

Nivelamento em Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto



- 1 Problema Computacional
- 2 Problemas Combinatórios
- 3 Paradigmas de Projeto de Algoritmos
 - Busca Completa
 - Backtracking
 - Algoritmos Gulosos
 - Divisão e Conquista
 - Programação Dinâmica

Licença

Este material está licenciado sob a Creative Commons BY-NC-SA 4.0. Isto significa que o material pode ser compartilhado e adaptado, desde que seja atribuído o devido crédito, que o material não seja utilizado de forma comercial e que o material resultante seja distribuído de acordo com a mesma licença.

Resumindo... Algoritmo

Definição Informal

É qualquer procedimento computacional bem definido que toma algum valor ou conjunto de valores como **entrada** e produz algum valor ou conjunto de valores como **saída**. Portanto, é uma sequência de passos computacionais que transformam uma entrada na saída.

Definição Informal 2

Uma ferramenta para resolver um **problema computacional** bem especificado. O enunciado do problema especifica em termos gerais o relacionamento entre entrada e saída. O algoritmo descreve um procedimento computacional para alcançar esse relacionamento da entrada com a saída.

Problema Computacional

Definição

Um problema computacional pode ser visto como uma coleção infinita de instâncias junto com uma solução para cada instância.

Definição Informal

Um problema computacional é uma questão geral a ser respondida, possuindo determinados parâmetros.

Exemplo

Problema de Ordenação

Ordenar uma sequência de números de maneira crescente.

Entrada

Uma sequência de n números $\langle a_1, a_2, \dots, a_n \rangle$.

Saída

Uma permutação $\langle a'_1, a'_2, \dots, a'_n \rangle$ da sequência de entrada, tal que $a'_1 < a'_2 < \dots < a'_n$.

Instância de um Problema

Uma **instância** de um problema consiste na entrada necessária para se calcular uma solução para o problema, por exemplo $\langle 31, 41, 59, 25, 47 \rangle$.

Problema Computacional

Problema de Decisão

Tipo de problema computacional em que a resposta para cada instância é **sim** ou **não**:

“Dadas uma lista de cidades e as distâncias entre todas, há uma rota que visite todas as cidades e retorne à cidade original com distância total menor do que 500 km?”

Problema de Otimização

Tipo de problema computacional em que é necessário determinar a **melhor** solução possível entre todas as soluções viáveis.

“Dadas uma lista de cidades e as distâncias entre todas, determine a menor rota que visite todas as cidades e retorne à cidade original.”

Problema Computacional

Problema de Busca

Tipo de problema computacional em que é necessário determinar uma **possível** solução:

“Dadas uma lista de cidades e as distâncias entre todas, determine uma rota que visite todas as cidades e retorne à cidade original.”

Problema de Contagem

Tipo de problema computacional em que é necessário determinar a **quantidade** de soluções para um problema de busca:

“Dadas uma lista de cidades e as distâncias entre todas, quantas rotas que visitem todas as cidades e retornam à cidade original existem?”

Classificação *International Collegiate Programming Contest*

Em competições de programação é primordial classificar o problema de acordo com suas características de solução:

- ▶ *Ad Hoc*;
- ▶ Busca Completa (iterativa ou *backtracking*);
- ▶ Divisão e Conquista;
- ▶ Guloso (clássico e original);
- ▶ Programação Dinâmica;
- ▶ Teoria dos Grafos;
- ▶ Matemática (aritmética e álgebra);
- ▶ Geometria Computacional;
- ▶ Manipulação de Strings;
- ▶ Problemas Combinatórios;
- ▶ Outros problemas “mais difíceis”.

Observações Sobre a Classificação ICPC

Note que “Ordenação” não é considerado um problema computacional pelo ICPC por sua trivialidade e por normalmente ser utilizado como uma sub-rotina;

Embora não figure na lista, “Recursividade” está incluído implicitamente em outras categorias;

Estruturas de dados não são utilizadas para classificar problemas, normalmente os problemas que requerem uma estrutura específica são listados como “*Ad Hoc*”;

Claramente, um problema pode ser classificado em duas classes diferentes, como por exemplo aqueles que requerem o algoritmo *Floyd-Warshall* (Teoria dos Grafos e Programação Dinâmica).

