Porquê estudar algoritmos elementares (de ordenação)

- Razões de ordem práctica
  - Fáceis de codificar e por vezes suficientes
  - Rápidos/Eficientes para problemas de dimensão média e por vezes os melhores em certas circunstâncias
- Razões pedagógicas
  - Bom exemplo para aprender terminologia, compreender contexto dos problemas e bom princípios para desenvolvimento de algoritmos mais sofisticados
  - Alguns são fáceis de generalizar para métodos mais eficientes ou para melhorar o desempenho de outros algoritmos
- Importante para compreensão das regras de "funcionamento"

Contexto e regras básicas [1]

- Objectivo
  - Estudar métodos de ordenação de ficheiros de dados em que cada elemento (item) é caracterizado por uma chave ("key")
  - Chaves são usadas para controlar a ordenação
  - objectivo é rearranjar os dados de forma a que as chaves estejam ordenadas de forma pré-definida (numérica ou alfabética, por exemplo)
- Opções em termos de implementação
  - macros ou subrotinas/funções?
    - compromisso entre desempenho, generalidade e simplicidade
    - utilizaremos uma mistura de ambos

Contexto e regras básicas [2]

- Metodologia
  - características específicas de cada item ou chave podem ser diferentes mas conceito abstracto é o mais importante
  - começaremos por estudar ordenação em tabelas
  - utilizaremos operações abstractas nos dados: comparação, troca
  - alterar items para outro tipo (ex: vírgula flutuante) é simples

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

- Tempo de execução usualmente proporcional ao número de comparações número de movimentações/trocas (ou ambos)

Nomenclatura [1]

- Tipos de Algoritmos de Ordenação
  - não adaptativos: sequência de operações independente da ordenação original dos dados
    - interessantes para implementação em hardware
  - adaptativos: sequência de operações dependente do resultado de comparações (operação "less")
    - a maioria dos que estudaremos
- Parâmetro de interesse é o desempenho, i.e. tempo de execução
  - algoritmos básicos: N^2 para ordenar N items
    - mas por vezes os melhores para N pequeno
  - algoritmos avançados: $N \log N$ para ordenar N items
- Olharemos também para os recursos de memória necessários
  - ordenação "in-place" ou utilizando memória adicional

Nomenclatura [2]

- **Definição**: um algoritmo de ordenação é dito **estável** se preserva a ordem relativa dos items com chaves repetidas

  ex: ordenar lista de alunos por nome ou por ano de graduação
  - é usualmente possível estabilizar um algoritmo alterando a sua chave (tem custo adicional)
  - algoritmos básicos são quase todos estáveis, mas poucos algoritmos avançados são estáveis

Nomenclatura [3]

- **Definição**: um algoritmo de ordenação é dito interno, se o conjunto de todos os dados a ordenar couber na memória; caso contrário é dito externo

  ex: ordenar dados lidos de tape ou disco é ordenação externa

- Distinção muito importante:
  - ordenação interna pode aceder a qualquer dado facilmente
  - ordenação externa tem de aceder a dados de forma sequencial (ou em blocos)
- Vamos estudar fundamentalmente ordenação interna

Nomenclatura [4]

- **Definição**: um algoritmo de ordenação é dito **directo** se os dados são acedidos directamente nas operações de comparação e troca; caso contrário é dito **indirecto**

- Exemplo de algoritmo indirecto:
  - se a chave for pequena e cada dado for "grande"
    - (por exemplo o nome completo de um aluno, mais morada, número de BI, etc)
    - nestes casos não convém estar a trocar os elementos
      - é dispendioso
    - basta trocar a informação correspondente aos seus índices
      - tabela de índices/ponteiros em que o primeiro indica o menor elemento, etc

Contexto de utilização

- Mesmo programa, diferentes drivers para diferentes algoritmos:

```
#include <stdio.h>
#include <stdlib.h>

main (int argc, char *argv[])
{
   int i, N = atoi (argv[1]), sw = atoi (argv[2]);
   int *a = malloc (N * sizeof(int));

   if (sw)
      for (i = 0: i < N; i++)
          a[i] = 1000 * (1.0 * rand() / RAND_MAX);
   else
      while (scanf("%d", &a[N]) == 1) N++;
   sort (a, 0, N-1);
   for (i = 0; i < N; i++) printf("%3d", a[i]);
   printf("\n");
}
```

