Universidade Federal de Uberlândia

Euller Henrique Bandeira Oliveira
Felipe Dantas Vitorino
Mateus Herrera Gobetti Borges

Métodos de Ordenação

Bucket, Counting, Heap e Shell

Uberlândia, MG
2019

Bucket Sort

O bucket sort é um método de ordenação que distribui os valores de um array em diversos arrays menores como se fossem "baldes" (por meio de uma struct que contem esses outros arrays de tamanho pre-definido). Cada "balde" será ordenado de maneira separada por um único método de ordenação (separado ou não), no código mostrado abaixo foi usado o bubble sort. Após a ordenação os "baldes" serão percorridos em ordem crescente e seus valores serão copiados no array a ser ordenado.

Complexidade

- Melhor Caso, O(n + k), onde k é o número de "baldes";
- Pior Caso, $O(n^2)$, quando o array é copiado para um mesmo "balde".

Vantagens

- Não altera a ordem dos dados iguais, ou seja, estável. Exceto se o algoritmo, utilizado para ordenar os "baldes", não for estável;
- Processamento Simples.

Desvantagens

- Dados devem estar igualmente distribuídos;
- Não recomendado para arrays muito grandes;
- Ordena apenas valores positivos, mas pode ser alterado.

Código em C:

```c
void bubbleSort(int* vet, int tam) {
    int i, aux, troca, fim = tam - 1;

    do{
        troca = -1;

        for (i = 0; i < fim; ++i){
            if (vet[i] > vet[i + 1]){
                aux = vet[i];
                vet[i] = vet[i + 1];
                vet[i + 1] = aux;
                troca = i;
            }// if
        }// for

        --fim;
    } while (troca != -1);
}// bubbleSort

void bucketSort(int* vet, int tam){
    int i, j, maior, menor, nroBaldes, pos;
    Bucket* bd;

    // Achar maior e menor valor:
    maior = menor = vet[0];
    for(i = 1; i < tam; ++i) {
```

```
        if(vet[i] < menor)  menor = vet[i];
        if(vet[i] > maior)  maior = vet[i];
    }// for


    // Inicializa baldes:
    nroBaldes = (maior - menor) / TAM + 1; // TAM eh o define para qtd de
                                           //valores em cada balde
    bd = (Bucket*) malloc(nroBaldes * sizeof(Bucket));
    if(!bd) // Se nao conseguiu alocar
        exit(1);

    for(i = 0; i < tam; ++i) {
        pos = (vet[i] - menor) / TAM; // TAM eh qtd de valores por balde
        bd[pos].valores[bd[pos].qtd++] = vet[i];
    }// for

    // Ordena baldes e coloca no array:
    pos = 0;
    for(i = 0; i < nroBaldes; ++i) {
        bubbleSort(bd[i].valores, bd[i].qtd);

        // Insere balde ordenado na lista:
        for(j = 0; j < bd[i].qtd; ++j)
            vet[pos++] = bd[i].valores[j];
    }// for

    free(bd);
}// bucketSort
```

Simulação Bucket Sort

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
4	1	7	2	8	11	5

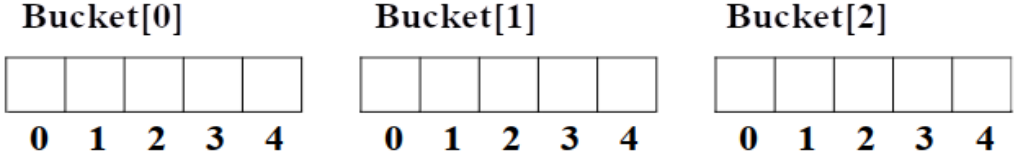
1º Achar maior e menor valor do array e definir a quantidade de baldes necessárias (quantidade dada pelo cálculo abaixo):

maior: 11
menor: 1

Tamanho de cada balde:
TAM: 5

nroBaldes: ((maior - menor) / TAM) + 1
nroBaldes: ((11 - 1) / 5) + 1 = 3

Baldes Criados:



2º Passo: Distribuir os elementos do array nos baldes, conforme a fomula:

pos = (v[i]-menor)/TAM;
bd[pos].valores[bd[pos].qtd] = v[i];

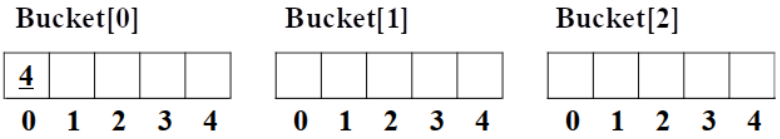
pos: em qual balde o elemento será Inserido

qtd: Próxima posição livre do balde acessado.

