

Aula 04

Algoritmos e Programação II

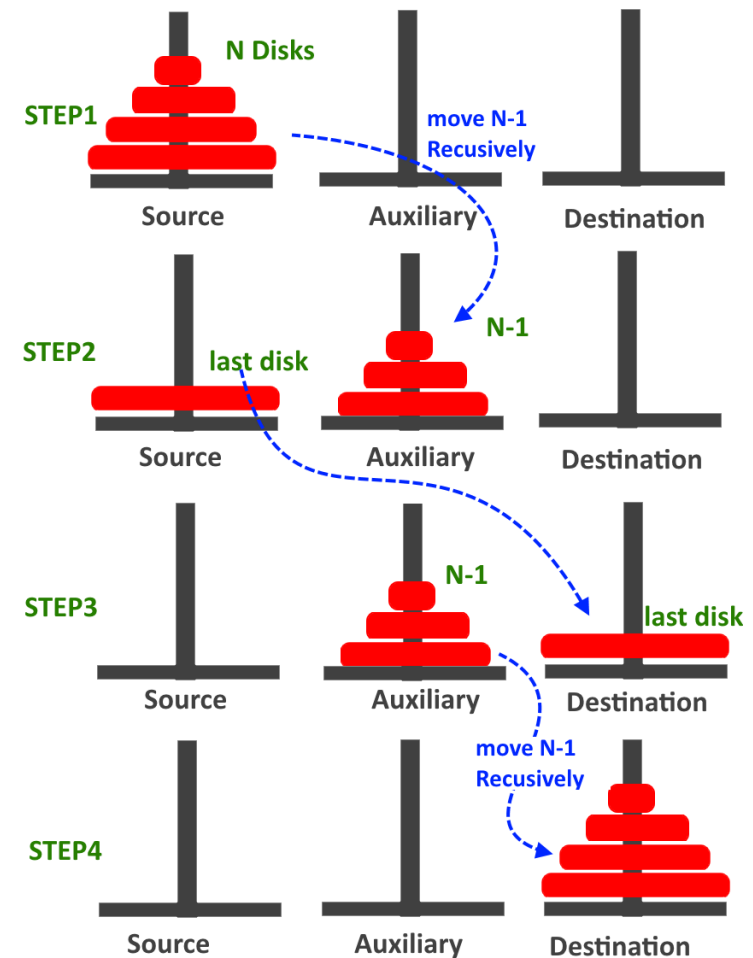
Setembro / 2017



Prof. Mario Liziér
lazier@ufscar.br

Exemplo - Hanoi

- Faça uma rotina resolver o problema da torre de Hanoi com n discos

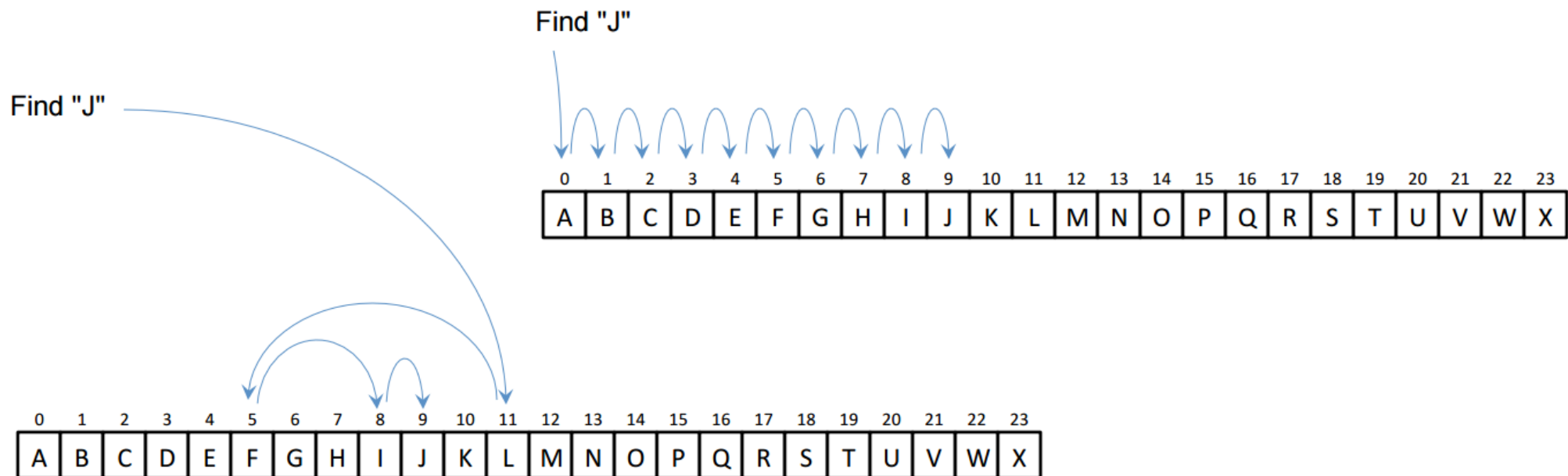


Exemplo – Torre de Hanoi

```
void hanoi (int discos, char origem, char destino, char auxiliar) {  
    if (discos==1)  
        printf("\t Mova o disco %d de %c para %c \n", discos, origem, destino);  
    else {  
        hanoi(discos-1, origem, auxiliar, destino);  
        printf("\t Mova o disco %d de %c para %c \n", discos, origem, destino);  
        hanoi(discos-1,auxiliar,destino,origem);  
    }  
}
```

