FGV EMAp

Escrita: Thalis Ambrosim Falqueto

Projeto e Análise de Algoritmos

Revisão para A2

Rio de Janeiro

2025

Conteúdo

Técnicas de Projeto

1.1 Método guloso (Greedy)

O método guloso é uma famoso paradigma utilizado para projetos de algoritmo, onde a estratégia consiste em escolher a cada iteração a opção com maior valor, e avaliar se deve ser adicionada ao resultado final.

Seguindo essa abordagem, as opções precisam ser ordenadas pro algum critério. Costuma ser simples e eficiente, porém nem todo projeto pode ser resolvido através dessa abordagem.

1.1.1 O problema do agendamento de tarefas

Dado o conjunto de tarefas $T = \{t_1, t_2, ..., t_n\}$ com n elementos, cada uma com tempo de ínicio `[tk]`, e um tempo de término `[tk]`, encontre o maior subconjunto de tarefas que pode ser alocado sem sobreposição temporal.

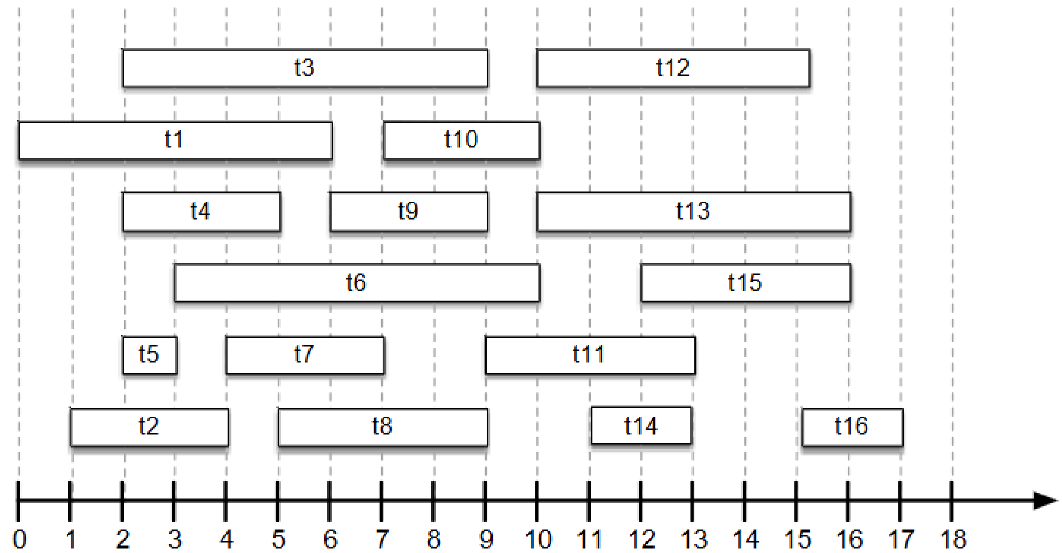


Figura 1: Exemplo do problema de agendamento

Vamos projetar a solução!

Perguntas:

1. Quais serão as opções a serem avaliadas a cada iteração?
   - Conjunto de tarefas que ainda não foi alocada ou descartada.
2. Qual critério iremos utilizar para ordenar as opções?
   - Tempo de ínicio?
   - Menor duração?
   - Menor número de projetos?
   - Tempo de término?

Vamos analisar cada critério tentando construir ao menos um cenário que demonstre que o critério gera um resultado não-ótimo.

Que tal se colocarmos o critério de seleção como o tempo de início do agendamento?

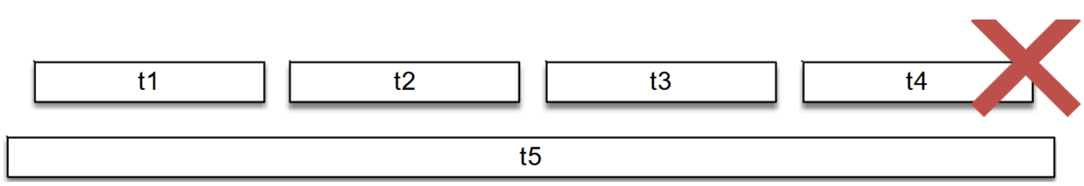


Figura 2: Contra-exemplo para uma possível solução do problema de agendamento

Isso não daria certo, pois nesse caso, por exemplo, o t_5 seria escolhido, enquanto a melhor escolha seria pegar as 4 primeiras tarefas.

E se escolhessemos pela menor duração?

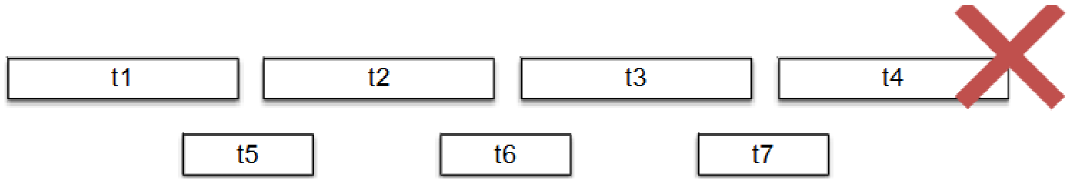


Figura 3: Contra-exemplo para uma possível solução do problema de agendamento

Isso também daria errado, já que escolheríamos t_5, t_6 e t_7 enquanto, novamente, a melhor escolha seriam os 4 primeiros.

Nada funcionou... Mas e se nosso critério fosse o tempo do término?

Ideia geral:

ordene T (pelo tempo de término)
$T_a[1, ..., n] = 0$
Insira a primeira tarefa da lista t_0 **em** T_a
$t_{\text{prev}} = t_0$
para t_k **em** T:
    se $\text{start}[t_k] \geq \text{end}[t_{\text{prev}}]$:
        adicione t_k **em** T_a
        $t_{\text{prev}} = t_k$
retorne T_a.

Essa ideia não é muito difícil. Como a lista é ordenada pelo horário de saída, então o primeiro elemento a ser adicionado é simplesmente o primeiro elemento. Note que, como o objetivo é apenas a quantidade máxima de tarefas, e não o tempo máximo que podemos otimizar para todas as tarefas que temos, então pegar o menor término **desde o início** é o que realmente faz o algoritmo funcionar (por exemplo, se tivéssemos os horários `[(5,10),(5,12)]`, pegar o menor tempo de saída nos ajudaria no caso de termos outra tarefa, como `(11,14)`).

Após selecionarmos a primeira tarefa da lista, basta compararmos os tempos de entrada das próximas tarefas, já que, pelo mesmo raciocínio do porque escolher a menor saída, se a próxima tarefa não colidir com a saída passada, então podemos pegar nossa nova tarefa e atulizar com o tempo de saída da nova tarefa atual (como a lista está ordenada pelo tempo de fim, a nova tarefa a ser pega garantiria que seria a melhor tarefa, já que seria a primeira que se encaixa com o tempo de finalização da última tarefa selecionada e a mais curta já que estamos olhando por ordenação).

Nosso pseudocódigo usa apenas um for sem nada demais dentro dele, mas precisamos ordenar a lista antes. Isso nos traz uma complexidade de $\Theta(n \log(n))$.

