Did you mean: recursion

Recursion - Wikipedia, the free encyclopedia
Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other, the nested ...

  Recursion (computer science)  Recursive definition
  Recursion in computer science is a method where the solution to a ...  A recursive definition (or inductive definition) in mathematical logic ...

CodingBat Java Recursion-1
codingbat.com/java/Recursion-1
Basic recursion problems. Recursion strategy: first test for one or two base cases that are so simple, the answer can be returned immediately. Otherwise, make a ...

Recursion - Learn You a Haskell for Great Good!
learnyouahaskell.com/recursion
We mention recursion briefly in the previous chapter. In this chapter, we'll take a closer look at recursion, why it's important to Haskell and how we can work out ...

Recursion
pages.cs.wisc.edu/.../6.RECURSION.htm... University of Wisconsin-Madison
The original call causes 2 to be output, and then a recursive call is made, creating a clone with \texttt{k == 1}. That clone executes line 1: the if condition is false; line 4: ...
RECURSION
administrivia…
- assignment 4 due on Thursday at midnight

- partners?

- midterm next Tuesday
  - exam review questions out later this week

- no office hours today
last time...
selection vs insertion

<table>
<thead>
<tr>
<th></th>
<th>WORST:</th>
<th>AVERAGE:</th>
<th>BEST:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td></td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
**selection vs insertion**

<table>
<thead>
<tr>
<th></th>
<th>Worst</th>
<th>Average</th>
<th>Best</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(N^2)$</td>
<td>$O(N^2)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

**Which one performs better in practice?**

A) selection

B) insertion
what we want...

-a sorting algorithm that has **subquadratic** complexity

-swapping adjacent items removes exactly 1 inversion

```
45  -3  9  76  11  -8  0
```

![SWAP REMOVES 1 INVERSION]

-what if we consider swapping nonadjacent pairs?

```
45  -3  9  76  11  -8  0
```

![SWAP REMOVES 7 INVERSION]

-removes inversions not involved with the swap
shellsort
the simplest subquadratic sorting algorithm
**shellsort**
insertion sort, with a twist

1) set the **gap size** to \( N/2 \)

2) consider the subarrays with elements at **gap size** from each other

3) do insertion sort on each of the subarrays

4) divide the **gap size** by 2

5) repeat steps 2 — 4 until the **gap size** is <1
shellsort
insertion sort, with a twist

1) set the gap size to \( N/2 \)

2) consider the subarrays with elements at gap size from each other

3) do insertion sort on each of the subarrays

4) divide the gap size by 2

5) repeat steps 2 — 4 until the is gap size is \(<1\)

WHAT DOES THIS LOOK LIKE?
HOW DO WE DESCRIBE INSERTION SORT WITH RESPECT TO SHELLSORT?
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr)
{
    for (gap = arr.length/2; gap > 0; gap /= 2)
    {
        for (i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for (j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr) {
    for (gap = arr.length/2; gap > 0; gap /= 2) {
        for (i = gap; i < arr.length; i++) {
            val = arr[i];
            for (j = i-gap; j >= 0 && arr[j] > val; j -= gap) {
                arr[j+gap] = arr[j];
            }
            arr[j+gap] = val;
        }
    }
}
void shellSort(int[] arr)
{
    for(gap = arr.length/2; gap > 0; gap /= 2)
    {
        for(i = gap; i < arr.length; i++)
        {
            val = arr[i];  // ITEM TO BE INSERTED
            for(j = i-gap; j >= 0 && arr[j] > val; j -= gap)
                arr[j+gap] = arr[j];
            arr[j+gap] = val;  // INSERT ITEM
        }
    }
}
today...
-what is recursion? and some examples…

-driver methods

-the overhead of recursion
re·cur·sion

[ri-kur-zhuh n]
noun

see recursion.
**recursion** is a problem solving technique in which the solution is defined in terms of a simpler (or smaller) version of the problem:

- break the problem into smaller parts
- solve the smaller problems
- combine the results

A recursive method calls itself.

Some functions are easiest to define recursively:

\[ \text{sum}(N) = \text{sum}(N-1) + N \]

There must be at least one *base case* that can be computed without recursion:

- any recursive call must make progress towards the base case!
a simple example

\[ \text{sum}(N) = \text{sum}(N-1) + N \]

```java
public static int sum(int n) {
    if(n == 1)
        return 1;
    return sum(n-1) + n;
}
```
a simple example

\[ \text{sum}(N) = \text{sum}(N-1) + N \]

```java
public static int sum(int n) {
    if(n == 1)
        return 1;
    return sum(n-1) + n;
}
```

FIX TO HANDLE ZERO OR NEGATIVE VALUES...
a simple example

\[ \text{sum}(N) = \text{sum}(N-1) + N \]

```java
public static int sum(int n) {
    if(n == 1)
        return 1;
    return sum(n-1) + n;
}
```

How can we solve the same problem without recursion?

Fix to handle zero or negative values. . .
a simple example

\[ \text{sum}(N) = \text{sum}(N-1) + N \]

```java
public static int sum(int n) {
    if(n == 1)
        return 1;
    return sum(n-1) + n;
}
```

How can we solve the same problem without recursion? Which is better, the recursive solution or the alternative?

Fix to handle zero or negative values...
exercise 1

- how to compute $N!$

$$N! = N \times N-1 \times N-2 \times \ldots \times 2 \times 1$$
exercise 1

- how to compute $N!$
  
  $N! = N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1$

- how would you compute this using a for-loop?
exercise 1

- how to compute $N!$
  
  $N! = N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1$

- how would you compute this using a for-loop?

