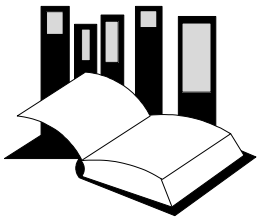
Ordenação em Vetores



- Esta aula introduz métodos avançados de ordenação em vetores:
  - Shellsort
  - Quicksort
  - Heapsort

Prof. Dr. José Augusto Baranauskas
DFM-FFCLRP-USP

Shellsort

- Refinamento do método de ordenação por inserção direta, proposto em 1959 por D.L.Shell
- A inserção direta troca itens adjacentes quando está procurando o ponto de inserção na seqüência destino
- Shellsort: troca de registros que estão distantes um do outro:
  - Itens que estão separados h posições são rearranjados de tal forma que todo h-ésimo item leva a uma seqüência ordenada
  - Também conhecimento como ordenação por inserção através de incrementos decrescentes

Shellsort

- Inicialmente, todos os elementos que estiverem a intervalos de quatro posições entre si na seqüência corrente são agrupados e ordenados separadamentes (ordenação de distância 4)
- Após este primeiro passo, os elementos são reagrupados em grupos com elementos cujo intervalo é de duas posições, sendo então ordenados novamente (ordenação de distância 2)
- Finalmente, em um terceiro passo, todos os elementos são ordenados através de uma ordenação simples (ordenação de distância 1)

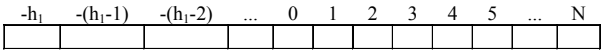
Vetor inicial	45	56	12	43	95	19	8	67
h = 4	45	19	8	43	95	56	12	67
h = 2	8	19	12	43	45	56	95	67
h = 1	8	12	19	43	45	56	67	95

Shellsort

- O algoritmo é, portanto, desenvolvido sem depender de uma seqüência específica de incrementos
- Os **t** incrementos são denotados por $\mathbf{h_1}$, $\mathbf{h_2}$, ..., $\mathbf{h_t}$ com as condições: $h_t = 1$; $h_{i+1} < h_i$
- Cada ordenação de distância **h** é programada na forma de uma ordenação por inserção direta

Shellsort

- Para fazer uso de sentinelas (para obter uma condição mais simples para o término da pesquisa do local correto de inserção), seria necessário estender o tamanho do vetor não apenas de um único item a[0] mas de um número $\mathbf{h_1}$ de elementos:



- Entretanto, linguagens como C/C++ e Java não admitem índices negativos para vetores
- Assim, nossa implementação fará uso de uma condição mais complexa devido a não utilizar sentinelas

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

Com k=2, h assumirá valores (em ordem reversa) 1,3,7,15,31,63,127,255,511,1023...

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

No exemplo N=8 então h=7,3,1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

j+h

N = 8

x = 67 h = 7

Início da ordenação de distância h=7

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

j+h

N = 8

x = 67 h = 7

Término da ordenação de distância h=7

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

j+h

N = 8

x = 43 h = 3

Início da ordenação de distância h=3

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j = -2 i

	1	2	3	4	5	6	7	8
a	45	56	12	45	95	19	8	67

j+h

N = 8

x = 43 h = 3

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j = -2 i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 43 h = 3

13

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 95 h = 3

14

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 95 h = 3

15

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 19 h = 3

16

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 19 h = 3

17

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	8	67

j+h

N = 8

x = 8 h = 3

18

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	43	56	12	45	95	19	45	67

j+h

N = 8

x = 8 h = 3

19

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j = -2 i

	1	2	3	4	5	6	7	8
a	43	56	12	43	95	19	45	67

j+h

N = 8

x = 8 h = 3

20

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j = -2 i

	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

j+h

N = 8

x = 8 h = 3

21

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

j+h

N = 8

x = 67 h = 3

22

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	95

j+h

N = 8

x = 67 h = 3

23

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

j i

	1	2	3	4	5	6	7	8
a	8	56	12	43	67	19	45	95

j+h

N = 8

x = 67 h = 3

Término da ordenação de distância h=3

24

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j		i					
	1	2	3	4	5	6	7	8
a	8	56	12	43	67	19	45	95

j+h

N = 8

x = 56 h = 1

Início da ordenação de distância h=1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j		i					
	1	2	3	4	5	6	7	8
a	8	56	12	43	67	19	45	95

j+h

N = 8

x = 56 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

		j		i				
	1	2	3	4	5	6	7	8
a	8	56	12	43	67	19	45	95

j+h

N = 8

x = 12 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

		j		i				
	1	2	3	4	5	6	7	8
a	8	56	56	43	67	19	45	95

j+h

N = 8

x = 12 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j		i					
	1	2	3	4	5	6	7	8
a	8	12	56	43	67	19	45	95

j+h

N = 8

x = 12 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

			j	i				
	1	2	3	4	5	6	7	8
a	8	12	56	43	67	19	45	95

j+h

N = 8

x = 43 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j			i			
	1	2	3	4	5	6	7	8
a	8	12	56	56	67	19	45	95

j+h

N = 8

x = 43 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j			i			
	1	2	3	4	5	6	7	8
a	8	12	43	56	67	19	45	95

j+h

N = 8

x = 43 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

				j	i			
	1	2	3	4	5	6	7	8
a	8	12	43	56	67	19	45	95

j+h

N = 8

x = 67 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

				j	i			
	1	2	3	4	5	6	7	8
a	8	12	43	56	67	19	45	95

j+h

N = 8

x = 67 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

					j	i		
	1	2	3	4	5	6	7	8
a	8	12	43	56	67	19	45	95

j+h

N = 8

x = 19 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

					j	i		
	1	2	3	4	5	6	7	8
a	8	12	43	56	67	67	45	95

j+h

N = 8

x = 19 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j					i	
	1	2	3	4	5	6	7	8
a	8	12	43	56	56	67	45	95

j+h

N = 8

x = 19 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j					i	
	1	2	3	4	5	6	7	8
a	8	12	43	43	56	67	45	95

j+h

N = 8

x = 19 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

	j					i	
	1	2	3	4	5	6	7	8
a	8	12	19	43	56	67	45	95

j+h

N = 8

x = 19 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

						j	i
	1	2	3	4	5	6	7	8
a	8	12	19	43	56	67	45	95

j+h

N = 8

x = 45 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

						j	i
	1	2	3	4	5	6	7	8
a	8	12	19	43	56	67	67	95

j+h

N = 8

x = 45 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

				j			i
	1	2	3	4	5	6	7	8
a	8	12	19	43	56	56	67	95

j+h

N = 8

x = 45 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

		1	2	3	4	5	6	7	8
					j			i	
a		8	12	19	43	45	56	67	95

j+h

N = 8

x = 45 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

		1	2	3	4	5	6	7	8
								j	i
a		8	12	19	43	45	56	67	95

j+h

N = 8

x = 95 h = 1

Shellsort

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

		1	2	3	4	5	6	7	8
								j	i
a		8	12	19	43	45	56	67	95

j+h

N = 8

x = 95 h = 1

Término da ordenação de distância h=1

Shellsort: Análise

- A análise deste algoritmo identifica alguns problemas difíceis, muitos ainda não solucionados até o momento
- Em particular, não se sabe qual a escolha de incrementos que deverá fornecer melhores resultados
- Um fato interessante é que eles não podem ser múltiplos uns dos outros
- É desejável que ocorra o maior número possível de interações entre as diversas seqüências

Shellsort: Análise

- $O(N^{1.2})$ para a seqüência 1, 3, 7, 15, 31, 63, 127, 255, 511, 1023 ...
- $O(N(\log_2 N)^2)$ para a seqüência 1, 2, 3, 4, 6, 9, 8, 12, 18, 27, 16, 24, 36, 54, 81
- Embora essas sejam melhorias significativas em relação à $O(N^2)$, este método não será detalhado, já que existem outros algoritmos, ainda melhores

