Algoritmos e Estruturas de Dados: Ordenação

Rômulo Silva de Oliveira
Departamento de Automação e Sistemas – DAS – UFSC

romulo@das.ufsc.br
http://www.das.ufsc.br/~romulo
Maio/2011

Referências

- Mastering Algorithms with C
  Kyle Loudon
  O´Reilly, 1999
- Livros de algoritmos e estruturas de dados em geral

Ordenação – Introdução 1/3

- Algoritmos de ordenação são extremamente úteis
- Existem as dezenas
- Algumas variações entre suas propriedades
- Melhor algoritmo de ordenação vai depender da situação em questão

Ordenação – Introdução 2/3

- Ordenação por Comparação
  - Compara elementos para coloca-los na ordem correta
  - Não é possível fazer mais rápido que $O(n \lg n)$
- Ordenação em Tempo Linear
  - Tempo proporcional ao número de elementos ordenados, ou $O(n)$
  - Baseia-se em certas características dos dados
  - Nem sempre pode ser aplicado
- Ordenação no Lugar
  - Usa a mesma memória para colocar a saída ordenada
- Outras ordenações precisam de memória extra para os dados de saída
  - Pode deletar os dados de entrada ao final

Ordenação – Introdução 3/3

- Insertion sort
  - Simples, ordena no lugar, melhor para ordenação incremental
- Quicksort
  - Ordena no lugar, melhor no caso geral
- Merge sort
  - Mesmo desempenho que quicksort, necessita o dobro da memória, facilita trabalhar com a divisão em partes dos dados originais
- Counting sort
  - Algoritmo com tempo linear, trabalha com inteiros cujo maior valor é conhecido
- Radix sort
  - Algorimo com tempo linear, ordena dígito por dígito, bom para elementos de tamanho fixo que podem ser quebrados em pedaços, na forma de inteiros

Ordenação – Header 1/6

```
#ifndef SORT_H
#define SORT_H


int issort(void *data, int size, int esize,
                int (*compare)(const void *key1, const void *key2));

int qksort(void *data, int size, int esize, int i, int k,
                int (*compare)(const void *key1, const void *key2));

int mgsort(void *data, int size, int esize, int i, int k,
                int (*compare)(const void *key1, const void *key2));

int ctsort(int *data, int size, int k);

int rxsort(int *data, int size, int p, int k);

#endif
```

Ordenação

- **Insertion sort**
- Quicksort
- Merge sort
- Counting sort
- Radix sort

Insertion Sort – Introdução 1/9

- Um dos mais simples
- Começa com um conjunto de dados desordenados
- Um por um, remove o dado do conjunto desordenado e insere na posição apropriada no conjunto ordenado
- Na inserção é necessário percorrer o conjunto ordenado para determinar o lugar deste novo elemento
- É possível ordenar sem gastar memória adicional
  - Apenas move o elemento de lugar
- Pode-se implementar com alocação contígua ou lista encadeada

Insertion Sort – Introdução 2/9

- Algoritmo simples
- Ineficiente para grandes conjuntos de dados
  - Determinar onde colocar o novo elemento pode requerer uma compara ção dele com todos os demais elementos já inseridos no conjunto ordenado
- Inserir um único elemento em um conjunto já ordenado é rápido
  - Eficiente para ordenação incremental
  - Por exemplo, incluir novas reservas em uma lista

Insertion Sort – Introdução 3/9

- Inicia posicionando o 23

23	21	76	16	52	43

Insertion Sort – Introdução 4/9

- Agora vai posicionar o 21

23	21	76	16	52	43
23	21	76	16	52	43

Insertion Sort – Introdução 5/9

- Agora vai posicionar o 76

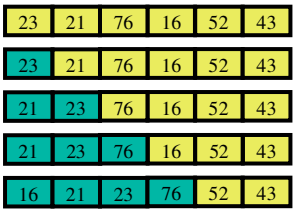
23	21	76	16	52	43
23	21	76	16	52	43
21	23	76	16	52	43

Insertion Sort – Introdução 6/9

- Agora vai posicionar o 16

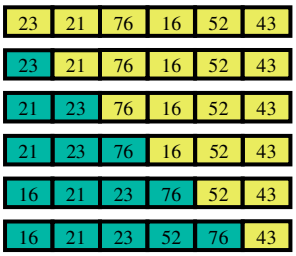
Insertion Sort – Introdução 7/9

- Agora vai posicionar o 52



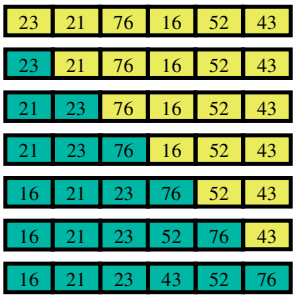
Insertion Sort – Introdução 8/9

- Agora vai posicionar o 43



Insertion Sort – Introdução 9/9

- Array está ordenado, no mesmo lugar
  - O (n^2)



