Alexandre Yuji Kajihara
Clodoaldo Basaglia da Fonseca

Análise empírica de algoritmos de subarranjo 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

Junho / 2017

Resumo

A análise empírica é uma maneira de verificar a execução de diferentes algoritmos, que tem o mesmo objetivo e conferir qual deles apresenta uma melhor performance.Um problema que apresenta algumas formas de se implementar é o problema subarranjo máximo, que dada uma sequência de valores, tentar encontrar os valores que tem o maior valor de soma. Com quatro das várias maneiras existentes de se implementar esse algoritmo, utilizamos a linguagem de programação Swift, na qual implementamos os algoritmos medindo o tempo de cada um deles. A nossa entrada foi um arranjo de valores negativos e positivos gerados aleatoriamente, na qual foram utilizados os mesmos em todos os algoritmos. Conseguimos obter sucesso em todos os tamanhos de vetores que propomos e foi visível a diferença de um algoritmo para outro que tem o mesmo objetivo.

Palavras-chave: análise de algoritmos. *Swift*,análise empirica.

Sumário

1 Introdução 4
2 Objetivos 4
3 Fundamentação 5
4 Materiais 6
5 Procedimentos e Resultados 6
6 Discussão dos Resultados 12
7 Conclusões 16
8 Referências 16

1 Introdução

A análise empírica também é importante na comparação de dois algoritmos que podem ou podem não tem a mesma ordem de complexidade (SANDERS, 2002).

Devido a existência do problema do subarranjo máximo, que de acordo Kamran Ashraf dado uma sequência de valores A= $\{a_1, a_2, ..., a_n\}$, nós iremos tentar encontrar a subsequência de A contínua a qual os valores tem a maior soma (ASHRAF, 2015). Dito isso, foi implementando quatro algoritmos que resolvem esse problema, em que um deles envolvem três iterações aninhadas, e o outro uma versão melhorada desse algoritmo anterior.

Uma das soluções para esse problema é a divisão e conqusita que consiste em: a instância dada do problema é dividida em duas ou mais instâncias menores, cada instância menor é resolvida usando o próprio algoritmo que está sendo definido, as soluções das instâncias menores são combinadas para produzir uma solução da instância original (FEOFILOFF, 2013a).

A última solução para esse problema é a de programação dinâmica: é uma espécie de tradução iterativa inteligente da recursão e pode ser definido, vagamente, como "recursão com apoio de uma tabela"(FEOFILOFF, 2013b).

Para a resolução dese problema e para comparação desse algoritmos, utilizamos a linguagem de programação *Swift*. O trabalho aqui apresentado está organizado da seguinte forma: na seção 2 é apresentado o objetivo desse trabalho do estudo, ou seja, mensurar o tempo de diferentes algoritmos que tem o mesmo propósito; na seção 3 são apresentadas as fundamentações que seriam o que foi necessário para realização desse trabalho; na seção 4 é descrito os materiais que foram utilizados; na seção 5 os procedimentos e resultados que foram obtidos após executar os quatro algoritmos; na seção 6 a disussão dos resultados que foram conquistados; na seção 7 a conclusão após realizar esse trabalho; e por fim na seção 8 as referências utilizadas.

2 Objetivos

O objetivo desse trabalho é realizar a análise empírica de quatro algoritmos que utilizam conceitos diferentes para resolução do problema do subarranjo máximo. As entradas foram de tamanhos diferentes, na qual foram escolhido um arranjo com tamanho de 10, 100, 1000 e 10000, e os valores presentes em cada uma dessas posições são diferentes. Após executar três vezes para cada um desses tamanhos, foi tirado uma média de tempo em milisegundos, que irá ser apresentando na seção 6.

3 Fundamentação

Os fundamentos necessários nesse trabalho foram basicamente, a instalação da linguagem de programação *Swift*. Primeiramente devemos fazer o *download* do *Swift* pelo terminal mesmo em que podemos utilizar o seguinte comando:

```
$ wget https://swift.org/builds/swift-3.1.1-release/ubuntu1610/swift-3
```

na qual escolhemos o *Swift* 3.1.1 para o Ubuntu 16.10, que foi o Sistema Operacional utilizado e ser a última versão até o momento da realização desse trabalho. Após isso, podemos extraír o arquivo realizando o seguinte comando:

```
$ tar -xf swift-3.1.1-RELEASE-ubuntu16.10
```

