https://brunoribas.com.br/apostila-eda/ordenacao-elementar.html

https://www.ime.usp.br/~pf/algoritmos/aulas/ordena.html

https://github.com/bcribas/benchmark-ordenacao

Importância da Ordenação

- Ordenação é organização
- Organização **otimiza as buscas**
  - Tanto computacionais quanto humanas!
- **Ordenação de itens** (arquivos, estruturas)
  - A **chave** é a parte do item utilizada como parâmetro/controle de ordenação

Características algoritmos de ordenação:

1) Complexidade assintótica

- **$O(n^2)$**
  - Adequado para arquivos pequenos
  - Muitas comparações, porém são comparações simples
- **O(n log n)**
  - Adequado para arquivos grandes
  - Menos comparações, porém são comparações complexos

2) Estabilidade

- É estável quando a **ordem relativa** dos elementos é **mantida**
- Quando é importante?
  - Múltiplas chaves
  - Manter a prévia ordenação de uma das chaves

3) **Adaptatividade**

- É adaptativo quando a **ordenação existente é aproveitada**

4) **Memória extra**

- A quantidade de memória extra usada na ordenação é um fator que afeta o tempo de execução
- **In-place:**
  - utiliza a própria estrutura
  - sem usar memória extra, exceto
    - por uma pequena pilha de execução ou
    - algumas variáveis auxiliares
- Quando não in-place:
  - utiliza uma estrutura extra
  - copia o conteúdo para outro array

Grupos de Ordenação

- Classificados pelo **local de memória** em que são realizadas as operações

- **Ordenação interna**:
  - Utiliza somente a memória principal
  - Todos os dados a serem ordenados cabem de uma vez na memória principal, sem necessidade de memória auxiliar (secundária)
  - Acesso mais rápido: direto com vetores

- **Ordenação externa**:
  - O arquivo a ser ordenado não cabe na memória principal
  - Tem de ser armazenado em memória secundária
    - Acesso sequencial ou em blocos

Algoritmos de Ordenação Interna

- **Métodos eficientes:**
  - Adequados para arquivos maiores
  - Vantagem: **O(n log n)** comparações
  - Desvantagem:
    - Mais complexas
    - Maior o *overhead*, sobrecarga (processamento, memória)

- **Métodos elementares**:
  - Vantagem:
    - São mais simples
    - Propósito geral (independente da estrutura)
  - Desvantagem
    - Custo
    - **O(n^2)** comparações
  - Se são piores, por que usar?
    - Arquivo pequeno?
    - Poucas ordenações?
    - Ordenação não é “gargalo”?
      - Ou seja, não é mais lento que o resto do processamento

do dado?

- Manipulação dos arquivos, impressão na tela..

- Sobrecarga dos algoritmos mais sofisticados não justifica

  seu uso?

  - Custo das constantes é maior que o custo dos métodos

    simples

- Porém, em geral, para muitos dados, os métodos simples não são os mais indicados

- **Array x Listas encadeadas**
  - **Métodos elementares** lidam bem com qualquer implementação
  - **Métodos mais eficientes**:
    - Trabalham com partes (metade) da estrutura
    - Implica em acesso direto para manter a complexidade

      esperada
    - Ideia é reduzir as comparações
    - Reduzir os loops aninhados
  - Alternativa para ordenar estruturas encadeadas:
    - Mantendo ordenadas em **árvores**

Método elementar: SELECTION SORT

- Ideias simples de ordenação?
- Selection → seleção do que?
- **Princípio de funcionamento:**
  - Selecione o menor item do vetor
    - Troque com o primeiro item do vetor
  - Selecione o segundo menor item do vetor
    - Troque com o segundo item do vetor
  - Repita para os **n** elementos do vetor