Insertion Sort – Interface 1/1

- ***issort***
  - int issort(void **data*, int *size*, int *esize*,
    int (***compare*)(const void ***key1*, const void ***key2*));
- **Retorno**
  - 0 se a ordenação tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Usa insertion sort para ordenar o array de elementos em *data*. O número de elementos em *data* é especificado por *size*. O tamanho de cada elemento é especificado por *esize*. O pointer para função *compare* especifica uma função definida por usuário que compara os elementos. Esta função deverá retornar 1 se *key1>key2*, 0 se *key1=key2*, e -1 se *key1<key2* para uma ordenação crescente. Para uma ordenação decrescente, *compare* deverá inverter os casos que retornam 1 e -1. Quando *issort* returna, *data* contem os elementos ordenados.

Insertion Sort – Implementação 1/1

```
#include <stdlib.h>
#include <string.h>
#include "sort.h"

int issort(void *data, int size, int esize, int (*compare)(const void *key1, const void *key2)) {
    char *a = data;
    void *key;
    int i, j;
    if ((key = (char *) malloc(esize)) == NULL)        return -1;

    for (j = 1; j < size; j++) {
        memcpy(key, &a[j * esize], esize);
        i = j - 1;
        while (i >= 0 && compare(&a[i * esize], key) > 0) {
            memcpy(&a[(i + 1) * esize], &a[i * esize], esize);
            i--;
        }
        memcpy(&a[(i + 1) * esize], key, esize);
    }
    free(key);
    return 0;
}
```

Ordenação

- Insertion sort
- **Quicksort**
- Merge sort
- Counting sort
- Radix sort

Quick Sort – Introdução 1/11

- Considerado o melhor para uso geral
- Ordena no lugar
- Começa com um conjunto desordenado
  que é dividido em dois sub-conjuntos
- Em um deles coloca-se todos os elementos menores ou iguais àquele
  que imagina-se ser a mediana
  - No outro sub-conjunto coloca-se os elementos que são maiores
- Uma vez com esses dois sub-conjuntos
  - Divide-se cada um deles da mesma maneira
  - Repete o processo até que cada sub-conjunto tenha 1 elemento apenas

Quick Sort – Introdução 2/11

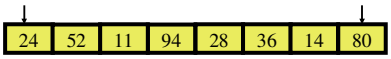
- Quicksort recursivamente
  particiona um conjunto desordenado de elementos
  até que todas as partições contenham um único elemento
- No pior caso, quicksort não é melhor que insertion sort
  - Truque é reduzir as chances de quicksort ter um pior caso
  - No caso médio ele é melhor que insertion sort

Quick Sort – Introdução 3/11

- Questão fundamental: como escolher o valor de particionamento
  - Quicksort tem pior desempenho quando os valores de particionamento colocam a maioria dos elementos na mesma partição
- O particionamento precisa ser balanceado
- Exemplo: {15, 20, 18, 51, 36, 10, 77, 43}
  - Usando 10: {10} e {20, 18, 51, 36, 15, 77, 43}
  - Usando 36: {15, 20, 18, 10} e {36, 51, 77, 43}
- Necessário um método para escolha do valor de particionamento
- Escolhe aleatoriamente 1 elemento do conjunto
- Escolhe aleatoriamente 3 elementos do conjunto e seleciona o valor mediado (método da mediana de três)

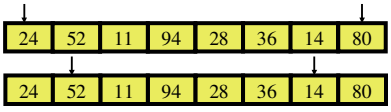
Quick Sort – Introdução 4/11

- Particionamento de um conjunto, começa nas extremidades, usa 40



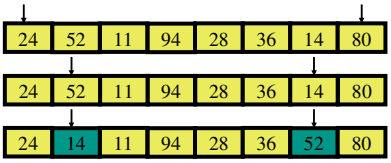
Quick Sort – Introdução 5/11

- Avança esqueda e direita até achar valores em subconjuntos errados



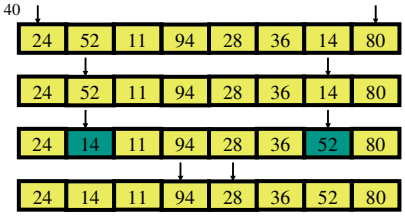
Quick Sort – Introdução 6/11

- Troca os valores de sub-conjunto



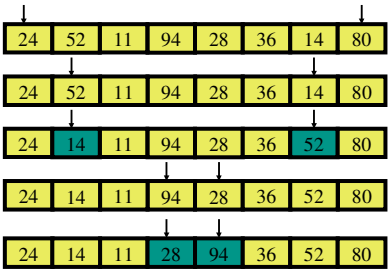
Quick Sort – Introdução 7/11

- Avança esquerda e direita até achar 2 valores em subconjuntos errados
  - Usa 40



