Capítulo 5
Algoritmos Gulosos e Programação Dinâmica

Manoel Ribeiro Filho

5.1 Algoritmos gulosos - **Algoritmos e complexidade Notas de aula - Marcus Ritt – Capítulo 4**

Algoritmos gulosos se aplicam a **problemas de otimização**.

É um problema de **encontrar uma melhor solução de todas as soluções viáveis**.
A otimização é o processo de pesquisa da melhor opção em termos do aproveitamento dos recursos dentro de uma categoria de possíveis soluções a partir das variáveis do projeto.

Ideia principal: Decide localmente.

Um algoritmo guloso constrói uma solução de um problema

1. Começa com uma solução inicial.
2. Melhora essa solução com uma decisão local (gulosamente!).
3. Nunca revisa uma decisão.

Por causa da localidade: Algoritmos gulosos frequentemente são apropriados para processamento online

Trocar Notas (ou moedas)

Troca mínima

Instância Valores (de moedas ou notas) $v_1 > v_2 > \cdots > v_n = 1$, uma soma s.

Solução Números $c_1, \ldots, c_n$ tal que $s = \sum_{1 \leq i \leq n} c_i v_i$

Objetivo Minimizar o número de unidades $\sum_{1 \leq i \leq n} c_i$.

A abordagem gulosa

```
1  for i:=1,...,n do
2     c_i := ⌊s/v_i⌋
3     s := s − c_i v_i
4  end for
```

Exemplo

Exemplo 4.1
Com $v_1 = 500, v_2 = 100, v_3 = 25, v_4 = 10, v_5 = 1$ e $s = 3.14$, obtemos $c_1 = 0, c_2 = 3, c_3 = 0, c_4 = 1, c_5 = 4$.
Com $v_1 = 300, v_2 = 157, v_3 = 1$, obtemos $v_1 = 1, v_2 = 0, v_3 = 14$.
No segundo exemplo, existe uma solução melhor: $v_1 = 0, v_2 = 2, v_3 = 0$.
No primeiro exemplo, parece que a abordagem gulosa acha a melhor solução. Qual a diferença? ◊

Uma condição simples é que todos valores maiores são múltiplos inteiros dos menores

Lema
A solução do algoritmo guloso é a única que satisfaz

$$\sum_{i \in [m,n]} c_i v_i < v_{m-1}$$

para $m \in [2, n]$. (Ela é chamada a *solução canônica.*)

Otimalidade da abordagem gulosa

A pergunta pode ser generalizada: Em quais circunstancias um algoritmo guloso produz uma solução ótima?

Se existe um solução gulosa: frequentemente ela tem uma implementação simples e eficiente.

Infelizmente, para um grande numero de problemas não tem algoritmo guloso ótimo .

Uma condição (que se aplica também para programação dinâmica) é a **subestrutura ótima**.

A teoria de matroides e greedoides estuda as condições de otimalidade de algoritmos gulosos

Definição (Subestrutura ótima)
Um problema de otimização tem *subestrutura ótima* se uma solução ótima (mínima ou máxima) do problema consiste em soluções ótimas das subproblemas.

Informalmente, um algoritmo guloso é caracterizado por uma subestrutura ótima e a característica adicional, que podemos escolher o subproblema que leva a solução ótima através de uma regra simples.

Portanto, o algoritmo guloso evita resolver todos os subproblemas.

Uma subestrutura ótima é uma condição necessária para um algoritmo guloso ou de programação dinâmica ser ótima, mas ela não é suficiente.

Exercício 4.2 (Análise de series)
Suponha uma série de eventos, por exemplo, as transações feitos na bolsa de forma

compra Dell, vende HP, compra Google, ...

Uma certa ação pode acontecer mais que uma vez nessa sequência. O problema: Dado uma outra sequência, decida o mais rápido possível se ela é uma subsequência da primeira.
Achar um algoritmo eficiente (de complexidade $O(m+n)$ com sequência de tamanho n e m), prova a corretude a analise a complexidade dele.
(Fonte: [57]).

Exercício 4.3 (Comunicação)
Imagine uma estrada comprida (pensa em uma linha) com casas ao longo dela. Suponha que todas casas querem acesso ao comunicação com celular. O problema: Posiciona o número mínimo de bases de comunicação ao longo da estrada, com a restrição que cada casa tem quer ser ao máximo 4 quilômetros distante de uma base.
Inventa um algoritmo eficiente, prova a corretude e analise a complexidade dele.
(Fonte: [57]).

