

Complexidade Algorítmica & Notação Assintótica

Como comparar complexidade de algoritmos diferentes? Notação assintótica!

- Aviso: a representação numérica do tamanho do input é em base 2 ou mais
- Interessa principalmente o cenário do pior caso

Técnica de Construção de Algoritmos:

1. Força Bruta
2. Algoritmos Guloses/Gananciosos
3. Dividi e Conquistar
4. Programação Dinâmica
5. Escavativo com Back-Tracking
6. Aproximação

Dividi e Conquistar:

1. critério de divisão
2. divisão (para a frente)
3. junção (para trás)

- Notação assintótica permite definir taxas de crescimento em função do tamanho do input
- Constantes são irrelevantes (assintoticamente)

- O: limite assintótico superior - "upper bound" - "no máximo"
 Ω : limite assintótico inferior - "lower bound" - "no mínimo"
 Θ : limite assintótico apertado - "igual" - "exatamente" - $O = \Omega$

ESTRUTURAS BÁSICAS DE DADOS

Set: elementos desordenados e não repetidos } inserir/remover/procurar
Bag: elementos desordenados & podem ser repetidos }

Queue: First-In-First-Out (FIFO) - inserir no fim; remover da frente

Stack: Last-In-First-Out (LIFO) - inserir e remover do topo

Tree: estrutura hierárquica pai/filho, equilibrada ou desequilibrada, completa ou incompleta $\log(n)$ n

Graph: nós e arestas, dirigido ou não dirigido, cíclico ou acíclico

Hash-Table: mapeamento entre domínios

Pesquisa Linear: array desordenado - $O(n)$

Pesquisa Binária: array ordenado - $O(\log(n))$

Min-Heap: array de valores interpretado como árvore binária em que
 $\text{Value}(A[\text{Parent}(i)]) \leq \text{Value}(A[i])$

Extract Min: retomar elemento do topo e reajustar - $O(\log(n))$

Shift Down: ajustar a min-heap - $O(\log(n))$

Shift Up: inverso de Shift Down - $O(\log(n))$

Insert: aumentar o tamanho e colocar elemento no fim - $O(\log(n))$

preorder(node *n){
visit(n); for all children
preorder(c);}

postorder(node *n){
for all children postorder(c);
visit(n);}

inorder(node *n){
inorder(n.left);
visit(n);
inorder(n.right);}

Ordenação

Como ganha informação? Comparar chaves

- Há $n!$ permutações de chaves
- Se todos as comparações não forem, a altura da árvore é $O(\log(n!))$
- O limite inferior de ordenação por comparação é $\Omega(n \log(n))$

Heap Sort: $\mathcal{O}(n \log(n))$

1. Selecionar o elemento do topo da max/min-heap e reorganizá-la
2. Inserir elemento no fim do array (ou da própria heap)
3. Quando todos os elementos foram extraídos da heap, o algoritmo termina

Quick Sort: $\mathcal{O}(n \log(n))$

1. Dividi o input em duas sequências usando um elemento pivot
2. Recursivamente, ordena cada sequência "in-place"

Counting Sort: $\mathcal{O}(n)$

1. Conta o número de ocorrências de cada chave
2. Insere cada chave no lugar certo do array ordenado
- É um algoritmo de ordenação estável

Radix Sort: $\mathcal{O}(K n)$

1. Inicia a ordenação usando o dígito/bit menos significativo
2. Quando todos os valores foram ordenados para todos os dígitos/bits, o algoritmo termina
- Requer um algoritmo interno de ordenação estável

Merge Sort: $\mathcal{O}(n \log(n))$

Algoritmos Básicos de Grafos

Componentes Fortemente Conexas: um grafo dirigido $G = (V, E)$ tem um componente fortemente conexo SCC como o maior conjunto de nós $U \subseteq V$ tal que $u, v \in U$, u é alcançável desde v e v é alcançável desde u (um nó sozinho é um SCC)

Grafo Reverso: $G^T = (V, E^T)$ tal que $E^T = \{(u, v) : (v, u) \in E\}$

• G e G^T têm os mesmos SCCs

function $\text{SCCs}(\text{Graph } G)$

↳ Gerar grafo reverso G^R

↳ Executar DFS(G^R) e contar a traescia em pós-ordem T

↳ Executar DFS(G) e visitar nós em pós-ordem reversa T^R

↳ Cada árvore DFS corresponde a um novo SCC

Complexidade: $O(V + E)$

Ordenação Topológica: de um DAG $G = (V, E)$ é uma ordenação de nós (linearização) tal que $(u, v) \in E$ então u aparece antes de v na ordenação

- a) Write in pseudo-code an algorithm to remove an edge from a graph, given the unique identifiers of its source and destination vertices.
- b) Indicate and justify the temporal complexity of the proposed algorithm, in the worst case, with respect to V and E, which denote, respectively, the graph's number of vertices and edges.

```

bool removeEdge (G, mcId, dtId) {
    O(1)   srcV = NULL;
    O(V) { for (v ∈ G.V)
            if (v.id == mcId)
                mcV = v
    O(1) { if (mcV == NULL)
            return false;
    O(E) { for (e ∈ mcV.E)
            if (e.id == dtId) {
                deleteEdge (e);
                return true;
            }
    return false;
}
}

```

$$O(1) + O(V) + O(1) + O(E) = O(V + E)$$

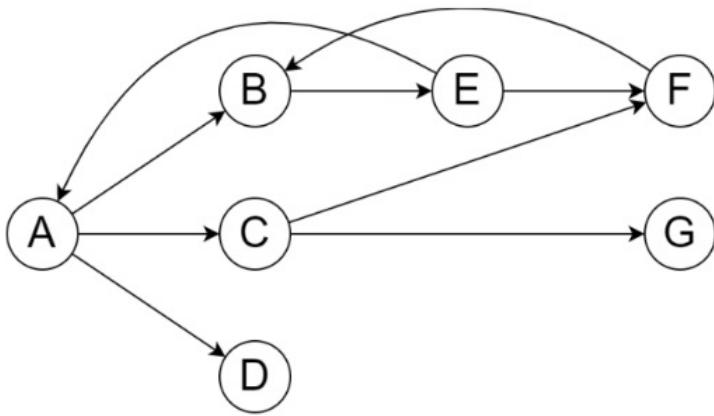
- d) Write in pseudo-code an algorithm to remove a vertex from a graph, given its unique identifier.
e) Indicate and justify the temporal complexity of the proposed algorithm, in the worst case, with respect to V and E, which denote, respectively, the graph's number of vertices and edges.

```

local removeVertex (G, id) {
    O(1)   v = NULL;
    O(V)   { for (n ∈ G.V)
              if (n.id == id)
                  v = n;
    O(1)   if (v == NULL)
              return false;
    O(E)   { for (e ∈ v.E)
              deleteEdge (e);
    O(E)   { for (e ∈ G.E)
              if (e.id == v.id)
                  deleteEdge (e);
    return true;
}

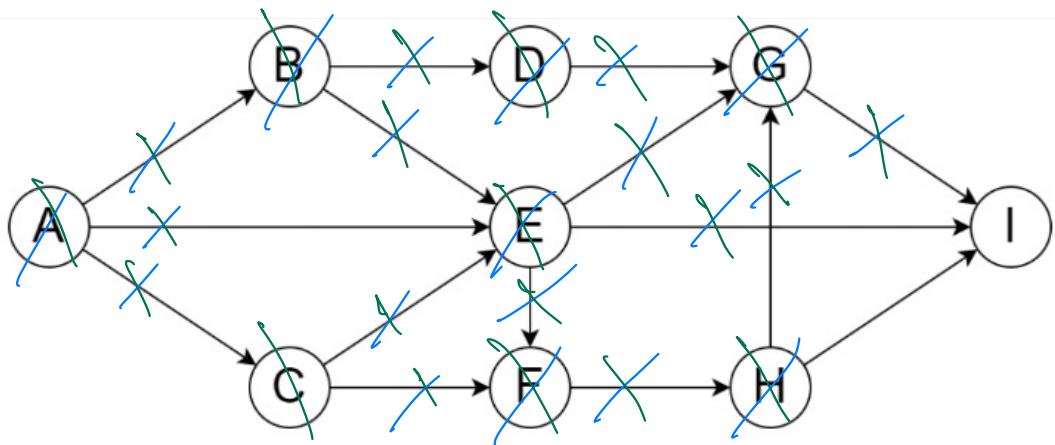
```

$$O(1) + O(V) + O(1) + O(E) + O(E) = O(V+E)$$



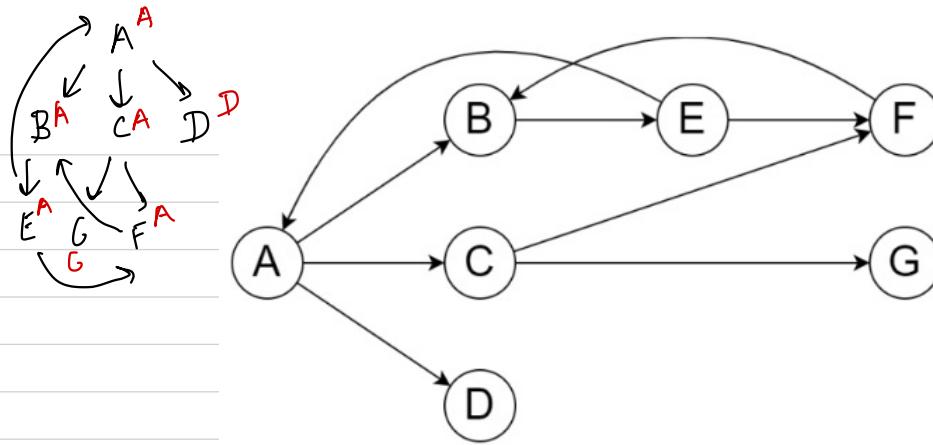
DFS: A, B, E, F, C, G, D

BFS: A, B, C, D, E, F, G
Q: A, B, C, D, E, F, G



Ordenação Topológica:

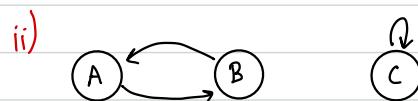
A, B, C, D, E, F, H, G, I
 A, C, B, E, D, F, H, G, I



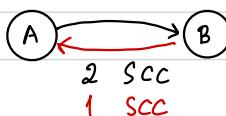
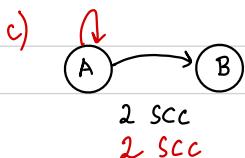
SCC: {A, B, C, E, F}, {D}, {G}

Considering directed graphs:

- Draw a directed graph with 3 vertices, at least 3 edges and (i) 1 SCC; (ii) 2 SCCs; (iii) 3 SCCs.
- What happens to SCC's when you add an edge to a directed graph?
- Show an example when adding an edge does not change the SCC's, and one where it does.



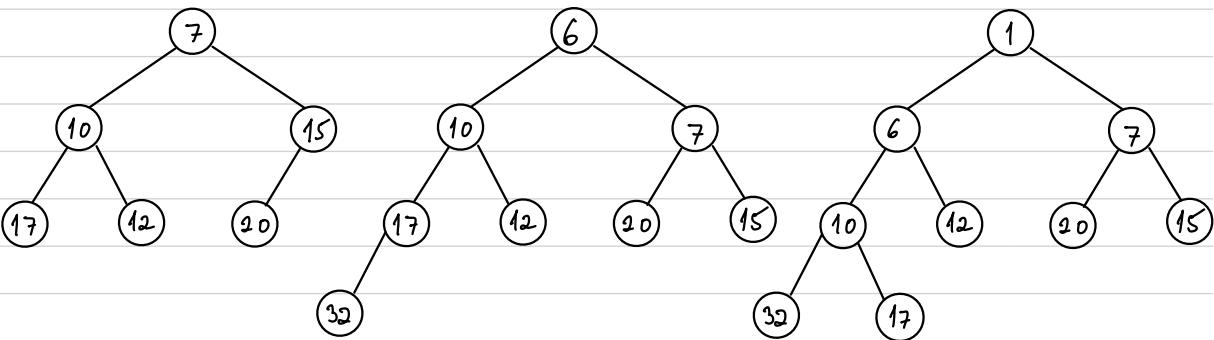
b) Se a aresta for adicionada entre vertex do mesmo SCC, o número de SCC mantém-se; Não, o número de SCC não pode diminuir ou manter-se, nunca aumentar



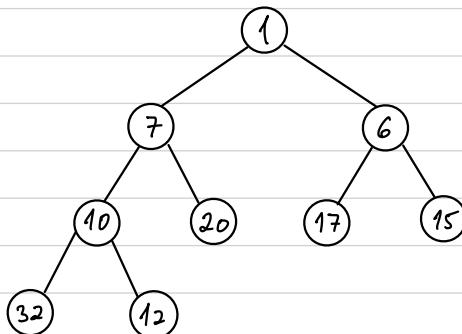
Consider the min-heap data structure using a binary tree. For this data structure:

- a) Show the organization of this min-heap before and after the insertion of the numbers 6 and 1, regarding the following sequence of insertion 10, 7, 15, 17, 12, 20, 6, 32, 1. Then, consider the insertion order 1, 32, 6, 20, 12, 17, 15, 7, 10. What do you conclude?

~~10, 7, 15, 17, 12, 20, 6, 32, 1~~



~~1, 32, 6, 20, 12, 17, 15, 7, 10~~



Conclusão: a ordem de inserções é relevante para a organização dos valores na heap e só a não desequilibrado tem de ser ajustado

This function sorts the provided vector in increasing order by using a heap as an auxiliary data structure.

- c) Indicate and justify the temporal complexity of *heapSort*, with respect to n, which denotes the number of elements of the vector to sort.

Uma função que tem de, em primeiro lugar, construir uma heap, isto é, inserir n elementos, o que tem complexidade $O(n \times \log(n))$, pois a inserção de um elemento tem complexidade $O(\log(n))$. Depois, o algoritmo tem de percorrer toda a heap (complexidade $O(n)$), retirando, em cada iteração, o elemento que estava no topo e colocando-o no final da heap (complexidade $O(h)$, sendo h a altura da heap), e que resulta em complexidade $O(n \cdot h) = O(n \cdot \log(n))$, $h = \log(n)$. Assim, as complexidades dos dois passos não são iguais, pelo que se conclui que a complexidade da função é $O(n \log(n))$.

Implement an algorithm that computes the kth smallest element in a set of n positive integers in $O(n + k \lg n)$ time, with the aid of a heap. If there are less than k elements in the set, then the algorithm should return -1.

```
int kthSmallest(unsigned int k, std::vector<int> v)
```

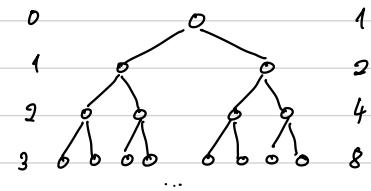
Justify why the algorithm has the desired complexity.

```
int kthSmallest (unsigned int k, std::vector<int> v) {
    O(1) { if (v.size() < k)
            return -1;
    O(1) minHeap heap (v.size());
    O(n) { for (i in v)
            heap.insert (i);
    O(k * log(n)) for (j = 1; j < k; j++)
        res = heap.top(); heap.pop();
    return res;
    } O(n + k log(n))
}
```

What are the minimum and maximum number of elements in a heap of height h ? Prove that these two values are correct by induction. **Note:** the height of a heap is the number of edges on the longest root-to-leaf path.

Numa heap de altura h , o número mínimo de elementos é 2^h e o máximo é $2^{h+1} - 1$

ALTURA	0	1	2	3	4
MÍNIMO	1	2	4	8	16
MÁXIMO	1	3	7	15	...



Prova: o nível de altura h tem no mínimo 1 elemento e no máximo 2^h elementos

A soma de todos os elementos de todos os níveis totalmente preenchidos de uma heap de altura h é:

$$S = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{h-1} = \sum_{i=1}^{h-1} 2^i \text{ que é a soma da progressão geométrica de razão } 2 \text{ e termo inicial } 1:$$

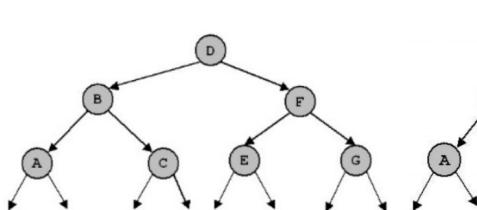
$$S = m \times \frac{2^m - 1}{2 - 1} = 1 \times \frac{2^h - 1}{2 - 1} = 2^h - 1$$

Como $h = n - 1 \Leftrightarrow m = h + 1$: $S = 2^{h+1} - 1$, sendo assim o número máximo de elementos de uma heap de altura h é o mínimo é o número máximo de elementos de uma heap de altura $h+1$ menos 1, ou seja: $2^{h+1} - 1 + 1 = 2^{h+1}$

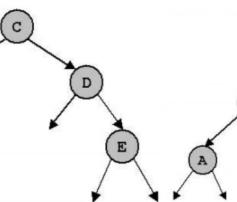
In a min-heap, where might the largest element be, assuming that all elements are distinct?

O maior elemento de uma min-heap pode estar em qualquer nó do último nível (nível com maior altura)

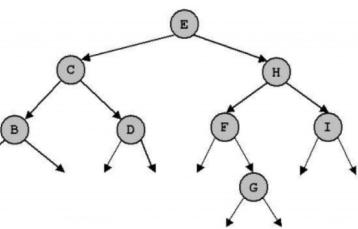
Give the preorder, inorder, postorder and level order traversals of the following binary trees.



(a)



(b)



(c)

(a) preorder: D, B, A, C, F, E, G
inorder: A, B, C, D, E, F, G
postorder: A, C, B, E, G, F, D

(b) preorder: C, B, A, D, E
inorder: A, B, C, D, E
postorder: A, B, E, D, C

(c) preorder: E, C, B, A, D, H, F, G, I
inorder: A, B, C, D, E, F, G, H, I
postorder: A, B, D, C, G, F, I, H, E

Brute-Force Algorithms

- Algoritmo simples e direto, baseado nas definições dos conceitos envolvidos no problema, que usa o poder de computação e não inteligência

1. $a^n \bmod m$

2. Selection Sort

3. Sequential Search

Traveling Salesman Problem: encontrar o caminho mais curto entre um dado conjunto de N cidades que visita cada cidade exatamente uma vez antes de volta à cidade em que começou. Encontra um ciclo que passa por todos os vértices exatamente uma vez (círculo hamiltoniano)

Solução: gerar todas as combinações possíveis gerando todas as permutações de $(n-1)$ cidades intermediárias e computar o comprimento do percurso; encontrar o mais curto entre eles.

Complexidade: $\mathcal{O}(n!)$

Knapsack Problem: dados N itens de pesos w_1, \dots, w_n e valores v_1, \dots, v_m e um raco de capacidade W , encontrar o subconjunto de itens mais valioso que cabe no raco

Solução: gerar todos os subconjuntos possíveis de n itens, computar o peso total de cada subconjunto para identificar subconjuntos fávveis e encontrar o subconjunto de maior valor.

Complexidade: $\Omega(2^n)$

Estratégia:

1. Gerar todos os elementos do domínio do problema
2. Selecionar os favoráveis (os que satisfazem restrições)
3. Encontrar o desejado (o que otimiza a função objetivo)

Consider the function *sum3* below.

```
bool sum3(unsigned int T, unsigned int selected[3])
```

The function finds three positive integers whose sum is equal to the value of the argument T. The function returns **True** and initializes the *selected* array with the three integers that add up to T. Otherwise, the function returns **False** (and the array named *selected* is not initialized).

Input example: T = 10

Expected result: selected = {1, 1, 8}, ..., selected = {2, 3, 5}, ...

- Implement *sum3* using an exhaustive search strategy (i.e., Brute-force) with $O(T^3)$ temporal asymptotic complexity.
- Improve the temporal efficiency of *sum3* by implementing a more efficient, but still brute-force approach with a lower temporal complexity.
- Indicate and justify the temporal complexity of the improved algorithm, in the worst case, with respect to T.

Handwritten pseudocode for *sum3*:

```
bool sum3(unsigned int T, unsigned int nums[3]) {
    for (unsigned int i = 1; i < T; i++)
        for (unsigned int j = 1; j < T; j++)
            if (i + j + 1 == T) {
                nums[0] = i;
                nums[1] = j;
                nums[2] = 1;
                return true;
            }
    return false;
}
```

Temporal complexity analysis:

$$O(T^2) \left(O(T) \left(O(1) \left(O(1) \left(O(1) \right) \right) \right) \right)$$
$$O(1) \rightarrow O(1)$$
$$O(T^2) + O(1) = O(T^2)$$

The **subset sum problem** consists of determining if there is a subset of a given array of non-negative numbers A whose sum is equal to a given integer T.

Note: To simplify this exercise, and should multiple subsets be valid, returning any of them will be acceptable. Any order for the valid elements of a subset is also acceptable.

Input example: A = [3, 34, 4, 12, 5, 2], n=6, T = 9

Expected result: [4, 5]

Input example: A = [3, 34, 4, 12, 5, 2], n=6, T = 20

Expected result: [3, 12, 5]

Input example: A = [3, 34, 4, 12, 5, 2], n=6, T = 30

Expected result: no solution

- Propose in pseudo-code a Brute-force algorithm solution for this problem. Your algorithm should return two outputs: a boolean indicating if the subset exists and, if so, the subset itself.
- Indicate and justify the algorithm's temporal and spatial complexity, in the worst case, with respect to the array's size, n.

bool subsetSumBF(A[], n, T, subset, &subsetYige) {

 O(1) vector<vector<int>> subsets;

 O(n) nige = $2^n - 1$;

 for (i=1; i < nige; i++) {

 O(1) mask = i;

 O(log mask) while (mask > 0) {

 O(1) bit = mask % 2; mask /= 2;

 O(1) if (bit) 1. push_back(A[1*pos*]);

 O(1) pos++;

 O(1) subsets.push_back(r); }

 O(n 2ⁿ) for (const r : subsets) {

 O(n) if (r.sum == T) {

 O(1) subsetYige = r.nige();

 O(1) for (j=0; j < r.nige(); j++) A[j] = r[j];

 O(1) return true; }

 O(1) return false; }

O(n 2ⁿ)

Given any one-dimensional array $A[1..n]$ of integers, the **maximum sum subarray problem** tries to find a contiguous subarray of A , starting with element i and ending with element j , with the largest sum: $\max \sum_{x=i}^j A[x]$ with $1 \leq i \leq j \leq n$. Consider the `maxSubsequenceBF` function below.

```
int maxSubsequenceBF(int A[], unsigned int n, int &i, int &j)
```

The function returns the sum of the maximum subarray, for which i and j are the indices of the first and last elements of this subsequence (respectively), starting at 0. The function uses an exhaustive search strategy (*i.e.*, Brute-force) so as to find a subarray of A with the largest sum, and updates the arguments i and j , accordingly.

