Lista 1

19 de março de 2014

1 Exercícios Básicos

1.1 Na bibliografia

Dasgupta: Capítulo 0, exercícios 1 e 2. Exercícios 2.16 e 2.17
Tardos & Kleinberg: Todos exercícios do cap 2 do livro texto, exceto 7 e 8 letra b.
Cormen: 10.3-2,10.3-5,10.3-6,10.3-7,10.3-9(Primeira Edição) ou 9.3-2,9.3-5,9.3-6,9.3-7,9.3-9(Segunda Edição)

1.2 Dasgupta

Solução 2.16. O algoritmo é dividido em duas etapas.

Na primeira etapa encontramos o menor inteiro $i^* \geq 0$ tal que $A[2^{i^*}] \leq x < A[2^{i^*+1}]$ utilizando uma busca exponencial, ou seja, comparamos x com $A[2^0]$, depois x com $A[2^1]$, depois com depois x com $A[2^2]$, até encontrar o valor de i^*. Se $A[2^{i^*}] = x$ então o algoritmo para, caso contrário ele vai para segunda etapa. Como $A[k] = \infty$ para $k > n$ temos que $2^{i^*} \leq n$ e, portanto, $i^* \leq \log n$. Logo, gastamos $O(\log n)$ nesta primeira etapa.

Na segunda etapa o algoritmo realiza uma busca binária no intervalo $A[2^{i^*}, \ldots, 2^{i^*+1}]$ para encontrar x ou determinar que x não está no vetor. Essa etapa também custa $O(\log n)$ já que $2^{i^*+1} - 2^{i^*} \leq n$

Solução 2.17. Este problema pode ser resolvido com um procedimento similar a uma busca binária. Inicialmente testamos se $A[mid] = mid$, onde $mid = \lfloor (1+n)/2 \rfloor$. Em caso positivo, devolvemos mid. Caso contrário, temos dois casos

(a) Se $A[mid] > mid$ podemos descartar a metade 'direita' do vetor e procurar o inteiro i recursivamente no intervalo $A[1, \ldots, mid-1]$.

(b) Se $A[mid] < mid$ podemos descartar a metade 'esquerda' do vetor e procurar o inteiro i recursivamente na intervalo $A[mid+1, \ldots, n]$.

A condição de parada é quando buscamos i em um vetor de um único elemento.

1.3 Tardos & Kleinberg

Exercício 3. Em ordem crescente $f_2, f_3, f_6, f_1, f_4, f_5$.

Exercício 4 Para resolver este exercício, em alguns casos, é interessante comparar o logaritmo das funções em vez de compará-las diretamente. Por exemplo, temos que $g_1(n) = 2^{\sqrt{\log n}}$ e, portanto, $\log g_1(n) = \sqrt{\log n}$. Por outro lado, $g_4(n) = n^{4/3}$ de modo que $\log g_4(n) = 4/3 \log n$. Como $4/3 \log n$ cresce mais rápido que $\sqrt{\log n}$ temos que Logo, $g_1(n)$ é $O(g_4(n))$.

A ordem correta é $g_1, g_3, g_4, g_5, g_2, g_7, g_6$.

Exercício 5

a) Verdadeiro. Como $f(n)$ é $O(g(n)$ temos que existe $c \geq 0$ e um inteiro n_0, independente de n, tal que $f(n) \leq cg(n)$ para todo $n > n_0$. Logo, $\log f(n) \leq \log(cg(n)) = \log c + \log g(n)$ para todo n maior que n_0, o que implica que $\log(f(n))$ é $O(\log(g(n)))$

b) Falso. $f(n) = 2n$ e $g(n) = n$ é um contra-exemplo.

c) Verdadeiro. Como $f(n)$ é $O(g(n))$ temos que existe $c \geq 0$ e um inteiro n_0, independente de n, tal que $f(n) \leq cg(n)$ para todo $n > n_0$. Logo, $f(n)^2 \leq c^2 g(n)^2$ para todo n maior que n_0, o que implica que $f(n)^2$ é $O(g(n)^2)$

Solução do Exercício 8.(a)

Seja C um inteiro menor que n cujo valor será determinado em nossa análise. Considere o seguinte algoritmo.

