Dynamic Programming

Let us recall Fibonacci numbers:

\[ 1, 1, 2, 3, 5, 8 \]

which satisfy the following recurrence equation

\[ x_n = x_{n-1} + x_{n-2} \]
\[ x_1 = 1, \quad x_2 = 1 \]

Here is Mathematica implementation:

\[
\begin{align*}
    f[x_] &:= f[x-1] + f[x-2] \\
    Table[f[k], \{k, 1, 7\}] \\
    &\{1, 1, 2, 3, 5, 8, 13\}
\end{align*}
\]

This is a divide-and-conquer algorithm. It has a top-to-bottom approach: to compute 5th term you need to compute 4th and 3rd. However, to find 4th term you need to computer 3rd term again. This is a reason why divide-and-conquer approach is so inefficient, and it has an exponential complexity. We can measure a computation time by `Timing` (which gives the CPU time):

\[
\text{Timing}[N[Zeta[3], 100];]
\]
\[\{0.\text{Second}, \text{Null}\}\]

In order to see the exponential time needed to compute n-th Fibonacci number, we make the following experiment
Dynamic Programming is similar divide-and-conquer: an instance of the problem is divided into smaller instances. However, it has an opposite direction. Dynamic programming is a bottom-up approach: to compute 5th term we need to compute 1st, 2nd, 3rd and 4th. In Mathematica this idea can be implemented by caching (saving all previously computed values):

$$f[x_] := f[x] = rhs$$

Here is the implementation

```mathematica
Clear[g];
g[x_] := g[x] = g[x - 1] + g[x - 2]
g[1] = g[2] = 1;
```

We compute 10th Fibonacci number

```mathematica
g[10]
```

55

All previous values are stored
If you ask for $g[7]$, Mathematica will just look up the value; it does not have to recompute it. The above implementation has a linear complexity:

$$\text{Table}[\text{Timing}[g[n]][[1]], \{n, 10, 5000, 200\}]$$

Exercise 1

Binomial coefficients are defined by the recurrence formula

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Implement them, using caching mechanism, assuming that $n$ and $k$ are positive integers

Run a few examples and compare CPU time of your implementation with the built ones.
Solution

Clear[bin];
bin[n_, k_] := bin[n, k] = bin[n - 1, k - 1] + bin[n - 1, k]
bin[n_, n_] = 1;
bin[n_, 0] = 1;

bin[5, 2]
10

bin[150, 10]; // Timing
{0.16 Second, Null}

Binomial[150, 10]; // Timing
{0. Second, Null}

Here is a more efficient version:

Clear[sbin];
sbin[n_, k_] /; n > k := Product[j, {j, Evaluate[n - k + 1], n}] / k!
sbin[n_, n_] = 1;
sbin[n_, 0] = 1;

sbin[1500000, 100001]; // Timing
{18.01 Second, Null}

Binomial[150000, 100001]; // Timing
{5.33 Second, Null}

Functional Operations

Mathematica combines procedural and functional styles of programming. Functional operations provide the most efficient way to use Mathematica. Why? Mathematica is an interpreted language, so there is a lot of overhead in doing routine calculations. The idea of functional programming is to reduce the amount of interpretation overhead.

There are many kinds of functional operations, some of them are hard to understand. In this section will discuss two of them: Apply and Map.
The typical basic structure is a list, an expression with the head `List`. If you are to perform any operations over the list (like sorting, for example), you need to know how to apply a function either to the list or directly to the elements, or separately to each of the elements. Here we apply `Sort` to the whole list:

```
Sort@{3, 5, 1}
```

```
{1, 3, 5}
```

`f@x` is a prefix form for `f[x]`. The above is identical to `Sort[{3, 5, 1}]`.

If you want to apply a function to elements, you do (in a prefix form)

```
Plus @@ {3, 5, 1}
```

```
9
```

**Quiz.**

Write a function which sums up digits of a given integer.

```
foo[n_Integer] := Plus @@ IntegerDigits[n]
```

What happened here? We replaced the head `List` by `Plus`, and then `Plus` carried out computation. Here is a standard form of the input

```
Apply[Plus, {3, 5, 1}]
```

```
9
```

Sometimes you want to apply a function to each element (the prefix form):

```
f /@ {3, 5, 1}
```

```
{2, 5, 1}
```

We `Map` the function `f` to each element. Here is the standard form

```
Map[f, {3, 5, 1}]
```

```
{2, 5, 1}
```

**Quiz.**

Given a list of integers. Write a function which inverts all positive integers, and replaces all negative integers by 0.

```
foo[e_List] := If[Negative[#], 0, 1/#] & /@ e
```
Exercise 2

How long is the longest function name in *Mathematica*?

Commands: `StringLength`, `ToString`, `Map`

Look up the explanations for the commands above in the help browser, and you will see that the following commands solve the stated problem.

**Solution**

We know that `Names["System`*"]` gives the list of strings. We need to find the longest string. Use the wildcard to guess the function name which gives the string length

```
?String*
```

Point mouse onto `StringLength` and press F1. Yes, this is what we need, let's use it:

```
StringLength["my name"]
```

7

So, what we need is to convert symbols to strings. What function to use? `ToString` looks promising...

```
StringLength[ToString[Plus]]
```

4

Now we put them all together.

```
Max[StringLength /@ Names["System`*"]]
```

36

We map the function `StringLength` over the list produced by `Names`, then we compute the maximum.
Exercise 3

What is the longest *Mathematica* function name?

Solution

We know that the longest function name has 36 characters. What is it?

```mathematica
Select[Names["System`*"], StringLength[#] == 36 &]
```

{NotebookSetupLayoutInformationPacket}

What is the second longest name?

```mathematica
Select[Names["System`*"], StringLength[#] == 35 &]
```

{}  

```mathematica
Select[Names["System`*"], StringLength[#] == 34 &]
```

{NotebookGetLayoutInformationPacket, UndocumentedTestGetSelectionPacket}