Elivelton Fernando de Oliveira
Matheus Sapia Guerra
Rafael Menezes Barboza
Vitor Yudi Shinohara

Avaliação empírica de algoritmos para solucionar o problema de Subvetor Máximo

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Análise de Algoritmos do Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Universidade Tecnológica Federal do Paraná – UTFPR
Departamento Acadêmico de Computação – DACOM
Bacharelado em Ciência da Computação – BCC

Campo Mourão
Dezembro / 2017

Resumo

O objetivo deste relatório técnico é realizar a análise empírica do algoritmo que soluciona o problema do subvetor máximo, para isto foi definido préviamente quatro linguagens de programação diferentes, sendo elas duas compiladas e duas interpreteadas. O experimento consiste em implementar 4 técnicas diferentes do algoritmo para cada uma das linguagens e executa-los com uma base de entrada padrão. Sendo assim é possível comparar o tempo gasto por cada técnica de implemtação e cada linguagem. Foi abordado análises matemáticas para provar a complexidade dos algoritmos, facilitando assim a visualização do comportamento dos algoritmos implementados.

Palavras-chave: análise de agoritmos. análise empírica. sub vetor máximo.

Sumário

1 Introdução

Na área da computação, o problema do subvetor máximo consiste em encontrar um subvetor de dimensão [1...k] onde $1 \leq k \leq n$, tal que a soma de todos os elementos do subvetor é também a maior soma do vetor [1...n], podendo conter números positivos e negativos.

Dentre diversas aplicações. 2 principais áreas da ciência da computação faz o uso destes algoritmos, tal qual visão computacional, utilizando o subvetor máximo para detectar áreas mais claras em uma imagem com auxílio de seu *bitmap* (WEDDELL et al., 2013), além do seu uso em mineração de dados (TAKAOKA, 2002).

Existe diversas maneiras de se solucionar o problema, nas seções futuras será abordado a análise empírica de 4 diferentes algoritmos propostos em 4 linguagens de programação diferentes para realizar o teste de performance e suas respectivas análises matemáticas.

2 Objetivos

O objetivo principal do relatório técnico é avaliar empiricamente e matematicamente o desempenho dos algoritmos implementados em 4 diferentes linguagens de programa para solucionar o problema do subarray máximo, os quais serão detalhados nas seções futuras com mais detalhes.

O relatório terá foco na análise de desempenho do algoritmo, sua complexidade e recorrência caso exista. Não será abordado a implementação com detalhes.

3 Fundamentação

3.1 Subvetor Máximo

O problema do subvetor máximo é solucionado através de um algoritmo o qual percorra o vetor e encontre a soma máxima de elementos presentes no mesmo (BORRADIALLE, 2017).

Dado um pequeno vetor de números inteiros, compute:

$$\max \sum_{k=i}^{j} a[k]$$

Por exemplo: $MaxSubArray([31, -41, \mathbf{59}, \mathbf{26}, \mathbf{\text{-}53}, \mathbf{58}, \mathbf{97}, -93, -23, 84]) = 197$

3.2 Algoritmo *Enumeration*

O algoritmo Enumeration quando comparado aos outros que serão abordados nas próximas subseções tem o pior desempenho. Dado um vetor de tamanho $[1...n]$, o mesmo é submetido a 3 laços de repetição, onde o primeiro realiza a execução n vezes, o segundo executa $n - i$ vezes, e finalmente o terceiro laço se repete $n - i - j$ vezes. O Algoritmo *Enumeration* é conhecido também por *Brute Force*.

Algorithm 1 Pseudo-código algoritmo Enumeration

```
1: ENTRADA: * Array
2: SAIDA: * SomaMaximaDoArray
3: max ← −∞
4: n ← length(a)
5: for i = 0 to n do
6:     sum ← 0
7:     for j = i to n do
8:         for k = i to k = j do
9:             sum ← a[n]
10:            if sum > max then
11:                max ← sum
12:            end if
13:        end for
14:    end for
15: end for
16: return max
```

Considerando a estruturação do algoritmo, podemos realizar o cálculo através da análise de trechos iterativos com o fim de obter a complexidade do mesmo, podendo ser calculada por:

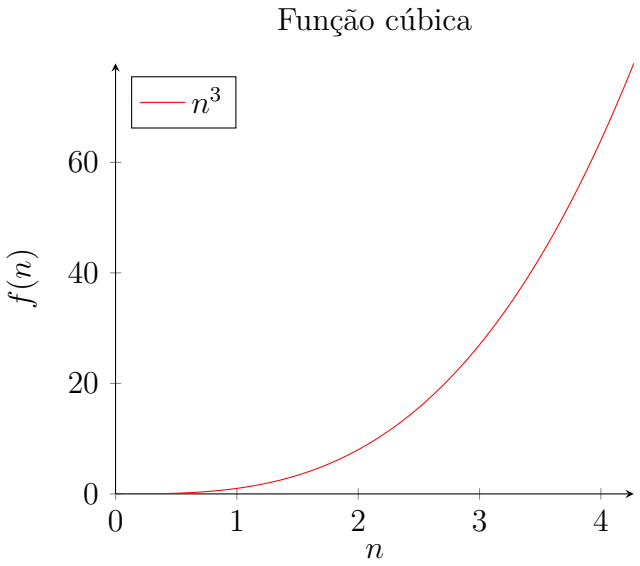
$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=j}^{j} n$$

De forma resumida, temos que:

$$T(n) = \Theta(n^3)$$

Podemos observar no algoritmo 1 a quantidade de laços aninhados resultando em um desempenho ruim.

O gráfico a seguir mostra o comportamento de uma função $f(x) = n^3$ o qual demonstra o comportamento do algoritmo, onde o eixo x representa o número de entradas e o eixo y ilustra o tempo.



É notável observar que o crescimento exponencial dificulta ou torna impossível a entrada de médias e grandes quantidades de dados, pois o tempo necessário para processamento aumenta de maneira muito rápida.

3.3 Algoritmo *Better Enumeration*

O algoritmo *Better Enumeration*, a partir de uma entrada $[1...n]$ submete a mesma para dois laços iterativos aninhados, onde o primeiro executa n vezes no total, e o segundo realiza $n - 1$ execuções como mostra o algoritmo 2.

Algorithm 2 Pseudo-código algoritmo BetterEnumeration

```
1: ENTRADA: * Array
2: SAIDA: * SomaMaximaDoArray
3: max ← −∞
4: n ← length(a)
5: for i = 0 to n do
6:     sum ← 0
7:     for j = i to n do
8:         sum ← a[n]
9:         if sum > max then
10:            max ← sum
11:        end if
12:    end for
13: end for
14: return max
```

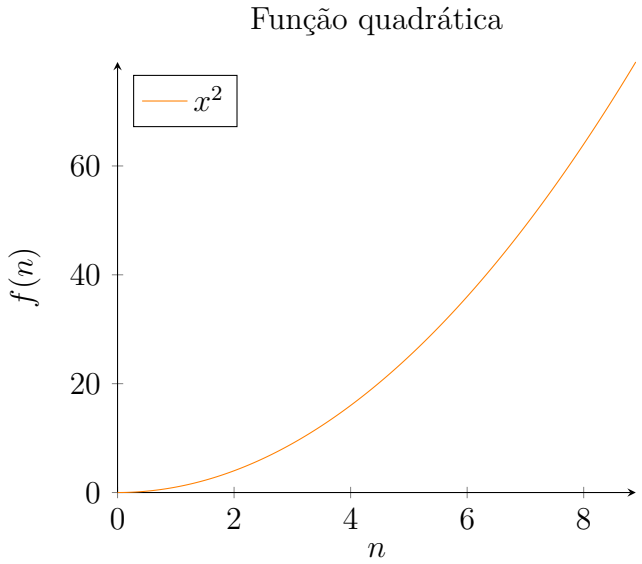
Sua complexidade é dada pela seguinte fórmula matemática:

$$T(n) = \sum_{i=1}^{n} \sum_{j=i}^{n} n$$

Resolvendo os somatórios a cima, temos que:

$$T(n) = \Theta(n^2)$$

Logo, o comportamento do algoritmo *BetterEnumeration* se assemelha com uma função quadrática, onde seu desempenho deixa a desejar para grandes quantidades de dados assim como o *Enumeration*, porém de forma mais eficiente.



Uma função quadrática significa que será processado todos os pares de valores possíveis de uma entrada, sendo prático apenas quando aplicado à pequenas quantidades de dados.

3.4 Algoritmo *Divide and Conquer*

O algoritmo *Divide and Conquer* é uma solução mais rápida que os algoritmos apresentados anteriormente, este adota da estratégia de divisão e conquista para resolver o problema. A idéia deste algoritmo é dividir o tamanho n por 2 até que se obtenha o caso base ($n = 1$), assim resolver e combinar as soluções para obter a resposta do problema.

Algorithm 3 Pseudo-código algoritmo Divide and Conquer

```
1: ENTRADA: * Array, , lo, hi
2: SAIDA: * SomaMaximaDoArray
3: if lo == hi then
4:    return  a[hi]
5: end if
6: midpoint ← (lo + hi)/2
7: leftMax ← divideAndConquer(a, lo, midpoint)
8: rightMax ← divideAndConquer(a, midpoint + 1, hi)
9: leftBothMax ← −∞
10: for i = midpoint to i ≥ 0 do
11:    leftBothSum ← leftBothSUm + a[i]
12:    if leftBothSum > leftBothMax then
13:       leftBothMax = leftBothSum
14:    end if
15: end for
16: rightBothMax = −∞
17: for i = midpoint to i ≥ 0 do
18:    RightBothSum ← RightBothSum + a[i]
19:    if rightBothSum > rightBothMax then
20:       rightBothMax = rightBothSum
21:    end if
22: end for
23: bothMax ← leftBothMax + rightBothMax
24: return  max(bothMax, leftMax, rightMax)
```

Apesar de ter um bom desempenho, a solução por divisão e coquista tem uma alta complexidade se tratando de implementação, pois geralmente não é trivial implementar algoritmos recursivos.

A complexidade do algoritmo *Divide and Conquer* pode ser dada pela seguinte recorrência:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Resolvendo a recorrência encontrada pelo teorma mestre, temos que:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$a = 2$$

$$b = 2$$

$$f(n) = n$$

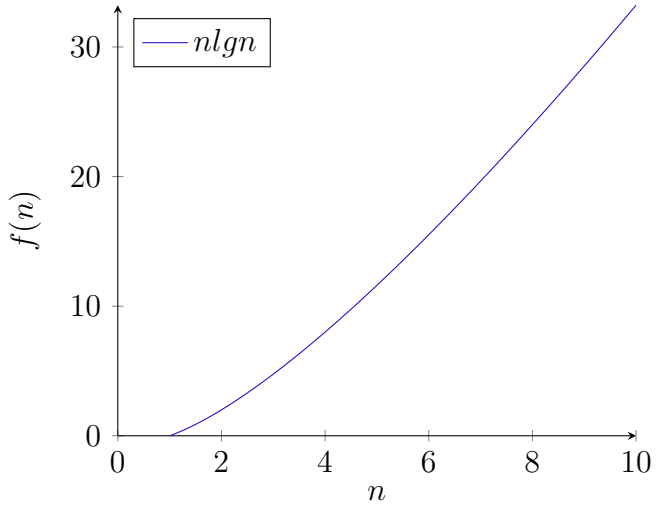
$$n^{log_2(2)} = n$$

Como n e $f(n)$ são polinomialmente iguais, caimos no caso II do teorema mestre, sendo assim temos que a complexidade do algotritmo *Divide and Conquer* para o problema do sub vetor máximo é:

$$T(n) = \Theta(nlog(n))$$

Sendo assim podemos concluir que para valores de n grande, a solução por *Divide and Conquer* se destaca comparado aos algoritmos com complexidade polinomial, tornando possível computar grandes quantidades de valores dentro de um tempo relativamente curto. O gráfico a seguir mostra como se comporta uma função $f(x) = nlgn$.

Função logarítmica



3.5 Algoritmo *Dynamic*

O algoritmo *Dynamic* é a solução linear para o problema do sub vetor máximo. A idéia do algoritmo é percorrer o vetor do inicio até o final mantendo as maiores somas parciais, ou seja, quando chegar ao último elemento do vetor o algoritmo já terá a resposta para o problema. Esta é uma abordagem de programação dinâmica.

Algorithm 4 Pseudo-código algoritmo *Dynamic*

```
1: ENTRADA: * Array
2: SAIDA: * SomaMaximaDoArray
3: maxSoFar ← 0
4: maxEndingHere ← 0
5: for i = 0 to a.length -1 do
6:    maxEndingHere ← maxEndingHere + a[i]
7:    if maxEndingHere < 0 then
8:       maxEndingHere ← 0
9:    else if maxSoFar < maxEndingHere then
10:      maxSoFar ← maxEndingHere
11:   end if
12: end for
13: return maxSoFar
```

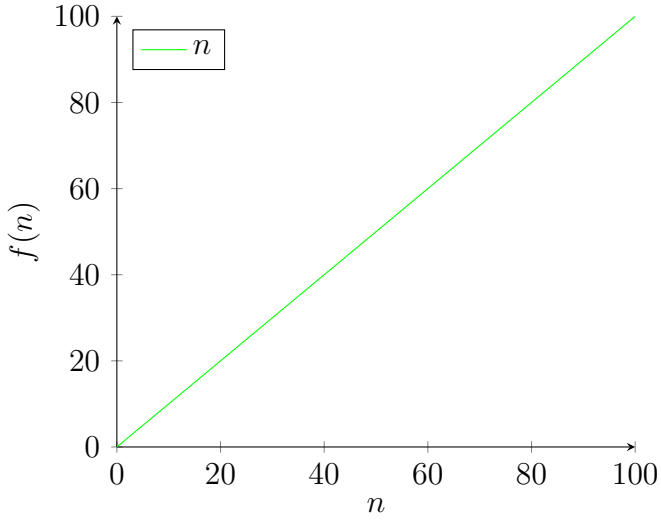
A complexidade do algoritmo apesentado é dado pela seguinte somatória:

$$T(n) = \sum_{i=0}^{n-1} c$$

Como as operações dentro do laço de repetição é constante, podemos dizer que o algoritmo tem a complexidade de:

$$T(n) = \Theta(n)$$

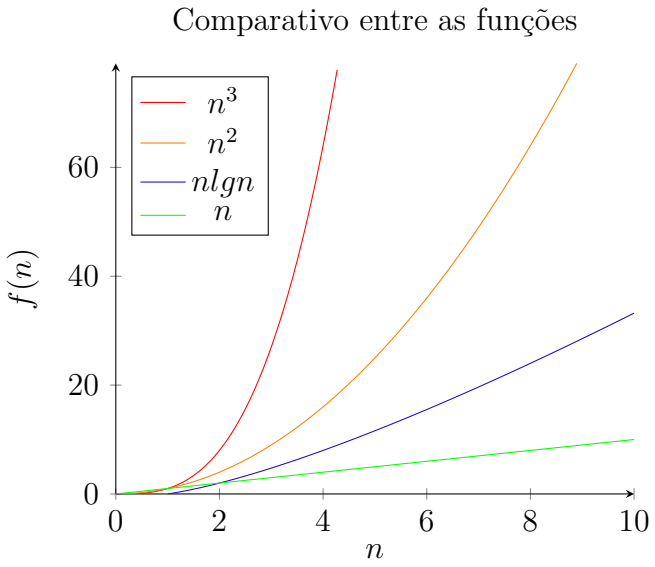
Função linear



A solução linear é a solução mais próxima do ideal que podemos chegar, pois em uma entrada grande o tempo de resposta será curto, viabilizando o uso deste algoritmo para resolver o problema do subvetor máximo para n grande.

3.6 Visão geral dos algoritmos

Com base na análise matemática feita nas subseções anteriores, podemos estipular dada uma quantidade de dados de entrada o tempo de execução através da complexidade do algoritmo.



Como podemos ver no gráfico anterior, o crescimento de uma função cúbica tal qual $f(n) = n^3$ é muito maior em relação a uma função quadrática $f(n) = n^2$, o que significa que dada uma entrada padrão para um algoritmo de complexidade cúbica, o mesmo levará muito mais tempo e realizará muito mais operações que um algoritmo quadrático.

4 Materiais

Para execução dos algoritmos foi utilizado um conjunto de dados padrão com vetores os quais continham elementos positivos e negativos de tamanhos $n = [2^2, 2^3, 2^4, ..., 2^{16}]$ utilizando o processador Intel Core i7-7500u *quad-core* 3.5 GHz com 8 GB de memória RAM com o sistema operacional Debian x64.

Foi utilizado os algoritmos *Enumeration*, *Better Enumeration*, *Division and Conquer* e finalmente o *Dynamic* em 4 linguagens de programação diferentes, sendo C, Java, Python e Ruby.

5 Procedimentos e Resultados

Para a execução dos algoritmos foi utilizado uma única máquina, a qual foi especificada na seção 4, onde apenas processos do sistema operacional estavam executando juntamente com os algoritmos.

Não foi implementado nenhum método de programação paralela, uso de *Threads* ou multi-processamento nos algoritmos, porém a linguagem Java realizou automaticamente a otimização, distribuindo o trabalho em 2 processadores.

Foi utilizado como métrica de tempo a unidade de milissegundos, o qual foi medido através de métodos nas linguagens, onde era retornado o tempo de relógio, sem levar em consideração o tempo de *CPU*. O tempo foi contabilizado no início da chamada de cada algoritmo ($T_{inicial}$) e no final (T_{final}), e finalmente foi calculado por:

$$T_{total} = T_{final} - T_{inicial}$$

É importante ressaltar que foi feito a medição de tempo de cada função separadamente e sem incluir *I/O*, para que não tenha erros de medição de tempo.

Para evitar interferências, os algoritmos foram executados 3 vezes com exceção da entrada de tamanho 16.384, visto que o tempo de execução seria impróprio para o teste, onde foi contabilizado 10 horas para o algoritmo *Enumeration* nas linguagens interpretadas . A partir disso, foi realizado o cálculo da média dos valores e exposto nas tabelas das subseções a seguir.

5.1 C

A implementação na linguagem de programação C consistiu em dividir a entrada em arquivos distintos, facilitando assim a manipulação ao escrever os algoritimos. Feito isso, é feito a leitura de cada arquivo armazenando os dados em forma de um vetor de inteiros e submetido aos algoritmos.

Após a execução do algoritmo, obtemos os resultados ilustrados na tabela 1.

Tabela 1: Tempo de execução dos algoritmos em C

Entrada / Tempo	**128**	**254**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Enumeration	1	6	49	382	3083	25111	207480	1571629
Better Enumeration	0	0	0	1	5	21	76	305
Divide and Conquer	0	0	0	0	0	0	0	1
Dynamic	0	0	0	0	0	0	0	0

5.2 Java

A implementação feita em Java consiste em, dado um vetor de inteiros, submeter aos algoritmos implementados, computar o tempo de execução dos mesmos e retornar o resultado para a função principal para verificar a consistência do resultado. Assim como em outros algoritmos, foi utilizado o tempo de relógio no código, e não o tempo de *CPU*.

A tabela a seguir mostra o desempenho do código para as entradas pré-definidas.

Tabela 2: Tempo de execução dos algoritmos em Java

Entrada / Tempo	**128**	**254**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Enumeration	11	23	18	133	1253	12210	96365	834138
Better Enumeration	1	2	6	22	5	11	48	160
Divide and Conquer	0	0	2	1	1	2	2	5
Dynamic	0	0	0	0	1	1	1	1

Uma peculiaridade da linguagem Java é que para quantidade de entradas maiores, as vezes o tempo de execução é menor.

Foi verificado a saída das funções e comparado com as outras linguagens, porém não foi notado nenhum resultado diferente.

5.3 Python

A implementação em Python consiste em, a partir de uma entrada, separar os dados e armazenar em um vetor com seu devido tamanho e a partir disso submeter para os algoritmos para computar o subvetor máximo e o tempo de processamento.

O resultado dos algoritmos foram armazenados em um arquivo, o qual continha a soma do subvetor máximo para verificar se os algoritmos estavam corretos e o tempo utilizado para executar tal método, ilustrado na tabela 3.

Tabela 3: Tempo de execução dos algoritmos em Python

Entrada / Tempo (ms)	**128**	**254**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Enumeration	21	115	884	7421	57575	509277	4389356	36046388
Better Enumeration	4	19	150	915	7811	62838	591922	4841982
Divide and Conquer	0	0	1	2	3	6	12	28
Dynamic	0	0	0	0	1	1	1	1

5.4 Ruby

Para a implementação do script em Ruby, o primeiro passo foi realizar a leitura de um arquivo contendo uma entrada padrão. Após realizar este passo, foi implementada quatro funções para as implementações do problema do subvetor máximo (*Enumeration, Better Enumeration, Divide and Conquer* e *Dynamic*.

Utilizando um temrinal e em um sistema Linux, pode-se executar um script Ruby da seguinte forma:

```
$ ruby myscript.rb
```

Após a execução do script, os tempos gastos para resolver os problemas foram armazenados em um arquivo. A Tabela 4 mostra o tempo gasto para cada algoritmo dado uma entrada.

Tabela 4: Tempo de execução dos algoritmos em Ruby

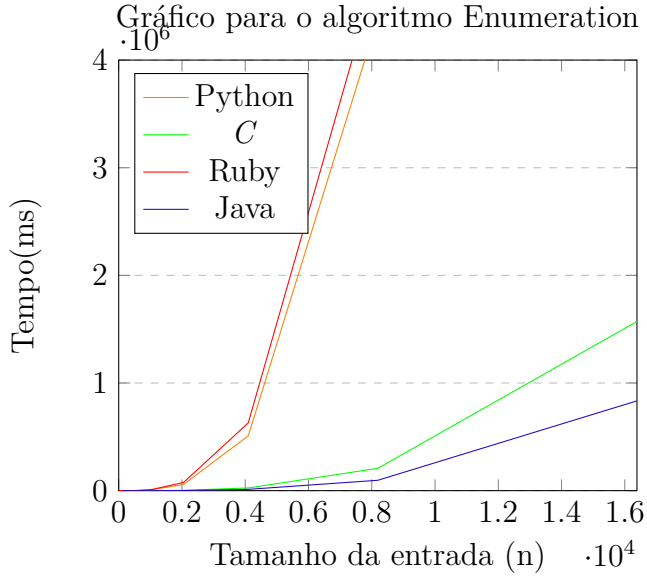
Entrada / Tempo(ms)	**128**	**254**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
Enumeration	28	152	1152	9138	75608	629392	4836653	37657843
Better Enumeration	0	2	8	37	141	589	2243	9054
Divide and Conquer	0	0	0	1	3	5	10	22
Dynamic	0	0	0	0	0	0	1	1

Como pode-se observar houve uma diferença muito grande nos tempos de execução dos algoritmos.

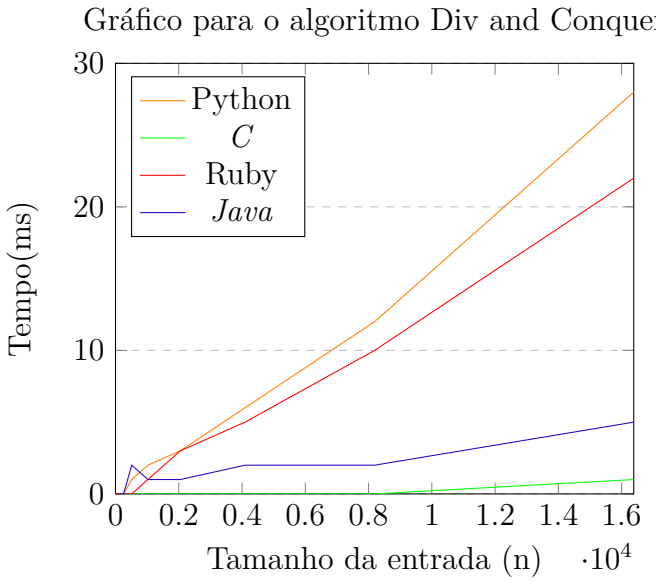
6 Discussão dos Resultados

Os resultados obtidos neste experimento reflete a complexidade do algotitmo implementado. O fato é que independente da liguagem utilizada, interpretada ou compilada, existe uma relação que se manteve em todos os testes realizados.

Para o algoritmo *Enumeration* e *BetterEnumeration*, foi notado que para uma grande quantidade de dados de entrada, o algoritmo se torna inviável para execução. Como discutido anteriormente, é possível observar que para uma entrada de aproximadamente 16.000 dados, foi levado aproximadamente 10 horas para realizar o processamento nas linguagens Python e Ruby.



Para os algoritmos *Divide and Conquer* e *Dynamic*, a quantidade de entrada de dados tem pouca influência no tempo de execução, visto que o crescimento da curva expressa pela quantidade de entradas sobre o tempo, é mais suave.



Foi notado que para as linguagens de programação C e Java, a execução da mesma entrada de dados para Python e Ruby foi mais rápida pelo fato de serem linguagens compiladas, porém as mesmas perdem em flexibilidade, por terem que passar pelo processo de compilação antes da execução. Já as linguagens interpretadas, tem vantagem em flexibilidade, mas sacrifica desempenho para isso. (FISCHER; GRODZINSKY, 1993)

7 Conclusões

Tomando base nas 4 diferentes linguagens de programação, podemos concluir que linguagens compiladas são mais rápidas em questão de desempenho, visualizando as tabelas 3 e 4 as ilustram o tempo de execução de algoritmos na linguagem Python e Ruby respectivamente, percebemos a diferença quando comparados ao C e Java.

Foi possível provar através de análises matemáticas e posteriormente os resultados, a complexidade do algoritmo e sua importância quando o assunto é performance.

Para aplicações cujo tempo de resposta deve ser rápido, não é viável utilizar complexidades cúbicas ($O(n^3)$) ou até quadráticas ($O(n^2)$) quando se tem uma grande quantidade de dados, pois, como mostrado no relatório, o tempo de processamento dos dados cresce exponencialmente e portanto, deixa de ser eficiente.

8 Referências

BORRADIALLE, G. *Test your knowledge: Solve the max subarray problem 4 ways*. 2017. Página Web. Acesso em: 01/12/2017. Disponível em: <https://web.engr-.oregonstate.edu/~glencora/wiki/uploads/max-subarray-project.pdf>. Citado na página 4.

FISCHER, A. E.; GRODZINSKY, F. S. *The anatomy of programming languages*. [S.l.]: Prentice Hall, 1993. Citado na página 15.

TAKAOKA, T. Efficient algorithms for the maximum subarray problem by distance matrix multiplication. *Electronic Notes in Theoretical Computer Science*, Elsevier, v. 61, p. 191–200, 2002. Citado na página 4.

WEDDELL, S. J. et al. Maximum subarray algorithms for use in astronomical imaging. *Journal of Electronic Imaging*, International Society for Optics and Photonics, v. 22, n. 4, p. 043011–043011, 2013. Citado na página 4.