Universidade Federal de Minas Gerais
Departamento de Ciência da Computação
Algoritmos e Estruturas de Dados II (Turmas N, W, F)
1º Semestre de 2012

3ª Lista de Exercícios (parte 1)

1. Sejam N registros armazenados em um vetor A. Proponha um algoritmo para ordenar o vetor em tempo linear O(n) que não necessite de espaço adicional para os seguintes casos. i) Todas as chaves são ou 0 ou 1; ii) Todas as chaves são números inteiros entre [0,..,k] onde k é uma constante.

i) Passe uma vez pelo vetor contanto o número n0 de chaves com valor zero (custo O(n)). Inicialize c0 = 0 e c1 = n0. Passe novamente pelo vetor. Para cada elemento com chave zero, coloque-o na posição c0 e incremente o valor de c0; para cada elemento com chave um, coloque-o na posição c1 e incremente o valor de c1.

ii) Igual à solução anterior, exceto que precisamos contar o número nX de chaves com valor X (0 <= X <= k), e inicializar contadores cY (0 <= Y <= k-1).

2. Use um método de ordenação instável A para criar um novo método de ordenação que seja estável e que apresente a mesma complexidade do algoritmo A.

Estenda a chave dos elementos no vetor de entrada com a *posição* (índice) do elemento no vetor de entrada. Use a posição original do elemento no vetor de entrada como critério de desempate caso dois elementos tenham chaves iguais.

3. Qual método roda mais rápido em um vetor com chaves idênticas: seleção ou inserção? Justifique sua resposta.

Inserção roda em tempo linear. Seleção não é adaptável e roda em tempo quadrático.

4. Um trabalhador de um depósito precisa rearranjar todas as caixas disponíveis no estoque de acordo com algum critério. Neste caso, o custo de comparações é pequeno comparado com o custo da troca de posições entre as caixas. Há espaço (extra) suficiente para apenas uma caixa. Qual método de ordenação deveria ser utilizado nesta situação.

O método da seleção é a melhor solução pois faz no máximo O(n) trocas (onde n é o número de caixas).

5. Execute o heapsort no seguinte vetor de entrada (Indique os passos intermediários utilizando a representação por árvore).

X	T	O	G	S	M	N	A	E	R	A	I

6. Altere o código do quicksort visto em sala para contemplar a estratégia que utiliza a mediana de três elementos para escolha do pivô.

```
void particao(struct item *v, int e, int d) {
   int i = e; int j = d;
   struct item pivo = mediana(v, e, d);
   while(1) {
      while(v[i].chave < pivo.chave) i++;
      while(pivo.chave < v[j].chave) j--;
      if(i >= j) break;
      troca(v[i], v[j]);
   }
   return i;
}

struct item mediana(struct item *v, int e, int d) {
   struct item a = v[e];
   struct item b = v[(e+d)/2];
   struct item c = v[d];
   struct item menor;
   struct item maior;
   if(a.chave > b.chave) {
      maior = a;
      menor = b;
   } else {
      maior = b;
      menor = a;
   }
   if(c.chave > maior.chave) return maior;
   if(c.chave < menor.chave) return menor;
   return c;
}
```

7. Considere o seguinte algoritmo (v: vetor de entrada, e: índice da esquerda, d: índice da direita do vetor).

```
void funcao(Item *v, int e, int d) {
   int m = (d + e) / 2;
   if(d <= e) return;
   funcao(v, e, m);
   funcao(v, m+1, d);
   aux(v, e, m, d);
}
```

Escreva o código para função aux de modo que o algoritmo mostrado ordene os elementos no vetor v.

```
void aux(Item *v, int e, int m, int d) {
   Item *tmp = malloc((d-e+1)*sizeof(Item));
   int s = m+1;
   int i = 0;
   while(i < d-e+1) {
      while(e <= m && (s == d || v[e] <= v[s])) {
         tmp[i] = v[e];
         e++;
         i++;
      while(s <= d && (e == m || v[e] > v[s])) {
         tmp[i] = v[s];
         i++;
         s++;
      }
   }
   memcpy(v+e, tmp, (d-e+1)*sizeof(Item));
}
```

8. Descreva como você modificaria o RadixSort para ordenar cadeias de caracteres de tamanho diferente. Por exemplo, a chave “carrega” deve estar antes de “carregamento” e depois de “borboleta”.

Faça todas as chaves ter o mesmo número de caracteres. Complete as chaves pequenas com um caractere marcador à direita. O caractere marcador deve ter o menor valor possível. Desta forma, ao executar o radixsort elas ficarão ordenadas:

```
borboletaXXX
carregaXXXXX
carregamento
```

9. Qual algoritmo de ordenação você usaria para cada um dos seguintes casos:

a) A ordenação original de elementos com chave idêntica precisa ser mantida.

Radixsort. Caso as chaves não sejam fáceis de usar com radixsort (e.g., números ponto flutuante), podemos usar o mecanismo descrito na questão 2.

b) O tempo de execução não deve apresentar grandes variações para nenhum caso.

Heapsort ou radixsort.

c) A lista a ser ordenada já está bem próxima da ordem final.

Inserção.

10. Modifique o algoritmo de ordenação por inserção de forma que ele utilize busca binária para encontrar a posição de inserção de um elemento no vetor destino. Considerando o número C(n) de comparações efetuadas, determine a complexidade do algoritmo obtido.

```
void insercao(struct item *v, int n) {
  int i, j, k;
  struct item aux;
  for(i = 1; i < n; i++) {
    j = binaria(v, v[i].chave, 0, i-1);
    aux = v[i];
    for(k = i; k > j; k--) {
      v[k] = v[k-1];
    }
    v[j] = aux;
  }
}

int binaria(struct item *v, int chave, int esq, int dir) {
    int i, esq, dir;
    do {
        i = (esq + dir) / 2;
        if(chave > v[i].chave) {
            esq = i + 1; /* procura na partição direita */
        } else {
            dir = i - 1; /* procura na part esquerda */
        }
    } while((chave != v[i].chave) && (esq <= dir));
    if(chave > v[i].chave) i++;
    return i;
}
```

O algoritmo modificado faz O(*n* lg *n*) comparações (mas continua fazendo O(n*n) movimentações). Note que o algoritmo modificado não tem custo linear para entradas pré-ordenadas.