SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 7

Alunos	**NUSP**
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 7 – Radix Sort e Merge Sort

Item a

Comentário

Neste trabalho, foram implementados dois algoritmos de ordenação para organizar um baralho especial fornecido como entrada: o Radix Sort e o Stooge Sort. Logo de início, devido ao teste de ambos na plataforma do Run Codes, pudemos perceber que **a complexidade temporal do Stooge Sort é bem mais elevada, dado que não passou nos casos 6 e 7**, enquanto o Radix Sort não demonstrou problemas na eficiência de tempo.

Radix Sort

O Radix Sort é um algoritmo que realiza a técnica de subdivisão do problema em detrimento dos dígitos dos valores a serem ordenados, e, através da utilização do Counting Sort como subrotina ele não realiza comparações diretas entre os elementos. Note que qualquer algoritmo de ordenação estável poderia ser utilizado no lugar do Counting Sort, no entanto a sua utilização gera uma complexidade linear em relação a quantidade de elementos. Portanto, é necessário primeiramente entendermos o counting sort.

Counting Sort

O Counting Sort ordena um array com base em um dos dígitos de cada elemento, seu funcionamento ocorre em três etapas:

1) Criamos um vetor auxiliar que conta a quantidade de ocorrências (menos um) de cada valor, de forma que os valores ficam relacionados aos índices desse array.
2) Logo em seguida, fazemos a operação de soma do prefixo nesse vetor auxiliar, assim, dado um índice i, a primeira ocorrência do valor i possui sua posição já ordenada correspondente ao dado contido no índice i do vetor auxiliar.
3) Dessa forma, para construir o vetor ordenado só será necessário percorrer o vetor original checando a posição de inserção de cada elemento de acordo com o que consta no vetor de prefixo auxiliar, sempre lembrando de decrementar um ao dado acessado no vetor auxiliar, a fim de acomodar as demais ocorrências desse mesmo valor corretamente.

Note que, será necessário alocar espaço para o vetor auxiliar de prefixo e um segundo vetor de tamanho **n** (seja **n** a quantidade de elementos no vetor a ser ordenado) para guardar a ordenação final, o que leva a uma complexidade espacial de **O(n)**. Além disso, considerando o intervalo de valores contido em **n**, a complexidade de tempo será **O(n)**.

Complexidade Radix Sort

Com isso, dado que a complexidade da subrotina Counting Sort é **O(n)**, e essa subrotina será realizada **k** vezes (seja **k** a quantidade de dígitos do maior elemento) **a complexidade de tempo do algoritmo é linear**, sendo **O(n k)**.

Por isso, como a quantidade de naipes e valores é pequena, e o número de dígitos **k** é constante e pequeno para cada casos teste (de acordo com a especificações de entrada), o **Radix Sort é muito eficiente para a ordenação de cartas**, já que o tempo de execução é linear em relação ao número de cartas.

Stooge Sort

O stooge Sort é um algoritmo simples de ordenação que raramente é utilizado em casos práticos devido a sua alta complexidade de **$O(n^{2.7})$.** Ele funciona comparando o primeiro e último elemento do array e os trocando de posição caso necessário, é feita então a chamada recursiva para os **primeiros** ⅔ do array, dos **últimos** ⅔ e novamente para os **primeiros** ⅔ . O caso base, por sua vez, ocorre se o vetor possui tamanho igual ou inferior a 2, note que a verificação do caso base ocorre após a comparação e troca se necessário entre o início e o fim.

Código

Para organizar e tratar os dados, armazenamos-os em um vetor de strings, em que a posicao i do vetor corresponde a carta de numero n-1.

Correspondência:

- Valores: 4 = '0' / 5 = '1' / 6 = '2' / 7 = '3' / Q = '4'
  J = '5' / K = '6' / A = '7' / 2 = '8' / 3 = '9'
- Naipe: ♦ = '0' / ♠ = '1' / ♥ = '2' / ♣ = '3

Vetor de strings: (matriz)

- string[0] = naipe (baralho[i][0])
- string[1 -> tam] = valor (baralho[i][1 -> tam])

Exemplo:

- entrada: ♦ 5K7 = string[] = {'0','1','8','3'}

Funções utilizadas

```
char **input(int n, int tam);  //funções utilizadas nas duas implementações
char converte_valor(char valor);
char converte_naipe(char naipe[5]);
char **aloca_matriz(int l, int c);
void libera_matriz(char ***matriz, int l);
void printa_baralho(char **baralho, int n, int tam);

void counting_sort(char **baralho, char **ordenado, int n, int digito);
char **radix_sort(char **baralho, int n ,int tam); //Funções utilizadas para
o Radix Sort

void stooge_sort(char **baralho, int inicio, int fim);
void swap(char **a, char **b); //Funções utilizadas para o Stooge Sort
```