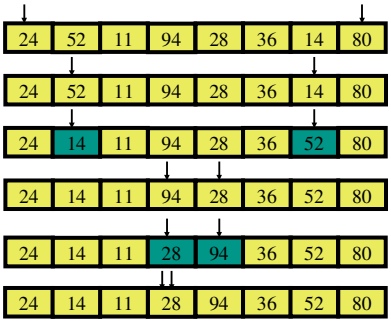
Quick Sort – Introdução 8/11

- Troca os valores de sub-conjunto



Quick Sort – Introdução 9/11

- Termina quando os pointers se encontram (cruzam)

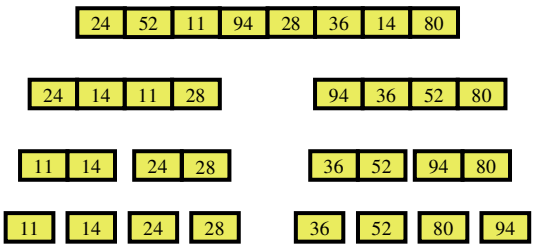


Quick Sort – Introdução 10/11

- Em seguida,
- Chama *qksort* recursivamente para a partição da esquerda
  - Continuará até chegar em uma partição de apenas 1 elemento
- Chama *qksort* recursivamente para a partição da direita
  - Continuará até chegar em uma partição de apenas 1 elemento
- Desempenho no pior caso é $O(n^2)$
- Dsempenho no caso médio é $O(n \lg n)$
- Quicksort ordena no lugar

Quick Sort – Introdução 11/11

- Aplica recursivamente o particionamento
  - Neste exemplo a escolha do valor de particionamento foi ideal



Quick Sort – Interface 1/1

- ***qksort***
  - int qksort(void **data*, int *size*, int *esize*, int *i*, int *k*, int (***compare*) (const void **key1*, const void **key2*));
- **Retorno**
  - 0 se a ordenação tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Usa quicksort para ordenar o array de elementos em *data*. O número de elementos em *data* é especificado por *size*. O tamanho de cada elemento é especificado por *esize*. Os argumentos *i* e *k* definem a partição corrente sendo ordenada e inicialmente deveriam ser 0 e *size* - 1, respectivamente. O pointer para função *compare* especifica uma função definida por usuário para comparar elementos. Ela deve executar de uma forma similar àquela descrita para para *issort*. Quando *qksort* retorna, *data* contem os elementos ordenados.

Quick Sort – Implementação 1/4

```
#include <stdlib.h>
#include <string.h>

#include "sort.h"

static int compare_int(const void *int1, const void *int2) {

    if (*(const int *)int1 > *(const int *)int2)
        return 1;
    else if (*(const int *)int1 < *(const int *)int2)
        return -1;
    else
        return 0;
}
```


Quick Sort – Implementação 2/4

```
static int partition(void *data, int esize, int i, int k,
                     int (*compare)(const void *key1, const void *key2)) {

    char *a = data;

    void *pval, *temp;
    int  r[3];

    if ((pval = malloc(esize)) == NULL)
        return -1;

    if ((temp = malloc(esize)) == NULL) {
        free(pval);
        return -1;
    }

    // Use the median-of-three method to find the partition value
    r[0] = (rand() % (k - i + 1)) + i;
    r[1] = (rand() % (k - i + 1)) + i;
    r[2] = (rand() % (k - i + 1)) + i;
    issort(r, 3, sizeof(int), compare_int);
    memcpy(pval, &a[r[1] * esize], esize);
```


Quick Sort – Implementação 3/4

```
    i--;                        // Create two partitions around the partition value
    k++;
    while (1) {
        // Move left until an element is found in the wrong partition.
        do {    k--;
        } while (compare(&a[k * esize], pval) > 0);

        // Move right until an element is found in the wrong partition.
        do {    i++;
        } while (compare(&a[i * esize], pval) < 0);

        if (i >= k)  // Stop partitioning when the left and right counters cross.
            break;
        else {       // Swap the elements now under the left and right counters.
            memcpy(temp, &a[i * esize], esize);
            memcpy(&a[i * esize], &a[k * esize], esize);
            memcpy(&a[k * esize], temp, esize);
        }
    }
free(pval);         free(temp);
return k;
}
```


