Construção e Análise de Algoritmos

aula 25: O problema da mochila

1 Introdução

> *Finalmente ele conseguiu entrar na casa, mas então ele não podia acreditar no que estava vendo.*
>
> *Havia tanta coisa ali.*
>
> *Mas a mochila que ele tinha trazido era tão pequena ...*
>
> *E, para piorar as coisas, era preciso agir rápido!*
>
> *Não havia tempo para algoritmos exponenciais, ou mesmo polinomiais muito grandes.*
>
> *A polícia podia chegar a qualquer momento.*

O problema de hoje está bem claro, não é?

O nosso amigo precisa escolher um subconjunto de objetos que caiba na sua mochila.

E, presumivelmente, ele quer levar pra casa (dele) coisas com o maior valor possível.

Nossa tarefa é ajudá-lo a fazer as contas bem rápido (i.e., encontrar um algoritmo eficiente para o problema).

O problema pode ser formalizado da seguinte maneira:

- Dada uma coleção de objetos

$$O \quad = \quad \{o_1,\, o_2,\, o_3,\, \ldots,\, o_n\}$$

  onde cada objeto o_i tem um certo valor v_i e um certo tamanho ou peso p_i, o problema consiste em encontrar um subconjunto de objetos S tal que

  1. a soma dos *pesos* dos objetos em S seja menor ou igual a *capacidade* C da mochila
  2. a soma dos *valores* dos objetos em S seja a maior possível.

Vejamos um exemplo simples.

Suponha que a mochila tem capacidade $C = 10$, e que temos a seguinte coleção de objetos

	o_1	o_2	o_3	o_4	o_5	o_6	o_7	o_8
valor	40	50	30	20	15	55	20	10
peso	5	4	6	3	2	8	2	3

Note que escolher os objetos de menor peso primeiro (gulosamente), para colocar o máximo de coisas na mochila, nos dá a solução

$$S_1 \quad = \quad \{o_4, o_5, o_7, o_8\} \quad \Rightarrow \quad \textsf{valor total} \quad = \quad 65$$

que não é a melhor possível.

E escolher os objetos de maior valor primeiro (gulosamente) nos dá a solução

$$S_2 \quad = \quad \{o_5, o_6\} \quad \Rightarrow \quad \textsf{valor total} \quad = \quad 70$$

que também não é a melhor possível.

A ideia de escolher objetos não muito pesados com um valor não muito baixo nos dá a solução

$$S_3 \quad = \quad \{o_1, o_2\} \quad \Rightarrow \quad \textsf{valor total} \quad = \quad 90$$

mas, será que isso é o melhor possível?

2 Estratégias de decomposição

A ideia é resolver o problema por programação dinâmica.

E nós já sabemos que a chave da programação dinâmica é encontrar uma maneira de decompor o problema.

O esquema abaixo ilustra as duas estratégias que nós temos utilizado para realizar essa tarefa

Alguns problemas podem ser decompostos das duas maneiras.

O problema da seleção de atividades, e os dois problemas sobre sequências da aula 22, são exemplos desse tipo.

Já os dois problemas sobre árvores que vimos na aula passada foram resolvidos utilizando a primeira estratégia.

Hoje nós estamos na situação inversa.

Quer dizer, não existe uma maneira evidente decompor o problema da mochila utilizando a primeira estratégia.

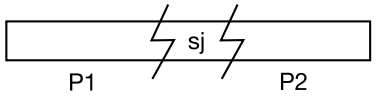
Mas, antes de apresentar a solução baseada na segunda estratégia, é útil entender porque isso é o caso.

De fato, a primeira estratégia funcionou bem em todos os problemas até agora (apesar de em geral não nos dar o algoritmo mais eficiente).

Mas, havia uma coisa em comum em todos esses problemas: uma relação de ordem natural estruturando os elementos do problema:

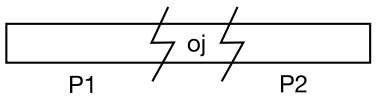
- no problema da seleção de atividades (aula 21), as atividades estavam distribuídas ao longo da linha do tempo
- nos dois problemas sobre sequências (aula 22), a própria noção de sequência estrutura as letras ou símbolos de maneira linear
- no problema da cadeia de multiplicação de matrizes (aula 23), as matrizes também aparecem em uma certa ordem (que não pode ser alterada)
- no problema da árvore binária de busca ótima (aula 23), as chaves dos registros também possuem uma relação de ordem.

