Programação Dinâmica

- Para cada um dos seguintes problemas:

  a) Defina os subproblemas

  b) Reconheça e resolva os casos base

  c) Apresente a recorrência que resolva os subproblemas

  d) Apresente uma implementação ingênua

  e) Apresente uma solução utilizando PD "top-down" ou "bottom-up"

  f) Analise a complexidade da solução utilizando PD

1. Suponha que você esteja gerenciando uma equipe de consultores de hackers especializados em computadores e cada semana você tem que escolher trabalhos para eles realizarem. Como você pode imaginar, o conjunto de trabalhos possíveis é dividido naqueles que são de baixo estresse (por exemplo, a criação de um site para uma escola elementar) e aqueles que são de alto estresse (por exemplo, proteger os segredos mais valiosos da nação, ou ajudar um grupo desesperado de estudantes da UFBA a terminar um projeto que tem algo a ver com compiladores). A questão básica, a cada semana, é se deve assumir um trabalho de baixo estresse ou um trabalho de alto estresse.
   Se você seleciona um trabalho de baixo estresse para sua equipe na semana i, então você ganha um receita de $l_i > 0$ reais; se você selecionar um trabalho de alto estresse, você obtém uma receita de $h_i > 0$ reais. O problema, no entanto, é que para a equipe assumir um trabalho de alto estresse na semana i, é necessário que eles não façam nenhum tipo de trabalho na semana $i-1$; eles precisam de uma semana inteira preparação para enfrentar o nível de estresse de esmagador. Por outro lado, está tudo bem para eles assumirem um trabalho de baixo estresse na semana i mesmo que eles tenham feito um trabalho de qualquer tipo na semana $i-1$.
   Então, dada uma sequência de n semanas, um plano é especificado por uma escolha de trabalhos de "baixo estresse", "alto estresse" ou "nenhum trabalho" para cada uma das n semanas, com a propriedade que se "alto estresse" é escolhido para a semana $i > 1$, então "nenhum trabalho" deve ser escolhido para a semana $i-1$. (Não há problema em escolher um trabalho de alto estresse na semana 1.) O valor do plano é determinado de maneira natural: para cada i, você adicione l_i ao valor se você escolher um trabalho "baixo-stress" na semana i e adicionar h_i ao o valor do plano se você escolher "alto estresse" na semana i. (Você adiciona 0 se você escolher "nenhum trabalho" na semana i.)
   O problema. Dado conjuntos de valores $l_1, l_2, \ldots, l_n$ e $h_1, h_2, \ldots, h_n$, encontre um plano de valor máximo. (Esse plano será chamado ótimo.)
   Exemplo. Suponha $n = 4$, e os valores de l_i e h_i são dados pelo Tabela 1. Então o plano de valor máximo seria escolher "Nenhum trabalho" na Semana 1, um trabalho de alto estresse na Semana 2 e trabalhos de baixo estresse em Semanas 3 e 4. O valor deste plano seria $0 + 50 + 10 + 10 = 70$.

2. Dada uma haste h de n metros de comprimento, e uma tabela de preços p_i, para $i = 1, 2, \ldots, n$, o problema do corte de hastes consiste em determinar qual a lucro máximo

	Semana 1	Semana 2	Semana 3	Semana 4
l	10	1	10	10
h	5	50	5	1

Tabela 1: Tabela de trabalhos e seus níveis de estresse.

(h_n^{max}) que se pode obter cortando a haste h e vendendo os seus pedaços, considerando que os comprimentos são sempre números inteiros de metros.

m (metros)	i	1	2	3	4	5	6	7	8	9	10
R$ (reais)	p_i	1	5	8	9	10	17	17	20	24	30

Tabela 2: Exemplo de preços adotados para hastes de tamanho i, para $1 \leq i \leq 10$.

Baseado nos valores apresentados na Tabela 1, podemos ver que uma haste de tamanho 4 pode ser vendida inteira por R$ 9, ou vendida em dois pedaços de tamanho 2 por R$10.

