Lista de Exercícios - Análise de algoritmos

Herberth Amaral

Departamento de Ciência da Computação
Universidade Estadual de Montes Claros
herberthamaral@gmail.com
5 de julho de 2015

Exercícios

1. A complexidade é dada pelo número de arestas: $O(|E|)$

2. Quer dizer que o algoritmo *find* executa um número de operações proporcional a $nlog(n)$, em que n é o tamanho da entrada do algoritmo. De fato, como a função de Ackermann cresce muito rapidamente (mesmo para valores muito pequenos, como 4,3), o seu inverso é usado para denotar funções que crescem muito devagar.

3. A principal diferença entre o QuickSort e o MergeSort é a complexidade no pior caso. No MergeSort é $O(nlogn)$ e no QuickSort é $O(n^2)$, entretanto é raro o QuickSort atingir o pior caso. Segue abaixo uma implementação em Python do QuickSort:

```
def quicksort(array):
    less = []
    equal = []
    greater = []

    if len(array) > 1:
        pivot = array[0]
        for x in array:
            if x < pivot:
                less.append(x)
            if x == pivot:
                equal.append(x)
            if x > pivot:
                greater.append(x)
        return quicksort(less)+equal+quicksort(greater)
    else:
        return array
```

A seguir, uma implementação do MergeSort em Python

```
def mergesort(x):
    if len(x) < 2:
        return x

    result,mid = [],int(len(x)/2)

    y = mergesort(x[:mid])
    z = mergesort(x[mid:])

    while (len(y) > 0) and (len(z) > 0):
            if y[0] > z[0]:
```

```
                result.append(z.pop(0))
            else:
                result.append(y.pop(0))

    result.extend(y+z)
    return result
```

4. Os dois algoritmos tem a mesma complexidade de tempo: $O(|V| + |E|)$, porém o algoritmo de Tarjan possui uma iteração a menos e, por isso, na prática, tende a ser mais rápido.

5. Os dois algoritmos podem apresentar complexidade $O(|E|log|V|)$. Essa complexidade pode variar de acordo com o uso de algoritmos de ordenação e estruturas de dados utilizadas para representar o grafo.

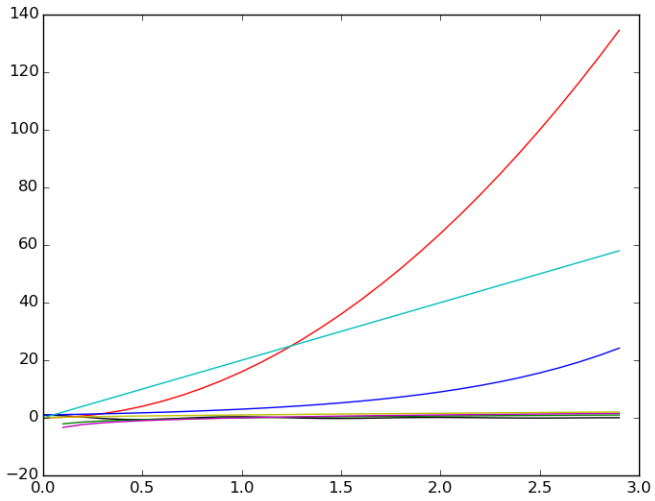
6. Sim, há o algoritmo de Thorup ([1]) que possui complexidade linear, ou $O(n)$. O algoritmo de Dijkstra possui complexidade proporcional a $O(|E| + |V|log|V|)$.

7. A complexidade dos algoritmos do heap binário são:

   (a) Busca: $O(n)$

   (b) Inserção: $O(logn)$

   (c) Exclusão: $O(logn)$

8. 