Quick Sort – Implementação 4/4

```
int qksort(void *data, int size, int esize, int i, int k,
        int (*compare)(const void *key1, const void *key2)) {

    int j;

    // Stop the recursion when it is not possible to partition further.
    if (i < k) {
        // Determine where to partition the elements
        if ((j = partition(data, esize, i, k, compare)) < 0)   return -1;

        if (qksort(data, size, esize, i, j, compare) < 0)
                return -1;

        if (qksort(data, size, esize, j + 1, k, compare) < 0)
            return -1;
    }

return 0;
}
```


Ordenação

- Insertion sort
- Quicksort
- **Merge sort**
- Counting sort
- Radix sort

Merge Sort – Introdução 1/14

- Realiza comparações entre elementos
- Não ordena no lugar

- Divide o conjunto desordenado na metade
- Cada um dos sub-conjuntos é dividido novamente na metade
- Continua este processo até terminar com sub-conjuntos de um único elemento

- A partir de agora:
  Junta os sub-conjuntos 2 a 2 em um sub-conjunto ordenado
- As junções continuam até chegar a um conjunto único
- O qual estará ordenado

Merge Sort – Introdução 2/14

- Juntar dois sub-conjuntos ordenados é eficiente
  - Basta uma passagem sobre cada sub-conjunto
- A divisão dos sub-conjuntos é feita de forma determinista
- Merge sort em todos os casos é tão bom quanto quicksort no caso médio

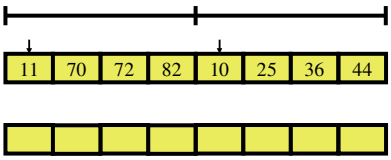
- Sua desvantagem é a necessidade de espaço

- A junção não pode ser feita no lugar, requer o dobro de memória
  - Neste aspecto quicksort é melhor no caso geral

- Merge sort é bom para conjunto de dados muito grandes (arquivos)
  - Não cabe tudo mesmo na memória
  - Merge sort divide os dados de uma forma facilmente gerenciável
  - Não precisa manter todos os dados na memória o tempo todo

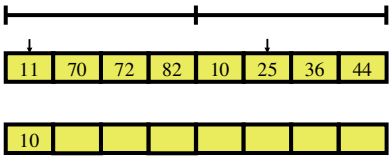
Merge Sort – Introdução 3/14

- Exemplo considera os dados originais em memória contígua
  - Poderia ser uma lista encadeada
- Necessário juntar 2 sub-conjuntos ordenados em um único (ordenado)



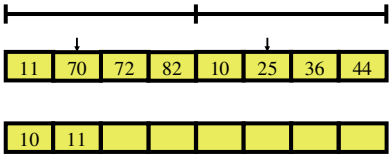
Merge Sort – Introdução 4/14

- Necessário juntar 2 sub-conjuntos ordenados em um único (ordenado)
  - Copia o 10 e move o pointer



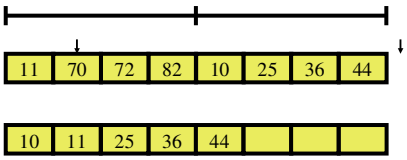
Merge Sort – Introdução 5/14

- Exemplo considera os dados originais em memória contígua
  - Poderia ser uma lista encadeada
- Necessário juntar 2 sub-conjuntos ordenados em um único (ordenado)



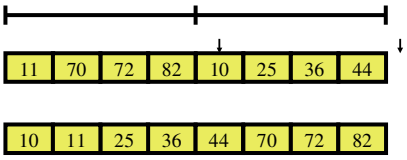
Merge Sort – Introdução 6/14

- Exemplo considera os dados originais em memória contígua
  - Poderia ser uma lista encadeada
- Necessário juntar 2 sub-conjuntos ordenados em um único (ordenado)



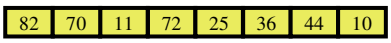
Merge Sort – Introdução 7/14

- Exemplo considera os dados originais em memória contígua
  - Poderia ser uma lista encadeada
- Necessário juntar 2 sub-conjuntos ordenados em um único (ordenado)



