Algoritmos em Grafos: Busca em Profundidade

Letícia Rodrigues Bueno

UFABC

Problema 1: Ordenação Topológica

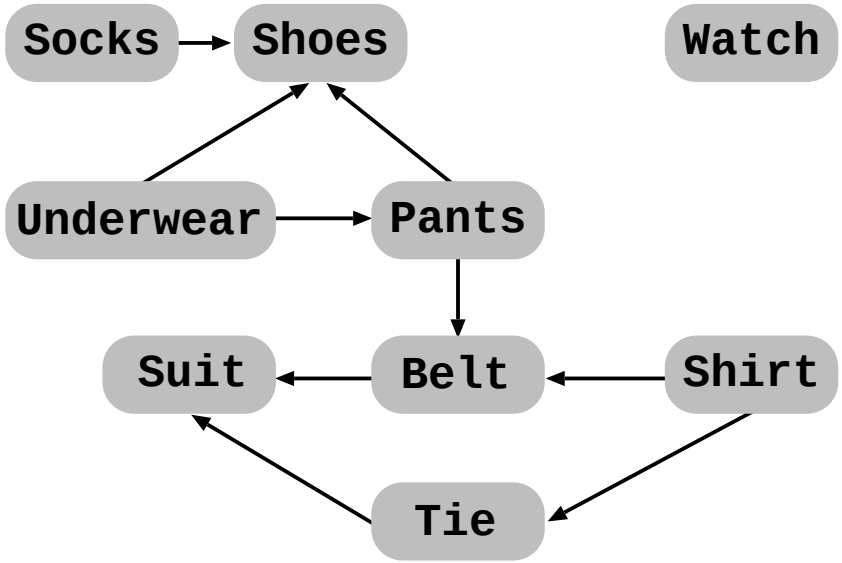
- **grafos direcionados acíclicos:** usados para indicar precedências entre eventos;

Problema 1: Ordenação Topológica

- **grafos direcionados acíclicos:** usados para indicar precedências entre eventos;
- **ordenação topológica:** ordenação linear tal que arestas orientadas sigam da esquerda para direita (pode haver várias);

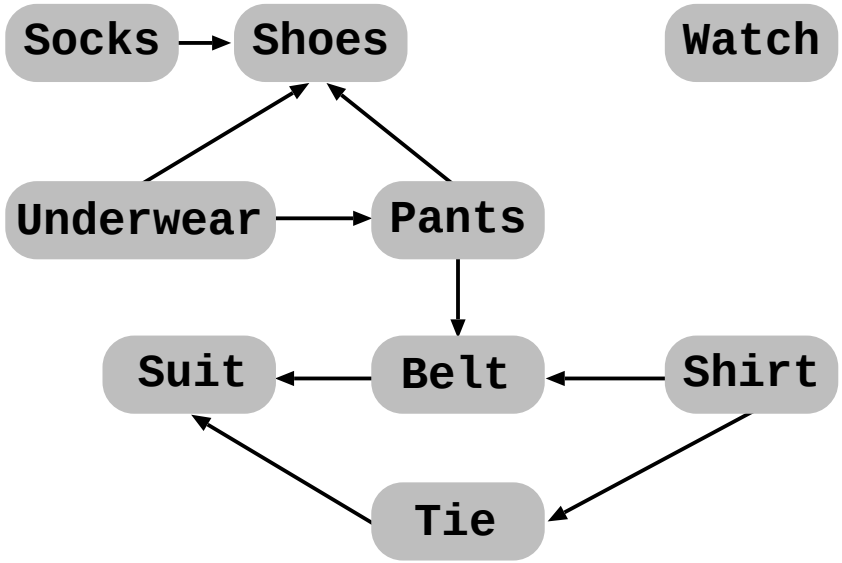
Problema 1: Ordenação Topológica

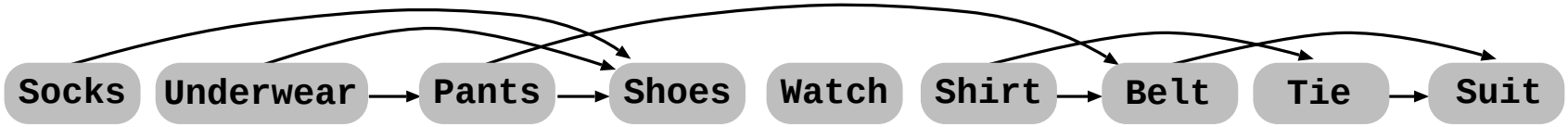
- **grafos direcionados acíclicos:** usados para indicar precedências entre eventos;
- **ordenação topológica:** ordenação linear tal que arestas orientadas sigam da esquerda para direita (pode haver várias);



Problema 1: Ordenação Topológica

- **grafos direcionados acíclicos:** usados para indicar precedências entre eventos;
- **ordenação topológica:** ordenação linear tal que arestas orientadas sigam da esquerda para direita (pode haver várias);





Busca em Profundidade (DFS - *Depth-First Search*)

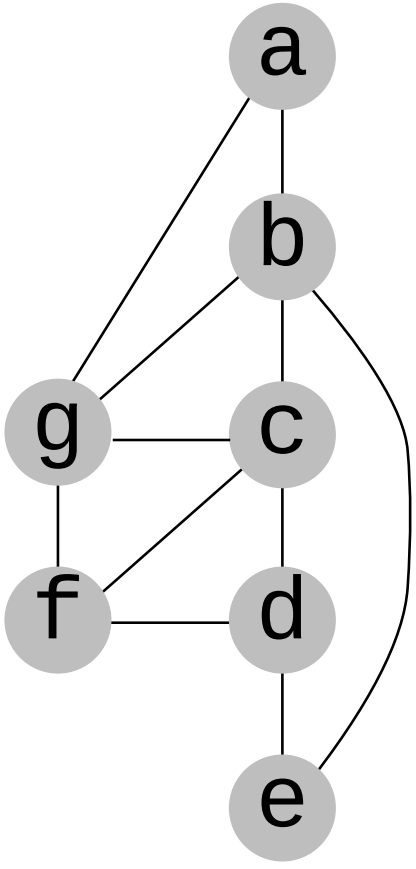
Como encontrar uma ordenação topológica em um grafo direcionado acíclico?

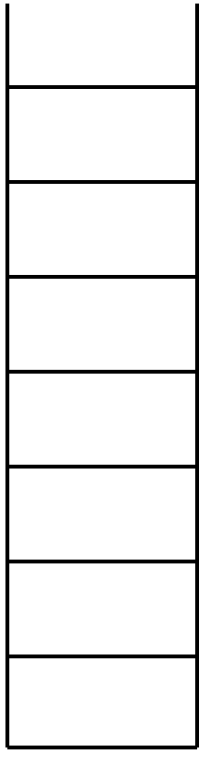
Busca em Profundidade (DFS - *Depth-First Search*)

Como encontrar uma ordenação topológica em um grafo direcionado acíclico?

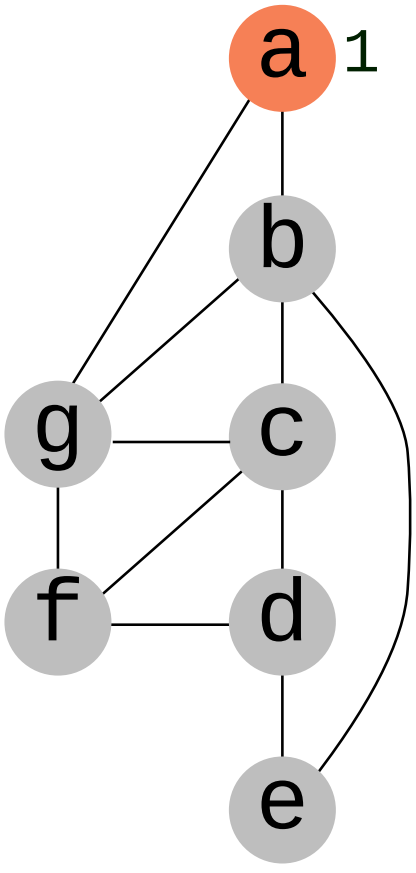
Usaremos a busca em profundidade!

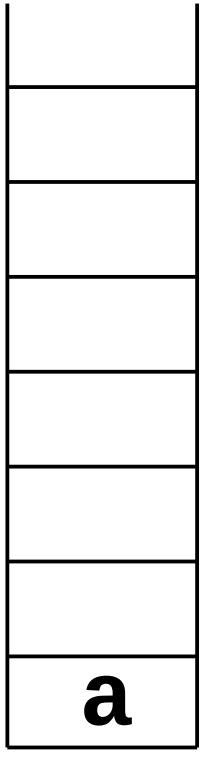
Busca em Profundidade (DFS - *Depth-First Search*)



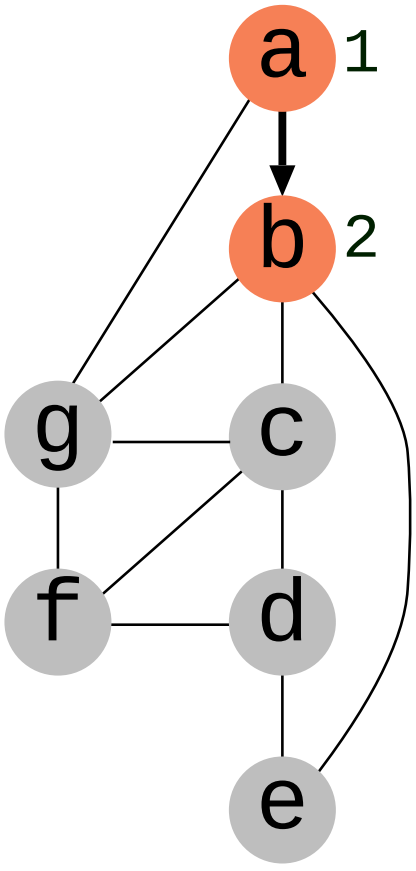


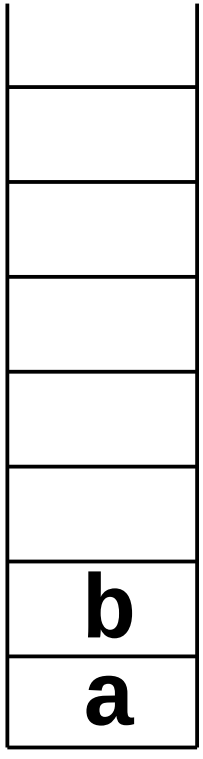
Busca em Profundidade (DFS - *Depth-First Search*)



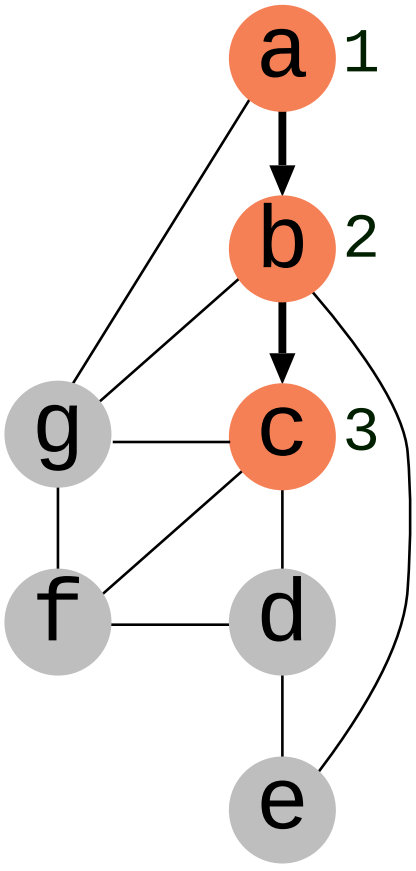


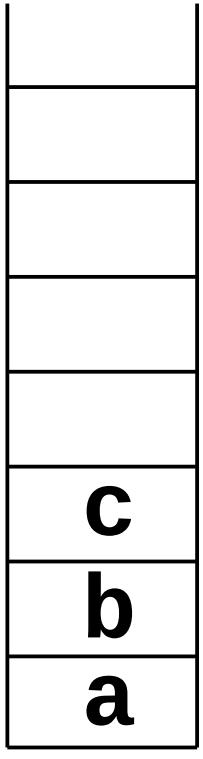
Busca em Profundidade (DFS - *Depth-First Search*)



