Universidade Federal de Viçosa
Campus Rio Paranaíba
Instituto de Ciências Exatas e Tecnológicas

SIN 343
Desafios de Programação

João Batista Ribeiro
joao42lbatista@gmail.com

Slides baseados no material do prof. Guilherme C. Pena

Universidade Federal de Viçosa
Campus Rio Paranaíba
Instituto de Ciências Exatas e Tecnológicas

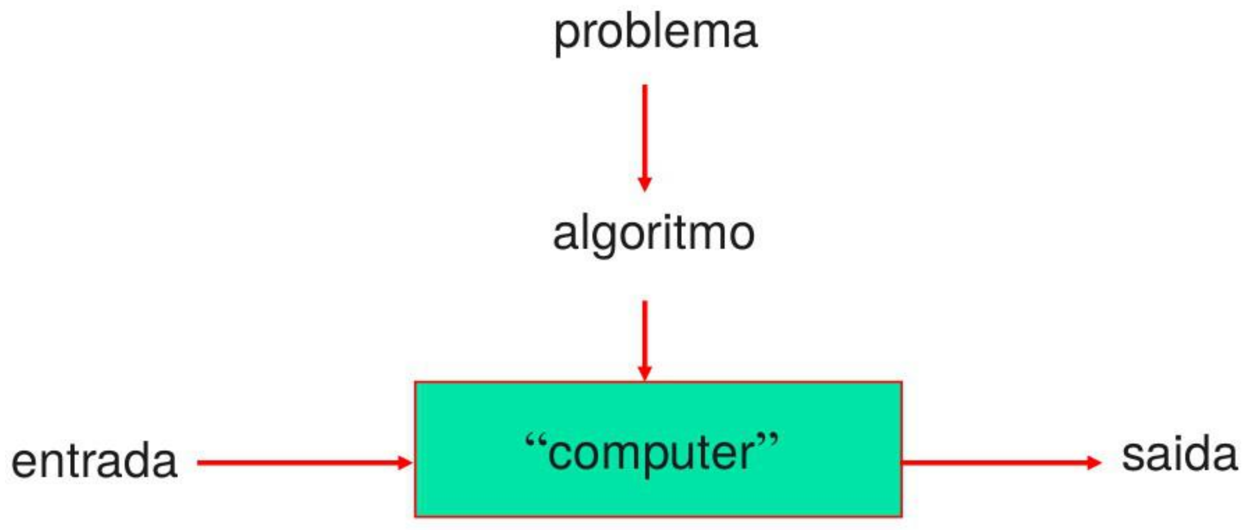
Aula de Hoje

Projeto de Algoritmos

Projeto de Algoritmos

Algoritmos:

Um ***algoritmo*** é uma sequência de instruções não ambíguas para resolver um problema, isto é, para obter uma determinada saída a partir de uma entrada válida dentro de um espaço de tempo finito.



Projeto de Algoritmos

Perspectiva Histórica:

O primeiro algoritmo de que se tem notícia é o *algoritmo de Euclides* para obtenção do **máximo divisor comum (mdc)** descrito no século 3 *B.C*.

Problema: Dados dois números m e n, ambos não negativos e pelos menos um diferente de zero, obter o máximo divisor comum mdc(m,n).

Por exemplo, ***mdc (60,24) = 12*** e ***mdc (60,0) = 60***

Projeto de Algoritmos

Perspectiva Histórica:

O algoritmo de Euclides é baseado na aplicação iterativa da igualdade:

$$\mathbf{mdc(m,n) = mdc(n,\ m\ mod\ n)}$$

até que o segundo número se torne zero.

