Software Development (CS2500)

Lecture 53: **Tail Recursion and Search**

M.R.C. van Dongen

March 4, 2011

Contents

1 Outline

This lecture is about *tail recursion* and search.

Tail recursive algorithms are an important class of algorithms. They are recursive, but at a very shallow level. They are "almost" iterative. This allows us to automatically translate them into iterative algorithms. The advantage of doing this is that it improves the performance.

Linear search outperforms binary search for small input sizes. It is an important requirement for the efficient implementation of many algorithms. Any further improvement to linear search will therefore

also improve these other algorithms. We shall study a number of ideas to improve the performance of linear search. These ideas can also be used for other algorithms. The part about linear search is partially based on [Knuth, 1998, Chapter 6.1].

Many of the code transformations are improvements which are carried out by a compiler. The result of the transformations will usually violate our coding conventions. To keep things simple we shall not bother about this most of the times.

2 Tail Recursion

Recursive algorithms are elegant and their correctness/termination properties are usually easy to prove. However, they have the disadvantage of having method call "overhead." To overcome this problem of overhead, programmers frequently first implement a recursive algorithm and (if possible) transform it to a more efficient non-recursive (iterative) algorithm.

A Java method is called *tail recursive* if (during its current invocation) any recursive call is immediately followed by a the return of the current invocation. Modern compilers are able to automatically transform any tail recursive method to an equivalent iterative method.

To study tail recursion we shall use the *Fibonacci number sequence*. The following are the Fibonacci numbers:

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

Except for the first two numbers, each number is the sum of its two predecessors. Writing f_n for the n-th Fibonacci number we have

$$f_n = \begin{cases} 1 & \text{if } n \leq 1\,; \\ f_{n-1} + f_{n-2} & \text{otherwise}\,. \end{cases}$$

The following method computes the n-th Fibonacci number.

```java
public static int
int f( int n ) {
    if (n <= 1) {
        return 1;
    } else {
        return f( n - 1 ) + f( n - 2 );
    }
}
```

The method is not tail recursive because there is an addition and a call to `f( n - 2 )` after the call to `f( n - 1 )`.

The previous method for Fibonacci numbers is hopelessly inefficient. The reason for the inefficiency is that the number of method calls that are required to compute f_{n-2} is almost twice the number of calls for that are required to compute f_{n-1}. As a consequence, the number of calls for f_n is in $\mathcal{O}(2^n)$. Figure 1 graphically shows this effect for $n \leq 6$.

When humans compute Fibonacci numbers they don't use the top-down recursion. Instead they

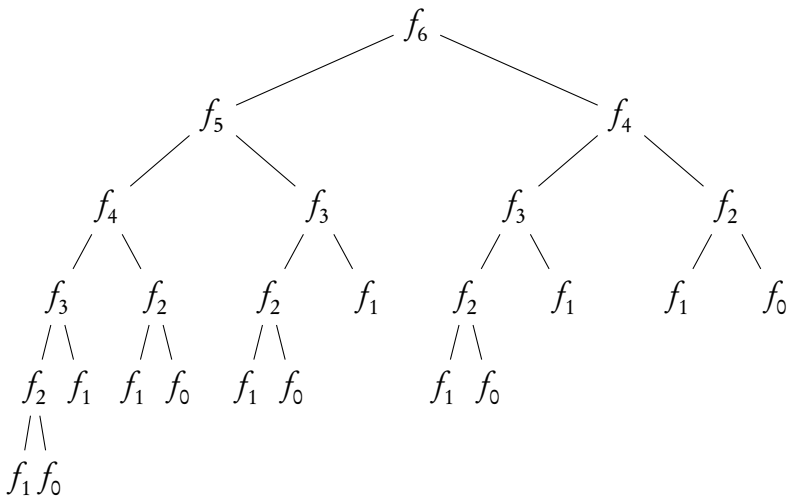

Figure 1: Fibonacci tree of order 6. The root at the top represents the computation of f_6.

