

Sai Nishanth Mettu (sm11326) – Assignment 5

1) Factorial Function – Non-Tail Recursive

```
student@pl2023:~/Desktop$ ./factorial
Factorial of 5 :
120
Factorial of 123456 :
26040699049291378729513930560926568818273270409503019584610185579952057379676834157935607166171279087355200170616
66000857261271456698589373086528293431724412115286581403020464598557341925130534223113557349105075627773504656933
73539659735869465710825190620341666868255704383656336382977870547298029477598891252511493479985597083202362548198
57950233238475301030341322536465099510321559236876610559963680670898109477391265996020702808577885204568468629754
05892240199795622185904449551072894142156396870803554498530395113467097692314079392617785551001374486925509809258
9701383656131240719620120988481474785469705567773024292709538983252312520579148756622549777722437694684345316392
77622693893695711468703110506186696702183962504330964793547158023182167199897119829395076737914410976787752768697
72692319069185008505356830236139649301382888122317876523528024348730982715765959261774213138968142299890293937760
34196287051827142583143471321201867439854515081725998506324093018265050081840771382590529446690657578611635149509
15837293446759149139720291850698461690541489434460217750327436956756068244018958445197887178337344725228170881151
72512616425029615944952496927994468437491902771950842156268356559709691300709532568162765316448284883426016667763
76590540808324894867626836971193131275528752013155994366445261470889888519159723838134322421224130659739789014938
36988171395856117033029891689843950318176848010676668233361694753098434731903687502510607608772281610050299549918
64619316621069032065889910623240968038869735805204599603574682539477109446234616125629634575537971766676775597818
08740625164743470723137632470538710307484607807304225997814319490254493248945760331910422895736611092167375036405
04433543826422229957075196812685234924306546789063543918689574606800907037812326455978472137620148372958976743417
43664279143520819105348425488234545484989163380740855056743634668982146092269361006667009463004116508758906424881
87805367189499053623251962925949384999230844523167531676839140741284864066493787408913247082793168218234519853988
71848056316564126182486127854964340189662512315115490083323559412075502514575814570846803124763291556184692683982
38882947441058377093982034347760642149543310623391728937484006719238701776042599670504040161681446355384131252756
00700950143924697420834534695169543445898097488418459910985631629808662751807076801105646773385459911401930302303
86910534909319274526701469049473565586505875959811486204158019637961205528595869144659010857026188079254869526465
98452438162533722988375261276334046137414840781685801852402217550152380056860881000055881624522688867082956977052
```

Explanation (Why Non-Tail Recursive Factorial is still able to handle large inputs)

Factorial of non-tail recursive implementation $O(n)$ can handle fairly large inputs without encountering stack overflow errors because **of Haskell's laziness and garbage collection mechanisms.** In Haskell, even though the non-tail recursive factorial function accumulates stack frames, those frames are lazily evaluated and garbage collected when no longer needed. This means that the memory usage remains bounded, allowing the function to compute factorials of large numbers without running into memory issues.

When you call **factorial 123456**, Haskell's lazy evaluation and garbage collection mechanisms allow it to compute the result without encountering stack overflow errors.

2) Factorial Function – Tail Recursive

```
1/1 + [icons] TiliX: student@pl2023: ~/Desktop
1: student@pl2023: ~/Desktop
| ~~~~~
Linking tail-factorial ...
student@pl2023:~/Desktop$ ./tail-factorial
Factorial of 5 using tail recursive form :
120
Factorial of 123456 using tail recursive form :
26040699049291378729513930560926568818273270409503019584610185579952057379676834157935607166171279087355200170616
66000857261271456698589373086528293431724412115286581403020464598557341925130534223113557349105075627773504656933
73539659735869465710825190620341666868255704383656336382977870547298029477598891252511493479985597083202362548198
57950233238475301030341322536465099510321559236876610559963680670898109477391265996020702808577885204568468629754
05892240199795622185904449551072894142156396870803554498530395113467097692314079392617785551001374486925509809258
970138365613124071962012098848147478546970556777302429270953898325231252057914875662254977722437694684345316392
77622693893695711468703110506186696702183962504330964793547158023182167199897119829395076737914410976787752768697
72692319069185008505356830236139649301382888122317876523528024348730982715765959261774213138968142299890293937760
34196287051827142583143471321201867439854515081725998506374093018265050081840771382590529446690657578611635149509
15837293446759149139720291850698461690541489434460217750327436956756068244018958445197887178337344725228170881151
72512616425029615944952496927994468437491902771950842156268356559709691300709532568162765316448284883426016667763
76590540808324894867626836971193131275528752013155994366445261470889888519159723838134322421224130659739789014938
36988171395856117033029891689843950318176848010676668233361694753098434731903687502510607608772281610050299549918
64619316621069032065889910623240968038869735805204599603574682539477109446234616125629634575537971766676775597818
08740625164743470723137632470538710307484607807304225997814319490254493248945760331910422895736611092167375036405
0443354382642229957075196812685234924306546789063543918689574606800907037812326455978472137620148372958976743417
43664279143520819105348425488234545484989163380740855056743634668982146092269361006667009463004116508758906424881
87805367189499053623251962925949384999230844523167531676839140741284864066493787408913247082793168218234519853988
71848056316564126182486127854964340189662512315115490083323559412075502514575814570846803124763291556184692683982
38882947441058377093982034347760642149543310623391728937484006719238701776042599670504040161681446355384131252756
00700950143924697420834534695169543445898097488418459910985631629808662751807076801105646773385459911401930302303
86910534909319274526701469049473565586505875959811486204158019637961205528595869144659010857026188079254869526465
98452438162533722988375261276334046137414840781685801852402217550152380056860881000955881624522688867082956977952
32008434757889679286811513503556584420061098956639846663429224396359450384165214602656651019693511015612206612478
62128498518228053076418319977614917228331123522277283118593509438718729494401041697565513762084544435393415884700
13640898815689834860889687103341972595565999517176549464113739851267020076302724505783488245448224533276048476506
71899564180093702584590843434148358189163634419115491806551688737554516210856996839240853885284058835910124726914
97240443450710215360281250689997875713903132936386413819156998289697885847504698571128915360166525404026337010874
31077978840698419079834228727137476250881239734147263427697180215097887715315110723865948942968519736489905680346
10172983710941968110887183196797066986550512910738947610301533588104990125546808891443119188031043407892708934184
73675003016107682856741908840240641109895686495013216276417696624990003683756116188218352015584692071399696665952
```



Explanation (Does Haskell Support Tail Recursion for Factorial and in General?)

Yes, In Haskell, tail recursion is supported, but it doesn't automatically “optimize” tail-recursive functions. Tail recursion means that the recursive call is the last operation in the function body, which can improve memory usage compared to non-tail-recursive functions. However, Haskell's runtime system doesn't guarantee automatic tail call optimization. GHC offers some tail call optimizations under certain conditions, known as "tail call optimization under certain circumstances". These optimizations can improve performance by optimizing tail calls in specific situations.

Overall, while Haskell supports Tail Recursion, it doesn't provide automatic tail call optimization, its lazy evaluation and garbage collection mechanisms help facilitate efficient execution of tail-recursive functions for many practical scenarios.

3) Fibonacci Function – Non Tail Recursive

```
Linking fibonacci ...
student@pl2023:~/Desktop$ ./fibonacci
Fibonacci of 10 using normal fibonacci recursion :
55
Fibonacci of 123456 using normal fibonacci recursion :
```

 **Online Haskell Compiler** 

Execute | Beautify | Share | Source Code

Terminal

```
1 fibonacci :: Integer -> Integer
2 fibonacci 0 = 0
3 fibonacci 1 = 1
4 fibonacci n = fibonacci (n - 1) +
    fibonacci (n - 2)
5
6 main = do
7     putStrLn "Fibonacci of 10 using: "
8     print $ fibonacci 10
9     putStrLn "Fibonacci of 20 using: "
10    print $ fibonacci 20
11    putStrLn "Fibonacci of 30 using: "
12    print $ fibonacci 30
13    putStrLn "Fibonacci of 40 using: "
14    print $ fibonacci 40
15    putStrLn "Fibonacci of 50 using: "
16    print $ fibonacci 50
17    putStrLn "Fibonacci of 60 using: "
18    print $ fibonacci 60
19    putStrLn "Fibonacci of 123456 : "
20    print $ fibonacci 123456
```

```
[1 of 1] Compiling Main (
    main.hs, main.o )
main.hs:7:1: warning: [-Wtabs]
    Tab character found here, and in 13
    further locations.
    Please use spaces instead.
|
7 |         putStrLn "Fibonacci of 10
  |         using: "
  |         ^^^^^^^^^
Linking main ...
Fibonacci of 10 using:
55
Fibonacci of 20 using:
6765
Fibonacci of 30 using:
832040
Fibonacci of 40 using:
102334155
Fibonacci of 50 using:
```

Why Non-Tail Recursion for Fibonacci eats up Memory, executes extremely slowly and times out eventually and why there isn't any error message, which we saw in Lisp?

The non-tail recursive implementation of the Fibonacci function in Haskell becomes increasingly slow for larger inputs due to its exponential time complexity. It is $O(2^n)$... This inefficiency arises because the function recalculates the Fibonacci numbers for smaller values multiple times, leading to redundant computations.

In a non-tail recursive Fibonacci implementation like the one you provided, the recursive calls branch out exponentially. For each value of n , two recursive calls are made to calculate $\text{fib}(n - 1)$ and $\text{fib}(n - 2)$. As n increases, the number of redundant computations grows rapidly, resulting in a significant slowdown in execution time, directly as the result of no optimization.

