Recursion

Table of contents

Introduction

Recursion is a way of **thinking about** problems, and a **method for solving** problems. The basic idea behind recursion is the following: it's a method that solves a problem by solving *smaller* (in size) versions of the same problem by breaking it down in smaller subproblems. Recursion is very closely related to **mathematical induction**.

We'll start with recursive definitions, which will lay the groundwork for recursive programming. We'll then look at a few prototypical examples of using recursion to solve problems. We'll finish with looking at problems and issues with recursion.

Back to Table of contents

Recursive definitions

We'll start **thinking recursively** with a few recursive definitions:

1. A list of numbers
2. Factorial
3. Fibonacci numbers

List of numbers

Consider the (*non-empty*) list of numbers `‹24, 88, 30, 37›`. How would we *define* such as list? The first one that comes to mind, at least at this point into this semester, would probably be the following:

```
A list is one or more numbers, separated by commas.
```

The **recursive** way of thinking would produce the following definition:

```
A list is either a: number
             or, a: number comma list
```

That is, a *list* is either a single number, or a number followed by a comma followed by a *list*. Two things to note here:

1. We're using *list* to define a *list*, which is to say that the concept of a *list* is used to define itself!
2. There is a **recursive** (the 2nd) part in the definition, and a **non-recursive** (the 1st) part in the definition, which is called the **base case** or the **stopping condition**. Without the base case, the recursion would never stop, and we would end up with *infinite recursion*!

With this **recursive definition** of a *list*, we can now ask whether ‹ `24` › is a *list* or not. It matches the 1st condition, which says that it's a single number, so it is indeed a *list*. What about ‹ `24,88` ›? Here we have a number `24` followed by a `comma` followed ‹ `88` ›; since we now know that ‹ `88` › is indeed a *list*, we can rephrase it as such: we have a number `24` followed by a `comma` followed by a *list* ‹ `88` ›, so it matches the 2nd definition, and so it is indeed a list. If we try a bit longer list of numbers ‹ `24,88,30,37` ›, we see it does indeed match our definition. Note how the recursive part of the *list* definition is used several times, terminating with the non-recursive part (the *base condition*, which stops the recursion).

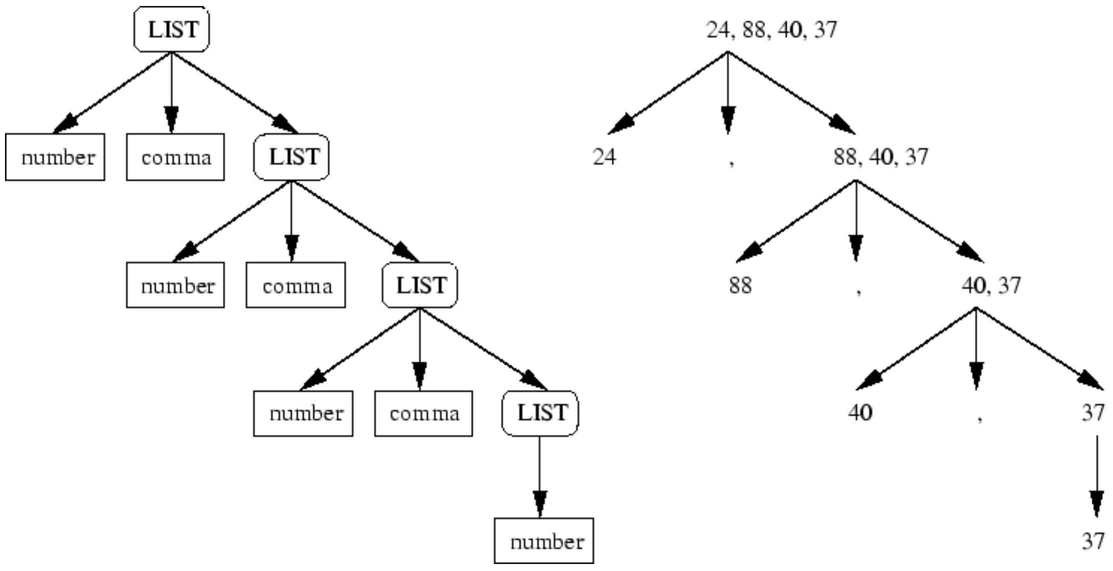
```
number  comma      list
  24      ,     88, 30, 37

             number  comma    list
               88      ,     30, 37

                          number  comma   list
                            30      ,      37

                                         number
                                          37
```

The following shows a graphical view of tracing this recursive definition of a *list* using the sequence ‹ `24,88,30,37` ›. This is often referred to as the **recursion tree** diagram.