Exemplo – Busca binária

- Faça uma rotina recursiva para procurar um determinado valor em um vetor de inteiros
 - Receba o tamanho do vetor
 - Receba os elementos do vetor (ou gere aleatoriamente e os imprima na tela)
 - Receba o valor a ser procurado
 - Imprima se o determinado valor está ou não presente no vetor



Exemplo – Busca binária

- Versão iterativa:

```
int binarySearch(int vet[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r-l)/2;

        if (vet[m] == x)
            return m;

        if (vet[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }

    return -1;
}
```

Exemplo – Busca binária

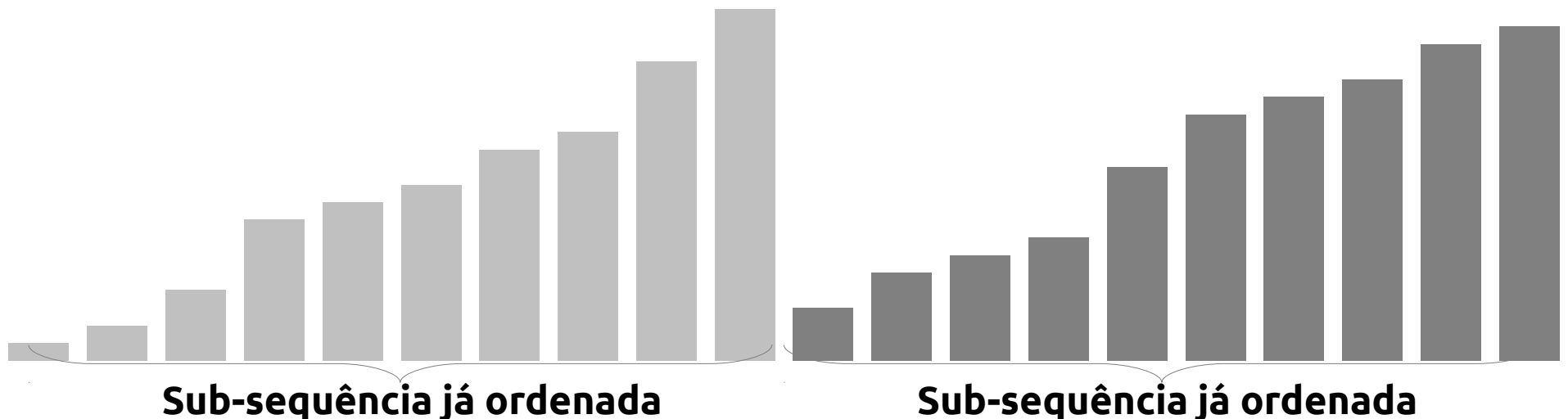
- Versão recursiva:

```
int binarySearch(int vet[], int l, int r, int x)
{
    if (r >= l)
    {
        int mid = l + (r - l)/2;

        if (vet[mid] == x)
            return mid;
        else if (vet[mid] > x)
            return binarySearch(vet, l, mid-1, x);
        else
            return binarySearch(vet, mid+1, r, x);
    }
    return -1;
}
```