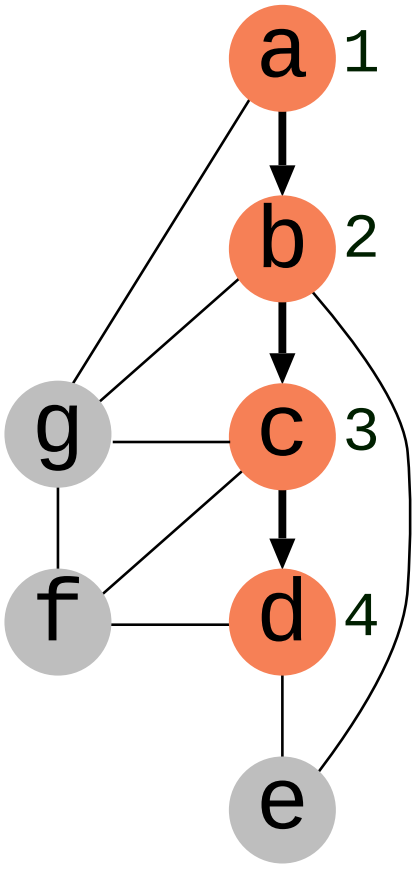


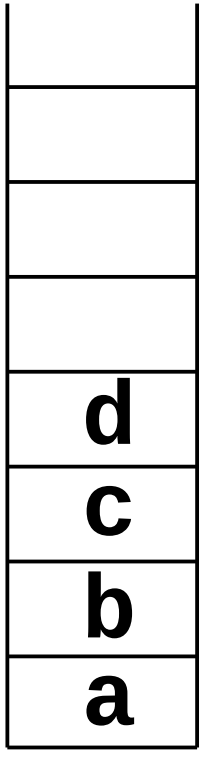
Busca em Profundidade (DFS - *Depth-First Search*)



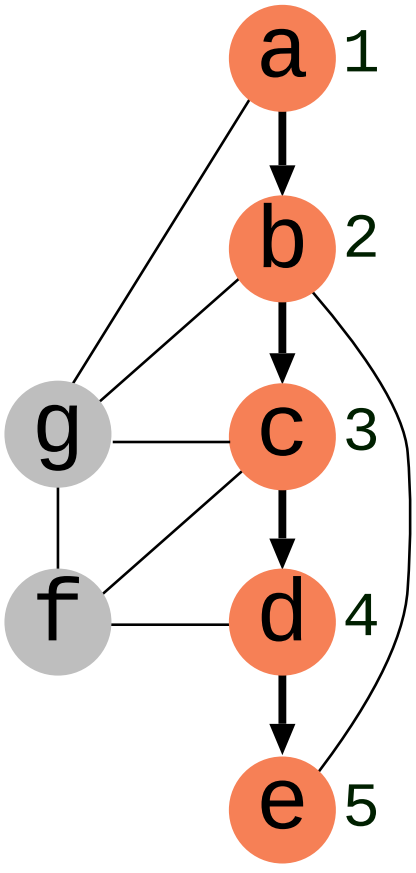


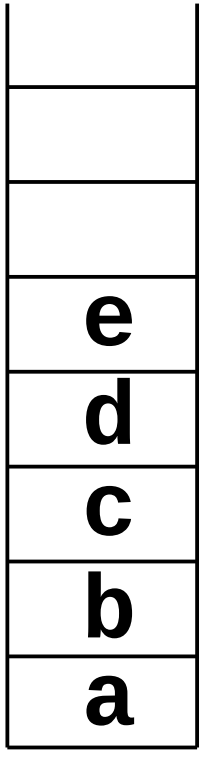
Busca em Profundidade (DFS - *Depth-First Search*)



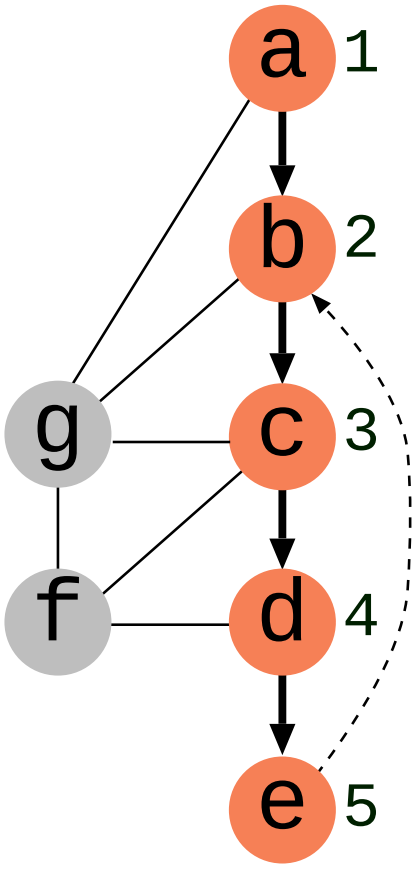


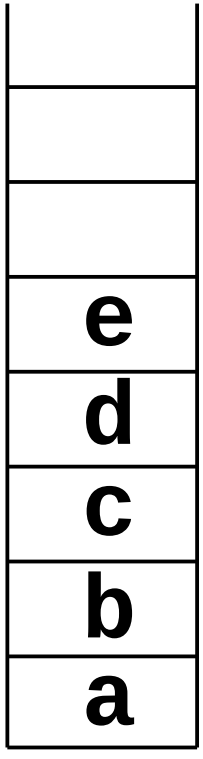
Busca em Profundidade (DFS - *Depth-First Search*)



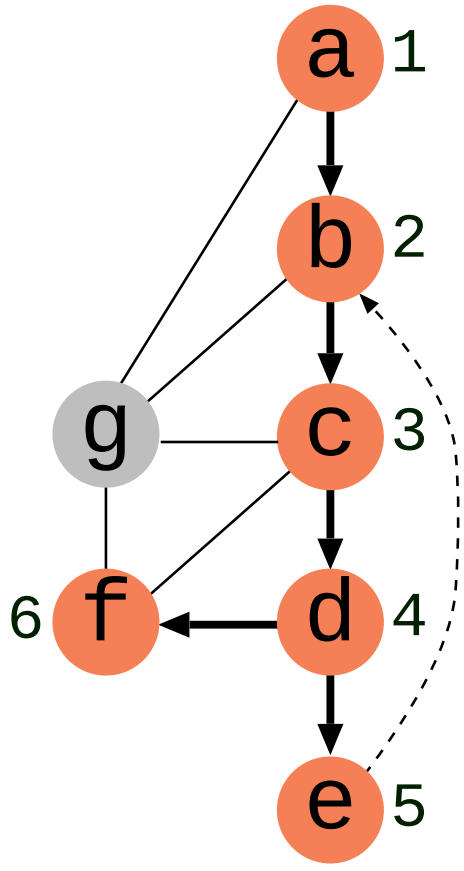


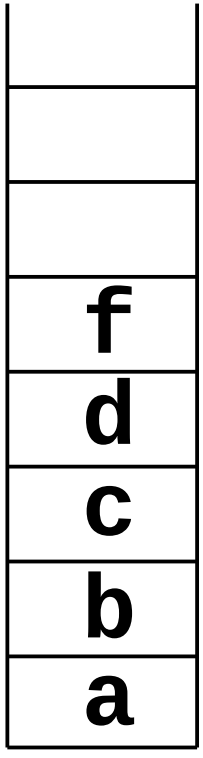
Busca em Profundidade (DFS - *Depth-First Search*)



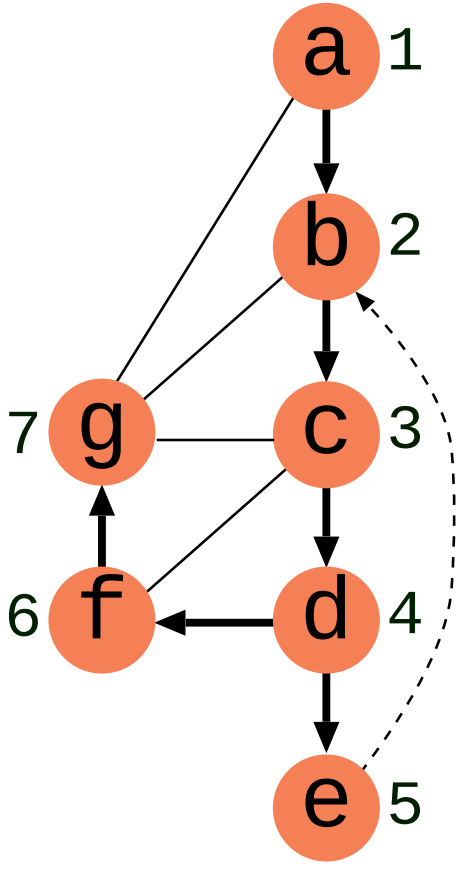


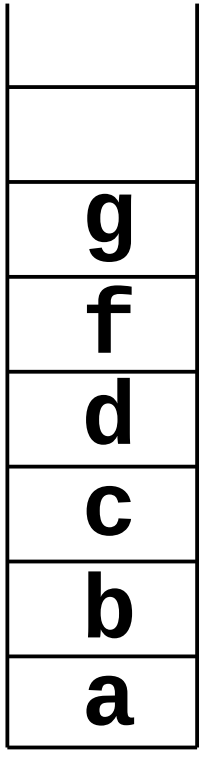
Busca em Profundidade (DFS - *Depth-First Search*)



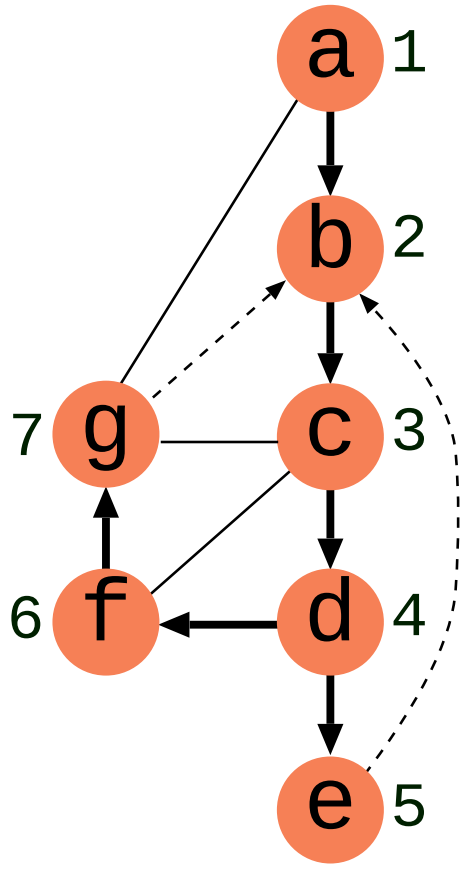


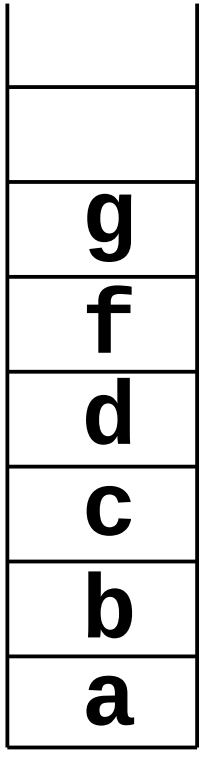
Busca em Profundidade (DFS - *Depth-First Search*)



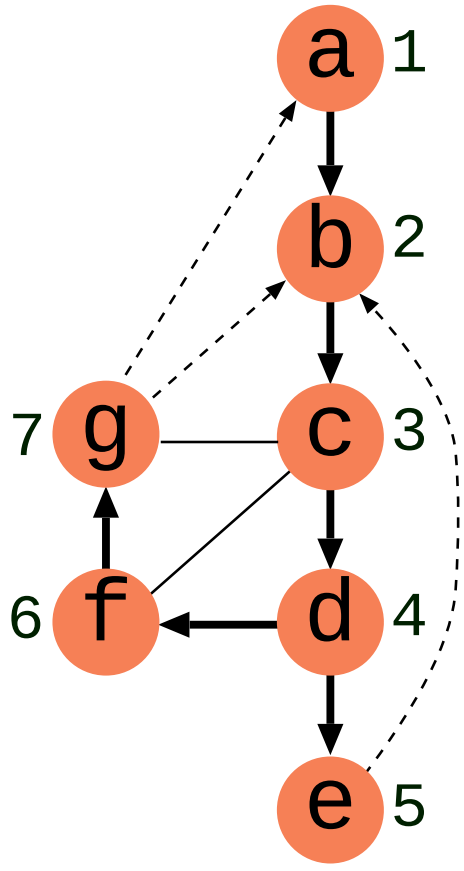


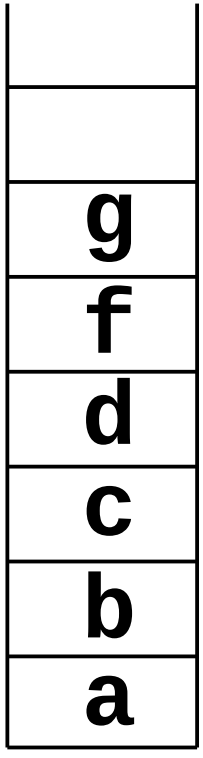
Busca em Profundidade (DFS - *Depth-First Search*)