Problemas *Ad Hoc*

Ad Hoc

- ▶ Para isto, para um determinado ato;
- ▶ Para este caso específico;
- ▶ Eventualmente investido em função provisória, para um fim especial;
- ▶ Único.

Problemas *Ad Hoc*

Problemas *Ad Hoc* são aqueles cujos algoritmos de solução não recaem em categorias bem estudadas:

- ▶ Cada problema é diferente do outro;
- ▶ Não existe técnica específica ou genérica para resolvê-los;
- ▶ A solução pode exigir uma estrutura nova ou um conjunto nada usual de laços e condições.

Classificação

Problemas *Ad Hoc* podem ser superficialmente categorizados:

- ▶ Triviais, fáceis, médios. . .
- ▶ Jogos (cartas, tabuleiros, etc.);
- ▶ Combinatórios (*Josephus*, Palíndromos, Anagramas);
- ▶ Problemas reais/Simulação;
- ▶ Simplesmente “*Ad Hoc*”.

Exemplo

Um mágico faz o seguinte truque tendo n cartas do naipe de espadas com a face para baixo:

- ▶ Se retirar a carta do topo, a próxima será o ás;
- ▶ Se retirar duas cartas, a próxima será o 2;
- ▶ Se retirar três cartas, a próxima será o 3...
- ▶ Cartas retiradas voltam ao fundo das $n - i$ cartas, menos as i cartas viradas.

Como embaralhar as cartas para que o truque funcione?



Definição

Um **problema combinatório** é um problema que possui um conjunto de **elementos** (ou **variáveis**) e para sua solução é exigida uma combinação de um subconjunto destes elementos.

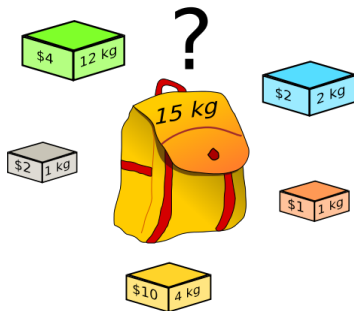
Diferentes combinações possuem diferentes valores, porém, o **objetivo** é **otimizar** a solução (achar a de maior valor – **maximização** ou a de menor valor – **minimização**) de acordo com a **função objetivo** ou **função de avaliação**.

As combinações são limitadas por **restrições**, que são regras que definem se uma combinação é **viável** ou **inviável**.

Problemas Combinatórios

O Problema da Mochila 0-1

Dadas uma mochila de capacidade W e uma lista de n itens distintos e únicos (enumerados de 1 a n), cada um com um peso w_1, w_2, \dots, w_n e um valor v_1, v_2, \dots, v_n , maximizar o valor carregado na mochila, respeitando sua capacidade.



Mais Definições

O **espaço de busca** de um problema combinatório é o conjunto de todas as soluções possíveis, podendo ser restrito as soluções viáveis ou não;

Uma **solução ótima global** é uma solução viável que atinge o melhor valor possível de acordo com a **função objetivo** de um problema combinatório;

Podemos ter uma ou múltiplas soluções ótimas para um problema, todas com o mesmo valor de avaliação da função objetivo, porém, com configurações diferentes;

Ao explorarmos o espaço de busca utilizando alguma técnica, realizamos **movimentos** entre soluções, ou seja, a partir de uma solução atual, a alteramos de uma determinada maneira e chegamos a uma outra solução;

Duas soluções que se diferem entre si por um movimento são ditas **vizinhas**.

Problemas Combinatórios

Exemplo - O Problema da Mochila 0-1

Os **elementos** a serem combinados são os itens;

O **espaço de busca** são todas as combinações de itens.