Exercício

```
const int k = 2;
int i, j, h;
item x;
h = 1;
do
{ h = k * h + 1;
} while (h <= N);
do
{ h = h / k;
  for (i = h + 1; i <= N; i++)
  { x = a[i];
    j = i - h;
    while (j >= 1 && x < a[j])
    { a[j+h] = a[j];
      j = j - h;
    }
    a[j+h] = x;
  }
} while (h != 1);
```

- Utilizando o algoritmo shellsort, obtenha o número de comparações e movimentações em cada passo (h e i) para os seguintes vetores
  - 45,56,12,43,95,19,8,67
  - 8,12,19,43,45,56,67,95
  - 95,67,56,45,43,19,12,8
  - 19,12,8,45,43,56,67,95

Solução

h	i	Ci	Mi	45	56	12	43	95	19	8	67
7	8	1	2	45	56	12	43	95	19	8	67
3	4	2	3	43	56	12	45	95	19	8	67
3	5	1	2	43	56	12	45	95	19	8	67
3	6	1	2	43	56	12	45	95	19	8	67
3	7	3	4	8	56	12	43	95	19	45	67
3	8	2	3	8	56	12	43	67	19	45	95
1	2	1	2	8	56	12	43	67	19	45	95
1	3	2	3	8	12	56	43	67	19	45	95
1	4	2	3	8	12	43	56	67	19	45	95
1	5	1	2	8	12	43	56	67	19	45	95
1	6	4	5	8	12	19	43	56	67	45	95
1	7	3	4	8	12	19	43	45	56	67	95
1	8	1	2	8	12	19	43	45	56	67	95
		24	37								

h	i	Ci	Mi	8	12	8	43	45	56	67	95
7	8	1	2	8	12	8	43	45	56	67	95
3	4	1	2	8	12	8	43	45	56	67	95
3	5	1	2	8	12	8	43	45	56	67	95
3	6	1	2	8	12	8	43	45	56	67	95
3	7	1	2	8	12	8	43	45	56	67	95
3	8	1	2	8	12	8	43	45	56	67	95
1	2	1	2	8	12	8	43	45	56	67	95
1	3	2	3	8	8	12	43	45	56	67	95
1	4	1	2	8	8	12	43	45	56	67	95
1	5	1	2	8	8	12	43	45	56	67	95
1	6	1	2	8	8	12	43	45	56	67	95
1	7	1	2	8	8	12	43	45	56	67	95
1	8	1	2	8	8	12	43	45	56	67	95
		14	27								

h	i	Ci	Mi	19	12	8	45	43	56	67	95
7	8	1	2	19	12	8	45	43	56	67	95
3	4	1	2	19	12	8	45	43	56	67	95
3	5	1	2	19	12	8	45	43	56	67	95
3	6	1	2	19	12	8	45	43	56	67	95
3	7	1	2	19	12	8	45	43	56	67	95
3	8	1	2	19	12	8	45	43	56	67	95
1	2	2	3	12	19	8	45	43	56	67	95
1	3	3	4	8	12	19	45	43	56	67	95
1	4	1	2	8	12	19	45	43	56	67	95
1	5	2	3	8	12	19	43	45	56	67	95
1	6	1	2	8	12	19	43	45	56	67	95
1	7	1	2	8	12	19	43	45	56	67	95
1	8	1	2	8	12	19	43	45	56	67	95
		17	30								

h	i	Ci	Mi	95	67	56	45	43	19	12	8
7	8	2	3	8	67	56	45	43	19	12	95
3	4	1	2	8	67	56	45	43	19	12	95
3	5	2	3	8	43	56	45	67	19	12	95
3	6	2	3	8	43	19	45	67	56	12	95
3	7	2	3	8	43	19	12	67	56	45	95
3	8	1	2	8	43	19	12	67	56	45	95
1	2	1	2	8	43	19	12	67	56	45	95
1	3	2	3	8	19	43	12	67	56	45	95
1	4	3	4	8	12	19	43	67	56	45	95
1	5	1	2	8	12	19	43	67	56	45	95
1	6	2	3	8	12	19	43	56	67	45	95
1	7	3	4	8	12	19	43	45	56	67	95
1	8	1	2	8	12	19	43	45	56	67	95
		23	36								

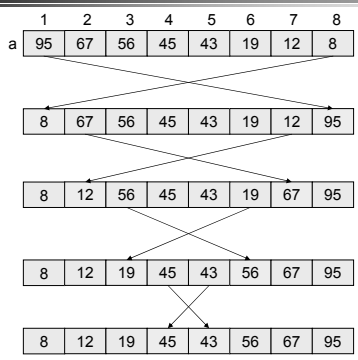
49

Quicksort

- Baseado no fato de que as permutações devem ser preferencialmente empregadas para pares de elementos que guardem entre si distâncias grandes, com a finalidade de se conseguir uma maior eficiência
- Por exemplo, se os **N** elementos estão na ordem inversa de suas chaves, é possível ordená-los com apenas **N**/2 permutações tomando-se primeiramente os elementos das extremidades à direita e à esquerda e convergindo gradualmente para o centro, pelos dois lados
- Obviamente, isto é possível se os elementos estiverem exatamente na ordem inversa

50

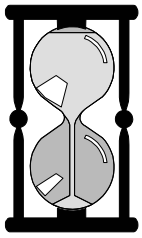
Exemplo



51

Exercício

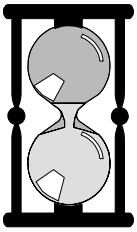
- Utilize essas idéias para escrever um algoritmo capaz de inverter a ordem dos elementos de um vetor de N elementos (índices 1,2,...N)
- Você tem 5 minutos para escrever o algoritmo



52

Solução

```
for(i=1; i <= N/2; i++)
{ x = a[i];
  a[i] = a[N-i+1];
  a[N-i+1] = x;
}
```



53

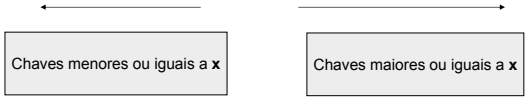
Partição

- Algoritmo de partição
  - escolha-se arbitrariamente um elemento **x** do vetor **a**;
  - o vetor é varrido da esquerda para a direita, até que seja encontrado um elemento a[i] > **x**;
  - após isso, o vetor é varrido da direita para a esquerda até que seja encontrado um elemento a[j] < **x**.
  - nesta ocasião, os dois elementos serão permutados, e este processo de *varredura* e *de permutação* continua até que os dois deslocamentos se encontrem em algum ponto intermediário do vetor
- O resultado desta prática é um vetor particionado, onde a partição esquerda contém apenas chaves cujos valores são menores (ou iguais) a **x** e a partição direita, apenas chaves cujos valores são maiores (ou iguais) a **x**

54

Partição

- Seja x = 43
  - 45 56 12 43 95 8 19 67
  - 45 56 12 43 95 8 19 67
  - 19 56 12 43 95 8 45 67
  - 19 56 12 43 95 8 45 67
  - 19 8 12 43 95 56 45 67



Algoritmo de Partição

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```


Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

	i							j
	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

	i							j
	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

	i						j	
	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		i					j	
	1	2	3	4	5	6	7	8
a	8	56	12	43	95	19	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		1	2	3	4	5	6	7	8
					i				j
a		8	19	12	43	95	56	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		1	2	3	4	5	6	7	8
					i				j
a		8	19	12	43	95	56	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		1	2	3	4	5	6	7	8
					i	j			
a		8	19	12	43	95	56	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		1	2	3	4	5	6	7	8
					j				
					i				
a		8	19	12	43	95	56	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

x = 43 N = 8

		1	2	3	4	5	6	7	8
					j				
					i				
a		8	19	12	43	95	56	45	67

Exemplo de Partição para x = 43

```
i = 1;
j = N;
x = selecionar um elemento aleatoriamente do vetor a;
do
{ while(a[i] < x)
    i++;
  while(x < a[j])
    j--;
  if(i <= j)
  { w = a[i];
    a[i] = a[j];
    a[j] = w;
    i++;
    j--;
  }
} while (i <= j);
```