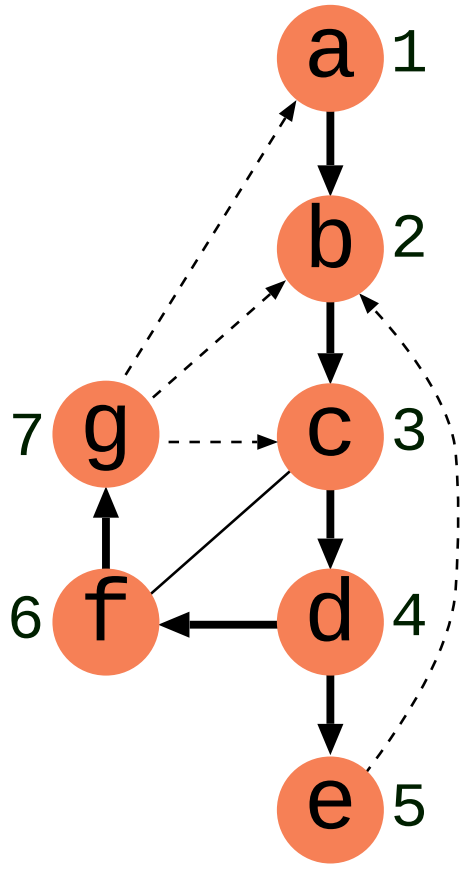


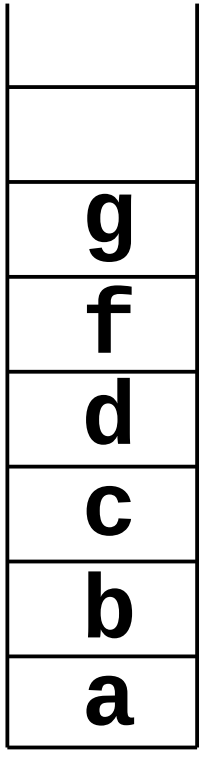
Busca em Profundidade (DFS - *Depth-First Search*)



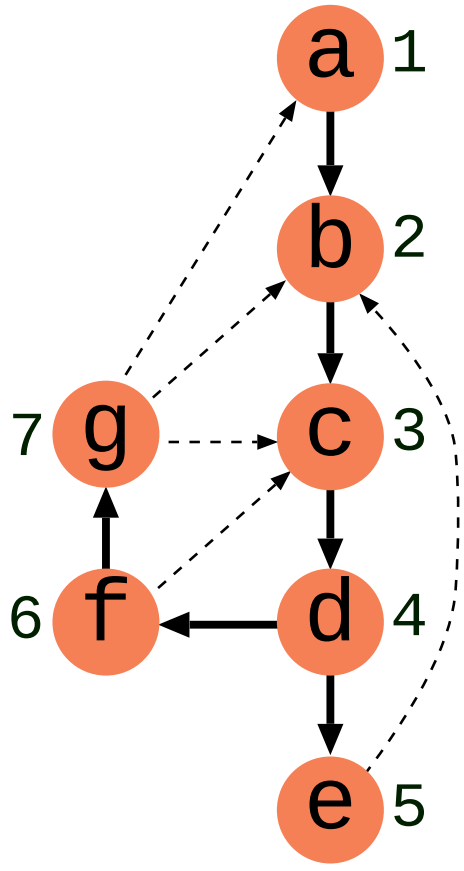


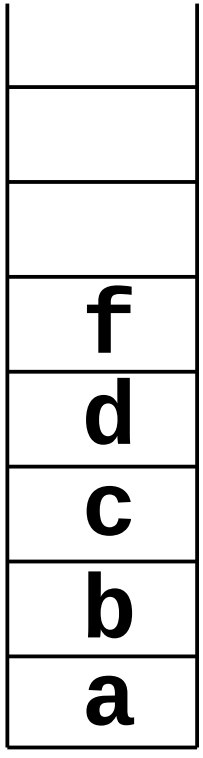
Busca em Profundidade (DFS - *Depth-First Search*)



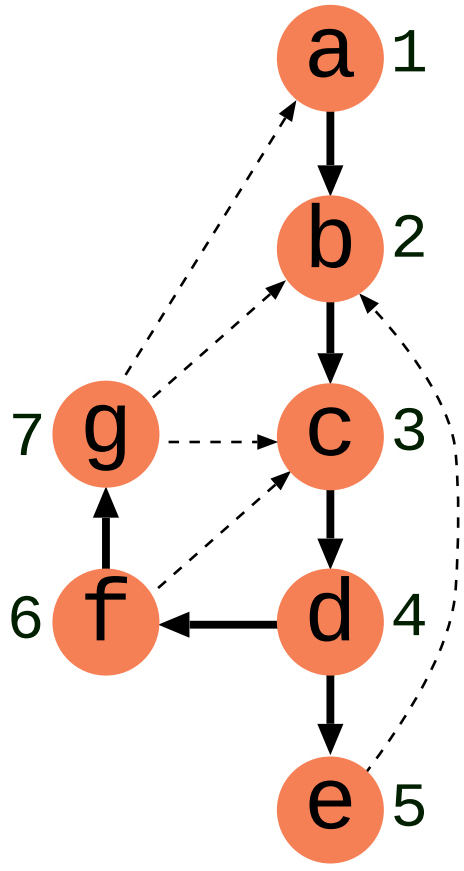


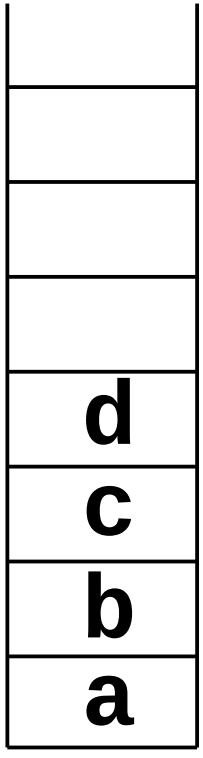
Busca em Profundidade (DFS - *Depth-First Search*)



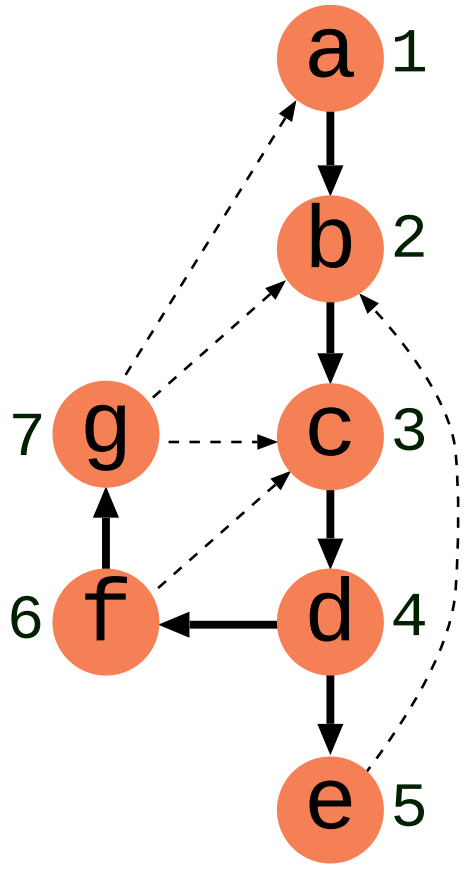


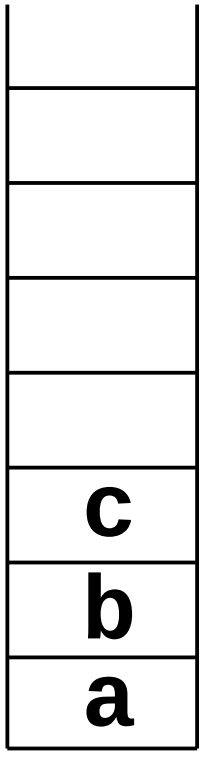
Busca em Profundidade (DFS - *Depth-First Search*)



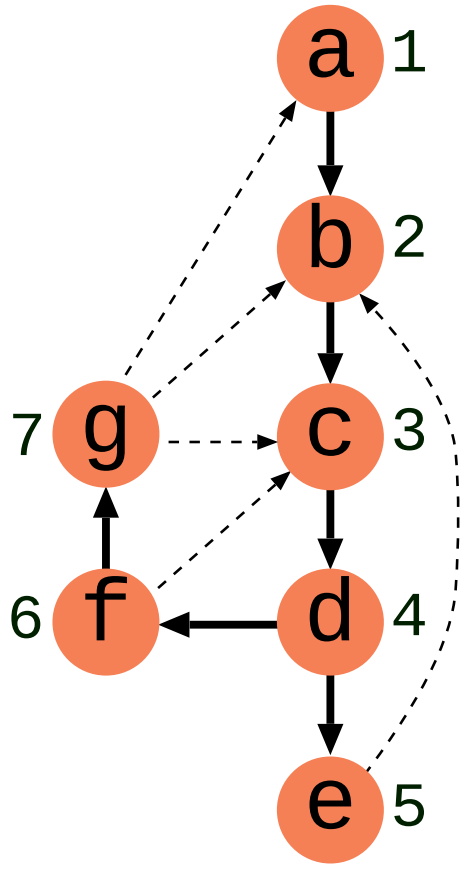


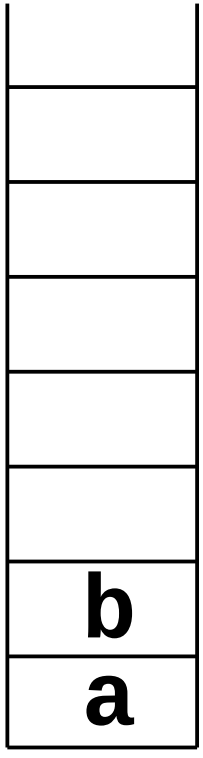
Busca em Profundidade (DFS - *Depth-First Search*)



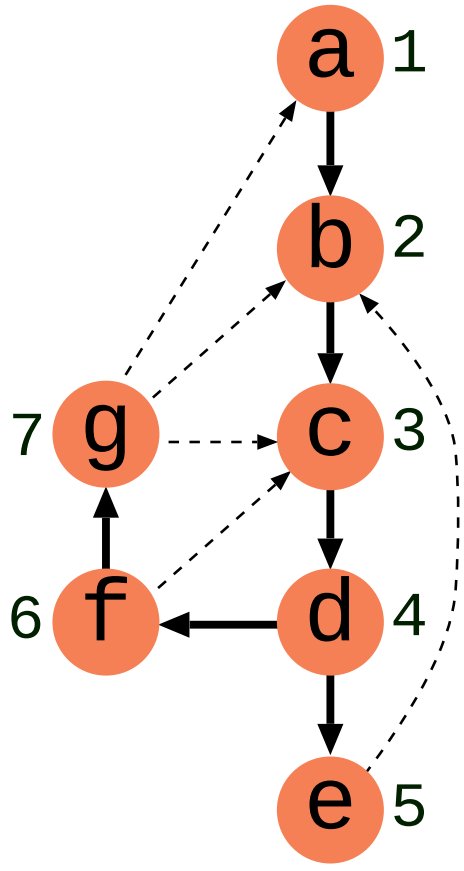


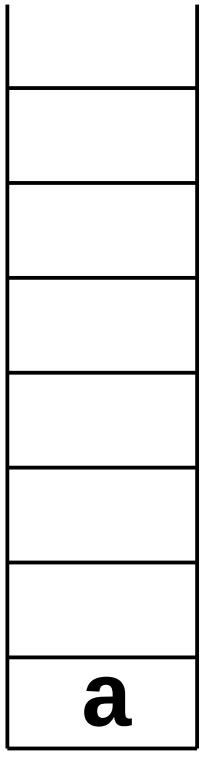
Busca em Profundidade (DFS - *Depth-First Search*)



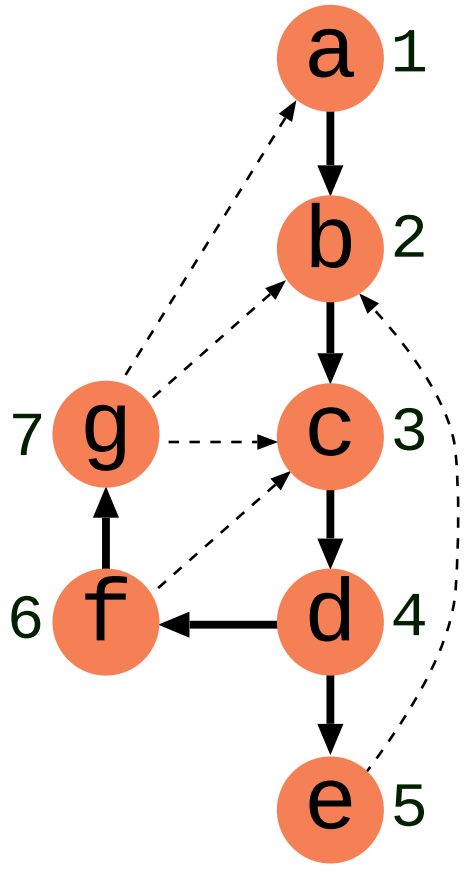


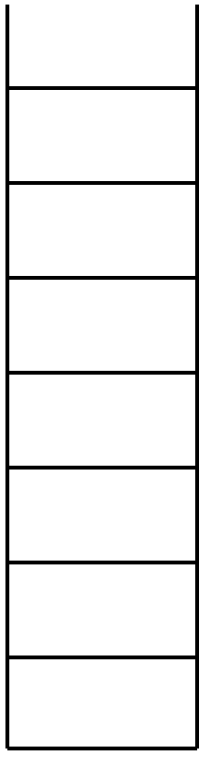
Busca em Profundidade (DFS - *Depth-First Search*)





