Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Campo Mourão
Bacharelado em Ciência da Computação

Disciplina:	BCC35A - Linguagens de Programação		
Aluno:	Breno Farias da Silva	**R.A.:**	2300516
Email:	brenofarias@alunos.utfpr.edu.br		

Comparação entre compiladores/interpretadores Python & C/C++.

Para facilitar a execução dos códigos, criei um makefile para a execução dos códigos. Tal makefile estará incluso no zip enviado no moodle.

Execução dos Algoritmos:

- **Binary Trees**:

- **C++ com o compilador do GCC:** A execução do Binary Trees em C++ com o GCC foi rápida, executando em, aproximadamente, 2,43 segundos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$ make C
g++ binary_trees.cpp -o cbinary_trees
time ./cbinary_trees 16
stretch tree of depth 17        check: 262143
65536     trees of depth 4      check: 2031616
16384     trees of depth 6      check: 2080768
4096      trees of depth 8      check: 2093056
1024      trees of depth 10     check: 2096128
256       trees of depth 12     check: 2096896
64        trees of depth 14     check: 2097088
16        trees of depth 16     check: 2097136
long lived tree of depth 16     check: 131071
2.431
2.43user 0.00system 0:02.43elapsed 100%CPU (0avgtext+0avgdata 12192maxresident)k
0inputs+0outputs (0major+2194minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$
```

- **C++ com o compilador do CLang++:** A execução do Binary Trees em C++ com o CLang foi ainda mais rápida, executando em, aproximadamente, 1,90 segundos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$ make Clang
clang++ binary_trees.cpp -o clangbinary_trees
time ./clangbinary_trees 16
stretch tree of depth 17        check: 262143
65536     trees of depth 4      check: 2031616
16384     trees of depth 6      check: 2080768
4096      trees of depth 8      check: 2093056
1024      trees of depth 10     check: 2096128
256       trees of depth 12     check: 2096896
64        trees of depth 14     check: 2097088
16        trees of depth 16     check: 2097136
long lived tree of depth 16     check: 131071
1.898
1.90user 0.00system 0:01.90elapsed 100%CPU (0avgtext+0avgdata 12084maxresident)k
0inputs+0outputs (0major+2192minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$
```

- **Python Puro (CPython):** A execução do Binary Trees em Python puro foi relativamente rápida, executando em, aproximadamente, 16,84 segundos**.**

```
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$ make Python
time python3 binary_trees.py 16
stretch tree of depth 17         check: 262143
65536     trees of depth 4       check: 2031616
16384     trees of depth 6       check: 2080768
4096      trees of depth 8       check: 2093056
1024      trees of depth 10      check: 2096128
256       trees of depth 12      check: 2096896
64        trees of depth 14      check: 2097088
16        trees of depth 16      check: 2097136
long lived tree of depth 16      check: 131071
16.69user 0.15system 0:16.84elapsed 99%CPU (0avgtext+0avgdata 49616maxresident)k
0inputs+0outputs (0major+134697minor)pagefaults 0swaps
```

- **Python com o compilador Pypy:** A execução do Binary Trees em Python com o compilador do Pypy foi muito mais rápida, executando em, aproximadamente, 0,90 segundos**,** o que é aproximadamente 18 vezes mais rápido do que o Python puro.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$ make Pypy-BinaryTrees
time pypy3 binary_trees.py 16
stretch tree of depth 17         check: 262143
65536     trees of depth 4       check: 2031616
16384     trees of depth 6       check: 2080768
4096      trees of depth 8       check: 2093056
1024      trees of depth 10      check: 2096128
256       trees of depth 12      check: 2096896
64        trees of depth 14      check: 2097088
16        trees of depth 16      check: 2097136
long lived tree of depth 16      check: 131071
0.87user 0.03system 0:00.90elapsed 100%CPU (0avgtext+0avgdata 92916maxresident)k
0inputs+0outputs (0major+19097minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$
```

- **Python com o compilador Codon:** A execução do Binary Trees em Python com o compilador do Codon foi muito mais rápida, mesmo se comparada ao Pypy, executando em, aproximadamente, 0,6 segundos, sendo 30% mais rápido do que o Python puro.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/Binary Trees$ make Codon
codon run -release binary_trees.codon 16
stretch tree of depth 17         check: 262143
65536     trees of depth 4       check: 2031616
16384     trees of depth 6       check: 2080768
4096      trees of depth 8       check: 2093056
1024      trees of depth 10      check: 2096128
256       trees of depth 12      check: 2096896
64        trees of depth 14      check: 2097088
16        trees of depth 16      check: 2097136
long lived tree of depth 16      check: 131071
0.599715
```

- **N-Body:**

- **C com o compilador GCC:** A execução do N-Body em C foi rápida, executando em, aproximadamente, 2,18 segundos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$ make C
gcc nbody.c -o cnbody -lm
time ./cnbody 10000000
2.18user 0.00system 0:02.18elapsed 100%CPU (0avgtext+0avgdata 1608maxresident)k
0inputs+0outputs (0major+74minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$
```

- **C com o compilador Clang:** A execução do N-Body em C foi ligeiramente mais rápida do que com o GCC, executando em, aproximadamente, 2,09 segundos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$ make Clang
clang nbody.c -o clangnbody -lm
time ./clangnbody 10000000
2.09user 0.00system 0:02.09elapsed 99%CPU (0avgtext+0avgdata 1644maxresident)k
0inputs+0outputs (0major+75minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$
```