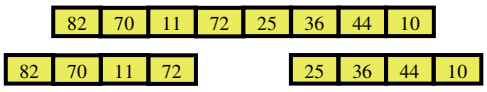
Merge Sort – Introdução 8/14

- Merge sort primeiro divide depois junta ordenado



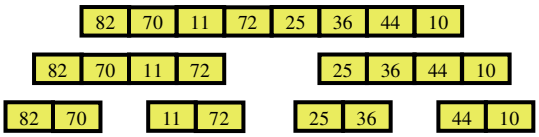
Merge Sort – Introdução 9/14

- Merge sort primeiro divide depois junta ordenado



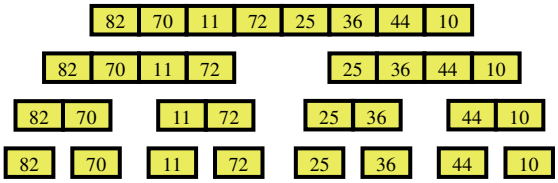
Merge Sort – Introdução 10/14

- Merge sort primeiro divide depois junta ordenado



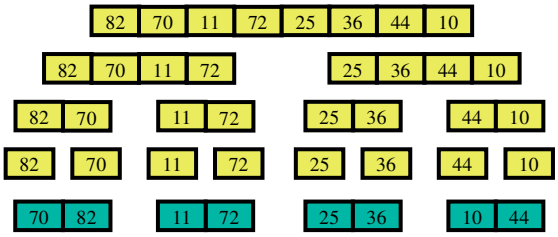
Merge Sort – Introdução 11/14

- Merge sort primeiro divide depois junta ordenado



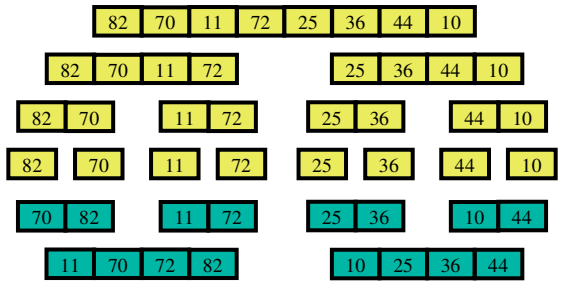
Merge Sort – Introdução 12/14

- Merge sort primeiro divide depois junta ordenado



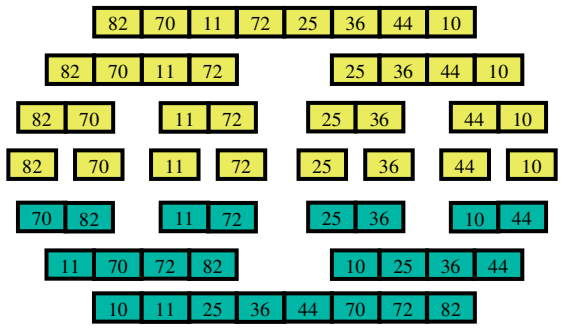
Merge Sort – Introdução 13/14

- Merge sort primeiro divide depois junta ordenado



Merge Sort – Introdução 14/14

- Merge sort primeiro divide depois junta ordenado



Merge Sort – Interface 1/1

- ***mgsort***
  - int mgsort(void **data*, int *size*, int *esize*, int *i*, int *k*,
    int (**compare*) (const void **key1*, const void **key2*));
- **Retorno**
  - 0 se a ordenação tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Usa merge sort para ordenar o array de elementos em *data*. O número de elementos em *data* é especificado por *size*. O tamanho de cada elemento é especificado por *esize*. Os argumentos *i* e *k* definem a divisão corrente sendo ordenada e inicialmente devem ser 0 e *size*-1, respectivamente. O pointer para função *compare* especifica uma função definida por usuário para comparar elementos. Ela deve funcionar como descrito para o *issort*. Quando *mgsort* retorna, *data* contem os elementos ordenados.

Merge Sort – Implementação 1/4

```
#include <stdlib.h>
#include <string.h>

#include "sort.h"

static int merge(void *data, int esize, int i, int j, int k,
    int (*compare)(const void *key1, const void *key2)) {

    char    *a = data,   *m;

    int   ipos, jpos, mpos;

    ipos = i;
    jpos = j + 1;
    mpos = 0;

    if ((m = (char *)malloc(esize * ((k - i) + 1))) == NULL)
    return -1;
```


Merge Sort – Implementação 2/4

```
    // Continue while either division has elements to merge
    while (ipos <= j || jpos <= k) {
        if (ipos > j) {
            while (jpos <= k) {
                memcpy(&m[mpos * esize], &a[jpos * esize], esize);
                jpos++;
                mpos++;
            }
            continue;
        }
        else if (jpos > k) {
            while (ipos <= j) {
                memcpy(&m[mpos * esize], &a[ipos * esize], esize);
                ipos++;
                mpos++;
            }
            continue;
        }
```