Passo 1 : Se n = 0, returne m e pare; senão vá para o passo 2

Passo 2 : Divida m por n e atribua o valor do resto a r;

Passo 3 : Atribua o valor de n a m e o valor de r a n; Vá para o passo 1.

```
if(n == 0)
     cout << m << endl;
else{
    while(n != 0){
        r = m%n;
        m = n;
        n = r;
    }
    cout << m << endl;
}
```

Projeto de Algoritmos

Principais Estratégias:

- Força Bruta
- Dividir para Conquistar
- Programação Dinâmica
- Backtracking (Tentativa e Erro) e Branch-and-Bound
- Estratégia Gulosa

Projeto de Algoritmos

Crescimento das funções:

Em relação ao tamanho da entrada

n	$Log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$	n^n
10	3,3	10	$3{,}3\times10$	10^2	10^3	10^3	$3{,}6\times10^6$	10^{10}
10^2	6,6	10^2	$6{,}6\times10^2$	10^4	10^6	$1{,}3\times10^{30}$	$9{,}3\times10^{157}$	10^{200}
10^3	10	10^3	$1{,}0\times10^4$	10^6	10^9			
10^4	13	10^4	$1{,}3\times10^5$	10^8	10^{12}			
10^5	17	10^5	$1{,}7\times10^6$	10^{10}	10^{15}			
10^6	20	10^6	$2{,}0\times10^7$	10^{12}	10^{18}			

Projeto de Algoritmos

Alguns problemas computacionais conhecidos:

- **Ordenação**
- **Busca**
- **Grafos**: caminhamento, fluxo, árvore geradora mínima, emparelhamento, cobertura de vértices e arestas, coloração, sub-grafos completos, caxeiro viajante
- **Mochila**
- **Par mais próximo, casco convexo**
- Dentre outros..

Projeto de Algoritmos

Principais Estruturas de Dados:

- **Lineares:** arranjos, listas, pilhas e filas
- **Árvores:** binária de pesquisa, árvore-B, quadtree, octree, Rtree, Kdtree
- **Grafos**
- Dentre outras..

Força Bruta

A força bruta é a técnica mais simples de projeto de algoritmos baseado, usualmente, no enunciado do problema e definições/conceitos envolvidos. Geralmente, é uma das mais fácil de aplicar.

Para alguns problemas importantes, a técnica de força bruta pode fornecer algoritmos de valor prático (eficientes), algoritmos razoáveis ou algoritmos ineficientes (para instancias grandes de problemas)

Força Bruta

Exemplos:

- Calcular n!
- Multiplicação de duas matrizes de ordem nXn.
- Ordenação por seleção (selection sort)
- Busca sequencial

Força Bruta

Uma primeira aplicação da técnica de força bruta geralmente resulta em um **algoritmo que pode ser melhorado** com uma quantidade modesta de esforço.

Para problemas mais complexos, raramente fornece algoritmos eficientes.

Força Bruta

Cálculo do MDC(m, n):

Algoritmo de força bruta que procura consecutivamente um inteiro t que seja divisor de m e n ($1 \leq t \leq \min(m, n)$):

```
int MDC (int m, int n){
    int t = min(m, n);
    while(t >=1)
    if((m%t == 0) && (n%t == 0))
        return t;
    else
        t = t – 1;
}
```

Para m = 60 e n = 24, os inteiros t testados são: 24, ..., 12. Sendo 12 o mdc.

Força Bruta

Ordenação por Seleção: Θ(n^2)

1. Varrer a toda a lista para encontrar o menor elemento e trocá-lo com o primeiro elemento.
2. Depois varrer a lista a partir do segundo elemento para encontrar o segundo menor (dentre os n-1 elementos) e trocá-lo com o segundo elemento.
3. Repetir este processo n-1 vezes

```
void SelectionSort(int vetor[], int tam) {
    for (int i = 0; i < tam; ++i) {
        int menor = i;
        for (int j = i+1; j < tam; ++j) {
            if (vetor[j] < vetor[menor])
                menor = j;
        }

        int aux = vetor[i];
        vetor[i] = vetor[menor];
        vetor[menor] = aux;
    }
}
```

Força Bruta

Comparação de Cadeia de Caracteres:

Dada uma string T de n caracteres chamada de texto, e uma string P de m caracteres (m≤n) chamada de padrão. Encontrar uma ***substring de T*** que coincida com ***P***.

