Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University
2010

Chapter 5. Introduction to Recursion

Outline

Introduction

A technique in which large problems are solved by reducing them to smaller problems of the same form.

Introduction

A technique in which large problems are solved by reducing them to smaller problems of the same form.

What is large? What is small?

A measurement of the size of the problem.

Introduction

A technique in which large problems are solved by reducing them to smaller problems of the same form.

What is large? What is small?

A measurement of the size of the problem.

The smaller problems must be of the same form as the large problem.

Outline

Factorial function

The function $f(n) = n!$.

Function prototype

```
int Fact(int n);
```

An iterative (nonrecursive) implementation

```
int Fact(int n) {
    int product;

    product = 1;
    for (int i = 1; i <= n; i++) {
        product *= i;
    }
    return product;
}
```

Factorial function (cont.)

The recursive formulation: $n! = n * (n-1)!$

A large problem (size n) is reduced to a smaller problem (size $n-1$) of the same form (factorial).

Stopping point (simple case, trivial case): $0! = 1$

Factorial function (cont.)

The recursive formulation: $n! = n * (n-1)!$

A large problem (size n) is reduced to a smaller problem (size $n-1$) of the same form (factorial).

Stopping point (simple case, trivial case): $0! = 1$

A recursive definition

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n(n-1)! & \text{otherwise} \end{cases}$$

A recursive implementation

```
int Fact(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * Fact(n-1);
    }
}
```

Tracing the recursive process

```
f = Fact(4)
```

main		
n	4	n * Fact(3)
n	3	n * Fact(2)
n	2	n * Fact(1)
n	1	n * Fact(0)
n	0	return 1

A stack of frames.

Outline

Fibonacci sequence

The sequence: $t_0, t_1, t_2, \ldots$

$$t_n = t_{n-1} + t_{n-2}, \quad t_0 = 0, \; t_1 = 1.$$

Fibonacci sequence

The sequence: $t_0, t_1, t_2, \ldots$

$$t_n = t_{n-1} + t_{n-2}, \quad t_0 = 0,\ t_1 = 1.$$

A recursive definition

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

Fibonacci sequence (cont.)

Function prototype

```
int Fib(int n);
```

A recursive implementation

```
int Fib(int n) {
    if (n < 2) {
        return n;
    } else {
        return (Fib(n - 1) + Fib(n - 2));
    }
}
```

Figure 5-1, p. 184.

Redundancy

`Fib(5)` calls `Fib(4)` and `Fib(3)`
`Fib(4)` calls `Fib(3)` and `Fib(2)`
`Fib(3)` calls `Fib(2)` and `Fib(1)` $\cdots$

Redundancy

`Fib(5)` calls `Fib(4)` and `Fib(3)`
`Fib(4)` calls `Fib(3)` and `Fib(2)`
`Fib(3)` calls `Fib(2)` and `Fib(1)` $\cdots$

one call to `Fib(4)`
two calls to `Fib(3)`
three calls to `Fib(2)`
five calls to `Fib(1)`
three calls to `Fib(0)`

Redundancy

`Fib(5)` calls `Fib(4)` and `Fib(3)`
`Fib(4)` calls `Fib(3)` and `Fib(2)`
`Fib(3)` calls `Fib(2)` and `Fib(1)` $\cdots$

one call to `Fib(4)`
two calls to `Fib(3)`
three calls to `Fib(2)`
five calls to `Fib(1)`
three calls to `Fib(0)`

Is recursion inefficient?

Outline

Additive sequence

A generalization of the Fibonacci sequence.

Given t_0 and t_1, $t_n = t_{n-1} + t_{n-2}$.

Function prototype

```
AdditiveSequence(int n, int t0, int t1);
```

Additive sequence

A generalization of the Fibonacci sequence.

Given t_0 and t_1, $t_n = t_{n-1} + t_{n-2}$.

Function prototype

```
AdditiveSequence(int n, int t0, int t1);
```

The Fibonacci sequence is a special case where $t_0 = 0$ and $t_1 = 1$.

Wrapper function

```
int Fib(int n) {
    return AdditiveSequence(n, 0, 1)
}
```

Additive sequence (cont.)

An observation:
The *n*th term in an additive sequence

$$t_0, t_1, t_2, t_3, \ldots$$

is the $(n-1)$st term in the additive sequence

$$t_1, t_2, t_3, \ldots \quad t_2 = t_0 + t_1$$

Additive sequence (cont.)

