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 ...

More results from wikipedia.org »

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  Learn You a Haskell for Great Good!
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.ht...  University of Wisconsin-Madison
The original call causes 2 to be output, and then a recursive call is made, creating a clone with k == 1. That clone executes line 1: if condition is false; line 4: ...
RECURSION
administrivia...
- assignment 4 due on Wednesday at midnight

- using the TA queue
  - for pair-programming assignments, only one partner can be in the queue at once

- TAs will be keeping help times short when the queue is busy
  - but, you should be better with debugging now
last time...
bubble sort
the (usually) most inefficient sorting algorithm
Compare each pair of adjacent items and swap them if necessary. Repeat.
bubble sort

1) for each item, compare it to its next neighbor and swap if necessary

2) repeat step 1 until sorted

what does this look like?
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
                {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
} 

1) same
2) different

what is the relationship of the best and worst case complexity?
void bubbleSort(int[] arr) {
    for(int i=0; i < arr.length-1; i++)
    {
        for(int j=0; j < arr.length-2; j++)
            if (arr[j] > arr[j+1])
                {
                    temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
}
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        boolean swapped = false;
        for(int j=0; j < arr.length-2; j++)
        {
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
                swapped = true;
            }
        }
        if (!swapped) return;
    }
}
void bubbleSort(int[] arr)
{
    for(int i=0; i < arr.length-1; i++)
    {
        boolean swapped = false;
        for(int j=0; j < arr.length-2; j++)
        {
            if (arr[j] > arr[j+1])
            {
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = arr[j];
                swapped = true;
            }
        }
        if (!swapped) return;
    }
    if (!swapped) return;
}

what is the best-case complexity?
1) O(1)
2) O(log N)
3) O(N)
4) O(N log N)
5) O(N^2)
6) O(N^3)
what we want...

-a sorting algorithm that has subquadratic complexity

-swapping adjacent items removes exactly 1 inversion

\[
\begin{array}{cccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

swap removes 1 inversion

-what if we consider swapping nonadjacent pairs?

\[
\begin{array}{cccccccc}
45 & -3 & 9 & 76 & 11 & -8 & 0 \\
\end{array}
\]

swap removes 7 inversions

-removes inversions not involved with the swap
shellsort
the simplest subquadratic sorting algorithm
Divide the array (smartly) into subarrays. Do insertion sort on the subarrays. Repeat.

* Take the first item in the unsorted portion of the array and **insert** it into the sorted portion of the array.
shellsort
insertion sort, with a twist

1) set the gap size to \( \frac{N}{2} \)

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 \(<1\)

what does this look like?
How can 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;
        }
    }
}

**what is the best-case complexity of shellsort?**
shellsort complexity

-worst case: $O(N^2)$ with Shell’s gaps, $O(N^{3/2})$ with better gaps

-average case: $O(N^{3/2})$ with Shell’s gaps, $O(N^{5/4})$ with better gaps

-proofs of these bounds are complicated
  -the $O(N^{5/4})$ bound is based on simulations only!

-insertion sort performs better the more sorted the array
  -remember, approaches $O(N)$ for a sorted array!
shell sort complexity

- still, $O(N^{5/4})$ is an encouraging bound for the average case

- for moderate $N$, this is better than $O(N \log N)$ algorithms

- around $N=100K$, $O(N \log N)$ wins

- best sorting algorithms are $O(N \log N)$
  - $\log N$ suggests repeated dividing by 2
  - “divide and conquer”

what algorithm do we know of that is $\log N$?
what does this imply about the “conquer” step?
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

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

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

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

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 recursive part?
-what is the base case?
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 recursive part?
    - what is the base case?

what is the complexity of the for-loop method?

A) $c$
B) $\log N$
C) $N$
D) $N \log N$
E) $N^2$
F) $N^3$
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 recursive part?
    - what is the base case?

what is the complexity of the recursive method?

- A) \( c \)
- B) \( \log N \)
- C) \( N \)
- D) \( N \log N \)
- E) \( N^2 \)
- F) \( N^3 \)
public static int divide(int a, int b) {
    ...
}

hint: \( \frac{9}{2} = 1 + \frac{7}{2} \)
exercise 2

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

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

hint: 9/2 = 1 + (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 index
  -else, figure out if item is the left or right half
    -repeat on that half
  -return -1 if not found in the array

-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 int 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 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?

```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!
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

---

factorial(4)

main

(call stack)
recursive calls

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

```javascript
main
  factorial(4)
  factorial(3)
```
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

```
call stack

factorial(4)
factorial(3)
factorial(2)
factorial(1)
main
```
recursive calls

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

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

factorial(4)
factorial(3)
factorial(2)
main
recursive calls

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

```
call stack

main
factorial(4)
factorial(3)
```
recursive calls

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

```
main
factorial(4)
```
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 Wednesday