Uma **função objetivo** possível é a soma dos valores dos itens da mochila;

A função objetivo deve ser **maximizada**;

A única **restrição** é a de capacidade da mochila;

Uma **solução ótima global** é a solução viável cujos itens possuem a maior soma, ou seja, o melhor valor de função objetivo;

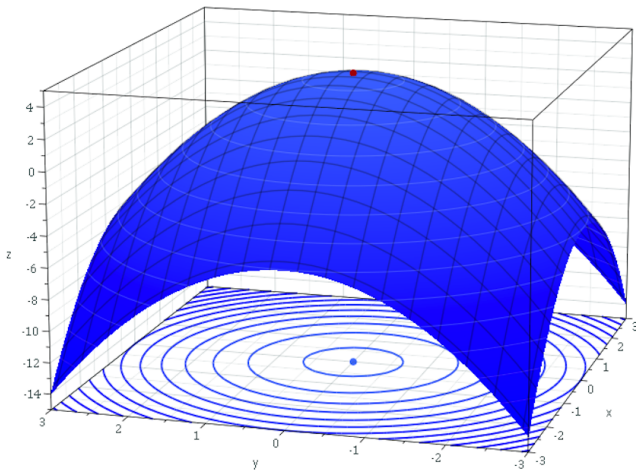
Remover um item da mochila, ou inserir um item caracteriza um **movimento**.

Exercício - O Problema da Mochila 0-1

Dê exemplos de soluções **vizinhas**.

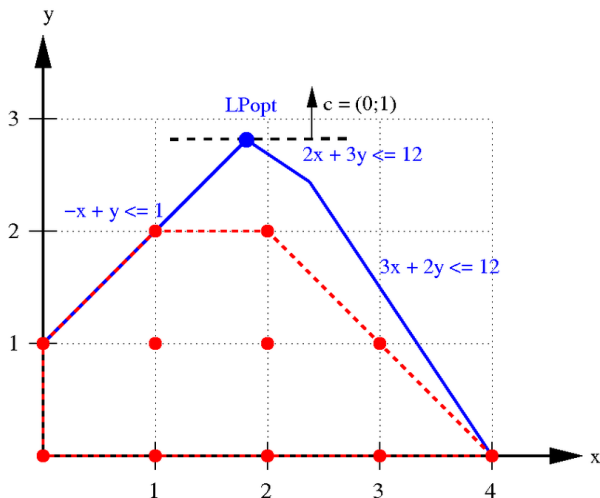
Função Objetivo e Ótimo Global

Gráfico da função $f(x, y) = -(x^2 + y^2) + 4$. A solução ótima $(0, 0, 4)$ que maximiza a função é indicada por um ponto vermelho.



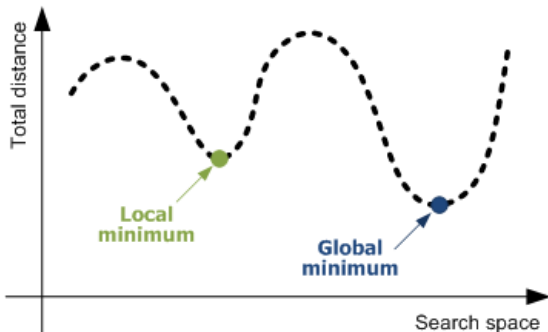
Viabilidade

Um problema com 5 restrições lineares (incluindo as 2 de não negatividade). Na ausência de restrições de integralidade, a região viável é representada em azul, caso contrário, em vermelho.



Mais Definições

Uma **solução ótima local** ou **subótima** é uma solução viável que atinge o melhor valor de função objetivo de um problema combinatório entre as soluções vizinhas.



Exemplo de ótimo local e global em um problema de minimização.

Caracterização

Em resumo, problema combinatório é composto dos seguintes componentes:

- ▶ Função objetivo;
- ▶ Conjunto de elementos;
- ▶ Conjunto de restrições.

Problemas Combinatórios

O Problema do Caixeiro Viajante

Dadas uma lista finita de cidades e as distâncias entre todas, determine a menor rota que visite todas as cidades e retorne à cidade original, sem repetir nenhuma cidade.



O Problema do Caixeiro Viajante

Elementos: Cidades;

Espaço de Busca: Todas as permutações da ordem das cidades;

Função Objetivo: Comprimento da rota;

Tipo de Função Objetivo: Minimização;

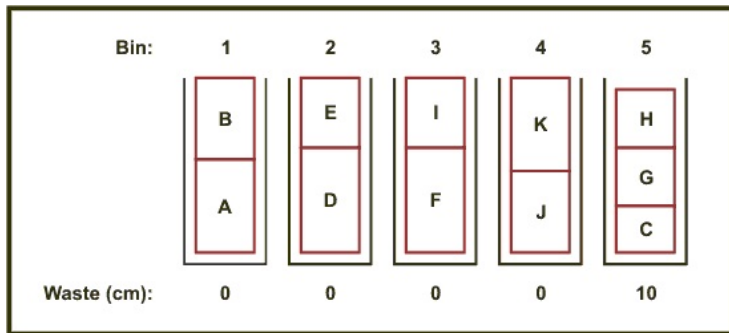
Restrições: Não repetir cidades, visitar todas as cidades;

Movimento: Trocar a posição de duas cidades na rota.

Problemas Combinatórios

Bin Packing

Dadas uma lista finita de objetos de volumes diferentes e uma lista finita de caixas de volume V , empacotar todos os objetos minimizando o número de caixas.



Bin Packing

Elementos: Objetos

Espaço de Busca: Todas as combinações de objetos e caixas;

Função Objetivo: Número de caixas utilizadas;

Tipo de Função Objetivo: Minimização

Restrição: Respeitar o volume das caixas;

Movimento: Inserir ou remover um objeto de uma caixa.

Subset Sum

Dado um conjunto (ou multiconjunto) de números inteiros, determinar se há um subconjunto não vazio cuja soma seja exatamente s .

Exemplo

Consideremos o conjunto $\{-7, -3, -2, 5, 8\}$ e $s = 0$.

A resposta é *sim*, porque o subconjunto $\{-3, -2, 5\}$ possui soma s .

Subset Sum

Elementos: os números inteiros;

Espaço de Busca: todos os subconjuntos do conjunto original;

Função Objetivo: soma dos elementos;

Tipo de Função Objetivo: problema de decisão;

Restrição: a soma dos elementos deve ser s ;

Movimento: adicionar ou remover um elemento do subconjunto.

O Problema de Minimização de Banda em Matrizes

Define-se o problema a partir de uma matriz M quadrada e esparsa, podendo esta ser simétrica ou não;

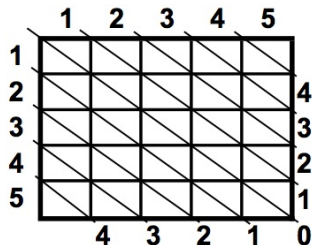
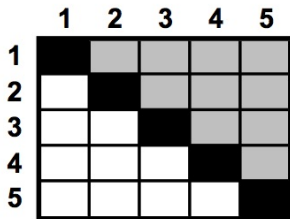
A banda de uma matriz é a banda diagonal que compreende as diagonais que contenham elementos não nulos mais distantes da diagonal principal em ambas as direções;

O objetivo é, portanto, minimizar as distâncias de tais diagonais mais afastadas da diagonal principal da matriz à esquerda e à direita.

O Problema de Minimização de Banda em Matrizes

Ao lado esquerdo da diagonal principal denomina-se meia banda esquerda e ao lado direito da diagonal principal denomina-se meia banda direita;


A figura abaixo identifica as regiões citadas e ilustra a numeração das diagonais de uma matriz.



O Problema de Minimização de Banda em Matrizes

Seja M uma matriz esparsa, k_1 a meia banda esquerda e k_2 a meia banda direita, a largura da banda de uma matriz é dada por $k_1 + k_2 + 1$.

	1	2	3	4	5
3	1	1	1	1	0
4	1	1	0	1	0
1	1	1	0	0	0
2	0	1	1	0	0
5	0	0	0	0	1



	1	2	3	4	5
1	1	1	0	0	0
4	1	1	0	1	0
3	1	1	1	1	0
2	0	1	1	0	0
5	0	0	0	0	1

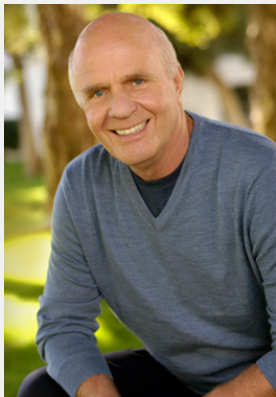
Citação I



"If all you have is a hammer, everything looks like a nail."

- Abraham Maslow, 1966.

