COL100 Assignment 1

2020CS10341

22 December 2020

1 Sum of three primes

1.1 Algorithm

Our main function is `findPrimes(n)` which returns a tuple `(a,b,c)` such that a, b, c are Prime and $a+b+c = n$. We check whether a tuple $(a, b, n-a-b)$ with the condition $a \leq b \leq n-a-b$ consists of primes only. First, we check whether a is a prime. If it is then we check whether b and $n-a-b$ are primes as well. If they are primes then we return $(a, b, n-a-b)$ as output.

If a is not a prime then we set (a, b) as $(a+1, a+1)$ and proceed till we find a prime $a' > a$.

If b and $n-a-b$ are not both primes then we increment b by 1 till it exceeds $n-a-b$ since checking for (a, b, c) and (a, c, b) is the same thing. So, we avoid double counting by ensuring that the condition $a \leq b \leq n-a-b$ holds. If b exceeds $n-a-b$ then we set (a, b) as $(a+1, a+1)$ and proceed.

We only check for such a tuple till $3a > n$ and we return $(0, 0, 0)$ as the output when this happens as after that there won't be any solutions since equality holds for every inequality in $a \leq b \leq n-a-b$ when $3a = n$ and for bigger values of a the right inequality would not hold.

Now, for checking whether $a, b, n-a-b$ are Primes we take the help of the helper function `isPrime(n)`. This function checks whether a given number is a prime by returning *true* if n is 2, and otherwise it checks by dividing n by all $i \geq 2$ and $i^2 \leq n$. If the number n is not divisible by all such i i.e $n \bmod i \neq 0$ then n is a prime as if it would have a divisor $2 \leq d < \sqrt{n}$ then it would also have a divisor $d' > \sqrt{n}$.

```
fun isPrime(n) = let fun f(a) = if n < a*a then true
else if n mod a = 0 then false else f(a+1) in f(2)
end;
fun findPrimes(n) = let fun f(a,b) = if a*3 > n then
(0,0,0) else if isPrime(a) then if isPrime(b) andalso
isPrime(n-a-b) then (a,b,n-a-b) else if b > n-a-b
then f(a+1,a+1) else f(a,b+1) else f(a+1,a+1) in
f(2,2) end;
findPrimes(998899998);
```

```
> val isPrime = fn: int → bool;
> val findPrimes = fn: int → int * int * int;
> val it = (2, 89, 998899907): int * int * int;
```

1.2 Time and Space Complexity

1.2.1 Time Complexity

Since we only check whether a number is prime by diving it by at most $\lfloor\sqrt{n}\rfloor - 1$ values $(2, 3, ..., \lfloor\sqrt{n}\rfloor)$, the time complexity of `isPrime(n)` is $O(\sqrt{n})$.

Since $a \leq b \leq n-a-b < n$, it would require at most $3\times O(\sqrt{n}) = O(\sqrt{n})$ time to check whether $(a, b, n-a-b)$ consists of primes, For a given a, we check at most $\left\lfloor \frac{n-a}{2} \right\rfloor$ values of b (till it exceeds $n-a-b$), and since we check a for at most $\left\lfloor \frac{n}{3} \right\rfloor - 1$ values $(2, 3, ..., \left\lfloor \frac{n}{3} \right\rfloor)$, we are checking at most

$$\sum_{a=2}^{\lfloor \frac{n}{3} \rfloor} \left\lfloor \frac{n-a}{2} \right\rfloor \leq \sum_{a=1}^{(\frac{n}{3})} \frac{n}{2} = \frac{n^2}{6}$$

values and for checking each value we take $O(\sqrt{n})$ time. So, total time taken is simply

$$\frac{n^2}{6} \times O(\sqrt{n}) = O(n^{2.5})$$

and that is the time complexity of the main function `findPrimes(n)`.

