Projetos de Algoritmos por Indução

Nessa seção usaremos a *técnica de indução* para desenvolver algoritmos para certos problemas. Isto é, a formulação do algoritmo será análoga ao desenvolvimento de uma demonstração por indução matemática.

Para resolver um problema *P* faremos o projeto de um algoritmo em dois passos:

1. exibir a resolução de uma ou mais instâncias pequenas de *P* (o caso base);
2. exibir como uma solução de toda a instância de *P* pode ser obtida a partir da solução de uma ou mais instâncias menores de *P*.

O processo indutivo resulta em algoritmos recursivos, em que:

- a base da indução corresponde à resolução dos casos base da recursão;
- a aplicação da hipótese de indução corresponde a uma ou mais chamadas recursivas; e
- o passo da indução corresponde ao processo de obtenção da resposta para o caso geral a partir daquelas retornadas pelas chamadas recursivas.

Um benefício imediato é que o uso (correto) da técnica nos dá uma prova da correção do algoritmo. A analise da complexidade será expressa em uma recorrência. Freqüentemente o algoritmo é eficiente, embora existam exemplos simples em que isso não acontece.

Vejamos um exemplo com o problema:

Dada uma seqüência de números reais a_n, a_{n-1}, ..., a_1, a_0, e um número real x, calcular o valor do polinômio: $P_n(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_1 x + a_0$

Primeira solução indutiva:

- Caso base: n = 0. A solução é a_0.
- Suponha que para um dado *n* saibamos calcular o valor de $P_{n-1}(x) = a_{n-1}x^{n-1} + ... + a_1 x + a_0$

Para calcular $P_n(x)$, basta calcular x^n, multiplicar o resultado por a_n e somar o valor obtido por $P_{n-1}(x)$.

Observe o algoritmo *Polinomio_1*, que tem como dados de entrada o vetor A[1..n+1] com os coeficientes: A[1]= a_0, A[2]= a_1, ..., A[n]= a_{n-1}, e A[n+1] = a_n, *n* que é o grau do polinômio, o valor de *x* e, na saída o valor calculado para $P_n(x)$.

```
Polinômio_1(A,n,x)
1  se n = 0
2     então devolve A[1]
3     senão y ← Polinômio_1(A,n-1,x)
4           v ← 1
5           para i ← 1 até n faça
6              v ← v * x
7              y ← y + A[n+1] * v
8           devolve y
```

Na analise da complexidade, chamaremos de T(n) o número de operações aritméticas realizadas pelo algoritmo.

Obtemos a seguinte recorrência para T(n), onde mult = multiplicação e ad = adição:

$$T(n) = 0 \qquad se \quad n = 0$$

$$T(n) = T(n\text{-}1) + n * mult + 1\ ad \qquad se \quad n > 0$$

Resolvendo a recorrência, obtemos: $T(n) = \sum_{i=1}^{n} (i \cdot mult + ad) = \frac{(n+1)n}{2} \cdot mult + n \cdot ad$

O número de multiplicações pode ser diminuído calculando x^n com um algoritmo que calcula rapidamente a exponenciação, que veremos no próximo exemplo.

Segunda solução indutiva:

Um dos desperdícios cometidos nessa primeira solução é o re-cálculo de potências de *x*. Vamos eliminar essa computação desnecessária trazendo o cálculo de x^{n-1} para dentro da hipótese de indução.

Suponha que para um dado *n* saibamos calcular não só o valor de $P_{n-1}(x) = a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$, mas também o valor de x^{n-1}.

Então no passo de indução, primeiramente calculamos x^n multiplicando *x* por x^{n-1} , conforme exigido na hipótese. Em seguida, calculamos $P_n(x)$ multiplicando x^n por a_n e somando o valor obtido com $P_{n-1}(x)$.

Note que para o caso base, $n = 0$, a solução agora é: $(a_0 , 1)$.

No algoritmo *Polinomio_2*, temos os mesmos dados de entrada: o vetor A[1..n+1] com os coeficientes, *n* o grau do polinômio, o valor de *x* e agora na saída temos o par de valores $(P_n(x), x^n)$.

```
Polinômio_2(A,n,x)
1  se n = 0
2     então devolve (A[1], 1)
3     senão (y,v) ← Polinômio_2(A,n-1,x)
4           v ← v * x
5           y ← y + A[n+1] * v
6           devolve (y, v)
```

Na análise desta versão, T(n) novamente representa o número de operações aritméticas. T(n) é dada pela recorrência:

$$T(n) = 0 \qquad se \quad n = 0$$

$$T(n) = T(n\text{-}1) + 2\ mult + 1\ ad \qquad se \quad n > 0$$

Resolvendo a recorrência: $T(n) = \sum_{i=1}^{n} (2 \cdot mult + ad) = n \cdot (2 \cdot mult + ad)$

Com uma grande melhora: $T(n) = \Theta(n)$, más ainda podemos evoluir.

Terceira solução indutiva

A escolha de considerar o polinômio $P_{n-1}(x)$ na hipótese de indução não é a única possível. Veja uma alternativa que resulta num ganho de complexidade:

- caso base: n = 0. A solução é a_0.
- Suponha que para um dado $n > 0$ saibamos calcular o valor de $P_{n-1}(x) = a_n x^{n-1} + a_{n-1}x^{n-2} + \ldots + a_1$.

Então $P_n(x) = xP_{n-1}(x) + a_0$

Assim, basta uma multiplicação e uma adição para obtermos $P_n(x)$ a partir de $P_{n-1}(x)$.

Observe que o algoritmo implementado nessa solução é a regra de Horner para avaliar polinômios.

Na entrada do algoritmo *Polinomio_3,* mantivemos os mesmos dados, más invertemos a ordem em que armazenamos os coeficientes: temos a_0 = A[n+1], a_1 = A[n], ...,a_{n-1} = A[2], a_n = A[1]), e na saída temos o valor calculado $P_n(x)$:

```
Polinômio_3(A, n, x)
1  se n = 0
2     então devolve A[n+1]
3     senão y ← x * Polinômio_3(A, n-1, x)+ A[n+1]
4           devolve y
```

A análise de complexidade obtém:

$$T(n) = o \qquad se \quad n = o$$

$$T(n) = T(n\text{-}1) + 1\ mult + 1\ ad \qquad se \quad n > o$$

Resolvendo a recorrência: $T(n) = \sum_{i=1}^{n} (mult + ad) = n \cdot mult + n \cdot ad$

Embora o resultado tenha a mesma ordem da solução anterior, T(n) = $\Theta(n)$, observamos uma melhora com a redução de uma operação de multiplicação em cada chamada.

Vejamos outro exemplo simples na *Exponenciação*:

Problema: *Calcular a^n, para todo real a e inteiro $n \geq o$.*

Nossa solução por indução será:

- Caso base: n = 0; a_0 = 1.
- Hipótese de indução: suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^{n-1};
- Passo da indução: n > 0; por hipótese de indução sei calcular a^{n-1}.

  Então, calculo $\boldsymbol{a^n}$ multiplicando $\boldsymbol{a^{n-1}}$ por $\boldsymbol{a}$.

Vejamos o algoritmo:

```
Exp_3(a,n)
1  se n = 0
2     então devolve 1
3     senão y ← a * Exp(a, n-1)
4           devolve y
```

Para análise, T(n) representa o número de operações executadas no cálculo de an , com a relação de recorrência:

$$T(n) = c_1 \qquad se \quad n = o$$

$$T(n) = T(n\text{-}1) + c_2 \qquad se \quad n > o$$

Onde c_1 e c_2 representam, respectivamente, o tempo (constante) executado na atribuição da base e multiplicação do passo. Resolvendo a recorrência chegamos ao resultado:

$$T(n) = c_1 + \sum_{i=1}^{n} c_2 = c_1 + nc_2 = \Theta(n)$$

Nessa análise *n* não representa o tamanho da entrada, más sim seu valor. Para expressar a complexidade deste algoritmo em termos do tamanho da entrada *m*, devemos considerar que este *m* é dado pelo número de bits de representação binária, ou seja, $m = \lfloor lgn \rfloor + 1$. como $n = \Theta(2^m)$, concluímos que este algoritmo é exponencial no tamanho da entrada.

Vejamos um terceiro exemplo de projeto por indução re-visitando *"o problema da seqüência de soma máxima"* na solução que roda em tempo linear, partindo do seguinte enunciado:

Dada uma seqüência $A = a_1, a_2, \ldots, a_{n-1}, a_n$ *de números inteiros (não necessariamente positivos) encontre uma subseqüência consecutiva*

$$A' = a_i, a_{i+1}, \ldots, a_{j-1}, a_j, \ 1 \leq i, j \leq n,$$

cuja soma seja máxima dentre todas .

Por exemplo:

i) $A = \{3, 0, -1, 2, 4, -1, 2, -2\}$ *obtemos:* $A' = \{3, 0, -1, 2, 4, -1, 2\}$ *com soma = 9.*

ii) $A = \{-1, -2, 0\}$ *obtemos:* $A' = \{0\}$ *com soma = 0.*

iii) $A = \{-3, -1\}$ *obtemos:* $A' = \{\ \}$ *que convencionamos igual a zero.*

Antes de empregar a técnica de indução vamos re-apresentar a solução iterativa ligeiramente modificada, para mostrar a soma máxima e também os índices de início e final da subseqüência:

```
Somax4(A,n)
 1  max ← aux ← j ← k ← 0; ini ← fim ← 1
 2  para i ← 1 até n faça
 3      aux ← aux + A[i]
 4      k ← k + 1
 5      se n = 0
 6         então max ← aux
 7                 fim ← k
 8         senão se aux < 0
 9                    então aux ← 0
10                          j ← k ← i+1
11                          ini ← j
12  devolve (max, ini, fim)
```

Inicialmente, vamos examinar o que podemos obter de uma hipótese de indução simples:

Para um n qualquer, sabemos calcular a soma máxima de seqüências cujo comprimento seja menor que n.

Seja então a seqüência $A = a_1, a_2, \ldots, a_{n-1}, a_n$, *uma seqüência qualquer de comprimento* $n > 1$.

Considere a seqüência $A_{n-1} = a_1, a_2, \ldots, a_{n-1}$ *obtida removendo-se o número* a_n *que apresenta a soma máxima na subseqüência* $A_{n-1}' = a_i, a_{i+1}, \ldots, a_{j-1}, a_j$, *obtida aplicando hipótese.*

Há três casos a examinar:

1. $A_{n-1}' = \{\ \}$; neste caso $A' = a_n$ se $a_n \geq 0$ ou $A' = \{\ \}$ se $a_n \leq 0$.
2. $j = n-1$; como no caso anterior, temos $A' = A_{n-1}', a_n$, se $a_n \geq 0$ ou $A' = A_{n-1}'$ se $a_n \leq 0$.
3. $j < n-1$; aqui também há dois casos a considerar:
   i) $A' = A_{n-1}'$, ou seja, a subseqüência de soma máxima é a mesma;
   ii) $A' \neq A_{n-1}'$, ou seja, o elemento a_n é parte da subseqüência de soma máxima A' de A, sendo portando da forma: $A' = a_k, a_{k+1}, \ldots, a_{n-1}, a_n$, para algum $k < n-1$.

Note que a hipótese de indução nos permite resolver todos os casos, exceto o último. Não há informação suficiente na hipótese de indução para permitir a resolução deste caso...

Más, o que falta à hipótese?...

É evidente que quando encontramos $A' = a_k, a_{k+1}, \ldots, a_{n-1}, a_n$, na verdade encontramos um sufixo dentre as subseqüências de A.

Assim se conhecemos o sufixo que nos dá a maior soma, teremos resolvido o problema completamente. Parece então natural enunciar a seguinte hipótese (agora fortalecida):

Para um n qualquer, sabemos calcular a soma máxima de seqüências cujo comprimento seja menor que n e o sufixo máximo de seqüências cujo comprimento seja menor que n.

É clara desta discussão também, a base da indução: *para n = 1,* a seqüência de $A = a_1$ é a_1 se $a_1 \geq 0$, e *vazia* (com valor nulo) caso contrário.

Escrevemos então o algoritmo:

```
Somax4-ind(A,n)
 1  se n = 1
 2     então se A[1]< 0
 3               então ini ← fim ← i-suf ← max ← suf  ← 0
 4               senão ini ← fim ← i-suf ← 1
 5                     max ← suf  ← 0
 6     senão (ini, fim, i-suf, max, suf) ← Somax4-ind(A,n-1)
 7            se i-suf = 0
 8               então i-suf ← n
 9            suf ← suf + A[n]
10            se suf > max
11               então ini ← i-suf
12                     fim ← n
13                     max ← suf
14               senão se suf < 0
15                        suf ← i-suf ← 0
16  devolve (ini, fim, i-suf, max, suf)
```

Analisando a complexidade $f(n)$ desse algoritmo temos:

$$f(n) = \Theta(1) \qquad se \quad n = 1$$
$$f(n) = f(n\text{-}1) + \Theta(1) \qquad se \quad n > 1$$

Com solução:

$$\sum_{1}^{n} \Theta(1) = n\Theta(1) = \Theta(n)$$ de mesma ordem que a solução anterior.

Exercícios de fixação

1. Para o problema de *verificar se um número x faz parte de uma seqüência de n números armazenados em um vetor A[1..n],* um algoritmo simples nos leva a uma solução de complexidade *O(n)* no pior caso. Projete um algoritmo incremental por indução, analisando sua correção e complexidade nessa nova versão.

2. Seja A um vetor ordenado de números inteiros e distintos (podem ser negativos). Projete, por indução, um algoritmo de complexidade *O(lgn)* para determinar se existe um índice *i*, $1 \leq i \leq n$, tal que A[*i*] = *i*.

3. Projete, por indução, um algoritmo que recebe um vetor A[e..d] e devolve o valor da soma dos elementos A[*i*] tais que $e \leq i \leq d$ e i é *par*.

4. Árvores AVL, proposta por Adel'son-Vel'skii e Landis em 1962, são exemplos de árvores binárias balanceadas. Essas estruturas de dados minimizam o tempo de busca de informações nela armazenadas. A idéia é que, para todo nó *N* da árvore, o *fator de balanceamento* de *N*, isto é, a *diferença* entre as alturas das duas subárvores esquerda de direita não desvie muito de zero. Para ser AVL o fator de balanceamento deverá ter valor: –1, 0, ou 1. Projete por indução, um algoritmo que "marque" os nós de uma árvore binária *A* qualquer, que não sejam AVL, mas cujos descendentes sejam todos AVL. Escreva e resolva a recorrência que expressa a complexidade do seu algoritmo em função do número de nós de *A*.

5. Seja *A[1..n]* um vetor com *n* inteiros. Um inteiro é um elemento majoritário em *A* se ocorre (estritamente) mais que *n/2* vezes em *A*. Projete, por indução, um algoritmo linear em *n* que determine se um dado vetor *A[1..n]* possui um elemento majoritário; caso positivo, seu algoritmo deve encontrar esse elemento.

6. Num conjunto S de n pessoas, uma *celebridade* é alguém que é conhecido por todas as pessoas de S, mas que não conhece ninguém. Isso implica que pode existir somente uma celebridade em S. (Celebridades são pessoas de difícil convívio...). Problema: *projetar por indução*, um algoritmo linear em *n* que determine se existe uma celebridade em S.

Divisão e Conquista

Historicamente, a expressão "*divisão e conquista*" tem suas origens na estratégia de batalha dos generais de Napoleão (1800-1814) que dividiam o exército inimigo em vários sub-exércitos separados, para poder vencer cada um mais facilmente. O desenvolvimento de algoritmos por *divisão e conquista* reflete esta estratégia militar, da seguinte forma:

> *Dada uma instância de um problema, de tamanho n, o método divide-a em k sub-instâncias disjuntas (1< $k \leq n$) que correspondem a k subproblemas distintos. Estes subproblemas são resolvidos separadamente e então se acha uma forma de combinar as soluções parciais para se obter a solução para a instância original. Se o tamanho das sub-instâncias é ainda relativamente grande, pode-se aplicar novamente o método para se obter sub-instâncias cada vez menores, até que se obtenham instâncias tão pequenas que os subproblemas resultantes da aplicação do método são do mesmo tipo que o problema original. Neste caso, a aplicação sucessiva do método pode ser expressa naturalmente por um algoritmo recursivo, isto é, dentro de um algoritmo X, que tenha um dado de entrada de tamanho n, usamos ao próprio X, k vezes, para resolver sub-entradas menores que n.*

O *algoritmo genérico* que emprega esse paradigma é baseado no princípio da indução:

```
Algoritmo_DC (entrada: instância x)
1 se x é suficientemente pequeno
2   então devolve (solucione x, com função adequada)
3   senão decomponha x em k instâncias menores
4         para i ← 1 até k faça
5             y[i] ← Algoritmo_DC(x/k)
6             obtenha y combinando as k soluções y[i]
7 devolve (saída: solução y para a instância x)
```

Exponenciação

Num primeiro estudo, vamos retomar ao exemplo simples da *Exponenciação* tratado na seção anterior:

Problema: *Calcular* a^n*, para todo real* a *e inteiro* $n \geq 0$.

Em nossa solução por indução obtivemos um algoritmo de complexidade $\Theta(n)$ no número de operações.

Vejamos uma nova solução onde empregamos a indução forte (essa difere da indução simples na suposição da hipótese: supomos que a propriedade em estudo vale para todos os casos anteriores e não somente para o anterior) de forma a obter um algoritmo de divisão e conquista:

- Caso base: n = 0; $a_0 = 1$.
- Hipótese de indução: suponha que, para qualquer inteiro $n > 0$ e real a, sei calcular a^k; para todo $k < n$.
- Passo da indução: n > 0; por hipótese de indução sei calcular $a^{\lfloor n/2 \rfloor}$.

Então, calculo a^n na forma: $$a^n = \begin{cases} \left(a^{\lfloor n/2 \rfloor}\right)^2 & n = par \\ a \cdot \left(a^{\lfloor n/2 \rfloor}\right)^2 & n = ímpar \end{cases}$$

Vejamos o algoritmo:

```
Exp_DC(a,n)
1 se n = 0
2   então devolve 1
3   senão x ← Exp_DC( a, ⌊n/2⌋ )
4         y ← x * x
5         se n MOD 2 = 1
6           então y = y * x
7 devolve y
```

Em nossa análise *T(n)* representará o número de operações executadas pelo algoritmo no cálculo de a^n , com a relação de recorrência:

$$T(n) = c_1 \qquad se \quad n = 0$$

$$T(n) = T(\lfloor n/2 \rfloor) + c_2 \qquad se \quad n > 0$$

Para determinar a fórmula fechada para essa relação, supomos que *n* é potência de 2, ou seja, $n = 2^k$ para algum k inteiro.

Assim:

$T(n) = T(n/2) + c_2 = [T(n/4) + c_2] + c_2 = [[T(n/8) + c_2] + c_2] + c_2 = \ldots$

$\ldots = [T(1) + c_2] + k*c_2 = c_1 + c_2 + k(c_2) = c_1 + c_2 + lgn(c_2)$, *pois* $n = 2^k$.

ou seja,

$$T(n) = \Theta(lgn)$$

Ao expressar a complexidade deste algoritmo em termos do tamanho da entrada *m*, obtemos agora um *algoritmo linear*, pois *m* é dado pelo número de bits de representação binária, ou seja, $m = \lfloor lgn \rfloor + 1$.

Busca Binária

Nesse exemplo vamos retomar o problema de pesquisar um elemento em um vetor ordenado através de uma Busca Binária para estudar a técnica de projeto e a análise do algoritmo.

> Problema: *dado um vetor ordenado A com n números reais e um real x, determinar a posição* $1 \leq i \leq n$ *tal que* $A[i] = x$, *ou que não existe tal i sinalizando com* $i = 0$.

Se utilizarmos indução forte para projetar o algoritmo, podemos obter um algoritmo de divisão e conquista que nos leva ao algoritmo de Busca Binária. Como o vetor está ordenado, conseguimos determinar, com apenas uma comparação, que *x* não está em metade das posições do vetor.

Eis o algoritmo:

```
Bbin(x,A,e,d)
 1 se e = d
 2    então se x = A[e]
 3                então devolve e
 4             senão devolve -1
 5    senão m ← ⌊(e+d)/2⌋
 6          se x = A[m]
 7            então devolve m
 8             senão se x < A[m]
 9                     então devolve Bbin(x, A, e, m-1)
10                     senão devolve Bbin(x, A, m+1, d)
```

Analisando esse algoritmo vamos supor que a complexidade é T(n) para o pior caso:

- Se $n = 1$ observamos a execução das linhas 1 – 4 com tempo constante (ordem $\Theta(1)$).
- Se $n > 1$, temos a execução das linhas 1, 5 – 10, consumindo tempo constante (ordem $\Theta(1)$) na verificação do provável trecho onde o elemento se encontra (linhas 6 - 8). Na seqüência, linha 9 ou 10, teremos a execução recursiva do algoritmo com tempo proporcional a $T(\lceil n/2 \rceil)$. Quando n é uma potência de 2, podemos deduzir a fórmula fechada como no exemplo anterior chegando a solução: $T(n) = O(lgn)$.