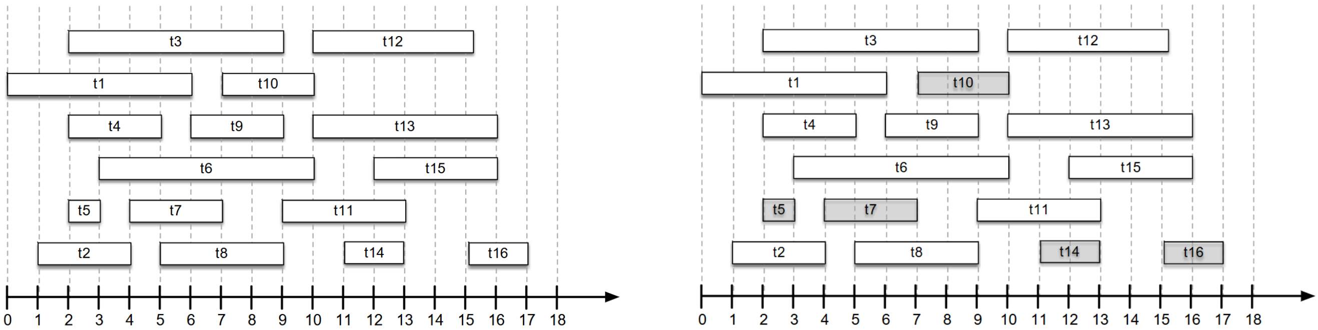


Figura 4: Solução para o problema de tarefas usando o algoritmo proposto

Por que essa solução é ótima?

Definição 1.1.1.1: Escolha gulosa

Uma solução ótima global pode ser atingida realizando uma sequência de escolhas locais ótimas (gulosas).

- A escolha local não considera o resultado das escolhas posteriores, e produz um sub-problema contendo um número menor de elementos.
- A definição do critério de escolha nos auxilia à organizar os elementos de forma que o algortimo seja eficiente.

Definição 1.1.1.2: Sub-estrutura ótima

Ocorre quando uma solução ótima de um problema apresenta dentro dela soluções ótimas para sub-problemas.

Definição 1.1.1.3: Swap argument (Argumento de troca)

Considere que temos uma solução ótima S, e a solução gulosa G. Então é possível substituir iterativamente os elementos de S por elementos de G sem que a solução deixe de ser viável e ótima, provando assim que G é, no mínimo, tão boa quanto S.

Vamos usar o que aprendemos então:

Seja $T_a = \{g_1, g_2, ..., g_k\}$ o conjunto de k tarefas selecionadas pelo nosso algoritmo guloso, já ordenadas pelo tempo de término (como no pseudocódigo). Seja $S = \{s_1, s_2, ..., s_m\}$ uma *solução ótima* qualquer, com m tarefas, também ordenadas por tempo de término.

Nosso objetivo é provar que T_a é ótima, ou seja, que $k = m$.

Queremos primeiro provar que a primeira escolha gulosa, g_1, pode fazer parte de *alguma* solução ótima.

1. g_1 é a tarefa escolhida por nosso algoritmo, então ela é a tarefa em *todo* o conjunto T com o *menor tempo de término*.
2. s_1 é a primeira tarefa da solução ótima S. Ela tem o menor tempo de término *dentro de* S.

Por definição, como g_1 tem o menor tempo de término de *todas* as tarefas, seu tempo de término deve ser menor ou igual ao de s_1:

$$\text{end}[g_1] \leq \text{end}[s_1] \tag{1}$$

Agora, vamos comparar g_1 e s_1.

1. *Caso 1:* $g_1 = s_1$. Se a primeira tarefa da solução ótima S é a mesma da solução gulosa T_a, então S já começa com a escolha gulosa.

2. *Caso 2:* $g_1 \neq s_1$. Vamos "trocar" s_1 por g_1 na solução ótima S. Considere uma nova solução S': $S' = \{g_1, s_2, s_3, ..., s_m\}$ Precisamos verificar se S' ainda é uma solução viável (sem sobreposições).
   - Como S era uma solução viável, todas as suas tarefas eram compatíveis. Sabemos que s_2 devia começar após s_1 terminar: $\mathrm{start}[s_2] \geq \mathrm{end}[s_1]$.
   - Mas, como vimos na Etapa 1, $\mathrm{end}[g_1] \leq \mathrm{end}[s_1]$.
   - Combinando os fatos, temos que $\mathrm{start}[s_2] \geq \mathrm{end}[g_1]$.
   - Isso significa que g_1 não se sobrepõe a s_2, e o resto das tarefas ($s_3, ...$) também não, pois já eram compatíveis com s_2.

A nova solução S' é, portanto, viável. O mais importante é que S' tem m tarefas, o *mesmo tamanho* da solução ótima S. Isso significa que S' *também é uma solução ótima*.

Concluímos que *sempre* existe uma solução ótima (seja S ou S') que começa com a primeira escolha gulosa g_1. Podemos repetir esse processo indutivamente. Em cada passo i, trocamos s_i por g_i, transformando a solução ótima S na solução gulosa T_a, sem nunca diminuir o número de tarefas, usando sub-estruturas ótimas. Isso só é possível se as duas soluções tiverem o mesmo tamanho desde o início. Portanto, $k = m$.

Logo, a solução gulosa T_a é, de fato, uma solução ótima.

Implementação em Python:

```py
def scheduling_problem(tasks):
    if len(tasks) == 0:                                     #caso de contorno
        return 0

    sorted_by_end = sorted(tasks, key= lambda x:x[1]) #ordena pelo término
    choosed_tasks = []
    choosed_tasks.append(sorted_by_end[0])
    t_prev = sorted_by_end[0]

    for task in sorted_by_end[1:]:                          #começa depois da primeira
        if task[0] >= t_prev[1]:                            #tempo maior que o de saída
            choosed_tasks.append(task)
            t_prev = task
    return choosed_tasks, len(choosed_tasks)                #retorna lista, quantidade
```

1.1.2 O problema da mochila fracionária

Dado um conjunto de itens $\mathbb{I} = \{1, 2, 3, ..., n\}$ em que cada item $i \in \mathbb{I}$ tem um peso w_i e um valor v_i, e uma mochila com capacidade de peso W, encontre o subconjunto $S \subseteq \mathbb{I}$ tal que $\sum_{i \in S}^{|S|} \alpha_i w_i \leq W$ e $\sum_{i \in S}^{|S|} \alpha_i v_i$ seja máximo, considerando que $0 < \alpha_k \leq 1$.

Item	Peso	Valor
1	1	2
2	3	4
3	4	6
4	5	10
5	7	8

Figura 5: Tabela de exemplo para o exemplo da mochila

Exemplo:

- W = 9
  - A escolha $\{1, 2, 3\}$ tem peso 8, valor 12 e cabe na mochila;
  - A escolha $\{3, 5\}$ tem peso 11, valor 14 e **não** cabe na mochila
  - A escolha $\{3_{50\%}, 5_{100\%}\}$ tem peso 9, valor 11 e cabe na mochila
  - A escolha $\{1_{100\%}, 3_{75\%}, 4_{100\%}\}$ tem peso 9, valor 16.5 e cabe na mochila

Seria possível criar um algoritmo capaz de encontrar uma solução ótima para esse problema?

- Pergunta 1: quais são as opções a serem avaliadas à cada iteração?
  - Itens (ou fragmentos de itens) que ainda não foram adicionados ou descartados.
- Pergunta 2: Qual critério iremos utilizar para ordenar as opções?
  - Menor peso?
  - Menor valor?
  - Maior razão peso/valor?

Essa ideia de razão parece fazer sentido, já que podemos separar e pegar a proporção que quisermos. Daí vem a ideia do algoritmo:

Mochila (I, v, w, n, W) :
    ordene I (pela razão valor/peso)
    $C, i = W, 1$
    $M[1, ..., n] = 0$
    enquanto $i \leq n \;\text{ e }\; C \geq w_i$:
        $M[i] = 1$
        $C = C - w_i$
        $i += 1$
    se $i \leq n$:
        $M[i] = \frac{C}{w_i}$
    retorne M

Onde I é o conjunto de itens, v o vetor de valores de cada item, w o vetor de pesos de cada item, n a quantidade de itens e W é a capacidade máxima da mochila.