Busca em Profundidade (DFS - *Depth-First Search*)





Algoritmo DFS

```
1  dfs(G, u, cont):
2      u.visitado = True
3      u.d = cont
4      para v em adj(u) faça
5          se não v.visitado então
6              v.p = u
7              dfs(G, v, cont+1)


8  para u em V(G) faça
9      u.visitado = False
10     u.d = ∞
11     u.p = None
12 cont = 1
13 dfs(G, u, cont)
```

Algoritmo DFS

```
1  dfs(G, u, cont):
2      u.visitado = True
3      u.d = cont
4      para v em adj(u) faça
5          se não v.visitado então
6              v.p = u
7              dfs(G, v, cont+1)


8  para u em V(G) faça
9      u.visitado = False
10     u.d = ∞
11     u.p = None
12 cont = 1
13 dfs(G, u, cont)
```

Análise da complexidade:

Algoritmo DFS

```
1  dfs(G, u, cont):
2      u.visitado = True
3      u.d = cont
4      para v em adj(u) faça
5          se não v.visitado então
6              v.p = u
7              dfs(G, v, cont+1)

8  para u em V(G) faça
9      u.visitado = False
10     u.d = ∞
11     u.p = None
12 cont = 1
13 dfs(G, u, cont)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 7 e 13): $O(n)$

Algoritmo DFS

```
1  dfs(G, u, cont):
2      u.visitado = True
3      u.d = cont
4      para v em adj(u) faça
5          se não v.visitado então
6              v.p = u
7              dfs(G, v, cont+1)

8  para u em V(G) faça
9      u.visitado = False
10     u.d = ∞
11     u.p = None
12 cont = 1
13 dfs(G, u, cont)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 7 e 13): $O(n)$
- A lista de adjacência de cada vértice percorrida uma vez (linha 4): $O(m)$