Exemplo: **padrão = NOT**

```
N O B O D Y _ N O T I C E D _ H I M
N O T
  N O T
    N O T
      N O T
        N O T
          N O T
            N O T
              N O T
```

SIN343 - Desafios de Programação

Força Bruta

Comparação de Cadeia de Caracteres: Θ(nm)

Devemos encontrar o primeiro índice i de T tal que:

$T[i] = P[0], T[i+1]=P[1], \ldots., T[i+j]=P[j], \ldots., T[i+m-1]=P[m-1],$

Para o exemplo anterior: i = 7

```
int StringMatch(char T[], char P[], int n, int m){
    int i, j;
    for(i = 0; i < n; i++){
        j = 0;
        while((j < m) && (P[j] == T[i+j]))
            j = j + 1;

        if(j == m)
            return i;
    }
    return -1; //busca sem sucesso
}
```

Força Bruta

Par mais Próximo:

Dado um conjunto de n (≥ 2) pontos no plano R^2. Determinar os dois pontos mais próximos (menor distância).

A distância entre os pontos $p_i = (x_i, y_i)$ e $p_j = (x_j, y_j)$ é determinada pela distância Euclidiana:

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Algoritmo de Força Bruta: Calcular a distância entre todos os pares de pontos diferentes e identificar os pontos com menor distância. Retornar o índice dos dois pontos.

Força Bruta

Par mais Próximo: $\Theta(n^2)$

```
struct Ponto {
    float x, y;
};
void PontosProx(Ponto P[], int n, int &i1, int &i2) {
    float dmim = “infinito”;

    for(int i = 0; i < n-1; i++)
        for(int j = i+1; j < n; j++) {
            d = sqrt((P[i].x-P[j].x)*(P[i].x-P[j].x)
 + (P[i].y-P[j].y)*(P[i].y-P[j].y));
            if(d < dmim) {
                dmin = d;
                i1 = i;
                i2 = j;
            }
        }
}
```

Busca Exaustiva

Em muitos problemas importantes é necessário **encontrar um elemento** com certas propriedades **dentro de um domínio que cresce exponencialmente** com o tamanho da instância do problema.

Tipicamente, tais problemas envolvem (explícita ou implicitamente) objetos combinatórios tais como **permutações**, **combinações** ou **subconjuntos** de um dado conjunto.

Busca Exaustiva

Muitos são **problemas de otimização combinatória** que consistem em encontrar um **elemento (solução) que minimize** ou **maximize** uma função objetivo satisfazendo algumas condições (restrições).

A **busca exaustiva** consiste simplesmente numa estratégia de força bruta para resolver problema combinatórios. Isto é, a solução é obtida analisando-se todas as possibilidades para obter a "***melhor solução***".

Busca exaustiva:

- Gere (de maneira sistemática) a lista de todas possíveis soluções;
- Avalie uma a uma as soluções possíveis e elimine as soluções não viáveis;
- Armazene a melhor solução obtida até então.

Busca Exaustiva

Problema da Mochila:

Dados n itens (objetos) cujos pesos são w_1, w_2, ..., w_n e os custos são c_1, c_2, ..., c_n, determine o subconjunto de itens mais valiosos (de maior custo total) que caibam dentro de uma mochila de capacidade W.

Algoritmo de Força Bruta:

- Determinar todos os subconjuntos de itens
- Dentre os subconjuntos viáveis (i.e. que satisfazem a capacidade da mochila), determinar o subconjunto de itens com o maior custo total.

Complexidade: $\Theta(2^n)$

Busca Exaustiva

Problema da Designação:

Dadas n pessoas e n tarefas a serem executadas sendo que cada tarefa deve ser executada por exatamente uma pessoa. Determine a designação de menor custo total sabendo que a designação de uma pessoa i a uma tarefa j tem um custo c_{ij} (i, j, = 1,...,n).

Algoritmo de Força Bruta:

- Determinar todas as permutações de 1 a n (cada permutação representa uma possível solução)
- Calcular o custo de cada solução e selecionar solução de menor custo total.