Factorial

The *factorial* of a non-negative integer `n` is defined to be the product of all positive integers less than or equal to `n`. For example, `5! = 5 × 4 × 3 × 2 × 1 = 120`. See Wikipedia's entry on Factorial for more info.

```
1! = 1
2! = 2 × 1
3! = 3 × 2 × 1
4! = 4 × 3 × 2 × 1
5! = 5 × 4 × 3 × 2 × 1
...
and so on.
```

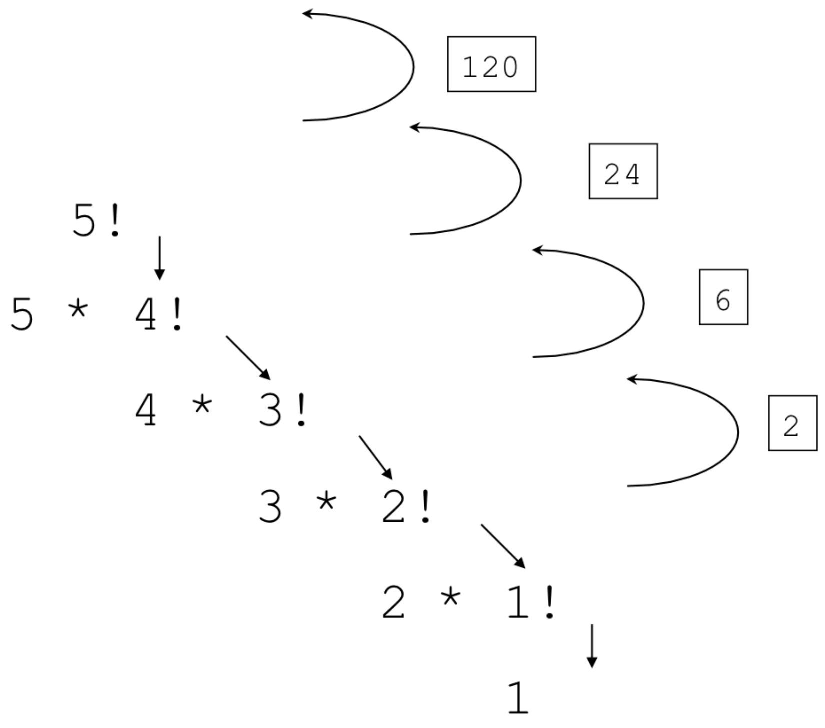
We can easily evaluate `n!` for any valid value of `n` by multiplying the values iteratively. However, there is a much more interesting recursive definition quite easily seen from the *factorial* expressions: `5!` is nothing other than `5 × 4!`. If we know `4!`, we can trivially compute `5!`. `4!` on the other hand is `4 × 3!`, and so on until we have `n! = n × (n - 1)!`, with `1! = 1` as the base case. The mathematicians have however added the special case of `0! = 1` to make easier (yes, it does, believe it or not). For the purposes of this discussion, we'll use both `0! = 1` and `1! = 1` as the two base cases (we can always add `2! = 2` as another base, as long as we keep the necessary one — `0!`).

```
     -
     | 0              if n = 1
     | 1              if n = 1
n! =|
     | n × (n - 1)!   if n > 1
     -
```

Now we can expand `5!` recursively, stopping at the base condition.

```
5! = 4 × 4!
         4 × 3!
             3 × 2!
                 2 × 1!
                     1
```

The recursion tree for `5!` shows the values as well.



Fibonacci numbers

The Fibonacci numbers are ‹ `0, 1, 1, 2, 3, 5, 8, 13, ...` › (some define it without the leading `0`, which is ok too). If we pay closer attention, we see that each number, except for the first two, is nothing but the sum of the previous two numbers. We can easily compute this iteratively, but let's stick with the recursive method. We already have the **recursive** part of the **recursive definition**, so all we need is the **non-recursive** part or the **base case**. The mathematicians have decided that the first two numbers in the sequence are `0` and `1`, which give us the base cases (notice the two base cases). See Wikipedia's entry on Fibonacci number for more info.