$i \leftarrow 1$

Enquanto o primeiro vaso não estiver quebrado

Jogue ele da altura $\min\{i \times C, n\}$

i++

Fim Enquanto

$j \leftarrow 1$

Enquanto o segundo vaso não estiver quebrado

Jogue o segundo vaso da altura $(i-1)C + j$

$j++$

Fim Enquanto

No primeiro loop o vaso é jogado no máximo n/C vezes e no segundo loop no máximo C vezes. Logo, o vaso é jogado no máximo $C + n/C$ vezes. Escolhendo $C = n^{0.5}$, valor que minimiza a função $f(C) = C + n/C$, temos que o vaso é jogado no máximo $2n^{0.5}$ vezes.

Claramente, o limite de $2n^{0.5}/n$ quando n tende a infinito é 0.

1.4 Cormen

Exercício 10.3.7

1. Encontre a mediana m em tempo linear utilizando o algoritmo de seleção em tempo linear mostrado em sala.
2. $S' \leftarrow \emptyset$.
3. Para cada elemento x de S faça $S' \leftarrow S' \cup |x - m|$
4. Seja q o $(k+1)$-ésimo menor elemento de S'. Encontre q utilizando o algoritmo de seleção em tempo linear mostrado em sala
5. Obtenha os $k+1$ menores elementos de S' pivoteando em relação ao elemento obtido em 3.

Devido a transformação no passo 2, no conjunto S', os k menores elementos correspondem aos k mais próximos da mediana

1.5 Outros

1. Analise a complexidade do algoritmo abaixo determinando uma função $f(n)$ tal que $T(n)$, a complexidade de pior caso do algoritmo, é $\Theta(f(n))$.

   Leia(n);

   $x \leftarrow 0$

   Para $i \leftarrow 1$ até n faça

       Para $j \leftarrow i+1$ até n faça

           Para $k \leftarrow 1$ até $j-i$ faça

               $x \leftarrow x+1$

   Solução. Seja $T(n)$ o número de operações que o algoritmo realiza no pior caso. Temos,

   $$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{j-i} c \leq \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} c \leq cn^3$$

   Por outro lado,

   $$T(n) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=1}^{j-i} c = \sum_{i=1}^{n} \sum_{j=i+1}^{n} c(j-i+1) \geq$$

   $$\sum_{i=1}^{n/4} \sum_{j=3n/4}^{n} c(j-i+1) \geq \sum_{i=1}^{n/4} \sum_{j=3n/4}^{n} n/2 \geq \frac{cn^3}{32}$$

   Portanto, $T(n) = \Theta(n^3)$

2. Dizemos que um vetor `P[1..m]` ocorre em um vetor `T[1..n]` se `P[1..m] = T[s+1,.., s+m]` para algum s. O valor de um tal s é um deslocamento válido. Projete um algoritmo para encontrar todos os deslocamentos válidos em um vetor e analise sua complexidade em função de m e n.

   Solução

   ```
   DESLOCAMENTOS-VALIDOS(T, P)
   n = T.length
   m = P.length
   for i = 1 to n − m + 1
       for j = 1 to m
           if T[i + j − 1] ≠ P[j]
               BREAK
       REPORTAR-DESLOCAMENTO(T, i, i + m − 1)
   ```

   A complexidade de pior caso do algoritmo é $O(nm)$ já que o algoritmo executa um loop externo que itera $n-m+1$ vezes, um loop interno que itera no máximo m vezes e, dentro de cada loop, $O(1)$ operações elementares são realizadas.

3. Seja $A[1..n]$ um vetor que pode conter números positivos e negativos.

   Projete um algoritmo com complexidade $O(n^3)$ para determinar os índices i e j, com $i \leq j$, tal que $A[i] + \ldots + A[j]$ é máximo. Tente reduzir a complexidade para $O(n^2)$ e depois para $O(n)$.

   Solução Para obter $O(n^3)$, basta executar o seguinte pseudo-código.

