Quicksort [1]

- Provavelmente o algoritmo mais usado
  - "inventado" nos anos 60
  - muito estudado e analisado
    - desempenho bem conhecido
  - popular devido à facilidade de implementação e eficiência
    - complexidade ***N log N***, em média, para ordenar ***N*** objectos
    - ciclo interno muito simples e conciso

  ➔ mas:
    - não é estável
    - quadrático (N^2) no pior caso!
    - "frágil": qualquer pequeno erro de implementação pode não ser detectado mas levar a ineficiência
- Função de ordenação do C (biblioteca) é o **`qsort()`**
  - uma implementação do *quicksort*

Quicksort [2]

Algoritmo do tipo *dividir para conquistar*

Idea chave: efectuar partição dos dados e ordenar as várias partes independentemente (de forma recursiva)

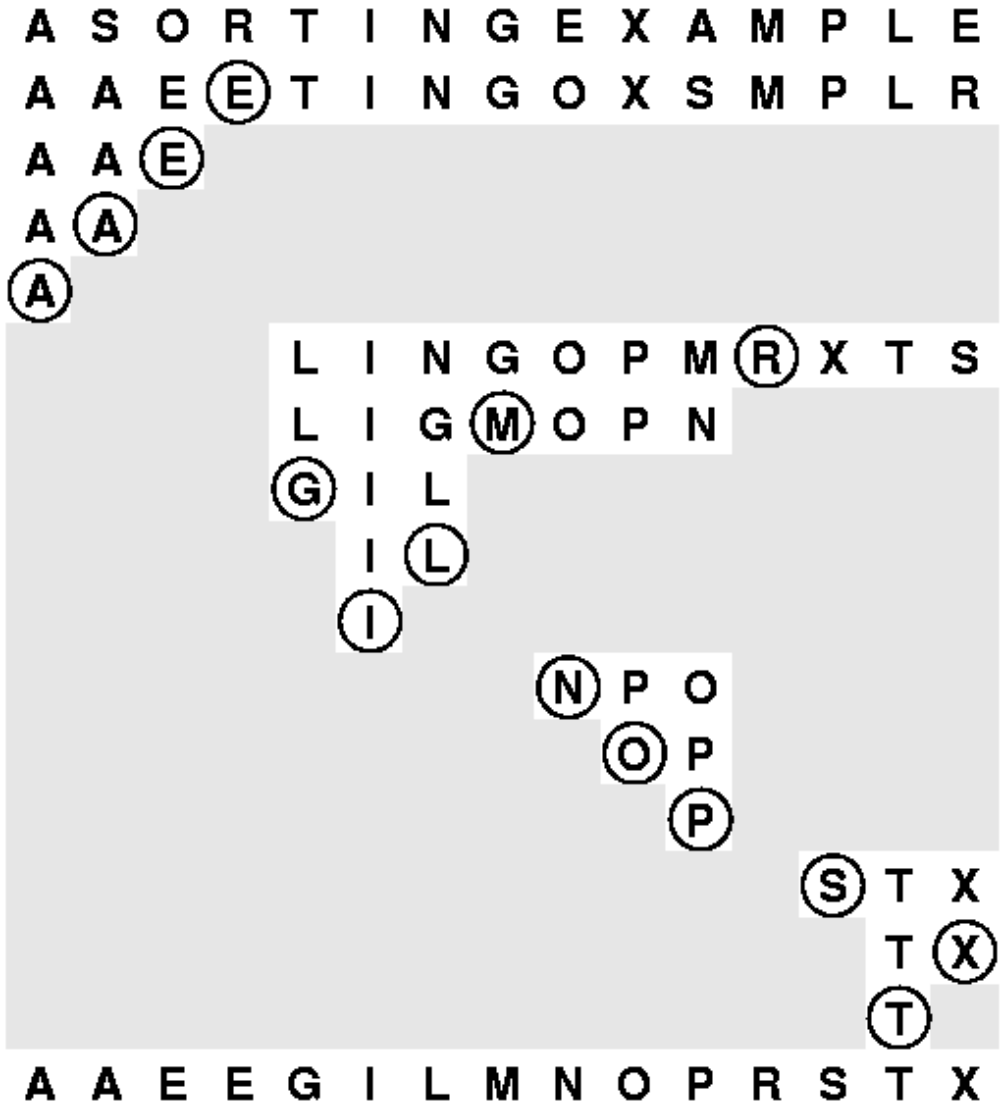
- posicionamento da partição a efectuar depende dos dados de entrada
- processo de partição é crítico
- algoritmo é recursivo por natureza
  - uma vez efectuada a partição, cada uma das partes pode por sua vez ser ordenada pelo mesmo algoritmo (o que implica nova partição dos dados)

Assuma os dados de entrada numa tabela ***a[...]*** de tamanho ***N***

Quicksort [3]

```
void quicksort(Item a[], int l, int r)
{
   int i;

   if (r <= l) return;

   i = partition(a, l, r);
   quicksort(a, l, i-1);
   quicksort(a, i+1, r);
}
```

Quicksort [4]


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

Quicksort - Partição [1]

- Processo de partição rearranja os dados de forma a que as três condições seguintes sejam válidas (de ***a[l]*** a ***a[r]***):
  - o elemento ***a[i]***, para algum ***i,*** fica, após a partição, na sua posição final
  - nenhum dos elementos em ***a[l] … a[i-1]*** é maior do que ***a[i]***
  - nenhum dos elementos em ***a[i+1] … a[r]*** é menor do que ***a[i]***

  ➔ Processo coloca pelo menos um elemento na sua posição final
    - é possível uma demonstração por indução para o processo recursivo

- Após partição, a tabela fica sub-dividida em duas partes
  - (sub-tabelas) que podem ser ordenadas separadamente