Analisando brevemente, ordenamos **descrescentemente** o vetor de itens $\mathbb{I}$, e declaramos a variável C, de capacidade, e i, de índice. Criamos o vetor de zeros M (de tamanho n), que é o vetor de porcentagem, referente a cada item. Note que como estamos ordenando pela proporção de valor por peso decrescentemente, pegar o primeiro item significa pegar o que item que mais vale a pena. Logo, o while serve para, enquanto couber a capacidade, pegara maior quantidade possível de valores. Quando o while quebra (no índice i), o algoritmo verifica se não chegou ao final, e, caso não tenha chegado, pega a proporção máxima da capacidade máxima restante sobre o peso, e retorna a lista de pesos ao final.

O mais complexo é a ordenação, que pode ser garantido com $\Theta(n \log(n))$.

Implementação em Python:

```py
def fractional_bag_problem(I, v, w, max_w):
  n = len(I)                          #as três listas têm o mesmo tamanho
  idx_w_ratio = []
  for i in range(n):
      ratio = v[i]/w[i]
      idx_w_ratio.append((i, w[i], ratio))#lista que armazena o índice, peso e razão
                                      #ordena por razão logo abaixo
  idx_w_ratio = sorted(idx_w_ratio, key = lambda x: x[2], reverse=True)
  capacity, i = max_w, 0
  M = [0] * n

  while i < n and capacity >= idx_w_ratio[i][1]:
      M[i] = 1                        #faz o while normal
      capacity -= idx_w_ratio[i][1]
      i += 1
  if i < n:
      M[i] = capacity/idx_w_ratio[i][1]

  itens_choosed = [0] * n             #lista que referencia a cada item a sua
  for j in range(n):                  #porcentagem escolhida
      if M[j] != 0:
          itens_choosed[idx_w_ratio[j][0]] = M[j]

  return itens_choosed                #retorna a lista de índices com a %
```

1.2 Dividir e Conquistar

O nome já diz muito, e esse paradigma é dividido em três etapas:

- Dividir o problema em um conjunto de sub-problemas menores.
- Resolver cada sub-problema recursivamente.
- Combinar os resultados de cada sub-problema gerando a solução.

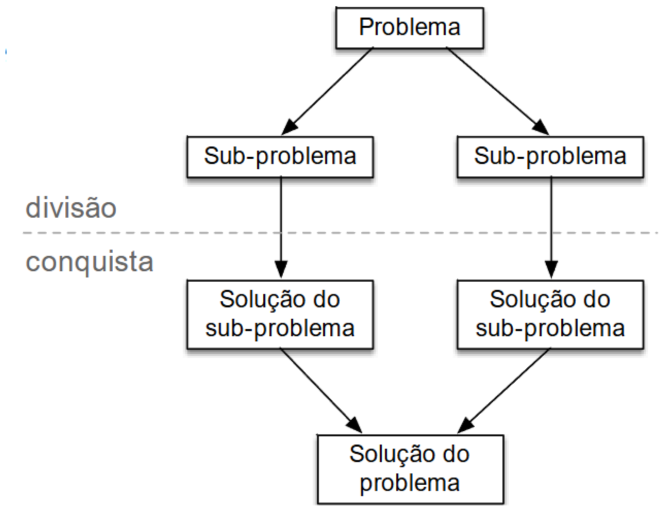


Figura 6: Exemplificação do paradigma Dividir e Conquistar.

1.2.1 O problema de contagem de inversões

Dado um problema com n números, calcule o número de inversões necessário para torná-la ordenada.

Exemplo:

Considere a sequência `A = [3,7,2,9,5]`

O número de inversões é 4: `(7,2),(3,2),(9,5),(7,5)`

A solução por força bruta seria verificar todos os pares, exigindo $\Theta(n^2)$.

A solução baseada em dividir e conquistar deverá definir estratégias para resolver cada sub-problema do número de inversões, e depois juntar, claro. Podemos dividir a sequência em dois grupos com aproximadamente metade (O primeiro array até $\frac{n}{2}$, o segundo de $\frac{n}{2}+1$ até n). Essa operação é constante, portanto $O(1)$.

A estratégia de resolução deve contar o número de inversões de cada grupo:

Exemplo:

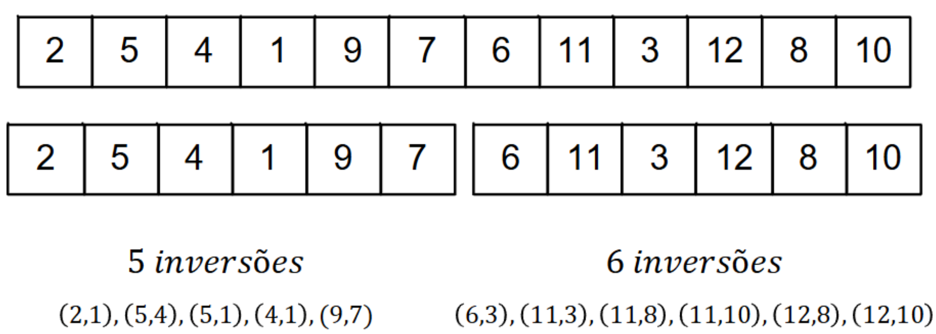


Figura 7: Exemplo do problema da contagem de inversões

Esse resultado pode ser obtido executando o algoritmo recursivamente ($\sim T(\frac{n}{2})$). Claro que, por fim, teremos que contar as inversões da junção das duas listas:

+ 7 *inversões ao combinar*

(5,3), (4,3), (9,6), (9,3), (9,8), (7,6), (7,3)

Totalizando, assim, 18 inversões.

Ok, a ideia está concisa, mas como fazer essa junção? Se ordenarmos cada segmento, e “juntarmos” direto, conseguiríamos fazer isso de forma fácil. Voltemos ao exemplo após ordenar:

1	2	4	5	7	9

3	6	8	10	11	12

Para a contagem de inversões para a junção, considere a sequências à esquerda de L e à direita de R e defina $i = 0$.

para a_j **de** R:
  incremente i **até** $L[i] > a_j$
  $\text{inv}_{\text{aj}} = |L| - i$.

Com inv_{aj} sendo a quantidade de inversões do elemento a_j de R.

Para explicar, considere o exemplo da imagem anterior e relembre um pouco do algoritmo de ordenação MergeSort. Dado que as listas menores já estão ordenadas, se colocarmos um ponteiro no início de cada lista, digamos `l` e `r`, então basta verificar se $L[l] \leq R[r]$, e incrementarmos l(ou r, dependendo da comparação).

Vamos continuar o exemplo (à princípio, pense que apenas juntamos as duas listas):
- Para `l = 0` e `r = 0`, temos $1 \leq 3$, que é verdade. Incrementamos o `l`.
- Para `l = 1` e `r = 0`, temos $2 \leq 3$, que ainda é verdade. Incrementamos o `l`.
- Para `l = 2` e `r = 0`, temos $4 \leq 3$, que é mentira. Então, podemos aplicar o raciocínio: sabendo que todos os elementos após o 4, como estão ordenados, são maiores ou iguais a 4 e, portanto, também teriam que ser considerados como maiores do que 3, então do ínidice l atual até $|L|$, uma inversão precisaria ser feita, se concatenássemos as duas listas. Logo, temos $|L| - l$ trocas a serem feitas a partir de $L[l]$. Isso explica o algoritmo para a contagem de inversões para a junção.

