Software Development (CS2500)

Lecture 27: **Recursion**

M.R.C. van Dongen

December 6, 2010

Contents

1 Recursion

This section studies *recursion*, one of the most important concepts in computer science.

The following is what Merriam Webster's on-line dictionary says about recursion.

Function: noun

Etymology: Late Latin recursion-, recursio, from recurrere

Date: 1616

1. return
2. the determination of a succession of elements (as numbers or functions) by operation on one or more preceding elements according to a rule or formula involving a finite number of steps
3. a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first

Many concepts in computer science and mathematics are defined or computed *recursively*, i.e. using *recursion*. The idea is to define complicated concept in terms of itself.

1.1 Definition

The following are the main components of a recursive method:

***Base Case*:** Simple computation. We know how to do it without the need to call the method itself.

***Recursive Computation*:** Complicated computation involving: simple computations and one or several computation(s) of "lower order."

1.2 Examples

A typical example of a recursive algorithm is dictionary search. Here we're looking up the meaning of a word in a dictionary. For simplicity we shall assume that the dictionary lists all possible words and has one word per page.

To *search* the word given n pages do the following:

- If there's only one page ($n = 1$): We've found the word.
- Otherwise ($n > 1$):
  - Find the page in the "middle."
  - Read the word on the middle page.
  - If that word is our word: We've found the word.
  - If our word is smaller: *search* to the left.

– Otherwise: *search* to the right.

Figure 1 depicts a picture of a Dutch add for a cocoa brand called Droste. It displays a nurse carrying a serving tray with a cup of hot chocolate and a box of Droste cocoa depicting the same image. The image on the box is a smaller version of the whole image: recursion.

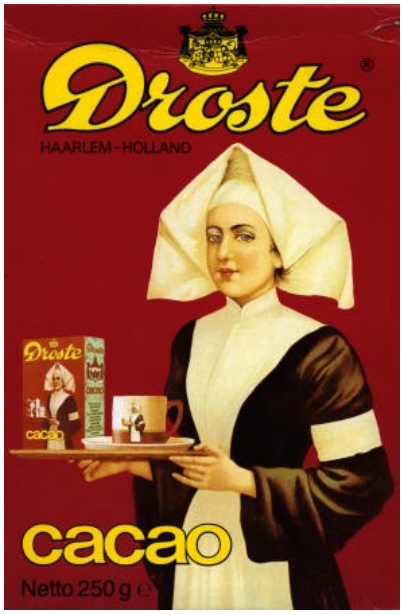

Figure 1: A recursive cocoa add.

Figure 2 shows that an old Tayto bag which uses a similar idea.

1.3 Potential Problems

Recursive computations involve themselves. If we're not careful we may get an infinite chain of computations. For example, we may be computing what's on Box 1 with Box 2 on it, which involves computing what's on Box 2 with Box 3 on it, which involves computing what's on Box 3 with Box 4 on it, which involves .... Each recursive computation should eventually terminate. We need to force some condition which guarantees that each recursive computation eventually reaches a base case condition.

The following helps us guarantee termination:

- Each computation should have a *size* which should be a non-negative integer.
- The size should depend on one or several method parameters.
- Each base-case computation corresponds to a small fixed size (usually 0 or 1). Different base-cases may correspond to different sizes.

Figure 2: A recursive Tayto add.

- Each recursive sub-computation which is part of a computation of size n should have a size which is *less* than n.
- If these conditions are met then termination is guaranteed and it can be proved using natural induction.

Again consider the dictionary search algorithm. It makes sense to define the size of the computation that looks up our word as the number of pages that are left.

The following argument proves that the algorithm terminates. Let's call the top computation C_0. Let C_1 be the recursive computation of C_0, let C_2 be the recursive computation of C_1, and so on. Finally, let S_i be the size of C_i. By nature of the algorithm a recursive search C_{i+1} has fewer pages than the number of pages of C_i. Therefore, $S_i > S_{i+1}$ for any computation C_i that has a recursive sub-computation. For sake of the argument, let's assume there is an infinite chain of computations C_0, C_1, C_2, ... Then we have an infinite chain of *integers* $S_0 > S_1 > S_2 > \cdots$. But this is impossible since $S_i \geq 0$, for all i. Therefore our assumption that there is an infinite chain of computations is false.