- Implement `maxSubsequence` using a brute-force strategy.
- Indicate and justify the temporal complexity of the algorithm, with respect to the array's size, n .

int max Subsequence (A[], i, &i, &j) {

num = INT - MIN;

for (a=0; a < n; a++) {

D(1) temp = 0;

for (b=a; b < n; b++) {

D(1) temp += A[b];

D(1) if (temp > num) {

D(1) num = temp;

D(1) i = a;

D(1) j = b;

}

}

D(1) return num;

}

$$O(n^2) + O(1) = O(n^2)$$

The **change-making problem** is the problem of representing a target amount of money, T , with the fewest number of coins possible from a given set of coins, C , with n possible denominations (monetary value). Consider the function *changeMakingBF* below, which considers a limited stock of coins of each denomination C_i in Stock, respectively.

```
bool changeMakingBF(unsigned int C[], unsigned int Stock[],  

unsigned int n, unsigned int T, unsigned int usedCoins[])
```

The arguments C and $Stock$ are unidimensional arrays of size n , and T is the target amount for the change. The function returns a boolean indicating whether or not the problem has a solution. If so, it set the *usedCoins* array with the total number of coins used for each denomination C_i .

Input example: $C = [1, 2, 5, 10]$, $Stock = [3, 5, 2, 1]$, $n = 4$, $T = 8$

Expected result: $[1, 1, 1, 0]$

Input example: $C = [1, 2, 5, 10]$, $Stock = [1, 2, 4, 2]$, $n = 4$, $T = 38$

Expected result: $[1, 1, 3, 2]$

- a) Implement *changeMakingBF* using a brute force strategy.
- b) Indicate and justify the temporal complexity of the algorithm, in the worst case, with respect to the number of coin denominations, n , and the maximum stock of any of the coins, S .

bool changeMakingBF (...)

$O(1)$ coins;

$O(n^2)$ $\left\{ \begin{array}{l} \text{for } (i=0; i < n; i++) \\ \quad \left\{ \begin{array}{l} O(n) \text{ for } (j=0; j < Stock[i]; j++) \\ \quad \quad \dots \\ \quad \quad O(1) \text{ coins.push}(C[i:j]); \end{array} \right. \end{array} \right.$

$O(2^n)$ subsets = generate-all-subsets (coins);

$O(1)$ min = INT-MAX; fewCoins;

$\} \text{for } (\text{subset} : \text{subsets})$

$O(n)$ $\} \text{if } (\text{subset.sum} = T \text{ & & subset.size} \leq \text{min})$

$O(1)$ $\quad \quad \quad \{ \text{fewCoins} = \text{subset}; \text{min} = \text{subset.size}; \}$

$O(1)$ $\text{if } (!\text{fewCoins.empty}()) \text{ O(n)} \text{ usedCoins}[i] = \text{count}(\text{fewCoins}, C[i]);$

$O(1)$ return true;

$O(1)$ return false;

$O(n2^n)$

```

mystery_func(S):
    d ← +∞      O(1)
    p1 ← NULL    O(1)
    p2 ← NULL    O(1)
    foreach (x in S)
        foreach (y in S)
            if(x ≠ y && euclidean_dist(x,y) < d)   O(n)
                d ← euclidean_dist(x,y)    O(1)
                p1 ← x      O(1)
                p2 ← y      O(1)
    return d, p1, p2  O(1)

```

- a) What does this algorithm do?
- b) Explain why the algorithm follows a brute-force approach.
- c) Derive and justify the temporal complexity of the algorithm, with respect to the array S's size, n.

a) Computes the closest two points and its distance

b) It tries all possible combination of two (different) points

c) $O(n^2)$

The **0-1 Knapsack problem** consists in selecting a subset of items from a set so that the total value is maximized without exceeding the Knapsack's maximum weight capacity. Each item has a value and a weight. Each item can only be placed in the Knapsack at most once.

Consider the *knapsackBF* function below, which solves the 0-1 knapsack problem.

```
unsigned int knapsackBF(unsigned int values[], unsigned int weights[],  
                        unsigned int n, unsigned int maxWeight, bool usedItems[])
```

Input example: values = [10, 7, 11, 15], weights = [1, 2, 1, 3], n = 4, maxWeight = 5

Expected result: [1, 0, 1, 1] (the total value is $10 + 11 + 15 = 36$)

Input example: values = [1, 2, 5, 9, 4], weights = [2, 3, 3, 4, 6], n = 5, maxWeight = 10

Expected result: [0, 1, 1, 1, 0] (the total value is $2 + 5 + 9 = 16$)

- Implement *knapsackBF*, which using a brute-force strategy.
- Derive and justify the temporal complexity of the algorithm, with respect to the number of items, n.

Knapsack BF }

$\mathcal{O}(2^n)$ subsets = generate_all_possible_subsets(values);

$\mathcal{O}(1)$ maxSet; maxValue;

for (subset: subsets)

$\mathcal{O}(n)$ if (subset.value > maxValue & subset.weight < maxWeight)

$\mathcal{O}(1)$ { maxSet = subset; maxValue = subsetValue; }

$\mathcal{O}(n)$ for (el: maxSet)

$\mathcal{O}(1)$ usedItems[i] = el; }

$\mathcal{O}(1)$ return maxValue; }

$\mathcal{O}(n \cdot 2^n)$

The **Traveling Salesman Problem (TSP)** consists in finding the path in a weighted graph that traverses each vertex once and exactly once and then returns to the initial node such that the sum of the weights of the edges used is minimized.

To simplify, consider that the initial city is always node 0 and that the graph is complete (*i.e.*, it has one edge connecting every node to every other node). The distances may not be symmetric, that is, the distance from node A to B may be different from the distance from B to A.

Consider the *tspBF* function below, which solves the TSP problem.

```
unsigned int tspBF(const unsigned int **dists, unsigned int n, unsigned int path[])
```

Input example: $\text{dists} = [[0, 10, 15, 20], [5, 0, 9, 10], [6, 13, 0, 1], [8, 8, 9, 0]]$, $n=4$

Expected result: 35, $\text{path} = [0, 1, 3, 2]$

- Implement *tspBF*, using a brute-force strategy.
- Derive and justify the temporal complexity of the algorithm, with respect to the number of vertices, n .

tspBF | $O(n)$
 $O(1)$ $\text{temp}[n]; \quad \text{for } (i=0; i < n; i++) \quad \text{temp}[i] = i;$
 $O(n)$ $\min[n];$
 $O(1)$ $\min = +\infty; \quad O(n)$
do | $\text{cost} = 0; \quad \text{actual} = 0;$
| $\text{for } (i=0; i < n; i++) \quad \{$
| | $O(1) \text{ next} = \text{temp}[(i+1) \% n];$
| | $O(1) \text{ cost} += \text{dists}[\text{actual}][\text{next}];$
| | $O(1) \text{ actual} = \text{next}; \}$
 $O(1) \text{ if } (\text{cost} < \min \text{ \&\& } \text{cost} != 0) \quad \{$
| | $O(1) \text{ cost} = \min;$
| | $O(1) \min = \text{temp}[i];$
| | while ($\text{next_permutation}(\text{temp}+1, \text{temp}+n));$
 $O(1) \text{ path} = \text{temp}; \quad \text{return min;}$ | $O(1)$
 $O(n \cdot n!)$

Greedy Algorithms

Estratégia: a cada passo do algoritmo, escolher a opção que é localmente a melhor para encontrar a solução ótima global

Seleção de Atividades: dado S um conjunto de atividades que partilham um recurso comum que só pode ser utilizado por uma atividade de cada vez e caracterizadas por hora de início (s_i) e de fim (f_i), encontrar o maior conjunto de atividades que são mutuamente compatíveis

Solução: assumindo que as atividades estão ordenadas por ordem de fim crescente ($f_1 \ll \dots \ll f_n$), escolher a atividade com menor tempo, de modo a maximizar o tempo para as atividades restantes

Complexidade: $O(n \log n) + O(n) = O(n \log n)$

Knapsack Problem: variação do problema original em que é possível transportar uma fração x_i de um objeto, $0 \leq x_i \leq 1$:
maximizar $\sum_{i=1}^n v_i x_i$ tal que $\sum_{i=1}^n w_i x_i \leq W$

Solução: selecionar os objetos por ordem crescente de v_i/w_i :

Complexidade: $O(n \log n)$

Minimizar Tempo de Processamento de Sistema: dado um servidor com N clientes, cada um com um tempo de serviço conhecido (o cliente i demora t_i tempo), minimizar o tempo total ocupado no sistema por todos os clientes

Solução: processar clients por ordem crescente de tempo de serviço, de modo a diminuir tempo agregado para todos

Códigos Huffman: construir a árvore que corresponde ao código "prefix-free" ótimo árvore
Solução: começar com C folhas e realizar $C-1$ operações de merge para obter a

Propriedade da Escolha Gananciosa: uma solução global ótima pode ser alcançada fazendo uma escolha local ótima (gananciosa)

Propriedade da Sub-Estrutura Ótima: uma solução ótima para o problema contém em si soluções ótimas para subproblemas

Consider the description for the **change-making problem** in exercise 3 of the TP2 class sheet.

```
bool changeMakingGR(unsigned int C[], unsigned int Stock[],  
                    unsigned int n, unsigned int T, unsigned int usedCoins[])
```

- Implement *changeMakingGR* using a greedy strategy.
- Does the algorithm always return the optimal solution (i.e. minimum amount of coins) for any set of coin values (i.e., for any coin system)? Give an example of a coin-system where the greedy algorithm does not give the optimal solution.

a) *changeMaking GR* {

```
    usedCoins = {0, 0, 0, ...}
```

```
    index = n - 1; total = 0
```

```
    while (total < T and index >= 0) {
```

```
        if (total + C[index] <= T and Stock[index] > 0) {
```

```
            usedCoins[index]++; Stock[index]--; total += C[index]; }
```

```
        else
```

```
            index--
```

```
}
```

```
    return total == T
```

b) $C = \{2, 3\}$ $Stock = \{2, 1\}$ $n = 2$ $T = 4$

greedy UsedCoin \rightarrow false

real UsedCoin $\rightarrow [2; 0]$

Consider the description for the **change-making problem** in exercise 3 of the TP2 class sheet.

```
bool changeMakingGR(unsigned int C[], unsigned int Stock[],  
                    unsigned int n, unsigned int T, unsigned int usedCoins[])
```

- Implement *changeMakingGR* using a greedy strategy.
- Does the algorithm always return the optimal solution (i.e. minimum amount of coins) for any set of coin values (i.e., for any coin system)? Give an example of a coin-system where the greedy algorithm does not give the optimal solution.

a)

```
bool changeMakingGR(int C[], int Stock[], int n, int T, int usedCoins[])
for (int i = 0; i < n; i++) O(n)
    usedCoins[i] = 0;
for (int i = n - 1; i >= 0; i--) } O(n)
    while (T >= C[i] && usedCoins[i] < Stock[i]) {
        usedCoins[i]++;
        T -= C[i];
    }
}
return T == 0; ↗ { int change = 0;
for (int i = 0; i < n; i++)
    change += usedCoins[i] * C[i];
return change == T;
```

The **activity selection problem** is concerned with the selection of non-conflicting activities to perform within a given time frame, given a set A of activities (a_i), each marked by a start time (s_i) and finish time (f_i). The problem is to select the maximum number of activities that can be performed by a single person or machine, assuming a given priority and that a person can only work on a single activity at a time. Consider the function *activitySelection* below (which uses the **Activity** class):

```
vector<Activity> activitySelectionGR(vector<Activity> A)
```

Input example: $A = \{a_1(10, 20), a_2(30, 35), a_3(5, 15), a_4(10, 40), a_5(40, 50)\}$

Expected result: $\{a_3, a_2, a_5\}$

- Formalize this problem.
- Implement *activitySelectionGR* using a greedy strategy, in which priority is given to activities with the earliest finish time.
- Indicate and justify the algorithm's time complexity, with respect to the number of activities, n .
- Prove that the greedy strategy leads to the optimal solution.

a) Parâmetros / Inputs: $a_i = (s_i, f_i), f_i > s_i \geq 0, \forall i \in \{1, \dots, N\}$

Variáveis de Decisão / Output: $selected_i, \forall i \in \{1, \dots, N\}$

Função Objetivo: $\max selected_size()$

Restrições: $\forall i, j \in \{1, \dots, N\} : (selected_i \wedge selected_j \wedge i < j) \rightarrow f_i \leq f_j$

b) 1. Ordena A por ordem crescente de tempo de fim $O(n \log n)$

2. $selected = \{A_0\}; j=1;$
 3. $\text{for } (i=2; i < A.size(); i++)$
 if $(s_i \geq f_j)$
 select(A_i); $j=i;$

$O(1)$

$O(n)$

c) $O(n \log n)$, devido à ordenação

d) 1. Existe uma solução ótima começando com a escolha greedy - atividade 1-, dado que esta maximiza o tempo disponível para as outras atividades, pois tem o menor tempo de fim

2. Depois da primeira escolha, o problema resume-se a encontrar uma solução de atividades compatíveis com a atividade 1, o que, por (1), se consegue com a escolha greedy

Consider a machine on a factory line that needs to have its tasks scheduled in order to minimize their average completion time. The machine can only process one task at a time and each task has a predefined quantity of time needed for execution. Consider the function *minimumAverageCompletionTime* below, which returns the minimum average task completion time and computes the optimal task ordering on the second argument (*orderedTasks*). The tasks are identified by their execution time, t_i .

```
double minimumAverageCompletionTime(vector<unsigned int> tasks,  
                                     vector<unsigned int> &orderedTasks)
```

Input example: **tasks** = [3, 7, 4]

Expected result: [3, 4, 7]

In the input example above, the machine has three tasks to carry out which take exactly 3, 7 and 4 units of time, respectively. The optimal scheduling is [3, 4, 7] (the task with a time of 3, followed by the one with 4 and ending with the one with 7) since the average completion time is $((3) + (3 + 4) + (3 + 4 + 7)) / 3 = 24 / 3 = 8$. Any other scheduling is sub-optimal. For example, [3, 7, 4] would give an average completion time of $((3) + (3 + 7) + (3 + 7 + 4)) / 3 = 27 / 3 = 9$.

- Formalize this problem.
- Implement *minimumAverageCompletionTime* using a greedy strategy.
- Indicate and justify the algorithm's time complexity, with respect to the number of tasks, n .
- Prove that the greedy strategy leads to the optimal solution.

a) Parâmetros/Inputs: $task_i = t_i, \forall i \in \{1, \dots, N\}$

Variáveis de Decisão/Outputs: ordered Tasks $i, \forall i \in \{1, \dots, N\}$

Função Objetivo: $\min \sum_{i=1}^n t_i \times (n - i + 1)$

Restrições: $[task_1] = [ordered\ Tasks]$

b) 1. $ordered\ Tasks \leftarrow tasks\ ordenados\ por\ t_i\ crescente$

$O(n \log n)$

2. $num = 0; n = |ordered\ Tasks|$

$O(1)$

3. $\text{for each } (task : ordered\ Tasks) \quad num += n-- \times task$

$O(n)$

4. $\text{return } num / |ordered\ Tasks|$

$O(1)$

c) $O(n \log n)$, devido à ordenação

d) A forma de minimizar o tempo total (e consequentemente a média) é realizar primeiro as tarefas mais rápidas, de modo a minimizar o número de tarefas à espera de tarefas realizadas, minimizando o tempo de espera total com a escolha greedy

Consider the **fractional knapsack problem**. This is a variant of the 0-1 knapsack problem (presented in the TP2 sheet) where only a percentage of an item can be placed in the knapsack. For instance, if an item has a value of 4 and a weight of 3 and only 50% of the item is used, then it adds a value of 2 and a weight of 1.5 to the knapsack. Consider the function *fractionalKnapsackGR* below.

```
double fractionalKnapsackGR(unsigned int values[], unsigned int
weights[], unsigned int n, unsigned int maxWeight, double usedItems[])
```

Input example: **values** = [60, 100, 120], **weights** = [10, 20, 30], **n** = 3, **maxWeight** = 50

Expected result: [1, 1, $\frac{2}{3}$] (the total value is $60*1 + 100*1 + 120*\frac{2}{3} = 240$)

- Formalize this problem.
- Implement *fractionalKnapsackGR* using a greedy strategy.
- Indicate and justify the algorithm's time complexity, in the worst case, with respect to the number of items, n.
- Prove that the greedy strategy leads to the optimal solution.
- Give an example of an integer 0-1 knapsack problem where using an adapted version of the greedy algorithm does not give the optimal solution.

a)

Parâmetros / Inputs:

- $W \geq 0$
- $v_i \geq 0, \forall i, i \in \{1, \dots, N\}$
- $w_i \geq 0, \forall i, i \in \{1, \dots, N\}$

Váriáveis de Decisão / Output: $x_i \in [0, 1], \forall i, i \in \{1, \dots, N\}$

Função Objetivo: max $V = \sum_{i=1}^n x_i v_i$

Restrições: $\sum_{i=1}^n x_i w_i \leq W \quad \wedge \quad x_i \in [0, 1]$

b) 1. Ordenar os itens de forma decrescente de $v_i/w_i \rightarrow$ sorted Items; $V=0$

2. forach ($i = 1$ to $\#$ items)

if ($w_i \leq W$)

$$w_i = 1$$

else

$$x_i = w/w_i$$

$$W -= w_i \cdot w_i$$

$$V += w_i \cdot v_i$$

c) A ordenação tem complexidade $O(n \log n)$, enquanto o ciclo tem complexidade linear, pelo que a complexidade do algoritmo é $O(n \log n)$.

d) Suponha os objetos ordenados por razão valor/peso decrescente ($v_1/w_1 > \dots > v_n/w_n$) e temos $X = (x_1, \dots, x_n)$ a solução computada pelo algoritmo "greedy"

• Se $w_i = 1, \forall i \in \{1, \dots, n\}$, a solução é claramente ótima

• Caso contrário, seja j o menor índice tal que $w_j < 1$ ($w_i = 1, \forall j < i \wedge w_i = 0, i > j$):

• A solução de pesos é $\sum_{i=1}^n x_i \cdot w_i = V$ e o valor da solução é $\sum_{i=1}^n x_i \cdot v_i = V(X)$

• Seja $Y = (y_1, \dots, y_n)$ outra solução possível:

• A solução de pesos é $\sum_{i=1}^n y_i \cdot w_i \leq V$ e o valor da solução é $\sum_{i=1}^n y_i \cdot v_i = V(Y)$

• Comparando X e Y quanto aos pesos: $\sum_{i=1}^n (x_i - y_i) \cdot w_i \geq 0$

• Notando que: $V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) \cdot v_i = \sum_{i=1}^n (x_i - y_i) \cdot w_i \cdot (v_i/w_i)$

• Em qualquer caso, $(x_i - y_i) \cdot (v_i/w_i) \geq (x_j - y_j) \cdot (v_j/w_j)$

• Então, como todos os itens antes de j têm razões v/w maiores:

$$V(X) - V(Y) = \sum_{i=1}^n (x_i - y_i) \cdot w_i \cdot (v_i/w_i) \geq x_j/w_j \cdot \sum_{i=1}^n (x_i - y_i) \cdot w_i \geq 0$$

• Pelo que $V(X)$ é a melhor solução possível

e) values = {4, 4, 6} weights = {1, 1, 1}

Knapsack GR. used Items = {0, 0, 1} $(V=6)$
 $(W=6)$

maxWeight = 8

last used Items = {1, 1, 0} $(V=8)$
 $W=8$

Conjuntos Disjuntos

- Aperte para conjuntos dinâmicos de elementos disjuntos

Representação: cada conjunto é representado por um elemento armazenado e inalterável do conjunto

Operações:

1. Make Set (v): cria um novo conjunto representado pelo elemento v
2. União (v, y): une os conjuntos que têm os elementos v e y , $S_v \cup S_y$, tendo o novo conjunto $S_v \cup S_y$ representado por v ou y e eliminando $S_v \cup S_y$
3. Find Set (v): retorna o conjunto que contém o elemento v

Connected Components (Graph G):

```
for  $v \in V[G]$  do
    MakeSet( $v$ )
for  $(u, v) \in E[G]$  do
    if FindSet( $u$ )  $\neq$  FindSet( $v$ ) then
        Union( $u, v$ )
```

Same Component (u, v):

```
return FindSet( $u$ ) = FindSet( $v$ )
```

Implementação em Listas Singulares: os elementos de cada conjunto estão numa lista simplesmente ligada, sendo o primeiro elemento de cada lista o elemento representativo e tendo cada nó da lista um apontador para a "cabeca" da lista

Complexidade: $O(m^2)$ para uma sequência de m operações

União Pesada: acrescenta a lista com menor número de elementos à lista com maior número de elementos (o peso de cada conjunto é o número de elementos)

Complexidade: $O(m + n \log m)$ para uma sequência de m operações (com n unões)

Implementação em Árvore: cada conjunto é representado por uma árvore e cada elemento aponta para o seu pai na árvore, sendo a raiz o elemento representativo e pai dele próprio.