Despite the inefficiency, the non-tail recursive Fibonacci function doesn't throw errors because it's still valid Haskell code. It will eventually compute the Fibonacci numbers correctly for reasonably small inputs, although it may take a long time for larger inputs due to the exponential time complexity. **While GHC Compiler does optimize (by Lazy Evaluation and Garbage Collection to a certain degree by itself, as seen in the earlier Non-Tail Factorial Problem.. it still isn't fast enough for exponential issues like $O(2^n)$.**

→ **Why Non Tail Recursive Factorial worked for larger inputs but not Fibonacci? (why is Fibonacci so slow)**

The reason why non-tail recursive factorial worked for larger inputs but not Fibonacci is because the factorial function's time complexity grows linearly with the input size. $O(n)$ In a non-tail recursive factorial implementation, each recursive call multiplies the current result by the next number until it reaches the base case. This linear growth allows the function to handle larger inputs without encountering stack overflow errors, although it may still suffer from performance issues for very large inputs due to the redundant calculations. However, the **exponential growth $O(2^n)$** of the Fibonacci function's time complexity in the non-tail recursive implementation causes it to become extremely slow for relatively small inputs like 50, making it impractical for larger inputs.

4) Fibonacci Function – Tail Recursive

```
Linking tail-fibonacci ...
student@pl2023:~/Desktop$ ./tail-fibonacci
Tail Recursive Fibonacci of 10 is :
55
Tail Recursive Fibonacci of 123456 is :
268305576502820904863831635364580578152316406572554082383476815325507031500921832968519843063
166742954880195378358679672503095874978770235997192296829947699315900184358007864172966752231
291515275893550316694407401423173184545648782874357753615252493441043792076592574814481700274
61510017775638842857458213262869332969146561943465819186783859633056542866179785312648315945
079332977361617640083510069383325506090535035875934480990556244337810711447567414739761582574
103468189833028953772346718574834116846465226544766394150604543112500808228901112417740723886
804949179256909951858933135313066627450470675721819932583543818632745482617590803860595139870
362476031944345095176947527893980467084073529269183124098291361331746364385688508029910748716
777245956634402847346358489347878861765103905235475824165921600275926831650284112027205657855
876703177629907372162251229704621377744051813062235773088955942849969761201216456213103500937
070310345154034500154897239016735114345726419540973696692541413816499519462708328092522442650
272278089253219518276667867934718607992515584246489273108404261808944102304396137455764986402
712902388140953924214672649910184945042365655405195410149464104449979195218764590161509100517
562068174878153443319797981978388348738352448963373025335891111807375386385002741872430943693
642189089906607631662946918198396042895833449839328930870077346191310098317661879957072286683
003377167661515201555616620887185827095667738622031671719727754687790456818170509656648879794
417165081545671230445924999852613306836203053924869774921647686214377275577313002946361412588
771957066935970871650804130896822809161388513586800021325842907655851557197434399582565824384
```

Explanation :

1. The recursive call **sub (n - 1) b (a + b)** is in the tail position, meaning it's the last operation performed in each recursive step. This allows the Haskell compiler GHC or GHCi to optimize the recursion by reusing the same stack frame for each recursive call, instead of creating a new stack frame for each call. This optimization prevents stack overflow errors for large inputs, allowing the function to compute Fibonacci numbers efficiently even for very large inputs.

The optimization process works as follows:

- When the compiler encounters a tail-recursive function like **fibonacciTail**, it recognizes that the recursive call is in the tail position and can be optimized.
- The compiler transforms the tail-recursive function into an equivalent iterative loop internally, removing the need to create a new stack frame for each recursive call.
- This optimization, known as tail call optimization (TCO) or tail call elimination (TCE), eliminates the risk of stack overflow errors for large inputs by reusing the same stack frame for each recursive call.

Overall, in Summary!... the tail-recursive Fibonacci function in Haskell is optimized by leveraging tail call optimization, which transforms the recursion into an iterative process, preventing stack overflow errors and allowing efficient computation of Fibonacci numbers even for very large inputs.

FINAL INFERENCE & KNOWLEDGE GAIN

Yes, tail recursion is supported in Haskell.

Haskell compilers like GHC (Glasgow Haskell Compiler) support tail recursion optimization (TCO) “only under certain conditions”. When a function meets the criteria for tail recursion, GHC is able to apply TCO, which effectively transforms the recursive function into an iterative one, thus avoiding the accumulation of stack frames and preventing stack overflow errors.

Under the right circumstances, GHC can optimize tail-recursive functions to execute efficiently even for large inputs, making tail recursion a viable and supported technique in Haskell for writing efficient and scalable code.

The criteria for tail recursion, which enables TCO, include:

1. **Tail Call:** The recursive call must be the last operation in the function body. In other words, there should be no further computation to be done after the recursive call.
2. **No Accumulation of Deferred Operations:** There should be no accumulation of deferred operations after the recursive call. Deferred operations include pending arithmetic operations or further function calls.
3. **Tail Call in All Branches:** In functions with multiple recursive branches (e.g., pattern matching), each branch should end with a tail call. This ensures that TCO can be applied to all branches of the recursion.

When these criteria are met, the compiler can optimize tail-recursive functions by reusing the same stack frame for each recursive call, effectively transforming the recursion into an iterative process. This optimization prevents the accumulation of stack frames, leading to improved memory usage and prevention of stack overflow errors for large inputs.