Uma argumentação entre *Backtracking* e *Branch and Bound*

Gustavo Lopes Rodrigues[1], **Rafael Amauri Diniz Augusto**[2]

[1]Instituto de Ciências Exatas e Informática –
Pontifícia Universidade Católica de Minas Gerais(PUC-MG)

Abstract. *Branch and Bound and Backtracking are two strategies in algorithm development. The objective of this project for the discipline of Algorithm Design and Analysis is talk about such strategies indicating, at least, the concepts, when it should be used, in addition to showing examples. We will also present the main similarities and differences between them. Also, if there are similarities or differences with other strategies such as Greedy Approach, Dynamic Programming and Division and Conquer, we will present them in a way to broaden the discussion.*

Resumo. *Branch and Bound e Backtracking são duas estratégias no desenvolvimento de algoritmos. O objetivo deste trabalho feito para a disciplina de Projeto e Análise de Algoritmos é discorrer sobre tais estratégias indicando, pelo menos, os conceitos, quando se deve utilizar, além de mostrar exemplos. Apresentaremos também as principais semelhanças e diferenças entre elas. Ainda, se houver semelhanças ou diferenças com outras estratégias como Abordagem Gulosa, Programação Dinâmica e Divisão e Conquista, apresentaremos elas de forma a ampliar a discussão.*

1. Informações gerais

Em discussões sobre desenvolvimento de algoritmos, a escolha da estratégia para resolver um problema é fundamental, visto que existem métodos diferentes, que possuem aplicações diferentes, e com complexidades diferentes. Os principais em questão, são o *Backtracking*, e *Branch and Bound*. Ambos são relativamente parecidos em conceito, porém possuem implementações diferentes, o que resulta em diferentes circunstâncias na qual um é preferível ao outro.

Neste artigo, iremos analisar os dois algoritmos, olhando com cuidado para os conceitos para então construir códigos em Python, e então resolver problemas conhecidos da computação. Por fim, iremos dar uma breve olhada em outras estratégias, e comparar com as abordagens mencionadas anteriormente.

2. Backtracking

Backtracking é uma estratégia para resolver problemas recursivamente e incrementalmente, removendo as soluções parciais que falham em ajudar na solução para o problema. Para aplicar essa técnica, a estratégia tenta encontrar a solução utilizando diversos pequenos *checkpoints*, para os quais o programa pode voltar se a iteração atual para a resolução do problema não ajudar a encontrar a solução final.

Essa estratégia é viável para resolver problemas modulares que requerem muita tentativa e erro, já que ele remove "caminhos" inválidos, e isso salva muito tempo de processamento.

2.1. Problema do labirinto

Considere o labirinto abaixo:

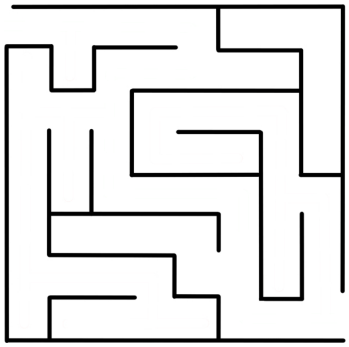

Figura 1. Um exemplo de labirinto

Resolver labirintos é uma aplicação clássica da estratégia de *backtracking*, pois envolve diversas possibilidades de caminhos e é fácil integrar *checkpoints* no problema.

Imaginando que podemos traduzir a imagem acima para uma matriz onde cada ponto de decisão do labirinto é uma célula, que todas células têm ponteiros que apontam para quatro direções: cima, baixo, esquerda e direita, e que esses ponteiros podem levar a outras células ou ser do tipo NULL (indicando que não há uma célula ligada naquela direção), a estratégia de Backtracking pode ser usada para resolver o problema da seguinte forma:

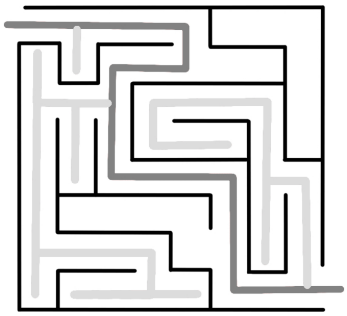

Figura 2. O mesmo labirinto de antes, porém com sua solução destacada

Admitindo que esse algoritmo busca os caminhos disponíveis na ordem [cima, esquerda, baixo, direita], os caminhos destacado em cor leve são caminhos percorridos pelo algoritmo e que foram detectados como caminhos que não contribuem para a solução. A ideia é preservar os checkpoints "certos" que foram percorridos e voltar ao último checkpoint correto ao invés de salvar os caminhos "errados" e recomeçar o percurso desde o início. Dessa forma, apenas os checkpoints "certos" vão ser mantidos e o caminho correto eventualmente será encontrado, sem a necessidade de recalcular e levar em conta os caminhos que já sabemos que são incorretos.

Algorithm 1 Backtracking

```
1:  procedure FIND_PATH(x,y)
2:      grid[x][y] == visited
3:      for i to 4 do
4:          if UP(x,y+1) then
5:              find_path(x,y+1)
6:          if LEFT(x-1,y) then
7:              find_path(x-1,y)
8:          if DOWN(x,y-1) then
9:              find_path(x,y-1)
10:         if RIGHT(x+1,y) then
11:             find_path(x+1,y)
```

3. Branch and Bound

Branch and Bound é uma técnica para algoritmos que visa melhorar o tempo de processamento ao eliminar as soluções-candidato que claramente não se aplicam e/ou não ajudam a resolver o problema. Esse método geralmente é aplicado em problemas que têm soluções finitas, no qual as soluções podem ser representadas como uma sequência de opções.

A primeira parte da técnica Branch and Bound pede que o algoritmo trate as possíveis soluções como se estivessem em uma estrutura de árvore, onde os nós da árvore são utilizados como partes de possíveis soluções garante que todas soluções sejam consideradas.

3.1. Problema da Mochila

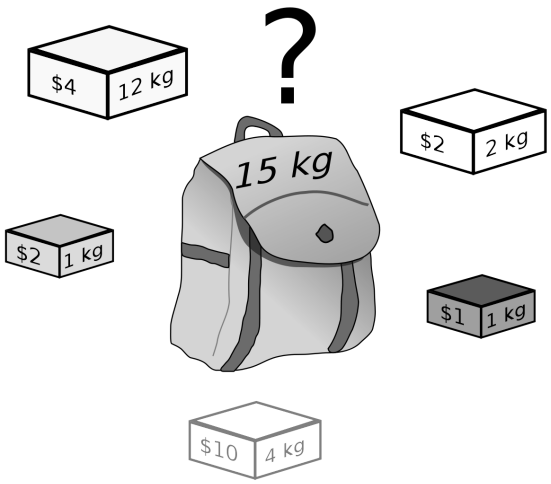

Figura 3. Ilustração do problema da mochila

O problema escolhido para mostrar essa estratégia é o problema do 0/1 Knapsack. A ideia por trás dele é um ladrão que quer roubar uma casa, mas sua mochila so aguenta carregar um peso P1. Existem vários items na casa, cada um com seu valor V e peso P2.

Escolha os items específicos que vão maximizar o valor roubado pelo ladrão de forma que a soma dos pesos dos items nunca seja maior que P1.

Podemos ilustrar as combinações de items em uma estrutura de árvore, de forma que cada nó é um item que o ladrão pode roubar, e um caminho percorrendo N nós significa um conjunto de N items.