Com arquivo extraído teremos que fazer uma alteração em um arquivo presente em nosso sistema que é o *./profile*, que pode ser acessado da seguinte maneira:

```
$ sudo nano ~/.profile
```

Você pode utilizar outros editores vi, gedit, xed, Atom,etc. Na última linha desse arquivo iremos adicionar a linha abaixo. Isso é necessário para que não tenhamos que executar toda vez que ligarmos nossa máquina esse comando, em que devemos passar o lugar onde foi extraído aquele arquivo que realizamos o *download* que no meu caso foi no Documentos.

```
export PATH=~/Documentos/swift-3.1.1-RELEASE-ubuntu16.10/usr/bin/:"{\$P
```

Agora podemos executar o comando abaixo para ver as opções fornecidas pela linguagem:

```
swift --help
```

Em alguns casos pode ser necessário a instalação do clang, que pode ser instalado da seguinte maneira:

```
sudo apt-get install clang
```

Enfim, podemos escrever códigos na linguagem de programação *Swift*, em que para compilar podemos utilizar o seguinte comando:

```
swiftc name-file.swift
```

em que no nosso caso compilamos dá seguinte maneira:

```
swiftc main.swift algoritmo.swift
```

No nosso caso para evitar de ter que escrever todo esse comando para compilar utilizamos o Makefile, que pode ser instalado:

```
sudo apt-get install make
```

Para evitar tudo de digitar, podemos dar os dois seguinte comandos:

```
make all
```

```
make
```

caso queira remover o arquivo executável, é só digitar:

```
make clear
```

Por fim, para executar o nosso programa é só utilizarmos o comando:

```
./main
```

4 Materiais

Os materiais necessários foram um caderno, lápis, um *laptop* com as seguintes configurações:

- Sistema Operacional Ubuntu 16.10 64 bits
- Intel Core i7-6500U
- 16 GB de memória RAM (*Random Access Memory*)
- *Swift* na versão 3.1.1
- Editor de textoSublime Text Build 3126

5 Procedimentos e Resultados

Os procedimentos necessários para realizar a análise empírica dos algoritmos que resolvem o problema do subarranjo máximo foi a instalação do *Swift* na qual citamos na seção 3.

Após isso, escrevemos o nosso código-fonte. Antes de implementar os quatros algoritmos, criamos dois arranjos unidimensionais de inteiros. O primeiro subarranjo preenchemos com valores, que começam com o metade do tamanho do vetor multiplicado por menos um. Com esses valores incrementamos um a cada vez que se alterava de posição. Exemplo: se o tamanho do nosso arranjo é 100, na posição zero do vetor irá estar o valor -50, na posição 1, o valor -49, na posição dois o valor -48, e assim por diante, até que todos as posições estejam preenchidas, como podemos ver no algoritmo 1. Com o primeiro arranjo preenchido, iremos preencher o segundo, em que geramos um valor no intervalo

de zero até o tamanho do primeiro arranjo, e pegamos o valor que está na posição gerada aleatória e adicionamos no segundo arranjo. Removemos esse valor do primeiro arranjo e repetimos esse processo até preencher o segundo arranjo, isso nos garante que não tenhamos valores repetidos e que tenhamos valores negativos, essa função pode ser vista no algoritmo 2.

Para fins de testes rápidos, foram comentados em meio ao código, vetores já preenxidos com deferentes tamanhos. Para utilizá-los, basta seguir as instruções que seguem comentadas junto a eles.

Algorithm 1 Algoritmo que preenche o primeiro arranjo

```
1:  function geraVetor(valores, n)
2:      i = 0
3:      valor = (n/2) * (−1)
4:      while i < n do
5:          valores[i] = valor
6:          valor = valor + 1
7:          i = i + 1
8:      end while
9:      return
10: end function
```

Algorithm 2 Algoritmo que preenche o segundo arranjo

```
1:  function preencheVetor(valores, vetor, n)
2:      i = 0
3:      valor = (n/2) * (−1)
4:      while i < n do
5:          posicao = (random()%valores.length)
6:          vetor[i] = valores[posicao]
7:          remove(valores[posicao])
8:          i = i + 1
9:      end while
10:     return
11: end function
```

Após termos o nosso arranjo com valores aleatórios iremos aplicar os mesmos em funções que tentam resolver o subarranjo máximo. O primeiro algoritmo é chamado de *enumeration()* utilizando três iterações aninhadas, para tentar achar o valor, como pode ser visto no algoritmo 3.

