Notas de Aula - AED1 – Algoritmos de Ordenação (parte 4)
Prof. Jefferson T. Oliva

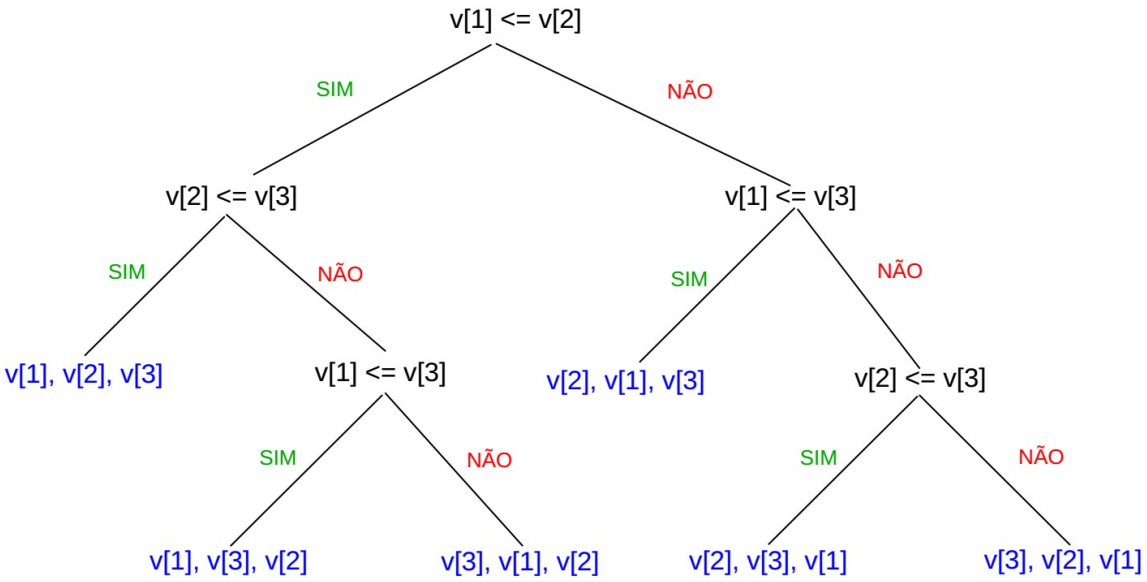
Neste material é descrito um algoritmo de ordenação de tempo linear e aplicação de alguns algoritmos na ordenação em strings e em TADs

Ordenação Linear

Até a última aula foram apresentados diversos algoritmos de ordenação, dos quais, os mais rápidos têm a complexidade de O(n log n), como heapsort (pior caso), mergesort (pior caso) e quicksort (caso médio).

Os algoritmos vistos possuem uma propriedade em comum: a ordenação se baseia apenas em comparações entre os elementos da entrada. Dessa forma, esses algoritmos são denominados de ordenação por comparação, para o qual, não existe método assintoticamente melhor do que O(n log n).

O algoritmo baseado em comparação é uma "árvores de decisão".



Existem algoritmos de ordenação que possuem tempo melhor que O(n log n), desde que:
- Desde que a entrada possua características específicas.
- Algumas restrições devem ser atendidas.
- Não sejam totalmente baseado em comparações.
- Atendendo os requisitos acima, podemos ter métodos de ordenação em tempo linear: O(n).

Exemplo de algoritmo de ordenação linear: *counting sort*
- Não são realizadas comparações entre os elementos.
- É assumido que cada elemento é um número inteiro entre 0 e *k*.
- A entrada é um arranjo *A[1..n]*, cujos elementos são números naturais menores iguais a *k*.
- A saída é um arranjo ordenado: *B[1..n]*.
-- É utilizado para determinar a quantidade de elementos que são menores que um determinado valor *x*.
-- Por exemplo, se há 14 elementos menores que x, então o mesmo deve ser posicionado na posição 15 (ou 14, caso o algoritmo tenha sido implementado em C ou outra linguagem onde o primeiro elemento de arranjo fica localizado na posição 0).

```
void counting_sort(int *A, int *B, int n, int k){
   int i, j, C[k + 1];

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

   for (j = 0; j < n; j++)
      C[A[j]]++;

   // Para cada i = {0, 1, ..., k}, aqui é determinado
   // quantos elementos são menores ou iguais a "i"
   for (i = 1; i <= k; i++)
      C[i] += C[i - 1];

   for (j = n - 1; j >= 0; j--){
      B[C[A[j]] - 1] = A[j];
      C[A[j]]--;
   }
}
```

Complexidade:
- *O(n + k)*, ou
- *O(max(n, k))*, ou
- *O(n)*, caso *k* seja menor que *n*

Ordenação em Strings

Esse tipo de ordenação é em ordem alfabética.

Comparação de strings é feita caractere por caractere.

Para diferenciar dois números, basta realizar uma única comparação.

Para diferenciar duas strings, a quantidade de comparações é de acordo com o número de caracteres.