Seleção 1

Índice do menor 0

0	1	2	3	4	5
3	2	4	6	1	5
i	j				

Índice do menor 1

0	1	2	3	4	5
3	2	4	6	1	5
i		j			

Índice do menor 1

0	1	2	3	4	5
3	2	4	6	1	5
i			j		

Índice do menor 4

0	1	2	3	4	5
3	2	4	6	1	5
i				j	

Índice do menor 4

0	1	2	3	4	5
3	2	4	6	1	5
i					j

Índice do menor 4 – swap i menor

0	1	2	3	4	5
1	2	4	6	4	5
i					j

Seleção 2

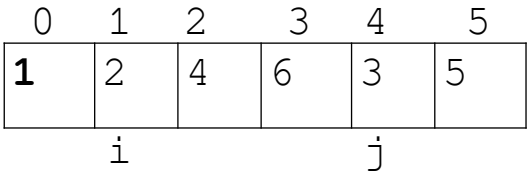
Índice do menor 1

0	1	2	3	4	5
1	2	4	6	3	5
	i	j			

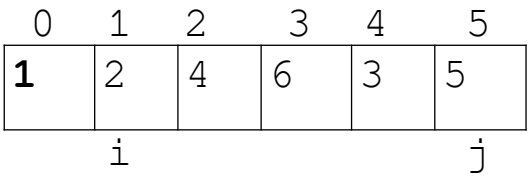
Índice do menor 1

0	1	2	3	4	5
1	2	4	6	3	5
	i		j		

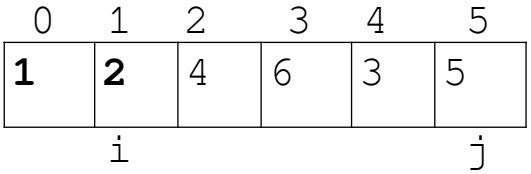
Índice do menor 1



Índice do menor 1

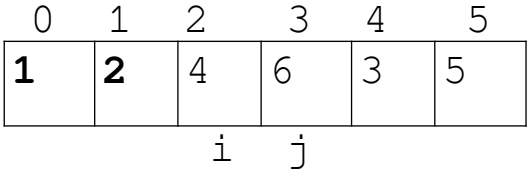


Índice do menor 1

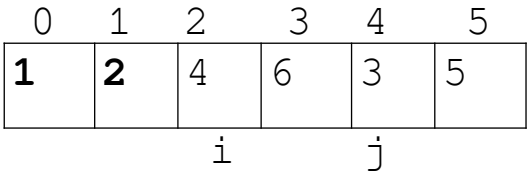


Seleção 3

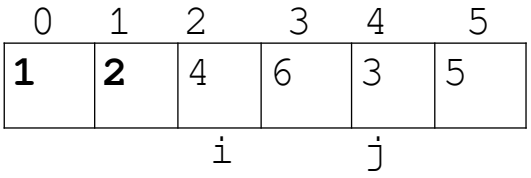
Índice do menor 2



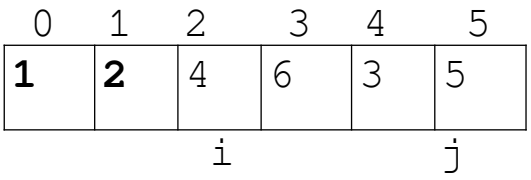
Índice do menor 2



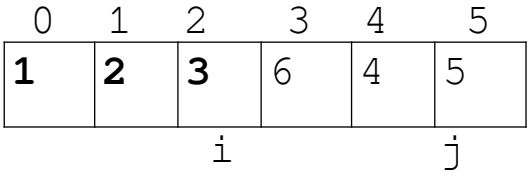
Índice do menor 4



Índice do menor 4

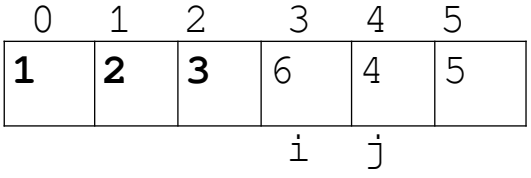


Índice do menor 4 – swap i menor

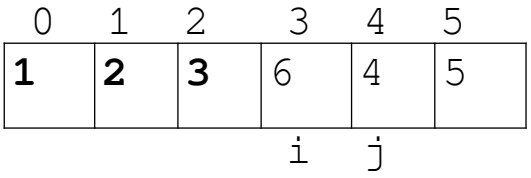


Seleção 4

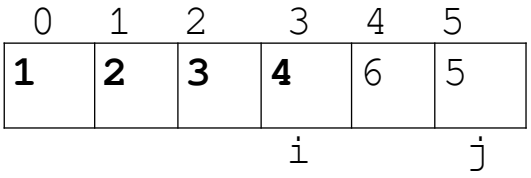
Índice do menor 3



Índice do menor 4

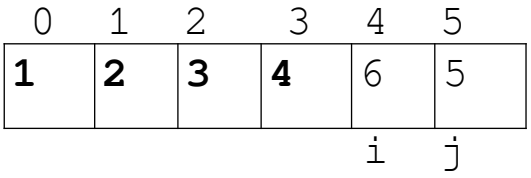


Índice do menor 4 – swap i menor

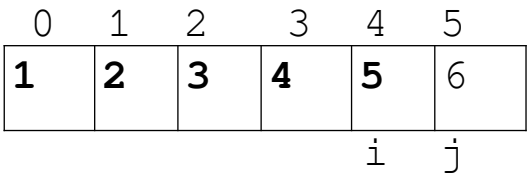


Seleção 4

Índice do menor 4



Índice do menor 5 – swap i menor



Complexidade assintótica?

- Cerca de $\mathbf{N^2/2}$ comparações e **N** trocas
- $\mathbf{O(N^2)}$

Adaptatividade?

- Se o primeiro item já for o menor, implica que não é necessário percorrer o vetor na primeira passada?!

Estabilidade?

- 4 3 4’ 1 → mantém a ordem relativa?
- Tem trocas com saltos?

In-place?

- Utiliza memória extra significativa?