enumerate the following sequence:

$$\underbrace{\langle f_0, f_1 \rangle, \langle f_1, f_2 \rangle, \langle f_2, f_3 \rangle, \ldots, \langle f_{n-1}, f_n \rangle}_{\text{length } n},$$

and return f_n.There's no need to explicitly construct pairs $\langle f_i, f_{i+2} \rangle$. In terms of recursion we can do this as follows:

$$f_n = \begin{cases} 1 & \text{if } n = 0\,; \\ F(1,1,1,n) & \text{otherwise}\,, \end{cases}$$

where $F(f_{i-1}, f_i, i, n)$ is given by:

$$F(f_{i-1}, f_i, i, n) = \begin{cases} f_i & \text{if } i = n\,; \\ F(f_i, f_i + f_{i-1}, i+1, n) & \text{otherwise}\,. \end{cases}$$

The following implements `F` in `Java`.

```java
public static int
int F( int fibPrev, int fibCurr, int curr, int n ) {
    if (curr == n) {
        return fibCurr;
    } else {
        return F( fibCurr, fibPrev + fibCurr, curr + 1, n );
    }
}
```

This definition is tail recursive. A clever `Java` compiler may translate the tail recursive definition of `F` to:

```java
public static int
int F( int fibPrev, int fibCurr, int curr, int n ) {
    while (curr != n) {
        int fibPrevOld = fibPrev;
        int fibCurrOld = fibCurr;
        fibPrev = fibCurrOld;
        fibCurr = fibCurrOld + fibCurrOld;
        curr ++;
    }
    return fibCurr;
}
```

To see that this works, let's trace the body of the `while` loop for `n == 6`. Table 1 depicts the trace. Now the computation of F_n requires n iterations of the `while` loop in `F`.

`fibPrev`	`fibCurr`	`curr`
1	1	1
1	2	2
2	3	3
3	5	4
5	8	5
8	13	6

Table 1: Trace of `while` loop of the method `F` for `n == 6`.

3 Search

In the remainder of these notes we shall study some more examples of tail-recursive methods. Each is based on the notion of search:

GIVEN: A key k and a collection of items.

TASK: Look up item with key k.

Examples:

- Given the name of a person, look up their phone number.
- Look up the meaning of the French word "entrepreneur" in a French to English dictionary.
- Enter a customer's account number in to the computer application and look up the balance of their current account.

It may not always be true that such queries result in a success:

- A person's phone number may not be listed.
- A word may not be listed in the dictionary.

- An account number may be invalid because of a typo.
- ....

In the remainder of the lecture we shall study some search algorithms. To simplify things we shall assume that we are given an array with keys and have to look up a given key. The type of our key is `int`. For simplicity we shall compare `int`s using the usual order.

4 Linear Search

Our first search problem is as follows.

GIVEN: *unordered* array, `keys`, of keys.

QUESTION: Does `keys` contain `key`, and if so at which position?

This problem is best solved using *linear search* which roughly boils down to seeking for `key` in `keys` from "left to right":

1. If there are no more keys then `key` is not in `keys`.
2. Otherwise, if the key is not equal to `key` then search for `key` in the remaining keys.
3. Otherwise `key` is in `keys` at the current position.

The following is a tail-recursive method implementing linear search.

```Java
/**
 * Linear search algorithm for deciding if keys[ lo..hi ] contains key.
 * ASSUMPTION: lo <= hi + 1.
 **/
public static
int linSearch( int[] keys, int key, int lo, int hi ) {
    if (lo > hi) {
        // There are no more keys left.
        return -1;
    } else if (keys[ lo ] != key) {
        // Search for key in remaining keys.
        return linSearch( keys, key, lo + 1, hi );
    } else {
        // We've located key.
        return lo;
    }
}
```

The following is an iterative version of the linear search algorithm.

```java
public static
int linSearch( int[] keys, int key, int lo, int hi ) {
    while (true) {
        if (lo > hi) {
            return -1;
        } else if (keys[ lo ] != key) {
            lo = lo + 1;
        } else {
            return lo;
        }
    }
}
```

Notice that the previous algorithm was obtained automatically from the tail-recursive definition. The resulting method is not particularly neat and it would not be wise to submit something like this for an assignment. For example, there are two `return` statements (exit points) in the method's body and a good algorithm should always have a single exit point.

5 Binary Search

Our second problem is as follows:

GIVEN: An array, `keys`, of (distinct) keys which are *ordered* in strictly ascending order. We are aslo given the index, `lo`, of the first key and the index, `hi`, of the last key.

QUESTION: Does `keys` contain a given key, `key`, and if so at which position?

