Recursividade

Pedro Ribeiro

DCC/FCUP

2020/2021

Recursividade

Matemática:

- Uma fórmula diz-se **recursiva** quando inclui referências a si própria
  Exemplos:
  - Factorial: $n! = n \times (n-1)!$
  - Fibonacci: $fib(n) = fib(n-1) + fib(n-2)$

Ciência de Computadores:

- Uma função diz-se **recursiva** quando inclui chamadas a si própria
  - Uma função não-recursiva é **iterativa**

- A **recursividade** é uma das técnicas algorítmicas mais usadas
  - Muitas soluções exprimem-se muito mais facilmente de forma recursiva
  - Vamos usar muita recursividade, por exemplo, com as árvores binárias

- Nesta aula vamos ver vários **exemplos de recursividade**

Recursividade: um primeiro exemplo

- Vamos imaginar que temos um array de inteiros e queremos **descobrir o maior número** entre as posições *start* e *end*.
- A solução **iterativa** típica implica fazer um ciclo entre *start* e *end* e ir guardando o maior até ao momento:

```
int maxIt(int v[], int start, int end) {
   int maxSoFar = v[start];
   for (int i=start+1; i<=end; i++)
      maxSoFar = Math.max(maxSoFar, v[i]);
   return maxSoFar;
}
```

Nota: **Math** é uma classe com várias funções de cariz matemático:
https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html
Em particular, `Math.max(a,b)` devolve o maior número entre *a* e *b*

Recursividade: um primeiro exemplo

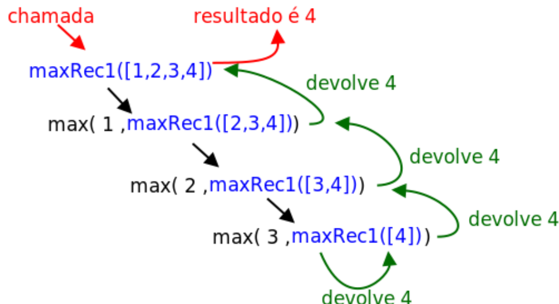
- Como podiamos definir **recursivamente** o máximo de uma lista?

- Em qualquer **função recursiva** precisamos tipicamente do seguinte: (relembrem a noção de dividir para conquistar)
  - **Caso base:** caso "pequeno" onde devolvemos a solução sem recursão
  - **Dividir** o problema em um ou mais casos mais pequenos que o original, mais próximos do caso base
  - **Chamar recursivamente** a função para os casos mais pequenos
  - **Combinar** os resultados recursivos para obter a solução global

Recursividade: um primeiro exemplo

- Como podemos **partir o array** em pedaços mais pequenos?
  - Uma hipótese é considerar todo o array excepto o primeiro elemento: max(v) = máximo entre 1º elemento e max(resto do array v)
- O **caso base** é um array de tamanho 1: o máximo é esse elemento

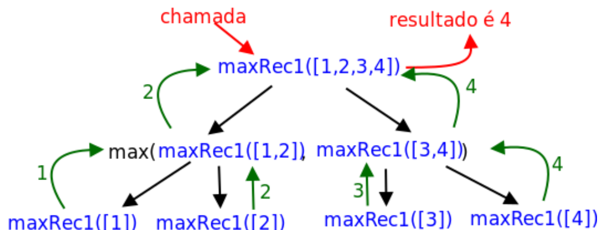
```
int maxRec1(int v[], int start, int end) {
   if (start == end) return v[start];  // caso base
   int max = maxRec1(v, start+1, end); // chamada recursiva
   return Math.max(v[start], max);     // combinar resultado
}
```



Recursividade: um primeiro exemplo

- Como podemos partir o array em pedaços mais pequenos?
  - Outra hipótese é partir o array em duas metades:
    max(v) = máximo entre max(metade esquerda) e max(metade direita)
- O **caso base** continua a ser um array de tamanho 1

```
int maxRec2(int v[], int start, int end) {
   if (start == end) return v[start];    // caso base
   int middle = (start + end) / 2;
   int max1 = maxRec2(v, start, middle); // chamada recursiva
   int max2 = maxRec2(v, middle+1, end); // chamada recursiva
   return Math.max(max1, max2);          // combinar resultado
}
```



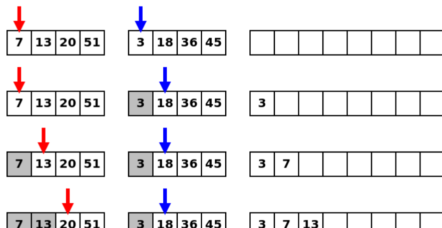
Recursividade: MergeSort

- Vejamos o mesmo padrão agora para ordenar números: **MergeSort** (já detalhamos este algoritmo anteriormente)
  - Partir o array em duas metades
  - Ordenar recursivamente cada uma das metades
  - Juntar as duas metades ordenadas num única array ordenado
- O **caso base** continua a ser um array de tamanho 1: já está ordenado
- Vamos assumir que mergeSort ordena entre posições *start* e *end* (para ordenar tudo é só chamar com argumentos 0 e $tamanho - 1$)

```
void mergeSort(int v[], int start, int end) {
      if (start == end) return;        // caso base
      int middle = (start + end) / 2;
      mergeSort(v, start, middle);     // chamada recursiva
      mergeSort(v, middle+1, end);     // chamada recursiva
      merge(v, start, middle, end);    // combinar resultados
   }
```