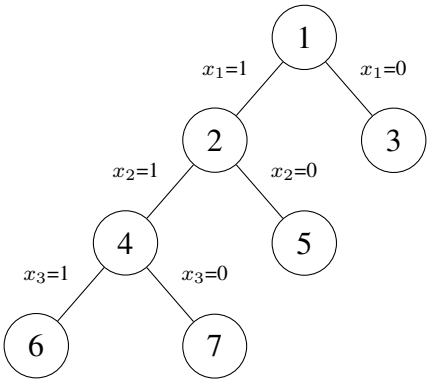

Figura 4. Space State Tree do problema do Knapsack

1	1	1	1	4	5	6	7
c	-38	-38	-32	-38	-36	-32	-38
u	-32	-32	-22	-32	-22	-38	-38

Tabela 1. Valores de C(custo) e U(upperBound)

Algorithm 2 Knapsack

```
1:  procedure KNAPSACK(S,F,LB)
2:      if LB < p(S) then
3:          LB ← p(S)
4:      A ← UpperBound(F, c − w(S))
5:      if A + p(S) ≥ LB then
6:          return
7:      for i ∈ F do
8:          B ← UB_i
9:          if A + p(S) ≥ LB then
10:             F ← F_i
11:         else
12:             break
```

4. Backtracking Vs Branch and Bound

Para terminar esta parte de nossa análise, iremos destacar quatro pontos importantes para comparar ambas estratégias.

- Ambos algoritmos usam uma árvore de estados(state space tree) para encontrar a solução, porém, enquanto Backtracking procura por ela exaustivamente até encontrar a solução, Branch and Bound é forçado a passar pela árvore inteira para encontrar a solução otimizada.
- A pesquisa feita na árvore de estados do Backtracking utiliza um DFS(Depth first search), enquanto que o Branch and Bound pode utilizar DFS, ou até mesmo BFS(Breadth first search)
- Como Backtracking não precisa percorrer por todas as possíveis soluções, esta abordagem é mais eficiente do que Branch and Bound, que precisa percorrer todas as ramificações da árvore de estados.
- O resultado do algoritmo de Backtracking pode ser a solução optimizada, porém existem problemas na implementação de backtracking em que o resultado será uma solução sub-optimizada. Como Branch and Bound varre procurando todas as soluções, este consegue achar a solução otimizada com maior frequência.

5. Outras estratégias

Com as principais estratégias discutidas, fica claro as diferenças e vantagens de ambas estratégias. Agora iremos discutir outras possíveis abordagens para resolução de algoritmos, e comparar com as já discutidas.

5.1. Abordagem gulosa

Abordagem Gulosa é uma estratégia que tenta resolver o problema fazendo a escolha ótima localmente em cada fase com o intuito de encontrar uma ótima global que resolve o problema por completo.

Para o uso dessa abordagem, duas propriedades são de suma importância:

- **Greedy Choice Property**: Uma solução global otimizada pode ser obtida, selecionado uma solução ótima local.
- **Optimal Substructure**: uma solução ótima pode ser construída, usando a solução ótima para os seus subproblemas.

Um exemplo de abordagem gulosa é o Algoritmo de Dijkstra.

5.1.1. Algoritmo de Dijkstra

Dado um grafo qualquer e dois pontos: partida e chegada, encontre o menor caminho possível. Para resolver esse problema, Dijkstra cria o que é chamado de *Shortest path tree*(spt) ou a árvore de caminho mais curto. A cada passo do algoritmo, a **spt** adiciona uma vértice não visitada, e então atualiza a distância das adjacências da vértice que foi adicionada. O algoritmo irá terminar quando a **spt** conter informação de todas as vértices.

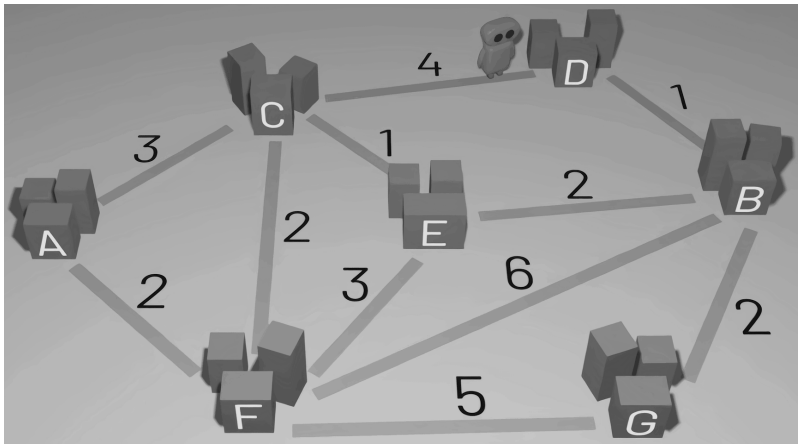

Figura 5. Um caso comum em encontrar o menor caminho entre dois pontos, é encontrar a menor distância entre duas cidades