- Copia os conteúdos para outra estrutura de dados?

Selection Sort com listas encadeadas??

Selection Sort estável??

- Não realizar o swap
- Ideia: “abrir” um espaço na posição, “empurrando” os itens para frente
- Boa solução?

Curiosidade

Selection Sort tem tempo linear??

Seja M a proporção entre o tamanho do item e da chave então podemos assumir que o valor de uma comparação é de 1 unidade de tempo e o custo de uma troca é de M unidades de tempo.

Selection sort executa cerca de $N^2/2$ comparações e NM unidades de tempo para trocas. Se **M** for maior do que uma constante múltipla de **N**, então **N*M** domina o $\mathbf{N}^2$, então o tempo é proporcional a N*M, que é proporcional a quantidade de tempo que pode ser requerido para mover todos os dados.

Método elementar: BUBBLE SORT

- Como as bolhas comportam-se?!
- A ideia é percorrer a estrutura diversas vezes
- A cada passagem fazendo flutuar para o topo o maior (ou menor) elemento da sequência
- Comparar chaves adjacentes
- A cada iteração um elemento é posicionado corretamente através de sucessivas trocas de posição

Flutuação 1

3	2	4	6	1	5

j-1 j

3	2	4	1	6	5

j-1 j

3	2	1	4	6	5

j-1 j

3	1	2	4	6	5

j-1 j

1	3	2	4	6	5

j-1 j

Flutuação 2

1	3	2	4	6	5

j-1 j

1	3	2	4	5	6

j-1 j

1	3	2	4	5	6

j-1 j

1	3	2	4	5	6

j-1 j

1	**2**	3	4	5	6

j-1 j

Flutuação 3

1	**2**	3	4	5	6

j-1 j

1	**2**	3	4	5	6

j-1 j

1	**2**	**3**	4	5	6

j-1 j

Flutuação 4

1	**2**	**3**	4	5	6
				j−1	j

1	**2**	**3**	**4**	5	6
			j−1	j	

Flutuação 5

1	**2**	**3**	**4**	**5**	**6**
				j−1	j

Complexidade assintótica

- Cerca de $N^2/2$ comparações e $N^2/2$ trocas
- No melhor caso: O(n) (como??)

