Projeto e Análise de Algoritmos
1º Trabalho Prático

Tiago Alves Macambira

2003.1

1 Avaliar as Somas

1.1 $\sum_{i=1}^{n} a^i$

Isso é simplesmente o somatório dos n primeiros elementos elementos de uma progressão geométrica, com $a_1 = q = a$. A dedução é trivial e encontra-se em qualquer livro de 2º grau. De qualquer forma, vamos a ela:

$S_n = a^1 + a^2 \ldots + a^n$

$S_n = a + aa \ldots + aa^{n-1}$

Introduzindo a^{n+1} em ambos os lados

$S_n + a^{n+1} = a + aa^1 + \ldots + aa^{n-1} + a^{n+1} = a + aa^1 \ldots + aa^{n-1} + aa^n$

$S_n + aa^n = a + aa \ldots + aa^{n-1} + aa^n$

$S_n + aa^n = a + a\left(a + aa \ldots + aa^{n-1}\right)$

$S_n + aa^n = a + aS_n$

$S_n = \frac{a(a^n-1)}{a-1}$

Assim $\sum_{i=1}^{n} a^i = \frac{a(a^n-1)}{a-1}$para $a \neq 1$.

Para $a = 1$, $\sum_{i=1}^{n} a^i = 1 + \ldots + 1 = n$

1.2 $\sum_{i=1}^{n} \frac{1}{i}$

Usando aproximação por integrais[1, p50]

A margem inferior:

$$\sum_{k=m}^{n} \frac{1}{k} \geq \int_{m}^{n+1} \frac{dx}{x} = \ln(n+1)$$

Para margem superior, derivamos a inequação:

$$\sum_{k=2}^{n} \frac{1}{k} \leq \int_{1}^{n+1} \frac{dx}{x} = \ln n$$

O que dá a margem:

$$\sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1$$

1.3 $\sum_{i=1}^{n} \log i$

$$\begin{aligned} \sum_{i=1}^{n} \log i &= \log 1 + \log 2 + \ldots + \log n \\ &= \log (1 \cdot 2 \cdot \ldots \cdot n) \\ &= \log n! \end{aligned}$$

Usando a "Aproximação de Stirling"[1, 3], chegamos em

$$\begin{aligned} \sum_{i=1}^{n} \log i &= \log n! \\ &= \Theta (n \log) \end{aligned}$$

2 Apresente a compliexidade de tempo para os procedimentos abaixo:

2.1 Pesquisa(n)

Caso Base:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 1 + (n-1) + T\left(\frac{3}{5}n\right) \\ &= n + T\left(\frac{3}{5}n\right) \end{aligned}$$

Aplicando o *Master Theorem*[1, p62], tendo que $a = 1$, $b = \frac{5}{3}$ e $f(n) = n$, chegamos ao resultado que

$$T(n) = \Theta(f(n)) = \Theta(n)$$

de qualquer forma, pelo método da substituição:

$$\begin{aligned} T(n) &= n + T\left(\frac{3}{5}n\right) && (1) \\ T\left(\frac{3}{5}n\right) &= \frac{3}{5}n + T\left(\frac{3^2}{5^2}n\right) && (2) \\ T\left(\frac{3^2}{5^2}n\right) &= \frac{3}{5}n + T\left(\frac{3^3}{5^3}n\right) && (3) \\ &\vdots && (4) \\ T\left(\left(\frac{3}{5}\right)^i n\right) &= \left(\frac{3}{5}\right)^i n + T\left(\left(\frac{3}{5}\right)^{i+1} n\right) && (5) \end{aligned}$$

$$
\begin{aligned}
& \vdots && (6) \\
T\left(\left(\frac{3}{5}\right)^k n\right) &= 1 && (7)
\end{aligned}
$$

Ora, chegaremos ao fim da recursão quando

$$
\begin{aligned}
\left(\frac{3}{5}\right)^k n &= 1 && (8) \\
k \log_{\frac{3}{5}} n &= 1 && (9) \\
k &= \log_{\frac{5}{3}} n && (10)
\end{aligned}
$$