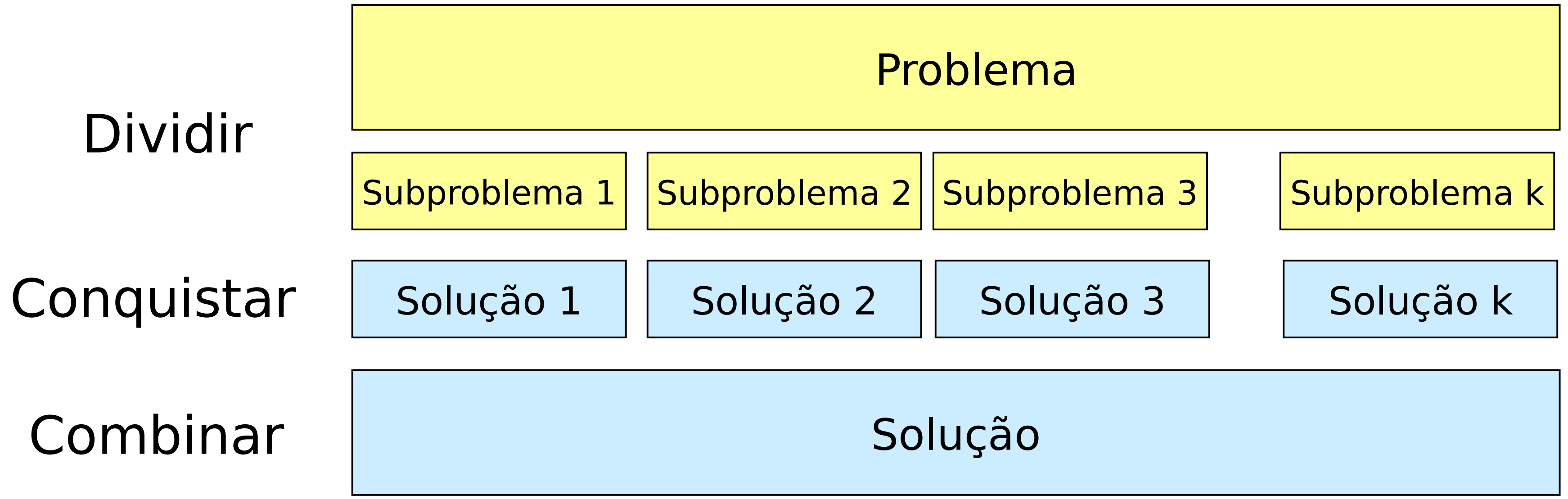
Complexidade: Θ(n!)

Divisão e Conquista

Divisão e Conquista

Divisão e Conquista é uma das técnicas mais importantes e eficientes em Ciência da Computação.

Graficamente,



Divisão e Conquista

Muitos problemas de Ciência da Computação podem ser resolvidos da forma eficiente utilizando a técnica de **divisão e conquista (DC)**.

Exemplos:

- Busca Binária
- Ordenação: mergesort e quicksort
- Percurso em árvores
- Multiplicação de Matrizes
- Par mais próximo
- Casco convexo

Divisão e Conquista

Em geral os problemas são reduzidos em subproblemas de forma recursiva até que atingimos um ***caso-base***.

Às vezes, além de subproblemas que são instâncias menores do mesmo problema original, temos de resolver subproblemas que não são exatamente iguais ao original.

Assim, consideramos a solução de tais problemas como parte da etapa de "**combinar**".

Divisão e Conquista

Busca Binária:

Problema: Encontrar um valor *x* num vetor ordenado *V*

Algoritmo Divisão e Conquista:
Compare *x* com o elemento da posição *n/2*. Se for igual fim.

Se x < V[n/2] podemos afirmar que *x* não está dentre os elementos V[n/2] .... V[n] e portanto, basta buscar x na parte V[1] ... V[n/2 – 1] do vetor.

Por outro lado, se x > V[n/2], de maneira semelhante, basta buscar na parte V[n/2 + 1] ... V[n].

Divisão e Conquista

Busca Binária: Θ(log n)

Problema: Encontrar um valor *x* num vetor ordenado *V*

```
int rec_buscaBin(int * v, int x, int inicio, int fim) {
    if(inicio > fim) // 1o caso base
        return -1;

    int meio = (inicio + fim)/2;
    if(x == v[meio]) // 2o caso base
        return meio;
    else if(x > v[meio])
        return rec_buscaBin(v, x, meio+1, fim);
    else
        return rec_buscaBin(v, x, inicio, meio-1);
}
```

Divisão e Conquista

Busca Binária: Θ(log n)