Vamos também analisar a correção desse algoritmo: *o algoritmo Bbin soluciona corretamente o problema de busca de um número real x em um veto ordenado A com n números reais.*

Demonstração: como apresentado no algoritmo, supomos, sem perda de generalidade, que A está em ordem crescente, e fazemos a demonstração por indução em n:

- Base indutiva $n = 1$; nesse caso, é óbvio que a execução da linha 2 detecta corretamente a presença ou ausência de x em A[1..n].
- Hipótese indutiva: supomos Bbin ser correto para vetores com menos que n elementos, para $n > 1$.
- Passo indutivo: quero então provar que Bbin é correto para uma entrada com n elementos. Como $n > 1$, executaremos as linhas 5 e 6, determinando, com apenas uma comparação na linha 8, em qual porção do vetor x não está presente: se $x < A[m]$, obviamente devemos verificar a presença de x somente na porção $A[m+1..n]$, caso contrário pesquisaremos na porção $A[e..m-1]$.

Recorrências de Divisão e Conquista

Todo algoritmo eficiente de Divisão e Conquista divide os problemas em subproblemas, onde cada um é uma fração do problema original, e então realiza algum trabalho adicional para computar a resposta final, resultando na expressão geral de recorrência: $f(n) = af(n/b) + g(n)$.

O teorema seguinte pode ser usado para determinar esse tempo para a maioria dos algoritmos de divisão e conquista.

Teorema: a solução para a equação $f(n) = af(n/b) + O(n^k)$, *onde* $a \geq 1 \text{ e } b > 1$, *é*

$$f(n) = \begin{cases} O(n^{\log_b a}) \text{ se } a > b^k \\ O(n^k \log n) \text{ se } a = b^k \\ O(n^k) \text{ se } a < b^k \end{cases}$$

Prova:

Seguindo a analise que realizamos acima, assumimos *m* ser potência de *b*; portanto $n = b^m$.

Então $n/b = b^{m-1}$ *e* $n^k = (b^m)^k = b^{mk} = (b^k)^m$.

Assumimos $f(1) = 1$ e ignoramos a constante $O(n^k)$ e assim temos:

$$f(b^m) = af(b^{m-1}) + (b^k)^m$$

dividindo por a^m:

$$\frac{f(b^m)}{a^m} = \frac{f(b^{m-1})}{a^{m-1}} + \left\{\frac{b^k}{a}\right\}^m$$

aplicando essa equação para outros valores de m obtemos:

$$\frac{f(b^{m-1})}{a^{m-1}} = \frac{f(b^{m-2})}{a^{m-2}} + \left\{\frac{b^k}{a}\right\}^{m-1}$$

$$\frac{f(b^{m-2})}{a^{m-2}} = \frac{f(b^{m-3})}{a^{m-3}} + \left\{\frac{b^k}{a}\right\}^{m-2}$$

...

$$\frac{f(b^1)}{a^1} = \frac{f(b^0)}{a^0} + \left\{\frac{b^k}{a}\right\}^1$$

Ao somarmos todas as equações cancelamos os termos iguais que aparecem de cada lado da igualdade, obtendo:

$$\frac{f(b^m)}{a^m} = 1 + \sum_{i=1}^{m} \left\{\frac{b^k}{a}\right\}^i = \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i$$

portanto,

$$f(n) = f(b^m) = a^m \sum_{i=0}^{m} \left\{\frac{b^k}{a}\right\}^i$$

i) *se* $a > b^k$, então a soma é uma série geométrica com quociente menor que 1. Desde que a soma de séries infinitas converge para uma constante, essa soma finita também converge, e a equação resulta:

$$f(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$$

ii) se $a = b^k$, então cada termo da soma será 1, desde que a soma contêm $1+\log_b n$ termos e $a = b^k$ implica que $\log_b a = k$, obtemos:

$$f(n) = O(a^m \log_b n) = O(n^{\log_b a} \log_b n) = O(n^k \log_b n) = O(n^k \log n)$$

iii) se $a < b^k$, então os termos na série serão maiores que 1, e obtermos:

$$f(n) = a^m \frac{\left(b^k/a\right)^{m+1} - 1}{\left(b^k/a\right) - 1} = O\left(a^m \left(b^k/a\right)^m\right) = O\left((b^k)^m\right) = O(n^k)$$

... provando o último caso do teorema.

No capítulo 4, seção 4.3, do livro texto, *Cormen e colaboradores* apresentam a seguinte formulação para este teorema, denominado *Teorema Mestre*:

Sejam $a \geq 1$ e $b > 2$ constantes, seja g(n) uma função e seja f(n) definida para os inteiros não – negativos pela relação de recorrência: $f(n) = af(n/b) + g(n)$.

Então *f(n)* pode ser limitada assintoticamente da seguinte maneira:

1. se $g(n) \in O(n^{\log_b a - \varepsilon})$ para alguma constante $\varepsilon > 0$, então $f(n) \in O(n^{\log_b a})$.
2. se $g(n) \in \Theta(n^{\log_b a})$, então $f(n) \in \Theta(n^{\log_b a} \lg n)$
3. se $g(n) \in \Omega(n^{\log_b a + \varepsilon})$ para alguma constante $\varepsilon > 0$, e se $af(n/b) \leq cf(n)$, para alguma constante $c < 1$ e para n suficiente grande, então $f(n) \in \Theta(g(n))$.

Com esses resultados facilitamos a análise das recorrências que encontramos nos algoritmos de divisão e conquista.

Entretanto, como já observamos em alguns exemplos, existem recorrências onde não podemos aplicar o Teorema Mestre, eis alguns exemplos:

i) $T(1) = 1$
$T(n) = T(n-1) + n$

ii) $T(b) = 1$
$T(n) = T(n-a) + T(a) + n$ *(para* $a \geq 1$, $b \leq a$, *a e b inteiros)*

iii) $T(1) = 1$
$T(n) = T(\alpha n) + T((1-\alpha)n) + n$ *(para* $0 < \alpha < 1$*)*

iv) $T(1) = 1$
$T(n) = T(n-1) + lgn$

v) $T(1) = 1$
$T(n) = 2T(n/2) + nlgn$

Vejamos agora, mais algumas aplicações e respectivas análises de complexidade.

Mergesort

Esse método de ordenação combina dois ou mais arquivos classificados num terceiro também ordenado: a intercalação.

Antes de resolver nosso problema de ordenar um arquivo, resolve mos o seguinte problema auxiliar: supondo $A[e..m]$ e $A[m+1..d]$ em ordem crescente; queremos colocar $A[e..d]$ em ordem crescente:

e					m	m+1			d
11	**33**	**33**	**55**	**55**	**77**	**22**	**44**	**66**	**88**

A solução é dada pela função Intercala que já analisamos em seção anterior:

```
Intercala (A, e, d, m)
1    para i ← e até m faça
2        B[i] ← A[i]
3    para j ← m+1 até d faça
4        B[d+m+1-j]← A[j]
5    i ← e
6    j ← d
7    para k ← e até d faça
8        se B[i] ≤ B[j]
9          então A[k] ← B[i]
10               i ← i+1
11         senão A[k] ← B[j]
12               j ← j-1
```

O tempo que Intercala consome em função do tamanho da instância: $T(n) = \Theta(n)$. Vamos usar Intercala para escrever o algoritmo de ordenação Mergesort. Esse método pode ser usado para classificar um arquivo supondo a divisão de um arquivo de n registros em n arquivos de tamanho 1, e a partir daí intercalarmos 2 *a* 2 os arquivos ordenando-os. Depois intercalamos os novos arquivos de 2 elementos formando arquivos de 4 registros, e assim sucessivamente até que todo o arquivo esteja ordenado.

O algoritmo é recursivo e a base da recursão é $e \geq d$; nesse caso não é preciso fazer nada.

```
Mergesort(A, e, d)
1 se e < d
2   então m ← ⌊(e + d)/2⌋
3         Mergesort(A, e, m)
4         Mergesort(A, m+1, d)
5         Intercala(A, e, d, m)
```

Vejamos uma simulação com o arquivo de 12 elementos:

92	**86**	**57**	**37**	**25**	**33**	**48**	**13**	**12**	**9**	**15**	**18**
92	86	57	37	25	33	**48**	**13**	**12**	**9**	**15**	**18**
92	86	57	**37**	**25**	**33**	48	13	12	**9**	**15**	**18**
92	**86**	**57**	37	**25**	**33**	48	**13**	**12**	9	**15**	**18**
	86	57		25	33		13	12		15	18

Completamos a divisão: n arquivos de 1 registro ...

... intercalando os arquivos ...

	57	86		25	33		12	13		15	18
92	**57**	**86**	37	**25**	**33**	48	**12**	**13**	9	**15**	**18**
57	86	92	**25**	**33**	**37**	12	13	48	**9**	**15**	**18**
25	33	37	57	86	92	**9**	**12**	**13**	**15**	**18**	**48**
9	**12**	**13**	**15**	**18**	**25**	**33**	**37**	**48**	**57**	**86**	**92**

Analisando o tempo de processamento do Mergesort, temos que resolver a recorrência:

$$T(n) = \begin{cases} 1 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 1 \end{cases}$$

Para uma instância de tamanho n temos: $T(n) = 2T(n/2) + n$

Utilizando o teorema Mestre, identificamos nessa equação o segundo caso analisado na equação geral de recorrências: $f(n) = af(n/b) + g(n)$, com $a = b^k$, com as constantes $a = 2, b = 2$ e $k = 1$, tendo com resultado: $T(n) = \Theta(n \lg n)$

Multiplicação de inteiros

Estudemos agora o *problema da multiplicação de inteiros* empregando a estratégia de divisão e conquista. Consideremos dois números *p* e *q*, inteiros e não negativos ambos expressos numa base b ≥ 2 com *n* algarismos onde $n \geq 1$.

Queremos resolver eficientemente o problema de se obter o produto $r = p * q$.

O algoritmo tradicional de multiplicação para este problema requer $O(n^2)$ multiplicações: algarismo a algarismo. Desenvolveremos outro algoritmo com tempo de ordem $O(n^{\lg 3})$, aproximadamente $O(n^{1,59})$.

Suponha, para simplificar, que *n* é potência de 2; se não for complete *p* e *q* com um número suficiente de zeros à esquerda. Dividimos tanto p quanto q em duas metades de n/2 algarismos cada (considere uma eqüipartição). Têm-se então:

$$p = p_1 * b^{n/2} + p_2 \quad e \quad q = q_1 * b^{n/2} + q_2 \qquad (1)$$

onde p_1, p_2, q_1 e q_2 tem $n/2$ algarismos.

Considere um caso particular em que $n = 1$, tem-se que $p_1 = q_1 = 0$.

Num caso geral considere, por exemplo, para $b = 10$, $p = 1234$, $q = 1982$ tem-se:

$$n = 4, \quad p_1 = 12, \quad p_2 = 34, \quad q_1 = 18 \quad e \quad q_2 = 82.$$

Tratando cada uma das metades como um número, podemos escrever:

$$r = p * q = p_1 * q_1 * b^{n/2} + (p_1 * q_2 + p_2 * q_1) * b^{n/2} + p_2 * q_2 \qquad (2)$$

Em (2) temos quatro multiplicações de dois números com n/2 algarismos cada, três somas, e duas multiplicações da forma $c * b^k$. Esta última operação consiste simplesmente em colocar-se ***k*** zeros à direita de ***c*** e, portanto pode ser efetuada em tempo proporcional a ***k***.

Queremos agora diminuir o número de multiplicações em (2), de quatro para três, reescrevendo-se a expressão entre parêntesis em (2) na forma:

$$p_1 * q_2 + p_2 * q_1 = (p_1 + p_2) * (q_1 + q_2) - p_1 * q_1 + p_2 * q_2 \qquad (3)$$

Através de (3) podemos então calcular $p*q$ com três multiplicações, quatro somas e duas subtrações. A motivação para se substituir uma única multiplicação por somas e subtrações é que estas últimas podem ser efetuadas mais rapidamente do que a multiplicação.

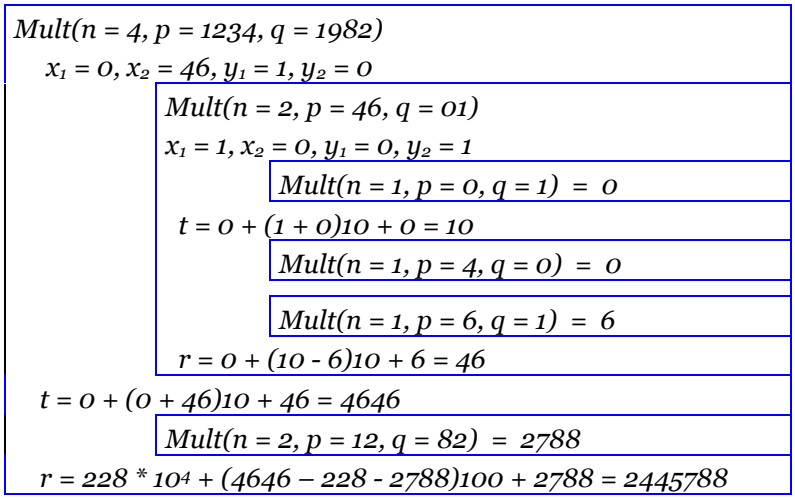
Finalmente, podemos escrever o algoritmo para a obtenção de $p*q$:

```
Mult(n,p,q,b)
1 se n = 1
2   então r = p*q
3   senão x = p1 + p2
4         y = q1 + q2
5         t = x1*y1*b^n + (x1*y2 + x2*y1)*b^(n/2) + Mult(n/2, x2, y2)
6         u = Mult(n/2,p1,q1,b)
7         v = Mult(n/2,p2,q2,b)
8         r = u*b^n + (t – u – v)*b^(n/2) + v
```

Observe que ***x,*** calculado na linha 4, tem $1 + n/2$ algarismos, pois p_1 e p_2 têm $n/2$ algarismos; sendo x composto por $x = x_1 * b^{n/2} + x_2$ tendo x_1 um algarismos.

Uma notação análoga é usada para o ***y*** calculado na linha 5 e em conseqüência, no cálculo de ***t*** na linha 6, x_1 e y_1 tem apenas um algarismo enquanto que x_2 e y_2 tem $n/2$ algarismos. Exceto por essas características, a expressão na linha 6 nada mais é do que a equação (2) adaptada para o cálculo de ***t = x * y.***

Apresentamos a seguir uma pilha de execução recursiva do Mult para uma entrada, na base $b = 10$:



Analisando o tempo para execução de Mult, se $n > 1$ supomos que todas as operações a partir da linha 4 são executadas, e também que n é potência de 2. As chamadas recursivas necessitam de um tempo $3f(n/2)$, para as três chamadas nas linhas 4, 5 e 6. Todas as operações de soma, subtração e multiplicação por um algarismo (x_1 ou y_1) e por potências de ***b,*** são efetuadas em um tempo proporcional a ***n***.

Temos então: $f(n) = 3f(n/2) + Cn$, para $n > 1$ e C uma constante (4)

Se $n = 1$, só a linha inicial e a final são executadas, em tempo constante, independe de ***n.***

Podemos adotar de forma consistente com (4): $f(1) = C$ (5)

E podemos provar que o algoritmo Mult tem execução proporcional a $O(n^{\lg 3})$, se ***n*** é uma potência de 2. Basta provarmos que a fórmula de recorrência (4) e (5) tem solução:

$$f(n) = 3Cn^{\lg 3} - 2Cn \qquad (6)$$

Para isso, façamos indução em ***n***.

A base $n = 1$ é trivial. Se (6) satisfaz (4) e (5), para valores de $n = k$, $k > 1$, provemos que o mesmo acontece para $n = 2k$.

$f(2k)$	=	$3f(k) + C(2k)$	(pela equação (4))
	=	$3[3Ck^{lg3} - 2Ck] + 2Ck$	(por hipótese indutiva)
	=	$3C(2k)^{lg3} - 2C(2k)$	... com isto completamos a indução e a prova.

Note que este algoritmo tem solução de acordo com o 1º caso, $a > b^k$, apresentado na solução do teorema sobre as recorrências dos algoritmos de Divisão e Conquista.

Em relação à expressão inicial: $f(n) = af(n/b) + O(n^k)$

Identificamos as constantes: $a = 3, b = 2 \; e \; k = 1$.

Algoritmo de Strassen para multiplicação de matrizes

O algoritmo de Strassen para multiplicação de matrizes é mais uma aplicação da técnica de divisão e conquista: vamos calcular o produto matricial *C = A * B, matrizes n × n*.

Supondo que *n* é uma potência de 2, dividimos *A, B* e *C* em quatro matrizes com dimensão *n/2 × n/2,* cada uma, reescrevendo a equação *C = A*B,* da seguinte forma:

$$\begin{pmatrix} A_{11}A_{12} \\ A_{21}A_{22} \end{pmatrix} * \begin{pmatrix} B_{11}B_{12} \\ B_{21}B_{22} \end{pmatrix} = \begin{pmatrix} C_{11}C_{12} \\ C_{21}C_{22} \end{pmatrix} \qquad (1)$$

onde

$$C_{ij} = A_{i1} * B_{1j} + A_{i2} * B_{2j}\text{, } 1 \le \text{ i, j} \le 2 \qquad (2)$$

Se n = 2, as sub-matrizes têm dimensão 1×1 e o produto A * B pode ser calculado diretamente pelas fórmulas de (2), por multiplicação de elementos. Para n > 2, os cálculos dos C_{ij} em (2) necessitam de multiplicações de sub-matrizes de dimensão *n/2x n/2.*

Como *n/2* também é uma potência de 2, a multiplicação das sub-matrizes pode ser feita pela aplicação recursiva do particionamento em quatro sub-matrizes (de *n/4xn/4*).

Prosseguindo recursivamente, nessa estratégia de dividir e conquistar observaremos que os particionamentos sucessivos resultarão em vários casos de multiplicação de matrizes 2x2, que podem ser efetuados diretamente.

Esse algoritmo requer 8 multiplicações e 4 somas de matrizes *n/2xn/2*, conforme (2). Sabendo que cada uma destas somas pode ser efetuada em tempo proporcional a n2/4.

Assim, a complexidade ***f(n)*** desse algoritmo pode ser expressa pela recorrência:

$$f(n) = c_1 \qquad n \le 2$$
$$f(n) = 8f(n/2) + c_2 n^2 \qquad n > 2 \quad (c_1 \text{ e } c_2 \text{ constantes})$$

A solução dessa recorrência resultará $f(n = \Theta(n^3)$ e, portanto esse método não é mais rápido que o algoritmo trivial (também de complexidade cúbica).

Observando que as multiplicações matriciais ($\Theta(n^3)$ versus $\Theta(n^2)$), *Strassen* propôs uma abordagem recursiva diferente para calcular os C_{ij} com menos multiplicações e, possivelmente, mais somas matriciais.

Seu método propõe, após a decomposição vista em (1),

o cálculo das submatrizes D_i , $1 \le i \le 7$, como vemos em (3) a seguir:

$$D_1 = A_{11}(B_{12} - B_{22}) \qquad D_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$D_2 = A_{22}(B_{21} - B_{11}) \qquad D_6 = (A_{12} - A_{22})(B_{21} + B_{22}) \quad (3)$$
$$D_3 = (A_{11} + A_{12})B_{22} \qquad D_7 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$D_4 = (A_{21} + A_{22})B_{11}$$

Esse cálculo exige apenas 7 multiplicações e 10 somas ou subtrações.

Para o cálculo dos C_{ij} empregamos 8 somas ou subtrações como listado em (4), abaixo:

$$C_{11} = D_5 + D_2 - D_3 + D_6 \qquad C_{12} = D_1 + D_3$$
$$C_{21} = D_4 + D_2 \qquad C_{22} = D_5 + D_1 - D_4 + D_7 \qquad (4)$$

O algoritmo de Strassen produz seguinte recorrência:

$$f(n) = c_1 \qquad n \le 2$$
$$f(n) = 7f(n/2) + c_2 n^2 \quad n > 2 \quad (c_1 \text{ e } c_2 \text{ constantes})$$

Lembrando que consideramos ***n*** uma potência de 2, trabalhamos a recorrência para obter a solução:

$$f(n) = c_2 n^2\{1 + 7/4 + (7/4)^2 + \ldots (7/4)^{lg(n-1)}\} + 7^{lg(n)}f(1)$$

$$\leq C \cdot n^2 (7/4)^{lg(n-1)}\} + 7^{lg(n)} = \Theta(n^{lg7}) \qquad (\text{C é uma constante})$$

Note que este algoritmo também pode ser resolvido de acordo com o 1º caso, $a > b^k$, apresentado na solução do teorema sobre as recorrências dos algoritmos de Divisão e Conquista. Em relação à expressão inicial $f(n) = af(n/b) + O(n^k)$ identificamos as constantes:

$$a = 7, \quad b = 2 \quad \text{e} \quad k = 2, \text{com o resultado } f(n) = \Theta(n^{\lg 7}) = \Theta(n^{2,81}).$$

Esse algoritmo é portanto assintóticamente mais rápido do que o algoritmo trivial, porém do ponto de vista prático o *algoritmo de Strassen* não é escolhido para a multiplicação de matrizes por razões como:

- não ser tão estável numericamente quanto o trivial,
- as constantes do método consomem mais tempo que as do algoritmo trivial, etc.

