14 – Projeto de Algoritmos: paradigmas

SCC201/501 - Introdução à Ciência de Computação II

Prof. Moacir Ponti Jr.

www.icmc.usp.br/~moacir

Instituto de Ciências Matemáticas e de Computação – USP

contém material extraído e adaptado das notas de aula dos Profs. Antonio Loureiro e Nivio Ziviani

2010/2

Sumário

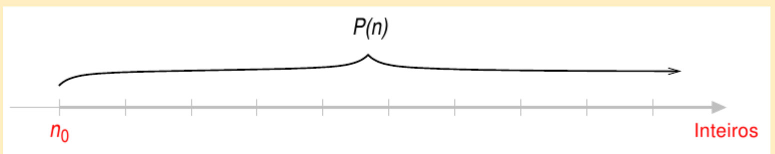
1. Indução
2. Recursividade
3. Tentativa e Erro
4. Divisão e Conquista
5. Algoritmos Gulosos
6. Programação Dinâmica
7. Algoritmos Aproximados

Indução matemática

- Útil para provar asserções sobre a correção e a eficiência de algoritmos
- Consiste em inferir uma lei geral a partir de instâncias particulares

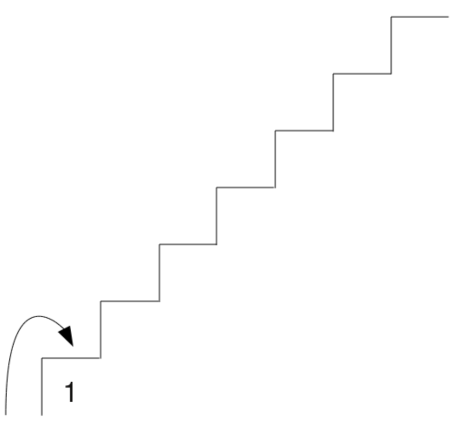
Princípio da indução matemática

- Seja $P(n)$ uma afirmação acerca dos números inteiros n, e seja n_0 um inteiro fixo (escalar).
- A prova de P por indução supõe que sejam verdadeiras:
  1. $P(n_0)$ é V.
  2. Para todo inteiro $k \geq n_0$, se $P(k)$ é V, então $P(k+1)$ é V.
- Logo, a afirmação $P(n)$ é V para todos os inteiros $n \geq n_0$.



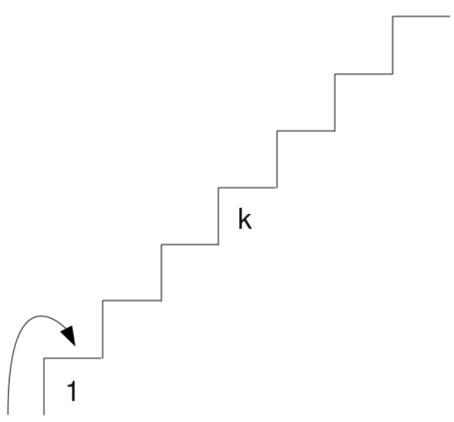
Indução matemática

- A idéia da prova por indução é primeiramente verificar a proposição para um caso trivial, chamado de caso base.



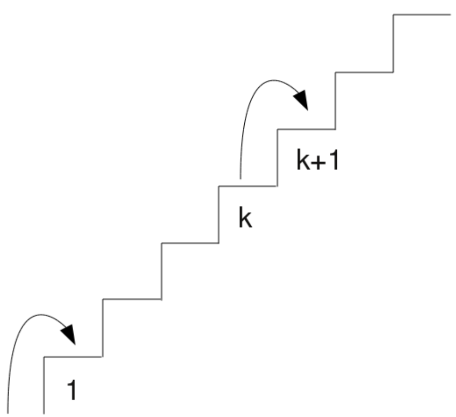
Indução matemática

- No passo indutivo, assumimos que a proposição é verdadeira para um dado k e tentamos provar para $k+1$



Indução matemática

- Se for verdadeira para $k+1$, provamos por indução que se assumirmos que $P(k)$ é verdadeiro, então $P(k+1)$ também é verdadeiro.