Now we can write the recursive definition for any Fibonacci number `n ≥ 0`.

```
          -
          | 0                           if n = 0
          | 1                           if n = 1
fib(n) =  |
          | fib(n-1) + fib(n-2)         n ≥ 2
          -
```

We can now compute `fib(5)` using this definition.

```
                       fib(5)
                        +
                   /           \
                  /             \
             fib(4)             fib(3)
               +                  +
          /         \          /       \
         /           \        /         \
     fib(3)       fib(2)   fib(2)     fib(1)
       +            +        +
      /     \         /    \       /   \
     /       \       /      \     /     \
    f        f     f        f    f      f
    i        i     i        i    i      i
    b        b     b        b    b      b
   (2)      (1)   (1)      (0)  (1)    (0)
    +
   / \
  ....
```

Before moving on, you should note how many times we're computing Fibonacci of 3 (`fib(3)` above), and Fibonacci of 2 (`fib(2)` above) and so on. This **redundancy in computation** leads to gross inefficiency, but something we can easily address to **Memoization**, which is the next topic we study.

Back to Table of contents

Recursive programming

A **recursive function** is one that calls itself, since it needs to solve the same problem, but on a smaller-sized input. In essence, a recursive function is a function that is defined in terms of itself.

Let's start with computing the factorial. We already have the recursive definition, so the question is how do we convert that a recursive method in Java that actually computes a factorial of a given non-negative integer. Here's the recursive definition again, along with the Java code. This is an example of *functional recursion*.

```
     -                                    static int fact(int n) {
     | 1              if n = 0                if (n == 0 || n == 1)
     | 1              if n = 1                    return 1;
n! =|                                         else
     | n × (n - 1)!   if n > 1                    return n * fact(n - 1);
     -                                    }
```

The first thing to note is how similar the code in `recursiveSum` is to the recursive definition. Once you have formulated the recursive definition, the rest is usually quite trivial. This is indeed one of the greatest advantages of recursive programming.

Let's now look an example of *structural recursion*, one that uses a **recursive data structure**: we want to compute the sum of the list of numbers `a = ‹3 8 2 1 13 4›`. The "easiest" (at least in Java, but certainly not in say Lisp or Scheme) way to solve it **iteratively**, as shown below.

```
static int iterativeSum(LinkedList list) {
    int sum = 0;
    Node n = list.head;
    while (n != null) {
        sum = sum + n.item;
        n = n.next;
    }
    return sum;
}
```

Now since we already know that a list of numbers has a recursive definition, meaning that it's a recursive data structure, any problem defined on it must also have a simple recursive solution. Thinking recursively, we note the following:

1. The sum of the numbers in a list is nothing but the sum of the first number **plus** the sum of the **rest of the list**. The problem of summing the **rest of the array** is the same problem as the original, except that it's smaller by one element! Ah, recursion at play.
2. The shortest list has a single number, which has the sum equal the number itself. Now we have a base case as well. Note that if we allowed our list of numbers to be empty (we said non-empty when defining it), then the base case will need to be adjusted as well: the sum of an empty list is simply `0`.

Note the key difference between the iterative and recursive approaches: **for each number in the list** vs **the rest of the list**.

Now we can write the **recursive definition** of this problem, and using that, write the recursive method:

```
         -
         | k                k is the only element
sum =    |
         | k + sum(n.next)  otherwise
         -


static int recursiveSum(LinkedList list) {
    if (list.size() == 1)
        return list.head.item;
    else {
        return list.head.item + recursiveSum(list.next);
    }
}
```

Again, note how similar the code in `recursiveSum` is to the recursive definition.

Our familiar *linked list* is a **recursive data structure** since it's defined in terms of itself. The following is a recursive definition of a `linked list`:

```
A linked list is either empty, or it is a node followed by another
linked list.
```

We've of course seen this already when discussing recursive definition — this is almost exactly the same as our example of list of numbers, with only base case differing — we allow a linked list to be empty.

