. Write a recursive code for computing x^n, given x and n (n>=0). Show the stack for the call of 5^4

11. Write recursive code converting a string with digits into a number, for example "1234" should become 1234

12. Write a recursive code for checking if a word is a palindrome. (reads the same from left to right and right to left such as "madam")

4.8 Exercises

13. Write a non-recursive code for computing Fibonacci numbers.

14. Write the shortest program you can that uses recursion!

15. Write a recursive and non-recursive version to print a non-negative number in binary. You cannot use any bitwise operators.