Algoritmos avançados - *Quicksort*

- Provavelmente o algoritmo mais usado
  - inventado em 1960 por Charles Hoare
  - muito estudado e analisado
    - desempenho bem conhecido
  - popular devido à facilidade de implementação e eficiência
    - complexidade $N \log_2 N$, em média, para ordenar N objectos
    - ciclo interno muito simples e conciso

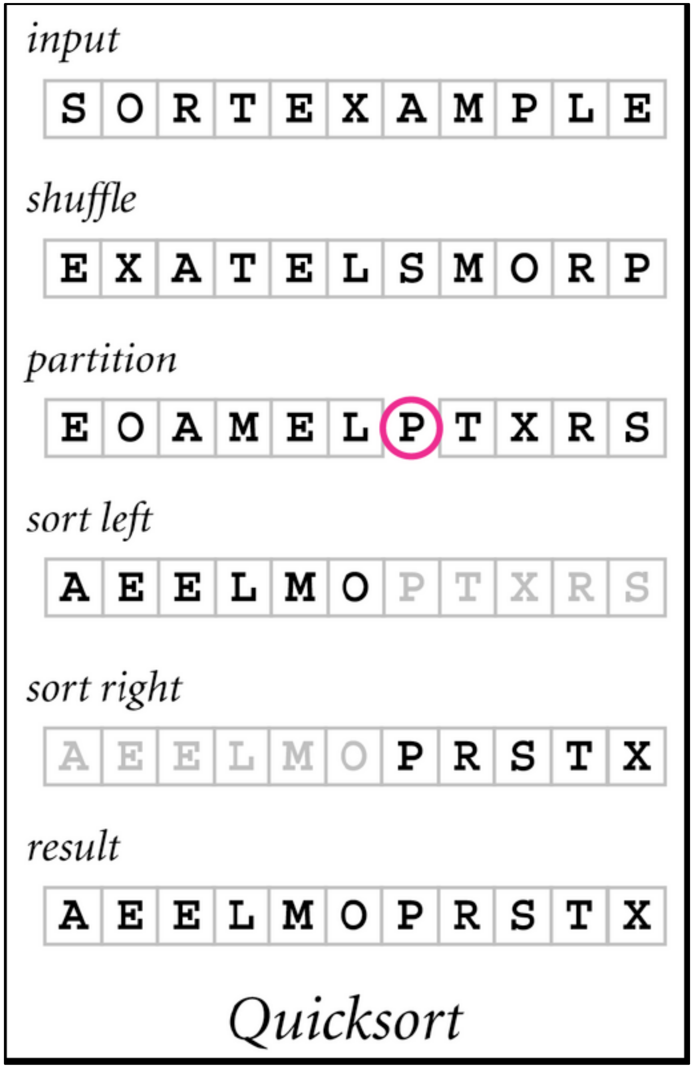
Quicksort

- *Quicksort* realiza 39% mais comparações do que o *Mergesort*
- Contudo, é mais rápido que o *Mergesort* na prática devido ao menor custo de outras instruções de alta frequência