Adaptatividade?

- Conjuntos já ordenados (zero movimentações), terão itens comparados?

Estabilidade?

- 2 4 3 4’ 1 → mantém a ordem relativa?
- Tem trocas com saltos?

In-place?

- Utiliza memória extra significativa?
- Copia os conteúdos para outra estrutura de dados?

Não é recomendado para programas que precisem de velocidade e operem com muitos de dados → pior que o selection sort

Método elementar: SHAKER SORT

- Otimização do Bubblesort
- Consiste em realizar uma iteração para colocar o menor elemento em cima e na volta colocar o maior elemento no fundo

Método elementar: INSERTION SORT

- Como organizam uma mão de cartas?
- Ideia:
  - Colocar (inserir) cada elemento na posição correta em relação aos seus antecessores
- Comparação item a item até achar um menor

Inserção 1

---

5 < 6?

6	5	4	3	2	1

j-1 j=i

swap → tem antecessores?

5	6	4	3	2	1

j-1 j=i

Inserção 2

---

4 < 6?

5	6	4	3	2	1

j-1 j=i

swap → tem antecessores?

5	4	6	3	2	1

j-1 j i

swap → tem antecessores?

4	5	6	3	2	1

j-1 j i

Inserção 3

4	5	6	3	2	1

j-1 j=i

4	5	3	6	2	1

j-1 j i

4	3	5	6	2	1

j-1 j i

3	4	5	6	2	1

j-1 j i

Inserção 4

3	4	5	6	2	1

j-1 j=i

3	4	5	2	6	1

j-1 j i

3	4	2	5	6	1

j-1 j i

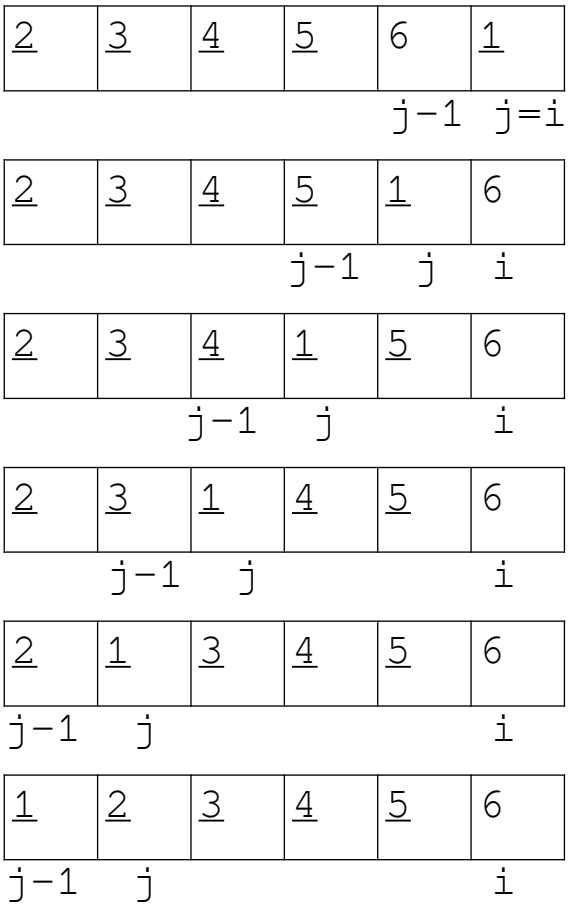
3	2	4	5	6	1

j-1 j i

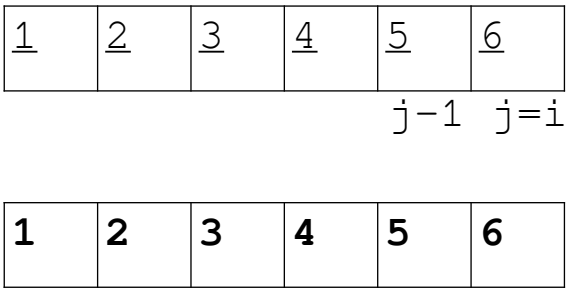
2	3	4	5	6	1

j-1 j i

Inserção 5



Inserção 6



1	**2**	**3**	**4**	**5**	**6**

Complexidade assintótica

- média: **$N^2/4$** comparações e **$N^2/4$** trocas
- pior caso: **$N^2/2$** comparações e **$N^2/2$** trocas
  - Não é indicado para grandes entradas totalmente desordenadas ou invertida
  - Desempenho do Bubble Sort
    - Envolve trocas com somente com os adjacentes
- melhor caso: **N–1** comparações e **0** trocas
  - quando?

