Análise e Técnicas de Algoritmos
Período 2003.1

Programação Dinâmica

Programação Dinâmica

- Uma solução ótima baseada no método guloso é definida por uma seqüência de decisões locais ótimas.
- Quando o método guloso não funciona, uma possível saída seria a geração de todas as possíveis seqüências de decisões. A melhor seqüência seria então escolhida.
- Essa solução é de ordem exponencial, portanto, ineficiente.
- Programação dinâmica é uma técnica que objetiva diminuir o número de seqüências geradas.

Exemplo: Número de Combinações

Vamos considerar n itens dentre os quais tenho que escolher r. Qual o número possível de combinações de r itens?

Para escolher r itens de n, podemos proceder de duas formas:

1. Escolher o primeiro item. Escolher depois $r-1$ itens dos $n-1$ itens restantes.
2. Não escolher o primeiro item. Então devemos escolher r itens dos $n-1$ itens restantes.

Podemos então traduzir a solução da seguinte forma: $\binom{n}{r} = \binom{n-1}{r-1} + \binom{n}{r-1}$

Uma estratégia de divisão e conquista pode ser utilizada para implementar a solução acima.

Escolha(n, r)

if $r = 0$ ou $n = r$ **then**
  $return(1)$
else
  $return(Escolha(n-1, r-1) + Escolha(n-1, r))$

A análise deste algoritmo requer a resolução da seguinte relação de recorrência:

- $T(n) = c$, se $n = 1$.
- $T(n) = 2T(n-1) + d$, em caso contrário.

Resolvendo a relação de recorrência:

$$\begin{aligned} T(n) &= 2T(n-1) + d \\ &= 2(2T(n-2) + d) + d \\ &= 4T(n-2) + 2d + d \\ &= 4(2T(n-3) + d) + 2d + d \end{aligned}$$

$= 8T(n-3) + 4d + 2d + d$
$= 2^i T(n-i) + d\Sigma_{j=0}^{i-1} 2^j$
$= 2^{n-1} T(1) + d\Sigma_{j=0}^{n-2} 2^j$
$= (c+d)2^{n-i} - d$

Logo, $T(n) = O(2^n)$.

O maior problema nesta solução é que o mesmo problema é resolvido várias vezes.

Uma solução seria utilizar uma tabela para guardar as soluções de problemas que vão se repetir. Essa é a idéia básica da programação dinâmica.

Uma solução por programação dinâmica seria:

Escolha(n, r)

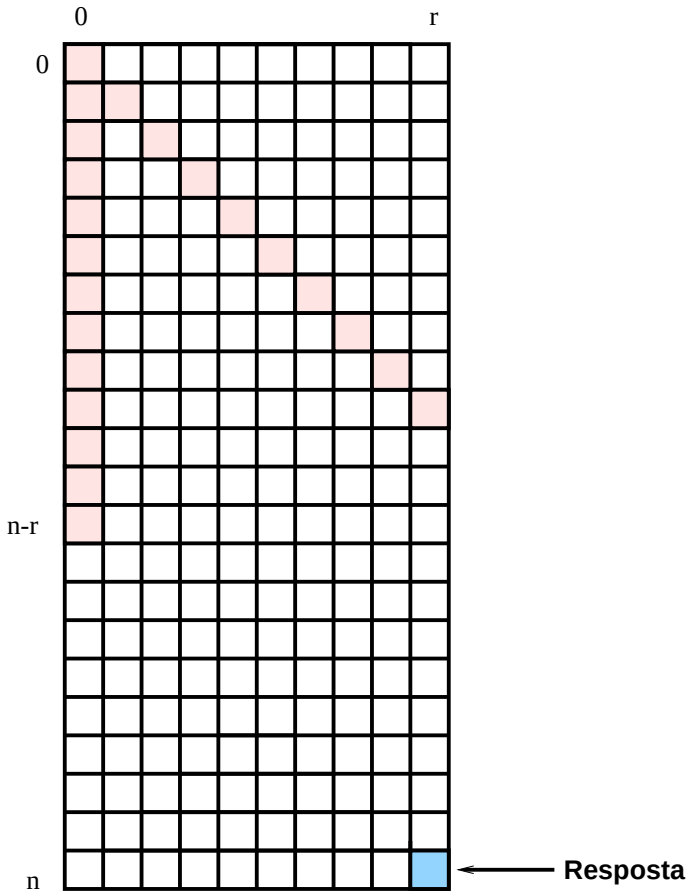
for $i = 0$ to $n - r$ **do**
  $T[i, 0] = 1$
for $i = 0$ to r **do**
  $T[i, i] = 1$
for $j = 1$ to r **do**
  for $i = j + 1$ to $n - r + j$ **do**
    $T[i, j] = T[i-1, j-1] + T[i-1, j]$
return$(T[n, r])$