Back to Table of contents

Examples

We look at the following examples:

1. Length of a String
2. Length of a linked list
3. Sequential search in a sequence

Length of a String

A character string is a recursive structure: a string is either empty, or a character followed by the rest of the string. This implies that recursion is a *natural* way to solve string related problems. Let's take the case of computing the length of a string; the length of "abc" is 1 longer than the length of the rest of the string, which is "bc"; the length of "bc" is 1 longer than the length of "c"; and the length of "c" is 1 longer than the length of "", which is the empty string (and hence has a length of 0). We now have our recursive case and a base case! An example of what is known as *structural recursion*.

Since we're using Java's String class to represent a character string, The length of a Java String object has the following recursive formulation: the length is 1 **plus** the length of the **rest of the String**. In Java, the `String` provides a `substr` method to extract the rest of the String: `s.substring(1)` produces the substring from index 1 onwards. The recursion stops when the String is empty, which gives us the base case.

```
          -
          | 0              if string s is empty
len(s) =  |
          | 1 + len(rest)  otherwise
          -


static int strLength(String s) {
    if (s.equals(""))
        return 0;
    else
        return 1 + strLength(s.substring(1));
}
```

Length of a linked list

A linked list is also a recursive structure: a list is either empty, or a node followed by the rest of the list. We can also compute the length of a list recursively as follows: the the length of a list is 1 longer than the rest of the list! The empty list has a length of 0, which is our base case.

```
          -
          | 0              if l is an empty list
len(l) =  |
          | 1 + len(rest)  otherwise
          -


static int listLength(List l) {
    if (l == null)
```

```
            return 0;
        else
            return 1 + listLength(l.next);
    }
```

Sequential search in a sequence

How would you find something in a list? Well, look at the first node and check if the `key` is in that node. If so, done. Otherwise, check the **rest of the list** for the given `key`. If you search an empty list for any `key`, the answer is false, so that's our base case. This is almost exactly the same, at least in form, as finding the length of a linked list, and also an example of *structural recursion*.

```
                  -
                  | false              if list l is empty
                  | true               if l.item = k
contains(l,k) =   |
                  | contains(l.next,k) otherwise
                  -


static boolean contains(List l, int k) {
    if (l == null)
        return false;
    else if (l.item == k)
        return true;
    else
        return contains(l.next, k);
}
```

What if the sequence is an array? How do we deal with the **rest of the array** part then? We can handle it in two ways:

1. We can create a new array that contains a copy of the rest of the array, and pass that instead to the next level of recursion. This is of course very inefficient in Java, and to be avoided at all costs. In programming languages which provide efficient **array slicing**, this would be the way to go.
2. We can maintain a *left* index, along with a reference to the array, that is used to indicate where the beginning of the array is. Initially, `left = 0`, meaning that the array begins at the expected index of 0. Eventually, a value of `l == a.length - 1` means that the rest of the array is simply the last element, and then `l == a.length` means that it's an 0-sized array. For Java, this is by far the **preferred** method.

```
                    -
                    | false                     if l ≥ a.length
                    | true                      if a[l] = k
contains(a,l,k) =   |
                    | return contains(a,l+1,k) otherwise
                    -


static boolean contains(int[] a, int l, int k) {
    if (l ≥ a.length)
        return false;
    else if (a[l] == k)
```

```
            return true;
        else
            return contains(a, l + 1, k);
    }
```

We start the search with **`contains(a, 0, key)`**, and then at each step, the rest of the array is given by advancing the left boundary `l`.

Instead of just a **yes/no** answer, what if we wanted the **position** of the key in the array? We can simply return **l** instead of **true** as the position if found, or use a sentinel **-1** instead of **false** if not.

Binary search in a sorted array

Given the abysmal performance of sequential search, we obviously want to use binary search whenever possible. Of course, the pre-conditions must be met first:

1. The sequence must support random access (an array that is)
2. The data must be sorted

Now we can formulate a recursive version of binary search of a `key` in an array `data` between the positions `l` and `r` (inclusive):

```
if the array is empty (if l > r that is):
  return false
else:
  Find the position of the middle element: mid = (l + r)/2
  If key == data[mid], then return true
  If key > data[mid], the search the right half data[mid+1..r]
  If key < data[mid], the search the left half data[l..mid-1]
end if
```

Now we can write the recursive definition, and translate that to Java.

```
                     -
                     | false                  if l > r
                     | true                   if k = a[mid]
contains(a,l,r,k) = |
                     | contains(a,mid+1,r,k) if k > a[mid]
                     | contains(a,l,mid-1,k) if k < a[mid]
                     -


static boolean contains(int[] a, int l, int r, int k) {
    if (l > r)
        return false;
    else {
        int mid = (l + r)/2;
        if (k == a[mid])
            return true;
        else if (k > a[mid])
            return contains(a, mid + 1, r, k);
        else
            return contains(a, l, mid - 1, k);
    }
}
```

We start the search with **`contains(a, 0, a.length - 1, key)`**, and then at each step, the rest of the array is given by one half of the array – left or right, depending on the comparison of the key with the middle element.