Um brilhante aluno de MATD74 observou que este problem possui subestrutura ótima, ou seja, soluções ótimas do problema incorporam soluções ótimas de subproblemas. Tendo isso em mente, o brilhante aluno projetou o seguinte algoritmo.

```
int h_max(int p[ ],int n){
   if ( n == 0 )
      return 0;
   int q = INT_MIN;
   for( int i=1; i <= n; i++ )
      q = max( q, p[ i ] + h_max( p, n-i ) );
   return q;
}
```

No entanto, é fácil perceber que esse algoritmo é ineficiente.

a) Utilize programação dinâmica para tornar o algoritmo `h_max` mais eficiente.

Resposta: Utilizaremos o vetor memo para memorizar a solução de cada subproblema e construiremos um novo algoritmo.

```
int memo[n+1];
for( int i=0; i <= n; i++ )
    memo[i]=INT_MIN; // -2^31
int h_memo_max(int p[ ],int memo[ ], int n){
    if (memo[ n ] >= 0) // Verifica subproblema de tamanho n
        return memo[n];
    int q;
    if ( n == 0 )
        q = 0;
    else {
        q = INT_MIN;
        for( int i=1; i <= n; i++ )
            q = max( q, p[ i ] + h_memo_max( p, memo, n-i ) );
    }
    memo[ n ] = q;
    return q;
}
```

b) Considere a seguinte modificação no problema: além de considerar o preço p_i para cada haste de tamanho i, cada corte possui um custo fixo c. O lucro máximo agora deve considerar a soma dos preços das peças menos a somas dos custos dos cortes. Altere o algoritmo apresentado para resolver essa nova versão do problema.

Resposta: Utilizaremos a variável global c para representar o custo por cada corte.

```
int c;
int memo[n+1];
for( int i=0; i <= n; i++ )
    memo[i]=INT_MIN; // -2^31
int h_memo_max(int p[ ],int memo[ ], int n){
    if (memo[ n ] >= 0) // Verifica subproblema de tamanho n
        return memo[n];
    int q;
    if ( n == 0 )
        q = 0;
    else {
        q = INT_MIN;
        for( int i=1; i <= n; i++ )
            if ( i < n ) // haste sofre um corte
                q = max( q, p[ i ] + h_memo_max( p, memo, n-i ) - c);
            else // haste eh vendida inteira
                q = max( q, p[ i ] + h_memo_max( p, memo, n-i ) );
    }
    memo[ n ] = q;
    return q;
}
```

3. Você decidiu criar uma lista de músicas para a sua próxima viagem para Chapada Diamantina. Você já escolheu as suas músicas favoritas, mas você ainda precisa decidir em qual ordem as músicas devem se tocadas. Sua tarefa é calcular a melhor ordem possível em que essas músicas devem ser tocadas. Para cada par de músicas i e j o "fator de musicalidade" FM_{ij}, que é um número positivo que indica o quão bom você pensa que seria tocar a música j logo depois da música i. Em geral, FM_{ij} e FM_{ji} são diferentes. Utilize o paradigma de programação dinâmica para encontrar a ordenação de músicas que maximize a soma dos fatores de musicalidade das músicas consecutivas. Você não precisa imprimir a sequência de músicas, mas apenas o somatório máximo do fator de musicalidade. Para uma entrada de n músicas, o seu algoritmo deve ser executado com complexidade $\text{O}(n^2 2^n)$.

4. Seja A uma sequência de n inteiros positivos $\{a_1, a_2, \ldots, a_n\}$. Encontre o subconjunto $S \subseteq A$ que tem a soma máxima, com a restrição de que se selecionamos o elemento a_i, não é permitido selecionar o elemento a_{i-1} nem o elemento a_{i+1}. Por exemplo, para o conjunto $A = \{1, 8, 6, 3, 7\}$, nós temos $S = \{8, 7\}$.