Adaptatividade?

- Ordenação existente, diminui as comparações?

1	2	3	5	4	6

j–1 j=i

1	2	3	5	4	6

j–1 j=i

1	2	3	5	4	6

j–1 j=i

1	2	3	5	4	6

j–1 j=i

1	2	3	4	5	6

j–1 j i

1	2	3	4	5	6

j-1 j=i

- Recomendado nos casos em que há ordenação quase completa

Estabilidade?

- Mantém a ordem relativa?
- Tem trocas com saltos?

3	2	2’	1

j-1 j=i

2	3	2’	1

j-1 j=i

2	2’	3	1

j-1 j=i

2	2’	3	1

j-1 j i

2	2’	3	1

j-1 j=i

2	2’	3	1

j-1 j=i

2	2’	1	3

j-1 j i

2	1	2’	3

j-1 j i

1	**2**	**2’**	3

j-1 j i

In-place?

- Utiliza memória extra significativa?
- Copia os conteúdos para outra estrutura de dados?

Insertion Sort x Bubble sort

- Bubble:
  - posiociona os itens do fim para o ínicio
  - o posicionamento de um item não garante a ordenação dos outros elementos
    - garante que os elementos à esquerda sejam menores e à direita maiores
    - não necessariamente ordenados a cada passagem
- Insertion:
  - posiociona os itens do início para o fim
  - o posicionamento de um item garante a ordenação dos elementos a sua esquerda

Insertion Sort x Selection sort

- Selection:
  - Relativo a uma posição atual:
    - itens à esquerda → ordenados e
      - na posição final
- Insertion:
  - Relativo a uma posição atual:
    - itens à esquerda → ordenados mas,
      - podem não estar posição final
      - podem ter que ser movidos para abrir espaço para itens menores

- Tempo de execução depende da ordenação inicial
  - É adaptativo
  - Quanto mais ordenado, mais rápido
    - O tempo tende a linear quanto mais ordenado
    - Selection, continua quadrático

Método elementar: SHELL SORT

- Extensão do algoritmo de ordenação Insertion Sort
- Ideia:
  - Ordenação parcial a cada passagem
  - Posteriormente, eficientemente, ordenados pelo Insertion Sort
- Diminui o número de movimentações
- Troca de itens que estão distantes um do outro
  - Separados a **h** distância
  - São rearranjados, resultando uma sequencia ordenada para a distância h (h-ordenada)
  - Quando h=1, corresponde ao Insertion Sort
  - A dificuldade é determinar o **valor de h**
    - **Donald Knuth** (cientista da computação): recomenda algo em torno de 1/3 da entrada
    - sequencias múltiplas de 2 não performam bem:
      - 1 2 4 8 16 32 64 128 256...
      - itens em posições pares não confrontam itens em posições ímpares até o fim do processo e, vice e versa

- Implementação é muito simples, similar ao algoritmo de inserção

---

h = 1

h = 3*h+1 → alternar pares e ímpares

h = 1, 4, 13, 40, 121, 364, 1093, …

r = 16 → 16/3 ~ 5

h = 1 < 5? (3*1+1) : 1

h = 4 < 5? (3*4+1) : 4

h = 13 < 5? (3*13+1) : 13

---

h = 13

7	4	9	12	6	11	15	5	16	13	3	10	2	8	1	14
j-h												j=i			

no swap → continua

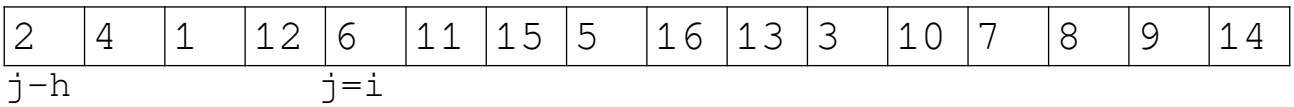
2	4	9	12	6	11	15	5	16	13	3	10	7	8	1	14
	j-h												j=i		