1.2.2 Space Complexity

Since the helper function `f(a)` defined in `isPrime(n)` recursively calls itself after every step when it doesn't output `false`, it only uses one stack frame and hence it's space complexity is simply $O(1)$. Similarly, the helper function `f(a,b)` defined in `findPrimes(n)` also recursively calls itself and doesn't use any additional stack frames at any step till it gives an output and hence it's space complexity is also $O(1)$.

1.3 Proof of Correctness

Function `isPrime(n)` returns `true` as output for `n=2`. For higher values of `n` the helper function `f(a)` checks whether `n` is divisible by `a` for all $2 \leq$ `a` $\leq \lfloor\sqrt{n}\rfloor$ and returns true when it does not find any value of `a` for which `n mod a` is `0` and returns `false` otherwise. So `isPrime(n)` returns `true or false` as output whenever `n` is a *prime or not* respectively.

Now `f(a,b)` defined in `findPrimes(n)` starts checking for tuples $(a, b, n-a-b)$ and ensures $a \leq b \leq n-a-b$ while doing so as it always starts with $a = b$ and then increments b ensuring $a \leq b$. Now whenever $b > n-a-b$, `f(a,b)` calls `f(a+1,a+1)` ensuring $b \leq n-a-b$. This ensures that we are checking for all possible ordered pairs (a, b, c) such that $a + b + c = n$ and $a \leq b \leq c$. Now the function `f(a,b)` checks whether a, b and $(n-a-b)$ are Primes before returning them as output as `(a,b,n-a-b)` and the function returns `(0,0,0)` as output only when it finds no such ordered pairs after checking for each one of them.

2 Packing the *dikki*

2.1 Algorithm

The helper function `MAX(a,b)` returns the greater number between `(a,b)`. `maximumValue(n,v,w,W)` is basically same as the helper function `iter(n,W)`. The function `iter(n,x)` represents the maximum value that can be carried by using `n` items and maximum weight limit `x`. So, `iter(0,x)=0` and `iter(n,x)=iter(n-1,x)` when weight of the n^{th} item exceeds total weight limit x. For a general case, *iter(n,x)* which represents the maximum value that can be carried without exceeding weight limit x can be thought of as the maximum value that can be achieved without including the n^{th} item i.e $iter(n-1, x)$ or by including the n^{th} item i.e the maximum value for weight $x-w(n)$ and the value of the n^{th} item $v(n)$ combined. So, `iter(n,x)=MAX(iter(n-1,x),iter(n-1,x-w(n))+v(n))` and this way the algorithm is a recursive one as the `iter` function calls itself in most of the steps.

```
fun MAX(a,b)= if a>=b then a else b;
fun maximumValue(n,v,w,W)= let fun iter(0,x) = 0
    |iter(n,x) = if w(n) > x then iter(n-1,x) else
MAX(iter(n-1,x-w(n))+v(n),iter(n-1,x)) in iter(n,W)
end;
fun value(1) = 500 |value(2) = 700 |value(3) = 300
|value(4) = 200 |value(5) = 300 |value(n) = 0;
fun weights(1) = 40 |weights(2) = 80 |weights(3) = 30
|weights(4) = 50 |weights(5) = 20 |weights(n) = 0;
maximumValue(5,value,weights,100);
```

```
> val MAX = fn: int * int → int;
> val maximumValue = fn: int * (int → int) * (int → int) * int → int;
> val value = fn: int → int;
> val weights = fn: int → int;
> val it = 1100: int;
```

2.2 Time and Space Complexities

2.2.1 Time Complexity

The function `iter(n,x)` calls itself at most twice recursively. Then it performs a comparison using the `MAX` function. Let *T(n)* be the time complexity of the `iter(n,x)` function. In the worst case scenario, the total weight will be sufficiently large and the `iter` function will call itself twice at every step. Suppose every comparison takes a constant time c. So, we get the relation

$$T(n) = 2T(n-1) + c$$

Now, we can compute $T(n)$ as follows:

$$\begin{aligned} T(n) &= 2T(n-1) + c \\ 2T(n-1) &= 4T(n-2) + 2c \\ 4T(n-2) &= 8T(n-3) + 4c \\ &. \\ &. \\ &. \\ 2^{n-1}T(n-2) &= 2^nT(0) + 2^{n-1}c \end{aligned}$$

$$2^{n-1}T(n-2) = 2^nT(0) + 2^{n-1}c$$

Summing both sides we get:

$$T(n) = 2^nT(0) + (2^n - 1)c$$

Since, $T(0)$ takes constant time, we get that the time complexity of the `iter` function (and hence the main function) is $O(2^n)$.