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
4	1	7	2	8	11	5

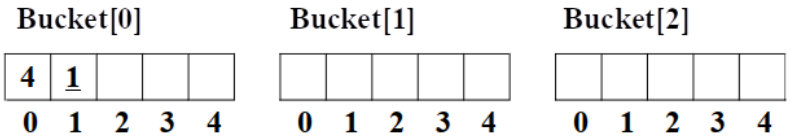
V[0]: 4

Pos = (4 – 1) / 5 == 0;
Bd[0].valores[bd[0].qtd] recebe 4;
bd[0].qtd: 0



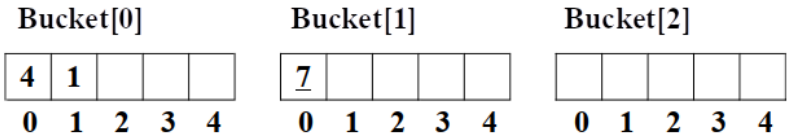
V[1]: 1

pos = (1 – 1) / 5 == 0;
bd[0].valores[bd[0].qtd] recebe 1;
bd[0].qtd: 1



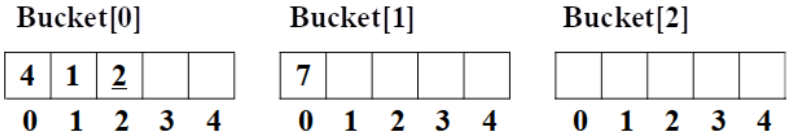
V[2]: 7

pos = (7 – 1) / 5 == 1;
bd[1].valores[bd[1].qtd] recebe 7;
bd[0].qtd: 0



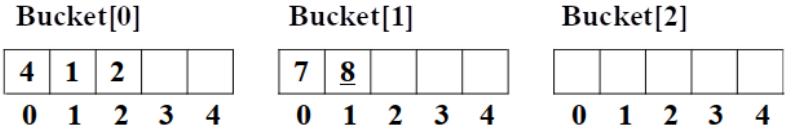
V[3]: 2

pos = (2 – 1) / 5 == 0;
bd[0].valores[bd[0].qtd] recebe 2;
bd[0].qtd: 2



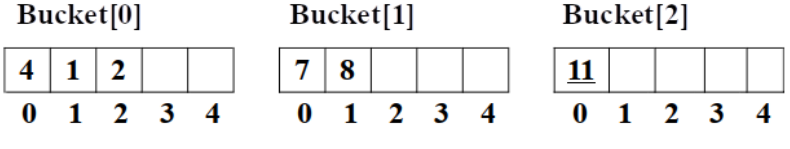
V[4]: 8

pos = (8 – 1) / 5 == 1;
bd[1].valores[bd[1].qtd] recebe 8;
bd[1].qtd: 1



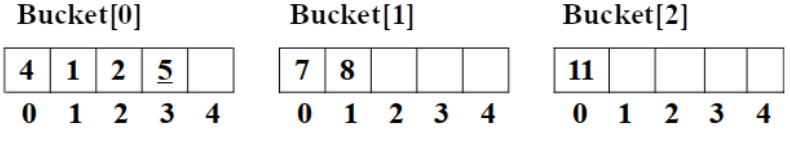
V[5]: 11

pos = (11 – 1) / 5 == 2;
bd[2].valores[bd[2].qtd] recebe 11;
bd[2].qtd: 0

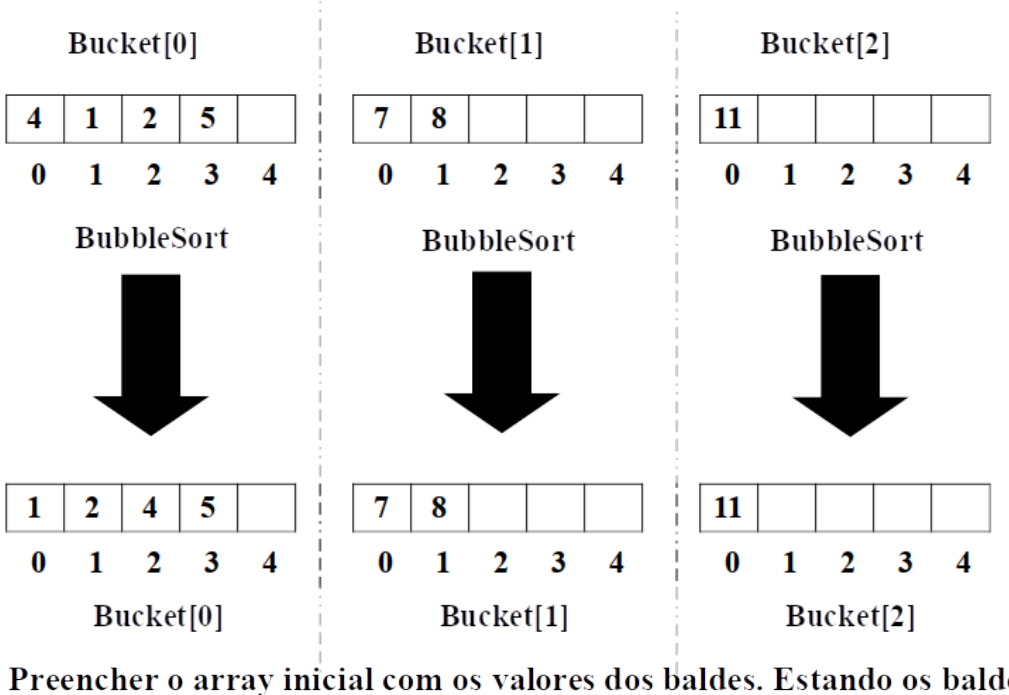


V[6]: 5

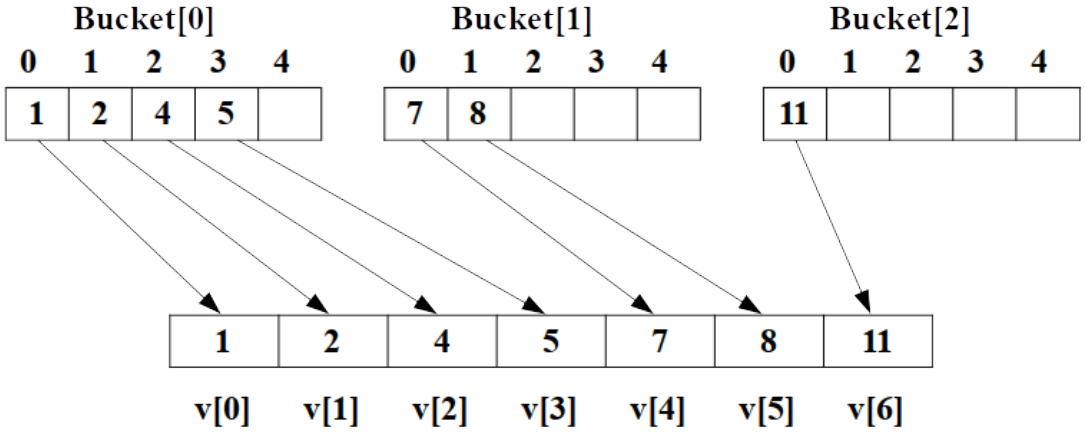
pos = (5 – 1) / 5 == 0;
bd[0].valores[bd[0].qtd] recebe 5;
bd[0].qtd: 3



3º Passo: Ordenar todos os baldes (com um loop), com um determinado método de ordenação. No algoritmo acima foi utilizado o BubbleSort para ordenação dos baldes.



4º Passo: Preencher o array inicial com os valores dos baldes. Estando os baldes em ordem crescente (o último elemento do balde é sempre menor que o primeiro do próximo balde) e ordenados (nesse caso, pelo BubbleSort) o array estará completamente ordenado.



Counting Sort

O counting sort usa um array auxiliar de tamanho igual ao maior valor armazenado do array original a ser ordenado (os valores do array serão tratados como indices no array auxiliar). O array auxiliar será usado para contar quantas vezes o indice aparece como elemento do array original. Depois o algoritmo percorre o array auxiliar e preenche o array original, onde o valor no indice do auxiliar será a quantidade de vezes que o valor, representado pelo indice, aparecerá no original. Dessa maneira, como os indices são crescentes (0 … n), o array ficará ordenado.

Complexidade

- O(n + k), onde k é tamanho do array auxiliar.

Vantagens

- Não altera a ordem de dados iguais, ou seja, é um algoritmo estável;
- Processamento simples.

Desvantagens

- Não é viável, quando o maior elemento do array (tamanho do array auxiliar) for muito grande;
- Ordena apenas valores positivos, mas pode ser alterado.

