CS29003 Algorithms Laboratory

Assignment No: 1

Date: 14–January–2020

Running Times of Algorithms

This assignment introduces the notion of *goodness* of algorithms. The same computational problem can be solved by many programs, of which some are bad, and some are good. A metric for assessing the quality of a program is the theoretical running time of the algorithm that the program implements. Reduction of these running times is a fundamental goal in algorithm design and analysis.

Counting neutrons in a fission bomb

The chain reaction in a fission bomb is triggered by bombarding a mass of plutonium by a neutron. For simplicity, assume that there are two types of bombarding neutrons: high-energy and low-energy. When a neutron hits a plutonium nucleus, it gets absorbed, but in the process, the nucleus breaks, releases energy, and produces more neutrons. A high-energy neutron leads to the release of two high-energy neutrons and one low-energy neutron, whereas a low-energy neutron causes the release of one high-energy neutron and one low-energy neutron. Assume that the chain reaction is clocked, that is, each fission happens at the beginning of a microsecond (our unit of time in this exercise). At $t = 0$, we start the chain reaction by bombardment with a single high-energy neutron.

Let H_n denote the number of high-energy neutrons and L_n the number of low-energy neutrons at time $t = n$. We have:

$$\begin{aligned} H_0 &= 1, \\ L_0 &= 0, \\ H_n &= 2H_{n-1} + L_{n-1} \text{ for } n \geqslant 1, \\ L_n &= H_{n-1} + L_{n-1} \text{ for } n \geqslant 1. \end{aligned}$$

Use a floating-point representation (preferably `double`) for H_n and L_n. These counts are integers but grow so rapidly that except for small values of n, `int` variables encounter overflow for storing them. Given an integer $n \geqslant 0$, we want to compute H_n and L_n. Three ways of doing this are explained below.

Method 0: Individual computation of H_n and L_n

Write a function $hirec(n)$ that returns H_n (and nothing else). Likewise, write a function $lorec(n)$ to return L_n (and nothing else). The only parameter that may be passed to each of these functions is n.

Method 1: Simultaneous computation of H_n and L_n

Write a function $hilorec(n)$ to return the pair (H_n, L_n) (in a structure or a two-element array). Again, the only parameter allowed to be passed to the function is n.

Method 2: Use of explicit formulas

Using techniques (slightly) beyond the scope of this course, we can obtain the following closed-form formulas valid for all $n \geqslant 0$.

$$\begin{aligned} H_n &= \left(\frac{5+\sqrt{5}}{10}\right)\left(\frac{3-\sqrt{5}}{2}\right)^{n+1} + \left(\frac{5-\sqrt{5}}{10}\right)\left(\frac{3+\sqrt{5}}{2}\right)^{n+1}, \\ L_n &= \left(\frac{-5-3\sqrt{5}}{10}\right)\left(\frac{3-\sqrt{5}}{2}\right)^{n+1} + \left(\frac{-5+3\sqrt{5}}{10}\right)\left(\frac{3+\sqrt{5}}{2}\right)^{n+1}. \end{aligned}$$

Write a function $hiloformula(n)$ that uses these formulas to compute and return the pair (H_n, L_n).

The *main*() function

Read n from the user. Call *hirec*(n), and print the return value (in scientific format). Likewise, call *lorec*(n), and print the return value. Then, call *hilorec*(n), and print the returned pair. Finally, call *hiloformula*(n), and print the returned pair.