Para cada algoritmo, "*sort*" tem nome apropriado

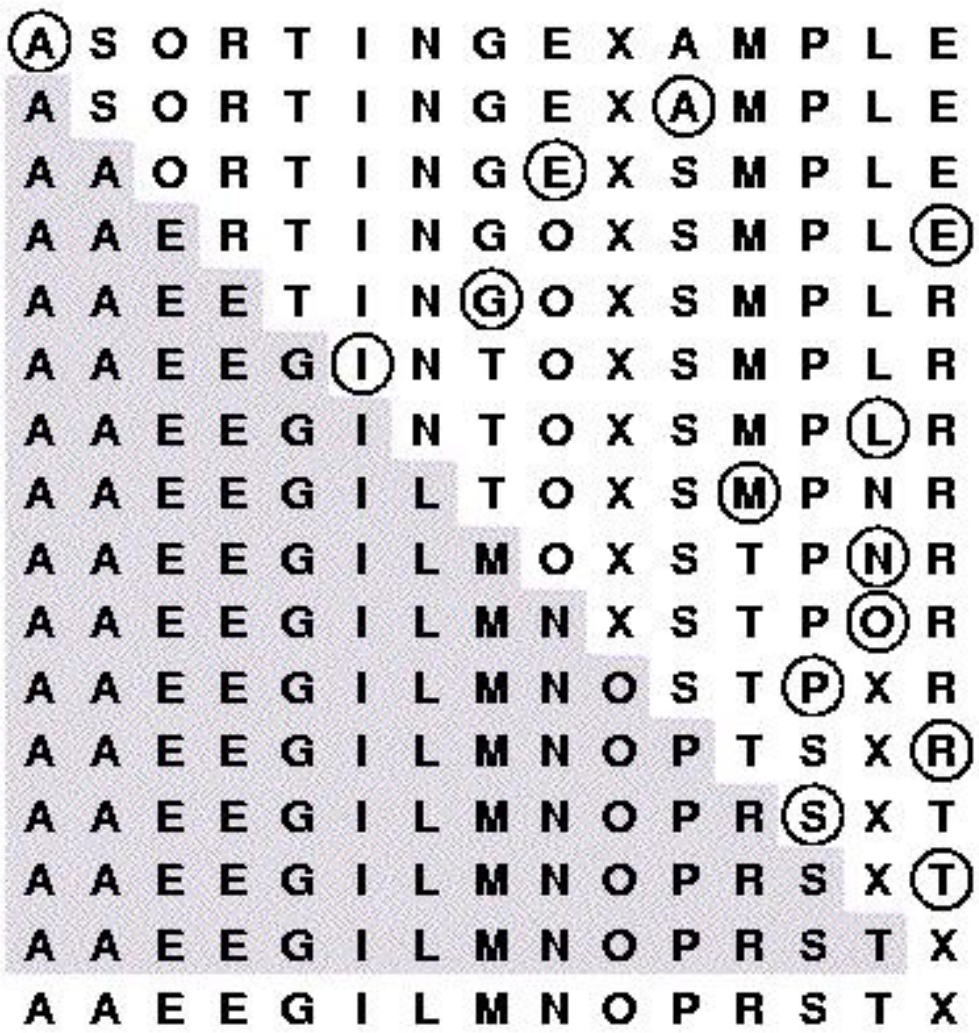
Ordenação por selecção - *Selection Sort*

- Um dos mais simples. **Algoritmo**:
  - procurar menor elemento e trocar com o elemento na 1ª posição
  - procurar 2º menor elemento e trocar com o elemento na 2ª posição
  - proceder assim até ordenação estar completa

```
void selection (itemType a[], int l, int r)
{
   int i, j;

   for (i = l; i < r; i++)  {
      int min = i;
      for (j = i+1; j <= r; j++)
         if  (less(a[j], a[min])) min = j;
      exch(a[i], a[min])
   }
}
```

Selection Sort - Exemplo de Aplicação



Selection Sort - Análise

- Ciclo interno apenas faz comparações
  - troca de elementos é feita fora do ciclo interno
  - cada troca coloca um elemento na sua posição final
    - número de trocas é ***N-1*** (porque não ***N***?)
  - tempo de execução dominado pelo número de comparações!

Propriedade: *Selection sort* usa aproximadamente $N^2/2$ comparações e N trocas

Demonstração: para cada item ***i*** de ***1*** a ***N-1*** há uma troca e ***N-i*** comparações (ver sombreados no exemplos)

Logo há ***N-1*** trocas e ***(N-1)+(N-2)+...+2+1=N(N-1)/2*** comparações

Factos: desempenho é independente da ordenação inicial dos dados; a única coisa que depende desta ordenação é o número de vezes que min é actualizado (quadrático no pior caso, ***N log N*** em média)

Ordenação por inserção - *Insertion Sort*

Ideia: considerar os elementos um a um e inseri-los no seu lugar entre os elementos já tratados (mantendo essa ordenação)

ex: ordenar cartas de jogar

- inserção implica arranjar novo espaço ou seja mover um número elevado de elementos uma posição para a direita
- inserir os elementos um a um começando pelo da 1ª posição

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

void insertion(itemType a[], int l, int r)
{
   int i, j;

   for (i = l+1; i <= r; i++)
      for (j = i; j > l; j--)
         compexch (a[j-1], a[j]);
}
```

Insertion Sort - Comentários

- Elementos à esquerda do índice corrente estão ordenados mas não necessariamente na sua posição final
  - podem ainda ter de ser deslocados para a direita para dar lugar a elementos menores encontrados posteriormente
- Implementação da ordenação por inserção na pág. 12 é ineficiente
  - código é simples, claro mas pouco eficiente; pode ser melhorado
  - ilustra bom raciocínio:
    - encontrar solução simples
    - estudar o seu funcionamento
    - melhorar desempenho através de pequenas transformações

Insertion Sort - Melhorar desempenho

- Demasiadas operações de comparação/troca (***compexch***)
  - podemos parar se encontramos uma chave que não é maior que a do item a ser inserido (tabela está ordenada à esquerda)
    - podemos sair do *loop* interno se ***less(a[j-1], a[j])*** é verdadeira
  - modificação torna o algoritmo adaptativo
  - aumenta desempenho aproximadamente por um factor de ***2***
- Passa a haver duas condições para sair do *loop*
  - mudar para um loop *while*
  - remover instruções irrelevantes
    - ***compexch*** não é o melhor processo de mover vários dados uma posição para a direita

Insertion Sort - Versão adaptativa

```
void insertion(itemType a[], int l, int r)
{
   int i, j;

  for (i = l+1; i <= r; i++) {
     itemType v = a[i];
     j = i;
     while (j > l && less(v, a[j-1])) {
       a[j] = a[j-1]; j--;
     }
     a[j] = v;
  }
}
```

Insertion Sort - Exemplo de Aplicação

Insertion Sort - Análise

Propriedade: *Insertion sort* usa aproximadamente $N^2/4$ comparações e $N^2/4$ pseudo-trocas (translações ou movimentos) no caso médio e o dobro destes valores no pior caso

Demonstração: Fácil de ver graficamente; elementos abaixo da diagonal são contabilizados (todos no pior caso)

Para dados aleatórios é expectável que cada elemento seja colocado aproximadamente a meio para trás pelo que apenas metade dos elementos abaixo da diagonal devem ser contabilizados

- **Factos:** em certos casos *Insertion sort* pode ter bom desempenho (a ver...)

Bubble Sort

- Talvez o algoritmo mais utilizado e o que muitas pessoas aprendem

Ideia: fazer múltiplas passagens pelos dados trocando de cada vez dois elementos adjacentes que estejam fora de ordem, até não haver mais trocas

- supostamente muito fácil de implementar
- usualmente mais lento que os dois métodos elementares estudados

```
void bubble(itemType a[], int l, int r)
{
     int i, j;
     for (i = l; i < r; i++)
         for (j = r; j > i; j--)
             compexch(a[j], a[j-1]);
}
```

Bubble Sort - Comentários [1]

- Movendo da direita para a esquerda no ficheiro de dados
  - quando o elemento mais pequeno é encontrado na primeira passagem
    - é sucessivamente trocado com todos à sua esquerda
    - acaba por ficar na primeira posição
  - na segunda passagem o 2º elemento mais pequeno é colocado na sua posição e por diante
  - ***N*** passagens pelos dados são suficientes!

- É semelhante ao método de selecção
  - tem mais trabalho para colocar cada elemento na sua posição final
    - todas as trocas sucessivas até chegar à posição certa

Bubble Sort - Comentários [2]

- Algoritmo pode ser melhorado, tal como o algoritmo de inserção
  - código é muito semelhantes mas não igual
    - *loop* interno de *Selection* percorre a parte esquerda(ordenada) da tabela
    - *loop* interno de *Bubble Sort* percorre a parte direita(não ordenada) da tabela
  - no final de cada passagem podemos testar se houve mudanças

Bubble Sort - Exemplo de Aplicação

Bubble Sort - Análise

Propriedade: *Bubble sort* usa aproximadamente $N^2/2$ comparações e $N^2/2$ trocas no caso médio e no pior caso

Demonstração: A ***i***-ésima passagem de *Bubble Sort* requer ***N-i*** operações de comparação/troca, logo a demonstração é semelhante a ***Selection sort***

Factos: Algoritmo pode depender criticamente dos dados se for modificado para terminar quando não houver mais trocas

- se o ficheiro estiver ordenado, apenas um passo é necessário
- se estiver em ordenação inversa então na ***i***-ésima passagem são precisas ***N-1*** comparações e trocas
- caso médio é semelhante (análise mais complexa)

Comparação dos algoritmos elementares de ordenação [1]

- Tempos de execução quadráticos

	Selection	Insertion	Bubble
Comparações	$N^2/2$	$N^2/4$	$N^2/2$
Trocas	N	$N^2/4$	$\approx N^2/2$

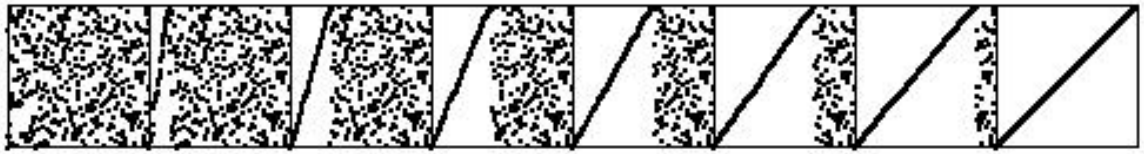
Comparação dos algoritmos elementares de ordenação [2]

- Ficheiros com elementos grandes e pequenas chaves
  - Selection Sort é linear no número de dados
    - ***N*** dados com tamanho ***M*** (palavras/words)
      - comparação - 1 unidade; troca - M unidades
    - $N^2/2$ comparações e ***NM*** custo de trocas
    - termo ***NM*** domina
    - custo proporcional ao tempo necessário para mover os dados
