Allison Alfredo O Sampaio
Mara Luci L Goulart
Rodrigo Paula da Silva

Análise Empírica de Algoritmos de Subarranjo Máximo

Relatório técnico de atividade prática solicitado pelo professor Rodrigo Campiolo na disciplina de Análise de Algoritmos do curso 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

A análise empírica se baseia na busca de dados relevantes e convenientes obtidos através de experências, vivência ou do pesquisar, para que assim se tenha como objetivo a chegada a novas conclusões. No relatório em questão, a análise empírica é apresentada de maneira a avaliar a performance de um algoritmo específico implementado de quatro maneiras diferentes em quatro linguagens distintas, em busca de demostrar qual deles apresenta o melhor desempenho. O algoritmo proposto para a análise dispõe-se a encontrar uma sublista contígua de maior soma a partir de uma lista de números inteiros, as linguagens de programação usadas para a implementação foram C, Python, Java e Ruby. A entrada apresenta um conjunto de valores aleatórios, mistos entre positivos e negativos, as mesmas foram utilizadas em todos os algoritmos. Os resultados obtidos são apresentados ao decorrer do relatório, a análise compara o tempo pelo tamanho das entradas e se constitui de duas formas, a implementação de quatro algoritmos em uma linguagem específica de programação e a implementação em cada linguagem de programação para um algoritmo específico.

Palavras-chave: analise empirica. analise de algoritmos.

Sumário

1 Introdução

A análise empírica visa auxiliar na avaliação de eficiência de um algoritmo, uma das medidas de eficiência é o tempo de execução. O tempo de execução geralmente varia de acordo com o tamanho das entradas (SILVA, 2012). A análise foi feita sobre o problema do subvetor máximo, que consiste em encontrar a maior soma em um vetor com números negativos e positivos (GOODRICH, 2002). Uma das soluções implementadas é a divisão e conquista na qual a instância dada do problema é dividida em duas ou mais instâncias menores, cada instância menor é resolvida usando o próprio algoritmo as soluções das instâncias menores são combinadas e então retornam com a solução da instância original (FEOLILOFF, 2003). Outra solução interessante é a programação dinâmica, aplicada quando o problema possui estrutura recursiva (FEOLILOFF, 2003). As soluções apresentadas foram implementadas nas linguagens de programação C, Python, Java e Ruby. O relatório está organizado da seguinte forma: na seção 2 é apresentado o objetivo, ou seja, medir a eficiência das implementações de acordo com o tempo de execução; na seção 3 é apresentada a fundamentação teórica dos conceitos presentes no trabalho; na seção 4 é descrito os materiais que foram utilizados; na seção 5 os resultados que foram obtidos após executar os quatro algoritmos em suas respectivas linguagens; na seção 6 a disussão dos resultados que foram conquistados; na seção 7 a conclusão ; e na seção 8 as referências utilizadas.

2 Objetivos

O objetivo deste trabalho é realizar a análise empírica de quatro algorítmos que utilizam os conceitos de: laços de repetição por força bruta (enumeration e better enumeration), divisão e conquista, e programação dinâmica. As entradas possuem os seguintes tamanhos, calculados a partir da potência de 2: $2^7(128)$, $2^8(256)$, $2^9(512)$, $2^{10}(1024)$, $2^{11}(2048)$, $2^{12}(4096)$. Após executar 5 entradas diferentes para cada tamanho calculou-se a média de tempo em milisegundos.

3 Fundamentação

3.1 Análise Empírica

A análise empírica visa mensurar a eficiência de um algoritmo, geralmente por meio do tempo de execução, considerando tempo de relógio como tempo de CPU. Essa medição se dá pela contagem de vezes que cada instrução no código é executada (HEINEMAN, 2016). Por meio dessa análise é possível detectar os pontos de melhoria necessário no algoritimo testado. Além da eficiência das intruções do código, ressalta-se que o hardware utilizado influencia no tempo de execução bem como no uso da memória (HEINEMAN,

2016). Os principais objetivos da análise empírica são: avaliar a corretude dos algoritmos, comparar a eficiência de diferentes algoritmos, comparar implementações alternativas do mesmo algoritmo, identificar a complexidade do algoritmo (BENTLEY, 1984).

3.2 Subvetor Máximo

No problema do subvetor máximo, temos um vetor com números inteiros positivos e negativos, e esperamos como saída a maior soma em um intervalo (HEINEMAN, 2016).

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k$$

Dentre os tipos de implementação do subvetor máximo estão o *enumeration*, *better enumeration*, *divisão e conquista*, e *programação dinâmica*, conforme descritos a seguir:

- Enumeration: Loop para cada par de indices i,j calculando a soma de a[k] tendo k=i até k=j e guardando a melhor soma encontrada até agora.
- Better Enumeration: Observe que no algoritmo anterior, a mesma soma é calculada muitas vezes. Em particular, observe que é calculado a soma de a[k] tendo k=i até k=j, isso pode ser calculado como a soma de a[k] tendo k=i até k=(j-1), sendo uma nova versão do primeiro algoritmo que aproveite esta observação.
- Divisão e Conquista: Se dividimos a entrada em duas metades, sabemos que a maior soma estará contida inteiramente na primeira parte, contida inteiramente na segunda parte ou começando na primeira parte e terminando na segunda. Os dois primeiros casos podem ser encontrados recursivamente. O último caso pode ser encontrado em tempo linear.
- Programação Dinâmica: O subarray máximo pode ou não usar o último elemento da entrada.