5. Teresa quer colher cacaus de uma certa forma, mas não pode colher mais do que ela é capaz de carregar e não pode violar certas restrições sobrenaturais. Durante o processo de colheita, ela notou que se ela colher os cacaus de um cacaueiro i, ela não consegue colher os cacaus do cacaueiro $i-1$, por conta de alguma manifestação que a ciência ainda não é capaz de explicar. Seja N o número de cacaueiros, K o número máximo de cacaus

que Teresa consegue transportas e seja o conjunto $C = \{c_1, c_2, ..., c_n\}$ que expressa a quantidade de cacaus em cada cacaueiro. Preocupada com esse fenômeno, Teresa deseja saber a quantidade máxima de cacaus que ela pode colher, dado o número de cacaueiros, número de cacaus em cada cacaueiro e a quantidade máxima de cacaus que ela pode transportar. **Determine a quantidade máxima de cacaus que Teresa conseguirá transportar**. Por exemplo, para N=5, K = 100 e $C = \{50, 10, 20, 30, 40\}$ o resultado esperado é 90.

```
// Resposta sem uso de programacao dinamica
int Cacaus(int N, int C[ ], int K ){
   if ( N == 0 || K == 0)
      return 0;
   else{
      if( C[N] > K)
         return Cacaus(N-1, C, K);
      else
         return max(Cacaus(N-1, C, K), Cacaus(N-2, C, K - C[N]) + C[N]);
   }
}
```

6. Você recebeu um chamado da ACM - Association for Computing Machinery para resolver um problema de planejamento de cidades. Existem n lotes em uma rua. Edifícios devem ser construídos nesses lotes com a restrição de que edifícios não podem ser construidos em lotes adjacentes para não prejudicar a circulação de ar dentro da cidade. Dado um valor n como entrada, encontre o número de maneiras em que edifícios podem ser construídos nos lotes. Por exemplo, para $n = 2$, existem 3 formas de construirmos tais edifícios: os dois lotes vazios, somente o primeiro lote vazio e somente o segundo lote vazio.

7. Assuma que conhecemos os pesos e valores de n itens, coloque um subconjunto desses itens em uma mochila de capacidade M para obter o valor total máximo na mochila. A sua única restrição é de que a soma dos pesos dos itens deve ser menor ou igual à capacidade da mochila. Por exemplo, dado um conjunto de $n = 4$ itens, em que seus pesos são dados por $w = \{2, 3, 4, 5\}$, seus valores por $v = \{3, 4, 5, 6\}$ e a capacidade da mochila sendo $M = 5$, o valor máximo da mochila será 7 se escolhermos os primeiros dois itens.

8. Dada uma seqüência de n números reais $A(1)...A(n)$, determine uma subseqüência contígua $A(i)...A(j)$ para a qual a soma de elementos na subseqüência é maximizada. Por exemplo, seja $A = \{3, -4, 9, -8, 8, 7\}$. A subsequência contígua máxima é $\{9, -8, 8, 7\}$ tal que a soma de seus elementos é 16.

9. Dado um número inteiro n, determine o número de formas diferentes de se escrever n como a soma dos números 1, 3 e 4.

   Dica: Tão fácil quanto Fibonacci.

10. Dada uma matriz formada por apenas 0s e 1s, encontre a maior submatriz quadrada formada apenas por 1s.

11. Dada uma matriz formada por apenas 0s e 1s, encontre a maior submatriz formada apenas por 1s.

12. Encontre um subvetor com a maior soma possivel. Por exemplo, para a seguinte sequência de valores $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ o segmento de soma máxima é $[4, -1, 2, 1]$ cuja soma é 6.