Fim de Algoritmos e complexidade Notas de aula - Marcus Ritt – Capítulo 4
Trabalho em equipe para a última prova – Implementar os exercícios 4.2 e 4.3, da página 94 do Ritt.

Algoritmos Gulosos - IME

Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" que vê pela frente. (A definição de "apetitoso" é estabelecida a priori.)
O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.

Um algoritmo guloso é "míope": ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás : as escolhas que faz em cada iteração são definitivas.

Embora algoritmos gulosos pareçam obviamente corretos, a prova de sua correção é, em geral, muito sutil. Para compensar, algoritmos gulosos são muito rápidos e eficientes.

Veja o capítulo 16 do CLRS

capítulo 16 do CLRS

Algoritmos para problemas de otimização, normalmente passam por uma sequência de etapas e cada etapa tem um conjunto de escolhas.

Para muitos problemas de otimização é um exagero utilizar programação dinâmica para determinar as melhores escolhas: algoritmos mais simples e mais eficientes servirão.

Um ***algoritmo guloso*** sempre faz a escolha que parece ser a melhor no momento em questão. Isto é, faz uma escolha localmente ótima, na esperança de que essa escolha leve a uma solução globalmente ótima.

Algoritmos gulosos nem sempre produzem soluções ótimas, mas as produzem para muitos problemas.

O método guloso é bastante poderoso e funciona bem para uma ampla faixa de problemas.

Primeiro, examinaremos na Seção 16.1 um problema simples, mas não trivial: **o problema da seleção de atividades**, para o qual um algoritmo guloso calcula eficientemente uma solução. Chegaremos ao algoritmo guloso, considerando primeiro uma abordagem de programação dinâmica e depois mostrando que sempre podemos fazer escolhas gulosas para chegar a uma solução ótima.

A Seção 16.2 revê os elementos básicos da abordagem gulosa, dando uma abordagem direta para provar a correção de algoritmos gulosos.

A Seção 16.3 apresenta uma aplicação importante das técnicas gulosas: o **projeto de códigos de compressão de dados (Huffman).**

Na Seção 16.4, investigamos um pouco da teoria subjacente às estruturas combinatórias denominadas “matroides”, para as quais um algoritmo guloso sempre produz uma solução ótima. Finalmente, a Seção 16.5 aplica matroides para resolver um problema de programação de tarefas de tempo unitário com prazos finais e multas.

Trabalho em equipe para a última prova – Implementar **o problema da seleção de atividades** (16.1 clrs) e **projeto de códigos de compressão de dados (Huffman).** (16.3 clrs)

16.1 UM PROBLEMA DE SELEÇÃO DE ATIVIDADES

Nosso primeiro exemplo é o problema de programar várias atividades concorrentes que requerem o uso exclusivo de um recurso comum, com o objetivo de selecionar um conjunto de tamanho máximo de atividades mutuamente compatíveis. Suponha que tenhamos um conjunto $S = \{a_1, a_2, \ldots, a_n\}$ de n ***atividades*** propostas que desejam usar um recurso (por exemplo, uma sala de conferências) que só pode ser utilizado por uma única atividade por vez. Cada atividade a_i tem um ***tempo de início*** s_i e um ***tempo de término*** f_i, onde $0 \leq s_i < f_i < \infty$. Se selecionada, a atividade a_i ocorre durante o intervalo de tempo meio aberto $[s_i, f_i)$. As atividades a_i e a_j são ***compatíveis*** se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem. Isto é, a_i e a_j são compatíveis se $s_i \geq f_j$ ou $s_j \geq f_i$). No ***problema de seleção de atividades***, queremos selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis. Supomos que as atividades estão organizadas em ordem monotonicamente crescente de tempo de término:

16.2 Elementos da estratégia gulosa

Um algoritmo guloso obtém uma solução ótima para um problema fazendo uma sequência de escolhas. Para cada ponto de decisão, o algoritmo escolhe a opção que parece melhor no momento. Essa estratégia heurística nem sempre produz uma solução ótima, mas, como vimos no problema de seleção de atividades, algumas vezes, funciona. Esta seção discute algumas propriedades gerais de métodos gulosos.

