Estrutura de Dados II (ED2)

Aula 10 – Métodos Elementares de Ordenação

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (UFES)

(Material baseado nos slides do Professor Eduardo Zambon)

Introdução

- Métodos de ordenação são essenciais nas mais diferentes aplicações.
- **Aula de hoje**: apresentação de alguns métodos clássicos de ordenação e suas principais características.
- **Objetivos**: compreender o funcionamento dos métodos básicos de ordenação *selection sort*, *insertion sort* e *shell sort*, e analisar o seu desempenho.

Referências

Chapter 6 – Elementary Sorting Methods
R. Sedgewick

Regras do Jogo

Problema de ordenação

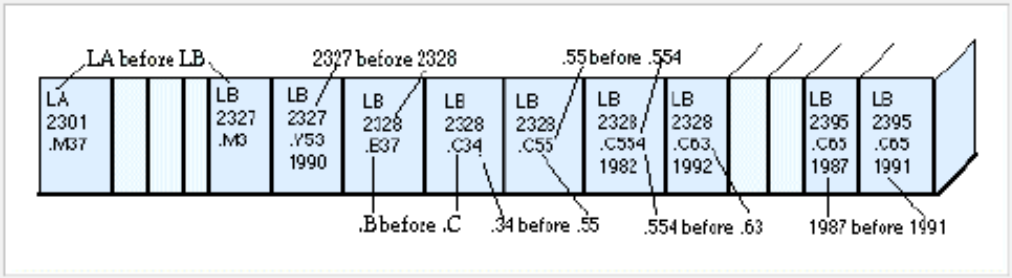
Exemplo: ordenar os registros em um banco de dados.

	Chen	3	A	991-878-4944	308 Blair
	Rohde	2	A	232-343-5555	343 Forbes
	Gazsi	4	B	766-093-9873	101 Brown
item →	Furia	1	A	766-093-9873	101 Brown
	Kanaga	3	B	898-122-9643	22 Brown
	Andrews	3	A	664-480-0023	097 Little
key →	Battle	4	C	874-088-1212	121 Whitman

Sort: Rearranjar um *array* de *N* itens em ordem crescente.

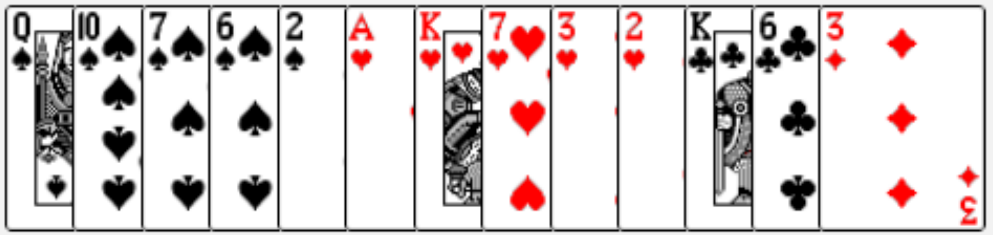
Andrews	3	A	664-480-0023	097 Little
Battle	4	C	874-088-1212	121 Whitman
Chen	3	A	991-878-4944	308 Blair
Furia	1	A	766-093-9873	101 Brown
Gazsi	4	B	766-093-9873	101 Brown
Kanaga	3	B	898-122-9643	22 Brown
Rohde	2	A	232-343-5555	343 Forbes

Aplicações de ordenação

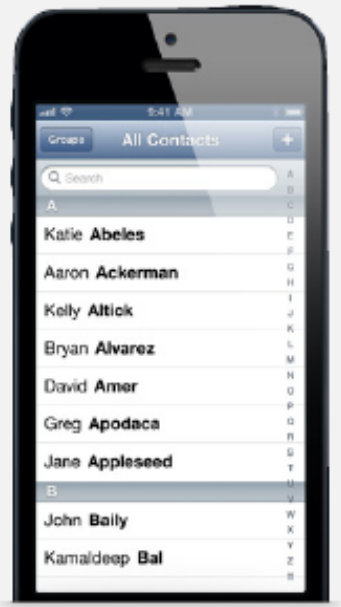


Library of Congress numbers

FedEx packages



playing cards



contacts

Hogwarts houses

Relação de ordem (*total order*)

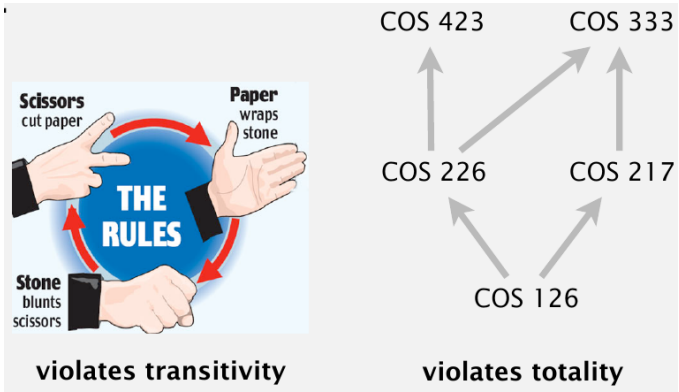
Objetivo: Ordenar qualquer tipo de dado (para o qual a ordenação é bem definida).

Uma relação de ordem (*total order*) é uma relação binária $\leq$ que satisfaz:

- Anti-simetria: se ambos $v \leq w$ e $w \leq v$, então $v = w$.
- Transitividade: se ambos $v \leq w$ e $w \leq x$, então $v \leq x$.
- Totalidade: ou $v \leq w$ ou $w \leq v$ ou ambos.

Exemplos de relação de ordem:

- Ordem padrão para números naturais e reais.
- Ordem cronológica para datas e horas.
- Ordem alfabética para *strings*.



Callbacks

Objetivo: Ordenar qualquer tipo de dado (para o qual a ordenação é bem definida).

Q: Como uma mesma função `sort()` pode ser usada para ordenar inteiros, reais, *strings*, etc?

Callback: referência para um código executável.

- O cliente passa um *array* de itens para a função `sort()`.
- A função `sort()` chama (*calls back*) os códigos de comparação dos itens conforme necessário.

Implementando *callbacks*:

- Python, ML, Javascript: funções de primeira classe.
- Java: interfaces.
- C: ponteiros para função, ou macros.

Macros para interface de itens

Comparação de inteiros:

```
typedef int Item;
#define key(A)           (A)
#define less(A, B)       (key(A) < key(B))
#define exch(A, B)       { Item t = A; A = B; B = t; }
#define compexch(A, B)   if (less(B, A)) exch(A, B)
```

Comparação de *strings*:

```
typedef char* Item;
#define key(A)           (A)
#define less(A, B)       (strcmp(A,B) < 0)
```

Comparação de estruturas:

```
typedef struct {
  int src, tgt, dist;
} Edge;

#define key(A)       (A.dist)
#define less(A, B)   (key(A) < key(B))
...
```

Selection Sort

Selection sort demo

Ideia geral:

- Na iteração `i`, encontre o índice `min` da menor chave restante.
- Troque `a[i]` e `a[min]`.