Problema: Encontrar um valor *x* num vetor ordenado *V*

```
int it_buscaBin(int * v, int x, int inicio, int fim) {
    int meio;
    while(inicio <= fim) {

        meio = (inicio + fim)/2;
        if(x == v[meio])
            return meio; // encontrou elemento
        else if(x > v[meio])
            inicio = meio + 1;
        else
            fim = meio - 1;
    }
    return -1; // se não encontrar retorna -1
}
```

Divisão e Conquista

Ordenação por Intercalação (*MergeSort*):

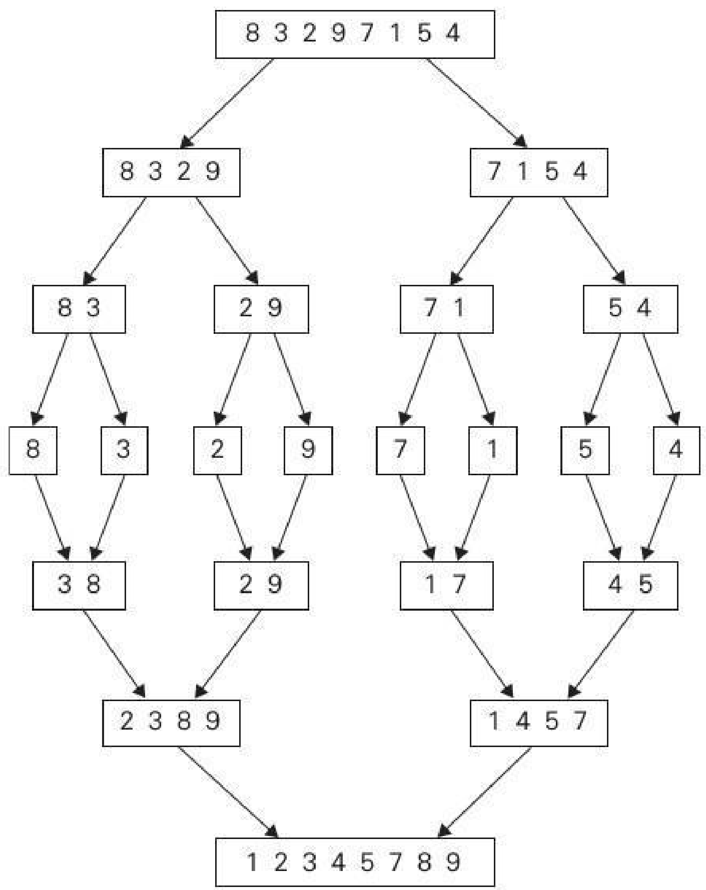
Algoritmo Divisão e Conquista:

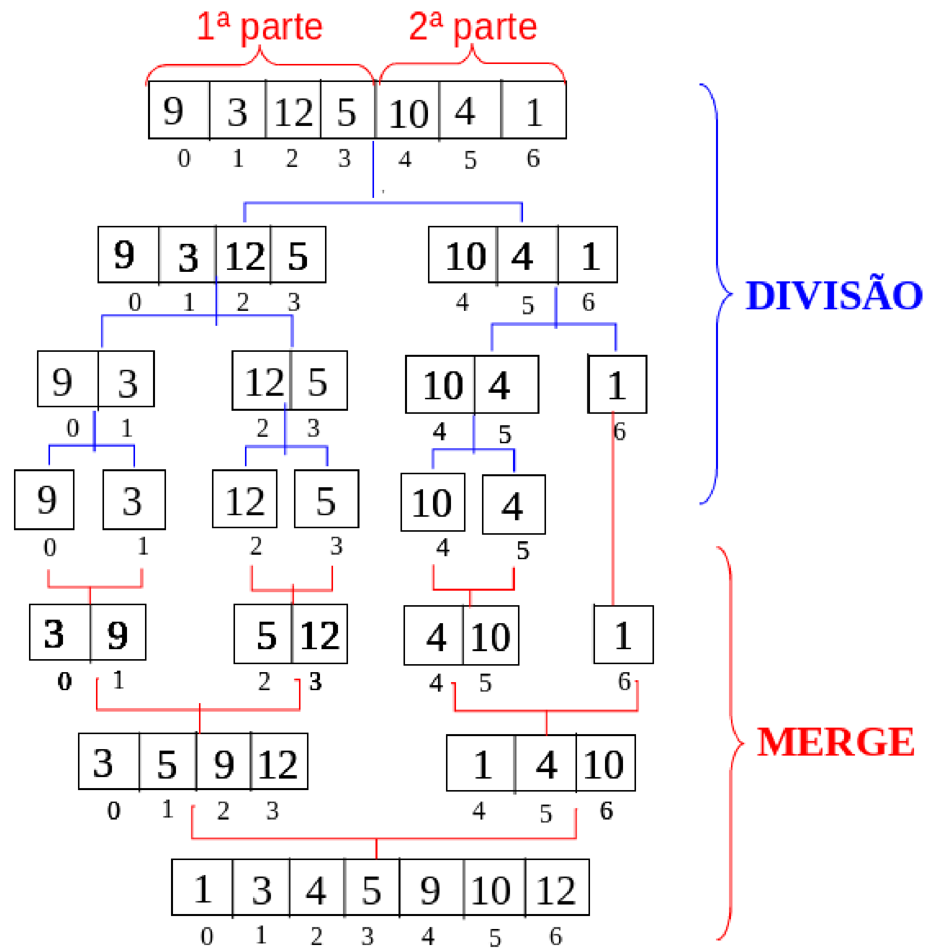
Divide um arranjo A [0 ... n -1] em duas partes A[0 ... (n/2)-1] A[n/2 ... n-1], ordena as partes obtidas recursivamente (até não poder dividí-las mais).