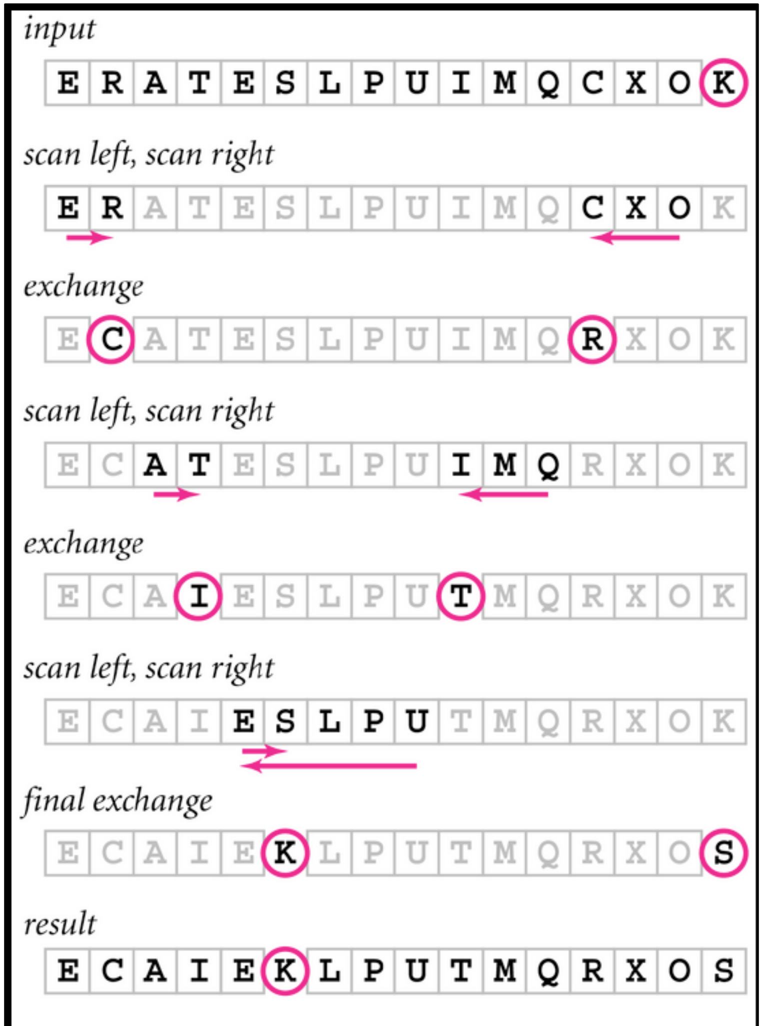
- Mas:
  - não é estável
  - quadrático *(N^2)* no pior caso!
  - “frágil”: qualquer pequeno erro de implementação pode não ser detectado mas levar a ineficiência

Quicksort

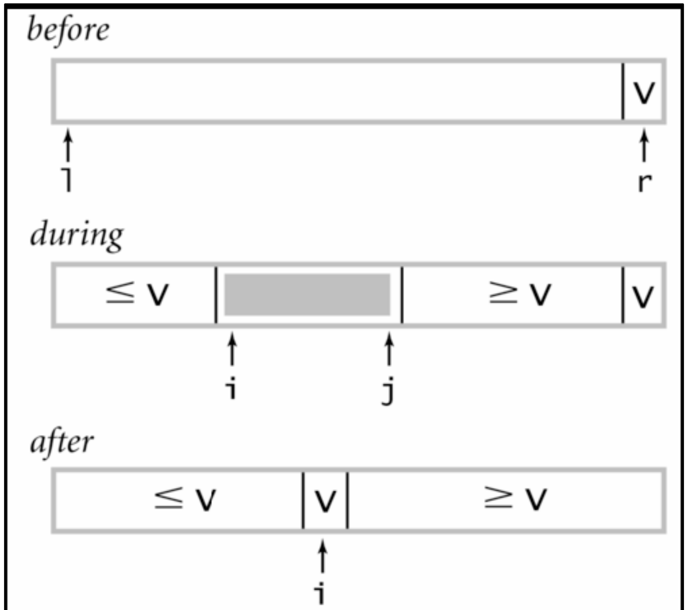
- Algoritmo do tipo *dividir para conquistar*
- Algoritmo *Quicksort:*
  - (**Opcional**): “Baralhar” (*Shuffle*) o *array*
  - Particionar (*Partition*) o *array* da seguinte forma:
    - elemento a[i] fica na sua posição final para um determinado i
    - não existem elementos maiores do que a[i] à esquerda de i
    - não existem elementos menores do que a[i] à direita de i
  - Ordenar cada partição recorrentemente



