

Recursion

Definition

(American Heritage Dictionary, 2000)

recursion

NOUN: *Mathematics* 1. An expression, such as a polynomial, each term of which is determined by application of a formula to preceding terms.
2. A formula that generates the successive terms of a recursion.

e.g., $x = 2*(1-x)$ or $x_n = 2*(1-x_{n-1})$

Some Low-hanging Fruit

- Factorials

- $n! = n \cdot (n-1)!$ and $0! = 1$
- the definition is recursive
- you'd probably calculate this with a loop

- Sums of consecutive integers

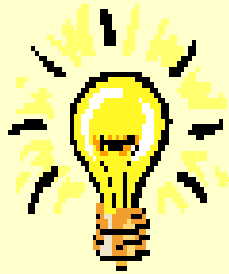
- $$\sum_{i=1}^N i = N + \sum_{i=1}^{N-1} i$$
- again, it's naturally recursive
- calculate with a loop, or use the formula $N \cdot (N+1) / 2$

Problem: Sort a large array

The simple sort algorithms take 4 times as long to sort an array of length $2N$ as they do to sort an array of length N .

- I can sort two arrays of length N much faster than I can sort one array of length $2N$.
- Recall that bubble sort (worst case) requires N^2 steps to sort an array of size N
 - so an array of size 10 may require 100 steps, while an array of size 20 may require 400 steps.
 - when the array size doubles, the time increases by a factor of 4.

Problem: Sort a large array



- Bright Idea: Split a large array into two equal halves, and sort each of them separately?
 - This might save us some time.

Problem: Sort a large array

- Some extra work (overhead) needs to be done:
 - I need to split the array into two halves
 - it's not hard to do, but it will take some time.
 - Combine two small sorted arrays into one large sorted array
 - this is called a *merge* operation
 - it is awkward to do this without using extra storage space
 - the length of the merged array is the sum of the lengths of the arrays being merged.

PseudoCode for our sort algorithm

MergeSort(array A[0 .. N])

```
/*assume we have an efficient sort algorithm,  
  sortSublist, that can sort smaller arrays */
```

```
mid = N/2 // find the middle element  
array L[0 .. mid] = A[0 .. mid] // get the left half  
sortSublist (L[ ]) // sort the left half  
array R[0 .. N-mid-1] = A[1+mid .. N] // get the right half  
sortSublist (R[ ]) // sort the right half  
A[] = merge(L[ ], R[ ]) // combine two sorted arrays  
return
```

Improvement: handle trivial lists (with 0 or 1 items) without delay

```
MergeSort(array A[0 .. N])  
  /*assume we have an efficient sort algorithm,  
   sortSublist, for sorting smaller arrays */  
  // a list with fewer than two elements is already sorted  
  if (length of A[ ] is greater than one)  
    mid = N/2  
    array L[0 .. mid] = A[0 .. mid]  
    sortSublist (L[ ])  
    array R[0 .. N-mid-1] = A[1+mid .. N]  
    sortSublist (R[ ])  
    A[ ] = merge(L[ ], R[ ])  
  return
```

What algorithm should we use to sort the smaller lists?

- Why not use this Mergesort itself? It now works for the smallest lists, so it should work for any list.

MergeSort(array A[0 .. N])

if (length of A[] is greater than one)

mid = N/2

array L[0 .. mid] = A[0 .. mid]

MergeSort (L[])

array R[0 .. N-mid-1] = A[1+mid .. N]

MergeSort (R[])

A[] = merge(L[], R[])

return

MergeSort – Space Requirement

MergeSort(array A[0 .. N])

if (length of A[] is greater than one)

mid = N/2

array L[0 .. mid] = A[0 .. mid]