Ao final, faz o "**merge**" (intercalação) dos elementos das duas partes ordenadas, obtendo a ordenação do arranjo original.

Divisão e Conquista

Ordenação por Intercalação (*MergeSort*):





Divisão e Conquista

Ordenação por Intercalação (*MergeSort*):

```
void MergeSort(int vet[], int ini, int fim){
    if(ini < fim) {
        int meio = (ini + fim)/2;

        MergeSort(vet, ini, meio);
        MergeSort(vet, meio+1, fim);

        Merge(vet, ini, meio, fim);
    }
}
```

```
MergeSort(vetor, 0, tam-1);
```

Complexidade: **Θ(n log n)**

Divisão e Conquista

Ordenação por Intercalação (*MergeSort*):

```
void Merge(int vet[], int ini, int meio, int fim) {
    int *vetAux = new int [fim - ini +1];
    int i = ini, j = meio + 1, k = 0;

    while(i <= meio && j <= fim) {
        if(vet[i] < vet[j]) { vetAux[k] = vet[i]; i++;
        } else { vetAux[k] = vet[j]; j++; }
        k++;
    }
    while(i <= meio) { vetAux[k] = vet[i]; i++; k++; }

    while(j <= fim) { vetAux[k] = vet[j]; j++; k++; }

    for(i = ini, k = 0; i <= fim; i++, k++)
        vet[i] = vetAux[k];

    delete [] vetAux;
}
```

Divisão e Conquista

O problema do Subarranjo Máximo:

Dado um conjunto de valores, seu objetivo é determinar um subarranjo contíguo deste conjunto cujos valores resultem na maior soma.