Note that it is crucial that the sizes are integers. For example, a chain of the following form does exist:

$$1, 1/2, 1/4, 1/8, \ldots .$$

The next few sections will study some more specific examples of recursion.

2 Factorial Computation

This section studies the application of recursion to the computation of factorials. Before we start let's recall the definition of factorials.

Let n be a positive integer. The *factorial* of n, denoted $n!$, is defined as follows:

$$n! = 1 \times 2 \times \cdots \times (n-1) \times n\,.$$

Using the product notation we may write this as follows:

$$n! = \prod_{i=1}^{n} i\,.$$

Usually, you compute factorials using a `for` statement:

```java
public static int factorial( int n ) {
    int product = 1;
    for (int i = 1; i != n; i ++) {
        product = product * i;
    }
    return product;
}
```

We can also compute factorials using recursion. The following is the key to the solution:

Base Case: Clearly $1! = 1$.[1] This is the base case.

Recursion: The recursion may be found by noticing that

$$\prod_{i=1}^{n} i = n \times \prod_{i=1}^{n-1} i\,.$$

This gives us

$$n! = (n-1)! \times n\,.$$

Note that this is clealy a recursive definition: the operator ! at the left hand side is defined in terms of itself since it also occurs in the right hand side.

The following socalled *case-based definition* combines the base case and the recursive case in a single formula.

$$n! = \begin{cases} 1 & \text{if } n = 1\,; \\ (n-1)! \times n & \text{if } n > 1\,. \end{cases}$$

Given this definition we almost get our Java for free:

```java
public static int factorial( int n ) {
    if (n == 1) {
        return 1;                    // Base Case
    } else {
        return factorial( n - 1 ) * n; // Recursion
    }
}
```

[1] Actually, $0! = 1$ is also true.

Note that the name of the previous method does not start with a verb. Strictly speaking this violates our Java naming conventions. However, the name is still quite acceptable because '`factorial( n )`' may be pronounced '`factorial` of `n`', which is exactly what the method computes. As a matter of fact, many methods in the `Math` class have similar names: `sin`, `cos`, `min`, `max`, ...

3 Fibonacci Numbers

This section studies another application of recursion. This time the application is a problem involving rabbits. The problem was invented by the famous Italian mathematician Fibonacci.

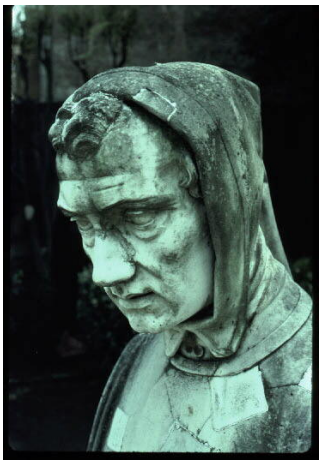

Figure 3: Leonardo of Pisa (Fibonacci).

The following are some basic facts.

- Born: about 1175 AD.
- Died: 1250 AD.
- Famous mathematician.
- Introduced the Decimal System into Europe.
- Well known for many of his problems.

3.1 A Fibonacci Problem

One of Fibonacci's problems is the following:

> A pair of rabbits are put in a field and, if rabbits take a month to become mature and then produce a new pair every month after that, how many pairs will there be in twelve months time?

He assumes the rabbits do not escape and none die.

3.2 Fibonacci's Solution

Fibonacci's problem gives rise to a famous sequence of numbers which are now called the Fibonacci Numbers. Table 1 lists the first few Fibonacci numbers. In this table the n-th Fibonacci number is denoted F_n. Remember that F_n is the number of pairs of rabbits, n months after a single pair starts breeding (and newly born bunnies start getting offspring when they are two months old).