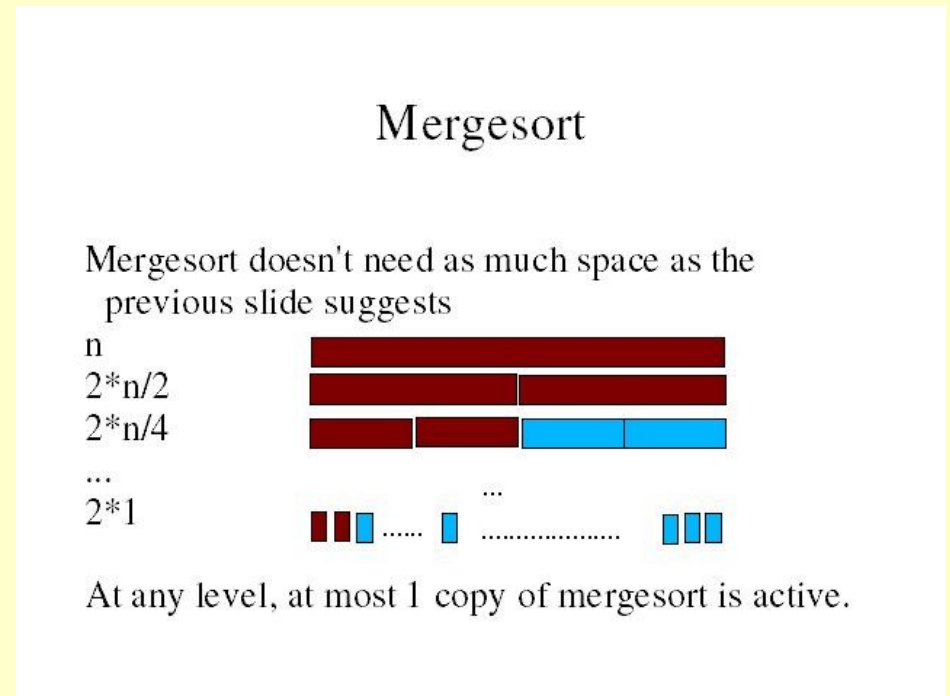
MergeSort (L[])

array R[0 .. N-mid-1] = A[1+mid .. N]

MergeSort (R[])

A[] = merge(L[], R[])

return



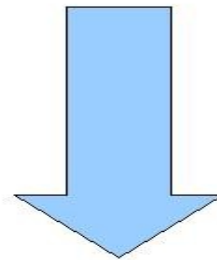
The speed of MergeSort

MergeSort can sort an array of size N in time
 $T(N) = O(N \cdot \log(N))$

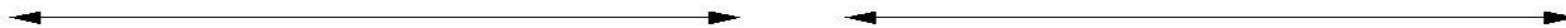
- $T(N) \leq C \cdot N \cdot \log(N)$ for some constant $C > 0$
- this is faster than the simple sorts (bubble sort, insertion sort, selection sort) which have time complexity $O(N^2)$
- MergeSort uses the “divide and conquer” strategy, which is naturally recursive
 - divide into smaller problems of comparable size, employ the same strategy to conquer each of those problems (i.e., divide them again ...)

Partitioning an array

(can you sort an array by doing this?)



Partition $A[1 \dots N]$



smaller than a_1

greater or equal to a_1

The idea behind QuickSort

- Partitioning an array splits it into two parts
- The two parts are not guaranteed to be of the same size (MergeSort, on the other hand, always produced two equal sized halves)
 - But, we might be lucky
- After splitting into two parts, sort each of those parts (using QuickSort again)
 - no merging is necessary, so this might even be faster than MergeSort

The QuickSort algorithm

```
QuickSort(array A[low .. high])  
  // sort a chunk of the array A  
  if (high > low) // more than one element?  
    int n = partition(A[low .. high])  
    QuickSort (A[low .. n-1])  
    QuickSort (A[n+1 .. high])  
  return
```

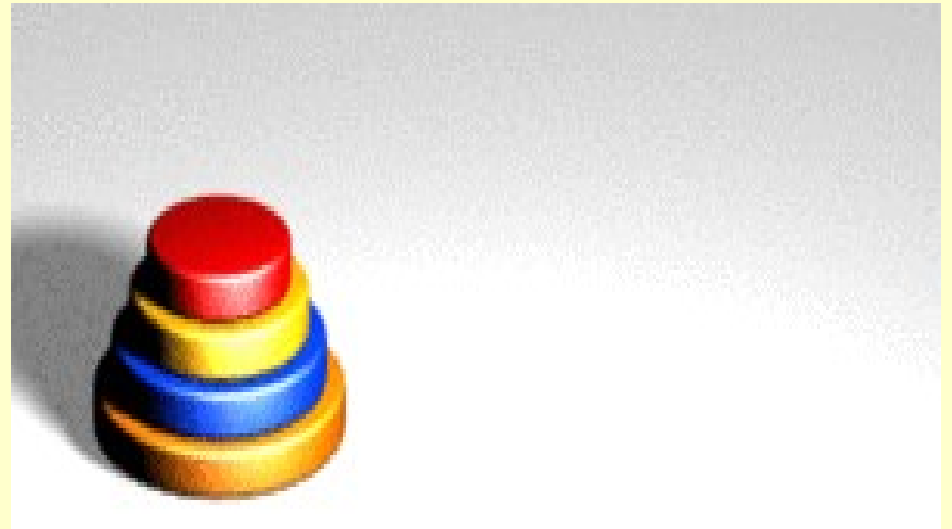
The Speed of QuickSort

- Worst Case
 - $T(N) = O(N^2)$
 - no better than bubble sort or insertion sort
 - this happens when the partitioning element always ends up at an endpoint of the array being partitioned
- Average Case
 - $T(N) = O(N \cdot \log(N))$, the same as MergeSort
 - most of the time, partitioning produces two parts of comparable size