Humans usually solve this problem using *binary search*.

To look up word `word` in `keys[ lo..hi ]`, the binary search algorithm proceeds as follows:

1. If `hi < lo` then `key` is not in `keys`.
2. Else assign `(lo + hi) / 2` to `mid`. This splits `keys` into three parts:
   (I) Keys before position `mid`. These keys are in `keys[ lo..mid - 1 ]`.
   (II) Keys after position `mid`. These keys are in `keys[ mid + 1..hi ]`.
   (III) Key `keys[ mid ]`.

   Note that (I) and (II) are about half the size of `keys[ lo..hi ]`.
3. There are three possibilities:
   (I) If `keys[ mid ] > key` then search for `key` in `keys[ lo..mid - 1 ]`.
   (II) Else if `keys[ mid ] < key` then search for `key` in `keys[ mid + 1..hi ]`.
   (III) Else `key` is in `keys` at position `mid`.

The following tail recursive method implements the binary search algorithm.

```java
public static
int binSearch( int[] keys, int key, int lo, int hi ) {
   if (lo > hi) {
      return -1;
   } else { // key is in keys[ lo..hi ]
      int mid = (lo + hi) / 2;

      if (key < keys[ mid ]) {
         return binSearch( keys, key, lo, mid - 1 );
      } else if (key > keys[ mid ]) {
         return binSearch( keys, key, mid + 1, hi );
      } else {
         return mid;
      }
   }
}
```

Exercise. *Transform the recursive method for binary search to an iterative version.*

6 Analysis

It is instructive to analyse the average running time of our iterative linear and binary search algorithms. We shall measure the running time *indirectly* by estimating the number of iterations of the algorithms for a random array `keys` of size `n` and a random key.

We shall first estimate the average running time of linear search. There are two cases to consider:

1. `key` is not in `keys`: We have to carry out the statements in the body of the `while` statement for each of the n keys.

2. `key` is in `keys`: It is reasonable to assume that `key` is "random" in the sense that it should be equally likely that `key == keys[ i ]` for all index position i. Then the probability that `key == keys[ i ]` is $1/n$. Given this probability the expected number of iterations is given by

$$\frac{1}{n}\sum_{i=1}^{n} i = (n+1)/2\,.$$

Our analysis gives us the worst-case running time for free:

Theorem. *The worst-case running time of linear search is $\mathcal{O}(n)$.*

We shall now analyse the worst-case time complexity of binary search.

As we've seen, the algorithm reduces the number of relevant index positions by a factor of two in each iterations. For such algorithms it is useful to assume that the input size is proportional to some (discrete) power of 2. To this end, let's assume that $n = 2^s$, for some integer s.

Next we consider the relationship between the i-th iteration and the number of keys left during the i-th iteration. Table 2 depicts this relationship. This gives us the following result: There are at most s, i.e.

number of iteration	1	2	3	...	s
number of keys left never exceeds	2^s	2^{s-1}	2^{s-2}	...	1

Table 2: Relationship between iteration and the maximum possible number of relevant keys in `keys`.

$\log_2(n)$ iterations. Therefore the worst-case running time of binary search is $\mathcal{O}(\log(n))$.

The *average* running time can only be better! (But not much.)

Using the Big-Oh notation we may now express the worst-case time complexity of linear search as follows.

Theorem. *The worst-case time complexity of binary search is* $\mathcal{O}(\log n)$.

7 Further Improvements to Linear Search

In our analysis we assumed that the time complexity only depends on the number of comparisons. This is a reasonable assumption if the number of keys, n, is large. However, it is not a reasonable assumption if n is small: an important case for many search applications. For example, it has been observed that linear search algorithm may outperform the binary search algorithm if n is small.

In the remainder of these notes we shall study some clever transformations which improve the linear search algorithm.

7.1 Sorted Keys

If the keys are sorted then the performance of the linear search algorithm may be improved. The following is an iterative implementation of linear search for sorted `keys`. The members of `keys` are sorted in non-decreasing order.

```Java
/**
 * Improved linear search algorithm.
 * The keys are sorted from small to large.
 **/
public static
int linSearch( int[] keys, int key, int lo, int hi ) {
    while (lo <= hi) {
        if (keys[ lo ] < key) {
            lo = lo + 1;
        } else if (keys[ lo ] > key) {
            return -1;
        } else {
            return lo;
        }
    }
    return -1;
}
```

We exploit the fact that the keys are ordered. Therefore, if `keys[ lo ] > key` then `keys[ i ] > key` for any remaining `i` in `lo+1 -hi`.