An observation:
The *n*th term in an additive sequence

$$t_0, t_1, t_2, t_3, \ldots$$

is the $(n-1)$st term in the additive sequence

$$t_1, t_2, t_3, \ldots \quad t_2 = t_0 + t_1$$

Implementation

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    if (n == 1) return t1;
    return AdditiveSequence(n-1, t1, t0 + t1);
}
```

Still a recursion, but no redundant calls!

Additive sequence (cont.)

An observation:
The *n*th term in an additive sequence

$$t_0, t_1, t_2, t_3, \ldots$$

is the $(n-1)$st term in the additive sequence

$$t_1, t_2, t_3, \ldots \quad t_2 = t_0 + t_1$$

Implementation

```
int AdditiveSequence(int n, int t0, int t1) {
    if (n == 0) return t0;
    if (n == 1) return t1;
    return AdditiveSequence(n-1, t1, t0 + t1);
}
```

Still a recursion, but no redundant calls!
Question: What happens if the `if (n == 1)` check is missing?

Additive sequence (cont.)

What makes the difference?

Additive sequence (cont.)

What makes the difference?

- `Fib(int n)` on p. 184 makes two overlapping recursive calls;
- `Fib(int n)` on p. 186 makes one recursive call.

Additive sequence (cont.)

What makes the difference?

- `Fib(int n)` on p. 184 makes two overlapping recursive calls;
- `Fib(int n)` on p. 186 makes one recursive call.

Note. Deep recursion can cause stack overflow.

Outline

Palindrome

A recursive formulation

- The first and last characters are the same.
- The substring generated by removing the first and last is a Palindrome.

Palindrome

A recursive formulation

- The first and last characters are the same.
- The substring generated by removing the first and last is a Palindrome.

Stopping points (trivial cases, simple cases):

Since we remove two characters (first and last) at a time, we end up with either a single-character string or an empty string.

Palindrome (cont.)

An implementation

```
bool IsPalindrome(string str) {
    int len = str.length();

    if (len <= 1) {
        return true;
    } else {
        return ((str[0] == str[len - 1]) &&
                IsPalindrome(str.substr(1, len - 2)));
    }
}
```

Improving efficiency

Using

- the positions of the first and last in the currently active substring.
- a wrapper.

Advantages of `CheckPalindrome`, p. 189:

- Calculate the length of the input string once;
- Avoid calling `substr` to make copy of substring.

`IsPalindrome`, Figure 5-4, p. 189.

Improving efficiency

Using

- the positions of the first and last in the currently active substring.
- a wrapper.

Advantages of `CheckPalindrome`, p. 189:

- Calculate the length of the input string once;
- Avoid calling `substr` to make copy of substring.

`IsPalindrome`, Figure 5-4, p. 189.

Why wrapper function?

Hide implementation. The interface of `IsPalindrome` is unlikely to be changed.

Outline

Binary search

Search for an element in an integer array sorted in ascending order.

Binary search

Search for an element in an integer array sorted in ascending order.

A recursive formulation:

Split the array in the middle, search the left half or right half depending on the given value.

Binary search

Search for an element in an integer array sorted in ascending order.

A recursive formulation:

Split the array in the middle, search the left half or right half depending on the given value.

Stopping point

- The mid-entry is the element.
- No elements in the active part of the array.

Binary search

Search for an element in an integer array sorted in ascending order.

A recursive formulation:

Split the array in the middle, search the left half or right half depending on the given value.

Stopping point

- The mid-entry is the element.
- No elements in the active part of the array.

Wrapper:

```
int FindIntInSortedArray(int key, int array[], int n) {
    return BinarySearch(key, array, 0, n-1);
}
```

Binary search (cont.)

```
int BinarySearch(int key, int array[],
                 int low, int high) {
    if (low > high) return -1;

    int mid = (low + high) / 2;
    if (key == array[mid] return mid;
    if (key < array[mid]) {
        return BinarySearch(key, array, low, mid - 1);
    } else {
        return BinarySearch(key, array, mid + 1, high);
    }
}
```

Outline

Mutual recursion

A general recursion.

Example.

`f` calls `g` and `g` calls `f`.

Function `IsEven`, Figure 5-6, p. 192.

Mutual recursion (cont.)

```
bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

Mutual recursion (cont.)

```
bool IsEven(unsigned int n) {
    if (n == 0) {
        return true;
    } else {
        return IsOdd(n - 1);
    }
}

bool IsOdd(unsigned int n) {
    return !IsEven(n);
}
```

Questions:

What happens if `if (n == 0)` check is missing in `IsEven`?

What happens if `if (n == 1)` check is added to `IsOdd`?

Thinking recursively

- Your program should look like

Standard recursion paradigm

```
if (test for simple case) {
    solve simple case
} else {
    call this function with smaller size
}
```

Thinking recursively (cont.)

- Find out all possible simple cases (stopping points). The recursion should end with a simple case.
- Test your program for the simple (trivial) cases.
- Determine a measurement of the size of the problem. Decompose a big problem into smaller problems of the same form. Apply the recursive leap of faith to make sure your program generates the complete solution.