Programação Recursiva
versão 1.02

4 de Maio de 2009

Este guião deve ser entregue, no mooshak e no moodle, até às 23h55 de 4 de Maio.

AVISO: O mooshak é um sistema de avaliação e não deve ser utilizado como ferramenta de verificação do seu estudo individual. Para tal serão fornecidos ficheiros de exemplo que deve ser verificados recorrendo a uma ferramenta do tipo diff. O número de submissões ao mooshak, das soluções deste guião e de guiões subsequentes, podem penalizar a sua nota. As primeiras cinco submissões de cada tarefa não penalizam a cotação da tarefa. No entanto, por cada submissão subsequente serão descontados 10% da quotação.

AVISO: Deve incluir o seu nome e número de aluno no seu código, num comentário com a instrução @author.

Neste guião iremos apresentar uma série de problemas que podem ser resolvidos com técnicas de programação recursiva. Aconselhamos os alunos a lerem este guião, calmamente, do principio ao fim antes de começarem a implementar os métodos descritos. Aconselhamos também a experimentarem os respectivos métodos com papel e lápis, para que estes se tornem mais familiares.

A recursão consiste na construção de objectos, ou procedimentos, recorrendo a objectos semelhantes, mas mais simples, de alguma forma. Atenção que o conceito de objecto é utilizado neste guião no sentido lato e não necessariamente no sentido técnico de programação orientada por objectos.

Os números naturais são um exemplo de uma construção recursiva. Existe um elemento elementar, ou base, o 1. Qualquer outro número natural é da forma $n + 1$ onde n também é um número natural. Este exemplo ilustra bem a flexibilidade das construções recursivas. Note que apenas com estas duas afirmações foi possivel caracterizar um conjunto infinito de números. As duas caracterizações são típicas das construções recursivas. Numa construção recursiva é preciso caracterizar os objectos mais simples de todos, as bases. Também é essencial explicar como os objectos mais complexos podem ser construídos a partir de objectos mais simples, garantindo sempre que esta definição é bem fundada, ou seja que a definição envolve uma sequência finita de objectos e que, portanto qualquer sequência de objectos envolvida na definição de um objecto atinge sempre um elemento base.

No parágrafo anterior tornamos evidente que alguns objectos da nossa realidade quotidiana podem ser descritos de forma recursiva. Pode parecer, à primeira vista, que os objectos recursivos surjem apenas no reino da matemática. Esta percepção deve-se ao facto de ser simples descrever objectos matemáticos desta forma. No entanto as contruções recursivas surgem naturalmente por uso

da razão humana, portanto podem ser encontradas na linguística, na pintura, etc. Os fractais são um exemplo de objectos recursivos bastante sofisticados. Sugerimos que faça uma busca deste tipo de objectos.

O presente guião serve para tomarmos consciência de como o conhecimento destes procedimentos recursivos pode ser usado para alterar a forma como interagimos com determinadas realidades.

1 Combinações

1. Vamos estudar um conjunto de números que é fácil de definir recursivamente, as combinações. Dados dois números naturais n e k, tais que $0 \leq k \leq n$ podemos definir o número $\binom{n}{k}$ da seguinte forma:

$$\binom{n}{k} = \begin{cases} 1 & \text{se } k = n \text{ ou } k = 0 \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{caso contrário} \end{cases}$$

   Note como os elementos base são identificados pelo primeiro caso da definição, ou seja os elementos base são os $\binom{n}{n} = \binom{n}{0} = 1$, para qualquer n. No segundo caso são definidos os elementos mais complexos. Vamos ver um exemplo de como utilizar esta definição.

$$\begin{array}{ll} \binom{4}{2}, & = \binom{3}{1} + \binom{3}{2} \qquad\qquad\qquad = \binom{2}{0} + \binom{2}{1} + \binom{2}{1} + \binom{2}{2} \\ & = 1 + 2 \times (\binom{1}{0} + \binom{1}{1}) + 1 \quad = 2 + 2 \times 2 = 6 \end{array}$$

2. Programe as combinações, de forma recursiva, num método recBinom(int n, int k), na classe Recursive e teste a sua validade com um método testBinom, na classe Main.

3. Teste com vários valores de n e k. Experimente calcular recBinom(-5, 2) ou recBinom(-5, -2). O que é que acontece? Qual é a causa disso? Extenda a função recBinom por forma a que quando não se verifica a condição $0 \leq k \leq n$ o resultado seja 1. Tente manter o seu código o mais simples possivel. É possivel programar este método com apenas um if, um else e dois returns.

