COL100 Assignment 3

2020CS10341

6 January 2021

1 Operations on very large numbers

1.1 Algorithm

Following are the *algorithms* for different functions:

1.1.1 Main functions

LgintToInt

```
fun LgintToInt(X)=
  if length(X)>9 then 1000000000
  else
    let
      fun iter([],y,z) = y
        |iter(x::xs,y,z) = iter(xs,(x*z) +y,10*z)
    in iter(X,0,1) end
```

The function `LgintToInt` converts a number in list format into its equivalent integer by maintaining a sub-list and output iteratively. It takes out the first element of the list, multiplies the next element by 10, then adds it, multiplies the next element by 100, then adds it and so on till the list becomes empty. Then it outputs the stored output at the final step (when the list becomes empty).

intToLgint

The function `intToLgint` converts the given integer number into the list format by appending the last digit `(n mod 10)` of the given number onto the front of the list obtained from the remaining digits by recursively calling itself to calculate the list form of the remaining digits. It terminates by returning a single digit as a list containing only that digit.

```
8  fun intToLgint(n)=
9    if n div 10 = 0 then [n]
10   else (n mod 10)::intToLgint(n div 10)
```

addLgint

```
11 fun addLgint(L1,L2) =
12   let fun iter([],[],0) = []
13     |iter([],[],c) = [c]
14     |iter([],x::xs,c) = ((x+c) mod 10)::iter([],xs,((x+c) div 10))
15     |iter(y,[],c) = iter([],y,c)
16     |iter(x::xs,y::ys,c) = ((y+x+c) mod 10)::iter(xs,ys,((y+x+c) div 10))
17   in iter(L1,L2,0) end
```

The function `addLgint` by using `iter` which sums up the input lists by taking out their first digits and then adding them. It also maintains a parameter that serves as *carry* just like carrying in conventional addition, then adds the carried part to the first elements of the respective lists and then stores the output iteratively that it returns at the last step. `addLgint(L1,L2) = iter(L1,L2,0)`

LgLesseq

```
18 fun LgLesseq(L1,L2) =
19   let fun iter([],[],c) = c
20     |iter([],y,c) = true
21     |iter(x,[],c) = false
22     |iter(x::xs,y::ys,c) =
23       if x=y then iter(xs,ys,c)
24       else iter(xs,ys,x<y)
25   in iter(L1,L2,true) end
```

`LgLesseq(L1, L2)` returns the Boolean value of the expression $l1 \leq l2$ where $l1, l2$ are the integers represented by `L1` and `L2`. It stores the value of the comparison for the last digit and then compares the next digit.

If the next digit is larger in value for the first list, then it stores false. If it is smaller for the first list, then it stores true. Otherwise, it does not change the stored value. This way, it stores only the comparison for the more significant digit at each step when the digits are unequal. The function initially starts with `true` so that if all the remaining digits turn out to be equal, the function will return `true` as the output.

1.1.2 Bonus

multiplyLgint

```
26 fun multiplyLgint(L1,L2) =
27   let fun multiply1(L,x) =
28     let fun iter([],0) = []
29       |iter([],c) = (c mod 10)::iter([], c div 10)
30       |iter(x::xs,c) = ((x+c) mod 10)::iter(xs, (x+c) div 10)
31     fun mapx([]) = []
32         |mapx(y::ys) = (x*y)::mapx(ys)
33     in iter(mapx(L),0) end
34   in
35     let fun iter(x,[],c) = c
36     |iter(x,y::ys,c) = iter(0::x,ys,addLgint(multiply1(x,y),c))
37     in iter(L1,L2,[]) end
38   end;
```

`multiply1(L,x)` multiplies the list `L` by `x` and gives the output. It does this with the help of `map(x)` function which multiplies each element of `L` by `x` and gives out a list. `iter` function just converts each element of this list into a single-digit integer to give the list of the desired output. `[56]` will be converted to `[6,5]`, for example. Then `multiplyLgint` uses the `iter` function to multiply the first list by the first element of the 2nd list, multiplies the first list by 10 (by appending `0`), and then multiplies the 2nd element of the 2nd element to this new list and adds it to the list obtained in the previous step and so on.

1.2 Correctness and Complexities

1.1.1 Correctness Proofs

Following are the *Correctness Proofs* for different functions:

LgintToInt

`LgintToInt` returns the integer corresponding to the given list. It returns `0` as output for a list of length 0 (null list). Assume that it gives the correct output for a list of length n. We need to show that it gives the correct output for a list of size $n+1$. We have a list of size $n+1$. The function takes out the first n elements of the list and stores the equivalent integer as the output based on our assumption. For the last item of the list, the function multiplies it by $z = 10^n$ (as z is getting multiplied by 10 at each step and there are total n additional steps till the `iter` function reaches the last element of the list) and then adds it to the equivalent

integer of the first n digits. Hence by *Principle of Mathematical Induction*, `LgintToInt` gives the correct output for lists of all sizes.

intToLgint

`intToLgint` $\equiv$ `iter` converts the given integer to the list format. For a single-digit number as input, the function returns a list containing only one element; that element is the digit itself. Now assume that the function gives the correct output for an integer consisting of k digits. We need to show that it gives the correct output for an integer consisting of $k+1$ digits. We have a $(k+1)$-digit integer. The function appends `n mod 10` (the last digit) to `intToLgint(n div 10)`. Since `n div 10` is a k-digit integer, `intToLgint(n div 10)` returns the list corresponding to the remaining digits based on our assumption. The required list is the last digit appended to the front of this list. Hence by *Principle of Mathematical Induction*, `intToLgint` returns the correct list for integers for all positive integers.

addLgint

`addLgint(L1,L2)` $\equiv$ `iter(L1,L2,0)` adds 2 integers in list format. For any two lists, the first element of the resulting list is the remainder obtained by dividing the sum of each list's first two elements respectively by 10, which represents the last digit of the sum of the two numbers in list format. Please assume that the function `iter` stores the correct digits up to the k^{th} digit from the last. We need to show that the rest of the digits are correct.