Quando o problema apresenta esse tipo de estrutura linear, a seleção de um elemento intermediário qualquer em geral decompõe o problema em dois subproblemas P_1 e P_2 independentes (ou quase independentes).



A questão é que não existe nenhuma relação de ordem natural entre os objetos do problema da mochila.

E a introdução de uma relação de ordem artificial tampouco ajuda, pois se decompomos o problema de acordo com essa ordem



os subproblemas P_1 e P_2, cada um com a sua subcoleção de objetos, não têm muito significado.

Quer dizer, não é claro como combinar as soluções de P_1 e P_2 para formar uma solução para o problema original.

(Você consegue pensar em alguma coisa?)

3 Solução do problema

Agora, considere a segunda estratégia de decomposição.

Esse esquema nos leva a pensar da seguinte maneira: "*E se nós colocarmos o objeto* o_1 *na mochila, o que acontece?*"

Bom, nesse caso nós acumulamos o valor v_1 e ocupamos uma porção p_1 da mochila.

Agora, nos resta ainda a capacidade $C - p_1$, e nós precisamos encontrar a melhor maneira possível de ocupá-la com os objetos $o_2, \ldots, o_n$.

Desta vez, nós temos um subproblema bem definido.

Calculando a solução desse subproblema e adicionando o objeto o_1, nós obtemos uma solução para o problema original.

Certo, mas ninguém garante que essa seja a solução ótima.

Quer dizer, pode ser que o objeto o_1 não faça parte da solução ótima.

Mas, nesse caso, a solução ótima pode ser encontrada resolvendo o subproblema onde nós ainda temos a capacidade total C da mochila, mas apenas os objetos $o_2, \ldots, o_n$.

Isto é, a ideia é calcular as duas soluções e ficar com a melhor delas.

Abaixo nós temos o pseudo-código que implementa essa ideia.

```
Procedimento  Prob-Moch-PD ( O = {oi,...,on}, C: capacidade da mochila )
 {
    Se ( O está vazio  ou  C = 0 )    Retorna ( vazio, 0 )

    (S1,V1)  <--  Prob-Moch-PD ( {o_i+1, ..., on}, C - pi )
         S1  <--  S1 + {oi}
         V1  <--  V1 + vi

    (S2,V2)  <--  Prob-Moch-PD ( {o_i+1, ..., on}, C )

    Se ( V1 > V2 )   Retorna (S1,V1)
    Senão            Retorna (S2,V2)
 }
```

Análise de complexidade

O primeiro passo da análise consiste em estimar o número de chamadas recursivas realizadas pelo algoritmo.

Note que cada chamada recursiva está associada a um subproblema que é definido por 2 parâmetros:

uma subcoleção de problema $\{o_i, \ldots, o_n\}$

- uma capacidade (residual) da mochila C

É fácil ver que existem n possibilidades para o índice i no primeiro parâmetro, de modo que existem $O(n)$ subcoleções de objetos dessa forma.

Com relação ao seguinte parâmetro, se assumirmos que todos os pesos p_i (e a capacidade inicial) são números inteiros, então todas as capacidades parciais são números inteiros também.

Ora, mas só existem C números inteiros entre 0 e C.

Portanto, essa é a quantidade de maneiras como o segundo parâmetro pode variar.

Juntando as duas coisas, nós temos que existem $O(nC)$ subproblemas diferentes.

Finalmente, observando que todas as instruções do procedimento `Prob-Moch-PD` executam em tempo $O(1)$, nós concluímos que o algoritmo executa em tempo

$$O(nC) \quad \times \quad O(1) \quad = \quad O(nC)$$

4 Discussão

O resultado que nós obtivemos acima parece bom, não é?

Um algoritmo que executa em tempo $O(nC)$.

Mas, examinando as coisas em detalhe

- nós vamos descobrir que isso pode ser terrível (em alguns casos)
- e vamos aprender o verdadeiro truque da programação dinâmica

As coisas estão começando a parecer fáceis demais, mas é preciso cuidado com a programação dinâmica.