Complexidade: $O(mn)$ para uma sequência de n operações $O(m)$

União por Rank: a árvore com menor número de elementos (estimada a partir da altura) aponta para a árvore com maior número de elementos

Compressão de Caminho: em cada operação de Find Set, fazer cada nó visitado no caminho até à raiz apontar diretamente para a raiz

Make Set(w):

$$\begin{aligned} \pi[w] &= w; \\ \text{rank}[w] &= 0; \end{aligned}$$

União (w, y):

Link ($\text{Find Set}(w), \text{Find Set}(y)$);

Find Set(w):

if $w \neq \pi[w]$ then
 $\pi[w] = \text{Find Set}(\pi[w]);$ Compressão do Caminho
return $\pi[w];$

Link (w, y):

if $\text{rank}[w] > \text{rank}[y]$ then $\pi[y] = w;$
else $\pi[w] = y;$
 if $\text{rank}[w] = \text{rank}[y]$ then
 $\text{rank}[y] = \text{rank}[y] + 1;$

União por Rank

Minimum Cost Spanning Trees (MSTs)

Grafo Convexo: grafo não dirigido $G = (V, E)$ em que para qualquer par de vértices/nós existe pelo menos um caminho a ligar os dois vértices

Spanning Tree: subconjunto acíclico das arestas $T \subseteq E$ de um grafo convexo e não dirigido que ligam todos os vértices/nós em G

Custo: soma de todos os custos das arestas de uma Spanning tree

MST: dado um grafo $G = (V, E)$ convexo, não dirigido e com uma função de peso $w: E \rightarrow \mathbb{R}$, Spanning tree cuja soma dos pesos das arestas é mínimo

$$\min W(T) = \sum_{(u,v) \in T} w(u, v)$$

function MST-Brute Force (Graph G , function w)

$S = \text{generate Spanning Trees } (G);$

$\text{for each } s \in S \text{ do}$

$s_c = \text{Cost}(s, w)$

$\text{select } s \in S \text{ with min cost;}$

$\text{return } s;$

function MST-Greedy (Graph G , function ω)

$$A = \emptyset$$

while A is not a spanning tree do
 identify safe edge (u, v) for A ;
 $A = A \cup \{(u, v)\}$

return A ;

- Um corte $(S, V-S)$ de um grafo não dirigido $G=(V, E)$ é uma partição de V em conjuntos disjuntos de nós
- Uma aresta $(u, v) \in E$ cria um corte $(S, V-S)$ se um dos seus nós está em S e o outro nó em $V-S$
- Um corte contém um conjunto de arestas A se nenhuma aresta $\in A$ cria o corte - todas as arestas ligam nós em ou S ou $(V-S)$
- Uma aresta que cria um corte com o menor custo é a aresta mais leve
- Uma aresta é segura para A se a sua inclusão em A não cria ciclos em A

Teorema: Sendo $G=(V, E)$ um grafo conexo e não dirigido com função peso ω e sendo $A \subseteq E$ incluído numa MST T , para qualquer corte $(S, V-S)$ que contém A , se (u, v) é a aresta mais leve que cria $(S, V-S)$, então (u, v) é uma aresta segura para A

Propriedade do Ciclo: a aresta mais pesada de um ciclo neverá é parte de uma MST

Propriedade do Corte: Separando os vértices do grafo de qualquer forma em dois conjuntos A e B , a aresta mais leve com vértices em A e B é sempre parte de uma MST

Algoritmo de Kruskal:

1. Começa com cada nó isolado como o seu próprio "cluster"
 2. Escolher a aresta mais leve $e \in E$
 - a. se a liga dois nós em "clusters" diferentes, então e é adicionado à MST e os "clusters" unidos
 - b. não, ignora-la
 3. Continuar até que $V-1$ arestas são adicionadas
- Complexidade: $O(E \log V)$

Algoritmo de Boruvka:

1. Começa com N conjuntos de nós (N árvores)
 2. Para cada árvore T , escolher a aresta de peso mínimo incidente em cada árvore T (uma aresta pode ser selecionada por múltiplas árvores)
 3. Unir árvores ligadas pelas arestas selecionadas
 4. Termina quando há uma árvore única
- Complexidade: $O(E \log V)$

Algoritmo de Prim:

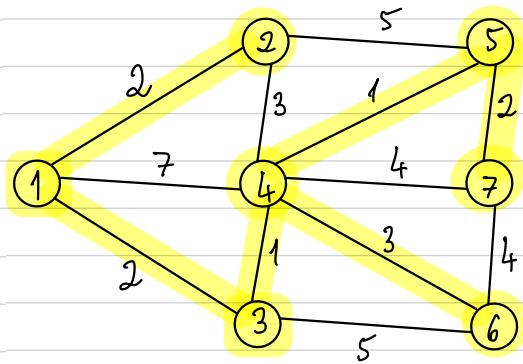
1. Começa MST a partir da raiz
 2. Expanda a árvore uma aresta de cada vez
 3. Em cada passo, escolher a aresta vizinha mais leve
- Complexidade: $O(E \log V)$

Using the **GreedyGraph** class provided in the *exercices.h* file, implement *kruskal*, which uses Kruskal's algorithm to find the minimum spanning tree.

```
std::vector<Vertex *> kruskal()
```

1. Ordenan edges de forma crescente
2. foreach ($v \in G.V$)
 create $\text{set}(v)$
3. foreach $((u, v) \in \text{sortedEdges})$
 if $(\text{set}(u) \neq \text{set}(v))$
 join $\text{set}(u, v)$
 add Edge $((u, v))$
 if $(|E| = |V| - 1)$
 finish algorithm

Ex:



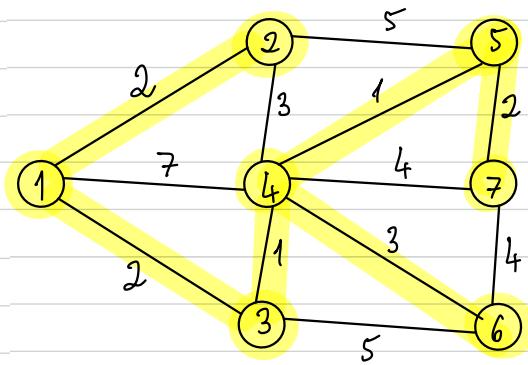
Implement `prim`, which uses Prim's algorithm to find the minimum spanning tree from the first vertex v in the graph, to all other vertices. The function returns the graph's set of vertices.

```
std::vector<Vertex *> prim()
```

MST-Prim (G, w, π)

```
Q = V[G];
foreach  $u \in Q$  do
    key[u] =  $\infty$ 
key[v] = 0;
pred[v] = NULL;
while  $Q \neq \emptyset$  do
    u = ExtractMin(Q);
    foreach  $v \in Adj[u]$  do
        if ( $v \in Q$  and  $w(u, v) < key[v]$ )
            pred[v] = u;
            key[v] = w(u, v);
```

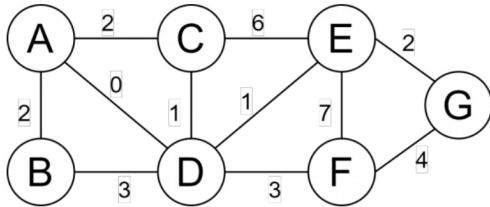
Ex:



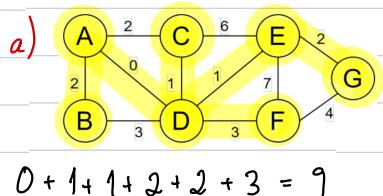
Bernard and Bianca are competing as a team in a computer network competition where each contestant must connect a series of routers between each other, so that there is a path between all pairs of routers. To achieve this, all of the teams in the competition (which are made up of 2 people) receive the same sheet with all of the connections they can create between consecutive routers and a score for each connection. One-by-one, each team member must create their own network, without knowledge about the network of their team member.

The goal is for the teams to **minimize** the sum of the score achieved by each member when creating their connected network. The teams know that there a **single** network that yields the minimum score. They also know that, according to the competition's rules, if both members create the exact same network, they receive 1.000.000 points – a guaranteed loss, taking into account the goal of minimizing the team's total score.

To avoid this scenario, Bernard agreed with his teammate Bianca that he will build the network with the lowest possible score, leaving the other options open for his teammate. On the day of the competition, they receive the following sheet:



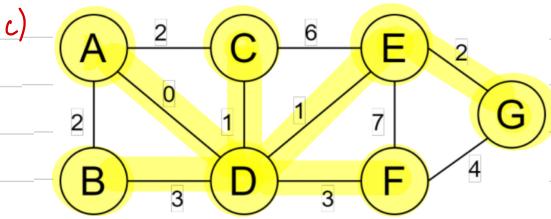
- Indicate which network Bernard created and the score he obtained. Use the algorithm you find the most convenient. Present the steps to reach the solution with the chosen algorithm.
- Propose an efficient algorithm, in pseudo-code or in C++, so that Bianca can create a network that allows the team to achieve the lowest possible score. The algorithm must work for the general case, not only on the graph above. Indicate the temporal complexity of the algorithm.
- Indicate the network and score obtained by Bianca on the specific case of the graph above with the algorithm that was proposed in the previous question.



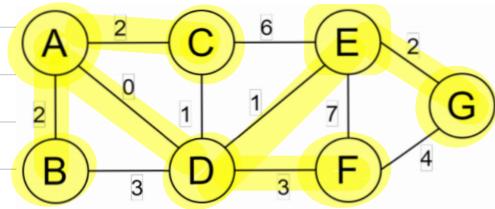
algoritmo de Kruskal: inicia cada nó como um conjunto e em cada passo escolher a aresta mais leve entre conjuntos de nós ainda não unidos

b) 1. Determine a MST (algorithm à Kruskal)
 2. $\min = \infty$; before = Null; after = Null; max = $-\infty$ $O(1)$
 foreach ($v \in G.V$)
 $t_{\min} = \infty$; before = Null; $t_{\text{after}} = Null$; $t_{\max} = -\infty$ $O(1)$
 for ($e \in v.\text{adj}$)
 if ($e \in \text{MST}$ $\&$ $e.w > t_{\max}$) $O(1)$
 $t_{\max} = e.w$; before = e $O(1)$
 for ($e \in v.\text{adj}$)
 if (! $e \in \text{MST.E}$ $\&&$ $e.w < t_{\min}$) $O(1)$
 $t_{\min} = e.w$; $t_{\text{after}} = e$ $O(1)$
 if ($\exists \text{edge } (e, \text{MST.before})$) $O(E)$
 $\min = t_{\min}$; $\max = t_{\max}$; before = before; after = t_{after} $O(1)$
 if (before $\&&$ after) $O(1)$
 MST.remove(before) $O(1)$
 MST.add(after) $O(1)$
 return true $O(1)$
 return false $O(1)$

$O(V E)$



$$0 + 1 + 1 + 2 + 3 + 3 = 10$$



$$0 + 1 + 2 + 2 + 2 + 3 = 10$$

Algoritmos de Maximização de Fluxo

Fluxo Máximo em Grafos: dado um grafo dirigido $G = (V, E)$, um nó de origem s e um nó de ralo t e temos cada aresta (u, v) uma capacidade não negativa $c(u, v)$ que indica o máximo valor de fluxo que é possível enviar de u para v através da aresta (u, v) , computa o valor máximo do fluxo que pode ser enviado da origem para o ralo, sujeito às restrições de capacidade das arestas.

Para redes com múltiplas origens e/ou ralos... deve-se aumentar o grafo para o fazer apenas uma origem e um ralo, isto é, definir uma super-fonte ligada a toda as fontes e um super-ralo ligado a todos os ralos, com capacidade infinita entre super-fonte/super-ralos e fontes/ralos

Fluxo de Fluxo: $G = (V, E)$ é um grafo dirigido em que aresta (u, v) tem uma capacidade $c(u, v) \geq 0$, tem dois nós especiais: fonte s e ralo t e todos os nós de G pertencem a um caminho Δ de s para t

Fluxo: de $G = (V, E)$ é uma função $f: V \times V \rightarrow \mathbb{R}$ tal que
(1) $f(u, v) \leq c(u, v), u, v \in V$ (restrição de capacidade)
(2) $f(u, v) = -f(v, u), u, v \in V$ (simetria)
(3) $\sum_{v \in V} f(u, v) = 0, u \in V - \{s, t\}$ (invariância/conservação do fluxo)

Valor do Fluxo: $|f| = \sum_{v \in V} f(s, v)$

Problema do Fluxo Máximo: dada a rede de fluxo G com origem s e ralo t , computa o valor do fluxo máximo de s para t

Propriedades: dadas conjuntas de nós X, Y, Z

1. $f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y)$
2. $f(X, X) = 0$ (cancelamento de termos)
3. $f(X, Y) = -f(Y, X)$ (simetria)
4. se $X \cap Y = \emptyset$, $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ (expansão da soma)

$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

Ford-Fulkerson Method (Graph G , node s , node t):

initialize flow f to 0;

while (exists an augmentation path P) do

increase flow along P ;

update residual network;

return f ;

Capacidade Residual: de (u, v) é o fluxo adicional que é possível enviar de $u \rightarrow v$

$$c_f(u, v) = c(u, v) - f(u, v)$$

Residual Graph: $G_f = (V, E_f)$, $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$

- cada aresta residual de G_f só permite fluxo positivo

Fluxo Adicional: para $(u, v) \in V$, $(f + f')(u, v) = f(u, v) + f'(u, v)$

$$|f + f'| = |f| + |f'|$$

"Augmenting Path": caminho simples de s para t na rede residual G_f

Capacidade Residual de f : $c_f(\pi) = \min \{ c_f(u, v) : (u, v) \in \pi \}$

Fluxo Líquido do Ete (S, T): $f(S, T) = \sum_{u \in S} \sum_{v \in T} f(u, v)$

Capacidade do Ete (S, T): $c(S, T) = \sum_{u \in S} \sum_{v \in T} c(u, v)$

- Qualquer valor de fluxo é limitado superiormente pela capacidade de qualquer rota

Teorema: Dado $G = (V, E)$, com fonte s , robo t e fluxo f , então as seguintes proposições não equivalentes:

1. f é o fluxo máximo de G
2. a rede residual G_f não tem "augmenting paths"
3. $|f| = c(S, T)$ para um coto (S, T) em G

Ford-Fulkerson (Graph G , node s , node t):

foreach $(u, v) \in E[G]$ do

$$f[u, v] = 0;$$

$$f[v, u] = 0;$$

while exists an augmenting path in residual network G_f do

compute $c_f(t)$;

foreach $(u, v) \in \tau$ do

$$f[u, v] = f[u, v] + c_f(t)$$

$$f[v, u] = -f[u, v]$$

Complexidade: $\mathcal{O}(E|f^*|)$

Algoritmo de Edmonds-Karp: escolher "augmenting paths" usando o caminho mais curto

1. Cada aresta tem uma distância de 1

2. Usar BFS em G_f que identifica o caminho mais curto

Complexidade: $\mathcal{O}(VE^2)$

$$f \rightarrow G_f \rightarrow \text{BFS} \rightarrow \tau \rightarrow f' \rightarrow G_f' \rightarrow \text{BFS} \rightarrow f''$$

Consider a file with the following text: "ban bad bananas". The goal is to encode this string using the minimum number of bits necessary.

- Define a fixed-length encoding for the text. How is the text represented and what is the minimum number of bits needed?
- Define a variable length-encoding for the text using Huffman encoding. How is the text represented and how many bits does it need?

a)

	b	a	n	d	^	l
f	3	5	3	1	1	2
c	000	001	010	011	100	101

$$15 \times 3 = 45 \text{ bits}$$

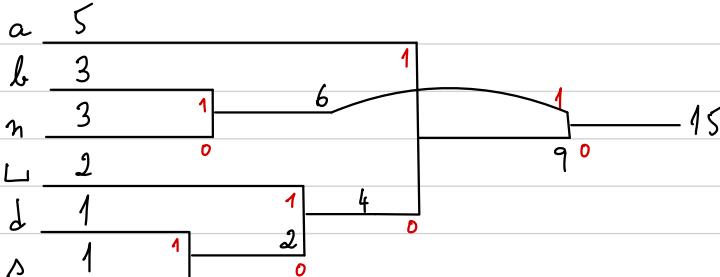
000.001.010.101.000.001.011.101.000.001.010.001.010.001.100

b)

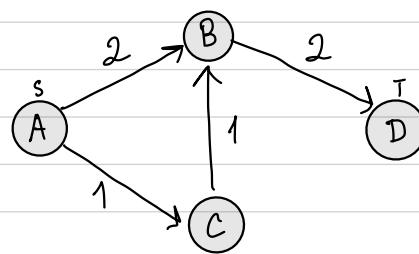
	b	a	n	d	^	l
f	3	5	3	1	1	2
c	11	01	10	0001	0000	001

11 01 10 001 11 01 0001 001 11 01 10 01 10 01 0000

36 bits



Give an example of a capacity graph to illustrate why the choice of the algorithm to find augmenting paths in the Ford-Fulkerson method is essential for achieving lower execution times when computing the maximum flow.



O fluxo máximo é 4, mas por ABD não é 3 mas, enquanto por ACBD não é 4

Using the **GreedyGraph** class provided in the *exercices.h* file, implement *edmondsKarp*, which uses the Edmonds-Karp algorithm to find the maximum flow from the source vertex source to the sink vertex target in the graph.

```
void edmondsKarp(int source, int target)
```

Ford-Fulkerson (Graph G , node s , node t)

foreach $(u, v) \in E[G]$ do

$$f[u, v] = 0;$$

$$f[v, u] = 0;$$

while exists an augmenting path τ in residual network G_f do

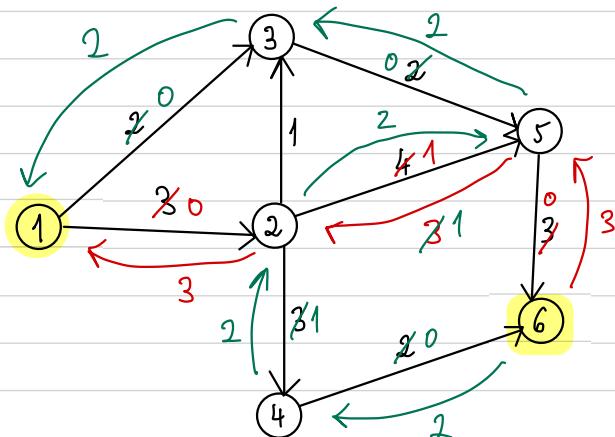
compute $c_f(\tau)$;

foreach $(u, v) \in \tau$ do

$$f[u, v] = f[u, v] + c_f(\tau);$$

$$f[v, u] = -f[u, v];$$

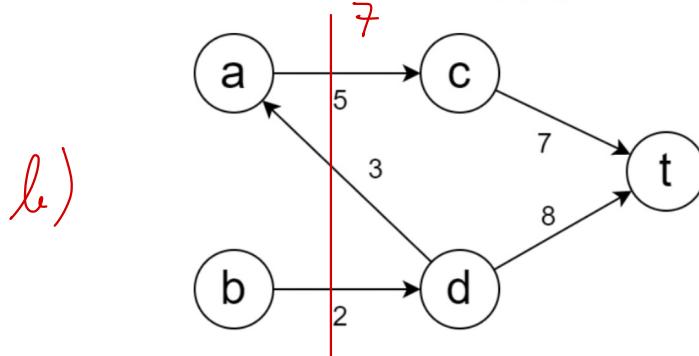
- Escolher "augmenting paths" usando o caminho mais curto
- Usar BFS em G_f para identificar o caminho mais curto



Path	Flow
1, 3, 5, 6	2
1, 2, 4, 6	2
1, 2, 5, 6	1
	5

while (find Augmentation Path BFS)
 $c_f = \min \text{ResidualCap}(\tau);$
 $f \leftarrow c_f;$
 $\text{update Residual Graph();}$

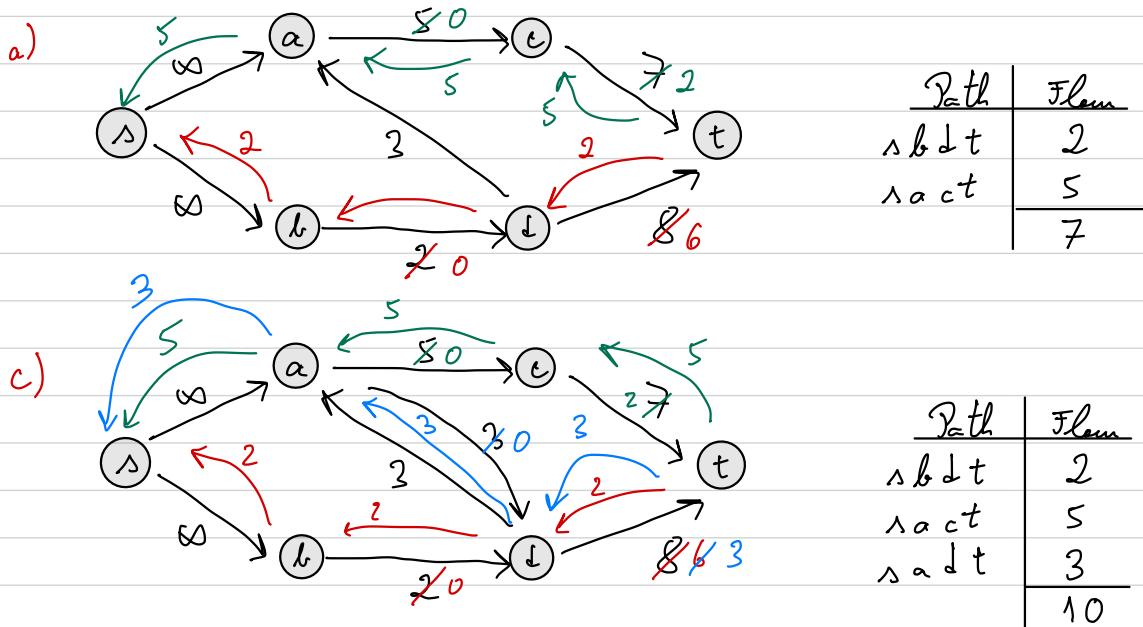
An oil extraction station is structured as a set of tanks connected by pipes. Fluids can only traverse the pipes in one direction, due to the pipes' slope. The structure of the station is depicted below, where the numbers denote how much liters can traverse each pipe per second.



b)

All the oil that is extracted from the underground is initially stored in one of two tanks, node a or node b, before moving though the structure to reach the destination tank, located in node t.

- What is the maximum flow for the oil that can traverse the system, from being mined underground to reaching the destination tank? Represent the problem using a capacity graph. Explain how you computed the maximum flow.
- In the capacity graph, determine a minimum s-t cut.
- A new pump is added to the system, allowing oil to be moved not only from tank d to a, but also in the reverse direction. The capacity of the pipe remains at 3 L/s. Does the pump increase the maximum possible amount of flow?