Indução matemática: exemplo

- Prove que, para todos inteiros $n \geq 1$, a soma abaixo é verdadeira:

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Prova por indução

1. Passo base: $P(n_0) = P(1)$: para $n_0 = 1$, $1 = \frac{1(1+1)}{2} = 1$, V.
2. Passo indutivo: se é V para $n = k$ então deve ser para $n = k + 1$, ou seja, $P(k) \rightarrow P(k+1)$.
   - Suponha que a fórmula é verdadeira para $n = k$ para algum inteiro $k \geq 1$ (hipótese indutiva), ou seja:

$$P(k) : 1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

Indução matemática: exemplo

- Devemos mostrar que

$$P(k+1) : 1 + 2 + \cdots + (k+1) = \frac{(k+1)(k+2)}{2}$$

- Pela hipótese indutiva temos:

$$\begin{aligned} 1 + 2 + \cdots + k + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\ &= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} \\ &= \frac{k^2 + 3k + 2}{2} \\ &= \frac{(k+1)(k+2)}{2} \end{aligned}$$

(que era o que devia ser provado)

Indução matemática e algoritmos

- Seja T um teorema que tenha como parâmetro um número natural n. Para provar que o T é válido para todo n, provamos que:

1. T é válido para $n = 1$; *(passo base)*
2. Para todo $n > 1$, *(passo indutivo)*
   se T é válido para n,
   então T é válido para $n + 1$,

- As condições 1 e 2 implicam T válido para $n = 2$, o que, junto com a condição 2 implica que é válido também para $n = 3$ e assim por diante.

Invariantes para laços

- A corretude de laços em um algoritmo ou programa pode ser provada por indução.
- Em geral essa prova envolve o conceito de invariante de laço.

Algoritmo (1): cálculo do quadrado de um número

```
int square(int n) {
   int S = 0, i = 0;
   while (i < n) {
      S = S + n;
      i++;
   }
   return S;
}
```

- Para provar a corretude desse algoritmo primeiro temos que garantir que ele termina.
  - Como `i` é incrementado de um em um, a partir de zero, eventualmente será igual a `n` e portanto o laço termina.
- Precisamos então provar que o algoritmo computa n^2

Algoritmo (1): cálculo do quadrado de um número

```
int square(int n) {
   int S = 0, i = 0;
   while (i < n) {
      S = S + n;
      i++;
   }
   return S;
}
```

- **Prova por indução**: mostraremos que as seguintes proposições são invariantes, após o algoritmo executar o loop k vezes:
  - $S = k \cdot n$, e
  - $i = k$
- Passo base: $k = 0$. Ocorre quando o algoritmo ainda não entrou no laço de repetição, e portanto $S = 0$, $i = 0$. A invariante é verdadeira pois:
  - $S = 0 \cdot n = 0$, e
  - $i = 0$

Invariantes para laços

Algoritmo (1): cálculo do quadrado de um número

```
int square(int n) {
   int S = 0, i = 0;
   while (i < n) {
      S = S + n;
      i++;
   }
   return S;
}
```

- Passo indutivo: assuma que para um valor arbitrário m de k, $S = m \cdot n$ e $i = m$ quando o laço for percorrido m vezes.
  - Provaremos que a invariante permanece para quando o loop for percorrido $m + 1$ vezes. Dentro do laço teremos:
    - $S = (m * n) + n$, e
    - $i = i + 1$;
  - produzindo, conforme queriamos provar:
    - $S = (m + 1) \cdot n$, e
    - $i = m + 1$;
- Observe que, quando o laço é terminado, $i = n$, e portanto o loop foi percorrido n vezes, e assim:

  $$S = n \cdot n = n^2$$

  e o algoritmo está correto

Limite superior de relações de recorrência

- A solução direta de uma recorrência pode ser difícil de obter
- Nesses casos tentar adivinhar a solução e depois verificá-la pode ser mais fácil
- Obter um limite superior para a ordem de complexidade também pode ser útil quando não estamos interessados na solução exata
  - mostrar que um certo limite existe é mais fácil do que obter **o** limite
- Exemplo:

$$T(2n) \leq 2T(n) + 2n - 1,$$
$$T(2) = 1,$$

definida para valores de n que são potências de 2.

- O objetivo é encontrar um limite superior na notação O, onde o lado direito da desigualdade representa o pior caso.

Limite superior de relações de recorrência: exemplo

$$T(2n) \leq 2T(n) + 2n - 1,$$
$$T(2) = 1,$$

definida para valores de n que são potências de 2.

- Procuramos uma função $f(n)$ tal que $T(n) \in O(f(n))$.
- considere o palpite $f(n) = n^2$.
- queremos provar que $T(n) \leq f(n) \in O(f(n))$ utilizando indução matemática em n.

Limite superior de relações de recorrência: exemplo

Provar por indução que $T(n) \leq f(n) \in O(f(n))$, para $f(n) = n^2$, sendo

$$T(2n) \leq 2T(n) + 2n - 1,$$
$$T(2) = 1,$$

definida para valores de n que são potências de 2.

Prova por indução:

1. Passo base: $T(n_0) = T(2)$: para $n_0 = 2$, $T(2) = 1 \leq f(2) = 4$, **V**.
2. Passo indutivo: se a recorrência é **V** para n então deve ser para $2n$, ou seja, $T(n) \rightarrow T(2n)$ (n é potência de 2 e portanto o número depois de n é $2n$).

   Reescrevendo o passo indutivo:

$$P(n) \rightarrow P(2n)$$
$$[T(n) \leq f(n)] \rightarrow [T(2n) \leq f(2n)]$$

$$\begin{aligned}
T(2n) &\leq 2T(n) + 2n - 1 && \text{(definição da recorrência)} \\
&\leq 2n^2 + 2n - 1 && \text{(pela hipótese indutiva, } T(n) = n^2\text{)} \\
&\leq 2n^2 + 2n - 1 <^? (2n)^2 && \text{(a conclusão é verdadeira?)} \\
&\leq 2n^2 + 2n - 1 < 4n^2 && \text{(sim e, logo, } T(n) \text{ é } O(n^2)\text{)}
\end{aligned}$$

Indução matemática: comentários finais

- Técnica matemática muito útil para provar asserções sobre a correção e eficiência de algoritmos.
- Pode ser usada para identificar (e verificar) invariantes de laços.
- Pode ser usada para encontrar um limite superior para uma equação de recorrência.

Recursividade

- Um procedimento que possui uma chamada a si mesmo (direta ou indiretamente) é dito **recursivo**.
- A recursividade fornece uma descrição clara e natural para muitos algoritmos:
  - árvore binária de busca
  - *heap*
  - ordenação por quicksort
  - busca binária

Recursividade: implementação

Pilha

- Uma **pilha** é usada para armazenar os dados usados em cada chamada de um procedimento que ainda não terminou.
- Todos as variáveis locais são alocadas na pilha registrando o estado atual da computação dentro da instância do procedimento.
  - o algoritmo termina quando a primeira instância criada é desempilhada.

Caso base

- O problema da terminação deve ser considerado, sendo o **caso base** bem definido e estudado para não cair em recursão infinita.
- A chamada recursiva é, portanto, sujeita a uma determinada **condição**, que deve se tornar obrigatoriamente falsa eventualmente.

Recursividade: implementação

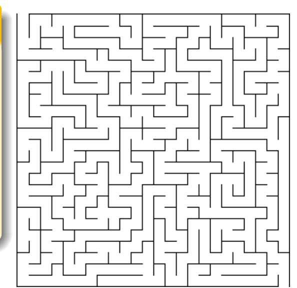
Memória

- É importante observar o comportamento da pilha (ou, de forma equivalente, a altura da árvore de recursão), evitando que muitos recursos de memória sejam utilizados.