- Ficheiros quase ordenados
  - *Bubble* e *Insertion sort* são quase lineares

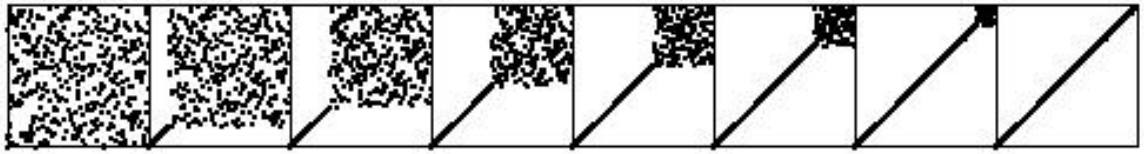
  ➔os melhores algoritmos de ordenação podem ser quadráticos neste caso!

Comparação dos algoritmos elementares de ordenação [3]

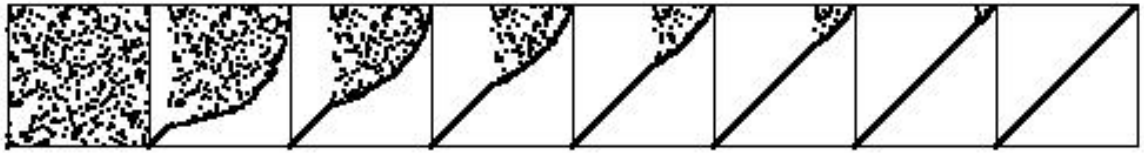
Insertion sort



Selection sort



Bubble sort



Shellsort - Melhorar a eficiência [1]

- *Insertion sort* é lento: apenas envolve trocas entre items adjacentes
  - se o menor item está no final da tabela, serão precisos ***N*** passos para o colocar na posição correcta

- *Shellsort*:
  - acelerar o algoritmo permitindo trocas entre elementos que estão afastados
  - como?

Shellsort - Melhorar a eficiência [2]

Ideia: 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 elementos na tabela grandes "distâncias" 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 vao produzir um ficheiro ordenado
  - cada passo torna o próximo mais simples

Shellsort - Melhorar a eficiência [3]

- **Outra forma de ver o algoritmo**:
  - a tabela é dividida em partições, cada uma contendo os objectos de índice ***% h***

    exemplo: para uma tabela de ***15*** posições e ***h=4***
    - índices resto ***0***, ***i%4=0***, são: ***0, 4, 8, 12***
    - índices resto ***1***, ***i%4=1***, são: ***1, 5, 9, 13***
    - índices resto ***2***, ***i%4=2***, são: ***2, 6, 10, 14***
    - índices resto ***3***, ***i%4=3***, são: ***3, 7, 11, 15***

    ➔ dentro de cada partição é feita uma ordenação por inserção

- Exemplo permite visualizar bem o algoritmo...

h-Ordenação - Exemplo: h=4

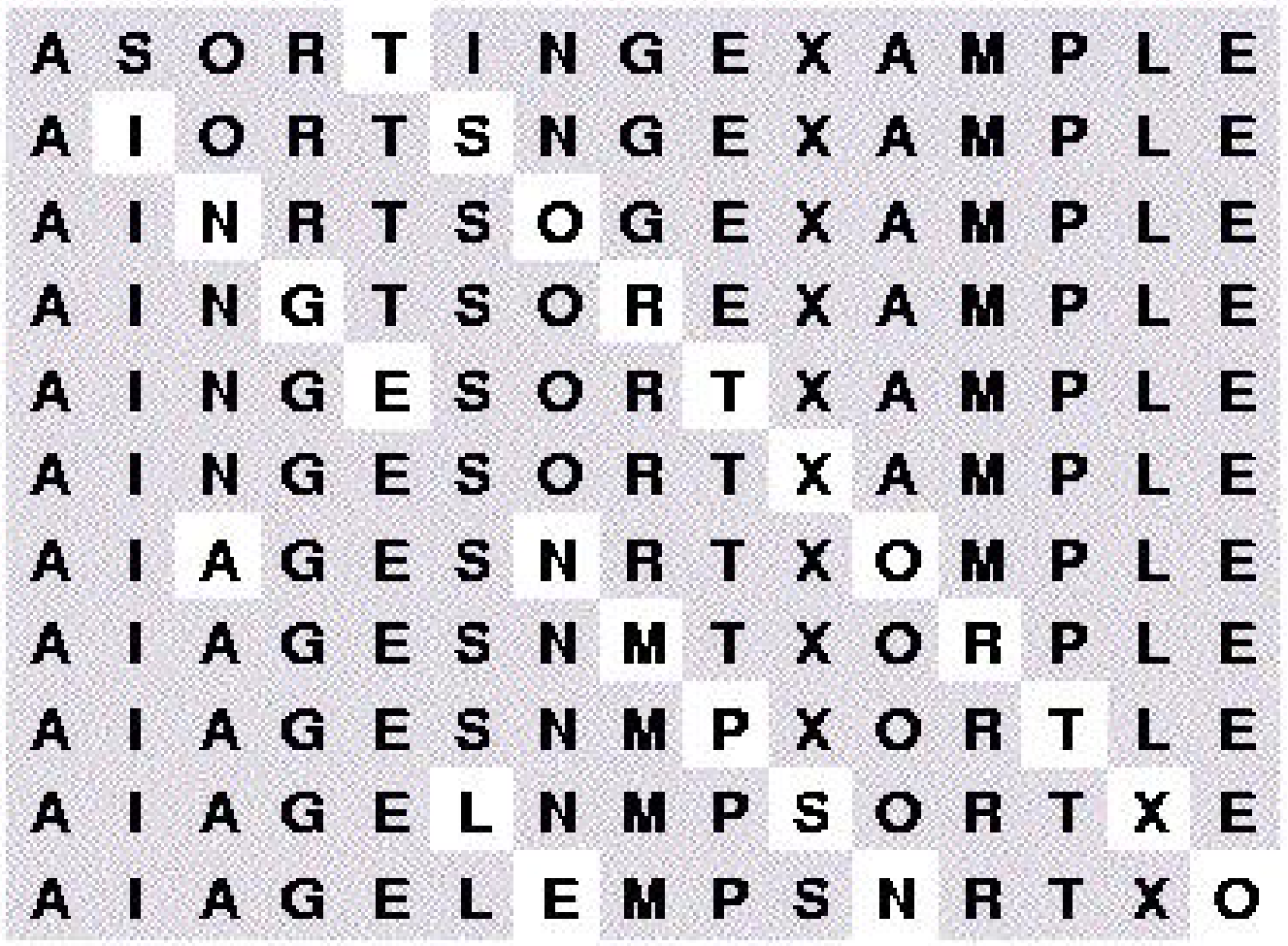
Implementação de *Shellsort* [1]

- Possível fazer melhor: sub-ficheiros são independentes
  - quando h-ordenamos os dados inserimos qualquer elemento entre os do seu (h) sub-ficheiro movendo elementos maiores para a direita
    - basta usar insertion code com incrementos/decrementos de h em vez de 1
  - implementação de *Shellsort* usa um passo de *Insertion sort* pelo ficheiro para cada incremento

- Tabela não fica completamente ordenada depois da 1ª ordenação das várias partições
  - é necessário repetir com uma série de partições
  - isto é, não basta escolher/usar um só ***h***!

Shellsort - Exemplo de aplicação [1]

Partições estão entrelaçadas



Implementação de *Shellsort* [2]

Solução óbvia: para cada h usar *Insertion Sort* independentemente em cada um dos h sub-ficheiros