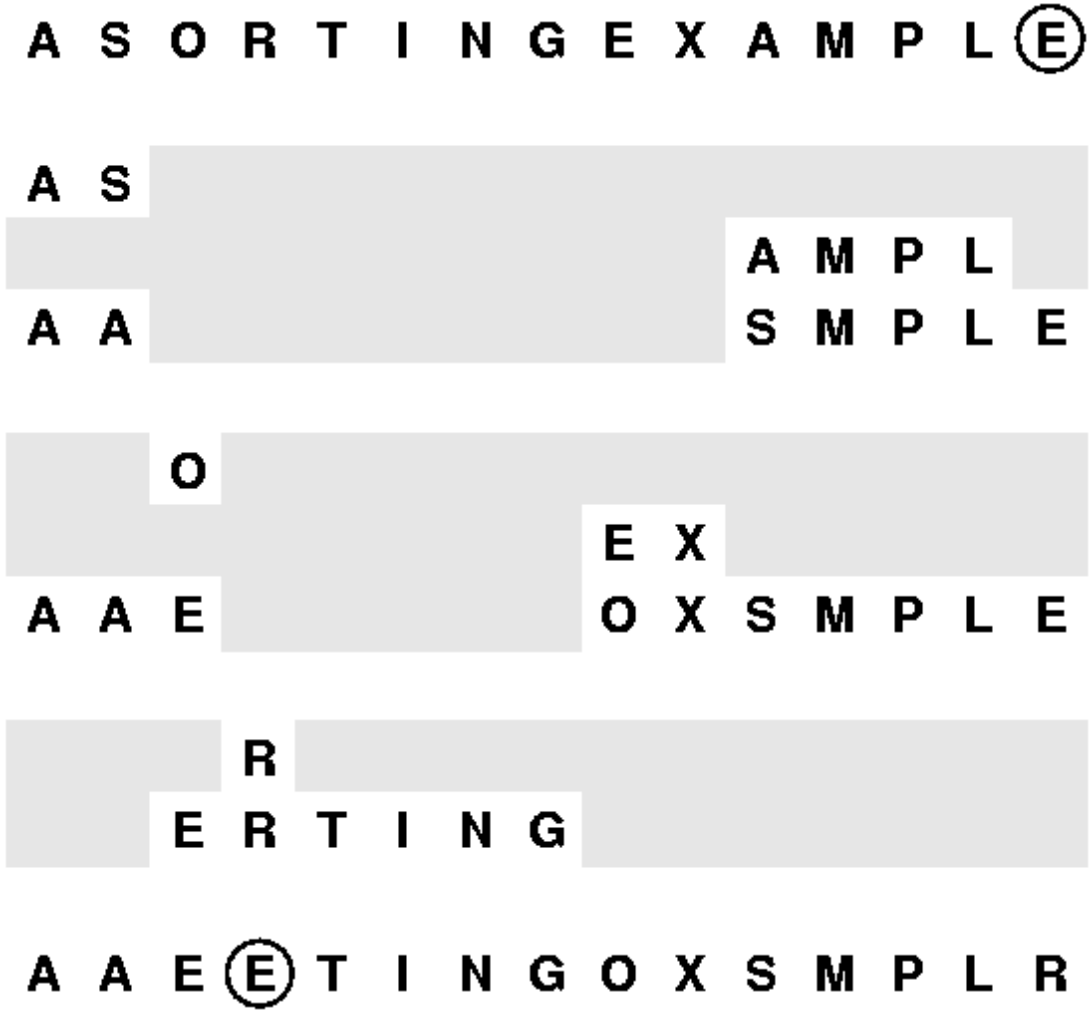
- Ordenação completa é conseguida através de
  - partição + aplicação **recursiva** do algoritmo aos dois subconjuntos de dados daí resultantes

Quicksort - Partição [2]

Estratégia para a partição:

- escolher ***a[r]*** arbitrariamente para ser o elemento de partição
  - o que é colocado na posição final
- percorrer a tabela a partir da esquerda até encontrar um elemento maior que ou igual ao elemento de partição (***a[r]***)
- percorrer a tabela a partir da direita até encontrar um elemento menor que ou igual ao elemento de partição (***a[r]***)
  - estes dois elementos estão deslocados; trocamos as suas posições!
- procedimento continua até nenhum elemento à esquerda de ***a[r]*** ser maior que ele, e nenhum elemento à direita de ***a[r]*** ser menor que ele
  - termina quando ponteiros se cruzam
  - completa-se trocando ***a[r]*** com o elemento mais à esquerda da sub-tabela da direita

Quicksort - Partição [3]



Quicksort - Partição [4]

- Ciclo interno de *Quicksort* é muito simples
  - incrementa um índice (“ponteiro”) e compara um elemento de uma tabela com um valor fixo
  - esta simplicidade é o que faz o quicksort rápido
    - díficil imaginar um ciclo interno mais curto e simples num algoritmo de ordenação
- Processo de partição não é estável
  - qualquer chave pode ser movida para trás de várias outras chaves iguais a si (que ainda não foram examinadas)
    - não é conhecida nenhuma forma simples de implementar uma versão estável de quicksort baseada em tabelas
- Procedimento tem de ser implementado cuidadosamente
  - não se deve chamar recursivamente se o ficheiro tiver tamanho 1
  - recursão acontece para valores estritamente menores que os de entrada
  - pode-se usar um teste explícito para ver se o elemento de partição é o menor de todo a tabela

Quicksort - Partição [5]

```
int partition(Item a[], int l, int r)
{
   int i, j;
   Item v;

   v = a[r]; i = l-1; j = r;
   for (;;) {
      while (less(a[++i], v)) ;
      while (less(v, a[--j]))
         if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
   }
   exch(a[i], a[r]);
   return i;
}
```

Questões:

