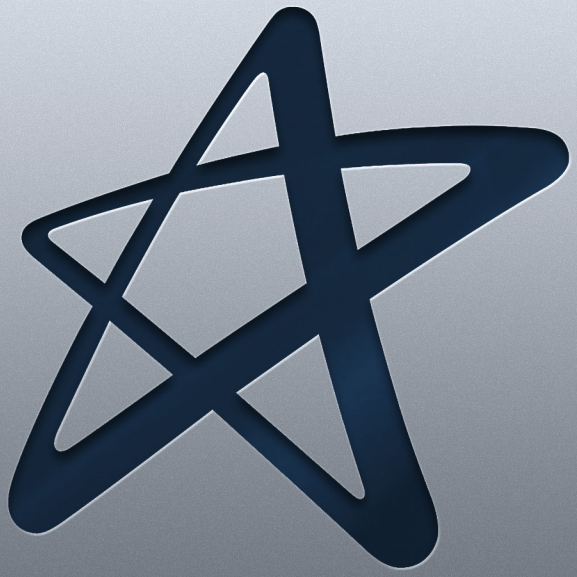
Computabilidade e Complexidade de Algoritmos



Cruzeiro do Sul Virtual
Educação a distância

Material Teórico

Técnicas de Projeto de Algoritmos

Responsável pelo Conteúdo:
Prof. Dr. Luciano Rossi

Revisão Textual:
Prof.ª Esp. Kelciane da Rocha Campos

UNIDADE Técnicas de Projeto de Algoritmos

- Refinamento;
- Iteração;
- Recursão;
- Divisão e Conquista;
- Programação Dinâmica;
- Programação Gulosa.

OBJETIVO DE APRENDIZADO

- Apresentar ao aluno as técnicas para o projeto de algoritmos, de modo a capacitá-lo para a utilização da melhor técnica associada aos contextos do problema, buscando a eficiência computacional.

Orientações de estudo

Para que o conteúdo desta Disciplina seja bem aproveitado e haja maior aplicabilidade na sua formação acadêmica e atuação profissional, siga algumas recomendações básicas:

Assim:

- ✓ Organize seus estudos de maneira que passem a fazer parte da sua rotina. Por exemplo, você poderá determinar um dia e horário fixos como seu "momento do estudo";
- ✓ Procure se alimentar e se hidratar quando for estudar; lembre-se de que uma alimentação saudável pode proporcionar melhor aproveitamento do estudo;
- ✓ No material de cada Unidade, há leituras indicadas e, entre elas, artigos científicos, livros, vídeos e *sites* para aprofundar os conhecimentos adquiridos ao longo da Unidade. Além disso, você também encontrará sugestões de conteúdo extra no item **Material Complementar**, que ampliarão sua interpretação e auxiliarão no pleno entendimento dos temas abordados;
- ✓ Após o contato com o conteúdo proposto, participe dos debates mediados em fóruns de discussão, pois irão auxiliar a verificar o quanto você absorveu de conhecimento, além de propiciar o contato com seus colegas e tutores, o que se apresenta como rico espaço de troca de ideias e de aprendizagem.

Refinamento

Seja bem-vindo(a) à unidade de Técnicas de Projeto de Algoritmos. Nas unidades anteriores, estudamos os princípios da análise de computabilidade e complexidade dos algoritmos, onde tivemos a oportunidade de identificar a importância do projeto de algoritmos eficientes.

Complementarmente, o estudo das técnicas de análise de algoritmos é importante para que possamos projetar algoritmos que sejam eficientes, tanto do ponto de vista do tempo de execução quanto do consumo de memória.

A necessidade de se projetar algo, como é o nosso caso aqui, pois pretendemos projetar algoritmos, é facilitada por meio da aplicação de alguma técnica que nos guie em nossa tarefa. Existem diferentes técnicas que são úteis para o projeto de algoritmos e, dependendo do objetivo que se busca, a aplicação da técnica mais adequada pode impactar no resultado final.

A primeira técnica que iremos estudar é chamada de refinamento. Nesse sentido, sabemos que um algoritmo é um conjunto de instruções que, quando executadas em uma ordem específica, produz o resultado pretendido. Um algoritmo é considerado completo quando as instruções que o compõem são perfeitamente compreendidas pelo destinatário – no caso, o computador.

O refinamento é caracterizado pelo desdobramento de instruções que não são compreendidas pelo destinatário em instruções mais específicas, que possam ser mais facilmente realizadas pelo destinatário.

Podemos observar vários exemplos do uso da técnica de refinamento de algoritmos. O cientista inglês Charles Babbage (1791-1871) é considerado um dos primeiros idealizadores do conceito de uma máquina programável de uso geral, que pudesse ser utilizada para diferentes aplicações. Sua invenção foi denominada de máquina analítica, a qual é considerada a precursora dos computadores eletrônicos. Apesar de nunca ter sido construída, a máquina analítica de Babbage é considerada uma das suas invenções mais bem sucedidas. A máquina analítica foi concebida por Babbage com base no desenvolvimento de um algoritmo que computava os valores de funções matemáticas. Esse algoritmo foi escrito por Augusta Ada Byron King (1815-1852), mais conhecida como Ada Lovelace, considerada a primeira programadora da história.

Antes do advento de máquinas que pudessem realizar cálculos matemáticos complexos, como a máquina analítica de Babbage, era comum que os matemáticos desdobrassem um processo de cálculo complexo em tarefas menores, que pudessem ser realizadas por pessoas sem grandes conhecimentos de matemática. Assim, partes do processo podiam ser executadas paralelamente, por várias pessoas com qualificações mínimas, cujos resultados eram combinados posteriormente, resultando em um ganho substancial de tempo.