Towers of Hanoi

(graphic from Wikipedia)

- You have a board with three spindles (let's call them left, middle, and right). There are a number of disks of different sizes on the left spindle, stacked so that each disk is smaller than the disk below it. The middle and right spindles are empty.
- Your task is to move the disks, one at a time without ever placing a larger disk on top of a smaller disk, so that the disks are all on the right spindle (and, of course, each disk is smaller than the disk below it).
- Not only do you want to transfer the disks to this new position, but you want to do it by making the smallest number of moves.
- It is known that it takes $2^N - 1$ moves to transfer the disks if you start with N disks on the left spindle.



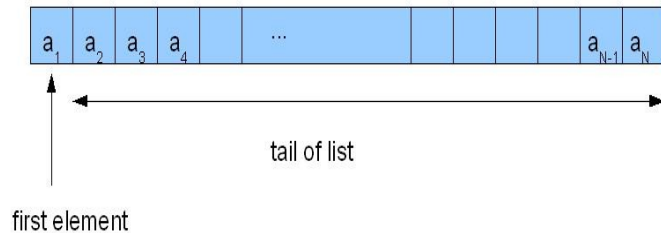
The (recursive) Solution to Towers of Hanoi

- To move all disks from the non-empty left spindle to the right spindle
 - Step 1: move all but the bottom disk from the left spindle to the middle spindle (*let's assume we know how to solve this smaller problem with $N-1$ disks*)
 - Step 2: move the remaining disk on the left spindle to the right spindle
 - Step 3: move all the disks from the middle spindle to the right spindle (the disk already on the right spindle can't get in the way since it is the largest of all, so pretend it isn't even there)

Towers of Hanoi: Recursive Solution

```
public class TowersOfHanoi {  
  
    public static void main(String[] args) {  
        int disks = Integer.parseInt(args[0]);  
        System.out.println("Solution for " + disks + " disks:");  
        solveTowers(disks, 'L', 'M', 'R');  
    }  
  
    public static void solveTowers (  
        int disks,  
        char initPos,  
        char middlePos,  
        char finalPos) {  
  
        if (disks > 0) {  
            solveTowers(disks - 1, initPos, finalPos, middlePos);  
            System.out.println("Move disk from "+initPos+" to "+finalPos);  
            solveTowers(disks - 1, middlePos, initPos, finalPos);  
        }  
    }  
}
```

Searching a List Recursively



```
boolean rsearch(list, target)
```

```
  if (list is empty)
```

```
    return false
```

```
  else if (first element == target)
```

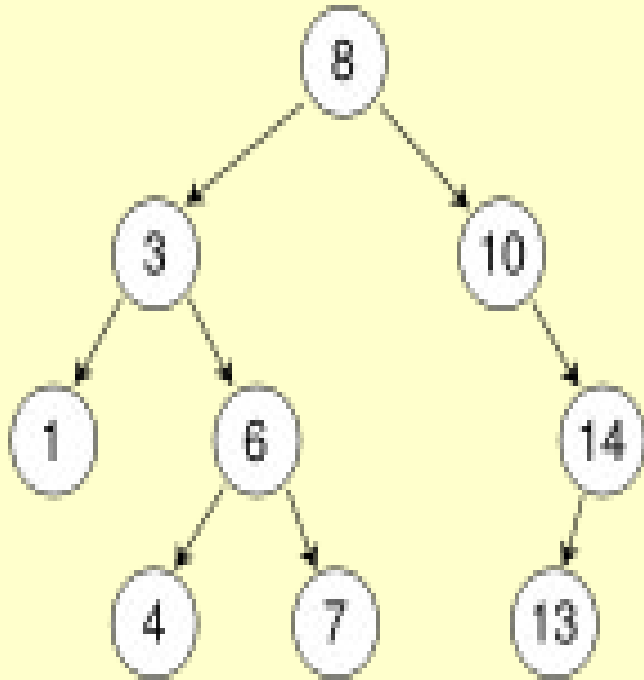
```
    return true
```

```
  else
```

```
    return rsearch(tail of list, target)
```

NOTE: It works, but later on we'll see why this isn't a good way to do it.