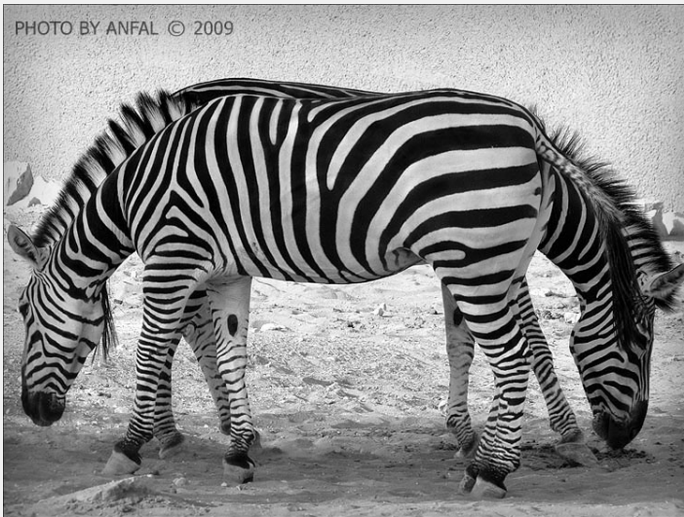
Citação II



*"If you change the way you look at things,
the things you look at change."*
- Wayne Dyer.

Solução de Problemas vs. Perspectiva Correta

PHOTO BY ANFAL © 2009



Introdução

Existem paradigmas clássicos para abordagem de problemas computacionais, cada um adequado a um tipo de problema:

- ▶ Busca Completa/*Backtracking*;
- ▶ Algoritmos Gulosos;
- ▶ Divisão e Conquista;
- ▶ Programação Dinâmica.

Dominar estes paradigmas nos auxilia a utilizarmos a “ferramenta” adequada, ao invés de martelarmos todos os problemas.

Ian Parberry e William Gasarch, *Problems on Algorithms*

"Often it appears that there is no better way to solve a problem than to try all possibilities.

This approach, called exhaustive search, is almost always slow, but sometimes it is better than nothing. It may actually be practical in real situations if the problem size is small enough."

Busca Completa

Definição

Em sua forma mais simples, a **Busca Completa** (enumeração completa, busca exaustiva ou força bruta) consiste em varrer o espaço de busca inteiro (se necessário) em busca da solução desejada.

Observações

Este método de solução não é eficiente em termos de complexidade, uma vez que tende a enumerar todas as possíveis soluções;

Por exemplo, em um problema combinatório em que o espaço de busca é definido pelas permutações de n elementos, a complexidade de um algoritmo de busca completa é $O(n!)$, o que pode ser impraticável mesmo para valores relativamente pequenos de n ;

Note que, algoritmos de busca completa possuem a propriedade de **corretude**: encontram a solução desejada para quaisquer instâncias.

Exemplo

Determine todos os pares de números de 5 dígitos que possuam os dígitos de 0 a 9 sem repetição, tal que o primeiro dividido pelo segundo seja igual a um inteiro N , em que $2 \leq N \leq 79$.

Em outras palavras $abcde/ghij = N$, em que cada letra representa um dígito diferente. O primeiro dígito pode ser zero, e.g., $79546/01283 = 62$; $94736/01528 = 62$.

Observações

Analisando o problema, temos que $ghij$ pode assumir valores entre 01234 e 98765, o que gera ≈ 100.000 possibilidades;

Para cada $ghij$, podemos obter o valor de $abcde$ pela multiplicação $ghij \times N$ e verificar se todos os dígitos são diferentes;

Neste caso, a busca completa é viável, embora frequentemente este não seja o caso.

Exemplo

Determine todos os pares de números de 5 dígitos que possuam os dígitos de 0 a 9 sem repetição, tal que o primeiro dividido pelo segundo seja igual a um inteiro N , em que $2 \leq N \leq 79$.

Em outras palavras $abcde/ghij = N$, em que cada letra representa um dígito diferente. O primeiro dígito pode ser zero, e.g., $79546/01283 = 62$; $94736/01528 = 62$.

Observações

Analisando o problema, temos que $ghij$ pode assumir valores entre 01234 e 98765, o que gera ≈ 100.000 possibilidades;

Para cada $ghij$, podemos obter o valor de $abcde$ pela multiplicação $ghij \times N$ e verificar se todos os dígitos são diferentes;

Neste caso, a busca completa é viável, embora frequentemente este não seja o caso.

Ian Parberry e William Gasarch, *Problems on Algorithms*

"Backtracking is a type of exhaustive search in which the combinatorial object is constructed recursively, and the recursion tree is pruned, that is, recursive calls are not made when the part of the current object that has been constructed cannot lead to a valid or optimal solution"

Definição

O paradigma *Backtracking* (ou “Tentativa e Erro”) é relacionado à Busca Completa (ou “Força Bruta”), no sentido de explorar o espaço de busca inteiro (se necessário) em busca da solução desejada;

É considerado um refinamento da Busca Completa, uma vez que pode eliminar parte do espaço de busca sem examiná-lo;

O princípio é construir uma solução gradativamente na base da tentativa e erro, desmanchando partes da solução quando necessário;

Normalmente, este paradigma é associado a algoritmos recursivos.

Aplicação

O *Backtracking* é aplicado em problemas cuja solução pode ser definida construtivamente, por meio de uma sequência de decisões;

Outra característica dos problemas passíveis de solução por *Backtracking* é a capacidade de modelagem por uma árvore que representa todas as possíveis sequências de decisão.

Justificativa

Se houver mais do que uma decisão disponível para cada uma das n decisões, a busca completa será exponencial;

A eficiência do *Backtracking* depende da capacidade de limitar esta busca, ou seja, podar a árvore, eliminando as ramificações não promissoras;

Desta forma, é necessário definir o espaço de busca para o problema:

- ▶ Que inclua a solução ótima;
- ▶ Que possa ser pesquisada de forma organizada, tipicamente, como uma árvore.

Ian Parberry e William Gasarch, *Problems on Algorithms*

"A greedy algorithm starts with a solution to a very small subproblem and augments it successively to a solution for the big problem.

The augmentation is done in a 'greedy' fashion, that is, paying attention to short-term or local gain, without regard to whether it will lead to a good long-term or global solution.

As in real life, greedy algorithms sometimes lead to the best solution, sometimes lead to pretty good solutions, and sometimes lead to lousy solutions. The trick is to determine when to be greedy.

[Most greedy algorithms are deceptively simple.]

One thing you will notice about greedy algorithms is that they are usually easy to design, easy to implement, easy to analyze, and they are very fast, but they are almost always difficult to prove correct."

Justificativa

Algoritmos para problemas de otimização tipicamente realizam uma sequência de passos, com um conjunto de opções a cada passo;

Para muitos deles, utilizar busca completa ou outros paradigmas pode ser lento demais – algoritmos mais “simples” ou mais “espertos” os resolveriam;

Ainda, o problema pode ser tão difícil que abrimos mão de obtermos a solução ótima em troco de uma solução “razoavelmente boa”.

Definição

Um algoritmo guloso sempre faz a escolha que parece ser correta no momento, ou seja, escolhe sempre o **ótimo local** na esperança que isto o leve até a solução **ótima global**;

No entanto, nem todos os problemas permitem que algoritmos gulosos obtenham soluções ótimas.

Caracterização dos Algoritmos

O processo de construção da solução é iterativo, dividido em passos;

A cada passo, existe um conjunto ou lista de elementos candidatos a fazerem parte da solução;

A cada passo, um destes elementos candidatos é adicionado à solução;

Deve haver uma maneira de verificar se um conjunto específico de elementos produzem uma **solução completa** ou uma **solução parcial**;

Deve haver também uma maneira de verificar se uma dada solução parcial é **viável** ou não;

Uma **função de seleção** determina a cada passo, entre os elementos candidatos, qual é o mais promissor;

A **função objetivo** determina o valor da solução encontrada, seja ela completa ou parcial.

Ian Parberry e William Gasarch, *Problems on Algorithms*

"Divide-and-Conquer is perhaps the most commonly used algorithm design technique in computer science.

Faced with a big problem P , divide it into smaller subproblems, solve these subproblems, and combine their solutions into a solution for P .

But how do you solve the smaller problems?