- deverá parar-se a busca em chaves iguais a v?
- Deveria haver sentinelas para testar os limites da tabela?
- Detalhes do "cruzamento" de índices são muito importantes

a - tabela de elementos a ordenar; ***l, r*** - gama a ordenar; ***v*** - elemento de partição
i - ponteiro da esquerda para a direita; ***j*** - ponteiro da direita para a esquerda

Quicksort - Partição [6]

- Quando há chaves duplicadas o cruzamento de índices é subtil
  - poder-se-ia parar o processo de partição se ***j*** **<** ***i***, e usar ***j*** em vez de ***i-1*** para delimitar a extremidade esquerda da sub-tabela à direita
    - deixar o algoritmo iterar mais uma vez é vantajoso: ficamos com dois elementos (nas posições ***i*** e ***j***) que já estão nas posições finais
      - o elemento de partição e o elemento que acabou com a procura (igual aquele)
  - há várias soluções possíveis sobre o que fazer quando os índices de procura são iguais ao elemento de partição
    - pode provar-se que o melhor é parar a procura
- Eficiência do processo de ordenação depende de quão bem a partição divide os dados
  - depende por seu turno do elemento de partição
    - será tanto mais equilibrada quanto mais perto este elemento estiver do meio da tabela na sua posição final

Quicksort - Características [1]

➔ Pode ser muito ineficiente em casos patológicos

Propriedade: *quicksort* usa cerca de $N^2/2$ comparações no pior caso

Demonstração: se o ficheiro já estiver ordenado, todas as partições degeneram e o programa chama-se a si próprio N vezes; o número de comparações é de

$$N + (N-1) + (N-2) + \ldots + 2 + 1 = (N + 1)\, N / 2$$

(mesma situação se o ficheiro estiver ordenado por ordem inversa)

➔ Não apenas o tempo necessário para a execução do algoritmo cresce quadraticamente como o espaço necessário para o processo recursivo é de cerca de N o que é inaceitável para ficheiros grandes

Quicksort - Características [2]

Melhor caso: quando cada partição divide o ficheiro de entrada exactamente em metade

- número de comparações usadas por *quicksort* satisfaz a recursão de dividir para conquistar

$$C_N = 2\,C_{N/2} + N$$

  - 1^0 termo cobre o custo de ordenar os dois sub-ficheiros
  - 2^0 termo refere-se a examinar cada elemento

- solução é $C_N = N \log N$ (vimos numa aula anterior)

Quicksort - Características [3]

Propriedade: *quicksort* usa cerca de ***2N ln N*** comparações em média

Demonstração: A fórmula de recorrência exacta para o número de comparações utilizado por *quicksort* para ordenar ***N*** números distintos aleatoriamente posicionados e

$$C_N = N + 1 + \frac{1}{N} \sum_{1 \le k \le N} \left(C_{k-1} + C_{N-k} \right) \qquad N \ge 2,\ C_0 = C_1 = 0$$

- termo ***N+1*** cobre o custo de comparar o elemento de partição com os restantes (2 comparações extra: ponteiros cruzam-se)
- resto vem do facto de que cada elemento tem probabilidade ***1/k*** de ser o elemento de partição após o que ficamos com duas sub-tabelas de tamanhos ***k-1*** e ***N-k***

Quicksort - Características [4]

- Recursão anterior é simples de resolver

  ➔ ver livro

  – aproximadamente ***2N ln N***

- Análise assume que os dados estão aleatoriamente ordenados e têm chaves diferentes
  – pode ser lento em situações em que as chaves não são distintas ou que os dados não estão aleatoriamente ordenados (como vimos)

➔ Algoritmo pode ser melhorado
  – para reduzir a probabilidade que estes casos sucedam!
  – necessário em ficheiros de grandes dimensões ou se o algoritmo for usado como função genérica numa biblioteca

Quicksort - Características [5]

Características dinâmicas: **visualização**

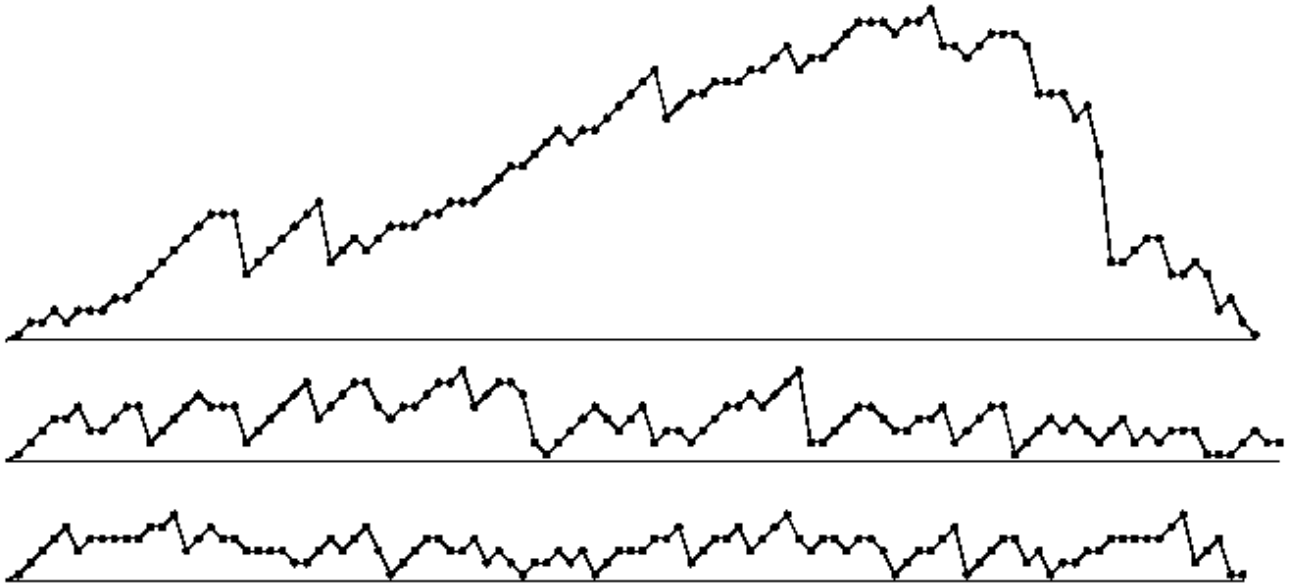
- partição divide em dois ficheiros que podem ser ordenados independentemente
- nenhum dos elementos para a esquerda do ponteiro de procura à esquerda é maior que o elemento de partição
  - não há elementos acima e à esquerda dele
- nenhum dos elementos para a direita do ponteiro de procura à direita é maior que ele
  - não hã elementos abaixo e à direita dele
- elemento de partição está na diagonal
  - a sua posição final