Tentativa e erro (backtracking)

Explorando o espaço de soluções

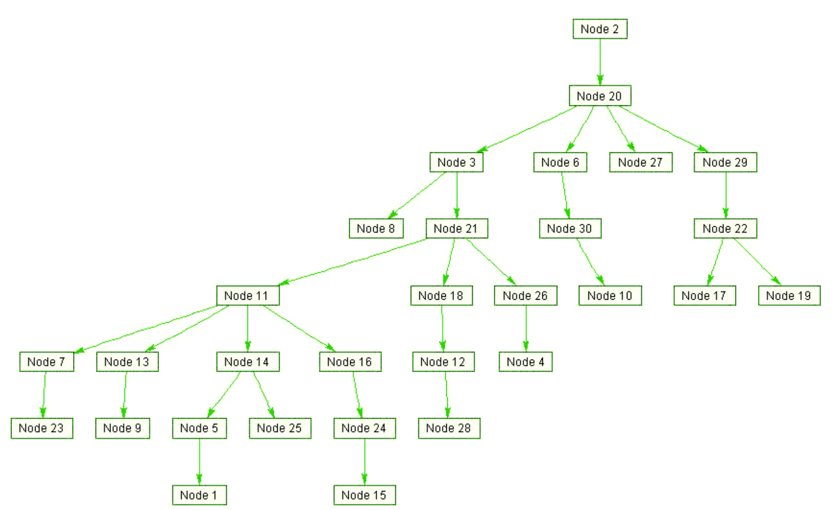
- decompõe o processo em um número finito de sub-tarefas parciais a serem exploradas exaustivamente
- gradualmente constrói e percorre uma "árvore" de soluções



Não há uma regra fixa

- passos em direção à solução final são executados e registrados
- se os passos não levarem à solução final, podem ser retirados e apagados do espaço de soluções

Tentativa e erro



Tentativa e erro (backtracking)

- Uma abordagem é **exaurir** as possibilidades de solução
- Há ainda algoritmos que inserem componentes aleatórios para chegar à solução
  - Algoritmo de ordenação Bozo-sort (ou Bogo-sort ou Vai-na-sort)
  - Algoritmos genéticos

- Quando a pesquisa pela solução exata é inviável, o uso de **algoritmos aproximados** ou **heurísticas** é necessário para viabilizar a obtenção de uma solução mais rápida (porém, sem garantia de solução ótima)

Tentativa e erro: passeio do cavalo

- Num tabuleiro de tamanho $n \times n$, considere uma peça cavalo.
- Problema: partindo de uma posição (x_0, y_0), encontrar, se existir, um caminho pelo o qual o cavalo possa visitar todos as casas do tabuleiro uma única vez.

```
01. repita
02.     seleciona próximo movimento candidato
03.     se (aceitavel)
04.        registra movimento
05.        se (tabuleiro não está cheio)
06.           tenta novo movimento [recursivamente]
07.           se (movimento inválido)
08.              apaga registro anterior
09.  enquanto !((movimento bem sucedido) ou
             (nao haja mais candidatos a movimento))
```

Tentativa e erro: passeio do cavalo

- O tabuleiro pode ser representado por uma matriz `n x n`
- A situação de cada posição pode ser um inteiro com o histórico das ocupações
  - `T[x,y]=0` – casa não visitada
  - `T[x,y]=i` – casa visitada no i-ésimo movimento $(1 \leq i \leq n^2)$.