    Dica: Solução bottom-up:

```
#include<iostream>
using namespace std;
int main(){
   int n = 9;
   int a[n] = {-2,1,-3,4,-1,2,1,-5,4};
   int memo[n+1];//Atencao ao tamanho do vetor !
   int resultado;
   memo[0] = 0;//Qual seria o caso base?
   for(int i = 1; i <= n; i++)
      memo[i] = max(memo[i-1]+a[i-1],a[i-1]);//Como eh a recorrencia?
   resultado = memo[0];//O que temos em cada posicao de memo?
   for(int j=1; j <= n; j++)
      if(resultado < memo[j])
         resultado = memo[j];
   cout<<resultado<<endl;
}
```

13. Dado um valor monetário de n, de quantas formas podemos trocar esse valor dado que temos um suprimento infinito de cada uma das cédulas $S = \{1, 2, 5, 10, 20, 50, 100\}$?

14. Dados números $p_1, p_2, \ldots, p_n, v_1, v_2, ldots, v_n$ e um subconjunto X de $1, 2, \ldots, n$, denotaremos por p(X) e v(X) as somas $\sum_{i \in X} pi$ e $\sum_{i \in X} vi$ respectivamente. Considere o seguinte problema: Dados números naturais $p_1, p_2, \ldots, p_n$, v_1, $v_2, \ldots, v_n$ e c, encontrar um subconjunto X de $1, 2, \ldots, n$ que maximize $v(X)$ sob a restrição $p(X) \leq c$.

15. Suponha que $A[1..n]$ é uma sequência de números naturais. Uma subsequência de $A[1..n]$ é o que sobra depois que um conjunto arbitrário de termos é apagado. Em outras palavras, $B[1..k]$ é uma subsequência de $A[1..n]$ se existem índices $i_1, i_2, \ldots, i_k$ tais que $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ e $B[1] = A[i_1]$, $B[2] = A[i_2]$, $\ldots$, $B[k] = A[i_k]$. Dizemos que o índice 1 de B casa com o índice i_1 de A, que o índice 2 de B casa com o índice i_2 de A, etc. Adotaremos a abreviatura ssc para dizer "subsequência crescente". Uma ssc de A[1..n] é máxima se não existe outra ssc mais longa. Considere o seguinte problema: Encontrar o comprimento de uma ssc máxima de $A[1..n]$.

```
// Dica: pense no um problema mais simples: de modo
// que o indice k de B case com o indice n de A
 int ssc_final_fixo(int* A, int n){
  int c = 1;
   for( i = n-1; i >= 1; i++ )
     if ( A[i] <= A[n] )
        c = max(c, ssc_final_fixo(A,i)+1);
   return c;
}
```

16. Dados números $p_1, p_2, \ldots, p_n$ e um subconjunto X de $\{1, 2, \ldots, n\}$, denotaremos por $p(X)$ a soma $\sum_{i \in X} p_i$. Considere o seguinte problema: Dados números naturais $p1, p2, \ldots, pn$ e c, decidir se existe[1] um subconjunto X de $\{1, 2, \ldots, n\}$ tal que $p(X) = c$.

    Dica: Seja $t[i, b]$ a solução (0 ou 1) da instância $(p_1, \ldots, p_i, b)$ do problema. Assim temos

$$t[i, n] = \begin{cases} t[i-1, b], & \text{se } p_i > b. \\ t[i-1, b] \vee t[i-1, b-p_i], & \text{se } p_i \leq b \end{cases}$$

    onde $\vee$ é o operador lógico OU.

    Dica: Capítulo 15 do Cormen.

17. **Multiplicação de cadeias de matrizes:** Dada uma sequencia de matrizes, determinar a forma mais eficiente de multiplicar estas matrizes.

    Dica:O problema não está na multiplicação, mas sim em determinar a ordem em que as multiplicações serão executadas.

18. **Resolvendo recorrências:** Seja a seguinte recorrência sobre o conjunto dos números naturais:

$$T(n) = \begin{cases} T(n-1) + T(n-3) & \text{se n} \geq 3. \\ 7 & \text{caso contrário} \end{cases}$$

    Escreva um algoritmo que, dado n, calcule o valor do n-ésimo termo da recorrência.

