

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INSTITUTO METRÓPOLE DIGITAL

ANÁLISE EMPÍRICA

DAYANE PABLA DE ARAÚJO VILAÇA

ANA CLARA NOBRE MENDES

Natal, RN.

2015

DAYANE PABLA DE ARAÚJO VILAÇA

ANA CLARA NOBRE MENDES

ANÁLISE EMPÍRICA

Relatório técnico que contém a implementação de dois algoritmos para um mesmo problema.  
Suas performances foram analisadas através de experimentos empíricos, em seguida comparadas com a sua análise matemática de complexidade.

Professor: Selan Rodrigues dos Santos

Natal, RN.

2015

**SUMÁRIO**

|  |  |  |
| --- | --- | --- |
| **1** | **INTRODUÇÃO………………………………………………………………….** | **03** |
| **2** | **MÉTODO UTILIZADO………………………………………………………...** | **04** |
| **3** | **RESULTADOS………………………………………………………………....** | **09** |
| **4** | **DISCUSSÃO…………………………………………………………………....** | **11** |

**1. INTRODUÇÃO**

Este relatório tem como objetivo apresentar dois algoritmos que solucionam o problema da soma máxima de uma subsequência. Definido por:

*Dada uma sequência de inteiros (possivelmente negativos) A1, A2,...,An, encontrar o máximo valor de ∑jk=i* Ak.

O primeiro algoritmo possuí complexidade linear e o segundo complexidade quadrática. Mostraremos a performace de cada um deles através de experimentos empíricos comparados com a sua análise matemática de complexidade.

**2. MÉTODO UTILIZADO**

Para a resolução do problema da soma máxima de uma sequência foi criado um arquivo maxSubSum, na linguagem C++. Este arquivo contém a função int maxSubSumLinear( const vector<int> & a ), algoritmo linear, e a função int maxSubSumQuadratic( const vector<int> & a ), algoritmo quadrático.

|  |
| --- |
| **maxSubSum.cpp** |
| #include <ctime>  #include <cmath>  #include <vector> #include <cstdlib> #include <iostream> #include <string.h> #include <assert.h> #include <unistd.h> // Define a função sleep no Linux   using namespace std;  // Declaração das assinaturas das funções do programa. // Dessa maneira, podemos chamar as funções antes de defini-las. void printVector( const vector<int> & a ); int maxSubSumLinear( const vector<int> & a ); int maxSubSumQuadratic( const vector<int> & a ); void showTimeResults( const clock\_t & begin, const clock\_t & end ); void showClockResults( const clock\_t & begin, const clock\_t & end );  int main ( int argc, char\* argv[]) {  vector<int> a;  clock\_t begin, end;  // O programa só continua se a quantidade de parâmetros estiver correta.  assert(argc >= 2);    // Alimenta o gerador randomico com um seed baseado no tempo atual.  // Assim, garantimos que os valores para o vetor a mudem em cada execução.  srand(std::time(0));   // for que contém o tamanho n da instância.  for ( long int c = 0; c < 100000000; c++ )  {  a.push\_back(rand());  }  sleep(7);  if ( strcmp(argv[1], "linear") == 0 )  {  // O valor 100 desse for é referente às 100 execuções do algoritmo.  for ( register int i = 0; i < 100; i++ )  {  begin = clock();  maxSubSumLinear(a);  end = clock();  showClockResults(begin, end);  }  }  else if ( strcmp(argv[1], "quadratic") == 0 )  {  for ( register int i = 0; i < 100; i++ )  {  begin = clock();  maxSubSumQuadratic(a);  end = clock();  showClockResults(begin, end);  }  }  return 0; }  void printVector( const vector<int> & a) {  cout << "[ ";  // Imprime o vetor a na tela.  // Usa o iterator para "visitar" cada elemento de a.  for ( vector<int>::const\_iterator i = a.begin(); i != a.end(); ++i )  {  cout << \*i << " ";  }  cout << "]" << endl; }  int maxSubSumLinear( const vector<int> & a ) {  int sum = 0, maxSum = 0;  for ( unsigned int i = 0; i < a.size(); ++i )  {  sum += a[i];  if ( sum > maxSum ) maxSum = sum;  if ( sum < 0 ) sum = 0;  }    return maxSum; }  int maxSubSumQuadratic( const vector<int> & a ) {  int sum, maxSum = 0;  for ( unsigned int i = 0; i < a.size(); ++i )  {  sum = 0;  for ( unsigned int j = i; j < a.size(); ++j )  {  sum += a[j];  if ( sum > maxSum ) maxSum = sum;  }  }  return maxSum;  }  void showClockResults( const clock\_t & begin, const clock\_t & end )  {  // Exibe tempo de execução em segundos.  cout << (double)(end - begin)/CLOCKS\_PER\_SEC << endl; } |
| Esse código pode ser acessado em: <http://goo.gl/y2I1P1> |