4. Experimente calcular recBinom(40, 10). Porque acha que demorou tanto tempo a calcular? Note que para usar a definição recursiva o computador vai ter que recorrer à base tantas vezes quanto o valor de recBinom(40, 10) = 847660528.

   Será que é possivel tornar este processo mais eficiente? O que acontece quando calculamos recBinom(10, 5) ? Para descobrir altere o seu método recBinom inserindo um System.out.printf("C(%d, %d)%n", n, k); na primeira linha do seu método recBinom. Experimente calcular recBinom(10, 5) vai obter uma lista com todas as chamadas recursivas que o método recBinom faz. Repare como, por exemplo, a string $C(5, 1)$ aparece repetidas vezes. Isto significa que o cálculo de recBinom(5, 1) é feito repetidas vezes. É precisamente devido a estes cálculos redundantes que este método demora tanto tempo.

   Porquê calcular duas vezes recBinom(10, 5) se vai devolver o mesmo valor ? O que precisamos é de guardar esse valor da primeira vez que é calculado

e usá-lo quando for preciso, evitando assim ter que o recalcular. Claro que esta decisão implica um compromisso entre o tempo que o programa demora a calcular um resultado e a quantidade de memória que precisa para o fazer, mas neste caso é um compromisso sensato. Programe um novo método itBinom, na classe Recursive, que calcula as combinações de forma iterativa com dois ciclos for. Note que para tal precisa de criar um vector bidimensional do tipo int[][], que vai guardando os valores de $\binom{n'}{k'}$ para $n' \leq n$ e $k' \leq k$. Note que para calcular os valores de $\binom{n'+1}{k'+1}$ apenas precisa dos valores de $\binom{n'}{k'+1}$ e $\binom{n'}{k'}$, por outras palavras para calcular a linha $n'+1$ do triângulo de Pascal basta guardar alguns valores da linha n'. Pode ocupar tanto espaço quanto ache que é necessário para guardar todos os valores intermédios. A forma mais simples de alocar este espaço em java é criar um objecto do tipo int[][], com $(n+1)\times(n+1)$ entradas. Caso esteja preocupado em minimizar o espaço que utiliza e esteja interessado num desafio, observe que para calcular o valor na coluna k apenas precisamos de guardar $k+1$ números. Portanto é possivel utilizar um vector de tamanho $k+2$, ou seja $k+1$ entradas para guardar os valores e uma entrada para um valor temporário. Uma maneira de guardar os valores neste vector é de forma circular, ou seja de uma linha para a outra do triângulo de Pascal a posição de coluna k é alterada uma posição para trás no vector.

Compare, empiricamente, o tempo que este novo método demora a calcular recBinom(10, 5) com o método recursivo. A conclusão a tirar desta experiência é que a recursão é uma técnica de programação poderosa mas que deve ser aplicada com alguns cuidados.

5. Apesar de termos tornado o método mais eficiente, rapidamente vamos encontrar problemas. Experimente calcular itBinom(100, 50). O que aconteceu ? Porque deu um número negativo ? Não é suposto, pois não?

   Quando o valor de $\binom{n}{k}$ é muito grande o método itBinom devolve valores errados. Vamos mudar este método para que devolva double em vez de int, esta nova função deve ser denominada por itBinomDouble. Assim, conseguimos calcular combinações para números maiores, mas também rapidamente atingimos o máximo dos números double.

6. Vamos agora à Tarefa A para o Mooshak, que se destina a validar automaticamente todo o trabalho anterior. Recorde que é necessário submeter um método main dentro de uma directoria poo. O seu programa deve ler da consola uma sequência de números inteiros, dois em cada linha e escrever na consola, para cada um deles, o número de combinações calculado pela função itBinomDouble, escrito em notação científica com quatro casas decimais. Neste exercício todos os números lidos estão entre 1 e 1000, inclusive. É fornecido um exemplo de input/output com este guião, exemplo esse que deve ser validado com um diff local.

2 Ordenação Rápida

2.1 Divisão

7. Neste problema vamos implementar um algoritmo de ordenação conhecido como *QuickSort*. O objectivo de um algoritmo de ordenação é reorganizar uma lista de inteiros por forma a que os números fiquem organizados por ordem crescente. Por exemplo transformar a lista $10, 23, 14, 5, 15$ na lista $5, 10, 14, 15, 23$.

