Caden Tierney and Shayan Daijavad

CSC 431

Professor Stephen Beard

The Mini Compiler (Java Edition)

1. Compiler Overview

Here is the link to our compiler. It is written entirely in Java.

https://github.com/CalPoly-Beard-Compilers/compiler-double-dog-man

1.1 Parsing

Our compiler uses the given ANTLR parser and lexer to generate an abstract syntax tree representation of original mini code. It uses a Mini.g4 as a grammar, which ANTLR takes and uses to produce a set of Java classes representing the structures of the language. The structure consists of a program class, which includes struct type definitions, global variable declarations, and functions. The functions consist of declarations (for parameters and local variables) as well as a body. The body is represented as a statement, which can be further broken down into additional statements and expressions. The internal class structure also keeps track of types for all declarations.

1.2 Static Semantics

Our type checking was implemented using a typeCheck function that almost all of the AST classes from parsing include. The typeCheck function takes a top down approach, starting from Program and then calling typeCheck on the program’s functions, which then type checks the function’s body and subsequently all statements and expressions. At the bottom level, typeCheck returns a Type, which gets passed all the way back up, and throws errors if any types mismatch. For example, BinaryExpression type-checks its left expression and the right expression, making sure they evaluate to the same type and that those types are legal (i.e. addition is only between two integer types).

The typeCheck function takes in an Environment, which contains a mapping for all of the different structs, symbols, and functions to their respective types. The environment also lets us keep track of what is currently defined so we can throw an error if a symbol doesn’t exist. The environment is initialized in Program’s typeCheck function, and then recursively passed down to all of the functions. This also allows us to keep track of variable scope, since each function gets their own copy of the environment and can extend it as they see fit without interfering with the rest of the program.

For return path checking, we created an interface called ReturnableStatement with an ensureReturn function that return statements, conditionals, whiles, functions, and block statements all implement. Return paths also take a top down approach, starting from functions and then calling ensureReturn on the body (only if the function is not a void type), which will call ensureReturn on all block statements, whiles, return statements, and conditionals within the body. Return statements always guarantee a return, conditional statements guarantee a return if both their then and else guarantee a return, and while statements only ensure a return if their guard is a TrueExpression. If anything within the function guarantees a return, then a return path has been found. Type checking handles whether or not the return path actually returns the correct type.

1.3 Intermediate Representations

1.3.1 Control Flow Graphs

A control flow graph is a way of representing the different paths a program can take while executing. The nodes in the graph are “blocks” filled with instructions that have one entry point and one exit point. The edges in the graph represent the flow of control from one block to another. For instance, if a conditional statement is encountered in a block, its guard remains in that block and creates a branch to two new blocks, one for the then portion of the conditional and one for the else. While loops are similar, where once the guard gets encountered, an edge is created to a new block containing the body of the while loop, and then the body gets an edge to another block that just has the guard again called a loopback block that will take the program back to the body block if the condition is met. The CFG provides a logical, easy to visualize way of representing the program that also makes analysis easier. For example, given the CFG, we can easily tell where variables are live by seeing what successor blocks they show up in, which allows us to do things like dead code elimination much more easily. The CFG is also incredibly valuable later on when doing register allocation.

1.3.2 LLVM

Our intermediate representation was a set of LLVM instructions that we represented using Java classes (one for each instruction). These instructions were what we filled our basic blocks with. This intermediate representation is useful because it allows us to perform analysis without being constrained by the physical restrictions of a particular architecture. The IR, once constructed, makes converting to a specific architecture much quicker, making the compiler more versatile. It also allows us to do certain optimizations and transformations before having to worry about architecture specific details, making the whole process simpler and more portable.

1.3.3 SSA Form

Static Single Assignment Form is useful because it forces variables to be assigned in exactly one place, which makes tracking variable usages much easier, since we don't have to worry about them potentially having two different definitions at any point in the code. This makes certain transformations (in our case, constant propagation and useless code elimination) much simpler to implement, because we know for certain whether or not a given variable definition is actually being used and whether or not that variable is a register or a literal.

1.4 Optimizations

The two optimizations our compiler implements are sparse simple constant propagation and relaxed critical instruction removal.

1.4.1 Sparse Simple Constant Propagation

Sparse simple constant propagation starts from the prologue block and works its way through all the blocks, checking each instruction to see if they are "operator" instructions. Operator instructions include things like add, mul, or, and, and so on which can potentially be simplified into a literal if they operate on literals. We created an "evaluate" function for these operator instructions that either returns the literal they simplify to or null if they can't be simplified. For each operator instruction that gets evaluated to a literal, our constant propagation method replaces all instances of the result register of that operator instruction with the literal. It keeps doing passes of all the blocks, starting from the prologue, until there are no more changes.

1.4.2 Relaxed Critical Instruction Removal

Relaxed critical instruction removal similarly starts from the prologue block and works its way through all the blocks, repeating until no blocks are changed. It is basically a mark and sweep algorithm, initially marking certain instructions as critical (returns, stores, function calls, control flow), then marking all other instructions that those initial ones depend on as critical, and then sweeping any instruction not marked as critical away. We accomplish this by marking all the initial critical instructions value dependencies as critical (i.e. the arguments for a call instruction, or the value to be stored for a store), and then on every subsequent passthrough each instruction that produces that value as a result will be marked as critical. For instance, if a store instruction stores the register r3 somewhere, then r3 gets marked as critical, and if there's an add instruction whose result gets put in r3, then that add instruction gets marked as critical. This continues until no more instructions get marked, and then all non-critical instructions get swept away by being removed from the block's list of instructions.

1.5 Code Generation and Register Allocation

1.5.1 Code Generation

LLVM generally converts to ARM cleanly, with a few minor modifications here and there. For instance, there are no more struct definitions anymore, and structs are simply allocated memory spaces. To access a struct's values, ARM uses a byte offset into that allocated memory. Another interesting difference is the way comparison conditions (i.e. less than, greater than) are handled between LLVM and ARM. LLVM keeps the conditions in the compare function and then branches based on the result of that compare function, ARM does the compare and then keeps the condition in the branch. To overcome this difference, we made icmp instructions translate to a CMP followed by a CSET, which can use the condition to set a register to either 1 or 0 depending on whether or not the condition is met. Then, the following branch turns into a comparison of that register to 1, and a branch to the then if it's equal to 1 and a branch to the else if not.

Phi instructions must be eliminated because ARM doesn't have a direct equivalent. This is largely because in LLVM, each variable is assigned its own register as if there are infinite registers, whereas ARM has a limited number of registers that must be shared and allocated accordingly. As a result, our translation to ARM has to find a way to accomplish a similar functionality to phi instructions. We do this by simply moving all of the phi instruction register choices into their respective predecessor blocks as a mov instruction into the result register of the phi instruction. For while header blocks, instead of placing the instructions in the loopback block (the block at the end of the loop that points back up to the while header), we create a new split block between the loopback and while header block and place mov instructions in it that ensure the parallelism of phi instructions. This is done by first creating temporary registers to store the initial values of the phi "source value". Then for each temp register, it is put into its respective phi result register. This way, any phis that use another phi as a source value are using its original value instead of its new value.

1.5.2 Register Allocation

For register allocation, we followed the algorithm provided in class, which does live range calculation, then builds an interference graph, then colors that graph, and for any node that can't be colored, we do register spilling onto the stack.

The first step of live range calculation is to generate the "gen" and "kill" sets for each block, gen being the set of registers that are source operands and don't get killed, and kill being the set of registers that are target operands (registers that get overwritten). After we generate the gen and kill sets, we generate the liveout set, which is basically the set of registers still alive at the end of the block, calculated by taking all the registers in each successor liveout, removing all the registers in each successor kill set, and then adding all registers in the successor gen set.

After generating the liveout set for each block, we generate the interference graph by creating a "live now" set that is initially the liveout set, and then for each instruction (starting from the bottom of the block) we add an interference edge between the target registers of the instruction and the livenow registers. We then add the instruction's sources to livenow and keep going up the block. Once we get all the edges, we construct the interference graph for the block. We then attempt to color this graph, and in any situation where we cannot color a node (meaning there are no remaining non-conflicting physical registers), we spill the register onto the stack using loads and stores. When an instruction attempts to use a spilled register, its contents are loaded from the stack using one of two temporary registers (x9 & x10). The instruction is then executed with the temporary registers, and if the result register is also spilled, the result of the instruction is stored onto the stack.

1.6 Other

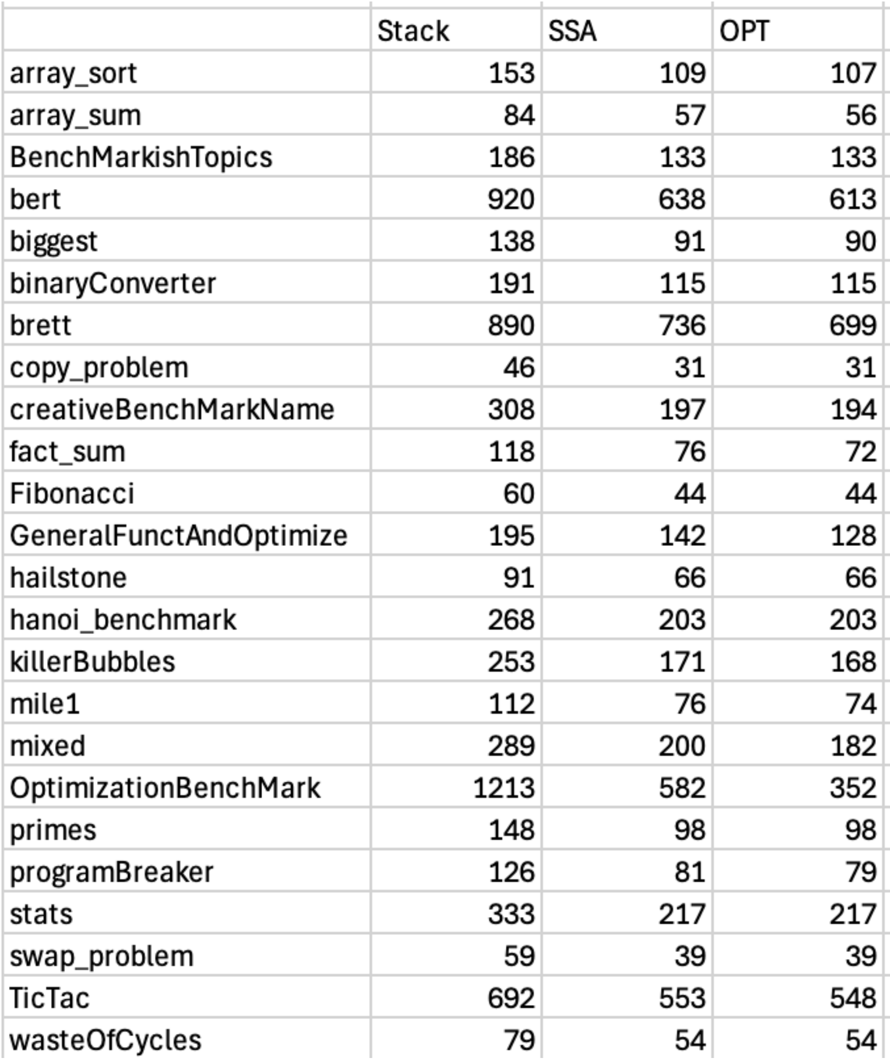
Thanks for a great quarter, Professor Beard. Below is a visual representation of how we felt most of the time while working on the compiler, especially every time bert failed.

2 Analysis

For each of the 24 test cases, we compiled all of the given benchmarks with gcc -O0, gcc -O3, and our compiler with the stack, ssa, and opt flags. We then ran the compiled code and analyzed how long each different version took. The results of said analysis are shown below in the form of bar charts. Each bar chart has a different bar for each compilation method, and the x axis represents time in seconds it took for the compiled program to run on the given input. We used the provided input and input.longer files for each benchmark. The array_sort and array_sum benchmarks don't have a provided C file associated with them, so their graphs do not have -O0 and -O3 included. We compiled and ran the compiled code on the Cal Poly ARM servers.

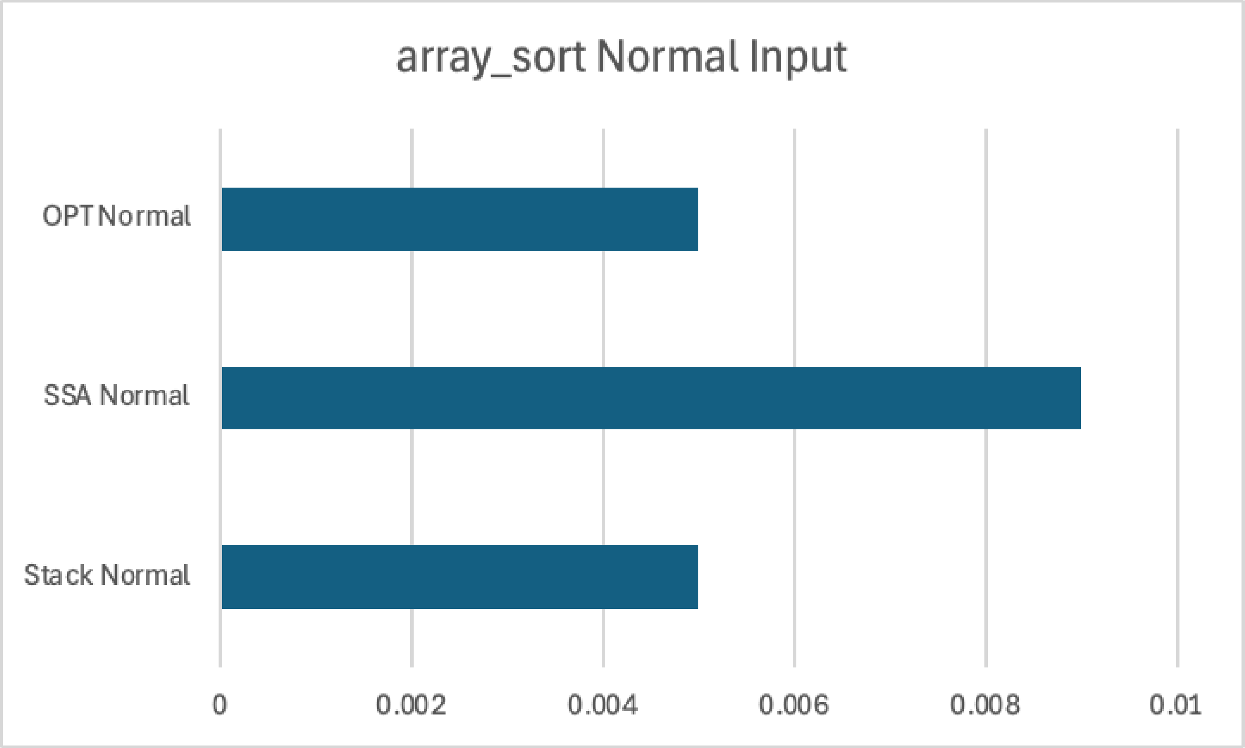
Additionally, below you will find a table showing the number of LLVM instructions for each different compilation version.

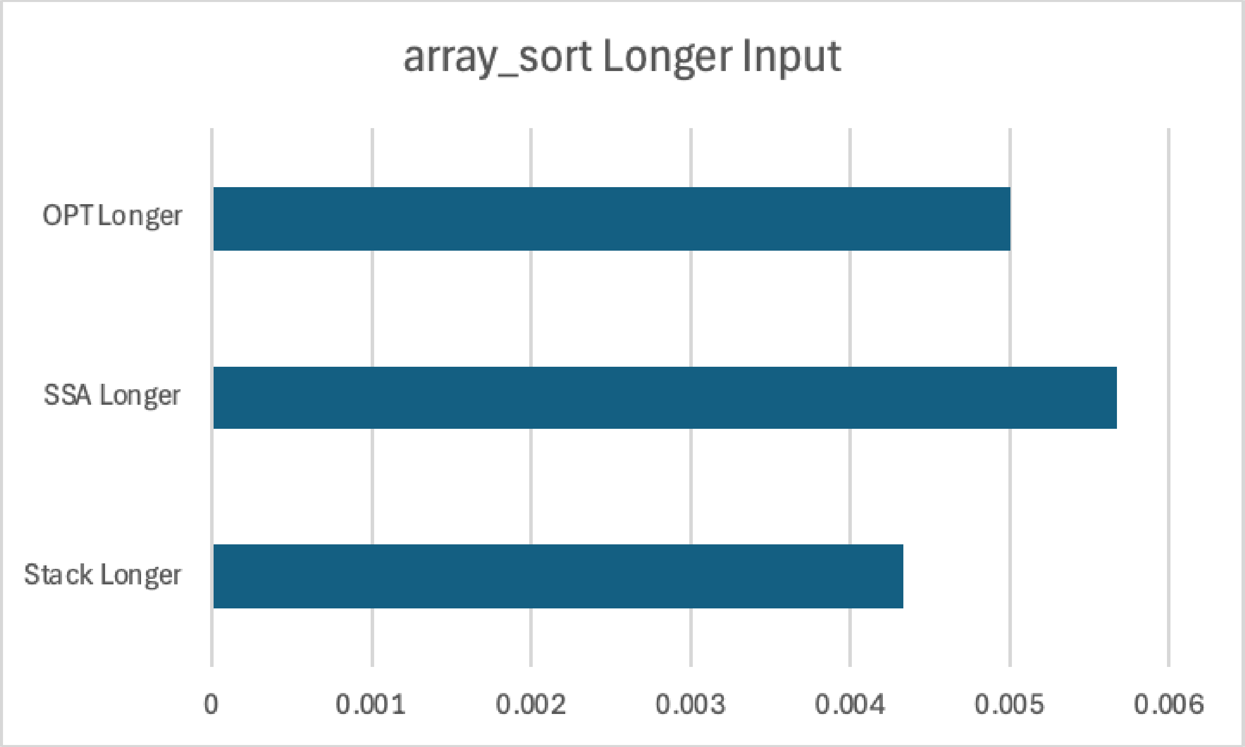
Number of LLVM Instructions



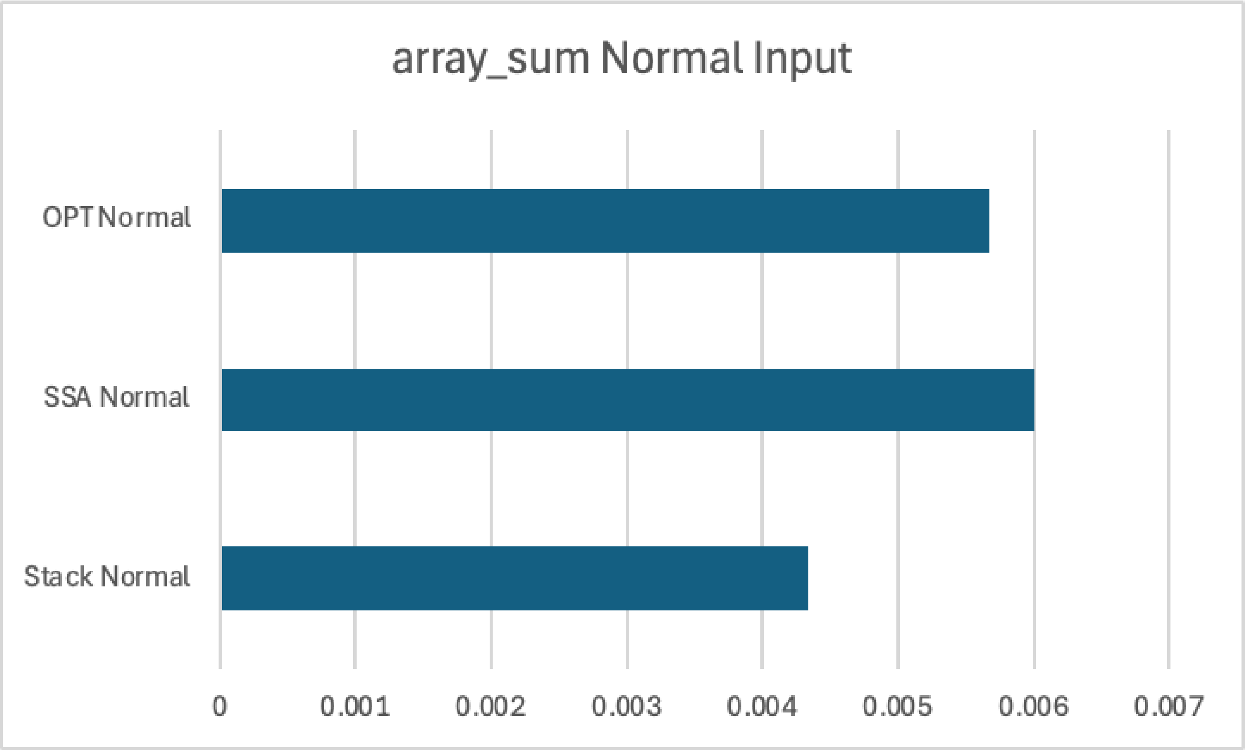
	Stack	SSA	OPT
array_sort	153	109	107
array_sum	84	57	56
BenchMarkishTopics	186	133	133
bert	920	638	613
biggest	138	91	90
binaryConverter	191	115	115
brett	890	736	699
copy_problem	46	31	31
creativeBenchMarkName	308	197	194
fact_sum	118	76	72
Fibonacci	60	44	44
GeneralFunctAndOptimize	195	142	128
hailstone	91	66	66
hanoi_benchmark	268	203	203
killerBubbles	253	171	168
mile1	112	76	74
mixed	289	200	182
OptimizationBenchMark	1213	582	352
primes	148	98	98
programBreaker	126	81	79
stats	333	217	217
swap_problem	59	39	39
TicTac	692	553	548
wasteOfCycles	79	54	54