Logo,

$$
\begin{aligned}
T(n) &= \left(\frac{3}{5}\right)^0 n + \left(\frac{3}{5}\right)^1 n + \ldots + \left(\frac{3}{5}\right)^k n && (11) \\
&= \sum_{i=0}^{k} \left(\frac{3}{5}\right)^i n && (12) \\
&= \sum_{i=0}^{\log_{\frac{3}{5}} n} \left(\frac{3}{5}\right)^i n && (13) \\
&= n \frac{1 - \frac{3}{5}^{1+\log_{\frac{5}{3}} n}}{1 - \frac{3}{5}} && (14)
\end{aligned}
$$

2.2 Fib

Vamos inicialmente ver a condição inicial dessa recursão

$$
\begin{aligned}
T(0) &= 1 \\
T(1) &= 1
\end{aligned}
$$

Para n qualquer, a expressão de tempo desse algoritmo fica

$$T(n) = T(n-1) + T(n-2)$$

Ora, a expressão acima é a função de recorrência que devolve o n-ésimo numero de Fibonacci. Logo, a complexidade de tempo desse algoritmo será, para uma entrada k, o valor do k-ésimo número de Fibonacci, cujo valor é dado pela fórmula:

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - \hat{\phi}^n\right)$$

Ou, arredondando para o inteiro mais próximo:

$$
\begin{aligned}
F_n &= \left\lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor \\
&= \frac{\phi^n}{\sqrt{5}}
\end{aligned}
$$

Onde $\phi = \frac{(1+\sqrt{5})}{2}$

A demonstração completa da obtenção de F_n pode ser obtida em[3, p.212-219]

Finalmente, para fechar esse item, seria bom resaltar que $T(n) = O(2^n)$

2.3 Sort1

O algoritmo é o merge-sort, que é $O(n \log n)$. De qualquer forma, vamos à análise desse algoritmo.

2.3.1 Merge

Antes de prosseguirmos, precisamos saber a complexidade de tempo do procedimento MERGE. É sabido que dá para se construir sem muito esforço um algoritmo que fazer intercalação de dois vetores de tamanho iguais, cada um deles ordenados, em $O(n)$. Para provar, propomos um algoritmo que faça isso e calcularemos a sua complexidade:

Algorithm 1 Merge de dois vetores

Require: $vetorAUX$
- $pos_i \leftarrow i$
- $pos_j \leftarrow m + 1$
- $k \leftarrow i$
- **while** $k \leq j$ **do**
  - **while** $(A[pos_i] \leq A[pos_j]) and (pos_i \leq m)$ **do**
    - $Aux[k] \leftarrow A[pos_i]$
    - $k \leftarrow k + 1$
    - $pos_i \leftarrow pos_i + 1$
  - **end while**
  - **while** $k \leq j$ **do**
    - $Aux[k] \leftarrow A[pos_j]$
    - $k \leftarrow k + 1$
    - $pos_j \leftarrow pos_j + 1$
  - **end while**
- **end while**
- **for** $x \leftarrow i..j$ **do**
  - $A[x] \leftarrow AUX[x]$
- **end for**

No algoritmo 1, acima o laco *while* mais externo garante que não haja sobreposição dos dois vetores que estão sendo. A cada iteração desse laço, os dois laços internos vão linearmente percorrendo e inserindo, ordenadamente, os elementos contidos nas respectivas partições de A que cada laço trata:

$$3(m-i+1)+3[j-(m+1)+1]+(j-i+1)=4(j-i+1)=O(n)$$

2.3.2 Sort1

Uma vez sabendo a complexidade de tempo do procedimento MERGE, sigamos para o cálculo da complexidade de tempo do algoritmo Sort1 como um todo.

