Análise de Algoritmos: Melhor caso, pior caso, caso médio

Fernando Lobo

Algoritmos e Estrutura de Dados

Sumário

- Rever um problema e um algoritmo que já devem conhecer.
- Descrevê-lo em pseudocódigo (como no livro).
- Começar a usar a notação assimptótica para descrever o tempo de execução de algoritmos.

O problema de ordenação

- Input: Uma sequência de n números $\langle a_1, a_2, \ldots, a_n \rangle$
- Output: Uma permutação $\langle a'_1, a'_2, \ldots, a'_n \rangle$ da sequência de input tal que:

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

- Sequência é tipicamente guardada num array.
- Exemplo:
  - Input: $\langle 8, 2, 4, 9, 3, 6 \rangle$
  - Output: $\langle 2, 3, 4, 6, 8, 9 \rangle$

Insertion Sort

- Bom para ordenar um número pequeno de elementos.
- Funciona de modo análogo ao modo como ordenamos uma mão de cartas:
  - Mantemos as cartas na mão sempre ordenadas.
  - Ao recolhermos mais uma carta, inserimo-la na posição correcta (para que as cartas na mão continuem ordenadas).

Pseudocódigo
(convenção: arrays começam na posição 1)

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1..j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```


Exemplo

```
8 2 4 9 3 6    // input
2 8 4 9 3 6    // fim da 1ª iteração
2 4 8 9 3 6    // fim da 2ª iteração
2 4 8 9 3 6    // fim da 3ª iteração
2 3 4 8 9 6    // fim da 4ª iteração
2 3 4 6 8 9    // fim da 5ª iteração
```

Correção do algoritmo

- Invariante: No começo de cada iteração do ciclo **for**, o subarray $A[1\,..\,j-1]$ está ordenado.
- Usamos o invariante para provar que o algoritmo funciona de modo correcto.

Correção do algoritmo (cont.)

Temos de provar 3 coisas:

1. Inicialização: Invariante é verdadeiro antes da 1ª iteração.
2. Manutenção: Se é verdadeiro no início de uma iteração, mantém-se verdadeiro no início da iteração seguinte.
3. Término: Quando o ciclo termina, o array $A[1\,..\,n]$ está ordenado.

Invariantes

A utilização de invariantes em ciclos é análogo ao método de indução matemática:

- Inicialização → base de indução.
- Manutenção → passo indutivo.
- Término: → não existe no método de indução matemática (é infinito). Aqui a "indução" pára quando o ciclo termina.

Como analisar o tempo de execução do algoritmo?

O tempo depende do input:

- Ordenar 10000 números demora mais do que ordenar 30.
- Pode demorar tempos diferentes em inputs do mesmo tamanho (ex: INSERTION-SORT é mais rápido se o input já estiver ordenado.)

Como analisar o tempo de execução do algoritmo?

O tamanho do input depende do problema:

- Normalmente é um só número (n no caso de ordenação).
- Mas pode ser mais:
  - Em algoritmos de grafos, o tamanho é geralmente expresso em termos de 2 quantidades: nº de nós e nº de arcos (veremos isso mais tarde).

Tempo de execução

O tempo de execução num determinado input, é o número de operações primitivas executadas.

- Operações primitivas: afectações, comparações, etc.
- Cada linha do pseudocódigo requer tempo constante (independente do tamanho do input).
- Linhas diferentes podem requerer tempos diferentes.
- Chamadas de funções também têm tempo constante, mas a execução da função poderá não ter.

Análise de Insertion-Sort

- Para $j = 2, 3, \ldots, n$, seja t_j o nº de vezes que o ciclo **while** é executado para esse valor de j.
- (NOTA: os ciclos **for** e **while** são testados mais uma vez que o corpo do ciclo.)
- Tempo de execução depende dos t_j's. Isto é, variam consoante o input.