Algoritmos Gulosos

19. O que são Algoritmos Gulosos?

- Para cada um dos seguintes problemas, apresente:

    a) O critério de escolha gulosa a ordem mágica de escolha

    b) Uma solução utilizando os critérios definidos

    c) Análise da complexidade da solução

20. Suponha que três dos seus amigos, inspirados pelo filme de terror The Blair Witch Project, decidiram caminhar pela Trilha dos Apalaches neste verão. Eles querem caminhar o máximo possível por dia, mas, por razões óbvias, não depois do anoitecer. Em um mapa, eles identificaram um grande conjunto de bons pontos de parada para acampar, e eles estão considerando o sistema seguinte para decidir quando parar para o dia. Cada vez que eles vêm a um potencial ponto de parada, eles determinam se eles podem chegar ao próximo ponto antes do anoitecer. Se eles conseguirem, continuarão caminhando; de outra forma, eles param. Apesar de muitos inconvenientes significativos, eles afirmam que este sistema tem Uma boa característica. "Dado que estamos apenas caminhando na luz do dia", afirmam eles, "Isso minimiza o número de paradas de acampamento que temos que fazer." Isso é verdade? O sistema proposto é um algoritmo guloso, e nós queremos determinar se minimiza o número de paradas necessárias. Para tornar essa questão precisa, vamos fazer o seguinte conjunto de simplificações. Modelaremos a Trilha dos Apalaches como um segmento de linha longa de comprimento L, e suponha que seus

[1] 0 ou 1, true ou false, sim ou não, etc.

amigos podem caminhar d milhas por dia (independente de terreno, condições climáticas e assim por diante). Vamos supor que o potencial os pontos de parada estão localizados nas distâncias $x_1, x_2, \ldots, x_n$ desde o início do trilha. Nós também vamos assumir (muito generosamente) que seus amigos estão sempre corretos quando eles estimam se eles podem chegar ao próximo ponto de parada antes anoitecer. Vamos dizer que um conjunto de pontos de parada é válido se a distância entre cada par de pontos adjacentes é no máximo d, sendo também a distância de no máximo d desde o início de a trilha até o primeiro ponto, e a distância de no máximo d do último ponto até o final da trilha. Portanto um conjunto de pontos de parada é válido se alguém pudesse acampar apenas nesses lugares e ainda atravessar toda a trilha. Vamos supor, naturalmente, que o conjunto completo de n pontos de parada são válidos; caso contrário, não seria possível percorre-lo inteiramente. Podemos agora declarar a questão da seguinte maneira. A estratégia do algoritmo guloso dos seus amigos - caminhar o maior tempo possível a cada dia - é ideal, no sentido de encontrar um conjunto válido cujo tamanho é tão pequeno quanto possível?

21. Vamos considerar uma estrada rural longa e tranquila com casas espalhadas escassamente ao longo dela. (Podemos imaginar a estrada como um longo segmento de linha, com um ponto inicial no extremo oriental e um ponto final no extremo ocidental.) Além disso, vamos supor que apesar do cenário bucólico, os moradores de todas essas casas são ávidos usuários de telefone (para compartilharem lindas fotos pelo Instagram). Você quer colocar estações base de telefone celular em certos pontos ao longo da estrada, de modo que cada casa esteja dentro de quatro quilômetros de uma das estações base. Dê um algoritmo eficiente que atinja esse objetivo, usando o menor número de estações base possíveis.

22. Sejam n programas P_1, $P_2, \ldots, P_n$ para serem armazenados em um disco com capacidade D megabytes. O programa P_i requer s_i megabytes de armazenamento. Nós não podemos armazenar todos eles porque $D < \sum_{i=1}^{n} s_i$.

    a) Algum algoritmo guloso que seleciona programas em ordem não decrescente de s_i maximizaria o número de programas mantidos no disco? Prove ou dê um contra-exemplo.

    b) Um algoritmo guloso que seleciona programas em ordem não crescente de s_i usaria o máximo possível da capacidade do disco? Prove ou dê um contra-exemplo.