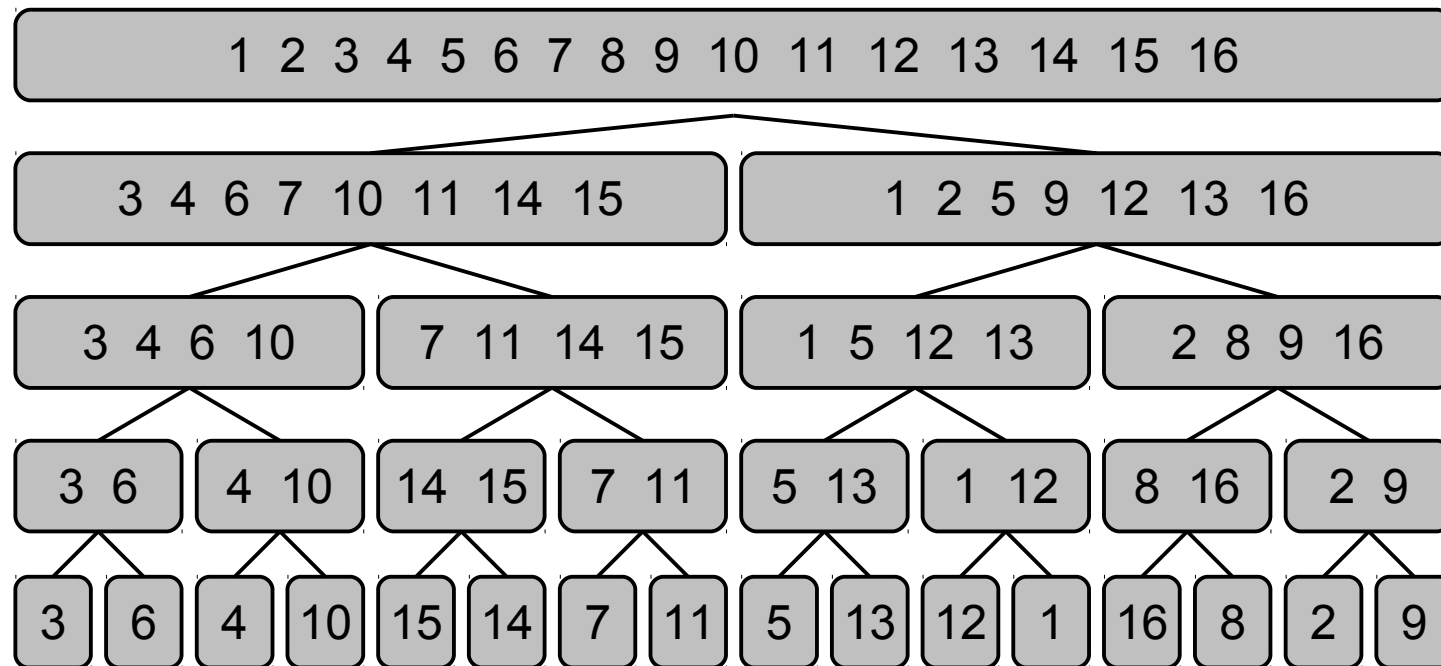
Exemplo - Merge Sort

- Dividir para conquistar
- Divisão da sequência em partes menores para facilitar a ordenação
- União de sequências menores já ordenadas, gerando sequências maiores ordenadas



Merge Sort

- Árvore de divisão



- Abordagens de implementação:
 - Top-Down - recursiva
 - Bottom-Up - iterativa

Procedimento Merge

```
void merge(Item vetor[], Item aux[], int imin, int imid, int imax)
{
    int i = imin, j = imid+1;

    for (int k = imin; k <= imax; k++)
        aux[k] = vetor[k];

    for (int k = imin; k <= imax; k++)
        if (i > imid)
            vetor[k] = aux[j++];
        else if(j > imax)
            vetor[k] = aux[i++];
        else if(aux[j] < aux[i])
            vetor[k] = aux[j++];
        else
            vetor[k] = aux[i++];
}
```

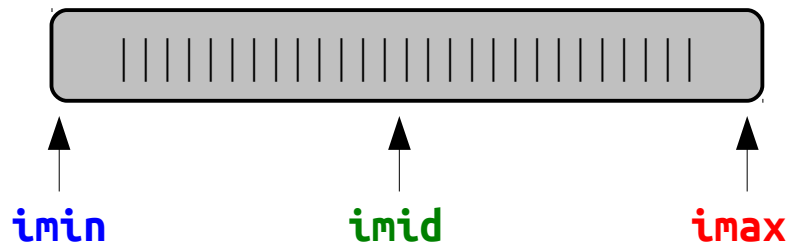
Merge Sort / Recursivo / Top-Down

```
void mergesort(Item vetor[], Item aux[], int imin, int imax)
{
    if (imax <= imin)
        return;

    int imid = imin + ((imax - imin) / 2);

    mergesort(vetor, aux, imin, imid);
    mergesort(vetor, aux, imid+1, imax);

    merge(vetor, aux, imin, imid, imax);
}
```



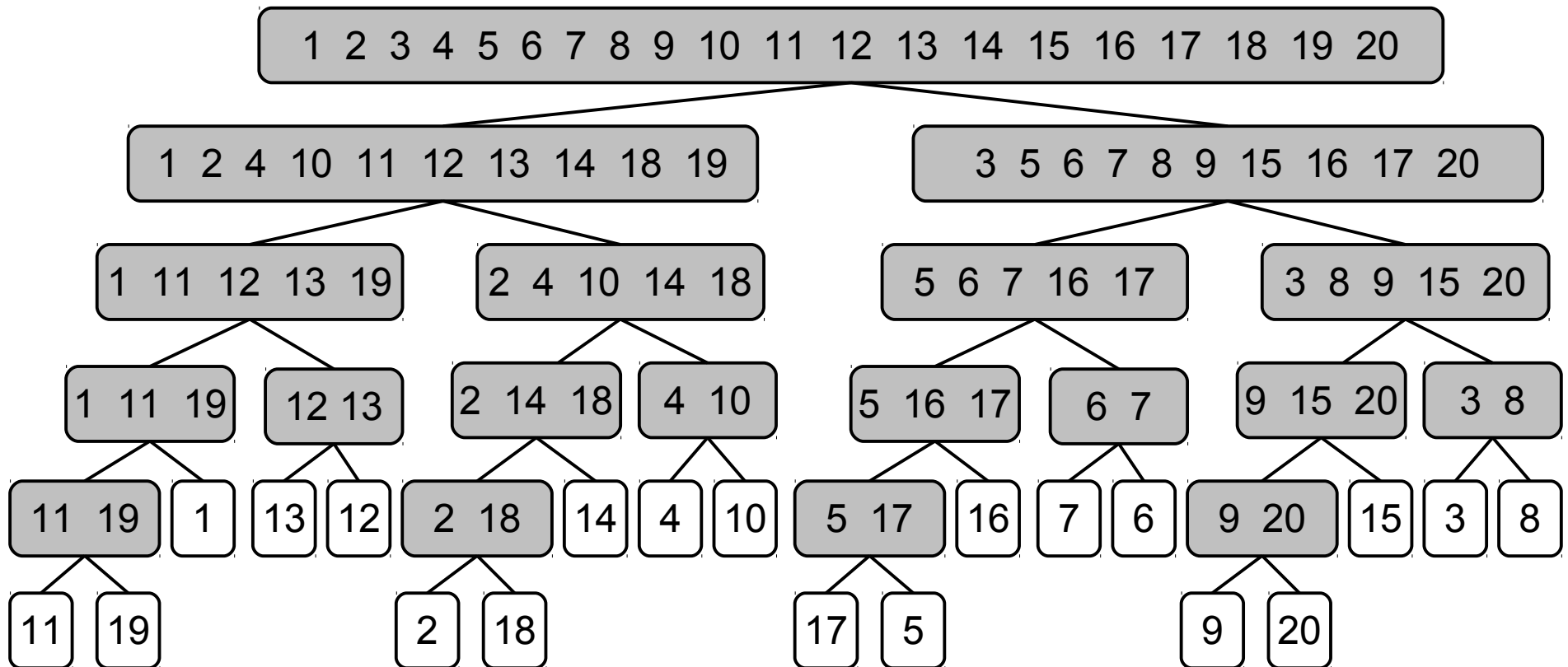
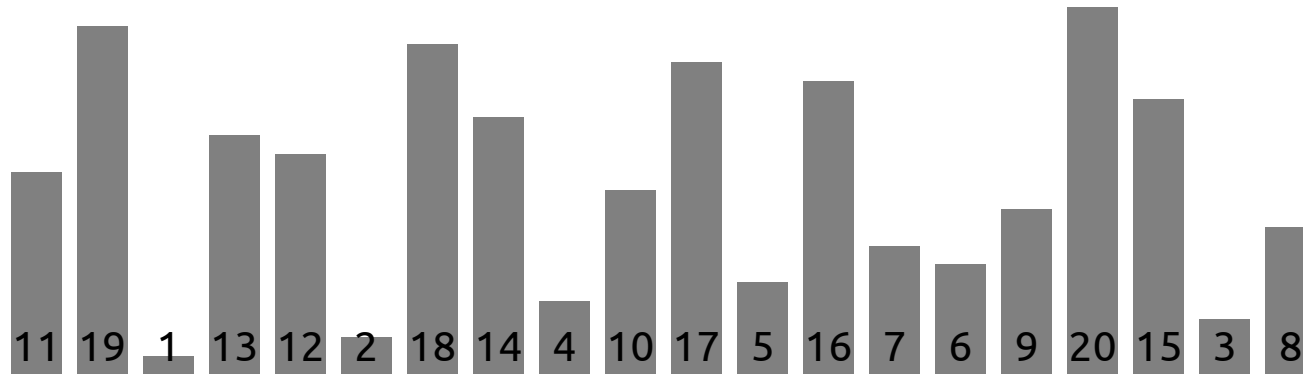
```
void merge(Item vetor[], Item aux[], int imin, int imid, int imax)
{
    int i = imin, j = imid+1;

    for (int k = imin; k <= imax; k++)
        aux[k] = vetor[k];

    for (int k = imin; k <= imax; k++)
        if (i > imid)
            vetor[k] = aux[j++];
        else if (j > imax)
            vetor[k] = aux[i++];
        else if (aux[j] < aux[i])
            vetor[k] = aux[j++];
        else
            vetor[k] = aux[i++];
}
```

MergeSort Top-Down


Entrada:



Merge Sort / Iterativo / Bottom-Up

```
inline int min(int A, int B)
{
    return (A < B) ? A : B;
}
```

```
void mergesortBU(Item vetor[], int imin, int imax)
{
    Item aux[imax+1];
    for (int m = 1; m <= imax-imin; m *= 2)
        for (int i = imin; i <= imax-m; i += 2*m)
            merge(vetor, aux, i, i+m-1, min(i+m+m-1, imax));
}
```



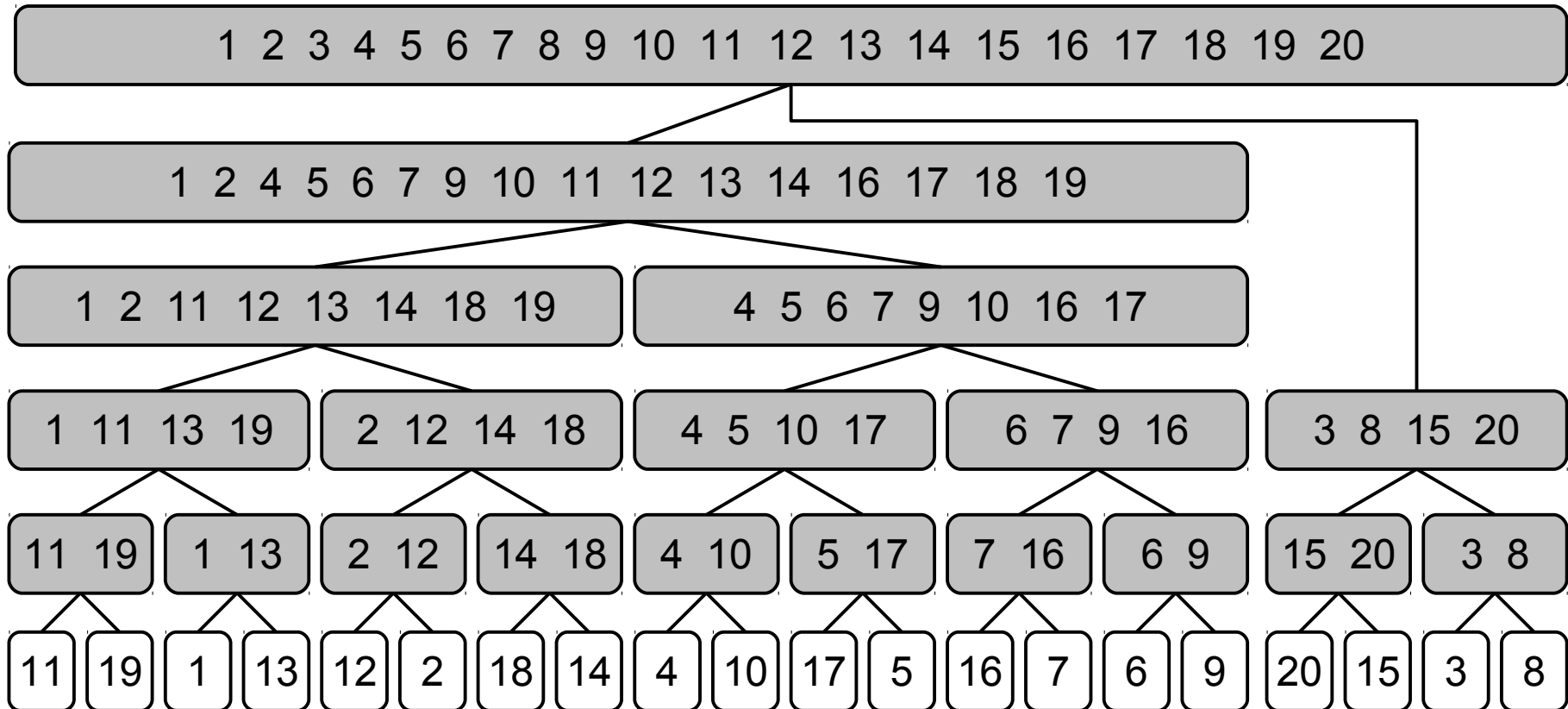
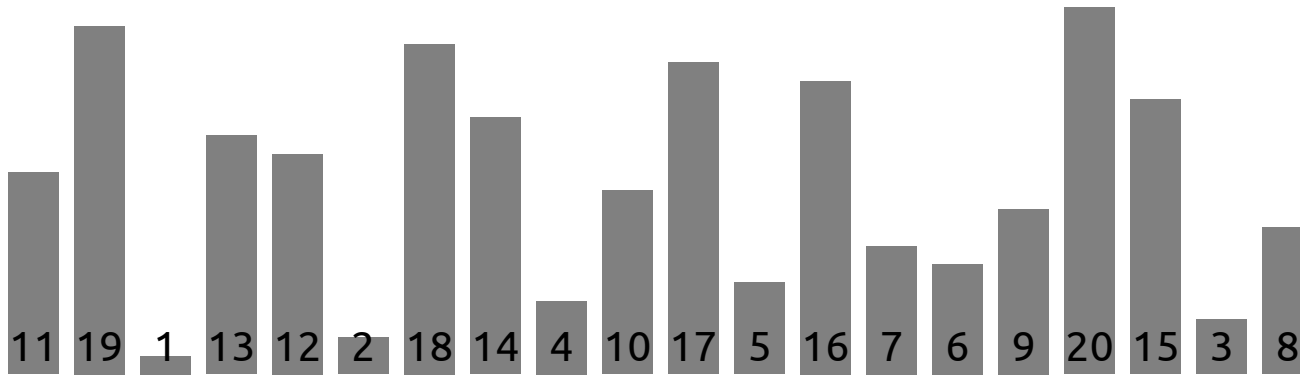
```
void merge(Item vetor[], Item aux[], int imin, int imid, int imax)
{
    int i = imin, j = imid+1;

    for (int k = imin; k <= imax; k++)
        aux[k] = vetor[k];

    for (int k = imin; k <= imax; k++)
        if (i > imid)
            vetor[k] = aux[j++];
        else if (j > imax)
            vetor[k] = aux[i++];
        else if (aux[j] < aux[i])
            vetor[k] = aux[j++];
        else
            vetor[k] = aux[i++];
}
```