Month (n)	Pairs of Rabbits		
	Babies	Mature	Total (F_n)
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8

Table 1: First Fibonacci numbers.

3.3 The Fibonacci Sequence

Fibonacci's solution involves the series of numbers:

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots .$$

Given the first two numbers we can compute each of the remaining numbers. The following shows how:

$$F_n = \begin{cases} 1 & \text{if } n = 0; \\ 1 & \text{if } n = 1; \\ F_{n-1} + F_{n-2} & \text{if } n > 1. \end{cases}$$

Given the case-based definition we can translate it into Java as follows:

```java
public static int fibonacci( int n ) {
    if (n <= 1) { /* Base Case */
        return 1;
    } else {      /* Recursion */
        return fibonacci( n - 1 ) + fibonacci( n - 2 );
    }
}
```

3.4 Tracing the Calls

It is instructive to study an example of an application of the `fibonacci` method to a small integer argument. Figure 4 depicts a socalled *call trace* of the call `fibonacci( 4 )`. for simplicty the tree uses $f(n)$ for `fibonacci( n )`. The node at the root of the tree corresponds to the top-level call $f(5)$. The nodes at the leaf positions correspond to the base cases: $f(n)$, where $n \leq 1$. The other nodes correspond to calls which require recursive method calls. Each of them has two children. For any such node, $f(n) = F_n$, the left child corresponds to the call $f(n-1)$ and the right child corresponds to the call $f(n-2)$.

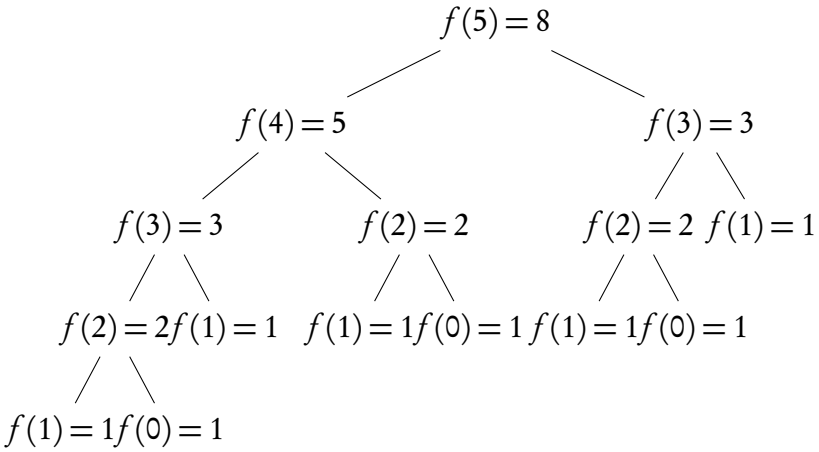

Figure 4: Call trace of $f(5)$, where f is given by $f(n) = 1$ if $n \leq 1$ and $f(n) = f(n-1) + f(n-2)$ if $1 < n$.

4 Towers of Hanoi

Our next problem originates from recreational mathematics. The problem was invented in 1883 by the French mathematician Edouard Lucas [Graham *et al.*, 1989, Chapter 1]. It is a textbook example of recursive problem solving. We're given a tower of 8 disks and three pegs: A, B, and C. Each disk has a hole in the centre. Initially, the disks are stacked in decreasing size on Peg A. The objective is to transfer the stack to a different peg, but

- we're only allowed to stack disks on pegs,
- we're only allowed to move one disk at a time, and
- we can only stack a smaller disk on top of a larger disk.

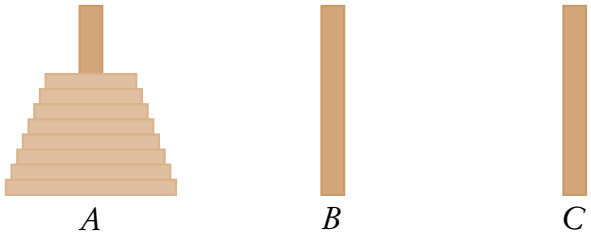

Figure 5: Initial state of the Towers of Hanoi.