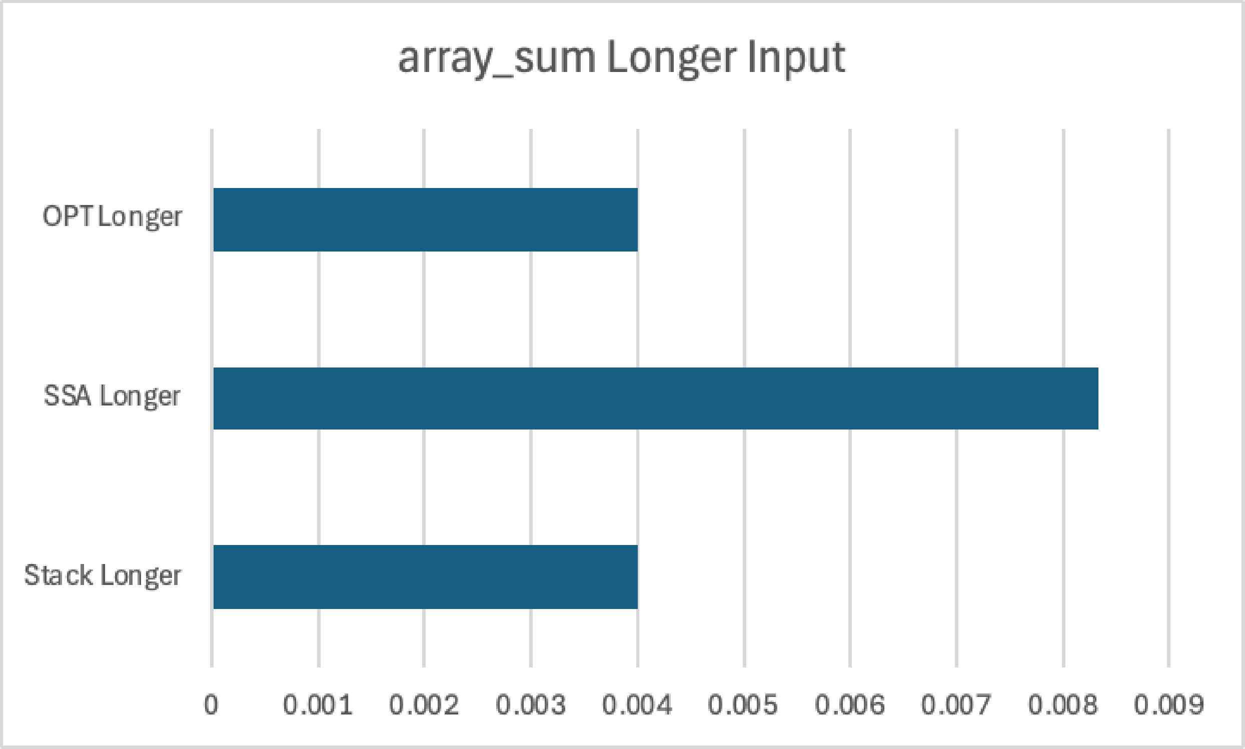
2.1 array_sort



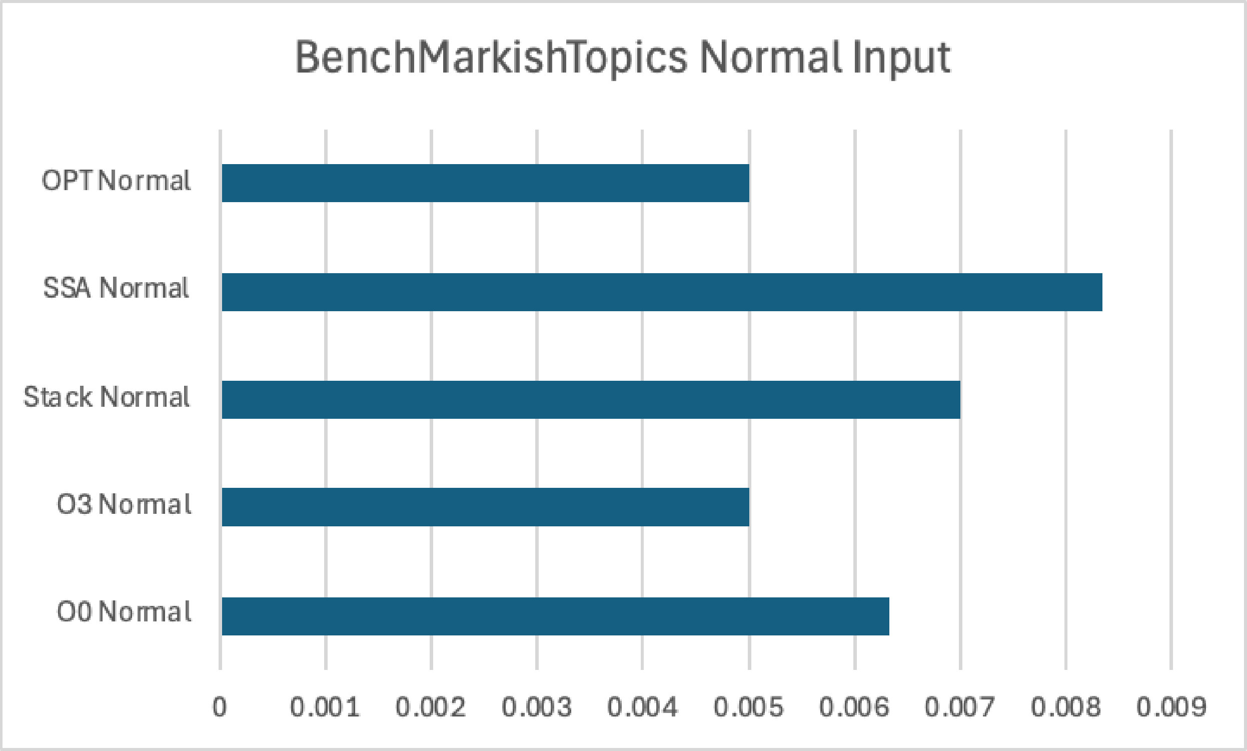


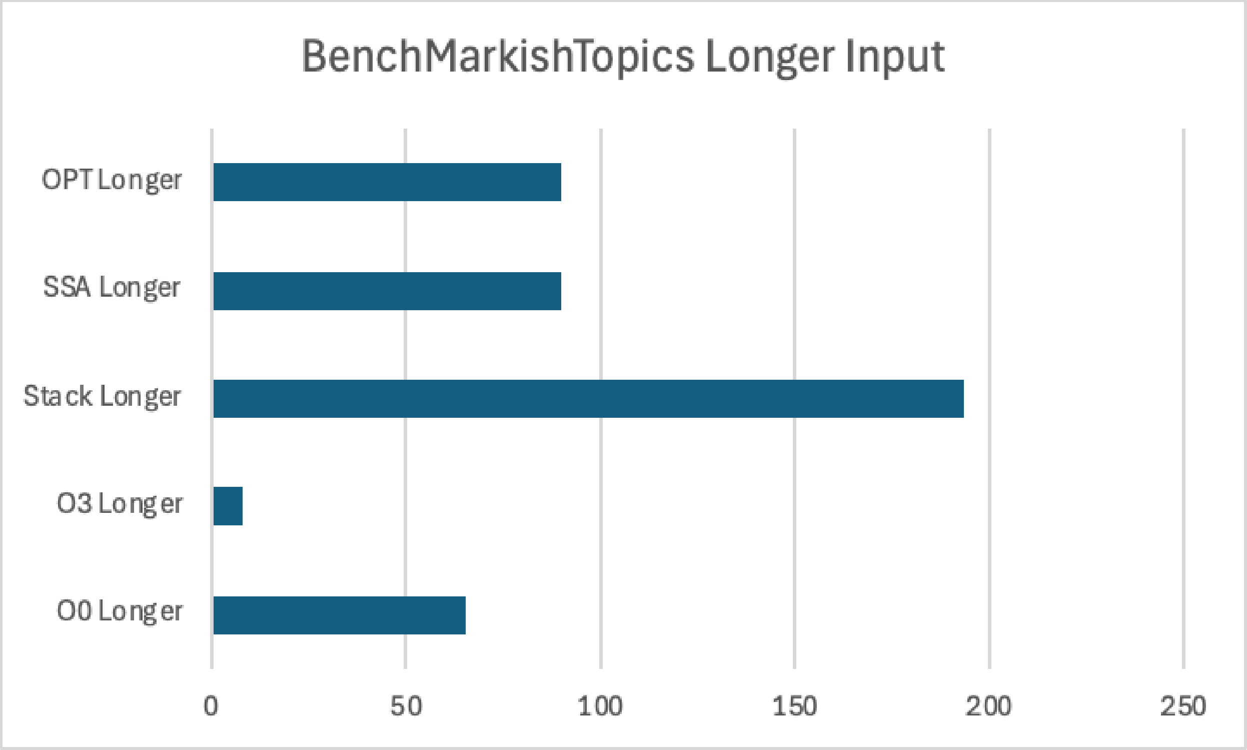
2.2 array_sum


array_sum Normal Input
OPT Normal
SSA Normal
Stack Normal
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007

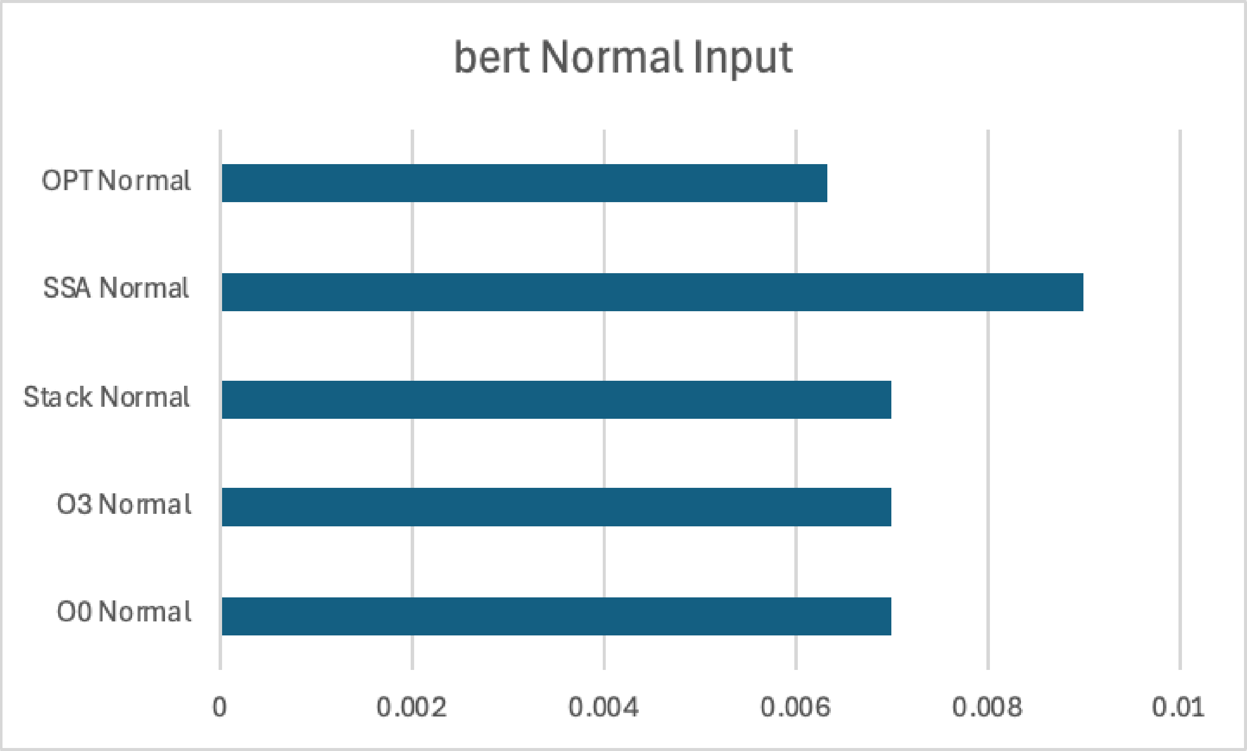

array_sum Longer Input
OPT Longer
SSA Longer
Stack Longer
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009