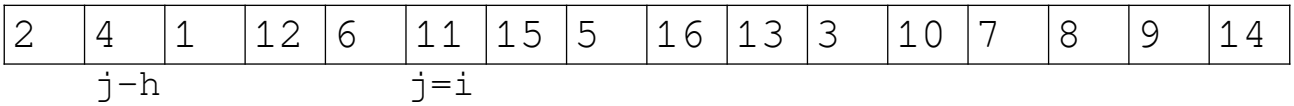
2	4	1	12	6	11	15	5	16	13	3	10	7	8	9	14
		j-h												j=i	

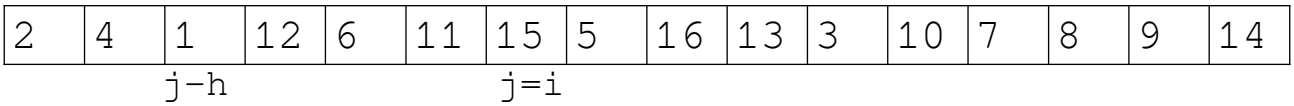
2	4	1	12	6	11	15	5	16	13	3	10	7	8	9	14
			j-h												j=i

---

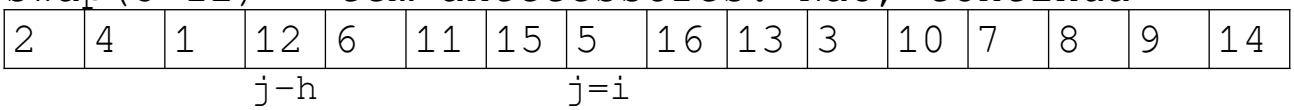
h = 13/3 ~ 4

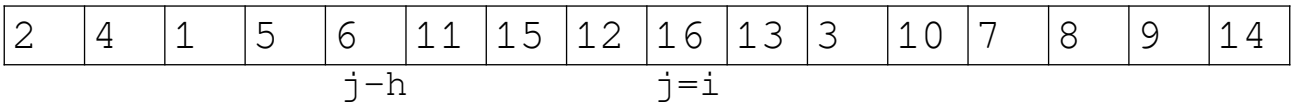


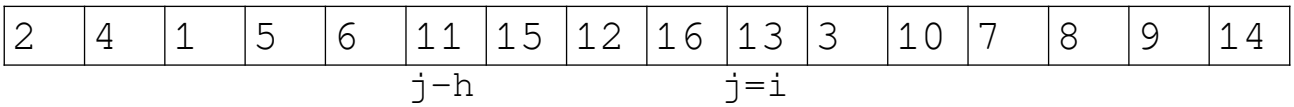




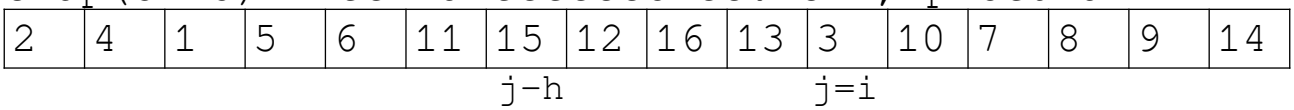
swap(5-12) → tem antecessores? Não, continua