O exemplo anterior ilustra o conceito de refinamento de algoritmo em um momento histórico no qual não havia, sequer, a proposição de uma definição e muito menos da aplicabilidade para esse tipo de solução. Atualmente, considerar a técnica de refinamento é uma forma de facilitar o projeto de algoritmos e, também, de proporcionar ao projetista alguns parâmetros que o ajudarão a obter algoritmos mais claros e eficientes.

Vamos considerar alguns exemplos de aplicação da técnica de refinamento de algoritmos. Considere que seja necessário projetar um algoritmo que calcule a área de um triângulo. Nesse sentido, um algoritmo que aborde esse problema, de forma ampla, poderia ter as seguintes instruções:

Algoritmo 1 – Cálculo da área de um triângulo sem refinamento

CalculaTriangulo( )

1. ***ler*** *os parâmetros*
2. ***calcular*** *a área*
3. ***retornar*** *a área*

A função *CalculaTriangulo*( ), descrita no Algoritmo 1, apresenta os passos para o cálculo da área do triângulo; porém, há alguns problemas nessa representação. Antes de discutirmos os problemas, a forma utilizada para a representação dos algoritmos nesta unidade seguirá de acordo com o formato utilizado nas unidades anteriores, apenas para a manutenção da coerência entre as representações.

Considere que uma pessoa vá executar os passos descritos pela função *CalculaTriangulo*( ); note que um primeiro questionamento possível de ser feito por essa pessoa é: quais são os parâmetros necessários para o cálculo da área do triângulo? Ou ainda: como se calcula a área de um triângulo? Essas questões são pertinentes, pois não há qualquer detalhe sobre como essas instruções devem ser realizadas e para o caso de a pessoa não ter ideia de como se procede ao cálculo da área de um triângulo, a sua execução torna-se impossibilitada.

Um algoritmo deve conter uma descrição sobre a resolução de uma determinada tarefa de forma inequívoca. Deixar lacunas que devam ser preenchidas pelo executor não é uma possibilidade no projeto de algoritmos. Obviamente, o nível de detalhe do algoritmo vai depender do destinatário; no caso de o computador ser o destinatário, há capacidades inerentes que podem ser consideradas, delimitando, assim, o nível de detalhamento do algoritmo.

Sabemos que para o cálculo da área de um triângulo é preciso saber quais são os valores da base e da altura do triângulo. Além disso, a forma de se calcular essa área é por meio da multiplicação da base pela altura, cujo resultado é dividido por dois. Nesse contexto, uma nova versão da função *CalculaTriangulo*( ), com um maior grau de refinamento, é apresentada no Algoritmo 2.

Algoritmo 2 – Cálculo da área de um triângulo com um maior grau de refinamento, comparado ao Algoritmo 1

CalculaTriangulo( )

1. $b \leftarrow$ ***ler*** *o valor da base*
2. $a \leftarrow$ ***ler*** *o valor da altura*
3. ***fazer*** $r \leftarrow b \times a \div 2$
3. ***retornar*** r

O exemplo descrito no Algoritmo 2 apresenta um maior nível de detalhes, quando comparado ao exemplo anterior. Veja que, nesse novo exemplo, estamos supondo algumas coisas sobre o destinatário do algoritmo. Consideramos que o destinatário sabe ler um valor fornecido pelo usuário, sabe realizar as operações aritméticas e sabe retornar um valor ao usuário. Esse exemplo ilustra a afirmação anterior, a qual descreve que o limite a ser considerado para o refinamento de algoritmos depende das características do destinatário. Como o nosso contexto é o projeto de algoritmos para computadores, devemos considerar as operações que ele é capaz de realizar e, também, descrever a estrutura de dados que será utilizada.

Iteração

Os algoritmos devem ter um nível específico de detalhes, como vimos na descrição da técnica de refinamento; além disso, outra característica sempre presente nos algoritmos é a iteratividade. Aqui é importante destacar as diferenças entre os termos iterativo e interativo. O primeiro faz referência ao que serve para iterar ou repetir. Por outro lado, interativo é aquele que interage, que se comunica em ambos os sentidos.

Podemos ter algoritmos interativos, nos quais há uma comunicação entre o usuário e o próprio algoritmo. Nesse contexto, podemos inserir nos algoritmos parâmetros que são demandados por ele em seu processo de execução. É comum a existência de programas de computador que solicitam que o usuário informe determinados dados para a continuidade da execução. Há diferentes níveis de interação entre os seres humanos (usuário) e o computador (programa/algoritmo), porém esse é outro assunto e, apesar da importância, não faz parte do escopo dessa disciplina.

A iteração é uma característica comum nos algoritmos e, como vimos anteriormente, é importante para os seus respectivos tempos de execução. Uma grande vantagem do computador, em relação ao ser humano, é sua capacidade de executar muitas tarefas rapidamente. Nesse contexto, muitas tarefas são iguais e devem ser executadas muitas vezes. Por exemplo, suponha que precisemos cadastrar um milhão de nomes e telefones de pessoas em um arquivo. Veja que a leitura e escrita desses dados é uma tarefa simples, porém a quantidade de vezes que deverá ser executada é grande. Assim, um algoritmo descreve as tarefas em poucas linhas de instruções, as quais serão repetidas quantas vezes forem necessárias.

