COL100 Assignment 2

Due date: 22 December, 2020

General assignment guidelines

1. For each question, the algorithm must be defined mathematically in your written work, as well as implemented in SML code in your programming component.
2. Analysis of correctness and efficiency must be carried out for all nontrivial helper functions as well as for the main function. Any function that performs recursion counts as nontrivial, as does any function that calls a nontrivial function.
3. Big O notation should be as tight as possible. For example, reporting $O(n^2)$ complexity for an $O(n)$ algorithm could lose you marks, even though it is technically correct.
4. For the programming component of each question, the name and the type of the main function must be exactly as specified in the question.
5. Any function that has integer input and output must not perform any intermediate computation using reals.

1 Sum of three primes

It is a famous unsolved problem in number theory to prove whether every integer $n > 5$ can be expressed as the sum of three prime numbers. But we don't need any number theory to simply verify whether the conjecture is true for any specific number. That is, given an integer $n > 5$, we want to find three prime numbers i, j, k, not necessarily distinct, whose sum is equal to n.

For example, if $n = 20$, a possible answer is $(2, 5, 13)$, since $2, 5, 13$ are all prime and $2+5+13 = 20$. Any permutation of these three numbers — or of any other triple of primes that add up to n, such as $(2, 7, 11)$ — would also be acceptable.

1. Design a recursive algorithm to find three primes which add up to a given number $n > 5$, or return $(0, 0, 0)$ if none exist. Prove the correctness of your algorithm.
2. Analyze the time and space complexity of your algorithm. For full marks, your algorithm should take $O(n^{2.5})$ time.

3. Implement your algorithm as a function $\texttt{findPrimes} : \texttt{int} \to \texttt{int} * \texttt{int} * \texttt{int}$.

2 Packing the *dikki*

I have a set of n different items I want to take to the market to sell. Each item has a certain value and a certain weight, both positive integers. Unfortunately, there is a limited amount of weight $W \in \mathbb{N}$ that I can carry in my vehicle. What items should I take to maximize the total value, without the total weight exceeding W? Assume the values and weights are given by two functions $v : \{1, \dots, n\} \to \mathbb{N}$ and $w : \{1, \dots, n\} \to \mathbb{N}$.

For example, suppose $n = 5$ and $W = 100$, with items

$$\begin{aligned} v(1) &= 500, & w(1) &= 40, \\ v(2) &= 700, & w(2) &= 80, \\ v(3) &= 300, & w(3) &= 30, \\ v(4) &= 200, & w(4) &= 50, \\ v(5) &= 300, & w(5) &= 20. \end{aligned}$$

Then the maximum possible value is 1100, obtained by choosing items 1, 3, and 5.

1. Design a recursive algorithm to find the maximum possible value for any set of items (described by n, v, and w) and any weight limit W. Prove the correctness of your algorithm.
2. Analyze the time and space complexity of your algorithm. This is a hard problem in the general case, so you are not expected to get a polynomial-time algorithm; you just have to correctly analyze its complexity.
3. Implement your algorithm as a function $\texttt{maximumValue} : \texttt{int} * (\texttt{int} \to \texttt{int}) * (\texttt{int} \to \texttt{int}) * \texttt{int} \to \texttt{int}$, which can be called as `maximumValue(n, v, w, W)`.

3 Human-friendly units

We often want to convert a raw measurement expressed in one unit into a human-readable form using a combination of large and small units. For example, it is easier to understand a height of 64 inches as “5 feet 4 inches”, and a duration of 10^6 seconds as “11 days 13 hours 46 minutes 40 seconds”. In this format, the number of each unit must be less than the size of the next larger unit (e.g. it would not be valid to say “4 feet 16 inches”, because 16 inches $\geq$ 1 foot).

Suppose you are given two functions specifying the names of the units and their conversion factors, for example

$$\begin{aligned} name(0) &= \text{“seconds”}, & factor(0) &= 60, \\ name(1) &= \text{“minutes”} & factor(1) &= 60, \\ name(2) &= \text{“hours”} & factor(2) &= 24, \\ name(3) &= \text{“days”} & factor(3) &= 365, \\ name(4) &= \text{“years”}, & factor(4) &= 0. \end{aligned}$$

Interpret a factor of zero to mean that there are no higher units to convert to. With these functions, for example, an input of 3602 should be converted to “1 hours 0 minutes 2 seconds” or “1 hours 2 seconds” (either is acceptable).

1. Design a recursive algorithm to convert an integer n of the smallest unit to a human-readable string using the *name* and *factor* functions. You may also need to design a *toString* function to convert a natural number to a string, e.g. $toString(64) =$ “64”.
2. Design an iterative algorithm for the same problem, which requires only $O(1)$ deferred computations / stack frames (no matter how many units are needed). Prove this fact.
3. Analyze the time complexity of your algorithms, assuming that: (i) $size(name(n)) = O(1)$ and $factor(n) = O(1)$, and (ii) the cost of string concatenation ($\wedge$) is $O(n + m)$ where n and m are the sizes of the two strings being joined.
4. Implement both algorithms as functions `convertUnitsRec` and `convertUnitsIter` of type `int` $*$ (`int` $\to$ `string`) $*$ (`int` $\to$ `int`) $\to$ `string`. Use your own `toString` function, not the built-in function `Int.toString`.

4 Iterative integer square root

Recall that the integer square root of a natural number n is the unique natural number k such that

$$k^2 \leq n < (k+1)^2.$$

In the lectures, we have discussed a fast recursive function for this problem, namely *intSqrt2*, which computes the integer square root of n in $O(\log n)$ time. However, it also requires $O(\log n)$ space to do so.

1. Design a fast iterative algorithm for this problem, which has a time complexity of $O(\log n)$, but a space complexity of only $O(1)$. Prove the correctness of your algorithm.

   Hint: Consider the invariant $i^2 \leq \lfloor n/4^p \rfloor < (i+1)^2$, for some natural numbers i and p.
2. Prove the $O(\log n)$ time complexity and $O(1)$ space complexity of your algorithm.

3. Implement your algorithm as a function `intSqrt` : `int` → `int`. Verify that it succeeds quickly even for large numbers, like 400,000,000.

5 Submission and other logistics

As usual, you should submit two files to the Moodle assignment page.

1. One file should be a PDF file containing all your written work, i.e. mathematical definitions of algorithms, correctness and efficiency analysis, etc. This can be handwritten with pen and paper and then scanned with your mobile, or it can be typed on your device using MS Word or LaTeX or any other software. The only requirement is that it should be clearly legible.
2. The other file should be an SML file containing all your code. Put your solutions to all four programming problems into a single file, along with any helper functions they require. We should be able to run any of the four required functions by typing in a function call at the bottom of the file.

The filenames of your submitted files should be exactly the same as your entry number. For example, if your entry number is 2017CS10389, you should name your SML file as `2017CS10389.sml` and your PDF file as `2017CS10389.pdf`. Your submission should consist of only these two files, uploaded individually. There should be no sub-folders or zip files in the submission on Moodle.

Failure to comply with these submission instructions will lead to a penalty.

Please ask any queries related to the assignment on the COL100 Piazza page (`https://piazza.com/iit_delhi/fall2020/col100`).