The $(k+1)^{th}$ digit of the sum of two integers is the sum of the $(k+1)^{th}$ digits of the two numbers being added, plus the value of the carried part, which is the $(k+1)^{th}$ digit obtained by adding the last k digits of both these numbers.

At every step, we are reducing the size of non-empty lists by 1. At the $(k+1)^{th}$ step, The head of the list `x::xs` (i.e., x) represents the $(k+1)^{th}$ digit of that number. The parameter `c` is the carried part from the previous sum, i.e., the sum of the last k digits of the two numbers.

So, our $(k+1)^{th}$ digit is `(x+y+c) mod 10` (where `x,y` are the $(k+1)^{th}$ digits of the two numbers in list format). In case one of the numbers does not have $(k+1)^{th}$ digit, we only add `(x+c)`. Moreover, when both the number do not have the $(k+1)^{th}$ digit, the function returns `[c]`, i.e., the carried part from the previous addition step. Thus, the function gives the correct value for the $(k+1)^{th}$ digit.

Hence, by *Principle of Mathematical Induction*, the function `addLgint(L1,L2)` gives the correct value for all the digits and terminates as the lists' size decrease at each step.

LgLesseq

`LgLesseq(L1,L2)` $\equiv$ `iter(L1,L2,true)` gives the Boolean expression $l1 \leq l2$ as output, where $l1, l2$ are the integers represented by L1, L2 respectively. For the last digits of these two numbers (x_1, y_1 respectively), the function stores $x_1 \leq y_1$ in the iteration (`true` for $x_1 = y_1$ and $x_1 < y_1$ otherwise). Please assume that the function `iter` stores the correct output up to the k^{th} digit from the last. We need to show that it shows the correct output for the remaining digits.

In case both the $(k+1)^{th}$ digits are equal (or do not exist for both the numbers), then the stored value tells us which the bigger number is. We know that if $(k+1)^{th}$ digits of both the numbers are unequal, the number with the bigger digit is bigger. If a number does not have the $(k+1)^{th}$ digit and the other has it, this number is smaller. The `iter` function deals with all the cases and stores the output for the next step accordingly, and it stores the correct output up to $(k+1)^{th}$ digit from the last.

Hence, by *Principle of Mathematical Induction*, `LgLesseq(L1,L2)` gives the correct values for all lists of all sizes and terminates as the lists' size decreases at each step.

multiplyLgint

`mapx`'s correctness is trivially correct and can be proved using *PMI*. `iter` function in `multiply1` can be thought of as addition of a null list and another list. Since we have already proved the correctness of `addLgint`,

`iter`, and thus `multiply1` can be proved correctly in a similar manner. We have already proved the correctness of `addLgint`, which is used repeatedly. We are just performing iterations on two lists, taking one element at a time from the second list, and adding`0` to the first list. Multiplication of two integers can be done in the same manner, by multiplying the first integer by the last digit, multiplying the first integer by the second last digit and adding a 0 at the last. This is same as the common method of multiplication by hand.

1.2.2 Complexities

Following are the *Complexities* for different functions:

LgintToInt

`LgintToInt(X)` has a time and space complexity of $O(n)$, where n is the list's length. At every step, `iter` calls itself recursively and reduces the length of the list by 1. It performs the same number of calculations at each step and does not use any additional stack frames. Since it takes n steps in total and only one stack frame, the time complexity is $O(n)$, and the space complexity is solely due to the list. Since the length of the list is n, space complexity is $O(n)$ as well.

intToLgint

`intToLgint(n)` has a time and space complexity of $O(log(n))$. At every step, the function calls itself and generates a new stack frame as it adds the item n mod 10 at each step to the list generated. It performs the same number of computations at each step. The function terminates as n becomes less than 10 on repeated division by 10 at the $\lfloor logn + 1 \rfloor^{th}$ step. Thus, the function has a time complexity of $O(log(n))$ as it takes a total $O(log(n))$ steps and the same space complexity as it generates $O(log(n))$ stack frames and a list of size $O(log(n))$ while computing the deferred computations.