Principais funções de tratamento dos dados:

```
char **input(int n, int tam) {
   char **baralho = aloca_matriz(n, tam +2); // +2 para o naipe e o '\0'

   char aux1; char aux2[5];
   for (int i = 0; i<n; i++) { //percorre todos as cartas
      scanf(" %s", aux2);
      //insere naipe na forma '0'/'1'/'2'/'3' no id 0
      baralho[i][0] = converte_naipe(aux2);

      //percorre todos os dígitos de cada valor
      for(int j = 1 ; j <= tam; j++) {
         scanf(" %c", &aux1);
         //recebe valores da forma '0'/'1'/'2'/'3'/'4'/'5'/'6'/'7'/'8'/'9'
         // no id j (entre 1 e tam)
         baralho[i][j] = converte_valor(aux1);        }

       baralho[i][tam+1] = '\0'; //fim da string
   }

   return baralho; }
```

```
void printa_baralho(char **baralho, int n, int tam){
   char valores[] = "4567QJKA23"; //matriz de correspondencia

   for (int i = 0; i<n; i++) {
      int naipe_id = (baralho[i][0] - '0') ; //id do naipe no switch case
      switch (naipe_id) { //printa naipe
           case(0): printf("♦ "); break;
           case(1): printf("♠ "); break;
           case(2): printf("♥ "); break;
           case(3): printf("♣ "); break;
      }
      //percorre todos os digitos do valor de cada carta
      for (int j = 1; j<= tam; j++) {
         //id do valor no vetor de  correspondencia
         int valor_id = baralho[i][j] - '0';

         //printa o digito j do valor da  carta i
         printf("%c", valores[valor_id]);
      }
      printf(";");
   }
   printf("\n"); }
```

Radix Sort

```
char **radix_sort(char **baralho, int n ,int tam) {
   char **ordenado = aloca_matriz(n, tam+2); // +2 para o naipe e o '\0'
   printa_baralho( baralho, n, tam); //printa baralho antes da ordenacao

   for(int i = tam; i > 0; i--) { //ordena valores
       counting_sort(baralho, ordenado, n, i);

       printf("Após ordenar o %d° dígito dos valores:\n", i);
       printa_baralho(baralho, n, tam);
   }
   counting_sort(baralho, ordenado, n, 0); //ordena naipe

   printf("Após ordenar por naipe:\n");
   printa_baralho(baralho, n, tam);

   return ordenado; }
```

```
void counting_sort(char **baralho, char **ordenado, int n, int digito) {
   int prefix[10] = {0,0,0,0,0,0,0,0,0,0},
   valor, insercao;

   for (int i = 0; i<n; i++) { //determina quantidade de cada valor
      valor = baralho[i][digito] - '0';
      //insere no vetor auxiliar a quantidade de cada valor
      prefix[valor]++;
   }
   for (int i = 1; i<=9; i++) { //seta o prefix
      prefix[i] += prefix[i-1];
   }
   for (int i = n -1; i>= 0; i--) {
      valor = baralho[i][digito] - '0'; //pega id do valor no prefix
      /*determina onde o valor será inserido e decrementa
      o prefix[valor] para a proxima insercao*/
      insercao = --prefix[valor];
      //insere o valor no vetor ordenado
      strcpy(ordenado[insercao], baralho[i]);
   }
   //copia a matriz ordenada para matriz original (baralho)
   for (int i =0; i<n; i++) {
      strcpy(baralho[i], ordenado[i]);
   }
}
```

Stooge Sort

```c
void stooge_sort(char **baralho, int inicio, int fim) { //ordena o baralho in
place

   //compara primeiro e ultimo elemento
   if (strcmp(baralho[inicio], baralho[fim]) > 0)
      swap(&baralho[inicio], &baralho[fim]); //faz a troca caso necessario

   if (inicio + 1 >= fim) return; //caso base

   //+1 garante que o arredondamento de tam/3 seja para cima
   int tam = fim - inicio +1;
   int novo_fim = fim - tam/3, novo_inicio = inicio + tam/3;

   stooge_sort(baralho, inicio,novo_fim); // primeiros 2/3
   stooge_sort(baralho, novo_inicio, fim); // ultimos 2/3
   stooge_sort(baralho, inicio, novo_fim); //primeiros 2/3
}
```

□ Saída

Para comprovar a eficiência dos dois métodos de ordenação, foi utilizada a biblioteca <time.h> da linguagem C para medir o tempo de execução dos algoritmos. Como as saídas são muito grandes, serão apresentados apenas os tempos de execução de cada algoritmo, com casos de teste do RunCodes

Tempos de Execução

Seja **n** a quantidade de cartas e **k** o tamanho dos valores de cada carta

	Caso 2: **n = 10** **k = 10**	**Caso 3:** **n = 1000** **k = 3**	**Caso 6:** **n = 10000** **k = 12**	**Caso 7:** **n = 100000** **k = 3**
Radix Sort	0.000125	0.001236	0.026651	0.046511
Stooge Sort	0.000009	0.252638	**177.583994**	**Muito maior a 1 minuto**

Algoritmo mais rápido

Caso 2: **n = 10** **k = 10**	**Caso 3:** **n = 1000** **k = 3**	**Caso 6:** **n = 10000** **k = 12**	**Caso 7:** **n = 100000** **k = 3**
Stooge Sort	Radix Sort	Radix Sort	Radix Sort

A partir dos testes, percebe-se que o Radix é um algoritmo muito mais eficiente do que o Stooge no que tange tempo de execução, e apresentou eficiência mais rápida para os casos testados com um número de cartas elevado, sendo excepcionalmente bom para ordenar o conjunto de dados (cartas) da maneira que foi fornecido, **essa constatação anteriormente explicada na discussão de complexidade do Radix Sort ficou clara ao passo que a quantidade de cartas teve impacto linear no tempo de processamento**, **sendo um fator importante o tamanho dos valores fornecidos (complexidade 0(nk)).** Vale ressaltar que apesar da grande eficiência temporal do Radix Sort em relação ao Stooge Sort, o algoritmo apresenta como desvantagem o uso de espaço extra **O(n)**, dado que o algoritmo requer armazenamento adicional para realizar as operações, ao contrário do Stooge Sort que opera in place.
