Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Lista de exercícios (Quick-Sort)

Exercícios (seleção): necessário entregar TODOS (moodle)!

Exercício 1) Implemente em linguagem C o algoritmo Quick-Sort visto em aula e o utilize para ordenar sequências com 10, 100, 1.000, 10.000, 100.000, 1 milhão e 10 milhões de números inteiros em ordem:

- aleatória ($V = \{9, 3, 7, 0, \dots\}$)
- crescente ($V = \{1, 2, 3, 4, \dots\}$)
- decrescente ($V = \{9, 8, 7, 6, \dots\}$)
- todos elementos iguais ($V = \{2, 2, 2, 2, \dots\}$)

Para qual dos casos acima o algoritmo foi mais rápido? E para qual caso foi pior? Mostre os tempos do algoritmo, para esses 3 casos. Explique o porquê desses tempos. Para este exercício utilize o programa **quick-sort.c** (junto ao material em **arquivos.zip**).

Exercício 2) A implementação do algoritmo Quick-Sort que vimos em aula não funciona satisfatoriamente para algumas seguintes configurações. Construa um algoritmo Quick-Sort aleatorizado para tentar melhorar estas execuções. Para tanto sorteie um pivô aleatoriamente dentro de uma partição correta e troque-o com o elemento que está na posição referente ao índice da direita (posição do pivô), ou seja:

```
int random_partition (int A[], int left, int right) {
  /*Sorteio aleatório da posição do pivô {p} tal que e <= p <= d.*/
  /*Troca do pivô com o sorteio (troca de elementos em {p} e {d}).*/
  return partition (A, left, right);
}
```

Repita os testes do exercício 1 para verificar o efeito desta otimização. Para este exercício utilize o programa **quick-random.c** (junto ao material em **arquivos.zip**).

Exercício 3) Crie um algoritmo chamado Quick-Find baseado no Quick-Sort para que, em vez de ordenar uma sequência de números inteiros, ele nos retorne o k-ésimo menor elemento dessa sequência. Por exemplo: suponha que os elementos $A = \{7, 1, 3, 10, 17, 2, 21, 9\}$ estejam armazenados nessa ordem em um vetor e que desejamos obter o quinto maior elemento dessa sequência. Então, uma chamada como Quick-Find (A,0,7,4), deverá retornar o número 9, onde S é o nome do vetor, 0 e 7 são, respectivamente, o menor e o maior índice do vetor e 4 indica que desejamos o quinto menor elemento (0 ... 4). Da mesma forma teste com (A,0,7,2) que deve retornar o valor 3. Obs.: uma

solução errada neste exercício é ordenar a sequência e depois tomar o k-ésimo elemento. Utilize o seguinte protótipo para a sua função:

```
void quick_find (int *A, int left, int right, int k);
```

Para este exercício utilize o programa **quick-find.c** (junto ao material em **arquivos.zip**).

Exercícios (aprofundamento): não é necessário entregar mas é importante estudar!

Exercício 4) O algoritmo Quick-Sort é estável? Qual o pior caso do algoritmo Quick-Sort em função do tamanho do vetor (n)?

Exercício 5) A análise de execução de alguns algoritmos recursivos as vezes é complexa. Uma estratégia interessante para facilitar a visualização da pilha de recursão é imprimir uma quantidade fixa de deslocamentos (vazios) para cada nível em que o programa está na pilha de recursão. Por exemplo, considere as chamadas recursivas que o algoritmo Quick-Sort realiza para o vetor $A = \{99, 33, 55, 77, 11, 33, 88, 66, 22, 44\}$ (o mesmo vetor exemplo utilizado em aula)

```
Input
A: {99,33,55,77,11,33,88,66,22,44}
quick-sort (0,9)
    P: {33,11,33,22,44,55,88,66,77,99}
    quick-sort (0,3)
        P: {11,22,33,33}
        quick-sort (0,0)
        quick-sort (2,3)
            P: {33,33}
            quick-sort (2,2)
            quick-sort (4,3)
    quick-sort (5,9)
        P: {55,88,66,77,99}
        quick-sort (5,8)
            P: {55,66,77,88}
            quick-sort (5,6)
                P: {55,66}
                quick-sort (5,5)
                quick-sort (7,6)
            quick-sort (8,8)
        quick-sort (10,9)
Number of recursive calls: 13
Sorted
A: {11,22,33,33,44,55,66,77,88,99}
```

Considere acima que cada chamada ao algoritmo Quick-Sort exibe os índices (esquerda, meio, direita). Note que os deslocamentos para a direita mostram os "filhos" de cada chamada recursiva. Como exercício faça:

- modifique o programa **quick-sort.c** (junto ao material em **arquivos.zip**), para explorar as chamadas recursivas conforme mostrado no exemplo acima
- modifique o algoritmo Quick-Sort para contar as chamadas recursivas conforme o exemplo acima
- quantas chamadas recursivas seriam feitas se o vetor fosse $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$? Como ficam as partições para essa entrada, balanceadas ou não?