TEOREMA

MAX-FLOW

MIN-CUT

Maximal Bipartite Matching

"Matching": $M \subseteq E$ tal que $\forall v \in V$, no máximo uma aresta de M incide em v

"Maximal Matching": "matching" de cardinalidade de M máxima

Grafo Bipartido: grafo que pode ser dividido em $V = L \cup R$, em que L e R são disjuntos e todas as arestas em E não entram L e R

Grafo Bipartido Máximo: "maximal matching" em que G é bipartido

- Um vértice $v \in V$ é "matched" por M se alguma aresta em M incide em V , caso contrário, v é "unmatched"

Algoritmo de Fluxo Máximo: "maximal bipartite matching" em G é equivalente a encontrar o fluxo máximo em G'

1. Constrói grafo auxiliar $G' = (V', E')$

$$= (V \cup \{s, t\}, \{(s, u) : u \in L\} \cup \{(u, v) : u \in L, v \in R, (u, v) \in E\} \cup \{(v, t) : v \in R\})$$

2. Define capacidades unitárias para as arestas em E'

Teoremas:

1. Se M é "matching" em G , então existe um fluxo interno f em G' tal que $|f| = |M|$
2. Se $|f|$ é um fluxo interno em G' , então existe um "matching" M em G tal que $|M| = |f|$
3. Se todas as capacidades das arestas são interiores, então o fluxo máximo $|f|$ é interno
4. "Maximal Bipartite Matching" em G corresponde a $|f|$, tendo f o fluxo máximo em G'

Complexidade: $O(|E|f^*) = O(E \min(|L|, |R|)) = O(|EV|)$

In a dancing event, a set of 7 men and 7 women dance in pairs over several rounds (for simplification purposes, assume all dancing pairs are composed of a man and a woman). These 7 women, however, are rather selective and will only dance with men they are friends with. The tables below show each woman's friends:

Women	Male friends
Anna	Anthony, Bruno
Berta	Anthony, Charles
Claire	Anthony
Diane	Bruno, Dennis
Elizabeth	Charles, George
Faith	Dennis, Fred
Grace	Eugene, George

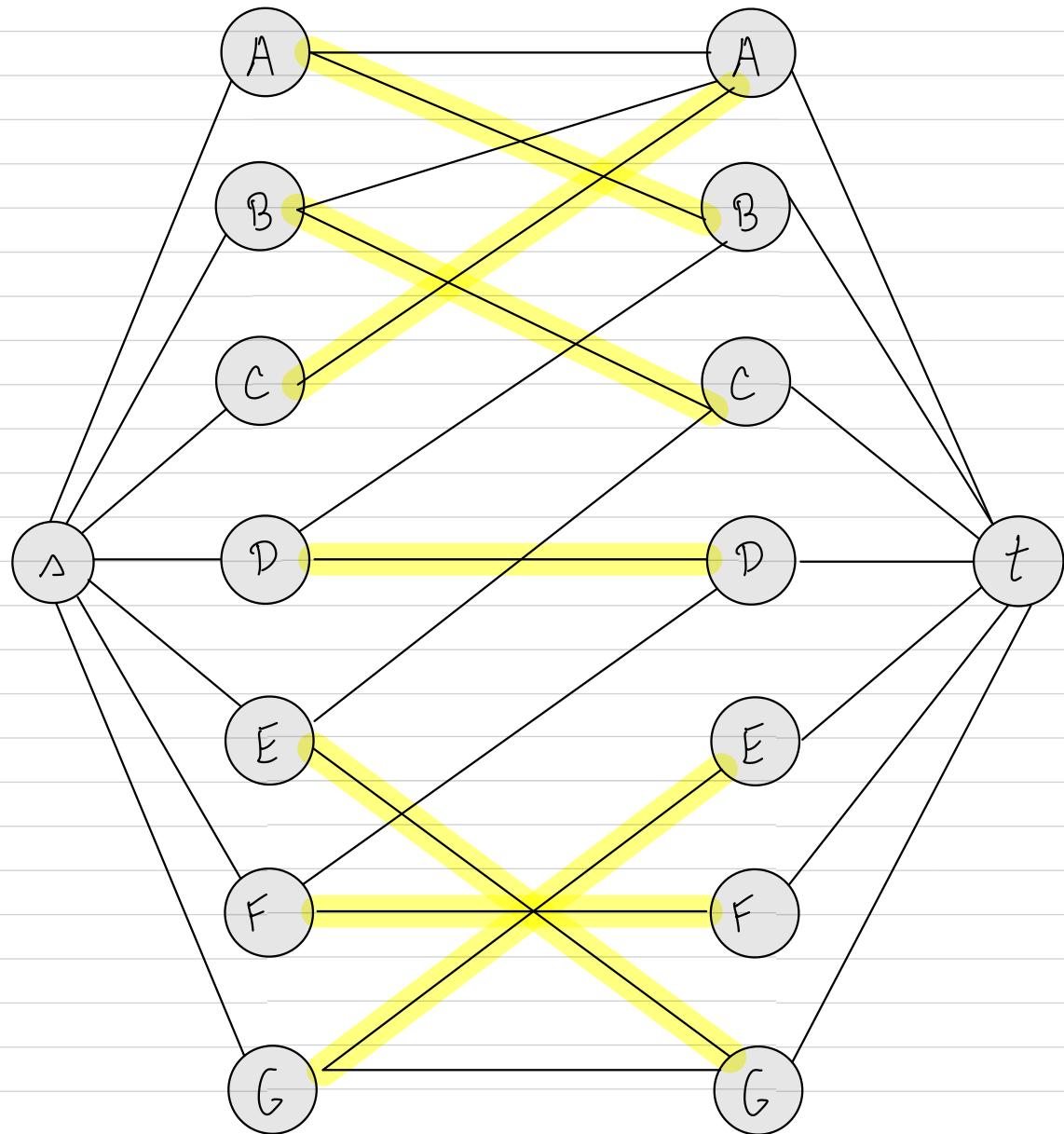
In a given round, the 5 following couples danced:

- Anna and Anthony
- Berta and Charles
- Diane and Bruno
- Faith and Dennis
- Grace and George

- a) Using these pairings as a starting point, determine a matching between the women and man that maximizes the number of dancing couples for the next round.
- b) Explain why using these initial pairings saves time over finding an optimal matching from scratch.

b) É um emparelhamento parcial que permite identificar os desemparelhados e fez os trocos relevantes para o emparelhamento.

a)



Shortest Path Problem

Defe: grafo dirigido $G = (V, E)$ com origem s , destino t e comprimento das arestas

Problema: encontrar o caminho dirigido mais curto de s para t

Single-Source Shortest Paths (SSSPs): identifica o caminho mais curto entre o nó de origem $s \in V$ e qualquer outro nó $v \in V$

All-Pairs Shortest Paths (APSPs): identifica o caminho mais curto entre cada par de nós em V

Relâncamento das arestas: tendo $\text{dist}[v]$ o comprimento de algum caminho de s para v e $\text{pred}[v]$ o antecessor de v no caminho atual de s para v , então o relâncamento segundo a aresta e de v para w é: se (v, w) dá um caminho mais curto para w através de v , então atualizar $\text{dist}[w]$ e $\text{pred}[w]$

if ($\text{dist}[w] > \text{dist}[v] + e.\text{weight}$) { $\text{dist}[w] = \text{dist}[v] + e.\text{weight}$; $\text{pred}[w] = v$;}
monotonicamente não crescente

Algoritmo de Dijkstra: mantém um conjunto S de nós explorados para os quais se determinou a distância mais curta $\delta(u)$ de s para u

1. Inicializa $S = \{s\}$, $\delta(s) = 0$; todos os outros nós têm $\delta(u) = \infty$
2. Escolher repetidamente o nó não explorado v que minimiza $\text{dist}(v) = \min(\text{dist}(u) + \text{len}(e))$, $e = (u, v) \in S$
3. Adicionar v a S e definir $\delta(v) = \text{dist}(v)$

Uma fila de prioridade para refletir o relâncamento da aresta recente

Dijkstra (Graph G , Function w , Node α)

Initialize Single Source (G, α);

$$S = \emptyset;$$

$O(V)$

$$Q = V[G];$$

// fila de prioridade Q

while $Q \neq \emptyset$ do

$O(\log V)$ $u = \text{Extract Min}(Q);$

$$S = S \cup \{u\};$$

$O(E)$ foreach $v \in \text{Adj}[u]$ do

$O(\log V)$ Relax (u, v, w)

// atualizar Q

Relax (u, v, w)

if $d[v] > d[u] + w(u, v)$ then

$$d[v] := d[u] + w(u, v)$$

$$\text{parent}[v] := u$$

Complexidade: $O((V+E) \log V)$

Invariante: para cada no $u \in S$, $d(u)$ é o comprimento do caminho $\alpha-u$ mais curto

$$l(P) \geq l(P') + w(u, y) \geq \text{dist}(u) + l(u, y) \geq \text{dist}(y) \geq \text{dist}(v)$$

$|$
não negativa

$|$
hipótese
indutiva

$|$
definição
de $\delta(y)$

$|$
algoritmo escolhe
 v em vez de y

Pesos/Ciclos Negativos:

- Pesos negativos, mas não ciclos negativos pode levar a resultados incorretos
- Pesos negativos e ciclos negativos levam a comportamento de ciclos infinitos

Bellman-Ford (Graph G , Function w , Node s)

$\Theta(V)$ Initialize Single Source (G, s);

$\Theta(VE)$ { for $i = 1$ to $|V[G]| - 1$ do
 foreach $(u, v) \in E[G]$ do
 Relax (u, v, w) ;

$\Theta(E)$ foreach $(u, v) \in E[G]$ do
 if $dist[v] > dist[u] + w(u, v)$ then
 return FALSE; // negative cycle
 return TRUE; // no negative cycle

Complexidade: $\Theta(VE)$

- Se o grafo não contém ciclos negativos, então $dist[v] = \delta(s, v)$ para todos os nós alcancáveis a partir de s quando o algoritmo termina e retorna verdadeiro.
- Se o grafo tem um ciclo negativo alcancável a partir de s , então o algoritmo retorna falso.

DAG-Shortest-Paths (Graph G , Function w , Node s)

Topological Sorting of nodes in G ;

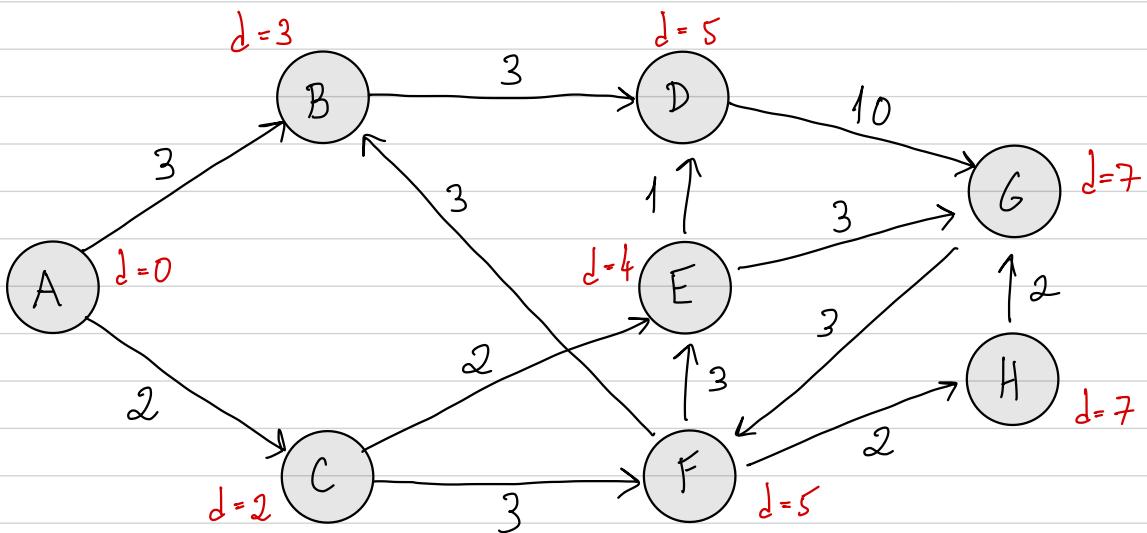
Initialize Single Source (G, s);

foreach $u \in V[G]$ in topological order do
 foreach $v \in Adj[u]$ do
 Relax (u, v, w)

Complexidade: $O(V+E)$

- Para encontrar o caminho mais curto entre todos os pares de nós, o algoritmo de Dijkstra tem complexidade $O(|V||E|)$ ou $O(|V|^2|E|)$?
 $|E| > |V|$

Ex:



NO'	$\{d(u, v), \text{pred}(v)\}$
A	(0, NULL)
B	(∞ , NULL)
C	—
D	—
E	—
F	—
G	—
H	—

Divide and Conquer

Merge Sort (A, l, r)

if $l < r$ then

$$q = \lfloor (l+r)/2 \rfloor$$

Merge Sort (A, l, q)

Merge Sort ($A, q+1, r$)

$\mathcal{O}(n)$ Merge (A, l, q, r)

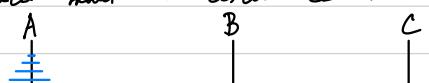
1. interior de divisão

2. divisão "para a frente"

3. junção "para trás"

Complexidade: $\mathcal{O}(n \log n)$

Torre de Hanoi: Hanoi (n, A, B, C) significa mover n discos de A para B usando C como auxiliar.



Hanoi (n, A, B, C) = Hanoi ($n-1, A, C, B$) + Hanoi ($1, A, B, C$) + Hanoi ($n-1, C, B, A$)

Complexidade: $\mathcal{O}(2^n)$

procedure Divide Conquer (l, r)

if Small (l, r) then

return G (l, r)

$m = \underline{\text{Divide}} (l, r)$

return Combine (Divide Conquer (l, m) + Divide Conquer ($m+1, r$));

Dividir: separar o problema original em k subproblemas menores e resolver cada um deles

Combinar: combinar soluções de subproblemas na solução do problema todo

Recorrência: relação que define uma sequência baseada numa regra que dá o próximo termo como uma função do (1) termo (n) anterior(es)

Teorema Master: $T(n) = \begin{cases} d, & n \leq k \\ aT\left(\frac{n}{b}\right) + f(n), & n > k \end{cases}$

$a \geq 1, b > 1, \lim_{n \rightarrow \infty} f(n) > 0, f(n) \in \Theta(n^c)$

Número de sub-problemas: a
Tamanho de cada sub-problema: $\left(\frac{n}{b}\right)$

Complexidade da divisão e combinação: $f(n)$

- a: mede quantas chamadas recursivas são ativadas por cada instância do método
- b: mede a razão de separação/divisão do input
- c: mede o termo dominante do trabalho não recursivo dentro do método recursivo
- d: mede o trabalho feito no caso base

- Se $\log_b a < c$, então $T(n) \in \Theta(n^c)$
- Se $\log_b a = c$, então $T(n) \in \Theta(n^c \log n)$
- Se $\log_b a > c$, então $T(n) \in \Theta(n^{\log_b a})$

• A árvore da recorrência $T(n) = aT(n/b) + f(n)$ tem $h = \log_b n$ e $n^{\log_b a}$ folhas

1. $f(n) = O(n^{\log_b a - \varepsilon})$, $\varepsilon > 0 \rightarrow f(n)$ cresce polinomialmente mais lentamente que $n^{\log_b a}$ por um fator de $n^\varepsilon \rightarrow T(n) = \Theta(n^{\log_b a}) \rightarrow$ o peso cresce geometricamente da raiz para as folhas, que contém uma fração constante do peso total

2. $f(n) = O(n^{\log_b a} \log^k n)$, $k > 0 \rightarrow f(n) \in n^{\log_b a}$ crescem similmente $\rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \rightarrow$ o peso é aproximadamente o mesmo em cada nível dos $\log_b n$ níveis

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$, $\varepsilon > 0 \rightarrow f(n)$ cresce polinomialmente mais rápido do que $n^{\log_b a}$ por um fator de n^ε e $f(n)$ satisfaz a condição de regularidade que $a f(n/b) \leq c f(n)$, $c < 1 \rightarrow T(n) = \Theta(f(n)) \rightarrow$ o peso diminui geometricamente da raiz para as folhas e a raiz contém uma fração constante do peso total

Teorema Master

$$T(n) = \begin{cases} f_1(n), & n \leq k \\ a T\left(\frac{n}{b}\right) + f_2(n), & n > k \end{cases}$$

$O(n^c)$

1) $\log_b a < c \rightarrow T(n) \in \Theta(n^c)$

2) $\log_b a = c \rightarrow T(n) \in \Theta(n^c \log n)$

3) $\log_b a > c \rightarrow T(n) \in \Theta(n^{\log_b a})$

Consider the same description for the **maximum sum subarray problem** as in the TP2 sheet.

a) Propose in pseudo-code a divide-and-conquer strategy for this problem.

b) Using the master theorem, indicate and justify the time complexity of the proposed algorithm.

a) $\text{max Subsequence DC}(\dots)$

$$i = 0; j = n - 1;$$

$\text{max Subsequence DC-rec}(\dots)$

$$T(n) = 2T\left(\frac{n}{2}\right) + f(n), \quad f(n) \in \Theta(n)$$

$$a = 2 \quad b = 2 \quad c = 1$$

$$\log_2 2 = 1 \rightarrow T(n) \in \Theta(n \log n)$$

$\text{max Subsequence DC-rec}(\dots)$

if $i == j$ // CASO BASE

return $A[i]$

$\text{mid} = \lfloor \frac{i+j}{2} \rfloor$ // DIVIDIR

// CONQUISTAR

$L = \text{max Subsequence DC-rec}(A, n, i, \text{mid})$

$R = \text{max Subsequence DC-rec}(A, n, \text{mid}+1, j)$

$X = \text{max Subsequence DC-auc}(A, i, \text{mid}, j)$

// COMBINAR

if $L > R \text{ and } L > X$

return L ;

if $R > L \text{ and } R > X$

return R ;

return X ;

$\text{max Subsequence DC-auc}(A, i, \text{mid}, j)$

$L_{\Delta} = -\infty; \Delta = 0;$

for $l = \text{mid} \rightarrow i$

$\Delta += A[l]$

if $\Delta > L$

$L_{\Delta} = \Delta$

$mL = l$

$i = mL; j = mR; \text{return } L + R$

$R_{\Delta} = -\infty; \Delta = 0;$

for $r = \text{mid} + 1 \rightarrow j$

$\Delta += A[r]$

if $\Delta > R$

$R_{\Delta} = \Delta$

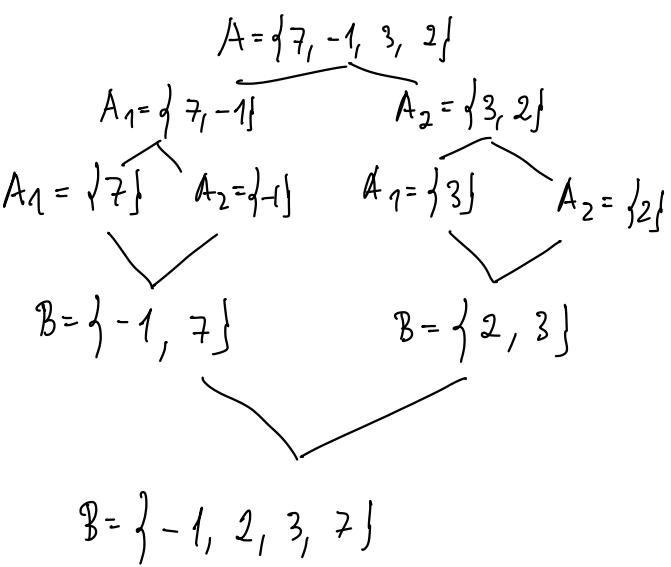
$mR = r$

Consider the following pseudo-code:

```

mysteryFunc(A)
if(A.size() ≤ 1)
    return A
A1 ← A[0 : A.size()/2]
A2 ← A[A.size()/2 : A.size()]
B1 ← mysteryFunc(A1)
B2 ← mysteryFunc(A2)
i ← 0
j ← 0
k ← 0
while(k < A.size())
    if (i ≥ B1.size())
        B[k++] ← B2[j++]
    else if (j ≥ B2.size())
        B[k++] ← B1[i++]
    else if (B1[i] < B2[j])
        B[k++] ← B1[i++]
    else
        B[k++] ← B2[j++]
return B

```



- What does this algorithm do? Identify the name of the algorithm.
- Explain why the algorithm follows a divide-and-conquer approach.
- Using the master theorem, indicate and justify the temporal complexity of the algorithm, with respect to the array's size, n.

a) o algoritmo ordena o array A — Merge Sort

b) O algoritmo segue uma estratégia de divisão e conquista porque:

- tem um caso base
- divide o array
- combina os resultados (com um critério)

c)

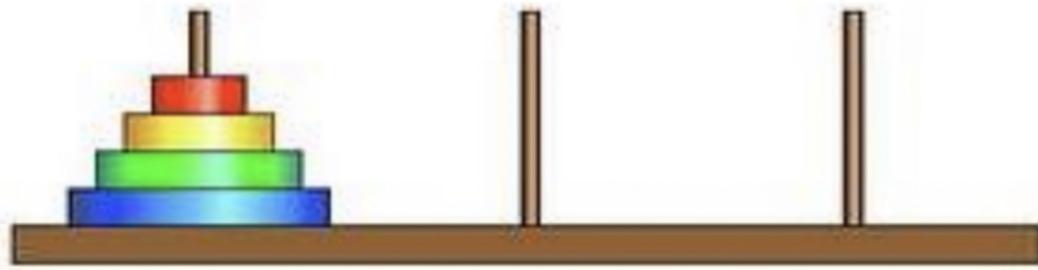
$$T(n) = \begin{cases} 1, & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + f(n), & f(n) \in \Theta(n) \end{cases}$$

$$\begin{aligned} a &= 2 \\ b &= 2 \\ c &= 1 \end{aligned}$$

$$\log_2 2 = 1 \rightarrow T(n) = \Theta(n \log n)$$

In the **Hanoi towers problem**, the goal is to move a stack of n disks of decreasing size from one peg to another, with the following constraints:

- Three pegs are available: A, B and C. The stack of n disks begins at one of the pegs.
- Only one disk can be moved at a time.
- At any time, the disk stack of any peg must be ordered in decreasing size order, with the largest disk at the bottom and the smallest one at the top.



- a) Propose an algorithm in pseudo-code that solves this problem using a divide-and-conquer strategy. The solution must minimize the number of disk movements.
- b) Using induction, prove that for n pegs, at most $(2^n - 1)$ moves are needed. Using this result indicate and justify the algorithm's time complexity, with respect to the number of disks, n.

$$\text{Hanoi}(n, A, B, C) = \text{Hanoi}(n-1, A, C, B) + \text{Hanoi}(1, A, B, C) + \text{Hanoi}(n-1, C, B, A)$$

hanoi DC (n, src, dest)

if $n == 1$ //CASO BASE
return src \rightarrow dest

return hanoiDC($n-1$, src, aux) + hanoiDC(1, src, dest) + hanoiDC($n-1$, aux, dest)

Para mover uma torre de n discos SRC \rightarrow DEST é necessário:

(1): Mover uma torre de $n-1$ discos SRC \rightarrow AUX

T_{n-1}

(2): Mover um disco SRC \rightarrow DEST

T_1

(3): Mover uma torre de $n-1$ discos AUX \rightarrow DEST

T_{n-1}

$$T_n < 2T_{n-1} + T_1$$

Assumindo $T_k = 2^k - 1$, pretende-se provar $T_{k+1} = 2^{k+1} - 1$

CASO BASE: $T_1 = 1 = 2^1 - 1$

PASSO INDUTIVO: $T_{k+1} = 2T_k + 1 = 2(2^k - 1) + 1 = 2^{k+1} - 1$ \square

$$T(n) = 2T(n-1) = O(2^n)$$

The goal of this exercise is to explore an efficient algorithm, called the Strassen algorithm, to multiply two matrices, with dimensions $a \times b$ and $b \times c$.

$$\begin{bmatrix} x & x & x \\ x & x & x \end{bmatrix}_{2 \times 3} \times \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}_{3 \times 4} = \begin{bmatrix} x & x & x & x \\ x & x & x & x \end{bmatrix}_{2 \times 4}$$

- a) Indicate and justify the time complexity of classic/naive matrix multiplication algorithm, commonly taught in algebra classes.

As an alternative to the classic algorithm, the Strassen algorithm proposes splitting the two matrices to be multiplied, X and Y, into four blocks and performing the operations detailed below, in order to produce the matrix Z. For now, to simplify, consider that X and Y are square matrices (i.e. same number of rows and columns) with a dimension that is a power of two (such as 1×1 , 2×2 , 4×4 ...).

$$X = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \quad Y = \begin{pmatrix} E & F \\ G & H \end{pmatrix} \quad Z = \begin{pmatrix} I & J \\ K & L \end{pmatrix}$$

$$M_1 = (A + D) \times (E + H)$$

$$M_2 = (C + D) \times E$$

$$M_3 = A \times (F - H)$$

$$I = M_1 + M_4 - M_5 + M_7$$

$$M_4 = D \times (G - E)$$

$$J = M_3 + M_5$$

$$M_5 = (A + B) \times H$$

$$K = M_2 + M_4$$

$$M_6 = (C - A) \times (E + F)$$

$$M_7 = (B - D) \times (G + H) \quad L = M_1 - M_2 + M_3 + M_6$$

The algorithm's base case corresponds to multiplying two 1×1 matrices, which yields a 1×1 matrix.