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

Tentativa e erro (backtracking): comentários finais

- Abordagem quando não se sabe exatamente qual caminho seguir para encontrar solução
- Em geral não garante solução ótima
- Pode ser vista como uma variante da recursividade (*backtracking*)
- Deve-se analisar o crescimento do espaço de soluções

- Quando a pesquisa pela solução exata é inviável, o uso de **algoritmos aproximados** ou **heurísticas** é necessário para viabilizar a obtenção de uma solução mais rápida (porém, sem garantia de solução ótima)

Divisão e Conquista

Procedimento básico

1. Dividir o problema em partes menores
2. Resolver o problema para essas partes (supostamente mais fácil, ou até trivial)
3. Combinar em uma solução global

- Geralmente leva a soluções eficientes e elegantes, muitas vezes de natureza recursiva
- Está normalmente relacionado a uma equação de recorrência que contém termos referentes ao próprio problema

Divisão e Conquista

Recorrência

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

onde:

- a indica o número de sub-problemas gerados
- b o tamanho de cada um dos problemas
- $f(n)$ o custo de resolver cada sub-problema

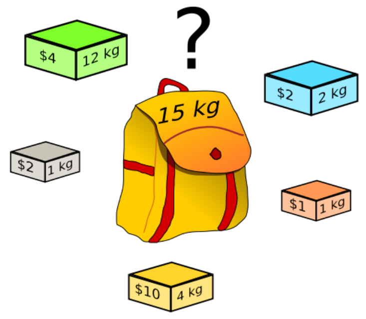
Exemplos

- Ordenação: mergesort, quicksort
- Busca: busca binária

Divisão e Conquista

- Não é aplicado unicamente a problemas recursivos
- Em geral as estratégias de resolução de problemas envolvem um dos três cenários abaixo:
  1. Processar independentemente partes do conjunto de dados
     - Ex: mergesort
  2. Eliminar partes do conjunto de dados a serem processados
     - Ex: pesquisa binária
  3. Processar separadamente partes do conjunto, mas no qual a solução de uma parte influencia no resultado da outra
     - Ex: somador de bits

O problema da mochila



Quais caixas escolher de forma a maximizar o valor total, obedecendo a restrição de um máximo de 15 kg?

por: Dake (Creative Commons non-commercial 2.5 License)

O problema da mochila: 0-1 / binário

- Dado um conjunto de n itens dos quais queremos selecionar alguns para serem levados em uma mochila.
- Cada item possui um **peso** e um **valor**.
- O objetivo é escolher um subconjunto de itens que caibam na mochila, **maximizando** o valor total.

O problema da mochila: 0-1 / binário

Sejam

- w_i o peso do item i,
- p_i o valor do item i,
- $x_i = 1$ se o item i está na mochila e $x_i = 0$ se não está na mochila,
- C a capacidade máxima da mochila.

Objetivo

Maximizar

$$\sum_{i=1}^{n} p_i x_i$$

Sujeito à restrição

$$\sum_{i=1}^{n} w_i x_i \leq C$$

O problema da mochila: 0-1 / binário

- É um problema de otimização, com natureza combinatória.
- Se tivermos n objetos possíveis, o número de possíveis soluções será 2^n (problema binário em x).
- É possível resolver esse problema por algum método já visto?
  - Tentativa e erro — força bruta.

- Vamos explorar melhores paradigmas para problemas desse tipo.

Sumário

1. Indução
2. Recursividade
3. Tentativa e Erro
4. Divisão e Conquista
5. Algoritmos Gulosos
6. Programação Dinâmica
7. Algoritmos Aproximados

Algoritmos Gulosos

- Resolve o problema a partir da idéia de escolher a estratégia ótima **local** supondo que esta leve à solução ótima **global**.

- Em outras palavras, realiza a escolha que parece ser a melhor no momento.
- A estratégia para a escolha do próximo passo pode ser uma **heurística**.
- Independente do que aconteça posteriormente, nunca reconsidera a decisão passada.
- Não necessita avaliar alternativas, ou usar procedimentos sofisticados para desfazer decisões prévias.

- Exemplo: caminho mais curto.