Este algoritmo é muito melhor. É fácil perceber que a complexidade é de $O(n^2)$.

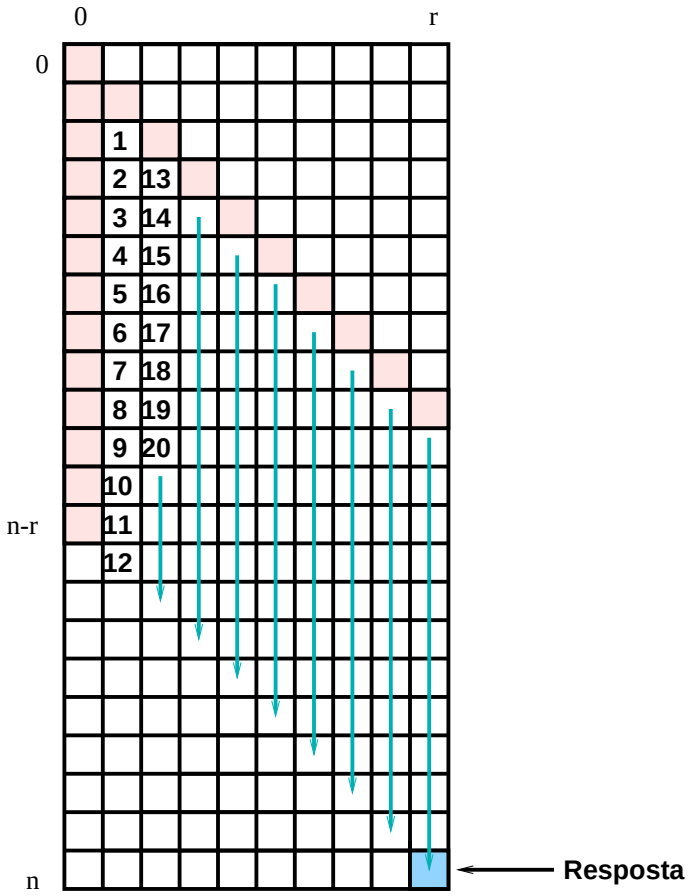
É possível distinguir duas partes no algoritmo: a primeira, estreitamente relacionada com o caso base da solução recursiva, corresponde à inicialização da tabela. A segunda parte define como o restante da tabela deve ser preenchida. Neste segundo caso, este preenchimento está relacionado com as chamadas recursivas no algoritmo recursivo.

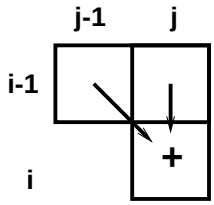
Para facilitar o entendimento, vamos considerar uma tabela $T[n, r]$, em que o valor armazenado em uma célula qualquer $T[i, j]$ indica o número possível de combinações de escolher j itens de um total de i itens.

A Tabela após o preenchimento inicial deve ter o seguinte formato:



O preenchimento da tabela, definido pelo aninhamento de laços, define um padrão a ser seguido.





Caracterização

Quando a estratégia de divisão e conquista gera um grande número de subproblemas idênticos, recursão se torna muito caro. Nesse caso, é melhor armazenar as soluções desses subproblemas em uma tabela. Isso caracteriza a programação dinâmica.