addLgint

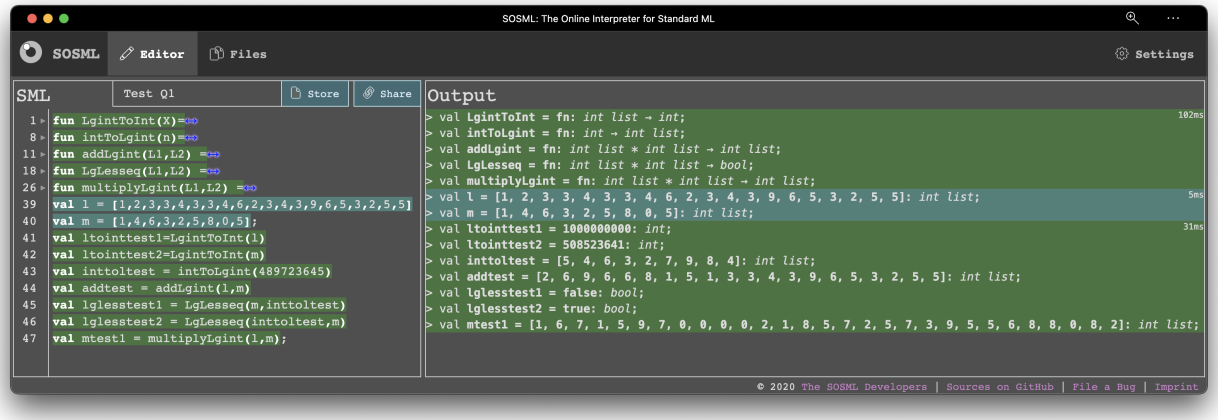
`addLgint(L1,L2)` has a time and space complexity of $O(n)$, where n is the maximum of the lengths of L1 and L2. At every step, the `iter` function calls itself and generates a new stack frame as it adds an item at each step to the list generated by `iter(xs,ys, some parameter)`. It performs the same number of computations at each step. The lists' lengths are decreasing by 1 at each step, and at the $(n+1)^{th}$ step, both of them become empty. The algorithm terminates at this step. Thus, the function has a time complexity of $O(n)$ as it takes a total $O(n)$ steps and the same space complexity as it generates $O(n)$ stack frames and a list of size $O(n)$ while computing the deferred computations.

LgLesseq

`LgLesseq(L1,L2)` has a time and space complexity of $O(n)$, where n is the minimum of the lengths of L1 and L2, and space complexity of $O(n')$, where n' is the maximum of the lengths of L1 and L2. The `iter` function performs the same number of calculations at each step and does not use any additional stack frames. The lists' lengths are decreasing by `1` at each step, and at the $(n+1)^{th}$ step, at least one of them becomes empty. The algorithm terminates at this step. Thus, the function has a time complexity of $O(n)$ as it takes a total $O(n)$ steps and the space complexity $O(n')$ as it uses lists of length $O(n')$ while not generating any additional stack frames.

multiplyLgint

Time complexity is $O(nm)$, where n and m are the lengths of the two lists. It's space complexity is $O(n+m)$ Let n, m be the lengths of the first and second lists respectively. `multiply(L1,x)` is the takes $O(n)$ time and space as both `mapx` and `iter` functions have the same complexities. Now, the main `iter` function takes m steps to reach the end of the list, and each step takes $O(n)$ time. So, the final time complexity is $O(nm)$. `iter` function does not generate any addidtional stack frames, and its space complexity is only due to the size of the lists. Thus, the final space complexity of `multiplyLgint` is $O(n+m)$.



2 Quarterly Performance

2.1 Algorithms

Following are the *algorithms* for different functions:

qPerformance

```
fun qPerformance(L) =
  let val (q1,q2,q3,q4,q5) =
    let fun sum((a1,a2,a3,a4,a5),(b1,b2,b3,b4,b5)) = (a1+b1,a2+b2,a3+b3,a4+b4,a5+b5)
    in foldl sum (0,0,0,0,0) L end
  val leng = length(L)
  fun hike(x1,x2,x3,x4,x5) =
    let fun q(1) = q1 |q(2) = q2 |q(3) = q3 |q(4) = q4 |q(x) = 0
    fun x(1) = x1 |x(2) = x2 |x(3) = x3 |x(4) = x4 |x(n) = 0
      in
        let fun qhike(n) =
          let fun iter(l,m) = if 10*leng*(x(n)) < l*(q(n)) then m else iter(l+1,m+1)
          in iter(11,0) end
        in (x5*(qhike(1)+qhike(2)+qhike(3)+qhike(4)+100)) div 100 end
      end
  in map hike L end
```

`qPerformance`(L) ≡ map hike L outputs a list of hiked salaries of all employees based on the input list. The function hike takes an element of the list (which is in the form of a 5-tuple) and outputs the incremented salary. It uses helper functions q and x for the same. q(x) = qx for x = 1,2,3,4. qx represents the sum of performance points of the xth quarter, calculated using `foldl sum (0,0,0,0,0) L`. sum function takes two 5-tuples as input and gives a 5-tuple as output. Each element of this 5-tuple is the sum of the respective elements of the input 5-tuples. `foldl sum (0,0,0,0,0) L` gives a 5-tuple with each element of the tuple being

the summation of those respective elements in each 5-tuple of the input list. x(n) is simply the n-th element of the 5-tuple input in hike for n = 1,2,3,4. `hike` ≡ `(x5*(qhike(1)+qhike(2)+qhike(3)+qhike(4)+100)) div 100` gives the hiked salary, where `qhike(i)` represents the percentage hike in salary based on ith quarter, and x5 represents the 5th element of the input 5-tuple, which denotes the base salary. `qhike(n)` ≡ `iter(11,0)`. `iter(l,m)` outputs `m` when the condition `10*leng*(x(n)) < l*(q(n))` is met. This condition is equivalent to the condition $x(n) < \frac{l}{10} \times \frac{q(n)}{leng}$. Now $\frac{q(n)}{leng}$ represents the average performance of the n-th quarter, and $\frac{l}{10}$ is initially $1.1 = 110\%$ and increases in increments of $0.1 = 10\%$. `m` is initially set to `0`. So, if the performance point of the employee for n^{th} quarter $()x(n))$ is less than 110% of the average performance points, salary increment is 0% else `l` and `m` increase by `1`, and the `iter` function checks whether the salary is less than 120% of the average and returns `m=1` when this happens. Else, it increases `l` and `m` until it finds a suitable value for `l` for which the inequality holds true, and we get the salary hike for that quarter to be $m\%$.