2.2.2 Space Complexity

At every step (assuming that `iter` calls itself twice at each step), the function performs a comparison (or a computation) that involves two values. For this comparison, the algorithm first calculates the first value and then the second value separately. Thus at every level, the function defers computation and gets to the lower level. Since the total depth of the levels in this manner is $n + 1$ (from `iter(n,`x_n`)` to `iter(0,`x_0`)`), the space complexity of the algorithm is $O(n)$.

3 Human-friendly units

3.1 Algorithm

The function `toString(n)` takes an integer `n` as input and returns a string `"n"` as output. It has been defined iteratively in terms of `(n div 10)`,`(n mod 10)` and 10 base cases for each digit. `toString(n)` is simply `toString(n div 10)^toString(n mod 10)` for `n>10`.

`convertUnitsRec(a,n,f)` and `convertUnitsIter(a,n,f)` are the required recursive and iterative functions respectively. These functions have been defined with the help of the helper function `alpha` that attaches the name and the corresponding number we get after getting the remainder by dividing it by the corresponding factor. `n(y)` and `f(y)` represent the name and the factor corresponding to `y` respectively and are the arguments of the main function.

Helper function `namer(x)` attaches the corresponding name i.e `name(y)` behind `x` for non zero x. When `x=0`, it simply returns `""`, i.e the empty string. In the recursive version, `alpha(x,y)` checks whether `f(y+1) = 0` or not. If `f(y+1)=0`, that means this is the last factor we have to deal with it and `alpha(x,y)` simply returns `namer(x)`, otherwise, it divides x by $f(y)$ and calls itself recursively by returning `alpha(x div f(y),y+1) ^ namer(x mod f(y))`. So, it divides the given number x by the corresponding factor at that stage i.e $f(y)$ and then attaches the corresponding name to the *remainder* and proceeds to process the *quotient*. The iterative version is not much different, with the only difference being an additional parameter in the function `alpha` i.e `c` which initially starts with `""` and then stores the output at every step until the last step. In the last step, `alpha(x,y,c)` attaches `namer(x)` before `c` and returns the resulting string as output.

```
fun toString(0) = "0"|toString(1) = "1"|toString(2) = "2"|toString(3) = "3"|toString(4) = "4"|toString(5) = "5"|toString(6) = "6"|toString(7) = "7"|toString(8) = "8"|toString(9) = "9"
|toString(n) = let fun iter(x,c) = if x = 0 then c else iter(x div 10,toString(x mod 10)^c) in iter(n,"") end;
fun convertUnitsRec(a,n,f) = let fun alpha(x,y) = let fun namer(x) = if x = 0 then "" else " "^toString(x)^" "^n(y) in if f(y+1) = 0 then namer(x) else alpha(x div f(y),y+1)^namer(x mod f(y)) end in alpha(a,0) end;
fun convertUnitsIter(a,n,f) = let fun alpha(x,y,c) = let fun namer(x) = if x = 0 then "" else " "^toString(x)^" "^n(y) in if f(y+1) = 0 then namer(x)^c else alpha(x div f(y),y+1,namer(x mod f(y))^c) end in alpha(a,0,"") end;
fun name(0) = "seconds" |name(1) = "minutes" |name(2) = "hours" |name(3) = "days" |name(4) = "years" |name(n) = "";
fun factor(0) = 60 |factor(1) = 60 |factor(2) = 24 |factor(3) = 365 |factor(n) = 0;
convertUnitsRec(21734687,name,factor);
convertUnitsIter(21734687-13*60*60,name,factor);
```

```
> val toString = fn: int → string;
> val convertUnitsRec = fn: int * (int → string) * (int → int) → string;
> val convertUnitsIter = fn: int * (int → string) * (int → int) → string;
> val name = fn: int → string;
> val factor = fn: int → int;
> val it = " 251 days 13 hours 24 minutes 47 seconds": string;
> val it = " 251 days 24 minutes 47 seconds": string;
```