3.3 Linguagens de Programação

A escolha das linguagens de programação utilizadas na implementação dos algoritmos se baseia nos conhecimentos prévios dos autores do relatório em questão. São elas:

- C: Linguagem de programação compilada de propósito geral, estruturada, imperativa, procedural, padronizada pela ISO, criada em 1972, por Dennis Ritchie, no ATT Bell Labs, para desenvolver o sistema operacional Unix (C, 2017).
- Python: Linguagem de programação de alto nível, interpretada, de script, imperativa, orientada a objetos, funcional, de tipagem dinâmica e forte. Foi lançada por

Guido van Rossum em 1991. Atualmente possui um modelo de desenvolvimento comunitário, aberto e gerenciado pela organização sem fins lucrativos Python Software Foundation (PYTHON, 2017).

- Java: Linguagem de programação interpretada orientada a objetos desenvolvida na década de 90 por uma equipe de programadores chefiada por James Gosling. Diferente das linguagens de programação convencionais, que são compiladas para código nativo, a linguagem Java é compilada para um bytecode que é interpretado por uma máquina virtual (JAVA, 2017).

- Ruby: Linguagem de programação interpretada multiparadigma, de tipagem dinâmica e forte, com gerenciamento de memória automático, originalmente planejada e desenvolvida no Japão em 1995, por Yukihiro "Matz"Matsumoto, para ser usada como linguagem de script (RUBY, 2017).

4 Materiais

O material necessário para a execução dos algoritmos e coleta de dados foi um notebook com as seguintes especificações:

- Sistema operacional: Ubuntu 16.04.3 LTS (Xenial)
- Processador: Intel(R) Core(TM) i3-4010U CPU @ 1.70GHz (4 CPUs)
- Memória: 4 GB
- Compiladores: GCC 5.4.0 / Java 1.8 / Python 3.5 / Ruby 2.3.1p112

5 Procedimentos e Resultados

O procedimento utilizado para realizar a análise empírica, foi a implementação dos algoritmos nas linguagens propostas junto com uma função de medição de tempo de execução. Após executar cinco vezes cada algoritmo em suas respectivas linguagens para seis tamanhos diferentes determinados a partir da potência de 2^n, coletamos os dados e geramos os gráficos presentes nesta seção.

5.1 Análise Matemática

A seguir são apresentados os pseudocódigos dos algoritmos utilizados na análise.

5.1.1 Algoritmo enumeration

Algorithm 1 Enumeration

```
1: function ENUMERATIO(v)
2:     soma ← 0
3:     max ← v[1]
4:     for i in 1..n do
5:         for j in i..n do
6:             for k in i..j do
7:                 soma ← soma + v[k]
8:             end for
9:             if soma > max then
10:                max ← soma
11:                start, end ← i, j
12:            end if
13:        end for
14:    end for
15:    return (max, start, end)
16: end function
```

Análise assintótica Cada laço *for* executa em tempo $O(n)$, portanto o tempo de execução do algoritmo no pior caso é $O(n^3)$.

5.1.2 Algoritmo better enumeration

Algorithm 2 Better Enumeration

```
1: function BETTER-ENUMERATION(v)
2:     soma ← 0
3:     max ← v[1]
4:     for i in 1..n do
5:         for j in i..n do
6:             soma ← soma + v[j]
7:             if soma > max then
8:                 max ← soma
9:                 start, end ← i, j
10:            end if
11:        end for
12:    end for
13:    return (max, start, end)
14: end function
```

Análise assintótica Como são dois laços *for* aninhados e cada um executa em tempo $O(n)$, o tempo de execução do algoritmo no pior caso é $O(n^2)$.

5.1.3 Algoritmo divisão e conquista

Algorithm 3 divisão e conquista

```
1: function MAX-CROSS(A, e, m, d)
2:     l_sum ← −∞
3:     sum ← 0
4:     for ∀i ∈ m..e do
5:         sum ← sum + A[i]
6:         if sum > l_sum then
7:             l_sum ← sum
8:             max_l ← i
9:         end if
10:    end for
11:    r_sum ← +∞
12:    sum ← 0
13:    for ∀j ∈ (m + 1)..d do
14:        sum ← sum + A[j]
15:        if sum > r_sum then
16:            r_sum ← sum
17:            max_r ← j
18:        end if
19:    end for
20:    return (max_l, max_r, l_sum + r_sum)
21: end function
22:
23: function MAX-SUBARRAY(A, e, d)
24:    if e = d then
25:        return (e, d, A[e])
26:    end if
27:    m ← ⌊(e + d)/2⌋
28:    (l_esq, l_dir, l_sum) ← MAX-SUBARRAY(A, e, m)
29:    (r_esq, r_dir, r_sum) ← MAX-SUBARRAY(A, m + 1, d)
30:    (c_esq, c_dir, c_sum) ← MAX-CROSS(A, e, m, d)
31:    if (l_sum ≥ r_sum) ∧ (l_sum ≥ c_sum) then
32:        return (l_esq, l_dir, l_sum)
33:    else if (r_sum ≥ l_sum) ∧ (r_sum ≥ c_sum) then
34:        return (r_esq, r_dir, r_sum)
35:    else
36:        return (c_esq, c_dir, c_sum)
37:    end if
38: end function
```