Vamos considerar um exemplo prático de algoritmo iterativo, a sequência de Fibonacci. O matemático italiano Leonardo de Pisa, conhecido também como Fibonacci, utilizou essa sequência para descrever, de maneira teórica, o crescimento populacional dos coelhos. Comumente, a sequência de Fibonacci tem como os dois primeiros elementos da série os valores 0 e 1. A partir do terceiro elemento, a obtenção dos elementos posteriores é possível pela soma dos dois elementos anteriores.

A sequência de Fibonacci tem aplicações em diversos contextos, como o mercado financeiro, na descrição de padrões biológicos e na computação. Considerar a utilização de algoritmos para a obtenção de determinados elementos da sequência de Fibonacci

é especialmente interessante, pois essa tarefa pode ser feita utilizando-se diferentes abordagens, como a iterativa, que veremos aqui, a recursiva e a da divisão e conquista, que veremos mais à frente ainda nesta unidade.

Considere o Algoritmo 3, que descreve as instruções que compõem a função iterativa para o cálculo do *n*-ésimo termo da sequência de Fibonacci. Os dois primeiros elementos dessa série são os valores 0 e 1, há algumas fontes que descrevem o primeiro elemento como sendo o valor 1, porém isso não impacta o funcionamento do nosso algoritmo.

Algoritmo 3 – Função iterativa que retorna o *n*-ésimo termo da sequência de Fibonacci

Fibonacci(*n*)
1. $i \leftarrow 0$
2. $j \leftarrow 1$
3. ***para*** $k \leftarrow 1$ ***até*** n
4. $\quad w \leftarrow i+j$
5. $\quad i \leftarrow j$
6. $\quad j \leftarrow w$
7. ***retornar*** i

O parâmetro de entrada *n* é a posição para a qual o valor na série deve ser retornado. Veja que as variáveis *i* e *j* assumem, inicialmente, os dois primeiros valores da série. O laço da linha 3 é responsável pela iteração do algoritmo, de modo que as variáveis são atualizadas até que se chegue à posição objetivo.

Veja que se $n = 1$, o laço não é executado e a função retorna 0, que é o primeiro valor da série. Se $n = 2$, o laço executa uma iteração e a função retornará 1, que é o segundo valor da série. A partir daí, a cada iteração do laço, é calculado o próximo valor, como sendo a soma dos dois valores anteriores, e o conteúdo das variáveis é atualizado. Ao final das iterações, definidas pelo valor de *n*, a função retorna o valor da *n*-ésima posição na série.

Trocando Ideias...

Faça a análise do tempo de execução do algoritmo iterativo para a sequência de Fibonacci, em função do tamanho da entrada, e apresente o resultado utilizando a notação assintótica (*Big Oh*).

Há uma versão recursiva para o algoritmo de Fibonacci iterativo. Aliás, para todo algoritmo recursivo há uma versão iterativa. Os algoritmos recursivos são, comumente, mais fáceis de serem interpretados e, por vezes, podem ser não recomendáveis para determinadas instâncias de problemas. O próximo tópico sobre técnicas de projetos de algoritmos de que trataremos é a recursão. A recursividade está presente em diversas aplicações algorítmicas, mas deve ser considerada com uma dose de moderação; assim, vamos explorar as características dessa técnica na próxima seção.

Recursão

A principal característica de problemas recursivos é a existência de um subproblema menor, que pode ser resolvido mais facilmente. Em outras palavras, a recursividade é a técnica que utiliza a redução de uma determinada instância de um problema em instâncias menores, do mesmo problema, de modo que se possa resolvê-lo diretamente.

As funções (ou procedimentos) recursivas são aquelas que chamam a si mesmas, enviando como parâmetro uma instância reduzida do problema original. Há, ainda, a possibilidade de duas funções que são acionadas uma pela outra, de modo a enviar uma instância, sempre menor, do problema original. A redução do tamanho da instância do problema é feita até que se chegue a um caso base. Assim, um caso base é aquele que pode ser resolvido diretamente.

Vamos considerar um exemplo prático de cálculo recursivo. O cálculo do fatorial de um número é um exemplo clássico de recursividade. A título de exemplificação, o fatorial de um número é:

$$n! = n \times (n - 1) \times \ldots \times 2 \times 1$$

Por exemplo, o valor de cinco fatorial é calculado da seguinte forma: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Nesse contexto, temos o caso base $0! = 1$; veja que quando a instância é igual a zero, podemos resolver de forma direta, pois sabemos que o resultado dessa instância é igual a um. Sabemos, também, que $n! = n(n - 1)!$, assim, a formalização do cálculo do fatorial, considerando o caso base e os demais casos, pode ser descrita da seguinte forma:

$$n! = \begin{cases} 1 & se\ n = 0 \\ n(n-1)! & se\ n > 0 \end{cases}$$

O Algoritmo 4 descreve uma função recursiva para o cálculo do fatorial de um número. Veja que uma característica importante em algoritmos recursivos é a simplicidade, o que torna sua interpretação mais facilitada. Atente para a linha 4 do algoritmo, na qual a instância é reduzida e passada como parâmetro na chamada recursiva à própria função *Fatorial*().

Algoritmo 4 – Função recursiva para o cálculo do fatorial de um número

```
Fatorial(n)
1. se n = 0
2.    retornar 1
3. senão
4.    retornar n × Fatorial(n – 1)
```

Os algoritmos recursivos podem consumir uma grande quantidade de memória, para os casos nos quais existam muitas chamadas recursivas. Cada chamada é empilhada e

fica aguardando até que o caso base seja atingido. Assim, podemos ter várias chamadas empilhadas na memória, o que pode ser um problema a depender da quantidade de memória disponível.