Algoritmo DFS

```
1  dfs(G, u, cont):
2      u.visitado = True
3      u.d = cont
4      para v em adj(u) faça
5          se não v.visitado então
6              v.p = u
7              dfs(G, v, cont+1)


8  para u em V(G) faça
9      u.visitado = False
10     u.d = ∞
11     u.p = None
12 cont = 1
13 dfs(G, u, cont)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 7 e 13): $O(n)$
- A lista de adjacência de cada vértice percorrida uma vez (linha 4): $O(m)$
- Complexidade total: $O(n+m)$

Ordenação Topológica

Voltando ao problema de ordenação topológica:

Ordenação Topológica

Voltando ao problema de ordenação topológica:

Como usar DFS para resolvê-lo?

Ordenação Topológica

Voltando ao problema de ordenação topológica:

Como usar DFS para resolvê-lo?

Lema

Um grafo direcionado acíclico sempre contém um vértice de grau de entrada 0*.*

Ordenação Topológica

Voltando ao problema de ordenação topológica:

Como usar DFS para resolvê-lo?

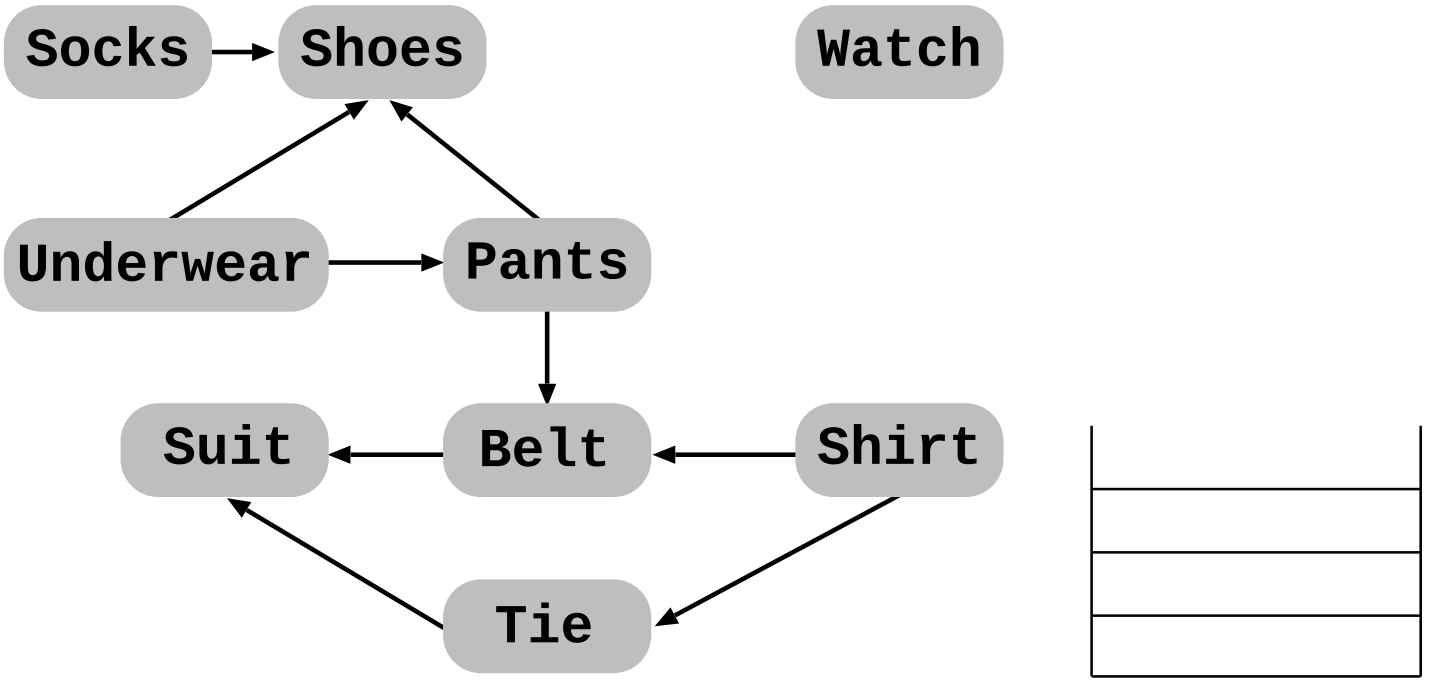
Lema

Um grafo direcionado acíclico sempre contém um vértice de grau de entrada 0*.*

Demonstração.

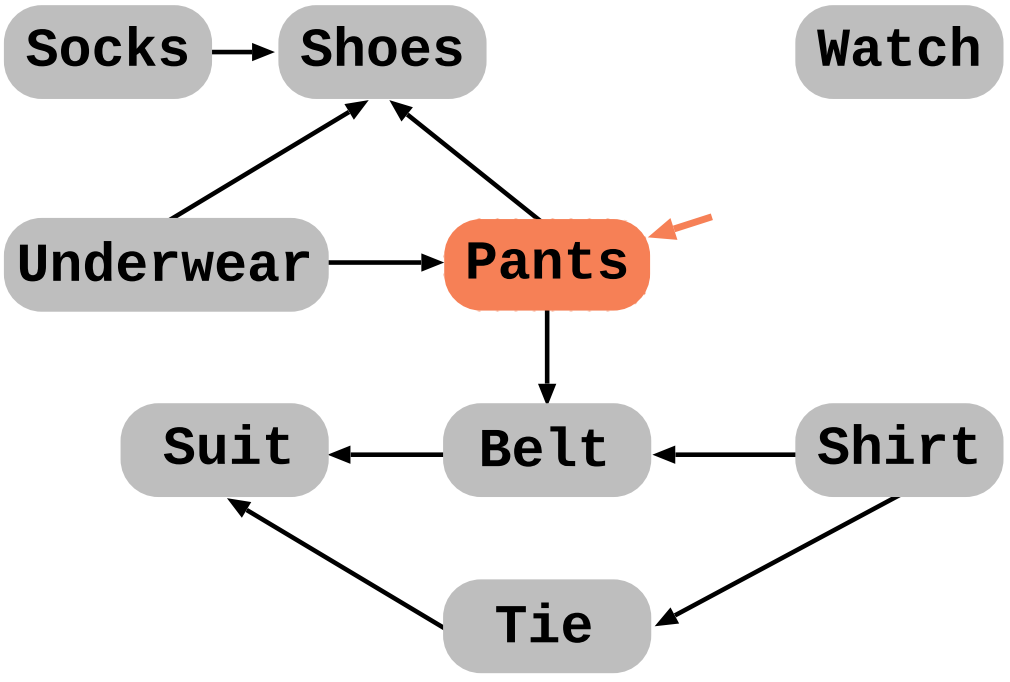
Se todo vértice tem grau de entrada maior que 0, podemos andar pelas arestas de trás para frente sem parar. Há uma quantidade finita de vértices, portanto fazemos isso em um ciclo, mas grafo acíclico não tem ciclos. □

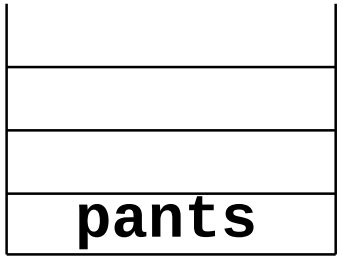
Busca em Profundidade (DFS - *Depth-First Search*)



pt → λ

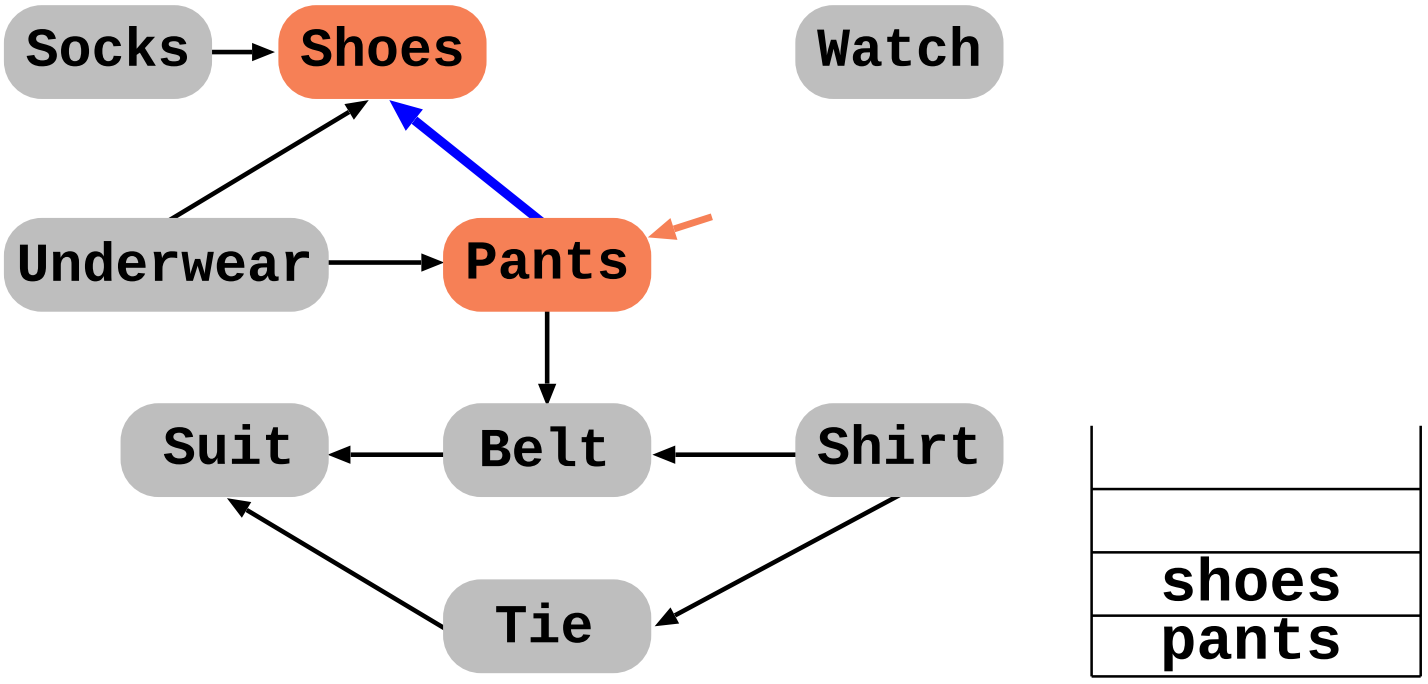
Busca em Profundidade (DFS - *Depth-First Search*)





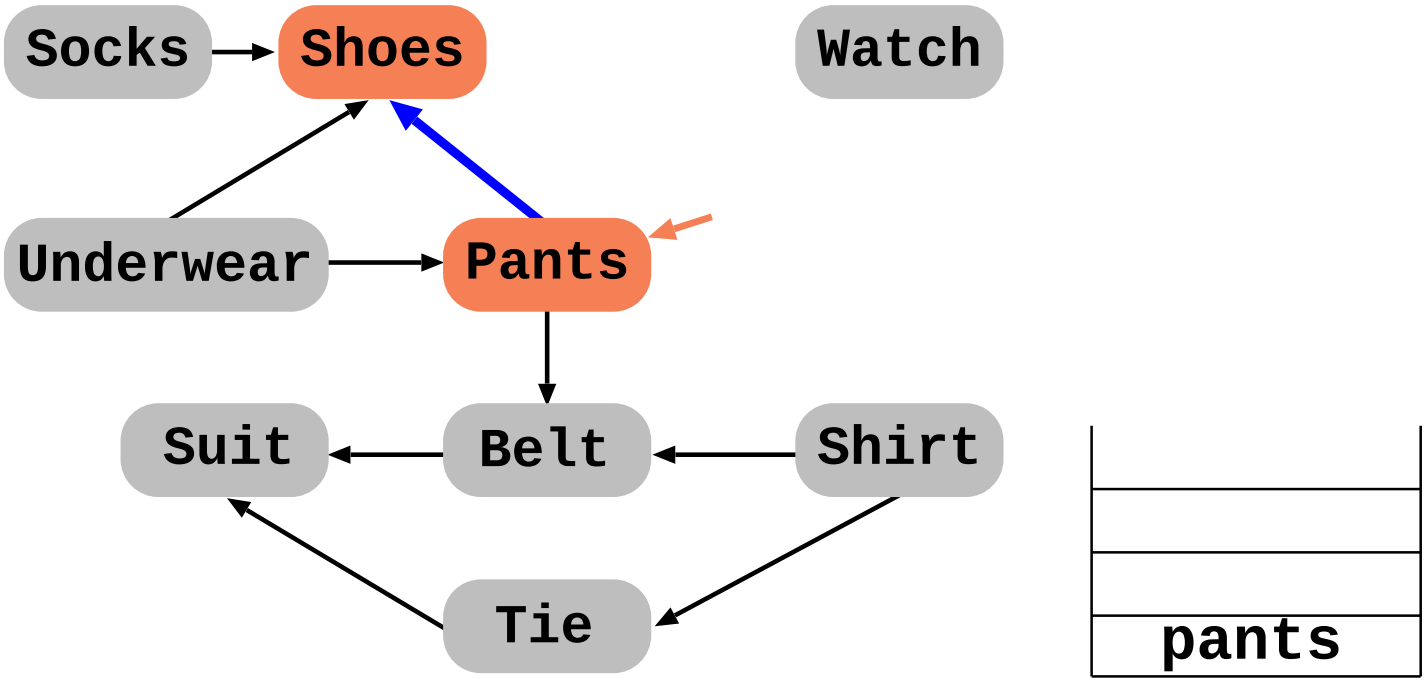
pt → λ