23. O problema do troco consiste em pagar um trôco com a menor quantidade possível de moedas. No Brasil, temos moedas de 1 real, 50, 25, 10, 5 e 1 centavos. Por exemplo, se tivéssemos que pagar um trôco de R$ 2,78, a solução ótima seria: 2 moedas de 1 real, 1 moeda de 50 centavos, 1 moeda de 25 centavos e 3 moedas de 1 centavo (total de 7 moedas).

24. Dados vetores naturais $(p_1, p_2, \ldots, p_n)$, $(v_1, v_2, \ldots, v_n)$ e um número natural c, encontrar um vetor racional $(x_1, x_2, \ldots, x_n)$ que maximize $x \cdot v$ sob as restrições $x \cdot p^t \leq c$ e $0 \leq xi \leq 1$ para todo i.

25. Tenho um grande número de arquivos digitais no meu computador. Cada arquivo ocupa um certo número de kilobytes. Quero gravar o maior número possível de arquivos num pen drive com capacidade para c kilobytes. O problema pode ser modelado assim: dados números naturais $p_1, p_2, \ldots, p_n$ e c, encontrar o maior subconjunto X de $\{1, 2, \ldots, n\}$ que satisfaça a restrição $\sum_{i \in X} p_i \leq c$. Mostre que um algoritmo guloso apropriado resolve o problema.

26. Suponha um conjunto de livros numerados de 1 a n. Suponha que cada livro i tem um peso $p[i]$ que é maior que 0 e menor que 1. Problema: Acondicionar os livros no menor número possível de envelopes de modo que

    - cada envelope tenha não mais que 2 livros e
    - o peso do conteúdo de cada envelope seja menor ou igual a 1.

    Escreva um algoritmo guloso que resolva o problema em tempo O($nlogn$).

27. Quero dirigir um carro de Salvador até Lençois ao longo das rodovias BR116, BR242 e BR324. O tanque de combustível do carro tem capacidade suficiente para cobrir n quilometros. O mapa indica a localização dos postos de combustível. Dê um algoritmo que garanta uma viagem com número mínimo de reabastecimentos.

28. Mostre, por contra exemplo, que o Problema da mochila booleana não possui solução gulosa.

Backtracking

29. (Fonte: GeeksforGeeks) Um labirinto para ratos é dado como uma matriz binária de dimensões $N \times N$ formada por compartimentos, armazenado em uma matriz *bool labirinto*$[N][N]$ como pode ser visto na Figura 1:

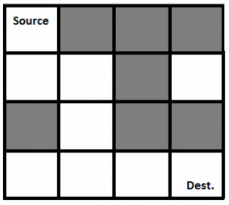

Figura 1: Representação de um labirinto organizado em blocos.

O compartimento de origem é o aquele no canto superior esquerdo, ou seja, o *labirinto*$[0][0]$, e o compartimento de destino é o bloco inferior mais à direita, ou seja, o *labirinto*$[N-1][N-1]$. Um rato começa a percorrer o labirinto a partir da origem e pode ou não chegar ao destino. O rato pode mover-se apenas em duas direções: para frente e para baixo.

