Algorithms/Ada Implementation

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

Welcome to the Ada implementations of the Algorithms Wikibook. For those who are new to Ada Programming a few notes:

- All examples are fully functional with all the needed input and output operations. However, only the code needed to outline the algorithms at hand is copied into the text - the full samples are available via the download links. (Note: It can take up to 48 hours until the cvs is updated).
- We seldom use predefined types in the sample code but define special types suitable for the algorithms at hand.
- Ada allows for default function parameters; however, we always fill in and name all parameters, so the reader can see which options are available.
- We seldom use shortcuts - like using the attributes Image or Value for String <=> Integer conversions.

All these rules make the code more elaborate than perhaps needed. However, we also hope it makes the code easier to understand

Chapter 1: Introduction

The following subprograms are implementations of the *Inventing an Algorithm* examples.

To Lower

The Ada example code does not append to the array as the algorithms. Instead we create an empty array of the desired length and then replace the characters inside.

File: to_lower_1.adb (____, ______ ____, ________________________ ________________, ________________________)

```
function To_Lower (C : Character) return Character renames
   Ada.Characters.Handling.To_Lower;

--  tolower - translates all alphabetic, uppercase characters—in str to lowercase
function To_Lower (Str : String) return String is
   Result : String (Str'Range);
begin
   for C in  Str'Range loop
      Result (C) := To_Lower (Str (C));
   end loop;
   return Result;
end To_Lower;
```

Would the append approach be impossible with Ada? No, but it would be significantly more complex and slower.

Equal Ignore Case

File: to_lower_2.adb (____, ______ ____, ________________________ ________________, ________________________)

—equal-ignore-case—returns true if s or t are equal,

```
--  ignoring case
function Equal_Ignore_Case
  (S    : String;
   T    : String)
   return Boolean
is
   O : constant Integer := S'First - T'First;
begin
   if T'Length /= S'Length then
      return False;  --  if they aren't the same length, they—aren't equal
   else
      for I in  S'Range loop
         if To_Lower (S (I)) /=
            To_Lower (T (I + O))
         then
            return False;
         end if;
      end loop;
   end if;
   return True;
end Equal_Ignore_Case;
```

Chapter 6: Dynamic Programming

Fibonacci numbers

The following codes are implementations of the Fibonacci-Numbers examples.

Simple Implementation

File: fibonacci_1.adb (______________________________________, ______ ______________________________, ______________________ ______________, ________________________)

```
...
```

To calculate Fibonacci numbers negative values are not needed so we define an integer type which starts at 0. With the integer type defined you can calculate up until `Fib (87)`. `Fib (88)` will result in an `Constraint_Error`.

```
type Integer_Type is range 0 .. 999_999_999_999_999_999;
```

You might notice that there is not equivalence for the `assert (n >= 0)` from the original example. Ada will test the correctness of the parameter *before* the function is called.

```
function Fib (n : Integer_Type) return Integer_Type is
begin
   if n = 0 then
      return 0;
   elsif n = 1 then
      return 1;
   else
      return Fib (n - 1) + Fib (n - 2);
   end if;
end Fib;

...
```

Cached Implementation

File: fibonacci_2.adb (______________________________________, ______ ______________________________, ______________________ ______________, ________________________)

```
...
```

For this implementation we need a special cache type can also store a `-1` as "not calculated" marker

```
type Cache_Type is range -1 .. 999_999_999_999_999_999;
```

The actual type for calculating the fibonacci numbers continues to start at 0. As it is a **subtype** of the cache type Ada will automatically convert between the two. (the conversion is - of course - checked for validity)

```
subtype Integer_Type is Cache_Type range
   0 .. Cache_Type'Last;
```

In order to know how large the cache need to be we first read the actual value from the command line.

```
Value : constant Integer_Type :=
   Integer_Type'Value (Ada.Command_Line.Argument (1));
```

The Cache array starts with element 2 since Fib (0) and Fib (1) are constants and ends with the value we want to calculate.

```
type Cache_Array is
   array (Integer_Type range 2 .. Value) of Cache_Type;
```

The Cache is initialized to the first valid value of the cache type — this is `-1`.

```
F : Cache_Array := (others => Cache_Type'First);
```

What follows is the actual algorithm.

```
function Fib (N : Integer_Type) return Integer_Type is
begin
   if N = 0 or else N = 1 then
      return N;
   elsif F (N) /= Cache_Type'First then
      return F (N);
   else
      F (N) := Fib (N - 1) + Fib (N - 2);
      return F (N);
   end if;
end Fib;

...
```

This implementation is faithful to the original from the Algorithms book. However, in Ada you would normally do it a little different:

File: fibonacci_3.adb (______, ______, ______, ______)

when you use a slightly larger array which also stores the elements 0 and 1 and initializes them to the correct values

```
type Cache_Array is
   array (Integer_Type range 0 .. Value) of Cache_Type;

F : Cache_Array :=
   (0      => 0,
    1      => 1,
    others => Cache_Type'First);
```

and then you can remove the first `if` path.

```
~~   if N = 0 or else N = 1 then~~
~~      return N;~~
~~   els~~if F (N) /= Cache_Type'First then
```

This will save about 45% of the execution-time (measured on Linux i686) while needing only two more elements in the cache array.

Memory Optimized Implementation

This version looks just like the original in WikiCode.

File: fibonacci_4.adb (______, ______, ______, ______)

```
type Integer_Type is range 0 .. 999_999_999_999_999_999;

function Fib (N : Integer_Type) return Integer_Type is
   U : Integer_Type := 0;
   V : Integer_Type := 1;
begin
   for I in  2 .. N loop
      Calculate_Next : declare
         T : constant Integer_Type := U + V;
      begin
         U := V;
         V := T;
      end Calculate_Next;
   end loop;
   return V;
end Fib;
```

No 64 bit integers

Your Ada compiler does not support 64 bit integer numbers? Then you could try to use decimal numbers instead. Using decimal numbers results in a slower program (takes about three times as long) but the result will be the same.

The following example shows you how to define a suitable decimal type. Do experiment with the `digits` and `range` parameters until you get the optimum out of your Ada compiler.

File: fibonacci_5.adb (______, ______, ______, ______)

```
type Integer_Type is delta 1.0 digits 18 range
   0.0 .. 999_999_999_999_999_999.0;
```

You should know that floating point numbers are unsuitable for the calculation of fibonacci numbers. They will not report an error condition when the number calculated becomes too large — instead they will lose in precision which makes the result meaningless.

Top, Chapters: 1, 2, 3, 4, 5, 6, 7, 8, 9, A

Retrieved from "https://en.wikibooks.org/w/index.php?title=Algorithms/Ada_Implementation&oldid=1797508"

This page was last edited on 17 May 2010, at 13:52.

[redacted]