Podemos construir uma nova versão para o algoritmo de Fibonacci, considerando a abordagem recursiva. O Algoritmo 5 apresenta a versão recursiva para o algoritmo de Fibonacci. Veja que, quando comparado à sua versão iterativa, a função recursiva é mais simples e clara. As características descritas para essa abordagem são observáveis na linha 4, onde é possível verificar tanto a redução no tamanho da instância, quanto a chamada recursiva. Note que, nesse caso, temos duas chamadas recursivas que terão seus resultados somados para a obtenção do valor posterior na sequência.

Algoritmo 5 – Função recursiva que retorna o *n*-ésimo termo da sequência de Fibonacci

Fibonacci(*n*)
1. ***se*** $n \leq 2$
2. ***retornar*** $n - 1$
3. ***senão***
4. ***retornar*** *Fibonacci*$(n - 1)$ + *Fibonacci*$(n - 2)$

A lógica funcional do algoritmo é simples de ser verificada; se, por exemplo, buscamos pelo primeiro elemento da série, ou seja, $n = 1$, o algoritmo retornará o valor 0. De forma similar, se desejamos o segundo elemento, $n = 2$, será retornado o valor 1. Esses dois exemplos constituem os casos base. Veja que para qualquer valor maior que 2, o que corresponde ao terceiro elemento em diante, o algoritmo retorna a soma das chamadas recursivas, as quais têm como parâmetros os dois elementos anteriores.

O exemplo recursivo para o algoritmo de Fibonacci é mais ineficiente que o seu correspondente iterativo. Note que podem ocorrer diversas chamadas recursivas independentes para um mesmo valor. A Figura 1 ilustra as chamadas recursivas que são realizadas a partir da passagem de $n = 5$ como parâmetro.

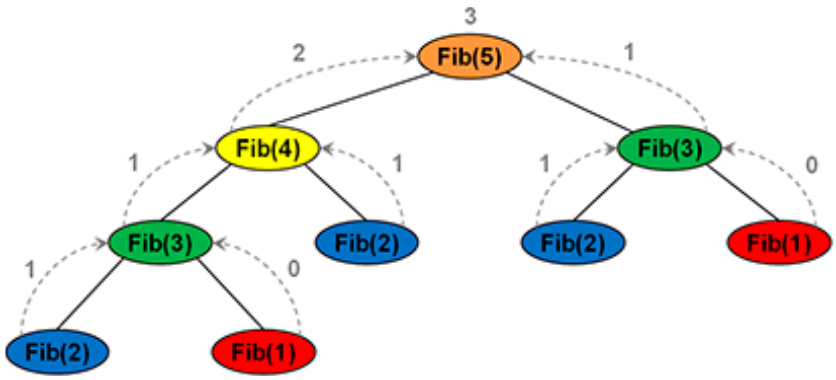


Figura 1 – Representação das instâncias criadas para o cálculo do quinto elemento da sequência de Fibonacci, cores iguais referem-se à mesma instância

Analisando-se a representação na Figura 1, é possível notar que existem chamadas independentes para um mesmo parâmetro de entrada. Por exemplo, veja que as chamadas

cujos parâmetros são 1, 2 e 3 ocorreram duas vezes, três vezes e duas vezes, respectivamente. Não é difícil notar que a realização do cálculo de um mesmo parâmetro, diversas vezes, consome um tempo adicional que impacta no desempenho do algoritmo. Além disso, há uma necessidade maior de memória, visto que as chamadas em espera devem ficar armazenadas até que o caso base ocorra e elas possam ser finalizadas.

Considerar a abordagem recursiva não é uma boa ideia para todos os problemas. Devido às características descritas anteriormente, a recursão deve ser aplicada em casos particulares, como, por exemplo, quando a versão iterativa do algoritmo for muito complexa ou quando a solução requeira o uso explícito de uma pilha. Além disso, casos que utilizam a abordagem, ou técnica, da divisão e conquista, são ótimos candidatos ao uso de recursão, como veremos a seguir.

"Ao tentar resolver o problema, encontrei obstáculos dentro de obstáculos. Por isso, adotei uma solução recursiva.".

Divisão e Conquista

A abordagem da divisão e conquista não é inteiramente nova para o nosso contexto. Quando estudamos o algoritmo de ordenação *Merge Sort*, vimos que ele divide o vetor de elementos em duas partes, sucessivas vezes, e após chegar a um vetor com um único elemento, o algoritmo combina esses vetores de modo que os elementos estejam ordenados ao final do processo.

Existem três etapas envolvidas na abordagem da divisão e conquista. Primeiro, dividir o problema em instâncias menores, conquistar as instâncias, resolvendo-as de forma recursiva ou diretamente, se forem pequenas o suficiente, e combinar as soluções das instâncias menores para obter a solução do problema original.

Quando comparamos o algoritmo recursivo de Fibonacci e o algoritmo de ordenação *Merge Sort* é possível observar diferenças fundamentais, apesar de ambos utilizarem recursão. Cada chamada recursiva, no *Merge Sort*, resolve metade do problema, ou seja, são instâncias diferentes do mesmo problema. Isso não ocorre com o Fibonacci, que computa as mesmas instâncias diversas vezes, podendo gerar ineficiência. Além disso, a redução de problemas pela abordagem da divisão e conquista não ocorre de forma trivial, como é o caso do cálculo do fatorial.