O processo que seguimos na Seção 16.1 para desenvolver um algoritmo guloso foi um pouco mais complicado que o normal. Seguimos estas etapas:

1. Determinar a subestrutura ótima do problema.
2. Desenvolver uma solução recursiva. (Para o problema de seleção de atividades, formulamos a recorrência (16.2), mas evitamos desenvolver um algoritmo recursivo baseado nessa recorrência.)
3. Provar que, se fizermos a escolha gulosa, restará somente um subproblema.
4. Provar que, é sempre seguro fazer a escolha gulosa. (As etapas 3 e 4 podem ocorrer em qualquer ordem.)
5. Desenvolver um algoritmo recursivo que implemente a estratégia gulosa.
6. Converter o algoritmo recursivo em um algoritmo iterativo.

16.3 Códigos de Huffman

Códigos de Huffman comprimem dados muito efetivamente: economias de 20-90% são típicas, dependendo das características dos dados que estão sendo comprimidos. Consideramos os dados como uma sequência de caracteres. O algoritmo guloso de Huffman utiliza uma tabela que dá o número de vezes que cada caractere ocorre (isto é, suas frequências) para elaborar um modo ótimo de representar cada caractere como uma cadeia binária.

Suponha que tenhamos um arquivo de dados de 100.000 caracteres que desejamos armazenar compactamente. Observamos que os caracteres no arquivo ocorrem com as frequências dadas pela Figura 16.3. Isto é, somente seis caracteres diferentes aparecem, e o caractere a ocorre 45.000 vezes.

Há muitas opções para representar tal arquivo de informações. Aqui, consideramos o problema de projetar um ***código de caracteres binários*** (ou ***código***, para abreviar) no qual cada caractere seja representado por uma cadeia binária única que denominaremos **palavra de código**. Se usarmos um ***código de comprimento fixo***, precisaremos de três bits para representar seis caracteres: a = 000, b = 001, ..., f = 101. Esse método requer 300.000 bits para codificar o arquivo inteiro. Podemos fazer algo melhor?

Trabalho em equipe para a última prova – Implementar os algoritmos listados abaixo:

Intervalos disjuntos (Cap. 7 – Analise de Algoritmos USP)

Mochila de valor quase máximo (Cap. 10 – Analise de Algoritmos USP)

E os dois seguintes sugeridos em **Algoritmos Gulosos - IME**

4. [PARES DE LIVROS] Suponha dado um conjunto de livros numerados de 1 a n. Suponha que cada livro i tem um peso $p[i]$ que é maior que 0 e menor que 1. PROBLEMA: Acondicionar os livros no menor número possível de envelopes de modo que
   - cada envelope tenha ≤ 2 livros e
   - o peso do conteúdo de cada envelope seja ≤ 1.

   Escreva um algoritmo guloso que resolva o problema em tempo $O(n \log n)$ no pior caso. Aplique seu algoritmo a um exemplo interessante. Mostre que seu algoritmo está correto.

5. [CLRS 16.2-4] Quero dirigir um carro de uma cidade A a uma cidade B ao longo de uma redovia. O tanque de combustível do carro tem capacidade suficiente para cobrir n quilometros. O mapa da rodovia indica a localização dos postos de combustível. Dê um algoritmo que garanta uma viagem com número mínimo de reabastecimentos.

Capítulo 7

Intervalos disjuntos

Imagine que vários eventos (feiras, congressos, etc.) querem usar um certo centro de convenções. Cada evento gostaria de usar o centro durante um intervalo de tempo que começa num dia a e termina num dia b. Dois eventos são compatíveis se os seus intervalos de tempo são disjuntos. A administração do centro quer atender o maior número possível de eventos que sejam mutuamente compatíveis.

A figura 7.1 mostra um exemplo com oito eventos. Há três eventos mutuamente compatíveis, mas não há quatro.

Figura 7.1: Cada linha horizontal representa o intervalo de tempo de um evento (o menor tem 5 dias e o maior tem 26 dias). A cor mais escura identifica um conjunto de eventos mutuamente compatíveis.

Capítulo 10

Mochila de valor quase máximo

O algoritmo de programação dinâmica para o problema da mochila (veja o Capítulo 9) é lento. Dada a dificuldade de encontrar um algoritmo mais rápido,[1] faz sentido reduzir nossas exigências e procurar por uma solução *quase* ótima. Um tal algoritmo deve calcular um conjunto viável de valor maior que uma fração predeterminada (digamos 50%) do valor máximo.