Estatísticas de Ordem

A i-ésima estatística de ordem de um conjunto de ***n*** elementos é o i-ésimo menor elemento. Por exemplo, o mínimo de um conjunto de elementos é a primeira estatística de ordem (i=1), e o máximo é a *n*-ésima estatística de ordem (***i=n***). Informalmente, uma mediana é o "ponto médio" do conjunto: se n é ímpar então a mediana é única (i=(n+1)/2), senão com n par teremos uma mediana superior (i=$\lceil$(n+1)/2$\rceil$) e outra inferior (i=$\lfloor$(n+1)/2$\rfloor$) - para simplificar podemos adotar só a mediana inferior. Estamos interessados em resolver o seguinte problema:

Dada uma lista de n elementos, qual é o k-ésimo elemento mínimo, para um dado k onde $1 \leq k \leq n$*?*

Por exemplo, na *lista = {2,6,1,6,8,3}* o número *"6"* é o 4º e também o 5º valor mínimo. Mais formalmente, o *k-ésimo mínimo* é um elemento x na lista tal que existem no máximo *(k-1)* valores de i para os quais $Mi < a$ e existem pelo menos k valores de i para os quais $Mi \leq a$.

Este problema pode ser resolvido através da ordenação da lista em ordem crescente, e daí determinar o *k-ésimo* elemento. Mas isto requer um tempo mínimo proporcional a *O(nlgn)* para as comparações. Para aplicação dessa estratégia podemos usar uma solução que roda em tempo linear!

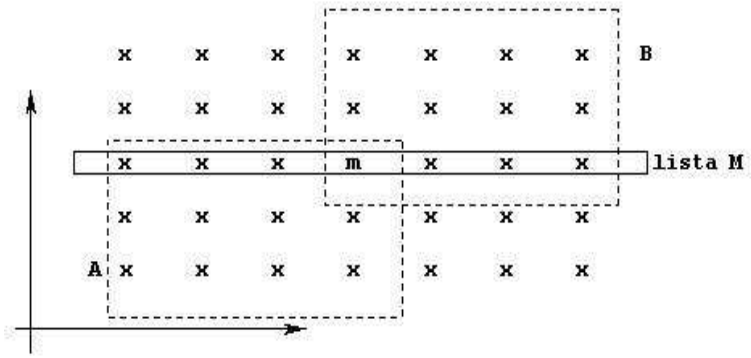
Quando $k = \lceil n/2 \rceil$tem-se um caso importante onde se determina a mediana de uma lista. A idéia básica é a de dividir a lista L em três sub-listas L_1, L_2 e L_3 de acordo com um elemento m de L, de tal forma que L_1 contenha todos os elementos menores do que m, L_2 contenha os elementos iguais a m, e L_3 os elementos maiores do que m. Pelo comprimento destas sub-listas, podemos determinar em que sub-listas o *k-ésimo* mínimo ocorre em L_1. Desta forma, podemos substituir a instância L por uma instância L_1 menor do que L.

Deve-se ter um cuidado especial na maneira de se determinar o elemento ***m***. Uma vez que queremos um algoritmo linear, devemos determinar m em tempo linear. Além disso, queremos que as sub-listas L_1 e L_3 tenham, cada uma, um comprimento máximo igual a uma função fixa do comprimento de L. A lista L é subdividida em $\lceil n/5 \rceil$sub-listas com cinco elementos cada. Cada sub-lista é ordenada e as medianas das sub-listas são determinadas para formar uma lista das medianas M. Como M contém apenas $\lceil n/5 \rceil$elementos, podemos calcular a sua mediana cinco vezes mais depressa do aquela de L.

Vejamos o algoritmo:

```
Seleção(k,L
 1  se n < 50
 2     então  Ordene(L)
 3            devolva o k-ésimo elemento de L
 4  Divida(L em ⌈n/5⌉ listas de 5 elementos)
 5  Ordene(as n/5 listas, separadamente)
 6  M[1..n/5] ← mediana de cada lista de 5 elementos
 7  m ← Seleção(|M|/2, M)
 8  L1 ← elementos menores que m da lista L
 9  L2 ← elementos iguais a m da lista L
10  L3 ← elementos maiores que m da lista L
11  se(|L1| ≥ k)então devolve Seleção(k,L1)
12               senão se (|L1|+|L2|) ≤ k
13                        então devolva m
14                        senão devolva Seleção(k-|L1|-|L2|,L3)
```

Ilustramos a seguir o fato de que pelo menos um quarto dos elementos de L serem menores ou iguais a m, e pelo menos um quarto dos elementos serem maiores ou iguais a m. Cada coluna representa uma das $\lceil n/5 \rceil$ sub-listas de 5 elementos ordenadas; os pontos na linha inferior representam os mínimos de cada sub-lista. Os elementos de A (e de B) são menores, (ou maiores) ou iguais a m.



Teorema: *dada uma lista L de n elementos, e um inteiro k, 1 ≤ k ≤ n, o algoritmo Seleção calcula o k-ésimo mínimo de L corretamente, em tempo proporcional a O(n).*

Prova: a certificação de *Seleção* pode ser feita por indução em ***n***.

Seja $f(n)$ o tempo de processamento do algoritmo Seleção.

A lista M das medianas contém no máximo n/5 elementos e então a execução da linha 7 requer um tempo $f(n/5)$ no máximo.

Como observamos, os elementos de B na ilustração não pertencem à sub-lista L_1.

Por outro lado, existem pelo menos $\lfloor n/10 \rfloor$ colunas, conforme ilustração, que contribuem com três elementos cada, para formação de B, isto é, B contém pelo menos $\lfloor 3n/10 \rfloor$ elementos.

Conclui-se daí que L_1 contém no máximo $n - \lfloor 3n/10 \rfloor$ elementos. Para $n \geq 50$, pode-se verificar que $n - \lfloor 3n/10 \rfloor < 3n/4$.

Assim, a execução do comando da linha 11 requer, no máximo um tempo $f(3n/4)$. Analogamente, ao comando da linha 13 requer no máximo um tempo $f(3n/4)$.

Como outras linhas requerem tempo $O(n)$, tem-se, para uma constante C:

$$f(n) \leq Cn, \text{ para } n \leq 49;$$

$$f(n) \leq f(n/5) + f(3n/4) + Cn, \text{ para } n \geq 50.$$

Para completar a demonstração, vamos provar por indução em n que f(n) deduzido acima, satisfazem:

$$f(n) \leq 20\,Cn$$

A verdade desta hipótese é óbvia para $n \leq 49$.

Vamos supor que também seja válida para valores menores que n, e provaremos sua validade para n.

Das deduções de $f(n)$, temos:

$$\begin{aligned} f(n) &\leq f(n/5) + f(3n/4) + Cn \\ &\leq 20C\,(n/5 + 3n/4) + Cn \qquad \text{(por hipótese)} \\ &= 20\,Cn. \end{aligned}$$

A pergunta que se pode fazer agora é: *por que as sub-listas têm 5, e não outro número fixo de elementos?*

Existem outros números além de "5" para os quais o algoritmo desejado ainda seria linear.

Note ainda que *Seleção* é aplicado recursivamente sobre duas listas: L_2 e L_1, ou L_2 e L_3,.

A soma dos comprimentos destas duas listas deve ser menor do que n para que o algoritmo continue linear.

Identificando a solução desse algoritmo com a solução do teorema sobre o tempo de processamento dos algoritmos de Divisão e Conquista, encontramos um exemplo do 3º caso, $a < b^k$.

Veremos um último exemplo num algoritmo de pesquisa.

Máximo e Mínimo

Problema: *dada uma lista A com n elementos, determinar o maior e o menor elemento da lista.*

Um algoritmo incremental, projetado por indução, para esse problema faz *2n-3* comparações: fazemos uma comparação no caso base e duas no passo... (confira!)

Usando a estratégia de divisão e conquista podemos melhorar um pouco o desempenho:

- Divida a lista em dois subconjuntos de mesmo tamanho, respectivamente A_1 com $\lfloor n/2 \rfloor$ elementos e A_2 com $\lceil n/2 \rceil$ elementos e, solucione os subproblemas;
- O máximo da lista A é o máximo dos máximos de A_1 e A_2 e o mínimo de A é o mínimo de A_1 e A_2.

Apresentamos o algoritmo abaixo que tem na entrada a lista armazenada no vetor A[1..n] e os índices *e* e *d* dos limites em análise e, na saída o par (max, min) desejado.

```
MaxMin(A, e, d)
 1  se d - e <= 1
 2    então se A(e) > A(d)
 3            então max ← A(e)
 4                  min  ← A(d)
 5            senão max ← A(d)
 6                  min ← A(e)
 7    senão m ← ⌊(d+e)/2⌋
 8          (max1, min1) ← MaxMin(A, e, m)
 9          (max2, min2) ← MaxMin(A, m+1, d)
10         se max1 > max2
11            então max ← max1
12            senão max ← max2
13          se min1 < min2
14            então min ← min1
15            senão min ← min2
16  devolve (max, min)
```

Na análise desse algoritmo vamos considerar o número de comparações efetuados para determinar ***max*** e ***min***, resultando a recorrência:

$$T(n) = 1 \qquad se \quad n \leq 2$$
$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 2 \qquad se \quad n > 2$$

Como podemos observar, não podemos empregar o teorema sobre as recorrências para obter uma fórmula fechada.

Assim, vamos supor que o número de elementos ***n*** é uma potência de 2, isto é $\boldsymbol{n = 2^k}$ para ***k*** inteiro, e trabalhar a relação:

$$T(n) = 2T(n/2) + 2 = 2[2T(n/4) + 2] + 2 = 2[2[2T(n/8) + 2] + 2] + 2 = \ldots = 2^k + [2^{k-1} - 2]$$

Obtendo a solução: $T(n) = 3n/2 - 2$

Assintóticamente, os dois algoritmos para esse problema são equivalentes, ambos apresentam solução: $T(n) = \Theta(n)$.

Entretanto, o algoritmo de divisão e conquista permite que menos comparações sejam feitas, cerca de 25% a menos. A estrutura hierárquica de comparações no retorno da recursão evita comparações desnecessárias.

Exercícios

1. Considere dois conjuntos de números inteiros A e B com *m* e *n* elementos, respectivamente. Eles não estão necessariamente ordenados, e assuma também que m ≤ n. Mostre como podemos computar A∪B e A∩B em um tempo proporcional a O(n lg m).

2. Considere como entrada um vetor com n números inteiros, onde cada elemento tem seu valor no intervalo [0,N], para algum N >> n.

Elabore um algoritmo que no pior caso roda em tempo proporcional a $O\left(\frac{\lg N}{\lg n}\right)$ e verifica se esses *n* números são diferentes. Seu algoritmo deve usar somente um espaço proporcional a *O(n)*.

3. Escreva um algoritmo que recebe um vetor A[e..d] e devolve o valor da soma dos elementos A[*i*] tais que e ≤ i ≤ d e i é *par*. A pilha de execução recursiva de seu algoritmo deverá ter profundidade máxima da ordem de lg(d–e+1).

4. Prove que:

i) a profundidade de recursão máxima da pilha de execução recursiva do algoritmo MaxMin é $\lfloor lgn \rfloor+1$, quando a lista de entrada possui *n* elementos,

ii) que a complexidade de MaxMin é $T(n) \leq \lceil 3n/2-2 \rceil$ para valores de *n* que não são potências de 2, *n* ímpar

5. Pode-se desenvolver um algoritmo não – recursivo para determinar o máximo e o mínimo de uma lista baseado na seguinte idéia: comparam-se pares de elementos consecutivos e, em seguida, compara-se o maior no par com o máximo temporário, e o menor do par com o mínimo temporário. Desenvolva esse algoritmo, analise e compare com nossa versão elaborada com a estratégia de divisão e conquista.

6. Projete um algoritmo para pesquisa de um elemento *x* em um vetor ordenado A[1..n] modificando a estratégia de divisão que empregamos na Busca Binária, isto é, a divisão deverá separar em dois conjuntos com n/3 e 2n/3 elementos. Projete essa "Busca Ternária", analise sua complexidade e correção e depois a compare com a Busca Binária.

Programação Dinâmica

A programação dinâmica, como o método de divisão e conquista, resolve problemas combinando as soluções para subproblemas. Os algoritmos de divisão e conquista particionam o problema em subproblemas independentes, resolve esses subproblemas recursivamente, combinando suas soluções para resolver o problema original.

Em contraste, a programação dinâmica é aplicável quando os subproblemas não são independentes, isto é, quando os subproblemas compartilham subproblemas. Nesse contexto, um algoritmo de dividir e conquistar trabalha mais que o necessário, resolvendo repetidamente os subproblemas comuns.

Um algoritmo de programação dinâmica resolve cada subproblema uma vez só e então grava sua resposta em uma tabela, evitando assim o trabalho de recalcular a resposta toda vez que o subproblema é encontrado. Nesse contexto, "programação" refere-se a uma tabulação e não à codificação de um algoritmo.

É aplicada a problemas de otimização com muitas soluções possíveis. Cada solução tem um valor, e desejamos encontrar uma solução com um valor ótimo (mínimo ou máximo). Chamamos tal solução uma *solução ótima* para o problema, em lugar de **a solução**, pois podem existir várias que alcançam o valor ótimo.

O desenvolvimento de um algoritmo de programação dinâmica pode ser desmembrado em uma seqüência de quatro etapas:

1. caracterizar a estrutura de uma solução ótima;
2. definir recursivamente o valor de uma solução ótima;
3. calcular o valor de uma solução ótima em um processo de baixo para cima(botton-up);
4. construir uma solução ótima a partir de informações calculadas.

As três primeiras etapas formam a base da programação dinâmica para um problema, enquanto a última pode ser omitida se apenas o valor de uma solução ótima é exigido. Algumas vezes. ao executar a quarta etapa, informações adicionais são mantidas durante a computação da terceira etapa para facilitar a construção de uma solução ótima.

A programação dinâmica é uma poderosa forma de desenvolver algoritmos que providencia um ponto de partida para uma solução. É essencialmente o paradigma da divisão e conquista resolvendo problemas mais simples primeiro. A diferença importante é que os problemas mais simples não representam uma divisão clara do original.

Porque subproblemas são repetidamente resolvidos, é importante guardar suas soluções para empregá-las em etapas posteriores. Em alguns casos a solução pode ser melhorada e, em outros a técnica apresenta-nos a melhor solução encontrada.

Ilustramos a seguir a utilização deste método aplicado a quatro problemas:

Números de Fibonacci

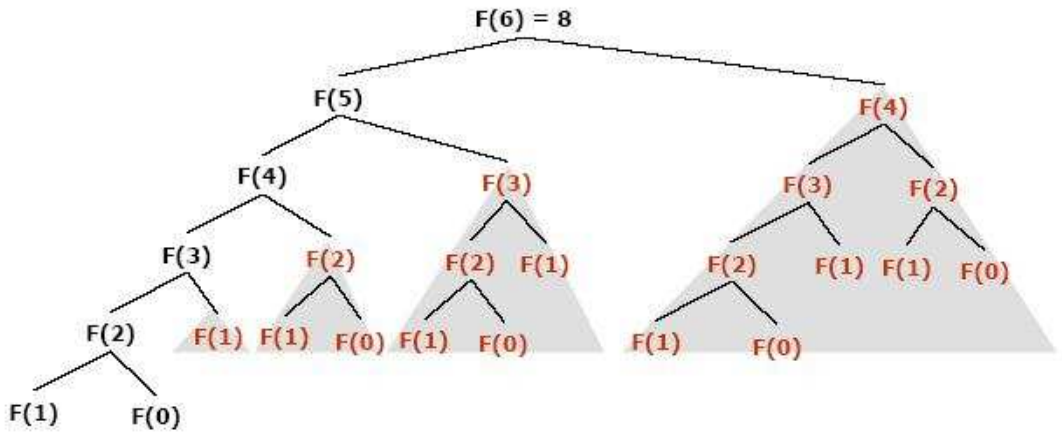
Iniciamos os exemplos com o cálculo dos números de Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21,... calculados a partir da relação:

$$F(0) = 0 \qquad F(1) = 1 \qquad F(n) = F(n-1) + F(n-2) \ (\text{para } n \geq 2)$$

Para a determinação de F[n], podemos aplicar o algoritmo recursivo:

```
Fibonacci_rec (n)
1  se n ≤ 1
2    então devolve n
3    senão  a  ←  Fibonacci_rec (n-1)
4           b  ←  Fibonacci_rec (n-2)
5           devolve a + b
```

Observe o calculo de F(6):



Analisando o desempenho deste algoritmo, podemos computar o tempo consumido, *verificando o número de somas* que serão realizadas conforme a relação de recorrência:

$$T(0) = 0 \text{ para } n = 0$$
$$T(1) = 0 \text{ para } n = 1$$
$$T(n) = T(n-1) + T(n-2) + 1 \text{ para } n \geq 2$$

Resolvendo essa relação para n = 2, 3, 4, 5, 6, 7, 8, 9, ... , n verificamos que:

$$T(n) \geq \Omega((3/2)^n) \quad \text{para } n \geq 6.$$

Prova:

Vamos aplicar a relação para n = 6, 7 e "n"...

$$T(6) = T(5) + T(4) + 1 = 12 > (3/2)^6$$

$T(7) = T(6) + T(5) + 1 = 20 > (3/2)^7$

$T(n) = T(n-1) + T(n-2) + 1$

$T(n) \geq^{hip} (3/2)^{n-1} + (3/2)^{n-2} + 1 = (3/2 + 1)(3/2)^{n-2} + 1$

$T(n) \geq (5/2)(3/2)^{n-2} = (10/4)(3/2)^{n-2} \geq (9/4)(3/2)^{n-2} = (3/2)^2(3/2)^{n-2}$

$T(n) = (3/2)^n$, ou seja, $T(n) = \Omega((3/2)^n)$

O comportamento assintótico terá uma solução de ordem exponencial: $T(n) = O(2^n)$.

Porque revolve o mesmo problema muitas vezes!

Vamos empregar a programação dinâmica, armazenando os resultados na "tabela" *F[0..n-1]*:

```
Fibonacci_progdin (n)
1  F[0] ← 0, F[1] ← 0
2  para i ← 2 até n
3         F[i] ← F[i-1]+F[i-2]
4  devolve  F[n]
```

Verificando o consumo de tempo, obtemos: $f(n) = O(n)$, um consumo linear em n .

Multiplicação de matrizes

Considere o cálculo do produto de *n* matrizes: $M = M_1 * M_2 * \ldots * M_n$; onde cada matriz M_i tem dimensões **b[i – 1]** linhas e **b[i]** colunas, $1 \leq i \leq n$.

O algoritmo trivial para se multiplicar uma matriz $p \times q$ por outra matriz $q \times r$ requer **pqr** operações aritméticas. Se considerarmos, por exemplo, o produto a seguir, com as dimensões indicadas:

$$M = \underset{200 \times 2}{M_1} * \underset{2 \times 30}{M_2} * \underset{30 \times 20}{M_3} * \underset{20 \times 5}{M_4}$$

Temos várias ordens possíveis de multiplicações para se obter M.

Por exemplo se a ordem é: $((M_1 * M_2) * M_3) * M_4$

o número total de operações necessárias é 152.000.

Por outro lado, se a ordem é: $M_1 * ((M_2 * M_3) * M_4)$

necessita-se de apenas 3400 operações!

Este exemplo mostra que a ordem em que as matrizes M_i são multiplicadas influi substancialmente no número total de operações (e, portanto, no tempo) necessárias para se calcular o produto **M**; e essa influência independe do algoritmo escolhido para se efetuar cada multiplicação matricial.

Queremos determinar qual a ordem das multiplicações que requer o mínimo número de operações, ou seja, encontrar uma seqüência ótima. Se fôssemos enumerar todas as ordens possíveis de multiplicações (ou todas as seqüências de decisões) com os respectivos números totais de operações necessárias, e em seguida escolhêssemos a seqüência ótima, o número total de operações seria uma função exponencial em n, o que é inviável para uma entrada com n grande, na prática.

Empregando o método de programação dinâmica, podemos reduzir drasticamente este número de operações para a determinação da seqüência ótima, como previsto no 1º passo desse método.

No 2º passo, procuramos definir o valor de uma solução ótima:

Seja M_{ij} o número total mínimo de operações necessário para se obter o produto

$$M_i * M_{i+1} * \ldots * M_j \quad (1 \leq i \leq j \leq n) \qquad (1)$$

Chamando **m[i,j]** de custo mínimo para essa multiplicação, temos **m[i,i] = 0** (2)

Consideremos agora, para $i \leq k < j$, o custo mínimo **m[i,k]** para o cálculo de **M'**, onde

$$M' = M_i * M_{i+1} * \ldots * M_k \,, \qquad (3)$$

E o custo mínimo **m[k+1,j]** para o cálculo de M", onde

$$M'' = M_{k+1} * M_{k+2} * \ldots * M_j \,, \qquad (4)$$

Note que (3) e (4) são subproblemas de (1).

Observemos que **M'** é uma matriz **b[i-1]×b[k]** e a matriz **M"** é uma matriz **b[k]×b[j]**.

Nestas condições, pode-se obter o produto em (1) pela multiplicação **M' × M"**.