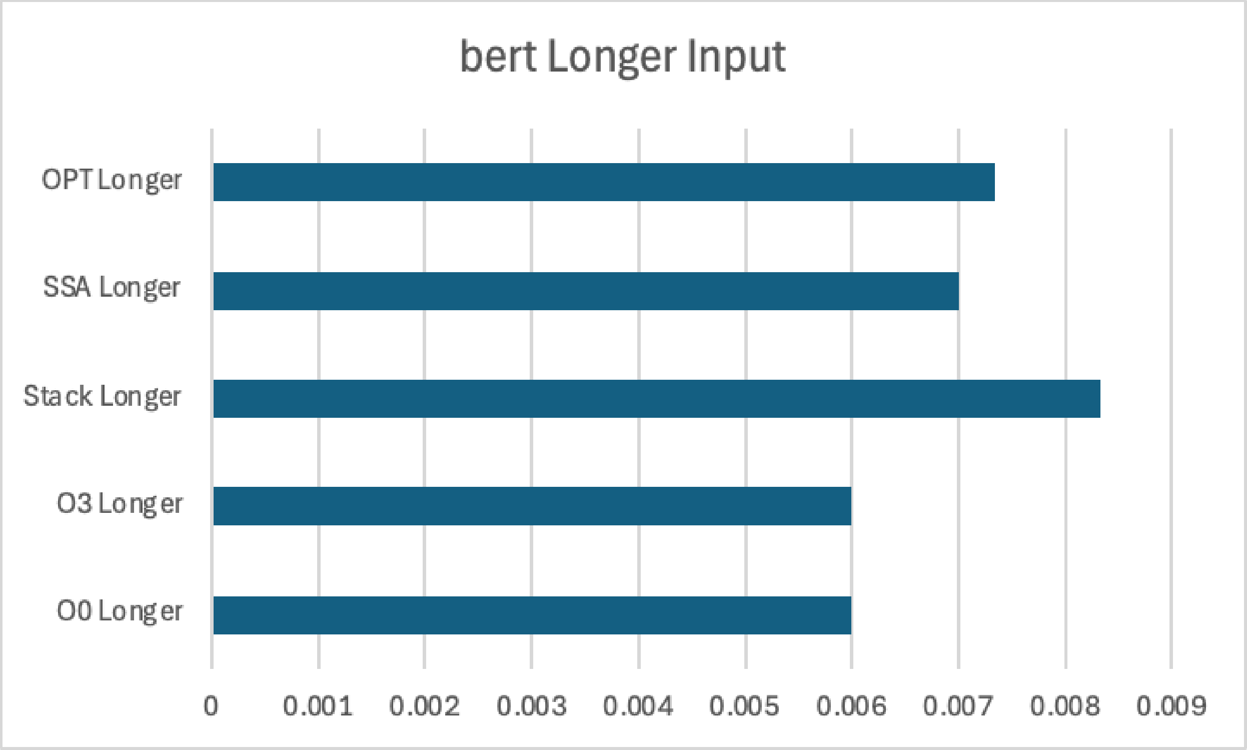
2.3 BenchMarkishTopics



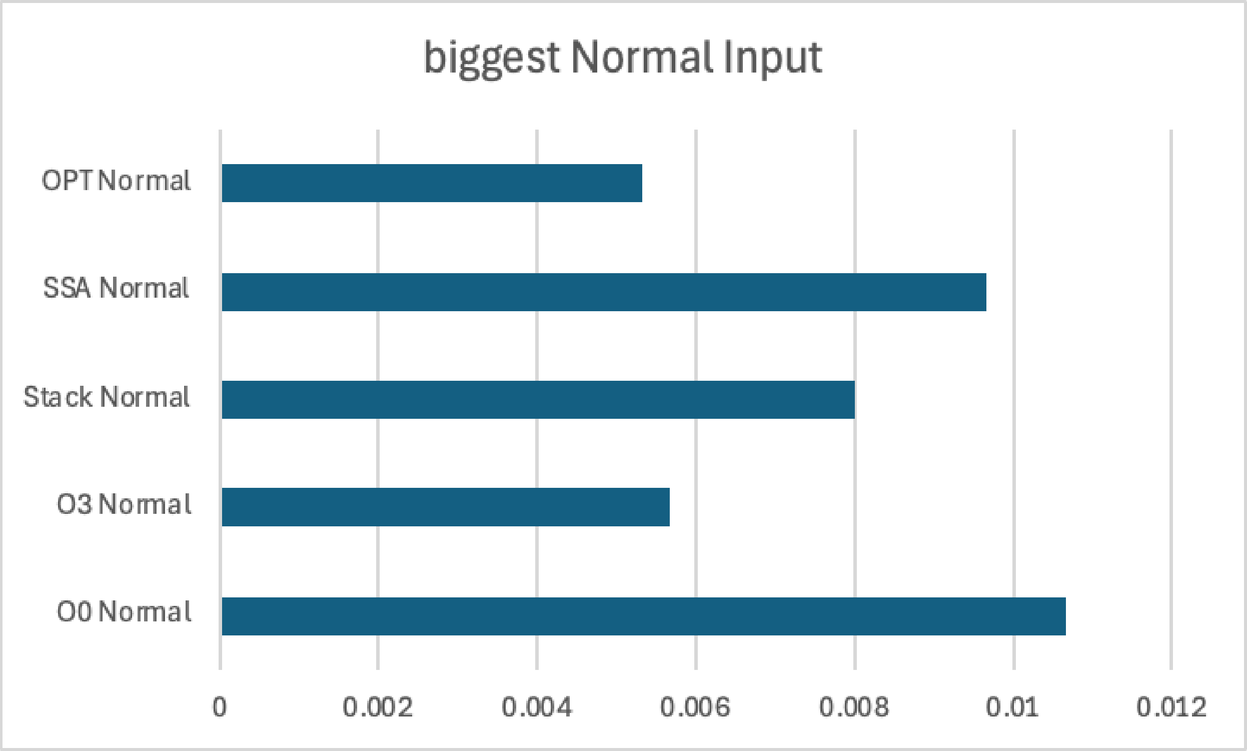


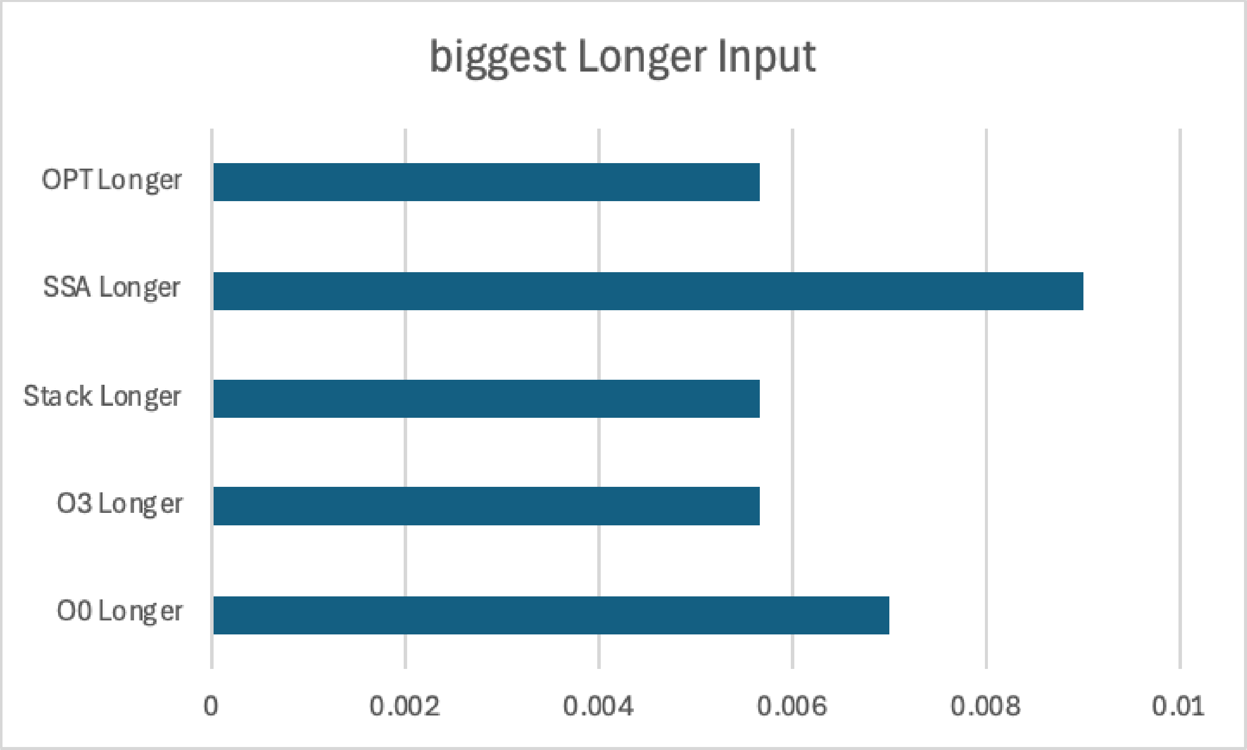
2.4 bert


bert Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.002
0.004
0.006
0.008
0.01

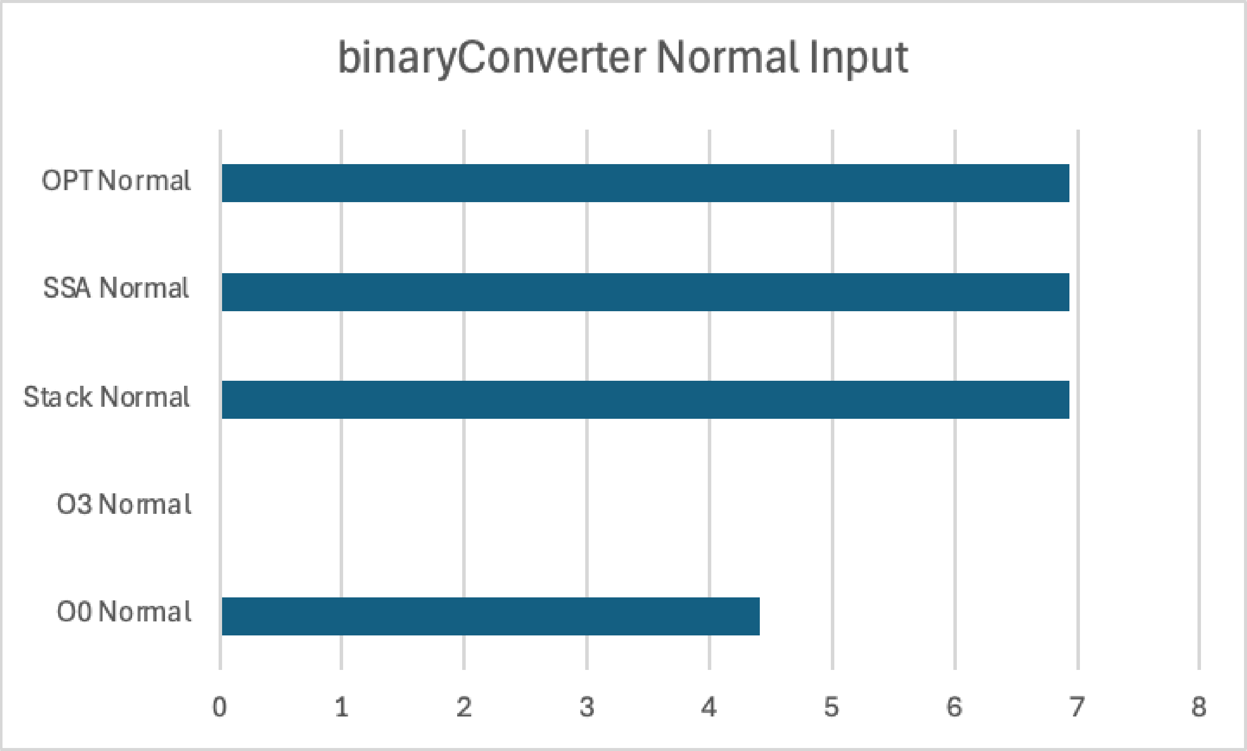

bert Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
0.009

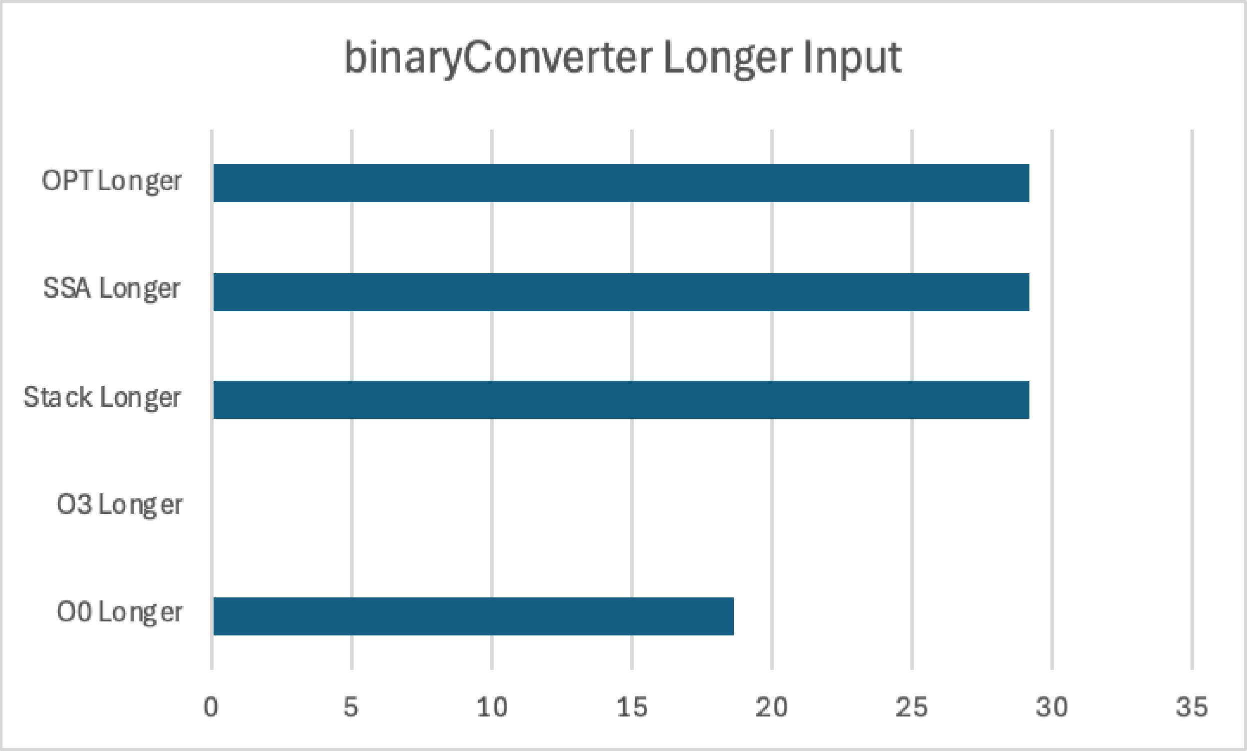
2.5 biggest


biggest Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.002
0.004
0.006
0.008
0.01
0.012

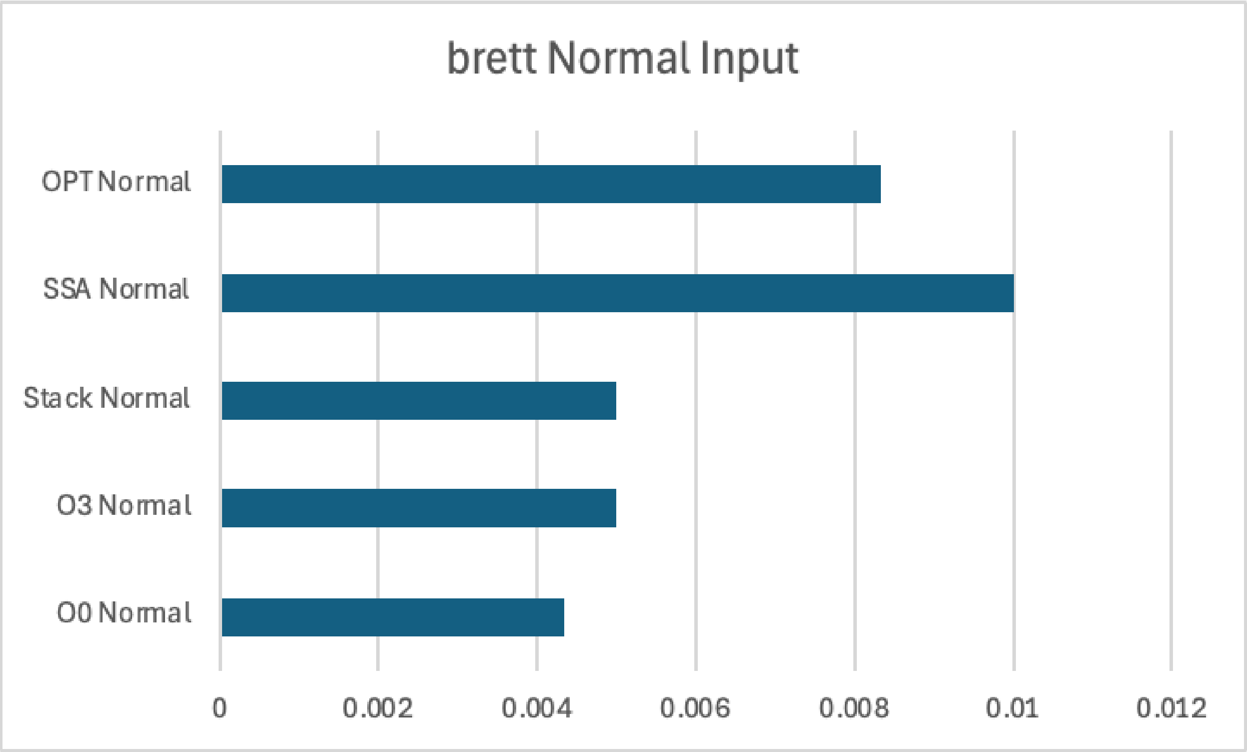

biggest Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
0.002
0.004
0.006
0.008
0.01

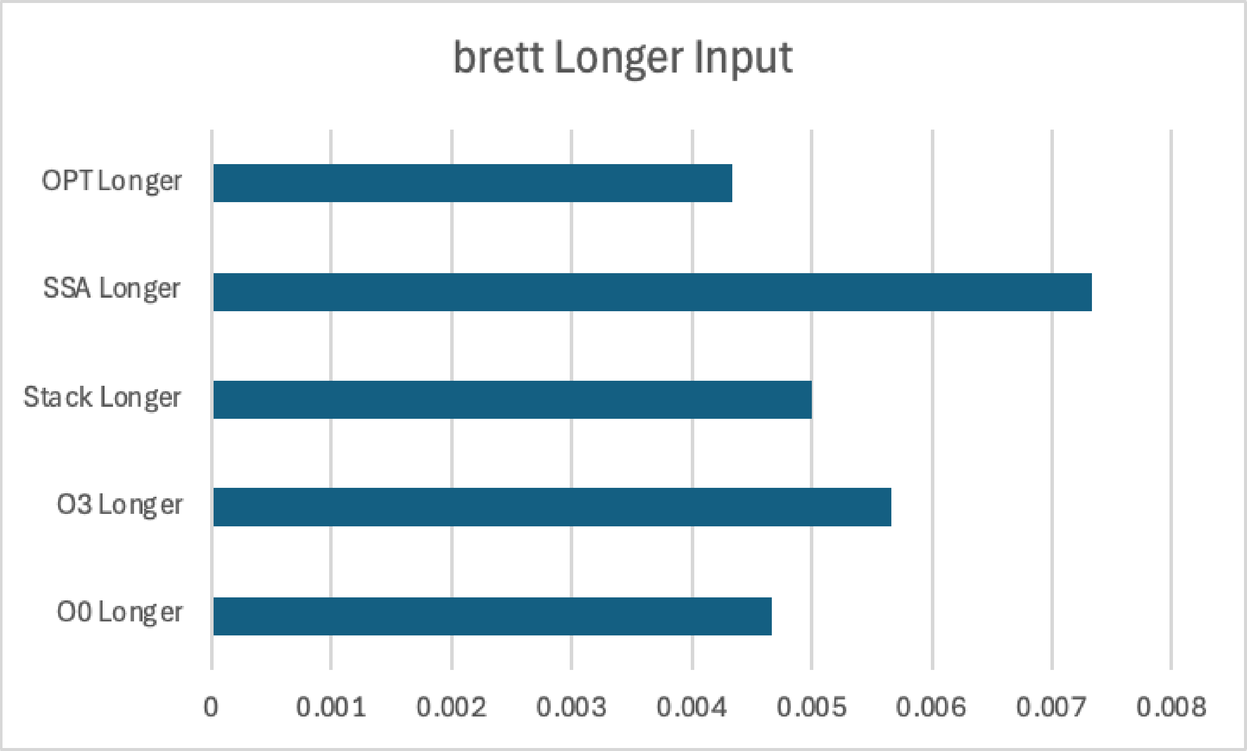
2.6 binaryConverter



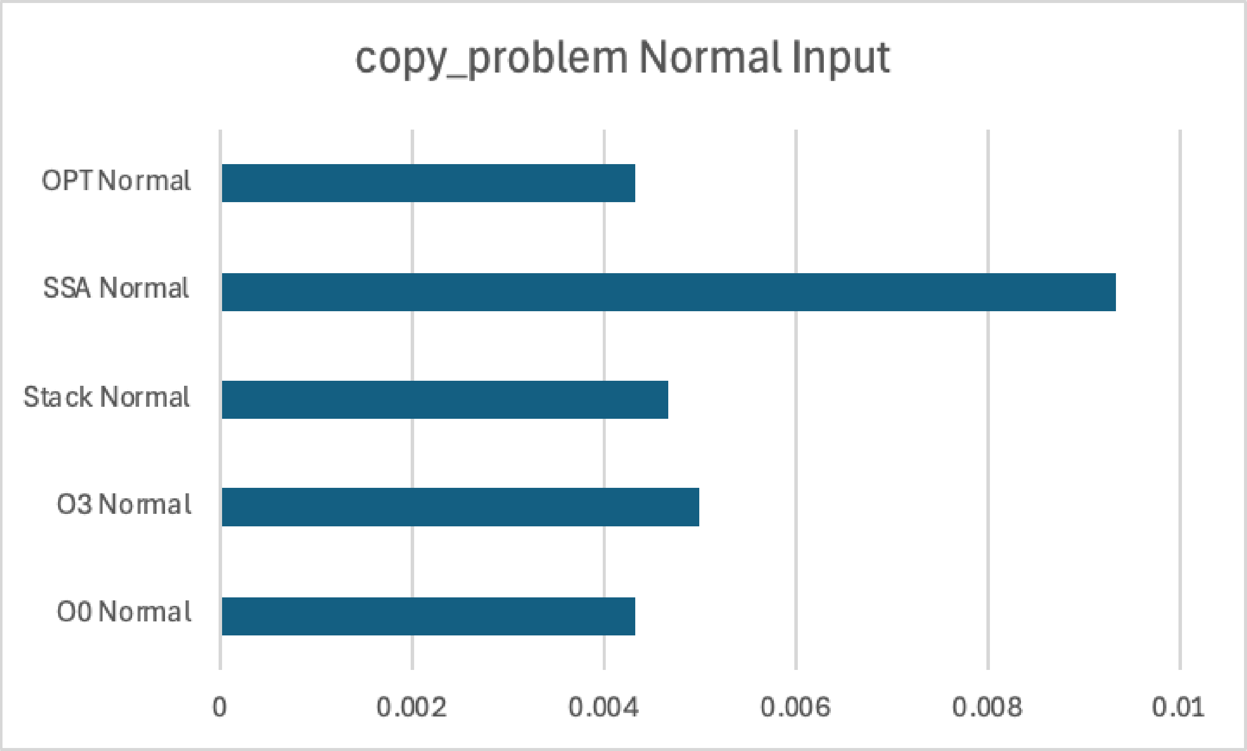


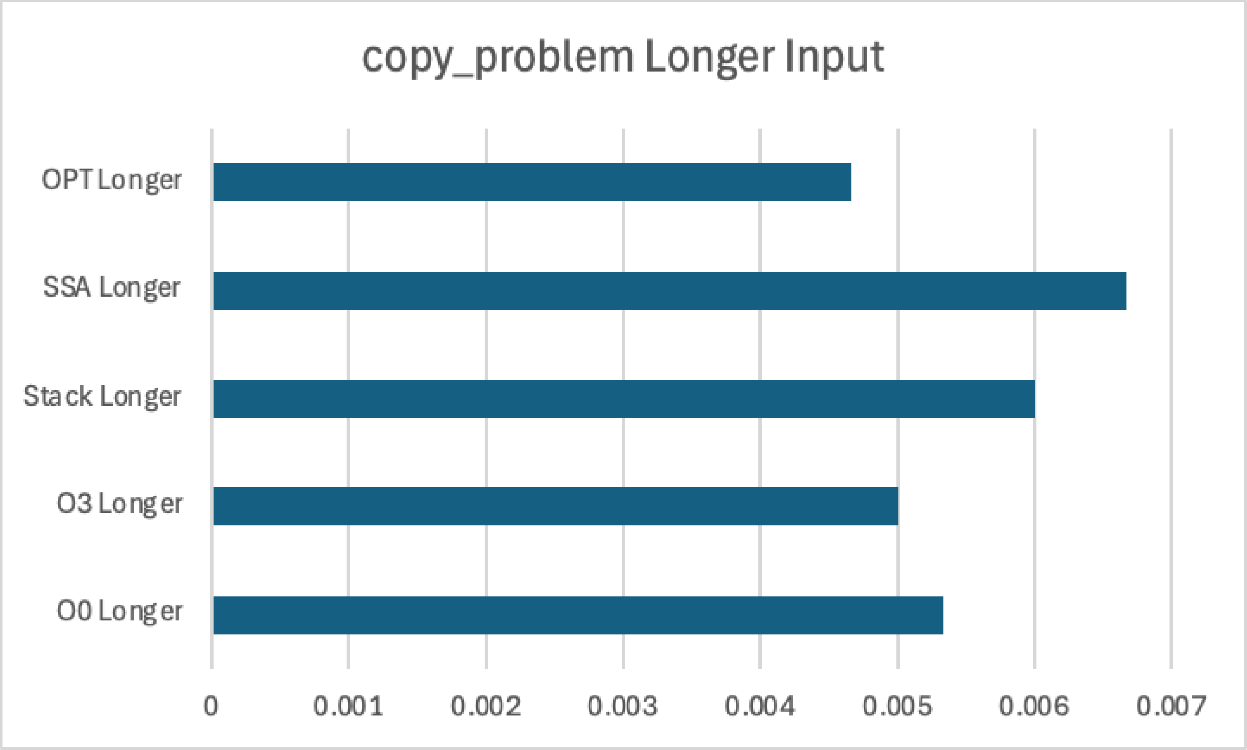
2.7 brett


brett Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.002
0.004
0.006
0.008
0.01
0.012