budgetRaise

```
54 fun budgetRaise(L) =
55   let fun sum1(a,b) = a+b
56     fun sum2((a1,a2,a3,a4,a5),b) = a5+b
57     val x = foldl sum2 0 L
58     val y = foldl sum1 0 (qPerformance(L))
59   in (real(y)/real(x))-1.0 end;
```

`budgetRaise` takes a list as input and outputs the hike in net salary. The hike in salary is $(s_{new})/(s_{old}) - 1$ where s_{new} and s_{old} represent the summation of new and old salaries respectively. `real(y)` and `real(x)` represent s_{new} and s_{old} in our SML programme. `y` is calculated using `qPerformance` function defined above and summing all the list elements using `foldl` and `sum1`. `sum1` outputs the sum of two input integers. We sum all respective elements of all 5-tuples. 5th element of each tuple represents the salary, and the 5-th element of the resulting 5-tuple (x) will denote the sum of all old salaries.

2.2 Correctness and Complexities

2.2.1 Correctness Proofs

qPerformance

It is trivial that the function sum used for calculating `q(i) (i = 1,2,3,4)`, gives the correct sum for two 5-tuples. `foldl` is a predefined function. So, `foldl sum(0,0,0,0,0) L` outputs a 5-tuple with each element of the tuple being the sum of the respective elements of the 5-tuples in list L. `qhike(n)` represents the hike in the n-th quarter. `qhike(n)` returns 0 as the output if $x(n) \leq (110\%) \times q(n)$. Assume that `qhike` returns the correct output "k" for $(100 + 10k)\% < \frac{x(n)}{q(n)} \leq (110 + 10k)\%$. We need to prove that it gives the correct output $(k+1)$ for $(100+10(k+1))\% < \frac{x(n)}{q(n)} \leq (110+10(k+1))\%$. When this inequality holds, $(100+10(m))\% < \frac{x(n)}{q(n)}$

holds for all $0 \leq m \leq k+1$. This inequality fails to hold for $m = k+2$. This is equivalent to

$$\frac{x(n)}{q(n)} \leq (100 + 10(k+2))\%$$

$$\frac{x(n)}{q(n)} \leq (110 + 10(k+1))\%$$

$$10 \times x(n) \leq q(n) \times (11 + (k+1))$$

$$10 \times x(n) \leq q(n) \times l \text{ (As } l = 10 + m = 10 + (k+1))$$

When this inequality is satisfied, `iter(l,m)` returns $m = k+1$ as output. Hence, the function `qhike` gives the correct output for all x(n) (and n=1,2,3,4) by the *Principle of Mathematical Induction*. Thus, `qPerformance` gives the correct output for all valid inputs.

budgetRaise

`y` represents s_{new}, i.e., the sum of new salaries of each employee. `qPerformance` represents the list of new salaries of employees. `sum1(a,b)` gives `a+b`. Using `foldl sum1 0 (qPerformance(L))`, we can calculate the sum of the employees' new salaries. `x` represents s_{old}, i.e., the sum of old salaries of each employee. 5^{th} element of a tuple represents the list of the old salary of the respective employee. `sum2(a,b)` gives the sum of `b`, and the last element of the 5-tuple `a`. 5^{th} element of each tuple in L represents the employee's old salary. Using `foldl sum2 0 (L)`, we can calculate the sum of the employees' old salaries as `foldl` will initially add the first employee's old salary to `0` and then store it as `b` for the tuple corresponding to the second employee and so on. Since we are using the `/` operator, we need to convert y and x to real. We use `real(y)` and `real(x)` and subtract `1.0` from `real(y)/real(x)` to get the desired output.

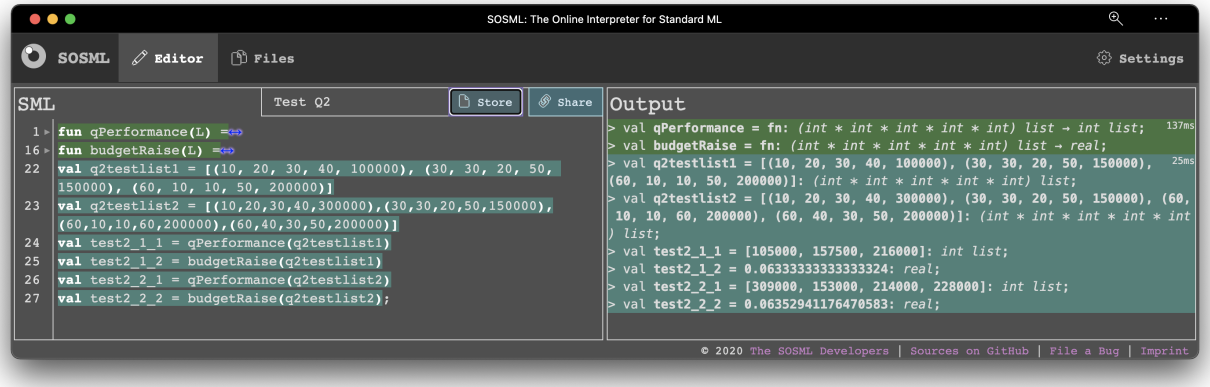
2.2.2 Complexities

Following are the *Complexities* for different functions:

qPerformance

The time and space complexities of `qPerformance` is $O(n^2)$ and $O(n)$, respectively, where n is the input list's length. First, the function calculates `(q1,q2,q3,q4,q5)` using `foldl` and `sum` functions. The `sum` function takes constant time and space for all inputs. The time complexity of `foldl` is $O(n)$ as it takes n steps to reach the list's last item. Its space complexity is also n as it works with a list of length n. `length` function also takes $O(n)$ space and time to calculate the value of `leng`. hike function has time complexity $O(n)$. `iter` function takes at most $10 \times (n-1)$ steps to terminate. The worst-case scenario is when the rest of the employees' performance points are zero for the i^{th} quarter. So $\frac{x(i)}{q(i)} = n$ for that case. `l` takes $10 * (n-1)$ steps to become $10 * n$ and the function terminates at that step. The function does not use more than 4 additional stake frames to calculate `qhike(i)` for `i = 1,2,3,4`. So, it has a space complexity of `O(1)`. `qPerformance` equivalent to map hike L takes n steps, and each step takes $O(n)$ time and $O(1)$ space. Thus, The time and space complexities of `qPerformance` is $O(n^2)$ and $O(n)$ respectively as the list if of $O(n)$ space.

The time and space complexities of `budgetRaise` is $O(n^2)$ and $O(n)$, respectively, where n is the input list's length. sum1 and sum2 have constant space and time complexities. The algorithm takes $O(n)$ space and time to calculate the value of x using foldl. The algorithm first calculates `qPerformance(L)` to calculate y. This evaluation takes $O(n^2)$ time and $O(n)$ space. The resulting list's length is n. Calculation of y after calculating `qPerformance(L)` takes $O(n)$ space and time. At the end, the function performs constant number of calculation to calculate `(real(y)/real(x))-1.0`. Thus, the final time and space complexities of `budgetRaise` are $O(n^2)$ and $O(n)$, respectively, and this is due to `qPerformance(L)`'s evaluation taking $O(n)$ time.



3 Lexicographic Permutations

3.1 Algorithm

Following are the *algorithms* for different functions:

lexicographicPerm

```
60 fun lexicographicPerm(li:char list) =
61   let val Li =...
78   fun Perm([]) = [[]]...
98   in Perm(Li) end;
```

`lexicographicPerm(li)` gives a list of char lists of all possible permutations of the input char list in lexicographic order.

sort

`Li` is the sorted version of the input list. `Li` ≡ `sort(li)`. sort function is the same as merge sort function; it splits a list into two parts, sorts them separately, and merges the two sorted lists.

`sort([])=[]`

`sort([x])=[x]`

are trivial. For lists of bigger length, `split` divides the list into two smaller lists. It adds the odd-numbered items to the first list and even-numbered items to the second list. `merge` merges two sorted lists by comparing their first elements, taking the smaller element out, and bringing it in front of the new list. Then it recursively calls itself with that element removed. The resulting merged list is sorted because the input lists are sorted. The input lists in merge are the sorted versions of the lists we obtain on splitting `L`, i.e., `sort(L1)` and `sort(L2)`. Since `sort` is called recursively till the input lists are of a length less than 2, our function terminates at that step and gives `sort([x])=[x]`. We split input lists in two recursively and then merged two lists sorted at a

```
60  fun lexicographicPerm(li:char list) =
61    let val Li =
62      let fun sort([])=[]
63        |sort([x])=[x]
64        |sort(L) =
65        let fun split([])=([],[])
66          |split([a])=([a],[])
67          |split(x::y::z)=
68            let val (a,b) = split(z)
69        in (x::a,y::b) end
70        fun merge(l1,[]) = l1
71          |merge([],l2) = l2
72          |merge(x::l1,y::l2) =
73            if x<y then x::merge(l1,y::l2)
74            else y::merge(x::l1,l2)
75        val (L1,L2) = split(L)
76        in merge(sort(L1),sort(L2)) end
77      in sort(li) end
78    fun Perm([]) = [[]]
98    in Perm(Li) end;
```

lower level.

Perm

```
60  fun lexicographicPerm(li:char list) =
61    let val Li =
78    fun Perm([]) = [[]]
79      |Perm([a]) = [[a]]
80      |Perm(L) =
81      let fun nlexi(n) =
82        let fun lexilist(n) =
83          let fun iter(i,[]) = raise Empty
84            |iter(i,x::xs) =
85            if i = n then (x,xs)
86            else
87              let val (a,b) = iter(i+1,xs)
88              in (a,x::b) end
89          val (a,b) = iter(0,L)
90          in (a,Perm(b)) end
91        fun append(a,[]) = []
92          |append(a,x::xs) = (a::x)::(append(a,xs))
93        in append(lexilist(n)) end
94      fun iterater(i) =
95        if i = length(L) then []
96        else (nlexi(i))@(iterater(i+1))
97      in iterater(0) end
98    in Perm(Li) end;
```

`Perm` function gives out all possible permutations of an input char list as a list of char lists. It permutates them in the order the elements appeared in the input list. If the input list is sorted, the permutations become lexicographic. Our required answer is `Perm(Li)`. `nlexi(n)` function takes out the $(n+1)^{th}$ element of the list, then generates a list of permutations of the remaining list, and then appends it. `lexilist(n)` gives a 2-tuple `(a,Perm(b))` as output, where a is the $(n+1)^{th}$ element, and b is the list consisting of all elements apart from a. `Perm(b)` represents the list of lexicographic permutations of the list `b`. It uses the help of `iter` function to find `(a,b)`. `iter(i,x::xs)` takes `x` out and increments `i` by `1` until its count reaches `n`. Then it takes the value of `x` at that point.

`iterater` iterates `nlexi(i)` for all possible `i` and then appends them to form a new list consisting of the required permutations.

3.2 Correctness and Complexities

3.2.1 Correctness

`sort(li)` sorts the list in increasing order. It firsts splits the input list into two lists using `split`.

`split([])=([],[])`

`split([a])=([a],[])`

are trivial. Assume that `split` gives the correct output for all lists of length $\leq k$. We need to show that it gives the correct output for a list of length $k+1$.

Let this list be `x::y::z`, where `z` is a list of length $k-1$. `split(x::y::z) = (x::a,y::b)` where `a,b` are the lists obtained on splitting `z` based on our assumption. So, `split` adds `x,y` to different lists, and hence by the *Principle of Mathematical Induction*, it gives the correct output for lists of all sizes.

The merge function merges two sorted lists. Suppose we have two lists `L1` and `L2` that are sorted we need to show that merge merges these two lists and the output list is a sorted list. Assume that merge works for two lists of length k, m with $k \geq m$ without loss of generality. We need to show that it works for lists of length $(k+1), m$.

Suppose that `L1` consists of characters x_i such that $i = 1, 2, 3, ..., k+1$ and L2 consists of characters y_j such that $j = 1, 2, 3, ..., m$. Since `L1` and `L2` are sorted, $x_1 \leq x_2 \leq x_3 ... \leq x_{k+1}$ and $y_1 \leq y_2 \leq y_3 .... \leq y_m$.

Case 1: $x_1 \geq ym$

In this case, merge takes out elements from the second list till it becomes empty. Using `merge(l1,[]) = l1`, we get the first list after the second list in sorted order.

Case 2: $x_1 < y_j$ for some $j = 1, 2, 3, ...m$

In this case, merge takes at most $j-1$ elements out of `L2` then it takes x_1 out of `L1`. In this case, the elements taken out till now are sorted, and since the remaining lists will be sorted correctly based on our induction hypothesis, merge merges them in. the correct order and gives a sorted list as output. By *Principle of Mathematical Induction*, merge merges all sorted lists correctly for lists of all sizes. Thus, `Li = sort(li)` gives a sorted list of the original list's characters.

`Perm` function outputs all possible characters of the input list in the order they appear. We can think of this as `Perm` function giving the lexicographic permutations of the input list as output , assuming the input list is sorted correctly.

`Perm([]) = [[]]`

`Perm([a]) = [[a]]`

is trivial. Assume that `Perm(L)` gives the correct output for some $n \geq 1$. where n is the length of the list `L`.

We will first prove the correctness of all associated functions to prove the correctness of `Perm(L')`, where L' is another list of length $n+1$.

`nlexi(i)` gives the $(i+1)^{th}$ element of the input list added to the remaining items' permutations. `lexilist(i)` gives a tuple `(a,Perm(b))` as output, where a is the $(i+1)^{th}$ element of `L` and `Perm(b)` represents the `char list list` of all elements except $(i+1)^{th}$ element of `L`. It uses the help of the function `iter`.

`iter(i,L)` gives a tuple containing $(i+1)^{th}$ element of L and the remaining list. `iter(n,x::xs)` gives `(x,xs)` as output as desired.

Assume that `iter(k,L)` gives the correct output for all L and an integer $k \leq n$. This output is `(a,b)`, where `a` is the $(n+1)^{th}$ element of the list and `b` is the remaining list. We need to show that it gives the correct output for $(k-1)$. `iter(k-1,x::xs) = (a,x::b)`. Since $k-1 \neq n$, x must be a part of the remaining list. So, `iter(k-1,x::xs)` gives the correct output for $(k-1)$. By *Principle of Mathematical Induction*, `iter(i,L)` gives the correct output for all $0 \leq i \leq n$.

`lexilist` is `(a,Perm(b))` and is correct whenever `Perm(b)` gives the correct output. Since, `b` is of length n, `Perm(b)` gives the correct output based on our initial assumption. `append(a,L)` appends the character `a` to each element of the `char list list L`. (Each element of `char list list` is a `char list`).

`append` function outputs a null list for a null list as input. Assume that `append` correctly appends `a` to a list of size n. We need to prove that it correctly appends to a list of size $n+1$.

Let `x::xs` be a `char list list` of length $(n+1)$.

Now, `append(a,x::xs) = (a::x)::(append(xs))` Since `append(xs)` gives a list of char lists with a appended and `(a::x)` represents `a` appended to the first item of the `char list list x::xs`, we now have a `char list list` of size $(n+1)$ with `a` appended to each item. By *Principle of Mathematical Induction*, `append(a,L)` appends the character `a` to each element of the `char list list L` and gives the resulting `char list list` as output.

`nlexi(i)` is `append(lexilist(i))`, i.e, $(i+1)^{th}$ element appended to the permutations of the remaining list, which is correct as the helper functions `append` and `lexilist` are correct as well. `iterater(i)` stops when the i equals the length of the list n. Otherwise, it appends the `char list list nlexi(i)` to the remaining list. Since `i` starts with `0`, we are calculating `nlexi(0)@nlexi(1)@nlexi(2)@...@nlexi(n-1), or more accurately, nlexi(0)@(nlexi(1)@(nlexi(2)@...@(nlexi(n-1)))....)`.

`nlexi(0)` has the first element appended in front, `nlexi(1)` has the second element appended in front, and so on. So, `Perm(L')` gives a list containing permutations of the elements of `L'` in the order they first appear. Since we input sorted list `Li` in `Perm(Li)`, we get a list of lexicographic permutations. Thus, we proved that Perm works for size n+1 if it works for lists of size n. By *Principle of Mathematical Induction*, Perm works for lists of all sizes and thus `Lexicographicperm` $\equiv$ `Perm(Li)` gives lexicographic permutations for lists of all sizes.

3.2.2 Complexities

Sort has a time complexity of $O(nlog(n))$ and space complexity of $O(n)$ as we have seen in class. Suppose $2^{k-1} < n \leq 2^k$ for some k. Then sorting of list of size n takes less than equal to steps taken to sort a list of size 2^k. Suppose that sorting a list of size 2^k takes $T(2^k)$ time. merge takes at most n steps to merge the given sorted lists as it takes an element of the lists at each step ad there are at most n elements. It also has a space complexity of $O(n)$ as it generates a list of size n and reduces the size of the input lists by one as it generates an additional stack frame. `split` also has a time complexity of $O(n)$ as it takes n steps to generate the two new lists. It also has $O(n)$ space complexity as it is working with lists of size n and does not generate any additional stack frames. Let $T(1) = c$. Now

$$T(2^k) = 2T(2^{k-1}) + 2O(2^k)$$

which is equivalent to

$$T(2^k) = 2T(2^{k-1}) + O(2^k)$$

$$2T(2^{k-1}) = 4T(2^{k-2}) + 2O(2^{k-1})$$

$$\vdots$$

$$2^{k-1}T(2) = 2^kT(1) + 2^{k-1}O(2)$$

Summing both sides, we get

$$T(2^k) = 2^kc + k * O(2^k)$$

As $2^i \times O\left(\frac{2^k}{2^i}\right) = O(2^k)$.

Now, $k = log(n)$. So $T(n) = log(n)O(n) = O(nlog(n))$. Let $T(n)$ be `Perm(L)`'s time complexity for a list of length `L`. For Perm, we first calculate the time complexities of the associated helper functions. `lexilist(i)` takes $i \leq n$ steps to calculate the value of `(a,b)` using `iter`. `iter` has the same space complexity as the input list. Thus, `iter` and `lexilist` have a space complexity of $O(n)$. Then, it uses $T(n-1)$ to calculate `Perm(b)`. Thus, the time complexity of `lexilist` is $T(n-1) + O(n)$. `append(a,Perm(b))` takes $(n-1)!$ steps

to append a to each element of `Perm(b)` containing $(n-1)!$ elements. It takes $O(n!)$ space while doing so as it works with an input list of length $(n-1)!$ (each element is of length n) and reduces the size of the list as it is generating additional stack frames. Thus, `nlexi` takes a total of $T(n-1)+O((n-1)!)$ time. `nlexi(i)` outputs a list of length $(n-1)!$. Calculating `nlexi(0)@(nlexi(1)@(nlexi(2)@...@(nlexi(n-1)))...)` takes a total of $n^2(n-1)!$ time. This can be proved as follows. Suppose we have two add k lists of length m. First, we add the two rightmost lists. This process takes $2m$ time. Then we add m with a list of length $2m$. This takes $3m$ time. We continue in this manner, and at the final step, it takes km time. So total time taken is $2m+3m+4m+....+km = O(k^2m)$ time. For our case, $k=n$ and $m=(n-1)!$ Since `nlexi` is called n times, the total time taken by `Perm` is

$$T(n) = n(T(n-1)+O((n-1)!))+O(n^2(n-1)!)$$
$$T(n) = n(T(n-1))+O((n+1)!)$$
$$T(n-1) = (n-1)(T(n-2))+O((n)!)$$
$$nT(n-1) = n(n-1)(T(n-2))+nO((n)!)$$
$$n(n-1)T(n-2) = n(n-1)(n-2)(T(n-3))+n(n-1)O((n-1)!)$$
$$.$$
$$.$$
$$.$$
$$n!T(2) = n!(T(1))+(n)(n-1)....(3)O(3!)$$