E o custo total será então **m[i,k] + m[k+1,j] + b[i-1]b[k]b[j]** (5)

Como queremos que o custo em (5) seja mínimo, temos que perguntar:

Qual é o valor de k, *tal que* $i \leq k < j$, *que minimiza (5) ?*

Em outras palavras, para i < j, queremos determinar:

$$m[i,j] = \text{Mínimo}_{\,i \leq k < j} \{ m[i,k] + m[k+1,j] + b[i-1]b[k]b[j] \} \qquad (6)$$

As equações (2) e (6) constituem uma fórmula de recorrência típica do método de programação dinâmica. O algoritmo baseado nesta fórmula tem os seguintes passos iterativos:

- os **m[i,i]** são calculados, para $1 \leq i \leq n$;
- os **m[i,i+1]** são calculados, para $1 \leq i \leq n-1$;
- os **m[i,i+2]** são calculados, para **1 ≤ i ≤ n-2**;

... e assim por diante, os **m[i,j]** são calculados em ordem crescente da diferença **j-i**. Desta forma, os termos **m[i,k]** e **m[k+1,j]** em (6) já são conhecidos quando **m[i,j]** é calculado, porque:

$$j - i > \text{Máximo } \{k-i,\ j-k+1\}$$

Note que o processo descrito é uma sucessão de soluções de subproblemas correspondentes a (1), em ordem crescente do "tamanho" dos subproblemas.

Como frisamos no início da seção, as soluções ótimas para os subproblemas (que neste caso são *os* **m[i,i]**, **m[i, i+1]**, …) são obtidas e armazenadas para serem utilizadas na solução de subproblemas maiores.

Passando ao 3º passo da solução por Programação Dinâmica, podemos agora propor um algoritmo que recebe as dimensões das matrizes (*vetor* **b[0..n]** *com* **n+1** *elementos*):

```
Ordena(b[0..n])
1  para i ← 1 até n faça
2    m[i,i] ← 0
3  para c ← 2 até n-1 faça
4     para i ← 1 até n-c+1 faça
4        j  ←  i+c-1
5        m[i,j] ← ∞
6        para k ← i até j-1 faça
7             Q ← m[i,k] + m[k+1,j] + b[i-1]b[k]b[j]
8             se Q < m[i,j]
9               então m[i,j] ← Q
10                    s[i,j] ← k
11 devolve m e s
```

O algoritmo devolve o custo mínimo **m[1,n]** com o número ótimo de multiplicações, e a matriz ***s*** com os ***k´s*** intermediários determinados em cada mínimo **m[i,j]** calculado que viabiliza o produto na ordem em que foi determinado como veremos na simulação a seguir.

Note que **Ordena()** tem uma estrutura iterativa, e que o tempo para execução no pior caso é da ordem de $O(n^3)$, pois cada uma das iterações em **c**,**i** *e* **k** são executadas $O(n)$ *vezes*.

Simulamos a seguir a execução de **Ordena()** para o produto de matrizes do exemplo (*onde* **b[0]= 200, b[1]=2, b[2]=30, b[3]=20 e b[4]=5**), mostrando os valores de **m[i,j]** em linhas correspondentes a valores crescentes de ***j-i***, correspondendo a cada valor **m[i,j]** a expressão (5) e o ***k*** utilizado, bem como a ordem das multiplicações das matrizes.

j – i = 0	$m_{11} = 0$	$m_{22} = 0$	$m_{33} = 0$	$m_{44} = 0$
j – i = 1	$m_{12} = m_{11} + m_{22} + b_0b_1b_2$ $m_{12} = 12000$ (k = 1); $M_1 * M_2$	$m_{23} = m_{22} + m_{33} + b_1b_2b_3$ $m_{23} = 1200$ (k = 2); $M_2 * M_3$	$m_{34} = m_{33} + m_{44} + b_2b_3b_4$ $m_{12} = 3000$ (k = 3); $M_3 * M_4$	
j – i = 2	$m_{13} = m_{11} + m_{23} + b_0b_1b_3$ $m_{13} = 9200$ (k = 1); $M_1 * (M_2 * M_3)$	$m_{24} = m_{23} + m_{44} + b_1b_3b_4$ $m_{24} = 1400$ (k = 3); $(M_2 * M_3) * M_4$		
j – i = 3	$m_{14} = m_{11} + m_{24} + b_0b_1b_4$ $m_{14} = 3400$ (k = 1); $M_1 *((M_2 * M_3) * M_4)$			

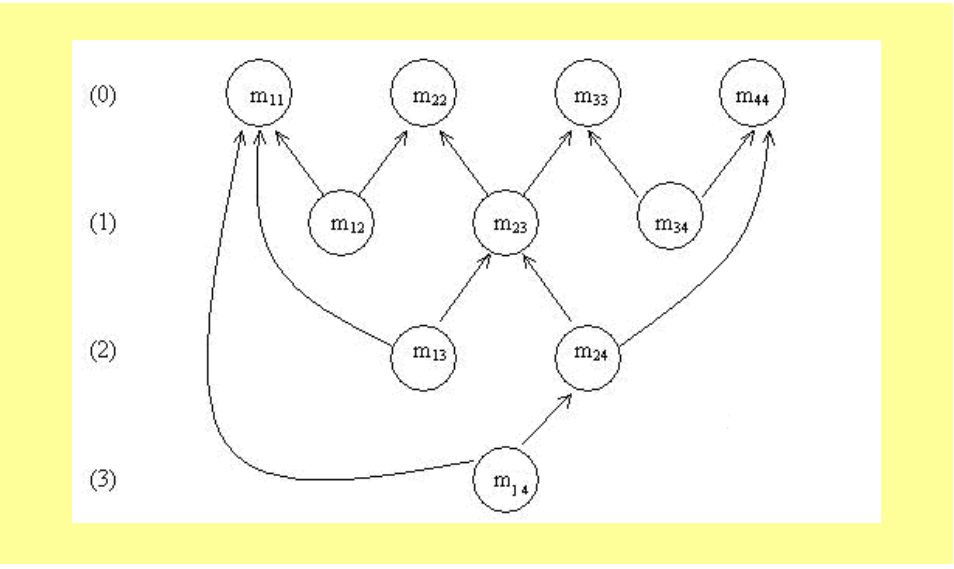
A partir dos valores de ***k*** utilizados para se obterem os correspondentes **m[i,j]**, como mostrado acima, pode-se calcular qualquer aresta (**v, w**) num grafo orientado que reflita o fato de que o valor ***m*** foi obtido a partir do valor **m'**.

Para demonstrar a ordem das multiplicações como demonstra a última linha da tabela acima, podemos empregar o algoritmo recursivo:

```
MostraOrdem(s, i, j)
1  se i = j
2    então escreve(" M",i)
3    senão escreve("( " )
4          MostraOrdem(s, i, s[i,j])
5          MostraOrdem(s, s[i,j]+1, j)
6          escreve(" )" )
```

Aplicando **MostraOrdem(**s, 1, n**)** obtemos: $M_1((M_2M_3)M_4)$.

Na ilustração abaixo mostramos um grafo, considerando os valores de ***k*** (anotados na simulação). Note que com o grafo também pode-se obter a ordem das multiplicações dos M_i**'s** correspondentes ao custo mínimo **m[1,n]**.



É interessante compararmos os mecanismos distintos de geração de subproblemas que existem em algoritmos.

Em algoritmos obtidos por programação dinâmica, os subproblemas menores são gerados antes dos subproblemas maiores, como ilustramos. Em contraste com algoritmos recursivos como vimos em exemplos anteriores que empregam a estratégia de divisão e conquista: os subproblemas maiores são gerados *antes* dos subproblemas menores.

Devido a essa distinção, os algoritmos recursivos são às vezes chamados de descendentes (em relação à ordem hierárquica de geração de subproblemas), e os algoritmos obtidos com a programação dinâmica são chamados algoritmos ascendentes.

Subseqüência Crescente Máxima

Outro exemplo de aplicação à programação será a determinação de uma subseqüência crescente máxima. Vamos enunciar o problema:

> "Suponha que *A* [1..n] é uma *seqüência* de números inteiros. Uma *subseqüência* de *A*[1..n] é a seqüência que sobra depois que um conjunto arbitrário de termos é apagado.
>
> Vale observar que um *segmento* de *A*[1..n] é o que sobra depois que apagamos um número arbitrário de termos no *início* de *A*[1..n] e um número arbitrário de termos no *fim* de *A*[1..n].
>
> Uma subseqüência B[1..k] de *A*[1..n] é crescente se B[1]≤...≤B[k]. Uma *subseqüência crescente é máxima* se não existe outra subseqüência crescente mais longa."

Exemplos:

Seqüência: **1, 2, 3, 9, 4, 5, 6** a subseqüência mais longa é: **1, 2, 3, 4, 5, 6.**

Outra seqüência: **9, 5, 6, 9, 6, 4, 7** têm solução: **5, 6, 6, 7**.

Note que **5,6,9** é uma seqüência crescente que não pode ser "prolongada", mas não é máxima.

Vamos empregar uma solução recursiva para determinar o comprimento máximo de uma seqüência *A*[1..n], supondo que *C*[j] é o comprimento de uma subseqüência crescente máxima de *A*[1..j] *dentre as que terminam com A*[j].

A solução do problema, *"achar o comprimento da subseqüência crescente máxima"*, será dada por **Máximo(C[1], C[2], ..., C[n])**.

Vejamos o algoritmo que nos dá essa solução, tem como entrada *A*[1..n] e devolve *C*[n]:

```
SSCmax_rec (A,n)
1  se n = 1
2     então devolve n
3     senão c ← 1
4           para i ← n-1 até 1 faça
5                 se A[i] ≤ A[n]
6                    então d ← SSCmax_rec(A,i)
7                          se d + 1 > c
8                             então c ← d+1
9  devolve c
```

Nosso algoritmo funciona, porém é muito ineficiente, pois processa **SSCmax_rec(A,i)** várias vezes para o mesmo valor de i, e como no exemplo anterior com *consumo de tempo exponencial.*

A melhora deste procedimento começa por armazenar as soluções dos subproblemas numa tabela, usaremos ***C*[1..n]**, observe o próximo algoritmo:

```
SSCmax_progrdin (A,n)
 1  para j ← 1 até n faça
 2       C[j] ← 1
 3       para i ← j-1 até 1
 4            se A[i] ≤ A[j] e C[i]=1 > c[j]
 5               então C[j] ← C[i]+1
 6  m ← 1
 7  para j ← 2 até n faça
 8       se m < C[j]
 9          então m ← C[j]
10 devolve m
```

Analisando o tempo consumido por este algoritmo, mais eficiente, encontramos $T(n) = O(n^2)$.

Problema do Caixeiro Viajante

Seja ***G* = (V, E)** um grafo orientado com custos positivos **c[i,j]** associados às arestas **(i, j)**; quando não existir uma aresta **(i, j)**, convencionemos que *c[i,j] seja infinito*, e também supomos **|V| = n > 1**. Uma viagem em *G* é um circuito(orientado) que contém cada vértice de V uma e uma só vez. *A soma dos custos* das arestas na viagem é chamada de *custo da viagem*. *O problema do caixeiro* viajante consiste em achar uma *viagem mínima*, sito é, entre todas as viagens possíveis em *G*, uma *viagem de custo mínimo*; observe que pode existir mais do que uma de tais viagens.

Podemos descobrir várias aplicações para este problema, por exemplo, a rota de um representante comercial (ou um caixeiro viajante) que periodicamente tem que visitar ***n*** cidades para efetuar certas tarefas em cada cidade (por exemplo, promover a empresa representada, ou realizar vendas). As ***n*** cidades podem ser vistas como os *vértices em um grafo*, sendo que o custo associado à aresta **(i, j)** seria o custo de se deslocar da cidade **i** até a cidade **j**. Quer-se então encontrar uma viagem de visita às ***n*** cidades que seja a *mais econômica* para o caixeiro viajante.

Outro exemplo, mais abstrato: considere um *conjunto de máquinas* que fabricam ***n*** produtos distintos, de forma cíclica. Suponhamos que todas as máquinas podem estar fabricando um produto ***i***, e serem *alterados* para produzirem outro produto ***j***, pagando-se um custo c_{ij} para se efetuar tal alteração: o custo para ajustar a máquina. Queremos determinar qual a ordem cíclica em que os ***n*** produtos devem ser fabricados, de tal maneira que a soma dos custos de alterações das máquinas sejam mínimos.

E mais um exemplo de instância deste problema do caixeiro viajante: um *braço mecânico móvel* que deve distribuir amostras de material radioativo entre ***n*** pontos diferentes dentro de uma câmara. Deseja-se então determinar qual a seqüência que minimiza o tempo necessário para o braço efetuar as *n* distribuições.

Vamos resolver o problema para qualquer instância aplicando programação dinâmica. Um algoritmo trivial para se resolver o problema consiste em simplesmente enumerar todas as **n!** permutações dos **n** vértices em V, calcular os custos de cada viagem correspondente a cada uma das permutações, e escolher uma viagem de custo mínimo. Mas o valor de **n!** torna este algoritmo inviável mesmo para valores pequenos de **n**. Este problema é um exemplo entre vários problemas combinatoriais para os quais a solução desejada pertence a um conjunto de **n!** permutações.

Com a programação dinâmica, reduzimos drasticamente o número de permutações dos **n** vértices entre os quais a viagem mínima deve ser procurada. Supomos em princípio que uma viagem começa e termina no vértice 1, após visitar cada um dos vértices 2, 3, ... , n uma única vez.

Note que qualquer viagem é constituída por uma aresta **(1, k), 2 ≤ k ≤ n**, e um *caminho R* de **k** até **1**. E este caminho R visita cada um dos vértices **V – { 1, k}** uma só vez. Se a viagem é considerada é de custo mínimo, o caminho R deve ser necessariamente de custo mínimo. Seja **m(i, *C*)** o *custo* do caminho do vértice **i** até **1**, e que visita todos os vértices no conjunto ***C***, assim:

$$m(1, V - \{1\}) = \text{Mínimo}_{2 \le k \le n} \{ c[1,k] + m(k, V - \{1, k\}) \} \quad (1)$$

Se conhecemos os valores de **m(k, V – {1, k})**, para todo **2 ≤ k ≤ n**, então resolve-se com (1) o problema do caixeiro, e podemos generalizar (1), escrevendo:

$$m(i, C) = \text{Mínimo}_{j \in C} \{ c[i,j] + m(j, C - \{j\}) \} \quad (2)$$

ou seja, o caminho mínimo do vértice **i** até **1**, que visita todos os vértices em ***C*** é constituído por **(i,j)** e um caminho de **j** até **1** que visita todos os vértices em ***C*** – **{j}**, para uma escolha adequada de **j** em ***C***.

Os valores de *m* necessários em (1) para resolver o problema podem ser obtidos através de (2) de uma forma iterativa ascendente, nas etapas:

- para **|*C*| = 0**, **m(i,∅) = c[i,1]**, onde **1 < i ≤ n**.
- para **|*C*| = 1**, **m(i,{j}) = c[i,j] + m(j,∅) = c[i,j] + c[j,1]**, onde **1<i≤ n , 1<j≤ n e i≠j**.
- para **|*C*| = 2, 3, ..., n–2** determinam-se os valores de **m(i,*C*)** através de (2), utilizando-se os valores de ***m*** calculados nas iterações anteriores. Deve-se considerar **1<i≤n**, e conjuntos ***C*** tais que ***C*** ∩ **{1,i}** = ∅.

Seguindo este roteiro podemos chegar ao algoritmo, que tem como entrada a matriz $C_{n\times n}$ com os custos de viagens entre dois vértices e como saída: o custo mínimo, e o vetor **s[1..n]** que guarda o "vértice j" da expressão (2) nas etapas 2 e 3 (no intervalo **s[1..n-1]**), e o vértice após "**1**" de ***C*[1,j]**, elemento **s[n]**, na expressão (1).

```
CustoMinimo(C)
 1  para i ← 2 até n faça
 2       Min[i,1] ← C[i,1]        (custo com |C| = 0, m[i,∅])
 3  para k ← 1 até n-1 faça       (custo min com |C| = 1,...,n-2)
 4       para i ← 2 até n faça
 5            Min[i,k] ← ∞
 6            para j ← 2 até n faça
 7                 se i≠j e (C[i,j] + Min[i,k-1])< Min[i,k]
 8                    então Min[i,k] ← C[i,j] + Min[i,k-1]
 9                          s[k] ← j
10  Q ← Min[2,n-1]
11  para i ← 3 até n faça
12       se Q > Min[i,n-1]
13          então Q ← Min[i,n-1]
14                s[n] ← i
15  Custo_min ← C[1,s[n]] + Q
16  devolve Custo_min e s
```

Ilustrando a aplicação através do um grafo orientado, mostrado na matriz 4×4:

	1	2	3	4
1	0	2	10	7
2	20	0	9	1
3	1	5	0	15
4	7	12	3	0

Temos os custos nos elementos C[i,j] da matriz, por exemplo **C[1,2] = 2** entre vértices **1** e **2**.

Determinando os valores de ***m***, temos pela expressão (2):

- na etapa 1:

$m(2, \varnothing) = C[2,1] = 20$

$m(3, \varnothing) = C[3,1] = 1 \quad m(4, \varnothing) = C[4,1] = 7$

- na etapa 2:

$m(2, \{3\}) = C[2,3] + C[3,1] = 20 + 1$

$m(2, \{4\}) = C[2,4] + C[4,1] = 1 + 7$

$m(3, \{2\}) = C[3,2] + C[2,1] = 5 + 20$

$m(3, \{4\}) = C[3,4] + C[4,1] = 15 + 7$

$m(4, \{2\}) = C[4,2] + C[2,1] = 12 + 20$

$m(4, \{3\}) = C[4,3] + C[3,1] = 3 + 1$

- na etapa 3:

$$m(2, \{3,4\}) = \min \{C[2,3] + m(3, \{4\}), C[2,4] + m(4, \{3\})\} = 5 \quad (j = 4)$$

$$m(3, \{2,4\}) = \min \{C[3,2] + m(2, \{4\}), C[3,4] + m(4, \{2\})\} = 13 \quad (j = 2)$$

$$m(4, \{2,3\}) = \min \{C[4,2] + m(2, \{3\}), C43 + m(3, \{2\})\} = 28 \quad (j = 3)$$

e finalmente pela expressão (1):

$$m(1, \{2,3,4\}) = \min \{C[1,2] + m(2, \{3,4\}), C[1,3] + m(3, \{2,4\}), C[1,4] + m(4, \{2,3\})\}$$

$$= \min \{2 + 5, 10 + 13, 7 + 28\} = 7 \quad (j = 2).$$

O custo do caminho neste exemplo é **7**, o caminho propriamente dito pode ser obtido se, em cada cálculo de **m(i,C)**, conhecemos o valor de **j** que minimiza (2). Como se pode observar acima, com **j = 2** no cálculo de **m(1,{2,3,4})** a primeira aresta do caminho mínimo é **(1,2)**. Prosseguindo, como **j = 4** no cálculo de **m(2,{3,4})**, a segunda aresta é **(2,4)**. A terceira aresta é **(4,3)**, pois **j = 3** no cálculo de **m(4,{3})**. Portanto a viagem mínima neste exemplo é **(1, 2, 4, 3, 1)** com custo **7**.

Analisando agora a complexidade deste algoritmo começamos estimando o número de valores de **m(i, C)** que devem ser calculados através de (2). Para cala valor de **C**, existem **n-1** possibilidades de escolha de **i**. Como **C** não deve conter **1** e **i**, o número de conjuntos, distintos, de tamanho **k** será: $\binom{n-2}{k}$

portanto, o número que queremos estimar é $$\sum_{k=0}^{n-2}(n-1)\binom{n-2}{k} = (n-1)2^{n-2} \qquad (3)$$

O número de comparações e somas necessárias para o cálculo de um valor **m(i, C)**, com **|C|=k**, é ***O(k)***. Assim o tempo de processamento, baseado em (2), é da ordem de $O(n^2 2^n)$, isto é, exponencial da ordem de **n** ao quadrado vezes dois elevado a ***n***. Note que este algoritmo é mais rápido do aquele que enumera as **n!** permutações dos ***n*** vértices.

Exercícios

1. Escreva um algoritmo de programação dinâmica que determine uma subseqüência crescente máxima de a[1..n], e não só o comprimento de uma tal subseqüência como foi apresentado anteriormente.

2. Mostre que são necessários exatamente n-1 pares de parêntesis para especificar exatamente a ordem de multiplicação da cadeia de matrizes: $M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$.

3. Desenhe a árvore de recursão para o algoritmo *Mergesort* aplicado a um vetor de 16 elementos. Por que a técnica de programação dinâmica não é capaz de acelerar o algoritmo?.

4. Considere o seguinte algoritmo para determinar a ordem de multiplicação de uma cadeia de matrizes $M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$ de dimensões $b_0, b_1, \ldots, b_n$: primeiro escolha *k* que minimize b_k; depois, determine recursivamente as ordens de multiplicação de $M_1 \cdot \ldots \cdot M_k$ e $M_{k+1} \cdot \ldots \cdot M_3$. Esse algoritmo produz uma ordem que minimiza o número total de multiplicações escalares? E se *k* for escolhido de modo a maximizar b_k? E se *k* for escolhido de modo a minimizar b_k?

5. Escreva um algoritmo de programação dinâmica para o seguinte problema: dados números inteiros não negativos $w_1, \ldots, w_n$ e *W*, encontrar um subconjunto K de {1, ..., n} que satisfaça $\sum_{k \in K} w_k \leq W$ e maximize $\sum_{k \in K} w_k$.(imagine que $w_1, \ldots, w_n$ são os tamanhos de arquivos digitais que voce deseja armazenar em um disquete de capacidade W.)