Bases

Em geral os algoritmos gulosos partem das seguintes bases:

1. Um conjunto ou lista de candidatos a partir do qual uma solução pode ser criada,
2. Uma função de viabilidade, determina se o candidato pode ser usado para compor a solução (sem considerar se esta é ótima),
3. Uma função de seleção escolhe o melhor candidato num dado instante para ser adicionado à solução,
4. Uma função objetivo, que atribui um valor à solução (ou a uma solução parcial),
5. Uma função de solução, que indica se a solução completa foi alcançada.

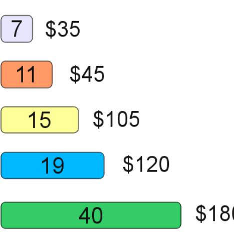
Algoritmos Gulosos

- A função de seleção está em geral relacionada com a função objetivo.
- Se o objetivo é:
  - Maximizar: provavelmente escolherá o candidato restante que proporcione o maior ganho individual.
  - Minimizar: o escolhido é aquele que adicione o menor custo à solução.

- Um candidato escolhido e adicionado à solução se torna permanente.
- Um candidato excluído não é mais reconsiderado.

- Exemplo: problema da mochila.
  1. *greedy by profit*: escolhe primeiro os itens de maior valor,
  2. *greedy by weight*: escolhe primeiro os itens de menor peso,
  3. *greedy by profit density*: escolhe primeiro os itens de maior densidade de valor, p_i/w_i, maximiza valor ao escolher itens com maior valor por unidade de peso.

Estratégia gulosa para o problema da mochila



item	Peso (w_i)	Valor (p_i)	D (p_i/w_i)
1	40	180	4.5
2	19	120	6.3
3	15	105	7
4	11	45	4.1
5	7	35	5

Soluções possíveis:

Modo	Soma dos valores
Valor	$(1) + (5) = 180 + 35 = 215$
Peso	$(5) + (4) + (3) = 35 + 45 + 105 = 185$
Densid.	$(3) + (2) + (5) = 105 + 120 + 35 = 260$
Ótima	$(2) + (3) + (4) = 120 + 105 + 45 = 270$

Sumário

1. Indução
2. Recursividade
3. Tentativa e Erro
4. Divisão e Conquista
5. Algoritmos Gulosos
6. Programação Dinâmica
7. Algoritmos Aproximados

Programação Dinâmica

- "Programação" nesse caso não está relacionado com programa de computador, mas com método de solução baseado em tabela.
- Programação dinâmica $\times$ Divisão e conquista
  1. Divisão e conquista particiona o problema em sub-problemas menores.
  2. Programação dinâmica resolve os sub-problemas, partindo dos menores para os maiores, armazenando os resultados em uma tabela: a seguir, somente reusa as soluções ótimas.

Programação Dinâmica

Princípio da otimalidade

- Em uma sequência ótima de escolhas (ou decisões), cada subsequência deve também ser ótima.
- Cada subsequência representa o custo mínimo assim como m_{ij}, $j > i$.
- Assim, todos os valores da tabela representam escolhas ótimas

Exemplo de aplicação do princípio da otimalidade

- suponha que o caminho mais curto entre São Carlos e Curitiba passa por Campinas; logo,
- o caminho entre São Carlos e Campinas também é o mais curto possível, como tambem é o caminho entre Campinas e Curitiba.
- Assim, o princípio da otimalidade se aplica.

Exemplo de não aplicação do princípio da otimalidade

- Seja o problema encontrar o caminho **simples** mais longo entre duas cidades:
  - um caminho simples nunca visita uma mesma cidade duas vezes
  - se o caminho mais longo entre São Carlos e Curitiba passa por Campinas, isso não significa que a solução possa ser obtida tomando: o caminho simples mais longo de São Carlos a Campinas e depois o de Campinas a Curitiba.
- Quando os dois caminhos simples são agrupados não existe uma garantia de que o caminho resultante também seja simples
- Logo o princípio da otimalidade não se aplica.

