Classificação por Particionamento

O método de particionamento é um caso de aplicação do princípio da divisão e conquista: classificar dois vetores de tamanho n/2 é mais fácil que classificar um vetor de tamanho n.

O método basicamente se resume em:

i. particionar o vetor;

ii. classificar cada partição.

Classificação por Particionamento - quicksort

Um processo que trabalha com base nessa proposta é o *quicksort*, assim denominado por seu inventor, C. A. R. Hoare. É um processo *in situ* e seu caso trivial é a partição de tamanho 1. pode ser visto, como uma melhoria do método das trocas, buscando reposicionar primeiro as chaves mais distantes do seu lugar final. Funda-se em separar o vetor em duas partições, delimitadas por uma certa chave, dita *pivô*, tais que numa

das partições estejam todas as chaves menores ou iguais ao pivô e na outra todas as chaves maiores ou iguais a ele. Em seguida, reclassifica-se cada partição.

Para uma melhor compreensão vamos analisar um exemplo (consideraremos o primeiro elemento da partição como o pivô, com o objetivo de facilitar o processo).

Classificação por Particionamento - quicksort

Se um vetor inicial for dado como:

25 57 48 37 12 92 86 33

e o pivô (25) for colocado na posição correta o vetor resultante será:

12 25 57 48 37 92 86 33

agora, o problema inicial foi decomposto na classificação de dois subvetores:

(12) e (57 48 37 92 86 33)

O primeiro já está classificado, uma vez que tem apenas um elemento. O processo se repete para classificar o segundo subvetor.

Agora, , o vetor pode ser visualizado como:

12 25 (57 48 37 92 86 33)

onde os parênteses encerram os subvetores que ainda serão classificados. Repetindo o processo sobre o subvetor a ordenar teremos:

12 25 (48 37 33) 57 (92 86)

e as aplicações posteriores resultam em:

12 25 (37 33) 48 57 (92 86)

12 25 (33) 37 48 57 (92 86)

12 25 33 37 48 57 (92 86)

12 25 33 37 48 57 (86) 92

12 25 33 37 48 57 86 92

Classificação por Particionamento - quicksort

Os observadores mais atentos devem ter percebido que o quicksort pode ser melhor compreendido definido recursiva-mente.

É interessante se observar que no método apresentado os elementos após o particionamento aparecem na mesma ordem relativa em que apareciam no vetor original. Entretanto, este método de particionamento é ineficiente de se implementar. Veremos agora um método eficiente de se implementar o particionamento.

Para tanto, com um cursor, digamos “esq”, percorre-se o vetor da esquerda para a direita, até se localizar um elemento maior que o pivô. Com outro cursor, “dir”, percorre-se o vetor da direita para a esquerda, até se encontrar um elemento menor ou igual ao pivô, esses valores são intercambiados caso “dir” seja maior que “esq”. Repete-se até os setores perscrutados esgotarem o vetor, o que se pode detectar pelo encontro dos cursores, ou seja, quando esq > dir. Neste momento, troca-se o valor do pivô pelo valor do elemento indexado por dir que é dito “ponto de partição”. O qual delimitará as partições a serem reclassificadas.

Este processo pode ser observado no exemplo:

25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	57	48	37	12	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
25	12	48	37	57	92	86	33
12	25	48	37	57	92	86	33

Classificação por Particionamento - quicksort

Exercício:

Com base no que foi discutido implemente a função particionar. A qual recebe um vetor, os índices do limite inferior e superior de um subvetor pertencente a este e retorna o índice do ponto de particionamento.

```
int particionar (int v[], int ii, int is) {
   int esq=ii, dir=is, pivo=v[ii];
   while (esq<dir) {
      while (v[esq]<=pivo && esq<is)
         esq++;
      while (v[dir]>pivo)
         dir--;
      if (esq<dir) {
         int temp;
         temp = v[esq];
         v[esq]=v[dir];
         v[dir]=temp; } }
   v[ii]=v[dir];
   v[dir]=pivo;
   return dir; }
```

