Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Estruturas de Dados Básicas I • IMD0029
– Lista de Exercícios –
7 de agosto de 2014

O objetivo desta lista é oferecer a oportunidade de implementar vários algoritmos para um mesmo problema, testando sua performance através de experimentos empíricos e comparando-a com sua análise matemática de complexidade. Além disso, algumas questões requerem a implementação de soluções (programas) cujo algoritmo deve possuir complexidades especificadas.

1 Problema da soma máxima de uma subsequência

O problema da **soma máxima de uma subsequência** pode ser definido como:

> *Dada uma sequência de inteiros (possivelmente negativos)* $A_1, A_2, \ldots, A_n$*, encontrar o máximo valor de* $\sum_{k=i}^{j} A_k$.

Por conveniência, a soma máxima de uma subsequência é *zero* se todos os inteiros forem negativos. Por exemplo, para a entrada $-2, 11, -4, 13, -5, -2$, a resposta é 20 (A_2 até A_4).

Para solucionar este problema existem, pelo menos, quatro algoritmos com complexidade temporal distinta. Execute as seguintes tarefas.

1.1 Tarefa #1

Projete e implemente, no mínimo, **dois** algoritmos de complexidades distintas para resolver o problema da soma máxima de uma subsequência. A seguir, realize testes empíricos comparativos entre seus desempenhos para entrada $n = 100$, 1000, 10000, 100000, onde n é o número de elementos na sequência. Você não necessariamente precisa utilizar os valores de n sugeridos aqui. Procure adequar os valores de n de acordo com capacidade de processamento da máquina usada nos experimentos. Tente também, dentro do possível, coletar dados com a maior variação de n possível, de maneira a melhorar a amostragem de dados necessária para gerar a curva de desempenho. Alternativamente, experimente gerar n em uma *escala logarítmica* ao invés de usar a *escala linear* sugerida[1].

Armazene os tempos de execução de cada algoritmo para os diversos valores de n e gere um gráfico mostrando uma curva de crescimento (n no eixo X e tempo de execução no eixo

[1] Consultar `http://en.wikipedia.org/wiki/Logarithmic_scale`

Y) para cada algoritmo. Faça o mesmo tipo de comparação considerando o número de passos executados.

Em seguida, elabore um relatório técnico com *i)* uma **introdução**, explicando o propósito do relatório; *ii)* uma seção descrevendo o **método** seguido, ou seja quais foram os materiais utilizados (caracterização técnica do computador utilizado, linguagem de programação adotada, compilador empregado, algoritmo implementado, etc.) e metodologia de comparação seguida (tempo, passos, memória); *iii)* os **resultados** alcançados (gráficos e tabelas) e, por fim; *iv)* a **discussão** dos resultados: O que você descobriu? Aconteceu algo inesperado? Se sim, por que? Afinal, qual o algoritmo mais eficiente? Que função matemática melhor se aproxima do gráfico gerado? A análise empírica é compatível com a análise matemática?

Um relatório técnico de algum valor acadêmico deve ser escrito de tal maneira que possibilite que uma outra pessoa que tenha lido o relatório consiga reproduzir o mesmo experimento. Este é o princípio científico da **reprodutibilidade**.

Note que para realizar uma comparação correta entre algoritmos, os *mesmos* valores gerados (randomicamente[2]) para cada tamanho n devem ser fornecidos como entrada para *os dois* algoritmos. Além disso, para cada valor de n é importante realizar, pelo menos, 100 medições para os mesmos dados e, ao final, calcular a média das 100 medições.

Cada algoritmo deve ser implementado na forma de uma função, cujo protótipo deve ter a seguinte assinatura, de acordo com a complexidade do algoritmo:

```
int maxSubSumLinear( const vector<int> & a );
int maxSubSumQuadratic( const vector<int> & a );
int maxSubSumCubic( const vector<int> & a );
int maxSubSumNLogN( const vector<int> & a );
```

onde a é o arranjo com a subsequência.

Os gráficos podem ser gerados usando uma planilha eletrônica ou programas de geração de gráficos, como o `gnuplot` que costuma vir instalado em sistemas linux.

1.2 Tarefa # 2

Fornecer versões modificadas dos programas do item anterior de forma que a função forneça qual é a subsequência que gera a soma máxima. Modifique os protótipos das funções de forma a retornar retornar *um único objeto* contendo o *valor* da soma máxima da subsequência e os *índices* de tal subsequência.

2 Permutação dos n primeiros inteiros

Suponha que você precisa gerar uma permutação *randômica* dos n primeiros inteiros. Por exemplo, $\{4, 3, 1, 5, 2\}$ e $\{3, 1, 4, 2, 5\}$ são permutações legais, mas $\{5, 4, 1, 2, 1\}$ não é, porque

[2]Para informações sobre como gerar números randômicos consulte o final deste documento.

o número 1 está duplicado e o 3 está faltando. Esta rotina é frequentemente usada em simulação de algoritmos. Assuma que existe um gerador de números (pseudo) randômicos, `randInt(i,j)`, que gera inteiros entre `i` e `j` com igual probabilidade. Considere os três algoritmos descritos à seguir:

I. Preencher o arranjo a de `a[0]` até `a[N-1]` da seguinte forma: para preencher `a[i]`, gere números aleatórios até obter um número que já não esteja em `a[0]`, `a[1]`,..., `a[i-1]`.