Merge Sort – Implementação 3/4

```
        // Append the next ordered element to the merged elements.
        if (compare(&a[ipos * esize], &a[jpos * esize]) < 0) {
            memcpy(&m[mpos * esize], &a[ipos * esize], esize);
            ipos++;
            mpos++;
        }
        else {
            memcpy(&m[mpos * esize], &a[jpos * esize], esize);
            jpos++;
            mpos++;
        }
    }

    memcpy(&a[i * esize], m, esize * ((k - i) + 1));
    free(m);

    return 0;
}
```

Merge Sort – Implementação 4/4

```
int mgsort(void *data, int size, int esize, int i, int k,
           int (*compare)(const void *key1, const void *key2)) {

    int j;

    if (i < k) {
        j = (int)(((i + k - 1)) / 2);

        if (mgsort(data, size, esize, i, j, compare) < 0)
            return -1;

        if (mgsort(data, size, esize, j + 1, k, compare) < 0)
            return -1;

        if (merge(data, esize, i, j, k, compare) < 0)
            return -1;
    }

    return 0;
}
```


Ordenação

- Insertion sort
- Quicksort
- Merge sort
- **Counting sort**
- Radix sort

Counting Sort – Introdução 1/12

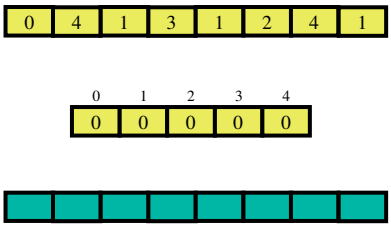
- Algoritmo de ordenação eficiente, em tempo linear
- Estável, deixa elementos de mesmo valor na mesma ordem original
- Conta quantas vezes cada elemento de um conjunto aparece
  - para determinar como o conjunto deve ser ordenado
- Funciona apenas com inteiros ou dados que podem ser representados como inteiros
- Usa array de contadores
  - Se o inteiro 3 aparece 4 vezes, 4 é armazenado na posição 3 do array
- Maior inteiro precisa ser conhecido para criar o array de contadores

Counting Sort – Introdução 2/12

- Uma vez que as ocorrências de cada elemento foram contadas
- Ajusta-se os contadores para refletir o número de elementos que virão no conjunto ordenado antes de cada elemento
- Adiciona-se a contagem de cada elemento no array à contagem de cada elemento que segue ele
- Isto resulta nos offsets de cada elemento no conjunto ordenado