Classificação por Particionamento - quicksort

Exercício:

Agora, construa uma função recursiva, em C, que recebe um vetor de inteiros e o número de elementos neste vetor. Esta função deve ordenar o vetor implementando o quicksort.

```
void quicksort (int v[], int n)
{
   if (n>1)
   {
      int pont_part=particionar(v, 0, n-1);
      quicksort (v, pont_part);
      quicksort (&v[pont_part+1],
       n-1-pont_part);
   }
}
```

Classificação por Particionamento - quicksort

A tarefa de escolher o pivô pode ser executada de forma mais eficiente. A chave ideal para o pivô seria a mediana das chaves, com a qual se teria uma divisão a mais balanceada possível. Só que para determinar a mediana é preciso ordenar o vetor! Como a distribuição das chaves é, em princípio, aleatória, qualquer uma tem a mesma chance de ser mediana. Mas, ao se tomar a primeira, como foi feito anteriormente, corre-se o risco de ser esta a menor (ou a maior) de todas, tornando praticamente inócuo o trabalho na primeira fase de particionamento (pois se teria uma partição sem nenhum elemento e outra com *n-1* elementos). Uma escolha melhor é a mediana entre a primeira chave, a última e a do meio do vetor.

Classificação por Particionamento - quicksort

Como um exercício, reescreva os algoritmos anteriores, considerando que o pivô não é mais o primeiro elemento mas, sim a mediana entre o primeiro elemento, o último e o elemento do meio do vetor.

Classificação por Particionamento - quicksort

O desempenho do algoritmo *quicksort* pode ser aquilatado com base nas considerações que seguem. Na fase de particionamento, todos os elementos são comparados com pivô. Na pior hipótese, todas as chaves seriam trocadas (caso do vetor invertido). Logo, este é o processo **O**(n). Por outro lado, as partições vão diminuindo, e, na melhor hipótese, vão se subdivindo em duas partições de mesmo tamanho (tal ocorre naturalmente quando o vetor está ordenado ou invertido). Nesse caso, gasta-se $\log_2 n$ reclassificações até se chegar às partições de tamanho 1. o melhor desempenho deste processo é então de ordem n log n.

Classificação por Particionamento - quicksort

O pior caso ocorre quando o pivô escolhido é uma chave mínima (ou máxima), pois acarreta uma partição nula e outra de n – 1 elementos. Se isto se repetir em todas as reclassificações, serão necessárias n subsivisões até a conclusão, e o desempenho para o pior caso é **O**(n^2). a chance de isso ocorrer é de apenas $1/n^3$ (se houver três chaves mínimas (ou máxima), ocupando exatamente a primeira posição, a intermediária e a última em cada partição).

Classificação por Inserção

A proposta da ordenação por inserção é a seguinte:

Para cada elemento do vetor **faça**

Inseri-lo na posição que lhe corresponde;

Um processo *in situ*, chamado **inserção direta**, pode ser assim descrito:

Considerar o subvetor ordenado v[0..k - 1];

Para j de k até n faça

Inserir v[j] na sua posição em v [0..k - 1];

Classificação por Inserção Direta

Para encontrar o lugar de v[k], basta comparar as chaves de índices [0..k-1] até encontrar uma chave v[k_ins] que seja maior que ele. v[k_ins] será a sucessora de v[k] depois deste ser inserido no subvetor ordenado (i.e., depois de ser localizado). O problema que surge agora é: como arranjar lugar em v[0..k - 1] para o valor v[k]?

Classificação por Inserção Direta

Para encontrar o lugar de v[k], basta comparar as chaves de índices [0..k-1] até encontrar uma chave v[k_ins] que seja maior que ele. v[k_ins] será a sucessora de v[k] depois deste ser inserido no subvetor ordenado (i.e., depois de ser localizado). O problema que surge agora é: como arranjar lugar em v[0..k - 1] para o valor v[k]?