Agora que resolvemos esse problema, podemos desenhar o algoritmo final:

CountInversions (A) :
  se $\text{len(A)} = 1$:
    retorne 0
  divida a lista em L e R
  $i_l = \text{CountInversions}(L)$
  $i_r = \text{CountInversions}(R)$
  $L = \text{Sort}(L)$
  $R = \text{Sort(R)}$
  $i = \text{Combine (L, R)}$
  retorne $i_l + i_r + i$

Avaliando a complexidade desse algoritmo, temos:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n \log(n)) = O(n(\log(n))^2) \tag{2}$$

O $O(n \log(n))$ vem da ordenação das duas listas, e o $2T(\frac{n}{2})$ das duas listas que separamos. Os métodos de complexidade aprendidos na A1 nos levam a uma solução simples de $O(n(\log(n))^2)$, já que sabemos que essa divisão pela metade traz um peso de $\log(n)$.

Seria possível trazer uma otimização ainda maior? Sim, se eliminassemos a etapa de ordenação explícita. Podemos fazer isso se, no algoritmo de contagem de inversão, além de contar, invertessemos e realizassemos o merge, trazendo o array ordenado direto.

Novo algoritmo:

CountInversions (A) :
- **se** $\mathrm{len(A)} = 1$:
  - **retorne** $0, A$
- **divida a lista em** L e R
- $i_l, L = \mathrm{CountInversions}(L)$
- $i_r, R = \mathrm{CountInversions}(R)$
- $i = \mathrm{Combine\ (L,\ R)}$
- $A = \mathrm{Merge}(L, R)$
- **retorne** $(i_l + i_r + i), A$

Aqui, nós nos aproveitamos da recursão para fazer a ordenação, em vez de fazer isso por outro algoritmo. Dado que a recursão vai até quando só tivermos um elemento, e que sempre fazemos um merge, a lista está garantidamente sempre ordenada. Isso permite a execução do Combine sem problemas.

A função Combine é $O(n)$, já que apenas conta a quantidade de inversões que precisaríamos. A função Merge também é $O(n)$, já que apenas junta as duas listas. Portanto, temos:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = n \log(n) \tag{3}$$

Implementação em python

Para essa implementação, vamos relembrar basicamente a função MergeSort, só que contando a quantidade de inversões. Ainda, vamos juntar as funções CountInversions e a Merge numa mesma.

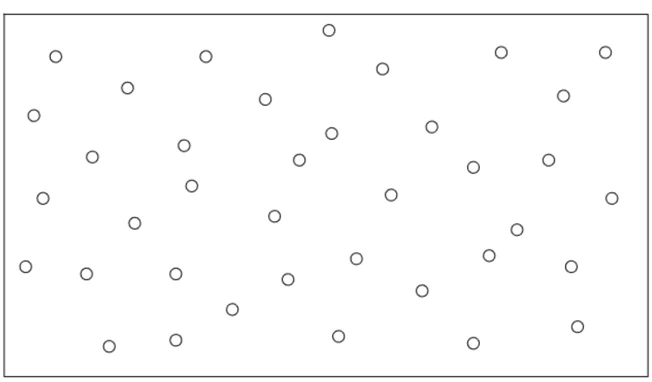
```py
def combine_merge(v, start_a, start_b, end_b):
    r = [0] * (end_b - start_a)
    a_idx = start_a
    b_idx = start_b
    r_idx = 0
    num_inv = 0                         #adicionado

    while a_idx < start_b and b_idx < end_b:
        if v[a_idx] <= v[b_idx]:
            r[r_idx] = v[a_idx]
            a_idx += 1
        else:
            num_inv += start_b - a_idx  #adicionado (explicado anteriormente)
            r[r_idx] = v[b_idx]
```

```
            b_idx += 1
        r_idx += 1

    while a_idx < start_b:
        r[r_idx] = v[a_idx]
        a_idx += 1
        r_idx += 1

    while b_idx < end_b:
        r[r_idx] = v[b_idx]
        b_idx += 1
        r_idx += 1

    for i in range(len(r)):
        v[start_a + i] = r[i]

    return num_inv                      #adicionado

def count_inversions(v, start_idx, end_idx):
    if (end_idx - start_idx) > 1:
        mid_idx = (start_idx + end_idx) // 2
        il = count_inversions(v, start_idx, mid_idx)      #essas linhas mudaram
        ir = count_inversions(v, mid_idx, end_idx)        #apenas a igualdade

        i = combine_merge(v, start_idx, mid_idx, end_idx) #antes não tinha
      return il + ir + i                #adicionado
    else:
      return 0
```

Todo o código (a menos de linhas comentadas) foi tirado da versão original do MergeSort. Como dito, fizemos o MergeSort contando a quantidade de inversões. :)

1.2.2 O problema de pares mais próximos

Dado uma sequência com n pontos em um plano, encontre o par com a menor distância euclidiana.



A primeira solução que vem a cabeça é simplesmente testar cada par com cada outro par, trazendo uma complexidade de $O(n^2)$

Como desenvolver uma solução melhor com o método que aprendemos?

Figura 10: Exemplificação do problema de pares mais próximos

Podemos dividir o plano de forma que cada lado tenha aproximadamente o mesmo número de pontos (ordenando pelo eixo x).

Em seguida, resolva cada lado encontrando o par mais próximo recursivamente.

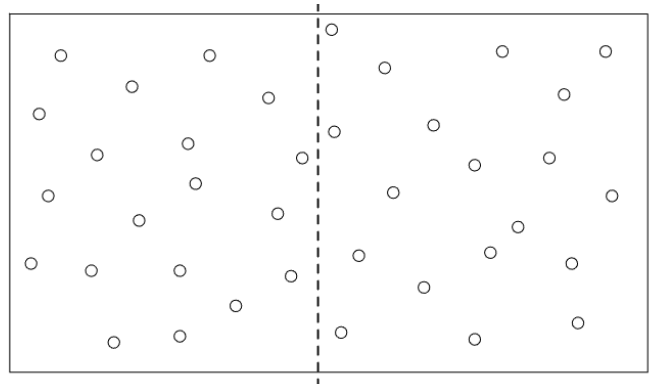
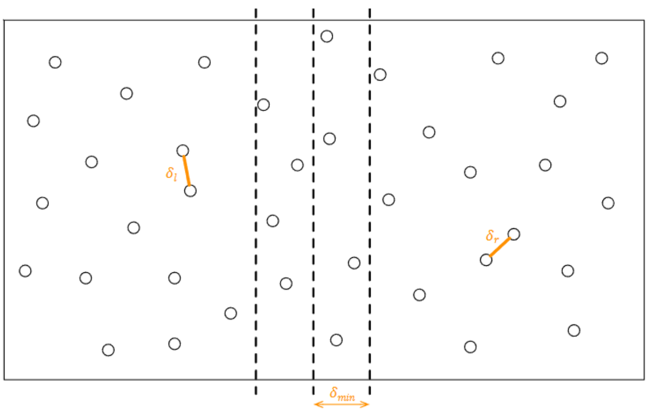


Figura 11: Exemplificação da solução do problema de pares mais próximos

Com o plano dividido, combine os resultados comparando O par mais próximo no lado direito, o par mais próximo do lado esquerdo, e o par mais próximo em cada lado. A última comparação parece exigir $\Theta(n^2)$, não parece muito bom.

Se pensarmos apenas na comparação da divisão dos planos, sejam δ_l e δ_r os pares com menor distância nos lados esquerdo e direito, respectivamente.