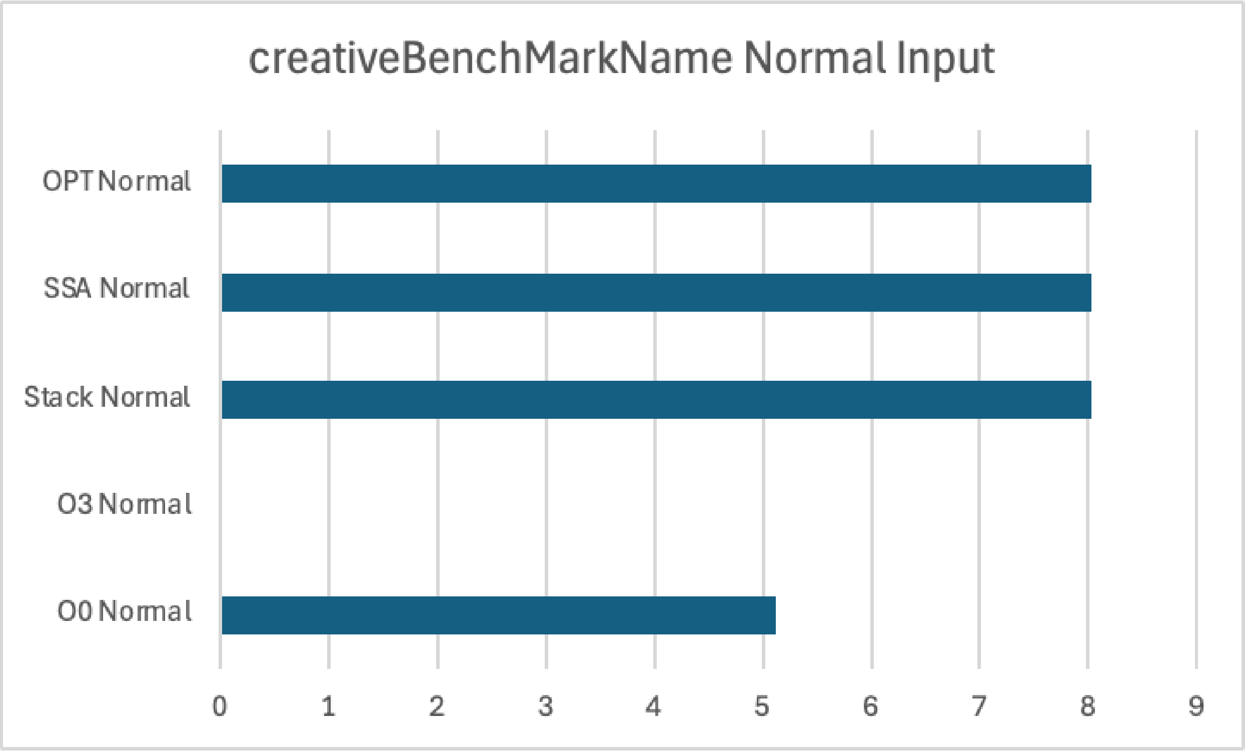

brett Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008

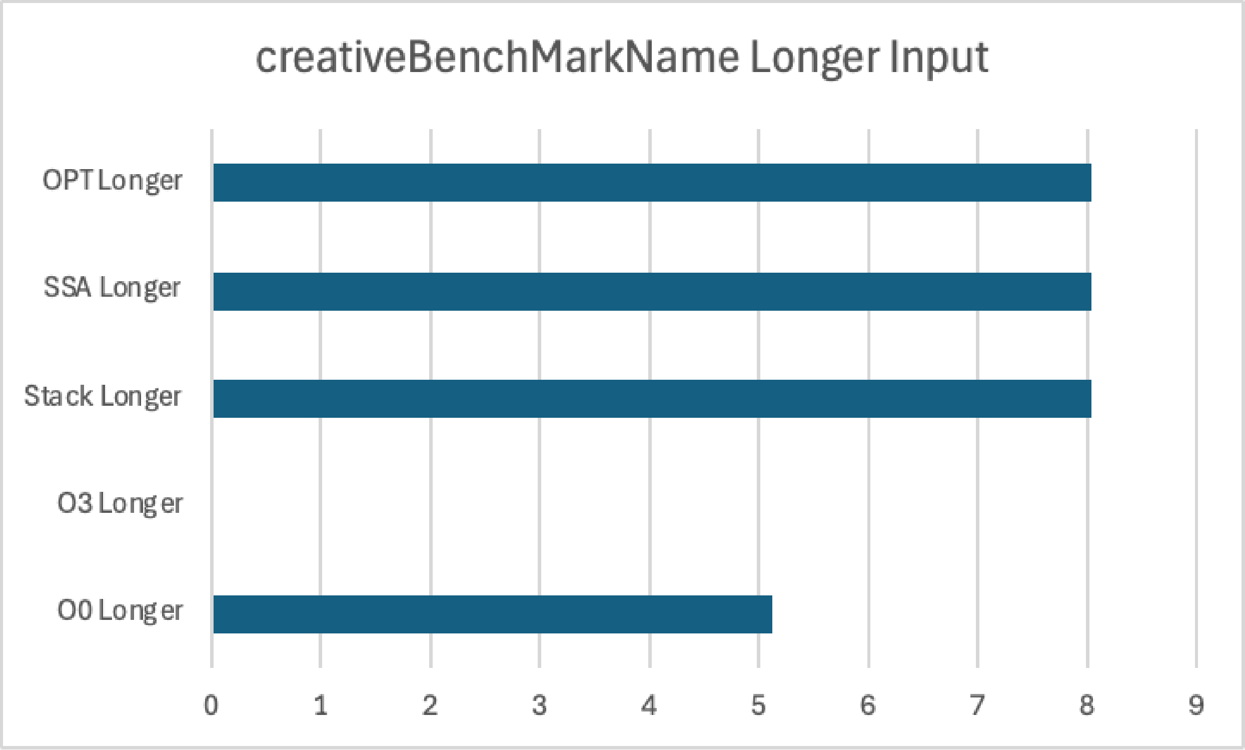
2.8 copy_problem



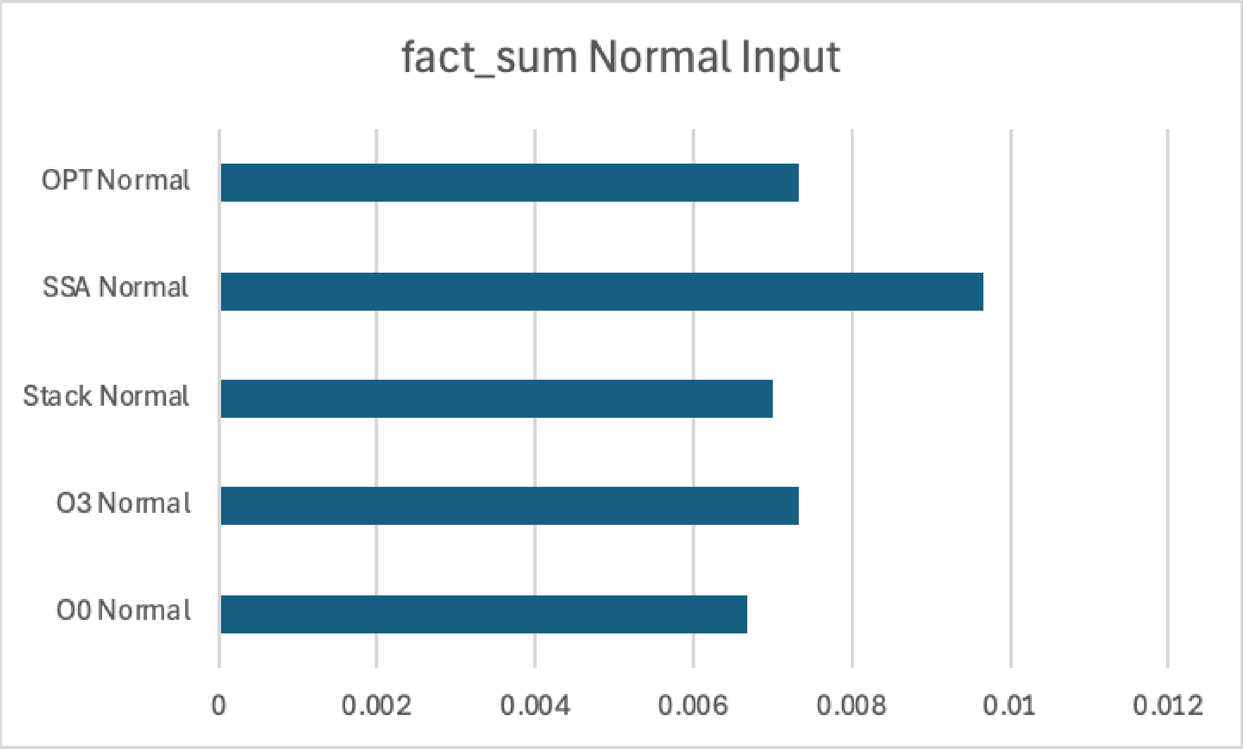


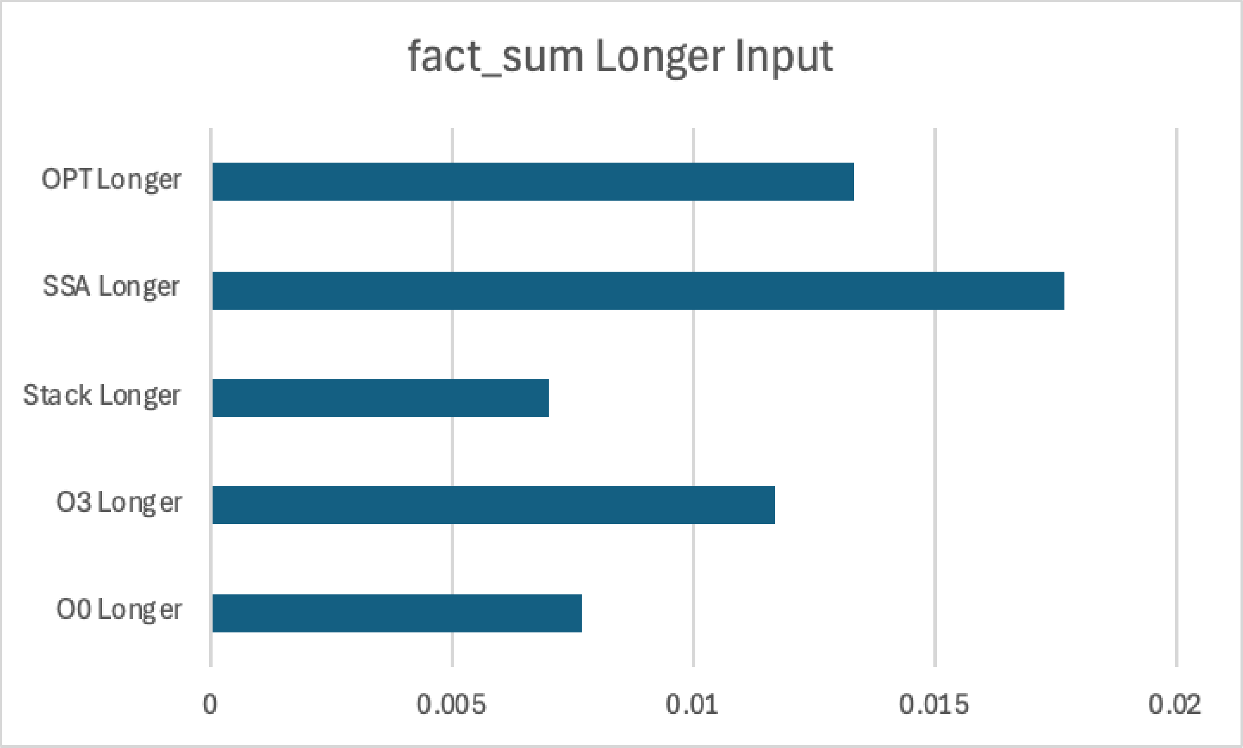
2.9 creativeBenchMarkName



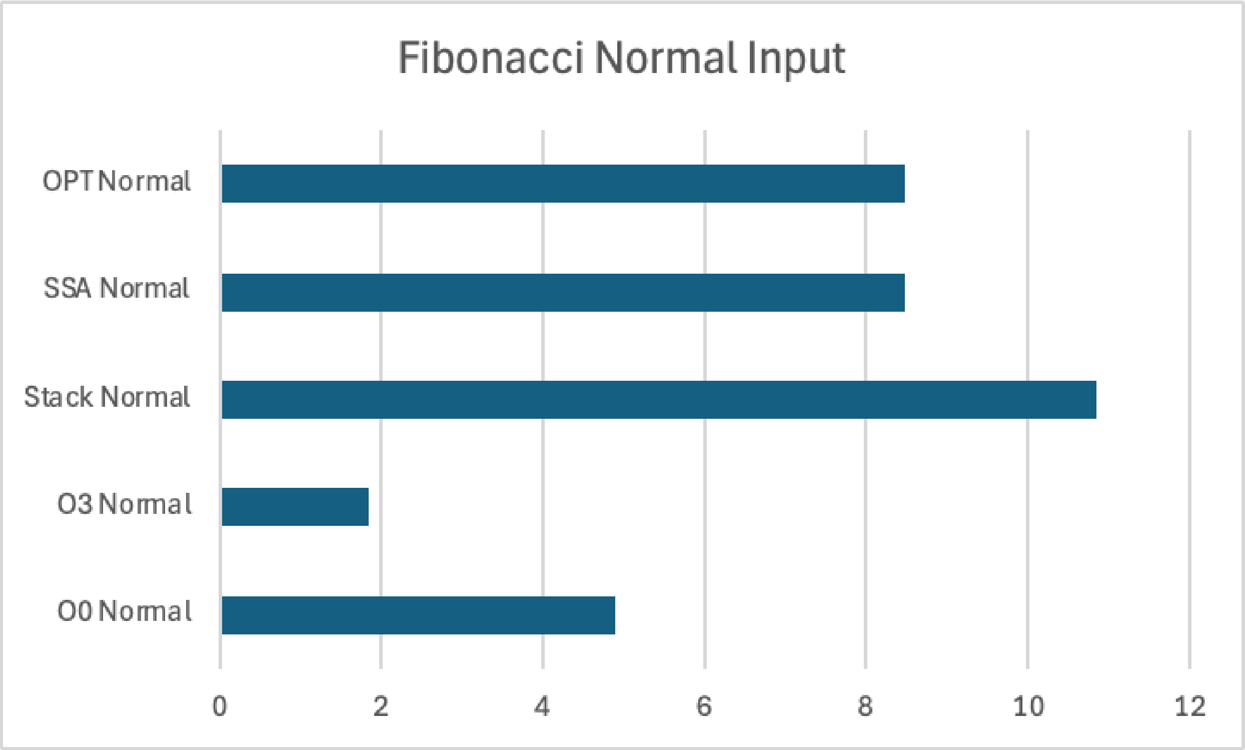


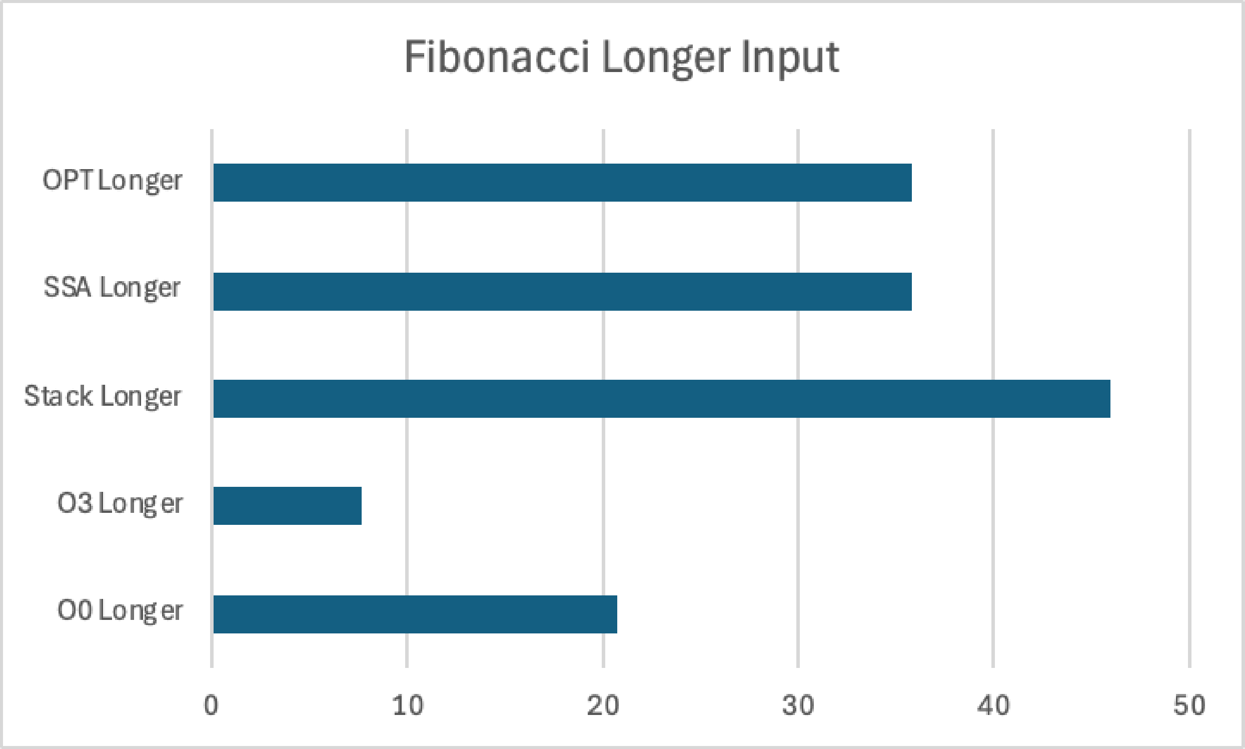
2.10 fact_sum


fact_sum Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.002
0.004
0.006
0.008
0.01
0.012