Feito o algoritmo 3 utilizamos a mesma idéia para realizar o segundo quese trata de uma melhoria do primeiro. Uma de suas diferenças que podemos notar é um laço aninhados a menos, do que em relação ao algoritmo 3, o segundo algoritmo pode ser visto 4.

O penúltimo algoritmo utilizado foi dito em seções anteriores, utilizam o conceito de divisão e conquista. Basicamente ele divide o arranjo em metade e vai calculando o

Algorithm 3 Algoritmo *enumeration()*

```
1: function ENUMERATION(vetor, n)
2:     i = 0, j = 0, k = 0
3:     somatorio = 0, melhor = 0
4:     while i < n do
5:         j = i
6:         while j < n do
7:             somatorio = 0
8:             k = i
9:             while k < j do
10:                somatorio = somatorio + vetor[k]
11:                k = k + 1
12:            end while
13:            if somatorio > melhor then
14:                melhor = somatorio
15:            end if
16:            j = j + 1
17:        end while
18:        i = i + 1
19:    end while
20:    return melhor
21: end function
```

Algorithm 4 Algoritmo *betterEnumeration()*

```
1: function BETTERENUMERATION(vetor, n)
2:     i = 0, j = 0
3:     somatorio = 0, melhor = 0
4:     while i < n do
5:         somatorio = 0
6:         j = i
7:         while j < n do
8:             somatorio = somatorio + vetor[j]
9:             if somatorio > melhor then
10:                melhor = somatorio
11:            end if
12:            j = j + 1
13:        end while
14:        i = i + 1
15:    end while
16:    return melhor
17: end function
```

somatório de cada uma dessas metades, e por fim verifica qual subarranjo possui o máximo valor, em que iremos apresentar logo abaixo o algoritmo 5.

O último algoritmo que iremos apresentar é o de programação dinâmica, em que criamos um arranjo cópia do original com uma posição a mais e preenchemos o mesmo

Algorithm 5 Algoritmo *divideAndConquer()*

```
1: function DIVIDEANDCONQUER(vetor, inicio, fim)
2:     if inicio > fim then
3:         return 0
4:     end if
5:     if inicio == fim then
6:         return max(0, vetor[inicio])
7:     end if
8:     meio = (inicio + fim)/2
9:     maiorEsquerda = 0
10:    somatorio = 0
11:    i = meio
12:    while i >= inicio do
13:        somatorio = somatorio + vetor[i]
14:        if somatorio > maiorEsquerda then
15:            maiorEsquerda = somatorio
16:        end if
17:        i = i − 1
18:    end while
19:    maiorDireita = 0
20:    somatorio = 0
21:    i = meio + 1
22:    while i <= fim do
23:        somatorio = somatorio + vetor[i]
24:        if somatorio > maiorDireita then
25:            maiorDireita = somatorio
26:        end if
27:        i = i + 1
28:    end while
29:    return max(maiorDireita+maiorEsquerda, max(divideAndConquer(vetor, inicio, meio), d
   1, fim)))
30: end function
```

com zeros. Após isso, verificamos quem é o maior o que está na cópia mais o original ou o que está na cópia e por fim, retornamos o maior valor, esse algoritmo pode ser visualizado abaixo 6.

Após termos todos esse algoritmos implementados que foram criados em um arquivo cujo nome é algoritmo.*swift*, a nossa função principal, que está no arquivo *main.swift* invoca as quatro funções presente no arquivo *algoritmo.swift* em que antes de invocar o algoritmo começamos a calcular o tempo, e após executar o algoritmo imprimimos o valor retornado de cada função, encerramos o tempo, multiplicamos o tempo calculado por 0.001, já que decidimos utiliziar a medida de milisegundos e imprimimos o resultado. Alguns detalhes importantes, dessa função e das demais, no algoritmo 7 a linha 2 é o tamanho do arranjo, em que ele pode ser alterado para qualquer valor inteiro positivo. Um outro detalhe que representamos nos algoritmos 7, 1, 2 é que quando temos somente

Algorithm 6 Algoritmo *dynamicProgramming()*

```
1: function DYNAMICPROGRAMMING(vetor, n)
2:     solucao[vetor.length + 1]
3:     i = 0
4:     resultado = 0
5:     while i < vetor.length + 1 do
6:         solucao[i] = 0
7:         i = i + 1
8:     end while
9:     i = 1
10:    while i < vetor.length + 1 do
11:        solucao[i] = max(solucao[i − 1] + vetor[i − 1], vetor[i − 1])
12:        i = i + 1
13:    end while
14:    resultado = solucao[0]
15:    i = 1
16:    while i <= vetor.length + 1 do
17:        if resultado < solucao[i] then
18:            resultado = solucao[i]
19:        end if
20:        i = i + 1
21:    end while
22:    return resultado
23: end function
```

a palavra *return* em qualquer linha representamos que essa função não retorna nada, já que as demais retornam um inteiro.