- b) Explain why the Strassen algorithm follows a divide-and-conquer approach.
 c) Using the master theorem, indicate and justify the temporal complexity of the Strassen algorithm.

a) As matrizes resultados têm $a \times c$ valores. Cada valor é a soma-produto de uma linha de A com uma coluna de B, ou seja, b produtos. Assim, a complexidade é $\Theta(a \times b \times c)$

b) O algoritmo de Strassen segue uma estratégia de divisão e conquista porque divide o problema em 7 sub-problemas e depois combina os seus resultados, para além de ter um caso base.

c)

$$T(n) = \begin{cases} 1, & n=1 \\ 7 T\left(\frac{n}{2}\right) + f(n), & f(n) \in \Theta(n^2) \end{cases}$$

$a = 7$
 $b = 2$
 $c = 2$

$$\log_2 7 > 2 \rightarrow T(n) \in \Theta(n^{\log_2 7})$$

- e) Generalize strassen so that it works with non-square matrices, whose dimensions are not necessarily powers of two.

Suggestion: the Strassen algorithm itself does not need to be modified. Instead, pad the both input matrices with 0's so that they are both squared, have the same dimensions and have dimensions that are a power of 2. After performing the multiplication (you can rename the function developed in the previous question to *strassenAux* to execute the actual Strassen algorithm), cut the matrix so that it has the correct dimensions.

Consider two n-bit integers x and y and assume for convenience that n is a power of 2 (the more general case is hardly any different) which they can be split into their left and right halves, which are $n/2$ bits long:

$$x = \boxed{x_L} \quad \boxed{x_R} = 2^{n/2}x_L + x_R$$

$$y = \boxed{y_L} \quad \boxed{y_R} = 2^{n/2}y_L + y_R.$$

For instance, if $x = 10110110_2$ then $x_L = 1011_2$, $x_R = 0110_2$, and $x = 1011_2 \times 2^4 + 0110_2$. The product of x and y can then be rewritten as

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

and based on the observation that $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$, the overall operation (after some simplifications) only requires 3 multiplications rather than the original 4 multiplications.

With this “trick” due to Gauss, develop an efficient method to multiply the numbers x and y (which is rather immediate) and derive its asymptotic complexity by showing the associated divide-and-conquer recurrence and its solution using the Master Theorem.

$$\begin{aligned} xy &= (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = \\ &= 2^{n/2}x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R = \\ &= 2^{n/2}x_L y_L + 2^{n/2}[(x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R] + x_R y_R = \\ &= x_L y_L (2^{n/2} - 2^{n/2}) + x_R y_R (1 - 2^{n/2}) + 2^{n/2}(x_L + x_R)(y_L + y_R) \end{aligned}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \quad \log_2 3 > 1 \quad T(n) = \Theta(n^{\log_2 3})$$

Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose if your problem instances exhibited large values of n ?

$$\text{Teorema Master: } T(n) = \begin{cases} 1, & n \leq k \\ aT\left(\frac{n}{b}\right) + f(n), & n > k \end{cases}$$

numero de sub-problemas a *tamanho de cada sub-problema* b $a \geq 1, b \geq 1, \lim_{n \rightarrow \infty} f(n) > 0, f(n) \in \Theta(n^c)$

complexidade da divisão e combinação

- Se $\log_b a < c$, então $T(n) \in \Theta(n^c)$
- Se $\log_b a = c$, então $T(n) \in \Theta(n^c \log n)$
- Se $\log_b a > c$, então $T(n) \in \Theta(n^{\log_b a})$

$$A: T(n) = \underbrace{5}_{a} T\left(\frac{n}{2}\right) + f(n), \quad f(n) \in \Theta(n^1)$$

$\rightarrow \log_2 5 > 1 \rightarrow T(n) \in \Theta(n^{\log_2 5})$

$$B: T(n) = \underbrace{2}_{a} T\left(\frac{n-1}{b}\right) + f(n), \quad f(n) \in \Theta(1)$$

$\rightarrow \log_2 2 > 0 \rightarrow T(n) \in \Theta(n^{\log_2 2}) \approx \infty$

$$C: T(n) = \underbrace{9}_{a} T\left(\frac{n}{3}\right) + f(n), \quad f(n) \in \Theta(n^2)$$

$\rightarrow \log_3 9 = 2 \rightarrow T(n) \in (n^2 \log n)$

R.: C, pela regra de L'Hôpital

Programação Dinâmica

Grafo Multi-Estágio: grafo dirigido e etiquetado $G = (V, E)$ em que os vértices V podem ser partitionados em $K \geq 1$ conjuntos disjuntos V_1, \dots, V_K , sendo que $\forall e = (u, v) \in E : u \in V_i \text{ e } v \in V_{i+1}$, $1 \leq i \leq K - 1$ tendo cada etapa um custo e havendo nós especiais de origem e destino $|V_1| = |V_K| = 1$

Como calcular o caminho de menor custo da origem para o destino?

Princípio da Ótimalidade: se uma solução é ótima, então qualquer posição de-la deve ser ótima em relação a todas as outras escolhas possíveis dentro da posição particular

Corolário: se uma solução parcial é sub-ótima, então não tem de ser mais explorada

É possível registrar soluções parciais numa tabela, de modo a evitar recomputação de soluções parciais

$$\text{Ex: } \text{cost}(i, j) = \min_{(j, k) \in E} \{ c(j, k) + \text{cost}(i+1, k) \} \quad O(V+E)$$

Greedy: falha porque não é possível tomar a decisão ótima só baseada em informação local

Divide-and-Conquer: falha porque as decisões não são independentes

1. Identifica casos primitivos para soluções ótimas
2. Constrói soluções iterativamente usando o critério de ótimalidade
3. Constrói uma tabela e preenchi-la usando a equação de recorrência e casos primitivos

1. Characterizar a estrutura ótima da solução
2. Definir os valores da solução ótima - a recorrência
3. Computar os valores da solução ótima de baixo para cima (usando uma tabela)
4. Construir a solução final ótima

Arranjos Combinatórios:

$$\binom{n}{k} = \begin{cases} 1, & \text{se } k=0 \text{ ou } k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k}, & \text{se } 0 < k < n \\ 0, & \text{caso contrário} \end{cases}$$

Solução: construir o triângulo de Pascal

Cadeia de Multiplicações Matriz-Matriz: o número de produtos depende da forma como os produtos estão organizados \rightarrow a colocação dos parênteses define a organização das multiplicações na cadeia \rightarrow colocar os parênteses de maneira a minimizar o número de multiplicações escalares \rightarrow o número de parênteses cresce exponencialmente com o número de matrizes

$$P(n) = \left\{ \begin{array}{ll} 1, & n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k), & n \geq 2 \end{array} \right\} = C(n-1), \quad C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{\sqrt{\pi}}\right)$$

```

m = length[t] - 1;
for i = 1 to n do m[i, i] = 0;
for i = 2 to n do
  for j = 1 to i-1 do
    m[i, j] = ∞
  
```

```

for k = i to j-1 do
  q = m[i, k] + m[k+1, j] + t[i-1] * t[k] * t[j];
  if (q < m[i, j]) then
    m[i, j] = q
    ^[i, j] = k
  
```

↳ encontra o melhor K?

Knapsack Problem: max $\sum_{i=1}^n x_i w_i$, $\sum_{i=1}^n x_i w_i \leq W$, $x_i \in \{0, 1\}$

A solução ótima para um peso W deve também ser a solução ótima para o problema com peso $W - w_m$ se inclui o objeto m .

Cada decisão de incluir o objeto i deve:

1. Considerar o uso do peso w_i e consequentemente o seu valor v_i
2. Guardar o peso para outros objetos

$v[i, j]$: define o valor máximo que é possível transportar com um peso limite de j ($j \leq W$), permitindo apenas objetos numerados de 1 até i para quem incluídos.

a solução ótima é definida por $v[n, W]$

$$\begin{cases} v[i, 0] = 0 \\ v[0, j] = 0, \quad j \geq 0 \\ v[i, j] = -\infty, \quad j < 0 \end{cases}$$

$$v[i, j] = \max(v[i-1, j], v[i-1, j-w_i] + v_i)$$

não inclui o objeto i incluir o objeto i

Complexidade: $\Theta(nW)$ — pseudo-polynomial

Maior Sub-Sequência Comum (LCS): encontrar a maior sub-sequência comum de duas sequências X e Y

Complexidade de Força Bruta: $O(2^{m+n})$

1. Se $x_m = y_m$, encontrar LCS para X_{m+1} e Y_{m+1}

2. Se $x_m \neq y_m$, encontrar LCS para X_{m+1} e Y e para X e Y_{m+1}

$c[i, j]$: comprimento da LCS para as sequências X_i e Y_j

$$c[i, j] = \begin{cases} 0, & \text{se } i=0 \text{ ou } j=0 \\ c[i-1, j-1] + 1, & \text{se } i, j > 0 \text{ e } u_i = v_j \\ \max(c[i-1, j], c[i, j-1]), & \text{se } i, j > 0 \text{ e } u_i \neq v_j \end{cases}$$

LCS (string u , string v , c , b)

Complexidade: $O(nm)$

for $i=1$ to m do

$$c[i, 0] = 0;$$

for $j=0$ to n do

$$c[0, j] = 0;$$

for $i=1$ to m do

for $j=1$ to n do

if ($u[i] == v[j]$) then

$$c[i, j] = c[i-1, j-1] + 1;$$

$$b[i, j] = 1;$$

else if ($c[i-1, j] \geq c[i, j-1]$) then

$$c[i, j] = c[i-1, j];$$

$$b[i, j] = 2;$$

else

$$c[i, j] = c[i, j-1];$$

$$b[i, j] = 3;$$

Nota: $b[i, j]$ guarda as decisões (1: \uparrow ; 2: \uparrow ; 3: \leftarrow)

Como continuar LCS?

1. $i = m$;
2. $j = n$;
3. if ($i == 0$ or $j == 0$) then exit
4. if ($b[i, j] == 1$) then

$$i = i-1;$$

$$j = j-1;$$

print " u_i ";

5. if ($b[i, j] == 2$) then $i = i-1$

6. if ($b[i, j] == 3$) then $j = j-1$

7. goto 3.

Complexidade: $O(nm)$

Troco de Moedas: dado um conjunto de moedas, de denominações c_1, \dots, c_n , com valores d_1, \dots, d_n , computar o menor número de moedas tal que a soma é N (assumindo um número ilimitado de moedas de cada denominação)

$c[i, j]$: menor número de moedas necessárias para cobrir j unidades, $0 \leq j \leq N$, não usando moedas com denominação entre 1 e i , $1 \leq i \leq n$

O objetivo é computar $c[n, N]$

$$\begin{cases} c[i, 0] = 0, & 1 \leq i \leq n \\ c[i, j] = +\infty, & i=0 \text{ ou } j > 0 \end{cases}$$

$$c[i, j] = \min(c[i-1, j], 1 + c[i, j - d_i])$$

não inclui a moeda $i \leftarrow$

incluir a moeda $i \rightarrow$

Complexidade: $\Theta(nN)$

Memorização: permite reutilizar solução de sub-problemas

Arraios Combinatórios: não é necessário computar todo o triângulo de Pascal, mas sim só a entrada necessária e apenas uma vez

Tabela $c[n, k]$ =

$$\begin{cases} c[i, j] = 1, & j = 0 \\ c[i, j] = 1, & j = n \\ c[i, j] = -1, & \text{caso contrário} \end{cases}$$

$C_m(n, k)$

```

if  $c[n-1, k] \geq 0$  then  $c[n-1, k] = C_m(n-1, k)$ 
if  $c[n-1, k-1] \geq 0$  then  $c[n-1, k-1] = C_m(n-1, k-1)$ 
return  $c[n-1, k-1] + c[n-1, k]$ 

```

APSP

All-Pairs Shortest Path Problem

Objetivo: encontrar os caminhos mais curtos entre todos os pares de nós

Se pesos não negativos: Algoritmo de Dijkstra — $\mathcal{O}(VE \log V)$

Se pesos negativos: Algoritmo de Bellman-Ford — $\mathcal{O}(V^2 E)$

Estrutura de Dados: matriz de adjacências

Pesos das arestas: matriz $(n \times n)$ $W = W_{ij} = \begin{cases} 0, & \text{se } i=j \\ w_{ij}, & \text{se } i \neq j \text{ e } (i,j) \in E \\ \infty, & \text{se } i \neq j \text{ e } (i,j) \notin E \end{cases}$

Representação do Caminho: matriz $(n \times n)$ $D = d_{ij} = \delta(v_i, v_j)$

peso rodado do caminho mais curto $i \rightarrow j$

Matriz dos Predecessores: $\Pi = \Pi_{ij} = \begin{cases} \text{NULL}, & \text{se } i=j \text{ ou não existe caminho } i \rightarrow j \\ \text{ predecessores de } j \text{ no caminho mais curto } i \rightarrow j, & \text{ se não} \end{cases}$

Sub-Gráfico dos Predecessores: $G_{\Pi, i} = (V_{\Pi, i}, E_{\Pi, i})$

$$V_{\Pi, i} = \{j \in V : \Pi_{i,j} \neq \text{NULL}\} \cup \{i\}$$

$$E_{\Pi, i} = \{\Pi_{i,j} : j \in V_{\Pi, i} - \{i\}\}$$

Algoritmo de Floyd-Warshall: aumentar os caminhos mais curtos

- Para todos os caminhos de quaisquer dois nós (v_1, \dots, v_k)
- Considerar caminhos que passam por nó interno do conjunto $\{1, \dots, k\}$
- Comerçar com caminhos diretos
- Aumentar o índice dos nós que um caminho pode atravessar e determinar o mais curto

$$d_{ij}^k = \begin{cases} w_{ij}, & \text{se } k=0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}), & \text{se } k>1 \end{cases}$$

Floyd-Warshall (V):

$$n = \text{rows}[W];$$

$$D^0 = W;$$

for $k=1$ to n do

for $i=1$ to n do

for $j=1$ to n do

return D^n

$$d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$$

Complejidad: $\Theta(n^3)$

Fecho Transitivo de un Grafo Dirigido: $G^* = (V, E^*)$, $E^* = \{(i, j) : \exists i \rightarrow j \text{ en } G\}$

Algoritmo: atribuir peso 1 a cada arista e usar o algoritmo de Floyd-Warshall
 $\rightarrow d_{ij} \neq \infty \rightarrow (i, j) \in E^*$

Complejidad: $\Theta(n^3)$

$$t_{ij}^0 = \begin{cases} 0, & \text{se } i \neq j \text{ e } (i, j) \notin E \\ 1, & \text{se } i \neq j \text{ e } (i, j) \in E \end{cases}$$

$$t_{ij}^k = t_{ij}^{k-1} \vee (t_{ik}^{k-1} \wedge t_{kj}^{k-1}), \quad k \geq 1$$

Transitive-Closure (G):

$$n = V[G]$$

for $i = 1$ to n do

for $j = 1$ to n do

if $i = j$ or $(i, j) \in E$ then $t_{ij}^0 = 1$

else $t_{ij}^0 = 0$

for $k = 1$ to n do

for $i = 1$ to n do

return T^n for $j = 1$ to n do

$$t_{ij}^k = t_{ij}^{k-1} \vee (t_{ik}^{k-1} \wedge t_{kj}^{k-1})$$

Algoritmo de Johnson: referir as arestas e usar os algoritmos de Dijkstra e BF

- Se todas as arestas têm pesos não negativos, usar algoritmo de Dijkstra
- Caso contrário, computar um novo conjunto de pesos não negativos w' tal que
 - (1) o caminho mais curto $u \rightarrow v$ usando a função peso w' também é o caminho mais curto usando a função peso w'
 - (2) $\forall (u, v) \in E: w'(u, v) \geq 0$

Como reatribuir pesos às arestas?

A função de re-pesagem w' usa a função $h: V \rightarrow \mathbb{R}$ definida como

$$w'(u, v) = w(u, v) + h(u) - h(v), \text{ tendo } h(u) = \delta(\lambda, u), \text{ i.e., a menor distância da origem } \lambda \text{ para } u \text{ usando } w$$

1. Um caminho mais curto com w é um caminho mais curto com w'

2. Existe um ciclo de pesos negativos com w se e só se existe um ciclo de pesos negativos com w'

Algoritmo:

1. Dado $G = (V, E)$, devemos $G' = (V', E') = \left(V \setminus \{s\}, E \cup \{(s, v) : v \in V\} \right)$, $w(s, v) = 0$

2.

a. com ciclos de pesos negativos: detectados com Bellman-Ford em G'

b. nem ciclos de pesos negativos:

i. definir $h(v) = \delta(s, v)$

ii. dado que $h(u) \leq h(v) + w(u, v)$

iii. verifica-se $w'(u, v) = w(u, v) + h(u) - h(v) \geq 0$

3. Executar o algoritmo de Dijkstra para todos os nós $u \in V$

↳ computar $\delta'(u, v) = \delta(u, v) + h(u) - h(v)$

johnson(G):

devemos G' adicionando o nó de origem s ;

if Bellman-Ford (G' , w , s) = FALSE then

print ("ciclo negativo detectado");

else

atribuir $h(v) = \delta(s, v)$, computado usando Bellman-Ford

computar $w'(u, v) = w(u, v) + h(u) - h(v)$, para todas as arestas (u, v)

foreach $v \in V[G]$ then

executar Dijkstra (G , w' , v); computar $\delta'(u, v)$;

$$d_{uv} = \delta'(u, v) + h(v) - h(u);$$

return D

Complexidade: $O(V(V+E) \log V)$

Dijkstra usado em cada nó para computar $\delta'(u, v)$ $\left[\delta(u, v) = \delta'(u, v) + h(v) - h(u) \right]$

Usar os caminhos mais curtos da solução re-pesada para devolver as menores distâncias no grafo original (usando os predecessores no grafo original)

The factorial of an integer n ($n!$, with $n \geq 0$) is the product of all the integers between 1 and n .

- a) Write the recurrence formula for $n!$.
- b) Implement and test a function that calculates the result in a recursive method (*factorialRecurs* function).
- c) Implement and test a function that calculates the result in an iterative method with dynamic programming (*factorialDP* function).
- d) Compare the two previous algorithms in terms of their time and space complexity, with respect to n .

Consider the **variant of the change-making problem with unlimited stock**, where one wants to produce m cents of change with the least amount of coins. In this variant, there is an unlimited supply of coins of each value for the change calculation.

Note: You can assume that the coin denominations are ordered by increasing value in C .

Input example: $C = [1, 2, 5, 10]$, $n = 4$, $T = 8$

Expected result: $[1, 1, 1, 0]$

Input example: $C = [1, 2, 5, 10]$, $n = 4$, $T = 38$

Expected result: $[1, 1, 1, 3]$

- a) Write in mathematical notation the recursive functions $\text{minCoins}(i, k)$ and $\text{lastCoin}(i, k)$ that return the minimum amount of coins and the value of the last coin used to produce k **value of change** ($0 \leq i \leq n$, where n is the number of the different coins available). Use a symbol or special value if a function is not defined.
- b) Calculate the table of values for $\text{minCoins}(i, k)$ and $\text{lastCoin}(i, k)$ for the input example of the 8-cent change.
- c) Implement $\text{changeMakingUnlimitedDP}$, which uses a dynamic programming algorithm to solve the problem.

```
bool changeMakingUnlimitedDP(unsigned int C[],  
                           unsigned int n, unsigned int T, unsigned int usedCoins[])
```

Suggestion: The values of minCoins and lastCoin should be calculated for the increasing values of i and k (as arrays), memoizing only values for the last i value (one single dimension array per function).

- d) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of coin denominations, n , and the desired change amount, T .

Consider the same description for the **change-making problem** as in the TP2 sheet. Unlike in exercise 2 of this sheet, there is a limited amount of coins for each stock. Consider the function *changeMakingDP* below.

```
bool changeMakingDP(unsigned int C[], unsigned int Stock[],  
                    unsigned int n, unsigned int T, unsigned int usedCoins[])
```

Note: You can assume that the coin denominations are ordered by increasing value in *C*.

- a) Implement *changeMakingDP* using a strategy based on dynamic programming.
- b) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of coin denominations, *n*, the maximum stock of any of the coins, *S*, and the desired change amount, *T*. You can assume that the coin denominations are ordered by increasing value in *Stock*.

Note: this exercise is more challenging than exercise 2.

Consider the *calcSum* function below, which given a sequence of n integers ($n > 0$), **returns a string with m parts. The m -th part ($0 \leq m < n$) has the format s, i** ; where i represents the index in the sequence of the first value of the subsequence of $m + 1$ contiguous elements whose sum (s) is the smallest possible ($0 \leq i < n-m$).

```
std::string calcSum(int sequence[], unsigned long n)
```

Input example: sequence = [4, 7, 2, 8, 1], n=4

Expected result: "1,4;9,1;11,2;18,1;22,0;"

In this example, we have the following smallest sum contiguous subarrays:

Subarray of size 1 ($m = 0$): [1], where $s = 1, i = 4$

Subarray of size 2 ($m = 1$): [7, 2], where $s = 9, i = 1$ (*if there is more than one solution, the first is returned*)

Subarray of size 3 ($m = 2$): [2, 8, 1], where $s = 11, i = 2$

Subarray of size 4 ($m = 3$): [7, 2, 8, 1], where $s = 18, i = 1$

Subarray of size 5 ($m = 4$): [4, 7, 2, 8, 1], where $s = 22, i = 0$

- Implement *calcSum* using dynamic programming.
- Indicate and justify the temporal complexity of the algorithm, in the worst case, with respect to n .
- Using the provided *testPerformanceCalcSum* function, produce a graph with the average execution times of the algorithm for the increasing values of n 500, 1000, ..., 10000 (*testPerformanceCalcSum* function). For each value of n , generate 1000 random sequences of integers between 1 and 10 $\times n$ (repetitions allowed) and measure the average elapsed time.
Suggestion: generate a CSV format file and generate a graph using Excel or similar tool; consider the code methods below to measure the elapsed time in milliseconds (ms).

The number of ways of dividing a set of n elements into k non-empty disjoint subsets ($1 \leq k \leq n$) is given by the Stirling number of the Second Kind, $S(n,k)$, which can be calculated with the recurrence relation formula:

$$S(n,k) = S(n-1,k-1) + k S(n-1,k), \text{ se } 1 < k < n$$
$$S(n,k) = 1, \text{ se } k=1 \text{ ou } k=n$$

On the other hand, the total number of ways of dividing a set of n elements ($n \geq 0$) in non-empty disjoint subsets is given by the n^{th} Bell number, denoted $B(n)$, which can be calculated with the formula:

$$B(n) = \sum_{k=1}^n S(n,k)$$

For example, the set $\{a, b, c\}$ may be divided 5 different ways:

- {a, b, c}
- {a, b}, {c}
- {a, c}, {b}
- {b, c}, {a}
- {a}, {b}, {c}

In this case we have $B(3) = S(3,1) + S(3,2) + S(3,3) = 1 + (S(2,1)+2S(2,2)) + 1 = 1 + 3 + 1 = 5$.

$B(n)$ grows quickly. For example, $B(15) = 1382958545$.

- a) Implement the $S(n,k)$ and $B(n)$ functions using a top-down recursive approach, considering their definitions ($s_recursive$ and $b_recursive$ functions).
- b) Implement the $S(n,k)$ and $B(n)$ functions using an iterative method with dynamic programming, based on the ${}^n C_k$ calculation method presented in the theoretic classes ($s_dynamic$ and $b_dynamic$ functions).
- c) What are the temporal complexities of the four algorithms?
- d) What are the spatial complexities of the four algorithms?