Programação Dinâmica

Quando usar

- Problema deve ter formulação recursiva
- Não deve haver ciclos na formulação
- Número total de instâncias do problema (n) deve ser pequeno
- **Subestrutura ótima**
  - solução ótima para o problema contém soluções ótimas para os subproblemas
- **Sobreposição de problemas**
  - número total de subproblemas distintos é pequeno comparado com o tempo de execução recursivo

Programação Dinâmica: um exemplo de uso de tabela

```
unsigned long fibmemo(unsigned long n, unsigned long *memo){
  if (memo[n]==0) // se a resposta ainda nao foi calculada, calcular
    memo[n] = fibmemo(n-1,memo) + fibmemo(n-2,memo);
  return memo[n];
}

unsigned long fibfast(unsigned long n){
  unsigned long *M, F;
  if (n <= 1) return n;
  M = (unsigned long *) calloc(n--, sizeof(unsigned long));
  M[0] = 1; M[1] = 1;

  if (M != NULL) { F = fibmemo(n,M); }
  else {
    printf("Erro de alocação\n");
    exit(EXIT_FAILURE);
  }
  free(M);
  return F;
}
```

Programação Dinâmica

- A implementação de Fibonacci anterior é apenas ilustrativa, visto que sua forma mais eficiente é a iterativa, obtendo o resultado de forma *bottom-up*, ou seja, do menor valor para o maior.
- No entanto, a implementação fornece uma idéia geral da solução de problemas utilizando memorização e armazenamento de soluções parciais em tabelas.

Programação dinâmica no problema da mochila

- Montar uma árvore de decisão baseada no problema
- Assuma o seguinte problema da mochila:
  - $n = 3$
  - $w = \{4, 3, 2\}$
  - $p = \{9, 7, 8\}$
  - $C = 5$

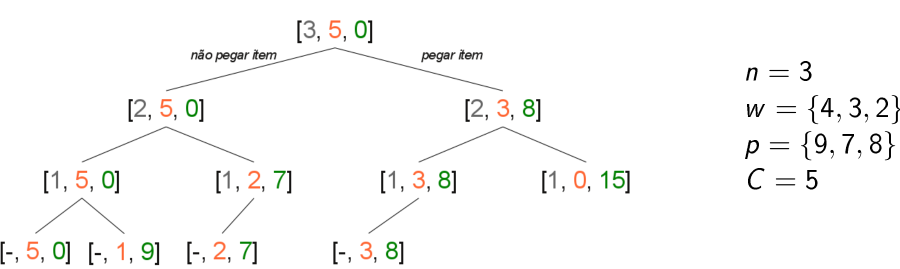
Operações possíveis

- Para cada item, decidiremos se iremos pegá-lo ou não.
- Por facilidade de implementação, começamos pelo último item da lista.
- Cada nó é uma tupla contendo: i) o índice do elemento observado, ii) a capacidade ainda disponível na mochila, e iii) o valor total atual na mochila.

Programação dinâmica no problema da mochila

Montagem da árvore de decisão com backtracking

- Será construída como em um percurso pré-ordem
- Cada nó da árvore é composto por:
  - índice do elemento sendo observado, capacidade disponível, e valor total



$n = 3$
$w = \{4, 3, 2\}$
$p = \{9, 7, 8\}$
$C = 5$

Programação dinâmica no problema da mochila

$n = 3$
$p = \{9, 7, 8\}$
$w = \{4, 3, 2\}$
$C = 5$

V	1	2	3	4	5
0					
1					
2					
3					

keep	1	2	3	4	5
0					
1					
2					
3					

- A tabela `V` armazena o valor máximo possível para diferentes instâncias do problema da mochila.
  - As linhas representam o problema da mochila considerando 0 a 3 itens.
  - As colunas representam o problema considerando uma mochila com capacidade de 1 a 5 unidades.
  - Cada célula (lin, col) representa o problema da mochila considerando os itens de 1 a *lin* numa mochila com capacidade *col*.
    - Por exemplo: a célula $(2, 4)$ representa o problema da mochila considerando os itens 1 e 2 numa mochila de capacidade 4.
- A tabela `keep` armazena 1 se desejamos manter o objeto *lin* ou 0 se não desejamos incluir o objeto.