Recursividade: MergeSort

```
void merge(int v[], int start, int middle, int end) {
   int aux[] = new int[end-start+1]; // Novo array temporário
   int p1 = start;     // "Apontador" do array da metade esquerda
   int p2 = middle+1; // "Apontador" do array da metade direita
   int cur = 0;        // "Apontador" do array aux[] a conter juncao
   while (p1 <= middle && p2 <= end) { // Enquanto der para comparar
      if (v[p1] <= v[p2]) aux[cur++] = v[p1++]; // Escolher menor
      else aux[cur++] = v[p2++];                // e adicionar
   }
   while (p1<=middle) aux[cur++] = v[p1++]; // Adicionar o que resta
   while (p2<=end)    aux[cur++] = v[p2++];
   // Copiar array aux[] para v[]
   for (int i=0; i<cur; i++) v[start+i] = aux[i];
}
```



Recursividade: alguns cuidados

- Os erros mais comuns com recursividade são:
  - O **caso base foi esquecido**, ou nem todos os casos base foram tratados
  - O caso recursivo não é aplicado a casos mais pequenos, e por isso a **recursão não converge**

- Em ambos os casos a recursão fica **"infinita"** e a função vai esgotar a memória até obter um erro de *Stack Overflow*

  Nota: uma recursão usa "internamente" uma pilha (stack) para guardar o estado das chamadas anteriores
  (quando sai de uma chamada, regressa à última chamada antes dessa)

Recursividade: invertendo um array

- O mais "complicado" é escolher a divisão recursiva ajustada ao problema
- Vejamos mais um caso: como **inverter o conteúdo de um array**?
  Exemplo: $[1, 2, 3, 4, 5] \rightarrow [5, 4, 3, 2, 1]$
  - Se trocarmos o primeiro com o último elemento... fica apenas a faltar **inverter o resto**
  - O **caso base** é qualquer array de tamanho inferior a 2: o array invertido é igual a ele
  - Vamos assumir que estamos a inverter entre posições *start* e *end*

```
void reverse(int v[], int start, int end) {
   if (start>=end) return;      // Caso base:  array de tamanho < 2
   int tmp = v[start];          // Trocar primeiro com último
   v[start] = v[end];
   v[end] = tmp;
   reverse(v, start+1, end-1); // Chamada recursiva para o resto
}
```

Recursividade: flood fill

- Pode dar jeito ter mais do que duas chamadas recursivas
- Considere uma matriz onde duas células são *vizinhas* se são adjacentes *vertical* ou *horizontalmente*
- Uma **mancha** é um conjunto de células vizinhas não vazias. Por exemplo, na figura seguinte, temos 3 manchas:
  - Uma mancha vermelha com 6 células
  - Uma mancha verde com 4 células
  - Uma mancha azul com 3 células

#	#	.	#	.	.	.
.	#	#	#	.	.	.
.	.	.	.	.	#	#
.	#	.	.	.	#	#
#	#	.	.	.	.	.

- Como calcular o **tamanho de uma mancha**?

Recursividade: flood fill

- Definição recursiva:
  - Seja m[L][C] a matriz de células com **L** linhas e **C** colunas
  - Seja $t(y, x)$ o tamanho da mancha na posição (y, x)
  - Se m[y][x] for célula vazia, então t(y,x) = 0
    Caso contrário, t(y,x) = 1 + t(y+1,x) + t(y-1,x) + t(y,x+1) + t(y,x-1)

Uma implementação incorrecta:

```
// Estamos a assumir que m[][], L e C são variáveis globais
int t(int y, int x) {
   if (m[y][x] == '.') return 0; // Caso base:  célula vazia
   int count = 1;                // célula não vazia
   count += t(y-1, x);           // Adicionando células não vizinhas
   count += t(y+1, x);
   count += t(y, x+1);
   count += t(y, x-1);
   return count;
}
```

- Problemas desta implementação? Limites da matriz!
  ex: t(0,0) vai chamar t(-1,0), que tenta aceder a uma posição fora dos limites da matriz

Recursividade: flood fill

- Uma implementação ainda incorrecta:

```
// Estamos a assumir que m[][], L e C são variáveis globais
int t(int y, int x) {
   if (y<0 || y>=L || x<0 || x>=C) return 0;// Caso base:  fora dos limites
   if (m[y][x] == '.') return 0; // Caso base:  célula vazia
   int count = 1; // célula não vazia
   count += t(y-1, x); // Adicionando células não vizinhas
   count += t(y+1, x);
   count += t(y, x+1);
   count += t(y, x-1);
   return count;
}
```