Código em C:

```c
void countingSort(int* vet, int tam) {
    int i, j, k, maior = vet[0];
    int* contadores;

    // Achar maior valor de vet:
    for(i = 1; i < tam; ++i) {
        if(vet[i] > maior)  maior = vet[i];
    }// for

    // Alocar array do tamanho do maior elemento de vet:
    contadores = (int*) calloc(maior + 1, sizeof(int));

    // Conta elementos de vet em contadores:
    for(i = 0; i < tam; ++i)
        ++contadores[vet[i]];

    // Repreenche vet ordenadamente:
    for(i = 0, j = 0; j <= maior; ++j){
        for(k = contadores[j]; k > 0; --k)
            vet[i++] = j;
    }// for

    free(contadores);
}// countingSort
```

Simulação Counting Sort

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	1	7	3	4

1º Passo: Achar o maior elemento e criar um novo array (inicializado com 0) de tamanho igual à: Maior + 1. (+1, porque o maior deve ser um índice do novo array).

Maior: 7
Tamanho do novo vetor: 8

Novo Array:

0	0	0	0	0	0	0	0
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]

2º Passo: Cada elemento do array original será utilizado como índice do array criado anteriormente. Para cada elemento, no primeiro array, o conteúdo do índice representado por ele será incrementado (No segundo array).

1)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	1	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
0	0	**1**	0	0	0	0	0

2)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	**2**	0	1	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
0	0	**2**	0	0	0	0	0

3)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	**0**	1	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	0	2	0	0	0	0	0

4)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	**1**	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	**1**	2	0	0	0	0	0

5)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	1	**7**	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	0	0	0	0	**1**

6)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	1	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	0	0	0	1

7)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
2	2	0	1	7	3	4

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

3º Passo: Preencher o array original de acordo com a quantidade presente em cada índice do segundo array, em que o índice desse será o elemento do primeiro array.

1)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	2	0	1	7	3	4

2)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	0	1	7	3	4

3)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	2	2	7	3	4

4)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	2	2	3	3	4

5)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	2	2	3	4	4

6)

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]
1	1	2	1	1	0	0	1

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	2	2	3	4	7

Vetor Ordenado:

v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
0	1	2	2	3	4	7

Heap Sort

O método heap utiliza o conceito de árvore binária completa, para ordenação do array original. Todo elemento pai terá seus dois filhos em: posição pos_do_pai * 2 + 1 e pos_do_pai * 2 + 2 (respectivamente, filho da esquerda e filho da direita), na construção dessa árvore o pai sempre será maior que seus filhos.

Com a árvore criada, o algoritimo começara a ordenar o array origem, sendo que o maior elemento estará sempre no topo da árvore, então este elemento será trocado com a última posição do array e a ávore será reconstruida (desconsiderando o antigo topo da árvore, que já estará ordenado). Esse processo será repetido até que todo array esteja ordenado.

Complexidade

- O(n * log n).

Vantagens

- Comportamento O(n * log n).

Desvantagens

- Devido sua complexidade não recomendado para arrays com poucos elementos.

Código em C:

```c
void heapSort(int* vet, int tam){
    int i, aux;

    for(i = (tam - 1) / 2; i >= 0; --i)
        criarHeap(vet, i, tam - 1);
    // for

    for(i = tam - 1; i >= 1; --i) {
        // Coloca maior elemento na ultima posicao do array:
        aux = vet[0];
        vet[0] = vet[i];
        vet[i] = aux;

        // Reconstroi Heap:
        criarHeap(vet, 0, i - 1);
    }// for
}// heapSort

void criarHeap(int* vet, int ini, int fim) {
    int aux = vet[ini];
    int j = ini * 2 + 1;

    while(j <= fim) {
        if(j < fim) {
            // Acha maior filho:
            if(vet[j] < vet[j + 1])
                ++j;
```

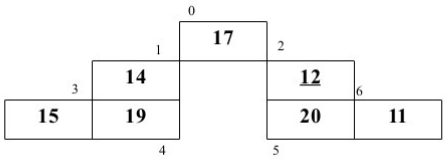
```
        }// if


        // Verifica se filho eh maior que pai se sim filho se torna pai:
        if(aux < vet[j]) {
            vet[ini] = vet[j];
            ini = j;
            j = ini * 2 + 1;

        } else
            break;
        // if - else
    }// while

    vet[ini] = aux; // Antigo primeiro pai oculpa posicao do ultimo
                    //filho analisado
}// criarHeap
```

Simulação Heap Sort

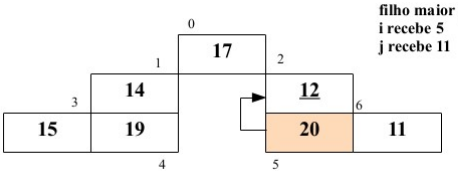
17	14	12	15	19	20	11
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]