Busca em Profundidade (DFS - *Depth-First Search*)

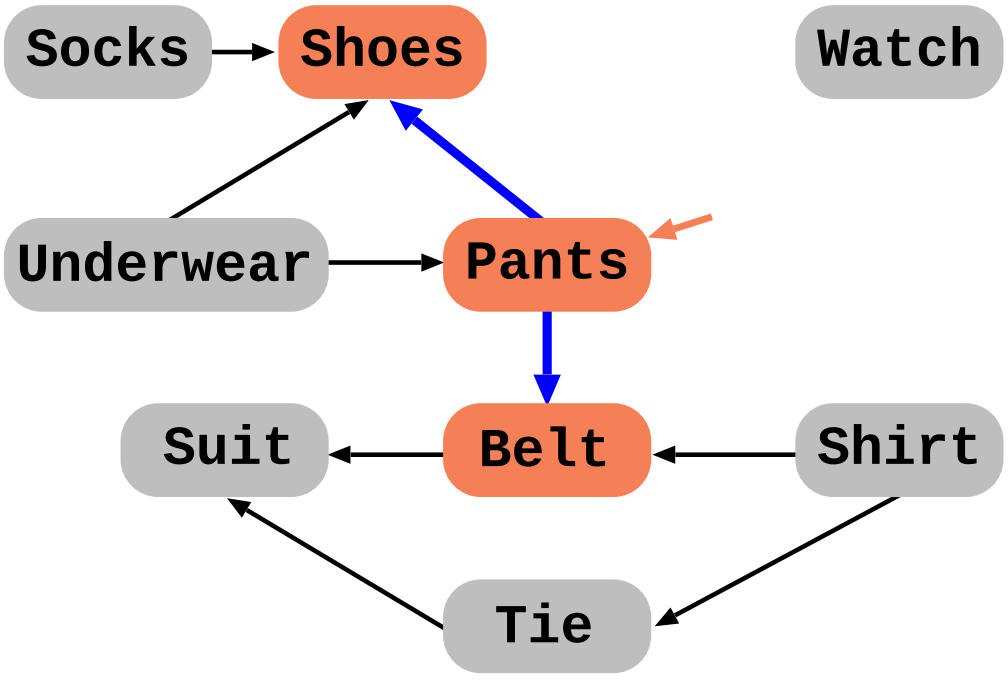


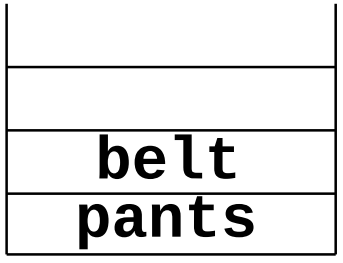
pt → λ

Busca em Profundidade (DFS - *Depth-First Search*)

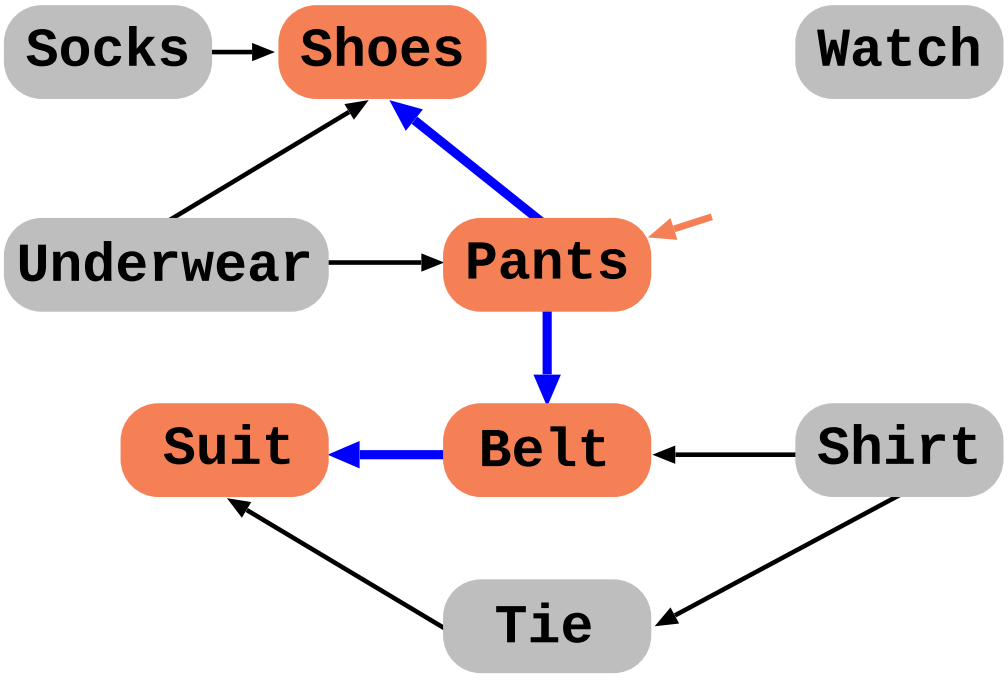


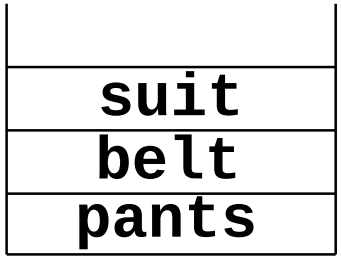
Busca em Profundidade (DFS - *Depth-First Search*)



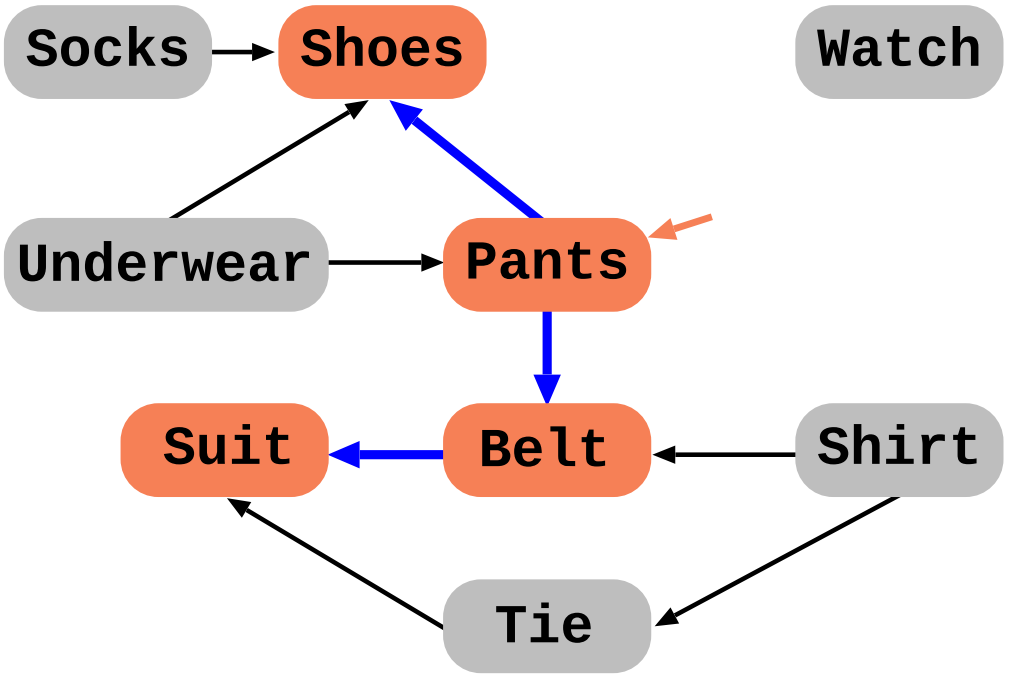


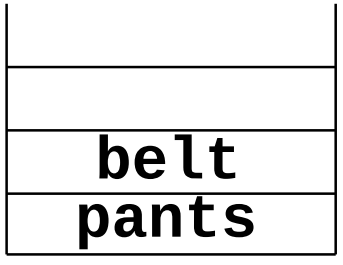
Busca em Profundidade (DFS - *Depth-First Search*)

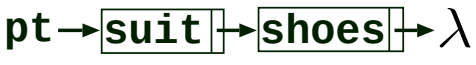




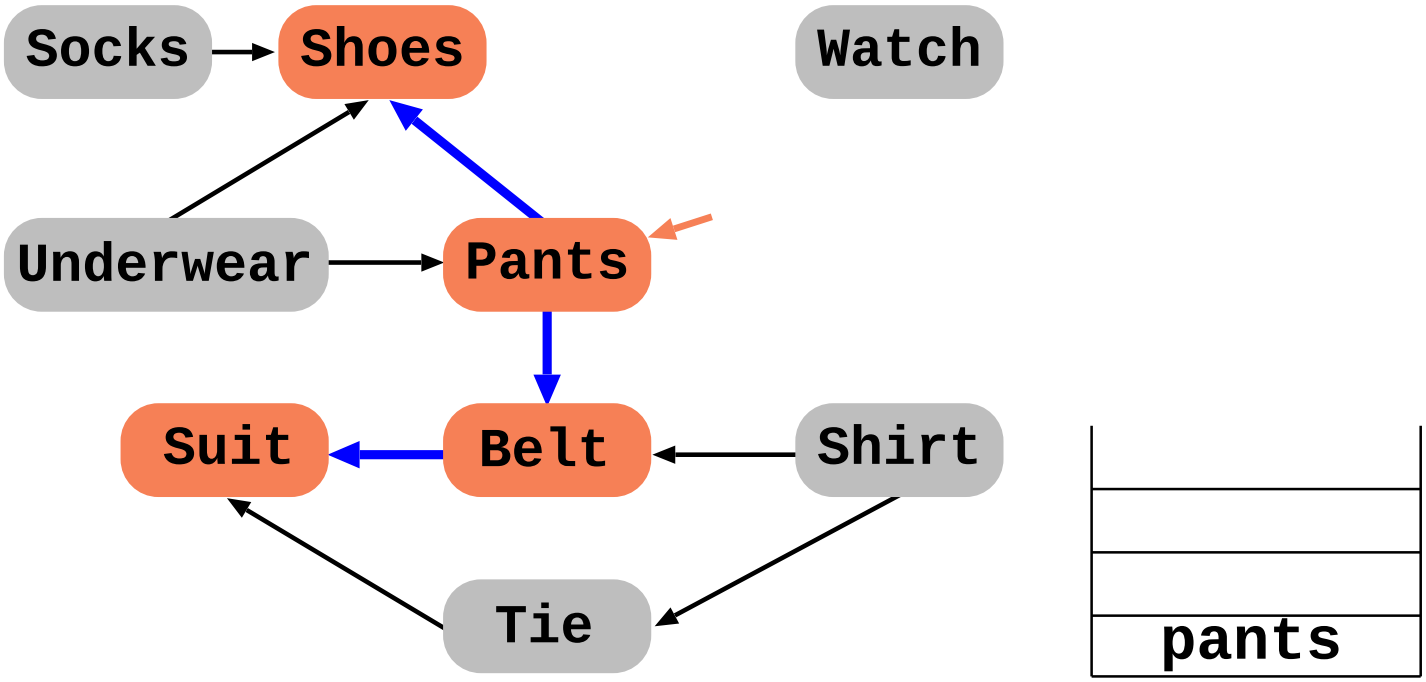
Busca em Profundidade (DFS - *Depth-First Search*)



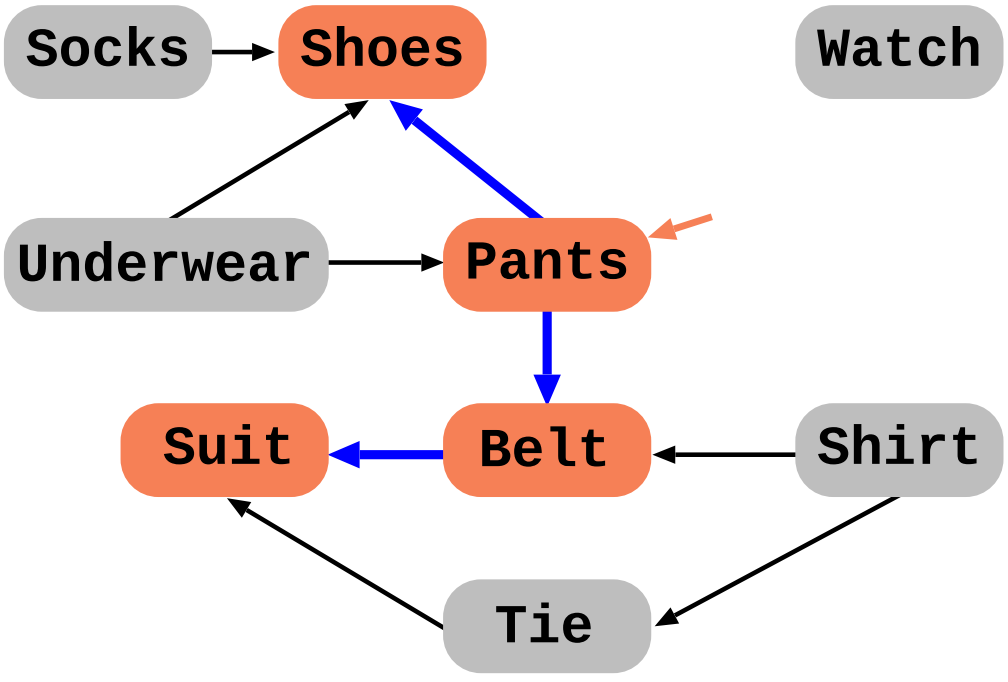


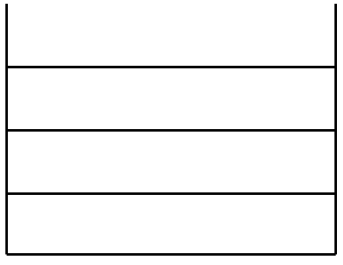


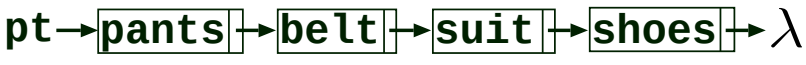
Busca em Profundidade (DFS - *Depth-First Search*)



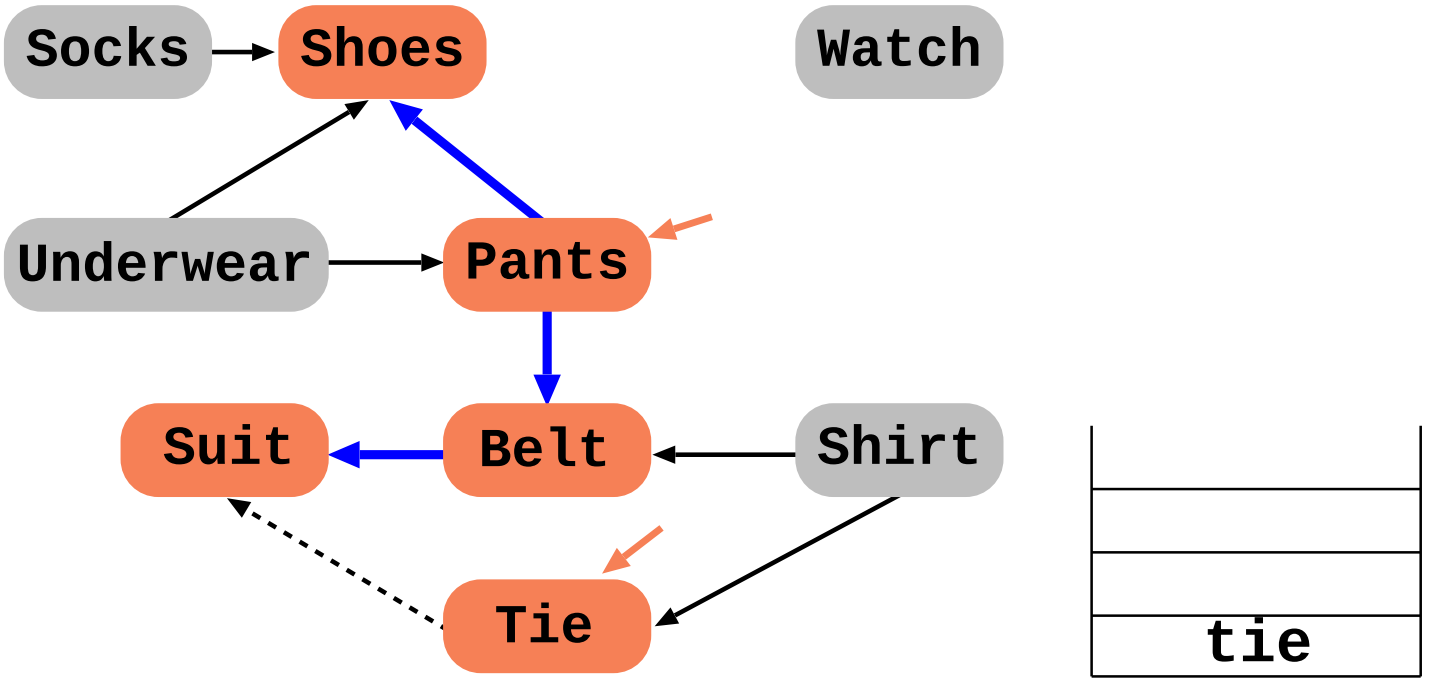
Busca em Profundidade (DFS - *Depth-First Search*)

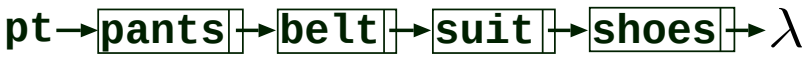




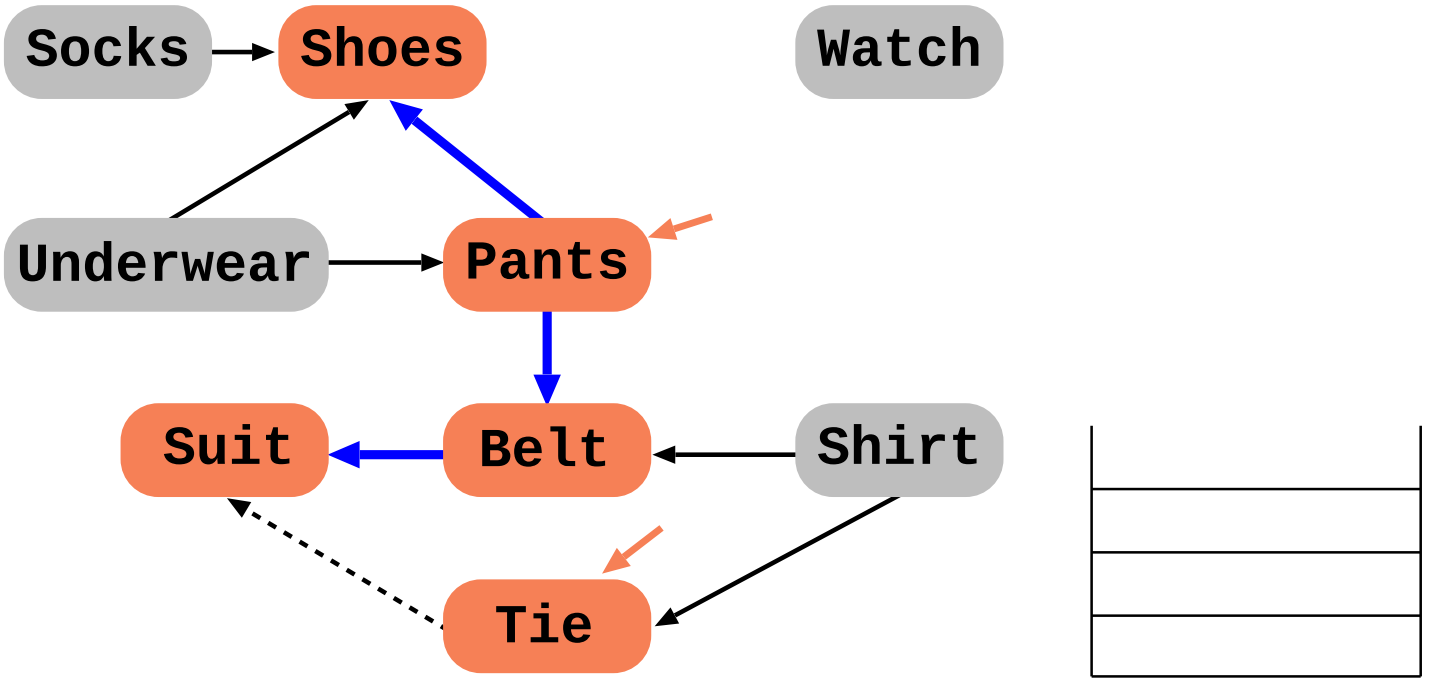


Busca em Profundidade (DFS - *Depth-First Search*)

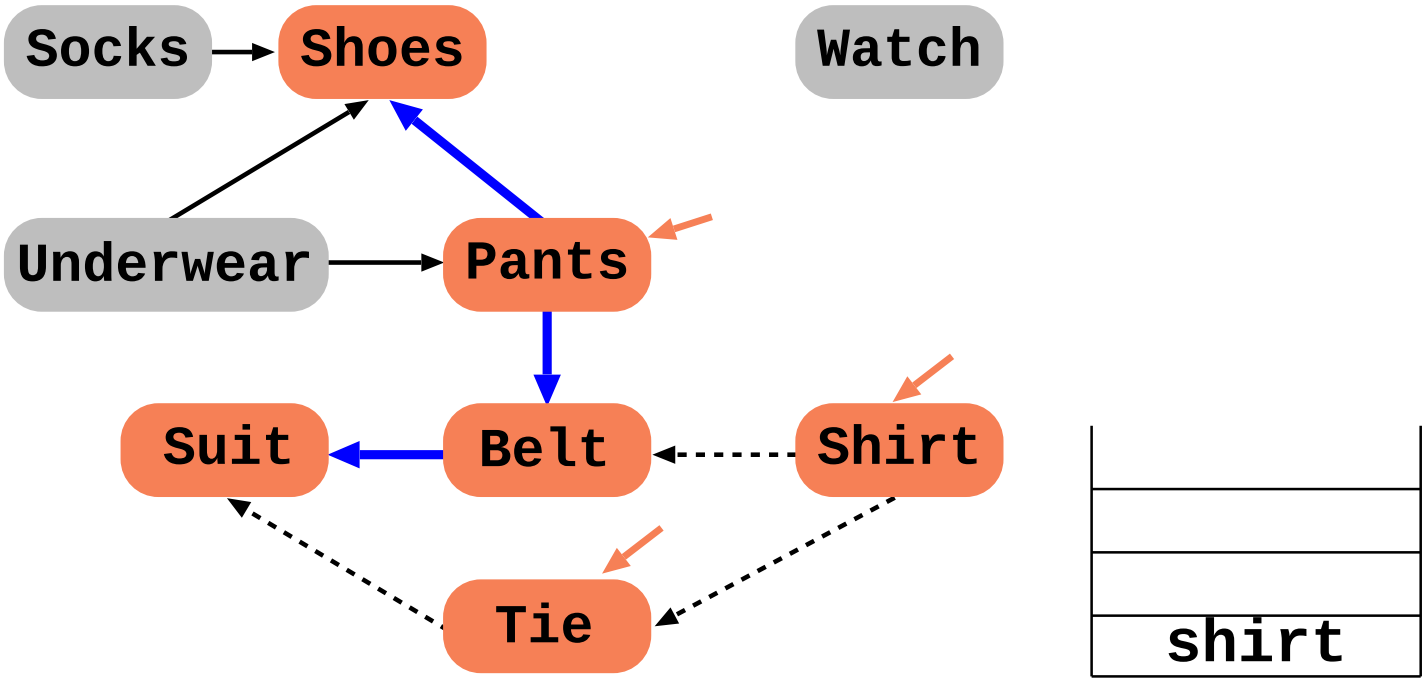




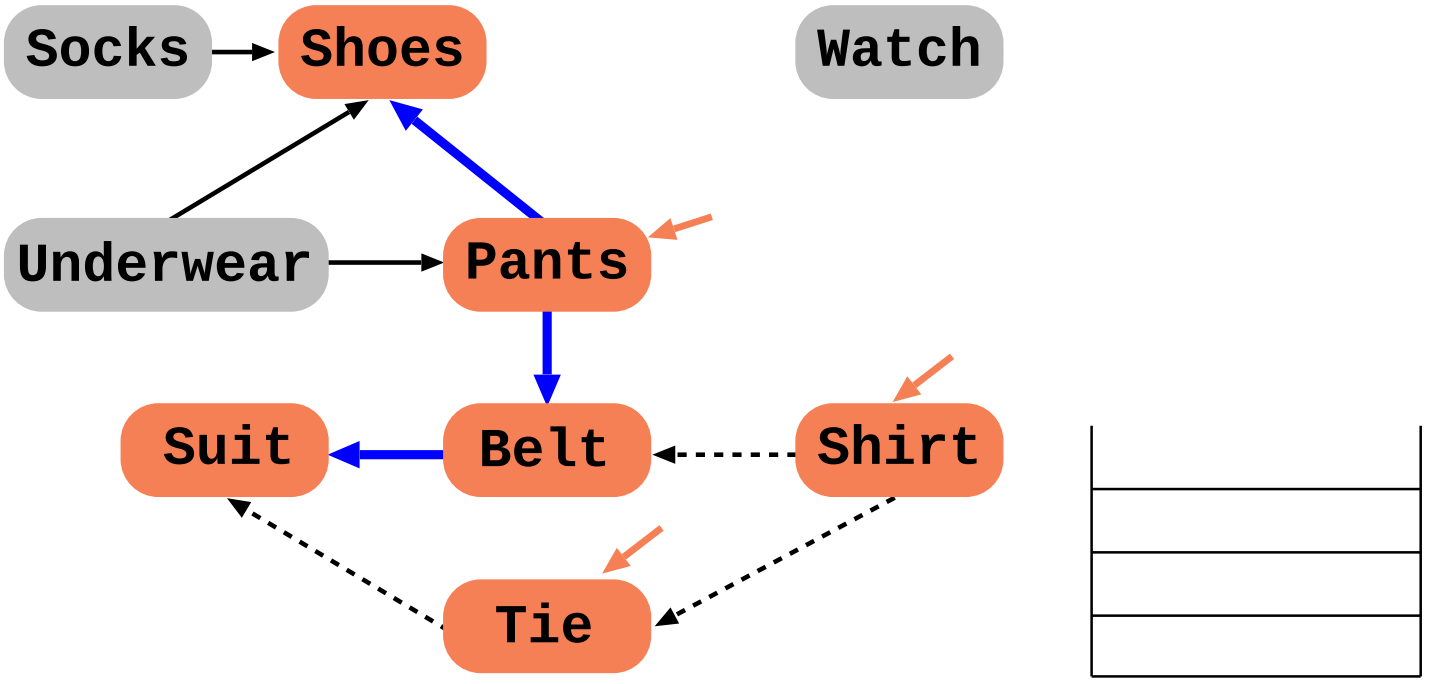
Busca em Profundidade (DFS - *Depth-First Search*)



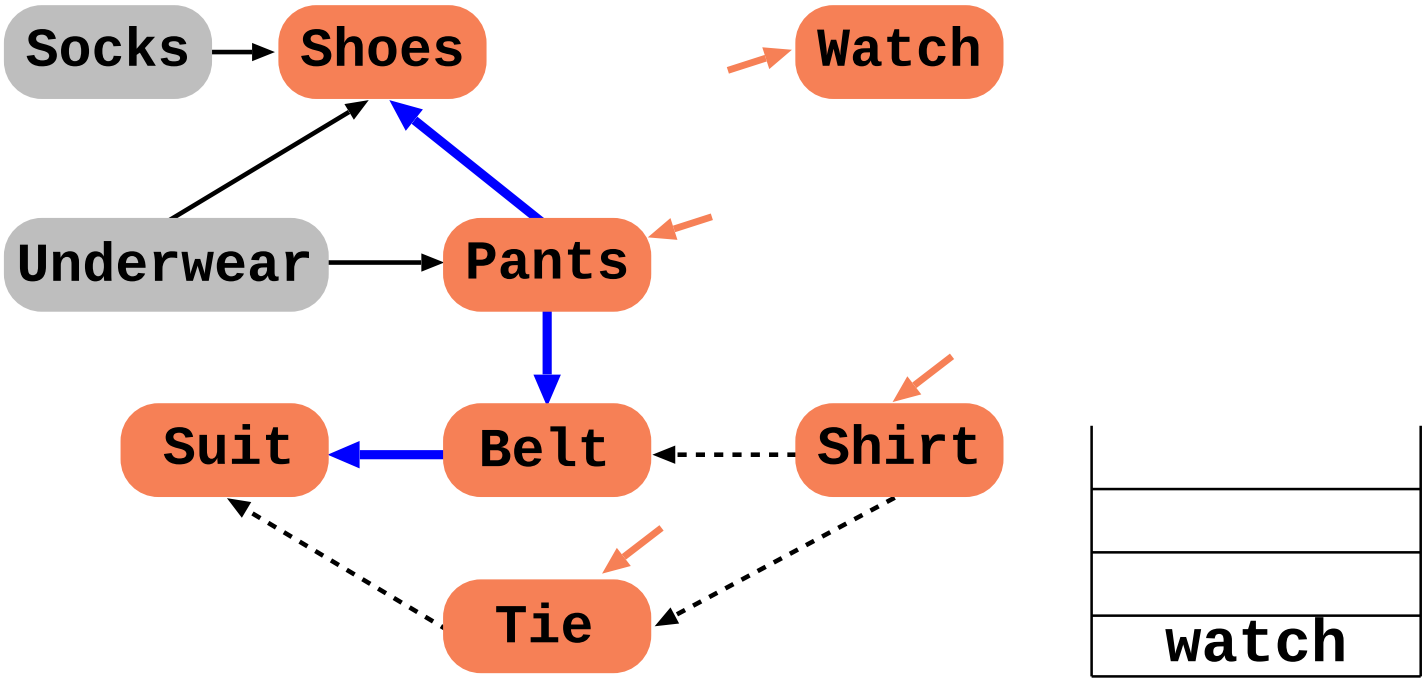
Busca em Profundidade (DFS - *Depth-First Search*)



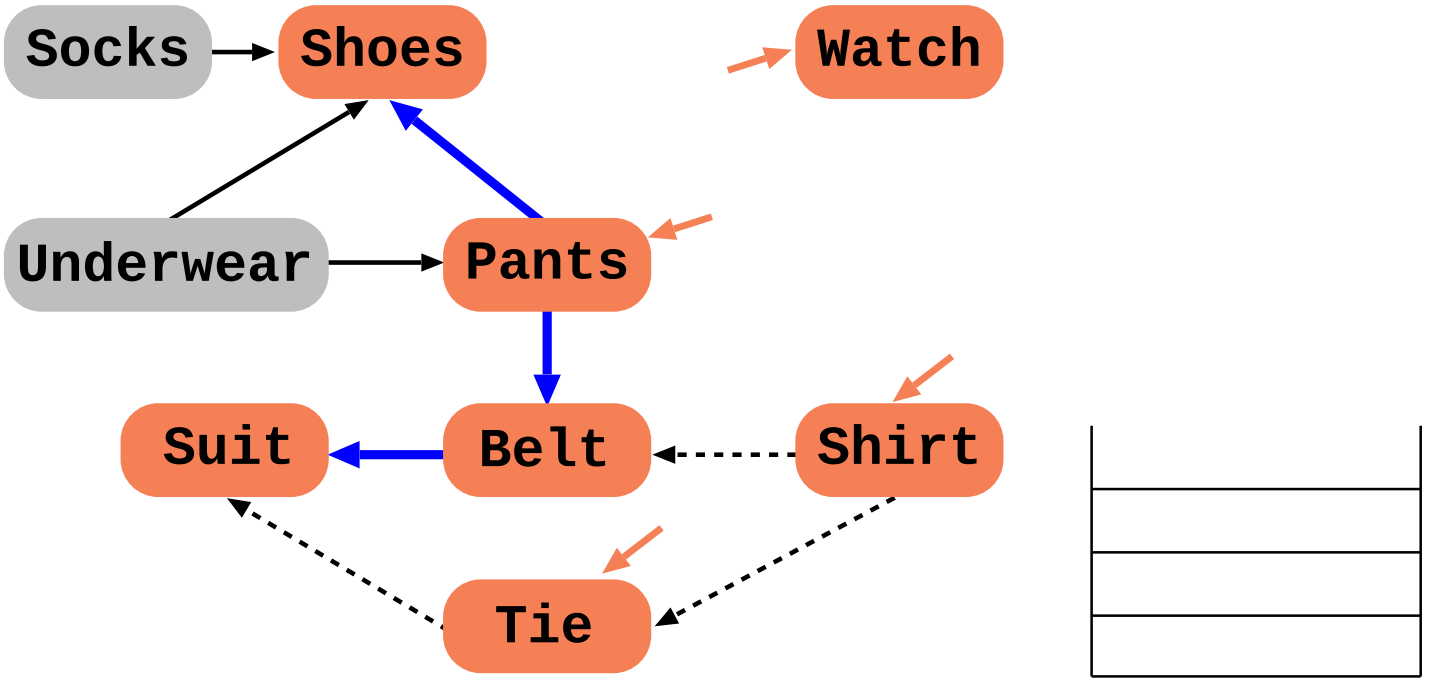
Busca em Profundidade (DFS - *Depth-First Search*)



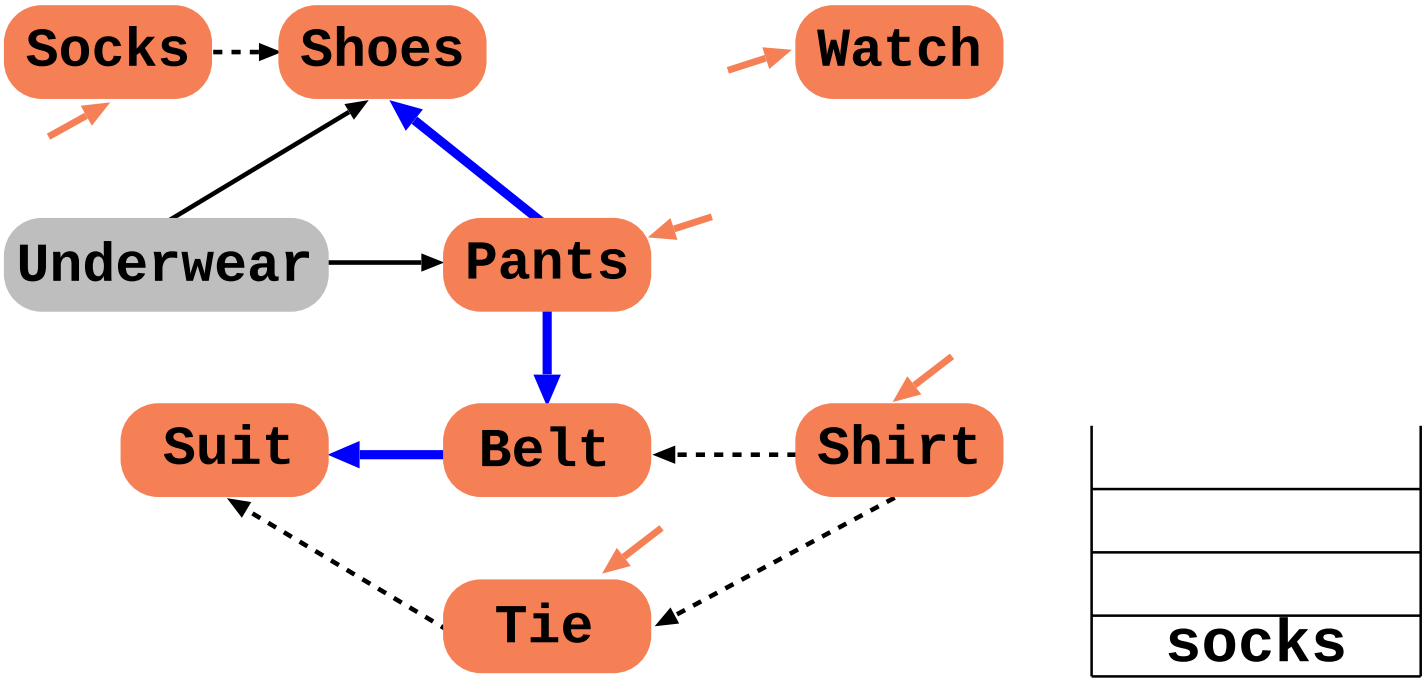
Busca em Profundidade (DFS - *Depth-First Search*)



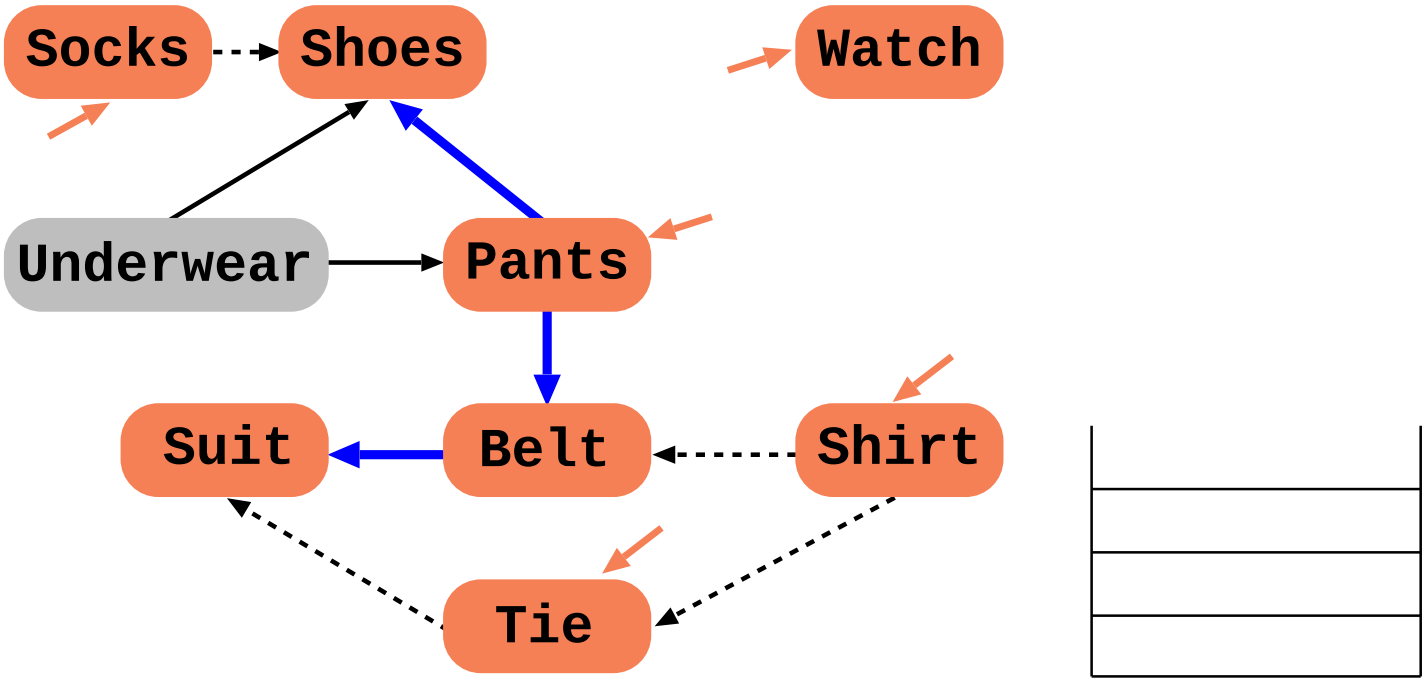
Busca em Profundidade (DFS - *Depth-First Search*)

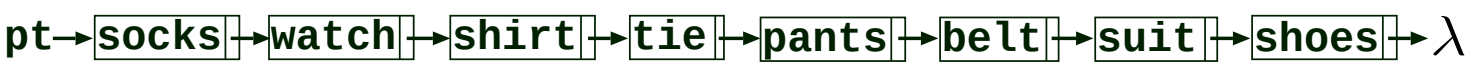


Busca em Profundidade (DFS - *Depth-First Search*)