A Figura 2 ilustra as etapas realizadas pelo algoritmo *Merge Sort* para a ordenação de um vetor. A instância original é dividida em duas instâncias menores; esse procedimento é repetido até que reste somente um elemento no vetor, dessa forma o vetor estará ordenado. Em seguida, os vetores são combinados considerando-se a ordem crescente para seleção dos elementos. Ao final, o vetor original apresenta seus elementos ordenados.

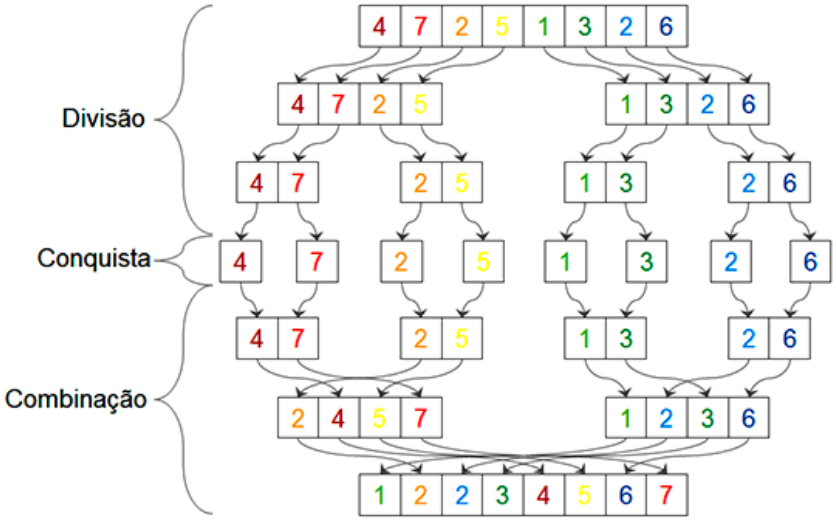


Figura 2 – Representação da abordagem de divisão e conquista aplicada pelo algoritmo de ordenação *Merge Sort*

A identificação das etapas de divisão, conquista e combinação, na Figura 2, é somente ilustrativa. Veja que essas etapas podem não apresentar uma definição específica de seus limites, quando consideramos a definição formal da abordagem de divisão e conquista.

Os algoritmos que têm chamadas recursivas a si próprios, comumente, podem ter seus tempos de execução descritos por uma equação de recorrência. Considerando, especificamente, a ordenação por intercalação, veja que a solução direta para os vetores com um único elemento é feita em tempo constante. A etapa de divisão resulta em subvetores com 1/2 do tamanho do vetor original e, assim, se o tempo de execução para o problema original é $T(n)$, então para cada subvetor o tempo de execução será $T(n/2)$. Nesse sentido, a equação de recorrência para a ordenação por intercalação é:

$$T(n) = \begin{cases} c, & se\ n = 1 \\ 2T\left(\frac{n}{2}\right) + cn, & se\ n > 1 \end{cases}$$

Consulte a unidade anterior para ver a descrição completa da análise do tempo de execução do algoritmo de ordenação por intercalação (*Merge Sort*).

Programação Dinâmica

De forma similar à abordagem de divisão e conquista, a programação dinâmica também resolve problemas combinando as soluções de instâncias menores do mesmo problema. Outra referência que podemos utilizar é o problema apresentado pelo algoritmo recursivo de Fibonacci, o qual, como vimos, resolve várias vezes a mesma instância do problema. Nesse contexto, a programação dinâmica é utilizada para solucionar o caso da resolução repetida de instâncias menores do problema.

Os algoritmos que consideram a programação dinâmica resolvem cada instância reduzida do mesmo problema uma única vez, armazenando o valor obtido para consultas

posteriores. Isso evita o reprocessamento de problemas, melhorando significativamente o desempenho desses tipos de algoritmos. Comumente, essa abordagem é utilizada em problemas de otimização. Nesses casos, podem existir muitas soluções para o mesmo problema e estamos interessados em encontrar a melhor solução, ou solução ótima.

A versão recursiva do algoritmo de Fibonacci calcula diversas vezes uma mesma instância, conforme vimos anteriormente. Por exemplo, se quisermos encontrar o valor do décimo elemento na sequência de Fibonacci, o algoritmo recursivo chamará *Fibonacci*(3) 21 vezes, sem considerar as demais chamadas. Esse reprocessamento característico da função Fibonacci recursiva torna o processo muito ineficiente. A utilização de programação dinâmica para esse problema consiste em memorizar os resultados já processados e, quando uma chamada for feita para uma instância já processada, retornar o resultado memorizado.

O algoritmo para calcular o *n*-ésimo termo da sequência de Fibonacci, considerando-se a abordagem da programação dinâmica, utiliza uma estrutura de dados auxiliar, nesse caso um vetor. O Algoritmo 6 descreve as características desse vetor. Note que o tamanho do vetor deve ser igual à posição que se pretende calcular, visto que precisamos de todos os valores precedentes para o cálculo do valor objetivo. Assim como foi feito anteriormente, armazenamos os casos base nas primeiras posições do vetor, ou seja, posições 0 e 1 (linhas 2 e 3) e, para todas as demais posições, armazenamos o valor $-\infty$.

Uma observação importante é que para não haver desperdício de espaço no vetor, a primeira posição é a posição 0. Assim, quando nos referirmos ao primeiro elemento na sequência, esse elemento estará na posição 0. Se quisermos o segundo elemento, pegaremos o valor da posição 1, e assim sucessivamente.