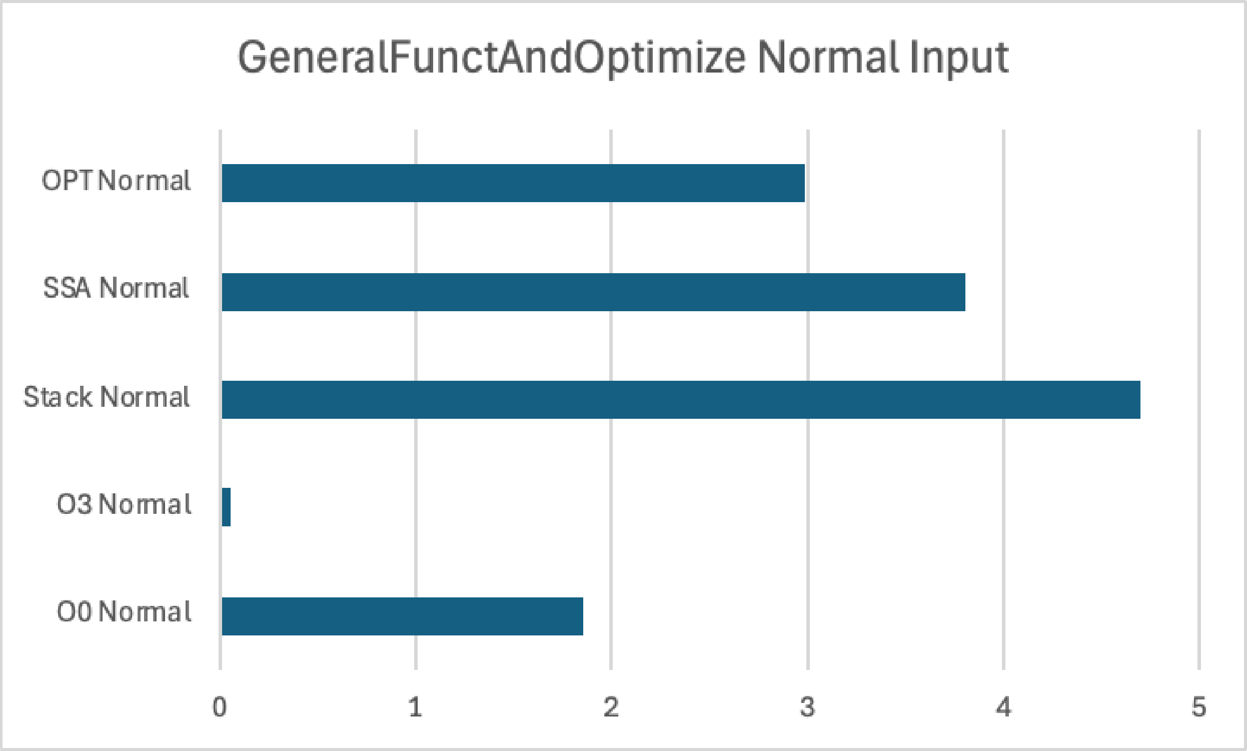

fact_sum Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
0.005
0.01
0.015
0.02