6. Seja S o conjunto de raízes quadradas dos números 1, 2, ..., 500. Escreva e teste um programa que determine uma partição (A, B) de S tal que a soma dos números em A seja tão próxima quanto possível da soma dos números em B. Seu algoritmo resolve o problema? Ou só dá uma solução aproximada? Uma vez calculados A e B seu progrma deve imprimir a diferença entre a soma de A e a soma de B e depois imprimir a lista dos quadrados dos números em um dos conjuntos.

7. N tarefas devem ser executadas em duas máquinas A e B. Se a tarefa i for processada na máquina A, serão consumidas a_i unidades de tempo; e se for processada na máquina B, b_i unidades de tempo. É possível que $a_i \geq b_i$, para algum i, e $a_j < b_j$ para algum $j \neq i$. Obtenha um algoritmo por programação dinâmica para determinar o mínimo tempo necessário para se executar todas as tarefas, supondo-se que as tarefas não podem ser divididas entre as duas máquinas. Deve-se iniciar também como determinar a atribuição das tarefas a cada máquina que minimiza o tempo mínimo.

Método Guloso

A palavra "guloso" na denominação desses algoritmos é uma tentativa de traduzir "greedy", de greedy algorithms, que alguns preferem traduzir por "gananciosos". É um método simples que pode ser aplicado a vários problemas, em geral do tipo: calcular um subconjunto S de um conjunto de n objetos. Este S deve satisfazer certas condições, e é denominado solução viável para o problema. Associado a cada solução viável, tem-se um valor f(S), onde f é uma função chamada função objetivo, cujo significado é dado na definição do problema. Entre todas as soluções viáveis S, deseja-se aquela que tem valor f(S) mínimo (ou máximo); e esta solução é denominada solução ótima.

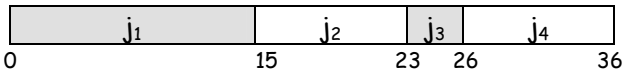
Pelo método guloso, pode-se obter um algoritmo iterativo que construa uma solução ótima S em etapas. Em cada etapa, o algoritmo possui uma solução viável parcial P que é um subconjunto (menor do que a solução ótima) satisfazendo as condições do problema; e em cada etapa, o algoritmo escolhe um novo objeto o; se $P \cup \{o\}$ é viável, então o é incluído em P, senão o é rejeitado. Assim o algoritmo constrói uma solução viável parcial cada vez maior, até obter uma solução final que seja ótima. A escolha de um a novo objeto o em cada etapa é orientada por alguma medida de otimização que forneça uma garantia de que a solução final seja ótima; esta medida de otimização se relaciona, direta ou indiretamente, com a função objetivo. Vamos ilustrar a aplicação do método para resolver dois problemas:

Planejando a execução de tarefas

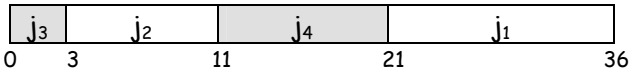
Iniciamos com um problema simples de organização de tarefas, que supomos listadas como $j_1, j_2, \ldots, j_n$ com tempos de processamento conhecidos e iguais a $t_1, t_2, \ldots, t_n$ respectivamente. Completando a identificação do problema considere que temos apenas um processador para executar cada um dos j_i programas e também que a execução uma vez iniciada cessará apenas quando for concluída, em geral após o respectivo tempo t_i. Para uma primeira organização considere a tabela abaixo com quatro programas:

Tarefa	j_1	j_2	j_3	j_4
Tempo	15	8	3	10

Uma primeira tentativa de organizar a execução poderia considerar a ordem de chegada, em que cada programa foi submetido para rodar, identificados pelo índice de j, e assim teríamos a conclusão de j_1 em 15(unidades de tempo), de j_2 em 23, j_3 em 26 e finalizando a série com j_4 em 36, como vemos na ilustração:



Com esse planejamento temos um tempo médio para conclusão de tarefas de (15+23+26+36)/4 = 25. Uma melhoria deste tempo médio pode ser alcançada com uma nova seqüência que considera o tempo de execução para determinar a ordem de início:



Priorizando os menores tempos temos um tempo médio de 17,75. A partir das duas soluções observadas podemos inferir que nosso problema terá sempre uma ótima organização sempre que utilizar o segundo critério. Podemos então generalizar nossa solução considerando a lista de tarefas na entrada: $j_{i1}, j_{i2}, \ldots, j_{in-1}, j_{in}$. O primeiro programa conclui

seu processamento em tempo t_{i1}, o segundo em tempo $t_{i1} + t_{i2}$, o terceiro após um tempo $t_{i1} + t_{i2} + t_{i3}$, e assim até completar a lista com custo total de:

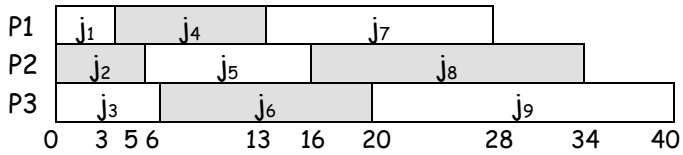
$$C = \sum_{k=1}^{n} (n-k+1)t_{i_k} \quad \text{portanto} \qquad C = (n+1)\sum_{k=1}^{n} t_{i_k} - \sum_{k=1}^{n} kt_{i_k}$$

Note que na equação acima, a primeira soma é independente da ordem em que as tarefas são apresentadas, e somente a Segunda soma afeta o custo total. Da mesma forma que ilustramos acima, suponha que tenhamos duas tarefas x > y na lista tais que $t_{ix} < t_{iy}$. Os cálculos apresentados acima mostram que se invertermos a ordem de x e y teremos um aumento da segunda soma e portando uma redução no custo total (ou no tempo médio para conclusão das tarefas). Portanto qualquer planejamento de tarefas nos quais os tempos não sejam monotonicamente decrescentes serão sub-ótimos. Os melhores planos serão aqueles que priorizam sempre as tarefas com menor tempo previsto de processamento, ignorando vínculos como a ordem de chegada. Isto nos dá uma pista dos procedimentos adotados por sistemas gerenciadores que geralmente colocam programas menores no topo da lista de execução.

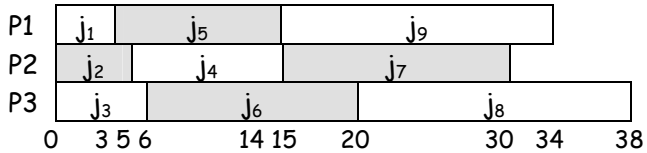
Vamos agora estender nosso problema para um sistema que disponha de mais que um processador, digamos três processadores e uma nova lista de programas:

tarefa	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
tempo	3	5	6	10	11	14	15	18	20

Nosso planejamento inicial para esses três processadores poderia ser também a ordem de chegada e assim teríamos as tarefas j_1, j_4 e j_7 executadas no processador P1. Em P2 teríamos j_2, j_5 e j_8 e o restante em P3. Isso totaliza um tempo total de 165 e um tempo médio de (3+5+6+13+16+20+28+34+40)/9 = 165/9 = 18,33, observe a ilustração:

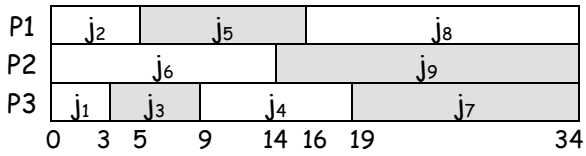


Com essa estratégia distribuímos as tarefas entre os três processadores ciclicamente, ocupando sempre aquele que está disponível. Não é difícil mostrar que nenhum outro esquema de divisão faria melhor, apesar de podermos considerar que o número P de processadores disponíveis divide o número *n* de tarefas apresentadas e assim há muitas seqüências ótimas de tarefas. Isso é obtido localizando cada uma das tarefas de j_{iP+1} a $j_{(i+1)P}$, para cada $0 \leq i < n/P$, em cada processador. Em nosso caso, a próxima ilustração mostra uma segunda solução ótima, que tem o mesmo tempo médio.



Mesmo quando P não divide n exatamente, haverá ainda muitas soluções ótimas mesmo se todos os tempos de execução forem diferentes. Uma vez mais vamos tentar minimizar o tempo final, ou seja, considerar o tempo para encerrar a execução de toda a lista de tarefas. Em nossos exemplos acima, o tempo medido foi de 40 e 38, respectivamente.

Podemos buscar um tempo menor e encontrar um mínimo em 34, como pode ser observado na re – distribuição a seguir:



Observe que o tempo final mínimo obtido de 34 no exemplo, não pode ser melhorado porque os processadores foram utilizados completamente durante todo tempo. Apesar desta ultima programação ter o tempo médio de 18,67, maior que os dois anteriores, tem o mérito de finalizar toda a seqüência de trabalhos em menos tempo. Se tivermos um conjunto de programas para executar de um mesmo usuário, esta parece ser a melhor escolha de programação. Apesar de similares este último problema é NP – completo, e pode ser visto como o problema da mochila, que será apresentado a seguir. Portanto, minimizar o tempo final onde se completam as tarefas é aparentemente muito mais difícil que minimizar o tempo médio de execução.

Código de Huffman

Consideremos uma segunda aplicação para o algoritmo guloso: a compressão de arquivos. O conjunto de caracteres ASCII mais empregados em textos soma aproximadamente 100 símbolos e para distingui-los precisamos ao menos de $\lceil \log_2 100 \rceil = 7 \text{ bits}$. Com 7 bits podemos representar 128 símbolos e no código ASCII temos outros 128 símbolos especiais, ao adicionar um oitavo bit de paridade na codificação. Um ponto a destacar: se o tamanho de um conjunto de caracteres é C, então $\lceil lgC \rceil$ bits serão necessários para sua codificação.

Suponha que temos um arquivo que contem somente os caracteres:

a, e, i, s, t, +, espaço em branco (sp) e nova linha (nl).

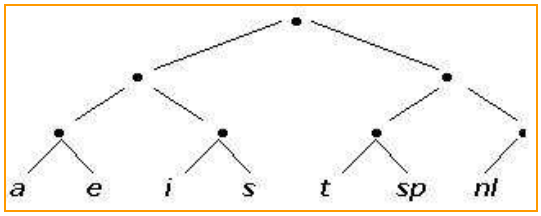
Suponha ainda que tenhamos 10 a's, 15 e's, 12 i's, 3 s's, 4 t's, 13 sp's e 1 nl, para representar cada um desses sete símbolos precisamos 3 bits, e no arquivo são 58 caracteres que totalizam 174 bits, como ilustra a próxima tabela:

caracter	código	freqüência	total
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
sp	101	13	39
nl	110	1	3
	Total		174

Normalmente encontramos arquivos muito maiores, podendo representar a saída de um programa, e em todos temos uma grande disparidade entre os símbolos mais e os menos utilizados. Por exemplo, arquivos de dados numéricos tem grandes quantidades de dígitos, brancos (sp) e nova_linha (nl), más poucos q's e x's. Podemos estar interessados em reduzir o tamanho do arquivo, para facilitar o transito da informação na rede por exemplo. Ou ainda economizar espaço de armazenamento possivelmente providenciando uma melhor codificação que possibilite a redução do número total de bits requeridos inicialmente.

A resposta a esta intenção é positiva, e uma estratégia simples pode alcançar 25% de economia em arquivos típicos e até mais que 60 % em grandes arquivos de dados. A estratégia geral é permitir o comprimento do código variar de um caracter para outro e assegurar que os caracteres que mais ocorrem tenham um código menor. Note que se os caracteres ocorrem com a mesma freqüência não conseguiremos economizar nada. A árvore que representa o pequeno alfabeto de nosso exemplo é mostrada a seguir, e os dados estão somente nas folhas. Trata-se de uma trie que já apresentamos na Pesquisa Digital.

Encontramos a representação partindo da raiz, fazendo as escolhas de caminho, 0 à esquerda e 1 à direita, até completar o código: por exemplo 011 corresponderia aos movimentos esquerda, direita e direita chegando à folha correspondente ao símbolo s. Se um caracter c_i está a uma profundidade d_i e ocorre f_i vezes, então o custo do código é igual a $\Sigma_i\ d_i f_i$.



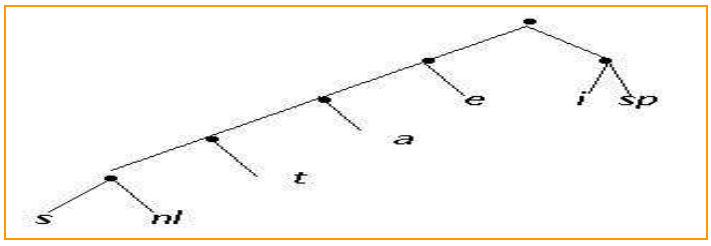
Uma melhoria nesta codificação pode ser obtida observando que o símbolo *nl* é filho único e pode ser localizado um nível acima (em lugar de seu pai), reduzimos assim em um bit (passando para 173 bits totais), ainda longe de um resultado ótimo.

Observe ainda que esta árvore está totalmente preenchida: todo nó é folha ou tem dois filhos. A otimização nos códigos pode considerar esta propriedade e eliminar de saída nós como *nl* acima. Se os caracteres são colocados somente nas folhas, qualquer seqüência de bits pode sempre ser codificada de forma única. Por exemplo, suponha a cadeia codificada:

0100111100010110001000111;

0 não é um código de caracter, tampouco 01, más 010 já representa o *i* e assim temos o primeiro caracter da seqüência. Nos próximos 3 bits, 011, temos um t, e um nova_linha nos dois próximos 11. No restante teremos *a, sp, t, i, e* e *nl*. Note que não houve qualquer dificuldade em reconhecer símbolos com comprimentos de códigos diferentes, pois nenhum código era prefixo para outro. Reciprocamente se um caracter está contido em um nó não-folha, isso não é garantia que a decodificação não terá ambigüidade. Juntando esses fatores, vemos que nosso problema básico é encontrar uma árvore binária plena com custo total mínimo, na qual todos os caracteres fiquem localizados nas folhas.

Nossa árvore a seguir reúne a qualidades de uma árvore ótima de codificação:



Note que nosso alfabeto inicial tem uma nova codificação, e uma diminuição no total de bits:

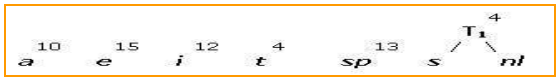
caracter	código	freqüência	total
a	001	10	30
e	01	15	30
i	10	12	24
s	00000	3	15
t	0001	4	16
sp	11	13	26
nl	00001	1	5
	Total		146

Há muitas codificações ótimas, basta podermos trocar os nós filho e codificarmos uma nova árvore. A principal questão em aberto está em como a árvore de codificação é construída. A resposta a essa questão foi dada por Huffman em 1952, como descrito a seguir. Inicia-se assumindo que o número de caracteres é N. O algoritmo pode ser descrito assim:

- Mantemos uma floresta de árvores, o peso de uma árvore é igual à soma das freqüências de suas folhas.
- *N – 1* vezes selecionamos duas árvores T_i e T_j de menor peso e formamos uma nova árvore com as sub-árvores T_i e T_j. No início do algoritmo, há *C* árvores somente com o nó raiz, uma para cada caracter.
- Ao final do algoritmo haverá uma árvore, e essa é uma árvore ótima, construída com o algoritmo de Huffman.

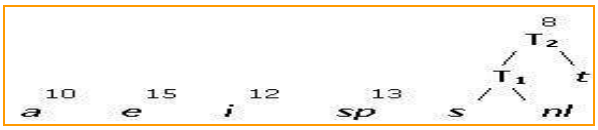
Vejamos as etapas empregando nosso arquivo exemplo de 7 símbolos:

Na primeira etapa formamos a árvore T_1 com *s* e *nl*:

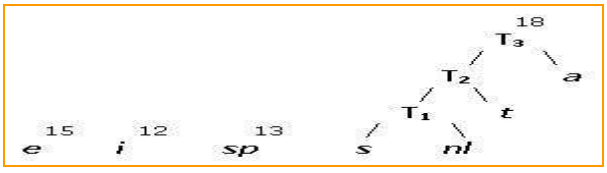


Fizemos *s* o filho esquerdo arbitrariamente, poderia ser o oposto pois qualquer procedimento pode ser usado. O peso total da nova árvore é a soma dos pesos (ou freqüências) de *s* e *nl*.

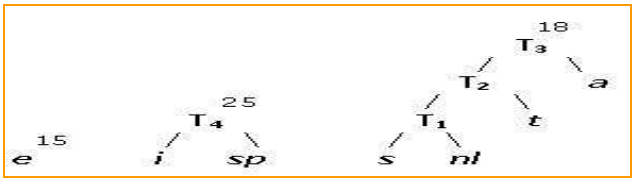
Temos agora seis árvores, e novamente selecionamos as que tem menor peso, juntando T_1 e *t* em uma nova árvore T_2 com peso 8:



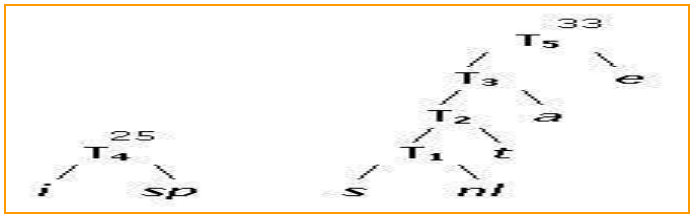
Na terceira etapa juntamos T_2 e *a* em T_3 com peso total 18.



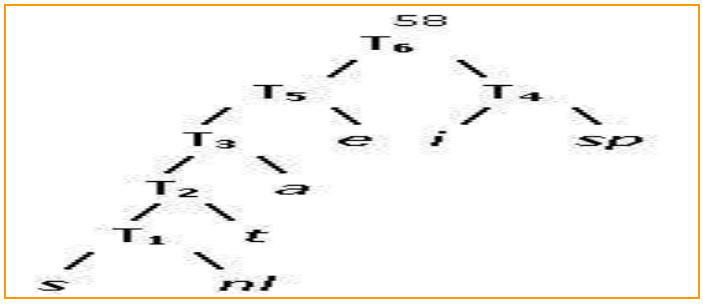
Juntamos em seguida, *i* e *sp* em T_4 com peso 25:



Depois, *e* e T_5 com peso 33:



E finalmente, T_6 com a junção de T_4 e T_5, com peso igual ao número de caracteres 58:



Codificação / Decodificação

Uma vez construída a árvore, seguindo o procedimento de Huffman, devemos decidir qual será o código de cada caractere. Isso pode ser feito se todo nó da árvore armazena seus pais e indica se é um filho esquerdo ou direito.

Se existem *N* caracteres, então existirão *2N – 1* nós. Podemos usar uma estrutura que contenha o peso, o pai, e o status de filho. Apresentamos a seguir uma tabela para a árvore que construímos:

	caracter	peso	pai	tipo_filho
0	*a*	10	9	1
1	*e*	15	11	1
2	*i*	12	10	0
3	*s*	3	7	0
4	*t*	4	8	1
5	*sp*	13	10	1
6	*nl*	1	7	1
7	T_1	4	8	0
8	T_2	8	9	0
9	T_3	18	11	0
10	T_4	25	12	1
11	T_5	33	12	0
12	T_6	58	0	

Os caracteres não são realmente armazenados, mas é usado um índice na tabela (presumimos então que um "*a*" poderia te índice 97, representando o valor em ASCII, "*b*" teria valor 98, e assim por diante. O pai de *nl*, por exemplo, é T_1 e *nl* é um filho direito. Podemos imprimir o código dos *nl*s, escrevendo recursivamente os código dos T_1's, e então o tipo do filho. Note que T_6 tem 0 como pai. Isso indica que devemos parar a recursão. Como o conjunto ASCII tem códigos de 0 a 127, o arranjo será entre 0 e 254, representando o fato que até *n – 1* símbolos internos podem ser necessários.

Uma vez gerado o arquivo comprimido, possivelmente precisaremos descomprimi-lo. Para isso, devemos incluir uma informação extra no arquivo comprimido. Obviamente, queremos incluir uma informação que ocupe o menor espaço possível. Vamos considerar como exemplo

que estamos usando o conjunto de códigos ASCII. Uma possibilidade é armazenar a contagem dos caracteres. Isso requer 128 inteiros; presumimos 24 bits (3 bytes) inteiros serão suficientes (2 bytes podem não atender pois alguns caracteres podem ter ocorrências superiores a 65.535). O armazenamento total necessário é de 384 bytes. Uma alternativa é armazenar as informações: pai e o tipo_filho. Como todo nó é um novo nó criado o valor do pai é um número entre 128 e 254. Conseqüentemente precisamos 7 bits para armazenar o pai. De fato podemos utilizar o caracter de 8 bits: os 7 bits iniciais mais 128 representarão o pai (a raiz pode usar 255 como seu pai). Isso deixa o 8º bit que é mais significativo livre para representar o tipo_filho. Conseqüentemente, o armazenamento total para codificar a informação precisa de 255 bytes.