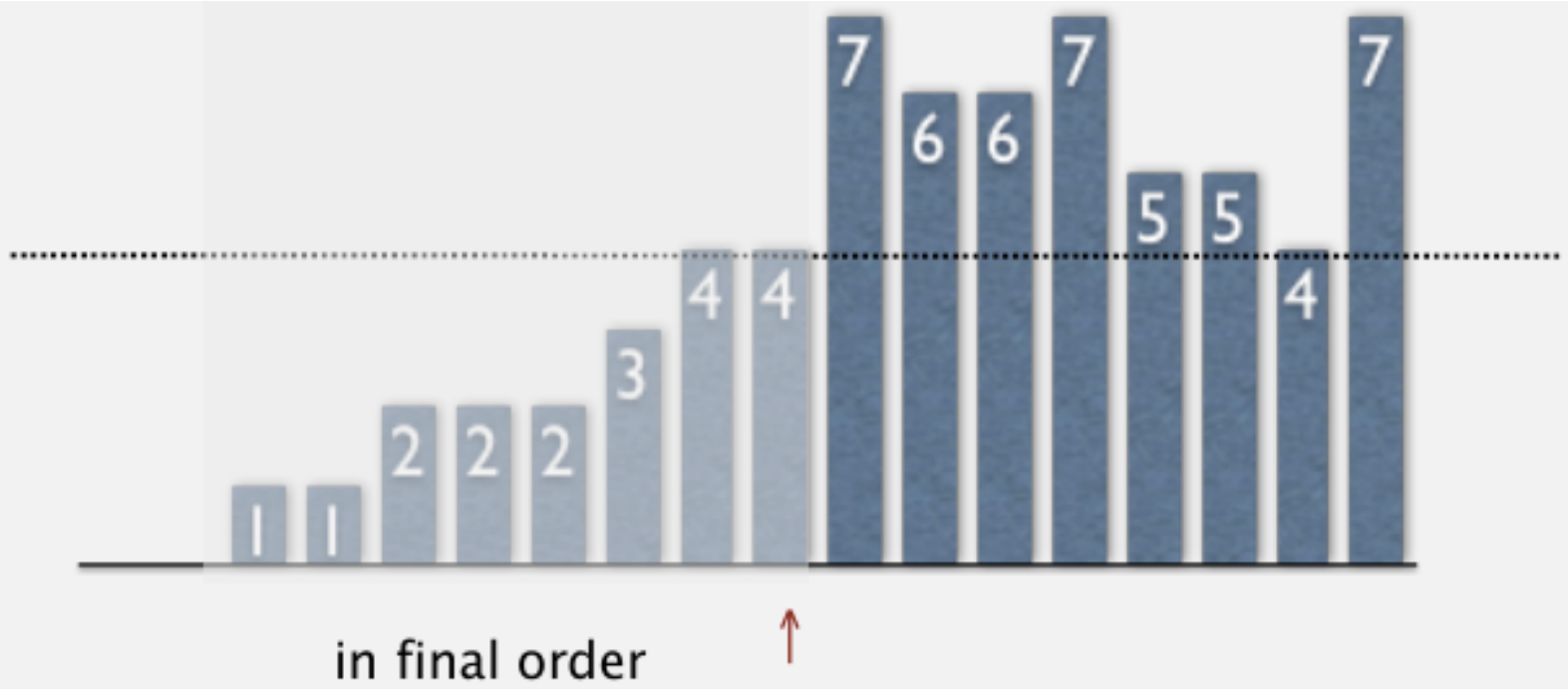
Ver arquivo `21DemoSelectionSort.mov`

Selection sort

Algoritmo: ↑ varre o *array* da esquerda para a direita.

Invariantes:

- Chaves à esquerda de ↑ (inclusive) estão fixas e em ordem não-decrescente (ascendente).
- Nenhuma chave à direita de ↑ é menor do que as chaves à esquerda de ↑.



Loop mais interno do *selection sort*

Para manter os invariantes:

Mova o índice para a direita.