- how would you compute this using recursion?

  - think about:
    
    - what is the base case?
    
    - what is recursive?
exercise 1

- how to compute \( N! \)
  \[ N! = N \times (N-1) \times (N-2) \times \ldots \times 2 \times 1 \]

- how would you compute this using a for-loop?

- how would you compute this using recursion? A) \( c \)  B) \( \log N \)  C) \( N \)  D) \( N \log N \)  E) \( N^2 \)  F) \( N^3 \)

- think about:
  - what is the base case?
  - what is recursive?

WHAT IS THE COMPLEXITY OF THE FOR-LOOP METHOD?
exercise 1

-how to compute $N!$

\[ N! = N \times N-1 \times N-2 \times \ldots \times 2 \times 1 \]

-how would you compute this using a for-loop?

-how would you compute this using recursion? A) $c$

-what is the base case?

-what is recursive?

WHAT IS THE COMPLEXITY OF THE RECURSIVE METHOD?

A) $c$
B) $\log N$
C) $N$
D) $N \log N$
E) $N^2$
F) $N^3$
exercise 2

public static int divide(int a, int b)
{
    ...
}

HINT: $9/2 = 1 + (7/2)$
exercise 2

- write a recursive method that computes \( \frac{A}{B} \)
  - do integer division
  - \(/\) operator not allowed, can only use -
  - don’t worry about negative input or divide-by-zero

```java
public static int divide(int a, int b) {
    ...
}

HINT: \( \frac{9}{2} = 1 + \frac{7}{2} \)
- recursion often seems like *magic*
  - use this to your advantage

- when writing a recursive method, just assume that the function you’re writing already works, so you can use it to help solve the problem

- once you’ve worked out the recursion, think about the base case, and you’re done
driver methods
divide and conquer

-divide and conquer is an important problem solving technique that makes use of recursion
  -**divide:** smaller problems are solved recursively (except for base cases!)
  -**conquer:** solutions to the subproblems form the solution to the original problem

-typically, an algorithm containing more than one recursive call is referred to as divide and conquer

-subproblems are usually disjoint (non-overlapping)
exercise 3
binary search (recursive)

-write a recursive method to perform a binary search
-assume an (ascending) sorted list
exercise 3
binary search (recursive)

- write a recursive method to perform a binary search
  - assume an (ascending) sorted list

-HINT
  - check if middle item is what we’re looking for
    - if so, return true
  - else, figure out if item is the left or right half
    - repeat on that half

-base case(s)???
-recursive methods often have unusual parameters
- at the top level, we just want:

    binarySearch(arr, item);

-but in reality, we have to call:

    binarySearch(arr, item, 0, arr.length-1);

-driver methods are wrappers for calling recursive methods
- driver makes the initial call to the recursive method, knowing what parameters to use
- is not recursive itself

    public static boolean binarySearch(arr, item){
        return binarySearchRecursive(
            arr, item, 0, arr.length-1);
    }
-another useful feature of driver methods is error checking (or, validity checks)

-do the error checking *only* in the driver method, instead of redundantly doing it every time in the recursion
-another useful feature of driver methods is error checking (or, validity checks)

-do the error checking *only* in the driver method, instead of redundantly doing it every time in the recursion

**WHAT IS SOMETHING TO CHECK FOR IN OUR BINARY SEARCH METHOD?**
- another useful feature of driver methods is error checking (or, validity checks)

- do the error checking \textit{only} in the driver method, instead of redundantly doing it every time in the recursion

\textbf{WHAT IS SOMETHING TO CHECK FOR IN OUR BINARY SEARCH METHOD?}

```java
public static boolean binarySearch(arr, item) {
    if (arr == null) // only check this once
        return false;
    return binarySearchRecursive(
            arr, item, 0, arr.length-1);
}
```
overhead of recursion
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the **call stack**

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!

<table>
<thead>
<tr>
<th>call stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>swap</td>
</tr>
<tr>
<td>sort</td>
</tr>
<tr>
<td>findAnagrams</td>
</tr>
<tr>
<td>main</td>
</tr>
</tbody>
</table>
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
method calls

- every time a method is invoked, a unique “frame” is created
  - contains local variables and state
  - put on the call stack

- when that method returns, execution resumes in the calling method

- this is how methods know where to return to!
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments

```
call stack
factorial(4)
main
```
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments
recursive calls

-create multiple frames of the same method
  -but each frame has different arguments
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments

```
call stack

main
factorial(4)
factorial(3)
factorial(2)
```

recursive calls

- create multiple frames of the same method
  - but each frame has different arguments
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments

factorial(4)
main
call stack
recursive calls

- create multiple frames of the same method
  - but each frame has different arguments
recursion, beware

-do not use recursion when a simple loop will do
  -growth rates may be the same, but...
  -...there is a lot of overhead involved in setting up the method frame
    -way more overhead than one iteration of a for-loop

-do not do redundant work in a recursive method
  -move validity checks to a driver method

-too many recursive calls will overflow the call stack
  -stack stores state from all preceding calls
recap
4 recursion rules

1. always have at least one case that can be solved without using recursion

2. any recursive call must progress toward a base case

3. always assume that the recursive call works, and use this assumption to design your algorithms

4. never duplicate work by solving the same instance of a problem in separate recursive calls
next time...
-reading
  - chapters 7 & 8.5 - 8.8 (recursion, mergesort, & quicksort)

-homework
  - assignment 4 due Thursday

-(short) midterm review on Thursday