Instead of just a **yes/no** answer, what if we wanted the **position** of the key in the array? We can simply return **mid** as the position instead of **true** if found, or use a sentinel **-1** instead of **false** if not.

Finding the maximum in a sequence (linear version)

Given a *sequence* of keys, our task is to find the **maximum key** in the sequence. This is of course trivially done iteratively (for a non-empty sequence): take the 1st one as maximum, and then iterate from the 2nd to the end, exchanging the current with the maximum if the current is larger than the maximum.

Formulating this recursively: the maximum key in a sequence is the larger of the following two:

1. the 1st key in the sequence
2. the **maximum** key in the **rest** of the sequence

Once we have (recursively) computed the maximum key in the rest of the sequence, we just have compare the 1st key with that, and we have our answer! The base is also trivial (for a non-empty sequence): the maximum key in a single-element sequence is the element itself.

Since the **rest of the sequence** does not need random access, we can easily do this for a linked list or an array. Let's write it for a list first.

```
                -
                | list.item                          if list.next = null
findMax(list) = |
                | max(list.item, findMax(list.next)) otherwise
                 -

static int findMax(Node l) {
    if (l.next == null)
        return l.item;
    else {
        int maxRest = findMax(l.next);
        return Math.max(l.item, maxRest);
    }
}
```

We find the maximum with **`findMax(head)`** (where `head` is the reference to the first node of the list), and then at each step, the rest of the array is given by advancing the head reference `head`.

What if the sequence is an array? Well, then we use the same technique we've used before — use a left (and optionally right) boundary to *window* into the array.

```
                 -
                 | a[l]                          if l =  r
```

```
findMax(a,l,r) = |
                 | max(a[l],findMax(a,l+1,r)) otherwise
                 -


static int findMax(int[] a, int l, int r) {
    if (l = r)
        return a[l];
    else {
        int maxRest = findMax(a, l + 1, r);
        return Math.max(a[l], maxRest);
    }
}
```

We find the maximum with **`findMax(a, 0, key)`**, and then at each step, the rest of the array is given by advancing the left boundary `l`.

Finding the maximum in an array (binary version)

If our sequence is an array, we can also find the maximum by formulating the following recursive definition: the maximum key in an array is the larger of the following two:

1. the maximum key in the **left half** of the array
2. the maximum key in the **right half** of the array

This will not produce an efficient version for linked lists, for the same reason binary search is not efficient for linked lists.

```
                 -
                 | a[l]                    if l =  r
findMax(a,l,r) = |
                 | max(findMax(a,l,mid),
                 |     findMax(a,mid+1,r)) otherwise
                 -


static int findMax(int[] a, int l, int r) {
    if (l = r)
        return a[l];
    else {
        int mid = (l + r)/2;
        int maxLeftHalf = findMax(a, l, mid);
        int maxRightHalf = findMax(a, mid + 1, r);
        return Math.max(maxLeftHalf, maxRightHalf);
    }
}
```

We find the maximum with **`findMax(a, 0, key)`**, and then at each step, the array is divided into two halves. If you draw the recursion tree, you will see why I refer to it as the *binary version*.

Selection sort

How about sorting a sequence recursively? We know from our iterative discussion of selection sort that it does not require random access, so we'll look at recursive versions for both linked lists and arrays.

The basic idea behind selection sort is the following: put the 1st minimum in the 1st position, the 2nd minimum in the 2nd position, the 3rd minimum in the 3rd position, and so on until each key is placed in its position according to its *rank*. To come up with a recursive formulation, the following observation is the key:

> Once the 1st minimum in the 1st position, it will never change its position. Now all we have to do is to sort the **rest of the sequence** (from 2nd position onwards), and we'll have a sorted sequence.