Quicksort - Partição

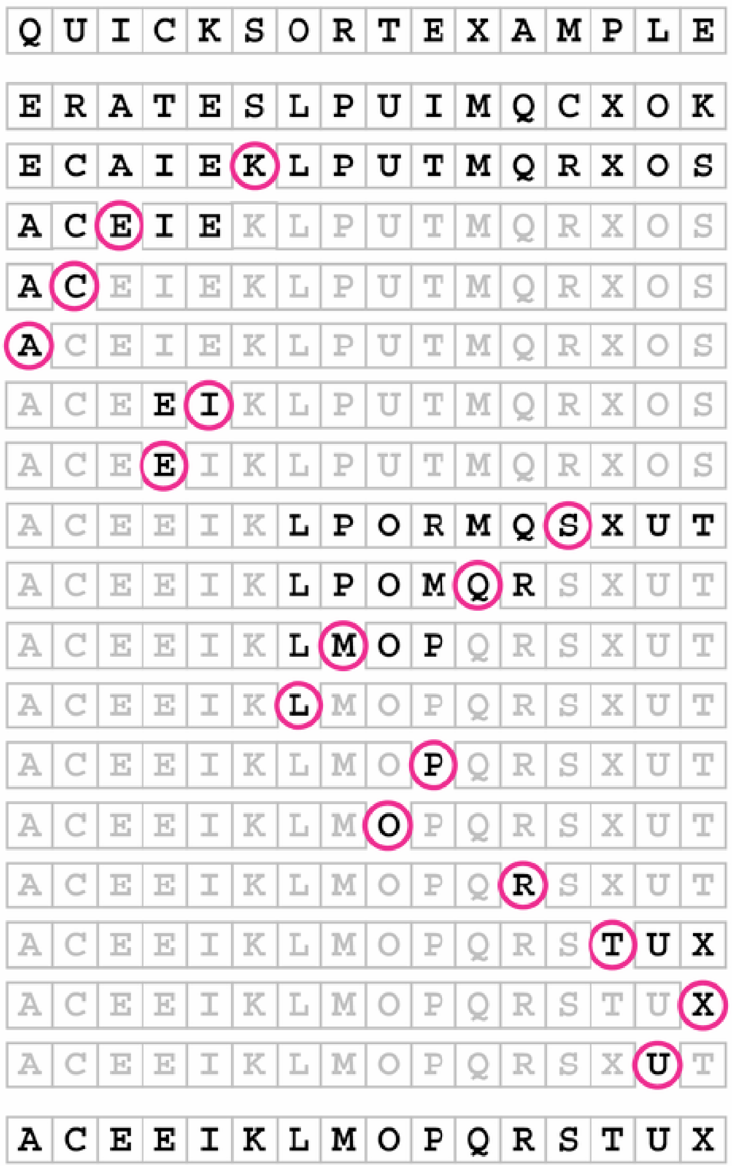


Partitioning example



Partitioning for quicksort

Quicksort - Exemplo



Full quicksort example

Estratégia para a partição

- Escolher *a[r]* para ser o elemento de partição
  - o que é colocado na posição final
- Percorrer o *array* a partir da esquerda até encontrar um elemento maior que ou igual ao elemento de partição (*a[r]*)
- Percorrer o *array* 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 os índices se cruzam
  - completa-se trocando *a[r]* com o elemento referenciado pelo índice *i* (que varre da esquerda para a direita)

Quicksort - Implementação

```
public class IntArraySort {
    …
    public static void quicksort(int[] a, int l, int r) {
        shuffle(a, l, r);  // Opcional
        qsort(a, 0, a.length - 1);
    }
    private static void qsort(int[] a, int l, int r) {
        if (r <= l) return;
        int m = partition(a, l, r);
        qsort(a, l, m-1);
        qsort(a, m+1, r);
    }

    …
```

Quicksort - Implementação

```
private static int partition(int[] a, int l, int r) {
    int i = l-1, j = r; int v = a[r];
    for (;;) {
        // Encontra índice de elemento >= pivot à esquerda
        while (less(a[++i], v)) ;
        // Encontra índice elemento<=pivot à direita
        while (less(v, a[--j]))
            if (j == l) break;
        if (i >= j)
            break;
        exch(a, i, j);
    }
    exch(a, i, r);  // Troca com pivot de partição
    return i; // Retorna índice do pivot
}
```

Quicksort - Partição

- 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
- 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 arrays