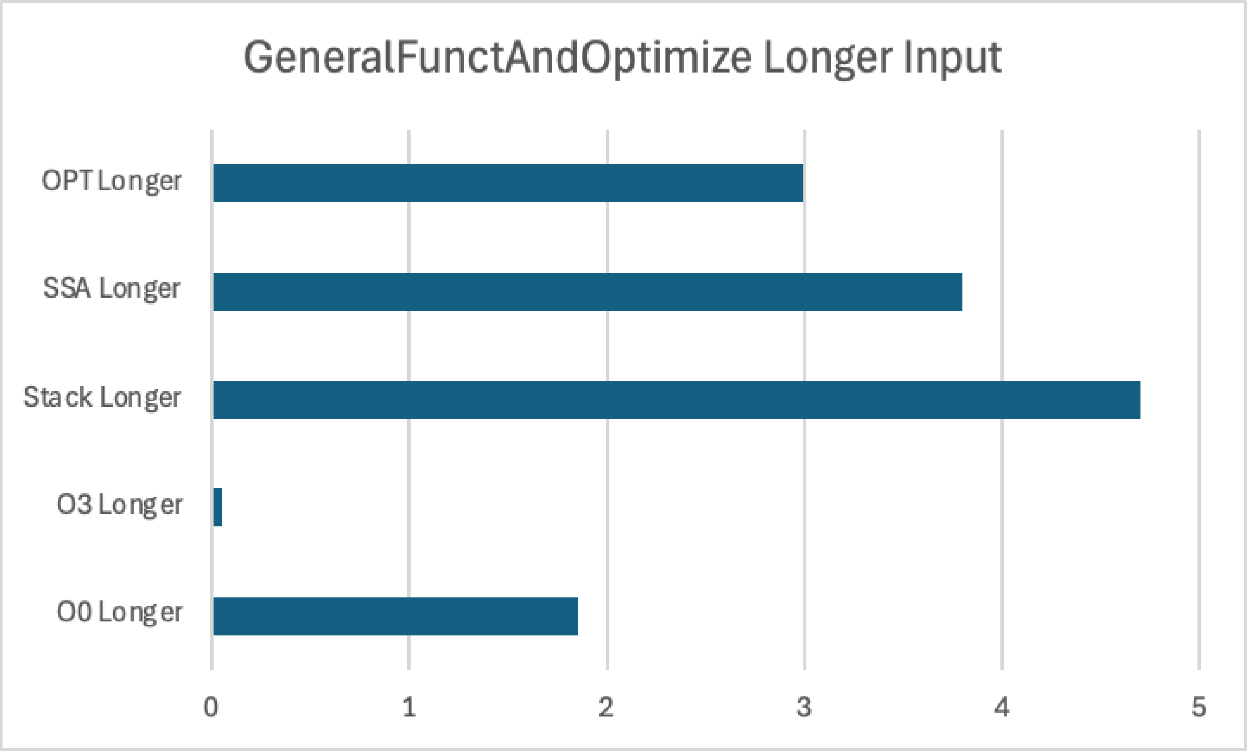
2.11 Fibonacci



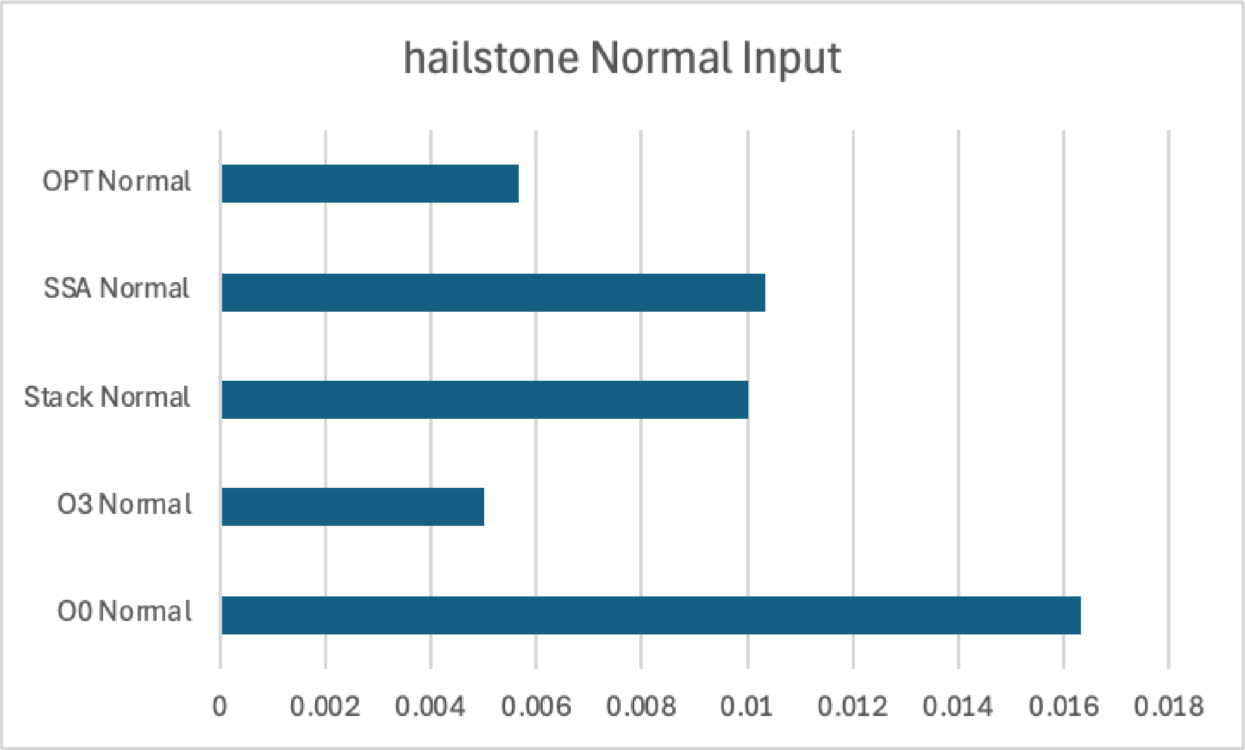


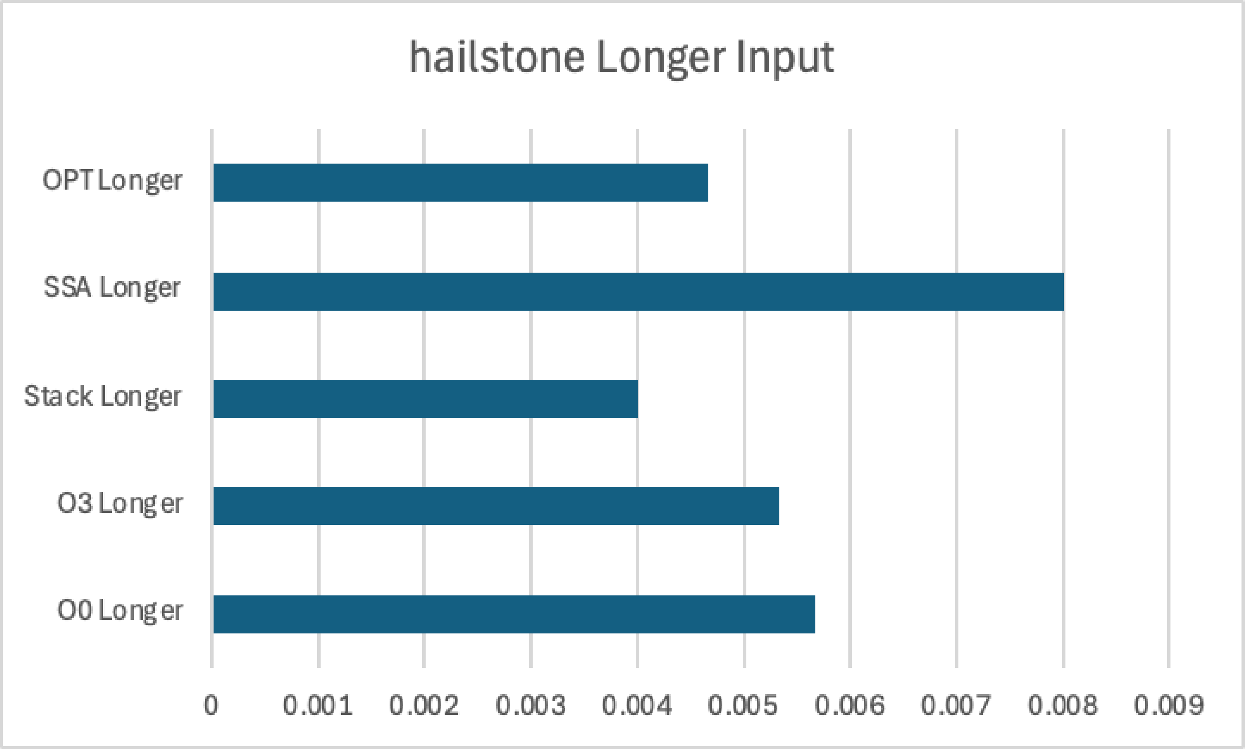
2.12 GeneralFunctAndOptimize



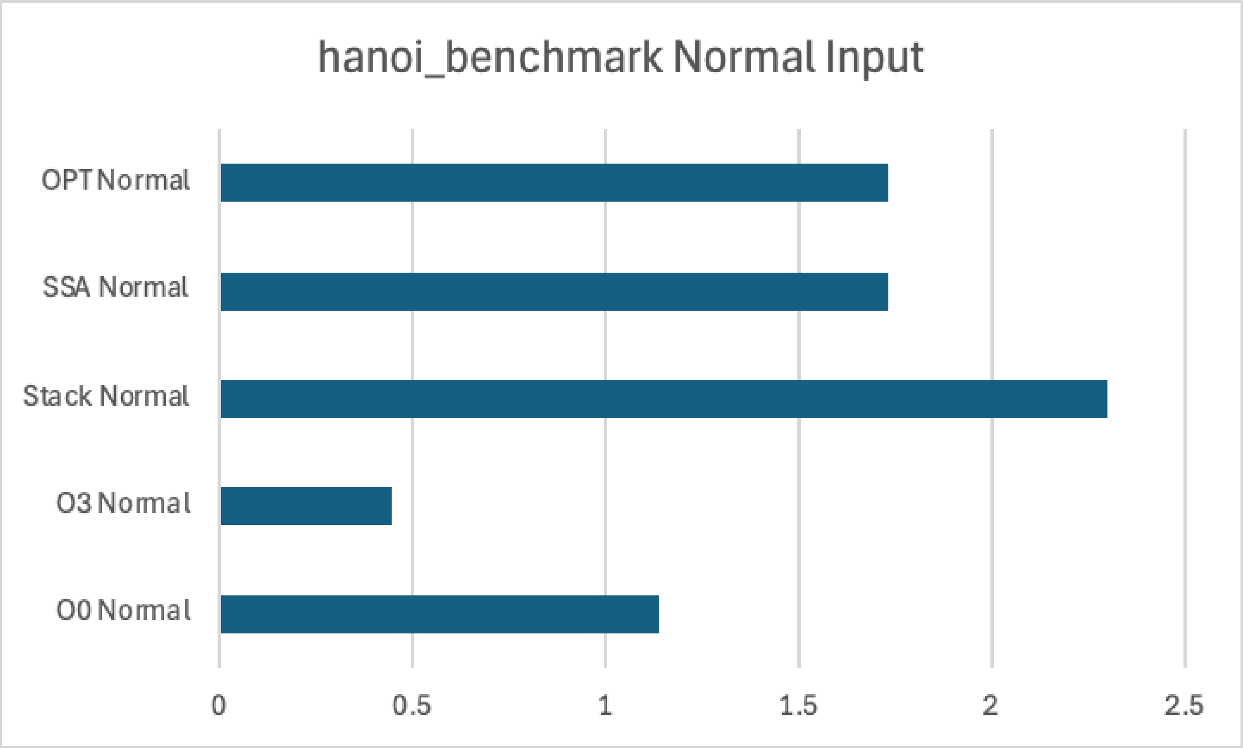


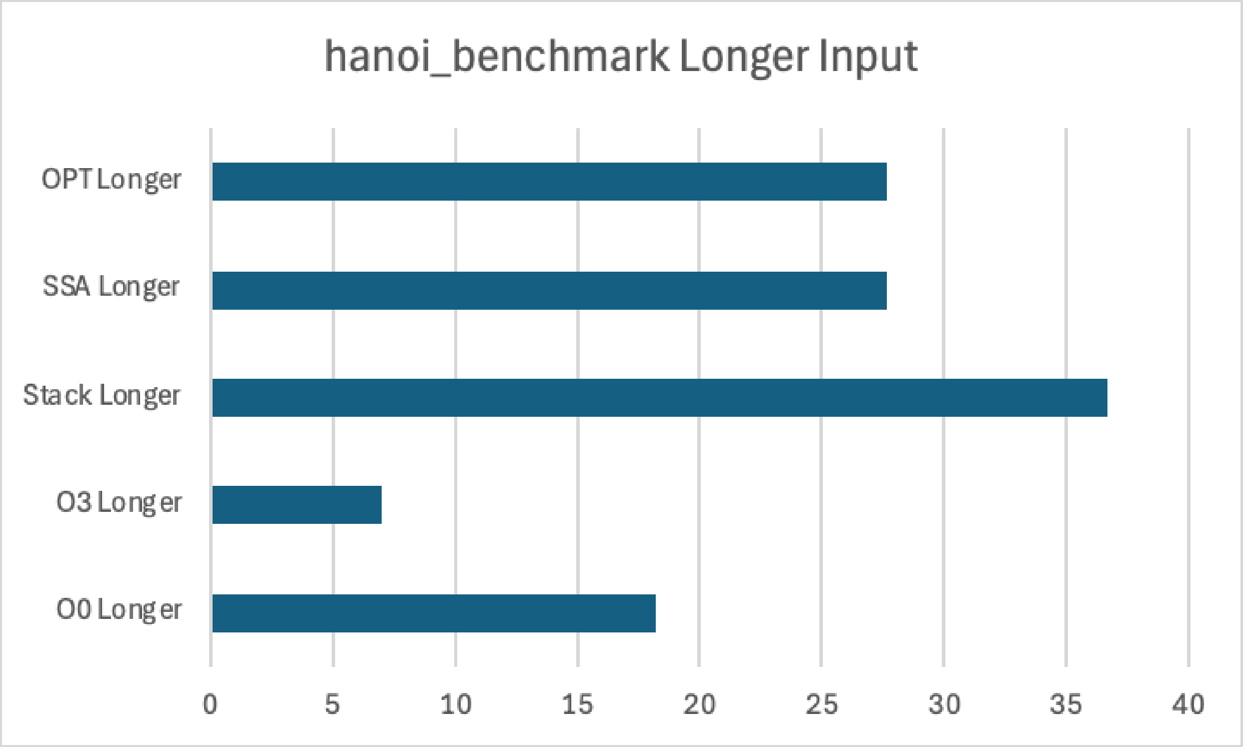
2.13 hailstone



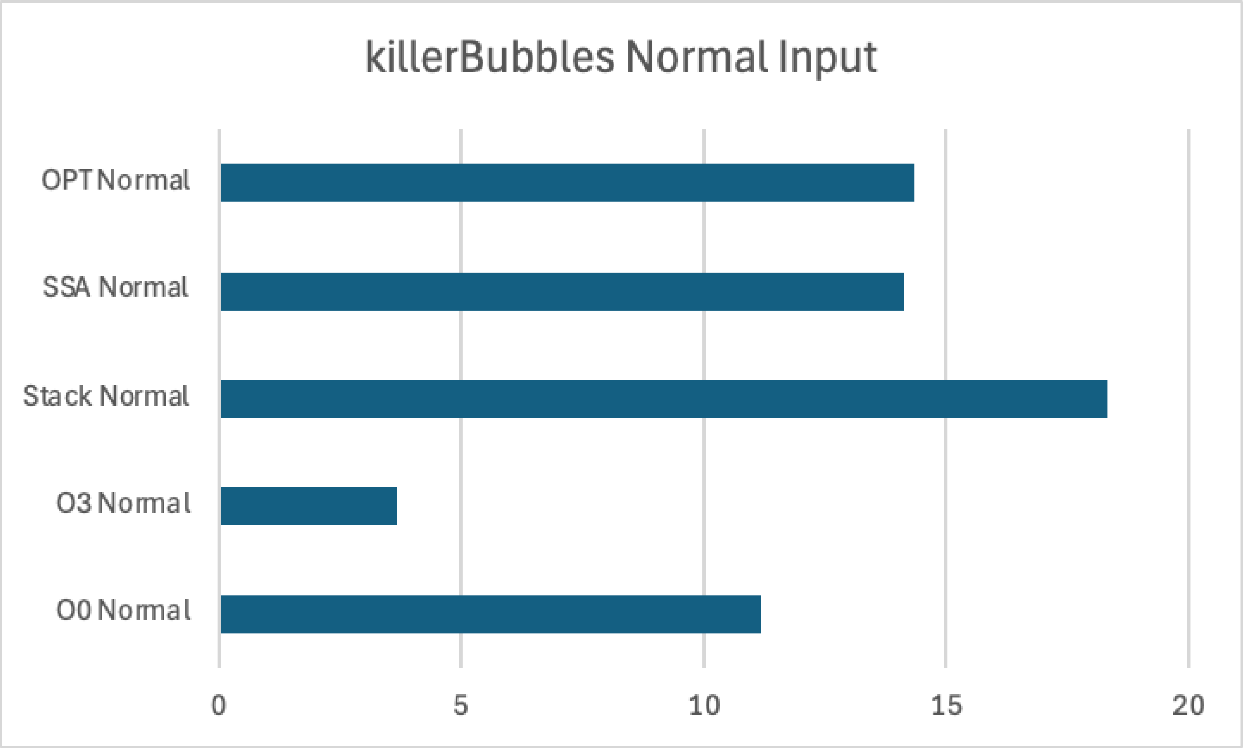


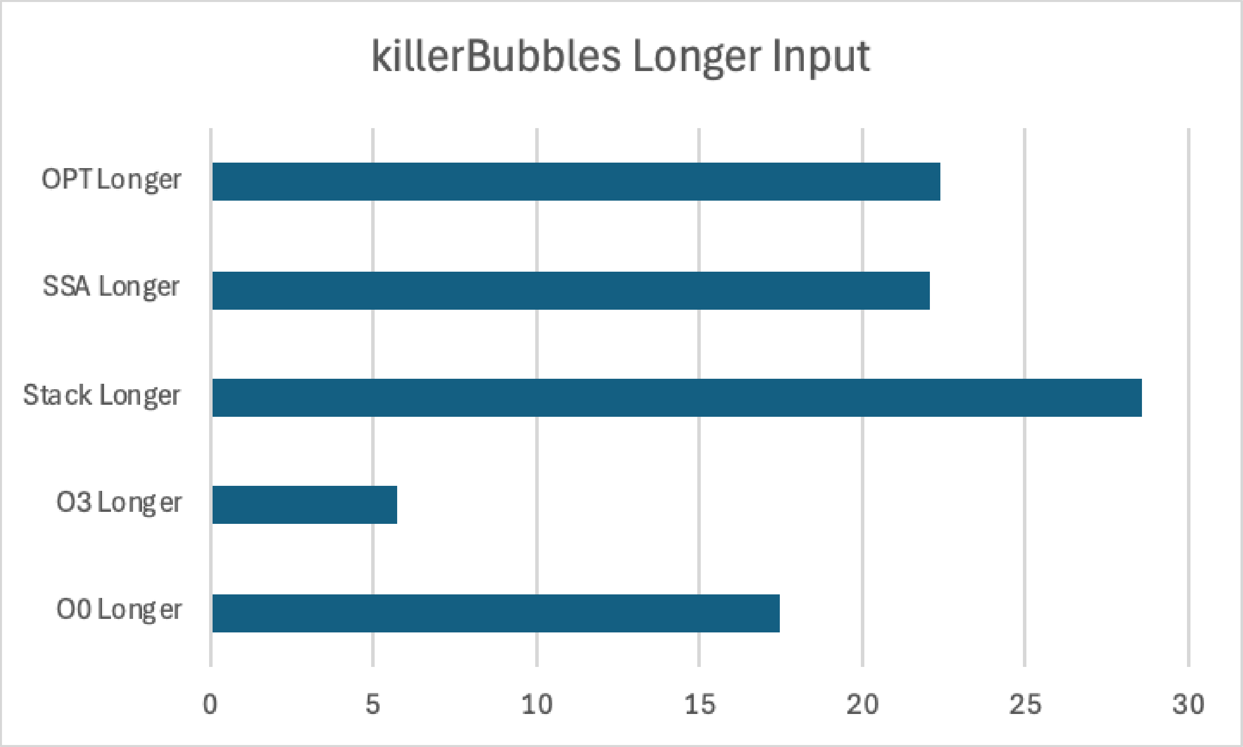
2.14 hanoi_benchmark



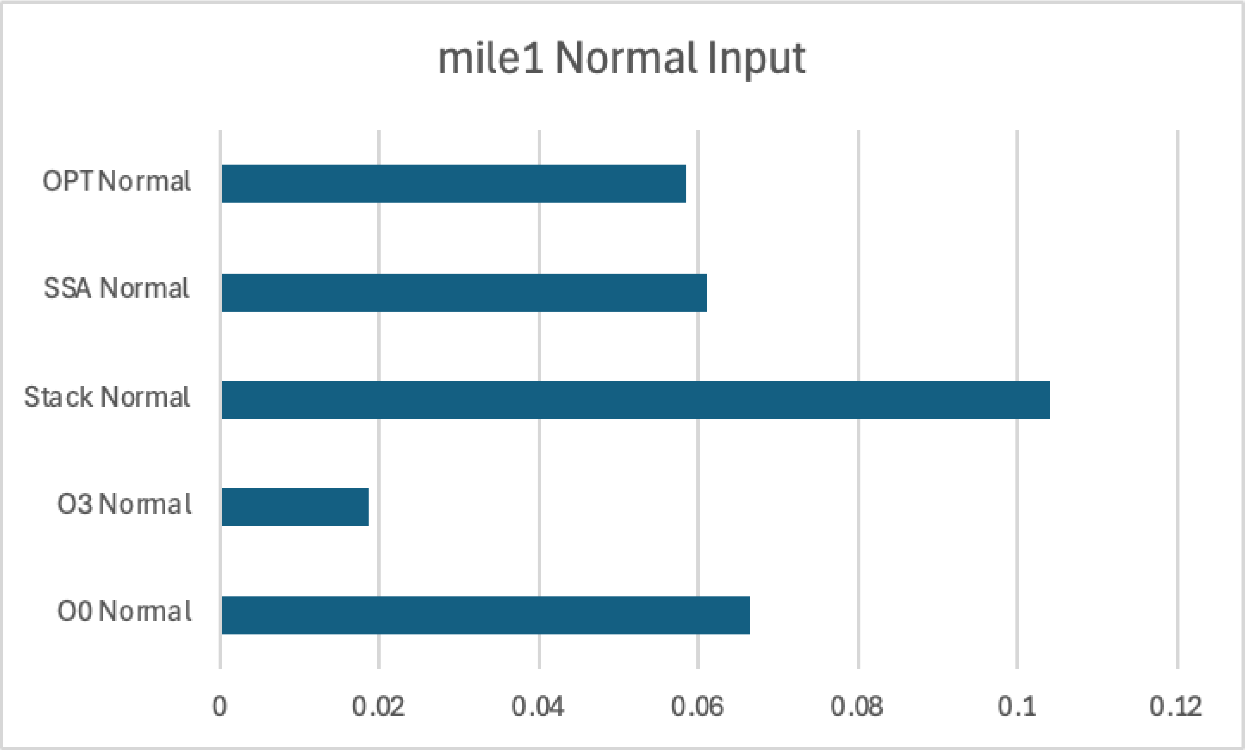


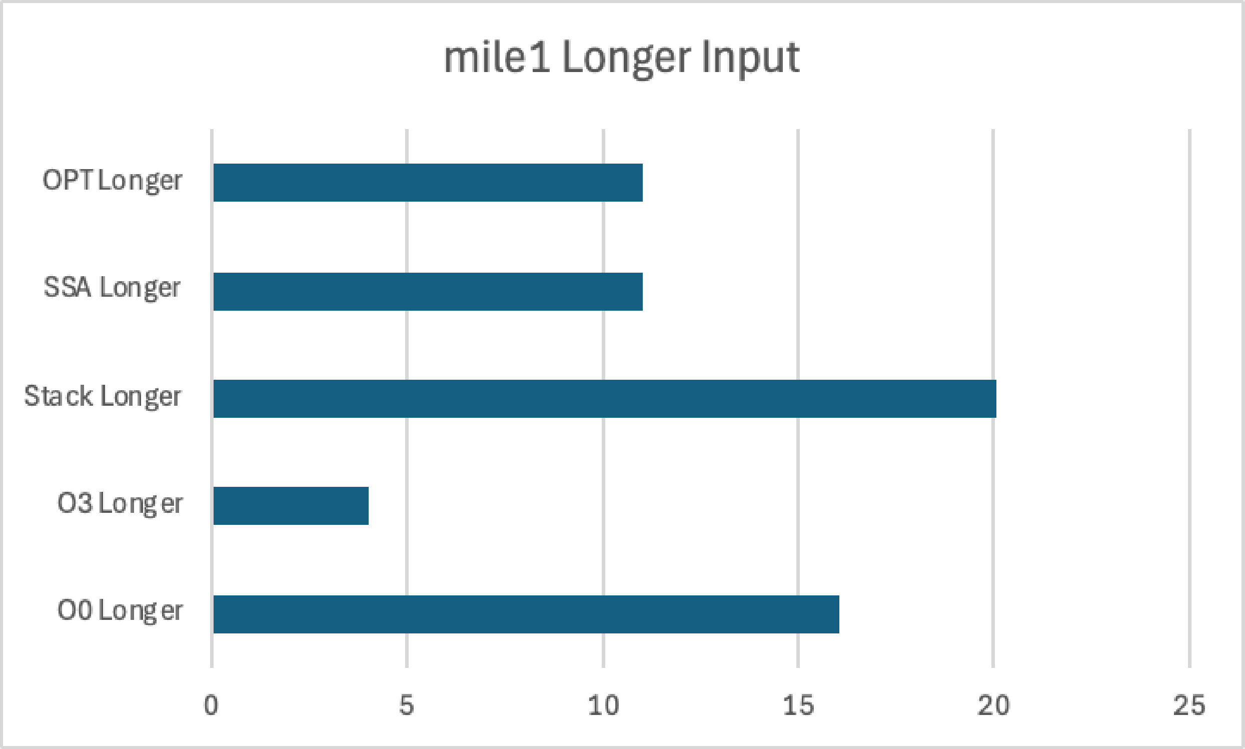
2.15 killerBubbles



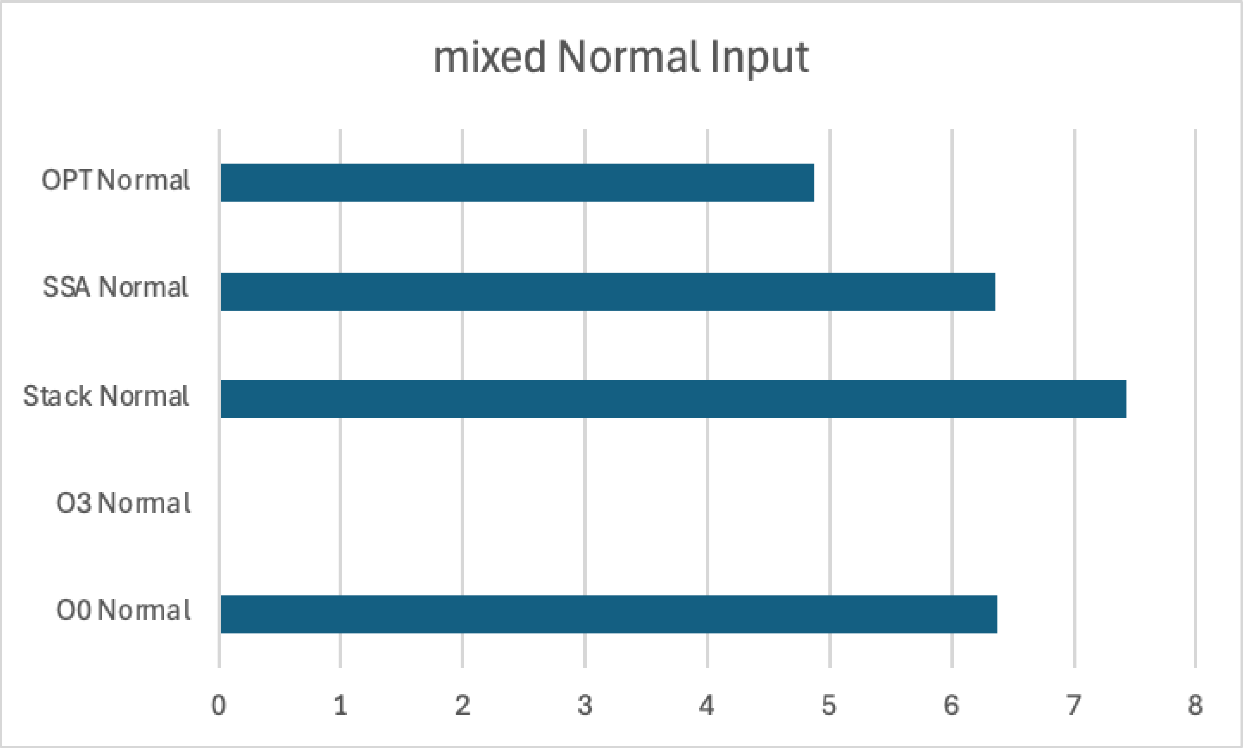


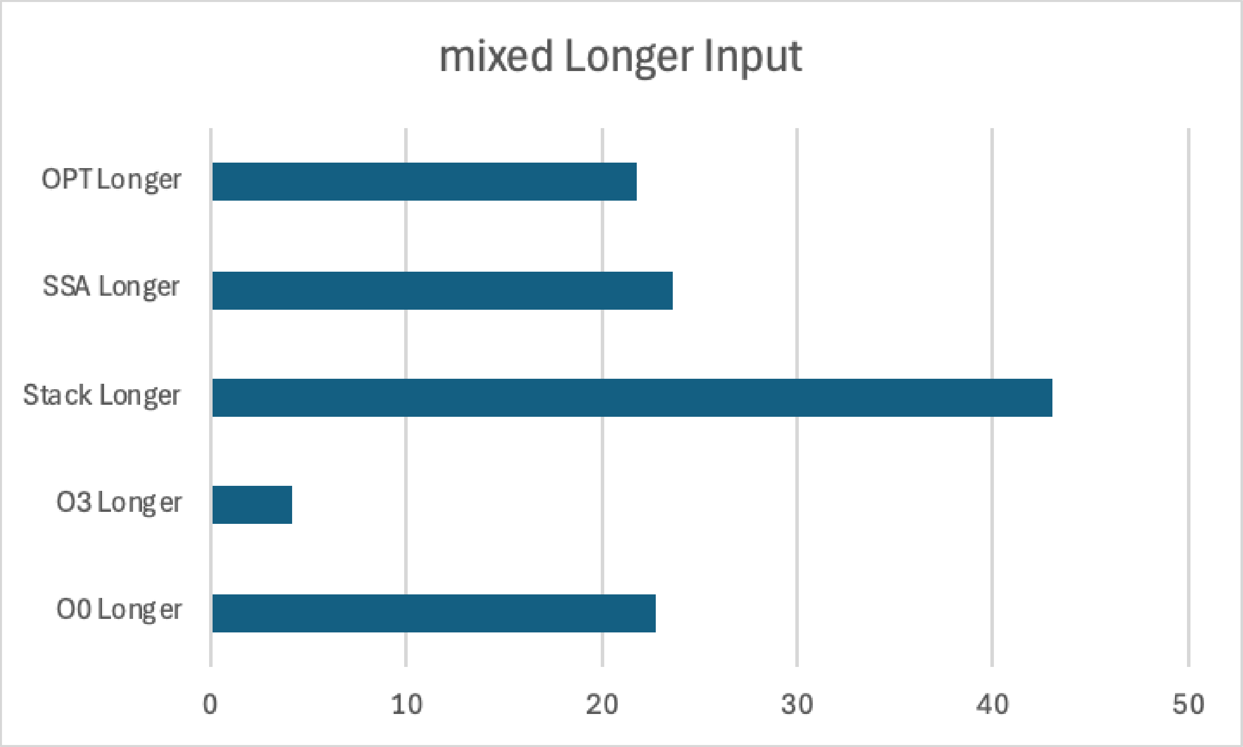
2.16 mile1


mile1 Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.02
0.04
0.06
0.08
0.1
0.12