Sample output

```
n = 25

+++ Method 0
    hi(25) = 2.0365011074e+10, lo(25) = 1.2586269025e+10

+++ Method 1
    hi(25) = 2.0365011074e+10, lo(25) = 1.2586269025e+10

+++ Method 2
    hi(25) = 2.0365011074e+10, lo(25) = 1.2586269025e+10
```

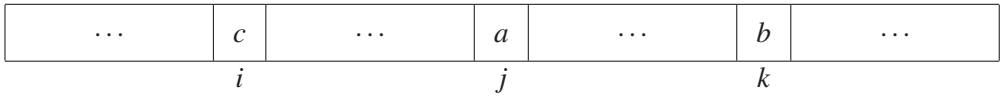
Check for algolicious sequences

The international chef Shri Shri Bhajohori Manna is in the making of a fabulous dish. He knows he has to add n sauces to the dish. These sauces are numbered $1, 2, 3, \ldots, n$. The dish becomes delicious if the sequence of adding the sauces is proper. Any improper sequence ruins the dish altogether, so Manna asks for your help to ensure that his sequence is proper. A proper sequence is defined as follows.

Let S be the array storing the sequence. Assume that each sauce (an integer in the range $[1, n]$) appears once and only once in S. In particular, the size of S is n. Take any three different sauces a, b, c with

$$1 \leqslant a < b < c \leqslant n.$$

An improper placement of a, b, c in S is described by a situation like this:



Here, c, a, b need not appear consecutively, but their relative position in S of this form is improper. The sequence S is called *algolicious* (and the dish tastes delicious) if S does not contain any improper placement of a, b, c for all possible choices of the sauces. For example, for $n = 10$, the sequence $3, 4, 5, 2, 6, 7, 9, 8, 1, 10$ is algolicious, whereas the sequence $3, 4, 5, 1, 6, 7, 9, 8, 2, 10$ is not (see the improper placement of $1, 2, 5$). Notice that the only improper order is largest-smallest-intermediate. All other combinations (like smallest-largest-intermediate) are proper.

Method 0: Brute force – $O(n^4)$

A direct translation of the above definition is to choose each combination a, b, c of sauces, find their positions j, k, i (as in the picture), check whether $i < j < k$, and if so, discard S as unalgolicious. If all checks discover proper placements, declare S as algolicious. There are $\binom{n}{3} \approx \frac{n^3}{6}$ sauce combinations. Since S is unsorted, you have to make linear search in S to locate the positions i, j, k. Therefore the running time is $O(n^4)$.

Method 1: Brute force, but better – $O(n^3)$

Instead of running the search on a, b, c, let us run the search on i, j, k. For each choice of the indices satisfying $0 \leqslant i < j < k \leqslant n - 1$, take $c = S[i]$, $a = S[j]$, and $b = S[k]$. Then, check whether $a < b < c$. Since every sauce appears at some position in S, this exhausts all sauce combinations. Still, a factor of n is gained.

Method 2: Brute force, more refinement – $O(n^2)$

For each i, set $c = S[i]$. Look at the subsequence of $S[i+1, n-1]$ consisting of numbers $< c$. S is algolicious if and only if for each i, this subsequence is (strictly) decreasing. Therefore for each i, you need to make a single pass through the rest of the sequence, and another factor of n is gained.

Method 3: Good bye brute force – $O(n)$

Welcome to the world of algorithms. Design an $O(n)$-time algorithm yourself.

What to do

Write four functions implementing the above four methods. In the *main* function, read n and the sequence S from the user. Assume that the user supplies an S of valid format (that is, each $i \in [1, n]$ appears once and only once), so you do not have to check the validity of S. Call the four functions, and report their findings.

Sample output

```
n = 25

+++ Sequence: 3 5 8 7 6 12 14 16 15 18 21 25 24 23 22 20 19 17 13 11 10 9 4 2 1
    Method 0: Algolicious
    Method 1: Algolicious
    Method 2: Algolicious
    Method 3: Algolicious

+++ Sequence: 8 19 9 4 25 5 7 2 6 10 11 12 13 14 15 16 17 21 18 3 20 23 24 22 1
    Method 0: Unalgolicious
    Method 1: Unalgolicious
    Method 2: Unalgolicious
    Method 3: Unalgolicious
```

Submit two C/C++ source files. Do not use global/static variables.