Exercício 6) O algoritmo clássico do Quick-Sort tem desempenho catastrófico em certas configurações de elementos como consequência de uma má escolha do pivô. Uma versão do Quick-Sort muito utilizada escolhe a mediana de três elementos de uma partição do vetor como pivô. Usualmente, os elementos do início, do meio e do final do vetor são os escolhidos no cálculo da mediana, ou seja, se estamos em uma partição em $A[l, \ldots, r]$, por exemplo

```
           l       m       r
A = {..., x, ..., z, ..., y, ...}
```

calcula-se o meio m de e e d e então seleciona-se a mediana dos três valores ($V[e]$, $V[m]$ ou $V[d]$) para ser o pivô, e realiza-se a troca do elemento da mediana com a posição d, usando o algoritmo particione visto em aula

```
int partition_median_of_three (int A[], int left, int right) {
  /*Escolhe a mediana de três valores entre {l}, {(l+r)/2} e {r}.*/
  /*Troca do elemento que está na posição do pivô ({r}) com o elemento da mediana.*/
  return partition (A, left, right);
}
```

Compare essa versão do particione com a do exercício anterior para vetores grandes (10 milhões de elementos ou mais).

Exercício 7) A versão da função particione que vimos em aula não é o algoritmo de particionamento original. A função particione original, devido a Hoare, é dada por:

```
int partition_hoare (int A[], int left, int right) {
  int pivot = A[left];
  int i = left - 1;
  int j = right + 1;
  while (1) {
   do {
     j--;
   } while (A[j] > pivot);
   do {
     i++;
   } while (A[i] < pivot);
   if (i < j)
     swap (A, i, j);
   else
     return j;
```

```
  }
}

void quick_sort (int *A, int left, int right) {
  if (left < right) {
    int pivot = partition_hoare (A, left, right);
    quick_sort (A, left, pivot);
    quick_sort (A, pivot+1, right);
  }
}
```

- Repita os testes do exercício 1 para dez milhões de elementos para verificar a performance do algoritmo acima.
- Mostre como modificar o algoritmo acima para realizar a escolha aleatória do pivô e repita os testes do item anterior.

Exercício 8) Escreva uma versão híbrida do algoritmo QuickSort com o objetivo de otimizar a execução em vetores pequenos: quando o vetor a ser ordenado tiver menos que T elementos (threshold que pode ser ajustado), a ordenação passa a ser feita pelo algoritmo Insertion-Sort. O valor de T pode ficar entre 5 e 20 (faça testes para verificar qual o melhor valor para ordenar dez milhões valores aleatórios). Experimentos mostram que um $5 < T < 25$ otimiza o tempo de execução em torno de 10% (verifique). (Esse truque é usado na prática porque o algoritmo de inserção é mais rápido que o QuickSort puro quando o vetor é pequeno. O fenômeno é muito comum: algoritmos sofisticados são tipicamente mais lentos que algoritmos simplórios quando o volume de dados é pequeno.)

Exercício 9) O algoritmo Quick-Sort visto em aula contém duas chamadas recursivas. No entanto, a segunda chamada recursiva não é realmente necessária, e podemos evitá-la utilizando uma estrutura de repetição. Essa técnica, denominada **recursão de cauda**, é automaticamente fornecida por bons compiladores. O algoritmo abaixo é uma variante do Quick-Sort com apenas uma chamada recursiva:

```
void quick_sort (int *A, int left, int right) {
  while (left < right) {
    int pivot = partition (A, left, right);
    quick_sort (A, left, pivot-1);
    left = pivot + 1;
  }
}
```

No entanto, a profundidade da pilha ainda assim pode ter tamanho $O(n)$ para o pior caso do algoritmo (quando todos os elementos ficam em um lado da partição), e desta forma o algoritmo pode causar um estouro de pilha para vetores grandes. Um jeito de amenizar essa situação é alterar o código acima para usar um espaço na pilha de recursão de no máximo $O(\log n)$. Descreva qual modificação que você faria no código acima para realizar essa otimização.

Exercício 10) Você tem uma pilha mista de n porcas e n parafusos e precisa encontrar rapidamente os pares correspondentes de porcas e parafusos. Cada porca coincide exatamente com um parafuso, e cada parafuso corresponde exatamente a uma porca. Ao encaixar uma porca em um parafuso, você pode ver qual é maior. No entanto, não é possível comparar diretamente duas porcas ou dois parafusos. Descreva (pode ser a ideia) como alterar o algoritmo Quick-Sort para resolver esse problema. Ps. se quiser codificar pode usar um esquema com simbolos para descrever cada tamanho de porca e parafuso (que podem ser comparados de forma simples devido a tabela ascii):

```
Entrada:
Porcas:    @ # $ % ^ &
Parafusos: $ % & ^ @ #

Saída:
Porcas:    # $ % & @ ^
Parafusos: # $ % & @ ^
```