Como estamos procurando o par mais próximo, seja $\delta_{\min} \leq \min(\delta_l, \delta_r)$ (sabemos que $\delta_{\min}$ está restrito a, no máximo, essa distância).

Ideia: procurar somente os pontos que estejam no máximo à $\delta_{\min}$ da divisória, ordenando os pontos na faixa $2\delta_{\min}$ pela posição o eixo y.

Qual seria a complexidade desse algoritmo?

Figura 11: Exemplo da distância de comparação.

Bom, não seria $O(n^2)$, pois a distância em cada lado é no mínimo $\delta_{\min}$.

1.2.3 como faz isso cara como é 11 7, 5 sla

1.2.4 Implementação em Python

1.3 Programação Dinâmica

O paradigma de programação dinâmica consiste em quebrar em sub-problemas menores e resolvê-los de forma independente. Semelhante ao dividir e conquistar, porém com foco em sub-problemas que usam repetição. Nessa técnica, um sub-problema só é resolvido caso não tenha sido resolvido antes (caso contrário é usado o resultado anterior guardado previamente).

1.3.1 O problema de Fibonacci

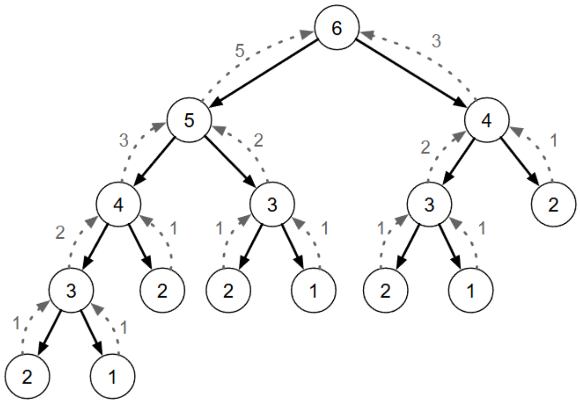


Figura 12: Exemplo de como seria $\mathrm{fib}(6)$

Dado um inteiro $n \geq 1$, encontre F_n. Solução recursiva (e ineficiente):

```py
def fib(n):
  if n <= 2:
    return 1
  return fib(n-1) + fib(n-2)
```

Note como, para n, o tempo de execução é exponencial, e que, grande parte dos problemas são re-computados. Podemos utilizar um cachê para reaproveitar resultados. Vamos fazer isso!

Solução Top-Down (recursiva):

```py
def Fib(n):
  if n == 0:
      return 0
  if n == 1:
      return 1

  F = [-1] * (n + 1)
  F[0], F[1], F[2] = 0,1,1

  def FibAux(k):
    if F[k] == -1:
      F[k] = FibAux(k - 1) + FibAux(k-2)
    return F[k]

  return FibAux(n)
```

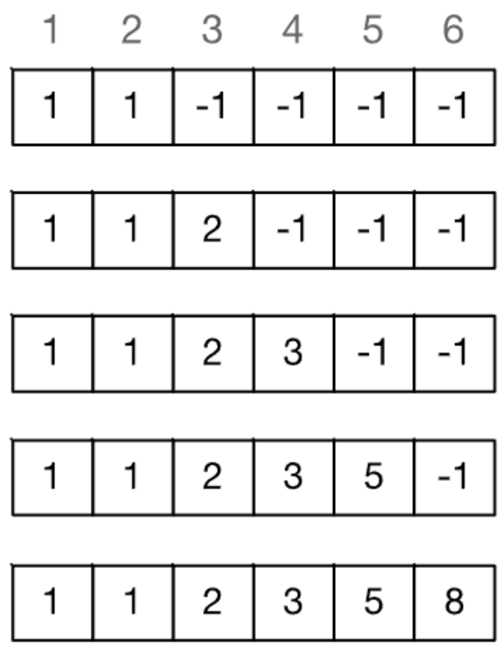


Figura 13: Exemplo de como fica o cachê para $\mathrm{Fib}(6)$

A complexidade desse algoritmo é $\Theta(n)$, sabendo que usamos apenas a lista para guardar os valores da lista de Fibonacci.

Solução Bottom-Up (interativa):

```py
def Fib(n):
  if n <= 2:
      return 1
  F = [0] * n
  F[1], F[2] = 1, 1
  for i in range (3,n + 1,1):
      F[i] = F[i - 1] + F[i - 2]
  return F[n]
```

A complexidade é a mesma da recursiva, desde que continuamos usando apenas um for e usando-a para guardar apenas uma lista.

A abordagem Top-Down é recursiva, somente executa a recursão caso o sub-problema não tenha sido resolvido e inicia no problema maior, enquanto em problemas Bottom-Up a solução é iterativa e resolve os sub-problemas do menor para o maior. Além disso, em geral apresentam a mesma complexidade.

1.3.2 O problema da mochila (não fracionária)

Dado um conjunto de itens $\mathbb{I} = \{1, 2, 3, ..., n\}$ em que cada item $i \in \mathbb{I}$ tem um peso w_i e um valor v_i, e uma mochila com capacidade de peso W, encontre o subconjunto $S \subseteq \mathbb{I}$ tal que $\sum_{i \in S}^{|S|} w_i \leq W$ e $\sum_{i \in S}^{|S|} v_i$ seja máximo.

A diferença desse problema para o que vimos no paradigma Guloso é que agora, não podemos pegar uma fração do item, apenas ou o pegamos ou não. Isso faz com que, agora, por mais que o item seja o mais valoroso possível na proporção valor/peso, ainda assim possa existir alguma outra combinação que tenha um valor maior.

Exemplo:

W = 11

A escolha $\{1, 2, 4\}$ tem peso 9, valor 29 e cabe na mochila.

A escolha $\{3, 5\}$ tem peso 12, valor 46 e não cabe na mochila.

Item	Peso	Valor
1	1	1
2	2	6
3	5	18
4	6	22
5	7	28

Figura 14: Tabela auxiliar para exemplo

Solução(ineficiente): criar um algoritmo de força bruta que testa todas as possibilidades e escolhe a que cabe na mochila com maior valor.

Tentando usar o que estamos aprendendo aqui (programação dinâmica), temos:

- para cada item i, considere a possibilidade de adicioná-lo ou não a mochila;
  - se adiconado, o valor é incrementado de v_i e a capacidade é reduzida de w_i
  - avalie qual o melhor valor obtido em cada caso.

Após considerar esse item, restam $n - 1$ itens disponíveis para serem avaliados (encontramos a sub-estrutura ótima).

Ideia geral (sem cachê):

Mochila (I, v, w, W) :
  se $|I| = 0$ **ou** $W = 0$
    retorne $\emptyset$
  escolha um item $i \in I$
  se $w_i > W$:
    retorne Mochila $(I - i, v, w, W)$:
  $\text{value\_using} = v_i +$ **Mochila** $(I - i, v, w, W - w_i)$
  $\text{value\_not\_using} =$ **Mochila** $(I - i, v, w, W)$

retorna $\max\{\text{value\_using}, \text{value\_not\_using}\}$

Vamos para as soluções definitivas, usando o paradigma que estamos aprendendo. A ideia, como temos que fazer uma comparação a cada item que podemos pegar com e sem ele, é usar uma matriz IxW, onde o valor de cada célula $M[i][w]$ responde a seguinte pergunta: Qual é o valor máximo que consigo obter usando apenas os itens de 1 até i, com uma mochila de capacidade máxima w (não W).

Solução Top-Down:

Mochila (n, v, w, W) :
  crie uma matriz nxW
  para $i = 0$ **até** W:
    $M[0][i] = 0$
    para $j = 1$ **até** n:
      $M[j][0] = 0$
      $M[j][i] = -1$
  retorna MocilhaAux(n, v, w, W)

onde n é o total de itens.