9. A soma de vetores em tempo $O(logn)$ não é possível, pois é necessário fazer uma soma par-a-par de todos os elementos do subvetor. Portanto, esse tipo de operação é $\Omega(n)$. Entretanto, a primeira função encontra-se abaixo:

```
A = []
def soma(i, y):
    return A[i] += y
```

10. (a) Realiza a soma dos a primeiros números naturais;
    (b) Como a função está escrita no formato *tail-recursive*, é possível reescreve-la para um formato iterativo de forma a facilitar os cálculos:

```
int X(int a)
{
    if (a<=0) // 1 vez
        return 0; // nao sera executado no pior caso
    int tmp = 0; // 1 vez
    for (i = 1; i<a;i++) // a vezes
    {
        tmp += i; // a -1 vezes
    }
    return tmp; // 1 vez
}
```

    Desta forma é fácil verificar que X é da ordem de $2n + 2$.
    (c) Feito no item anterior;
    (d) As duas funções são equivalentes em termos de complexidade assintótica. No entanto, na prática, caso o compilador não tenha otimização de *tail-call*, a implementação iterativa é mais eficiente.

11. A partir de $n = 3$.

12. (a) Afirmativa verdadeira. 2^{n+1} domina assintoticamente 2^n;
    (b) Afirmativa falsa. 2^{2n} domina assintoticamente 2^n, pois não há uma constante que multiplicada por 2^n seja $>= 2^{n+1}$;
    (c) Afirmativa falsa. Contra-exemplo: $f(n) = O(n), g(n) = O(1) \rightarrow f(n) + g(n) = O(n)$.

13. $O(n * i * j)$

14. A pizza ganha dois novos pedaços a cada corte. Portanto $P(n) = 2n$.

15. Prova por indução. Assumindo que o algoritmo analisado considera potenciação como uma série de multiplicações e que o algoritmo considere uma mútiplicação como uma série de somas, $n = 2 \rightarrow \Sigma_{i=0}^{2} 3^i = 3.3 = (3 + 3 + 3) = \Sigma_{j=0}^{3^1} 3$. Para $n = 3 \rightarrow \Sigma_{i=0}^{3} 3^i = 3.3.3 = (3 + 3 + 3 + 3 + 3 + 3 + 3 + 3 + 3) = \Sigma_{j=0}^{3^2} 3$. Portanto, para $n = x \rightarrow \Sigma_{i=0}^{x} 3^i = (3.3.\cdots) = (3 + 3 + 3 + \cdots) = \Sigma_{j=0}^{3^{x-1}} 3$. Como $3^{x-1})$ é dominado assintoticamente por 3^x, podemos concluir que $\Sigma_{i=0}^{n} 3^i$ é $O(3^n)$.

16. 16

17. O limite inferior para ordenação por comparação é $O(nlogn)$.

18. Segue tabela preenchida abaixo

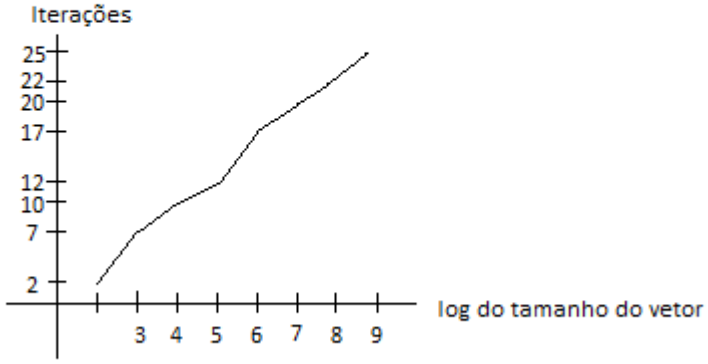
Expressão	Termo(s) dominantes	O(...)
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50nlog_{10}n$	$100n^{1.5}$	$O(n^{1.5})$
$0.3n + 5n^{1.5} + 2.5n^{1.75}$	$2.5n^{1.75}$	$O(n^{1.75})$
$n^2log_2n + n(log_2n)^2$	n^2log_2n	$O(n^2logn)$
$nlog_3n + nlog_2n$	$nlog_3n$	$O(nlogn)$
$3log_8n + log_2log_2log_2n$	$3log_8n$	$O(logn)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$
$0.01nlog2n + n(log_2n)^2$	$n(log_2n)$	$O(nlogn)$
$100nlog3n + n^3 + 100n$	n^3	$O(n^3)$
$0.003log_4n + log2log2n$	$0.003log_4n$	$O(nlogn)$