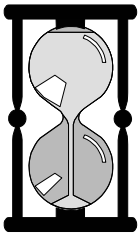
x = 43 N = 8

		1	2	3	4	5	6	7	8
					j	i			
a		8	19	12	43	95	56	45	67

Exercício

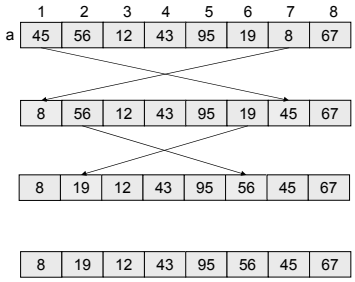
- Particione o vetor para x=45, 56, 12, 95, 19, 8, 67
- Você tem 10 minutos

	1	2	3	4	5	6	7	8
a	45	56	12	43	95	19	8	67



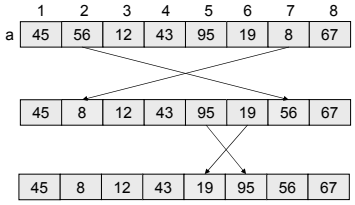
67

Solução x = 45



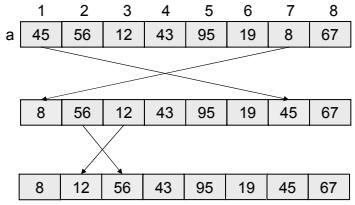
68

Solução x = 56



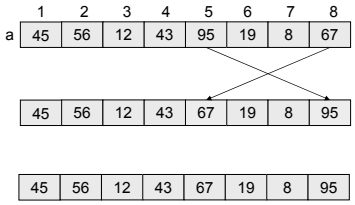
69

Solução x = 12



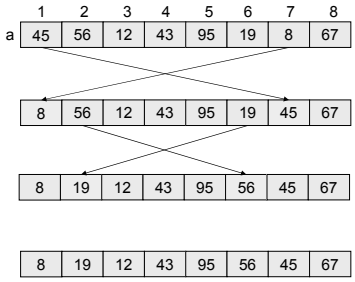
70

Solução x = 95



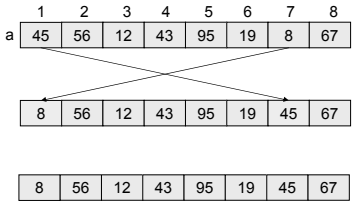
71

Solução x = 19

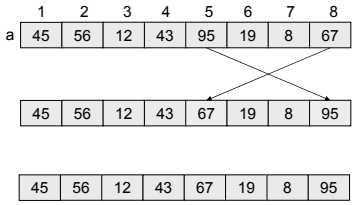


72

Solução x = 8

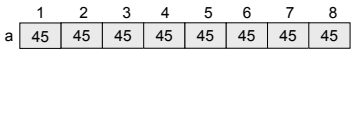


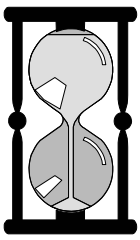
Solução x = 67



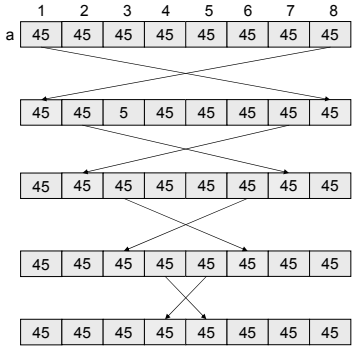
Exercício

- Particione o vetor para x=45
- Você tem 5 minutos





Solução x = 45



Algoritmo de Partição

- Este algoritmo é bastante direto e eficiente
- Entretanto no caso das **N** chaves idênticas são necessárias **N**/2 permutações
- Estas permutações desnecessárias podem ser eliminadas trocando-se os comandos de varredura para

```
while(a[i] <= x)
  i++;
while(x <= a[j])
  j--;
```

- Neste caso, entretanto, o elemento escolhido **x**, que está presente como elemento do vetor, já funcionará como sentinela para as duas varreduras
- Um vetor que possuísse todas as suas chaves idênticas provocaria uma varredura para além da extensão do vetor, a menos que venham a ser utilizadas condições de término mais complexas
- A simplicidade das condições empregadas no algoritmo certamente compensam uma permutação extra, que dificilmente ocorre de fato na média dos casos reais de aplicação

Quicksort

- É necessário lembrar que o objetivo almejado não é só o de encontrar partições do vetor original, mas também ordená-lo
- Entretanto é simples o passo que leva à ordenação a partir do particionamento
  - após ter sido particionado o vetor, aplica-se o mesmo processo para ambas as partições
  - em seguida, para as partições oriundas de cada uma das partições obtidas
  - e assim por diante, até que todas as partições consistam de apenas um único elemento.

Quicksort

```
void qsort(item a[], int L, int R)
{ int i,j;
  item w,x;
  i = L; j = R; x = a[(L + R) / 2];
  do
  { while(a[i] < x) i++;
    while(x < a[j]) j--;
    if(i <= j)
    { w = a[i];
      a[i] = a[j];
      a[j] = w;
      i++; j--;
    }
  } while (i <= j);
  if(L < j)
    qsort(a,L,j);
  if(i < R)
    qsort(a,i,R);
}
```

```
void Quicksort(item a[], int N)
{ qsort(a,1,N);
}
```

- Note que o procedimento *qsort* ativa recursivamente a si próprio para particionar cada partição novamente
- A utilização de recursão em algoritmos é uma ferramenta muito poderosa quando bem empregada

Quicksort

```
void qsort(item a[], int L, int R)
{ int i,j;
  item w,x;
  i = L; j = R; x = a[(L + R) / 2];
  do
  { while(a[i] < x) i++;
    while(x < a[j]) j--;
    if(i <= j)
    { w = a[i];
      a[i] = a[j];
      a[j] = w;
      i++; j--;
    }
  } while (i <= j);
  if(L < j)
    qsort(a,L,j);
  if(i < R)
    qsort(a,i,R);
}
```

```
void Quicksort(item a[], int N)
{ qsort(a,1,N);
}
```

		1	2	3	4	5	6	7	8
		L							R
x=43	a	45	56	12	43	95	19	8	67
		L		R					
x=19		8	19	12	43	95	56	45	67
		L	R						
x=8		8	12	19	43	95	56	45	67
						L			R
x=56		8	12	19	43	95	56	45	67
								L	R
x=95		8	12	19	43	45	56	95	67
		8	12	19	43	45	56	67	95

Quicksort: Análise

- Para analisar o desempenho do algoritmo é necessário verificar o comportamento do processo de partição
- Depois de selecionado um limite **m** uma varredura completa é executada no vetor
- Assim, são realizadas exatamente **N** comparações
- O número de permutações pode ser determinado utilizando conceitos probabilísticos. Para **m** fixado o número esperado de permutações é igual ao número de elementos existentes à esquerda da partição, ou seja, **m-1**, multiplicado pela probabilidade de um elemento encontrar a sua posição certa com uma única permutação
- Uma permutação ocorre desde que o elemento tenha feito parte, anteriormente, da partição à direita; a probabilidade disso ocorrer é **(N-(m-1))/N**
- Portanto, o número esperado de permutações é a média destes valores esperados para todos os possíveis limites **m**:

$$M_{\text{máx}} = \frac{1}{N}\sum_{m=1}^{N}(m-1)\frac{N-(m-1)}{N} = \frac{1}{N^2}\sum_{u=0}^{N-1}u(N-u) = \frac{1}{N^2}\left[N\sum_{u=0}^{N-1}u - \sum_{u=0}^{N-1}u^2\right]$$

$$= \frac{N(N-1)}{2N} - \frac{2N^2-3N+1}{6N} = \frac{N-1/N}{6}$$

Quicksort: Análise

- Admitindo-se que sempre ocorra o melhor caso (o limite escolhido é o ponto médio da partição), então em cada particionamento, o vetor é dividido em duas metades e o número de passos necessários para a ordenação será log(**N**)
  - Então o número total resultante de comparações será **N** log(**N**) e o número total de permutações será **N** log(**N**)/6
- Quanto ao pior caso, considere, por exemplo, a situação na qual o elemento **x** de comparação como sendo o maior dos valores de uma partição. Então, em cada passo, um segmento de **N** elementos será dividido em uma partição esquerda com **N-1** elementos e uma partição direita com um único elemento
  - O resultado é que são necessários **N** (ao invés de log(**N**)) divisões e que o desempenho para o pior caso é O(**N²**), que ocorre quando o vetor está na ordem reversa