$W = 11$

i	w_i	v_i	0	1	2	3	...
0			0	0	0	0	...
1	1	1	0	-1	-1	-1	...
2	2	6	0	-1	-1	-1	...
3	5	18	0	-1	-1	-1	...
4	6	22	0	-1	-1	-1	...
5	7	28	0	-1	-1	-1	...

Figura 15: Exemplo da matriz para o algoritmo Top-Down e valores anteriores.

Continuação da solução:

MochilaAux (i, v, w, W) :
  se $M[i][W] = -1$:
    se $w_i > W$:
      $M[i][W] =$ **MochilaAux** $(i-1, v, w, W)$
    caso contrário:
      $\text{using} = v_i +$ **MochilaAux** $(i-1, v, w, W - w_i)$
      $\text{not\_using} =$ **MochilaAux** $(i-1, v, w, W)$
      $M[i][W] = \max\{\text{using,not\_using}\}$
  retorna $M[i][W]$

Onde i é o item que estamos considerando no momento. Essa solução usa a ideia explicada anteriormente, de fazer a verificação entre o melhor caso, adicionando e não adicionando. Vamos agora para a solução Bottom-Up:

Mochila (n, v, w, W) :
  crie uma matriz nxW
  para $i = 0$ **até** W:
    $M[0][i] = 0$
  para $j = 1$ **até** n:
    $M[j][0] = 0$
  para $j = 1$ **até** n:
    para $i = 1$ **até** W:
      se $w_j > i$:

```
            | M[j][i] = M[j − 1][i]
        caso contrário:
            | using = v_j + M[j − 1][i − w_j]
            | not_using = M[j − 1][i]
            | M[j][i] = max{using,not_using}
    retorna M[n][W]
```

Sabendo que toda a análise e o algoritmo é baseado na criação da matriz, onde dentro da criação de cada item acontecem apenas verificações, então a complexidade $\Theta(nW)$ (o que **não** é polinomial, já que W é um tamanho, não um valor). Vamos ver agora como essa matriz ficaria no final:

$W + 1$

$W = 11$

$n + 1$

i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10	11
0			0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	6	0	1	6	7	7	7	7	7	7	7	7	7
3	5	18	0	1	6	7	7	18	19	24	25	25	25	25
4	6	22	0	1	6	7	7	18	22	24	28	29	29	40
5	7	28	0	1	6	7	7	18	22	28	29	34	34	40

Figura 16: Resultado final da matriz finalizando o primeiro exemplo da mochila fracionária.

Lembre qual a função da matriz: o índice i (na linha) representa que podemos pegar qualquer dos itens 1 até i, e o peso W (na coluna) é o peso w que foi escolhido, e o número no índice nxw é o valor que conseguimos nessa combinação. Portanto, podemos interpretar que, na segunda linha, na coluna de $w = 0$, temos 0 itens para ser colocados e podemos colocar até um peso 0, logo, o valor máximo é 0. Ao continuar dessa linha, conseguimos ver que, a partir de quando o peso fica ≥ 1, conseguimos colocar o único item liberado (1), com peso 1 e valor 1. Por isso, toda a segunda linha é igual a 1 a partir do momento que $w \geq 1$.

Nota: Seguindo esse raciocínio, você, caro leitor, pode verificar cada valor da tabela. Existe **um** erro na tabela. Convido a você interpretá-la e entendê-lá e encontrar o erro. Se quiser validar que encontrou o erro, mande uma mensagem (Thalis).

Para finalizar, precisamos definir quais itens devem ser adicionados à mochila:

i	w_i	v_i	0	1	2	3	4	5	6	7	8	9	10	11
0			0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
2	2	6	0	1	6	7	7	7	7	7	7	7	7	7
3	5	18	0	1	6	7	7	18	19	24	25	25	25	25
4	6	22	0	1	6	7	7	18	22	24	28	29	29	40
5	7	28	0	1	6	7	7	18	22	28	29	34	34	40

Figura 17: Exemplo da busca dos itens adicionados (as células pintadas de laranja são as células visitadas pelo algoritmo)

$S, i, j = \{\}, W, n$
enquanto $j \geq 1$:
  se $M[j][i] = M[j-1]\left[i - w_j\right] + v_j$
    $S = S \cup \{j\}$
    $i = i - w_j$
  $j = j - 1$
retorna S

Vamos entender o código: j itera nas linhas, W nas colunas. Em teoria, a última célula da matriz $(n\mathrm{x}W)$ carrega com certeza o maior valor que satisfaz a condição do problema, e por isso começamos por ela. O que estamos fazendo é verificar se $M[j][i] = M[j-1]\left[i - w_j\right] + v_j$, ou seja, se o valor da célula voltando o peso do item atual (supondo que ele foi adicionado) e voltando um item $(j-1)$ somado ao valor de v_j é igual ao valor da célula atual, pois, se isso for verdade, significa que adicionamos esse valor ao descobrir o item j.

Vamos olhar para o exemplo da tabela:

- Ponto de partida: $M[5][11]$. Valor $= 40$. O item 5 (peso 7, valor 28) foi usado para obter esse valor de 40?
  - Comparamos o valor atual $(M[5][11] = 40)$ com o valor da célula de cima $(M[4][4] = 7 + v_j = 7 + 28 \neq 40)$.
  - Como os valores não são iguais, significa que o item 5 **não** foi incluído. A solução ótima para capacidade 11 já existia sem ele.
  - Então o algoritmo "sobe" para a célula $M[4][11]$.
- Posição Atual: Célula $M[4][11]$. Valor $= 40$. O item 4 (peso 6, valor 22) foi usado?
  - Comparamos o valor atual $(M[4][11] = 40)$ com o valor da célula de cima $(M[3][5] = 18 + v_j = 18 + 22 = 40)$.
  - Os valores são iguais. Isso significa que o item 4 **foi** incluído!
  - Adicionamos o item 4 ao nosso conjunto de solução S.O algoritmo "sobe" para a linha anterior $(i = 3)$ e "anda para a esquerda" subtraindo o peso do item 4 da capacidade: $11 - 6 = 5$. O novo ponto de análise é $M[3][5]$.

E assim sucessivamente!

Implementação em Python

```py
def bag_problem_bottom_up(n, v, w, W):
  #primeira parte (criar a matriz e inserir os valores)
  M = [[0] * (W + 1) for _ in range(n + 1)]
  for j in range(1, n + 1):
      for i in range(1, W + 1):
          peso_item_j = w[j-1]
          valor_item_j = v[j-1]
          if peso_item_j > i:
              M[j][i] = M[j - 1][i]
```

```python
            else:
                not_using = M[j - 1][i]
                using = valor_item_j + M[j - 1][i - peso_item_j]
                M[j][i] = max(using, not_using)

    #segunda parte (identificar os itens selecionados)
    itens_selecionados = []
    valor_maximo = M[n][W]
    j, i = n, W
    while j > 0 and i > 0:
        peso_item_j = w[j-1]
        valor_item_j = v[j-1]
        if peso_item_j <= i and M[j][i] == (M[j- 1][i - peso_item_j] + valor_item_j):
            itens_selecionados.append(j)
            i -= peso_item_j
        j -= 1
    itens_selecionados.reverse()
    return valor_maximo, M, itens_selecionados
```

Convido o caro leitor a implementar a solução Top-Down.