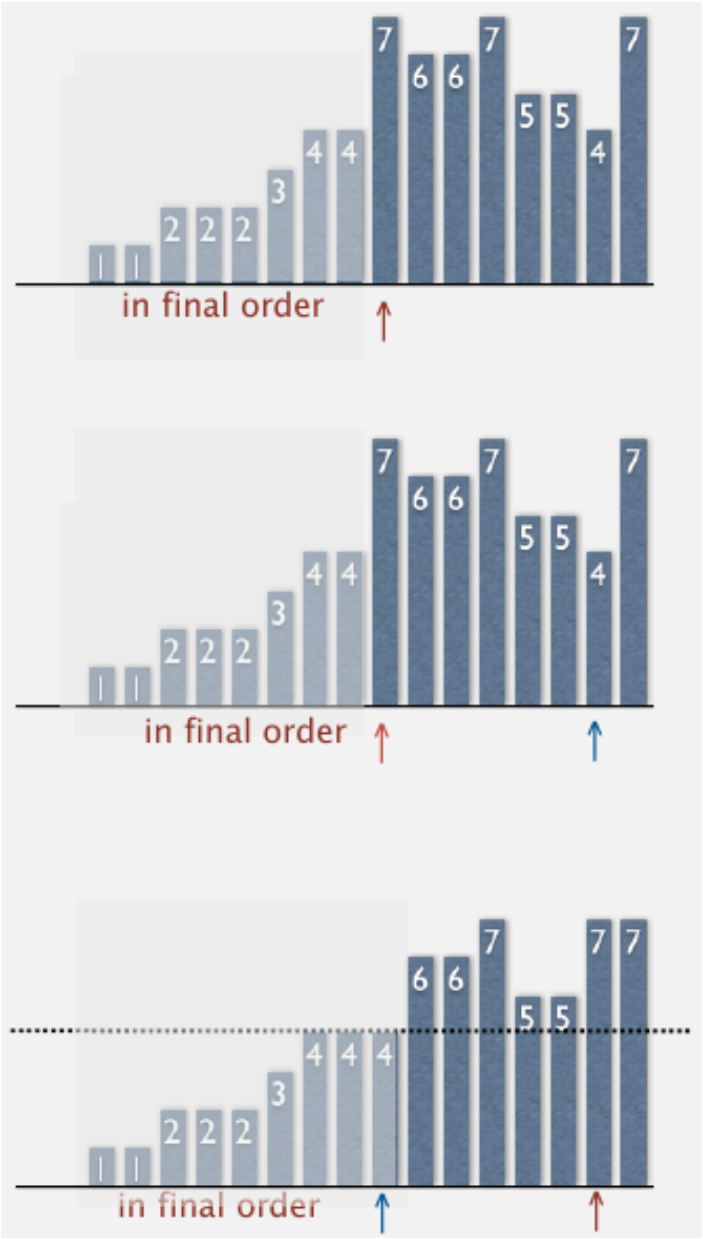
```
i++;
```

Encontre o índice da menor chave à direita.

```
int min = i;
for (int j = i+1; j <= hi; j++)
    if (less(a[j], a[min]))
        min = j;
```

Troque as posições.

```
exch(a[i], a[min]);
```



Selection sort: implementação em C

Tendo definido o tipo `Item` e suas operações, a função de ordenação fica como abaixo.

```
void sort(Item *a, int lo, int hi) {
    for (int i = lo; i < hi; i++) {
        int min = i;
        for (int j = i+1; j <= hi; j++) {
            if (less(a[j], a[min])) {
                min = j;
            }
        }
        exch(a[i], a[min]);
    }
}
```

Para usar: `sort(a, 0, N-1);`

Veja as animações em:
`https://www.toptal.com/developers/sorting-algorithms/selection-sort`

Selection sort: análise matemática

Selection sort usa $(N-1)+(N-2)+\ldots+1+0 \sim N^2/2$ comparações e N trocas.

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R
5	7	A	E	E	L	M	X	S	O	P	T	R
6	8	A	E	E	L	M	O	S	X	P	T	R
7	10	A	E	E	L	M	O	P	X	S	T	R
8	8	A	E	E	L	M	O	P	R	S	T	X
9	9	A	E	E	L	M	O	P	R	S	T	X
10	10	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in black are examined to find the minimum

entries in red are a[min]

entries in gray are in final position

Trace of selection sort (array contents just after each exchange)

Tempo de execução é insensível à entrada. Tempo quadrático, mesmo se a entrada estiver ordenada.

Movimento dos dados é mínimo. Número linear de trocas.

Selection sort: ordem de crescimento

Ordem de crescimento do tempo de execução para ordenar um *array* de N itens.

Algoritmo	Melhor caso	Caso médio	Pior caso
Selection sort	N^2	N^2	N^2

Insertion Sort

Insertion sort demo

Ideia geral:

- Na iteração `i`, troque `a[i]` com o maior valor à sua esquerda.