Exercício

```
void qsort(item a[], int L, int R)
{ int i,j;
  item w,x;
  i = L; j = R; x = a[(L + R) / 2];
  do
  { while(a[i] < x) i++;
    while(x < a[j]) j--;
    if(i <= j)
    { w = a[i];
      a[i] = a[j];
      a[j] = w;
      i++; j--;
    }
  } while (i <= j);
  if(L < j)
    qsort(a,L,j);
  if(i < R)
    qsort(a,i,R);
}

void Quicksort(item a[], int N)
{ qsort(a,1,N);
}
```

- Utilizando o algoritmo quicksort, obtenha o número de comparações e movimentações em cada passo (**i** e **j**) para os seguintes vetores
  - 45,56,12,43,95,19,8,67
  - 8,12,19,43,45,56,67,95
  - 95,67,56,45,43,19,12,8
  - 19,12,8,45,43,56,67,95

Solução

i	j	Ci	Mi	45	56	12	43	95	19	8	67
5	3	9	10	8	19	12	43	95	56	45	67
3	2	3	4	8	12	19	43	95	56	45	67
2	0	3	4	8	12	19	43	95	56	45	67
7	5	5	7	8	12	19	43	45	56	95	67
8	7	2	4	8	12	19	43	45	56	67	95
		22	29								

i	j	Ci	Mi	8	12	19	43	45	56	67	95
5	3	9	4	8	12	19	43	45	56	67	95
3	1	4	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		21	16								

i	j	Ci	Mi	19	12	8	45	43	56	67	95
5	4	8	4	19	12	8	43	45	56	67	95
3	1	5	7	8	12	19	43	45	56	67	95
4	2	3	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		24	23								

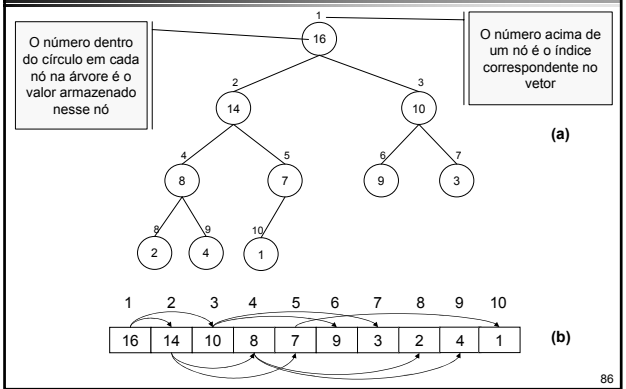
i	j	Ci	Mi	95	67	56	45	43	19	12	8
5	4	8	13	8	12	19	43	45	56	67	95
3	1	5	4	8	12	19	43	45	56	67	95
4	2	3	4	8	12	19	43	45	56	67	95
7	5	5	4	8	12	19	43	45	56	67	95
8	6	3	4	8	12	19	43	45	56	67	95
		24	29								

Heaps

- A estrutura de dados *heap* (binário) é um objeto arranjo (vetor) que pode ser visto como uma árvore binária praticamente completa
- Cada nó da árvore corresponde a um elemento do vetor que armazena o valor do nó
- A árvore é completamente preenchida em todos os níveis, exceto talvez o nível mais baixo, que é preenchido da esquerda até determinado ponto

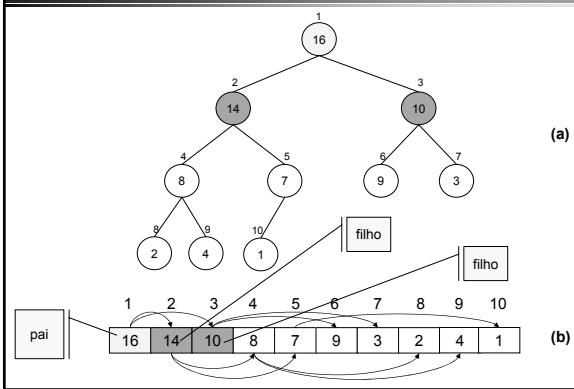
85

Heaps



86

Heaps



87

Heaps

- Um vetor **a** que representa um heap é um objeto com dois atributos:
  - **N** = *comprimento*(**a**), que é o número de elementos do vetor e
  - **R** = *tamanho-do-heap*(**a**), que é o número de elementos no heap armazenado no vetor **a**
- Assim, embora a[1], ...a[**N**] possa conter valores válidos, nenhum elemento além de a[**R**], onde **R<=N**, é um elemento do heap
- A raiz da árvore é a[1] e, dado o índice **i** de um nó, os índices
  - de seu pai: PARENT(**i**) = $\lfloor i/2 \rfloor$
  - do filho da esquerda: LEFT(**i**) = 2*i
  - do filho da direita: RIGHT(**i**) = 2*i + 1

88

Heaps Máximos e Mínimos

- Existem dois tipos de heaps binários: heaps máximos e heaps mínimos
  - Em ambos os tipos, os valores nos nós satisfazem a uma **propriedade de heap**, cujos detalhes específicos dependem do tipo de heap
- Em um **heap máximo**, a propriedade de heap máximo é que para todo nó **i** diferente da raiz:
  - a[PARENT(i)] >= a[i]
  - isto é, o valor de um nó é, no máximo, o valor de seu pai. Desse modo, o maior elemento em um heap máximo é armazenado na raiz, e a subárvore que tem raiz em um nó contém valores menores que o próprio nó
- Um **heap mínimo** é organizado de modo oposto; a propriedade de heap mínimo é que para todo nó **i** diferente da raiz
  - a[PARENT(i)] <= a[i]
  - O menor elemento de um heap mínimo está na raiz

89

Altura de um Heap

- A **altura** de um nó em um heap é o número de arestas (ou arcos) no caminho descendente simples mais longo desde o nó até uma folha
- A **altura** do heap é a altura de sua raiz
- Tendo em vista que um heap de **N** elementos é baseado em uma árvore binária completa, sua altura é $\lfloor \log_2 N \rfloor$
- As operações básicas sobre heaps são executadas em um tempo máximo proporcional à altura da árvore, e assim demoram um tempo O(log2 N)
- Para o algoritmo Heapsort, utilizaremos heaps máximos; assim toda menção a um heap deste ponto em diante se refere a um heap máximo

90

Manutenção de um Heap

- Dado um vetor a[1],..., a[**N**] os elementos a[**m**], ..., a[**N**], com **m** = $\lfloor N/2 \rfloor + 1$ já formam um heap, uma vez que nenhum par de índices (**i**,**j**) é tal que **j** = 2**i** ou j = 2**i**+1
- Esses elementos formam a linha inferior da árvore binária a eles associada, entre os quais nenhuma relação de ordem é exigida
- O heap é agora estendido para a esquerda, sendo que um novo elemento é incluído a cada passo e posicionado apropriadamente por meio de uma operação de escorregamento, que nos leva ao procedimento **heapify**