Grafos

Sinceramente, fazer uma revisão de conceitos muito básicos de grafos que já vimos em Matemática Discreta é um pouco chato, para não dizer desnecessário. Irei relembrar alguns termos e definir outros que não me lembro de termos vistos. Vamos lá:

2.1 Relembrando conceitos

- $V \to$ vértices;
- $E \to$ arestas;
- Uma aresta é definida pelo par (v_i, v_j);
- O **tamanho** de um grafo é definido por $|V| + |E|$, onde $|.|$ é a cardinalidade (quantidade de elementos);
- Dado $e = (v_i, v_j)$, v_i e v_j são **extremos** da aresta se e é incidente em v_i e v_j, e v_i e v_j é incidente em e;
- Vértices relacionados por uma aresta são **adjacentes**;
- Duas arestas são **paralelas** se incidem ao mesmo vértice;
- **Laço** $= (v_i, v_i)$;
- $\sum_{i=1}^{|V|} g(v_i) = 2\ |E|$, onde $g(v_i)$ é o **grau** do vértice i;
- Um grafo é **completo** se cada vértice possuir todos os demais adjacentes à ele;
- O número de arestas em um grafo completo é definido por: $\frac{|V|(|V|\ -1)}{2}$ (observe que isso é ligeiramente menor que $\frac{(|V|)^2}{2}$;
- Um grafo é **regular** se todos os vértices possuírem o mesmo grau (ou k-regular , para grau k);
- O número de arestas em um grafo k-regular é $|V|\frac{k}{2}$
- Um grafo é **denso** se o seu tamanho for proporcional ao quadrado do número de vértices ($|V| + |E| \propto |V|^2$), e é esparso se $(|V| + |E|) \propto |V|$.
- O grafo $H = G(V', E')$ é um **subgrafo** de $G = (V, E)$ se $V' \subseteq V$ e $E' \subseteq E$.
- O grafo $H = G(V', E')$ é um **subgrafo gerador** de $G = (V, E)$ se H for um subgrafo de G e $V' = V$.

  Exemplo:

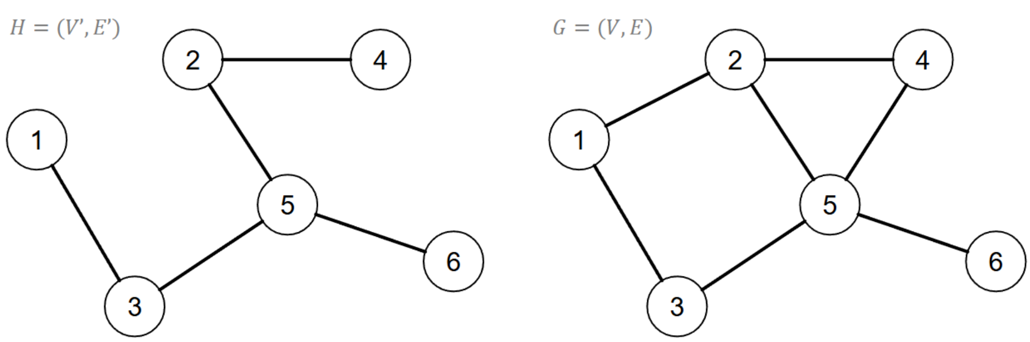


Figura 18: Exemplo de subgrafo gerador.

- O grafo $H = G(V', E')$ é um **grafo induzido** de $G = (V, E)$ se E' for definido por todas as arestas de E adjacentes a um par de vértices V'.

  Exemplo:

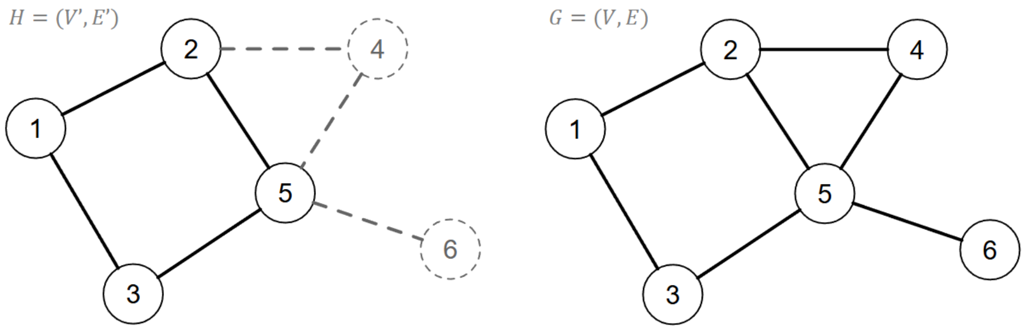


Figura 19: Exemplo de grafo induzido (os vértices escolhidos foram $\{1, 2, 3, 5\}$ e as arestas(e vértices) que não são desses vértices não aparecem no subgrafo induzido).

- O grafo $H = G(V', E')$ é um **grafo próprio** de $G = (V, E)$ se $H \subset G$.

  Exemplo:

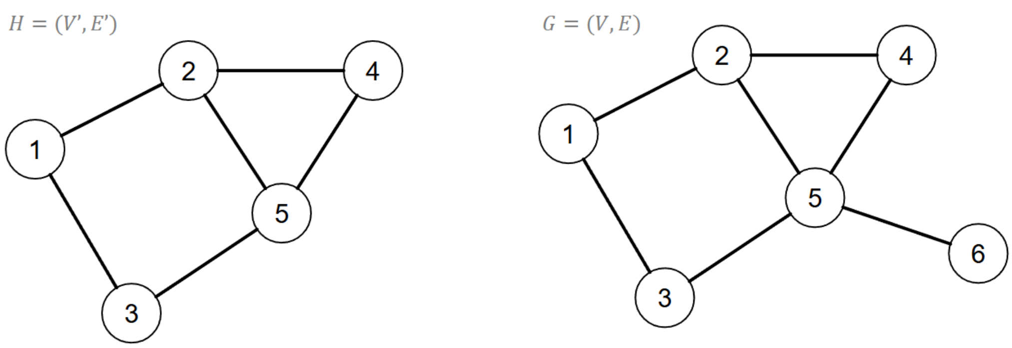


Figura 20: Exemplo de grafo próprio (note que é $\subset$, não $\subseteq$. Então, um subgrafo próprio é um subgrafo menor, e não igual ao grafo original).

- Um **caminho** P em $G(V, E)$ consiste em uma sequência de n vértices, finita e não vazia tal que v_{i+1} é adjacente a v_i.
- Um caminho é **simples** se não possuir vértices repetidos.
- Um caminho é **fechado** se $v_1 = v_n$.
- O **comprimento** de um caminho é definido pelo número de arestas do caminho.
- Um grafo $G = (V, E)$ é **conexo** se para qualquer par de vértices existe um caminho em G.
- Quando um grafo não é conexo podemos segmentá-lo em **componentes conexos** (um par está no mesmo componente se existe um caminho).
- Um grafo $G(V, E)$ é uma **árvore** se G for conexo e acíclico (possui $|V| - 1$ arestas, a remoção de qualquer aresta torna o grafo não-conexo e para todo par de vértices existe um único caminho);
- Um grafo $G(V, E)$ é uma **floresta** se for um grafo acíclico;
- Um grafo é **planar** se puder ser representado graficamente em um plano de tal forma que não haja cruzamento de arestas;
- Um grafo $G(V, E)$ é **bipartido** se os vértices puderem ser divididos em dois conjuntos V_1 e V_2 de forma que toda aresta e_k é incidente em (v_i, v_j) tal que $v_i \in V_1$ e $v_j \in V_2$;
- Um grafo $G(V, E)$ é **orientado** se as arestas possuirem um sentido. Nesse caso, a nomenclatura que definimos (v_i, v_j) significa que ela começa em v_i e termina em v_j.
- O grau de **sáida** $g_s(v_i)$ é definido pelo número de arestas que saem de v_i. Raciocíno análogo para grau de **entrada** $g_e(v_i)$;
- $\sum_{i=1}^{|V|} g_e(v_i) = \sum_{i=1}^{|V|} g_s(v_i) = |E|$;
- O vértice v_i é uma **fonte** se $g_e(v_i) = 0$;
- O vértice v_i é um **sorvedouro** se $g_s(v_i) = 0$;
- O vértice v_i é **isolado** se for sorvedouro e fonte;

