Complexidade de Algoritmos

Paulino Ng

2020-05-29

Plano da aula

Esta aula apresenta alguns algoritmos de ordenação e uma análise de sua complexidade.

1. Problema de ordenação.
2. Ordenação por seleção
3. Ordenação por inserção
4. *Mergesort*, ordenação por intercalação
5. *Quicksort*

Problema da Ordenação

- **Entrada**: sequência (vetor ou lista) de n dados $\{a_0, a_1, \ldots, a_{n-1}\}$
- **Saída**: a sequência dos n dados ordenados $\{a'_0, a'_1, \ldots, a'_{n-1}\}$ tal que $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$

Ordenação por seleção

```
Procedimento ordena_selecao(A,n)

Entradas:
  A: um vetor.
  n: número de elementos de A a serem ordenados.
Saída:
  Os elementos de A ordenados em ordem não decrescente

1. Para i = 0 até n-2:
   A. Faça menor ser o índice do menor elemento do
         subvetor A[i..n-1]
   B. Troque A[i] com A[menor]
```

Complexidade: $\Theta(n^2)$*, por que?*

Análise do ordena_selecao()

- temos dois laços, um dentro do outro:
  - o mais interno procura o índice do menor valor do subvetor vec[i..n-1] $\rightarrow \Theta(n)$
  - o externo, depois de encontrado o menor valor do subvetor, troca o valor de vec[i] com o valor de vec[menor] para cada i do primeiro ao penúltimo $\rightarrow (n-1).\Theta(n) = \Theta(n^2)$
- Logo, a complexidade do ordena_selecao() é $Theta(n^2)$

Código Python para implementar este algoritmo

```
def select_sort(vec,n):
  for i in range(0,n-1):
    menor = i
    for j in range(i+1,n):
      if vec[j] < vec[menor]: menor = j
    vec[i], vec[menor] = vec[menor], vec[i]
```

Ordenação por inserção

```
Procedimento ordena_insercao(A,n)

Entradas e Saída como no ordena_selecao()

1. Para i=1 até n-1:
   A. Faça chave = A[i] e j = i - 1
   B. Enquanto j >= 0 e A[j] > chave faça:
      i.   A[j+1] = A[j]
      ii.  j = j - 1
   C. A[j+1] = chave
```

Análise do insert_sort()

- A ideia básica do funcionamento do insert_sort() é semelhante ao que faz um jogador de cartas que recebe as cartas uma a uma e vai colocando elas na ordem. As cartas já recebidas estão em ordem na mão do jogador, a carta nova é inserida na sua posição deslocando todas as cartas maiores para a direita de uma posição.
- O algoritmo, de novo, tem dois laços, um dentro do outro;
  - o laço externo percorre o vetor do segundo elemento para o último, isto é, executa as instruções A, B e C $n - 1$ vezes;
  - o laço interno desloca para a direita os valores maiores do que A[i] para a direita para abrir espaço para colocar o A[i] na sua posição correta (instrução C)

cont. da Análise

- Se a chave (valor de A[i]) for maior do que o valor mais a direita dos elementos já ordenados em ordem crescente, então, não há a necessidade de deslocar nenhum elemento já ordenado. Isto quer dizer que se o vetor já estiver ordenado, o laço interno não precisa executar as suas instruções e a complexidade do algoritmo é $\mathcal{O}(n)$ (melhor caso).
- Se o vetor estiver ordenado na ordem decrescente, o laço interno tem de deslocar todos os elementos já ordenados. Neste caso, a complexidade do algoritmo é $\mathcal{O}(n^2)$ (pior caso).
- No caso médio, a complexidade é $\mathcal{O}(n^2)$, mas existem muitas aplicações em que os dados já estão quase ordenados e, nestes casos, o algoritmo da ordenação por inserção é muito bom.