Recall the TP2 sheet, when we solved the **maximum subarray problem** using brute-force and only managed to get a $O(n^3)$ time complexity. Then, in the TP4 sheet, we solved the problem with a divide-and-conquer solution, achieving a $O(n * \log(n))$ complexity. Consider the function `maxSubsequence` below.

```
int maxSubsequenceDP (int A[], unsigned int n , int &i, int &j)
```

- a) Implement `maxSubsequenceDP` using dynamic programming in order to solve this problem in linear time, $O(n)$.
- b) Using the provided `testPerformanceMaxSubsequence` function, compare the average execution times of the dynamic programming, brute-force (TP2) and divide-and-conquer algorithms (TP4) for the increasing values of n , in a similar fashion to exercise 5. Compare the results. Are they consistent with the respective temporal complexities of each solution?

Consider the same description for the **Hanoi towers problem** as in the TP4 sheet. Consider the function *hanoiDP* below.

```
std::string hanoiDP(unsigned int n, char src, char dest)
```

- a) Explain why it makes more sense to solve the Hanoi towers problem with dynamic programming than with a divide-and-conquer approach.
- b) Implement *changeMakingDP* using a strategy based on dynamic programming.

Consider the same description for the **0-1 knapsack problem** as in the TP2 sheet.

Implement *knapsackDP* using a strategy based on dynamic programming.

- a) Write in mathematical notation the recursive functions $\maxValue(i, k)$ and $\lastItem(i, k)$ that return the maximum total value and the index of the last item used in a knapsack with maximum capacity k ($0 \leq i \leq n$, $0 \leq k \leq \maxWeight$) using only the first i items ($0 \leq i \leq n$, where n is the number of the different items available). Use a symbol or special value if a function is not defined.
- b) Calculate the table of values for $\maxValue(i, k)$ and $\lastItem(i, k)$ for the input example below:

Input: values = [10, 7, 11, 15], weights = [1, 2, 1, 3], n = 4, maxWeight = 5

Expected result: [1, 0, 1, 1] (the total value is $10 + 11 + 15 = 36$)

- c) Implement *knapsackDP*, which uses a dynamic programming algorithm to solve the problem.

```
unsigned int knapsackDP(unsigned int values[], unsigned int weights[],  
                        unsigned int n, unsigned int maxWeight, bool usedItems[])
```

- d) Indicate and justify the algorithm's temporal and spatial complexity, with respect to the number of items, n , and the knapsack's maximum capacity, T .

The **edit/Levenshtein distance** between two strings is defined as the minimum number of operations to convert a string A to a string B. Three operations are possible to perform this conversion:

- Insertion: Adding a character in any position of the string.
- Substitution: Swapping a character in any position of the string with any other character.
- Deletion: Removing a character in any position of the string.

Let $D(i,j)$ be the edit distance between the first $(i+1)$ characters of a string A, $A[0:i]$, and the first $(j+1)$ characters of a string B, $B[0:j]$ (to convert $A[0:i]$ to $B[0:j]$).

- Indicate the recursive formula for the edit/Levenshtein distance, $D(i,j)$.
- Compute the edit distance between “money” and “note” (i.e. to convert “money” to “note”).

After searching for documents which include words similar to a given one, there is the need to sort them by relevance (edit distance). Consider the function *numApproximateStringMatching*, which returns the average edit distance of words in the file (named *filename*) to the searched for expression (*toSearch*). The average distance is computed by the formula (sum of distances / number of words).

```
float numApproximateStringMatching(std::string filename,
                                    std::string toSearch)
```

- Implement the auxiliary function *editDistance*, which computes the edit distance between two words, using dynamic programming.
- ```
int editDistance(std::string pattern, std::string text)
```
- Indicate the temporal and spatial complexities of *editDistance* with respect to the lengths of strings A and B ( $|A|$  and  $|B|$ , respectively). Justify your answers.
  - Using the *editDistance* function, implement *numApproximateStringMatching*.

a)

$$D(i, j) = \begin{cases} i, & j = 0 \\ j, & i = 0 \\ \min \left( 1 + D(i-1, j), 1 + D(i, j-1), D(i-1, j-1) \right), & A[i] = B[j] \\ \min \left( 1 + D(i-1, j), 1 + D(i-1, j), 1 + D(i-1, j-1) \right), & A[i] \neq B[j] \end{cases}$$

b)

|   | E | M | O | N | F | Y |
|---|---|---|---|---|---|---|
| E | 0 | 1 | 2 | 3 | 4 | 5 |
| N | 1 | 1 | 2 | 2 | 3 | 4 |
| O | 2 | 2 | 1 | 2 | 3 | 4 |
| F | 3 | 3 | 2 | 2 | 3 | 4 |
| Y | 4 | 4 | 3 | 3 | 2 | ⑤ |

To solve this and the following exercises, use the **DPGraph** class provided in the *exercises.h* file.

**DPGraph** derives from the **Graph** base class (declared in *data\_structures/Graph.h*), which implements a graph's basic functionalities and includes all the necessary attributes for the **Vertex**, **Edge** and **Graph** classes to solve these exercises.

All exercises can be solved without modifying the **Graph**, **Vertex** and **Edge** classes and by only adding auxiliary methods to the **DPGraph** class. No class attribute needs to be created on any class.

- a) Using the **DPGraph** class provided in the *exercises.h* file (which derives from the **Graph** base class), implement *dijkstra*, which uses the Dijkstra algorithm to find the shortest path from a node to all other nodes in a directed graph.

```
std::vector<Vertex *> dijkstra(const int &origin)
```

### Suggestions:

- use the provided **MutablePriorityQueue** class.
- use the *dist* and *path* attributes (and associated getters and setters) from the **Vertex** class.
- implement the private auxiliary method *relax* in the **DPGraph** class. This method performs relaxation along an edge with weight equal to *weight* between vertices *v* and *w* and returns a boolean indicating if the relaxation actually occurred.

```
bool relax(Vertex *v, Vertex *w, double weight)
```

- b) Implement the *getPath* public method in the **Graph** class.

```
vector<int> getPath(const int &origin, const int &dest)
```

Considering that the *path* property of the graph's vertices has been updated by invoking a shortest path algorithm from one vertex *origin* to all others, this function returns a vector with the sequence of the vertices of the path, from the *origin* to *dest*, inclusively (*dest* is the identifier of the destination vertex of the path). It is assumed that a path calculation function, such as *dijkstra*, was previously called with the *origin* argument, which is the origin vertex.

*Dijkstra:* Em cada momento, escolher o nó não visitado com menor distância e relaxar todos os seus vizinhos/arestas adjacentes  
 $O((V+E) \log V)$

Using the **DPGraph** class provided in the *exercises.h* file, implement *bellmanFord*, which uses the Bellman-Ford algorithm to find the shortest path from a node to all other nodes in a directed graph that can have edges of negative weight.

```
void bellmanFord(const int &origin)
```

### Suggestions:

- use the *relax* auxiliary method from exercise 10.
- use the *dist* and *path* attributes (and associated getters and setters) from the **Vertex** class.

Bellman - Ford:

1. para cada nó, percorrer todas as arestas do grafo e relaxá-las  $O(VE)$
2. para cada aresta  $(u, v)$ , se  $dist[v] > dist[u] + w(u, v)$ , indicar que há um ciclo negativo  $O(E)$

a) Using the **DPGraph** class provided in the *exercises.h* file, implement *floydWarshall*, which uses the Floyd-Warshall algorithm to find the shortest path between all pairs of nodes in a directed graph.

```
void floydWarshall()
```

**Suggestion:** Use the *distMatrix* and *pathMatrix* attributes from the **Graph** class.

- b) Implement the *getFloydWarshallPath* public method of the **DPGraph** class, which returns a vector with the sequence of elements in the graph in the path from *origin* to *dest* (where *origin* and *dest* are the values of the identifiers of the origin and destination vertices, respectively).

```
vector<int> getFloydWarshallPath(const int &origin, const int &dest)
```

*Floyd-Warshall:* construi as tabelas de distâncias e predecessores, aumentando, em cada momento, o caminho mais curto de acordo com a recorrência

$$d_{ij}^k = \begin{cases} w_{ij}, & k=0 \\ \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}), & k > 0 \end{cases}$$

$\mathcal{O}(V^3)$

johnson:

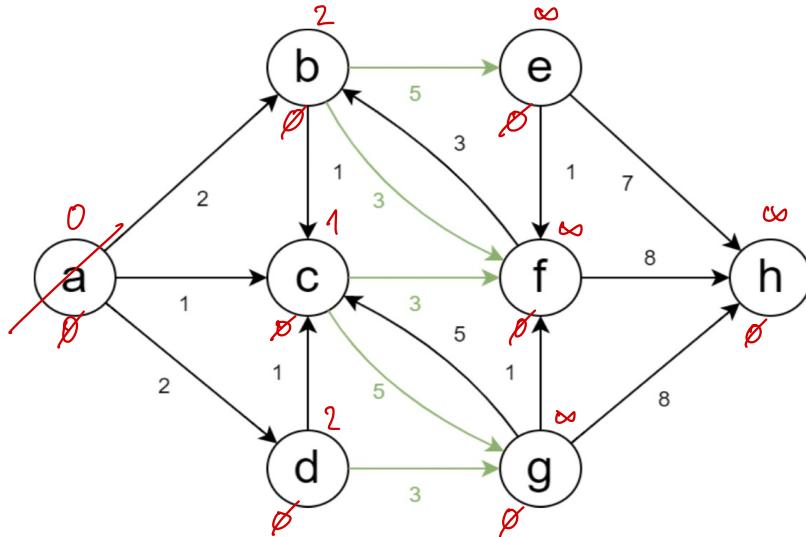
1. criar um grafo auxiliar, adicionando um nó de origem com arestas a 0
2. executar Bellman-Ford para detectar ciclos negativos e definir a distância mais curta da origem a cada nó como  $h(v)$
3. reforçar as arestas  $(u, v)$  com  $w'(u, v) = w(u, v) + h(u) - h(v)$
4. para cada nó, executar Dijkstra

$$\mathcal{O}(V(V+E) \log V)$$

A small town is modeled as a set of intersections and roads in a small town. Two vehicles are parked in node a want to travel to node h while minimizing energy consumption. One of the vehicles (vehicle A) runs on fuel while the other one (vehicle B) is hybrid. Vehicle B is able to produce energy (rather than consume) when traveling downwards.

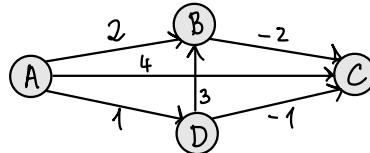
Consider the graph below where the nodes represent the town's intersections and the edges, the town's roads. The edges' weight reflects the energy needed to traverse the road (in MJ) according to the following rules:

- for vehicle A: the energy consumption in each edge corresponds to the weights shown in the graph.
- for vehicle B: the energy consumption in the black edges corresponds to the edges' weights, but for the green edges the weights indicate the energy that was produced.



- Determine the shortest path from node a to h for vehicle A. Detail all the steps of the computation.
- Determine the shortest path from node a to h for vehicle B. Detail all the steps of the computation.
- Would it make sense to have a cycle in the graph that includes the green edges? Explain.

a) Dijksta



Dijkstra: A

Bellman-Ford: A

$$d_{i,j}^k = \begin{cases} w_{i,j} \\ \min(d_{i,j}^{k-1}, d_{i,k}^{k-1} + d_{k,j}^{k-1}) \end{cases}$$

Floyd-Warshall:

|       |          | j        |          |          |          |   |          |          |     |
|-------|----------|----------|----------|----------|----------|---|----------|----------|-----|
|       |          | A        | B        | C        | D        | 1 | 0        | 1        | 2 3 |
| i \ j | A        | 0, 2     | 4        | -2       | 1        | A | 0        | 2        | B   |
| B     | $\infty$ | 0        | $\infty$ | $\infty$ | $\infty$ | B | $\infty$ | $\infty$ | C   |
| C     | $\infty$ | $\infty$ | 0        | $\infty$ | $\infty$ | C | $\infty$ | $\infty$ | D   |
| D     | $\infty$ | 3        | -1       | 0        | $\infty$ | D | $\infty$ | $\infty$ |     |

$$4, 2 - 2 = 0$$

Which algorithm design techniques (brute-force, greedy, divide-and-conquer, dynamic programming, backtracking) does the Dijkstra algorithm employ? Justify your answer.

A group of countries decided to form an allegiance. The headquarters of the annual meeting of the allegiance must be located in the capital city of one of the forming countries. The leaders agreed to build the headquarters in the city that minimized the travel time by airplane to the farthest city away.

You have access to the travel time by airplane between all pairs of capital cities. These times are represented as a 2D matrix and are measured in hours. Some cities may not be accessible directly by airplane, requiring other capital cities to be visited first. Example:

| from/to    | Bluekistan | Blackistan | Pinkistan | Redkistan |
|------------|------------|------------|-----------|-----------|
| Bluekistan | 0          | 2          | 2.5       | 6         |
| Blackistan | 2          | 0          | 1         | x         |
| Pinkistan  | 2.5        | 1          | 0         | 5         |
| Redkistan  | 6          | x          | 5         | 0         |

- Write in pseudo-code (or in C++) an efficient algorithm to solve this problem.
- Indicate the temporal and spatial complexity of the algorithm. Justify your answers.
- Apply your algorithm to the specific case presented above while detailing all the algorithm's steps.

Prove by induction that, in an undirected weight graph, after applying the Floyd-Warshall algorithm, the resulting 2D matrix of shortest path distances between nodes is symmetric (i.e. that all the shortest path distance between any pair of nodes  $(i, j)$  is the same as the distance for the pair  $(j, i)$ ).

## Backtracking

- Não há informação suficiente para saber o que escolher
- Cada decisão leva a um novo conjunto de escolhas
- Algumas sequências de escolhas podem ser uma solução para o problema

Backtracking: forma metódica de tentar várias sequências de decisões até encontrar a solução correta que funciona

Alvo: classe dos problemas de procura/busca

Ideia: de forma sistemática, procurar exaustivamente no espaço de soluções

Soma de Subconjuntos: dados  $n+1$  números positivos  $w_i$ ,  $1 \leq i \leq n$ , e  $M$ , encontrar todos os subconjuntos de  $W$  cuja soma é  $M$

Fórmula Bruta: enumera todos os subconjuntos e conta a soma de cada ( $2^n$ )

VS.

Backtracking: sequencialmente, toma as decisões de incluir/excluir cada item na solução, explorando todas as escolhas e fazendo "backtrack" se uma escolha específica for má ↴  
implicitamente, define uma Árvore de Escolhas/Estados

Recursivo Back Tracking (real  $X[1:n]$ , inteiro  $k$ )

intger  $n$ , real  $X[1:n]$ ;

for each  $X[k]$  possible choice in Choices( $X[1:k-1]$ ) do

if Solution( $X[1:k]$ ) then

return  $X[1:k]$ ;

call Recursivo Back Tracking( $X[1:n]$ ,  $k+1$ )

## Notas:

- Se, num dado estado, adicionas todos os elementos restantes não alcança  $M$ , então a escolha atual é impraticável
  - Se os valores estão ordenados, e se, num dado estado, adicionas o próximo elemento leva a um valor superior a  $M$ , então a escolha atual é impraticável

**Função Delimitadora:** se falsa, ignorar a exploração do nó atual; se verdadeira, continuar a explorar os nós

Recursive Back Tracking (real  $X[1:m]$ , integer  $k$ )

integer  $n$ , real  $X[1:n]$ ;  
for each  $X[k]$  possible choice in  $\text{Choices}(X[1:k-1])$  and  $\text{Bound}(X[1:k])$  do  
if  $\text{Solution}(X[1:k])$  then  
return  $X[1:k]$ ;  
call Recursive Back Tracking ( $X[1:n]$ ,  $k+1$ )

Intuitive Back Tracking (real  $X[1:m]$ , integer  $m$ )

integer  $k, m$ , real  $\lambda[1:n]$ ;

while ( $k > 0$ ) do

If there is an untried  $X[k]$  in  $\text{Choices}(X[1:k-1])$  and  $\text{Bound}(X[1:k])$  then  
if  $\text{Solution}(X[1:k])$  then  
Save  $\text{Solution}(X[1:k])$

$$k = k + 1$$

else

$$k = k - 1$$

## Backtracking:

- Seqüência de Decisões que formam uma Árvore de Decisões ou Estados resultantes de cada Decisão
- Estados deve ser representados como um  $n$ -tuplo  $(k_1, \dots, k_n)$  em os  $k_i$  são escolhidos de um conjunto Finito  $S$
- Todos os caminhos desde a Raiz aos outros nós definem o Espaço de Estado do Problema
- Normalmente, o Problema a Resolver procura arranjar um vetor que Maximiza/Minimiza Satisfaz uma Função Entíero  $P(k_1, \dots, k_n)$
- Restrições definem que Estados não Jávies

Cores Gráficas: Dado  $G$  um grafo e  $m$  um dado inteiro positivo, pretende-se descobrir se os nós em  $G$  podem ser coloridos de modo que nenhum dois nós adjacentes tenham a mesma cor, usando apenas  $m$  cores.

Ciclos Hamiltonianos: Dado  $G = (V, E)$  um grafo conexo com  $n$  vértices, um ciclo hamiltoniano é um caminho de ida e volta ao longo de  $n$  arestas de  $G$  que visita cada vértice exatamente uma vez e regressa à sua posição inicial.

Knapsack 0/1: dados  $n$  pesos positivos  $w_i$ ,  $n$  lucros positivos  $p_i$  e um número positivo  $M$  que é a capacidade da mochila, o problema consiste em escolher um subconjunto de pesos tal que

$$\sum_{i=1}^n w_i x_i \leq M \quad \text{e} \quad \sum_{i=1}^n p_i x_i \text{ é maximizado}$$

Os  $x_i$ 's constituem um vetor de valores 0-1

The objective of this exercise is to find the exit of a 10 by 10 labyrinth. The initial position is always at (1, 1). The labyrinth is modeled as a 10x10 matrix of integers, where a 0 represents a wall, a 1 represents free space, and a 2 represents the exit.

- a) Implement the *findGoal* function (see *Labirinth.h* and *Labirinth.cpp*), which finds the way to the exit using backtracking algorithms. This function should call itself recursively until it finds the solution. In each decision point in the labyrinth, the only possible actions are to move left, right, up or down (see *Labirinth.h* and *Labirinth.cpp*). Once the exit is found, a message should be printed to the screen. It is suggested that you use a matrix to keep track of the points which have been visited already.
- b) Assuming that the maze is located within a square  $n \times n$  grid, what is the temporal complexity of the algorithm in the worst case, with respect to  $n$ ?

Consider the same description for the **subset sum problem** as in the TP2 sheet.

```
bool subsetSumBT(unsigned int A[], unsigned int n, unsigned int T,
 unsigned int subset[], unsigned int &subsetSize)
```

- Propose in pseudo-code a backtracking algorithm to solve this problem. Your algorithm should return two outputs: a boolean indicating if the subset exists and the subset itself.
- Indicate and justify the algorithm's temporal and spatial complexity, in the worst case, with respect to  $n$ .
- Implement *subsetSumBT* using the proposed algorithm.

*//* **bool** *subsetSumBT* (*A*[ $\cdot$ ],  $n$ ,  $T$ , *subset*[ $\cdot$ ], *subsetSize*)  
*// sort(A);*  
*subsetSize* = 0;  
return *subsetSumBT\_rec* (*A*,  $n$ ,  $T$ , 0, *subset*, *subsetSize*);

**bool** *subsetSumBT\_rec* (*A*[ $\cdot$ ],  $n$ ,  $T$ ,  $i$ , *subset*[ $\cdot$ ], *subsetSize*)  
if ( $T == 0$ ) return true;  
if ( $i == n$ ) return  $T == 0$ ;  
if ( $A[i] \leq T$ )  
 *subset*[*subsetSize* + 1] = *A*[ $i$ ];  
 if (*subsetSumBT\_rec* (*A*,  $n$ ,  $T - A[i]$ ,  $i + 1$ , *subset*, *subsetSize*)  
 return true;  
 *subsetSize* --;  
return *subsetSumBT\_rec* (*A*,  $n$ ,  $T$ ,  $i + 1$ , *subset*, *subsetSize*);

Complexidade:  $O(2^n)$

$$\cdot T(n) = \begin{cases} 1, \dots \\ T(n-1) + 1, \dots \end{cases} \longrightarrow T(n) = O(2^n)$$

Consider the same description for the **change-making problem** as in the TP2 sheet.

```
bool changeMakingBT(unsigned int C[], unsigned int Stock[],
unsigned int n, unsigned int T, unsigned int usedCoins[])
```

- a) Implement *changeMakingBT* using a strategy based on backtracking.
- b) Indicate and justify the algorithm's temporal complexity, in the worst case, with respect to the number of coin denominations, n, and the maximum stock of any of the coins, S.

Consider the same description for the **activity selection problem** as in the TP3 sheet.

```
vector<Activity> activitySelectionBT(vector<Activity> A)
```

Implement *activitySelectionBT* using a strategy based on backtracking.

Consider the same description for the **0-1 knapsack problem** as in the TP2 sheet.

```
unsigned int knapsackBT(unsigned int values[], unsigned int weights[],
 unsigned int n, unsigned int maxWeight, bool usedItems[])
```

Implement *knapsackBT* using a strategy based on backtracking.

Consider the same description for the traveling salesman problem (TSP) as in the TP2 sheet.

```
unsigned int tspBT(const unsigned int **dists, unsigned int n, unsigned
int path[])
```

Implement *tspBT* using a strategy based on backtracking.

*tspBT* ( $\text{**dists}$ ,  $n$ ,  $\text{path}[]$ )

$\text{path}[0] = 0;$

return *tspBT- rec* ( $\text{dists}$ ,  $n$ ,  $\text{path}$ , 1)

*tspBT- rec* ( $\text{**dists}$ ,  $n$ ,  $\text{path}[]$ ,  $i$ )

if ( $i == n$ ) return  $\text{path}. \text{cost}();$

Sudoku [<http://en.wikipedia.org/wiki/Sudoku>] is a game in which the objective is to fill a 9x9 matrix with numbers from 1 to 9, without repeating numbers in any row, column or 3x3 blocks.

- a) Implement the *solve* function, which (efficiently) solves Sudoku of any degree of difficulty, using a backtracking algorithm. The algorithm should work recursively: it should fill in a cell and call itself to solve the remaining puzzle.

Suggestion: Use the following greedy algorithm to choose the cell to fill in: select the cell with the minimum number of possible values (ideally 1). Implement this algorithm in the *findBestCell* function.

- b) Implement the *countSolutions* function, which determines the multiplicity of solutions (no solution, one solution or more than one solution) of a given Sudoku.