   $SomaMax \leftarrow -\infty$

   For a=1 to n

       For b=a to n

           $aux \leftarrow 0$

           For c= a to b

               $aux \leftarrow aux + a[c]$

           End For

           If $SomaMax < Aux$ then

               $SomaMax \leftarrow aux$

               $i \leftarrow a$; $j \leftarrow b$

           End If

       End For

   End For

   Para obter um procedimento $O(n^2)$, basta utilizar o fato de que a soma de todos elementos da sublista que começa na posição a e termina na posição b é igual a $PREF[b] - PREF[a-1]$, aonde $PREF[j]$ é igual a $a[1] + a[2] + \ldots + A[j]$. O vetor $PREF$ pode ser calculado em $O(n)$ no início do procedimento.

   $PREF[aux] \leftarrow A[1]$; $SomaMax \leftarrow -\infty$

   For aux=2 to n

       $PREF[aux] \leftarrow PREF[aux-1] + A[aux]$

   End For

   For a=1 to n

       For b=a to n

           If $SomaMax < PREF[b] - PREF[a-1]$ then

               $SomaMax \leftarrow PREF[b] - PREF[a-1]$

               $i \leftarrow a$; $j \leftarrow b$

           End If

       End For

   End For

Para obter um procedimento $O(n)$, precisamos raciocinar um pouco mais. Seja j o menor inteiro no conjunto $\{1, 2, \ldots, n\}$ para o qual $a[1] + a[2] + \cdots + a[j]$ é negativo. Temos dois casos:

(a) j não existe. Neste caso, existe uma soma máxima que começa na posição 1.

De fato, assuma que a soma máxima comece em uma posição i maior que 1 e termine em uma posição $i' \geq i$. Neste caso, a soma que começa em 1 e termina em i' é maior ou igual a que começa em i e termina em $i' \geq i$, o que implica que ela também é máxima.

(b) j existe. Neste caso, ou existe uma soma máxima que começa em uma posição maior que j ou existe uma soma máxima que começa na posição 1 e termina em uma posição menor que j.

De fato, assuma que não existe uma soma máxima que começa em uma posição maior que j nem que existe uma soma máxima que começa na posição 1. Logo, existe uma soma máxima que começa em uma posição i menor ou igual a j e termina em uma posição $i' \geq i$. Novamente, a soma que começa em 1 e termina em i' é maior ou igual a que começa em i e termina em $i' \geq i$, o que gera uma contradição. Finalmente, não pode existir uma soma máxima que que começa na posição 1 e termina em uma posição j' maior ou igual a j porque tal soma é negativa se $j' = j$ e é menor que que a soma que começa em $j+1$ e termina em j', se $j' > j$.

A conclusão é que basta comparar a maior soma que começa na posição 1 e termina em uma posição menor que j com a maior soma que começa em uma posição maior que j. Isso pode ser feito de forma recursiva através do seguinte procedimento:

`EncontraSomaMaxima`(A,j)

Se $j > n$

Return 0

Fim Se

Sum $\leftarrow$ 0; MaxSum $\leftarrow$ 0;

Enquanto Sum ≥ 0 e $j \leq n$

Sum $\leftarrow$ Sum+A[j]

MaxSum $\leftarrow$ max{Sum,MaxSum}

j++

Fim Enquanto

Return max{ MaxSum,EncontraSomaMaxima(A,j) }

`Fim`

`Main`

Leia um vetor A indexado de 1 an n contendo n números.

EncontraSomaMaxima(A,1)

`Fim`

Seja $T(n)$ a complexidade de pior caso do procedimento para uma lista de tamanho n. Se $n = 1$ temos que $T(1) = 1$. Por outro lado, se $n > 1$, temos $T(n) = j + T(n - j)$ para algum $j \in \{1, \ldots, n\}$. Resolvendo a recursão temos que $T(n) = \Theta(n)$.