Enfim, estamos satisfeitos com os resultados que obtivemos, já que todos os valores que propomos foram executados com sucesso, todos obtendo o mesmo valor de subarranjo máximo, mas com valores de tempo diferentes.

Algorithm 7 Algoritmo *main()*

```
1: function MAIN
2:     n = 10
3:     vetor[n]
4:     valores[n]
5:     geraVetor(valores, n)
6:     preencheVetor(valores, vetor, n)
7:     imprime(vetor)
8:     incializaTempo()
9:     imprime(enumeration(vetor, n))
10:    tempoExecucao = encerraTempo()
11:    imprime(tempoExecucao * 0.001)
12:    incializaTempo()
13:    imprime(betterEnumeration(vetor, n))
14:    tempoExecucao = encerraTempo()
15:    imprime(tempoExecucao * 0.001)
16:    incializaTempo()
17:    imprime(divideAndConquer(vetor, 0, n − 1))
18:    tempoExecucao = encerraTempo()
19:    imprime(tempoExecucao * 0.001)
20:    incializaTempo()
21:    imprime(dynamicProgramming(vetor, n))
22:    tempoExecucao = encerraTempo()
23:    imprime(tempoExecucao * 0.001)
24:    return
25: end function
```

6 Discussão dos Resultados

Como dito na seção anterior, estamos contentes com os resultados já que conseguimos de executar todas as entradas que propusemos. O programa e os resultados que alcançamos estão no GitHub (KAJIHARA; FONSECA, 2017), divididos em pastas de acordo o tamanho da entrada. Os resultados que iremos mostrar é uma média, já que executamos três vezes cada o tamanho do arranjo, utilizando o mesmo vetor. Os valores abaixo apresentados consideramos os valores até quatro casas depois da vírgula, mas nos resultados que temos armazenado em (KAJIHARA; FONSECA, 2017) temos até 14 números após a vírgula, tendo casos em que temos até mais que 14 números depois da vírgula. Todos os valores que iremos apresentar estão sendo mensurados em milisegundos.

Acreditávamos já antes de realizar essa análise empírica, do problema do subarranjo máximo é que o tamanho de entrada iria fazer toda diferença. Após realizarmos, pudemos comprovar que o tamanho do arranjo faz totalmente diferença, já que quanto maior o tamanho, mais iterações seriam feitas para tentar resolver o problema.

Essa diferença pode ser visto nas tabelas 1, 2, 3 e 4 que apresentam os valores das três execuções para arranjos de tamanho 10, 100, 1000 e 1000 para cada um dos algoritmos. Foi possivel observar que o tempo aumenta proporcionalmente ao tamanho da entrada. Foi notado que , por exemplo, na tabela 4 o arranjo com tamanho 10 apresentou diferença entre a primeira e a terceira execução, já que em outros casos, como o arranjo de tamanho 100, na tabela 1 tivemos o tempo nas três execuções mais constante, com pouca variação entre cada uma das execuções.

Após ter executado, 12 vezes o programa apenas mudando o tamanho do arranjo, notamos o quanto ineficiente é o algoritmo 3, em questão de tempo de execução. Isso porque, como executamos um algoritmo por vez e imprimimos o valor do subarranjo máximo e o tempo de execução, ele é o que demora mais para resolver o problema, para imprimir as informações, já que com relação aos três demais algoritmos, logo que o primeiro termina os demais são muito mais rápidos para imprimir o valor do subarranjo máximo e o tempo de execução. Um dos fatores que acreditamos ser é o fato de ter três laçps de repetição aninhados. Também tentamos ir mais além na qual tentamos executar com um arranjo de 100000 posições, em que deixamos executar por cerca de 12 horas e mesmo assim nem se conseguiu obter o resultado o problema do subarranjo máximo, para o algoritmo 3.