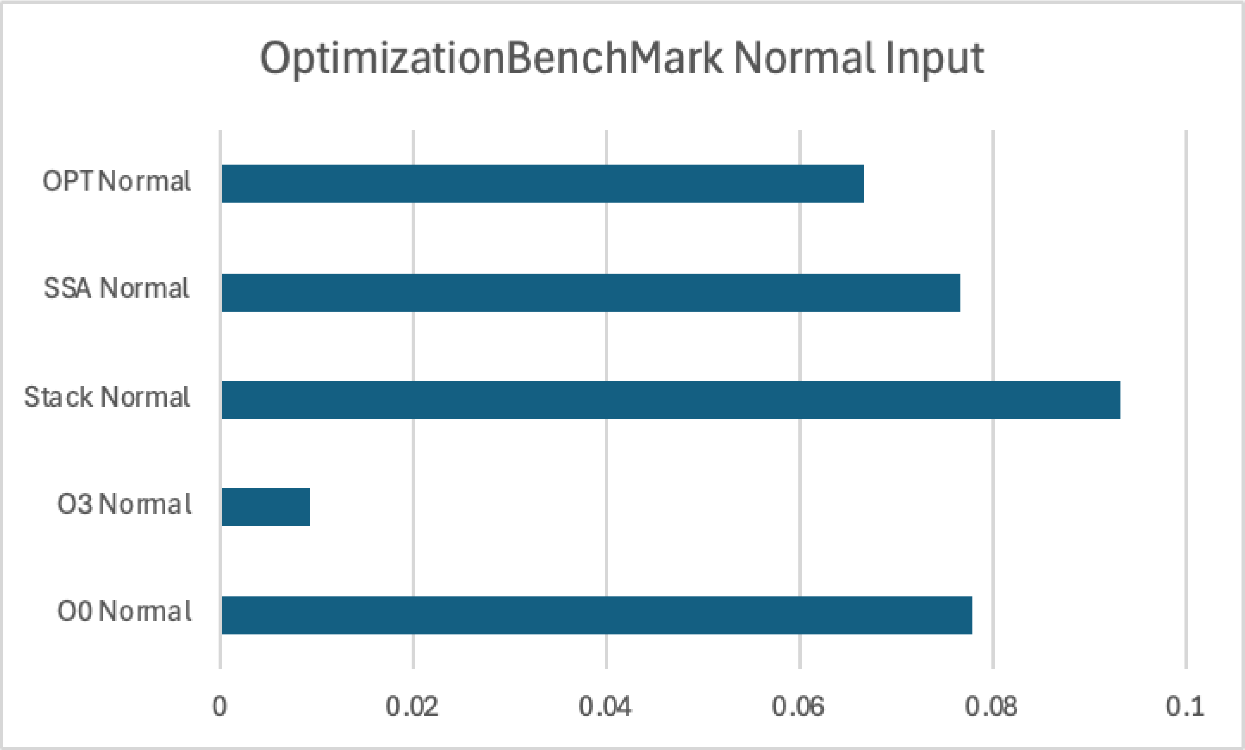

mile1 Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
5
10
15
20
25

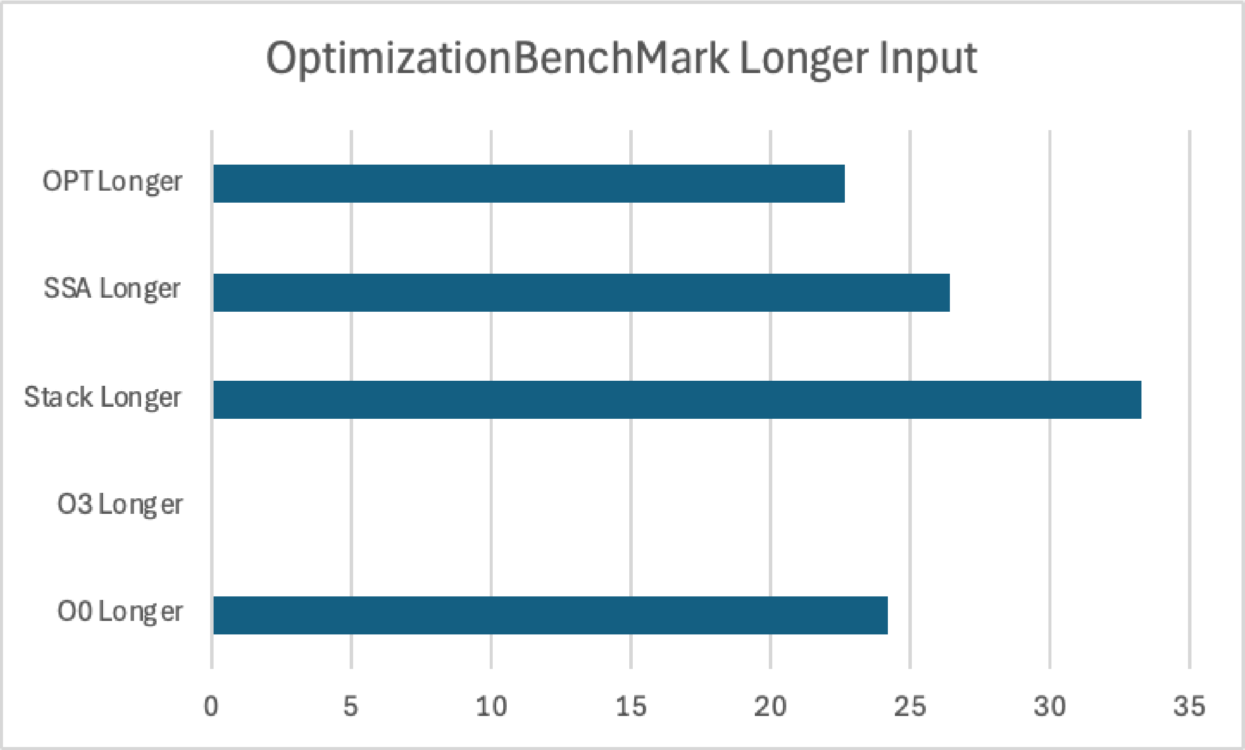
2.17 mixed


mixed Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
1
2
3
4
5
6
7
8

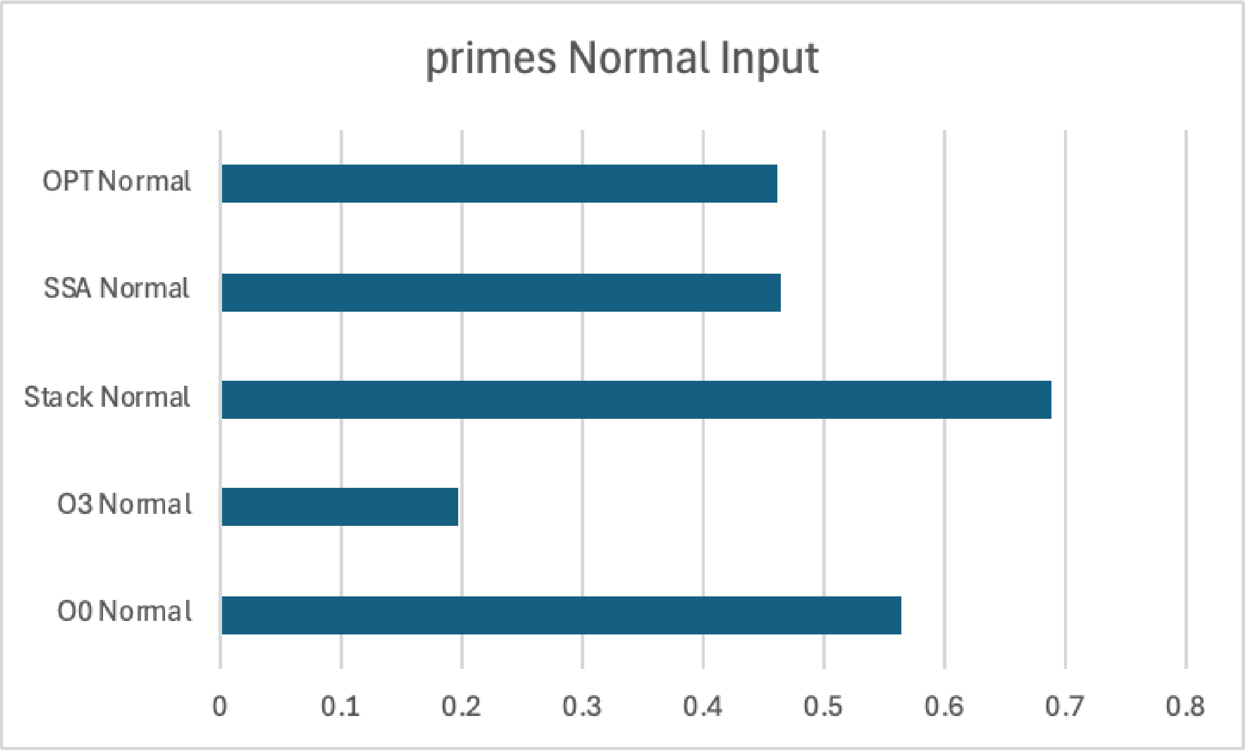

mixed Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
10
20
30
40
50

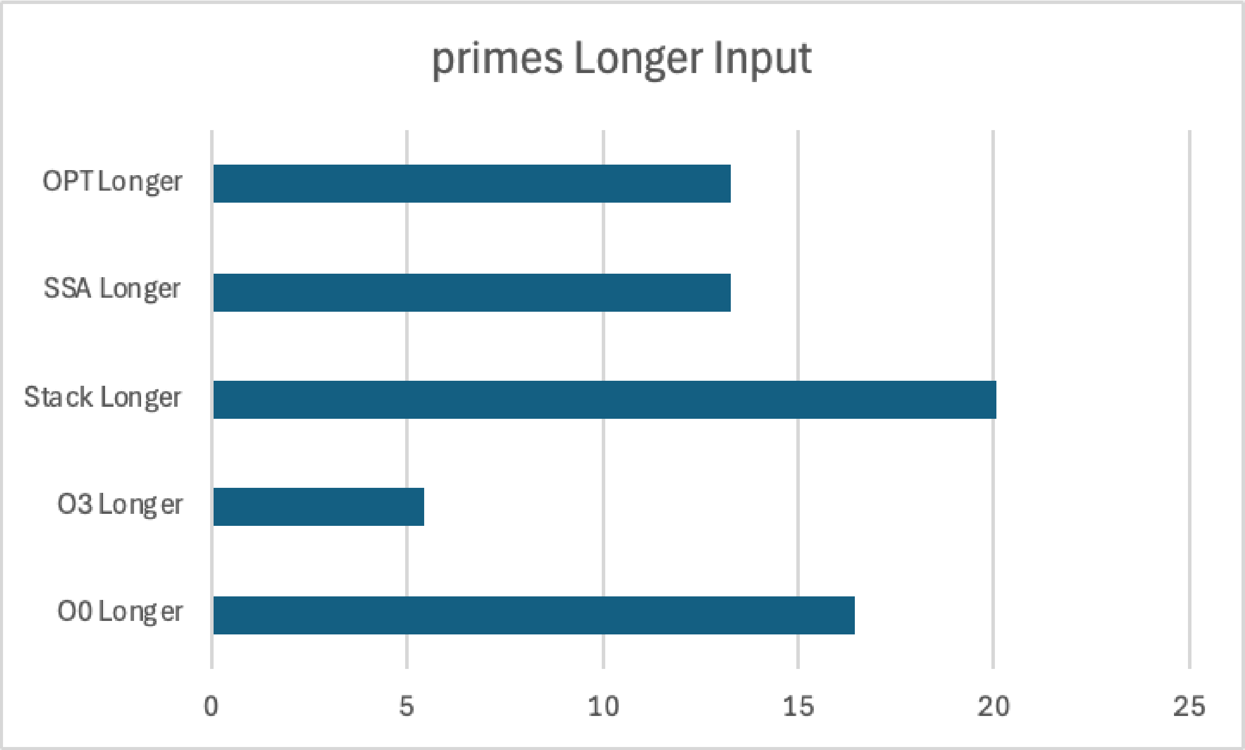
2.18 OptimizationBenchmark



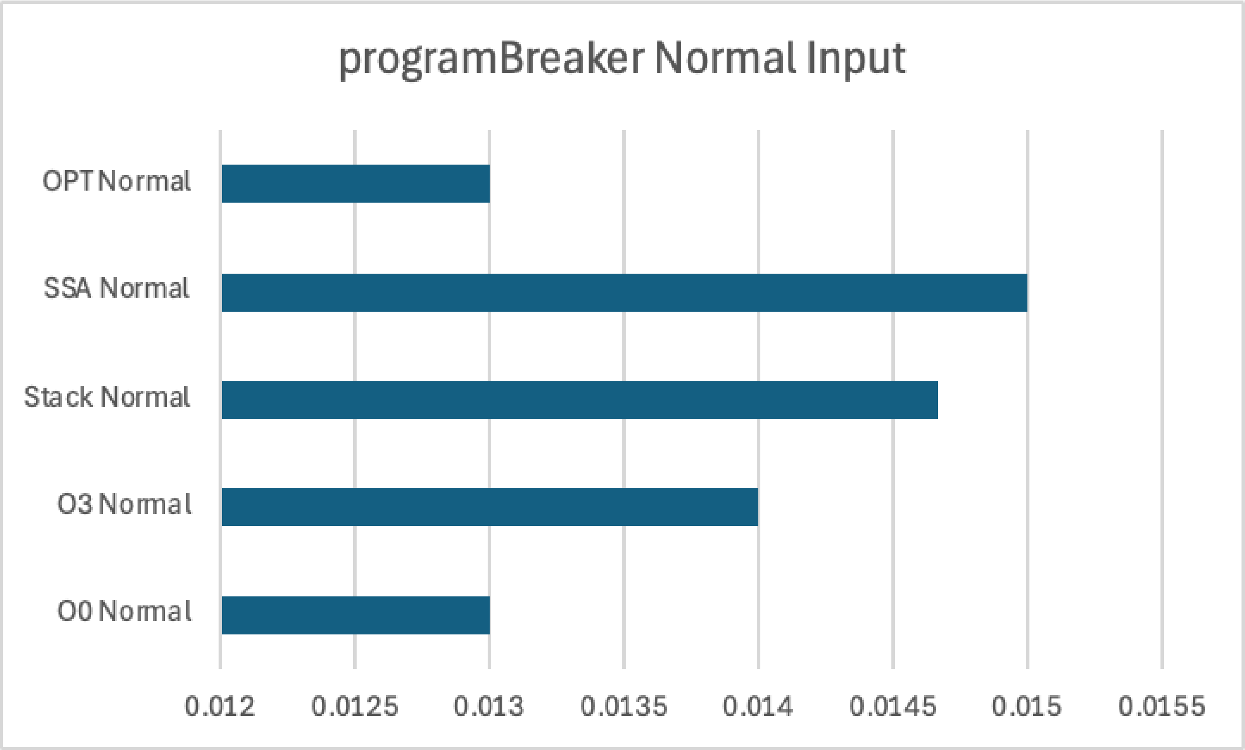


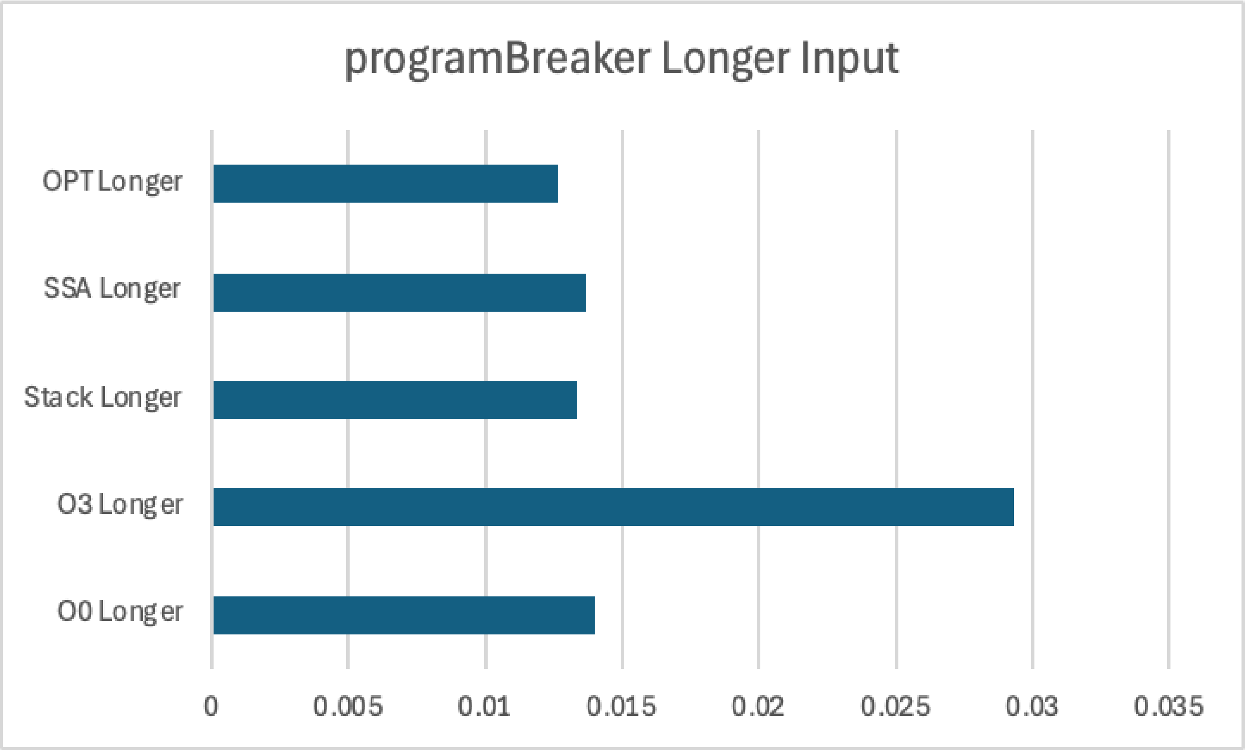
2.19 primes


primes Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8

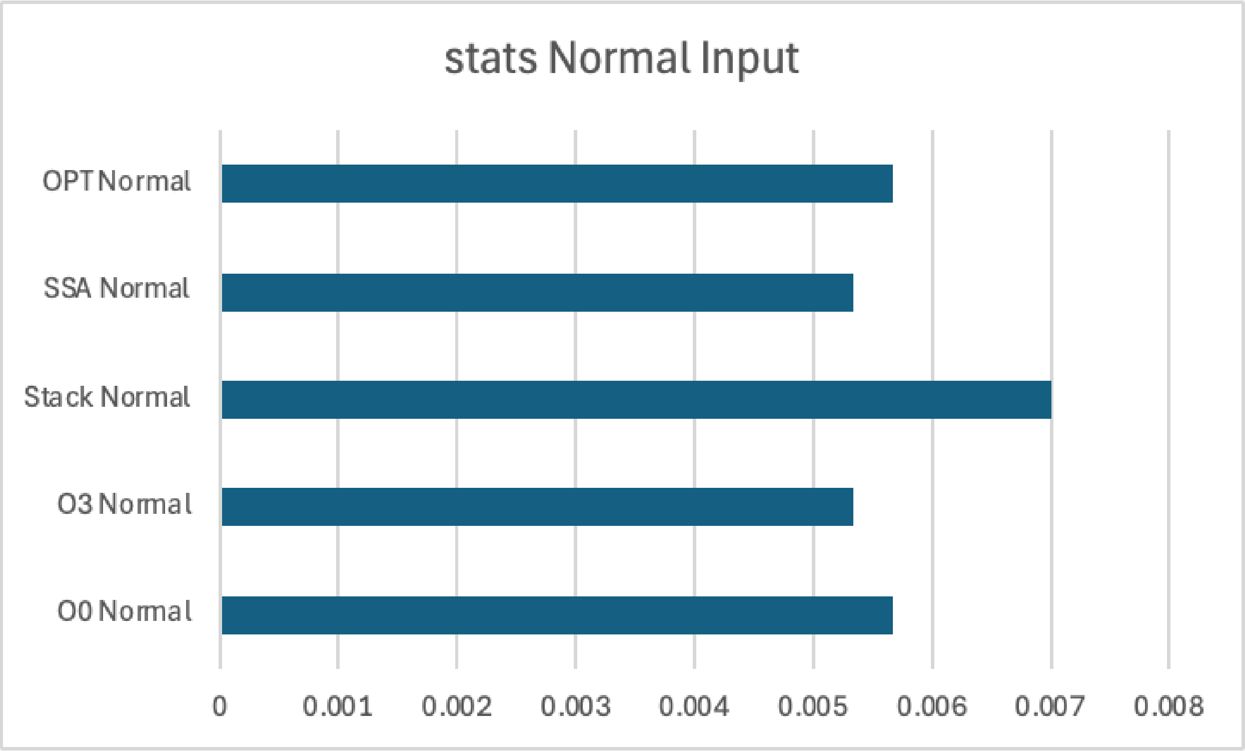

primes Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
5
10
15
20
25

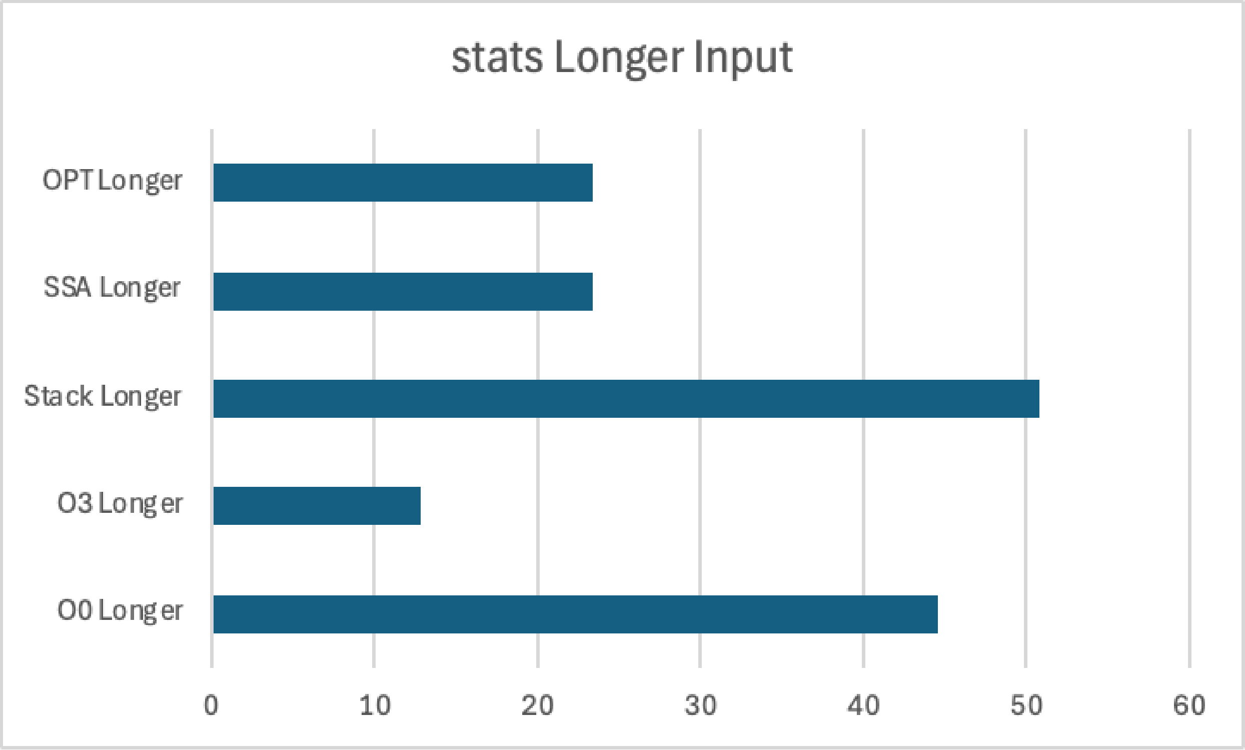
2.20 programBreaker



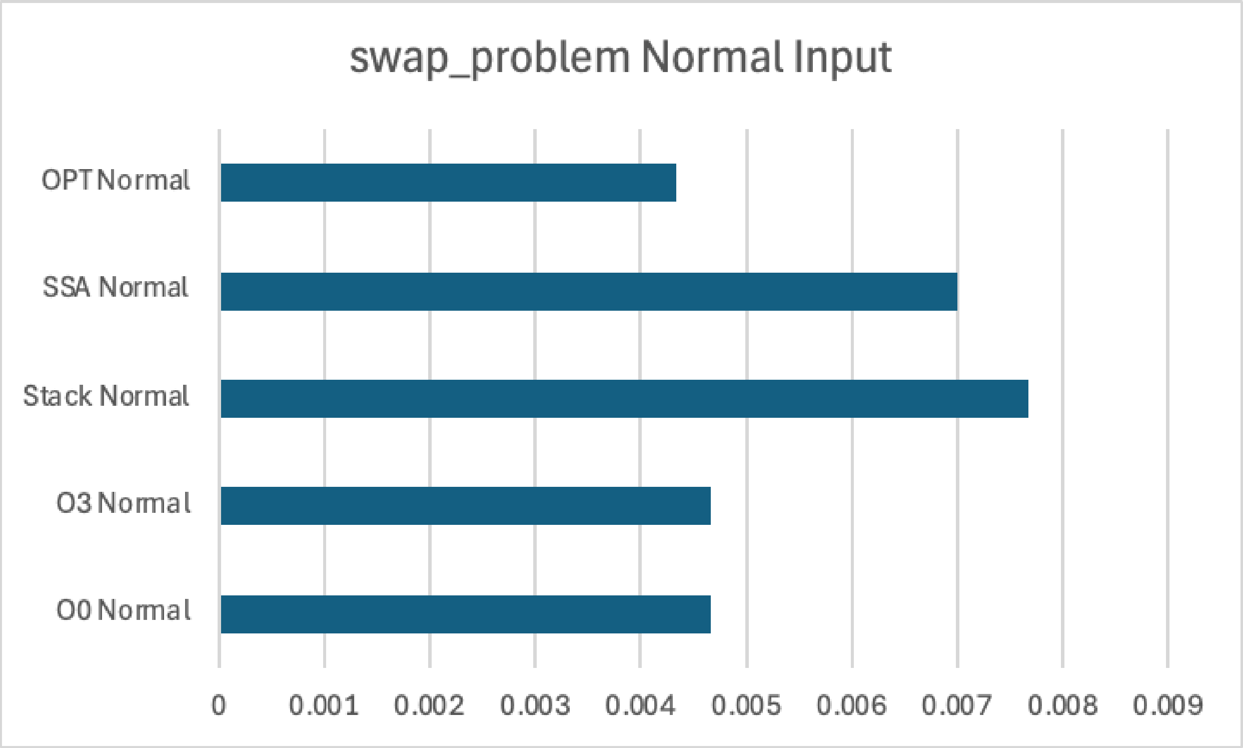