Antes de podermos começar a compressão, precisamos um contador de freqüências para cada caractere. Se o arquivo é pequeno o suficiente para ser carregado na memória principal, podemos ler o arquivo em um vetor de caracteres e então computamos as contagens, aplicando o algoritmo. Mas, e se for muito grande? Certamente não queremos ler o arquivo duas vezes, as operações de I/O com o disco são extremamente lentas. Uma possibilidade é quebrar o arquivo em grandes blocos que possam ser armazenados na memória principal e então ser comprimidos separadamente.

Computadores têm memória suficiente para armazenar arquivos da ordem de megabytes, de tal forma que um custo adicional de 255 bytes para cada tabela de codificação não parece tão grande. Uma alternativa é assumir que o arquivo de entrada é "típico", por exemplo, a distribuição de caracteres em C/C++ é bem conhecida. Uma vantagem desse método é que não precisamos armazenar a codificação da tabela: o que é particularmente interessante para arquivos enormes. Há muitas outras possibilidades que podem ser exploradas, assim como também; em há métodos de compressão mais complexos que podem funcionar melhor que a codificação de Huffman.

Implementação

Vejamos uma implementação do algoritmo demonstrado, que constrói uma árvore ótima. Considerando que o algoritmo recebe o *um conjunto C de caracteres e as respectivas freqüências f*, em vetores de *n* posições. Inicialmente é criada uma "fila" com prioridades com os elementos do conjunto C, onde cada nó z corresponde a um dos caracteres do meu alfabeto e tem esq[z] = dir[z] = NIL.

A fila com prioridades é implementada num heap e é armazenada no vetor H[1..m] cujos elementos são endereços de nós. Mais precisamente, para cada i, H[i] será o endereço da raiz de uma das árvores da floresta, assim: fr[H[i/2]] ≤ fr[H[i]], para cada i maior que 1.

Para simplificar, supomos que cada caractere do conjunto C, nosso alfabeto, é 1,2,...,n. ou seja, é a posição de C. Desta forma temos como *dados de entrada* o número de símbolos *n* do alfabeto e o vetor f[1..n] com as respectivas freqüências.

Supomos que a freqüência de um caractere i é f[i] e que cada nó *z* da árvore tem 4 campos: ch, fr, esq e dir. Se z é o endereço de um nó então:

- esq[z] = endereço do filho esquerdo do nó z (vale NIL se z não tem filho esquerdo)
- dir[z] = endereço do filho direito do nó z (vale NIL se z não tem filho direito)
- ch[z] = caracter associado ao nó z (só é relevante se esq[z]=dir[z]=NIL)
- fr[z]= freqüência de z (se esq[z]=dir[z]=NIL então fr[z]= fr[ch[z]]; senão fr[z]= fr[esq[z]] + fr[dir[z]])

O algoritmo Huffman constrói uma árvore ótima relativa ao vetor de freqüências f[1..n]. O algoritmo devolve o endereço da raiz da árvore que construiu.

```
Huffman(n, f)
 1  para i ← 1 até n faça          (de início alocamos n nós)
 2       z ← Aloca ()
 3       esq(z) ← NIL
 4       dir(z) ← NIL
 5       fr(z) ← f(i)
 6       H(i) ← z
 7  N ← n
 8  para i ← N/2 até 1 faça
 9       Posiciona(H, N, i)          (constrói o heap H)
10  para i ← 1 até n-1 faça
11       z ← Aloca ()                (cada iteração constrói um nó interno)
12       x ← ExtraiMin(H, N)
13       N ← N - 1
14       y ← ExtraiMin(H, N)
15       N ← N - 1
16       esq(z) ← x
17       dir(z) ← y
18       fr(z) ← fr(x) + fr(y)
19       Inclui(z, H, N)
20       N ← N + 1
21  devolve H(1)
```

Obs: o procedimento Aloca() gera um novo nó e devolve o endereço desse nó. Descrevemos abaixo os procedimentos auxiliares para retirar um nó da fila com prioridades que tem fr mínimo, e incluir um novo nó interno e posicionar o nó no heap:

```
Extraimin(H, N)
 1  m ← H(1)              (obs: fr[m] é mínimo)
 2  H(1) ← H(N)
 3  Posiciona(H, N, 1)
 4  devolve m
```

```
Inclui(z, H, N)
 1  i ← N+1
 2  enquanto i > 1 e fr(H(i/2)) > fr(z) faça
 3       H(i) ← H(i/2)
 4       i ← i/2
 5  H(i) ← z
```

```
Posiciona(H, N, i)
 1  se 2i ≤ N   e  fr( H(2i) ) < fr( H(i) )
 2          então m ← 2i
 3          senão m ← i
 4  se 2i+1 ≤ N e fr( H(2i+1) ) < fr( H(m) )
 5           então m ← 2i+1
 6  se m ≠ i
 7           então troque(H[m], H[i])
 8                    Posiciona(H, N, m)
```

Analisando a Codificação

Vamos apresentar argumentos que provam que a codificação de Huffman providencia um código ótimo:

- Primeiro, não é difícil mostrar por contradição que a árvore deve estar completamente preenchida, pois vimos que uma árvore de códigos que não esta completa deve ser melhorada.
- Em seguida, devemos mostrar que os dois caracteres menos freqüentes, no exemplo *s* e *nl*, devem ser os nós mais profundos (apesar de outros nós também poderem ser ...). Mais uma vez, é fácil mostrar por contradição, pois se *s* ou *nl*, não são os nós mais profundos, então deve haver algum outro nó, digamos *z*, que o será – lembre que a árvore está completa. Se *s* é menos freqüente que *z*, então podemos melhorar o custo trocando-os na árvore.
- Podemos então argumentar que os caracteres em quaisquer dois nós de mesma profundidade podem ser trocados sem afetar a otimização. Isso mostra que sempre podemos encontrar uma árvore ótima que tem os dois símbolos menos freqüentes irmãos; portanto o primeiro passo que executamos ao montar a árvore não foi errado. A prova pode ser completada usando indução. Como as árvores são intercaladas, consideramos o novo conjunto de símbolos como os caracteres nas raízes. Portanto, em nosso exemplo após 4 intercalações, podemos ver um conjunto composto do *e* e dos meta-caracteres T_3 e T_4.

A razão desse algoritmo ser uma aplicação do método guloso é que em cada etapa fizemos uma fusão sem manter considerações globais, meramente selecionamos as duas menores árvores. Se mantemos as árvores em uma fila de prioridades, ordenadas pelo peso, então o tempo para rodar será de ordem $O(n \lg n)$.

Uma implementação simples da lista de prioridades, usando listas ligadas nos daria um algoritmo de ordem $O(n^2)$. A escolha das filas de prioridade, depende de quão grande é n. Em casos típicos, para caracteres em ASCII, n é pequeno suficiente para que o algoritmo em tempo quadrático seja aceitável.

Outras aplicações que empregam a estratégia gulosa podem ser listadas como descobrir uma seqüência ótima de intercalações (quando temos mais que 2 arquivos, ampliando o Mergesort que estudamos), o problema da mochila, encontrar uma árvore espalhada mínima, etc.

O problema da mochila

Vamos apresentar a solução do *problema da mochila*, ou *Knapsack Problem*, empregando os dois últimos métodos estudados: a *versão contínua* do problema da mochila resolvida por um algoritmo guloso e a *versão booleana* que é resolvida via programação dinâmica.

Problema:

Dados vetores x(1..n) *e* w(1..n), denotamos $x \cdot w$ o produto escalar de x por w:

$$x \cdot w = x(1)w(1) + \ldots + x(n)w(n).$$

Suponha dado um número *inteiro não-negativo* M e vetores *inteiros não-negativos* $w(1..n)$ e $v(1..n)$. Uma mochila é qualquer vetor $x(1..n)$ tal que

$$x \cdot w <= M \quad e \quad 0 <= x(i) <= 1 \text{ para todo } i.$$

O valor de uma mochila x é o número $x \cdot v$. O *problema contínuo* (ou o *fracionário*) da mochila consiste no seguinte:

dados w, v, n *e M, encontrar uma mochila de valor máximo.*

A versão booleana, ou *0-1*, do problema tem uma restrição adicional:

$$\text{para todo } i,\ x(i) = 0 \quad \text{ou} \quad x(i) = 1\ .$$

Motivação: Imagine que sua mochila é um disquete. Suponha que a capacidade do disquete é M. Suponha que tenho n arquivos e cada arquivo i tem tamanho $w(i)$ e valor $v(i)$. Minha tarefa é produzir um disquete que tenha o maior valor possível. A variável x seleciona os arquivos: o arquivo i é gravado no disquete se e só se $x(i)=1$. A versão contínua do problema é um pouco artificial pois permite gravar no disquete apenas uma *parte* de um arquivo.

*Exemplo: s*uponha $M = 50$ e $n = 4$. A tabela a seguir dá os valores de $w(1..4)$ e $v(1..4)$. Mais abaixo, temos uma solução $x(1..4)$ do *problema contínuo* e uma solução $x'(1..4)$ do *problema booleano*

O valor da mochila contínua é $x \cdot v = 1040$. O valor da mochila booleana é $x' \cdot v = 1000$.

w	40	30	20	10
v	840	600	400	100
x	1	1/3	0	0
x'	0	1	1	0

Algoritmo guloso para a versão contínua

O seguinte algoritmo guloso resolve a versão contínua do problema da mochila. O algoritmo exige que os dados estejam em ordem decrescente de "valor específico" v/w:

$$V(1)/w(1) >= v(2)/w(2) >= \ldots >= v(n)/w(n)$$

(para evitar divisão por 0, podemos supor que todos os $w(i)$ são positivos).

É nesta ordem "mágica" que está o segredo do funcionamento do algoritmo.

```
Mochila_Gulosa (w,v,M,n){
1  para j ← 1 até n faça
2      se w(j]) ≤ M então x(j) ← 1
3                          M ← M – w(j)
4                  senão x(j) ← M/w(j)
5                          M ← 0
6  devolve x
```

O algoritmo é *guloso* porque abocanha o primeiro objeto viável (na ordem dada), sem se preocupar com o que vai acontecer depois e sem jamais se arrepender do valor atribuído a um componente de x. Por que o algoritmo dá a resposta correta? . . . É evidente que o algoritmo dá o valor de uma mochila; o difícil é provar que o valor é *máximo*. O fato é que,

no início de cada iteração, existe uma mochila máxima que contém os itens selecionados pelo algoritmo (e possivelmente mais alguns).

O consumo de tempo do algoritmo é *O(n)*. Isso não inclui o tempo *O(nlgn)* necessário para colocar os objetos em ordem antes de aplicar o algoritmo. O algoritmo guloso não resolve a mochila booleana ...

Mochila booleana: algoritmo recursivo

O problema da mochila booleana pode ser resolvido por um algoritmo recursivo empregando a Divisão e Conquista. O algoritmo é interessante, mas muito ineficiente.

A idéia, já conhecemos: se o problema é pequeno, resolva-o diretamente; senão, *reduza o problema a um problema menor do mesmo tipo*.

Qual o problema menor nesse caso?

Que tal tentar o problema da mochila com dados *w(1..n-1)*, *v(1..n-1)* e M ?

Isso faz sentido, sim, *se desistirmos de colocar o objeto n* na mochila. Caso contrário, se colocarmos o objeto *n* na mochila, a capacidade da mochila baixa para *M – w(n)*. É claro que esta segunda alternativa só faz sentido se *w(n)* <= *M*.

Vamos colocar essa idéia em prática para uma versão simplificada do problema: nosso algoritmo devolverá apenas o *valor x·v* de uma mochila de valor máximo.

```
Mochila_DC(w, v, M, n)
1  se n = 0
2        então devolve 0
3        senão A ← Mochila_DC(w, v, M, n-1)
4              se w(n) > M
5                  então devolve A
6                  senão B ← v(n) + Mochila_DC(w, v, M – w(n), n-1)
7  devolve Máximo(A,B)
```

O algoritmo é muito ineficiente porque refaz, várias vezes, a solução de vários dos subproblemas.

Qual o remédio? ...

Guardar as soluções dos subproblemas em uma tabela. É o que faremos em seguida.

Mochila booleana: algoritmo de programação dinâmica

O problema da mochila booleana pode ser resolvido por um algoritmo de *programação dinâmica* que consome *O(nM)* unidades de tempo. Isso é bem melhor que o algoritmo recursivo, mas não é nenhuma maravilha, porque o consumo de tempo depende de M.

Aqui a *programação dinâmica* transforma recursão em iteração com o apoio de uma tabela. A idéia é guardar em uma tabela, digamos *t*, as soluções dos subproblemas do problema. A tabela é definida assim:

- *t(i,Y)* é o valor máximo da expressão *x(1..i)·v(1..i)* sujeita à restrição *x(1..i)·w(1..i)* <= *Y*.
- os possíveis valores de *Y* são *0,1,* . . , M (é claro que isso só funciona porque estamos supondo M inteiro) e os possíveis valores de *i* são *0,1,* . . , *n*.

- é importante que *0* seja um valor possível de *Y* e de *i*. Por exemplo, se *w(1)=0* então *t(1,0)=v(1)*. É fácil ver que *t(0,Y)=0* para todo *Y*.
- se *i > 0* então *t(i,Y) = A* se *w(i) > Y* e *t(i,Y) = máximo(A,B)* se *w(i) <= Y*, onde *A = t(i-1,Y)* e *B = t(i-1,Y-w(i))+v(i)*. Ou seja, se a mochila máxima para *1, . . ,i* não usa *i* então ela é também uma mochila máxima para *1, . . ,i-1*.
- se a mochila máxima para *1, . . ,i* usa *i* então ela é o resultado de juntar *i* com uma mochila máxima para *1, . . ,i-1*.

A partir dessa recorrência podemos escrever um algoritmo de programação dinâmica para determinar *t* e o vetor *x*:

```
Mochila_Dinamica(w,v,M, n){
 1  para Y ← 0 até M faça
 2        t(0,Y) ← 0
 3        para i ← 1 até n faça
 4              A ← t(i-1,Y)
 5              se w(i) > Y
 6                    então B ← 0
 7                    senão B ← t(i-1,Y-w(i)) + v(i)
 8              t(i,Y) ← Máximo(A,B)
 9  Y ← M
10  para i ← n até 1 faça
11        se t(i,Y) = t(i-1,Y)
12              então x(i) ← 0
13              senão x(i) ← 1
14                    Y ←  Y-w(i)
15  devolve t e x
```

Note que a coluna *t(... ,0)* não é inicializada com zeros. Ela pode conter elementos não nulos se *w(i)=0* para algum *i*. É óbvio que o consumo de tempo do algoritmo é *O(nM)*.

Essa complexidade *não é satisfatória*: imagine que uma mudança trivial em nossa escala de medida de pesos multiplica por *1000* os números *w(1), . . , w(n)* e M; nosso problema continua essencialmente o mesmo, mas o algoritmo pode consumir *1000* vezes mais tempo! Em termos mais técnicos, diz-se que o algoritmo é "exponencial em M", pois o "tamanho", digamos *t*, do número M é *lg(M)*, e portanto o consumo de tempo é $O(n\,2^t)$.

Exemplo: Suponha que *M = 5* e *n* = 4. Os vetores *w* e *v* são dados pela tabela esquerda e à direita temos a tabela *t*.

	1	2	3	4
w	**4**	**2**	**1**	**3**
v	**500**	**400**	**300**	**450**

	0	1	2	3	4	5
0	**0**	**0**	**0**	**0**	**0**	**0**
1	**0**	**0**	**0**	**0**	**500**	**500**
2	**0**	**0**	**400**	**400**	**500**	**500**
3	**0**	**300**	**400**	**400**	**700**	**800**
4	**0**	**300**	**400**	**450**	**750**	**850**

Árvores geradoras de peso mínimo (Minimum - Spanning Trees)

problema: *Dado um grafo simétrico, um vértice r, e uma função w que atribui um peso a cada arco, encontrar uma árvore geradora com raiz r que tenha peso mínimo.* (obs: o peso de uma árvore é a soma dos pesos de suas arestas.)

Como o grafo é simétrico, qualquer árvore geradora com raiz *r* "cobre" todos os vértices do componente que contém *r*, ou seja, para cada vértice *v* do componente existe um (e um só) caminho de *r* a *v* na árvore. Se existe uma árvore geradora com raiz *r* e peso *P* então, para qualquer vértice *s* no componente que contém *r*, existe uma árvore geradora de peso *P* e raiz *s*.

Algoritmo de Prim

O algoritmo de Prim usa uma "fila" *de vértices*, com prioridade ditada por uma chave a ser discutida em seguida. No início de cada iteração temos dois tipos de vértices, os PRETOs e os BRANCOs, e uma árvore com raiz *r* que "cobre" o conjunto dos vértices PRETOs.

Há também arestas da árvore ligando vértices PRETOs a vértices BRANCOs, mas essas arestas são provisórias. Cada iteração acrescenta à árvore a aresta mais leve dentre as que ligam um vértice PRETO a um vértice BRANCO.

Vamos supor que os parâmetros de entrada são o grafo G (número de vértices, lista de adjacências e pesos) cujos vértices são identificados por 1,2,...,n.

```
MST_Prim(G,r)
 1.  para u ← 1 até n faça
 2.     cor[u] ← BRANCO
 3.     pred[u] ← NIL
 4.     chave[u] ← ∞
 5.  Q ← Crie-Fila-Vazia()
 6.  para u ← 1 até n faça
 7.        Insira-na-Fila(u,Q)
 8.  chave[r] ← 0
 9.  pred[r] ← r
10.  enquanto Q ≠ ∅ faça
11.     u ← Retire-Mínimo(Q)
12.     para cada v em Adj[u]faça
13.           se cor[v] = BRANCO e w(u,v) < chave[v]
14.                 então chave[v]← w(u,v)
15.                       Refaça-Fila(v,Q)
16.                       pred[v]←u
17.        cor[u] ← PRETO
18.  devolve pred
```

O comando Crie-Fila-Vazia()cria uma "fila" vazia com prioridades ditadas por *chave*. Na implementação tudo se passa como se os vértices estivessem nessa fila *sempre em ordem crescente* da *chave*.

O comando Retire-Mínimo(Q) retira de *Q* um vértice com *chave* mínima. A alteração do valor de *chave[v]* na linha 14 pode afetar a estrutura da fila; a função Refaça-Fila(), conserta as coisas.

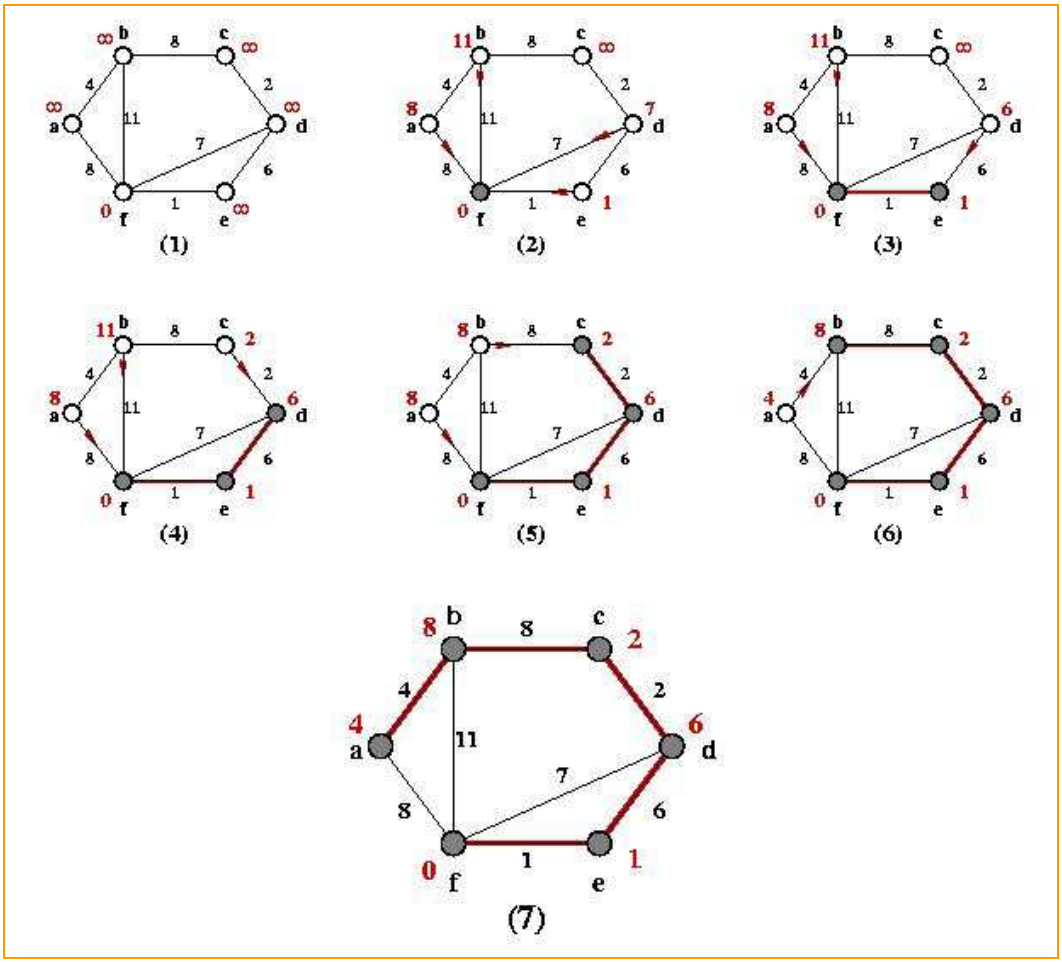
No início de cada iteração:

1. a cor de todo vértice em *Q* é BRANCO e todo vértice de cor BRANCO está em *Q*;
2. para todo vértice *x*, *pred[x] = 0* se e só se *chave[x] = ∞;*
3. para todo vértice *x*, se *pred[x]* não é *0*, então:
    a. *chave[x] = w(pred[x], x);*
    b. a cor de *pred[x]* é PRETO;
    c. o arco *(pred[x], x)* tem peso mínimo dentre todos os arestas que vão de um vértice de cor PRETO a *x*.