Analisando os dados também notamos que a escolha de um algoritmo para a resolução de um problema faz toda diferença, principalmente para entradas de tamanho maiores. Por exemplo, se olharmos o gráfico 4 e percebemos notamos que há uma grande diferença entre o algortimo 3 em que ele executa em um tempo superior ao tempo dos demais.

Tabela 1: Tempo em milisegundos usando o algoritmo *enumeration()*

enumeration()	10	100	1000	10000
1ª execução	3,2007x10^{-8}	3,4340x10^{-6}	0,0030	3,1034
2ª execução	3,8027x10^{-9}	3,4379x10^{-6}	0,0031	3,0893
3ª execução	1,2302x10^{-7}	3,4379x10^{-6}	0,0030	3,0887
média	5,2943x10^{-8}	3,4366x10^{-6}	3,0333x10^{-3}	3,0938

Tabela 2: Tempo em milisegundos usando o algoritmo *betterEnumeration()*

betterEnumeration()	10	100	1000	10000
1ª execução	3,9935x10^{-9}	1,0502x10^{-7}	9,2869x10^{-6}	0,0009
2ª execução	6,9737x10^{-9}	1,0401x10^{-7}	8,9359x10^{-7}	0,0009
3ª execução	9,9539x10^{-7}	1,0800x10^{-7}	8,9830x10^{-6}	0,0009
média	3,3545x10^{-7}	3,6000x10^{-8}	1,8509x10^{-5}	9x10^{-4}

Tabela 3: Tempo em milisegundos usando o algoritmo *divideAndConquer()*

divideAndConquer()	10	100	1000	10000
1ª execução	7,0333x10^{-9}	2,9981x10^{-8}	3,0797x10^{-7}	3,2719x10^{-6}
2ª execução	1,1980x10^{-8}	2,9981x10^{-8}	3,1399x10^{-7}	3,3989x10^{-6}
3ª execução	1,5974x10^{-8}	3,3974x10^{-8}	3,2299x10^{-7}	3,3909x10^{-6}
média	1,1662x10^{-8}	3,1312x10^{-8}	3,1498x10^{-7}	3,3539x10^{-6}

Tabela 4: Tempo em milisegundos usando o algoritmo *dynamicProgramming()*

dynamicProgramming()	10	100	1000	10000
1ª execução	1,9967x10^{-8}	8,3029x10^{-8}	6,6000x10^{-7}	6,2209x10^{-6}
2ª execução	2,4020x10^{-8}	7,7962x10^{-8}	6,2400x10^{-7}	6,1360x10^{-6}
3ª execução	5,5015x10^{-8}	8,6009x10^{-8}	6,4998x10^{-7}	6,2929x10^{-6}
média	3,3000x10^{-8}	8,2333x10^{-8}	6,4466x10^{-7}	6,2166x10^{-6}