MergeSort Bottom-Up

Entrada:

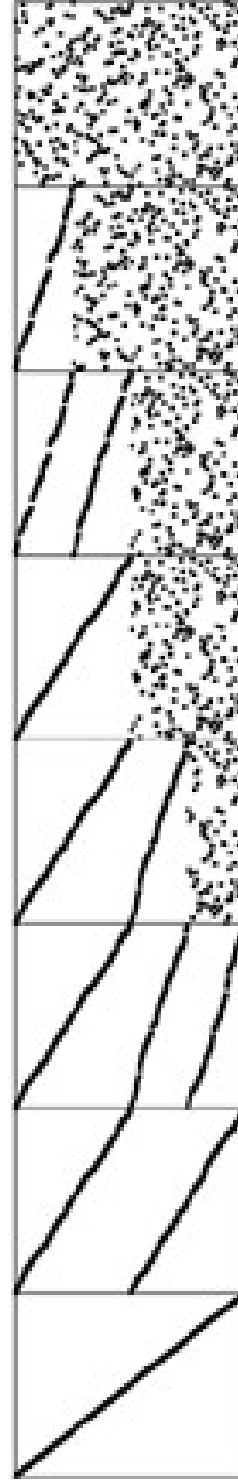
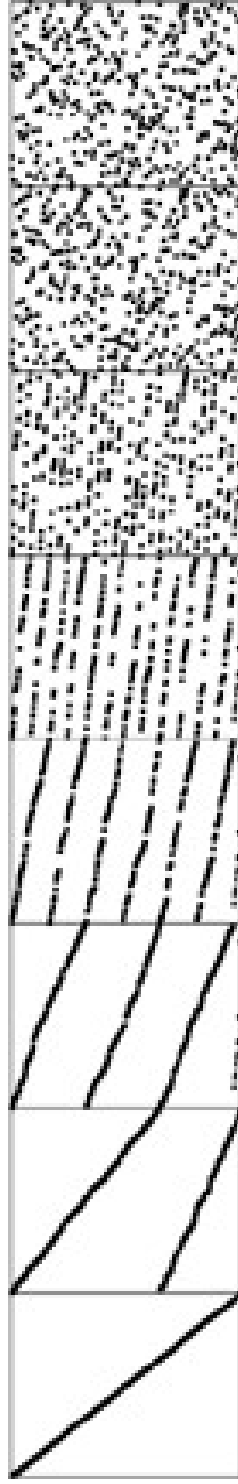


Merge Sort

- Quantas comparações são executadas?
- Quantas trocas são executadas?
- É estável?
- Quantidade de memória?

Merge Sort

- Os valores dos dados não-interferem na execução do algoritmo
- Crescimento do número de comparações em relação ao tamanho de entrada:
 - **linear-logarítmico**
- Crescimento do número de trocas em relação ao tamanho de entrada:
 - **linear-logarítmico**
- Crescimento do uso de memória em relação ao tamanho da entrada: **linear**
- O algoritmo é estável? Sim!

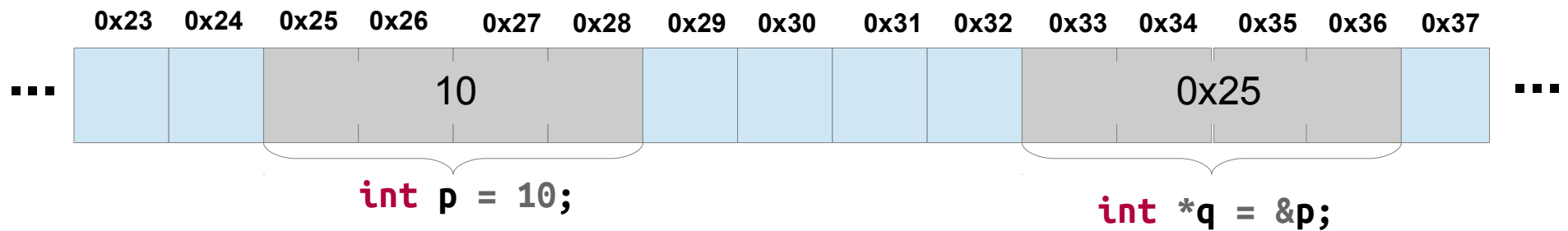


Memória

- A memória de um computador é organizada linearmente, ou seja, uma sequência de *bytes* acessados por um único endereço
- Quando declaramos uma variável e solicitamos sua alocação, um espaço da memória é reservada para armazenar o valor que esta variável irá conter
 - A partir daí, nos referenciamos a este espaço de memória (para ler ou escrever algum valor) pelo nome atribuído a esta variável
- O tamanho do espaço utilizado para cada variável é definido pelo seu tipo
 - Podemos consultar o tamanho em *bytes* de cada tipo pelo operador ***sizeof***
 - Por ex, um *char* ocupa 1 *byte*, um *int* ocupa 4 *bytes*, ... (estes valores podem ser diferentes, a linguagem C exige apenas um mínimo de *bytes* para cada tipo, logo é conveniente utilizar o *sizeof*)
- Todo lugar na memória possui um endereço, então outra forma de acessarmos uma região da memória é pelo seu endereço do primeiro *byte* diretamente (ao invés de utilizarmos um nome de variável)
- Para armazenar um endereço de memória, precisaremos das variáveis do tipo **ponteiro!**

Ponteiros

- **Ponteiro** é uma variável que armazena um endereço de memória
 - Podemos atribuir um tipo de dado a um ponteiro (*int*, *char*, *float*, *double*, ...) , mas neste caso estamos especificando o tipo do dado que estará no endereço de memória armazenado no ponteiro, ou seja, o tipo do dado que o ponteiro aponta!