- Problemas desta implementação? Recursão infinita!
  ex: t(0,0) chama t(1,0), que chama t(0,0), que chama t(0,1), que chama t(0,0), que chama t(0,1), ...
- Precisamos de garantir que não voltamos a uma célula que já visitamos

Recursividade: flood fill

- Uma implementação correcta
  - *visited*[*L*][*C*] é uma matriz de booleanos inicialmente com tudo a *false*

```
// Estamos a assumir que m[][], L, C e visited[][] são variáveis globais
int t(int y, int x) {
   if (y<0 || y>=L || x<0 || x>=C) return 0;// Caso base:  fora dos limites
   if (visited[y][x]) return 0;  // Caso base:  célula já visitada
   if (m[y][x] == '.') return 0; // Caso base:  célula vazia
   int count = 1; // célula não vazia
   visited[y][x] = true;
   count += t(y-1, x); // Adicionando células não vizinhas
   count += t(y+1, x);
   count += t(y, x+1);
   count += t(y, x-1);
   return count;
}
```

Recursividade: gerando subconjuntos

- Como **gerar todos os subconjuntos** de um dado conjunto?
  Exemplo: $\{1,2,3\}$ tem 8 subconjuntos:
  - $\{1,2,3\}$, $\{1,2\}$, $\{1,3\}$, $\{2,3\}$, $\{1\}$, $\{2\}$, $\{3\}$, $\{\}$
- **Definição recursiva?** Subconjuntos de $\{1,2,3\}$ são:
  - $\{1\} \cup$ subconjuntos de $\{2,3\}$: $\{1,2,3\}$, $\{1,2\}$, $\{1,3\}$, $\{1\}$
    e
  - $\{\} \cup$ subconjuntos de $\{2,3\}$: $\{2,3\}$, $\{2\}$, $\{3\}$, $\{\}$
- Dito de outro modo, o 1º elemento ou está no conjunto ou não está, e para cada um destes casos, temos todos os subconjuntos do "resto"

Recursividade: gerando subconjuntos

- Seja a inclusão no conjunto representada por array de booleanos:
  Exemplos: [T,T,T] representa {1,2,3}; [T,T,F] representa {1,2}
- Então todos os subconjuntos são:
  - Arrays onde a 1º posição é T + todos os subconjuntos seguintes mais
  - Arrays onde a 1º posição é F + todos os subconjuntos seguintes

  [T,T,T] = {1,2,3}
  [T,T,F] = {1,2}
  [T,F,T] = {1,3}
  [T,F,F] = {1}
  [F,T,T] = {2,3}
  [F,T,F] = {2}
  [F,F,T] = {3}
  [F,F,F] = {}

Recursividade: gerando subconjuntos

- Implementando:

```
// Escrever todos os subconjuntos do array v[]
void sets(int v[]) {
   // array de booleanos para representar o conjunto
   boolean used[] = new boolean[v.length];
   goSets(0, v, used); // chamar função recursiva
}

// Gera todos os subconjuntos a partir da posição 'cur'
void goSets(int cur, int v[], boolean used[]) {
   if (cur == v.length) {  // Caso base: terminamos o conjunto
      for (int i=0; i<v.length; i++) // Escrever conjunto
         if (used[i]) System.out.print(v[i] + " ");
      System.out.println();
   } else {                     // Se não terminamos, continuar a gerar
      used[cur] = true;         // Subconjuntos que incluem o elemento actual
      goSets(cur+1, v, used);// Chamada recursiva
      used[cur] = false;        // Subconjuntos que não incluem o el. actual
      goSets(cur+1, v, used);// Chamada recursiva
   }
}
```

Recursividade: gerando permutações

- Como gerar todas as **permutações** de um array?
  Exemplo: 6 permutações de [1,2,3]: 123, 132, 213, 231, 312, 321
- **Definição recursiva?** Permutações de [1,2,3] são:
  - 1 seguido das permutações de [2,3]
    e
  - 2 seguido das permutações de [1,3]
    e
  - 3 seguido das permutações de [1,2]
- Dito de outro modo, o 1º elemento da permutação pode ser qualquer um dos elementos do array e para cada um desses casos temos de permutar o "resto"

Recursividade: gerando permutações

- Implementando:

```
// Escrever todos as permutações do array v[]
void permutations(int v[]) {
   boolean used[] = new boolean[v.length]; // i está na permutação?
   int perm[] = new int[v.length];         // permutação actual
   goPerm(0, v, used, perm); // chamar função recursiva
}
// Gera todas as permutacoes a partir da posição 'cur'
void goPerm(int cur, int v[], boolean used[], int perm[]) {
   if (cur == v.length) {  // Caso base:  terminamos a permutação
      for (int i=0; i<v.length; i++) // Escrever a permutação
         System.out.print(v[perm[i]] + " ");
      System.out.println();
   } else { // Se não terminamos, continuar a gerar
      for (int i=0; i<v.length; i++) // Tentar todos os elementos
         if (!used[i]) {
            used[i] = true; perm[cur] = i;
            goPerm(cur+1, v, used, perm);
            used[i] = false;
         }
   }
}
```