- Um grafo (orientado ou não) é **ponderado** se cada aresta estiver associado a um peso;

2.2 Estruturas de dados para representar grafos

Dependendo do problema, a escolha da estrutura pode variar, e, em geral, usamos duas formas de implementar essa representação:

2.2.1 Matriz de adjacência

Consiste em um matriz quadrada A de ordem $|V|$ cujas linhas e colunas são indexadas pelos vértices de V. Exemplo para grafos orienteados:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

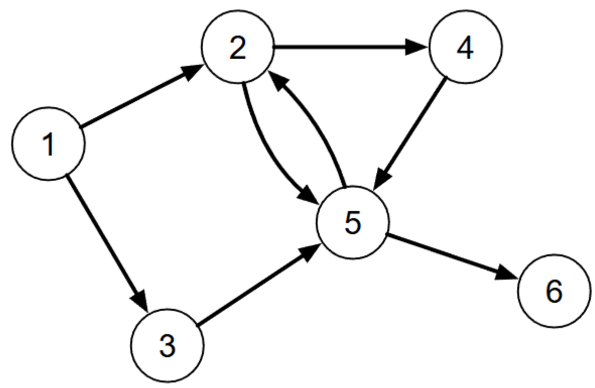


Figura 21: Exemplo de matriz de adjacência para o grafo à direita.

Analogamente, para não orientados:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

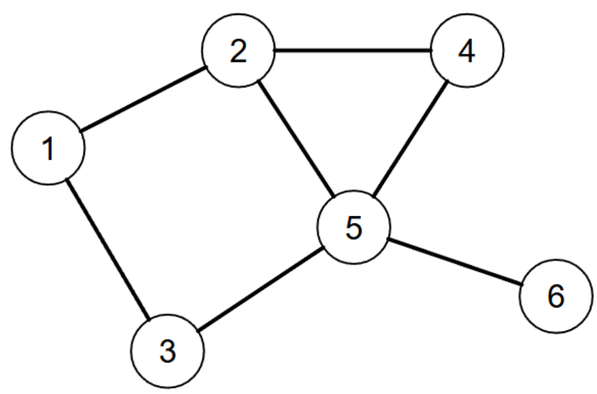


Figura 22: Exemplo de matriz de adjacência para o grafo à direita. Nota: a matriz é simétrica!

A complexidade de acessar(ou verificar) uma aresta é $\Theta(1)$, e claramente conta com uma complexidade de espaço de $\Theta(|V|^2)$. Além disso, o fato da matriz ser simétrica para grafos não-orientados faz com que o tamanho se reduza para a metade, podendo se armazenar apenas a diagonal superior ou inferior da matriz.

2.2.2 Lista de adjacência

Consiste em uma sequência de vértices contendo na estrutura de cada ponteiro para uma lista encadeada com elemento representando as arestas adjacentes ao vértices. Exemplo para grafo dirigido:

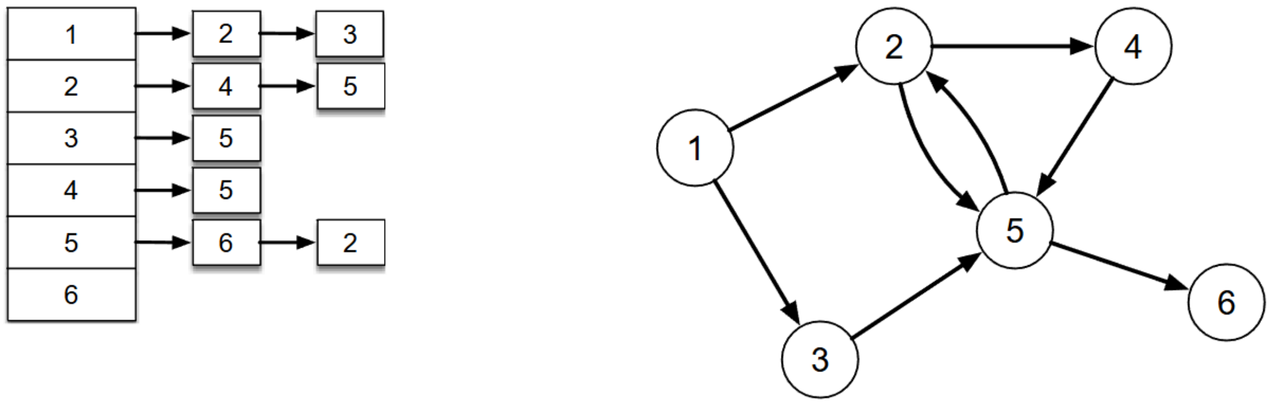


Figura 23: Exemplo da lista de adjacência para o grafo à direita.

Exemplo para grafo não-dirigido:

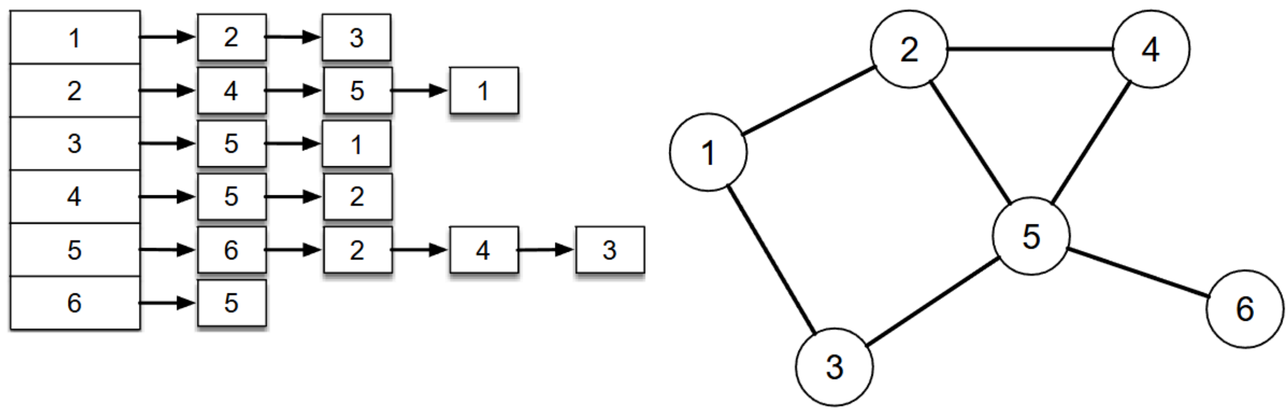


Figura 24: Exemplo da lista de adjacência para o grafo à direita.

A complexidade de acessar o conjunto de arestas de um vértice é $\Theta(1)$ (mas encontrar uma aresta específica é $\Theta(|V|)$ no pior caso). Ainda, uma lista de adjacência exige um espaço $\Theta(|V| + |E|)$

As estruturas de dados do vértice e da aresta podem ser estendidas para armazenar informações específicas do problema.

Nota: Os exercícios passados no slide não serão feitos aqui (pois isso é um “resumo” teórico), e sim no GitHub, na pasta Exercises.