II. Mesmo que algoritmo (I), mas mantenha uma arranjo extra chamado `used`. Quando um número aleatório, `ran`, é armazenado pela primeira vez em `a`, faça `used[ran]=true`. Isto significa que quando for preencher `a[i]` com um número aleatório, é possível testar em um único passo se o número aleatório gerado já foi usado ou não, ao invés de (possivelmente) usar `i` passos para obter a mesma informação como no algoritmo (I).

III. Preencher o arranjo tal que `a[i]=i+1`. Então

```
for ( i = 1; i < n; i++ )
    swap( a[ i ], a[ randInt( 0, i ) ] );
```

1. Verifique, experimentalmente, a seguinte afirmação: *"Todos os três algoritmos geram apenas permutações legais e todas as permutações possuem a mesma probabilidade."*

2. Escreva um programa de teste que implemente cada algoritmo e os execute 10 vezes para se obter uma boa média do tempo de execução. Execute o programa (I) para $n = 250$, 500, 1000 e 2000; programa (II) para $n = 25000$, 50000, 100000 e 200000; programa (III) para $n = 100000$, 200000, 400000 e 800000.

3. Qual é o algoritmo com a pior performance? Qual o melhor?

3 Busca do maior elemento em uma matriz

A entrada do programa `findInMatrix.cpp` é uma matriz $n \times n$ de números inteiros sem repetição[3]. Cada linha individual possui elementos em ordem crescente da esquerda para direita. Cada coluna individual possui elementos em ordem crescente de cima para baixo. Desenvolva uma função com desempenho $O(n)$ no pior caso que decide se um número `X` está ou não na matriz. Esta função deverá receber a matriz como argumento e retornar `true` caso `X` esteja na matriz, ou `false`, caso contrário.

4 Crivo de Eratóstenes

A *Crivo de Eratóstenes* é um método usado para computar todos os primos[4] menores do que n. Começamos criando uma tabela de inteiros de 2 até n. Encontramos o menor inteiro, i,

[3] Você pode preencher a matriz diretamente ou solicitar ao usuário que forneça os números.

[4] Um número natural maior do que 1 cujos únicos divisores naturais são 1 e o próprio número.

que não foi 'marcado', imprimimos i e 'marcamos' seus múltiplos, i.e. $i, 2i, 3i, \ldots$. Quando $i > \sqrt{n}$, então o algoritmo termina. Agora para exibir o restante dos primos basta percorrer a tabela imprimindo todos os números que *não* estão marcados. Implemente tal algoritmo e analise sua complexidade.

5 Elemento majoritário

Um elemento *majoritário* em um arranjo A de tamanho n é um elemento que aparece mais do que $n/2$ vezes (portanto existe, no máximo, um elemento majoritário). Por exemplo, o arranjo

$$3, 3, 4, 2, 4, 4, 2, 4, 4$$

tem o 4 como elemento majoritário, enquanto que o arranjo

$$3, 3, 4, 2, 4, 4, 2$$

não possui elemento majoritário. Caso não exista elemento majoritário o programa deve indicar tal situação. Abaixo segue um esboço de um algoritmo para resolver o problema de identificar um elemento majoritário (se houver) em um arranjo:

> *Primeiro, um* candidato *a elemento majoritário é encontrado (esta é a parte difícil). O candidato é o único elemento que pode possivelmente ser o elemento majoritário. O segundo passo determina se o candidato é de fato o elemento majoritário. Isto é apenas uma busca sequencial sobre o arranjo.*
>
> *Para encontrar um candidato no arranjo A, utilize um segundo arranjo B. Então compare A_1 e A_2. Se eles são iguais, adicione um deles à B; caso contrário não faça nada. Então compare A_3 e A_4. Novamente, se são iguais adicione um à B, caso contrário não faça nada. Continue desta forma até que todo o arranjo A seja percorrido. Então, de forma recursiva, ache um candidato para B; este é o candidato para A (por que?)*

1. Como a recursão deve ser encerrada? (i.e. qual o caso base?)
2. Como lidar com o caso em que n é um número ímpar?
3. Como é possível evitar o uso de um arranjo extra B?
4. Escreva uma função para computar o elemento majoritário de um arranjo. Esta função deve receber o arranjo como uma referência constante para um `vector`; o elemento majoritário, se houver, deve ser retornado através de um parâmetro declarado como referência simples; a função deve retornar `true` se houver elemento majoritário, ou `false`, caso contrário. Confira o protótipo da função:

```
bool findMajority( const vector<int> & a, int & maj );
```

Timing e Randomização

Para a medição do tempo decorrido na execução de um determinado trecho de código é possível utilizar-se de duas funções diferentes. A primeira, `time`, captura o tempo *de relógio* decorrido entre o início e o fim do trecho. A segunda, `clock`, captura o *número de ciclos* de processador decorridos entre o início e o fim da marcação — portanto é possível obter o tempo aproximado de processamento real.