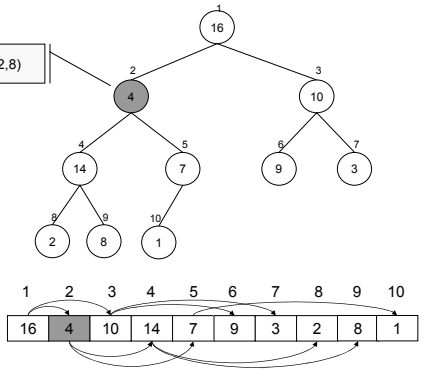
Manutenção de um Heap

```
void heapify(item a[], int L, int R)
{ int i,j;
  item x;

  i = L;
  j = 2 * L;
  x = a[L];
  if (j < R && a[j] < a[j+1])
    j++;
  while(j <= R && x < a[j])
  { a[i] = a[j];
    i = j;
    j = 2 * j;
    if (j < R && a[j] < a[j+1])
      j++;
  }
  a[i] = x;
}
```

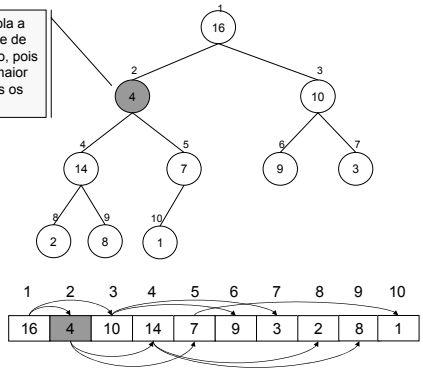

Manutenção de um Heap

heapify(a,2,8)



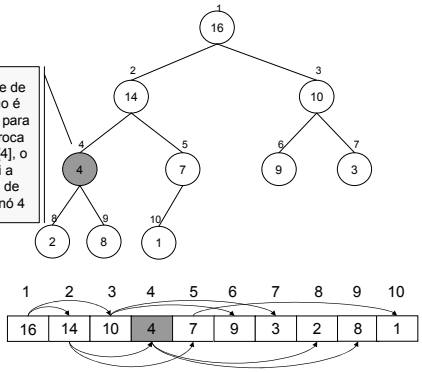
Manutenção de um Heap

a[2] = 4 viola a propriedade de heap máximo, pois ele não é maior que ambos os filhos



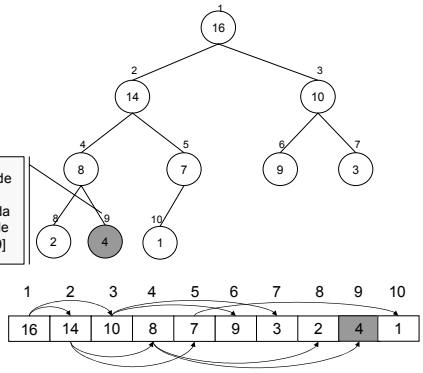
Manutenção de um Heap

A propriedade de heap máximo é restabelecida para o nó 2 pela troca de a[2] por a[4], o que destrói a propriedade de heap para o nó 4



Manutenção de um Heap

a propriedade de heap é restabelecida pela troca de a[4] por a[9]



Construção de um Heap

- Podemos utilizar o procedimento **heapify** de baixo para cima, a fim de converter um vetor a[1],...a[N] em um heap
- Os elementos a[N/2+1],...,a[N] são todos folhas da árvore, então cada um deles é um heap de 1 elemento com o qual podemos começar
- O procedimento para construção de um heap percorre os nós restantes da árvore e executa **heapify** sobre cada um:

```
for(L = N/2; L >= 1; L--)
  heapify(a,L,N);
```

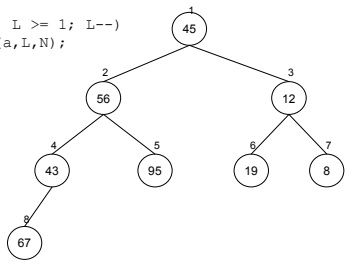
97

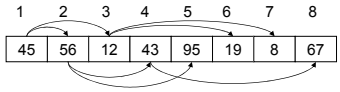
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 4





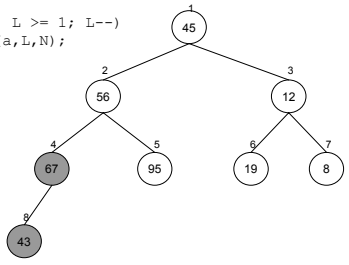
98

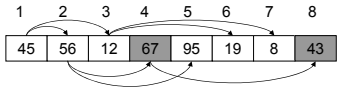
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 4





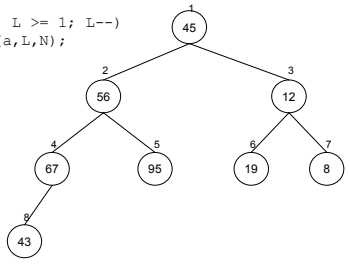
99

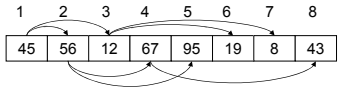
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 3





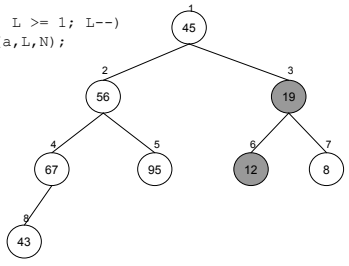
100

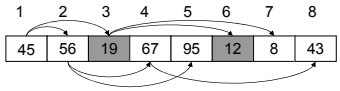
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 3





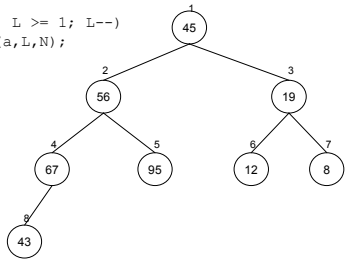
101

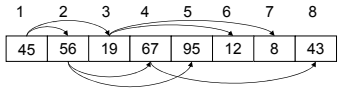
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 2





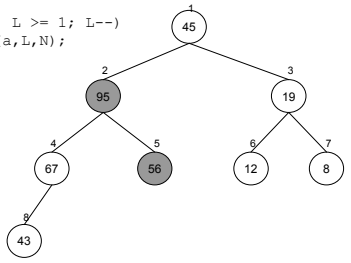
102

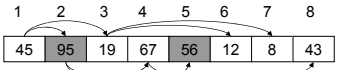
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 2



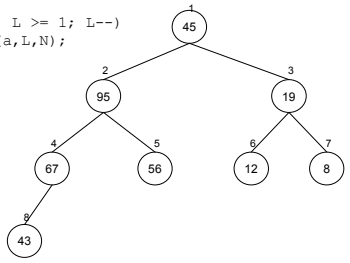


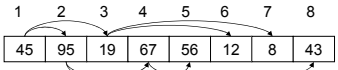
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 1



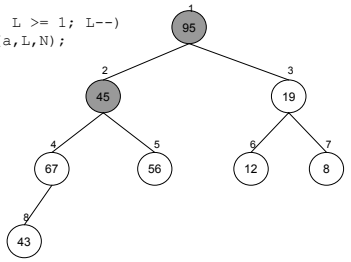


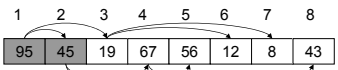
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 1



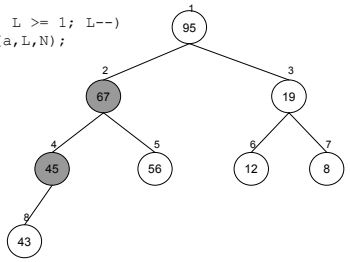


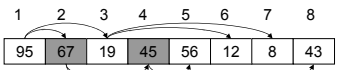
Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8