Ponteiros

- Declaração:

```
<tipo> *<nome_variável_ponteiro>;
```

- Exemplos:

```
int *p; char *c; double *f;
```

- Podemos apontar para qualquer tipo de variável
- Se usarmos como tipo “void” para o ponteiro, indicaremos a ausência de tipo, ou seja, podemos armazenar um endereço de memória sem indicar o tipo de dado que ele aponta
- Ponteiros podem ser declarados junto com variáveis do mesmo tipo:

```
int p, *q, i;
```

Ponteiros

- Operador de endereço &
 - Quando utilizado antes do nome de uma variável, conseguimos obter seu endereço na memória
 - Exemplo:
`int *p = &q; // será atribuído no ponteiro p, o endereço da variável q`
- Operador conteúdo (ou indireto) *
 - Quando utilizado antes de um ponteiro, conseguimos obter o valor armazenado no endereço apontado pelo ponteiro
 - Exemplo:
`*p = 3; // será atribuído o valor 3 no endereço de memória armazenado em p`
 - Cuidado: o operador * possui 3 utilizações distintas!
 - Operador conteúdo
 - Multiplicação
 - Declaração de ponteiros

Ponteiros

- Quando declaramos um ponteiro, estamos pedindo uma variável para armazenar um endereço de memória, e não o dado em si, ou seja, nada mais é alocado!

```
int *p; // 32-bits ou 64-bits para armazenar um endereço! Não há  
espaço em p para armazenar um número inteiro
```

```
double *p; // 32-bits ou 64-bits para armazenar um endereço! Não há  
espaço em p para armazenar um ponto flutuante
```

- Assim como variáveis comuns, não podemos utilizar (ler ou escrever) um ponteiro não inicializado
 - Sempre existirá um valor como endereço, mas será inválido (lixo) se não foi atribuído pelo programador!

Ponteiros

- Podemos indicar explicitamente que um ponteiro não está armazenando um endereço de memória válido! Assim conseguimos distinguir do caso de um lixo de memória

```
int *p = 0;
```

- Se incluirmos a `stdlib.h`, podemos utilizar o `NULL`

```
int *p = NULL;
```

- Ponteiros com valores 0, `NULL` ou lixo de memória (ou qualquer endereço de memória que não seja “seu”)
 - NÃO podem ser acessados!

```
int *p, *q = 0, *r;
```

```
*p = 3; // ERRO!
```

```
printf(“%d”, *q); // ERRO!
```

```
r = 4;
```

```
printf(“%d”, *r); // ERRO!
```

Ponteiros

- Exemplos:

```
int i, j, *p, *q;
```

```
p = &i; // p recebe o endereço de i
q = p; // q recebe o endereço armazenado em p
i = 4; // a variável i passa a armazenar o valor 4
printf("%d %d %d", i, *p, *q); // saída padrão recebe o valor de i e os conteúdos dos
                                // endereços armazenados em p e em q, ou seja, 4 4 4
```

```
*p = 3; // no endereço armazenado em p, o valor 3 é atribuído
printf("%d %d %d", i, *p, *q); // saída padrão recebe o valor de i e os conteúdos dos
                                // endereços armazenados em p e em q, ou seja, 3 3 3
```

```
j = 2; // j recebe o valor 2
q = &j; // o ponteiro q recebe o endereço da variável j
printf("%d %d %d", j, *p, *q); // saída padrão recebe o valor de j e os conteúdos dos
                                // endereços armazenados em p e em q, ou seja, 2 3 2
```

```
*p = *q; // o conteúdo que está no endereço em q é atribuído no endereço em p
printf("%d %d %d %d", i, j, *p, *q); // saída padrão recebe o valor de i, j e os conteúdos dos
                                        // endereços armazenados em p e em q, ou seja, ??????
```

Passagem de parâmetros

- Quando queríamos que um subprograma alterasse o valor de uma variável que foi passada como parâmetro, utilizamos ponteiros!

```
void proc2( int a ) {  
    ...  
    a = 3;  
    ...  
}
```

```
void proc1( int *a ) {  
    ...  
    *a = 3;  
    ...  
}
```

```
...  
int b = 2;  
proc2(b);  
printf("%d", b); // saída padrão recebe o valor 2  
...
```

```
...  
int b = 2;  
proc1(&b);  
printf("%d", b); // saída padrão recebe o valor 3  
...
```


Ponteiro

- Podemos ver o endereço armazenado em um ponteiro

```
int i = 20;
```

```
int *q = &i;
```

```
int *r;
```

```
printf("%d %p %p", i, q, r); // ok!
```

```
printf("%d %d %p", i, *q, r); // ok!
```

```
printf("%d %d %d", i, *q, *r); // ERRO!
```

- Normalmente o valor do endereço de memória, em si, não é importante!
 - Entre cada execução, as mesmas variáveis podem ser alocadas em regiões (endereços) distintos!