Now we can write the recursive definition for a linked list, and translate it to Java.

```
                     -
                     | done                if list = null or list.next = null
selectSort(list) =   |
                     | exchange the minimum key with list.item,
                     |    and sort the rest of the list
                     |    with selectSort(list.next)
                     -

static void selectSort(Node l) {
    if (l == null || l.next == null)
        return;
    else {
        Node minNode = findMinNode(l);
        swap(l, minNode);
        selectSort(l.next);
    }
}
```

We sort a list headed by `head` by calling `selectSort(head)`. Note that we're not find the minimum key, but rather the node that **contains** the minimum key since we need to exchange the left key with the minimum one. We can write that iterative of course, but a recursive one is simply more fun. This is of course almost identical to finding the maximum in a sequence, with two differences: we're find the minimum, and we're return the node that contains the minimum, not the actual minimum key.

```
                      -
                      | list.item                   if list.next = null
findMinNode(list) =   |
                      | get the node that contains the minimum in
                      |   the rest with findMinNode(list.next)), and
                      |   return either list.item or the other one
                      |   depending on which one has the smaller key
                      -

static Node findMinNode(Node l) {
    if (l.next == null)
        return l;
    else {
        Node minNode = l;
        Node minNodeRest = findMinNode(l.next);
        if (minNodeRest.item < l.item)
            minNode = minNodeRest;
        return minNode;
    }
}
```

If the sequence is an array, then we have to use the left (and optionally right) boundary to window into the array.

```
                      -
                      | done                    if l >= r
selectSort(a,l,r) =  |
                      | exchange the minimum key with a[l],
                      |    and sort the rest of the array
                      |    with selectSort(a,l+1,r)
                      -

static void selectSort(int[] a, int l, int r) {
    if (l >= r)
        return;
    else {
        Node minIndex = findMinIndex(a, l, r);
        swap(a, l, minIndex);
        selectSort(a, l + 1, r);
    }
}
```

We sort an array `a` by calling `selectSort(a, 0, a.length - 1)`. And we can write `findMinIndex` recursively of course, which is again almost exactly the same as either finding the maximum in a sequence, or finding the maximum in an array — we'll use the binary method just for illustration.

```
                        -
                        | a[l]                        if l = r
findMinIndex(a,l,r)  = |
                        | get the index of minimum in the rest with
                        |   findMinIndex(a,l+1,r)), and return either
                        |   l or the other one depending on which one
                        |   has the smaller key
                        -

static Node findMinIndex(int[] a, int l, int r) {
    if (l == r)
        return l;
    else {
        int mid = (l + r)/2;
        Node minIndexLeft = findMinIndex(a, l, mid);
        Node minIndexRight = findMinIndex(a, mid + 1, r);
        int minIndex = minIndexLeft;
        if (a[minIndexRight] < a[minIndexLeft])
            minIndex = minIndexRight;

        return minIndex;
    }
}
```

Insertion sort

Insertion works by inserting each new key in a sorted array so that it is placed in its rightful position, which now extends the sorted array by the new key. In the beginning, there is a single key in the array, which by definition is sorted. Then the second key arrives, which is then inserted into the already sorted array (of one key at this point), and now the sorted array has two keys. Then the third key arrives, which is inserted into the already sorted array, creating a sorted array of 3 keys. And so on. Iteratively, it's a fairly simple operation. The question is how can we formulate this recursively. The following observation is the key to this recursive formulation:

```
Given an array of n keys, sort the first n-1 keys and then
insert the nth key in the sorted array such that all
n keys are now sorted.
```

Note how the recursive part comes first, and then the n^{th} key is inserted (iteratively) into the sorted array. Unlike recursive selection sort, we're going from **right to left** in the recursive version of insertion sort.

Note that we don't need random access, but need to be able to iterate in both directions (reverse direction to insert the new key in the sorted partition). So, if we're sorting a linked list using the insertion sort algorithm, the list must be doubly-linked.