4. Resolvas as equações abaixo encontrando uma função $f(n)$ tal que $T(n) = \theta(f(n))$

   a $T(n) = 2T(n/2) + n^2$ para $n > 1$; $T(1) = 1$

   b $T(n) = 2T(n/2) + n$, para $n > 1$; $T(1) = 1$

   c $T(n) = T(n/2) + 1$, para $n > 1$; $T(1) = 1$

   d $T(n) = T(n/2) + n$, para $n > 1$; $T(1) = 1$

   Gabarito [a] $\Theta(n^2)$; [b] $\Theta((n \log n)$; $\Theta(\log n)$; $\Theta(n)$

5. Seja $A = \{a_1 < \ldots < a_n\}$ uma lista ordenada de números reais. A proximidade entre a_i e a_j é definida como $|a_i - a_j|$. Dados os inteiros j e k, encontre os k elementos de A mais próximos de a_j em $O(k)$.

   Solução. Execute o procedimento abaixo que em cada iteração compara o elemento da posição esq com o elemento na posiçao dir. Se o elemento da posição esq estiver mais próximo de a_j que o elemento da posição dir, diminuimos esq de uma unidade, caso contrário aumentamos dir de uma unidade. Utilizamos o 'truque' de inicializar $a[0]$ e $a[n+1]$ com valor infinito para evitar tratar casos em que os ponteiros esq e dir chegam ao início e final da lista A, respectivamente.

   $a[0] \leftarrow \infty$; $a[n+1] \leftarrow \infty$

   $esq \leftarrow j - 1$; $dir \leftarrow j + 1$

   For i=1 to k

   Se $|a_{esq} - a_j| < |a_{dir} - a_j|$

   Adicione a_{esq} a lista dos elementos mais próximos de a_j.

   $esq \leftarrow esq - 1$

   Senão

   Adicione a_{dir} a lista dos elementos mais próximos de a_j.

   $dir \leftarrow dir + 1$

   Fim Se

   Fim For

6. Seja $S = \{a_1, \ldots, a_n\}$ um conjunto de n números naturais distintos e um inteiro x. Considere o problema $\mathcal{P}$ de determinar se existem três números naturais distintos em S cuja soma é x

a) Seja $T(n)$ a complexidade de pior caso do algoritmo abaixo para resolver $\mathcal{P}$. Encontre $f(n)$ tal que $T(n) = \Theta(f(n))$.

Para i=1,...,n-2
  Para j=i+1,...,n-1
    Para k=j+1,...,n
      Se a_i+a_j+a_k =x
        Return SIM
Return NÃO

b) Projete um algoritmo com complexidade $O(n^2 \log n)$ para resolver o problema $\mathcal{P}$. Não é necessário apresentar o pseudo-código mas sim explicar com clareza os passos que o algoritmo deve realizar e explicar a complexidade.

Solução. (a) Seja $T(n)$ a complexidade do algoritmo. Temos que

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} c,$$

aonde c é uma constante.

Note que

$$\sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} c \leq \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} c = cn^3.$$

Logo $T(n)$ é $O(n^3)$.

Por outro lado,

$$\sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^{n} c \geq \sum_{i=1}^{n/3} \sum_{j=n/3}^{2n/3} \sum_{k=2n/3+1}^{n} c \geq \frac{cn^3}{27}$$

Logo $T(n)$ é $\Omega(n^3)$. Portanto, podemos concluir que $T(n)$ é $\Theta(n^3)$.

Solução (b). Ordene a lista em $O(n \log n)$. Após, para todo par de elementos i, j com $i < j$, procure através de uma busca binária se existe um elemento a_k em A que satisfaz simultaneamente: $k \neq i$, $k \neq j$ e $a_k = x - a_i - a_j$. Se tal elemento existir responda SIM, caso contrário responda NÃO.

7. Considere os pseudo-códigos abaixo.

a) Determine para o pseudo código 1 uma função $f(n)$ tal que $T(n) = \theta(f(n))$.

Pseudo1

t ← 0

Cont ← 1

Para i=1 até n

Cont ← cont+1

Fim Para

Enquanto $cont \geq 1$

Cont ← cont/2

Para $j = 1$ a n

$t++$

Fim Para

Fim Enquanto

b) Determine para o pseudo código 2 uma função $g(n)$ tal que $T(n) = \theta(g(n))$.