8. Como seria de esperar num guião dedicado à recursão o algoritmo de *QuickSort* funciona de forma recursiva. A estratégia do algoritmo pode ser descrita como "dividir para conquistar". A base da recursão consiste na lista vazia ou na lista que tem apenas um elemento, que já está ordenada. As listas com mais elementos são divididas em duas listas mais pequenas. Cada uma dessas listas é ordenada recursivamente. Depois de ordenadas essas listas são fundidas numa lista maior.

9. Vamos primeiro estudar o processo de divisão no *QuickSort*. O processo de divisão consiste em escolher um elemento da lista, chamado pivô. A lista inicial é dividida na sublista dos elementos que são estritamente menores que o pivô, seguida pelo pivô e pela lista dos elementos que são maiores ou iguais ao pivô.

10. Uma vez percebida a ideia geral da divisão da lista é preciso analisar, ao pormenor, qual a melhor forma de implementar esta divisão. Defina um método int divide(int[] array, int pvt, int a, int b). A semântica do método é a seguinte. O método recebe um array de inteiros denominada por array. A lista de inteiros que queremos ordenar encontra-se entre as posições a e b do array, ou seja array[a] é o primeiro elemento e array[b] é o último elemento. O número pvt é o pivô. Após o método divide executar, a estrutura do array entre a e b consiste nos elementos estritamente menores que o pivô seguidos pelos elementos maiores ou iguais ao pivô. O método deve devolver a posição m, entre a e b onde ocorre o primeiro elemento que é maior ou igual ao pivô. É importante que a ordem relativa dos elementos que são menores que o pivô não seja alterada. A ordem relativa dos elementos maiores ou iguais ao pivô pode ser alterada.

11. O método divide deve ser implementado com apenas 1 ciclo for com dois contadores, 1 comando if e uma variável auxiliar do tipo int. A cada iteração do ciclo for um dos contadores indica quanto elementos do array já foram processados. A estrutura dos elementos do array processados consiste em duas sublistas, uma dos elementos menores e uma dos elementos maiores ou iguais ao pivô. O segundo contador indica a posição onde começa a sublista dos elementos maiores ou iguais ao pivô. Quando o próximo elemento a ser processado é maior ou igual que o pivô, não é preciso realizar nenhuma operação. Quando o elemento a processar é estritamente menor do que o pivô, trocamos este elemento com o primeiro elemento da lista de elementos maiores ou iguais ao pivô e incrementamos o contador da posição onde começa a lista dos elementos maiores ou iguais que o pivô.

12. Vamos testar este método no mooshak, que se destina a avaliar automaticamente todo o trabalho anterior. Esta é a tarefa B. O seu programa deve ler da consola uma sequência de números inteiros separados por "caracteres brancos". O primeiro número indica o valor de a, o segundo número indica o valor de b, o terceiro número indica o valor de pvt, o quarto número indica quandos elementos devem existir no array, seguem-se os elementos que devem estar dentro do array. Após estes números existirá uma mudança de linha, os números da próxima linha são interpretados da mesma forma que os anteriores. O ouput desta tarefa vai consistir em imprimir os elementos do array, separados por um espaço, e com parênteses rectos à volta da posição retornada pelo método divide. Consulte os ficheiros de input/output para esta tarefa.

2.2 Recursão

13. Agora que já temos o processo de divisão das listas vamos passar á implementação recursiva do mesmo. Defina um método qsort(int array[], int a, int b), que ordena o array entre os índices a e b. O método deve começar por dividir a lista, com o método anterior, utilizando como pivô o elemento em array[a]. A seguir deve ordenar recursivamente os elementos estritamente menores do que o pivô e depois os elementos maiores ou iguais que o pivô, excluindo o próprio pivô. Note que para isolar o pivô a forma mais simples é invocar divide(array, array[a], a+1, b), desta forma o pivô fica na posição a, depois é necessário coloca-lo na posição adequada do array. Para tal pode trocar o pivô com o último dos elementos que lhe são estritamente menores. Repare que os elementos que ficam no sub-array da esquerda são todos estritamente menores que o pivô e que os que estão no sub-array da direita são todos maiores ou iguais ao pivô. Ordenando cada sub-array e mantendo o pivô ficamos com todo o array ordenado.

14. Vamos também testar este método no mooshak. Esta é a tarefa C. O seu programa deve ler da consola uma sequência de números inteiros separados por "caracteres brancos". O primeiro número indica quandos elementos devem existir no array, seguem-se os elementos que devem estar dentro do array. Após estes números existirá uma mudança de linha, os números da próxima linha são interpretados da mesma forma que os anteriores. O ouput desta tarefa vai consistir nas posições dos sucessivos pivôs escolhidos pelo método qsort invocado para ordenar o array todo, separados por um espaço. A forma mais simples de obter este ouput é fazer System.out.printf ao resultado do método divide menos 1. Consulte os ficheiros de input/output para esta tarefa.