1º Cria Heap:

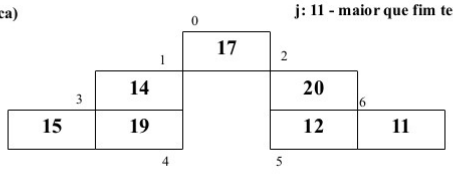
1)



i: 2
maior filho: 20 (j: 5)

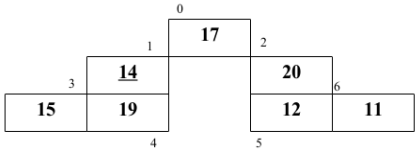


filho maior que o pai (troca)
i recebe 5
j recebe 11

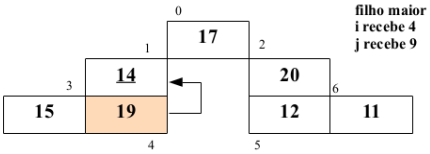


j: 11 - maior que fim termina

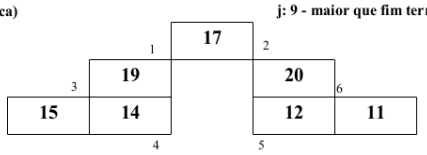
2)



i: 1
maior filho: 19 (j: 4)

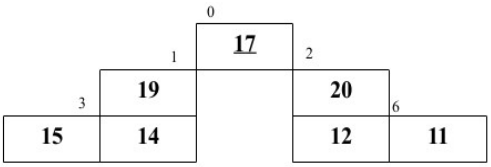


filho maior que o pai (troca)
i recebe 4
j recebe 9

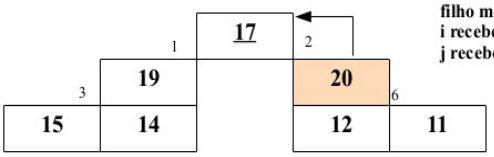


j: 9 - maior que fim termina

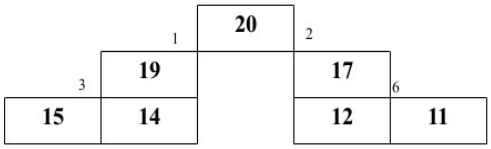
3)



i: 0
maior filho: 20 (j: 2)



filho maior que o pai (troca)
i recebe 2
j recebe 5



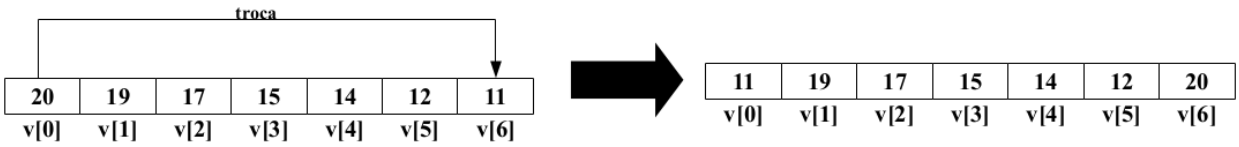
j: 9 - maior que fim termina
aux: 17

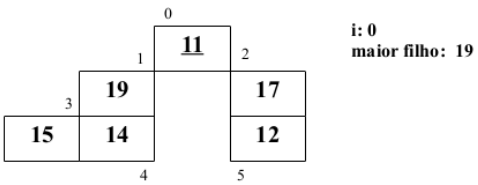
vet[i] recebe aux

Termina aux maior que
Filhos da pos: 6

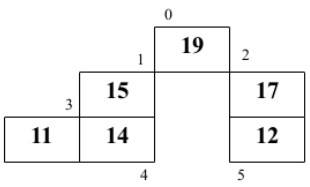
2º Troca primeiro (maior) elemento com o último, reduz fim do array e reconstroi a Heap. Esse processo é repetido até o array estar ordenado:

1)