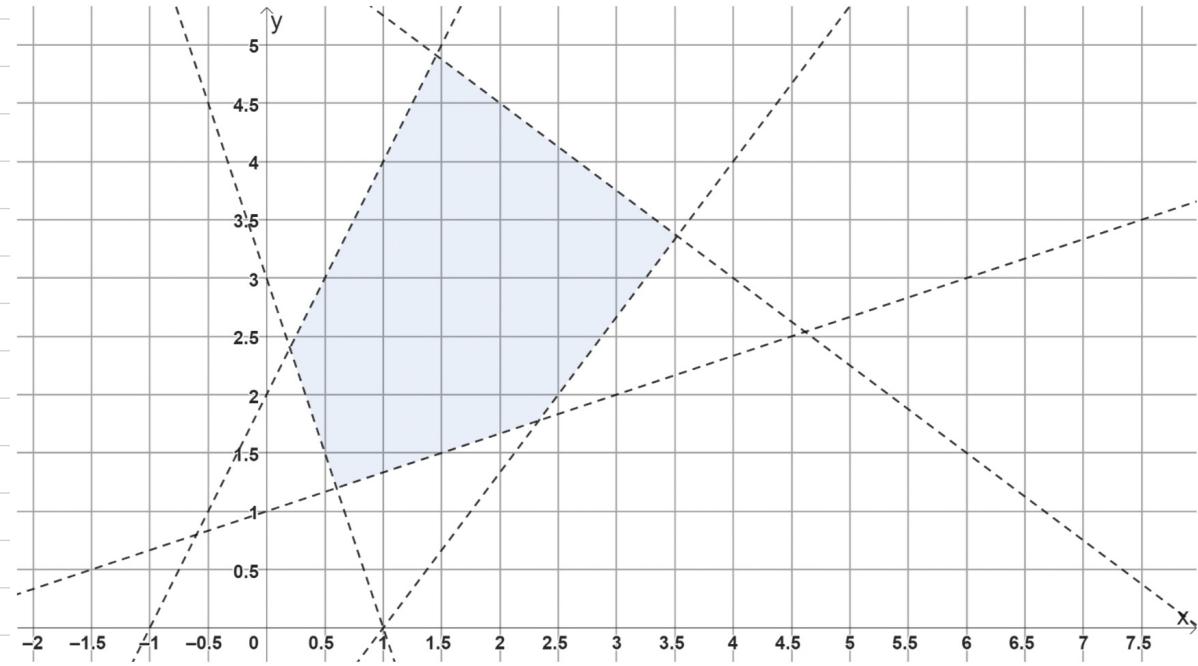
Suggestion: adapt the implementation of the *solve* function.

- c) Implement the *generate* function, which automatically generates Sudoku.

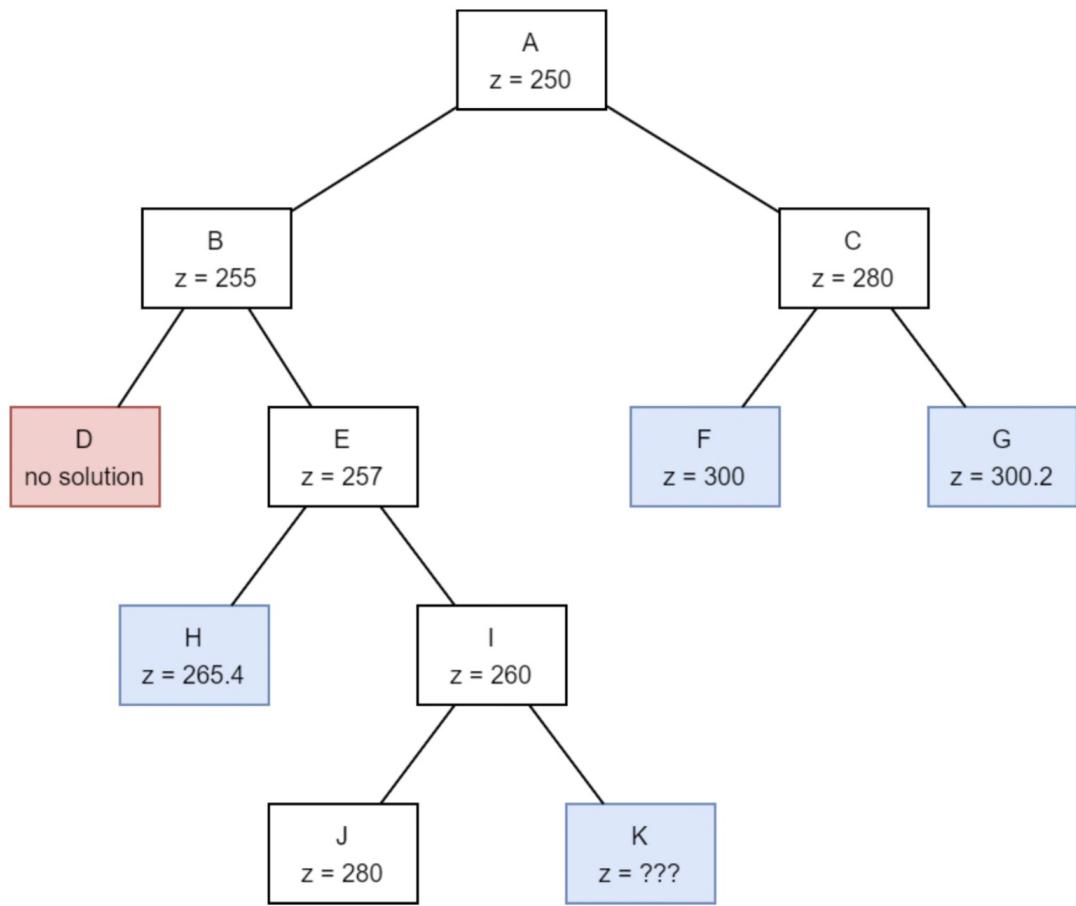
Suggestion: starting with an empty Sudoku, randomly choose a cell and a number; if the cell is not filled in and the chosen number is valid for that cell, fill it in and analyze the multiplicity of solutions; if the Sudoku becomes impossible, clear the cell; if it has one solution, terminate; otherwise, continue the process.

The figure below depicts the graphical representation of an Integer Programming (IP) problem with two decision variables,  $x$  and  $y$ . A set of constraints in the problem limited the feasible region to the polygon highlighted in blue. The objective function, to be minimized, is  $z = x - 2y$ .

Using the Branch-and-Bound algorithm, find the optimal solution. Explain the main steps of the algorithm by drawing a search tree and represent the optimal solution graphically in the plot. The search for the optimal solution should be conducted using Depth-First Search (DFS).



A computer running the Branch-and-Bound algorithm for an Integer Programming problem produced the search tree below. This problem has an objective function known as  $z$ .



The nodes of the search tree were explored in alphabetical order. The nodes in blue correspond to all-integer solutions, while the node in red represents an inadmissible solution to the relaxed Linear Programming problem.

- Determine and justify whether this is a maximization or minimization problem.
- Assuming node K contains the optimal solution, determine this node's lower and upper bounds for  $z$ .
- Which algorithm was used to explore the solution space (Depth-First Search or Breadth-First Search)? Justify your answer. Is this strategy the best suited one for this problem?

Another computer running the Branch-and-Bound algorithm for an Integer Programming problem produced the set of solutions below, in order. This problem has five decision variables ( $x_1, x_2, x_3, x_4, x_5$ ) and one objective function known as F.

| Solution | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | F           |
|----------|-------|-------|-------|-------|-------|-------------|
| A        | 13    | 6.667 | 30    | 0     | 0     | 650.7       |
| B        | 12.4  | 7     | 30    | 0     | 0     | 650         |
| C        | 13    | 7     | 30    | 0     | 0     | no solution |
| D        | 12    | 7     | 30    | 0     | 0.25  | 647.8       |
| E        | 11    | 7     | 30    | 0     | 1     | 645.5       |
| F        | 12    | 7.2   | 30    | 0     | 0     | 647.7       |
| G        | 10.5  | 8     | 30    | 0     | 0     | 642.7       |
| H        | 12    | 7     | 30.45 | 0     | 0     | 647.5       |
| I        | 12.33 | 7     | 31    | 0     | 0     | 642         |
| J        | 12    | 7     | 30    | 0     | 0     | 646         |
| K        | 13    | 6     | 30    | 1.5   | 0     | 648         |
| L        | 16    | 5     | 30    | 2     | 0     | 647.2       |
| M        | 12.5  | 5     | 30    | 0     | 0     | 647         |

- a) Determine and justify whether this is a maximization or minimization problem.
- b) Draw the search tree produced by the computer. In each node, indicate the current lower and upper bound for the objective function F.
- c) In the search tree, identify the nodes that correspond to:
  - i) Inadmissible solutions to the relaxed Linear Programming problem.
  - ii) All-integer solutions. Distinguish those that are considered the best solution found so far from those that are pruned when they are found.
- d) Which node of the tree corresponds to the optimal solution?
- e) Which algorithm was used to explore the solution space? Justify your answer.

We now consider the sum-of-subsets problem as described in class for the instance  $S = \{5, 2, 9, 3, 5, 8\}$  and  $M = 20$ . Does this problem have a solution for  $M = 20$ ? And what about for  $M = 4$ ? Provide a solution example and argue that for infeasibility.

$$S = \{5, 2, 9, 3, 5, 8\}$$

$\downarrow$   
SORT

$$S = \{2, 3, 5, 5, 8, 9\}$$

$$M = 20 \rightarrow S = \{2, 5, 5, 8\}$$

$$M = 4 \rightarrow \text{NÃO EXISTE}$$

Solve the following sum-of-subset problem  $W = \{1, 3, 4, 5\}$ ,  $M=8$  using backtracking showing the complete tree of states and use the bounding functions described in class and determine which states can be pruned.

$$W = \{1, 3, 4, 5\}$$

$$M = 8$$

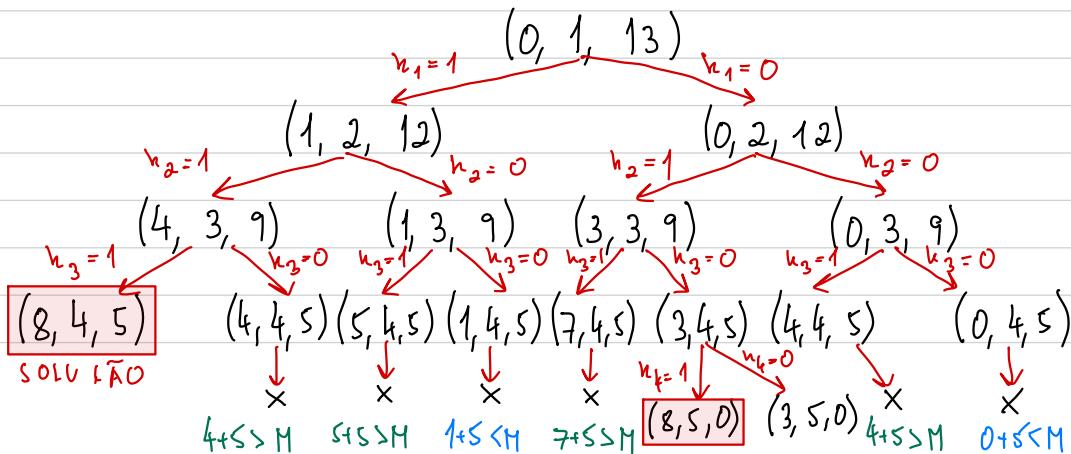
"Bounding Functions"

$$\sum_{i=1}^k x_i w_i + w_{k+1} > M$$

$$\sum_{i=1}^k x_i w_i + \sum_{i=k+1}^n w_i < M$$

$x_i \in \{0, 1\} \rightarrow$  inclui  $w_i$

tuple: (índice atual, elemento analisado, nome elementos restantes)



# Problemas

Problemas Tratáveis: algoritmos que podem ser resolvidos em tempo polinomial -  $T(n) = O(n^k)$

Problemas Intratáveis: não podem ser resolvidos em tempo polinomial; requerem tempo super-polinomial

Problema da Paragem: problema de decisão em que dada a descrição de um programa e um input finito, decidir se o problema termina ou corre infinitamente, dado esse input

Halt (P, I):

Input: um programa  $P$  e um input  $I$  para  $P$

Output: true se  $P$  termina em  $I$ , falso caso contrário

Break (x):

```
if (ValidCode (r) AND Halt (r, x)) then
 while true do nothing
else return true
```

Se  $\text{Break}(\text{Break})$  para, então  $\text{Halt}(\text{Break}, \text{Break})$  é falso

↳ Isto é, por contradição,  $\text{Break}(\text{Break})$  não para

Se  $\text{Break}(\text{Break})$  não para, então  $\text{Halt}(\text{Break}, \text{Break})$  é verdade

↳ Isto é, por contradição,  $\text{Break}(\text{Break})$  para

Contradição em ambos os casos  $\rightarrow \text{Halt}$  não pode existir!

Problemas NP-Completo: não demonstrado se podem ou não ser resolvidos polynomialmente

Conjectura: os problemas NP-Completo são intractáveis

Ciclo de Euler: visitar todas as arestas apenas uma vez — um grafo conexo tem um ciclo de Euler se e só se todos os vértices têm grau par — Polynomial

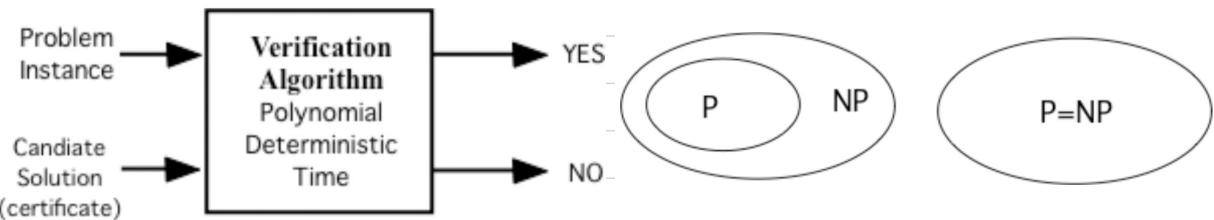
Ciclo Hamiltoniano: visitar todos os nós apenas uma vez — NP-Completo

Satisfatibilidade Booleana (SAT): dada uma fórmula booleana com variáveis booleanas  $x_i$  e operadores  $\wedge, \vee, \neg$ , há alguma atribuição de valores às variáveis  $x_i$  que torna a fórmula verdadeira? — NP-Completo

Problemas NP: resolvíveis em tempo polynomial de forma não determinística e verificáveis em tempo polynomial

• Todos os problemas em P também pertencem a NP, mas não é preciso tratar provisamente do não-determinismo

• É  $P \neq NP$ , i.e., há problemas em NP que não podem estar em P?  
• É NP-Completo? São todos os problemas em NP igualmente difíceis?



Como mostrar que um problema é NP-Completo?

- Decisão VS Otimização
- Redução
- Problema NP-Completo Base

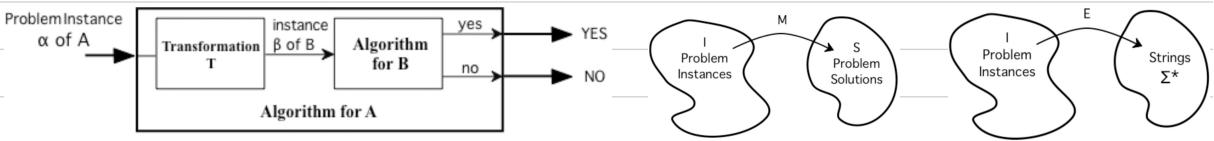
Problema de Otimização: encontrar o máximo/mínimo h tal que algo se verifica

Problema de Decisão: impor um limite/fronteira h e ver se o problema é viável

- (1) Se o Problema de Otimização é fácil, o Problema de Decisão também
- (2) Se o Problema de Decisão é difícil, o Problema de Otimização também
- (3) O Problema de Decisão não é mais difícil do que o de Otimização!

Redução: dados um problema A a resolver em tempo polinomial e um problema B com um algoritmo de tempo polinomial, é uma transformação T de uma instância  $\alpha$  de A para uma instância  $\beta$  de B tal que T demora tempo polinomial e gera respostas idênticas entre  $\alpha$  e  $\beta$

Redução de Tempo Polinomial: permite resolver A em tempo polinomial - A  $\leq_f$  B



Problema Abstrato: mapeamento M de instâncias do problema para soluções

Codificação: os problemas devem ser representados como strings em  $\{0,1\}^*$

O tempo é polinomial no tamanho da representação codificada

• M é pelo menos tão difícil como L E NP se e só se  $L \leq_f M$ , i.e.,  
L é polynomialmente reduzível para M, i.e.,  $n \in L \Leftrightarrow f(n) \in M$

• Um problema M é NP-Completo se:

1. M E NP, i.e., pode ser verificado em tempo polynomial
2. É NP-Difícil, i.e., tão difícil como qualquer problema L E NP

① Se existir um algoritmo polynomial para  $M \in NPC$ , então todos os problemas  $L \in NP$  podem ser resolvidos em tempo polynomial, tendo  $L \leq_f M$ , resolvendo M e resolvendo L e  $N = NP$

Como mostrar que um problema é NP-Completo? Reduzir NPC L a M

1. Mostrar que M E NP (só verificável em tempo polynomial)
2. Selecionar uma linguagem apropriada NPC M
3. Descrever um algoritmo que computa  $f$ , uma função de mapeamento de L para M, tal que para qualquer instância  $x$  de L temos  $f(x)$  é uma instância de M
4. Provar que  $\forall x, x \in L \Leftrightarrow f(x) \in M$
5. Provar que o algoritmo que computa  $f$  corre em tempo polynomial em relação ao tamanho da instância de L

CIRCUIT-SAT: dado um circuito combinatório booleano com N inputs  $x_i$  e um output  $y$ , existe alguma atribuição de valores booleanos a  $x_i$  que faz  $y = 1$ ?

Algoritmo: força-bruta  $\rightarrow O(2^n)$ , sendo  $n$  o número de portas ?? NP-Completo

SAT: determinar se uma fórmula booleana tem uma atribuição verdadeira

$\left. \begin{array}{l} \text{NP - equivalente a travessia de grafo} \\ \text{NP-Difícil - CIRCUIT-SAT} \end{array} \right\} NPC!$

**CNF SAT**: SAT na forma normal conjuntiva — SAT  $\leq_p$  CNFSAT — NP-Completo

**3CNFSAT**: CNFSAT em que cada cláusula tem exatamente 3 literais — NPC

**2CNFSAT**: CNFSAT em que cada cláusula tem exatamente 2 literais — P!

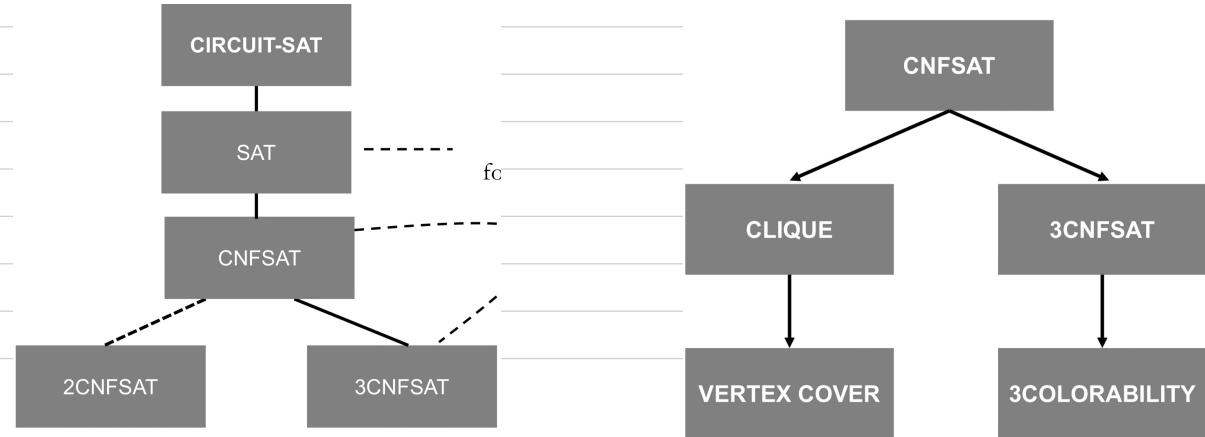
- ↓
1. Para cada variável inclui dois nós no grafo ( $n_i$  e  $\neg n_i$ )
  2. Para a cláusula ( $n \vee y$ ) adiciona as arestas ( $n_i \rightarrow y_j$ ) e ( $\neg n_i \rightarrow y_j$ )
  3. Se existir um SCC com  $n_i \in \neg n_i$ , então não é satisfazível. Senão, sim!

**CLIQUE**: existe sub-grafo completo de  $G = (V, E)$  com  $k$  nós? — NP-Completo

**Colaboração dos Vértices**: existe um conjunto de vértices  $V' \subseteq V$  tal que os vértices em  $G$  incidem em pelo menos um vértice de  $V'$ ? — NP-Completo (NP e Difícil)

**3-Cores**: decide se  $G$  pode ser colorido usando 3 cores, atribuindo cores aos vértices de modo que vértices adjacentes tenham cores diferentes

Todos estes problemas NP não NP porque a complexidade da verificação é linear no tamanho da instância, i.e., não equivalentes a travessia de grafos em que  $T(n) = O(V+E)$



## Algoritmos de Aproximação

- Para problemas NP-Completo, desenvolvem-se heurísticas eficientes
- Um algoritmo de aproximação retorna soluções quase ótimas e pode ter casas de prasar fronteiras/límites no rácio entre a solução aproximada e a solução ótima

Algoritmo de Aproximação  $\rho(n)$ : se o custo  $C$  da solução produzida pelo algoritmo é um fator de  $\rho(n)$  do custo  $C^*$  da solução ótima

$$\max \left( \frac{C}{C^*}, \frac{C^*}{C} \right) \leq \rho(n)$$

Maximização:  $O \leq C \leq C^*$

Minimização:  $O \leq C^* \leq C$

$$\rho(n) \geq 1$$

$$\text{Idealmente, } \rho(n) = 1$$

$$\rho(n) = 1 + \varepsilon$$

Esquema de Aproximação  $1 + \varepsilon$ : algoritmo de aproximação  $(1 + \varepsilon)$ ,  $\varepsilon$  fixo

Esquema de Aproximação de Tempo Completamente Polinomial: esquema de aproximação cujo tempo de execução é polinomial em  $1/\varepsilon$  e em  $n$

Normalmente, o tempo de execução de um esquema de aproximação varia rapidamente com a diminuição de  $\varepsilon$

$$T(n) = O \left( \frac{n^3}{\varepsilon^2} \right)$$

Problema da Cobertura dos Vértices: encontrar o menor conjunto  $V' \subseteq V$  tal que as arestas em  $G$  não incidentes em pelo menos um vértice  $v \in V'$

Algoritmo "Greedy":

1. Escolher uma aresta arbitrária
2. Remover as arestas incidentes nos nós da aresta escolhida
3. Até que não fiquem ser escolhidas/removidas mais arestas

Approx-Vertex-Cover ( $G$ ):

$$C = \emptyset$$

$$E' = E[G]$$

while  $E' \neq \emptyset$  do

    Seja  $(u, v)$  uma aresta arbitrária de  $E'$

$$C = C \cup \{u, v\}$$

    Remover de  $E'$  qualquer aresta incidente em  $u$  ou  $v$

return  $C$

Complexidade:  $\mathcal{O}(V+E)$

Teorema: o algoritmo é uma Aproximação Polinomial de Tempo 2

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) = \frac{C}{C^*} \ll 2$$

iterativamente

Heurística: retocar o grafo obtido por aproximação baseando-se no facto de que se todos os vizinhos de um nó estão em  $V'$ , então é possível remover esse nó de  $V'$ ... mas a ordem é importante!