10.1 O problema

Convém repetir algumas das definições da Seção 9.1. Dados números naturais $p_1, \ldots, p_n$, c e $v_1, \ldots, v_n$, diremos que p_i é o **peso** e v_i é o **valor** de i. Para qualquer subconjunto S de $\{1, \ldots, n\}$, denotaremos por $p(S)$ a soma $\sum_{i \in S} p_i$ e diremos que S é **viável** se $p(S) \leq c$. O **valor** de S é o número $v(S) := \sum_{i \in S} v_i$. Um conjunto viável S^* é **ótimo** se $v(S^*) \geq v(S)$ para todo conjunto viável S.

Algoritmos Gulosos - IME

Algoritmos gulosos

Esta página é uma introdução à *estratégia gulosa* (= gananciosa = *greedy*) de solução de problemas. Poderíamos também apresentar o assunto como *paradigma guloso* de concepção de algoritmos.

Para resolver um problema, um algoritmo guloso escolhe, em cada iteração, o objeto mais "apetitoso" que vê pela frente. (A definição de "apetitoso" é estabelecida a priori.) O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.

Um algoritmo guloso é "míope": ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás: as escolhas que faz em cada iteração são definitivas.

Embora algoritmos gulosos pareçam óbviamente corretos, a prova de sua correção é, em geral, muito sutil. Para compensar, algoritmos gulosos são muito rápidos e eficientes. (É preciso dizer, entretanto, que os problemas que admitem soluções gulosas são um tanto raros.)

Veja o capítulo 16 do CLRS. Veja também o verbete *Greedy algorithm* na Wikipedia.

5.2 Programação Dinâmica

Muitos algoritmos eficientes seguem o paradigma da *programação dinâmica*. Esse paradigma, ou estratégia de projeto de algoritmos, é uma espécie de tradução iterativa inteligente da recursão e pode ser definido, vagamente, como "recursão com o apoio de uma tabela".

Como em um algoritmo recursivo, cada instância do problema é resolvida a partir da solução de instâncias menores, ou melhor, de subinstâncias da instância original.

A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias.

O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

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 das subinstâncias

Para que o paradigma da programação dinâmica possa ser aplicado, é preciso que o problema tenha *estrutura recursiva*: a solução de toda instância do problema deve "conter" soluções de subinstâncias da instância.

Essa estrutura recursiva pode, em geral, ser representada por uma recorrência e a recorrência pode ser traduzida em um algoritmo recursivo.

O algoritmo recursivo é tipicamente ineficiente porque refaz, muitas vezes, a solução de cada subinstância.

Uma vez escrito o algoritmo recursivo, entretanto, é fácil construir uma tabela para armazenar as solução das subinstância e assim evitar que elas sejam recalculadas.

A tabela é a base de um algoritmo de programação dinâmica.

Para poder aplicar a programação dinâmica, é necessário decompor um problema em subproblemas menores, tal que o princípio de otimalidade (ou de subestrutura ótima) é satisfeito: uma solução ótima do problema pode ser obtida através de uma solução ótima de subproblemas.

Passos do desenvolvimento de um algoritmo de PD

1. Verificar a subestrutura ótima.
2. Expressar a solução em forma de recorrência.
3. Para implementar: usar memoização ou armazenar os possíveis valores da recorrência numa tabela, preenchendo a tabela de forma ascendente

Um exemplo de Programação Dinâmica, para resolução do Segmento de Soma Máxima, é o algoritmo ALTURAIV, do capítulo 4, da referência AnáliseAlgoritmosUSP.

A **altura** de um vetor A[1 ... n] é a soma de um segmento de soma máxima. Por exemplo para o vetor 20 -30 15 -10 30 -20 -30 30
É 15 – 10 + 30 = 35

Problema do segmento de soma máxima: Calcular a altura de um vetor A[1 ... n] de números inteiros.

O método da programação dinâmica, consiste em armazenar os resultados de cálculos intermediários numa tabela e assim evitar que eles sejam refeitos.

Para aplicar o método, será preciso introduzir o seguinte problema auxiliar, que restringe a atenção aos segmentos terminais do vetor: dado um vetor A[1 .. q], encontrar
a maior soma da forma A[i] + . . . + A[q] , com 1<= i <= q
Para facilitar a discussão, diremos que a maior soma dessa forma é a **semialtura** do vetor A[1 ... q]. A altura do vetor A[1 ... n] é o máximo das semialturas de
A[1 ... n], A[1 ... n - 1], A[1 ... n - 2], etc.
A semialtura de um vetor tem uma propriedade recursiva que a altura não tem:
se a soma de A[i ... q] é a semialtura de A[1 ... q] e i < q então a soma de A[i .. q-1] é a semialtura de A[1 .. q-1].
Se denotarmos a semialtura de A[1 ... q] por S[q], podemos resumir a propriedade recursiva por meio de uma recorrência: para qualquer vetor A[1 ... q],

S[q] = max(S[q-1] + A[q] , A[q]) **Essa é a recorrência encontrada**

Em outras palavras, S[q] = S[q - 1] + A[q] a menos que S[q - 1] seja estritamente negativo, caso em que S[q] = A[q].