Simply divide each of the small problems into smaller problems, and keep doing this until the problems become so small that it is trivial to solve them.

Sound like recursion? Not surprisingly, a recursive procedure is usually the easiest way of implementing divide-and-conquer"

Definição

Divisão e Conquista (ou Dividir e Conquistar, ou ainda, D&C) é um paradigma de solução de problemas no qual tentamos simplificar a solução do problema original dividindo-o em subproblemas menores e resolvendo-os (ou “conquistando-os”) separadamente;

O processo:

- ▶ **Dividir** o problema original em **subproblemas** – normalmente com a metade (ou algo próximo disto) do tamanho do problema original, porém com a mesma estrutura;
- ▶ Determinar a solução dos subproblemas (comumente, de maneira recursiva) – que agora se tornam mais “fáceis”;
- ▶ Se necessário, **combinar** as soluções dos subproblemas para produzir a solução completa para o problema original.

Utilização

Existem quatro condições que indicam se o paradigma D&C pode ser aplicado com sucesso:

- ▶ Deve ser possível dividir o problema em subproblemas;
- ▶ A combinação de resultados deve ser eficiente;
- ▶ Os subproblemas devem possuir tamanhos parecidos dentro de um mesmo nível;
- ▶ A solução dos subproblemas são operações repetidas ou correlacionadas.

Utilização

Classicamente, o paradigma D&C é utilizado em vários algoritmos de ordenação:

- ▶ *Quick Sort*;
- ▶ *Merge Sort*;
- ▶ *Heap Sort*.

Outro caso de sucesso é a Busca Binária, embora este algoritmo não resolva todos os subproblemas nem combine os resultados.

A maneira como os dados são organizados em estruturas como Árvore Binária de Busca, *Heap* e Árvore de Segmentos também possui o espírito do D&C.

Ian Parberry e William Gasarch, *Problems on Algorithms*

"Dynamic programming is a fancy name for [recursion] with a table. Instead of solving subproblems recursively, solve them sequentially and store their solutions in a table.

The trick is to solve them in the right order so that whenever the solution to a subproblem is needed, it is already available in the table.

Dynamic programming is particularly useful on problems for which divide-and-conquer appears to yield an exponential number of subproblems, but there are really only a small number of subproblems repeated exponentially often.

In this case, it makes sense to compute each solution the first time and store it away in a table for later use, instead of recomputing it recursively every time it is needed."

Observação

A palavra **programação** na expressão “programação dinâmica” não tem relação direta com programação de computadores;

Ela significa planejamento e refere-se à construção da tabela que armazena as soluções dos subproblemas.

Justificativa

Se a soma do tamanho dos subproblemas é $O(n)$, temos uma boa probabilidade de que o algoritmo recursivo tenha complexidade polinomial;

Entretanto, se temos n subproblemas de tamanho $n - 1$ cada, a tendência é de que o algoritmo recursivo associado tenha complexidade exponencial;

A Programação Dinâmica nos fornece uma maneira de projetar algoritmos que:

- ▶ Sistemáticamente exploram todas as possibilidades (corretude);
- ▶ Evitam computação redundante (eficiência).

Características

À medida em que resolvemos subproblemas, armazenamos os resultados parciais, acelerando o algoritmo:

- ▶ Para cada subproblema (ou **estado**) inédito, o resolvemos e armazenamos o resultado;
- ▶ Para os subproblemas repetidos, apenas consultamos o resultado.

É importante, primeiro nos certificamos de que o algoritmo recursivo é correto, depois o aceleramos;

As soluções dos subproblemas são mantidas em uma **tabela**, a qual é consultada a cada subproblema encontrado;

Cada entrada na tabela é chamada de **estado** ou **estágio**.

Caracterização dos Problemas

A PD é aplicável a problemas que possuam as seguintes propriedades:

Formulação Recursiva Bem Definida: no caso de formulação recursiva, esta não deve possuir ciclos;

Subestrutura Ótima: O problema pode ser dividido sucessivamente, e a combinação das soluções ótimas dos subproblemas corresponde à solução ótima do problema original;

Superposição de Subproblemas: O espaço de subproblemas é pequeno e eles se repetem com frequência durante a solução do problema original.

Dúvidas?