```
h = 4;
for (i = l+h; i <= r; i++) {
   itemType v = a[i];
   j = i;
   while (j >= l+h && less(v, a[j-h])) {
      a[j] = a[j-h];
      j -= h;
   }
   a[j] = v;
}
```

Sequência de Ordenação

- Difícil de escolher
  - propriedades de muitas sequências foram já estudadas
  - possível provar que umas melhores que outras
    - ex: 1, 4, 13, 40, 121, 364, 1093, 3280, ... (Knuth, $3*h_{ant}+1$)

      melhor que

      1, 2, 4, 8, 16, 32, 64, 128, 256, 512, ... (Shell, 2^i)
      - Porquê?

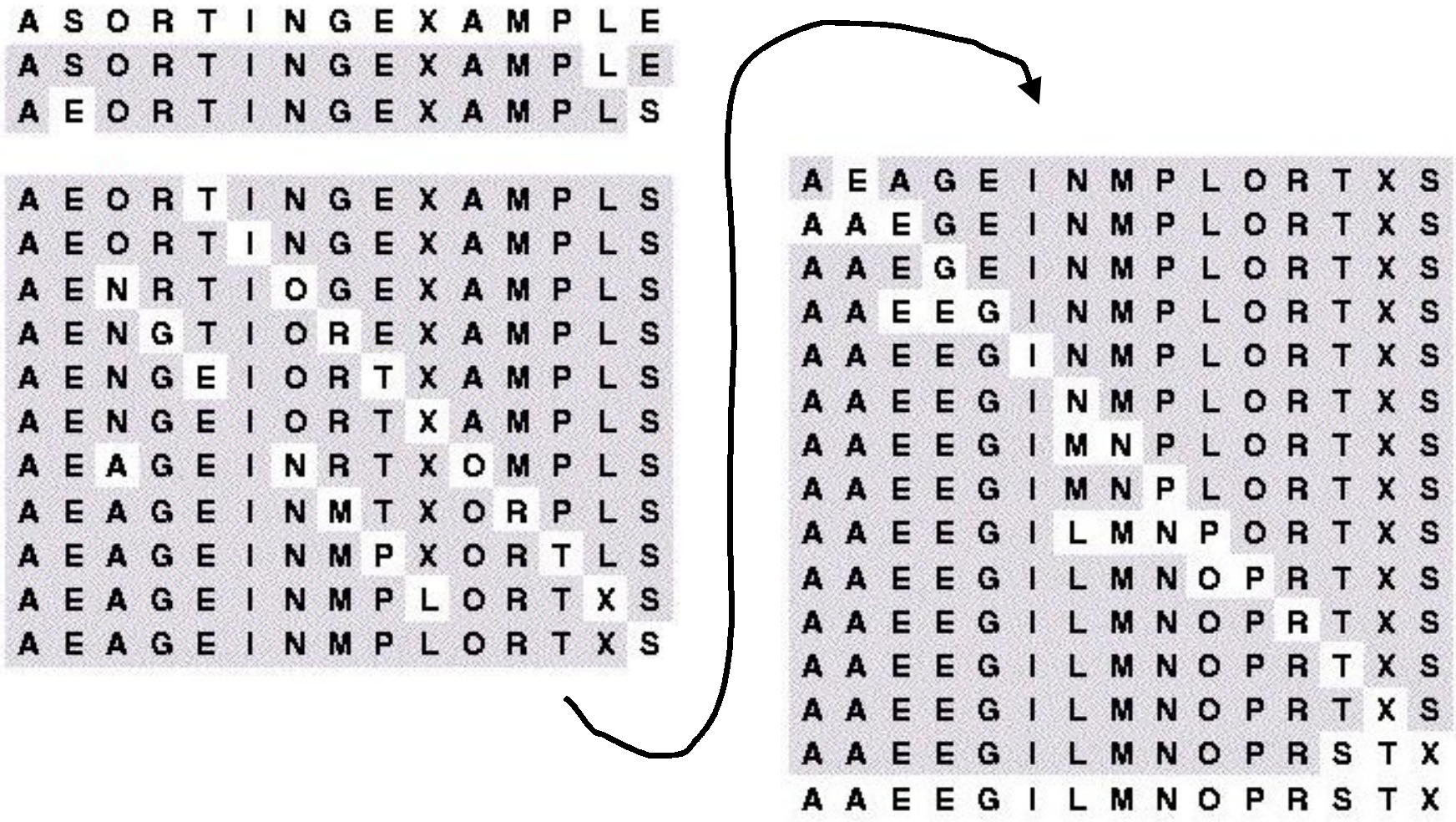
      mas pior (20%) que 1, 8, 23, 77, 281, 1073, 4193, … ($4^{i+1} + 3\ 2^i + 1$)
  - na prática utilizam-se sequencias que decrescem geometricamente para que o número de incrementos seja logaritmico
  - a sequência óptima não foi ainda descoberta (se é que existe)
  - análise do algoritmo é desconhecida
    - ninguem encontrou a fórmula que define a complexidade
    - complexidade depende da sequência

Shellsort