Para auxiliar a compilação foi criado o arquivo Makefile:

|  |
| --- |
| **Makefile** |
| CC = g++  all: maxSubSum.cpp  $(CC) -Wall maxSubSum.cpp -o maxSubSum -g |
| Esse código pode ser acessado em: <http://goo.gl/uCzzg6> |

Em seguida a execução é feita da seguinte forma para os dois tipos de algoritmos testados:

>>> ./maxSubSum linear

>>> ./maxSubSum quadratic

Para armazenar os tempos das 100 execuções aleatórias de cada algoritmo com instância de tamanho *n*, e em seguida calcular a média aritmética dos tempos das 100 execuções foi criando um script python que nos retornará exatamente esses valores.

|  |
| --- |
| **runner.py** |
| #!/usr/bin/env python  from \_\_future\_\_ import division  import os, sys  try:  import sh except ImportError:  sys.exit("Can't import sh! Try install it with: \n pip install sh")  CURRENT\_DIR = os.path.dirname(os.path.realpath(\_\_file\_\_)) MAX\_SUM\_PATH = os.path.join(CURRENT\_DIR, 'maxSubSum')  if not os.path.exists(MAX\_SUM\_PATH):  sys.exit("No such executable file: %s" % MAX\_SUM\_PATH)  def execute(program, param):  outputs = []  print 'Executing %s algorithm...' % param  for i in xrange(1, 26):  print '>> Round %i' % i  output = program(param)  outputs.append(output)  print\_summary(output, param)  return outputs  def average\_time(times):  return sum(times) / float(len(times))  def print\_summary(output, param):  lines = output.split()  times = [float(x) for x in lines]    print "Average time: %f" % average\_time(times)  print "Execution times: %s" % ', '.join(lines)  def main():  max\_sum = sh.Command(MAX\_SUM\_PATH)  linear\_outputs = execute(max\_sum, 'linear')  quadratic\_outputs = execute(max\_sum, 'quadratic')  if \_\_name\_\_ == '\_\_main\_\_':  main() |
| Esse código pode ser acessado em: <http://goo.gl/ryjRST> |

Para a execução do script será necessária a instalação da biblioteca sh do python, que em uma máquina Linux pode ser instalada com o seguinte comando:

>>> sudo pip install sh

Feito isso o script poderá ser rodado da seguinte forma:

>>> python runner.py

Começará a ser listado no terminal informações como no exemplo a seguir até que seja completada as 25 entradas diferentes, finalizando o algoritmo linear logo em seguida o quadrático irá iniciar.

|  |
| --- |
| Executing linear algorithm...  >> Round 1  Average time: 0.001700  Execution times: 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0.01, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0  >> Round 2  Average time: 0.001800  Execution times: 0, 0.01, 0, 0.01, 0, 0.01, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0, 0, 0.01, 0, 0, 0, 0, 0  >> Round 3 |

Os algoritmos foram testados em dois computadores, o primeiro x86\_64-linux-gnu e o segundo, uma máquina virtual x86\_64-linux-gnu. Ambos tiveram praticamente os mesmos resultados, apenas o segundo em alguns momentos demorou um pouco mais para finalizar os testes.

O compilador utilizado nos dois computadores foi o gcc version 4.9.2 20150212 (Red Hat 4.9.2-6).

**3. RESULTADOS**

A seguir temos os gráficos mostrando uma curva de crescimento (n no eixo X e tempo de execução no eixo Y) para cada algoritmo.

**4. DISCUSSÃO**

Neste trabalho, tínhamos como objetivo comparar o desempenho dos algoritmos linear e quadrático na resolução de um mesmo problema, analisando a complexidade de tempo, que é comumente estimada pela contagem do número de operações elementares performadas pelo algoritmo, onde a operação elementar toma a quantia fixa de tempo para realizar.

Após a análise dos resultados obtidos nos testes, e espressos pelos gráficos, podemos perceber e confirmar os conhecimentos teóricos sobre o desempenho dos algorítimos linear e quadrático.

Portanto, vemos que o algoritmo linear tem, em geral, um desempenho melhor em relação ao quadrático, e processa os dados em um menor espaço de tempo. Essa diferença pode ser notada de maneira ainda mais expressiva de acordo com o aumento das entradas, pois quanto maior o número de dados, maior ainda é o tempo gasto pelo algoritmo quadrático.

Temos isso porque um algoritmo é dito que usa tempo linear, ou tempo O(n), se sua complexidade de tempo é O(n). Informalmente, isto significa que para entradas grandes o suficiente o tempo de execução delas aumentam linearmente com o tamanho da entrada. Por exemplo, um procedimento que adiciona todos os elementos em uma lista requer tempo proporcional ao tamanho da lista. Esta descrição é levemente imprecisa, visto que o tempo de execução pode desviar significantemente de uma proporção precisa, especialmente para valores pequenos de n.

Já o quadrático , representado por O(*n*2), ocorre quando os itens de dados são processados aos pares, muitas vezes em uma repetição dentro da outra. Por exemplo, quando *n* é mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos, como podemos notar claramente nos gráficos.