Na biblioteca string.h contém a função strcmp para a comparação de cadeias de caracteres:
- Entrada: duas strings (str1 e str2)
- Saída:
→ -1: conteúdo de str1 menor do que str2
→ 0: ambas strings são iguais
→ 1: conteúdo de str1 maior do que str2

Exemplos
- strcmp("goku", "vegeta") = -1
- strcmp("sheena", "sheena") = 0
- strcmp("shaka", "aldebaran") = 1

Implementação do método de comparação de strings:

```
int comparar(char s1[], char s2[]){
	int i, t;

	if (strlen(s1) < strlen(s2))
		t = strlen(s1);
	else
		t = strlen(s2);

	for (i = 0; (i < t) && (s1[i] == s2[i]); i++);

	if (i == t){
		if (strlen(s1) < strlen(s2)) return -1;
		else if (strlen(s1) > strlen(s2)) return 1;
		else return 0;
	}else if (s1[i] < s2[i]) return -1;
	else return 1;
}
```

Outra versão:

```
int comparar_char(char c1, char c2){
	if (c1 == c2) return 0;
	else if (c1 < ce2) return -1;
	else return 1;
}


int comparar(char s1[], char s2[]){
	int i;

	for (i = 0; (s1[i] == s2[i]) && (s1[i] != '\0') && (s2[i] != '\0'); i++);

	comparar_char(s1[i], s2[i]);
}
```

Complexidade?
O(n), onde *n* é comprimento da string.

Vetor de strings
- char vstr[n][m];
- char *vstr[m];
- char **vstr;

Exemplo para vetor de string

```
char vstr[5][10];
strcpy(vstr[0], "a");
strcpy(vstr[1], "bb");
strcpy(vstr[2], "ccc");
strcpy(vstr[3], "ddd");
strcpy(vstr[4], "eeee");
printf("%s\n", vstr[0]);
printf("%s\n", vstr[1]);
printf("%s\n", vstr[2]);
printf("%s\n", vstr[3]);
printf("%s\n", vstr[4]);
```

- Troca de posição entre duas strings (linhas 0 e 4):

```
strcpy(str, vstr[0]);
strcpy(vstr[0], vstr[4]);
strcpy(vstr[4], str);
```

A ordenação de strings é custosa
-Além da quantidade de comparações entre um par de strings, na abordagem apresentada acima ainda teremos que lidar com 3 cópias.
- Assim, serão realizadas 4 * l operações em cada movimentação entre strings, onde l é o tamanho de uma string: l comparações + 3*l cópias

Como amenizar esse problema?

O problema apresentado no slide anterior pode ser amenizado por meio do uso de ponteiros:

```
void troca(char **vstr, int p1, int p2){
    char *str;

    str = vstr[p1];
    vstr[p1] = vstr[p2];
    vstr[p2] = str;
}
```

Para o uso dessa função (ou aplicação de ponteiros), o vetor de strings deve estar alocado dinamicamente

```
char **vstr = malloc(sizeof(char*) * 5);
char str[20];

for (i = 0; i < 5; i++)
    vstr[i] = malloc(sizeof(char) * 10);
```

Com o uso de ponteiros, a quantidade de movimentações para uma troca de posição entre strings passa a ser l + 3
- Por mais que ainda seja necessária a comparação entre strings (complexidade na ordem de l caracteres), a movimentação para a troca de posições de strings passa a ser similar em relação aos dados do tipo numérico.
- É muito menos custoso realizar l + 3 operações em comparação com 4l

Ordenação por bolha (bubblesort) adaptada para vetores de strings

```
void bubblesort(char **vstr, int n){
    int i, j, x, t = 1;

    for (i = 0; (i < n - 1) && t; i++){
        t = 0;

        for (j = 0; j < n - i - 1; j++)
            if (comparar(vstr[j], vstr[j + 1]) > 0){
                troca(vstr, j, j + 1);
                t = 1;
            }
    }
}
```

Complexidade: O(l * n^2)

Algoritmo quicksort adaptado para vetores de strings

```
void quicksort(int **vstr, int n_cima, int n_baixo){
    int i = n_cima, j = n_baixo, aux;
    char *pivo = vstr[(i + j) / 2];

    do{
        while ((comparar(vstr[i], pivo) < 0) && (i < j))
            i++;

        while ((comparar(vstr[j], pivo) > 0) && (j > i))
        j--;

        if (i <= j){
            troca(vstr, i, j);
            i++;
            j--;
        }
    }while (i <= j);




if (j > n_cima)
    quicksort(vstr, n_cima, j);
if (i < dir)
    quicksort(vstr, i, n_baixo);
}
```

Complexidade: O(l * n * log(n))

A ordenação de strings pode ser feita utilizando qualquer um dos métodos que vimos até o momento em sala de aula. Entretanto, cada algoritmo será mais custoso.

Ordenação em TAD

A ordenação pode ser aplicada em structs também.

Assim, podemos aplicar tanto algoritmos de ordenação de números quanto de strings.

As structs podem ter chaves de identificação (e.g. registro acadêmico).

Uma struct pode ter chave secundária (e.g. código do curso de graduação).

Assim, há diversas formas diferentes de ordenação que podem ser aplicadas em structs
- Listar nome de alunos ordenados por nome
- Listar nome de alunos ordenados por registro acadêmico
- Etc

Assim como para números, não há algoritmo de ordenação que se destaca em relação aos outros para a ordenação de strings e structs.

Existem inúmeros algoritmos de ordenação que não foram apresentados em sala de aula
- Radix sort
- Bucket sort
- Shake sort
- Twistsort
- etc

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007