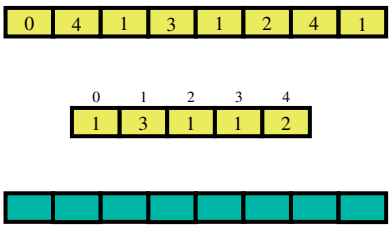
Counting Sort – Introdução 3/12

- Parte com os contadores zerados



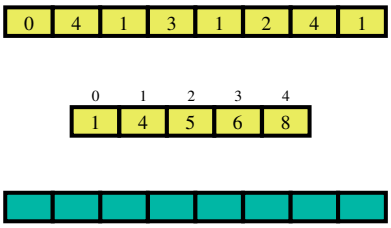
Counting Sort – Introdução 4/12

- Conta as ocorrências de cada número



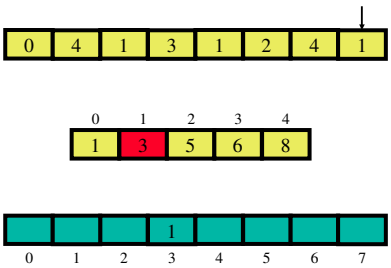
Counting Sort – Introdução 5/12

- Ajusta contadores para refletir contagens dos anteriores



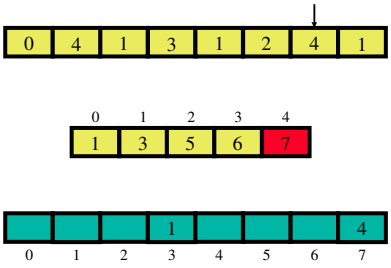
Counting Sort – Introdução 6/12

- Copia número para posição indicada, decrementa



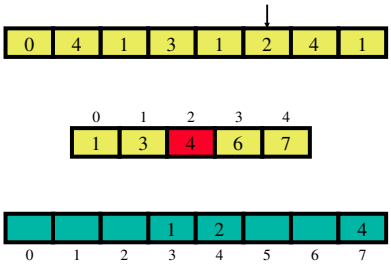
Counting Sort – Introdução 7/12

- Copia número para posição indicada, decrementa



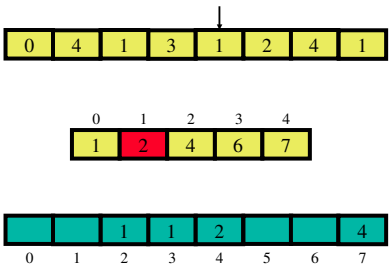
Counting Sort – Introdução 8/12

- Copia número para posição indicada, decrementa



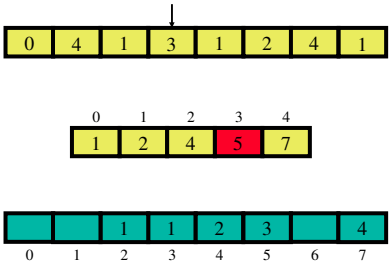
Counting Sort – Introdução 9/12

- Copia número para posição indicada, decrementa



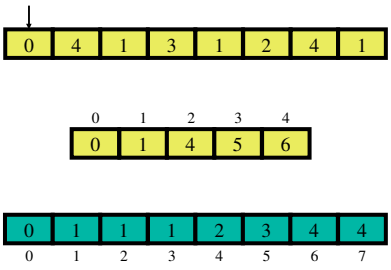
Counting Sort – Introdução 10/12

- Copia número para posição indicada, decrementa



Counting Sort – Introdução 11/12

- Ao final, array auxiliar está ordenado



Counting Sort – Introdução 12/12

- Complexidade do tempo de execução é O(n+k)
  - onde n é o número de inteiros nos dados
  - k é o inteiro de maior valor + 1
- Quanto à memória:
  requer 2 arrays de tamanho n
  e um array de tamanho k

Counting Sort – Interface 1/1

- ***ctsort***
  - int ctsort(int **data*, int *size*, int *k*);
- **Retorno**
  - 0 se a ordenação tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Usa counting sort para ordenar um array de inteiros em *data*. O número de inteiros em *data* é especificado por *size*. O argumento *k* especifica o máximo inteiro em *data*, mais 1. Quando *ctsort* retorna, *data* contem os inteiros ordenados.

Counting Sort – Implementação 1/2

```
#include <stdlib.h>
#include <string.h>

#include "sort.h"

int ctsort(int *data, int size, int k) {

    int  *counts,  *temp;
    int  i, j;

    if ((counts = (int *)malloc(k * sizeof(int))) == NULL)
            return -1;

    if ((temp = (int *)malloc(size * sizeof(int))) == NULL)
            return -1;

    for (i = 0; i < k; i++)
            counts[i] = 0;

    for (j = 0; j < size; j++)
            counts[data[j]] = counts[data[j]] + 1;
```


Counting Sort – Implementação 2/2

```
    //  Adjust each count to reflect the counts before it.

    for (i = 1; i < k; i++)
            counts[i] = counts[i] + counts[i - 1];

    // Use the counts to position each element where it belongs.

    for (j = size - 1; j >= 0; j--) {

            temp[counts[data[j]] - 1] = data[j];
            counts[data[j]] = counts[data[j]] - 1;
    }

    memcpy(data, temp, size * sizeof(int));

    free(counts);
    free(temp);

    return 0;
}
```


Ordenação

- Insertion sort
- Quicksort
- Merge sort
- Counting sort
- **Radix sort**

Radix Sort – Introdução 1/8

- Algoritmo eficiente, com tempo linear
- Ordena os dados em pedaços chamados digitos, um digito por vez
  - da posição menos significativa até a posição mais significativa
- Por exemplo, números de base 10 (radix-10)
- {15, 12, 49, 16, 36, 40}
- Produz {40, 12, 15, 16, 36, 49} após ordenar o digito menos significativo
- e {12, 15, 16, 36, 40, 49} após ordenar o dígito mais significativo

Radix Sort – Introdução 2/8

- Radix sort precisa usar uma ordenação estável para ordenar os dígitos
- Depois que um elemento recebeu um lugar conforme o dígito menos significativo
- Este lugar só poderá mudar se necessário pela ordenacao dos digitos mais significativos
- Por exemplo, 12 e 15 não podem inverter quando o digito mais significativo é considerado
- Counting sort é bom pois é
  - Estável
  - Tempo linear
  - O maior dígito é conhecido

Radix Sort – Introdução 3/8

- Radix sort pode ser usado com qualquer dado que possa ser dividido em pedaços que são inteiros
  - String visto como um inteiro de base 2^4
  - Inteiros de 64 bits como inteiros de 4 dígitos na base 2^{16}
- Que valor escolher como base depende dos dados
- Busca-se minimizar o gasto de memória pn + pk
  - p é o número de digitos em cada elemento
  - n é o número de elementos
  - k é a base (radix)
- Geralmente, k próximo porém não maior que n