Quer dizer, praticamente todo problema de otimização pode ser resolvido com um raciocínio do tipo

- *Bom, o elemento e_1 do problema pode estar ou não na solução. Então,*
  - *primeiro eu assumo que e_1 está na solução, e resolvo o subproblema que aparece (recursivamente)*
  - *depois assumo que ele não está, e resolvo o subproblema que aparece (recursivamente)*
  - *e, no final das contas, eu escolho a melhor solução das duas*

O problema é que isso quase sempre nos dá um algoritmo exponencial.

Mas, com a programação dinâmica não dá — isto é, a coisa fica polinomial.

Porque?

Vejamos um exemplo concreto do problema da mochila.

Suponha que a capacidade da mochila é $C = 2$, e que todos os n objetos tem peso igual a 1 — os valores não importam, para esse raciocínio.

É fácil ver que, nesse caso, qualquer solução interessante consiste em ocupar a mochila com 2 objetos.

Pode-se imaginar, então, que o nosso algoritmo de programação dinâmica irá examinar todos os subconjuntos de objetos de tamanho 2, e selecionar aquele que tem o maior valor agregado.

Mas, não é isso o que acontece.

Até porque isso levaria tempo $O(n^2)$, enquanto que o nosso algoritmo executa em tempo $O(2n)$ nesse exemplo.

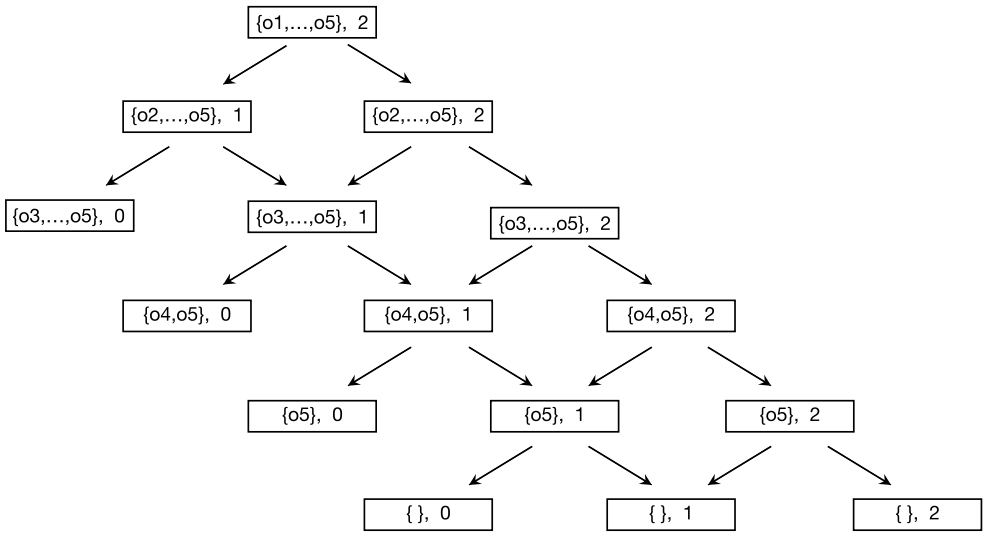
Mas, o que está acontecendo então?

Bom, é só examinar a árvore de chamadas recursivas.

Cada chamada recursiva do nosso algoritmo é caracterizada por:

1. um subconjunto de objetos da forma $\{o_i, \ldots, o_n\}$;
2. uma capacidade residual C.

Quando $n = 5$, nós temos a seguinte árvore de recursão



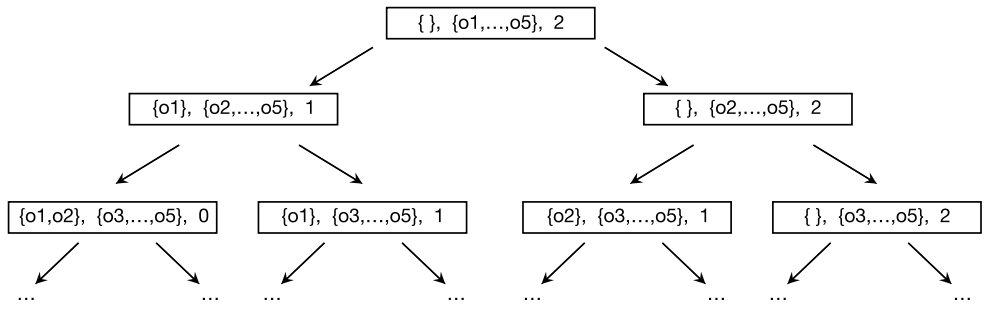
(onde as chamadas repetidas foram representadas apenas uma vez, devido ao recurso da memoização)

Agora, considere o algoritmo exponencial que examina todas as soluções possíveis.