Apresente um algoritmo em C/C++ que, para qualquer labirindo de dimensões $N \times N$ com alguns compartimentos bloqueados, identifique, caso exista, um percurso pelo labirinto da origem ao destino. Como saída, seu algoritmo deve imprimir o *labirinto* indicando o caminho percorrido pelo rato, caso tal caminho exista, ou imprimir "Nao existe caminho" caso o caminho da origem ao destino não exista.

```
/* codigo para percorrer o labirinto usando backtracking
   backtracking */
#include<stdio.h>

// Tamanho do labirinto do exemplo
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);
```

```
/* Funcao para imprimir a solucao da matriz sol[N][N] */
void printSolution(int sol[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf(" %d ", sol[i][j]);
        printf("\n");
    }
}

/* Uma funcao para saber se a posicao (x,y) do labirinto eh valida */
bool isSafe(int maze[N][N], int x, int y)
{
    // se (x,y) esta fora do labitindo retorne false
    if(x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
        return true;

    return false;
}

/* Esta funcao resolve o problema do labirinto usando backtracking */
bool solveMaze(int maze[N][N])
{
    int sol[N][N] = { {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0},
        {0, 0, 0, 0}
    };

    if(solveMazeUtil(maze, 0, 0, sol) == false)
    {
        printf("Nao existe caminho");
        return false;
    }

    printSolution(sol);
    return true;
}

/* funcao recursiva para resolver o problema do labirinto */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
    // se a coordenada (x,y) eh o destino retorne true
    if(x == N-1 && y == N-1)
    {
        sol[x][y] = 1;
        return true;
```

```
    }

    // Verifica se maze[x][y] eh uma posicao valida
    if(isSafe(maze, x, y) == true)
    {
        // marque a posicao (x,y) como parte do caminho
        sol[x][y] = 2;

        /* Mova na direcao x */
        if (solveMazeUtil(maze, x+1, y, sol) == true)
            return true;

        /* Se o movimento na direcao x nao der a solucao entao
            mova para baixo na direcao y  */
        if (solveMazeUtil(maze, x, y+1, sol) == true)
            return true;

        /* Se nenhum dos movimentos der certo faca BACKTRACK:
            dermarque o compartimento na coordenada (x,y) como parte da
   solucao */
        sol[x][y] = 0;
        return false;
    }

    return false;
}

// funcao principal para testar o programa
int main()
{
    int maze[N][N]  =  { {1, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 1, 0, 0},
        {1, 1, 1, 1}
    };

    solveMaze(maze);
    return 0;
}
```

30. Sudoku é um quebra-cabeça baseado na colocação lógica de números. O objetivo do jogo é a colocação de números de 1 a 9 em cada uma das células vazias numa grade de 9x9, constituída por 3x3 subgrades chamadas regiões. O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Cada coluna, linha e região só pode ter um número de cada um dos 1 a 9. Resolver o problema requer apenas raciocínio lógico e algum tempo. Os problemas são normalmente classificados em relação à sua realização. Apresente uma modelagem e um algoritmo em C/C++ para resolver este problema.

```
bool SolveSudoku(int grid[N][N])
{
    int row, col;
     // Se todas as posicoes possuem um numero, acabou
    if (!FindUnassignedLocation(grid, row, col))
       return true; // Sucesso!
    // Considere apenas digitos de 1 a 9
    for (int num = 1; num <= 9; num++)
    {
        // se for permitido colocar o numero...
        if (isSafe(grid, row, col, num))
        {
            // ... faca uma tentativa de preenchimento
            grid[row][col] = num;
            // Retorne true se essa atribuicao te fez chegar na resposta
   correta!
            if (SolveSudoku(grid))
                return true;
            // Em caso de falha, desmarque e tente novamente
            grid[row][col] = UNASSIGNED;
        }
    }
    return false; // Esse retorno aciona o backtracking
}
```

31. Iremos definir a brincadeira do cabo de guerra da seguinte forma: Dado um conjunto de n inteiros, cada um correpondendo à força de um participante da brincadeira, divida o conjunto em dois subconjuntos de $n/2$ tamanhos cada um, de modo que a diferença da soma de dois subconjuntos seja a menor possível. Se n for um número par, então os tamanhos de dois subconjuntos devem ser exatamente $n/2$ e se n for ímpar, então o tamanho de um subconjunto deve ser $(n-1)/2$ e o tamanho de outro subconjunto deve ser $(n+1)/2$.