Radix Sort – Introdução 4/8

- Posição por posição é aplicado counting sort
  - A partir da posição menos significativa

3	0	2
2	5	3
6	1	1
9	0	1
5	2	9
1	0	2

Radix Sort – Introdução 5/8

- Posição por posição é aplicado counting sort
  - A partir da posição menos significativa

3	0	2
2	5	3
6	1	1
9	0	1
5	2	9
1	0	2

6	1	1
9	0	1
3	0	2
1	0	2
2	5	3
5	2	9

Radix Sort – Introdução 6/8

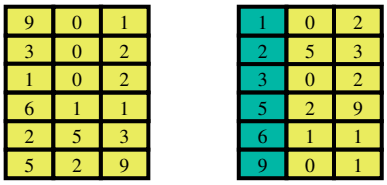
- Posição por posição é aplicado counting sort
  - A partir da posição menos significativa

6	1	1
9	0	1
3	0	2
1	0	2
2	5	3
5	2	9

9	0	1
3	0	2
1	0	2
6	1	1
2	5	3
5	2	9

Radix Sort – Introdução 7/8

- Posição por posição é aplicado counting sort
  - A partir da posição menos significativa



9	0	1
3	0	2
1	0	2
6	1	1
2	5	3
5	2	9

1	0	2
2	5	3
3	0	2
5	2	9
6	1	1
9	0	1

Radix Sort – Introdução 8/8

- Radix sort aplica counting sort uma vez
  para cada uma das p posições de digitos nos números
- Radix sort apresenta complexidade
  p vezes a complexidade do counting sort
  - ou O (pn + pk)
- Necessidade de espaço é a mesma do counting sort:
  - 2 arrays de tamanho n
  - 1 array de tamanho k

Radix Sort – Interface 1/1

- ***rxsort***
  - int rxsort(int *data, int size, int p, int k);
- **Retorno**
  - 0 se a ordenação teve sucesso, ou -1 caso contrário.
- **Descrição**
  - Usa radix sort para ordenar o array de inteiros em *data*. O número de inteiros em *data* é especificado por *size*. O argumento p especifica o número de posições de dígitos em cada inteiro. O argumento k especifica a base num érica.Quando *rxsort* retorna, *data* contem os inteiros ordenados.

Radix Sort – Implementação 1/3

```
#include <limits.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#include "sort.h"

int rxsort(int *data, int size, int p, int k) {

    int   *counts,   *temp;

    int   index,   pval,   i,   j,   n;

    if ((counts = (int *)malloc(k * sizeof(int))) == NULL)
        return -1;

    if ((temp = (int *)malloc(size * sizeof(int))) == NULL)
        return -1;
```


Radix Sort – Implementação 2/3

```
    //  Sort from the least significant position to the most significant.

    for (n = 0; n < p; n++) {
        // Initialize the counts

        for (i = 0; i < k; i++)
            counts[i] = 0;

        // Calculate the position value
        pval = (int)pow((double)k, (double)n);

        // Count the occurrences of each digit value
        for (j = 0; j < size; j++) {
            index = (int)(data[j] / pval) % k;
            counts[index] = counts[index] + 1;
        }
```


Radix Sort – Implementação 3/3

```
        //  Adjust each count to reflect the counts before it.

        for (i = 1; i < k; i++)
            counts[i] = counts[i] + counts[i - 1];

        //  Use the counts to position each element where it belongs.
        for (j = size - 1; j >= 0; j--) {

            index = (int)(data[j] / pval) % k;
            temp[counts[index] - 1] = data[j];
            counts[index] = counts[index] - 1;
        }
        memcpy(data, temp, size * sizeof(int));
    }

    free(counts);
    free(temp);
    return 0;
}
```

Ordenação – Sumário

- Insertion sort
  - Simples, ordena no lugar, melhor para ordenação incremental
- Quicksort
  - Ordena no lugar, melhor no caso geral
- Merge sort
  - Mesmo desempenho que quicksort, necessita o dobro da memória, permite trabalhar com a divisão em partes dos dados originais
- Counting sort
  - Algoritmo com tempo linear, trabalha com inteiros cujo maior valor é conhecido
- Radix sort
  - Algorimo com tempo linear, ordena dígito por dígito, bom para elementos de tamanho fixo que podem ser quebrados em pedaços, na forma de inteiros
- Existem ainda vários outros métodos