Quicksort – Características de desempenho

- Se não se usar o *shuffling*, o *Quicksort* pode ser muito ineficiente em casos patológicos
- **Propriedade:** *Quicksort* usa cerca de $N^2/2$ comparações no <u>pior caso</u>
  - No pior caso, o número de comparações usadas por *Quicksort* satisfaz a recorrência de dividir para conquistar

$$C(n) = C(n-1) + O(n)$$

    - 1º termo cobre o custo de ordenar um sub-ficheiro (partição degenerada)
    - 2º termo refere-se a examinar cada elemento
    - $C(n) = O(n^2)$

Quicksort – Análise do pior caso

- 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 <u>quadraticamente</u> como o espaço necessário para o processo recorrente é de cerca de N o que é <u>inaceitável</u> para ficheiros grandes

Quicksort – Características de desempenho

- Melhor caso: quando cada partição divide o ficheiro de entrada exactamente em metade
  - número de comparações usadas por *Quicksort* satisfaz a recorrência de dividir para conquistar

$$C(n) = 2C(n/2) + O(n)$$

    - 1º termo cobre o custo de ordenar os dois sub-ficheiros
    - 2º termo refere-se a examinar cada elemento
- solução é $C(n) = O(n \log_2 n)$ (vimos numa aula anterior)
- Espaço usado nas chamadas recorrentes é $\log_2(n)$
  - Demonstração...

Quicksort – Características de desempenho

- **Propriedade:** *Quicksort* usa cerca de $O(2N \mathbf{log_2} N)$ comparações <u>em média</u>
- **Demonstração:** A fórmula de recorrência exacta para o número de comparações utilizado por *Quicksort* para ordenar *N* números distintos aleatoriamente posicionados é

$$C(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C(k-1) + C(n-k)) \qquad n \geq 2, C(0) = C(1) = 0$$

  - 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/n* de ser o elemento de partição após o que ficamos com dois sub-arrays de tamanhos *k-1* e *n-k*

Quicksort – Características de desempenho

- Demonstração (cont.)

$$C(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C(k-1) + C(n-k))$$

$$= n + 1 + \frac{2}{n}\sum_{k=1}^{n}C(k-1)$$

- Multiplicar ambos os lados da equação por *n* e subtrair a mesma fórmula por *n-1*

$$nC(n) - (n-1)C(n-1) = n(n+1) - (n-1)n + 2C(n-1)$$

- Simplificar para:

$$nC(n) = (n+1)C(n-1) + 2n$$

Quicksort – Características de desempenho

- Dividir ambos os lados da equação por *n(n-1)* de forma a obter a soma:

$$\begin{aligned}\frac{C(n)}{n+1} &= \frac{C(n-1)}{n} + \frac{2}{n+1}\\ &= \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}\\ &= \frac{C(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}\\ &= \vdots\\ &= \frac{C(2)}{3} + \sum_{k=3}^{n} \frac{2}{k+1}\end{aligned}$$

- Aproximar a resposta exacta por um integral:

$$\frac{C(n)}{n+1} = \sum_{k=1}^{n} \frac{2}{k} \approx \int_{k=1}^{n} \frac{2}{k} = 2\ln n$$

- Finalmente, obtém-se a solução: $C(n) = 2(n+1)\ln n \approx 1.39n \log_2 n.$

Quicksort – Características de desempenho

- 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

Ordenação – Síntese da Aula 4

- Algoritmo *Quicksort*
  - Ideia chave + Motivação
    - Algoritmo que recorre à divisão em instâncias menores – *“divide and conquer”*
  - Código
  - Exemplo de aplicação
  - Descrição detalhada do mecanismo de partição
  - Análise de eficiência
    - Pior caso
    - Melhor caso
    - Caso médio

Quicksort – Questões mais relevantes

- Possível redução de desempenho devido ao uso de recorrência
- 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 recorrência:
  - recorrência implica chamada a função e logo a carregar dados na pilha/*stack* do computador
  - no pior caso todas as partições degeneram e há $O(N)$ níveis de recorrência (em vez de $O(\mathbf{log}_2(N))$ no melhor caso)
    - pilha cresce até ordem N

Quicksort - Espaço necessário (1)