Pseudocódigo (outra vez)

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2       key = A[j]
3       // Insert A[j] into the sorted sequence A[1..j − 1].
4       i = j − 1
5       while i > 0 and A[i] > key
6            A[i + 1] = A[i]
7            i = i − 1
8       A[i + 1] = key
```

Número de vezes que cada linha é executada

Seja n o número de elementos do array A (i.e., $n == A.length$).

- linha 1: n
- linha 2: $n - 1$
- linha 3: $n - 1$
- linha 4: $n - 1$
- linha 5: $t_2 + t_3 + \ldots + t_n$
- linha 6: $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$
- linha 7: $(t_2 - 1) + (t_3 - 1) + \ldots + (t_n - 1)$
- linha 8: $n - 1$

Tempo de execução do algoritmo

- $\sum_i$ (custo da linha i) $\times$ (nº de vezes que a linha i é exec.)
- Seja c_i o custo da linha i.
- Seja $T(n)$ o tempo de execução de INSERTION-SORT.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Melhor caso

- O array já está ordenado. Isto é,
- $A[i] \leq key$ no teste do ciclo **while** $\implies$ todos os $t_{j's} = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- $T(n) = an + b$, em que a e b são constantes.
- $\implies$ $T(n)$ é uma função linear de n.

Pior caso

- O array está ordenado por ordem inversa.
- $A[i] > key$ em todas as iterações do ciclo **while**.
- Temos de comparar *key* com todos os elementos à esquerda da posição j $\implies$ $j - 1$ comparações.
- Como o ciclo **while** termina quando $i = 0$, há um teste adicional depois das $j - 1$ comparações $\implies$ $t_j = j$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j$$

$$\sum_{j=2}^{n} (t_j - 1) = \sum_{j=2}^{n} (j - 1)$$

(estes somatórios são somas de termos de progressões aritméticas)

Pior caso (cont.)

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\frac{(2+n)(n-1)}{2} \\
&\quad + c_6\frac{n(n-1)}{2} + c_7\frac{n(n-1)}{2} + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 \\
&\quad + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\
&\quad - \left(c_2 + c_4 + c_5 + c_8\right)
\end{aligned}
$$

- $T(n) = an^2 + bn + c$, em que a, b e c são constantes.
- $\implies$ $T(n)$ é uma função quadrática de n.

Caso médio

- No geral é difícil (ou mesmo impossível) de calcular.
- Temos de assumir certas coisas. Por exemplo, todas as sequências de input são igualmente prováveis (o que nem sempre é verdade).

Caso médio (cont.)

- Imaginemos que o input para INSERTION-SORT é um array de n números aleatórios gerados a partir de uma distribuição uniforme.
- Em média, *key* em $A[j]$ é menor que metade dos elementos em $A[1..j-1]$ e é maior que a outra metade.
- $\Longrightarrow$ em média o ciclo **while** tem de olhar até metade do subarray $A[1..j-1]$ $\Longrightarrow$ $t_j = j/2$
- O tempo de execução é aprox metade do tempo do pior caso. Mas continua a ser uma função quadrática de n.

Casos: Melhor, pior e médio

- Análise do melhor caso normalmente não nos interessa.
- Porquê? Porque é fácil fazer batota.
- $\Longrightarrow$ Podemos ter um algoritmo de ordenação muito mau que é linear no melhor caso e supra-exponencial no pior caso (e também no caso médio).
- Que algoritmo é esse?

Casos: Melhor, pior e médio

- Normalmente estamos interessados na análise do pior caso: o tempo de execução mais longo de qualquer input de tamanho n.
- Porquê?
  - Dá-nos um limite superior do tempo de execução, qualquer que seja o input.
  - O caso médio costuma ser quase tão mau como o pior caso e é normalmente mais difícil de analisar.
  - Nalguns algoritmos, o pior caso ocorre com muita frequência (ex: pesquisa de um elemento que não está presente na coleção.)

Ordem de crescimento

- Simplificamos para nos concentrar no essencial.
  - eliminamos termos de ordem inferior.
  - ignoramos o coeficiente do termo de ordem superior.
- Exemplo: O tempo de execução de INSERTION-SORT no pior caso é dado por $an^2 + bn + c$
  - eliminando termos de ordem inferior $\implies an^2$
  - ignorando o coeficiente $\implies n^2$
- IMPORTANTE: Mas não podemos dizer que $T(n) = n^2$
- $\implies$ o que podemos dizer é que $T(n)$ "cresce" de modo proporcional a n^2.

Ordem de crescimento (cont.)

- Dizemos que $T(n) = \Theta(n^2)$ para capturar a noção de que a ordem de crescimento é proporcional a n^2.
- Chama-se a isto análise assimptótica.
- Normalmente dizemos que um algoritmo é mais eficiente que outro, se o seu tempo de execução no pior caso tiver uma menor ordem de crescimento (i.e. menor complexidade assimptótica).

Nota final

- Esta análise detalhada foi meramente ilustrativa.
- Não é algo que façamos a toda a hora...
- Olhando para o pseudocódigo vê-se logo que a ordem de crescimento é quadrática. Porquê?