Quicksort - Características [6]

- Questões mais relevantes:
  - possível redução de desempenho devido ao uso de recursão
  - tempo de execução dependente dos dados de entrada
  - tempo de execução quadrático no pior caso
    - um problema
  - espaço/memória necessário no pior caso é linear
    - um problema sério (para ficheiros de grandes dimensões)
- Problema do espaço está associado ao uso de recursão:
  - recursão implica chamada a função e logo a carregar dados na pilha/stack do computador
  - no pior caso <u>todas</u> as partições degeneram e há ***O(N)*** níveis de recursão
    - pilha cresce até ordem ***N*** !!!

Quicksort - Espaço necessário [1]

- No pior caso espaço extra para a ordenação é linear em ***N***
  - inaceitável, mas pode ser melhorado!
    - Exemplos: 1º - ficheiro parcialmente ordenado
      2º e 3º - ficheiros aleatóriamente ordenados

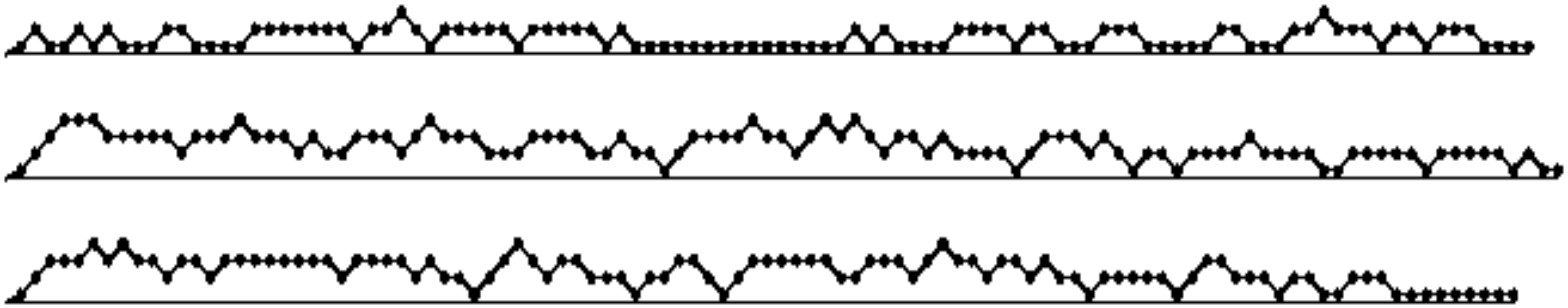


Quicksort - Espaço necessário [2]

- Usamos uma pilha (stack) explícita
  - pilha contém "trabalho" a ser processado, na forma de sub-tabelas a ordenar
  - quando precisamos de uma sub-tabela para processar tiramo-la da pilha (i.e. fazemos um ***pop()*** do stack)
  - por cada partição criamos duas sub-tabelas e metemos ambas na pilha (i.e. fazemos dois ***push()*** para o stack)
  - substitui a pilha do computador que é usado na implementação recursiva
- Conduz a uma versão não recursiva de ***quicksort()***
  - verifica os tamanhos das duas sub-tabelas e põe a maior delas primeiro na pilha (e a menor depois; logo a menor é retirada e tratada primeiro)
  - ordem de processamento das sub-tabelas não afecta a correcta operação da função ou o tempo de processamento mas afecta o tamanho da pilha

Quicksort - Espaço necessário [3]

- No pior caso espaço extra para a ordenação é logaritmico em ***N***
  - Exemplos: 1º - ficheiro parcialmente ordenado
    2º e 3º - ficheiros aleatóriamente ordenados



- Tempo de execução continua a ser quadrático no pior caso!

Quicksort - Versão não-recursiva [1]

```
#define push2(A, B)  push(A); push(B);

void quicksort(Item a[], int l, int r)
{
   int i;
   stackinit(); push2(l, r);
   while (!stackempty()) {
      r = pop(); l = pop();
      if (r <= l) continue;
      i = partition(a, l, r);
      if (i-l > r-i) {
         push2(l, i-1); push2(i+1, r);
      } else {
         push2(i+1, r); push2(l, i-1);
      }
   }
}
```

Quicksort - Versão não-recursiva [2]

- Política de colocar a maior das sub-tabelas primeiro na pilha
  - garante que cada entrada na pilha não é maior do que metade da que estiver antes dela na pilha
  - pilha apenas ocupa ***lg N*** no pior caso
    - que ocorre agora quando a partição ocorre sempre no meio da tabela
    - em ficheiros aleatórios o tamanho máximo da pilha é bastante menor

Propriedade: se a maior das duas sub-tabelas é ordenada primeiro a pilha nunca necessita mais do que ***lg N*** entradas quando *quicksort* é usado para ordenar ***N*** elementos

Demonstração: no pior caso o tamanho da pilha é inferior a T_N em que T_N satisfaz a recorrência

$$T_N = T_{\lfloor N/2 \rfloor} + 1 \qquad (T_N = T_N\, 0)$$

que foi já estudada anteriormente

Quicksort - Melhoramentos [1]

- Algoritmo pode ainda ser melhorado com alterações triviais
  - porquê colocar ambas as sub-tabelas na pilha se uma delas é de imediato retirada?
  - Teste para `r <= l` é feito assim que as sub-tabelas saem da pilha
    - seria melhor nunca as lá ter colocado!
    - parece insignificante mas a natureza recursiva de *quicksort* garante que uma fracção grande das sub-tabelas terão tamanho ***0*** ou ***1***
  - ordenação de ficheiros/sub-tabelas de pequenas dimensões pode ser efectuada de forma mais eficiente
  - como escolher "correctamente" o elemento de partição?
  - Como melhorar o desempenho se os dados tiverem um grande número de chaves repetidas?