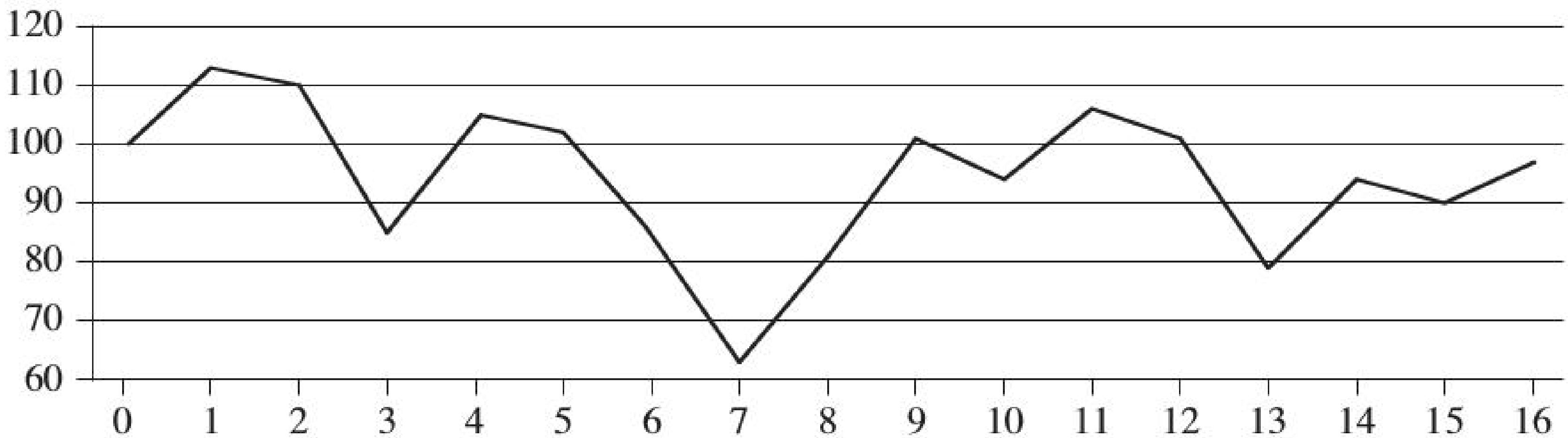
Supondo uma situação:
Você ganhou a oportunidade de investir em uma empresa comprando uma única unidade de ação e vendendo-a em uma data posterior.

A operação de compra ou venda só pode ser realizada após o fechamento do pregão do dia e para compensar essa restrição, você pode saber qual será o preço da ação no futuro. Sua meta é maximizar o seu lucro.

Divisão e Conquista

O problema do Subarranjo Máximo:

Suponha o preço da ação num período de 17 dias.



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Você pode comprá-la em qualquer dia após o dia 0, quando o preço é $100,00

Divisão e Conquista

O problema do Subarranjo Máximo:

Força Bruta: Experimentar todo par possível de datas de compra e venda no qual a data de compra seja anterior à data de venda. Neste caso teríamos $\binom{n}{2}$ combinações levando em conta um período de n dias. Na complexidade daria algo como (**n²**).

Divisão e Conquista

O problema do Subarranjo Máximo:

Observação: O problema é interessante somente quando o arranjo contém alguns números negativos. Se todas as entradas do arranjo fossem não negativas, não haveria desafio já que o arranjo inteiro daria a maior soma.

Divisão e Conquista

O problema do Subarranjo Máximo:

Divisão e Conquista: Pensando em termos de DC, suponha que queiramos determinar o subarranjo máximo dentro de A[low ... high].

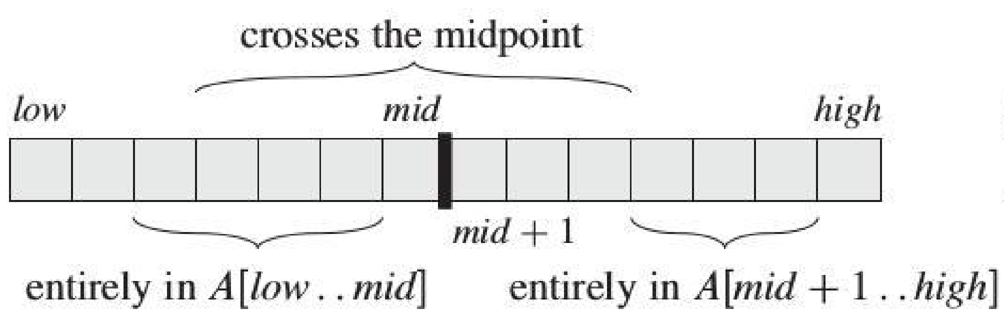
O método DC sugere que dividamos o subarranjo em dois subarranjos com tamanhos mais iguais possíveis supondo um ponto médio (mid): A[low … mid] e A[mid+1 … high].

Divisão e Conquista

O problema do Subarranjo Máximo:

Desta forma, qualquer subarranjo contíguo A[i ... j] dentro de A[low ... high] deve-se encontrar exatamente em um dos seguintes lugares:

- Inteiramente no subarranjo A[low ... mid], low ≤ i ≤ j ≤ mid
- Inteiramente no subarranjo A[mid+1 ... high], low < i ≤ j ≤ mid
- cruzando o ponto médio, low ≤ I ≤ mid < j ≤ high



Divisão e Conquista

O problema do Subarranjo Máximo:

Portanto, um subarranjo máximo de A[low … high] deve-se encontrar exatamente em um dos 3 casos.