A recorrência encontrada na transparência anterior serve de base o algoritmo, de programação dinâmica, para encontrar a altura A[1 ... n]. É o algoritmo seguinte:

```
ALTURAIV (A, n)
1 S[1] =  A[1]
2 para q =  2 até n faça
3         se S[q -1] >= 0
4                   então S[q] =  S[q-1] + A[q]
5                   senão S[q]  = A[q]
6 x  = S[1]
7 para q  = 2 até n faça
8         se S[q] > x então x =  S[q]
9 devolva x
```

O algoritmo está correto

Entre as linhas 1 e 5 são calculadas todas as semialturas e armazenadas na tabela S.

Entre as linhas 6 e 8 é calculada a maior semialtura, ou seja, a altura do vetor. E portanto, no fim do algoritmo, x é a altura correta de A[1 ... n].

Desempenho

A estimativa do consumo de tempo do algoritmo é muito fácil.
Cada linha do algoritmo é executada no máximo n vezes, e portanto o algoritmo consome tempo proporcional a n. Em outras palavras, o desempenho do algoritmo no pior caso está em $\theta(n)$ e portanto o algoritmo é linear. (O desempenho no melhor caso também está em $\theta(n)$.)

O algoritmo ALTURAIV é um exemplo de **programação dinâmica**.

O método só se aplica a problemas dotados de estrutura recursiva: qualquer solução de uma instância deve conter soluções de instâncias menores. Assim, o ponto de partida de qualquer algoritmo de programação dinâmica é uma recorrência.

A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias da instância original. (Nesse caso, trata-se da tabela S[1 ... n].) O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

Outros exemplos de programação dinâmica
As linhas de um parágrafo (Cap. 8 – Analise de Algoritmos USP)
Mochila de valor máximo (Cap. 9 – Analise de Algoritmos USP)

Da referência 12_Programação dinâmica
algoritmo da subsequência crescente máxima
algoritmo da subsequência comum máxima
multiplicação de cadeias de matrizes
algoritmo para o problema subset-sum
algoritmo para o problema da mochila booleana
algoritmo de Dijkstra

Capítulo 8

As linhas de um parágrafo

Um parágrafo de texto é uma sequência de palavras, sendo cada palavra uma sequência de caracteres. Queremos imprimir um parágrafo em uma ou mais linhas consecutivas de uma folha de papel de tal modo que cada linha tenha no máximo L caracteres. As palavras do parágrafo não devem ser quebradas entre linhas e cada duas palavras consecutivas numa linha devem ser separadas por um espaço.

Para que a margem direita fique razoavelmente uniforme, queremos distribuir as palavras pelas linhas de modo a minimizar a soma dos cubos dos espaços em branco que sobram no fim de todas as linhas exceto a última.

8.3 Um algoritmo de programação dinâmica

Para usar a recorrência (8.1) de maneira eficiente, basta interpretar X como uma tabela $X[1\,..\,n]$ e calcular $X[n]$, depois $X[n-1]$, e assim por diante. Esta é a técnica da programação dinâmica. O seguinte algoritmo implementa esta ideia. Ele devolve o defeito mínimo de um parágrafo $c_n, c_{n-1}, \ldots, c_1$ supondo $n \geq 1$, linhas de capacidade L e $c_i \leq L$ para todo i:

```
DefeitoMínimo (c_n, ..., c_1, L)
 1  para m ← 1 até n faça
 2      X[m] ← ∞
 3      k ← m
 4      s ← c_m
 5      enquanto k > 1 e s ≤ L faça
 6          X' ← (L − s)^3 + X[k−1]
 7          se X' < X[m] então X[m] ← X'
 8          k ← k − 1
 9          s ← s + 1 + c_k
10      se s ≤ L então X[m] ← 0
11  devolva X[n]
```

Capítulo 9

Mochila de valor máximo

Suponha dado um conjunto de objetos e uma mochila. Cada objeto tem um certo peso e um certo valor. Queremos escolher um conjunto de objetos que tenha o maior valor possível sem ultrapassar a capacidade (ou seja, o limite de peso) da mochila. Este célebre problema aparece em muitos contextos e faz parte de muitos problemas mais elaborados.