A figura a seguir mostra um exemplo de execução do algoritmo MST_Prim. Os números em vermelho (ao lado dos vértices nomeados por a, b, c, ..., f) representam os valores das chaves, enquanto pequenas setas vermelhas indicam os predecessores. Vértices que estão fora da fila ou que possuem predecessores "NIL" não têm setas vermelhas.

Cada passo na figura pode ser considerado como sendo uma "foto" do grafo exatamente no momento do teste do **enquanto** na linha 10. Assim, o passo 1 representa o grafo tal como ele se encontra logo após a execução da linha 9, com todos os vértices com chave igual a infinito, com exceção de "f", que é a raiz, e todos os pais iguais a "NIL". O passo 2 mostra o resultado da primeira iteração do laço das linhas 10-17. Nele, os vizinhos da raiz já têm um valor de chave diferente de infinito e o campo pred apontando para a raiz.

Note que enquanto um vértice não sai da fila de prioridade, seu pai não é fixo: o pai do vértice *d* muda entre os passos 2 e 3. Quando um nó é extraído da fila, muda a cor para PRETO e seu pai passa a ser definitivo.



Exemplo de execução do algoritmo de Prim

Eficiência

Suponha que a fila *Q* é implementada como um *heap*. Qual a eficiência do algoritmo MST_Prim() ?

As linhas 5-7, que criam a fila de vértices, consomem *O(n)*. O loop das linhas 10-17 é executado *n* vezes. Cada execução da linha 11 consome *O(lgn)*. A soma de todas as execuções da linha 11 será *O(nlgn)*. Cada execução da linha 15 consome *O(lgn)*.

E o número total de execuções do bloco de linhas 13-16 ao longo da execução do algoritmo é *O(m)*, onde *m* é o número de arestas do grafo.

Conclusão: o algoritmo é *O((n+m)lgn)*.

Algoritmo de Kruskal

O algoritmo de Kruskal começa com uma floresta de árvores formada por apenas um vértice e procura, a cada passo, uma aresta que, além de conectar duas árvores distintas da floresta, possua peso mínimo. Suponha que, numa determinada iteração, o algoritmo escolheu a aresta *(u,v)* para ser inserida no conjunto *A* e que a aresta *(u,v)* conecte a árvore C_1 à árvore C_2. Note que, dentre todas as arestas que conectam duas componentes distintas nesta iteração, *(u,v)* é uma das que possui o menor peso, pois ela foi escolhida assim. Logo, nesta iteração, para qualquer corte que separe *u* de *v*, *(u,v)* é uma aresta leve (simplesmente pelo fato de não haver outra aresta com peso menor separando duas componentes nesta iteração). Assim, *(u,v)* certamente é uma aresta leve que conecta C_1 a uma outra componente conexa (no caso, C_2) da floresta determinada por *A*, o que significa que *(u,v)* é uma aresta segura para *A*.

A implementação do algoritmo de Kruskal utiliza uma estrutura de dados de conjuntos disjuntos para testar se dois vértices pertencem a uma mesma componente conexa, ou não, no procedimento Conjunto() . A princípio, o conjunto de arestas é ordenado em ordem crescente de peso. As arestas são verificadas uma a uma, em ordem, e o algoritmo executa uma chamada da função Conjunto() para cada vértice adjacente a elas.

Se os vértices não pertencerem ao mesmo conjunto, o que significa que a aresta une vértices de componentes distintas, a aresta é inserida em *A* e os conjuntos de *u* e *v* são unidos com uma chamada do procedimento Une_Arvore. Vejamos uma versão do algoritmo de Kruskal :

```
MST_Kruskal(G,w)
1.  A ← ∅
2.  para u ← 1 até n faça Construa_Conjunto(u)
3.  Ordene(E)                                  {arestas de E em ordem crescente de peso (w)}
4.  para cada aresta (u,v) faça     {tomadas em ordem crescente de peso (w)}
5.        se Conjunto(u) ≠ Conjunto(v)
6.              então A ← A ∪ {(u,v)}
7.                    Une_Árvores(u,v)
8.  devolve A
```

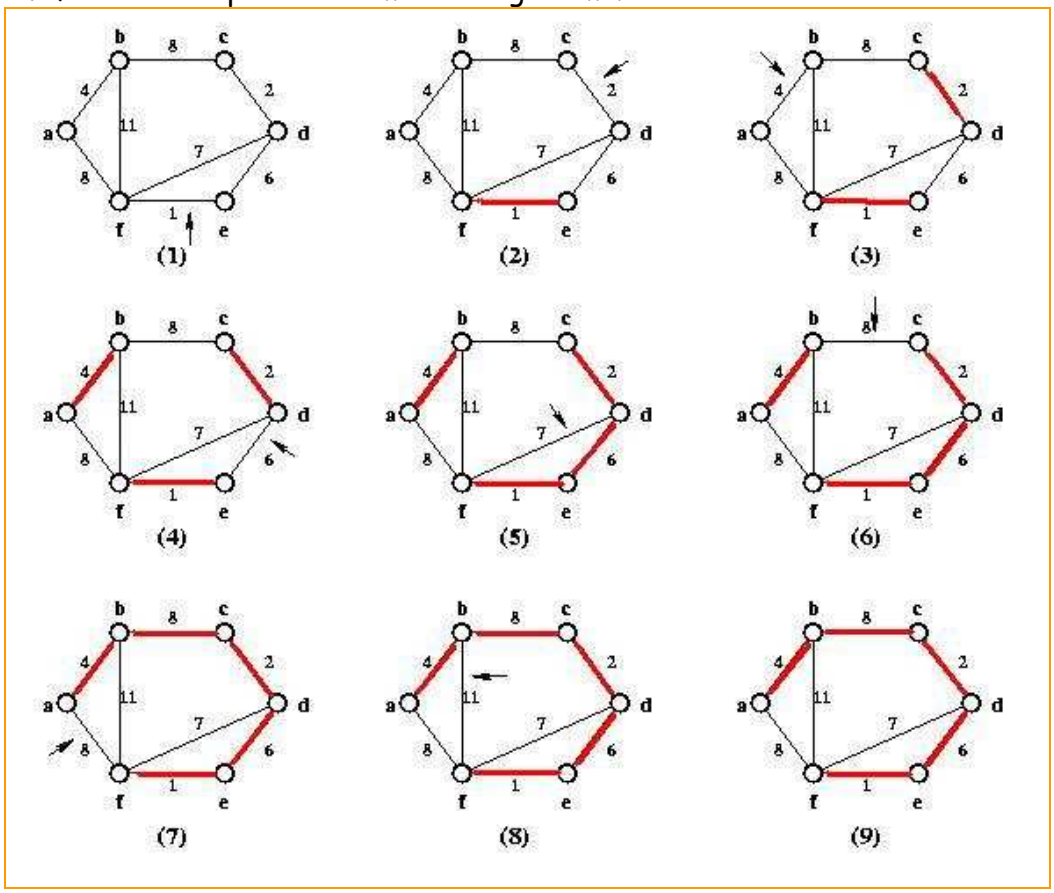
A figura a seguir exemplifica uma execução do algoritmo de Kruskal. Note que as arestas são "visitadas" em ordem crescente de peso (no caso: 1, 2, 4, 6, 7, 8, 8 e 11) e, para que uma

aresta seja inserida no conjunto *A* (aqui representado pelas arestas vermelhas), os seus vértices adjacentes devem pertencer a componentes diferentes.

Note que nos passos 5, 7 e 8, as arestas verificadas acabam não sendo inseridas no conjunto *A*. Na figura, em cada passo, a aresta indicada é analisada (linha 6 do algoritmo) e o fato de uma aresta ser inserida ou não só é percebido no passo seguinte.

Podemos entender os passos 1 a 8 na figura como "fotos" do grafo exatamente no momento em que o teste da linha 6 é executado.

A "foto" 9 foi tirada depois do término do algoritmo.



Exemplo de execução do algoritmo de Kruskal

Eficiência

O tempo de execução do algoritmo de Kruskal depende muito do tipo de implementação da estrutura de dados para conjuntos disjuntos: assumindo a implementação de florestas com as heurísticas de união por posto e compressão de caminhos, teremos uma implementação mais eficiente.

O algoritmo gasta tempo proporcional a O(n) operações de Construa_Conjunto, feitas na linha 2, para criar as n árvores iniciais(supondo que G tem n vértices). Então, ordena as arestas, na linha 3, em gastando um tempo *O(m lgm) – aqui vamos supor |E| = m.*

Cada aresta é verificada uma vez por união das sub-árvores, em tempo proporcional a O(m).

Portanto, tempo de execução do algoritmo de Kruskal será dado pela operação que consome mais tempo - a ordenação - *O(m lgm).*

Observando que $m < n^2$, isto é que $|E| < |V|^2$, podemos redefinir o tempo de execução do algoritmo de Kruskal como *O(m lgn)*

Caminhos de peso mínimo

Suponha que cada aresta *(u,v)* de um grafo tem um *peso w(u,v)*. Para qualquer caminho *C*, vamos denotar por *w(C)* o peso do caminho, ou seja, a soma dos pesos das arestas que formam o caminho. Dados vértices *s* e *t*, um caminho *C* de *s* a *t tem peso mínimo* (ou *é* w-mínimo) se $w(C) \leq w(C')$ para qualquer caminho *C'* de *s* a *t*.

problema: *Dado um grafo, dois vértices s e t, e uma função w que atribui um peso a cada aresta, encontrar um caminho de peso mínimo de s a t.*

Vamos supor que $w(u,v) \geq 0$ para cada aresta *(u,v)*.

Algoritmo de Dijkstra

A experiência mostra que não adianta preocupar-se apenas com os caminhos de *s* a *t*: é preciso considerar também os caminhos que vão de *s a cada um dos demais vértices* do grafo. Ou seja, é preciso encontrar uma *árvore geradora* com raiz s que tenha a seguinte propriedade: para cada vértice *v*, o único caminho de *s* a *v* na árvore é um caminho de peso mínimo (dentre os que começam em *s* e terminam em *v*).

Vemos a seguir um algoritmo que resolve o problema quando não há arestas de peso negativo. O algoritmo recebe um grafo G com vértices 1,2,...,n e arestas definidas por listas de adjacências *Adj*, recebe também um vértice *s* e um peso não-negativo *w(u,v)* para cada aresta *(u,v)*. O algoritmo determina um caminho *w-mínimo* de *s* a cada um dos demais vértices; os caminhos são representados por um vetor de predecessores.

```
Dijsktra(G,w,r)
 1.   para u ← 1 até n faça
 2.          cor(u) ← BRANCO, pred(u) ← 0, d(u) ← ∞
 3.   d(r) ← 0
 4.   Q ← Crie-Fila( )
 5.   enquanto Q ≠ ∅ faça
 6.          u ← Retire-Mínimo(Q)
 7.          cor[u] ← PRETO
 8.          para cada v em Adj[u]faça
 9.                 se cor(v) = BRANCO e d(u) + w(u,v) < d(v)
10.                    então d(v)← d(u)+w(u,v), pred(v)←u
11. devolve pred
```

O comando Crie-Fila() cria uma "fila" com todos os vértices. A prioridade dos vértices na fila é dada pelo vetor *d*. O comando Retire-Mínimo(Q) retira de *Q* um vértice *u* para o qual *d[u]* é mínimo. A alteração do valor de d(v) na linha 10 pode afetar a estrutura da fila; isso deve ser levado em conta quando a fila for implementada.

Para entender o funcionamento do algoritmo, é preciso observar que no início de cada iteração valem as seguintes propriedades:

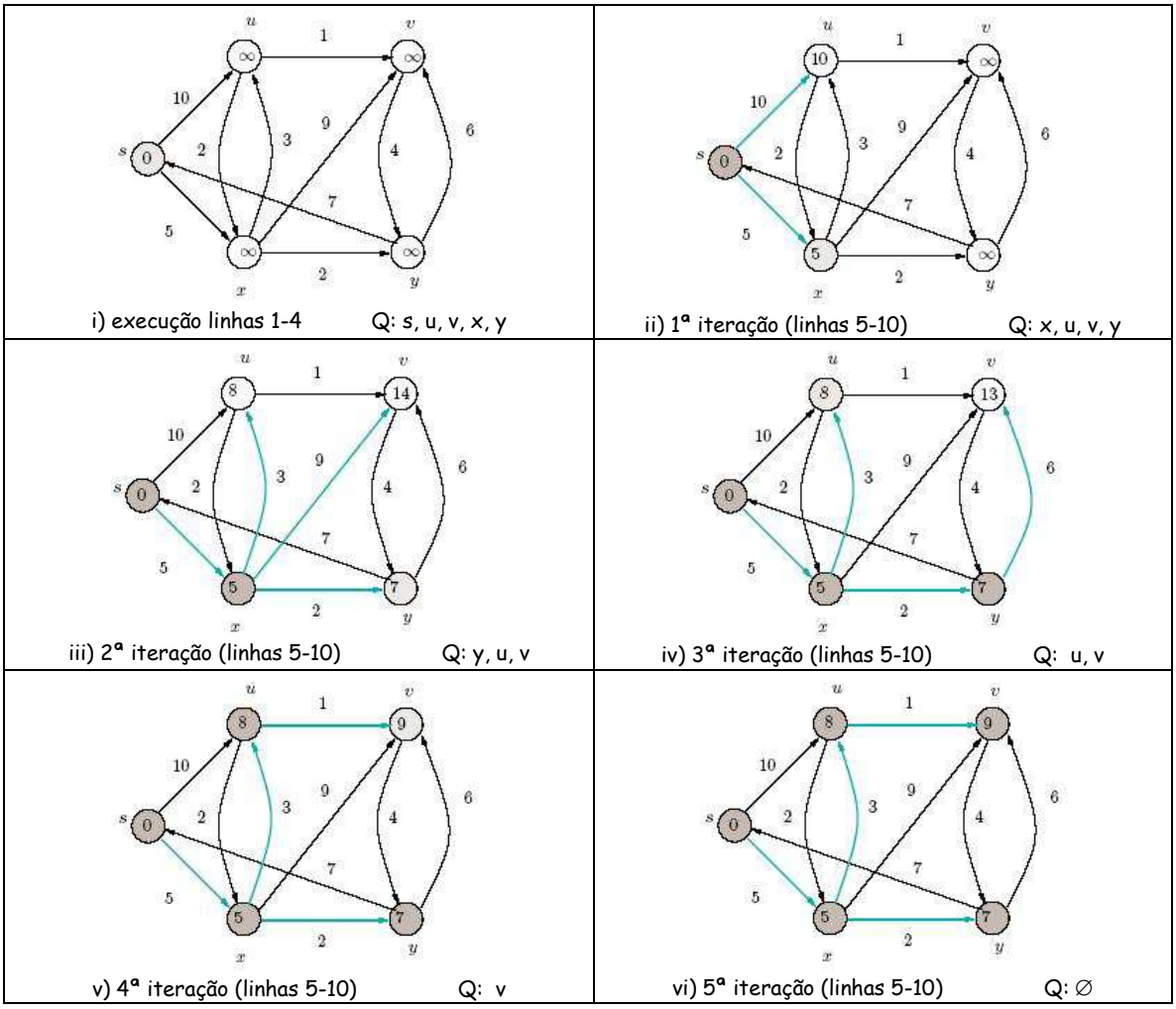
1. todo vértice em *Q* é BRANCO e todo vértice BRANCO está em *Q*;
2. para todo vértice *x* existe um caminho de *s* a *x* cujo peso é d(x) (portanto, d(x) é sempre uma *estimativa* da distância de *s* a *x*);
3. $w(C) \geq d(x)$ para todo caminho *C* que começa em *s* e termina em um vértice PRETO, ou seja, se *x* é PRETO então *d(x)* é a "distância" correta de *s* a *x*;
4. $w(C) \geq d(x)$ para todo caminho *C* que começa em *s*, termina em um vértice BRANCO *u* e não tem outros vértices BRANCO s além de *u*.

Resumindo 2,3,4: *d(x)* é o peso de um caminho de peso mínimo dentre os que não têm vértices BRANCO s exceto talvez o último.

A operação nas linhas 9 e 10 é conhecida como *relaxação* da aresta *(u,v):* se existe um caminho de peso *d(u)* de *s* até *u* então também existe um caminho de peso *d(u)+w(u,v)* de *s* até *v*.

As figuras a seguir demonstram a execução do algoritmo Dijsktra em um grafo onde a origem é o vértice "s" mais à esquerda.

As estimativas de caminhos mais curtos são mostradas dentro dos vértices, e as arestas "verdes" indicam valores de predecessores.



Eficiência do algoritmo

Comece considerando a possibilidade de implementar *Q* em uma fila sem prioridades: faça busca linear para achar o mínimo. Suponha matriz de adjacências no lugar de *Adj*. Temos *n* iterações. Cada iteração gasta *n* para retirar o mínimo. Cada vértice sai de *Q* e fica PRETO uma só vez. Logo, cada linha da matriz é examinada apenas uma vez. Temos aí n^2. Total: n^2+n^2. O consumo de tempo é então $O(n^2)$. Se usarmos listas de adjacências, teremos $O(n^2+m)$.

Exercícios

1. Implemente o algoritmo que *planeja a execução de tarefas* dentro da estratégia gulosa.

2. Encontre um algoritmo que dê uma seqüência ótima para intercalar dez arquivos que tem os seguintes tamanhos: 28, 32, 12, 5, 84, 53, 91, 35, 3 e 11.

3. Num arquivo temos seguintes símbolos e respectivas freqüências: *virgulas*(100), *espaços*(605), *nova_linha*(100), *aspas*(705), *0* (431), *1* (242), *2*(176), *3* (59), *4* (185), *5* (250), *6* (174), *7* (199), *8* (205) e *9* (217). Construa o código de Huffman para este arquivo.

4. [Bin-packing] São dados objetos 1, ..., n e um número ilimitado de "latas". Cada objeto *i* tem "peso" w_i e cada lata não pode passar de 1. *Problema: distribuir os objetos pelo menor número possível de latas*. Programe e teste as seguintes heurísticas: (1) examine os objetos na ordem dada; tente colocar cada objeto em uma lata já parcialmente ocupada que tiver mais "espaço" livre sobrando; se isso for impossível, pegue uma nova lata. (2) rearranje os objetos em ordem decrescente de peso; em seguida, aplique a heurística (1). Essas heurísticas resolvem o problema?

5. Prove que o problema da mochila fracionária tem propriedade gulosa.

6. Estude o problema da mochila booleana no caso particular em que v = w para cada i. Comece por simplificar a formulação do problema.

7. Suponha dado um número não-negativo M e vetores não-negativos w(1..n) e v(1..n). Digamos que um *segmento* é qualquer par (i,k) tal que $1 \le i \le k \le n$ e $w(i)+w(i+1)+ \ldots +w(k) \le M$. O *valor* de um segmento (i,k) é o número $v(i) + v(i+1) + \ldots + v(k)$. Problema: Determinar um segmento de valor máximo.

8. Suponha dado um grafo simétrico com pesos nos arestas. Suponha que um vetor de predecessores *pred* representa uma árvore geradora de peso *P*. Escreva um algoritmo que receba *pred* e um vértice *s* e devolva o vetor de predecessores de uma árvore geradora com raiz *s* e peso *P*.

9. Seja *r* um vértice de um grafo *G não necessariamente simétrico*. Para cada aresta*(u,v)*, seja *w(u,v)* o peso. Escreva um algoritmo que encontre uma árvore de peso mínimo dentre as que têm raiz *r*. (É claro que a árvore deve ser orientada: cada aresta da árvore deve estar apontando "para longe de" *r*.)

10. Suponha dado um grafo com vértices 1,2,...,n. Suponha que todas os arcos são da forma *(j,i)*, sendo $i < j$. Suponha que cada arco *(u,v)* tem um peso *w(u,v)*, que pode ser positivo ou negativo.

    a) Escreva um algoritmo que encontre um caminho de peso mínimo de um vértice *s* a um vértice *t*. Quanto tempo o seu algoritmo consome?

    b) A *altura* de um vértice *u* é o peso de um caminho de peso mínimo de *u* até o vértice 1. (OBS: "de *u* até 1".) Escreva um algoritmo que resolva o seguinte problema: "*Encontrar a altura de cada um dos vértices do grafo*".