Programação dinâmica no problema da mochila

- As tabelas serão preenchidas da esquerda para a direita e de cima para baixo.
- Após calcular uma instância do problema, esta é reutilizada para resolver instâncias mais complexas.

$n = 3$
$p = \{9, 7, 8\}$
$w = \{4, 3, 2\}$
$C = 5$

V	*1*	*2*	*3*	*4*	*5*
0	0	0	0	0	0
1	0	0	0	9	9
2	0	0	7	9	9
3	0	8	8	9	15

keep	*1*	*2*	*3*	*4*	*5*
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	1	1	0	1

- A combinação final dos elementos será obtida percorrendo a tabela `keep`.

Programação dinâmica no problema da mochila

keep	*1*	*2*	*3*	*4*	*5*
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	1	1	0	1

- A tabela `keep` é percorrida considerando do último elemento até o primeiro, e armazenando numa variável, `C`, a capacidade restante na mochila.
  1. `item = 3, C = 5`: se `keep(3,5) = 1`, incluir item 3.
     - como o item 3 foi incluído e tinha peso 2, agora a capacidade será 3.
  2. `item = 2, C = 3`: se `keep(2,3) = 1`, incluir item 2.
     - como o item 2 foi incluído eesse tinha peso 3, agora a capacidade será 0 e o algoritmo termina.

Sumário

1. Indução
2. Recursividade
3. Tentativa e Erro
4. Divisão e Conquista
5. Algoritmos Gulosos
6. Programação Dinâmica
7. Algoritmos Aproximados

Algoritmos Aproximados

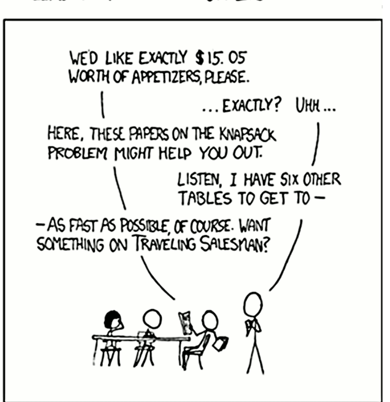
- Existem problemas para os quais é improvável que haja um algoritmo exato que possa obter uma solução em tempo polinomial.
- Esses problemas são chamados de problemas NP-difícil (*NP-hard*) — tempo polinomial não determinístico.

- Para esses problemas, **algoritmos aproximados** são desenvolvidos para:
  1. obter soluções que se aproximem da solução ótima por um fator constante (por exemplo 5% da solução ótima), e
  2. garantir um tempo de execução viável para a obtenção da solução
- São utilizados em problemas para os quias algoritmos polinomiais exatos existem, mas são muito caros computacionalmente para instâncias grandes.

Algoritmos Aproximados

- Os algoritmos aproximados estão ligados à problemas de otimização. Por isso, não se aplicam, por exemplo, a problemas de decisão "puros".
- Assim, para um dado problema a ser resolvido, é preciso concebê-lo como um problema de otimização.

- Os algoritmos aproximados são diferentes daqueles que utilizam **heurísticas**.
- Os algoritmos que utilizam heurísticas podem produzir um bom resultado, ou até mesmo a solução ótima, mas também podem não encontrar uma solução ou obter resultado distante da solução ótima.



por: xkcd (Creative Commons non-commercial 2.5 License)

Bibliografia I

ZIVIANI, N.
Projeto de Algoritmos. 3.ed.
Cengage, 2004.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C.
Algoritmos: teoria e prática (Seção 2.3, Capítulo 15, Capítulo 16).
Campus, 2002.

LOUREIRO, A.A.F.
Notas de aula: Paradigmas de projeto de algoritmos
UFMG, 2007, http://www.dcc.ufmg.br/~loureiro.