3.2 Space Complexity of the iterative algorithm

The function `toString(n)` is an iterative process that stores only 1 value for `n>9` hence it's space complexity is $O(1)$. Function `convertUnitsIter` has the same space complexity as the helper function `alpha`. The helper function `namer(x)` has 2 stack frames/ deferred computations i.e one for `toString(x)` and other for `n(y)`. Since both these functions have $O(1)$ space complexity, the space complexity of `namer(x)` is $2 \times O(1) = O(1)$. Now, since `alpha(x,y,c)` doesn't use any additional stack frames for computations and only uses 3 parameters till the last step where it computes `namer(x)` before attaching it with `c`, it also has $O(1)$ space complexity.

3.3 Time Complexities

`toString(n)` takes $\lfloor log_{10}(n) \rfloor + 1$ steps to reach one of the 10 base cases. Since we are catenating a string of length 1 to a string of length k at each step where k varies from $(1, 2, ..., \lfloor log_{10}(n) \rfloor)$, the number of steps is

$$\sum_{k=1}^{\lfloor log_{10}(n) \rfloor} (1+k) = \frac{\lfloor log_{10}(n) \rfloor \lfloor log_{10}(n) + 3 \rfloor}{2}$$

So, time complexity of `toString(n)` is $O((log(n))^2)$. Time taken to concatenate two strings is time taken to find out what those two strings are and an additional time of $O(n+m)$ for the concatenation. For the function `namer(x)`, the time complexity is $O((log(x))^2) + O(log(x)+c) = O((log(x))^2 + c)$ where c is a constant depending on the maximum length of `n(y)`. Suppose there are p factors. So, the function `namer(x)` is called at most kp times. For $p = 2$, we are computing `(namer(((x div f(0)) div f(1)) mod f(2))^namer((x div f(0)) mod f(1)))^namer(x mod f(0))`. Let x_k represent `(....((x div f(0) div f(1))...... div f(k-1))`. So, in the end we are computing

$$(.....(namer(x_p)^\wedge namer(x_{p-1} \; mod \; f(p-1))).....namer(x \; mod \; f(0)))$$

in the recursive algorithm and

$$(namer(x_p)^\wedge(namer(x_{p-1} \; mod \; f(p-1))).....(namer(x_1 \; mod \; f(1))^\wedge namer(x \; mod \; f(0))).....)$$

in the iterative algorithm. (Folded from left in the recursive function and from the right in the iterative function.)

Now concatenating in this manner to give a string of length l will takes less steps than concatenating l individual characters to give the same string. Now, concatenating a string of length l in this manner will take $O(l^2)$ steps. This can be proved similarly to how we found out the time complexity of `toString` function. The resulting string will be of length of $O(log(x)+p)$ and hence the time complexity of the concatenation will be $O((log(x)+p)^2)$. Additionally, we require some time whenever `namer` is called. This time is simply $\sum_{i=0}^{p}((log(y_i))^2 + c)$ where y_i is the input given to `namer` function at different steps. Now, $y_i < max(f(x))$ for $i = (0, 1, ..., p-1)$ as `(a mod b)<b` and $y_p < x$. So, our summation $\sum_{i=0}^{p}((log(y_i))^2 + c)$ is of the order $O(p \cdot (log(max(f(x))))^2 + (log(x))^2)$. So, our final time complexity is simply $O((log(x)+p)^2) + O(p \cdot (log(max(f(x))))^2 + (log(x))^2)$ or $O((log(x)+p)^2)$. This can also be written as $O(l^2)$ where l represents the length of the resulting string.

