CK0226 - Programação
Professor Wladimir A. Tavares
Programação em C

O objetivo da aula é aprender:

1. Noções básica de comparações de algoritmo (Encontrar o Maior Elemento)
2. a usar o comando while e if/else (busca Linear), (Jump Search passo 2) e (Jump Search passo $\sqrt{n}$).
3. Comportamento linear
4. Comportamento raiz quadrada.
5. Comportamento logarítmico.
6. Desenvolvimento de funções `colocar_maior_ultimo`.
7. Usar funções para resolver outros problemas (Ordena Colocando o maior por último)

Encontrando o maior elemento

Quando um algoritmo é bom? Quando um algoritmo é ruim?

```
int maior_elemento1(int L[], int n){
    for(int i = 0; i < n; i++){
        int maior_todos = 1;
        for(int j = 0; j < n; j++){
            if(L[j] > L[i]){
                maior_todos = 0;
                break;
            }
        }
        if(maior_todos == 1){
            return L[i];
        }
    }
}
```

```
int maior_elemento2(int L[], int n){
    int M = L[0];
    for(int i = 1; i < n; i++){
        if(L[i] > M){
            M = L[i];
        }
    }
    return M;
}

```

As duas funções acima encontram o maior elemento numa lista de números. No caso acima, o algoritmo `maior_elemento2` é melhor que `maior_elemento1` (Por quê?).

Podemos tentar olhar para o número de comparações para tentar comparar os dois algoritmos. O algoritmo `maior_elemento2` realiza n^2 comparações e `maior_elemento1` realiza $n-1$ comparações.

Busca Linear

Nesse problema, recebemos um lista de inteiros L ordenada em ordem crescente e um inteiro k, queremos saber se o inteiro k é um elemento da lista de inteiros L.

O algoritmo de busca linear consiste em avançar a variável i em uma unidade até encontrar um elemento $L[i] \geq k$ ou $i = n$. Neste caso, podemos analisar os seguintes casos:

1. $L[i] == k$, o elemento está presente na lista.
2. $L[i] > k$, o elemento não está presente na lista.
3. $i == n$, o elemento não está presente na lista.

```c
#include <stdio.h>
#define N 10
int L[N] = {2,5,6,9,11,15,18,20,21,25};
int k = 14;

int main(){
    int i;
    i = 0;
    while(i<N && L[i] < k) {
        printf("%d < %d\n", L[i], k);
        i++;
    }

    if(i==N) printf("nao achei\n");
    else if(L[i]==k) printf("achei\n");
    else printf("nao achei\n");
    /*
    if(i==N || L[i]>k)
        printf("nao achei");
    else
        printf("achei");
    */
}
```

O programa acima realiza aproximadamente n comparações. Neste caso, dizemos que o crescimento do tempo de execução tem um comportamento linear.

Jump Search passo 2

Nesse problema, recebemos um lista de inteiros L ordenada em ordem crescente e um inteiro k, queremos saber se o inteiro k é um elemento da lista de inteiros L.

O algoritmo de busca linear consiste em avançar a variável i em duas unidade até encontrar um elemento $L[i] \geq k$ ou $i \geq nn$. Neste caso, podemos analisar os seguintes casos:

1. $i == n + 1$, o último elemento foi testado e o elemento k não está presente.
2. $i == n$, o último elemento não foi testado.
3. $i < n$ and $L[i] \geq k$, precisamos testar $L[i] == k$ e $L[i - 1] == k$

```c
#include <stdio.h>
#define N 10
int L[N] = {2,5,6,9,11,15,18,20,21,25};
int k = 25;

int main(){
    int i;
    i = 0;
    while(i<N && L[i] < k) {
        printf("i %d : %d < %d\n", i, L[i], k);
        i = i + 2;
    }
    printf("i = %d\n", i);
    /*
    //testou o ultimo
    if(i == N+1)
        printf("nao achei\n");
    else if(i == N) { //pulou o ultimo
        if(L[i-1] == k) printf("achei\n");
        else printf("nao achei");
    }else{ // i < N
        if(L[i] == k) printf("achei");
        else if(i > 0 && L[i-1] == k)
                printf("achei\n");
        else printf("nao achei");
    }
    */
    if(i>=N){
        if(i-1<N && L[i-1]==k) printf("achei");
        else printf("nao achei\n");
    }else{
        if(L[i]==k || (i > 0 && L[i-1] == k))
            printf("achei\n");
        else
            printf("nao achei\n");
    }

}
```

O programa acima realiza aproximadamente $\frac{n}{2}$ comparações. Neste caso, dizemos que o crescimento do tempo de execução tem um comportamento linear.

Jump Search passo $\sqrt{n}$

No algoritmo Jump Search, o número de comparações realizada será aproximadamente ao número de blocos + tamanho do bloco. Podemos tentar encontrar o melhor tamanho de bloco, analisando os seguintes casos para N = 10:

pulo	número de blocos	tamanho do bloco
2	$\frac{n}{2}$	2
3	$\frac{n}{3}$	3

Quando escolhemos o tamanho do pulo = $\sqrt{n}$ temos que o número de blocos e o tamanho do bloco tem quase o mesmo tamanho.

```c
#include <stdio.h>
#include <math.h>
#define N 9
int L[N] = {2,5,6,9,11,15,18,20,21};
int k = 11;
int pulo = sqrt(N);
int main(){
    int i;
    i = 0;
    while(i<N && L[i] < k) {
        printf("i %d : %d < %d\n", i, L[i], k);
        i = i + pulo;
    }
    printf("i = %d\n", i);
    if(i >= N){
        i = N-1;
        while(i >= 0 && L[i] > k) {
            printf("i %d : %d > %d\n", i, L[i], k);
            i = i-1;
        }
        if( i < 0 || L[i] < k) printf("nao achei\n");
        else printf("achei");
    }else{
        while(i >= 0 && L[i] > k) {
            printf("i %d : %d > %d\n", i, L[i], k);
            i--;
        }
        if( i < 0 || L[i] < k) printf("nao achei\n");
        else printf("achei\n");
    }
}
```