$$T(n)=2T(n/2)+O(n)$$

Aplicando o *Master Theorem*[1, p62], tendo que $a=2$, $b=2$ e $f(n)=O(n)$, chegamos ao resultado que

$$f(n)=\Theta\left(n^{\log_b a}\right)\Rightarrow T(n)=\Theta\left(n^{\log_b a}\log n\right)=\Theta\left(n^{\log_2 2}\log n\right)=\Theta(n\log n)$$

Já temos uma idéia do que procurar. Desenvolvamos a equação recursiva por substituições:

$$\begin{aligned}
T(n) &= 2T(\frac{n}{2})+O(n)\\
&= 4T\left(\frac{n}{4}\right)+20(n)\\
&\vdots\\
T(n) &= 2^iT\left(\frac{n}{2^i}\right)+iO(n)
\end{aligned}$$

Fazendo $i=\lg n$

$$\begin{aligned}
T(n) &= n+O(n)\lg n\\
&= o(n\lg n)
\end{aligned}$$

2.4 Sort2

Esse algoritmo é muito parecido com o anterior.

A sua fórmula de recorrência é:

$$T(n)=\begin{cases} 1, & \\ 3T\left(\frac{n}{3}\right)+\frac{5n}{3}-2 & para\ n=1\end{cases}$$

Aplicando o *Master Theorem*[1, p62], tendo que $a=3$, $b=3$ e $f(n)=\frac{5n}{3}-2$, chegamos ao resultado que

$$f(n)=\Theta\left(n^{\log_b a}\right)\Rightarrow T(n)=\Theta\left(n^{\log_b a}\log n\right)=\Theta\left(n^{\log_3 3}\log n\right)=\Theta(n\log n)$$

3 Limite Inferior

Ordenações através de comparações podem ser vistas abstratamente em termos de *árvores de decisão*.Uma arvore de decisão representa as comparações realizadas pelo algoritmo de ordenação quando ele trabalha em uma entrada de um dado tamanho.

O tamanho do caminho mais longo partindo da raíz de uma árvore de decisão para qualquer um de suas folhas representa o pior-caso no número de comparações que um algoritmo de ordenação realiza. Conseqüentemente, o pior-caso no número de comprarações para uma ordenação por comparação corresponde à altura da sua árvore de decisão.

Assim, achar o limite inferior para a altura de uma árvore de decisão equivale a achar o limite inferior para o tempo de execução de qualquer algoritmo de ordenação por comparação.

Considere a arvore de decisão de altua h que ordena n elementos. Como existem $n!$ permutações dos n elementos, cada permutação representa uma ordem de ordenação distinta, a árvore tem que ter pelo menos $n!$ folhas. Já que uma árvore binária de altura h tem não maus do que 2^h folhas, temos:

$$n! \leq 2^h$$

que, através do uso de logaritmos, fica

$$h \geq \lg(n!)$$

, já que a função é monotonicamente crescente. Usand as aproximação de Stirling, temos:

$$n! \geq \left(\frac{n}{e}\right)^n$$

onde e é a base dos logaritmos naturais; assim:

$$\begin{aligned} h &\geq \lg\left(\frac{n}{e}\right)^n \\ &= n\lg n - n\lg e \\ &= \Omega(n\lg) \end{aligned}$$

-

[2, 1]

4 Algoritmos de ordenação parcial: estudos comparativos

Em anexo, no documento *estatisticas/tempo.txt*, há um lista com os médios tempos de execução de vários algoritmos, para vários valores de n e k.

Também em anexo, gráficos relativos a essas estatisticas em *graficos/time_graph_ *.ps.*

4.1 Experimentalmente, o número de comparações e movimentos-de-registro

Os algoritmos originados de algoritmos $O\left(n^2\right)$ perdem muito tempo fazendo comparações. O partital_Selection é destes aquele que mais realiza comparaçõe, seguido do insertion (também originado de um algoritmo $O(n2)$. Heap e quick sort realizam um número bem parecido de movimentações de regi:stros.

Em relação a comparação de registros, partial_insertion sort é o que mais se destaca quando do crescimento do n. Todos os outros ficam relativamente estáveis.