O Código 1 demonstra o uso das duas formas de medição de tempo em um program. O trecho a ser medido está destacado em vermelho e vai da linha 18 até 26.

Código 1 Programa para medir tempo de execução de um trecho de código. A medição é feita em tempo total (linha 16 e 27) e tempo de CPU (linha 17 e 28); trecho medido: linha 18 à 26 (em vermelho).

```
1  /**
2   *  Timing test
3   *  To compile use: g++ -Wall timing.cpp -o timing
4   */
5  #include <ctime>
6  #include <cmath>
7  #include <iostream>
8  using std::cout;
9  using std::endl;
10
11 int main() {
12   time_t  t0, t1; // time_t is defined on <ctime> as long
13   clock_t c0, c1; // clock_t is defined on <ctime> as int
14   double a, b, c;
15
16   t0 = time( NULL );
17   c0 = clock();
18   cout << "\tbegin (wall):  "<< t0 << endl;
19   cout << "\tbegin (CPU): "<< c0 << endl;
20   cout << "\t\tsleep for 7 seconds ...  \n";
21   sleep(7);  // process is suspended for 7 seconds: no CPU time wasten.
22   cout << "\t\tperform some computation ...  \n";
23   for (long dummyCount= 1; dummyCount < 100; dummyCount++)
24       for (long count = 1; count < 10000000; count++) {
25           a = sqrt(count); b = 1.0/a; c = b - a;
26       }
27   t1 = time( NULL );
28   c1 = clock();
29
30   cout << "\tend (wall):               " << t1 << endl;
31   cout << "\tend (CPU);                " << c1 << endl << endl;
32   cout << "\telapsed wall clock time: " << (t1 - t0) << " second(s)\n";
33   cout << "\telapsed CPU time:        " << (c1 - c0)/CLOCKS_PER_SEC <<
34           " second(s)\n";
35
36   return ( EXIT_SUCCESS );
37 }
```

A geração de números randômicos ou aleatórios é feita através de uma função, `rand()`, da biblioteca padrão do C++. Esta função retorna números inteiros entre *zero* e uma constante numérica identificada por `RAND_MAX`. Note, contudo, que a função `rand()` é um gerador de números *pseudo-aleatórios*. Isto quer dizer que a sequência de números gerados será sempre a mesma.

Para aumentar a flexibilidade de tal geração, permitindo que uma sequência de números seja diferente a cada execução, é necessário invocar previamente uma outra função denominada de `srand()`. Esta função permite fornecer uma *semente* para a geração de números aleatórios: diferentes sementes implicam em diferentes sequências de números aleatórios.

Mas como gerar uma semente diferente a cada execução? (caso contrário estaríamos com o mesmo problema de gerar sementes aleatórias!) A resposta é utilizar a função `time` (vista anteriormente) para recuperar qual o valor do relógio do sistema expresso em segundos decorridos à partir de meia noite das 00:00 horas de 1º de janeiro de 1970 UTC (Tempo Universal Coordenado). Como a cada execução o tempo será diferente (o tempo decorrido em segundos continua aumentando), tem-se a garantia (parcial) da geração de números aleatórios. O Código 2 demonstra um programa que faz uso destas funções para gerar números aleatórios.

Código 2 Programa para gerar números aleatórios.

```
#include <ctime>
#include <iostream>
using std::cout;
using std::endl;

#define NTrials 10 // Number of trials

void randomize(void) {  srand((int) time(NULL)); }

int randInt (int low, int high) {
  int k;
  double d;

  d = (double) rand() / ((double) RAND_MAX + 1);
  k = (int) (d * (high - low + 1));
  return (low + k);
}

int main() {
  cout << ">> On this computer, RAND\_MAX = " << RAND_MAX << endl;
  cout << ">>  " << NTrials << " calls to rand (same seed):\n";
  for ( int i = 0; i < NTrials; i++ )
      cout << rand() << endl;

  randomize();
  cout << "\n>>  " << NTrials << " calls to rand (time-based seed):\n";
  for ( int i = 0; i < NTrials; i++ )
      cout << rand() << endl;

  cout << "\n>>  " << NTrials << " calls to rand (range[1-10]):\n";
  for ( int i = 0; i < NTrials; i++ )
      cout << randInt(1,10) << endl;

  return ( EXIT_SUCCESS );
}
```

Considerações Finais

O trabalho deve ser desenvolvido em duplas e submetido via Sigaa de acordo com a data estabelecida na turma virtua. Você deve submeter um único arquivo (zip) contendo todas as respostas organizadas em subpastas para cada questão da lista.

$\sim$ FIM $\sim$