    Por exemplo, considere o conjunto $\{3, 4, 5, -3, 100, 1, 89, 54, 23, 20\}$, de tamanho 10. A saída para este conjunto deve ser $\{4, 100, 1, 23, 20\}$ e $\{3, 5, -3, 89, 54\}$. Ambos os subconjuntos de saída são do tamanho 5 e a soma de elementos em ambos os subconjuntos é igual (148 e 148). Consideremos outro exemplo em que n é ímpar. Seja o conjunto $\{23, 45, -34, 12, 0, 98, -99, 4, 189, -1, 4\}$ de tamanho 11. Os subconjuntos de saída devem ser $\{45, -34, 12, 98, -1\}$ e $\{23, 0, -99, 4, 189, 4\}$. As somas de elementos em dois subconjuntos são 120 e 121, respectivamente.

    Uma forma de resolver este problema é utilizando força bruta, tentando todos os subconjuntos com a metade dos elementos. Se um subconjunto tem a metade dos elementos, os elementos restantes formam o outro subconjunto. Inicializamos o conjunto atual como vazio e, um a um, o construímos. Existem duas possibilidades para cada elemento, seja fazer parte do conjunto atual, ou fazer parte dos elementos restantes (outro subconjunto). Consideramos as duas possibilidades para cada elemento. Quando o tamanho do conjunto atual se torna $n/2$, verificamos se essa solução é melhor do que a melhor solução disponível até o momento. Se assim for, atualizamos a melhor solução. Apresente um algoritmo em $C/C++$ para resolver este problema que imprima os elementos em cada subconjunto.

```
#include <iostream>
#include <cmath>
using namespace std;

int n;
int *cabo;
int *caboOtimo;
int entropia;

int soma(int l, int r){
   int total = 0;
   for(int i=l; i < r; i++)
      total += cabo[i];
   return total;
}

void swap(int x, int y)
{
    int temp;
    temp = cabo[x];
    cabo[x] = cabo[y];
    cabo[y] = temp;
}

void guerra(int l, int r)
{
   int i;
   if (l == r){
      if(entropia > abs(soma(0,n/2) - soma(n/2,n)))
      {
         entropia = abs(soma(0,n/2) - soma(n/2,n));
         for(int i=0; i < n; i++)
            caboOtimo[i] = cabo[i];
      }
      if(entropia == 0)
         return;
   }
   else
   {
       for (i = l; i <= r; i++)
       {
          swap(l, i);
          guerra(l+1, r);
          swap(l,i); //backtrack
       }
   }
}

int main()
```

```
{
   entropia = INT_MAX;
   cout<<"Numero de participantes: ";
   cin>>n;
   cabo = new int[n];
   caboOtimo = new int[n];
    cout<<"Forca de cada participante: "<<endl;
    for(int i=0; i < n; i++)
      cin>>cabo[i];
    guerra(0, n-1);

    // Imprime a saida
    cout<<"{";
    for(int i=0; i < n/2; i++){
      cout<<caboOtimo[i];
      if(i+1 < n/2)
         cout<<",";
   }
   cout<<"}-{";
   for(int i=n/2; i < n; i++){
      cout<<caboOtimo[i];
      if(i+1 < n)
         cout<<",";
   }
   cout<<"} = "<<entropia<<endl;

   delete [] cabo;
   delete [] caboOtimo;

    return 0;
}
```

Algoritmos Gulosos vs Programação Dinâmica

Às vezes é difícil distinguir um algoritmo guloso de um algoritmo de programação dinâmica. A seguinte lista grosseira de características pode ajudar.

Um Algoritmo Guloso:

- escolhe a alternativa mais promissora (sem explorar as outras),
- nunca se arrepende de uma decisão já tomada,
- é muito rápido,
- não tem prova de correção simples.

Um algoritmo de Programação Dinâmica

- explora todas as alternativa (mas faz isso de maneira eficiente),
- a cada iteração pode se arrepender de decisões tomadas anteriormente,
- é um tanto lento,
- tem prova de correção simples.