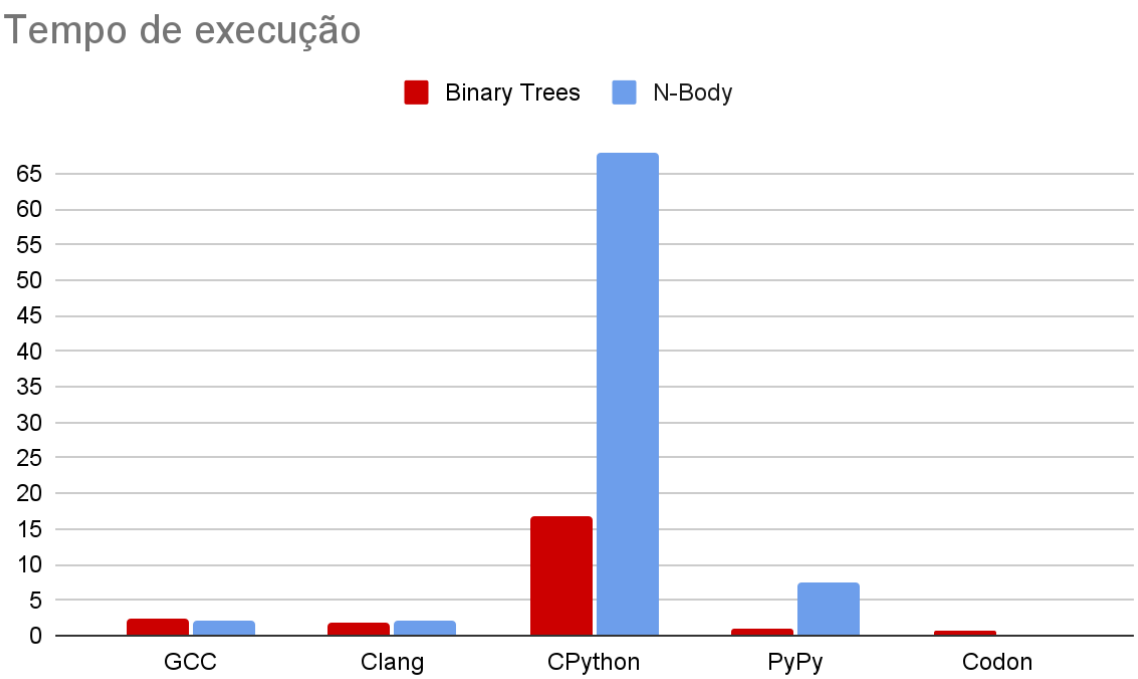
- **Python Puro (CPython):** A execução do N-Body em Python puro demorou, aproximadamente, 1 minuto, 8 segundos e 15 milésimos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$ make Python
time python3 nbody.py 10000000
68.14user 0.00system 1:08.15elapsed 99%CPU (0avgtext+0avgdata 8916maxresident)k
0inputs+0outputs (0major+895minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$
```

- **Python com o compilador Pypy:** A execução do N-Body em Python com o compilador do Pypy foi quase 10 vezes mais rápida, com um tempo de execução de, aproximadamente, 7,43 segundos.