Figure 5 depicts the initial situation.

When solving a problem like this, try solving a few small instances by hand. If you're lucky you may spot a pattern. Let's consider the 3-disk version of the general problem. Initially, the disks were stacked on Peg A. In the final state, the disks were stacked on Peg C. Figure 6 depicts one of the intermediate states. We arrived at this state by the following moves: $A \to B$, $A \to C$, $B \to C$, and $A \to B$. In the

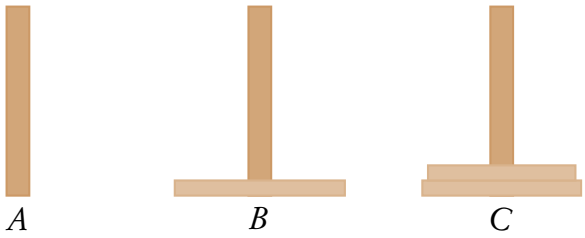

Figure 6: Intermediate state of the 3-disk version of the Towers of Hanoi.

intermediate state the largest disk is in its final position. The purpose in the intermediate state is to move the disks from Peg C to Peg B. But this is just the 2-disk version with C as initial and B as destination peg. We know how to solve the 2-disk version. How did we arrive at the intermediate state? If we can solve this sub-problem then we can solve the whole problem:

1. Start at initial state.
2. Solve the sub-problem to arrive at the intermediate state.
3. Use recursion to go from the intermediate to the target state.

So, how did we get at the intermediate state?

1. We started with all disks stacked on Peg A.

2. We moved all disks except for the largest one from A to C. But this is just the 2-disk version with A as the initial and C as the destination disk.

3. We moved the largest disk to Peg B.

Let's see if we can generalise this to a solution strategy for the n disk version. We already noticed that in the general version the source and destination pegs may be different. We have to take this into account.

Base case: If $n = 1$:

1. *Move disk n to target peg.*

Recursion: If $n > 1$:

1. Move disks 1, ..., $n - 1$ from source to intermediate peg.
2. *Move disk n to target peg.*
3. Move disks 1, ..., $n - 1$ from intermediate to target peg.

We could combine the two cases and have an `if` statement with two clauses: the `if` clause deals with the base-case and the `else` clause deals with the recursion. However, there is a more elegant solution which has a different base-case: if there are *no* disks then do nothing. Using this base-case we may express the algorithm more elegantly:

- If $n \geq 1$ then

1. Move disks 1, ..., $n - 1$ from source to intermediate peg.
2. Move disk n to target disk.
3. Move disks 1, ..., $n - 1$ from intermediate to target peg.

Exercise 1. *Prove that the algorithm terminates.*

The Java almost comes for free. For ease of use we provide a wrapper method `void hanoi( int n )` which moves `n` disks from the first to the second peg. To carry out this task the wrapper method calls the more complex helper method `void hanoi( int n, int source, int target )`.

```Java
/**
 * @param n Number of disks.
 * @param source The source peg: should be 0, 1, or 2.
 * @param target The target peg: should be 0, 1, or 2.
 * <PAR> {@code source} and {@code target} should be different.</PAR>
 */
private static
void hanoi( int n, int source, int target ) {
    if (n >= 1) {
        // Compute the number of the intermediate peg:
        final int intermediate = 3 - source - target;
        hanoi( n - 1, source, intermediate );
        moveDisk( n, source, target );
        hanoi( n - 1, intermediate, target );
    }
}

public static
void hanoi( int n ) {
    // move n disks from Peg 0 to Peg 1.
    hanoi( n, 0, 1 );
}
```

The method `moveDisk` just prints some text indicating which disk is moved from which peg to which other peg. Before we study this method, it is interesting to study the technique which computes the intermediate disk as a function of the two remaining disk numbers. This trick works because 3 is the sum of all disk numbers. Since 3 is the sum of *all* disk numbers, $3 = \texttt{source} + \texttt{intermediate} + \texttt{target}$, we should get `intermediate` if we subtract `source` and `target` from 3.[2] This trick saves an extra parameter at the expense of a small computation.

The following is the method `moveDisk`.

```Java
private static void moveDisk( int disk, int source, int target ) {
    final String pegNames[] = { "A", "B", "C" };
    System.out.println( "Move disk " + disk
                        + " from " + pegNames[ source ]
                        + " to " + pegNames[ target ] );
}
```

5 Binary Search

Binary search is an algorithm which(1) determines whether a given item is in a sorted list, and (2) if it is, returns the position of that element in the list. It works like the "dictionary search" algorithm. It repeatedly halves the number of elements needed to be checked. It is a typical case of a *divide and conquer* algorithm. Because of the halving it is sometimes called *dichotomic*. It requires a (worst-case) time which is logarithmic in the size of the input list.

5.1 The Basic Idea

Before studying the algorithm let's define its main task.

[2] We've seen this trick before when we were flipping bits: $1 - 1 \rightarrow 0$, and $1 - 0 \rightarrow 1$. Here the things are 0 and 1. The number 1 also plays the role of the sum.

Input: The input of the algorithm consists of (1) an item and (2) a list of items which is sorted in non-decreasing order. For simplicity we'll assume the items in the list are unique. With this assumption, the list is now sorted in increasing order.

Output: The output of the algorithm is an `int`. The output depends on one of the following cases.

Item is in list: If the item is in the list, the algorithm returns the index of the item in the list.

Item is not in list: If the item is not in the list the algorithms returns a negative number.

For simplicity we'll assume that all items are `int`s. Furthermore, we'll assume that the list is presented as an array. At the end of this section we shall study a version of the algorithm that works for arrays consisting of `Comparable` objects.

5.2 The Algorithm

The following is the algorithm. The input item is given by an `int` called `item` and the list of items is given by an `int` array called `items`. In addition we're given two `int`s called `lo` and `hi`. The purpose of these `int`s is to specify the range in `items` which the algorithm is supposed to search. Specifically, the algorithm's range is restricted to the indices which are not below `lo` and do not exceed `hi`.

`lo > hi:` Here there are no more valid index positions left. We terminate by returning `-1`.

`lo <= hi:` Here we still don't know if `item` is in `items[ lo .. hi ]`.

1. Determine "the" middle index. We implement this as `mid = (lo + hi) / 2`. Unfortunately, this is not correct due to overflow. You can fix this by implementing it as '`mid = lo + (hi - lo) / 2`' or as '`mid = (hi + lo) >>> 1`'.[3] For simplicity we shall assume that `mid = (lo + hi) / 2` is correct.
2. Compare `item` and `items[ mid ]`. There are three cases:
   - **`item == items[ mid ]`**: If this happens we've found the location of `item` in `items` and we return `mid`.
   - **`item < items[ mid ]`**: If this happens `item < items[ index ]` for all indices `index` such that `mid <= index`. The reason for this follows from the fact that `items` is sorted in increasing order. Effectively, this rules out the indices `index` where `mid <= index`. If `item` is among `index[ lo .. hi ]` then this can only be for index positions which are greater than or equal to `lo` and less than `mid`. This justifies the decision to recurse and return `binSearch( item, items, lo, mid - 1 )`.
   - **`item > items[ mid ]`**: If this happens `item > items[ index ]` for all indices `index` such that `index <= mid`. Here we recurse by returning `binSearch( item, items, mid + 1, hi )`.

[3] The operator `>>>` is a so called *shift* operator. The operation `lhs >>> bits` shifts the bits of `lhs` by `bits` bits to the right.

5.3 Implementation in Java

The following is a possible implementation in Java.

```java
public static
int binSearch( int item, int[] items ) {
    return binSearch( item, items, 0, items.length - 1 );
}

public static
int binSearch( int item, int[] items, int lo, int hi ) {
    if (lo > hi) {
        return - 1;
    } else {
       int mid = (lo + hi) / 2;
       if (item == items[ mid ]) {
           return mid;
       } else if (item < items[ mid ]) {
           return binSearch( item, items, lo, mid - 1 );
       } else {
           return binSearch( item, items, mid + 1, hi );
       }
    }
}
```

5.4 `Comparable` Interface

We've seen how to use binary search for `int`s. We should be able to generalise it for other *comparable* things. In the remainder of this section we shall study how to generalise the algorithm. The key to the idea is *implementing* the `Comparable` *interface*.

Implementing an *interface* is almost the same as extending a class.[4] For example, if class B implements interface A, then B behaves as a "subclass" of A. A class *implements* the `Comparable` *interface* if it overrides the following method:

`int compareTo( Object that )` The method should return a negative value if `this` is smaller than `that`, a positive value if `this` is greater than `that`, and zero otherwise.

Many classes implement the `Comparable` interface: `Integer`, `Double`, `String`, ...

The following implementation of the binary search algorithm works for arrays of `Comparable` objects. Note that we're using the method `compareTo( )` to get an `int` which we can use to compare `item` and `items[ mid ]`.

[4] We shall study interfaces in a few lectures time.

```java
public static
int binSearch( Comparable item, Comparable[] items, int lo, int hi ) {
    if (lo > hi) {
        return - 1;
    } else {
       int mid = (lo + hi) / 2;
       int compare = item.compareTo( items[ mid ] );
       if (compare == 0) {
           return mid;
       } else if (compare < 0) {
           return binSearch( item, items, lo, mid - 1 );
       } else {
           return binSearch( item, items, mid + 1, hi );
       }
    }
}
```

The implementation of `binSearch` is not ideal. For example, the method may cause a run time error when `item` is an `Integer` and `items` is an array of `String`. When we shall study *generic* types we shall learn how to implement the method properly. Still the implementation works if `item` and the members of `items` are "compatible", for example, if they are of the same class.

6 Quicksort

Sorting algorithms are a very important class of algorithms. Sorting efficiently is crucial to many applications. The `quicksort` algorithm is one of the most commonly used sorting algorithms. Given n items its expected number of comparisons is $O(n \log n)$. Here the notation $O(f(n))$ means 'proportional to $f(n)$', so the expected number of comparisons required by `quicksort` is proportional to $n \log n$. However, despite the $O(n \log n)$ (expected) comparisons for random input, there is input for which the algorithm requires $O(n^2)$ comparisons. These cases occur when the input data are almost sorted or almost reversed sorted. Even for relatively small n, say $n = 10^{10}$, algorithms with a $O(n^2)$ time complexity are prohibitive because they will (virtually) never terminate.

One of the advantages of `quicksort` is that if the input is provided as an array then the sorting can be carried out *in-situ*. Here sorting the array *in-situ* means sorting the array without the need of extra space (except for a constant number of variables). The algorithm was invented by C. A. R. Hoare in 1962.

For simplicity we shall study the version for sorting `int` arrays. The class `Arrays` defines several `quicksort`-based sorting methods.

6.1 Main Ideas

The following are the main ideas behind `quicksort`. In the following, n is the number of items which need to be sorted.

Base case: If $n \leq 1$ then the input is sorted.

Recursion: If $n > 1$:

1. Select any item from the input. This item is usually called the *pivot*. Ideally the pivot should be an item from the array such that about half the remaining items are smaller than or or equal to the pivot.
2. Partition the remaining items into two classes, L and G. L are the items less than or equal to the pivot. G are the remaining items.
3. The members in L should end up before the members of G.
4. After the partitioning, the pivot is put between the members of L and G.
5. Recursively sort L and G.

Exercise 2. *Prove that* `quicksort` *terminates.*

6.2 Implementation in `Java`

The following is the algorithm, which acts as a wrapper for the core algorithm. The core algorithm is parameterised over the start and end indices. Providing a wrapper function like the one below is a proper thing to do: you don't want to have the programmer provide the same lower index `0` and upper index `items.length - 1` each time the algorithm is used.

```Java
public static
void qsort( int[] items ) {
    qsort( items, 0, items.length - 1 );
}
```

The core algorithm is as follows.

```Java
// Sorts items[ lo .. hi ] in non-descending order.
private static
void qsort( int[] items, int lo, int hi ) {
   if (hi - lo >= 1) {
      int pivotPosition = partition( items, lo, hi );
      qsort( items, lo, pivotPosition - 1 );
      qsort( items, pivotPosition + 1, hi );
   }
}
```

The following is the `partition` algorithm. The pivot can be any member of the input array. Ideally it should be any number which ends up in the middle of the sorted array. Unfortunately, there is no cheap way to determine such a member. A reasonable choice is to select the number in the centre.

```Java
private static
int partition( int[] items, int lo, int hi ) {
   int destination = lo;
   swop( items, (hi + lo) >>> 1, hi );
   // The pivot is now stored in items[ hi ].
   for (int index = lo; index != hi; index ++) {
      if (items[ hi ] >= items[ index ]) {
         // Move current item to start.
         swop( items, destination, index );
         destination ++;
      }
      // items[ i ] <= items[ hi ] if lo <= i < destination.
      // items[ i ] >  items[ hi ] if destination <= i <= index.
   }
   // items[ i ] <= items[ hi ] if lo <= i < destination.
   // items[ i ] >  items[ hi ] if destination <= i < hi.
   swop( items, destination, hi );
   // items[ i ] <= items[ destination ] if lo <= i <= destination.
   // items[ i ] >  items[ destination ] if destination < i <= hi.
   return destination;
}
```

The brilliant aspect of the algorithm is that we temporarily swop the pivot with the item at position `hi`. We then partition `items[ lo .. hi - 1 ]` such that all items less than or equal to the pivot are at the start and the remaining items at the end. While doing this, we compute the position where the pivot will end up upon return. This position is called `destination`. Initially it is set to `lo`. After the partitioning of `items[ lo .. hi - 1 ]` we swop the items at positions `destination` and `hi`. We finish by returning the position of the pivot.

The implementation of the helper method `swop` is left as an exercise.

6.3 A Call Trace Study

Figure 7 depicts a call trace of `qsort`. The input of the method is the array `{2,5,4,1,3,8}`. The top-level node corresponds to the call `qsort( items, 0, 5 )`, where `items` is the input array. Each node corresponds to a call of the form `qsort( items, lo, hi )`. The value of `items` at the moment of the call is listed as part of the nodes. The values of `lo` and `hi` are listed as subscripts of the array `items`. The value of `lo` is listed first. The left subtree of a node corresponds to the first recursive call and the right subtree to the second.

Exercise 3. *Carry out a simulation of the algorithm* `partition` *for each of the internal nodes (the ones with children). You should be able to check the result of your computation by looking at the node's left (or right) child node.*

7 For Wednesday

For Wednesday:

- Study the lecture notes.
- Carry out Exercise 3.

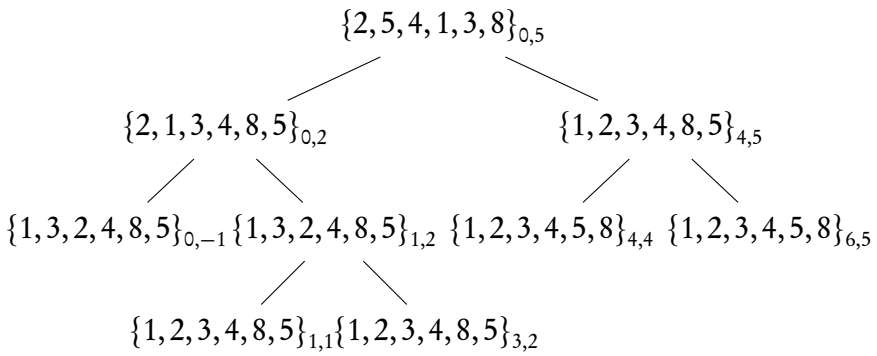

Figure 7: Call trace of `qsort`.

References

[Graham *et al.*, 1989] R.L. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989.