De seguida veremos como efectuar alguns destes melhoramentos

Quicksort - Melhoramentos [2]

- Pequenos ficheiros/sub-tabelas
  - um programa recursivo é garantido instanciar-se a si próprio múltiplas vezes para pequenos ficheiros!
  - conveniente utilizar o melhor método possível quando encontra tais ficheiros
  - forma óbvia de obter este comportamento é mudar o teste no início da função recursiva para uma chamada a *insertion sort*

    ```
    if (r-l <= M) insertion(a, l, r)
    ```

    em que ***M*** é um parâmetro a definir na implementação
  - outra solução é a de simplesmente ignorar ficheiros pequenos (tamanho menor que ***M***) durante a partição:

    ```
    if (r-l <= M) return;
    ```

    neste caso no final teremos um ficheiro que está praticamente todo ordenado

➔ boa solução neste caso é usar *insertion sort*

- algoritmo híbrido: bom método em geral!

Quicksort - Melhoramentos [3]

- Utilizar um elemento de partição que com alta probabilidade divida o ficheiro pela metade
  - pode-se usar um elemento aleatoriamente escolhido
    - evita o pior caso (I.e. pior caso tem baixa probabilidade de acontecer)
    - é um exemplo de um algoritmo probabilistico
      - um que usa aleatoriedade para obter bom desempenho com alta probabilidade independentemente dos dados de entrada
    - no caso de *quicksort* ter um gerador de números aleatórios não se justifica
  - pode-se escolher alguns (ex: três) elementos do ficheiro e usar a mediana dos três como elemento de partição
    - escolhendo os três elementos da esquerda, meio e direita da tabela podemos incorporar sentinelas na ordenação
    - ordenamos os três elementos, depois trocamos o do meio com ***a[r-1]*** e corremos o algoritmo departição em ***a[l+1] … a[r-2]***

    ➔ este melhoramento chama-se o método da **mediana de três**
      - *median - of - three*

Quicksort - Melhoramentos [4]

- Método da media de três melhora *quicksort* por três razões
  - o pior caso é mais improvável de acontecer na prática
    - dois dos três elementos teriam de ser dos maiores ou menores do ficheiro
      - e isto teria de acontecer constantemente a todos os níveis de partição
  - elimina o uso de uma sentinela para a partição
    - esta função pode ser feita por um dos três elementos analisados
  - reduz o tempo médio de execução do algoritmo
    - embora apenas por cerca de 5%
  - caso particular de métodos em que se faz amostragem dos dados para estimar as suas propriedades

  ➔ junto com o método de tratar de pequenos ficheiros pode dar ganhos de 20 a 25%

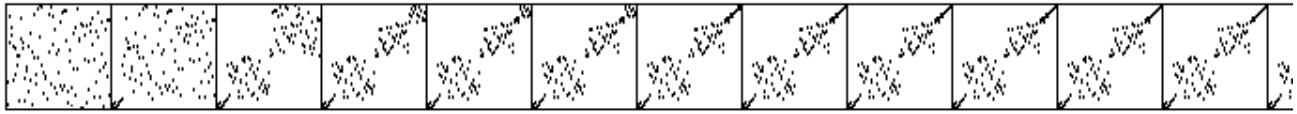
- É possível pensar em outros melhoramentos mas o acréscimo de eficiência é marginal

  ex: porque não fazer a mediana de cinco?

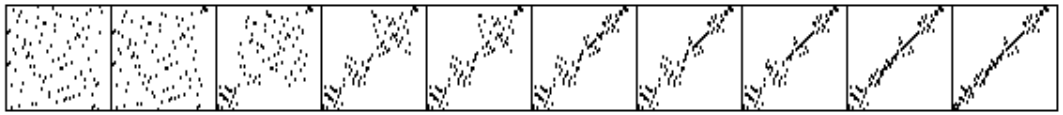
Quicksort - Melhoramentos [5]

Método Standard

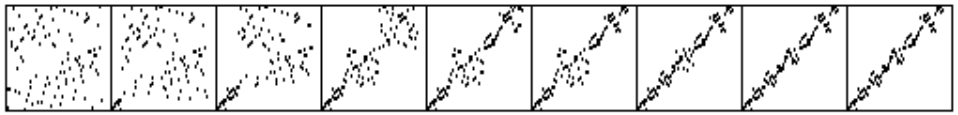
- pode ter má (1ª no exemplo) ou boa partição (2ª no exemplo)



Tratamento de ficheiros pequenos



Mediana de três



Quicksort - Melhoramentos [6]

É bastante melhor do que parece!!