4.2 Tempo de Execução

4.2.1 Interpretação

É interessante notar que os algoritmos descendentes de algoritmos $O\left(n^2\right)$, a exceção de n muito pequenos, a medida que k aumenta, sua performance diminui drasticamente. A perforamance desses algoritmos, contudo, ainda dá margens para interessantes surpresas: o partial_selection é disparadoo melhor algoritmo para $k = 1$. Mas também esse é o unico caso onde ele se dá bem. O partial_insertion_sort também apresenta uma acelerada taxa de crescimento com um aumento de k. Ainda assim é, para valores de k proximos de $\sqrt{n}$ uma boa opção, rivalizando nas faixas iniciais de k até com o quicksort.

Enrtre os algoritmos cujos originais eram $O(n \log n)$ o quicksort é sempre a melhor opção. O partial_heapsort apresente, em alguns momentos, um custo inicial de montar a heap alto demais, em comparação com os algoritmos cujos originais são $O(n^2)$. Não pouca vezes perdeu para o insetion_sort com valores pequenos de k.

4.3 Resultados Analíticos

4.3.1 Partial insertion sort

o que difere este algoritmo do original é que ele

1. Realiza um insertion sort normal para os k primeiros elementos ($O(k^2)$)
2. Para cada elemento fora desse intervalo, ele compara o valor destes com o do ultimo (e maior) elemento contudo no intervalo e, se for o caso, troca esses elementos de lugar, chamando novamente insertion sort nos k primeiros elementos para ordená-los ($O(n-k)k^2$)

A complexidade desse algoritmo fica então:

$$T(n) = O(nk^2 + k^2 - k^3)$$

Logo, esse algoritmo, com L pequenos é bom, pois o fator nk^2 fica minimizado. Mas, aumentos em k acarreta um crescimento muito grande na parcela nk^2

4.3.2 Partial Selection Sort

O que difere esse do original é que o laço externo é limitado a rodar k vezes. Complexidade:

$$\begin{aligned} T(n) &= \sum_{i=0}^{k-1} n - i \\ &= k\frac{n+n-k-1}{2} \\ &= \frac{2nk - k^2 - k}{2} \\ &= O\left(nk + k^2\right) \end{aligned}$$

Pela fórmula analítica fica fácil de ver pq o selection decresce tanto em relação ao insertion: apeser de ser inicialmente menor do que o insertions, k tem um grande peso nesse algoritmo.

4.3.3 Partial Heapsort

Aqui nada de estranho: inicialmente cria-se um heap_invertido. Faz-se k vezes o o passo de remoção do raiz da heap para o final da fila. Ao final da ordenação dos k elementos , estes, que estavam em ordem reversa ao no final do vertor, são copiado de volta.

- O custo para montar o heap é o mesmo: $O(n)$
- O custo apra retirar os k primeiros: $k \cdot$ (logm)
- O custo para re-ordenar os k elementos no incio do vertor: $O(k)$

Total: $k \log n + k{=}O(k \log n)$

Um k pequeno ajuda mas o $\log n$ estará sempre presente.

4.3.4 Partial Quick

Melhor algoritmo para k e n grandes - ele simplesmente se dá bem de qualquer forma.

A complexidade dele, no pior caso é iqual á do quicksort normal: $O\left(n^2\right)$. Esse pior caso poderia ser facilmente evitado se fosse utilizado antes um algoritmo $O(n)$ pra selecionar o k-esimo elemento do vetor e, com ele em mãos, fazer o particiomento de forma "perfeita". Contudo, a nossa implementação no caso médio é boa o suficiente para atender as nossas espectativas.

Referências

[1] "Introduction to Altorithms", Cormen et al, 1st Edition

[2] "Algorithms, Data Structures, and Problem Solving With C++" , Mark Allen Weiss.

[3] "Matemática Concreta - Fundantos para a Ciência da Computação", Graham, Knuth, Patashnik; 2ª edição; LTC Editora