As posições que estão ocupadas com o valor são aquelas que ainda não tiveram seus valores calculados. Assim, inicialmente o vetor conta apenas com os casos base e, durante a execução do algoritmo, os outros valores vão sendo calculados e armazenados nas respectivas posições.

Algoritmo 6 – Função para criar um vetor que será utilizado pela função *FibonacciDynamicProgramming*()

FibonacciPreparing(*n*)
1. ***fazer*** $F[0 \ldots n-1]$
2. $F[0] \leftarrow 0$
3. $F[1] \leftarrow 1$
4. ***para*** $i \leftarrow 2$ ***até*** $n-1$
5. $F[i] \leftarrow -\infty$
6. ***retornar*** F

A função *FibonacciDynamicProgramming*(), descrita no Algoritmo 7, recebe, como parâmetros de entrada, a posição *n* que queremos calcular na sequência de Fibonacci e o vetor auxiliar *F*, este já previamente concebido. Inicialmente, a função verifica se o valor que queremos existe no vetor, ou seja, se ele já foi calculado previamente ou se ele é um dos casos base (linha 1). Caso essa verificação se confirme, a função retornará o valor correspondente à posição *n*, lembrando que a posição *n* na sequência de Fibonacci é a posição $n-1$ no vetor auxiliar.

Caso o valor objetivo ainda não tenha sido calculado, são realizadas duas chamadas recursivas considerando as posições imediatamente anteriores à posição objetivo, da mesma forma que foi feita para a versão recursiva do algoritmo. Os retornos dessas chamadas serão somados e o resultado será armazenado na posição correspondente, para futuras consultas, e retornada pela função.

Algoritmo 7 – Função recursiva que retorna o *n*-ésimo termo da sequência de Fibonacci com memorização de instâncias já processadas (programação dinâmica)

FibonacciDynamicProgramming(*n*,*F*)

1. ***se*** $F[n-1] \geq 0$
2. ***retornar*** $F[n-1]$
3. ***senão***
4. $F[n-1] \leftarrow$ *Fibonacci*$(n-1, F)$ + *Fibonacci*$(n-2,F)$
5. ***retornar*** $F[n-1]$

Outro exemplo de aplicação de programação dinâmica é o problema da mochila. Esse é um problema de otimização que considera uma coleção de objetos para os quais existem um peso e um valor correspondentes. Há, ainda, uma mochila com uma determinada capacidade. O objetivo é coletar objetos de modo que a soma de seus valores seja a maior possível, respeitando-se a capacidade da mochila.

Existem diferentes versões para o problema da mochila, aqui vamos considerar o problema da mochila booleana (ou binária). Nessa versão, temos a opção de coletar ou não os objetos, sem repetição dos mesmos.

Uma solução para o problema da mochila booleana é obtida a partir de recursão. Por exemplo, considere que temos uma mochila com capacidade $p = 20$ e uma coleção de objetos $X = \{x_1, x_2, \ldots, x_n\}$, suponha que tenhamos selecionado um objeto x_i cujo peso é $p_i = 6$. Nesse sentido, podemos considerar que com a escolha de x_i o problema se reduziu a uma mochila com capacidade $p = 14$ (pois já temos um objeto na mochila; portanto, a capacidade atual é $p = 20 - 6 = 14$) e a escolha de um objeto em um conjunto de $n - 1$ objetos (veja que se tínhamos n objetos, com a primeira escolha temos agora $n - 1$ objetos). Note que houve uma redução no tamanho da instância original e o problema mantém suas características, ou seja, a solução será a mesma para a instância reduzida, daí a recursão.

No caso de o objeto selecionado estar na solução ótima, ou seja, a soma dos valores dos objetos selecionados é a maior possível, então a solução será a mochila com o objeto x_i combinado com as demais escolhas dos $n - 1$ objetos, porém a capacidade da mochila agora é igual a 14. Por outro lado, se o objeto selecionado x_1 não contribuir para uma solução ótima, a solução será a mochila com sua capacidade original com $n - 1$ objetos selecionados. Devemos comparar os dois casos descritos e escolher aquele que produz o maior valor. Veja o Algoritmo 8, que descreve essa função recursiva para o problema da mochila booleana.

Algoritmo 8 – Versão recursiva para a resolução do problema da mochila booleana

Mochila(*v*,*p*,*w*,*n*)

1. ***se*** $n=0$
2. ***retornar*** 0
3. ***para*** $i \leftarrow 0$ ***até*** $n-1$
4. ***se*** $p[n-1-i] > w$
5. ***retornar*** *Mochila*(v, p, w, $n-1$)
6. ***senão***
7. ***retornar*** *MAX*(*Mochila*(v, p, $w - p[n-1-i]$, $n-1$) $+ v[n-1-i]$, *Mochila*(v, p, w, $n-1$))

A função *Mochila*() retorna o valor máximo para a escolha dos objetos que estarão na mochila, respeitando sua capacidade. Os parâmetros de entrada da função são um vetor $v = [v_0, v_1, \ldots, v_{(n-1)}]$ com os valores v_i dos objetos, um vetor $p = [p_0, p_1, \ldots, p_{n-1}]$ com os pesos p_i dos objetos, a capacidade w da mochila e a quantidade n de objetos. Inicialmente, o algoritmo verifica se há elementos a serem selecionados (linha 1) e, caso não existam objetos, o algoritmo retorna zero.

Para cada objeto disponível, se o respectivo peso for maior que a capacidade da mochila, o algoritmo reduzirá o tamanho da instância e chamará, recursivamente, a si mesmo passando como parâmetro a instância reduzida (linhas 4 e 5). Caso contrário, o algoritmo retorna o maior valor, considerando a seleção do respectivo objeto ou não (linha 7).

A função *Mochila*() chama a si mesma, recursivamente, repetidas vezes com os mesmos valores de parâmetros, o mesmo comportamento que observamos para a versão recursiva do algoritmo de Fibonacci (Algoritmo 5). Assim, essa primeira versão do algoritmo *Mochila*() é muito ineficiente para instâncias grandes do problema.

A utilização de uma abordagem baseada em programação dinâmica, na qual os resultados das instâncias é memorizado para consultas posteriores considerando instâncias iguais, elimina o reprocessamento observado na versão anterior da função *Mochila*(). O Algoritmo 9 descreve os passos para a implementação da função *MochilaDinamica*().

Algoritmo 9 – Versão em programação dinâmica para a resolução do problema da mochila booleana

MochilaDinamica(*v*,*p*,*w*,*n*)

1. ***fazer*** $z[n+1][w+1]$
2. ***para*** $d \leftarrow 0$ ***até*** w
3. $z[0][d] \leftarrow 0$
4. ***para*** $k \leftarrow 1$ ***até*** n
5. $z[k][0] \leftarrow 0$
6. ***para*** $k \leftarrow 1$ ***até*** n

```
7.    para d ← 1 até w
8.      z[k][d] ← z[k − 1][d]
9.      se p[k − 1] ≤ d e v[k − 1] + z[k − 1][d − p[k − 1]] ≥ z[k][d]
10.       z[k][d] ← v[k − 1] + z[k − 1][d − p[k − 1]]
11.   retornar z[n][w]
```

Veja que, inicialmente, é criada uma matriz com $n + 1$ linhas e $w + 1$ colunas, essa matriz será utilizada para a memorização dos resultados de instâncias intermediárias (linha 1). As primeiras linha e coluna são preenchidas com zero (linhas 2-5). Há nas linhas 6 e 7 dois laços aninhados, o laço externo (k) gerará os coeficientes que representam os objetos e o laço interno (w) gerará os coeficientes para a capacidade da mochila, de 1 até o seu limite.

O teste condicional, na linha 9, segue de maneira similar à versão anterior. Primeiro, verifica-se se o peso do objeto em questão não excede a capacidade da mochila e, depois, decide-se sobre a escolha, ou não, desse objeto. Ao final do processo, a matriz z estará completamente preenchida e, em sua última célula, o valor máximo da soma dos valores dos objetos selecionados estará armazenado, o qual será retornado pela função.

Considerando, a título de exemplo, uma instância na qual $v = [10, 7, 25, 24]$, $p = [2, 1, 6, 5]$, $n = 4$ e $w = 7$, o preenchimento da matriz z é apresentado na Tabela 1, onde as linhas e colunas referem-se ao número de objetos disponíveis e à capacidade da mochila, respectivamente.

Tabela 1 – Preenchimento da matriz após a finalização da execução da função *MochilaDinamica*(), considerando, como instância de exemplo, $v = [10, 7, 25, 24]$, $p = [2, 1, 6, 5]$, $n = 4$ e $w = 7$

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	10	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

A melhor forma de compreender o funcionamento de um algoritmo é realizando sua simulação. Considere o algoritmo *MochilaDinamica*() e a mesma instância do exemplo anterior. Considere, também, $w = 8$ e faça a simulação, preenchendo a matriz , de acordo com as instruções do algoritmo.

Programação Gulosa

Na seção anterior tratamos do problema da mochila. Vimos que esse tipo de problema pertence à classe dos problemas de otimização, ou seja, queremos encontrar o resultado ótimo. A solução para o problema da mochila booleana foi desenvolvida

considerando-se duas estratégias, uma envolvendo a programação dinâmica e outra não. Ambas as estratégias resultam em um resultado ótimo para o problema.

Há outra técnica de projeto de algoritmo cujo objetivo é encontrar uma solução, mesmo que não seja ótima. Essa abordagem é denominada de estratégia gulosa para a solução de problemas. Um algoritmo guloso, como o próprio nome sugere, seleciona, a cada iteração, o objeto que lhe parece mais "saboroso" naquele momento, sem qualquer tipo de avaliação global sobre a escolha feita. Assim, um algoritmo guloso preocupa-se com a melhor escolha em um momento específico, sem considerar as implicações posteriores. Nessa abordagem, depois da escolha feita não se volta atrás e, por conta disso, geralmente os algoritmos gulosos são rápidos.

Vamos retornar ao problema da mochila booleana para compreender as características de uma solução gulosa. Lembrando que temos uma coleção de objetos e queremos colocá-los em uma mochila, com capacidade limitada, de modo que tenhamos o maior valor possível. Cada objeto possui um peso e um valor; assim, podemos determinar um valor de utilidade para os objetos. Por exemplo, se dividirmos o valor pelo peso, teríamos o valor de cada objeto por unidade de peso. Dessa forma, a estratégia é escolher os objetos de acordo com os respectivos valores de utilidade, ordenados do maior para o menor, até o limite da capacidade da mochila.

Para ilustrar a estratégia descrita anteriormente, vamos considerar a mesma instância que utilizamos como exemplo na seção anterior, ou seja, temos quatro objetos ($n = 4$), os seus respectivos valores são descritos pelo vetor $v = [10, 7, 25, 24]$, os pesos correspondentes são dados pelo vetor $p = [2, 1, 6, 5]$ e a capacidade da mochila é $w = 7$.