Pseudo2

$i \leftarrow 0$

Enquanto $i^2 \leq n$

i ++

$t \leftarrow 0$

Enquanto $t \leq i$

$t++$

Fim Enquanto

Fim Enquanto

Solução de (a). O primeiro loop gasta $\Theta(n)$. O segundo loop (Enquanto) executa $\log n$ vezes e o custo de cada loop é n devido ao loop interno. Portanto, a complexidade do algoritmo é $\Theta(n \log n)$.

Solução de (b). O algoritmo gasta

$$\sum_{i=1}^{\sqrt{n}} c \times i. = \frac{c\sqrt{n}(\sqrt{n}+1)}{2}.$$

Portanto, o algoritmo executa em $\Theta(n)$.

8. Seja A um vetor contendo n números reais.

   a) Explique como seria um algoritmo para devolver o número em A que aparece mais vezes. Em caso de empate, o algoritmo pode devolver qualquer um dos números empatados. Análise a complexidade do algoritmo proposto. Quanto mais eficiente melhor.

   b) Assuma agora que todo número em A pertence ao conjunto $\{n^2, n^2 + 1, ..., n^2 + n\}$. Responda o item anterior tendo em vista esta hipótese.

   Solução de (a). Ordene o vetor em $O(n \log n)$ e depois percorra o vetor acumulando quantas vezes cada elemento aparece.

   Solução de (b). Crie um vetor V de $n + 1$ posições indexado de 0 a n. Incialize V colocando 0 em todas posições. Execute um loop para percorrer a lista A: ao encontrar o elemento a_i, incremente $V[a_i - n^2]$ de uma unidade. Após o loop, seja j a posição de V com valor máximo. O elemento de A que aparece mais é $n^2 + j$. Esta solução tem complexidade $O(n)$.

9. Seja S um conjunto de n numeros reais distintos. Explique como seria um algoritmo eficiente para encontrar os $\sqrt{n}$ menores números do conjunto S e analise sua complexidade. Quanto mais eficiente o algoritmo melhor.

   Solução. Encontre o $\sqrt{n}$-ésimo menor elemento do vetor utilizando seleção em tempo linear. Percorra o vetor imprimindo todo elemento que for menor que $\sqrt{n}$-ésimo menor elemento encontrado.

2 Extras (Mais difíceis)

1. Seja uma matriz quadrada A com n^2 números inteiros que satisfaz as seguintes propriedades:

   (a) $A[i,j] \leq A[i+1,j]$ para $1 \leq i \leq n-1$ e $1 \leq j \leq n$

   (b) $A[i,j] \leq A[i,j+1]$, para $1 \leq i \leq n$ e $1 \leq j \leq n-1$

   Dado um elemento x, descreva um procedimento eficiente para determinar se x pertence a A ou não. Analise a complexidade do algoritmo proposto. Mostre que este problema tem complexidade $\theta(n)$

   Solução Aplique o procedimento abaixo que a cada iteraçao procura o elemento x no canto superior direito da submatriz corrente de A. Se o elemento deste canto for maior que x, a coluna corrente é descartada da busca. Se o elemento for menor que x a linha corrente é descartada da busca. O procedimento tem complexidade linear pois a cada iteração, ou o valor de *lin* ou o de *col* diminui de uma unidade e quando um deles chega a 0, o procedimento para.

   $lin \leftarrow n$; $col \leftarrow n$; $encontrou \leftarrow false$

   Enquanto $linha > 0$ e $col > 0$ e $encontrou = false$

   Compare $a(lin, col)$ com x

   Case $a(lin, col) = x$ do $encontrou \leftarrow true$

   Case $a(lin, col) > x$ do $col --$

   Case $a(lin, col) > x$ do $lin --$

   Fim Enquanto

   Para mostrar que n operações são necessárias, considere um adversário que preenche dinamicamente a matriz A. O adversário depende do algoritmo utilizado e procede da seguinte forma:

   (a) $A[i,j] = 0$, se $i + j < n + 1$; (abaixo da diagonal secundária)

   (b) $A[i,j] = n + 1$ se $i + j > n + 1$ (acima da diagonal secundária)

   (c) Se o algoritmo testar alguma posição da diagonal secundária procedemos da seguinte forma: se todas as demais posições ja tiverem sido preenchida, devolvemos x com probabilidade 1/2 e um número diferente de x e com probabilidade 1/2 devolvemos x. Se alguma posições não tiverem sido preenchida, devolvemos algum número do conjunto $\{1, \ldots, n\} - \{x\}$ que ainda não foi utilizado.