Análise assintótica Cada chamada de MAX-SUBARRAY executa em tempo $T(n) = T(n/2)$, e o procedimento MAX-CROSS em tempo $\theta(n)$, portanto a recorrência do algo-

ritmo é dada por: $T(n) = 2T(n/2) + \theta(n)$. Pelo método mestre, $T(n) = \theta(n \log n)$ para o pior caso.

5.1.4 Programação dinâmica

Algorithm 4 programação dinâmica (Kadane's algorithm)

```
1: function MAX-SUBARRAY(v)
2:     current ← maximum ← v[1]
3:     for ∀x ∈ v do
4:         current ← max(x, current + x)
5:         maximum ← max(maximum, current)
6:     end for
7:     return maximum
8: end function
```

Análise assintótica Tempo de execução no pior caso: $O(n)$.

5.2 Resultados Python

A Tabela 1 apresenta a média de tempo dos resultados da execução de cada algoritmo em Python:

Tabela 1: Tempo em milisegundos utilizando cada algoritmo em Python

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Enumeration	41,9128	317,2804	2720,5762	22657,0328	184467,2862	1492802,713
Better Enumeration	4,1868038	17,1467958	68,5272922	271,4727162	1088,7628242	4320,3160314
Divide and Conquer	0,7718672	1,6408722	3,5756074	7,6058672	16,4137414	34,952661
Dynamic	0,1008058	0,1776342	0,3386812	0,7952252	1,2868458	2,595719

5.3 Resultados C

A Tabela 2 apresenta a média de tempo dos resultados da execução de cada algoritmo em C:

Tabela 2: Tempo em milisegundos utilizando cada algoritmo em C

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Enumeration	2,8864	16,5286	119,8916	939,4412	7479,771	55494,3894
Better Enumeration	0,0426	0,1608	0,63	2,4924	9,945	39,711
Divide and Conquer	0,0198	0,038	0,0728	0,1486	0,3014	0,6196
Dynamic	0,007	0,0126	0,0234	0,0408	0,0836	0,2528

5.4 Resultados Java

A Tabela 3 apresenta a média de tempo dos resultados da execução de cada algoritmo em Java:

Tabela 3: Tempo em milisegundos utilizando cada algoritmo em Java

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Enumeration	8,4	16,6	49	223,6	1730,8	13050,6
Better Enumeration	3,4	4,6	9,2	16,4	24,2	60,6
Divide and Conquer	3,8	4,2	6,2	7,4	7,6	9
Dynamic	3,4	3,8	4,6	5,2	5	5,4

5.5 Resultados Ruby

A Tabela 4 apresenta a média de tempo dos resultados da execução de cada algoritmo em Ruby:

Tabela 4: Tempo em milisegundos utilizando cada algoritmo em Ruby

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Enumeration	37,2079984	295,3568084	2053,3077692	17024,4681768	126279,8878782	1003486,1
Better Enumeration	1,0114974	4,0858984	15,8537784	62,529755	262,2592524	985,69246
Divide and Conquer	0,3232428	0,764659	1,5209966	3,0886588	6,1329074	12,426728
Dynamic	0,0236498	0,0464976	0,0893338	0,1851136	0,3241004	0,7343924

6 Discussão dos Resultados

O procedimento utilizado para realizar a análise empírica, foi a implementação dos algoritmos nas linguagens propostas junto com uma função de medição de tempo de execução. Após executar cinco vezes cada algoritmo em suas respectivas linguagens para seis tamanhos diferentes determinados a partir da potência de 2^n, coletamos os dados e geramos os gráficos presentes nesta seção.

Os gráficos a seguir apresentam uma comparação entre as linguagens utilizadas:

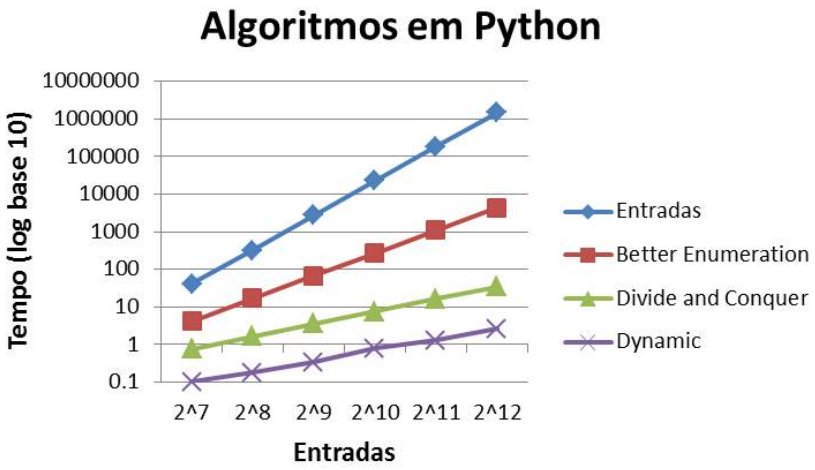

Figura 1: Média do tempo de execução em Python

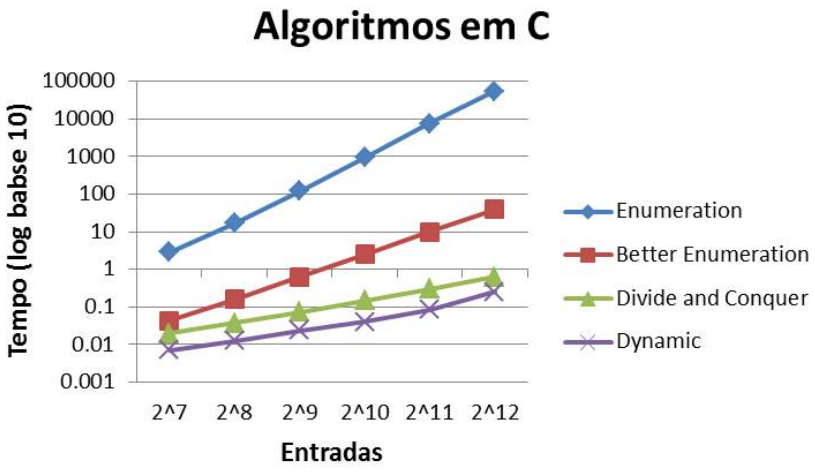

Figura 2: Média do tempo de execução em C

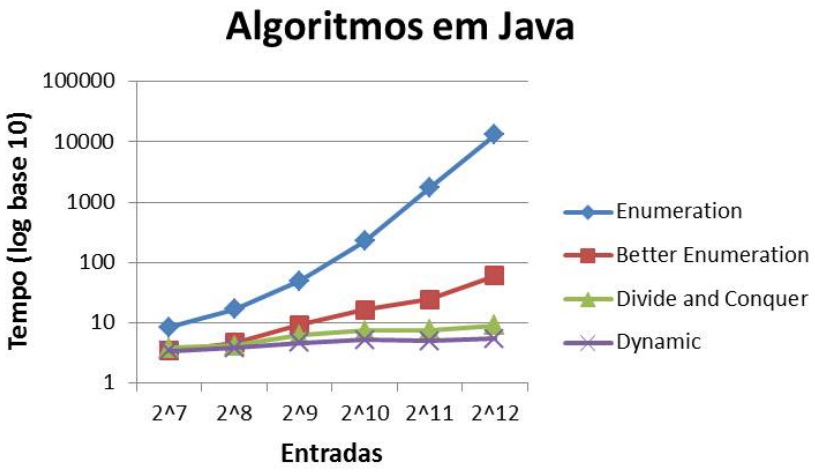

Figura 3: Média do tempo de execução em Java

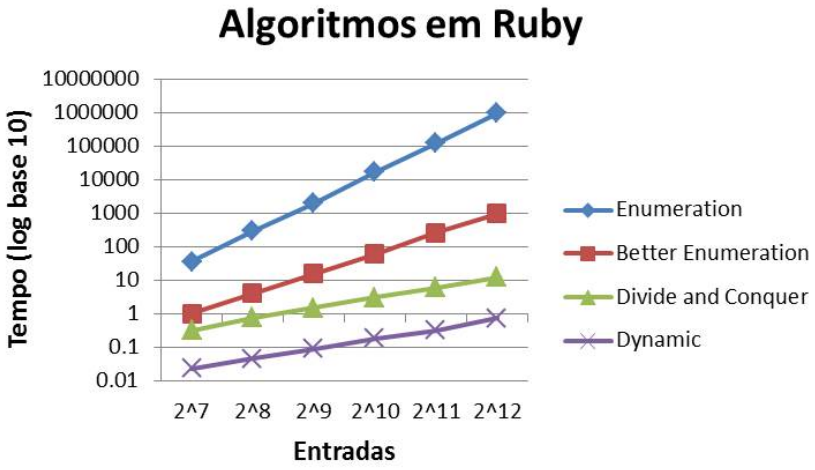

Figura 4:Média do tempo de execução em Ruby

As tabelas 5,6,7 e 8 apresentam a comparação entre as linguagens na implementação de cada algoritmo.

Tabela 5: Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Python	41.9128	317.2804	2720.5762	22657.03218	184467.2862	1492802.713
C	2.8864	16.5286	119.8916	939.4412	7479.771	55494.3894
Java	8.4	16.6	49	223.6	1730.8	13050.6
Ruby	37.2079984	295.3568084	2053.307769	17024.46818	126279.8879	1003486.179

Tabela 6: Better Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Python	4.1868038	17.1467958	68.527922	271.4727162	1088.762824	4320.316031
C	0.0426	0.1608	0.63	2.4924	9.945	39.711
Java	3.4	4.6	9.2	16.4	24.2	60.6
Ruby	1.0114974	4.0858984	15.8537784	62.529755	262.2592524	985.6924684

Tabela 7: Divisão e Conquista

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Python	0.7718672	1.6408722	3.5756074	7.6058672	16.4137414	34.952661
C	0.0198	0.038	0.0728	0.1486	0.3014	0.6196
Java	3.8	4.2	6.2	7.4	7.6	9
Ruby	0.3232428	0.764659	1.5209966	3.0886588	6.1329074	12.4267286

Tabela 8: Dinâmico

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Python	0.1008058	0.1776342	0.3386812	0.7952252	1.2868458	2.595719
C	0.007	0.0126	0.0234	0.0408	0.0836	0.2518
Java	3.4	3.8	4.6	5.2	5.0	5.4
Ruby	0.0236498	0.0464976	0.0893338	0.181136	0.3241004	0.7343924

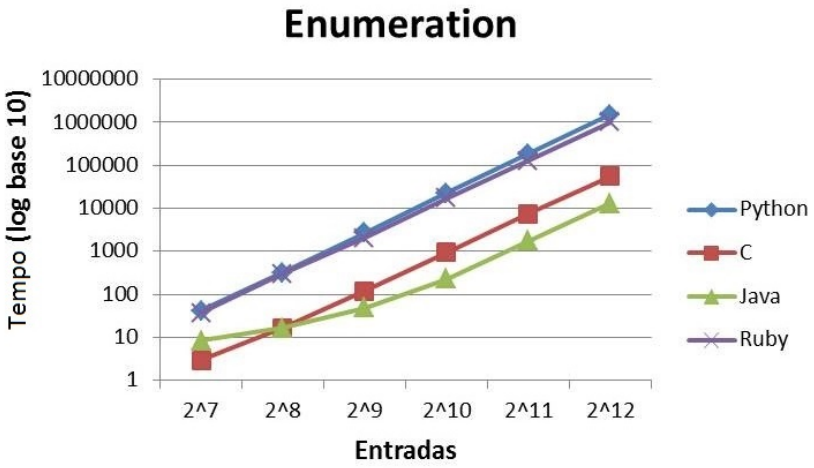

Figura 5: Tempo de execução Enumeration

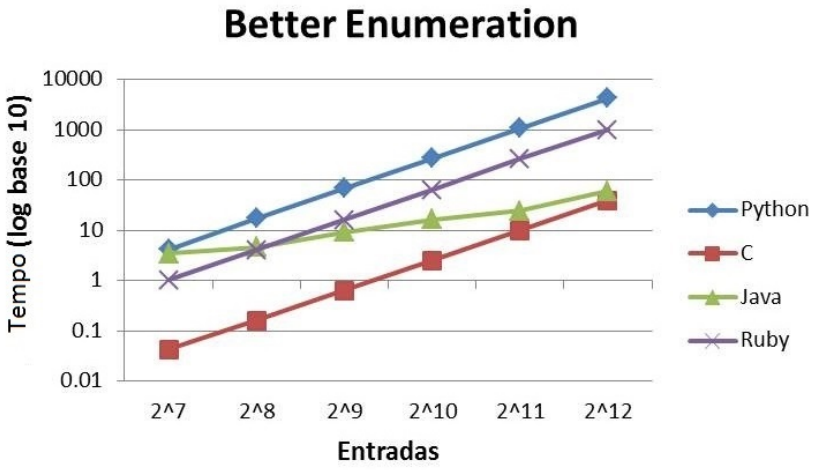

Figura 6: Tempo de execução Better Enumeration

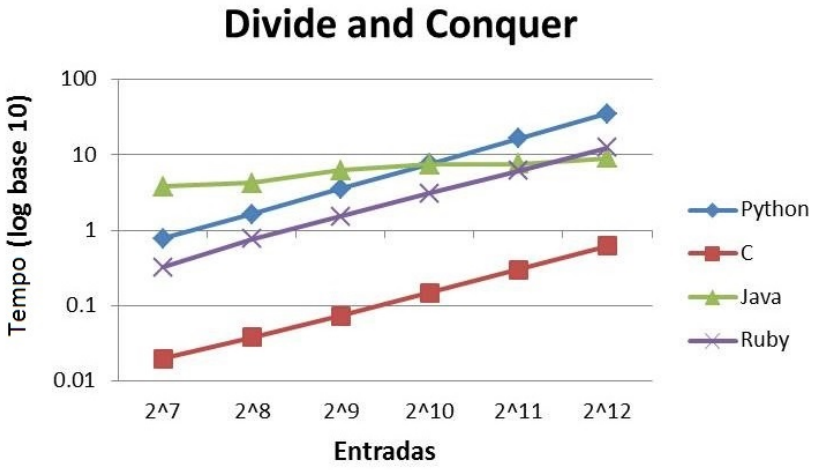

Figura 7: Tempo de execução Divide and Conquer

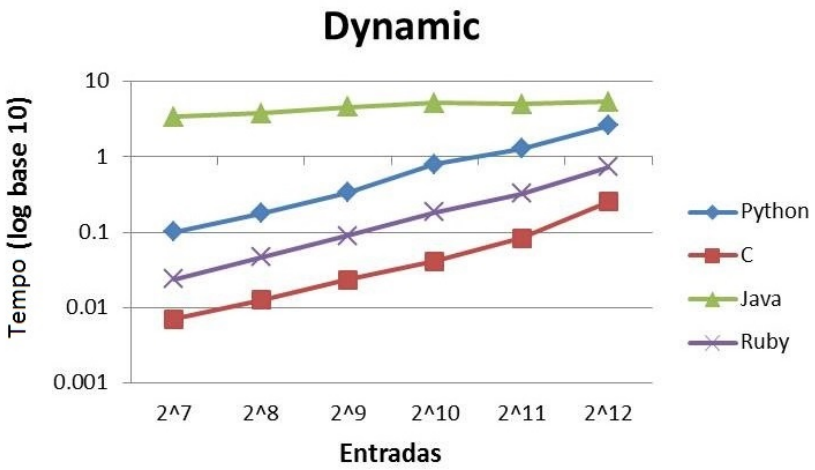

Figura 8: Tempo de execução Dynamic

O gráfico a seguir mostra a relação entre a complexidade de cada algoritmo e a linguagem na qual cada um foi implementado e seus respectivos tempos de execução, utilizando o a média dos tempos de execução com o tamanho do vetor 2^{12}.

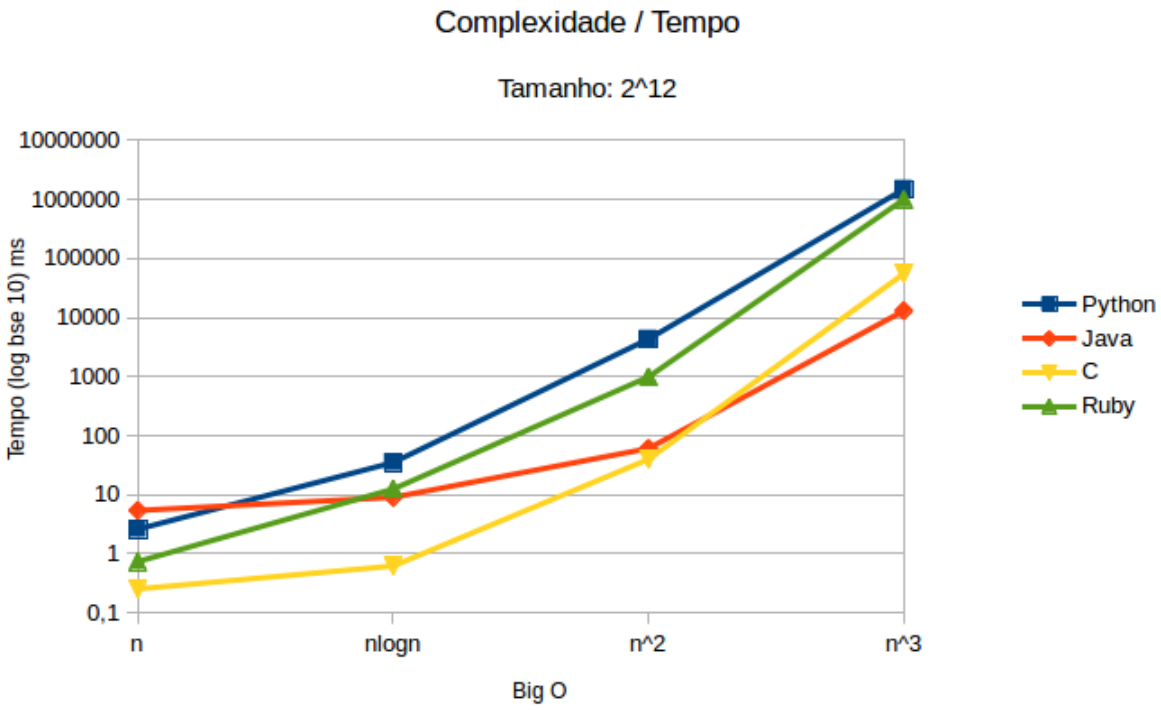

Figura 8: Gráfico da complexidade. Programação dinâmica (n), Divisão e conquista ($n \log n$), Better enumeration (n^2) e Enumeration (n^3).