11. Mostre que o algoritmo Dijkstra pode produzir resultados errados se o peso $w(u,v) < 0$ para alguma aresta *(u,v)*.

Backtraking

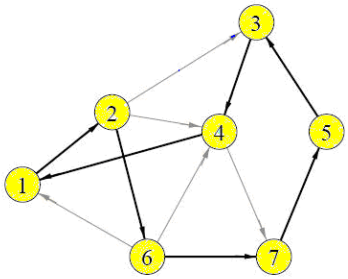
Em muitos casos, um algoritmo backtracking, ou algoritmo de reversão, equivale a uma implementação de pesquisa exaustiva, com desempenho geralmente desfavorável. Entretanto, esse não é sempre o caso, em alguns casos pode ser mais econômico que os esforços sobre pesquisas exaustivas usando força bruta. Com certeza, o desempenho é relativa, um algoritmo $O(n^2)$ para classificação é muito ruim, mas um algoritmo $O(n^5)$ para um problema como o do caixeiro viajante (ou qualquer problema NP - completo) poderia ser um grande resultado.

Um exemplo prático de algoritmo backtracking é o problema de arranjar a mobília em uma casa nova. Há muitas possibilidades para testar, mas tipicamente somente poucas serão realmente consideradas. Começando com tudo desarrumado, cada peça de mobília é colocada em alguma parte da casa. Se toda mobília é colocada e o proprietário está contente, então o algoritmo termina. Se pesquisarmos um ponto onde toda subseqüente colocação de mobília é indesejável, temos que desfazer o último passo e tentar uma alternativa. Certamente, isso poderia forçar a desfazer outro passo e assim por diante.

Se verificarmos que desfizemos todos os primeiros passos, então não há colocação da mobília que seja satisfatória. De outra forma, eventualmente, terminaremos com um arranjo satisfatório. Note que apesar desse algoritmo ser essencialmente de força bruta, ele não tenta todas as possibilidades diretamente. Por exemplo, arranjos que consideram o sofá na cozinha nunca são tentados. Muitos outros arranjos ruins são previamente descartados porque um subconjunto de arranjos indesejáveis é detectado. A eliminação de um grande grupo de possibilidades em um passo é conhecido como poda.

Exemplo com grafos: *encontrar um ciclo Hamiltoniano.*

Um ciclo Hamiltoniano em um grafo direcionado (ou não) é um ciclo que passa exatamente uma vez em cada vértice e retorna ao vértice inicial.



A solução para encontrar "um ciclo Hamiltoniano" em um grafo direcionado G = (V, E), como ilustra a figura acima, começa por armazenar o grafo em uma lista de adjacências (representada na matriz Adj apresentada a seguir) que guarda a lista dos vértices (u,v) ∈ E e, também um arranjo D(1..n) com o grau de saída de cada um dos vértices de G. Armazenamos o ciclo Hamiltoniano em um arranjo A(1..n), onde o ciclo (vide figura) será: A(n) → A(n-1) → ... → A(2) → A(1) → A(n).

Adj	1	2	3
1	2		
2	3	4	6
3	4		
4	1	7	
5	3		
6	1	4	7
7	5		

Arranjo com o grau de saída dos vértices:

	1	2	3	4	5	6	7
D:	1	3	1	2	1	3	1

Ciclo Hamiltoniano no grafo da figura:

	1	2	3	4	5	6	7
A:	6	2	1	4	3	5	7

O algoritmo para a solução mantêm a sinalização de poda em um arranjo U(1..n) que sinaliza se um vértice "m" (onde 1< m< n) é válido ou não: U(m) = 1 é válido e U(m) = 0 se não for é aí ocorre a poda. A lista de adjacências guardada na matriz Adj, o arranjo D(1..n) do grau de saída dos vértices são os dados de entrada do algoritmo Encontra( ) descrito a seguir que assume existirem n > 1 vértices:

```
Encontra( Adj, D)
1.  para i ← 1 até n-1 faça U(i) ← 1
2.  U(n)  ← 0
3.  A(n) ← n
4.  Hamillton(n-1)
```

```
Hamilton(m)
1.  se m = 0
2.      então Processa(A)
3.      senão para j ← 1 até D(A(m+1)) faça
4.                  w ← Adj( A(m+1), j)
5.                  se U(w) = 1
6.                        então U(w) ← 0
7.                              A(m) ← w
8.                              Hamilton(m-1)
9.                              U(w) ← 1
```

```
Processa(A)
1.  ciclo ← 0
2.  para j ← 1 até D(A(1)) faça
3.        se Adj( A(1), j)
4.             então ciclo ← 1
5.  se ciclo = 1
6.      então imprime A(1..n)
```

Observações:

- o procedimento Hamilton( ) completa o ciclo Hamiltoniano em A(m+1) com m vértices não utilizados.
- O procedimento Processa(a) fecha o ciclo quando é chamado (no momento que m = 0) pois verifica se os vértices indicados em A(n) e A(1) contem uma aresta do grafo. Se existe do ciclo, imprime o arranjo A(1..n).
- A poda é realizada na linha 5 do procedimento Hamilton( ), baseado em se um novo vértice foi ou não usado (marcado na linha 6). Na linha 9, o vértice é marcado como não usado quando retornamos de uma visita a ele.

Analisando a complexidade desse algoritmo para "encontrar um ciclo Hamiltoniano", vamos assumir inicialmente que não exista nenhum ciclo. Seja T(n) o tempo para processar o procedimento Hamilton(v, n), supondo que o grafo tenha o valor "d" como maior grau de saída, obtemos:

$$T(n) \leq bd \quad (\text{se } n = 0) \quad \text{e} \quad T(n) \leq dT(n-1) + c \quad (\text{se } n > 0) \text{ onde b, c são constantes.}$$

Dessa forma, substituindo T(n-1), T(n-2), ..., obtemos: $T(n) = O(d^n)$.

Portanto, o tempo total de processamento em "n" vértices do grafo será $O(d^n)$ se não há ciclo Hamiltoniano, porém se encontrarmos "um ciclo Hamiltoniano" gastaremos um tempo:

$$T(n) = O(d^n + d) = O(d^n)$$

Veremos agora um exemplo de algoritmo backtracking associado a geometria computacional, observando que esta técnica também é usada na elaboração de jogos como na seleção de movimentos de peças num tabuleiro de xadrez por exemplo, ou a previsão do próximo lance no "jogo da velha".

Para ilustrar como funcionam esses algoritmos vamos trabalhar com um problema.

O problema da reconstrução de auto-estradas

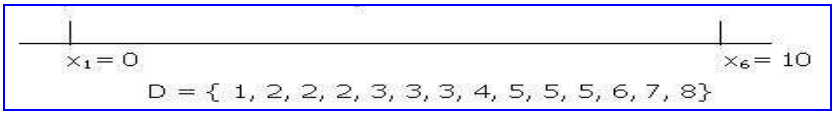
Suponha que são dados n pontos $p_1, p_2, \ldots, p_n$, localizados sobre um eixo x, onde x_i é a coordenada x do ponto p_i. Vamos assumir que $x_1 = 0$ e os pontos são dados da esquerda para a direita. Esses n pontos determinam n(n-1)/2 (não necessariamente únicas) distâncias $d_1, d_2, \ldots, d_n$ entre cada par de pontos da forma $|x_i - x_j|$ $(i \neq j)$. esta claro que se damos o conjunto de pontos, é fácil construir o conjunto de distâncias em tempo $O(n^2)$. Esse conjunto não é ordenado, mas se estamos dispostos a estabelecer um limite de tempo de $O(n^2 lgn)$, as distâncias podem ser ordenadas também. Essa aplicação é conhecida como *o problema da 'reconstrução de "auto-estradas com cobrança de pedágio"* onde reconstrói-se um conjunto de pontos a partir de distâncias conhecidas.

Temos aplicações desse problema na Física ou na Biologia Molecular, onde uma estrutura de dimensões atômicas apresenta seus contornos a partir de experimentos que indicam as distâncias inter-atômicas como por exemplo na imagem de superfícies cristalina com detalhes de formatos e profundidades revelando as disposições e ligações atômicas. O nome deriva da analogia dos pontos de saída (acessos) das auto-estradas para reconstruir a auto-estrada, sendo a construção naturalmente muito mais simples que a reconstrução a partir de um conjunto de distâncias já pré existentes. Observamos ainda que não se tem notícia de uma algoritmo em tempo polinomial para resolver este problema. Vamos apresentar a seguir a solução do problema observando que o algoritmo desenvolvido para ele roda dentro da meta acima, ou seja, em tempo proporcional a $O(n^2 lgn)$.

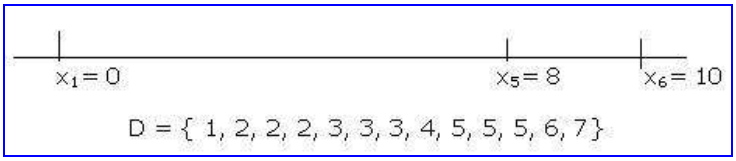
Seja D um conjunto de distâncias, e assumimos que |D| = m = n(n-1)/2. Como um exemplo, suponha que

D = { 1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 5, 6, 7, 8, 10}

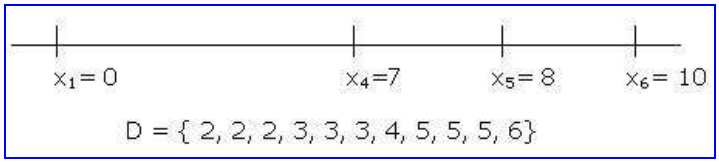
Como |d| = 15, sabemos que n = 6. Começamos o algoritmo ajustando $x_1 = 0$, e $x_6 = 10$, pois é o maior elemento de D. Removemos então 10 de D, e o conjunto de pontos e distâncias localizadas fica:



A maior distância restante é 8, que implica em $x_2 = 2$ ou $x_5 = 8$. Por simetria, podemos concluir que a escolha não é importante desde que ambas levam a soluções (as soluções são como imagens no espelho – equivalentes). Vamos optar por $x_5 = 8$. Então removemos 8 do conjunto D, e obtemos:

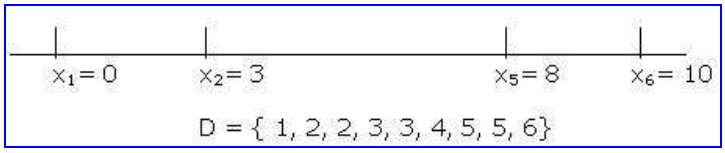


O próximo passo não é óbvio. Desde que 7 é o maior valor em D, $x_4 = 7$ ou $x_2 = 3$. Se $x_4 = 7$, então as distâncias $x_6 - 7 = 3$ e $x_5 - 7 = 1$ devem estar presentes em D. Uma verificada, mostra-nos que estão. Por outro lado, se escolhemos $x_2 = 2$ então $3 - x_1 = 3$ e $x_5 - 3 = 5$ devem estar presentes em D, elas estão! Não há uma referencia de quem escolher, vamos tentar a primeira possibilidade $x_4 = 7$, que nos dá:

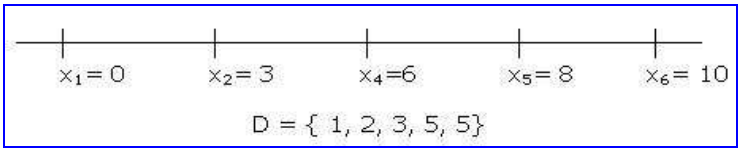


E nesse ponto, temos $x_1 = 0$, $x_4 = 7$, $x_5 = 8$ e $x_6 = 10$. Agora a maior distância é 6: $x_3 = 6$ ou $x_2=4$. Mas se $x_3 = 6$, então $x_4 - x_3 = 1$, que é impossível, pois 1 já foi eliminado. Por outro lado, se $x_2 = 4$ então $x_2 - x_0 = 4$ e $x_5 - x_2 = 4$. Isso também não é possível, pois 4 aparece só uma vez em D. Portanto, essa linha de raciocínio não leva a uma solução, e nos voltamos.

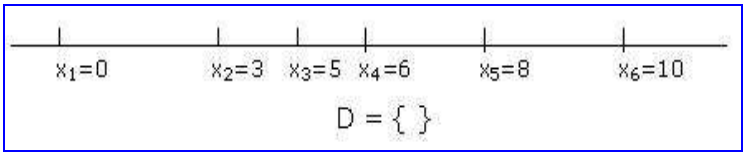
Como $x_4 = 7$ falhou, tentamos a outra com $x_2 = 3$. Se essa também falhar, damos o caso por encerrado sem solução. Assim temos:



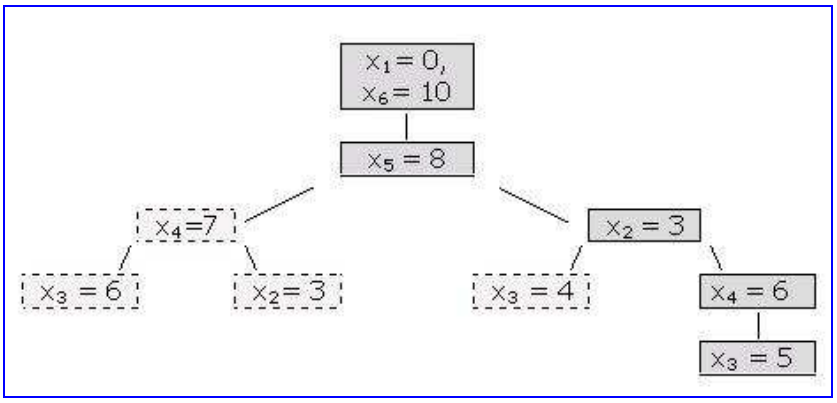
Uma vez mais, temos que escolher entre $x_4 = 6$ e $x_3 = 4$. Escolher $x_3 = 4$ é impossível, porque D somente tem um 4, e essa escolha implicaria em dois. Por outro lado, $x_4 = 6$ é possível, e assim obtemos:



A ultima escolha que restou foi $x_3 = 5$, que funciona pois deixa D vazio, como precisávamos para ter uma solução, e assim:



Observe a árvore de decisões na ilustração a seguir, representando as ações tomadas até chegar a solução do problema:



Anotamos em cada nó destino a escolha do ponto realizada, note que os nós com bordas pontilhadas e coloração mais clara correspondem às escolhas que falharam, pois as escolhas eram incompatíveis com as distâncias dadas, enquanto o nó com bordas duplas e coloração clara significou um caminho incorreto, pois seus descendentes são inconsistentes.

Quando caímos em um desses nós o algoritmo retornou à última escolha acertada e recomeçou o processo de verificação de outras alternativas.

Vamos apresentar um algoritmo que resume o procedimento adotado para resolver este problema, ele recebe um vetor para as saídas dos pontos (X (1..n)), o conjunto de distâncias no vetor D(1.. max) e a quantidade de pontos que devemos procurar n. se uma solução é descoberta então devolverá 1, a resposta será colocada em X(i), e D(1..max) vai sendo esvaziado com *Remove* (|distância, D).

Se a hipótese é falsa (inconsistente) devolverá "0" e X(1..n) fica indefinido, não modificando D(1..max).

O procedimento ajusta X(1), X(n-1) e X(n) na mesma forma que ilustrado acima, altera D(1..max) e chama a função Localiza para colocar outros pontos. Supomos também que uma verificação terá sido feita para que $|D| = n(n-1)/2$.

```
Reconstrói(X(1..n),D(1..max),n) (função para orientar as escolhas)
1  X(1) ← 0;
2  X(n) ← D(max);
3  X(n-1) ← D(max-1));
4  se |X(n)–X(n-1)| ∈ D()
5       então Remove(|X(n) – X(n-1)|,D)
6            devolve Localiza(X(), D(), n, 2, n-2)
7    senão devolve 0
```

A parte mais difícil é o algoritmo backtracking, apresentado logo abaixo. Como a maioria dos algoritmos desenvolvidos com essa técnica, a implementação mais conveniente é recursiva. Passamos os mesmos argumentos mais as direções Esq e Dir, e ainda Xesq e Xdir como coordenadas que estamos tentando colocar.

Se D está vazio (ou Esq > Dir), então a solução terá sido encontrada, e podemos finalizar. Se não é o caso, primeiro tentamos Xdir=Dmax.

Se todas as distâncias apropriadas estão presentes (na quantidade correta), então tentamos colocar este ponto, removendo essas distâncias, e tentando preencher de Esq para a Dir - 1. Se as distâncias não estão presentes, ou a tentativa de preencher Esq para Dir - 1 falha, então tentamos ajustar Xesq = X(n) – Dmax usando uma estratégia similar.

Se isso não funciona, então não há solução. Por outro lado, se encontramos uma solução ela é passada para *Reconstrói* através de *Encontrou* e as variáveis X(1..n).

A analise do algoritmo envolve dois fatores. Suponha que as linhas de comando de 10 a 12 e de 19 a 21, ou seja, quando ocorrem os retornos e restauram-se as distâncias apagadas, nunca sejam executadas. Podemos manter D(1..max) como uma árvore binária de pesquisa balanceada.

Se nunca retornamos (não encontramos soluções incompatíveis) haverá até $O(n^2)$ operações envolvendo D(1..max), tais como apagar as distâncias e as pesquisas de *Encontrou* (incluídas na seleção (if ...) das linhas 4 e 12). Esse fato é claro para remoções, desde que D(1..max) tenha $O(n^2)$ elementos, e nenhum elemento seja re-inserido.

Cada chamada de *Localiza* usa até 2*n *Encontrou"*s, e se *Localiza* nunca volta atrás em sua analise, pode haver até $2*n^2$ *Encontrou"*s.

Portanto se não há retornos, o tempo para rodar será $O(n^2 \lg n)$.

```
➢algoritmo backtracking para colocar os pontos X(Esq..Dir)
➢ X(1 .. Esq-1)  e  X(Dir+1 .. n) já foram colocados
➢                se Localiza devolve 1, então X(Esq..Dir) foi preenchido.

Localiza(X(1..n), D(1..max), n, Esq, Dir)
 1  Encontrou ← 0;
 2  se D(1..max) = 0
 3     então devolve 1                              (D(1..max) foi esvaziado)
 4  Dmax ← EncontraMax(D(1..max))      (verifica se X(Dir) = Dmax é possível
 5  se |X(j) – Dmax| ∈ D( ), 1 ≤ j < Esq, Dir < j ≤ n
 6     então X(Dir) = Dmax                          (tenta X(Dir) = Dmax)
 7           para 1 ≤ j < Esq e Dir < j ≤ n faça
 8              Remove(|X( j ) – Dmax|, D(1..max))
 9           Encontrou ← Localiza(X(1..n), D(1..max), n, Esq, Dir-1)
10           se Encontrou = 0          (volta pois não obteve resposta consistente)
11              então para 1≤j<Esq, Dir<j≤n faça     (retorna apagados de D(1..max))
12                    Insere(|X(j) – Dmax|, D(1..max))
        ( se a 1ªtentativa falhou, verifica-se a viabilidade do ajuste X(Esq) = n–Dmax...)
13  se  Encontrou=0 e (|X(j)– Dmax – X(j)| ∈ D(1..max), 1≤j<Esq, Dir<j≤n)
14     então X(Esq) = X(n) – Dmax                   (com a mesma lógica anterior)
15        para 1 ≤ j < Esq, Dir < j ≤ n faça
16              Remove(|X(n) – Dmax| - X(j))|, D( ))
17           Encontrou = Localiza(X(1..n), D(1..max), n, Esq+1, Dir)
18           se Encontrou=0                                  ( volta...)
19              então para 1 ≤ j < Esq, Dir < j ≤ n faça     (desfaz remoções...)
20                    Insere(|X(n) – Dmax - X(j) |, D(1..max))
21  devolve Encontrou
```

Como sempre ocorrem retornos (backtracking) ou reconsiderações como vimos no exemplo, e se ocorrem repetidamente, a performance do algoritmo é afetada. Nenhum limite polinomial para o backtracking é conhecido, más por outro lado, não há exemplos patológicos que mostrem que os retornos devem ocorrer sempre mais que *O(1)* vezes. Portanto, é inteiramente possível que esse algoritmo seja de ordem $O(n^2 \lg n)$.

Experimentos mostram que se os pontos têm coordenadas (números inteiros) distribuídos uniformemente e aleatoriamente no intervalo (0, D_{max}), onde $D_{max} = \theta(n^2)$, então é quase certo que ao menos um retorno será realizado durante a execução do algoritmo.

Exercícios

1. Suponha que você emitiu cheques em maio nos valores de p(1), ..., p(n) ao longo do mês de setembro último. No fim do mês, o banco informa que um total T foi descontado de sua conta. Quais cheques foram descontados? Por exemplo, se p = {61, 62, 63, 64} e T = 125 então só há duas possibilidades: ou foram descontados os cheques 1 e 4 ou foram descontados os cheques 2 e 3. Esse é o "*problema da soma de subconjuntos*". Desenvolva um algoritmo para resolver este problema empregando a estratégia backtracking.

2. Suponha que sejam dados *n* homens e *n* mulheres e duas matrizes $n \times n$ P e Q tal que P(i,j) é a preferência de um homem *i* por uma mulher *j* e Q(i,j) é a preferência de uma mulher *i* por um homem *j*. Desenvolva um algoritmo que encontre os pares de homens e mulheres tais que a soma do produto das preferências é maximizada.

3. *O problema das oito damas*: oito damas devem ser colocadas em um tabuleiro de xadrez, de tal modo que nenhuma delas ataque, ou seja, atacada por nenhuma outra. Implemente um algoritmo para resolver este problema considerando como dado de entrada uma casa C_{ij} do tabuleiro, que pode ser interpretado como uma matriz 8x8, e a partir daí o algoritmo deverá solucionar o problema.