9.3 Algoritmo de programação dinâmica

Para obter um algoritmo eficiente a partir da recorrência (9.1-9.2), é preciso armazenar as soluções das subinstâncias numa tabela à medida que elas forem sendo obtidas, evitando assim que elas sejam recalculadas. Esta é a técnica da programação dinâmica, que já encontramos em capítulos anteriores.

Como c e todos os p_i são números naturais, podemos tratar X como uma tabela com linhas indexadas por $0\,..\,n$ e colunas indexadas por $0\,..\,c$. Para cada i entre 0 e n e cada b entre 0 e c, a casa $X[i,b]$ da tabela será o valor de uma solução da instância $(i,\, p,\, b,\, v)$. As casas da tabela X precisam ser preenchidas na ordem certa, de modo que toda vez que uma casa for requisitada o seu valor já tenha sido calculado.

O algoritmo abaixo recebe uma instância $(n,\, p,\, c,\, v)$ do problema e devolve o valor de uma solução da instância:[1]

```
MOCHILAPD (n, p, c, v)
1  para b ← 0 até c faça
2      X[0,b] ← 0
3      para j ← 1 até n faça
4          x ← X[j − 1, b]
5          se b − p_j ≥ 0
6              então y ← X[j − 1, b − p_j] + v_j
7                    se x < y então x ← y
8          X[j,b] ← x
9  devolva X[n,c]
```

Do CLRS

15.1 Corte de Hastes
15.2 Multiplicação de Cadeias de Matrizes

15.3 Elementos de programação dinâmica

Embora tenhamos acabado de analisar dois exemplos do método de programação dinâmica, é bem possível que você ainda esteja imaginando exatamente quando aplicar o método. Do ponto de vista da engenharia, quando devemos procurar uma solução de programação dinâmica para um problema? Nesta seção, examinamos os dois elementos fundamentais que um problema de otimização deve ter para que a programação dinâmica seja aplicável: subestrutura ótima e sobreposição de subproblemas. Também voltaremos a discutir mais completamente como a memoização pode nos ajudar a aproveitar a propriedade de sobreposição de subproblemas em uma abordagem recursiva de cima para baixo.

15.4 Subsequência comum mais longa

Em aplicações biológicas, muitas vezes, é preciso comparar o DNA de dois (ou mais) organismos diferentes. Um filamento de DNA consiste em uma cadeia de moléculas denominadas ***bases***, na qual as bases possíveis são adenina, guanina, citosina e timina. Representando cada uma dessas bases por sua letra inicial, podemos expressar um filamento de DNA como uma cadeia no conjunto finito {A,C,G,T}. (O Apêndice C dá a definição de uma cadeia.) Por exemplo, o DNA de um organismo pode ser S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, e o DNA de outro organismo pode ser S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Uma razão para a comparação de dois filamentos de DNA é determinar o grau de "semelhança" entre eles, que serve como alguma medida da magnitude da relação entre os dois organismos. Podemos definir (e definimos) a semelhança de muitas maneiras diferentes. Por exemplo, podemos dizer que dois filamentos de DNA são semelhantes se um deles for uma subcadeia do outro. (O Capítulo 32 explora algoritmos para resolver esse problema.) Em nosso exemplo, nem S_1 nem S_2 é uma subcadeia do outro. Alternativamente, poderíamos dizer que dois filamentos são semelhantes se o número de mudanças necessárias para transformar um no outro for pequeno. (O Problema 15-3 explora essa noção.) Ainda uma outra maneira de medir a semelhança entre filamentos S_1 e S_2 é encontrar um terceiro filamento S_3 no qual as bases em S_3 aparecem em cada um

Gula *versus* programação dinâmica(Algoritmos Gulosos – IME)

Às vezes é difícil distinguir um algoritmo guloso de um algoritmo de programação dinâmica. A seguinte lista grosseira de características pode ajudar.

Um algoritmo guloso

1. abocanha a alternativa mais promissora (sem explorar as outras)
2. é muito rápido,
3. nunca se arrepende de uma decisão já tomada,
4. não tem prova de correção simples.

Um algoritmo de programação dinâmica

1. explora todas as alternativa (mas faz isso de maneira eficiente),
2. é um tanto lento,
3. a cada iteração pode se arrepender de decisões tomadas anteriormente (ou seja, pode rever o "ótimocorrente"),
4. tem prova de correção simples.