7 Conclusões

Conforme a realização das análises de acordo com a complexidade dos algoritmos e seus tamanhos de entrada, os resultados obtidos nos levaram a conclusão de que cada algoritmo se comportou da maneira esperada. É notória a diferença em relação as linguagens de programação, como por exemplo, os tempos de execução dos algoritmos implementados em Python e Ruby, no qual foram semelhantes e consideravelmente mais altos que os algoritmos implementados em C e Java, com exceção do algoritmo implementação de forma dinâmica onde o Java se saiu pior que as demais linguagens. Uma observação interessante é que os testes feitos em Java mostraram uma curva de tempo inferior as outras linguagens. A linguagem C se mostrou na média a mais rápida para todos os casos testados. Por outro lado, Python foi a que demandou mais tempo de execução.

8 Referências

BENTLEY, J. *Programming pearls: algorithm design techniques*. [S.l.]: Communications of ACM, 1984. Citado na página 5.

C. *C Documentation*. [S.l.: s.n.], 2017. Citado na página 5.

FEOLILOFF. *Análise de Algoritmos*. [S.l.]: IME-USP, 2003. Citado na página 4.

GOODRICH, M. *Simplified Analyses of Randomized Algorithms*. [S.l.]: Stamford, 2002. Citado na página 4.

HEINEMAN, G. T. *Algorithms in a Nutshell*. [S.l.]: O'Relly, 2016. Citado 2 vezes nas páginas 4 e 5.

JAVA. *Java Documentation*. [S.l.: s.n.], 2017. Citado na página 6.

PYTHON. *Python Documentation*. [S.l.: s.n.], 2017. Citado na página 6.

RUBY. *Ruby Documentation*. [S.l.: s.n.], 2017. Citado na página 6.

SILVA, F. *Introdução á Complexidade de Algoritmos*. [S.l.]: USP, 2012. Citado na página 4.

Apêndice A - Execução em Python

As tabelas 9,10,11 e 12 apresentam em detalhes os resultados da execução dos algoritmos Enumeration, Better Enumeration, Divisão e Conquista e Dinâmico em Python:

Tabela 9: Tempo em milisegundos utilizando o algoritmo Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	42.151	310.904	2657.217	22434.462	184035.392	1545474.16
Entrada 2	41.413	317.333	2672.191	22615.633	184488.747	1476061.11
Entrada 3	43.539	325.1	2803.734	22721.924	184068.904	1469058.68
Entrada 4	41.575	310.47	2698.633	22553.105	184614.694	1479633.35
Entrada 5	40.886	322.595	2771.106	22960.04	185128.694	1493786.27

Tabela 10: Tempo em milisegundos utilizando o algoritmo Better Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	4.146335	16.572707	68.384231	269.795032	1093.729749	4416.517415
Entrada 2	4.152714	16.686511	68.788464	272.6177	1086.526093	4290.938856
Entrada 3	4.163052	16.736765	68.965081	271.764406	1084.971064	4283.029582
Entrada 4	4.251667	18.106293	68.465386	271.777104	1086.896538	4305.711889
Entrada 5	4.220251	17.631703	68.033299	271.409339	1091.690677	4305.382415

Tabela 11: Tempo em milisegundos utilizando o algoritmo Divisão e Conquista

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.667828	1.423543	3.046748	6.485194	14.017516	29.307443
Entrada 2	0.798053	1.706415	3.643792	7.883161	16.948674	35.697267
Entrada 3	0.79769	1.678185	3.702937	7.887057	16.965383	35.720155
Entrada 4	0.798676	1.701212	3.668637	7.864486	17.20358	38.363922
Entrada 5	0.797089	1.695006	3.815923	7.909438	16.933554	35.674518

Tabela 12: Tempo em milisegundos utilizando o algoritmo Dinâmico

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.093169	0.176887	0.329079	0.653361	1.272396	2.70259
Entrada 2	0.093514	0.175446	0.349052	0.650099	1.27342	2.567602
Entrada 3	0.095778	0.179314	0.335256	0.685758	1.303668	2.587185
Entrada 4	0.127196	0.175583	0.34712	1.327424	1.289522	2.559444
Entrada 5	0.094372	0.180941	0.332899	0.659484	1.295223	2.561774

Apêndice B - Execução em C

As tabelas 13,14,15 e 16 apresentam em detalhes os resultados da execução dos algoritmos Enumeration, Better Enumeration, Divisão e Conquista e Dinâmico em C:

Tabela 13: Tempo em milisegundos utilizando o algoritmo Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	4.196	21.92	24.559	941.707	7463.329	55395.621
Entrada 2	3.942	15.213	118.678	938.911	7463.067	55416.682
Entrada 3	2.09	15.093	118.661	938.816	7502.048	55615.719
Entrada 4	2.157	15.326	118.671	939.035	7493.533	55541.399
Entrada 5	2.047	15.091	118.889	938.737	7476.878	55502.526

Tabela 14: Tempo em milisegundos utilizando o algoritmo Better Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.043	0.162	0.633	2.492	9.932	39.699
Entrada 2	0.042	0.16	0.631	2.494	9.945	39.702
Entrada 3	0.043	0.161	0.629	2.491	9.945	39.665
Entrada 4	0.042	0.161	0.629	2.495	9.956	39.82
Entrada 5	0.043	0.16	0.628	2.49	9.947	39.669