Algorithm 3 Dijkstra

```
1:  procedure DIJKSTRA(origin,destination)
2:      for each v ∈ G.vertices do
3:          distance[v] = ∞
4:      A ← G.vertices
5:      distance[origin] = 0
6:      for j = 1 in | G.vertices | -1 do
7:          u := Extract-Min(A)
8:          A.pop(u)
9:          for each v ∈ Neighbors(u) do
10:             Update(distance,v)
11:     return distance[destination]
```

5.1.2. Algoritmo Guloso Vs Backtracking

- Por sua natureza, a estratégia Gulosa pode não alcançar imediatamente o resultado global ótimo, um efeito que também está presente nos algoritmos que utilizam *Backtracking*
- Devido a natureza da estratégia de backtracking, tais algoritmos costumam ter implementações recursivas, enquanto que algoritmos gulosos tentam buscar a melhor escolha local de forma iterativa.

5.1.3. Algoritmo Guloso Vs Branch and Bound

- Algoritmos gulosos costumam ter resultados mais rápidos do que técnicas de *Branch and Bound* para problemas com menos *input*, mas à medida que a entrada cresce
- *Branch and Bound*, assim como *Backtracking*, consegue voltar a estados anteriores da árvore para encontrar a solução, enquanto isso, algoritmos gulosos tentam buscar a melhor escolha local de forma iterativa e não dividem o problema em *"checkpoints"*

5.2. Divisão e conquista

Divisão e Conquista é uma técnica que pode ser dividida em três partes fundamentais: dividir um problema maior recursivamente em problemas menores(**Dividir**), resolver todos os sub-problemas (**Conquistar**) e, por fim, a solução do problema inicial é dada através da combinação dos resultados de todos os problemas menores computados(**Combine**).

Problemas que usam divisão e conquista são característiscos em possuir um princípio chamado **Overlapping subproblems**, ou seja, o problema pode ser divido em subproblemas, que são reutilizados várias vezes e que não geram novos subproblemas.

Exemplos de problemas conhecidos que usam desta estratégia são o problema de ordenação interna(usando quicksort ou mergesort) e o problema dos pares de pontos próximos.

5.2.1. Pares de pontos próximos

Dado um conjunto de N pontos em um espaço, é preciso encontrar os dois pontos do conjunto que possuem a menor distância um do outro. Este problema possui mais de uma implementação, porém uma das mais eficientes é utilizando divisão e conquista.

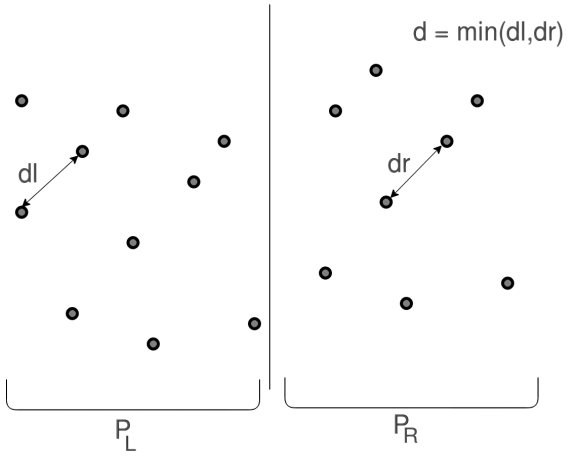

Figura 6. Uma representação visual da separação dos pontos

Supondo que a entrada para o problema é um conjunto de pontos com o eixo y já ordenado, encontramos o ponto do meio, separamos o conjunto ao meio, e então recursivamente descobrimos as menores distâncias entre os conjuntos separados, para então achar a solução geral.