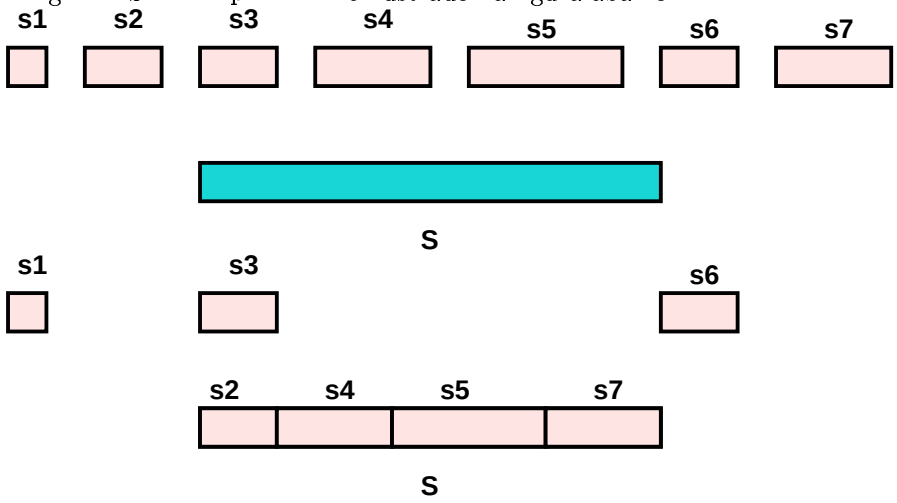
O projeto de um algoritmo baseado em programação dinâmica é dividido em duas partes:

1. Identificação:
   - determinar uma solução por divisão e conquista.
   - Analisar - e verificar que o tempo de execução é exponencial.
   - Mesmo subproblema resolvido várias vezes.
2. Construção:

- Pegar a parte do algoritmo de divisão e conquista que corresponde à parte da conquista e substituir as chamadas recursivas por *olhada* na tabela.
- Em vez de retornar um valor, armazená-lo na tabela.
- Usar o caso base do algoritmo de divisão e conquista para inicializar a tabela.
- Determinar o padrão de preenchimento da tabela.
- Definir um laço que utilizao padrão para preencher os demais valores da tabela.

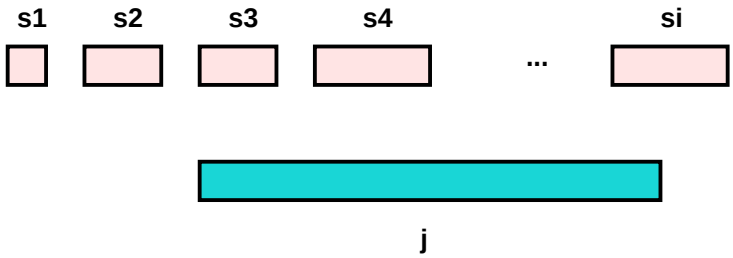
O Exemplo da Mochila Binária

A mochila binária é um exemplo de problema onde a utilização de programação dinâmica permite encontrar a solução ótima. Seja n itens de tamanhos $s_1, s_2, ..., s_n$. Existe um subconjunto desses itens cujo tamanho total é exatamente igual a S? Esse problema é ilustrado na figura abaixo.



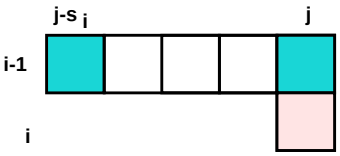
Em uma solução por divisão-e-conquista, sabemos que devemos ter problemas menores da mesma natureza. podemos generalizar para a situação em que temos i itens e o tamanho total é j. Para saber se retornamos verdadeiro, temos que analisar duas possibilidades:

1. O i-ésimo item é usado para completar o tamanho j.
2. O i-ésimo item não é usado e, portanto, j é alcançado até os $i-1$ primeiros itens.



Devemos usar uma tabela $t[n, S]$ para armazenar t, caso seja possível completar S com os n elementos. Caso não seja possível, preenchemos com f.

Pelo que definimos acima, uma célula $t[i, j]$ deve ser preenchida com t se uma das duas situações é verdadeira: $t[i-1, j-s_i]$ ou $T[i-1, j]$. O padrão de preenchimento seria então:



A tabela final teria o seguinte formato: