Problemas Tratáveis . . .

Até aqui: todos os algoritmos executavam em tempo $O(n^3)$.

Tempo (μs)		$33n$	$46n \lg n$	$13n^2$	$3.4n^3$	
Tempo assimp.		n	$n \lg n$	n^2	n^3	2^n
Tempo de	n=10	$.00033s$	$.0015s$	$.0013s$	$.0034s$	$.001s$
execução por	n=100	$.003s$	$.03s$	$.13s$	$3.4s$	$4.10^{14}s$
tamanho do	n=1000	$.033s$	$.45s$	$13s$	$.94h$	séculos
input	n=10000	$.33s$	$6.1s$	$22m$	39 dias	$\cdots$
	n=100000	$3.3s$	$1.3m$	1.5 dias	108 anos	$\cdots$
Tamanho máx.	$1s$	3.10^4	2000	280	67	20
do input para	$1m$	18.10^5	82000	2200	260	26

Concentramo-nos agora no estudo de problemas para os quais o melhor algoritmo *conhecido* tem complexidade exponencial, e cuja resolução demoraria pelo menos alguns anos ou séculos para inputs razoavelmente grandes.

Problemas de Optimização vs. Problemas de Decisão

A resolução de um problema de optimização consiste na selecção da melhor solução para outro problema.

- **Árvore Geradora Mínima – Opt**: escolher a melhor solução (i.e. de menor peso) para o problema da determinação de uma árvore geradora (qualquer).

A cada problema de optimização está normalmente associado um problema de decisão, i.e., um problema cuja solução é uma resposta sim/não:

- **Árvore Geradora Mínima – Dec**: dado um valor k, existirá alguma árvore geradora para G com peso $\leq k$?

Alguns Problemas Difíceis Famosos

Coloração de um Grafo $G = (V, E)$: é uma função $C : V \to S$, com S um conjunto finito de cores, verificando a restrição

$$(v, w) \in E \Longrightarrow C(v) \neq C(w)$$

(vértices adjacentes são coloridos com cores diferentes)

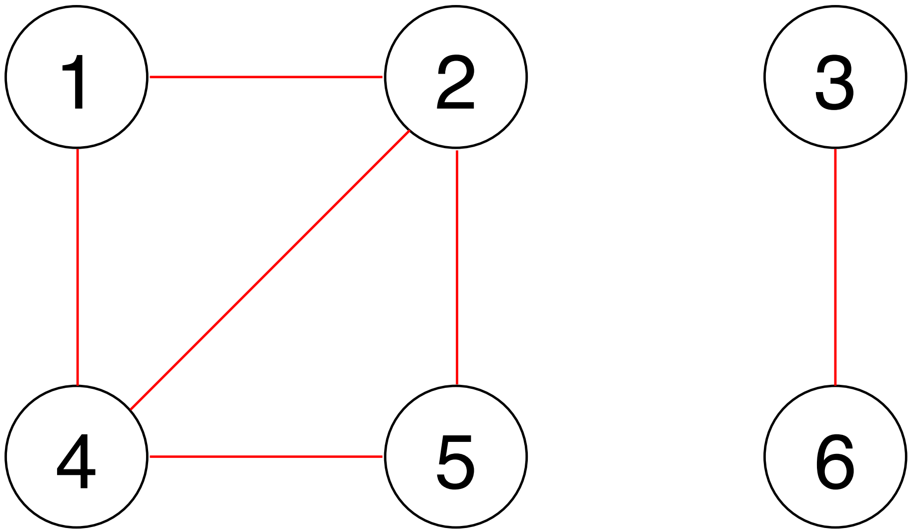
- **Problema Opt**: Dado G, determinar uma coloração C tal que $|C(V)|$ – o número de cores usadas – é mínimo.

- **Problema Dec**: Dado G e k inteiro, haverá alguma coloração de G usando no máximo k cores?

Coloração de Grafos

Aplicação: problemas de *escalonamento* – por exemplo o problema de determinação de horários dos exames de um conjunto de disciplinas (V) sujeito a incompatibilidades (pares de disciplinas cujos exames não podem acontecer em simultâneo - E). Qual o número de slots de tempo necessários?

Exemplo:



Alguns Problemas Difíceis Famosos

"Bin Packing": Dados n objectos de dimensões $s_1, \ldots, s_n$, com $0 < s_i \leq 1$,

- **Problema Opt**: Quantas gavetas de dimensão 1 serão necessárias para os arrumar? (E qual a disposição dos objectos correspondente?)
- **Problema Dec**: Dado um inteiro k, será possível arrumar os n objectos em k gavetas?

Aplicações:

- Sistemas Operativos: dispor programas em páginas de memória; dispor dados em palavras de tamanho fixo;
- Investigação Operacional – problemas de corte de componentes (e.g. tecido) em peças de dimensão normalizada.

Alguns Problemas Difíceis Famosos

"Knapsack": Dada uma mochila de capacidade C e n objectos de dimensões $s_1, \ldots, s_n$ e valores $p_1, \ldots, p_n$,

- **Problema Opt**: Determinar o valor máximo dos objectos que se consegue colocar na mochila (e a lista desses objectos).

- **Problema Dec**: Dado um inteiro k, existirá um conjunto de objectos que caiba na mochila e corresponda a um valor $\geq k$?

Aplicações: Planeamento económico; investimentos (tamanhos correspondem a capital investido, valor corresponde a lucro esperado).

Alguns Problemas Difíceis Famosos

Caminhos e Circuitos de Hamilton: Num grafo G, um *caminho de Hamilton* é um caminho que passa por cada vértice exactamente uma vez. Um *circuito de Hamilton* é qualquer ciclo que seja um caminho de Hamilton.

- **Problema Dec**: Decidir se G contém ou não um caminho de Hamilton (ou um circuito).

Alguns Problemas Difíceis Famosos

Caixeiro Viajante ("Traveling Salesman"): Dado um grafo pesado G,

- **Problema Opt**: Determinar o circuito de Hamilton de peso mínimo.
- **Problema Dec**: Para um inteiro k, haverá algum circuito de Hamilton em G, com peso $\leq k$?

Aplicações: O caixeiro viajante pretende minimizar a distância total percorrida para passar por todas as cidades que deve visitar. Mas também: circuito óptimo para recolha de lixo ou entrega de correio numa cidade ...

A Classe de Problemas P

- Um algoritmo é *limitado polinomialmente* se tem comportamento no pior caso em $O(P(n))$, com $P(n)$ um polinómio em n (a dimensão do input).

- Um problema diz-se limitado polinomialmente se existe um algoritmo limitado polinomialmente que o resolve.

- A Classe P é constituída pelos problemas de decisão limitados polinomialmente.

- A classe P *inclui* todos os problemas *razoáveis* mas também problemas de difícil resolução (há polinómios de crescimento muito rápido!)

- No entanto, é certo que um problema que *não pertença a P* será de resolução praticamente impossível.

- A classe P é *fechada por operações* diversas (e.g. $+$,$*$) $\Rightarrow$ útil?

A Classe de Problemas NP

Tipicamente um problema de decisão corresponde à obtenção de uma resposta para um problema de *existência* de um objecto, e uma solução para o problema corresponde a um tal objecto que justifica uma resposta verdadeira:

- Uma coloração de um grafo com k cores no máximo;
- Um circuito de Hamilton com peso $\leq k$ ...

Uma *solução proposta* para um problema de decisão é um objecto do tipo procurado, mas sobre o qual não se sabe se é uma solução:

- Uma coloração *qualquer* do grafo;
- Um circuito *qualquer* de Hamilton no grafo.

A Classe de Problemas NP

Para cada problema faz sentido que exista um processo (algoritmo) que, dada uma solução proposta, *verifica* se ela é ou não solução do problema.

Uma solução proposta será descrita por uma string de algum conjunto finito arbitrário de caracteres. A verificação da solução implica

1. Verificar que a string obedece ao formato utilizado para descrever as soluções, ou seja, que é *sintacticamente correcta*.

2. Verificar que a solução proposta descrita pela string verifica o critério do problema, ou seja é de facto uma solução.

Informalmente, a Classe NP é constituída pelos problemas de decisão em que a verificação de soluções pode ser feita em tempo polinomial.

Algoritmos Não-determinísticos

Trata-se de algoritmos de aplicação teórica, utilizados para a *classificação de problemas*. Um algoritmo não-determinístico tem duas fases:

1. **Fase não-determinística**: é escrita algures (em memória) uma string arbitrária s. Em cada execução do algoritmo esta string pode ser diferente.

2. **Fase determinística**: o algoritmo lê agora s e processa-a, após o que pode suceder uma de três situações:

   (a) algoritmo pára com resposta sim
   (b) algoritmo pára com resposta não
   (c) algoritmo não pára

A primeira fase pode ser vista como uma "tentativa de adivinhar" uma solução. A segunda fase verifica se este "palpite" obtido na primeira fase corresponde ou não a uma solução para o problema.

Algoritmos Não-determinísticos

- Estes algoritmos (ND) distinguem-se dos tradicionais pelo facto de a execuções diferentes poderem corresponder outputs (sim/não) diferentes.

- A resposta de um algoritmo A não-determinístico a um input x define-se como sendo "sim" sse *existe uma execução de A sobre x que produz output "sim"*.

  A resposta "sim" de A a x corresponde então à existência de uma solução para o problema de decisão com input x.

- O número de passos de execução de um algoritmo ND corresponde à soma dos passos das duas fases.

A Classe de Problemas NP

Um algoritmo ND A diz-se *limitado polinomialmente* se existe um polinómio $P(x)$ tal que para cada input (de dimensão n) para o qual a resposta de A seja “sim”, existe uma execução do algoritmo que produz output “sim” em tempo $O(P(x))$.

A Classe NP é constituída por todos os problemas de decisão para os quais existe um algoritmo não-determinístico limitado polinomialmente.

[NP $=$ *Nondeterministic Polynomial-bounded*]

Teorema. *Os problemas de* Coloração de Grafos, Bin Packing, Knapsack, Caminhos e Ciclos de Hamilton, *e* Caixeiro Viajante *são todos NP.*

Exemplo de Problema de Decisão

Coloração de Grafos: existirá uma coloração de $G = (V, E)$ com k ou menos cores?

- formato das soluções propostas: strings contendo caracteres correspondendo a cores (R = vermelho, G = verde, . . .), encontrando-se na posição i da string a cor atribuída ao vértice de índice i no grafo.

- Verificação de uma solução:
  1. verificar que a string tem comprimento $|V|$ e todos os caracteres correspondem a cores (verif. sintáctica);
  2. percorrer as listas (ou matriz) de adjacências do grafo e verificar que todos os pares de vértices adjacentes têm cores diferentes;
  3. verificar se a string contém no máximo k cores diferentes.

Exercício: Coloração de Grafos

Seja $G = (\{1,2,3,4,5\}, \{(1,2),(1,4),(2,4),(2,3),(3,5),(2,5),(3,4),(4,5)\})$. Poderá G ser colorido com 4 cores?

Considere-se um algoritmo não-determinístico que gera sucessivamente as seguintes soluções propostas. Quais são de facto soluções (i.e., qual o output do algoritmo em cada caso?)

```
RGRBG
RGRB
RBYGO
RGRBY
R%*,G@
```

Verificação é de facto feita em tempo polinomial!

P e NP

Teorema. $P \subseteq NP$

Prova. *Qualquer algoritmo determinístico para um problema de decisão é um caso particular de um algoritmo ND. Seja A um algoritmo determinístico para um problema $p_0 \in P$. Então podemos construír um algoritmo ND A' a partir de A:*

1. *a fase não-determinística escreve $s =$"" em zero passos;*

2. *a fase determinística é constituída por A (ignorando a string s escrita pela primeira fase).*

Sendo assim,

- *A' dá a resposta* sim *ou* não *correcta;*

- *A executa em tempo polinomial, logo A' executa também em tempo polinomial.*

Fica assim provado que $p_0 \in NP$.

A Grande Questão

Será que $NP \subseteq P$, ou seja $P = NP$?

Será que o não-determinismo é mais poderoso do que o determinismo?

Por outras palavras: será que alguns problemas que não podem ser resolvidos em tempo polinomial por um algoritmo vulgar podem ser resolvidos em tempo polinomial por algoritmos contendo um "gerador de palpites" não-determinístico?

Conjectura-se que NP seja muito maior do que P, no entanto não existe *nenhum problema* $p_0 \in NP$ para o qual tenha sido provado $p \notin P$!

Por exemplo, para todos os problemas NP apresentados anteriormente, não são conhecidos algoritmos determinísticos limitados polinomialmente, no entanto para nenhum deles foi provado um limite $O(f(n))$ com $f(n)$ assimptoticamente superior a qualquer polinómio.

A questão $[P = NP\ ?]$ permanece pois aberta (e muito valiosa!)

P e NP

Qual a dificuldade da definição de algoritmos limitados polinomialmente para os problemas que apresentámos como exemplos?

Os métodos para o desenho de algoritmos estudados neste curso podem ser resumidamente descritos como se segue:

Procurar uma **estratégia algorítmica "inteligente"** (divisão e conquista, "greedy", progr.dinâmica. . .) que utilize propriedades específicas do problema e evite assim a análise de *todas as combinações possíveis*.

- um algoritmo de ordenação não produz todas as permutações possíveis de uma sequência para depois escolher a correctamente ordenada;
- um algoritmo de caminhos mais curtos não examina todos os caminhos entre A e B para depois escolher o de menor peso;

e assim sucessivamente.

P e NP

O problema é que para nenhum dos problemas expostos se encontrou uma tal estratégia algorítmica que resulte num algoritmo limitado polinomialmente.

Resta-nos a estratégia **"força bruta"**: para qualquer problema NP, a resposta (*sim* / *não*) correcta para o problema pode ser obtida *deterministicamente*:

1. Seja A um algoritmo ND para o problema, e $p(n)$ o polinómio que o limita;
2. Cada string gerada pela primeira fase de A tem comprimento máximo $p(n)$;
3. Se o conjunto de caracteres utilizado contiver c caracteres, existirão $c^{p(n)}$ strings possíveis – um número *exponencial* em n.
4. Para resolver o problema, basta executar a segunda fase de A sucessivamente para cada string gerada na primeira fase, parando se se obtiver output "sim".

Assim, a estratégia "força bruta" resolve os problemas NP em tempo exponencial

O Tamanho de um Problema – Questão Subtil!

Problema: Dado um inteiro positivo n, haverá dois inteiros $j, k > 1$ tais que $n = jk$? (i.e, será n não-primo?)

Considere-se o seguinte algoritmo do tipo "força bruta":

```
found = 0;
j = 2;
while ((!found) && j < n) {
  if (n mod j == 0) found = 1;
  else j++;
}
```

Este algoritmo executa em tempo $O(n)$, no entanto trata-se de um problema famoso pela sua dificuldade (e é por isso utilizado em muitos algoritmos criptográficos).

De facto é importante identificar correctamente o tamanho de n, uma vez que disso depende a classificação do algoritmo como polinomial ou exponencial.

Tamanho e Representação

O tamanho de um número é o número de caracteres necessários para o escrever: o tamanho de 3500 é 4. Um inteiro n em notação decimal ocupa aproximadamente $\log_{10} n$ dígitos; em notação binária (representação em máquina) ocupa $\log_2 n$ dígitos.

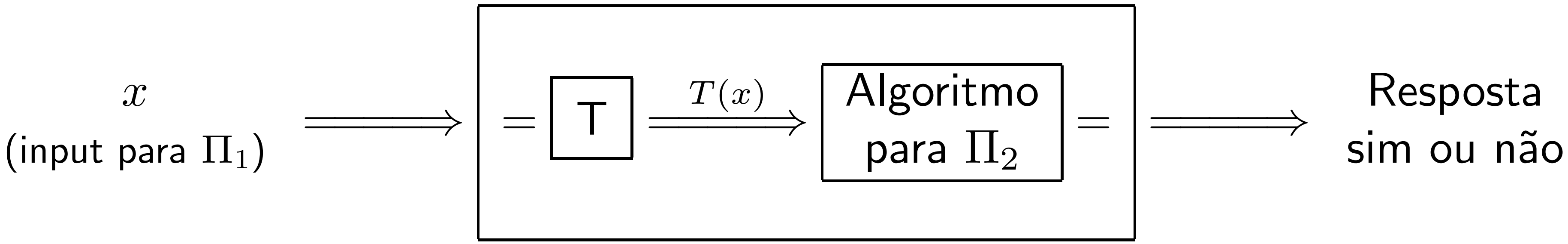
Observe-se: se um algoritmo executa no pior caso em tempo linear em n, e n tem tamanho $s = \log_k n$, então o algoritmo executa em tempo k^s, ou seja $T(s) = O(k^s)$. Um algoritmo de tempo aparentemente linear é de facto de tempo exponencial!

Redução de Problemas

Considere-se dois problemas Π_1 e Π_2. Dispomos por hipótese de um algoritmo para resolver Π_2, e de uma *função de transformação de inputs* $T()$, que transforma um input x para Π_1 num input $T(x)$ para Π_2, obedecendo à segunte restrição:

> A resposta correcta para Π_1 com input x é *sim* sse a resposta correcta para Π_2 com input $T(x)$ é *sim*.

Por composição obtém-se facilmente um algoritmo para Π_1:



Nestas circunstâncias diz-se que **reduzimos** Π_1 **a** Π_2.

Exemplo de Redução de Problemas

Π_1: Dadas n variáveis booleanas, será que pelo menos uma delas tem valor *verdadeiro*?

Π_2: Dados n inteiros, será positivo o maior deles?

$T()$: Seja $T(x_1, \ldots, x_n) = y_1, \ldots, y_n$ com $y_i = 1$ se x_i for verdadeiro, e $y_i = 0$ se x_i for falso.

qualquer algoritmo para Π_2 resolve Π_1 quando aplicado a $T(x_1, \ldots, x_n)$.

Reduções Polinomiais

Uma função $T()$ do conjunto de inputs de um problema de decisão Π_1 para o conjunto de inputs de um problema de decisão Π_2 diz-se uma redução polinomial de Π_1 em Π_2 se

1. $T()$ pode ser calculada em tempo polinomial; e

2. Para cada input x para Π_1, a resposta correcta para Π_2 sobre $T(x)$ é igual à resposta correcta para Π_1 sobre x.

Π_1 diz-se polinomialmente redutível a Π_2 se existe uma redução polinomial de Π_1 em Π_2, e escreve-se $\Pi_1 \propto \Pi_2$.

Reduções Polinomiais

A ideia que se pretende exprimir é que Π_2 é *pelo menos tão difícil de resolver como* Π_1.

Teorema. *Se* $\Pi_1 \propto \Pi_2$ *e* Π_2 *está em P, então* Π_1 *está em P.*

Prova. *Sejam* $p()$ *o polinómio que limita o comportamento da função de redução* $T()$*, e* $q()$ *o que limita o comportamento de um algoritmo para* Π_2*.*

Se x *for um input para* Π_1 *de tamanho* n*, então* $T(x)$ *tem tamanho* $p(n)$ no máximo*. Então o algoritmo para* Π_2 *executará no máximo* $q(p(n))$ *passos.*

O tempo total dispendido é pois limitado polinomialmente *por* $p(n) + q(p(n))$*.*

Problemas NP-completos

Um problema Π diz-se NP-completo se está em NP, e para qualquer outro problema Π_1 em NP se tem $\Pi_1 \propto \Pi$.

Teorema. *Seja Π um problema NP-completo. Se Π está em P então P=NP.*

Prova. *Para qualquer outro problema Π_1 em NP tem-se $\Pi_1 \propto \Pi$, logo, pelo teorema anterior, Π_1 está em P. Então $NP \subseteq P$.*

Conclusões a reter:

1. Para provar P=NP (o que é altamente improvável) bastaria provar que *um qualquer* problema NP-completo pode ser resolvido por um algoritmo limitado polinomialmente.

2. Como é improvável que seja P=NP, é também improvável que tal algoritmo exista!

Problemas NP-completos

Para provar que um problema Π é NP-completo basta mostrar que qualquer problema em NP é polinomialmente redutível a Π.

Por exemplo, para mostrar que o problema de coloração de grafos com 4 cores é NP-completo, poderíamos mostrar que para qualquer outro problema Π_1 em NP, existe uma função $T()$ tal que:

- $T()$ transforma em tempo polinomial um input x de Π_1 num grafo G;
- Este grafo descreve (num sentido informal) a computação de um algoritmo ND para Π_1, actuando sobre o input x;
- G poderá ser colorido com 4 cores sse a computação acima resultar em *sim*.

Problemas NP-completos

No entanto, depois de provado que um qualquer problema é NP-completo (pelo método anterior), surge outro método:

Teorema. *Para provar que um problema* Π *em NP é NP-completo basta mostrar, para outro problema NP-completo* $\widehat{\Pi}$ *conhecido, que* $\widehat{\Pi} \propto \Pi$.

Prova. *Como* $\widehat{\Pi}$ *é NP-completo, todos os problemas de* $NP \propto \widehat{\Pi}$. *Se provarmos* $\widehat{\Pi} \propto \Pi$, *teremos por transitividade de* $\propto$ *que todos os problemas de* $NP \propto \Pi$, *logo* Π *é NP-completo.*

Teorema. *Todos os problemas apresentados neste capítulo são NP-completos.*

Exemplo

Suponha-se conhecido que o problema dos Circuitos de Hamilton para grafos orientados (HO) é NP-completo.

Desejamos agora provar que o mesmo problema, mas para grafos não-orientados (HNO), é também NP-completo. Basta provar que HO $\propto$ HNO.

Seja $G = (V, E)$ orientado, e $G' = (V', E')$ com

- $V' = \{v^i \mid v \in V, i = 1, 2, 3\}$
- $E' = \{(v^1, v^2), (v^2, v^3) \mid v \in V\} \cup \{v^3, w^1) \mid (v, w) \in E\}$.

A função $T : G \mapsto G'$ constroi um grafo com $3|V|$ vértices e $2|V| + |E|$ arcos, pelo que executa em tempo polinomial.

É fácil provar que G tem um circuito de Hamilton (orientado) sse G' tem um circuito de Hamilton (não-orientado). Logo $T()$ é uma redução polinomial de HO em HNO.

Restrições sobre os Problemas

A introdução de restrições pode simplificar muito (ou não!) um problema.

Por exemplo: se restringirmos os problemas sobre grafos a grafos de Grau máximo ≤ 2 (nenhum vértice tem mais de dois arcos), os problemas dos circuitos de Hamilton e da coloração são resolúveis em tempo polinomial. Para o grau máximo 3 o primeiro torna-se NP-completo; para o grau máximo 4 o segundo torna-se também NP-completo.

O problema de decisão de coloração só é NP-completo para $k \geq 3$ cores; para $k \leq 2$ cores, a resolução é fácil.

Semelhanças Aparentes

Alguns problemas aparentemente parecidos podem ter complexidades muito diferentes:

- O problema do caminho mais curto entre dois vértices está em P; no entanto o do caminho mais longo é NP-completo.

- Circuito de Euler: ciclo que atravessa cada *arco* de um grafo orientado e ligado exactamente uma vez. Pode ser determinado (ou provada a sua não-existência) em tempo linear em $|E|$. Mas a determinação de circuitos de Hamilton é NP-completa.

