Aula prática #9

Notação Assintótica

a. $f(n) = 5n^2 - 100n + 34$; $g(n) = 2n^3 - 97$

A função g(n) cresce mais rapido para n->infinito, e portanto existe um c a partir de $n \leq n_0$ tal que $f(n) \leq c * g(n)$. Logo $f(n) = O(g(n))$ e $g(n) = \Omega(f(n))$.

b. $f(n) = 54n + 100$; $g(n) = 6n - 25$

Existe nos dois casos uma constante tal que $f(n) \leq c_1 * g(n)$ e $g(n) \leq c_2 * f(n)$ para n maior que um n_0, logo $f(n) = \Theta(g(n))$ e vice versa.

c. $f(n) = 2^n$; $g(n) = 6n^2$

Quando n->infinito, a função exponencial domina e existe um n_0 tal que para $n \geq n_0$ $g(n) \leq c * f(n)$ e portanto: $g(n) = O(f(n))$, $f(n) = \Omega(g(n))$.

d. $f(n) = 30; g(n) = log30$

Nos dois casos é possível achar constantes c_1 e c_2 para que $f(n) \leq c1 * g(n)$ e $g(n) \leq c2 * f(n)$. Logo $f(n) = \Theta(g(n))$ e vice e versa.

e. $f(n) = n \log n + n; g(n) = n^2$

Como $\log n \leq n$, então para n->infinito n^2 domina e existe, novamente, um n_0 e uma constante c para que $f(n) \leq c * g(n)$ e, portanto, $f(n) = O(g(n))$ e $g(n) = \Omega(f(n))$.

f. $f(n) = \sqrt{n}$; $g(n) = \log n$

Colocando no programa gnuplot para altos n's, é possível ver que sqrt(n) domina. Logo, existe um n0 e uma constante c para que $g(n) \leq c * f(n)$ e, portanto, $g(n) = O(f(n))$ e $f(n) = \Omega(g(n))$.

Complexidade do TAD Conjunto

1.

a. A função contains, uma vez que é necessário verificar os n elementos do conjunto no pior caso, é linear - $\Theta(n)$.

b. Linear - $\Theta(n)$: esta usa a função contains.

c. Linear - $\Theta(n)$: no pior do cenário, algorítimo percorre todo o array de size n (também usa contains).

d. Contante - $\Theta(1)$.

e. Constante - $\Theta(1)$

2.

a. A função contains passa a ter complexidade constante: $\Theta(1)$.

b. Contante - $\Theta(1)$.

c. Contante - $\Theta(1)$.

d. Constante - $\Theta(1)$.

e. Constante - $\Theta(1)$.

Mesmo diminuindo a complexidade, esta nova abordagem usa muito mais memória que o IntSet usando arrays. Caso um dos elementos seja 1000000, então o array deve ter 1000000 elementos (e true no arr[1000000]). É aquela troca entre eficiência e memória.

Previsão do tempo de execução

Para prever o tempo de execução de um algorítimo, usa-se a equação:

$$t_2 = \frac{f(n_2)}{f(n_1)} * t_1 \tag{1}$$

Imagine que tem um programa P implementado, que recebe como input n números. Ao experimentar executá-lo com testes aleatorizados, obteve os seguintes tempos de execução:

- Com n=300 demorou 1.5 segundos
- Com n=600 demorou 12.0 segundos
- Com n=1200 demorou 1m36s

a. Qual será a complexidade temporal mais provável do programa P? Justifique.

   Ao duplicar os inputs (de $n = 300$ para $n = 600$), o tempo aumenta em $\frac{12.0}{1.5} = 8$. Ao duplicar novamente de $n = 600$ a $n = 1200$, o tempo aumenta por um fator de 8 novamente. Logo, como $2^3 = 8$, a complexidade temporal de P é O(n^3) (no pior dos casos).

b. Indique uma estimativa de quanto tempo demoraria o programa para um caso com 5000 números. Justifique.

   Como se sabe que o programa tem uma complexidade temporal $O(n^3)$, então:

$$t_2 = \frac{5000^3}{300^3} * 1.5 \tag{2}$$

Portanto, t_2 é aproximadamente 2 horas.

Problema do subarray contíguo de soma máxima (Maximum subarray problem)

Em ciência dos computadores, o *problema do subarray contíguo de soma máxima* é um problema clássico e pode ser resolvido utilizando várias tecnicas de algorítimo, incluindo-se nestas o brute force, divide and conquer, dynamic programming e outras.