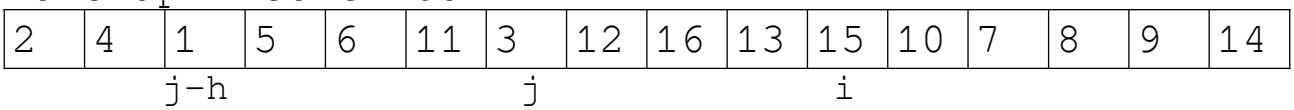




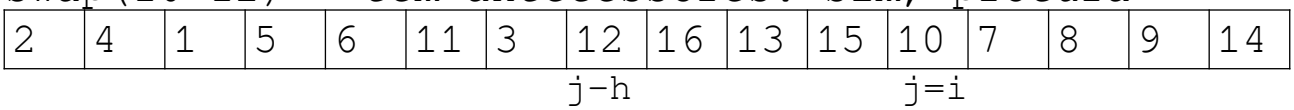
swap(3-15) → tem antecessores? Sim, procura



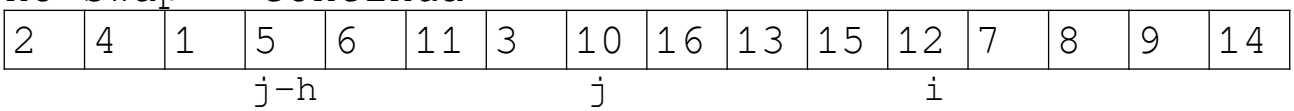
no swap → continua



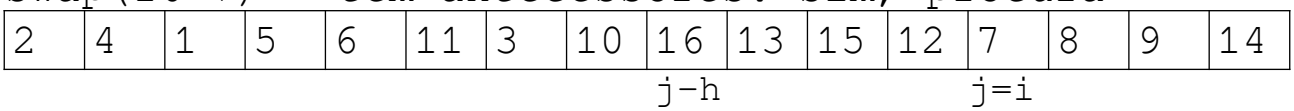
swap(10-12) → tem antecessores? Sim, procura



