

SEMANAS:

- INICIO/APRESENTAÇÃO
- INTRODUÇÃO AOS ALGORITMOS E ESTRUTURAS DE DADOS
- ESTRUTURAS DE CONTROLO DE FLUXO EM C#
- CONCEITO DE CLASSE E OBJETO
- SEPARAÇÃO ENTRE INTERFACE PÚBLICA E IMPLEMENTAÇÃO PRIVADA
- ESTRUTURAS DE DADOS – INTRODUÇÃO
- ANÁLISE DE ALGORITMOS E COMPLEXIDADE
- ALGORITMOS DE ORDENAÇÃO
- ALGORITMOS DE ORDENAÇÃO (PARTE 2)
- PROVA DE AVALIAÇÃO
- LISTAS E FILAS
- TABELAS DE HASH
- TRATAMENTO DE COLISÕES
- ÁRVORES BINÁRIAS
- TIPOS GENÉRICOS E TRATAMENTO DE EXCEPÇÕES
- PROVA DE AVALIAÇÃO

Sumário:

- Noção de recursividade.
- Algoritmos de ordenação não elementares:
 - ShellSort
 - QuickSort
- Preparação para prova.

Algoritmos e Estruturas de Dados

CTeSP - Tecnologias e Programação de Sistemas de Informação 2020/21



Docente: [Ricardo Henriques](#).

Algoritmos de ordenação não elementares

(continuação)

ShellSort

Como se estudou, o *insertion sort* é lento e tal deve-se a que apenas envolve trocas entre items adjacentes. É com base nesta observação que o *shell sort* se baseia. Como acelerar o algoritmo? Como permitir trocas entre elementos que estão afastados?

A ideia por detrás deste algoritmo passa por rearranjar os dados de forma a que tenham a propriedade que olhando para cada h -ésimo elemento estão ordenados:

- Dados dizem-se *h-ordenados*;
- É equivalente a h sequências ordenadas entreligadas (entrelaçadas).
- Usando valores de h grandes é possível mover grandes "distâncias" elementos, o que torna mais fácil *h-ordenar* mais tarde com h pequenos:
 - Usando este procedimento para qualquer sequência de h 's que termine em 1 vão produzir um ficheiro ordenado;
 - Cada passo torna o próximo mais simples.

Vamos exemplificar pictoricamente para uma tabela de entrada com 15 elementos e $h=4$:

- A tabela é dividida em partições, cada uma contendo os objectos de índice $\%h$, exemplo:
 - indices de resto 0, $i \% 4 = 0$, são: 0, 4, 8, 12
 - indices de resto 1, $i \% 4 = 1$, são: 1, 5, 9, 13
 - indices de resto 2, $i \% 4 = 2$, são: 2, 6, 10, 14
 - indices de resto 3, $i \% 4 = 3$, são: 3, 7, 11



A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S	O	R	E	I	N	G	T	X	A	M	P	L	E
A	S	O	R	E	I	N	G	P	X	A	M	T	L	E

$$i \% 4 = 0 \\ 0, 4, 8, 12$$

A	I	O	R	E	S	N	G	P	X	A	M	T	L	E
A	I	O	R	E	S	N	G	P	X	A	M	T	L	E
A	I	O	R	E	L	N	G	P	S	A	M	T	X	E

$$i \% 4 = 1 \\ 1, 5, 9, 13$$

A	I	N	R	E	L	O	G	P	S	A	M	T	X	E
A	I	A	R	E	L	N	G	P	S	O	M	T	X	E
A	I	A	R	E	L	E	G	P	S	N	M	T	X	O

$$i \% 4 = 2 \\ 2, 6, 10, 14$$

A	I	A	G	E	L	E	R	P	S	N	M	T	X	O
A	I	A	G	E	L	E	M	P	S	N	R	T	X	O

$$i \% 4 = 3 \\ 3, 7, 11$$

Neste caso foi definido um número fixo para h , mas é comum ter implementações em que o próprio h é variável, na maior parte das vezes por valores de sequências geométricas. Acredita-se porém que a sequência óptima ainda não foi encontrada.

```
public static class ShellSort {
    public static string[] Sort(string[] vector) {
        int h = 1;
        while (h < vector.Length / 3)
            h = 3 * h + 1; //1, 4, 13, 40, 121, 364, 1093, ...
        while (h >= 1) {
            for (int i = h; i < vector.Length; i++) {
                for (int j = i; j >= h &&
                    MetodosAuxiliares.Less(vector[j], vector[j - h]); j -= h) {
                    MetodosAuxiliares.Swap(ref vector[j], ref vector[j - h]);
                }
            }
            h = h / 3;
        }
        return vector;
    }
}
```

- Da mesma forma a análise do algoritmo é desconhecida, dado que ninguém encontrou a fórmula que define a complexidade e a mesma depende da sequência, exemplos:

- o 1, 4, 13, 40, 121, 364, 1093, 3280, ... ($3 \cdot h_{ant} + 1$)
- o 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... (2^i)
- o 1, 8, 23, 77, 281, 1073, 4193, ($4^{i+1} + 3 \cdot 2^{i+1}$)

Em resumo, no *Shell Sort*:

- Faz menos do que $O(N^{3/2})$ quando utilizados os incrementos 1, 4, 13, 40, 121, 364, 1093, ...
- Faz menos do que $O(N^{4/3})$ quando utilizados os incrementos de 1, 8, 23, 77, 281, 1073, 4193,
- O código é rápido e eficiente;
- O algoritmo implementa-se em pouco código;
- É o melhor método para entradas pequenas e médias.
- Com um desempenho aceitável para grandes volumes;
- É muito utilizado na prática, mas difícil de compreender.

QuickSort

É considerado o algoritmo mais utilizado, apesar da sua origem remontar aos anos 60 do século passado. O seu desempenho é bem conhecido e a sua fácil implementação e eficiência tornam-o popular. A ideia por detrás deste algoritmo passa por:

- Efetuar a partição dos dados e ordenar as várias partes independentemente (de forma recursiva);
- O posicionamento da partição a efetuar depende dos dados de entrada;
- O processo de partição é crítico;
- O algoritmo, por natureza, é recursivo:
 - Uma vez efetuada a partição, cada uma das partes pode, por sua vez, ser ordenada pelo mesmo algoritmo (o que implica nova partição dos dados).

Transpondo para um algoritmo:

1. Escolhe-se um elemento designado *pivot*;
2. Faz-se a divisão - reordenar o vector de forma que todos os elementos com valores inferiores ao *pivot* fiquem "antes" dele e (iguais) ou maiores "depois". Assim considerando os índices do vector *v* entre [1..r] tem-se:
 - i. o elemento *v[i]*, para algum *i* fica, após a partição, na sua posição final;
 - ii. nenhum dos elementos em *v[1]..v[i-1]* é maior que *v[i]*
3. Aplicar recursivamente os passos acima aos subvectores separadamente - o caso de paragem da recursividade será sempre quando um vector atinge 1 ou 0 elementos.

```
public static string[] Sort(string[] vector) {
    SortRecursive(ref vector, 0, vector.Length - 1);
    return vector;
}
private static void SortRecursive(ref string[] vector, int lo, int hi) {
    int p;
    if (hi <= lo) return;
    p = Partition(vector, lo, hi);
    SortRecursive(ref vector, lo, p - 1);
    SortRecursive(ref vector, p + 1, hi);
}
```

Porque o processo de partição sempre fixa um item na sua posição, pode-se fazer a prova por indução que o método recursivo constitui uma ordenação. Se a componente esquerda do vector e a componente direita do vector ficarem ambos ordenados, então o resultado é que o vector está ordenado.



```

private static int Partition(string[] vector, int lo, int hi) {
    string pivot = vector[lo]; // ou hi, tanto vale!
    while (true) {
        // procura à esquerda
        while (MetodosAuxiliares.Less(vector[lo], pivot)) lo++;
        // procura à direita
        while (MetodosAuxiliares.Less(pivot, vector[hi])) hi--;
        // se já procurou e se for necessário troca
        if (lo < hi && vector[lo] != vector[hi]) {
            MetodosAuxiliares.Swap(ref vector[lo], ref vector[hi]);
        } else {
            return hi;
        }
    }
}

```

Neste código usa-se a estratégia mais comum de escolher um elemento arbitrário para ser o *item de partição*. De seguida trabalha-se na parte esquerda do vector até encontrar um valor maior (ou igual) ao *item de partição*. De imediato ataca-se a parte direita até que se encontre um valor menor (ou igual) ao *item de partição*.

Os dois elementos que param o "tratamento" poderão estar na ordem inversa; se for esse o caso trocam-se.

Em resumo, no *Quick Sort*:

- A eficiência do processo de ordenação depende de quanto bem a partição divide os dados - será tanto mais equilibrada quanto mais perto este elemento estiver do meio da tabela na sua posição final.
- No pior caso usa cerca de $N^2/2$ comparações - uma vez que, se os dados de entrada estiverem ordenados (crescente ou decrescente), todas as partições degeneram e o programa chama-se a si próprio N vezes; o número de comparações é de:
 $N+((N-1)+(N-2)+...+2+1)=(N+1)N/2$
- No caso de cada partição dividir a entrada em metade o número de comparações usadas satisfaz a recursão de dividir para reinar, sendo o custo de **$N \log N$** .
- Dada a recursividade (se for aplicada), não é apenas o tempo necessário à execução do algoritmo que cresce quadraticamente, é também o espaço necessário (memória) para o processo, o que dificulta em situações em que os dados de entrada sejam extensos.

O *QuickSort* tem versões iterativas (não recursivas) e um conjunto de melhoramentos que optimizam o desempenho, por exemplo por melhor escolha do elemento de partição. Neste curso não iremos detalhar mais, mas pode encontrar os mesmos na documentação da bibliografia.

Exercícios de aprendizagem

Implemente dois *dry-runnings*, um sobre o algoritmo *QuickSort* e outro para o *ShellSort*. Utilize como exemplo o vector:



```
string[] nomes = { "Serafim", "Manuel", "Frederica", "Ana", "Luís", "Maria",
"Carlos", "Abel" };
```

Exercício 9: exercício acompanhado durante a aula

Apresenta-se aqui um conjunto de exercícios de grau de dificuldade semelhante aos que poderiam surgir numa prova de avaliação individual.

Durante a aula, resolva apenas o exercício A

- A. Constitua um sistema em programação OO capaz de gerir o histórico das intervenções veterinárias (simplificada: registo, edição, remoção) sobre animais registados na clínica, em que:
 - Cada intervenção é caracterizada por:
 - Intervenção - vacinação | tratamento | cirurgia | outra
 - Medicamentos receitados (`List<T>`);
 - Sumário da intervenção (texto opcional);
 - Data/hora da intervenção/consulta (`DateTime`).
 - Cada animal:
 - Código interno (da clínica);
 - Tipo: pássaro | gato | cão | batráquio | réptil | outro;
 - NIF do dono atual (ie. não tem histórico);
 - Observações (texto opcional).
 - E o sistema deve permitir:
 - i. Operações de registo, edição e remoção;
 - ii. Listagem da contagem de animais por tipo;
 - iii. Listagem da informação da última consulta de um animal.
- B. Construa um algoritmo que devolva os números primos de uma lista (parâmetro de entrada sob forma de vector)
- C. Construa um algoritmo que devolva os números primos de uma lista (parâmetro de entrada sob forma de `List<int>`)
- D. Calcule o máximo denominador comum de 2 números.
- E. Faça um *dry-running* de um algoritmo dos que implementou.