4 Iterative integer square root

4.1 Algorithm and Proof of its Correctness

4.1.1 Algorithm

`intsqrt(n)` is our main function. The helper function `alpha` calculates the highest number b of the form 4^x such that $4^x \leq n < 4^{x+1}$. Another helper function `iter(x,z)` has been defined which stores the integer square root of `n div z` in the form of `x`. `z` varies from `b` to `1`, getting divided by `4` at each step. When `z=1`, `x` represents the integer square root of `n div z` = `n` and the function `iter(x,1)` returns `1` as output.

4.1.2 Proof of Correctness

We want prove that for the function `iter(x,z)`,`x` represents the integer square root of `n div z` by induction. *To Prove*: `x` represents the integer square root of `n div z` *Base Case*: `1` represents the integer square root of `n div b` Initially, we start with `iter(1,b)`. Since $b \leq n < 4b$

$$1 \leq \left\lfloor \frac{n}{b} \right\rfloor < 4$$

And hence `1` represents the integer square root of `n div b` and the base case of our induction hypothesis is true.

Induction Hypothesis: x_k represents the integer square root of $n\ div\ z_k$ for some (x_k, z_k). Now, we show that x_{k+1} represents the integer square root of z_{k+1}. We have $z_{k+1} = z_k\ div\ 4$ and

$$(x_k)^2 \leq \left\lfloor \frac{n}{z_k} \right\rfloor < (x_k + 1)^2$$

$$(2x_k)^2 \leq 4 \left\lfloor \frac{n}{z_k} \right\rfloor < (2x_k + 2)^2$$

$$(2x_k)^2 \leq 4 \left\lfloor \frac{n}{z_k} \right\rfloor \leq \left\lfloor \frac{4n}{z_k} \right\rfloor = \left\lfloor \frac{n}{z_{k+1}} \right\rfloor < (2x_k + 2)^2$$

Now by definition, integer square root of $\left\lfloor \frac{n}{z_{k+1}} \right\rfloor$ is either $2x_k$ or $2x_k + 1$. Now, `iter` function checks whether $(2x_k + 1)^2 > \left\lfloor \frac{n}{z_{k+1}} \right\rfloor$. If it is then it sets $x_{k+1} = 2x_k$. Else, it sets $x_{k+1} = 2x_k + 1$. Thus, x_{k+1} represents the integer square root of `n div z`$_{k+1}$ and by Principle of Mathematical Induction, it follows that `iter(1,b)` and hence `intsqrt(n)`

```
fun intsqrt(n) = let val b = let fun alpha(b) = if b > n div 4 then b
else alpha(4*b) in alpha(1) end in let fun iter(x,z) = if z = 1 then x
else if (2*x+1)*(2*x+1) > n div (z div 4) then iter(2*x,z div 4) else
iter(2*x + 1,z div 4) in iter(1,b) end end;
intsqrt(472364567);
```

```
> val intsqrt = fn: int → int;
> val it = 21733: int;
```

4.2 Time and Space Complexities

4.2.1 Time Complexity

It takes $\lfloor log_4 n \rfloor + 1$ steps to calculate `b` and a further $\lfloor log_4 n \rfloor + 1$ steps to reach `z=1`. Hence, the time complexity is $O(log(n))$.

4.2.2 Space Complexity

The function `alpha` doesn't defer any computation while calculating for `b` hence it is of $O(1)$ space complexity. The function `iter` performs same number of calculations in each step and it recursively calls itself without deferring any computations.

$$\texttt{iter(x,z)} = \begin{cases} \texttt{x,} & \texttt{if z=1} \\ \texttt{iter(2x,z div 4),} & \texttt{(2x+1)*(2x+1) > n div z} \\ \texttt{iter(2x+1,z div 4)} & \texttt{otherwise} \end{cases}$$

Hence, it also has $O(1)$ space complexity.