no swap → continua



swap(16-7) → tem antecessores? Sim, procura



no swap → continua

2	4	1	5	6	11	3	10	7	13	15	12	16	8	9	14
				j-h				j				i			

2	4	1	5	6	11	3	10	7	13	15	12	16	8	9	14
									j-h				j=i		

2	4	1	5	6	11	3	10	7	8	15	12	16	13	9	14
					j-h				j				i		

2	4	1	5	6	8	3	10	7	11	15	12	16	13	9	14
	j-h				j								i		

2	4	1	5	6	8	3	10	7	11	15	12	16	13	9	14
										j-h				j=i	

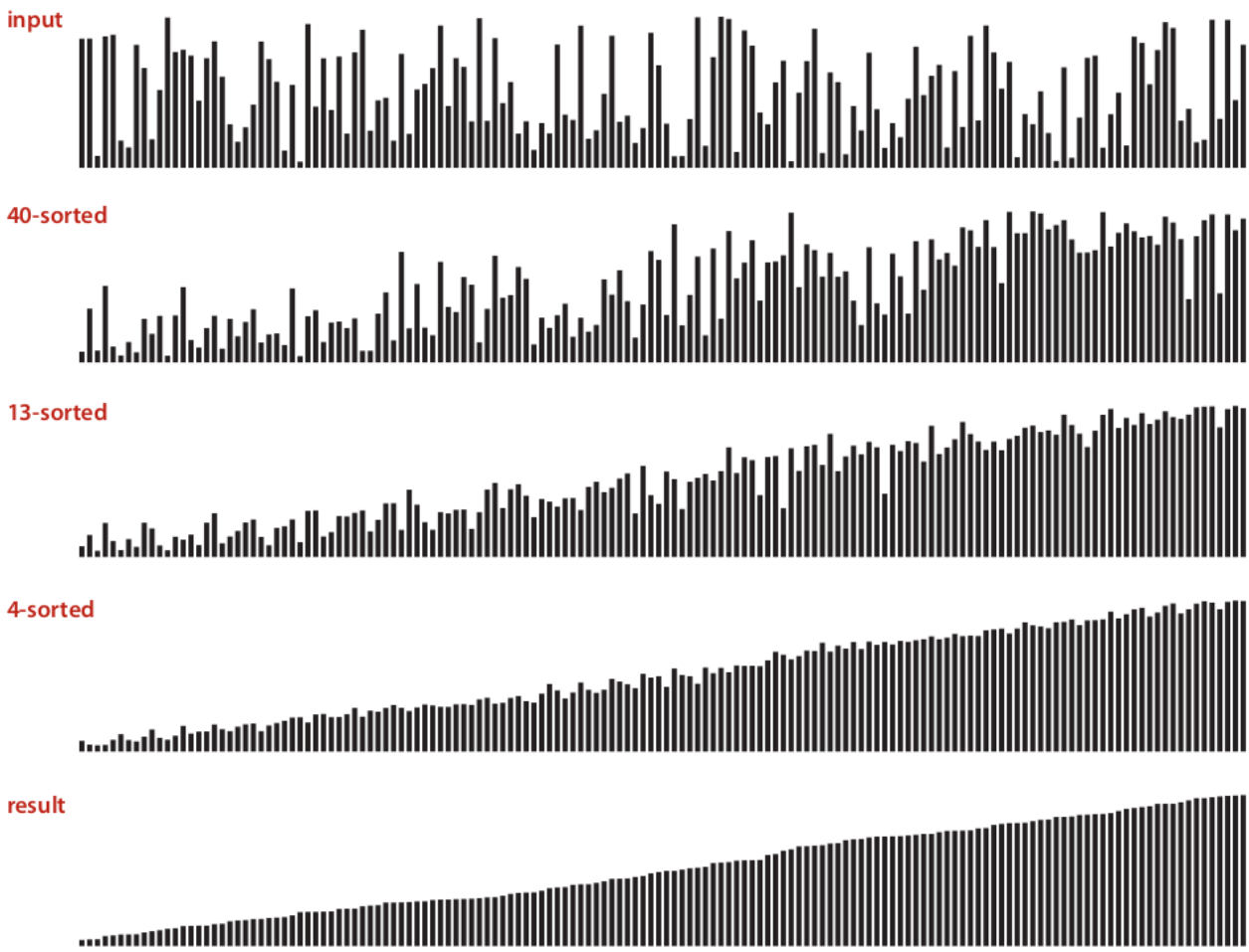
2	4	1	5	6	8	3	10	7	11	9	12	16	13	15	14
						j-h				j				i	

2	4	1	5	6	8	3	10	7	11	9	12	16	13	15	14
											j-h				j=i

---

h = 4/3 ~ 1 → Insertion Sort

2	4	1	5	6	8	3	10	7	11	9	12	16	13	15	14
j-h	j=i														



Visual trace of shellsort

fonte: Algorithms – 4 edição, Robert Sedgewick e Kevin Wayne

Complexidade assintótica

- Tempo de execução: muito sensível à ordem inicial dos elementos
- Cada passagem de k em k, temos um vetor mais ordenado
  - Como é adaptativo, menos comparações serão efetudas
  - Conta com a possibilidade de acertar a posições dos elementos
- Empiricamente, observou-se sua eficiência em diversos casos
- Importante, **no pior caso, shellsort não é necessariamente quadrático (Sedgewick)**
  - As comparações são proporcionais a $\mathbf{N^{3/2}}$
    - Pior caso com pior sequencia de intervalos *h*: $\mathbf{O(n^2)}$
    - Melhor caso com pior sequencia de intervalos *h*: $\mathbf{O(n \log^2 n)}$
      - Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks (Outstanding Dissertations in the Computer Sciences)
- Melhor caso com uma boa sequencia de intervalos *h*: $\mathbf{O(n \log n)}$
- **Caso médio:**
  - Segundo Sedgewick (2011) nenhum resultado matemático estava disponível sobre o número médio de comparações para shellsort para entrada ordenada aleatoriamente

Adaptatividade?

- Ordenação → diminui comparações?

Estabilidade?

- Mantém a ordem relativa?
- Tem trocas com saltos?

In-place?

- Utiliza memória extra significativa?
- Copia os conteúdos para outra estrutura de dados?

Vamos testar