Código Python para implementar este algoritmo

```
def insert_sort(vec,n):
  for i in range(1,n):
    chave = vec[i]
    j = i - 1
    while j >= 0 and A[j] > chave:
      vec[j+1] = vec[j]
      j = j - 1
    vec[j+1] = chave
```

Mergesort

- A ideia de base do algoritmo *mergesort* é juntar (intercalar) 2 vetores ordenados. Dado um vetor com n elementos, para simplificar, vamos supor que n é uma potência de 2. Vamos dividir o vetor em duas metades, ordenar as duas metades e juntá-las.
- Como ordenamos as duas metades? Da mesma maneira que o vetor inicial, a não ser que o vetor seja unitário (com um único elemento). Observe o processo de recursão, a solução base (quando o vetor só tem um elemento, ele já está ordenado).

```
merge_sort(vec, ini, fim)
  if fim - ini <=1 return
  meio = (ini + fim + 1)/2
  merge_sort(vec, ini, meio - 1)
  merge_sort(vec, meio, fim)
  merge(vec, ini, fim)
```

```
merge(vec, ini, meio, fim)
  esquerda = vec[ini .. meio]
  direita  = vec[meio + 1 .. r]
  i, j, k = 0, 0 , ini
  while i <= meio - ini and j <= fim = meio - 1
    if esquerda[i] < direita[j]
      vec[k] = esquerda[i]
      i += 1
    else
      vec[k] = direita[j]
      j += 1
    k += 1
  while i <= meio - ini
    A[k] = esquerda[i]
    i += 1
    k += 1
  while j <= fim - meio - 1
    A[k] = direita[j]
    j += 1
    k += 1
```

Observações:

- O `merge_sort()` é recursivo. A cada vez o tamanho do problema é reduzido pela metade.
- A operação de intercalar (`merge()`) 2 vetores ordenados é $\Theta(n)$
- Para simplificar a operação de intercalagem, uma cópia do vetor (das partes do vetor) é feita. Isto faz com que o *mergesort* ocupe mais espaço memória do que outros algoritmos que ordenam os vetores no lugar (*in-place*). É possível imaginar uma implementação *in-place* da operação de intercalagem, mas ela ia exigir trocas de elementos feitas de uma maneira não trivial.
- A equação de recursão é dada por: $T(n) = 2T(\frac{n}{2}) + \Theta(n)$, o que resulta em uma complexidade temporal: $T(n) = \Theta(n \log n)$

Quicksort

Um algoritmo com uma complexidade quase tão boa quanto a do *mergesort* que tem a vantagem de ser *in-place* é o *quicksort*. O *quicksort* geralmente já está implementado em bibliotecas nas diferentes linguagens de programação. Veremos aqui o algoritmo geral e a analise deste.

Algoritmo Quicksort

```
quicksort(vec, ini, fim)
  if ini >= fim return
  q = particione(vec, ini, fim)
  quicksort(vec, ini, q - 1)
  quicksort(vec, q + 1, fim)
```

```
particione(vec, p, r)
  q = p
  for u = p until r - 1
    if vec[u] <= vec[r]
      swap(vec, u, q)
      q++
  swap(vec, q, r)
  return q
```

Análise do Quicksort

- De novo temos um algoritmo recursivo, com características muito próximas do mergesort
- Diferente do mergesort, o quicksort não precisa de cópias do vetor.
- O elemento vec[r] é chamado de pivô do particionamento. Nesta operação, todos os elementos menores que o pivô são colocados à esquerda do pivô e todos os maiores ou iguais, à direita. O pivô é colocado na sua posição correta.
- Depois do particionamento, os subvetores à esquerda e à direita podem estar fora de ordem, por isso eles são recursivamente ordenados pelo quicksort.
- Idealmente, o pivô divide os subvetores em tamanhos quase iguais e com isto a complexidade temporal do algoritmo é $\mathcal{O}(n \log n)$.
- Se um dos subvetores é nulo, isto é, o pivô é um dos elementos da ponta, temos o pior caso, e o algoritmo é $\mathcal{O}(n^2)$