Agora, calculamos os respectivos valores de utilidade da seguinte forma: v_i/p_i, assim poderíamos armazenar os valores de utilidade, também, em um vetor, cujos valores calculados seriam $u = [5; 7; 4, 17; 4, 8]$. A seleção dos objeto é feita por meio do valor de utilidade, do maior para o menor. Assim, o primeiro objeto colocado na mochila é aquele cujo valor de utilidade é igual a 7 (u_1) e, assim, temos um valor acumulado na mochila igual a 7 (v_1) e a capacidade atualizada da mochila é $w = 7 - 1 = 6$.

O próximo objeto a ser inserido na mochila é aquele cujo valor de utilidade é igual a 5 (u_0); com a seleção desse objeto, o valor acumulado na mochila é igual a 17 ($v_1 + v_0$) e a capacidade atualizada da mochila é igual a $w = 6 - 2 = 4$. Seguindo a mesma lógica, o próximo objeto a ser selecionado é aquele cujo valor de utilidade é igual a 4,8 (u_3); veja que esse objeto tem peso igual a 5 (p_3), o que supera a capacidade atual da mochila, que é 4. O último objeto restante ($u_2 = 4, 17$) tem peso igual a 6 (p_2), que também supera a capacidade atual da mochila ($w = 4$). Assim, não há mais objetos possíveis de serem selecionados, considerando-se a restrição de peso da mochila.

O resultado do processo anterior, que considera uma abordagem gulosa, é um valor total dos objetos na mochila igual a 17. Veja que esse mesmo exemplo considerado com o algoritmo anterior resultou em um valor acumulado igual a 34, selecionando os objetos i = 0 e $i = 3$. Veja que esse resultado é o dobro do resultado obtido pela estratégia gulosa. Isso ocorreu porque definimos um critério de atratividade (valor de utilidade) e o utilizamos olhando para o contexto de cada iteração, individualmente, sem nos preocuparmos com o impacto nas iterações posteriores.

Algoritmo 10 – Versão considerando a abordagem gulosa para a resolução do problema da mochila booleana

MochilaGulosa(v,p,w,n)
1. $q \leftarrow 0$
2. ***para*** $i \leftarrow 0$ *até* $n–1$
3. $u[i] \leftarrow v[i]/p[i]$
4. *Ordenar(u,v,p)*
5. ***para*** $i \leftarrow 0$ até $n–1$
6. ***se*** $p[i] \leq w$
7. $q \leftarrow q + v[i]$
8. $w \leftarrow w - p[i]$
9. ***retornar*** q

Uma representação possível para a abordagem gulosa do problema da mochila é descrita no Algoritmo 10. Veja que os parâmetros de entrada seguem em conformidade com os exemplos anteriores. O cálculo dos valores de utilidade é feito na linha 3, onde armazenamos em um novo vetor $u[0, \ldots, n-1]$ os quocientes resultantes da divisão dos valores pelos pesos.

A ordenação é feita considerando-se o vetor dos valores de utilidade. Uma observação importante é que todas as trocas realizadas em u são repetidas em *v* e *p*, de modo a obter os valores e pesos, de cada objeto, na ordem especificada pelos valores de utilidade. Esse procedimento é abstraído pela chamada ao algoritmo *Ordenar*() (linha 4), o qual pode ser implementado utilizando-se, por exemplo, o *Merge Sort*.

O laço, na linha 5, verifica se o peso do objeto é menor ou igual à capacidade atualizada da mochila. Lembrando que os pesos estão ordenados em acordo com os respectivos valores de utilidade. Os objetos selecionados têm seus valores acumulados na variável *q* e seus respectivos pesos são descontados da capacidade da mochila, representada pela variável *w*.

Considerando a linguagem de programação de sua preferência, faça a implementação do algoritmo *MochilaGulosa*(), teste diferentes instâncias e verifique se os resultados são ótimos.

Nesta unidade estudamos diferentes técnicas de projeto de algoritmos. Essas técnicas não são excludentes, podendo ser consideradas mais que uma técnica no projeto de um mesmo algoritmo. A escolha da abordagem que será utilizada na concepção de um algoritmo deve ser feita considerando-se o objetivo pretendido e as características de cada problema.

Os algoritmos apresentados nesta unidade descrevem a ideia central de cada uma das abordagens de projeto. Veja que o aprendizado das técnicas descritas é condicionado à prática da implementação. Assim como na matemática, aprender sobre programação e conceitos correlatos é uma questão de prática; assim, a implementação dos algoritmos propostos nesta unidade é de fundamental importância para a interiorização dos conceitos que foram discutidos.

Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

Vídeos

Recursividade além da matemática
https://youtu.be/VebJ3qeF7xs

Pesquisa operacional II – Aula 05 – O problema da mochila (*knapsack problem*)
https://youtu.be/0GR8T8uL8iw

Técnicas de projeto de algoritmos e problemas de otimização
https://youtu.be/IvtCMZSZ5EE

Merge Sort vs Quick Sort
https://youtu.be/es2T6KY45cA

Referências

CORMEN, T. H. *et al*. ***Introduction to algorithms.*** MIT press, 2009.

TOSCANI, L. V. **Complexidade de algoritmos**. v. 13: UFRGS. 3ª edição. Porto Alegre Bookman 2012. (*e-book*)

ZIVIANI, N. **Projeto de algoritmos**: com implementações em JAVA e C. São Paulo: Cengage Learning, 2012. (*e-book*)


Cruzeiro do Sul
Educacional