Podemos determinar subarranjos máximos de A[low ... mid] e A[mid+1 ... high] de forma recursiva, pois são subproblemas menores do problema original.

Assim, resta apenas encontrar um subarranjo máximo que cruze o ponto médio e tomar o subarranjo que tenha a maior soma dos 3.

Divisão e Conquista

O problema do Subarranjo Máximo:

No entanto, o subarranjo que cruza o ponto médio não é um subproblema do original porque existe essa restrição de ter que cruzar o ponto médio.

Porém, qualquer subarranjo A[i ... j] que cruza o ponto médio é composto por dois subarranjos A[i ... mid] e A[mid+1 ... j].

Portanto, precisamos apenas encontrar subarranjos máximos da forma A[i ... mid] e A[mid+1 ... j] e então combiná-los.

Divisão e Conquista

O problema do Subarranjo Máximo:

O procedimento seguinte toma o arranjo A, os índices, low, mid e high e retorna uma tupla que contém os índices que demarcam o subarranjo máximo que cruza o ponto médio e a soma dos valores em um subarranjo máximo.

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

Divisão e Conquista

O problema do Subarranjo Máximo:

Com o procedimento feito, o pseudocódigo para o algoritmo DC ficaria assim:

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Divisão e Conquista

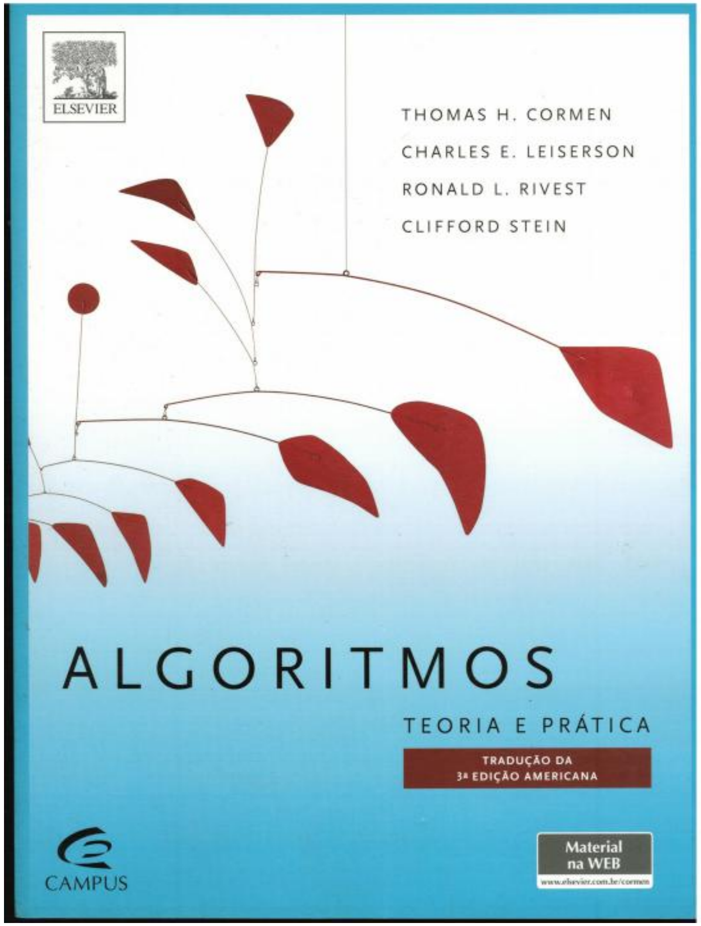
O problema do Subarranjo Máximo:

Esta solução tem complexidade: **Θ(n log n)**

Ver arquivo: **MaxSubArranjo.cpp**

Referência Bibliográfica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. Tradução da 2. ed. Americana. Rio de Janeiro: Campus, 2002.



Referência Bibliográfica

- SKIENA, S.S. REVILLA, M. A. Programming challenges: the programming contest training manual. Springer, 2003.