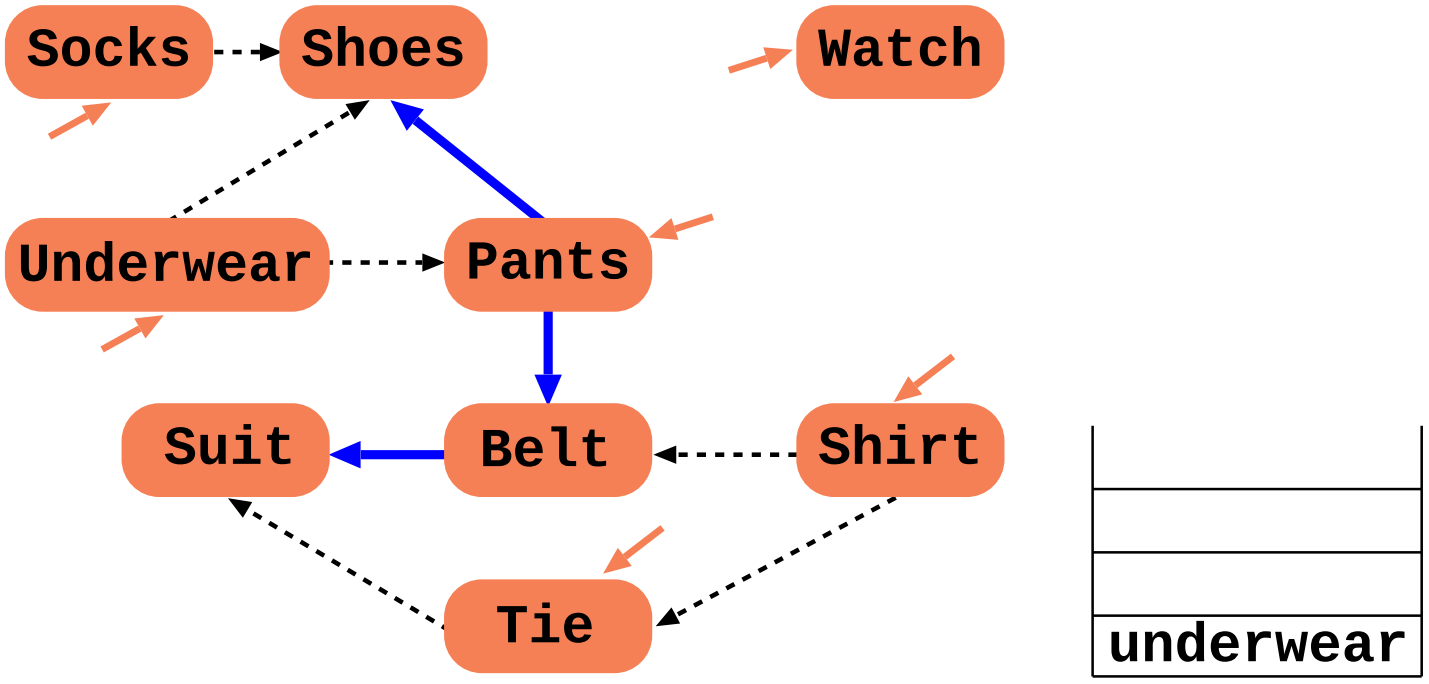


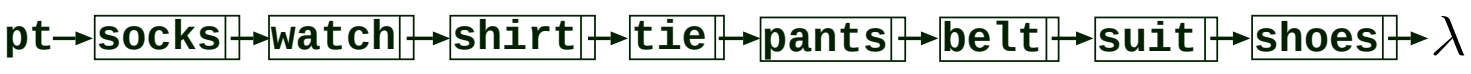
Busca em Profundidade (DFS - *Depth-First Search*)



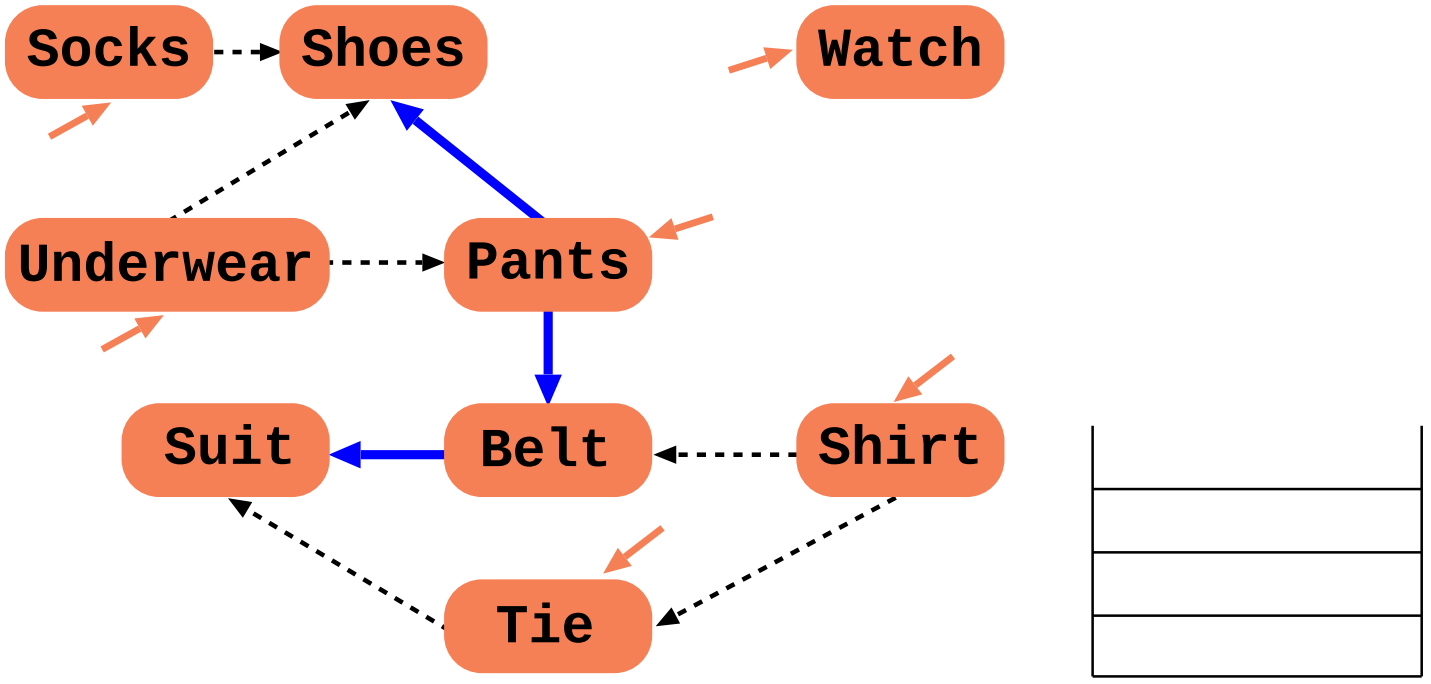


Busca em Profundidade (DFS - *Depth-First Search*)





Busca em Profundidade (DFS - *Depth-First Search*)



Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)


7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Algoritmo Ordenação Topológica

Análise da complexidade:

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)


7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 5 e 12): $O(n)$

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 5 e 12): $O(n)$
- A lista de adjacência de cada vértice percorrida uma vez (linha 3): $O(m)$

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 5 e 12): $O(n)$
- A lista de adjacência de cada vértice percorrida uma vez (linha 3): $O(m)$
- inserePrimeiro insere no início de lista ligada: $O(1)$;

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8      u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da complexidade:

- Cada vértice passado para método dfs (linha 5 e 12): $O(n)$
- A lista de adjacência de cada vértice percorrida uma vez (linha 3): $O(m)$
- inserePrimeiro insere no início de lista ligada: $O(1)$;
- Complexidade total: $O(n+m)$

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)


7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)


7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da corretude:

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da corretude:

- Vértices são adicionados no começo da lista somente se não tem mais arestas de saída;

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da corretude:

- Vértices são adicionados no começo da lista somente se não tem mais arestas de saída;
- Sumidouros são adicionados primeiro;

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da corretude:

- Vértices são adicionados no começo da lista somente se não tem mais arestas de saída;
- Sumidouros são adicionados primeiro;
- Depois, são adicionados vértices “satisfeitos”;

Algoritmo Ordenação Topológica

```
1  dfs(G, u):
2      u.visitado = True
3      para v em adj(u) faça
4          se não v.visitado então
5              dfs(G, v)
6      inserePrimeiro(L, u)

7  para u em V(G) faça
8       u.visitado = False
9  L = Lista();
10 para u em V(G) faça
11     se não u.visitado então
12         dfs(G, u)
```

Análise da corretude:

- Vértices são adicionados no começo da lista somente se não tem mais arestas de saída;
- Sumidouros são adicionados primeiro;
- Depois, são adicionados vértices “satisfeitos”;
- Finalmente, fontes são adicionadas;

Exercícios

1. Modifique o algoritmo DFS para verificar se um grafo é acíclico.
2. Supondo que *G* seja conexo, como podemos usar o algoritmo DFS para obter uma árvore geradora do grafo *G*?
3. O algoritmo DFS pode ser usado para checar se um grafo é conexo?
4. Proponha um algoritmo alternativo para resolver o problema de ordenação topológica sem utilizar DFS mas que tenha mesma complexidade de tempo.

Bibliografia

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L. e STEIN, C. *Introduction to Algorithms*, 3ª edição, MIT Press, 2009.