   Devemos notar que sem examinar todos elementos da diagonal principal, o algoritmo não consegue determinar se x pertence a matriz ou não.

2. Mostre como ordenar n inteiros no intervalo $[1, n^2]$ em tempo linear $O(n)$.

   Solução. Represente cada número x da lista dos números que devem ser ordenados como $x = (a, b)$ tal que $0 \leq a \leq n$, $0 \leq b \leq n-1$ e $x = an + b$. Aplique o Radix-sort na lista dos números representados desta forma.

3. Mostre que para fazer o merge de duas listas com n elementos é necesário realizar pelo menos $2n-1$ comparações no pior caso.

4. Mostre que para fazer o merge de duas listas, uma com n elementos e outra com m elementos, é necesário realizar pelo menos comparações $\log\binom{n+m}{n}$ no pior caso.

   Solução. O número de ordens possíveis a partir de duas listas, uma com n elementos e outra com m elementos, é $\binom{n+m}{n}$ já que devemos escolher n posições a partir das $(m+n)$ disponíveis para colocar os elementos da primeira lista. Note que após escolher as posições dos elementos da primeira lista, as posições dos elementos da segunda lista ficam determinadas.

   Portanto, o número de folhas de uma árvore de decisão para encontrar a ordem desejada é pelo menos $\binom{n+m}{n}$ e, como consequência, sua altura é $\Omega(\log\binom{n+m}{n})$.

5. Perdido em uma terra muito distante, você se encontra em frente a um muro de comprimento infinito para os dois lados (esquerda e direita). Em meio a uma escuridão total, você carrega um lampião que lhe possibilita ver apenas a porção do muro que se encontra exatamente à sua frente (o campo de visão que o lampião lhe proporciona equivale exatamente ao tamanho de um passo seu). Existe uma porta no muro que você deseja atravessar. Supondo que a mesma esteja a n passos de sua posição inicial (não se sabe se à direita ou à esquerda), elabore um algoritmo para caminhar ao longo do muro que encontre a porta em $O(n)$ passos. Considere que n é um valor desconhecido (informação pertencente à instância). Considere que a ação composta por dar um passo e verificar a posição do muro correspondente custa $O(1)$

   Sugestão. Analise o número de passos do algoritmo que anda 1 passo para esquerda, depois 2 passos para direita, depois 4 para esquerda, depois 8 para direita e assim por diante até encontrar a porta.

   Solução. O algoritmo funciona em etapas. Na etapa 1 o algoritmo anda para a direita 1 passo depois anda para esquerda 2 passos e, finalmente, anda 1 passo para direita. Na etapa 2 o algoritmo anda para a direita 2 passos depois anda para esquerda 4 passos e, finalmeete, anda 2 passos para direita. Em geral, na etapa i, o algoritmo anda para a direita 2^{i-1} passos depois anda para esquerda 2^i passos e, finalmente, anda 2^{i-1} passos para a direita. Em qualquer uma das etapas a pessoa para se encontrar o muro.

   Temos então que o muro vair ser encontrado na etapa j, onde j é o menor inteiro tal que 2^{j-1} é maior ou igual a n. Em cada etapa i, com $i<j$, o algoritmo anda exatamente 2^{i+1} passos. Na etapa j o algoritmo anda no máximo $3\times 2^{j-1}$ passos (verifique isso).

   Portanto, o número de passos P é no máximo

$$P=\left(\sum_{i=1}^{j-1}2^{i+1}\right)+3\times 2^{j-1}.$$

   Somando a PG obtemos que $P\le 3\times 2^{j-1}+2^{j+1}-2$. Como $n\ge 2^{j-1}$ temos que $P\le 7n-2$.