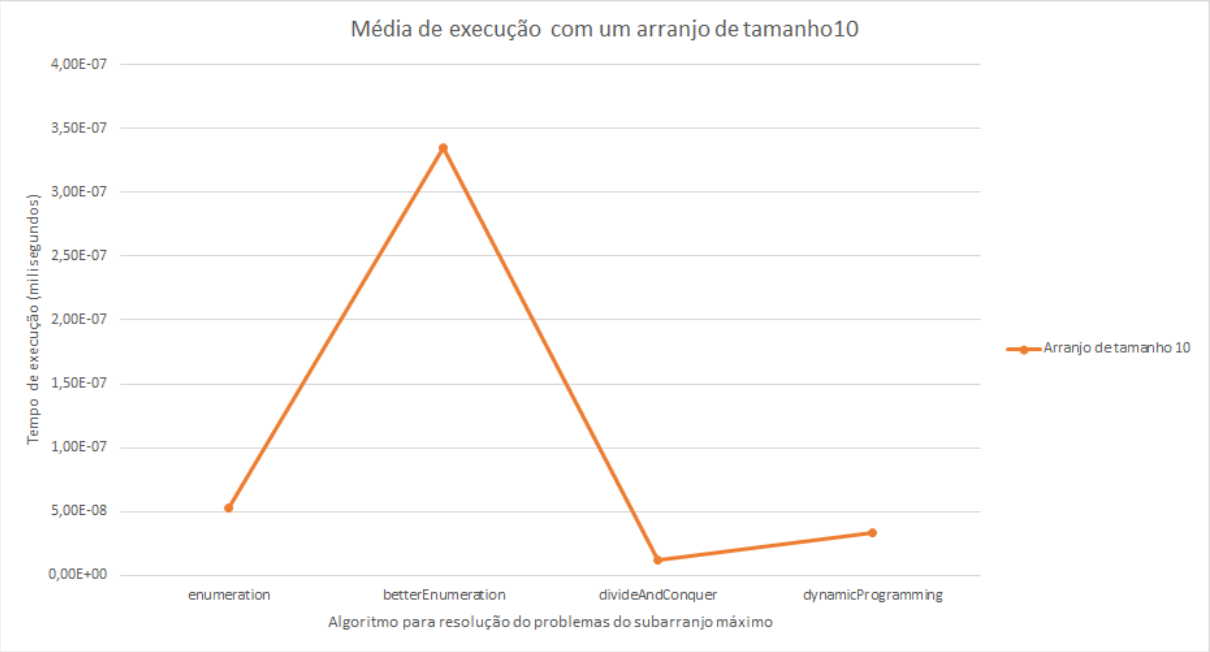

Figura 1: gráfico com a média de execução com o tamanho do arranjo sendo 10.

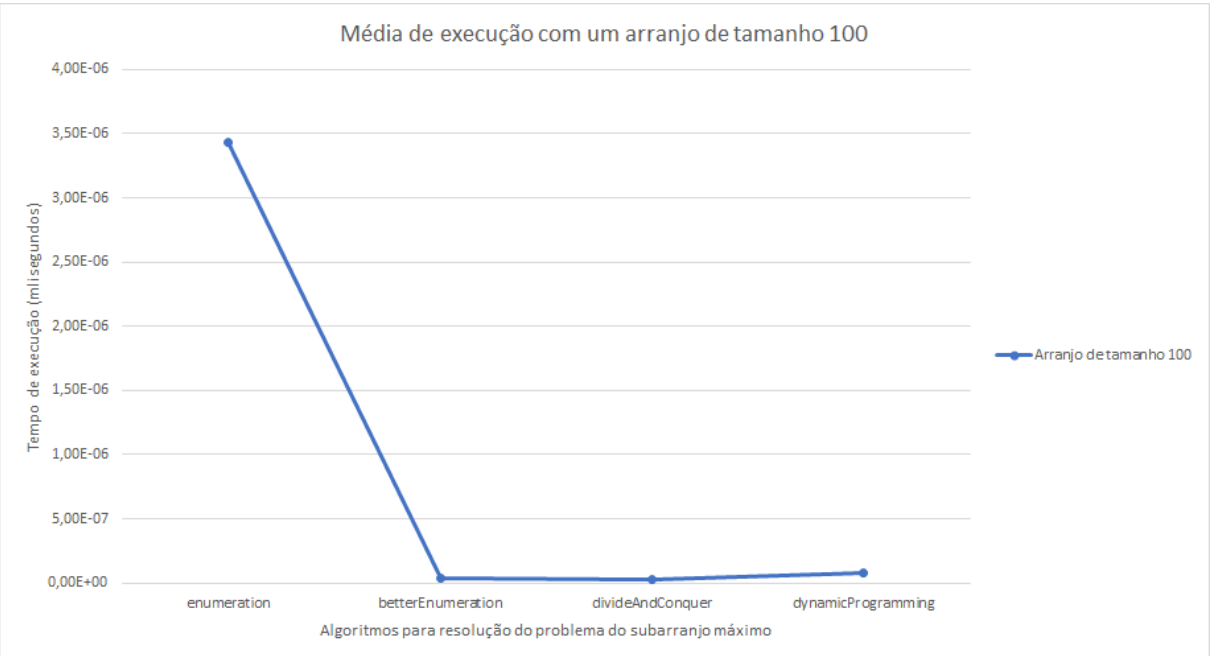

Figura 2: gráfico com a média de execução com o tamanho do arranjo sendo 100.

Figura 3: gráfico com a média de execução com o tamanho do arranjo sendo 1000.