On summing up both sides we get

$$T(n) = n!(T(1))+O(n!)((n+1)+(n)+(n-1)+...+(3))$$

Taking $T(1)$ as $O(1)$ and $O(n!)((n+1)+(n)+(n-1)+...+(3))$ as $O(n^2n!)$,

$$T(n) = n!(O(1))+O(n^2n!)$$
$$T(n) = (O(n!))+O(n^2n!)$$
$$T(n) = O(n^2n!)$$

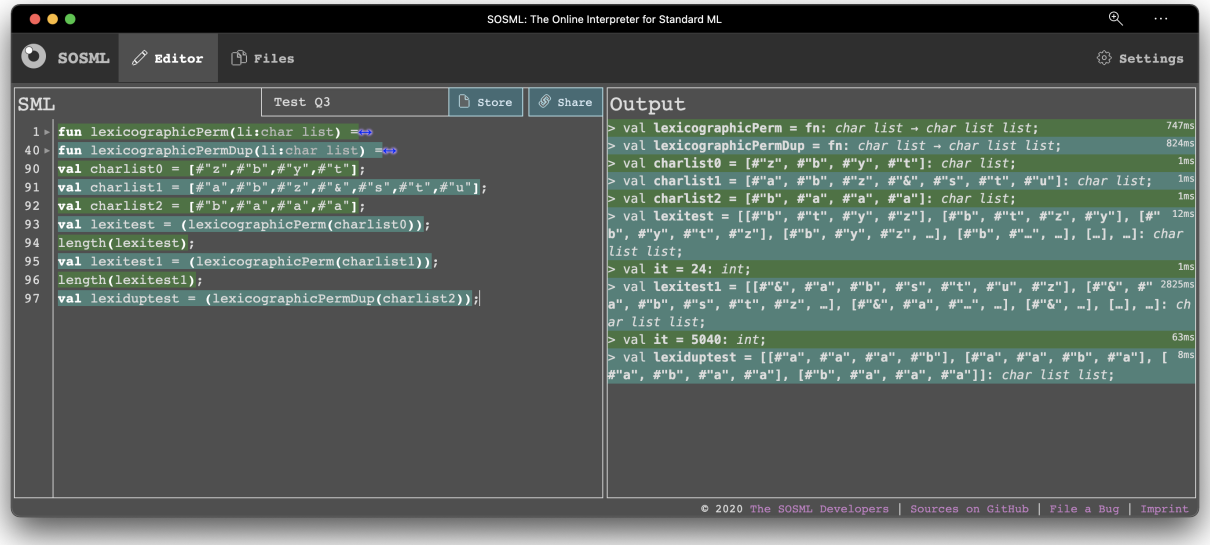
So, the time complexity of Perm is $O(n^2n!)$. Which is also the final time complexity as $O(nlog(n))+O(n^2n!) = O(n^2n!)$. `iterater` has the same space complexity as the output list. So, it has a space complexity of $O(n*n!)$ as it gives a list of length n! as output, each element's length being n. So, the space complexity of `LexicographicPerm` is $O(n*n!)$.

```
fun lexicographicPermDup(li:char list) =
  let val Li =…
  fun Perm([]) = [[]]
    |Perm([a]) = [[a]]
    |Perm(L) =
    let fun nlexi(n) =…
    val Leng = length(L)
    fun nterm(n) =
      let fun iter(i,[]) = raise Empty
        |iter(i,x::xs) =
        if i = n then x
        else iter(i+1,xs)
      in iter(0,L) end
    fun iterater(i) =
      if i = Leng then []
      else
        if i = Leng - 1 then (nlexi(i))
        else
        if (nterm(i))=(nterm(i+1)) then (iterater(i+1))
        else (nlexi(i))@(iterater(i+1))
    in iterater(0) end
  in Perm(Li) end;
```

3.3 Bonus

The algorithm for the bonus part is almost the same as the main function. It just takes care of duplicates in the iteration step in the `iterater` function. It does this with the help of the `nterm` function. `nterm(n)` is almost the same as the `lexilist(n)` function, except that it returns the value of the $(n+1)^{th}$ term only. Note that Permutations generated using `Perm` will be identical when the first item and the remaining list items are the same. The sorted list will have duplicates next to each other.

Thus, `nlexi(i)=nlexi(i+1)` when $(i+1)^{th}$ and $(i+2)^{th}$ terms of the sorted list are same. Since, `Perm` functions at each level of the list, this way we avoid duplication at each level and the output list does not have any duplicate permutations. `nterm(i)` has a space and time complexity of `O(n)` which can be shown just like we showed the space and time complexity of `iter` function in `lexilist`. Thus, our final time and space complexities are the same as the original function.


SOSML: The Online Interpreter for Standard ML
SOSML
Editor
Files
Settings
SML
Test Q3
Store
Share
1 ▸ fun lexicographicPerm(li:char list) =⇔
40 ▸ fun lexicographicPermDup(li:char list) =⇔
90 val charlist0 = [#"z",#"b",#"y",#"t"];
91 val charlist1 = [#"a",#"b",#"z",#"&",#"s",#"t",#"u"];
92 val charlist2 = [#"b",#"a",#"a",#"a"];
93 val lexitest = (lexicographicPerm(charlist0));
94 length(lexitest);
95 val lexitest1 = (lexicographicPerm(charlist1));
96 length(lexitest1);
97 val lexiduptest = (lexicographicPermDup(charlist2));
Output
> val lexicographicPerm = fn: char list → char list list; 747ms
> val lexicographicPermDup = fn: char list → char list list; 824ms
> val charlist0 = [#"z", #"b", #"y", #"t"]: char list; 1ms
> val charlist1 = [#"a", #"b", #"z", #"&", #"s", #"t", #"u"]: char list; 1ms
> val charlist2 = [#"b", #"a", #"a", #"a"]: char list; 1ms
> val lexitest = [[#"b", #"t", #"y", #"z"], [#"b", #"t", #"z", #"y"], [#" 12ms
b", #"y", #"t", #"z"], [#"b", #"y", #"z", …], [#"b", #"…", …], […], …]: char
list list;
> val it = 24: int; 1ms
> val lexitest1 = [[#"&", #"a", #"b", #"s", #"t", #"u", #"z"], [#"&", #" 2825ms
a", #"b", #"s", #"t", #"z", …], [#"&", #"a", #"…", …], [#"&", …], […], …]: ch
ar list list;
> val it = 5040: int; 63ms
> val lexiduptest = [[#"a", #"a", #"a", #"b"], [#"a", #"a", #"b", #"a"], [8ms
#"a", #"b", #"a", #"a"], [#"b", #"a", #"a", #"a"]]: char list list;
© 2020 The SOSML Developers | Sources on GitHub | File a Bug | Imprint