19. Esse algoritmo de busca é a busca binária. Como todos os algoritmos que usam. Segue abaixo um pseudocodigo anotado com as funções de custo:

```
1:  function BuscaBinaria(Vetor, valor)
2:      pivo ← Length(Vetor)/2                          ▷ 1 vez
3:      fim ← Length(Vetor)                             ▷ 1 vez
4:      inicio ← 0                                      ▷ 1 vez
5:      while pivo ≠ fimandpivo ≠ inicio do             ▷ log2Length(Vetor) vezes
6:          if Vetor[pivo] == valor then                ▷ log2Length(Vetor) − 1 vezes
7:              return pivo                             ▷ 1 vez
8:          end if
9:          if Vetor[pivo] < valor then                 ▷ log2Length(Vetor) − 1 vezes
10:             inicio ← pivo
11:         else
12:             fim ← pivo/2
13:         end if
14:         pivo ← (inicio + fim)/2                     ▷ log2Length(Vetor) − 1 vezes
15:     end while
16:     return −1                                       ▷ 1 vez
17: end function
```

Como visto acima, o custo da função é $5log(N) + 2$, em que N é o tamanho do vetor. A linha 12 não foi contada porque o custo de processa-la implica no não-custo da linha 10, então é seguro dizer que a linha 10 e 12 juntas tem custo $log_2Length(Vetor) - 1$. Desta forma, pode-se concluir que a complexidade da busca binária é $O(logN)$.

20. Esse teste é deveras difícil de ser executado em ambiente real de forma que a diferença de tempo de execução em função do tamanho da entrada seja perceptível. Se estivéssemos procurando um átomo específico no Universo utilizando busca binária, encontraríamos com apenas 273 comparações, no máximo ($log_2 10^{82}$). Sem contar na *impossibilidade* de notar diferenças significativas. Uma máquina com memória suficiente para guardar 10^{82} bytes (2 KY, ou Kilo-Yotta) está prevista para ser lançada em 63 anos se mantivermos o ritmo de dobrar a capacidade de armazenamento a cada 18 meses no mínimo. No entanto, para efeitos de verificação do algoritmo, podemos medir a quantidade de vezes que o loop foi executado. Segue gráfico abaixo.



O gráfico está em escala exponencial em X, portanto ele dá a impressão que cresce linearmente. Porém, na realidade, o crescimento é logarítmico.

21. Esse algoritmo pode ser descrito utilizando dois algoritmos: um de iteração ($\Theta(n)$) e outro de busca ($\Theta(logn)$). Segue abaixo uma representação em pseudocódigo do algoritmo:

```
1: function ExisteSoma(S, x)
2:     for all i ← S do                          ▷ i é o índice, não o elemento
3:         indiceComplemento ← BuscaBinaria(S, S[i] − x)
4:         if indiceComplemento ≠ i then
5:             return true
6:         end if
7:     end for
8:     return false
9: end function
```

22. (a) Falso. Contra-exemplo: $O(n)$ é $O(n^2)$, mas $O(n^2)$ não é $O(n)$;
    (b) Falso. Contra-exemplo: $\Theta(n) + \Theta(n^2) = \Theta(n^2)$;
    (c) Verdadeiro. $g(n)$ domina assintoticamente $f(n)$, então $2^{g(n)}$ também domina $2^{f(n)}$;
    (d) Falso. Considere $f(n) = n$ e $g(n) = n^2$. Não se pode afirmar que $g(n) = \Omega(n)$, pois $g(n)$ pode ser $\Omega(n^2)$.

Referências

[1] M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time *JACM* **46** (1999).