- Para resolver este problema usamos uma pilha (*stack*) explícita
  - pilha contém "trabalho" a ser processado, na forma de sub-arrays a ordenar
  - quando precisamos de um sub-array para processar tiramo-lo da pilha (i.e. fazemos um *pop()* do *stack*)
  - por cada partição criamos dois sub-arrays e metemos ambos na pilha (i.e. fazemos dois *push()* para o *stack*)
  - substitui a pilha do computador que é usada na implementação recorrente

Quicksort - Espaço necessário (2)

- Conduz a uma versão não recorrente de *Quicksort*
  - verifica os tamanhos dos dois subarrays e põe o maior deles primeiro na pilha (e o menor depois; logo o menor é retirado e tratado primeiro)
  - ordem de processamento dos subarrays não afecta a correcta operação da função ou o tempo de processamento mas afecta o tamanho da pilha
- No pior caso espaço extra para a ordenação é logarítmico em *N*
  - garante que dimensão máxima do *stack* é $O(\mathbf{log}_2 N)$

Quicksort - Versão não-recursiva (1)

```
static void nonRecursiveQuicksort(int[] a, int l, int r) {
	shuffle(a, l, r); // Shuffle
	IntStack s = new IntStackArray(50);
	s.push(l); s.push(r);
	while (!s.isEmpty()) {
		r = s.pop(); l = s.pop();
		if (r <= l) continue;
		int i = partition(a, l, r);
		if (i-l > r-i) { s.push(l); s.push(i-1); }
		s.push(i+1); s.push(r);
		if (r-i >= i-l) { s.push(l); s.push(i-1); }
	}
}
```

Quicksort - Versão não-recursiva (2)

- Política de colocar o maior dos subarrays 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 $\mathbf{log}_2\ N$ 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 o menor dos dois subarrays é ordenado primeiro a pilha nunca necessita mais do que $\mathbf{log}_2\ N$ entradas quando *Quicksort* é usado para ordenar *N* elementos

Quicksort - Versão não-recursiva (3)

- **Demonstração:** no pior caso o tamanho da pilha é inferior a $T(n)$ em que $T(n)$ satisfaz a recorrência

$$T(n) = T_{\lfloor n/2 \rfloor} + 1 \qquad \text{com} \quad T(0) = T(1) = 0$$

que foi já estudada anteriormente

Quicksort - Melhoramentos (1)

- Algoritmo pode ainda ser melhorado com alterações triviais
  - porquê colocar ambos os subarrays na pilha se um deles é de imediato retirado?
  - Teste para **r <= l** é feito assim que os subarrays saem da pilha
    - seria melhor nunca os lá ter colocado!
  - ordenação de ficheiros/subarrays 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?

Quicksort - Melhoramentos (2)

- Pequenos ficheiros/sub-arrays
  - Até mesmo o *QuickSort* apresenta um *overhead* excessivo quando é usado com ficheiros de pequena dimensão
    - conveniente utilizar o melhor método possível quando encontra tais ficheiros
  - forma óbvia de obter este comportamento é mudar o teste no início da função recorrente para uma chamada a *Insertion sort*

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

em que *M* é um parâmetro a definir na implementação

Quicksort - Melhoramentos (3)

- Pequenos ficheiros/subarrays
  - 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 (4)

- 1º Método:
  - Utilizar um elemento de partição que com alta probabilidade divida o ficheiro pela metade
    - pode usar-se um elemento aleatoriamente escolhido
      - evita o pior caso (i.e. pior caso tem baixa probabilidade de acontecer)
      - é um exemplo de um algoritmo probabilístico
        - um que usa aleatoriedade para obter bom desempenho com alta probabilidade independentemente dos dados de entrada
    - pode ser demasiado pesado incluir gerador aleatório no algoritmo; existem outros métodos igualmente simples

Quicksort - Melhoramentos (5)

- 2º Método:
  - pode escolher-se 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 executamos o algoritmo de partição em *a[l+1] … a[r-1]*
  - Este melhoramento chama-se o método da **mediana de três**
    - *median - of - three*

Mediana de três - Implementação

```
private final static int M = 10;
static void quickSort(int[] a, int l, int r) {
        if (r-l <= M) return;
        exch(a, (l+r)/2, r-1);
        lessExch(a, r-1, l);
        lessExch(a, r, l);
        lessExch(a, r, r-1);
        int i = partition(a, l+1, r-1);
        quicksort(a, l, i-1);
        quicksort(a, i+1, r);
}
static void hybridSort(int a[], int l, int r)
   {   quicksort(a, l, r); insertionSort(a, l, r);  }
```

Quicksort - Melhoramentos (6)

- Método da mediana de três melhora *Quicksort* por duas 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
  - reduz o tempo médio de execução do algoritmo
    - embora apenas por cerca de 5%
  - junto com o método de tratar de pequenos ficheiros pode resultar em 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 - Chaves duplicadas (1)

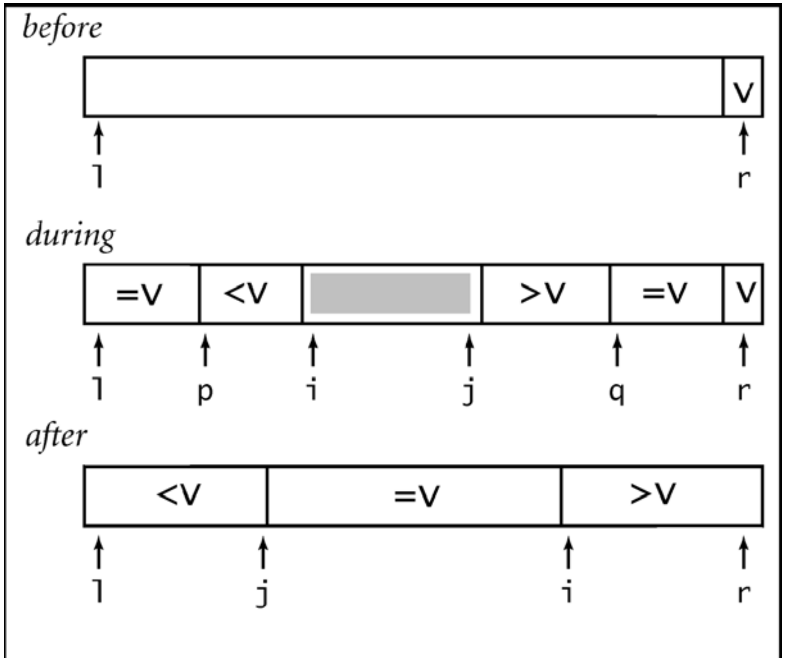
- Ficheiros com um grande número de chaves duplicadas são frequentes na prática
  - ex: ordenar população por idade; remover duplicados de uma lista
  - desempenho de *Quicksort* pode ser substancialmente melhorado
    - se todas as chaves forem iguais
      - *Quicksort* mesmo assim faz $O(N \log_2 N)$ comparações

Quicksort - Chaves duplicadas (2)

- 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
- 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

Quicksort - Chaves duplicadas (3)

- 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!!



Partitioning with duplicate keys

Quicksort - Partição em três

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

Junção versus partição

- *Quicksort* é baseado na operação de selecção do elemento de partição (*pivot*)
  - a partição divide um ficheiro em duas partes
  - quando as duas metades do ficheiro estão ordenadas, o ficheiro está ordenado
- Operação complementar é de **junção (*merge*)**
  - dividir o ficheiro 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*, <u>independentemente dos dados</u>!
- Outros algoritmos, p.ex. *HeapSort*, também têm desempenho desta ordem

Ordenação – Síntese da Aula 5

- Análise do algoritmo *Quicksort*
  - Discussão relativa à memória utilizada na versão recorrente
  - Alternativa de implementação por pilha e suas vantagens na perspectiva da memória utilizada
  - Código para *Quicksort* em versão não recorrente
- Melhoramentos na versão não recorrente
  - Mecanismos de partição alternativos
    - Aleatórios
    - Mediana de três
  - Estratégia de melhoramento em presença de chaves duplicadas
    - Ideia base
    - Código