Suas chamadas recursivas são caracterizadas por:

1. a solução parcial que ele tem no momento;
2. um subconjunto de objetos $\{o_i, \ldots, o_n\}$;
3. uma capacidade residual C.

E a sua árvore de recursão tem a seguinte forma:



Você já viu o que está acontecendo?

Bom, existem realmente duas coisas.

A primeira delas é a seguinte observação:

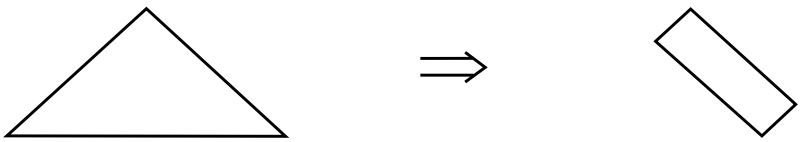
- a solução parcial, que o algoritmo exponencial carrega com ele ao longo da recursão, não ajuda em nada a resolver o subproblema que ele tem no momento

Isso significa que nós podemos eliminar esse parâmetro da chamada recursiva — ou *esquecer o que fizemos antes de chegar ali* (o que é o primeiro truque da programação dinâmica).

Fazendo isso, as chamadas repetidas aparecem outra vez.

E o segundo truque da programação dinâmica consiste em realizar cada chamada apenas uma vez — i.e., eliminar as repetições.

Ao fazer isso, nós transformamos uma árvore exponencial em uma árvore polinomial



Não é legal!

Mas, ainda nos resta a observação de que o desempenho do algoritmo que encontramos para o problema da mochila pode ser terrível (em alguns casos).

Porque acontece isso?

A resposta é: porque o primeiro truque pode falhar ...

Quer dizer, as chamadas recursivas do procedimento `Prob-Moch-PD` podem lembrar "*sem querer*" o que aconteceu antes de se chegar até ali.

Vejamos um exemplo concreto.

Suponha desta vez que a capacidade da mochila é igual $C = 2^n$, e que cada objeto o_i tem peso $p_i = 2^{i-1}$.

O que tem de especial nessa configuração é que todos os subconjuntos de objetos possuem um peso total diferente.

Por exemplo,

$$\mathsf{Peso}(o_1, o_2, o_3) \;=\; 7 \qquad \mathsf{Peso}(o_1, o_3, o_5) \;=\; 24 \qquad \mathsf{Peso}(o_2, o_4) \;=\; 10$$

Porque acontece isso?

Bom, é fácil de ver quando a gente examina os números em binário:

- $p_1 \;=\; \texttt{1}$
- $p_2 \;=\; \texttt{10}$
- $p_3 \;=\; \texttt{100}$
- $p_4 \;=\; \texttt{1000}$
- e assim por diante

Quando somamos os pesos de um subconjunto de objetos qualquer

```
          1 0
      1 0 0 0 0
  1 0 0 0 0 0 0
-----------------
  1 0 1 0 0 1 0
```

nós sempre obtemos uma sequência de `0`'s e `1`'s diferente.

Em particular, isso significa que se alguém nos diz qual é o peso total de um subconjunto, então nós já sabemos quais são os objetos que ele contém.

E isso significa também que, se alguém nos diz a capacidade residual da mochila, então nós conseguimos descobrir quais são os objetos que foram colocados lá dentro.

Em outras palavras, ao receber o parâmetro C que indica a capacidade residual da mochila, o procedimento `Prob-Moch-PD` está recebendo um histórico (implícito) de tudo o que aconteceu até se chegar até ali.

E, por causa disso, não haverão chamadas recursivas repetidas que podem ser eliminadas.

Isto é, o segundo truque falha também, e o tempo de execução do algoritmo é exponencial em n.

É preciso tomar muito cuidado com a programação dinâmica ...