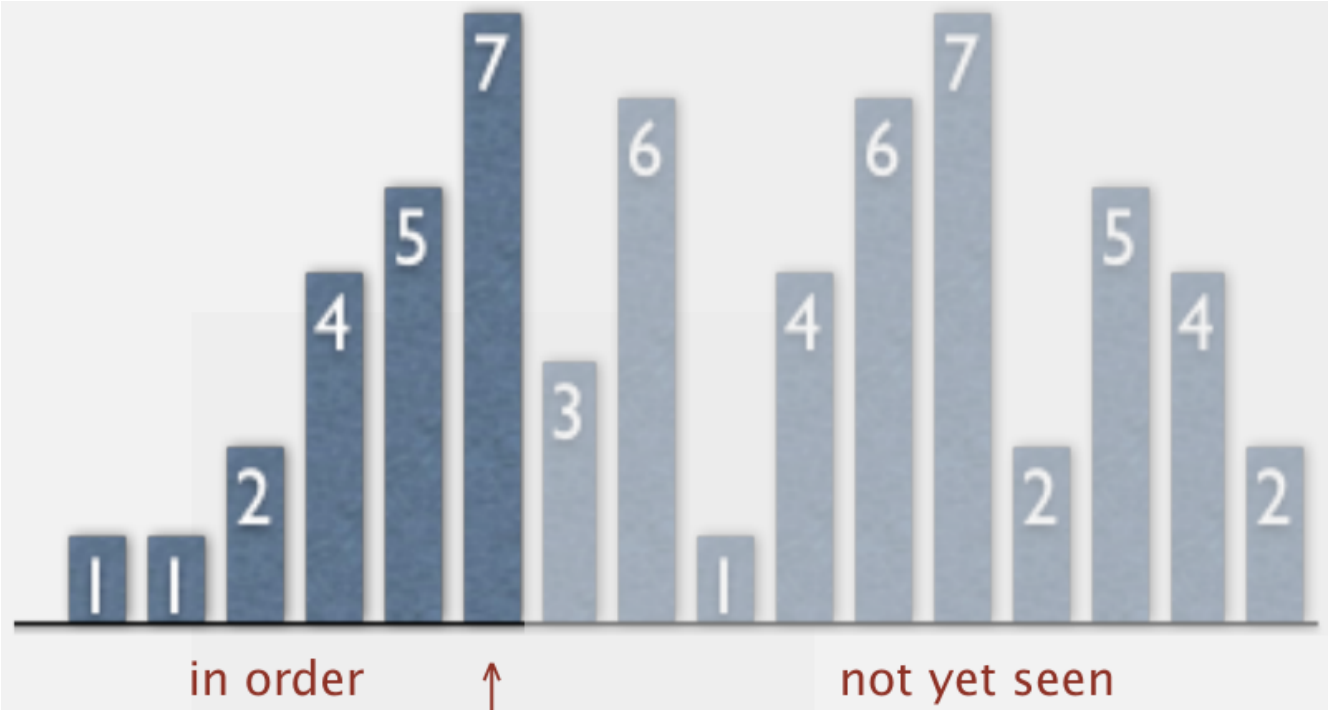
Ver arquivo `21DemoInsertionSort.mov`

Insertion sort

Algoritmo: ↑ varre o *array* da esquerda para a direita.

Invariantes:

- Chaves à esquerda de ↑ (inclusive) estão em ordem não-decrescente (ascendente).
- Chaves à direita de ↑ ainda não foram vistas.



Loop mais interno do *insertion sort*

Para manter os invariantes:

Mova o índice para a direita.

```
i++;
```

Movendo da direita para esquerda, troque `a[i]` todas as chaves maiores à esquerda.

```
for (int j = i; j > lo; j--)
    compexch(a[j-1], a[j]);
```

Insertion sort: implementação em C

Tendo definido o tipo `Item` e suas operações, a função de ordenação fica como abaixo.

```
void sort(Item *a, int lo, int hi) {
    for (int i = lo+1; i <= hi; i++) {
        for (int j = i; j > lo; j--) {
            compexch(a[j-1], a[j]);
        }
    }
}
```

Para usar: `sort(a, 0, N-1);`

Veja as animações em:
`https://www.toptal.com/developers/sorting-algorithms/insertion-sort`

Insertion sort: análise matemática

Para ordenar um *array* com chaves aleatórias (todas distintas), *insertion sort* faz $\sim \frac{1}{4}N^2$ comparações e $\sim \frac{1}{4}N^2$ trocas.

Intuição: Cada chave (em média) deve se mover metade do caminho para trás.

		a[]										
i	j	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
1	0	O	S	R	T	E	X	A	M	P	L	E
2	1	O	R	S	T	E	X	A	M	P	L	E
3	3	O	R	S	T	E	X	A	M	P	L	E
4	0	E	O	R	S	T	X	A	M	P	L	E
5	5	E	O	R	S	T	X	A	M	P	L	E
6	0	A	E	O	R	S	T	X	M	P	L	E
7	2	A	E	M	O	R	S	T	X	P	L	E
8	4	A	E	M	O	P	R	S	T	X	L	E
9	2	A	E	L	M	O	P	R	S	T	X	E
10	2	A	E	E	L	M	O	P	R	S	T	X
		A	E	E	L	M	O	P	R	S	T	X

entries in gray do not move

entry in red is `a[j]`

entries in black moved one position right for insertion

Trace of insertion sort (array contents just after each insertion)

Insertion sort: análise matemática

Análise do slide anterior: caso médio.

Melhor caso: se o *array* já está ordenado, *insertion sort* faz $N - 1$ comparações e 0 trocas.

Pior caso: se o *array* está em ordem inversa, *insertion sort* faz $\sim {}^1\!/_2 N^2$ comparações e $\sim {}^1\!/_2 N^2$ trocas.

Insertion sort: *arrays* parcialmente ordenados

Definição: Uma inversão é um par de chaves que estão fora de ordem. A sequência

A E E L M O T R X P S

possui 6 inversões:

T–R T–P T–S R–P X–P X–S .

Definição: Um *array* está parcialmente ordenado se o seu número de inversões é $\leq cN$.

- No melhor caso o *array* já está ordenado e o número de inversões é zero.

Proposição: Para *arrays* parcialmente ordenados, *insertion sort* executa em tempo linear.

- O número de trocas é igual ao número de inversões.
- Número de comparações = # trocas $+(N-1)$.

Insertion sort: melhorias práticas

1. Iniciar colocando o menor elemento no começo do *array* para servir de sentinela.
2. Fazer uma atribuição ao invés de uma troca no *loop* interno.
3. Terminar o *loop* interno quando o elemento a ser inserido está em posição.

```
void sort(Item *a, int lo, int hi) {
    for (int i = hi; i > lo; i--)
        compexch(a[i-1], a[i]);
    for (int i = lo+2; i <= hi; i++) {
        int j = i;
        Item v = a[i];
        while (less(v, a[j-1])) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = v;
    }
}       // Ordem de crescimento continua igual.
```

Insertion sort: ordem de crescimento

Ordem de crescimento do tempo de execução para ordenar um *array* de N itens.

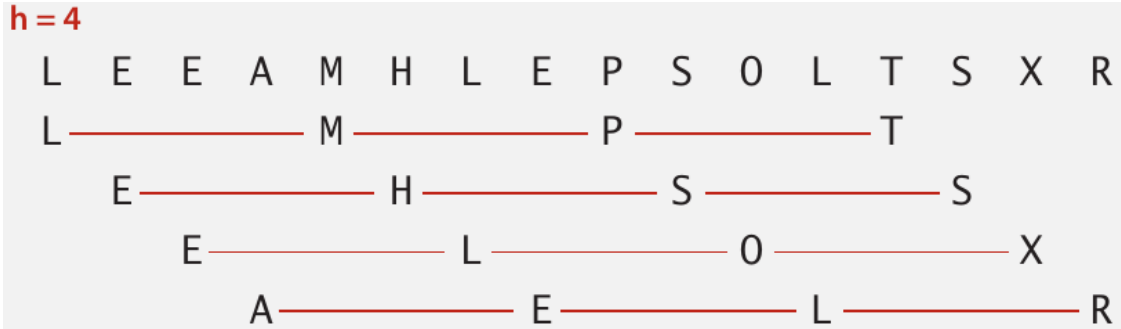
Algoritmo	Melhor caso	Caso médio	Pior caso
Selection sort	N^2	N^2	N^2
Insertion sort	N	N^2	N^2

Shell Sort

Shell sort: visão geral

Ideia:

- Mover as chaves mais de uma posição por vez fazendo *h-sorting* do *array*.
- Um *array h-sorted* é formado por *h* subsequências ordenadas intercaladas.



Shell sort [Shell 1959]: faça um *h-sort* do *array* para uma sequência decrescente de valores de *h*.

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

h-sorting demo

Ideia geral:

- Na iteração `i`, troque `a[i]` com o maior valor à sua esquerda distante *h* posições.

Ver arquivo `21DemoShellSort.mov`

h-sorting

Q: Como fazer um *h-sort* de um *array*?

A: *Insertion sort* com passo de tamanho *h*.

Por que *insertion sort*?

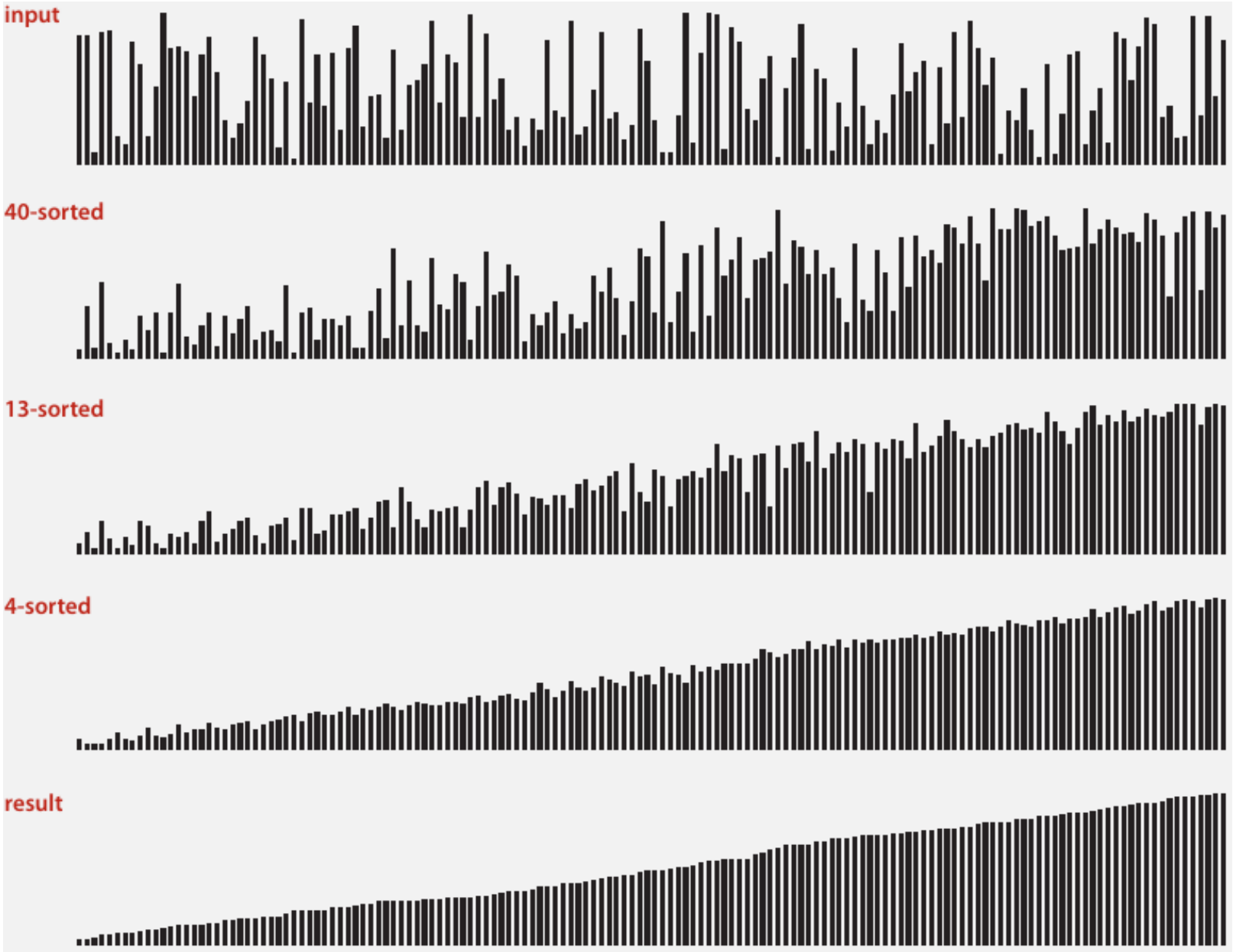
- Porque é um método que se adapta bem ao problema.
- Incrementos grandes $\Rightarrow$ *subarray* pequeno.
- Incrementos pequenos $\Rightarrow$ quase ordenado.