Note que para encontrar o valor 20, o programa vai imprimir o seguinte relatório:

```
i 0 : 2 < 20
i 3 : 9 < 20
i 6 : 18 < 20
i = 9
i 8 : 21 > 20
achei
```

Note a variável i pula até 3 blocos e sai do lista chegando ao valor $i = 9$. Neste caso, ele precisa analisar o último bloco da esquerda para direita pulando sequencialmente os números maiores que o número buscado. No caso do número 20, o laço termina com a variável i apontando para o número 20.

Busca Binária

Na busca binária, precisamos jogar fora sempre a metade do problema. Quando escolhemos um elemento na posição do meio, pode acontecer três coisas:

- se $L[meio] == k$, então a busca pode ser interrompida.
- se $L[meio] > k$ então sabemos que o elemento buscado está na primeira metade ($L[inicio..meio-1]$).
- se $L[meio] < k$ então sabemos que o elemento buscado está na segunda metade ($L[meio+1..fim]$).

```c
#include <stdio.h>
#include <math.h>
#define N 9
int L[N] = {2,5,6,9,11,15,18,20,21};
int k = 20;
int main(){
    int inicio, fim, meio, encontrei;
    inicio = 0;
    fim = N-1;
    encontrei = 0;
    while(inicio<fim){
        printf("L[%d...%d]\n", inicio, fim);
        meio = (inicio+fim)/2;
        if(L[meio] == k){
            encontrei = 1;
            break;
        }else if(L[meio] > k){
            fim = meio-1;
        }else{
            inicio = meio+1;
        }
    }
}
```

Trocar o maior elemento com o último

Podemos resolver o problema em duas etapas:

1. Encontrar a posição do maior.
2. Trocar o maior com a última posição

A ideia do algoritmo é utilizar a variável M para guardar o maior encontrado visto até agora e $posM$ guarda a posição do maior encontrado visto até agora. Ao final do primeiro laço, encontramos o maior elemento e a sua posição. Em seguida, realizamos a troca desse elemento com o último.

```c
#include <stdio.h>
#include "prog.h"
int main(){
    int L[6] = {7,9,1,2,3,5};
    int i, posM, M, n, aux;
    n = 6;
    //Encontrar a posição do maior
    M = L[0];
    posM = 0;
    for(i = 0; i < 6; i++){
        if(L[i] > M){
            M = L[i];
            posM = i;
        }
    }
    //Trocar o maior com o último
    aux = L[posM];
    L[posM] = L[n-1];
    L[n-1] = aux;

    imprime_lista(L, 6);
}
```

Coloca_maior_ultimo

Imagine que queremos realizar essa tarefa muitas vezes para diferentes faixas da listas. A ideia agora é definir uma função para isso precisamos definir um nome e quais são os parâmetros de entrada para a função.

```c
#include <stdio.h>
#define N 6
int L[6] = {7,9,1,2,3,5};

void coloca_maior_ultimo(int L[], int n){
    int i, M, posM, aux;

    M = L[0];
    posM = 0;
    for(i = 0; i < n; i++){
        if(L[i] > M){
            M = L[i];
            posM = i;
        }
    }
    //Trocar o maior com o último
    aux = L[posM];
    L[posM] = L[n-1];
    L[n-1] = aux;

}


int main(){

                            //[7,9,1,2,3,5]
coloca_maior_ultimo(L, 6); //[7,5,1,2,3|,9]
coloca_maior_ultimo(L, 5); //[3,5,1,2|,7,9]
coloca_maior_ultimo(L, 4); //[3,2,1|,5,7,9]
coloca_maior_ultimo(L, 3); //[1,2|,3,5,7,9]
coloca_maior_ultimo(L, 2); //[1,|2,3,5,7,9]


}
```

O número de instruções realizadas pela função `colocar_maior_ultimo` é proporcional a n. Logo, dizemos que a função tem comportamento linear com relação ao tempo de execução.

Ordena colocando o maior por último

A ideia do método de ordenação de colocar o maior na última posição consiste em cada passo do laço, o maior é colocado na última posição do vetor de tamanho i. O problema pode ser reduzido em 1 unidade. Note que o processo continua até que o vetor fique o com o tamanho igual a 1.

```c
#include <stdio.h>
#define N 6
int L[6] = {7,9,1,2,3,5};

void coloca_maior_ultimo(int L[], int n){
    int i, M, posM, aux;

    M = L[0];
    posM = 0;
    for(i = 0; i < n; i++){
        if(L[i] > M){
            M = L[i];
            posM = i;
        }
    }
    //Trocar o maior com o último
    aux = L[posM];
    L[posM] = L[n-1];
    L[n-1] = aux;

}


int main(){

    int i;
    for(i = N; i > 1; i++){
        coloca_maior_ultimo(L, i);
    }

}
```

O número de instruções realizadas pelo programa acima é proporcional a n^2. Logo, dizemos que o programa tem comportamento quadrático com relação ao tempo de execução.