Tabela 15: Tempo em milisegundos utilizando o algoritmo Divisão e Conquista

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.02	0.038	0.076	0.149	0.303	0.624
Entrada 2	0.02	0.037	0.073	0.149	0.304	0.618
Entrada 3	0.019	0.037	0.073	0.152	0.308	0.639
Entrada 4	0.02	0.039	0.073	0.148	0.297	0.611
Entrada 5	0.02	0.039	0.069	0.145	0.295	0.606

Tabela 16: Tempo em milisegundos utilizando o algoritmo Dinâmico

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.008	0.014	0.023	0.042	0.082	0.359
Entrada 2	0.007	0.012	0.023	0.04	0.099	0.367
Entrada 3	0.008	0.013	0.024	0.04	0.079	0.197
Entrada 4	0.006	0.012	0.023	0.041	0.078	0.17
Entrada 5	0.006	0.012	0.024	0.041	0.08	0.171

Apêndice C - Execução em Java

As tabelas 17,18,19 e 20 apresentam em detalhes os resultados da execução dos algoritmos Enumeration, Better Enumeration, Divisão e Conquista e Dinâmico em Java:

Tabela 17: Tempo em milisegundos utilizando o algoritmo Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	9	15	44	220	1519	11474
Entrada 2	8	18	49	207	1551	11259
Entrada 3	10	19	54	254	1813	14209
Entrada 4	8	16	53	199	1978	14046
Entrada 5	7	15	45	238	1793	14265

Tabela 18: Tempo em milisegundos utilizando o algoritmo Better Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	3	4	8	13	21	63
Entrada 2	3	4	8	15	21	54
Entrada 3	4	6	10	18	25	61
Entrada 4	4	4	11	17	30	62
Entrada 5	3	5	9	19	24	63

Tabela 19: Tempo em milisegundos utilizando o algoritmo Divisão e Conquista

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	3	5	6	8	6	10
Entrada 2	5	5	6	7	9	8
Entrada 3	4	5	6	7	8	10
Entrada 4	3	3	6	8	7	8
Entrada 5	4	3	7	7	8	9

Tabela 20: Tempo em milisegundos utilizando o algoritmo Dinâmico

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	3	3	5	5	5	7
Entrada 2	3	5	4	6	5	5
Entrada 3	5	3	4	6	5	5
Entrada 4	3	3	5	4	6	5
Entrada 5	3	5	5	5	4	5

Apêndice D - Execução em Ruby

As tabelas 21,22,23 e 24 apresentam em detalhes os resultados da execução dos algoritmos Enumeration, Better Enumeration, Divisão e Conquista e Dinâmico em Ruby:

Tabela 21: Tempo em milisegundos utilizando o algoritmo Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	37.475306	287.575254	2053.913105	15859.35741	124891.1271	1046631.26
Entrada 2	37.578997	295.818833	2054.799787	17472.3414	126578.5627	993004.1591
Entrada 3	37.018164	295.483732	2055.586099	17264.50218	126969.5165	994713.8862
Entrada 4	36.973827	295.939704	2053.536377	17264.53854	126474.5239	991552.6958
Entrada 5	36.993698	301.966519	2048.703478	17261.60135	126485.7092	991528.896

Tabela 22: Tempo em milisegundos utilizando o algoritmo Better Enumeration

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.993899	4.045848	15.901743	62.444654	257.524619	970.600161
Entrada 2	1.019504	4.206251	15.944256	62.815821	264.614168	994.41845
Entrada 3	1.006231	4.019069	16.133071	62.325197	262.938729	987.063743
Entrada 4	1.017184	4.106712	15.370222	62.667425	262.620891	987.714609
Entrada 5	1.020669	4.051612	15.9196	62.395678	263.597855	988.665379

Tabela 23: Tempo em milisegundos utilizando o algoritmo Divisão e Conquista

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.323958	0.667924	1.561697	3.269651	6.475671	13.141231
Entrada 2	0.326489	0.692811	1.804508	2.95368	5.794951	12.102943
Entrada 3	0.322917	0.723338	1.480201	2.968366	6.25653	12.076218
Entrada 4	0.3204	1.020464	1.446125	3.151476	6.2062	12.46555
Entrada 5	0.32245	0.718758	1.312452	3.100121	5.931185	12.347701

Tabela 24: Tempo em milisegundos utilizando o algoritmo Dinâmico

	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
Entrada 1	0.027963	0.050674	0.088554	0.167804	0.330539	0.643661
Entrada 2	0.023293	0.046227	0.090354	0.194647	0.32373	0.688758
Entrada 3	0.022569	0.04559	0.089591	0.1871	0.322464	0.662466
Entrada 4	0.022253	0.045106	0.089217	0.186277	0.321869	0.947824
Entrada 5	0.022171	0.044891	0.088953	0.18974	0.3219	0.729253