```
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$ make Pypy
time pypy3 nbody.py 10000000
7.37user 0.05system 0:07.43elapsed 99%CPU (0avgtext+0avgdata 64884maxresident)k
0inputs+0outputs (0major+6865minor)pagefaults 0swaps
brenofarias@DESKTOP-VRM37Q8:~/LP/N-Body$
```

1 - Realizar um gráfico de comparação (tempo de execução), utilizando uma entrada resultando em um tempo suficiente para ver a diferença entre as execuções dos diversos compiladores. Por exemplo, nbody utilizando uma entrada de 10.000.000 e binary_trees uma entrada de 16:

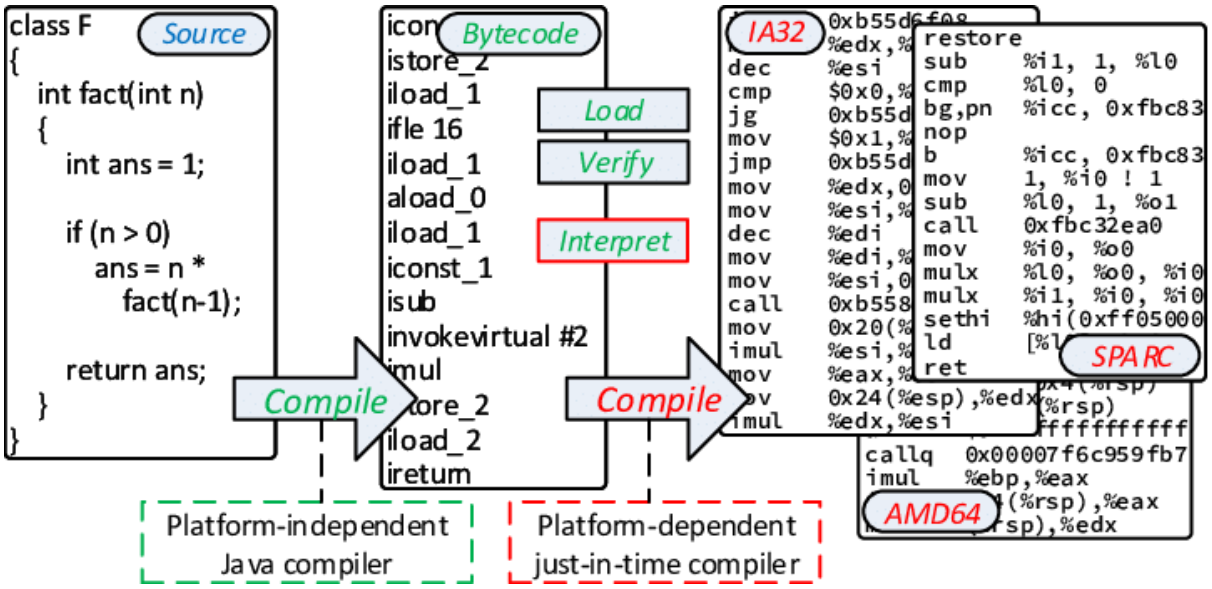


2 - Descrever motivos que faz com que a execução de um compilador seja maior que outro. Por exemplo, presença de JIT?, compila direto para LP de máquina?, possui alto grau de otimizações? etc. Note que a página principal de cada compilador (pelo menos os modernos), discutem as suas principais vantagens no modelo de execução. Compare o conteúdo visto em aula e presente nas páginas oficiais para se embasar nas hipóteses.

- **Python Puro (CPython) vs Pypy vs Codon:** Vimos que tanto no *Binary Tree* quanto no *N-Body*, houveram melhorias (acima de 10 vezes mais rápido) no tempo de execução dos códigos compilados pelo *Pypy* e pelo *Codon* com relação ao Python puro, mas qual seria a razão disso? Bem, tanto o *Pypy* quanto o Codon tem uma implementação do compilador JIT (*Just-in-Time*), que nada mais é do que um software que combina bem o processo de compilação com o de interpretação. Isso se deve ao fato de que o JIT, durante o processo de conversão do código para bytecode, que é um código que será interpretado por uma VM (*Virtual Machine*, como a JVM do Java ou a LLVM do Clang, por exemplo). Todavía, é relevante lembrar que bytecode é diferente de código de máquina. Código de máquina é o código composto de 0s e 1s. O Bytecode digamos que está em um nível

intermediário entre o código fonte e o código de máquina, no qual ele não é executável e que será interpretado pela VM e, só depois, compilado novamente de modo a virar código de máquina. No fim deste tópico há um diagrama (Figura 1) tirado do ResearchGate que demonstrará melhor isso.
O Pypy em si usa a JIT, como já foi mencionado e o motivo dele ser tão rápido é porque o JIT dentro do PyPy analisa o tipo de informação e objetos que são criados no programa e depois usa essa informação para otimizar as coisas. Por exemplo, se tem apenas um tipo diferente de objeto no código, o PyPy vai gerar um código de máquina para acelerar o tempo de execução desse objeto. O Codon faz o mesmo e vai além, pois ele usa a LLVM e usa o *ahead-of-time (AOT)* no lugar do *Just-in-time*, que a diferença entre eles se dá ao fato que de o JIT compila o software em *runtime* e o AOT compila o software em *build time*. Além disso, ele implementa funcionalidades funções *inline* (ou funções anônimas), que são melhores, pois não aumenta a pilha de execução, além de paralelismo e multithreading, mas tudo isso sem gerar overhead, segundo a própria descrição do *Codon* no Github.

Figura 1 - Bytecode e Código Fonte



Fonte: Research Gate - Java source code is first compiled to bytecode, and subsequently interpreted or executed as native code. Heavy optimizations are reserved for the JIT-compilation phase.

- **GCC vs CLang:** A execução usando o compilador do Clang mostrou-se ligeiramente melhor pois ele usa a LLVM, a qual implementa o *Just-in-time* (JIT), o qual já foi explicado anteriormente. Mas por que que a diferença é pequena entre os tempos

de execução usando o GCC e o CLang? Bom, isso se deve ao fato de que as otimizações feitas no próprio GCC já são muito eficientes, além de que, dependendo do contexto, a vantagem sobre um ou outro varia, pois o contexto importa. Mas falando melhor da LLVM, ela pode otimizar as linguagens de programação e os link durante a compilação, tempo de execução (*runtime*).Outros fatores relevantes se dão também pelo fato do Clang ter uma infraestrutura modular, que permite rápida análise e manipulação do código, um diagnóstico mais detalhado, que permite identificar e resolver os erros do código de forma mais eficiente, além de algoritmos nos geral mais eficientes de otimização do que o GCC padrão, apesar de no GCC podermos especificar o nível de otimização que queremos no nosso código, o qual vai ser 0 até 4.