Problema da Mochila 0/1: max  $\sum_{i=1}^k v_{a_i}$  tal que  $\sum_{i=1}^k w_{a_i} \leq W$

Algoritmo A: ordenar objetos por valores decrescentes de  $v_i/w_i$  e escolher os primeiros  $k$  objetos enquanto ainda couberem — não ótimo

Algoritmo B: escolher de forma "greedy" os itens por ordem decrescente de valor  $v_i$  e preencher a mochila até ao último item não caber — não ótimo

Algoritmo de Aproximação:

1. Usar o algoritmo "greedy" A para computar a aproximação  $V_a$
2. Usar o algoritmo "greedy" B para computar a aproximação  $V_b$
3. Escolher o melhor/máximo de entre  $V_a$  e  $V_b$

Teorema: o algoritmo é uma Aproximação Polinomial de Tempo 2

$$\sum_{i \in S} v_i + v_j \geq C^* \Leftrightarrow V_a + V_b \geq C^* \xrightarrow{\max(v_{ij}) \geq \frac{h}{2}} \max(V_a, V_b) \geq C^*/2 \rightarrow C^*/C \leq 2$$

Como Provar que um problema não pode ser bem aproximado?

- Dado um problema de otimização NP-Completo  $X$
- Prodigi-lo em tempo polinomial para um problema de minimização  $Y$  t.g.:
  - Uma instância de  $X$  corresponde a uma instância de  $Y$  com valor no máximo  $k$
  - Nenhuma instância de  $X$  corresponde a instâncias de  $Y$  com valor maior do que  $p_k$
  - A não ser que  $P = NP$ , não existe algoritmo de aproximação em tempo polinomial para  $Y$

Problema do Caixão Viajante (TSP): dado um grafo não dirigido  $G = (V, E)$  com função de custo  $c$ , encontrar um círculo que visita todos os nós em  $G$  com mínimo peso agregado

$$c(A) = \sum_{(u,v) \in A} c(u,v)$$

Círculo: caminho que visita todos os nós em  $G$  exatamente uma vez e termina onde começou

Desigualdade Triangular:  $c(t, v) \leq c(t, u) + c(u, v), \quad \forall (t, v), (t, u), (u, v) \in E$

Algoritmo "Greedy":

1. Selecionar um vértice como "raiz"
2. Computar MST desde esse vértice
3. Usar o caminho indireto por  $T$  como o círculo

Aprox-TSP-Tour:

Selecionar qualquer  $r \in V$  como vértice "raiz"

Computar MST  $T$  com raiz em  $r$

$L$ : visita pré-ordem de  $T$

Definir o círculo  $H$  que visita  $G$  usando a ordem em  $L$   
return  $H$

Complexidade:  $O(V+E)$

Teorema: o algoritmo é uma Aproximação Polinomial de Tempo 2 para instâncias do TSP que verificam a desigualdade triangular

Teorema: se  $P \neq NP$ , então para qualquer constante  $p > 1$ , não existe algoritmo de aproximação em tempo polinomial com razão de aproximação  $p$  para o problema geral TSP

Problema da Cobertura de Conjuntos: dadas um conjunto finito de pontos  $X$ , um conjunto de sub-conjuntos de pontos  $F \subseteq X$  tal que  $X = \bigcup_{S \in F} S$ , encontrar o menor conjunto  $C$  de  $F$  tal que  $X = \bigcup_{S \in C} S$ , i.e., o menor número de instâncias de  $F$  que cobrem todos os pontos no conjunto de input  $X$

Algoritmo "Greedy":

1. Escolher o conjunto de  $S$  que cobre o maior número de pontos por cobrir de  $X$
2. Atualizar o conjunto de pontos cobertos
3. Repetir até todos os pontos estarem cobertos

Approx-Greedy-Covering ( $X, F$ ):

$$U = X$$

$$C = \emptyset$$

while  $U \neq \emptyset$  do

  selecionar um  $S \in F$  que maximiza  $|S \cap U|$

$$U = U - S$$

$$C = C \cup \{S\}$$

return  $C$

Complexidade:  $O(|X||F|)$ , pois o número de iterações é limitado por  $\min(X, F)$

Teorema: o algoritmo é uma Aproximação Polinomial de Tempo  $\rho(n)$ , sendo  $\rho(n) = H(\max\{|S| : S \in F\})$ , com  $H(d) = 1 + \frac{1}{2} + \dots + \frac{1}{d}$  e  $H(0) = 0$

Prova:  $\sum_{k=1}^n \frac{1}{k} \leq \ln n + 1$

Conclusão: o algoritmo é uma Aproximação Polinomial de Tempo  $\ln(|X|) + 1$

# Programação Linear

Região Convexa: todos os pontos de um segmento que ligue quaisquer dois pontos dentro da região também estão dentro da região

Facto: região factível é um polígono convexo

- A solução ótima, se existir, ocorre sempre na fronteira e pode não ser única
- A solução ótima não existe se a região convexa for vazia ou se não for limitada

Solução Factível: qualquer solução para o conjunto de restrições

• A cada solução factível corresponde um valor da função objetivo

Região Factível: conjunto de soluções factíveis  $\longrightarrow$  SIMPLEX

• A região factível é uma região convexa num espaço N-dimensional

Programação Linear: otimizar (maximizar/minimizar) uma função linear sujeita a restrições lineares

$$\text{Função Linear: } f(x_1, \dots, x_n) = \sum_{j=1}^n c_j x_j$$

$$\text{Restrições Lineares: } g(x_1, \dots, x_n) = \sum_{j=1}^n a_{ij} x_j \stackrel{?}{\leq} b_i$$

Algoritmo Simplex: se a região fejável é limitada, então existe solução e ela ocorre num ou mais vértices.

1. Calcular a localização de todos os vértices
2. Calcular a função objetivo em cada um deles
3. Encontrar o menor/maior valor

Algoritmo Simplex: segue um caminho através dos vértices da região fejável que aumenta a função objetivo de forma "greedy"

Forma Standard: maximizar  $\sum_{j=1}^n c_j x_j$  sujeito a  $\sum_{j=1}^n a_{ij} x_j \leq b_i, x_j \geq 0$

$$i = 1, \dots, m \quad j = 1, \dots, n \quad a_{ij}, c_j, b_i \in \mathbb{Z}$$

Matriz: maximizar  $c^T x$  sujeito a  $Ax \leq b, x \geq 0$

Se houver variáveis sem restrições, substituir  $x_i$  por 2 variáveis  $x_{i1}$  e  $x_{i2}$  e multiplicar os coeficientes de  $x_{i2}$  por -1, tendo ambas as variáveis restrições de  $\leq$

$$\min \underbrace{x^{(-1)}}_{\text{max}} = \underbrace{\geq}_{\leq} \quad \underbrace{\leq}_{\geq} \quad \underbrace{\geq}_{\leq} \quad \underbrace{\leq}_{\geq}$$

Forma Slack:  $s_i = b_i - \sum_{j=1}^n a_{ij} x_j, s_i \geq 0$

Substituir todas as restrições de desigualdade por restrições de igualdade, exceto para variáveis individuais com restrições não negativas  
 Para cada restrição, introduzir uma "folga" adicional

Variáveis Básicas: variáveis expressas em termo de outras variáveis

Variáveis Não-Básicas: variáveis que definem as variáveis básicas

Objetivo a Maximizar:  $z = \sum_{j=1}^n c_j x_j + v$

Forma Slack:

$$(N, B, A, b, c, v)$$

N: variáveis não-básicas,  $|N| = n$   
B: variáveis básicas,  $|B| = m$   
v: constante da função objetivo

Algoritmo Simplex:

- Problema na Forma Slack
- Solução Inicial Fazível
- Pivots Iterativos para Melhorar a Função Objetivo
- Escolha Greedy das Variáveis Pivot

1. Encontra um vértice dentro da região fejável
2. Enquanto houver um vizinho do vértice atual com um valor maior de  $c_x$ , mover para ele
3. Terminar quando alcança um ótimo local

Vértice? O conjunto de variáveis não-básicas é 0 e o valor da função custo é dado pelo valor de  $v$

Mover para Vértice? Toma uma variável que é atualmente zero (não básica) em não-zero (básica)

Variável que Sai: variável que é substituída (básica  $\rightarrow$  não-básica)

Variável que Entra: variável que substitui (não-básica  $\rightarrow$  básica)

Solução Inicial Básica: defini variáveis não-básicas a 0, básicas a  $b_i$ .

• Escolher uma variável pivot não-básica  $x_k$  tal que:

1. O valor da função objetivo cresce ( $\rightarrow$  máximo)

2. É limitada por restrições de variáveis básicas (positividade)

3. A variável não-básica  $x_k$  vai se tornar básica

• Escolher variável básica  $x_l$  para se tornar não-básica

Variável que entra:  $N' = N - \{X_k\} \cup \{X_l\}$

Variável que sai:  $B' = B - \{X_l\} \cup \{X_k\}$

Nova Forma Slack:  $(N', B', A, b, c, v)$

• Determinar forma slack inicial para uma solução inicial fejável, caso contrário o problema é infejável e o algoritmo termina

$$z = \sum_{j=1}^n c_j x_j + v$$

• Enquanto existir  $c_j$  tal que  $c_j > 0$  (*i.e.*, o valor de  $z$  vai crescer):

•  $x_{entada}$  define uma nova variável de entrada (*i.e.*, nova variável básica)

• Escolher  $x_{saída}$

•  $x_{saída}$  corresponde à restrição  $i$  que minimiza  $b_i/a_i$  para  $a_i < 0$

• Se  $a_i > 0, \forall i$ , o problema é ilimitado

• Pivot com  $(N, B, A, b, c, v, saída, entada)$

# RESUMO

1. começar com o problema na Forma Glack

2. repetir:

- escolher uma variável  $x_i$  com coeficiente positivo na função objetivo
- encontrar a equação  $\text{NÃO-BÁSICA} = 0$   $b + c_1x_1 + \dots + c_nx_n = 0$  com  $c_i < 0$  e  $-b/c$  mínimo
- rescrever a equação com  $x_i = -b/c + 1/c_i + \dots$
- substituir  $-b/c + 1/c_i + \dots$  por  $x_i$  em todas as equações e função objetivo
- até: não existe variável  $x_i$  em equação

3. Se não existe  $x_i$ : solução ótima encontrada!

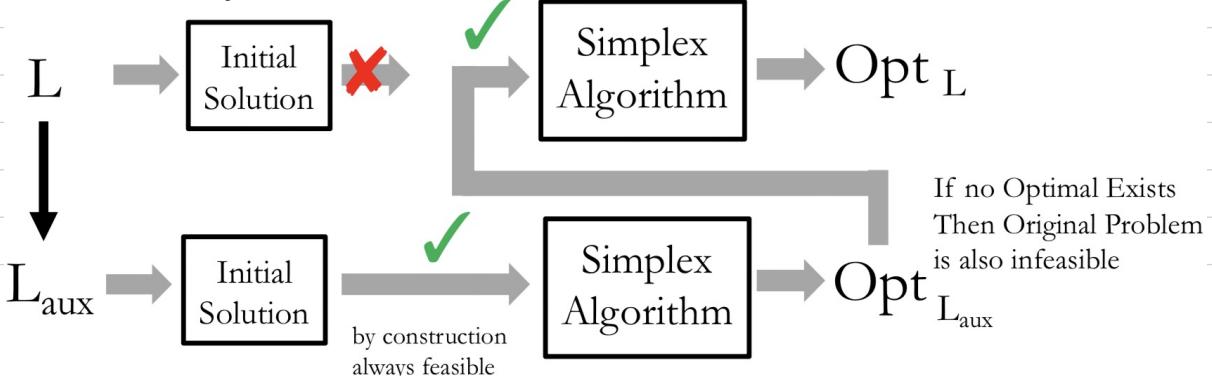
Não: não existe solução ótima

Problema: um problema de Programação Linear pode ser fezível, mas a solução básica inicial pode não ser fezível ...

Solução: usar um problema de Programações Lineas auxiliares, resolvê-lo e revisitar o problema original com a solução do problema auxiliar como nova solução inicial

Auxiliar: maximizar  $-x_0$  sujeito a  $\sum_{j=1}^n a_{ij}x_j - x_0 \leq b_i, x_j \geq 0$

$L$  é fezível  $\Leftrightarrow$  a solução de  $L_{aux}$  tem  $x_0 = 0$



## Resultados:

- A Forma Yslack é única
  - O valor da Função Objetivo nunca decrece
  - O algoritmo Simplex vai entrar em ciclo quando existem formas Yslack idênticas para duas iterações
- Se o algoritmo não terminar ao fim de  $\frac{m}{m}$  iterações, está em ciclo
- EVITAR!** Como num empate, escolha a variável com menor índice

**Dualidade:** Para cada problema primo existe um problema dual correspondente

$$\begin{array}{l} \text{Primo: maximizar } \sum_{j=1}^n c_j x_j \text{ sujeito a } \sum_{j=1}^n a_{ij} x_j \leq b_i, \quad x_j \geq 0 \\ \text{Dual: minimizar } \sum_{i=1}^m b_i y_i \text{ sujeito a } \sum_{i=1}^m a_{ij} y_i \geq c_j, \quad y_i \geq 0 \end{array}$$

n variáveis  
 m restrições  
 m variáveis  
 n restrições

Yendo x uma solução fazível do problema primo e y uma solução fazível do problema dual:

$$\sum_{j=1}^n c_j x_j \leq \sum_{i=1}^m b_i y_i$$

DUALIDADE FIRMA CA

Yendo x qualquira solução derivada pelo algoritmo Simplex, N e B os conjuntos de variáveis da forma Yslack final, c o vetor de coeficientes da forma Yslack final e  $y_i = -c_{m+i}$ ,  $m+i \in N$ :

$$\sum_{j=1}^n c_j x_j = \sum_{i=1}^m b_i y_i$$

x é a solução ótima para o problema primo  
 y é a solução ótima para o problema dual

Teorema Fundamental: qualquer problema na Forma Standard ou tem uma solução ótima com valor finito (que o algoritmo Simplex) retorna, ou é infinita ou é ilimitada (e o algoritmo Simplex identifica ambas)

Programação Linear Inteira (ILP): a região fejável é um conjunto de pontos discretos (limitada por uma região convexa) em que a solução ótima pode não ser um vértice — problema NP-completo

Como resolver? "Branch and Bound"

1. Resolver o problema como um problema LP pelo método Simplex
2. Tomar uma variável da solução e restringi-la aos inteiros permitidos imediatamente superior e inferior
3. Resolver os dois novos problemas pelo método Simplex
4. Repetir sucessivamente com outras variáveis

- Quando se alcança uma solução, alcança-se um limite (superior/inferior) do problema — fronteira
- A estrutura de soluções é uma árvore binária em que cada nó representa um problema LP para resolver pelo método Simplex

DFS: toma o ramo de lado esquerdo até à árvore poder ser "jodada", depois fazer "backtracking" até ao último nó em que pode tomar outro ramo (para a direita) e depois continua com ramo esquerdo

BFS: avalia os nós por nível crescente ( $2 \rightarrow 3 \rightarrow \dots$ )

If we express the linear program below, in standard form describe the various elements of this matrix formulation, respectively,  $n$ ,  $m$ ,  $A$ ,  $b$  and  $c$ ? Provide two feasible solutions to this problem and indicate the value of the objective function for each one.

Objective function:

$$\max 2x_1 - 3x_2 + 3x_3$$

subject to:

$$\begin{array}{rcl} x_1 & x_2 & -x_3 \leq 7 \\ -x_1 & -x_2 & +x_3 \leq -7 \\ -x_1 & -2x_2 & +2x_3 \leq 4 \end{array}$$

and

$$x_1, x_2, x_3 \geq 0$$

Convert the LP formulation in Exercise 1 in Slack Form and derive  $N$ ,  $B$ ,  $A$ ,  $b$ ,  $c$  and  $v$ .

Convert the following linear program into Standard form.

$$\min 2x_1 + 7x_2$$

subject to:

$$x_1 = 7$$

$$3x_1 + x_2 \geq 24$$

$$x_2 \geq 0$$

$$x_3 \leq 0$$

$$x_1, x_2, x_3 \geq 0$$

Convert the following linear program into slack form

$$\max 2x_1 - 6x_3$$

subject to:

$$x_1 + x_2 - x_3 \leq 7$$

$$3x_1 - x_2 \geq 8$$

$$-x_1 + 2x_2 + 2x_3 \geq 0$$

$$x_1, x_2, x_3 \geq 0$$

$$\max 2x_1 - 6x_3$$

$$z = 2x_1 - 6x_3$$

$$x_1 + x_2 - x_3 \leq 7$$

$$x_4 = 7 - x_1 - x_2 + x_3$$

$$-3x_1 + x_2 \leq -8$$

$$x_5 = -8 + 3x_1 - x_2$$

$$x_1 - 2x_2 - 2x_3 \leq 0$$

$$x_6 = -x_1 + 2x_2 + 2x_3$$

$$x_1, x_2, x_3 \geq 0$$

$$x_1, x_2, x_3, x_4, x_5, x_6 \geq 0$$

Show that the following linear program is infeasible.

$$\max 3x_1 - 2x_2$$

subject to:

$$\begin{array}{lcl} x_1 & + x_2 & \leq 2 \\ -2x_1 & - 2x_2 & \leq -10 \end{array}$$

$$x_1, x_2 \geq 0$$

$$(x_1, x_2, x_3, x_4)$$

$$y = 3x_1 - 2x_2$$

$$(0, 0, 2, -10)$$

$$x_3 = 2 - x_1 - x_2$$

$$\begin{aligned} -b/c &= -2/-1 = 2 & y &= -x_0 \\ -b/c &= 10/-1 = -10 & x_3 &= 2 - x_1 - x_2 + x_0 \\ x_4 &= -10 + 2x_1 + 2x_2 & x_4 &= -10 + 2x_1 + 2x_2 + x_0 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned}$$

$$x_1, x_2, x_3, x_4 \geq 0$$

$$x_0 = 10 - 2x_1 - 2x_2 + x_4$$

$$y = -10 + 2x_1 + 2x_2 - x_4$$

$$\leftarrow -b/c = -10/-3 = 4$$

$$x_3 = 12 - 3x_1 - 3x_2 + x_4$$

$$-b/c = -10/-2 = 5$$

$$x_0 = 10 - 2x_1 - 2x_2 + x_4$$

$$x_0, x_1, x_2, x_3, x_4 \geq 0$$

$$x_1 = 4 - x_2 - \frac{x_3}{3} + \frac{x_4}{3}$$

$$\begin{aligned} \rightarrow y &= -2 - \frac{2x_3}{3} - \frac{x_4}{3} \\ x_1 &= 4 - x_2 - \frac{x_3}{3} + \frac{x_4}{3} \\ x_0 &= 2 + \frac{2x_3}{3} + \frac{x_4}{3} \end{aligned}$$

$$\left( \frac{2}{0}, 1, 2, 3, 4 \right)$$

Show that the following linear program is unbounded.

$$\max x_1 - x_2$$

STANDARD RD

subject to:

$$\begin{array}{rcl} -2x_1 + x_2 & \leq & -1 \\ -x_1 - 2x_2 & \leq & -2 \end{array}$$

$$x_1, x_2 \geq 0$$

$$(x_1, x_2, x_3, x_4)$$

$$y = x_1 - x_2$$

$$(0, 0, -1, -2)$$

$$x_3 = -1 + 2x_1 - x_2$$

$$x_4 = -2 + x_1 + 2x_2$$

$$x_1, x_2, x_3, x_4 \geq 0$$

$$-\frac{b}{c} = \frac{1}{-1} = -1$$

$$-\frac{b}{c} = \frac{2}{-1} = -2$$

$$y = -x_0$$

$$x_3 = -1 + 2x_1 - x_2 + x_0$$

$$x_4 = -2 + x_1 + 2x_2 + x_0$$

$$(x_0, x_1, x_2, x_3, x_4) \geq 0$$

$$x_0 = 2 - x_1 - 2x_2 + x_4$$

...

Given the LP problem below, already in Standard Form, determine its feasible region and show the various pivot steps of the use of the Simplex algorithm to find the optimal solution, should it exist. At each pivot step, explain the decision of the choice of variables.

maximize:  $z = 4x_1 + x_2$

subject to:  $x_2 \leq 6$

$$x_1 + x_2 \leq 8$$

$$x_1 \leq 4$$

$$x_1 - x_2 \leq 2$$

$$x_1, x_2 \geq 0$$

Suppose that we have a general linear program with  $n$  variables and  $m$  constraints, and suppose that we convert it to the standard form. Give an upper bound on the number of variables and constraints in the resulting linear program. What about a conversion to the slack form, does the bound change?

Give an example of a liner program for which the feasible region is not bounded but the optimal objective value is finite. Can you also find a linear program that has a feasible region that is unbounded but has an infinite number of solutions with a finite value?

Solve the following linear program using the Simplex algorithm:

$$\max 18x_1 + 12.5x_2$$

subject to:

$$x_1 + x_2 \leq 20$$

$$x_1 \leq 12$$

$$x_2 \leq 16$$

$$x_1, x_2 \geq 0$$

Solve the following linear program using the SIMPLEX algorithm:

$$\max -5x_1 - 3x_2$$

subject to:

$$x_1 - x_2 \leq 1$$

$$2x_1 + x_2 \leq 2$$

$$x_1, x_2 \geq 0$$

Consider the LP problem below. Show that it has an infeasible initial solution and set up a feasible solution to the corresponding auxiliary problem that once solved would provide the initial solution to this original LP problem. Do not solve the original LP problem.

$$\min z = x_1 + x_2 + x_3$$

- subject to:

$$2x_1 + 7.5x_2 + 3x_3 \geq 10$$

$$20x_1 + 5x_2 + 10x_3 \geq 30$$

$$x_1, x_2, x_3 \geq 0$$

# Programação Linear Inteira

• A região fejável é um conjunto de pontos discretos limitados por uma região convexa — a solução ótima pode não ser um canto

• ILP é NP-Completo

(1) ILP é NP

(2) ILP reduz-se a 3CNF-SAT

Usar LP para resolver ILP? Se a solução não for inteira, abandonar

“Cutting-Plane”: restringir a região fejável sem excluir pontos inteiros, adicionando restrições lineares de cada vez até a solução ser inteira (usando o algoritmo de programação linear)

“Branch-and-Bound”: tomar uma das variáveis da solução LP e restringi-la ao inteiro permitido abaixo e acima do seu valor

“Fathoming”: avaliar nós numa dada ordem pode facilitar

Bounding: limite inferior já encontrado

Integralidade: solução inteira encontra prior ao que o limite

Infeasibilidade: região fejável vazia