```
void shellsort(itemType a[], int l, int r)
{
   int i, j;
   int incs[16] = { 1391376, 463792, 198768, 86961, 33936,
                    13776, 4592, 1968, 861, 336, 112, 48,
                    21, 7, 3, 1 };
   for ( k = 0; k < 16; k++) {
      int h = incs[k];

      for (i = l+h; i <= r; i++)  {
         itemType v = a[i];
         j = i;
         while (j >= h && less(v, a[j-h]))  {
            a[j] = a[j-h];
            j -= h;
         }
         a[j] = v;
      }
   }
}
```

Shellsort - Exemplo de aplicação [2]



Análise de *Shellsort*

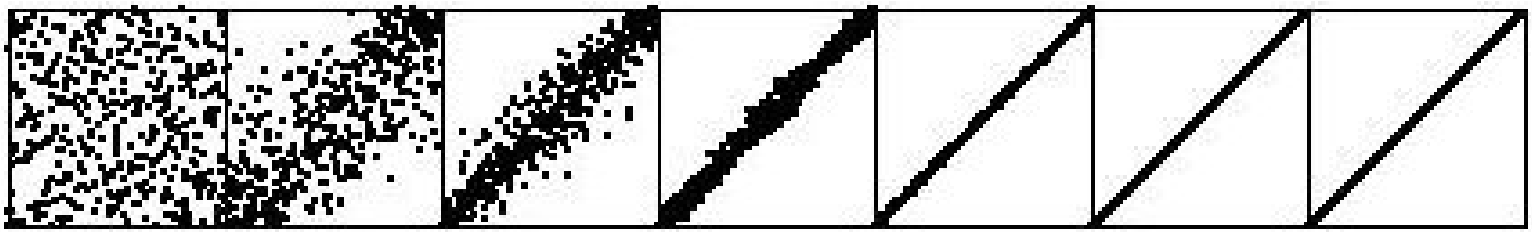
- Propriedades:
  - o resultado de h-ordenar um ficheiro que está k-ordenado é um ficheiro que está simultaneamente h- e k-ordenado
  - *Shellsort* faz menos do que $O(N^3/2)$ comparações para os incrementos 1, 4, 13, 40, 121, 364, 1093, ...
  - *Shellsort* faz menos do que $O(N^4/3)$ comparações para os incrementos 1, 8, 23, 77, 281, 1073, 4193, 16577, ...
    - Nota: sequências até agora usam incrementos que são primos entre si
  - *Shellsort* faz menos do que $O(N \log^2 N)$ comparações para os incrementos 1, 2, 3, 4, 6, , 9, 8, 12, 18, 27, 16, 24, 36, 54, 81
- *Shellsort* é um exemplo de um algoritmo simples cuja análise é complicada:
  - sequência **óptima**, e análise completa não são conhecidos!
- Alguém consegue encontrar uma sequência que bata

  1, 2, 7, 21, 48, 112, 336, ...??

Shellsort - Exemplo Gráfico

- Graficamente parece que um elástico, preso nos cantos puxa os pontos para a diagonal

Shellsort



Vantagens de *Shellsort*

- rápido/eficiente
- pouco código
- **melhor** método para ficheiros pequenos e médios
- aceitável para elevados volumes de dados

➔ Muito utilizado na prática, embora díficil de compreender!

Ordenação de outros tipos de dados [1]

- Algoritmos estudados são independentes do tipo de dados
  - necessário o interface apropriado e a implementação correspondente

```
#include <stdlib.h>
#include <stdio.h>

main (int argc, char *argv[])
{
   int  i, N = atoi (argv[1]), sw = atoi (argv[2]);
   int *a = malloc (N * sizeof(int));

   if (sw)
      randinit(a, N);
   else
      scaninit(a, &N);
   sort (a, 0, N-1);
   show (a, 0, N-1);
}
```

```
for (i = 0: i < N; i++)
    a[i] = 1000 * (1.0*rand()/RAND_MAX);
```

```
while (scanf("%d", &a[N]) == 1) N++;
```

```
for (i = 0; i < N; i++)
   printf("%3d", a[i]);
printf("\n");
```

Ordenação de outros tipos de dados [2]

- Algoritmos estudados são independentes do tipo de dados
  - necessário o interface apropriado e a implementação correspondente

```
#include <stdlib.h>
#include <stdio.h>

main (int argc, char *argv[])
{
   int i, N = atoi (argv[1]), sw = atoi (argv[2]);
   Item *a = malloc (N * sizeof(Item));

   if (sw)
      randinit(a, N);
   else
      scaninit(a, &N);
   sort (a, 0, N-1);
   show (a, 0, N-1);
}
```

Ordenação de outros tipos de dados [3]

- Definição do interface: o tipo apropriado

```
void randinit (Item [], int);
void scaninit (Item [], int *);
void show (Item [], int, int);
void sort (Item [], int, int);
```

Interface para tipo tabela de Items

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

Item ITEMrand(void);
int  ITEMscan(Item *);
void ITEMshow(Item);
```

Interface para tipo Item

Ordenação de outros tipos de dados [4]

- Definição do interface: as funções que sobre ele operam

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "Array.h"

void randinit(Item a[], int N)
{
    int i;
    for (i = 0; i < N; i++) a[i] = ITEMrand();
}

void scaninit(Item a[], int *N)
{
    int i =0;;
    for (i = 0; i < *N; i++)
       if (ITEMscan(&a[i]) == EOF) break;
    *N = i;
}

void show(Item a[], int l, int r)
{
    int i;
    for (i = l; i <= r; i++) ITEMshow(a[i]);
    printf("\n");
}
```

Inplementação para tipo tabela de Items

Ordenação de outros tipos de dados [5]

- Implementação: as funções que sobre ele operam

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"

double ITEMrand(void)
{
   return 1.0 / rand() / RAND_MAX;
}

int  ITEMscan(double *x)
{
   return scanf("%f", x);
}

void ITEMshow(double x)
{
   printf("%7.5f ", x);
}
```