7.2 Sentinel

Before implementing the next improvement, we need to tidy up our previous algorithm. The reason for doing this is that there is a `return` inside the `while` loop. Some people argue that this is not a good idea. For example, several `return` statements makes it more difficult to understand the algorithm. The following code fragment is equivalent to the previous algorithm (prove this) but it has the advantage that the algorithm has only one entry and one exit point.

```Java
public static
int linSearch( int[] keys, int key, int lo, int hi ) {
    int cmp = -1;

    while ((lo <= hi) && (cmp < 0)) {
        cmp = ( keys[ lo ] < key ? -1
              : keys[ lo ] > key ?  1 : 0 );
        if (cmp < 0) {
            lo = lo + 1;
        }
    }
    return (cmp != 0 ? -1 : lo);
}
```

Our second improvement is more subtle. This improvement works if comparing keys is relatively

cheap and if we have a special key, `KEY`, which is larger than any other (allowed) key. If we have such key then we can eliminate the check `(lo <= hi)`.

As with the previous case we sort the keys from small to large. However, this time we put `KEY` immediately after the last key in `keys`. Then, when looking for a key `key` which is not in `keys[ lo..hi ]`, we will eventually run into the key `KEY` and then `KEY > key`, which proves that `key` is *not* in `keys[ lo..hi+1 ]`. What is more, the comparison `KEY > key` may be implemented as `key[ lo ] > key`, which is just the same as before. Effectively, this merges the termination condition for the case where `key` is not in `keys` with the termination condition for the case where `key` is in `keys`.

Compared to the previous algorithm we save one comparison in every iteration at the expense of having one more comparison if `key` is larger than the last member in `keys`. In addition there is no longer any need to pass the parameter `hi`.

The following is the improved algorithm. The variable `lo` has been renamed to `position`. Note that an ideal candidate for the sentinel `s` in the algorithm is the largest possible `int` is `Integer.MAX_VALUE`.

```Java
/**
 * Linear search with sentinel, s, at end of keys. The
 * keys are sorted from small to large. The sentinel
 * has the property that s > k for any valid key k.
 **/
public static
int linSearch( int[] keys, int key, int position ) {
    int cmp = -1;

    while (cmp < 0) {
        cmp = ( keys[ position ] < key ? -1
              : keys[ position ] > key ?  1 : 0 );
        if (cmp < 0) {
            position = position + 1;
        }
    }
    return (cmp != 0 ? -1 : position);
}
```

7.3 Final Improvement

Our final improvement is to increment `position` regardless of the value of `cmp`. Compared to the previous implementation it saves one comparison in every iteration, but if `key` is in `keys` then `position` will be the *successor* of the position of `key` in `keys`. If we return `position - 1` in this case then the result is still correct.

```Java
/**
 * Linear search with sentinel, s, at end of keys. The
 * keys are sorted from small to large. The sentinel
 * has the property that s > k for any valid key k.
 **/
public static
int linSearch( int[] keys, int key, int position ) {
    int cmp = -1;

    while (cmp < 0) {
        cmp = ( keys[ position ] < key ? -1
              : keys[ position ] > key ?  1 : 0 );
        position ++;
    }
    return (cmp != 0 ? -1 : position - 1);
}
```

Using the `do-while` construct we may avoid the initial assignment of `-1` to `cmp`.

```Java
/**
 * Linear search with sentinel, s, at end of keys. The
 * keys are sorted from small to large. The sentinel
 * has the property that s > k for any valid key k.
 **/
public static
int linSearch( int[] keys, int key, int position ) {
    int cmp;

    do {
        cmp = ( keys[ position ] < key ? -1
              : keys[ position ] > key ?  1 : 0 );
        position ++;
    } while (cmp < 0);
    return (cmp != 0 ? -1 : position - 1);
}
```

8 For Monday

- Study the lecture notes.
- Translate the tail recursive version of `binSearch` to an iterative version.

9 Bibliography

References

[Knuth, 1998] D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching, second edition. Addison–Wesley, 1998.