2.3 Acesso Ordenado

15. Nesta tarefa queremos aceder a um elemento do array na posição pos, como se o array estivesse ordenado. A forma simples de resolver este problema, consiste em ordenar o array e devolver array[pos]. Defina um método int element(int array[], int pos, int a, int b) que devolve o elemento array[pos] do array ordenada.

16. Devido a preocupações com a eficiência o método element não pode ordenar o array, porque isso demora muito tempo. Em alternativa o método element deve utilizar o método divide para dividir o array com array[a]. Note que tambem neste caso é preciso isolar o pivô. Caso a posição do pivô coincida com pos o valor do que estamos à procura é o do pivô. Caso contrário, se a posição do pivô for menor do que pos o valor que estamos à procura encontra-se na lista do elementos maiores ou iguais ao pivô, no caso que falta o valor que estamos à procura esta na lista dos elementos menores que o pivô.

17. Vamos também testar este método no mooshak. Esta é a tarefa D. O seu programa deve ler da consola uma sequência de números inteiros separados pos "caracteres brancos". O primeiro número indica o valor de pos, o segundo número indica quandos elementos devem existir no array, seguem-se os elementos que devem estar dentro do array. Após estes números existirá uma mudança de linha, os números da próxima linha são interpretados da mesma forma que os anteriores. O ouput desta tarefa vai consistir nas posições dos sucessivos pivôs escolhidos pelo método element, separados por um espaço. A forma mais simples de obter este ouput é fazer System.out.printf ao resultado do método divide. Consulte os ficheiros de input/output para esta tarefa.

3 Eficiência

18. Nesta tarefa vamos constatar experimentalmente a vantagem em termos de eficiência temporal de usar a Ordenação Rápida. Primeiro temos de implementar um *InsertionSort*. O InsertionSort é o método que a maior parte dos jogadores de cartas usa para ordenar as suas cartas. Essencialmente consiste em ordenar as cartas da esquerda para direita.

19. O algoritmo *InsertionSort* pode ser implementado com dois ciclos for encadeados. O ciclo exterior itera sobre os elementos do array, da esquerda para a direita. O ciclo interior pega no elemento iterado pelo ciclo exterior o compara-o com todos os elementos anteriores até encontrar o primeiro que é menor ou igual a esse elemento, quando se percorre essa parte inicial da direita para a esquerda. Após econtrar essa posição, insere o elemento e move todos os outros para a direita. Implemente um método isort(int array[]) que ordena os elementos do array. Note que este método deve fazer uso da operação $>$, ou $<$, e que esta operação deve ser usada para comparar elementos do array apenas uma vez por cada iteração do ciclo interior.

20. Como já foi salientado o objectivo desta secção é comprovar a eficiência do método de ordenação *QuickSort*, para tal vamos contar o número de vezes que a operação $>$ ou $<$ é utilizada para comparar dois elementos do array. A ideia desta tarefa é contar o número de vezes que a operação $>$ ou $<$ é utilizada para comparar elementos do array, primeiro para o algoritmo de *InsertionSort* e depois para o algoritmo de *QuickSort*. Note que no caso do *InsertionSort* o valor que procuramos coincide com o total de números que são deslocados para a direita. No caso do *QuickSort* o

número que procuramos é a soma dos valores $b - a$ calculados de cada vez que divide(array, pvt, a + 1, b) é invocado. Esta tarefa não precisa de ser submetida ao mooshak, no entanto nós fornecemos exemplos de input/output, em que o input é como o da tarefa C e o ouput é o número de operações > do *InsertionSort* seguido pelo número de operações > do *QuickSort*.

21. Gere 100 listas de números aleatórios, com tamanhos entre 1000 e 10000 elementos usando a classe java.util.Random, ou seja 10 listas de cada tamanho. Os números aleatórios devem estar entre 0 e 1000. Desenhe um gráfico, com duas curvas, por exemplo em excel, com o número de operações > dos algoritmos *InsertionSort* e *QuickSort*. Submeta os seus gráficos, em jpeg, no moodle e inclua um comentário quanto á eficiência relativa destes dois algoritmos. Nota: o jpeg tem que incluir obrigatóriamente o seu nome e número de aluno.