Shell sort exemplo: incrementos 7, 3, 1

Shell sort: implementação em C

```
void sort(Item *a, int lo, int hi) {
    int h = 1;
    while (h < (hi-lo)/9) { // 3x+1 increment sequence
        h = 3*h + 1;         // 1, 4, 13, 40, 121, 364, ...
    }
    while (h > 0) {// h-sort the array.
        for (int i = lo+h; i <= hi; i++) {
            int j = i;       // Insertion sort
            Item v = a[i];
            while (j >= lo+h && less(v, a[j-h])) {
                a[j] = a[j-h];
                j -= h;
            }
            a[j] = v;
        }
        h /= 3; // Move to next increment
    }
}
```

Shell sort: visualização



Shell sort: animação

Veja as animações em:
`https://www.toptal.com/developers/sorting-algorithms/shell-sort`

Shell sort: qual sequência de incrementos usar?

Potência de dois: 1, 2, 4, 8, 16, 32, ...

- Originalmente proposta por Shell em 1959.
- Ruim. Elementos nas posições pares e ímpares só são comparados na última passada.

Potência de dois menos um: 1, 3, 7, 15, 31, 63, ...

- Talvez. Mas há sequências melhores.

$3x + 1$: 1, 4, 13, 40, 121, 364, ...

- OK. Fácil de computar.
- Proposta por Knuth em 1969.

Sedgewick: 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

- Boa. Difícil de bater em experimentos empíricos.
- Mais difícil de calcular.

Shell sort: análise matemática

Pior caso:

- A ordem de crescimento para o pior caso do número de comparações do *shell sort* depende da sequência de incremento.
- Para *shell sort* com $3x + 1$ incrementos a ordem é $N^{3/2}$.
- Para outros incrementos melhores, a ordem pode cair para $N^{4/3}$ ou até mesmo $N \lg^2 N$.

Melhor caso:

- Já foi provado que o limite inferior do melhor caso é $N \log N$.
- Prova utiliza matemática "esotérica": complexidade de Kolmogorov.
- Fala qual é o limite, mas não a sequência de incrementos que leva a esse caso.

Shell sort: análise matemática

Caso médio: O número esperado de comparações que o *shell sort* faz para ordenar um *array* aleatório é...

N	compares	2.5 N ln N	0.25 N ln ^{2} N	N $^{1.3}$
5,000	93K	106K	91K	64K
10,000	209K	230K	213K	158K
20,000	467K	495K	490K	390K
40,000	1022K	1059K	1122K	960K
80,000	2266K	2258K	2549K	2366K

Um modelo matemático preciso ainda não foi descoberto (!)

Por que estamos interessados no *shell sort*?

Exemplo de uma ideia simples que leva a ganhos de desempenho consideráveis.

Útil na prática.

- Rápido para *arrays* não absurdamente grandes.
- Implementação simples, código enxuto.
- Utilizado em variadas aplicações: `bzip2`, `uClibc`, *kernel* do Linux.

Questões em aberto.

- Taxa de crescimento assintótica?
- Melhor sequência de incrementos?
- Desempenho do caso médio?

Lição: Ainda há bons algoritmos esperando para serem descobertos.

Sumário: ordem de crescimento

Ordem de crescimento do tempo de execução para ordenar um *array* de N itens.

Algoritmo	Melhor caso	Caso médio	Pior caso
Selection sort	N^2	N^2	N^2
Insertion sort	N	N^2	N^2
Shell sort $(3x+1)$	$N \log N$	?	$N^{3/2}$
Objetivo	N	$N \log N$	$N \log N$

Em breve: algoritmos de ordenação $N \log N$ (no pior caso).

Sumário: análise empírica

Tempo de execução dos algoritmos em segundos para entradas de tamanho 10^i, $i = 3, 4, 5$.

RANDOM NUMBERS

N	1K	10K	100K
select	0.006	0.171	16.626
insert	0.003	0.084	7.905
shell	0.000	0.002	0.029

SORTED

N	1K	10K	100K
select	0.005	0.170	16.606
insert	0.000	0.000	0.000
shell	0.000	0.001	0.005

REVERSE SORTED

N	1K	10K	100K
select	0.005	0.162	15.612
insert	0.005	0.161	15.839
shell	0.000	0.001	0.008

NEARLY SORTED

N	1K	10K	100K
select	0.005	0.169	16.653
insert	0.000	0.000	0.001
shell	0.000	0.000	0.005