L = 1

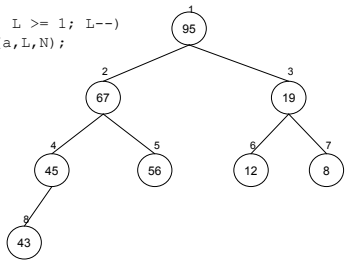


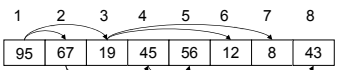


Construção de um Heap

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
```

N = 8





Heapsort

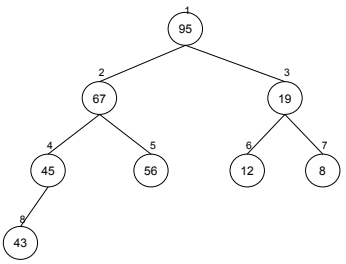
- Com a finalidade de obter uma ordenação completa dos **N** elementos deve-se seguir **N** passos de escorregamento, em que, após a execução de cada passo, o novo elemento pode ser retirado do topo do heap
- Uma questão que surge é onde armazenar os elementos que emergem do topo e se seria possível uma ordenação *in situ*
- Tal solução existe: em cada passo, é necessário retirar o elemento do topo do heap na posição liberada, antes ocupada por **w** e permitir que **w** escorregue para a sua posição adequada:

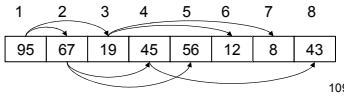
```
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 8



