

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

Estruturas de Dados Básicas I • IMD0029

– Lista de Exercícios –

9 de fevereiro de 2015

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, \dots, 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 pelo menos 25 entradas diferentes.

Sugere-se o uso de uma *escala logarítmica* para determinar o número de dados para a instância de entrada. Exemplos de escala são 2^i , e^i e 10^i , com $i \in [0, 25]$. Claro, você não necessariamente precisa utilizar todos os valores de n sugeridos aqui. Procure adequar os valores de n aqui sugeridos de acordo com capacidade de processamento da máquina usada nos experimentos. Pode ser que para determinados algoritmos, uma entrada com $n = 10^{15}$ seja impossível de ser medida.

Para cada algoritmo armazene os tempos de 100 execuções aleatórias de cada instância com tamanho n . Depois calcule a média aritmética dos tempos das 100 execuções: este será o tempo médio da execução do algoritmo para uma instância do problema com tamanho n . Após coletar e calcular a média dos tempos de execução para todos os tamanhos de instância

n , gere um gráfico mostrando uma curva de crescimento (n no eixo X e tempo de execução no eixo Y) para cada algoritmo. Faça o mesmo tipo de comparação considerando o número de passos executados. No total, serão gerados 4 gráficos, se dois algoritmos forem comparados.

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 (randomicamente¹) para cada tamanho n devem ser fornecidos como entrada para os *dois* algoritmos.

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 *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

¹Para 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 $a[0], a[1], \dots, a[i-1]$.
- II. Mesmo que algoritmo (I), mas mantenha um 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?

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.

```
1  #include <ctime>
2  #include <iostream>
3  using std::cout;
4  using std::endl;
5
6  #define NTrials 10 // Number of trials
7
8  void randomize(void) { srand((int) time(NULL)); }
9
10 int randInt (int low, int high) {
11     int k;
12     double d;
13
14     d = (double) rand() / ((double) RAND_MAX + 1);
15     k = (int) (d * (high - low + 1));
16     return (low + k);
17 }
18
19 int main() {
20     cout << ">> On this computer, RAND_MAX = " << RAND_MAX << endl;
21     cout << ">> " << NTrials << " calls to rand (same seed):\n";
22     for ( int i = 0; i < NTrials; i++ )
23         cout << rand() << endl;
24
25     randomize();
26     cout << "\n>> " << NTrials << " calls to rand (time-based seed):\n";
27     for ( int i = 0; i < NTrials; i++ )
28         cout << rand() << endl;
29
30     cout << "\n>> " << NTrials << " calls to rand (range[1-10]):\n";
31     for ( int i = 0; i < NTrials; i++ )
32         cout << randInt(1,10) << endl;
33
34     return ( EXIT_SUCCESS );
35 }
```

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.

~ FIM ~