Ponteiros

- Encontre os erros e acertos!

```
int* sub1( int *a ) {  
    *a = 2;  
    return a;  
}
```

```
int* sub2( int *a ) {  
    *a = 2;  
    return &a;  
}
```

```
int* sub3( int a ) {  
    a = 2;  
    return &a;  
}
```

```
int* sub4( int a ) {  
    int b = a + 3;  
    return &a;  
}
```

```
int* sub5( int *a ) {  
    int b = *a + 3;  
    return &b;  
}
```

```
// ...  
int a = 1, *p;  
p = sub1(p);  
p = sub1(&a);  
p = sub2(&a);  
p = sub3(a);  
p = sub4(a);  
p = sub5(&a);  
// ...
```

Ponteiros

- Temos que ter muito, mas muito mesmo, cuidado ao programar com ponteiros!
 - Qualquer erro nos faz acessar posições de memória que não nos pertence
 - As vezes que nos pertenceu no passado, mas não mais!
- Ponteiros é uma grande fonte de erros, mas permite uma flexibilidade enorme!
 - Muitas linguagens não permitem que o programador acesse diretamente os endereços de memória, visando minimizar a quantidade de erros, em detrimento desta flexibilidade

Aritmética de ponteiros

- Podemos fazer contas com os endereços de memória!

- Mais perigos a vista!

- Útil quando utilizamos *arrays*

```
double v[10]; // v é um ponteiro para a primeira posição de um bloco de memória
```

```
double *f = &v[3];
```

- Adição de inteiro

```
printf("%lf = %lf", v[5], *(v+5) );
```

```
printf("%lf = %lf", v[5], *(f+2) );
```

```
printf("%p = %p", &v[4], f+1 );
```

- Subtração de inteiro

```
printf("%lf = %lf", v[1], *(f-2) );
```

```
printf("%p = %p", &v[0], f-3 );
```

- Subtração entre ponteiros

```
printf("%d = 3 e %d = -3", f - v, v - f );
```

Aritmética de ponteiros

- O tamanho do tipos dos elementos apontados por um ponteiro já é considerado na aritmética!

$p+1$ irá para o endereço do próximo elemento, somando o número de *bytes* correspondente ao tipo de p , assim como quando utilizamos os colchetes `[]`

- Podemos utilizar os operadores relacionais `>`, `<`, `>=`, `<=`, `==` e `!=`

```
float f[2] = {3.5, 1.2};
```

`f[0] < f[1]` é falso (0)

`&f[0] < &f[1]` é verdadeiro (1)

Aritmética de ponteiros

- Podemos utilizar o ++ e -- da mesma forma que com inteiros!

```
int v[] = { 4, 8, 2, 0 };  
int *p = v;  
  
printf( "%d", *p ); // 4  
  
printf( "%d", *p++ ); // 4  
  
printf( "%d", *p ); // 8  
  
printf( "%d", (*p)++ ); // 8  
  
printf( "%d", *p ); // 9  
  
printf( "%d", ++*p ); // 10  
  
printf( "%d", ++(*p) ); // 11  
  
printf( "%d", *++p ); // 2  
  
printf( "%d", *(++p) ); // 0  
  
printf( "%d", *p ); // 0
```

Registros

- Podemos ter um ponteiro para uma estrutura (registro)

```
typedef struct {           // ...
    float x, y;
} ponto;

ponto p;
ponto *ptr = &p;

p.x = 1.0;
(*ptr).y = 4.2;

printf("%f", p.y ); // 4.2
printf("%f", (*ptr).x ); // 1.0

// ...
```

- Operador seta ou seTinha ->

```
printf("%f , %f", ptr->x, ptr->y ); // 1.0 , 4.2
```

```
scanf("%f", &ptr->x );
scanf("%f", &(*ptr).y );
```

Registros

- Quando utilizamos registros como parâmetros de subprogramas

```
void proc( ponto p, ponto *q) {  
    q->x = p.x;  
    q->y = p.y;  
}
```

```
// . . .
```

```
ponto f, g;
```

```
g.x = 1.0;
```

```
g.y = 2.0;
```

```
proc(g, &f);
```

```
printf("%f , %f", f.x, f.y); // 1.0 , 2.0
```

```
// . . .
```


Exercício

- Receba um vetor de RA e notas (pode ser aleatório)
- Imprima este vetor na tela
- Faça um procedimento que ordene os elementos deste vetor, considerando o RA
 - Utilize um procedimento separado para trocar de posição dois elementos do vetor
 - Percorra o vetor utilizando aritmética de ponteiros
- Imprima este vetor na tela