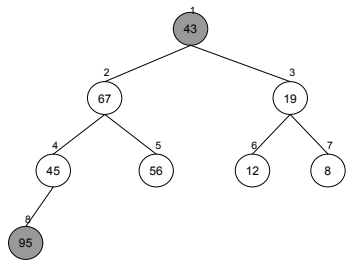


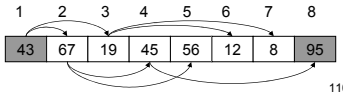
109

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 8



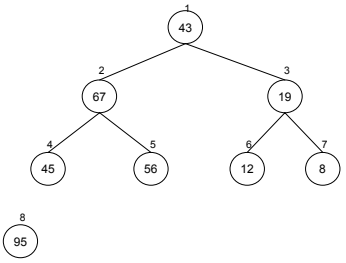


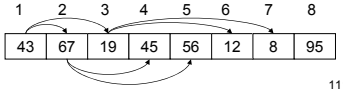
110

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 8



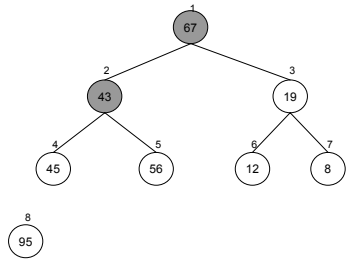


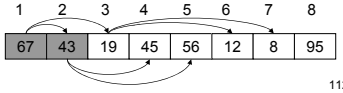
111

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 8



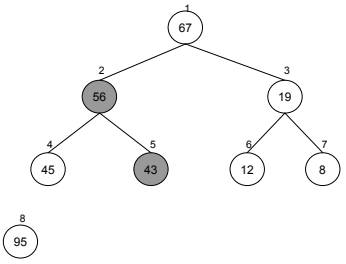


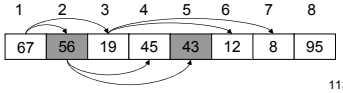
112

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 8



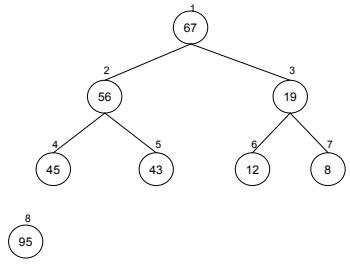


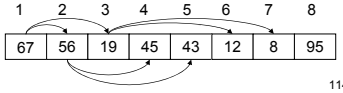
113

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

N = 8
R = 7

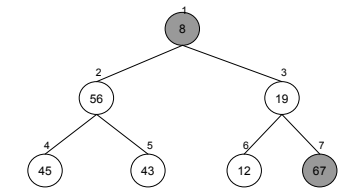




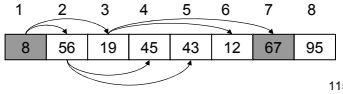
114

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

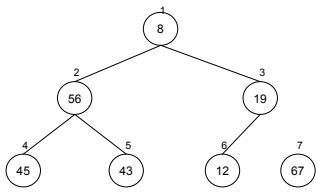


N = 8
R = 7

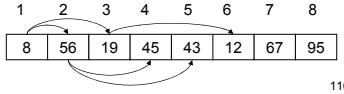


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

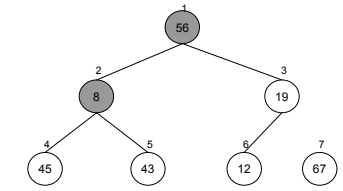


N = 8
R = 7

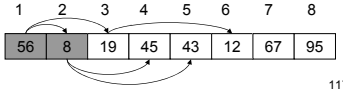


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

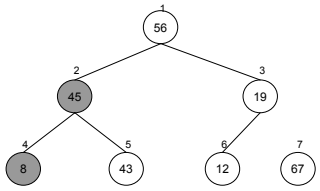


N = 8
R = 7

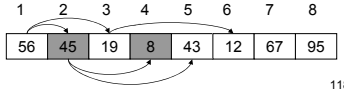


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

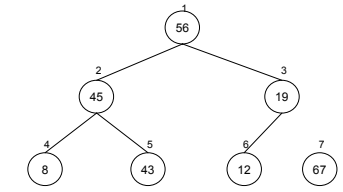


N = 8
R = 7

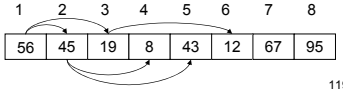


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

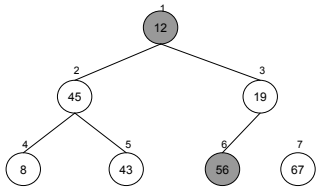


N = 8
R = 6

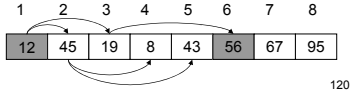


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

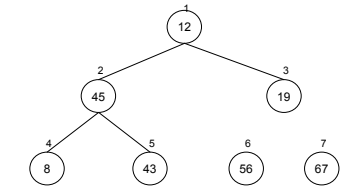


N = 8
R = 6

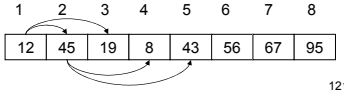


Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



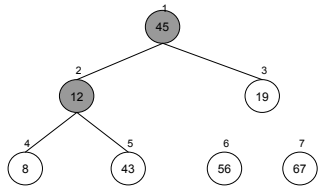
N = 8
R = 6



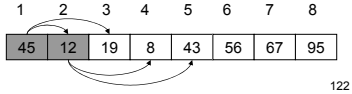
121

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



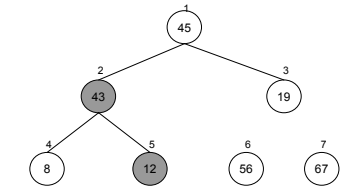
N = 8
R = 6



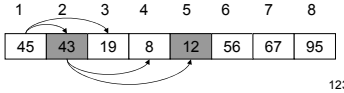
122

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



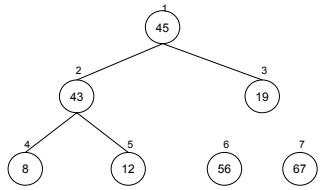
N = 8
R = 6



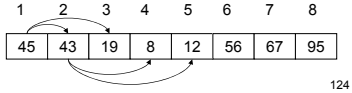
123

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



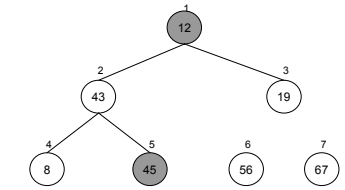
N = 8
R = 5



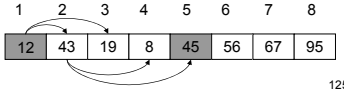
124

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



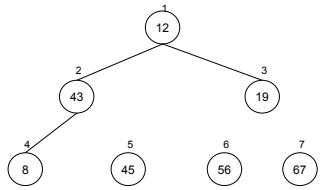
N = 8
R = 5



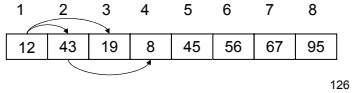
125

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



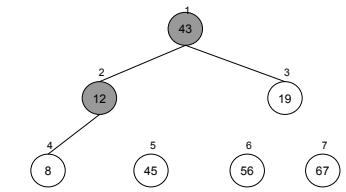
N = 8
R = 5



126

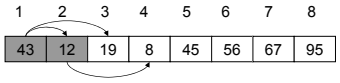
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

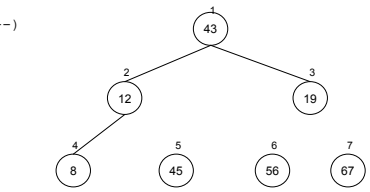
R = 5



127

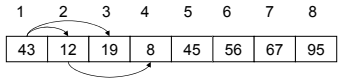
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

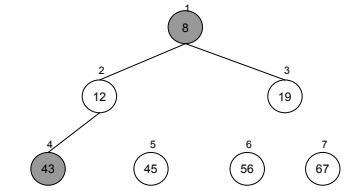
R = 4



128

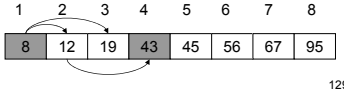
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

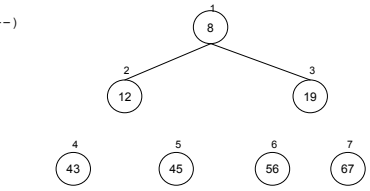
R = 4



129

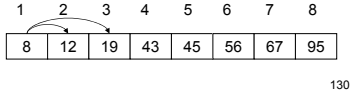
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

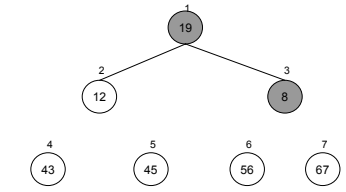
R = 4



130

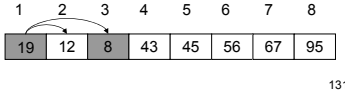
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

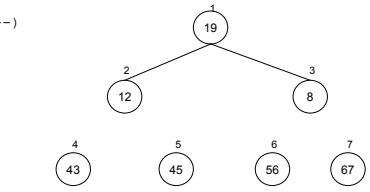
R = 4



131

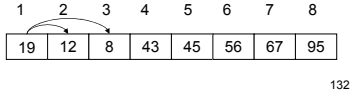
Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

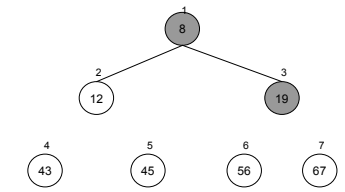
R = 3



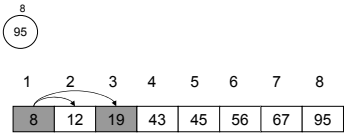
132

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



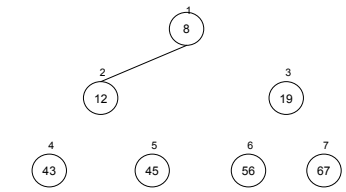
N = 8
R = 3



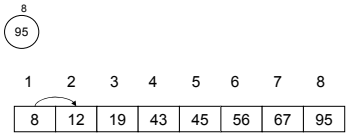
133

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



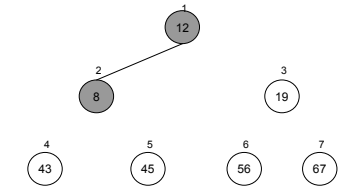
N = 8
R = 3



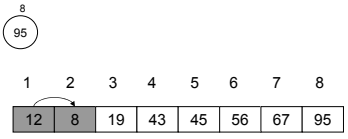
134

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



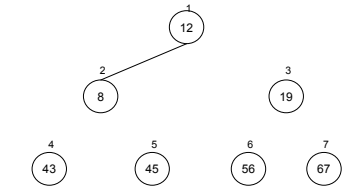
N = 8
R = 3



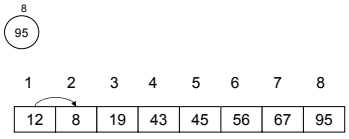
135

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



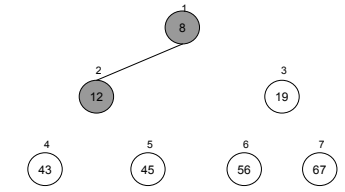
N = 8
R = 2



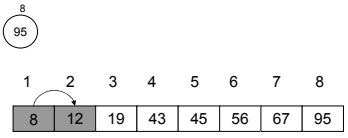
136

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



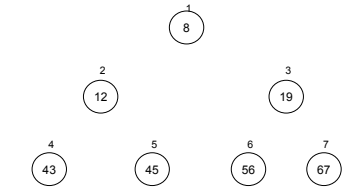
N = 8
R = 2



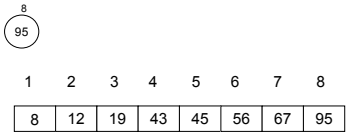
137

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



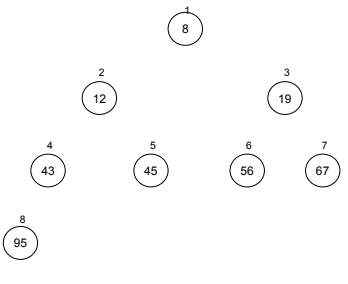
N = 8
R = 2



138

Heapsort

```
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```



N = 8

1	2	3	4	5	6	7	8
8	12	19	43	45	56	67	95

Vetor ordenado

Heapsort: Análise

- À primeira vista não é evidente que este método de ordenação ofereça bons resultados, pois os elementos escorregam para a esquerda em primeiro lugar antes de serem finalmente colocados na sua posição correta, na extremidade direita
- De fato, o procedimento não é recomendado para pequenos valores de **N**
- Todavia, para valores grandes de **N**, o método Heapsort é muito eficiente, e quanto maior for o valor de **N**, melhor será o seu desempenho, mesmo comparado com o do Shellsort
- O número médio de movimentos é de aproximadamente **N**/2*log **N** e os desvios relativos a este valor são relativamente pequenos

Exercício