O objectivo deste exercício é ter algoritmos de diferentes complexidades para um mesmo problema e ir verificando a sua eficiência temporal.

Uma primeira solução com força bruta com tempo $\Theta(n^3)$

Quais são todas as subsequências contíguas possíveis? Seja $v[]$ um array contendo a sequência e começado na posição 0. As sequências possíveis são então todos os subarrays $v[i..j]$ tal que $0 \leq i \leq j \leq n$.

Uma solução exaustiva seria passar por todas estas subsequências e para cada uma delas calcular o valor da respectiva soma, escolhendo a melhor possível. Supondo que já temos a leitura feita para o array v[], uma maneira de fazer isto seria a indicada pelo código seguinte:

```
int maxSum = s[0];
for(int i=0;i<s.length;i++){
    for(int j=i;j<s.length;j++){
        int sum = 0;
        for(int k=i;k<=j;k++)
            sum+=s[k];
        if(sum>maxSum) maxSum = sum;
    }
}
System.out.println(maxSum);
```

Melhorando a solução para $\Theta(n^2)$

Intuitivamente, olhando para o código anterior podemos notar que em cada soma estamos a repetir muitos cálculos. Quando passamos do cálculo de soma(v[i..j]) para soma(v[i..j+1]) não precisamos de voltar a recalcular tudo (começando novamente em i, e basta-nos adicionar v[j+1] à soma anterior!

Dito de outro modo, soma(v[i..j+1]) = soma(v[i..j]) + v[j+1]. Podemos utilizar isto para remover o terceiro ciclo com k que tínhamos na solução anterior.

```
int maxSum = s[0];
for(int i=0;i<s.length;i++){
    int sum = s[i];
    for(int j=i;j<s.length;j++){
        if(j!=i)
           sum+=s[j];
    if(sum>maxSum) maxSum = sum;
    }
}
System.out.println(maxSum);
```

Melhorando ainda mais para $\Theta(n)$

Uma solução quadrática ainda não é suficiente e temos de a melhorar para tempo linear. Para isso vamos usar o *algoritmo de Kadane*.

Considere que best(i) representa o melhor subarray que termina na posição i. Sabemos como "caso base" que best(0) = v[0] (é o único subarray possível que termina na primeira posição).

Se soubermos o valor de best(i), como calcular o valor de best(i+1)?

- Se *best(i)* for um valor positivo, então *best(i+1) = best(i) + v[i+1]* pois como os subarrays têm de ser contíguos, então se aproveitarmos algo "de trás", terá que ser o melhor possível a terminar na posição anterior. Imagine por exemplo o array [4,-2,1,5]:

i	0	1	2	3
v[i]	4	-2	1	5
best(i)	4	2	3	??

best(2) = 3, obtido através do subarray [4,-2,1].
best(3) = best(2) + v[3] = 3 + 5 = 8, obtido através subarray [4,-2,1,5] (estendeu-se o subarray anterior).

- Se *best(i)* for um valor negativo, então *best(i+1) = v[i+1]*, pois o que está "para trás" só pode fazer decrescer a soma do array a terminar na posição *i+1*. Imagine por exemplo o array [-5,2,-3,4]:

i	0	1	2	3
v[i]	-5	2	-3	4
best(i)	-5	2	-1	4

best(2) = -1, obtido através do subarray [2,-1].
best(3) = v[3] = 4, obtido através subarray [4] (não se estendeu o subarray anterior).

Figure 1: Descrição do algorítimo de Kadane

Uma solução, utilizando o algorítimo de Kadane, é:

```
int[] best = new int[s.length];
best[0] = s[0];//base case
for(int i=1;i<s.length;i++){
    if(best[i-1]>0)//if positive, add sequence item to sum
        best[i] = best[i-1]+s[i];
    else //if negative, the sum is less then sequence item, so don't add
        best[i] = s[i];
}
```

```java
int maxSum = best[0];//initially assume the max sum is the first int on array best
for(int i=1;i<best.length;i++){
    if(best[i]>maxSum) maxSum=best[i];
}
System.out.println(maxSum);
```

No final de tudo, o melhor subarray é o melhor valor de best(i) para um qualquer i (o melhor pode terminar em qualquer posição).

Para calcular isto basta-nos percorrer uma única vez o array v[] e em cada iteração calcular em tempo constante o valor da melhor soma a terminar na posição actual usando a melhor soma a terminar na posição anterior. A complexidade temporal fica portanto linear.

Exercícios extras

ED222 - Salvando os Ruivaços

ED199 - Tesouros de Kilmia