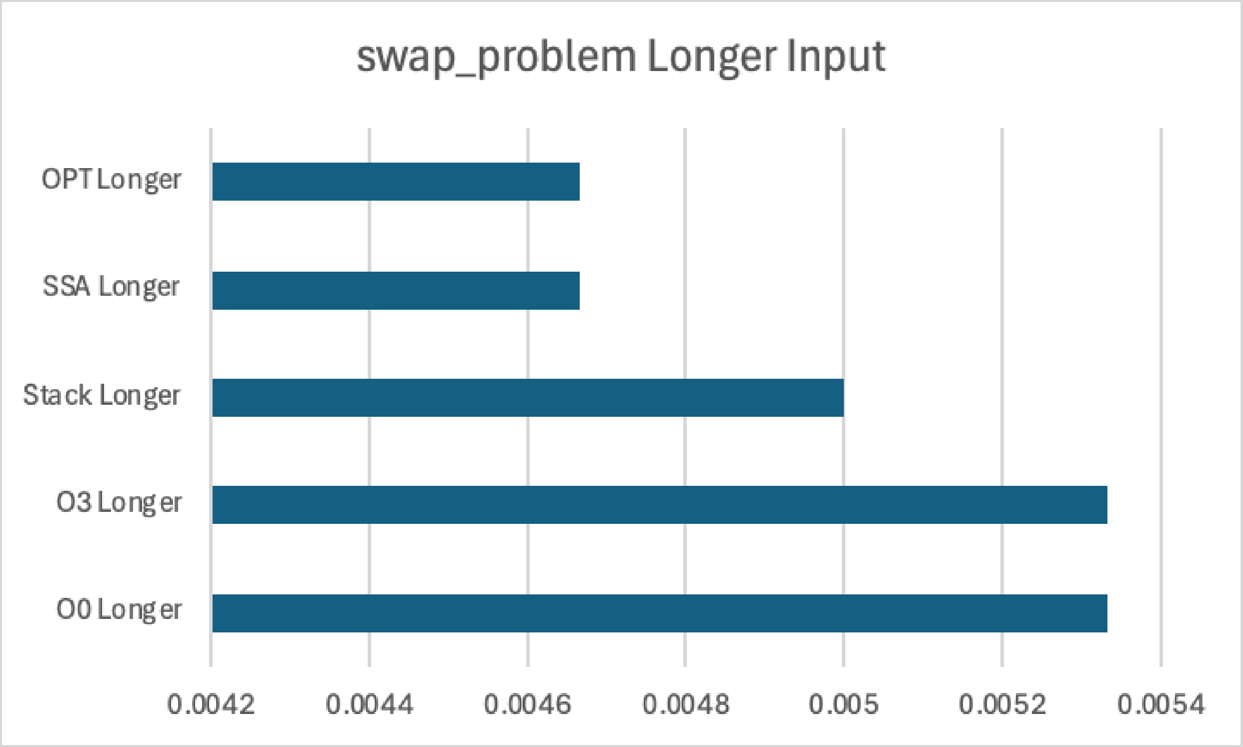
2.21 stats


stats Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008

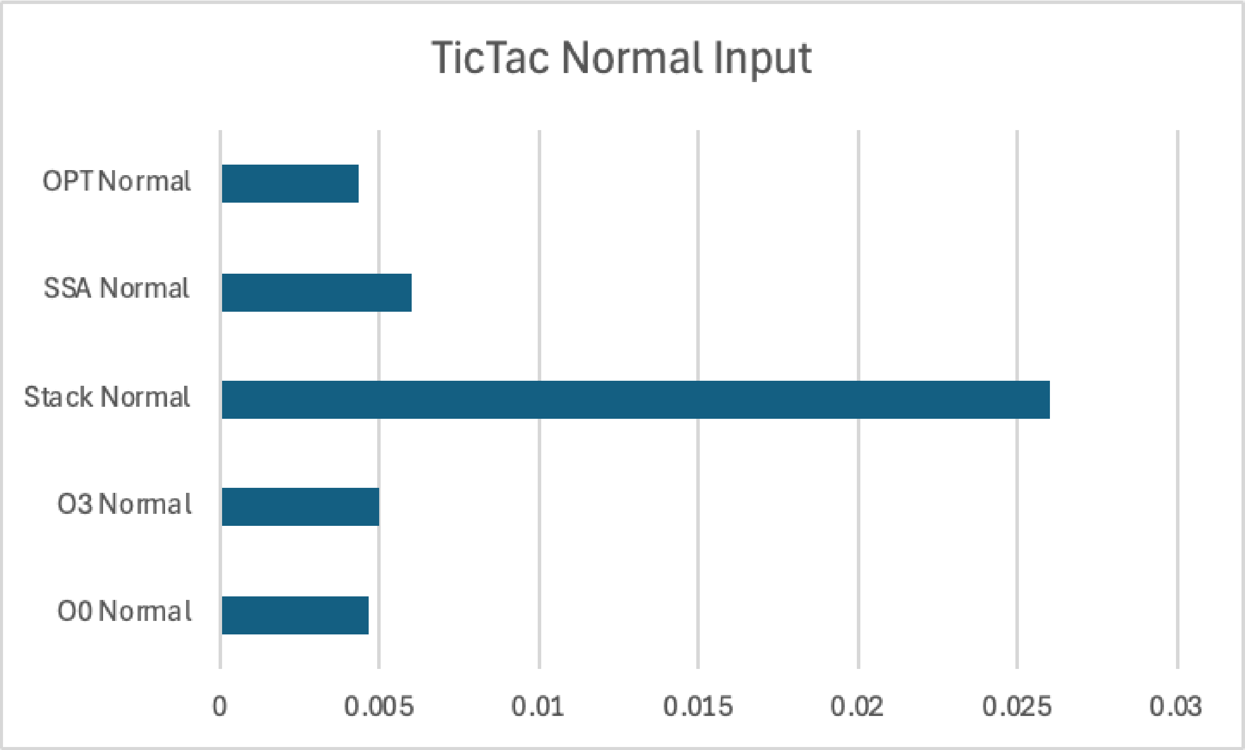

stats Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0
10
20
30
40
50
60

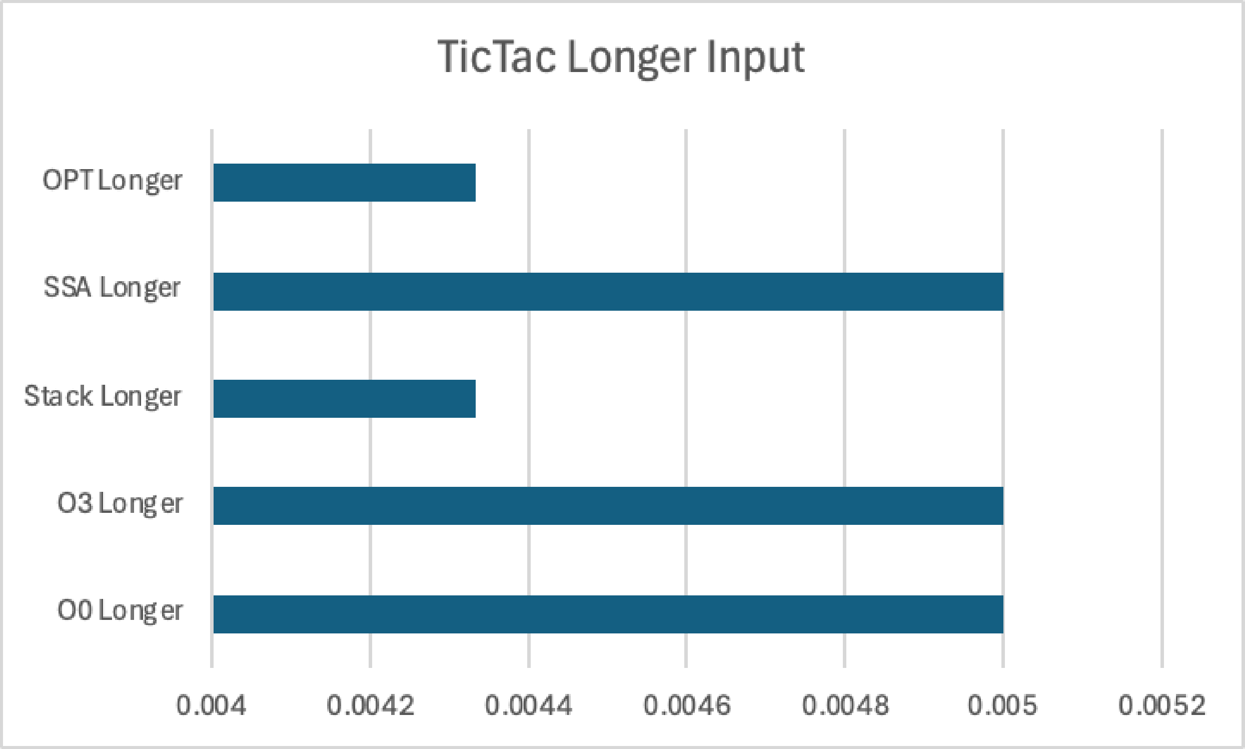
2.22 swap_problem



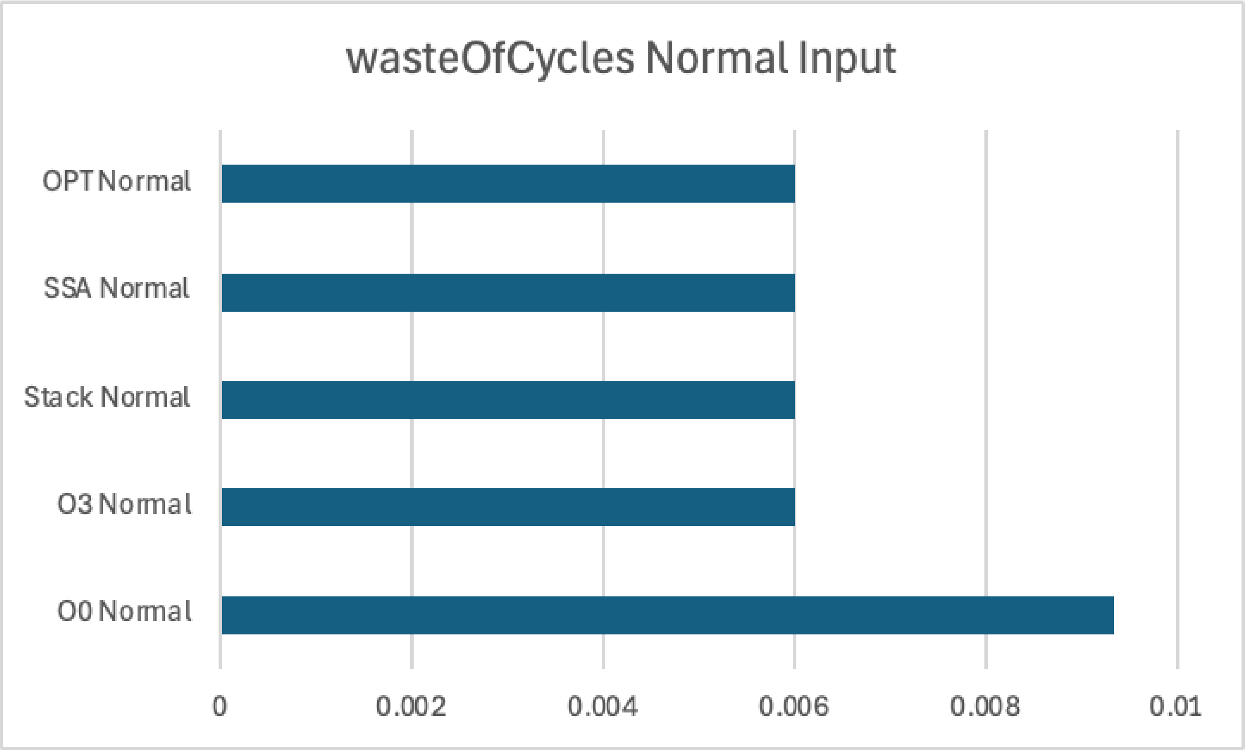


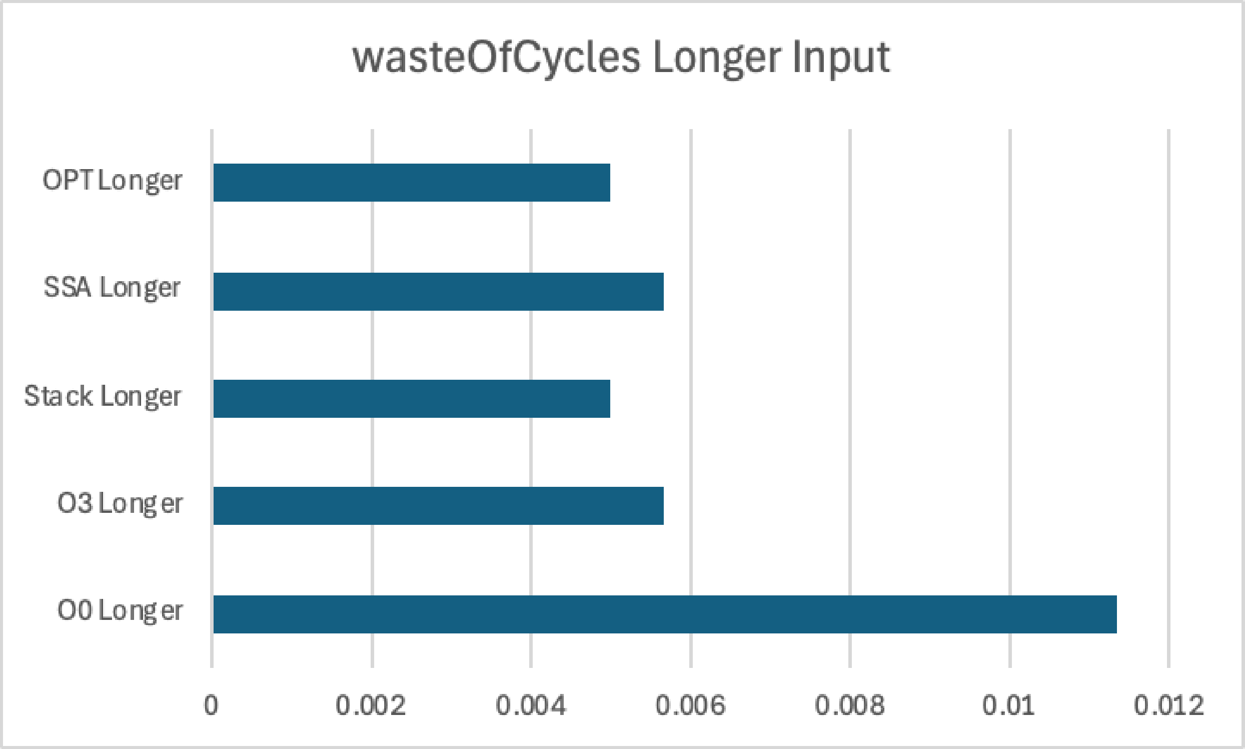
2.23 TicTac


TicTac Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal
0
0.005
0.01
0.015
0.02
0.025
0.03


TicTac Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer
0.004
0.0042
0.0044
0.0046
0.0048
0.005
0.0052

2.24 wasteOfCycles


wasteOfCycles Normal Input
OPT Normal
SSA Normal
Stack Normal
O3 Normal
O0 Normal


wasteOfCycles Longer Input
OPT Longer
SSA Longer
Stack Longer
O3 Longer
O0 Longer