Problemas de Decisão vs. Problemas de Optimização

Todo o estudo dos problemas NP e NP-completos foi efectuado para *problemas de decisão*, para os quais a formalização é mais fácil.

Claramente o problema de optimização é de resolução mais difícil do que o correspondente problema de decisão – basta observar que a partir da solução do primeiro se obtém facilmente a solução do segundo.

Por exemplo, conhecendo-se o número mínimo de cores com que se pode colorir um grafo G, é trivial responder à questão "poderá G ser colorido com k cores?" para qualquer k.

Se P$=$NP, ou seja se existissem algoritmos polinomiais para os problemas NP de decisão, poder-se-ia em muitos casos obter algoritmos polinomiais para os correspondentes problemas de optimização.

$\Rightarrow$ Como?

Resolução de Problemas NP-completos

Estratégias possíveis:

- Escolher o mais eficiente dos algoritmos exponenciais . . .
- Concentrar a escolha na análise de *caso médio* em vez de *pior caso*.
- Em particular, um estudo dos padrões de inputs que ocorrem com mais frequência pode levar à escolha de um algoritmo que se comporte melhor para esses inputs.
- Escolha pode depender mais de *resultados empíricos* do que de uma análise rigorosa.
- Estratégia alternativa: Algoritmos de Aproximação.

Algoritmos de Aproximação ou Heurísticos

Princípio: utilizar algoritmos rápidos (da classe P) que não produzem garantidamente uma solução óptima, mas sim *próxima* da solução óptima.

Muitas heurísticas utilizadas são simples e eficientes, resultando apesar disso em soluções muito próximas da optimalidade.

A definição de "proximidade à solução óptima" depende do problema.

Uma solução aproximada para, por exemplo, o problema do caixeiro viajante, não é uma solução que passa por "quase todos" os vértices do grafo, mas sim uma solução que passa por todos, e cujo peso é próximo do mínimo possível.

Por outras palavras, as soluções óptimas devem sempre ser *soluções propostas* por alguma execução de um algoritmo não-determinístico para o problema.

■ Exemplo de um Algoritmo Heurístico para "Bin Packing" ■

Distribuír n objectos de dimensões $s_1, \ldots, s_n$, com $0 < s_i \leq 1$ pelo número mínimo de gavetas de dimensão 1.

- Estratégia óptima: considerar todas as distribuições possíveis (número máximo de gavetas $= n$). O número de soluções é naturalmente exponencial em n.

- Estratégia **First Fit**: colocar cada objecto na primeira gaveta em que ele couber.

- Estratégia **First Fit Decreasing**: ordenar os objectos por dimensão decrescente antes de aplicar a estratégia "first fit".

Exercício: aplicar as estratégias ao conjunto de objectos de dimensões: 0.2, 0.3, 0.4, 0.5, 0.2, 0.2, 0.8, 0.4

$\Rightarrow$ Serão óptimas as soluções obtidas?

“First Fit”: Algoritmo Detalhado

```
int FF (float S[], int n, int bin[])
{
  float used[n];
  int i;          /* objectos */
  int j;          /* gavetas  */
  int m = 0;      /* numero necessário de gavetas */
  for (j=1 ; j<=n ; j++) used[j] = 0;
  for (i=1 ; i<=n ; i++) {
    j = 1;
    while (used[j]+S[i] > 1) j++;
    bin[i] = j;
    if (j>m) m=j;
    used[j] = used[j]+S[i];
  }
  return m;
}
```

Observações

- Tempo de execução de "First Fit": $\Theta(n^2)$.
- Qual o tempo de execução de FFD?
- **Teorema** [limite superior]: o número de gavetas usadas por FFD nunca excede em mais de 22
- No entanto o algoritmo comporta-se geralmente muito melhor do que indicado por este limite. Estudos empíricos sobre inputs grandes (com uma distribuição uniforme dos tamanhos) mostram que o número de gavetas extra é aproximadamente $0.3\sqrt{n}$.

$\Rightarrow$ Algo de estranho na realização de estudos empíricos ?