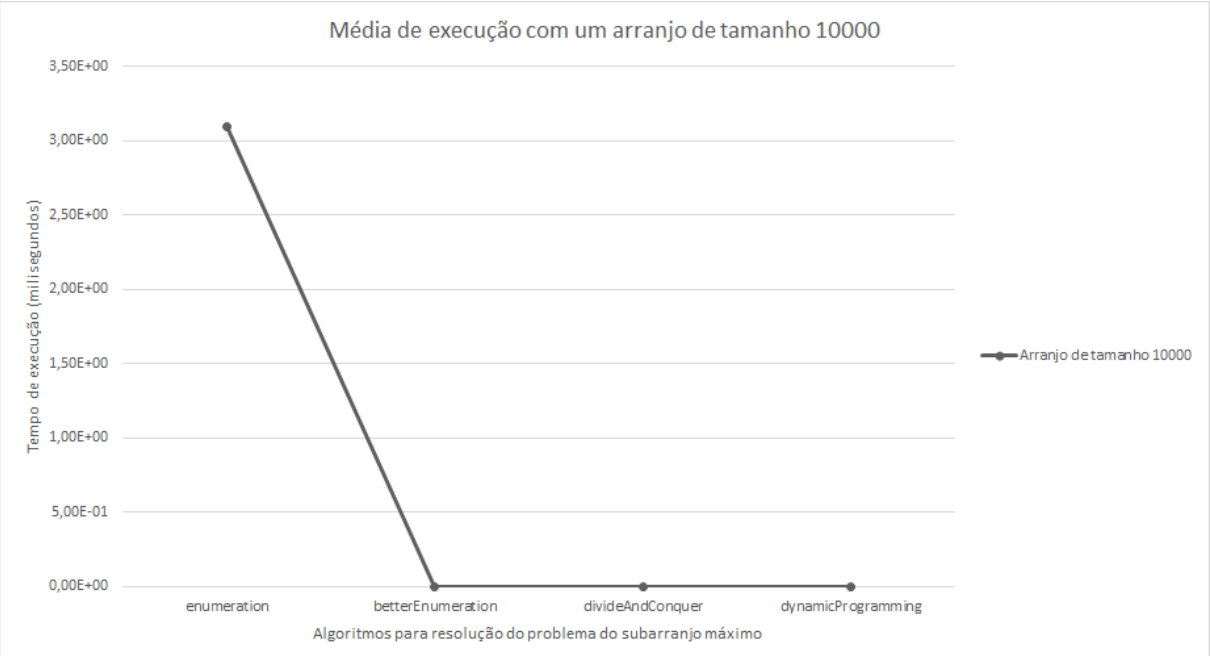

Figura 4: gráfico com a média de execução com o tamanho do arranjo sendo 10000.

7 Conclusões

Após utilizarmos a análise empírica para comparar o tempo de execução de algoritmos que procuram resolver o problema do subarranjo máximo, com tamanhos de entradas diferentes tiramos muitas lições. Uma delas é verificar as opções de algoritmos que temos para resolver um determinado problema, já que como vimos nesse problema do subarranjo máximo o algoritmo influenciam totalmente no tempo de execução de um programa. A outra lição que aprendemos com a realização desse trabalho foi que a entrada influenciam totalmente no algoritmo, já que mais iterações terão que ser feitas e o tempo de execução ficará mais elevado. Então se a nossa entrada for pequena, podemos utilizar qualquer um dos algoritmos, como podemos ver na figura 1 em que a diferença de tempo de execução não é tão grande assim. Agora se utilizarmos entradas de tamanho elevado, aí devemos considerar algoritmos de divisão e conquista ou programação dinâmica, em que há uma enorme diferença de tempo de execução, como visto na figura 4.

8 Referências

ASHRAF, K. *The Maximum Subarray Problem*. [S.l.], 2015. Disponível em: <https://pt.slideshare.net/KamranAshraf6/the-maximum-subarray-problem>. Acesso em: 29.6.2017. Citado na página 4.

SANDERS lan (Ed.). *Teaching Empirical Analysis of Algorithms*, v. 34. [S.l.]: Association for Computing Machinery (ACM), 2002. 321-325 p. Citado na página 4.

FEOFILOFF, P. *Divisão e conquista*. [S.l.], 2013. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/divide-and-conquer.html>. Acesso em: 29.6.2017. Citado na página 4.

FEOFILOFF, P. *Programação dinâmica*. [S.l.], 2013. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html>. Acesso em: 29.6.2017. Citado na página 4.

KAJIHARA, A.; FONSECA, C. B. da. *analise-empirica-20171*. [S.l.], 2017. Disponível em: <https://github.com/xaaaandao/analise-empirica-20171.git>. Acesso em: 30.6.2017. Citado na página 12.