Cursos de Verão 2010 - IME/USP
Tópicos de Programação
Countingsort e Radixsort
Prof. Alexandre da Silva Freire

1 Countingsort

A seguir veremos mais um algoritmo de ordenação. O algoritmo que chamaremos de `countingsort` recebe um inteiro n, um vetor de inteiros v tal que $v[0\,..\,n-1]$, e um inteiro $k = \max(v[0], v[1], \ldots, v[n-1])$.

```
    void countingsort(int v[], int n, int k) {
       int* t; int* x; int i;

       /* aloca os vetores auxiliares */
       t = malloc(sizeof(int) * n);
       x = malloc(sizeof(int) * (k + 1));

       /* zera todas as posições do vetor x */
(1)    for(i = 0; i <= k; i++) {
          x[i] = 0;
       }

       /* faz com que x[j] seja o número de i's tais que v[i]=j */
(2)    for(i = 0; i < n; i++) {
          x[v[i]]++;
       }

       /* faz com que x[i] seja o número de j's tais que v[j]<=i */
(3)    for(i = 1; i <= k; i++) {
          x[i] = x[i] + x[i - 1];
       }

       /* copia em t os elementos de v em ordem crescente */
       /* IMPORTANTE: preserva a ordem relativa dos elementos */
(4)    for(i = n-1; i >= 0; i--) {
          t[x[v[i]] - 1] = v[i];
          x[v[i]]--;
       }

       /* copia os elelementos de t em v */
(5)    for(i = 0; i < n; i++) {
          v[i] = t[i];
       }
    }
```

Qual é o consumo de tempo do algoritmo `countingsort`? Os laços das linhas (2), (4) e (5) consomem tempo $\Theta(n)$ cada um. Os laços das linhas (1) e (3) consomem tempo $\Theta(k)$ cada um. Portanto, o consumo total de tempo é $4 \cdot \Theta(n) + 2 \cdot \Theta(k) = \Theta(n + k)$.

2 Radixsort

Um algoritmo de ordenação é dito *estável* se ele preserva a ordem relativa dos elementos da entrada. Ou seja, se $a = b$ e a aparece antes de b na entrada, então a deve aparecer antes de b após o trabalho feito pelo algoritmo. Dos algoritmos vistos nas aulas anteriores, quais são estáveis? O `countingsort` é um exemplo de algoritmo estável.

O algoritmo que chamaremos de `radixsort` utiliza como subrotina uma função que chamaremos de `countingsort2`. Esta função é uma versão do `countingsort` na qual os elementos são ordenados de acordo com o i-ésimo dígito da direita para a esquerda (do menos significativo para o mais significativo), sendo que i é um parâmetro desta função.

Exemplo: para $n = 4$ e $v = [233\mathbf{2}0, 020\mathbf{2}3, 000\mathbf{1}7, 123\mathbf{0}2]$, após uma chamada `countingsort2(v,n,2)` (ordena pelo segundo dígito), deve valer que $v = [123\mathbf{0}2, 000\mathbf{1}7, 233\mathbf{2}0, 020\mathbf{2}3]$. Repare que devido à estabilidade do `countingsort`, o elemento $233\mathbf{2}0$ deve aparecer antes de $020\mathbf{2}3$.

Além de v e n, o `radixsort` recebe como parâmetro um inteiro d que indica o número de dígitos que possui o maior elemento de $v[0, 1, \ldots, n-1]$. Eis o algoritmo:

```
    void radixsort(int v[], int n, int d) {
       int i;
(1)    for(i = 1; i <= d; i++) {
          /* ordena (com algoritmo estável) pelo dígito i */
(2)       countingsort2(v, n, i);
       }
    }
```

Qual é o consumo de tempo do algoritmo `radixsort`? O laço da linha (1) é executado $\Theta(d)$ vezes, sendo que cada uma delas consome tempo $\Theta(n + k) = \Theta(n)$, pois $k \leq 9$ (k é limitado por uma constante). Portanto, o consumo total do algoritmo `radixsort` é $\Theta(n \cdot d)$. Infelizmente, d não é limitado por uma constante! É claro que em um computador d é limitado por uma constante. Para avaliar na prática qual algoritmo utilizar em uma determinada situação, precisamos fazer contas mais detalhadas (olhar todas as constantes "escondidas" na análise assintótica).