If sorting an array, we can formulate the solution as follows:

```
                      -
                      | done                if l >= r
insertSort(a,l,r) =   |
                      | recursively sort the first n-1 keys using
                      |    insertSort(a,l,r-1), and then insert the
                      |    nth key (index r in
                      |    case) such that the result is sorted.
                      -

static void insertSort(int[] a, int l, int r) {
    if (l >= r)
        return;
    else {
        insertSort(a, l, r - 1);

        // Now insert the r'th key by moving leftwards
        Object key = a[r];      // save to avoid being overwritten
        int j = r - 1;

        // Shift left, making room for the key
        while (j >= 0 && key < data[j]) {
            data[j + 1] = data[j];
            j--;
        }
        // Now insert the key in the right position
        data[j + 1] = key;
    }
}
```

[illegible]

We sort an array `a` by calling `insertSort(a, 0, a.length - 1)`.

Sum of integers 1 through N

Computing the sum of the 1st N integers has a trivial iterative solution. It is however instructive to see the recursive formulation. This is an example of *functional recursion*.

The sum of the 1st `N` integers is `N` **plus** the sum of the 1st `N-1` integers. The base case is when we sum only the first integer, namely when `N = 1`.

```
        -                                 static int sumN(int n) {
        | 1              n = 1                if (n == 1)
  sum = |                                         return 1;
        | n + sum(n - 1) n > 1                else
        -                                         return n + sumN(n - 1);
                                          }
```

Exponentiation – a^n

This is another example of *functional recursion*. To compute a^n, we can iteratively multiply `a` `n` times, and that's that. Thinking recursively, $a^n = a \times a^{n-1}$, and $a^{n-1} = a \times a^{n-2}$, and so on. The recursion stops when the exponent `n = 0`, since by definition $a^0 = 1$.

```
       -                               static int exp(int a, int n) {
       | 1           n = 0                 if (n == 0)
  a^n = |                                       return 1;
       | a × n^(n-1)  n > 0                else
       -                                        return a * exp(a, n - 1);
                                       }
```

As it turns out, there is actually a much more efficient recursive formulation for the exponentiation of a number. We start by noting that $2^8 = 2^4 \times 2^4$, and that $2^7 = 2^3 \times 2^3 \times 2$. We can generalize that with the following recursive definition, and its implementation.

```
       -
       | 1                            n = 0
  a^n = |
       | a^(n/2) × a^(n/2)            n is even
       | a^((n-1)/2) × a^((n-1)/2) × a  n is odd
       -


  static int exp(int a, int n) {
      if (n == 0)
          return 1;
      else if (n % 2 == 0)
          return exp(a, n/2) * exp(a, n/2);
      else
          return exp(a, (n - 1)/2) * exp(a, (n - 1)/2) * a;
  }
```

[illegible]

Ok, but why would we care about this formulation over the more familiar one? If we solve for the running time, both solutions take the same time, so what is the benefit of this approach? Notice how we're computing the following expressions **twice**:

1. `exp(a, n/2)`
2. `exp(a, (n - 1)/2)`

Why not compute it once, and then use the **result** twice (or as many times as needed)? We can, and as we'll find out, that'll give us a huge boost when we compute the running time of this algorithm. Here's the modified version.

```
static int exp(int a, int n) {
    if (n == 0)
        return 1;
    else if (n % 2 == 0) {
        int tmp = exp(a, n/2);
        return tmp * tmp;
    } else {
        int tmp = exp(a, (n - 1)/2);
        return tmp * tmp * a;
    }
}
```

All we're doing is removing the **redundancy** in computations by saving the **intermediate** results in temporary variables. This is a simple case of a technique known as **Memoization**, which is the next topic we study. Remember that we've already seen such redundancy in recursive computation — when computing the Fibonacci numbers.

Back to Table of contents

Issues/problems to watch out for

Inefficient recursion

The recursive solution for Fibonacci numbers outlined in these notes shows massive **redundancy**, leading to very inefficient computation. The 1st recursive solution for exponentiation also shows how redundancy shows up in recursive programs. There are ways to avoid computing the same value more than once by *caching* the intermediate results, either using **Memoization** (a *top-down technique* — see next lecture), or **Dynamic Programming** (a *bottom-up technique* — survive this semester to enjoy it in the next one).

Space for activation frames

Each recursive method call requires that it's **activation record** be put on the system *call stack*. As the depth of recursion gets larger and larger, it puts pressure on the system stack, and the stack may potentially run out of space.

Infinite recursion

Ever forgot a base case? Or miss one of the base cases? You end up with infinite recursion, since there nothing stopping your recursion! Whenever you see a **Stack Overflow** error, check your recursion!

Back to Table of contents

Discussion

Back to Table of contents