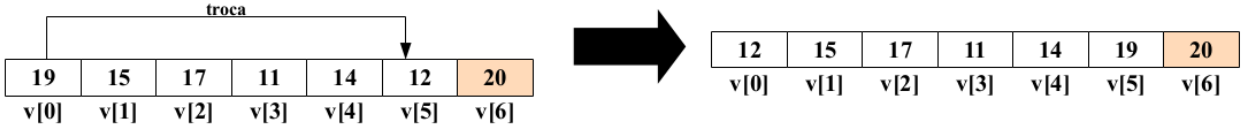


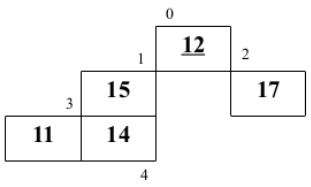


i: 0
maior filho: 19

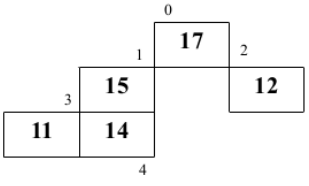


2)

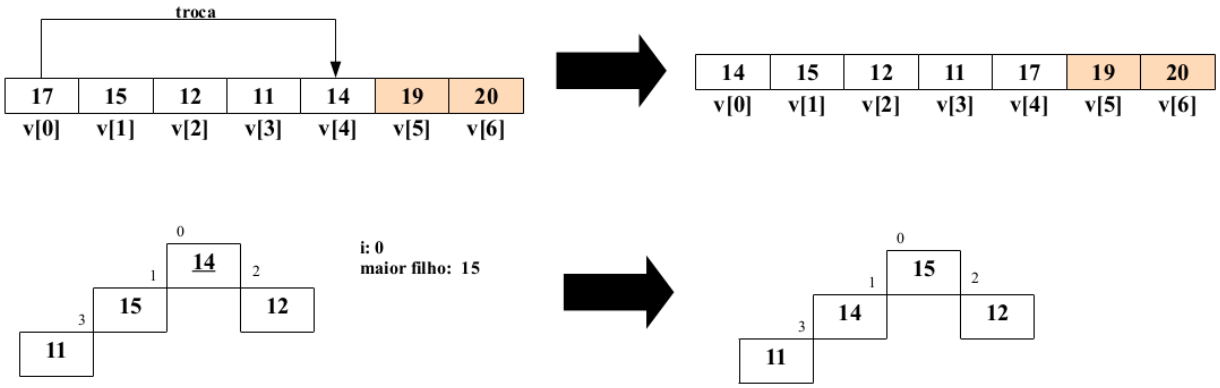




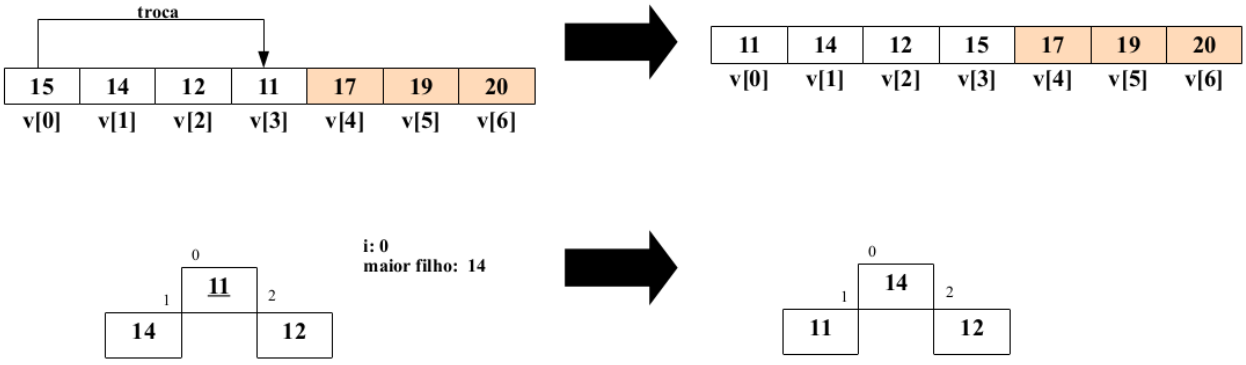
i: 0
maior filho: 17



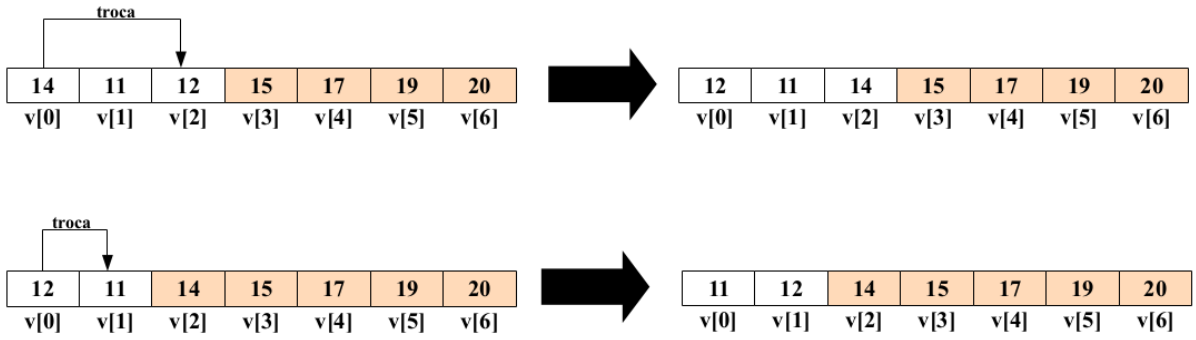
3)



4)



5) A heap já está pronta para as próximas duas trocas:



Array Ordenado:

11	12	14	15	17	19	20
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]

Shell Sort

O método shell sort faz trocas ao comparar dois elementos presentes em um array distanciados pela formula: h=tamanho_do_array / 2. A cada iteração, a comparação é realizada, a troca é feita se necessária, ao final da iteração a nova distancia será calculada dividindo novamente por 2.

O algoritmo utilizara três loops: o primeiro será responsável por determinar o indice de partida (índice ‘i’) de cada iteração deste loop, (este loop também ajusta o valor de ‘h’ após o segundo loop); o segundo encontrará o elemento que será comparado com o valore do índice ‘i’, definido no loop anterior, guardando seu índice em ‘j’ (este loop também ajusta o valor em ‘j’ após o terceiro loop); o terceiro (último) loop comparará o valor em ‘i’ e o valor em ‘j-h’, trocando-os se necessário. Quando o primeiro loop se encerra o array estará ordenado.

Complexidade

- O(n^2).

Vantagens

- Tal método de ordenação é uma excelente opção para arquivos de tamanho mediano;
- A implementação do Shell Sort requer um algoritmo simples e pequeno.

Desvantagens

- Não é estável.

Código em C:

```c
void shellSort(int* vet, int tam) {
    int i, j, aux, h;
    h = tam / 2;

    while(h > 0) {
        i = h;

        while (i < tam) {
            aux = vet[i];
            j = i;

            //Testa se aux < v[j-h]
            while(j >= h && aux < vet[j-h]){
                //V[j-h] se torna o v[j]
                vet[j] = vet[j-h];
                j = j - h;
            }// while
```

```
            // O aux se torna o v[j]
            vet[j] = aux;
            i++;
        }// while

        h = h/2;
    }// while
}// shellSort
```

Simulação ShellSort

41	47	3	90	22	9	51	66
V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]

Passo 1: Encontrar o valor do espaço inicial de comparação(h):
h = tam_vet / 2
h = 4
Passo 2: Comparar os elementos com a distância h e realizar a troca se necessário:

41	47	3	90	22	9	51	66
↑		Troca		↑			

22	47	3	90	41	9	51	66
	↑		Troca		↑		

22	9	3	90	41	47	51	66
		↑		Não Troca		↑	

22	9	3	90	41	47	51	66
			↑		Troca		↑

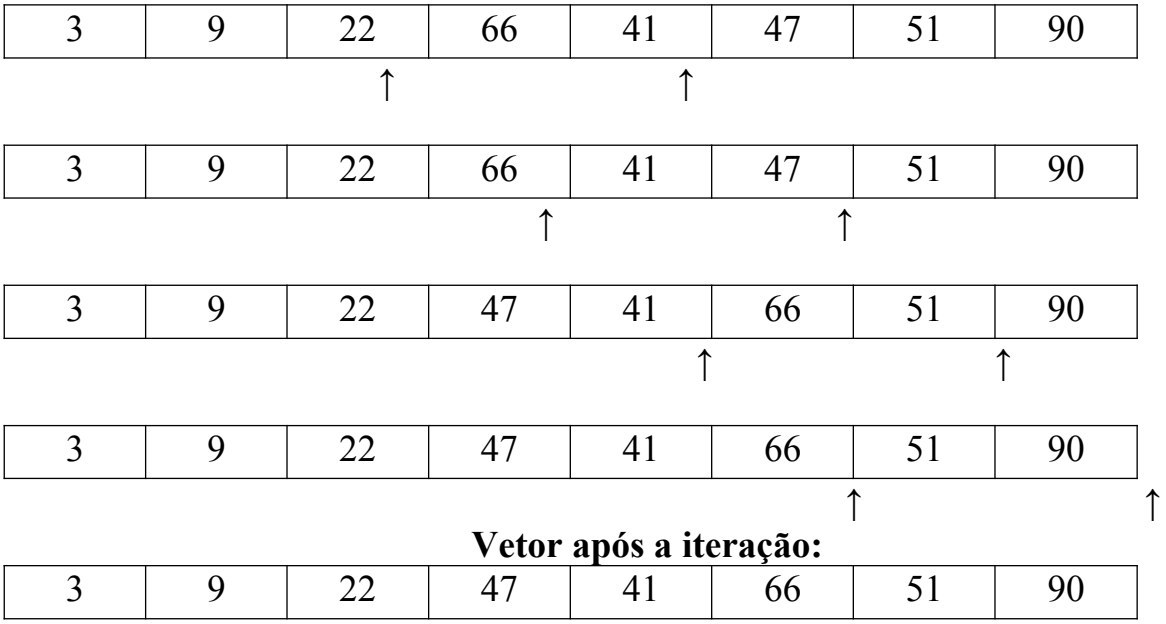
Vetor após a iteração:

22	9	3	66	41	47	51	90

Ao realizar todas as comparações possíveis com h = 4, o valor h é dividido por 2, desta forma:
h = h/2, logo h = 2
Continuando as comparações, agora com h = 2:

22	9	3	66	41	47	51	90
↑		↑					

3	9	22	66	41	47	51	90
	↑		↑				

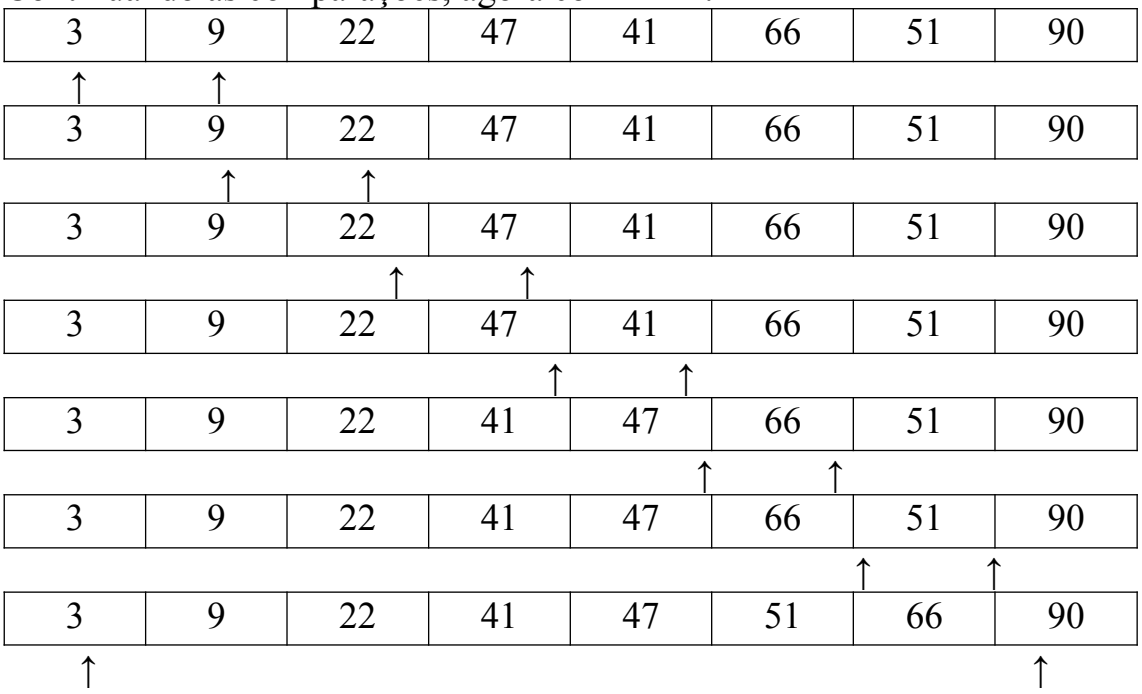


Vetor após a iteração:

3	9	22	47	41	66	51	90

Ao realizar todas as comparações possíveis com h = 2, o valor h é dividido por 2 novamente, desta forma:
h = h/2
h = 1

Continuando as comparações, agora com h = 1:



Vetor após a iteração (vetor ordenado):

3	9	22	41	47	51	66	90

Nota-se que neste ponto do algoritmo, h = 1, ao dividirmos h por 2 temos:
h = 0.5, porém h é um inteiro, portanto a parte fracionária do número é descartada, então h passa a valer zero, e com h = 0 o algoritmo sai do loop e é encerrado retornando o vetor ordenado.