Quicksort - Estudo empírico

		Quicksort básico			***Quicksort melhorado***		
N	***shellsort***	***M=0***	***M=10***	***M=20***	***M=0***	***M=10***	***M=20***
12500	***6***	***2***	***2***	***2***	***3***	***2***	***3***
25000	***10***	***5***	***5***	***5***	***5***	***4***	***6***
50000	***26***	***11***	***10***	***10***	***2***	***9***	***14***
100000	***58***	***24***	***22***	***22***	***25***	***20***	***28***
200000	***126***	***53***	***48***	***50***	***52***	***44***	***54***
400000	***278***	***116***	***105***	***110***	***114***	***97***	***118***
800000	***616***	***255***	***231***	***241***	***252***	***213***	***258***

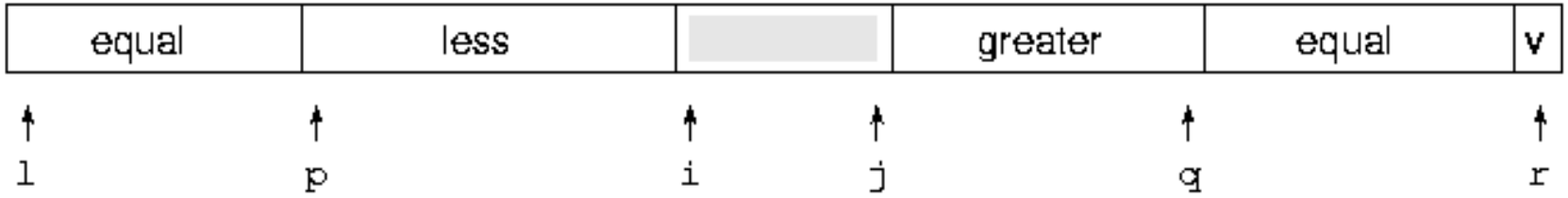
- *Quicksort* é cerca de 2 vezes mais rápido que *shellsort* para ficheiros grandes aleatoriamente ordenados.
- Usando *insertion* para pequenos ficheiros e a estratégia de mediana-de-três melhoram cada um a eficiência por um factor de 10%

Quicksort - Chaves duplicadas [1]

- Ficheiros com um grande número de chaves duplicadas são frequentes na prática
  - desempenho de *quicksort* pode ser substancialmente melhorado
    - se todas as chaves forem iguais
      - *quicksort* mesmo assim faz ***N lg N*** comparações
    - se houver duas chaves distintas
      - reduz-se ao problema anterior para cada sub-ficheiro
      - é melhor completar a ordenação com uma única partição
  - natureza recursiva de *quicksort* garante que haverá frequentemente sub-ficheiros de items com poucas chaves
  - uma possibilidade é dividir o ficheiro em três partes
    - cada uma para chaves menores, iguais e maiores que o elemento de partição
    - não é trivial de implementar, sobretudo se se impuser que a ordenação deverá ser feita com apenas uma passagem pelos dados

Quicksort - Chaves duplicadas [2]

- Solução simples para este problema é fazer uma partição em três partes
  - manter chaves iguais ao elemento de partição que são encontradas no sub-ficheiro da esquerda do lado esquerdo do ficheiro
  - manter chaves iguais ao elemento de partição que são encontradas no sub-ficheiro da direita do lado direito do ficheiro



- Quando os ponteiros/índices de pesquisa se cruzam sabemos onde estão os elementos iguais ao de partição e é fácil colocá-los em posição
  - não faz exactamente tudo num só passo mas quase...
  - trabalho extra para chaves duplicadas é proporcional ao número de chaves duplicadas: funciona bem se não houver chaves duplicadas
  - **linear** quando há um número constante de chaves!!

Quicksort - Partição em três [1]

```
#define eq(A,B) (!less(A,B) && !less(B,A))

void quicksort(Item a[], int l, int r)
{
   int i, j, k, p, q;
   Item v;

   if (r <= l) return;
   v = a[r]; i = l-1; j = r; p = l-1; q = r;
   for (;;) {
      while (less(a[++i], v)) ;
      while (less(v, a[--j])) if (j == l) break;
      if (i >= j) break;
      exch(a[i], a[j]);
      if (eq(a[i],v)) {
         p++; exch(a[p],a[i]);
      }
      if (eq(v,a[j])) {
         q--; exch(a[q],a[j]);
     }
   }
   exch(a[i], a[r]); j = i-1; i = i+1;
   for (k = l  ; k < p; k++, j--) exch(a[k], a[j]);
   for (k = r-1; k > q; k--, i++) exch(a[k], a[i]);
   quicksort(a, l, j);
   quicksort(a, i, r);
}
```

Selecção [1]

- É uma operação importante relacionada com ordenação mas para a qual uma ordenação completa não é necessária
  - exemplo é calculo de mediana de um conjunto de dados
    - ou muitas outras operações estatísticas ou de amostragem
  - com generalidade pode ser descrito como o problema de encontrar o conjunto dos ***k***-menor números

    ex: seja a tabela ***[15, 3, 47, 9, 12, 0]***. O 3º menor element é o ***9***

- Um algoritmo para este problema é o *Selection sort*
  - 1º procura o menor elemento, depois o 2º menor, etc
  - se ***k*** for pequeno custo é ***N k***
  - há outros métodos de custo ***N log k***

- Melhor solução pode ser obtida usando a partição de *quicksort*
  - custo linear em média!

Selecção [2]

- Partição coloca um elemento na sua posição final: ***i***
  - elementos à esquerda são menores que ***a[i]***
  - elementos à direita são maiores que ***a[i]***

- Dada a posição do ***k***-ésimo menor elemento faz-se uma partição
  - se ***i*** = ***k***, terminamos; se ***i*** > ***k*** continuamos no sub-ficheiro da direita; se ***i*** < ***k*** continuamos no sub-ficheiro da esquerda

Propriedade: selecção baseada em *quicksort* é linear em média

- No pior caso, tempo de execução é quadrático, como *quicksort*

  ex: procurar o menor elemento num ficheiro já ordenado

Junção versus partição [1]

- *Quicksort* como vimos é baseado na operação de selecção
  - fazer selecção é semelhante a dividir um ficheiro em duas partes
  - é efectuada uma partição e quando as duas metades do ficheiro estão ordenadas, o ficheiro está ordenado
- Operação complementar é de **junção** (***merge***)
  - combinar dois ficheiros para obter um, maior, ordenado
  - dividir os ficheiros em duas partes para serem ordenados e depois combinar as partes de forma a que o ficheiro total fique ordenado

  ➔ *mergesort*

- *Mergesort* tem uma propriedade muito interessante:
  - ordenação de um ficheiro de ***N*** elementos é feito em tempo proporcional a ***N log N***, **independentemente dos dados**!!

Junção de dois ficheiros ordenados [1]

- Dados dois ficheiros combinamo-los para obter um único ficheiro ordenado
  - em cada iteração um elemento é retirado de ***a*** ou ***b*** e colocado em ***c***
  - termina quando ambos os ficheiros de entrada foram lidos
  - trivial mas utiliza **espaço adicional** proporcional aos dados!!

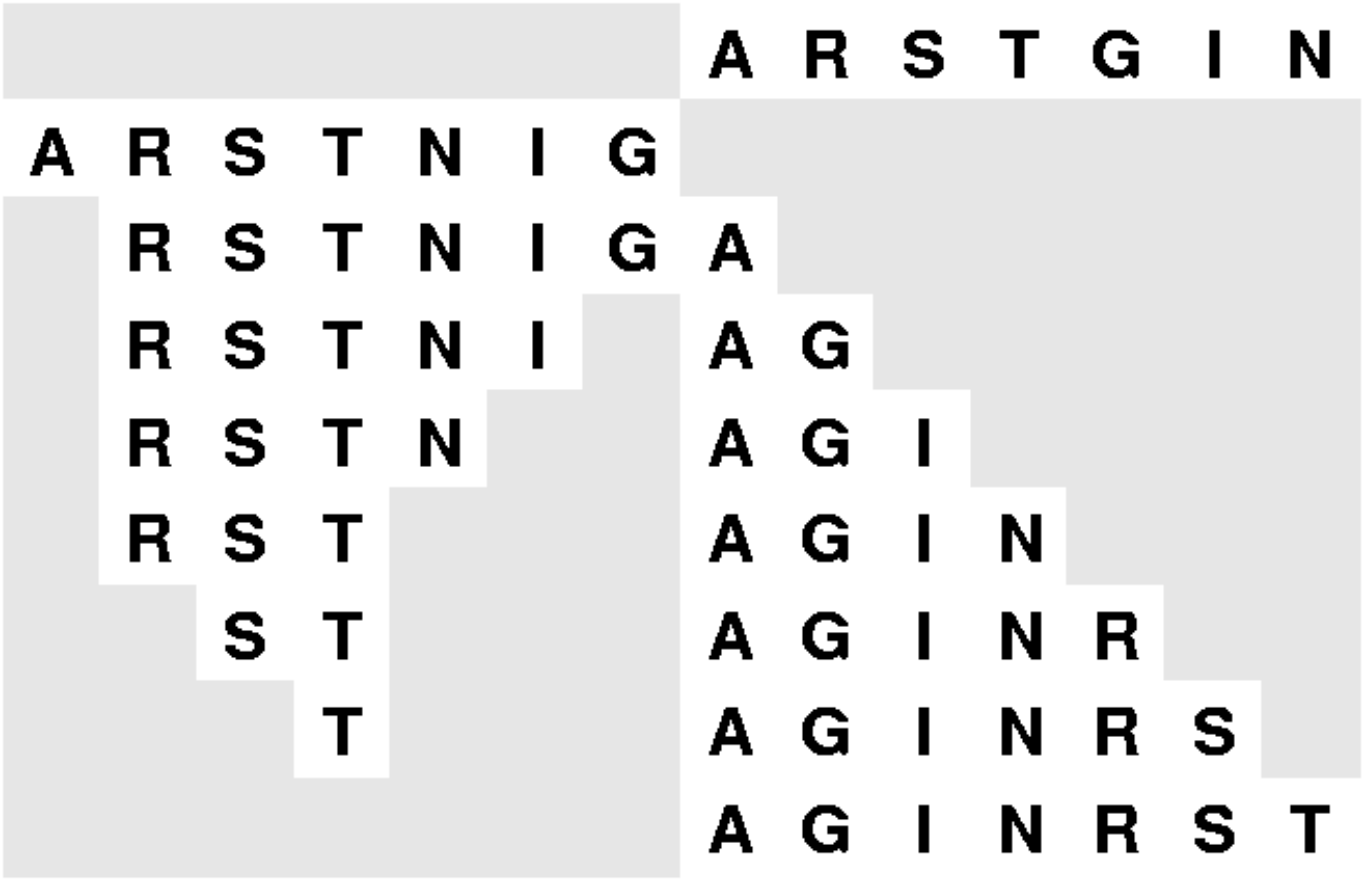
```
merge(Item c[], Item a[], int N, Item b[], int M)
{
     int i, j, k;

    for (i = 0, j = 0, k = 0; k < N+M; k++)  {
        if (i == N)  {
            c[k] = b[j++];  continue;
        }
        if (j == M) {
            c[k] = a[i++]; continue;
        }
        if (less(a[i], b[j]))
           c[k] = a[i++]; else c[k] = b[j++];
     }
}
```

Junção de dois ficheiros ordenados [2]

- Implementação (aparentemente) sem espaço adicional (*in-place*)
  - copiamos ***a*** e ***b*** para uma tabela extra ***c*** com ***b*** em ordenação revertida
  - ***c*** é espaço adicional mas não para quem chama função (e pode ser evitado)
  - evita testes pelo fim das tabelas ***a*** e ***b***
  - sequência de chaves aumenta e depois diminui sequência bitónica