Bem, como v[k] vai deixar seu lugar, este pode ser ocupado pelo elemento v[k-1], ao se fazer avançar uma posição para frente todo o subvetor v[k_ins..k – 1].

Classificação por Inserção Direta

O processo começa adotando-se o subvetor ordenado v[0..0], e passando-se a inserir os elementos de índice 1..n. Para melhorar um pouco o desempenho, inicia-se a pesquisa a partir da posição k, ao mesmo tempo que se vai deslocando os elementos v[k_ins..k – 1]. A busca da posição de inserção deve prosseguir enquanto v[i $\in$ 0..k-1] > v[k]. Se a chave v[k] é menor que qualquer chave v[0.. k -1], acaba-se percorrendo o vetor até o seu início, correndo-se o risco de um índice inválido (-1). Por isso, o critério de parada deve incluir uma segunda condição: k_ins >= 0.

Classificação por Inserção Direta

Observe no exemplo a seguir o comportamento de um vetor submetido á classificação por inserção direta. Em cada linha é listado um vetor depois de uma passagem completa sobre o mesmo, estando sublinhado o subvetor [0..k-1] e em negrito o elemento nele inserido por último.

Vetor original:	75 25 95 87 64 59 86 40 16 49
passagem	**vetor resultante**
1	**25** 75 95 87 64 59 86 40 16 49

passagem	vetor resultante
2	25 75 **95** 87 64 59 86 40 16 49
3	25 75 **87** 95 64 59 86 40 16 49
4	25 **64** 75 87 95 59 86 40 16 49
5	25 **59** 64 75 87 95 86 40 16 49
6	25 59 64 75 **86** 87 95 40 16 49
7	25 **40** 59 64 75 86 87 95 16 49
8	**16** 25 40 59 64 75 86 87 95 49
9	16 25 40 **49** 59 64 75 86 87 95

Com base no que foi discutido, codifique uma função que receba um vetor (de inteiros) e o número de elementos no mesmo e através da inserção direta ordene de forma crescente os elementos do vetor.

```
void insercao_direta (int v[ ], int n)
{
   int k_ins, k, elemento;
   for (k=1;k<n;k++)
   {
      k_ins=k-1;
      elemento=v[k];
      while (k_ins>=0 && v[k_ins]>elemento)
         v[k_ins+1]=v[k_ins--];
      v[k_ins+1]=elemento;
   }
}
```

Classificação por Inserção Direta

Uma análise do algoritmo de inserção direta confirma que ele é **O**(n^2) o laço externo força a execução n-1 vezes do laço interno que, por sua vez, poderá ser executado até n-1 vezes (quando o último elemento é o maior de todos). O pior caso ocorre quando o vetor está invertido: os elementos extremos têm de atravessar todo o vetor para relocalização. O melhor caso corresponde ao vetor ordenado, para o qual não são necessários deslocamentos. O caso médio ocorre quando cada chave está a n/2 posições de sua

UNIVASF

Classificação por Inserção Direta

posição final, exigindo então $n^2/2$ movimentos de chave. Observa-se também que é um processo *estável*, ou seja: a ordem relativa original das chaves já ordenadas entre si se mantém durante o processo de ordenação. Dito de outra forma: um processo é *estável* se, por conta das operações que realiza sobre o vetor, nunca uma chave é posicionada de modo incorreto, mesmo durante as passagens intermediárias.

Classificação por Inserção Direta

O processo de inserção direta pode ser melhorado utilizando-se do processo de busca binária para localizar a posição de inserção de v[k] no subvetor v[0..k-1] (estudaremos o processo de busca binária em nosso próximo módulo). Outra forma de melhoria do processo de inserção direta é a aplicação do método sobre listas encadeadas, evitando assim a necessidade de deslocamento dos elementos para uma posterior inserção.