Implementação para tipo Item (double)

Ordenação de outros tipos de dados [6]

- Para qualquer tipo de dados
  - apenas alterar o interface
  - reescrever as funções que operam sobre os objectos
    - rand
    - show
    - exch
    - etc, etc
  - algoritmos e rotinas de sorting não necessitam de qualquer alteração

Ordenação por índices [1]

- Suponha-se que os objectos a ordenar são strings (vectores de caracteres):
  - funções que operam sobre os dados precisam de ter em conta a questão da alocação de memória para strings
  - quem deve ser responsável pela gestão desta memória?
  - e se os objectos são "grandes"? Comparar e mover os objectos pode ser dispendioso!
    - imagine que cada objecto é o nome completo de um aluno (ou que é toda a sua informação: nome, morada, etc)
  - mesmo que haja uma boa abstracção para operar sobre os objectos ainda há a questão do custo
- Porquê movê-los?
  - porque não alterar apenas referência para a sua posição relativa?

Ordenação por índices [2]

- Solução 1:
  - dados numa tabela ***data[0], ..., data[N-1]***
  - usar uma segunda tabela, ***a[.]***, apenas de índices
    - inicializado de forma a que ***a[i] = i, i = 0, ..., N-1***
  - objectivo é rearranjar a tabela de índices de forma a que ***a[0]*** aponte para o objecto com a menor chave, etc
  - objectos são apenas acedidos para comparação
- Rotinas de ordenação apenas acedem aos dados através de funções de interface:, ***less***, ***exch***, etc
  - apenas estas têm de ser reescritas

Ordenação por índices - Exemplo

- Suponha os seguintes dados: ***data = [“rui”, “carlos”, “luis”]***
  usamos uma tabela, de índices ***a = [0, 1, 2]***
  - 1º passo: comparar ***data[a[1]]*** com ***data[a[0]]***: ***“carlos” < “rui”***
    pelo que há troca de ***a[1]*** com ***a[0]***: ***a = [1, 0, 2]***
  - 2º passo: comparar ***data[a[2]]*** com ***data[a[1]]***: ***“rui” < “luis”***
    pelo que há troca de ***a[2]*** com ***a[1]***: ***a = [1, 2, 0]***
  - 3º passo: comparar ***data[a[1]]*** com ***data[a[0]]***: ***“carlos” < “luis”***
    pelo que não há troca

- Os valores ordenados sao portanto ***data[a[0]], data[a[1]]*** e ***data[a[2]]***
  ou seja ***“carlos” < “luis” < “rui”***

(de forma encaputada usámos uma “espécie” de *Selection sort*)

Ordenação por ponteiros

- Outra solução é a tabela de índices conter de facto ponteiros para os dados
  - mais geral pois os ponteiros podem apontar para qualquer lado
    - items nao precisam de ser membros de uma tabela
    - nem de ter todos o mesmo tamanho
  - depois da ordenação, acesso sequencial à tabela de ponteiros devolve os elementos ordenados

- um exemplo é a função ***qsort***, do C, que implementa o algoritmo **quicksort** (estudaremos adiante)

Ordenação por ponteiros ou índices

- Não-intrusiva em relação aos dados
  - pode ser efectuada se os dados forem apenas de leitura
- Possível efectuar ordenação em chaves múltiplas

  ex: listagens de alunos, com nome, número e nota
- Evita o custo de mover/trocar os items
  - pode ser elevado se estes forem grandes quantidades de informação
  - mais eficiente em problemas com dados grandes e chaves pequenas
- E se for preciso retornar os dados ordenados?
  - ordenar por índice/ponteiro
  - fazer permutações in-situ (como?)

Ordenação de listas ligadas

- Semelhante a ordenação de tabelas
  - funções de interface diferentes
  - necessário trabalhar com ponteiros e não apenas com índices
- Trocas podem não ser possíveis
  - elementos podem ser acedidos por outras estruturas de dados
  - trocam-se apenas os ponteiros
  - inserções são na verdade mais simples
    - não é preciso mover elementos para a direita, apenas ajustar dois ponteiros
- Algoritmos básicos
  - alterações simples

Exercício - usar *Selection Sort* para ordenar uma lista ligada

- Estruturas de Dados:
  - lista de entrada
  - lista de saída: dados colocados na sua posição final
  - ponteiros adicionais

Ideia básica

  - procurar maior elemento na lista
  - manter ponteiro para o anterior
  - retirá-lo da lista de entrada
  - introduzi-lo no início da lista de saída

➔ Implemente como exemplo!