Binary Search Trees and Recursion



```
boolean rBSTsearch(root, tgt)
```

```
if (root is null)  
    return false
```

```
else if (tgt == root.value)  
    return true
```

```
else if (tgt < root.value)  
    return rBSTsearch(root.left, tgt)
```

```
else  
    return rBSTsearch(root.right, tgt)
```

NOTE: This logic looks pretty simple.
Could you do it as simply without
using recursion?

When is recursion a good idea?

- Recursion is usually a good idea if
 - if simplifies your program logic
 - if it's no simpler than using a loop, you should prefer the loop
 - you understand why and how it will work
 - if you don't know what you're trying to do, you will have trouble giving correct instructions to a machine
 - it isn't a bad idea (see later slides)

Low-hanging Fruit Revisited

- Factorials

- $n! = n \cdot (n-1)!$ and $0! = 1$
- should be calculated with a loop

- Sums of consecutive integers

- $$\sum_{i=1}^N i = N + \sum_{i=1}^{N-1} i$$

- a loop should be used for straightforward calculations like this, recursion will be slower and no simpler

When is recursion a bad idea?

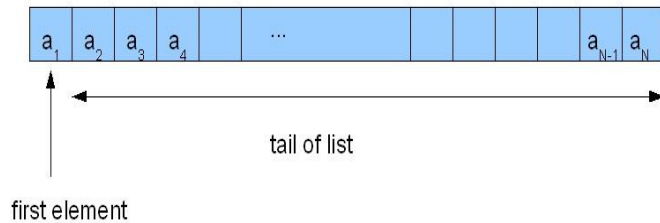
- Consider using recursion to compute Fibonacci numbers
 - $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, $F_1 = 1$
 - $$\begin{aligned} F_4 &= F_3 + F_2 \\ &= (F_2 + F_1) + (F_1 + F_0) \\ &= ((F_1 + F_0) + F_1) + (F_1 + F_0) \end{aligned}$$
- notice that F_2 is computed twice
- it gets worse for larger Fibonacci numbers
- You don't want to solve the same simple instance a whole bunch of times

When is recursion a bad idea?

- **Tail Recursion** should be replaced by a loop
 - tail recursion occurs when the last operation of the function is the only line where a recursive call is made
 - think about the “stream of instructions” being executed
 - you are going to repeat the entire body of the function, possibly with different parameter values
 - all you really need to do is modify some variable values and branch back to the first statement in the method,
 - of course, you'll also need to add the looping control statement to your code and determine a condition for terminating the loop
- You will save all the overhead of calling new instances of your method, passing parameters, returning values, etc.

Searching a List Recursively

(note the “tail recursion”)



boolean `rsearch`(list, target)

if (list is empty)

return false

else if (first element == target)

return true

else

return `rsearch`(tail of list, target)

How would you translate this into a loop? ... While the sublist to be searched is not empty and the first element is not the target value ...

Infinite Regress

(this could happen to you when you use recursion)

- An **infinite regress** in a series of propositions arises if the truth of proposition P_1 requires the support of proposition P_2 , and for any proposition in the series P_n , the truth of P_n requires the support of the truth of P_{n+1} .
- There would never be adequate support for P_1 , because the infinite sequence needed to provide such support could not be completed.

A Recursion CheckList

To be sure that a recursive method will work correctly, it is necessary to

- **identify a base case**, a simple version of the problem being solved, which can be solved without recursion
 - The recursive method must check for this base case and solve it without making any additional recursive calls.
- **reduce the problem size before making each recursive call**, thereby assuring that the base case will eventually be reached
 - Each recursive call must present the recursive subroutine with a simpler version of the problem. This assures the base case will eventually be reached, avoiding infinite regress.
- after a recursive call returns a solution to a simpler version of the problem, the calling routine must know how to **use the solution of the simpler instance(s) to solve its more complex instance of that same problem**
 - Presumably, the recursive call was made because the caller wanted the solution to that simpler problem instance. Why did it want that solution? How is it planning to use it?

If a recursive method implements these three notions correctly, there is an assurance that the initial call to the recursive method will eventually terminate and control will be returned to the routine that called it (as long as these subroutines don't crash for some other reason).

Does MergeSort do all our checklist items?

base case

MergeSort(array A[0 .. N])

if (length of A[] is greater than one)

mid = N/2

array L[0 .. mid] = A[0 .. mid]

MergeSort (L[])

array R[0 .. N-mid-1] = A[1+mid .. N]

MergeSort (R[])

A[] = merge(L[], R[])

return

simpler instance

simpler instance

arrays must be sorted before merging