```
void heapify(item a[], int L, int R)
{ int i,j;
  item x;
  i = L;
  j = 2 * L;
  x = a[L];
  if (j < R && a[j] < a[j+1])
    j++;
  while(j <= R && x < a[j])
  { a[i] = a[j];
    i = j;
    j = 2 * j;
    if (j < R && a[j] < a[j+1])
      j++;
  }
  a[i] = x;
}
//------------------------------
for(L = N/2; L >= 1; L--)
    heapify(a,L,N);
for(R = N; R >= 2; R--)
{ w = a[1];
  a[1] = a[R];
  a[R] = w;
  heapify(a,1,R-1);
}
```

- Utilizando o algoritmo quicksort, obtenha o número de comparações e movimentações em cada passo (**i** e **j**) para os seguintes vetores
  - 45,56,12,43,95,19,8,67
  - 8,12,19,43,45,56,67,95
  - 95,67,56,45,43,19,12,8
  - 19,12,8,45,43,56,67,95

Solução

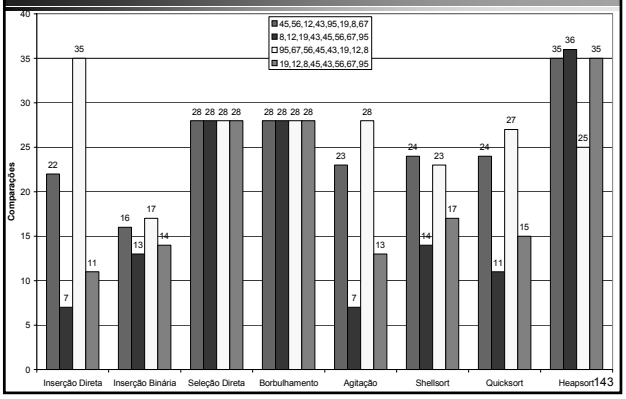
L/R	Ci	Mi	45	56	12	43	95	19	8	67
L=4	2	3	45	56	12	67	95	19	8	43
L=3	3	3	45	56	19	67	95	12	8	43
L=2	3	3	45	95	19	67	56	12	8	43
L=1	4	4	95	67	19	45	56	12	8	43
R=8	5	7	67	56	19	45	43	12	8	95
R=7	5	7	56	45	19	8	43	12	67	95
R=6	5	7	45	43	19	8	12	56	67	95
R=5	2	6	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	35	57								

L/R	Ci	Mi	8	12	19	43	45	56	67	95
L=4	2	3	8	12	19	95	45	56	67	43
L=3	3	3	8	12	67	95	45	56	19	43
L=2	4	4	8	95	67	43	45	56	19	12
L=1	5	4	95	45	67	43	8	56	19	12
R=8	5	7	67	45	56	43	8	12	19	95
R=7	2	6	56	45	19	43	8	12	67	95
R=6	5	7	45	43	19	12	8	56	67	95
R=5	4	7	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
		36	58							

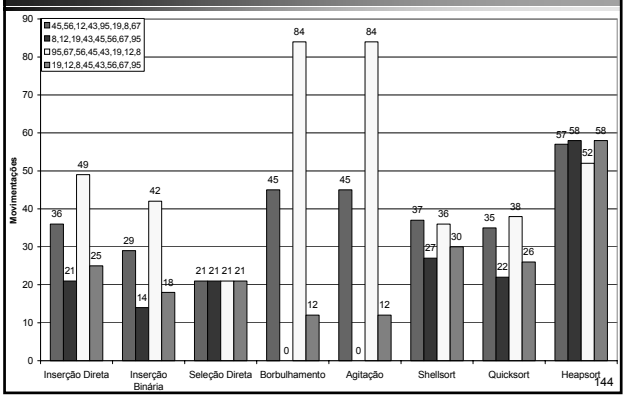
L/R	Ci	Mi	19	12	8	45	43	56	67	95
L=4	2	3	19	12	8	95	43	56	67	45
L=3	3	3	19	12	67	95	43	56	8	45
L=2	4	4	19	95	67	45	43	56	8	12
L=1	4	4	95	45	67	19	43	56	8	12
R=8	5	7	67	45	56	19	43	12	8	95
R=7	4	7	56	45	12	19	43	8	67	95
R=6	5	7	45	43	12	19	8	56	67	95
R=5	4	7	43	19	12	8	45	56	67	95
R=4	3	6	19	8	12	43	45	56	67	95
R=3	0	5	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	35	58								

L/R	Ci	Mi	95	67	56	45	43	19	12	8
L=4	0	2	95	67	56	45	43	19	12	8
L=3	1	2	95	67	56	45	43	19	12	8
L=2	1	2	95	67	56	45	43	19	12	8
L=1	1	2	95	67	56	45	43	19	12	8
R=8	5	7	67	45	56	8	43	19	12	95
R=7	4	7	56	45	19	8	43	12	67	95
R=6	5	7	45	43	19	8	12	56	67	95
R=5	2	6	43	12	19	8	45	56	67	95
R=4	3	6	19	12	8	43	45	56	67	95
R=3	2	6	12	8	19	43	45	56	67	95
R=2	1	5	8	12	19	43	45	56	67	95
	25	52								

Quadro Geral: Comparações



Quadro Geral: Movimentações



Análise dos Algoritmos de Ordenação

- Considerando os algoritmos de ordenação vistos, a tabela seguinte mostra a ordem de grandeza dos
  - números mínimo ($C_{mín}$), médio (C_{med}) e máximo (C_{max}) de comparações de chaves
  - números mínimo ($M_{mín}$), médio (M_{med}) e máximo (M_{max}) de movimentos de chaves.

Algoritmo	C_{min}	$C_{méd}$	$C_{máx}$	M_{min}	$M_{méd}$	$M_{máx}$
Inserção Direta	O(N)	$O(N^2)$	$O(N^2)$	O(N)	$O(N^2)$	$O(N^2)$
Inserção Binária	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	O(N)	$O(N^2)$	$O(N^2)$
Seleção Direta	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(N)	O(N)	O(N)
Bubblesort	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(1)	$O(N^2)$	$O(N^2)$
Shakersort	O(N)	$O(N^2)$	-	O(1)	$O(N^2)$	$O(N^2)$
Heapsort	-	$O(N*\log_2 N)$	$O(N*\log_2 N)$	-	$O(N*\log_2 N)$	$O(N*\log_2 N)$
Quicksort	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N^2)$	$O(N*\log_2 N)$	$O(N*\log_2 N)$	$O(N^2)$

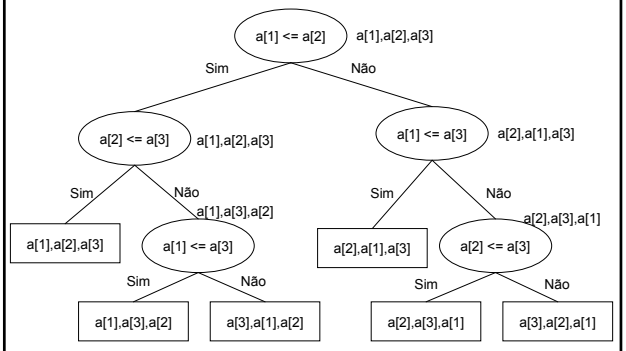
145

Com que Velocidade podemos Ordenar?

- Qualquer algoritmo de ordenação baseado em comparações pode ser visto como uma árvore de decisão, onde
  - os nós internos representam uma comparação de chaves
  - as arestas indicam o resultado do teste (verdadeiro ou falso) e
  - os nós folhas (terminais) representam o resultado final do processo de ordenação

146

Árvore de Decisão (3 elementos)



147

Com que Velocidade podemos Ordenar?

- Ordenando N elementos: N! resultados possíveis
  - A árvore deve ter N! folhas
- Como uma árvore binária de altura h tem, no máximo, 2^{h-1} folhas, sabemos que
  - $N! <= 2^{h-1} \equiv h >= \log_2(N!)+1$
- Portanto qualquer árvore de decisões que classifica N elementos distintos tem uma altura mínima log2(N!)+1
- Sabendo que (pois pelo menos N/2 termos do produto são maiores que N/2)
  - $N! = N(N-1)(N-2)...(2)(1) >= (N/2)^{N/2}$
  - Então $\log_2(N!) >= (N/2)\log_2(N/2) = O(N \log_2 N)$
- O melhor tempo possível é $O(N \log_2 N)$

148

Resumo

- Nesta aula foram vistos alguns métodos avançados de ordenação
- Comparando os métodos diretos e os avançados, nota-se que nem sempre o algoritmo mais “simples” possui o melhor tempo de computação para um grande número de elementos

149