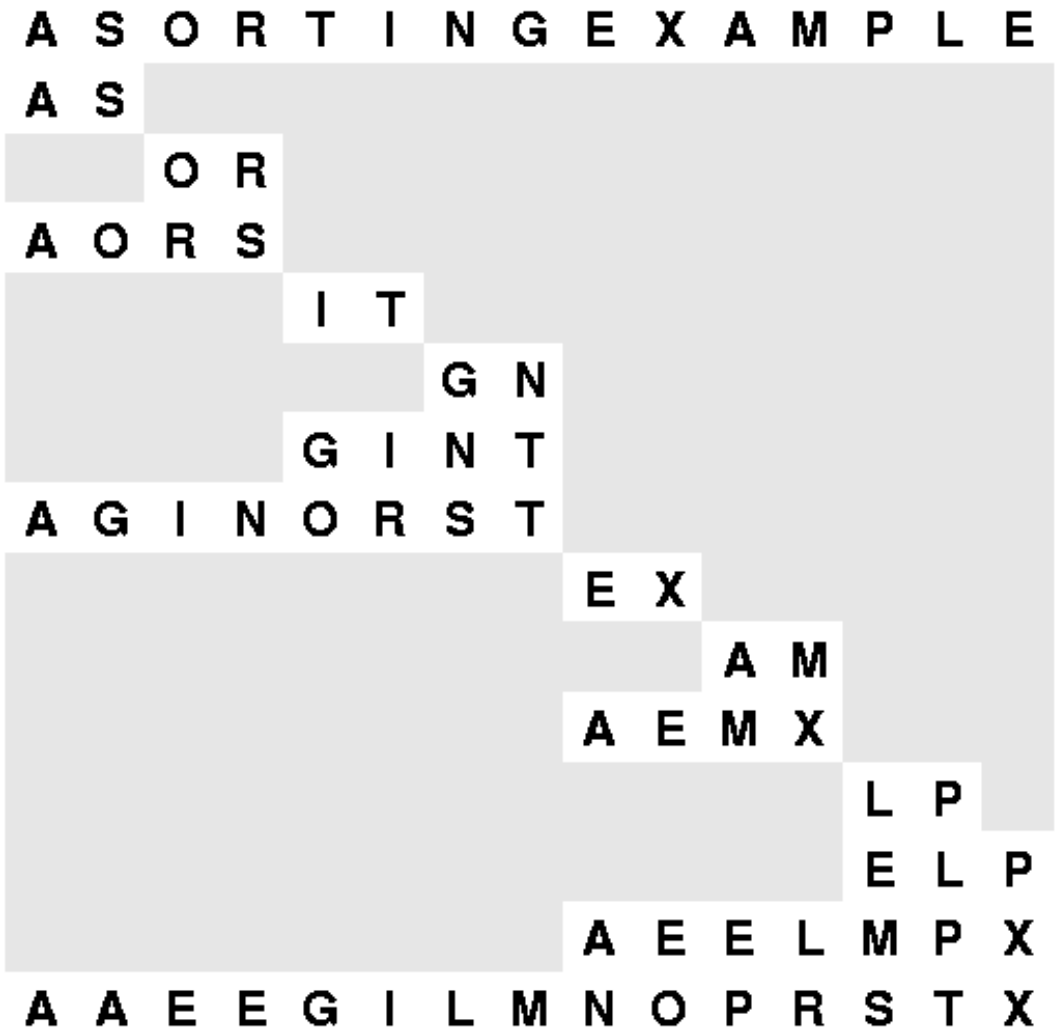
```
Item aux[maxN];

merge(Item a[], int l, int m, int r)
{
   int i, j, k;

   for (i = m+1; i > l; i--) aux[i-1] = a[i-1];
   for (j = m; j < r; j++) aux[r+m-j] = a[j+1];
   for (k = l; k <= r; k++)
      if (less(aux[i], aux[j]))
         a[k] = aux[i++];
      else a[k] = aux[j--];
}
```

Junção de dois ficheiros ordenados [3]



Mergesort

Ordenar um ficheiro:
ordenando duas metades do mesmo (recursivamente) e depois fazendo a junção dos resultados



Implementação de *mergesort* - Versão recursiva *(descendente)*

- Para ordenar um ficheiro (processo *top-down mergesort*)
  - divide-se em duas metades
  - ordenam-se essas metades (recursivamente)
  - faz-se a junção dos resultados

```
void mergesort(Item a[], int l, int r)
{
   int m = (r+l)/2;

   if (r <= l) return;
   mergesort(a, l, m);
   mergesort(a, m+1, r);
   merge(a, l, m, r);
}
```

Mergesort - Propriedades

- Complexidade é garantidamente ***N log N***
  - mas requer espaço extra proporcional a ***N***!
    - dados apenas definem a ordem em que os elementos são processados nas junções
  - complexidade garantida pode ser um problema
    - não pode haver nenhum caso melhor
    - não se pode adaptar consoante os dados
- É um algoritmo **estável**
  - (se a junção o for)
  - propriedade que muitas vezes faz com que seja escolhido!
- Normalmente implementado de forma a aceder aos dados sequencialmente
  - por vezes o único processo de acesso
  - muito usado para ordenação de listas ligadas

Implementação de *mergesort* - Versão não-recursiva *(ascendente)*

- Tal como em *quicksort* também há uma versão não-recursiva de *mergesort (*computações efectuadas por outra ordem)
  - sub-ficheiros são processados independentemente, logo junções podem ser feitas numa sequência diferente
    - mesmo conjunto de divisões e de junções
    - apenas as junções são efectuadas por outra ordem
      - primeiro as de ficheiros com um elemento, depois as de 2 elementos, etc
    - em cada passo o tamanho dos sub-ficheiros ordenados duplica

- Estratégia de **dividir para conquistar** é substituida por uma de **combinar para conquistar** (na prática é o mesmo)

Implementação: no livro...

Mergesort - Exemplo Gráfico

- *Mergesort* ascendente (cima) consiste numa série de passagens pelo ficheiro fazendo a junção de sub-ficheiros ordenados até apenas existir um
- *Mergesort* descendente (baixo) ordena a primeira metade do ficheiro antes de prosseguir para a outra metade (recursivamente)