Algorithm 4 Closest pair of points

```
1: procedure DIVIDE-AND-CONQUER(P,Q)
2:     if n ≥ 3 then
3:         bruteForce(P,n)
4:     midPoint ← P[n/2]
5:     P_l ← P[1...n/2 − 1]
6:     Q_l ← P[1...n/2 − 1]
7:     P_r ← P[n/2...n]
8:     Q_r ← P[n/2...n]
9:     d_l ← DIVIDE − AND − CONQUER(P_l, Q_l)
10:    d_r ← DIVIDE − AND − CONQUER(P_r, Q_r)
11:    d ← min(d_l, d_r)
12:    strip_p ← []
13:    strip_q ← []
14:    lr ← P_l + P_r
15:    for i to n do
16:        if abs(lr[i].x − midPoint.x) < d then
17:            stripP.append(lr[i])
18:        if abs(Q[i].x − midPoint.x) < d then
19:            stripQ.append(Q[i])
20:    sort(stripP, key = y)
21:    min_p ← min(d, stripClosest(striP), | stripP |, d)
22:    min_q ← min(d, stripClosest(striQ), | stripQ |, d)
23:    return min(min_p, min_q)
```

5.2.2. Divisão e Conquista Vs Backtracking

- Devido à maneira como ambos são conceitualizados, ambos possuem a natureza de serem recursivos.
- Em divisão e conquista, precisamos analisar a entrada inteira do usuário para resolver o problema, enquanto que Backtracking pode ou não analisar a todas as possibilidades dentro da entrada do usuário

5.2.3. Divisão e Conquista Vs Branch and Bound

- Enquanto divisão e conquista divide a entrada do usuário(Ex: divide o vetor de números para ordená-los) para resolver os sub-problemas e então resolver a combinação desses problemas, a outra estratégia divide o espaço da solução para o problema.
- Tanto Branch and Bound e Divide and Conquer são estratégias que partem dos mesmos princípios: ambas querem dividir o problema em partes menores para resolver o problema todo.

5.3. Programação Dinâmica

Programação dinâmica é uma estratégia de construção de algoritmos que possui uma grande eficácia na resolução de problemas de otimização combinatória. Em outras palavras, essa estratégia é ideal quando o problema original é divido em subproblemas, que se repetem, logo podem ser memorizados para evitar recálculos.

Antes de aplicar essa técnica, o problema precisa atender duas propriedades: um problema com subproblemas **(Overlapping subproblems)**, propriedade já vista em Divisão e conquista e uma subestrutura ideal **(Optimal substructure)**, propriedade já vista em algoritmos gulosos.

Para melhor exemplificar esses conceitos, vamos analisar um clássico problema da computação, e também um caso perfeito para ilustrar a estratégia de programação dinâmica.

5.3.1. Fibonacci

Descoberto pelo matemático italiano do século 12, Leonardo Fibonacci, a sequência Fibonacci é uma sucessão de números que aparece codificada nos mais diversos fenômenos da natureza(espiral de galáxias, ciclones, conchas e vários outros). Na computação, temos o problema de criar um algoritmo capaz de calcular o número da sequência de Fibonacci.

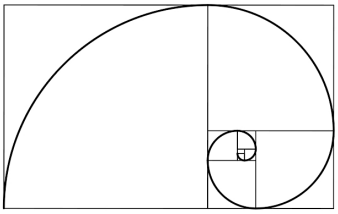

Figura 7. Uma imagem ilustrativa da sequência Fibonacci

A ideia para resolução desse problema é bem simples: sabendo que a sequência inicia com 0 e 1, os próximos números serão sempre a soma dos dois números anteriores. A seguir temos um exemplo de uma implementação ingênua do algoritmo da sequência de Fibonacci.

Algorithm 5 Naive Fibonacci

```
1: procedure FIB(n)
2:     if n ≥ 2 then
3:         return n
4:     return fib(n-1) + fib(n-2)
```

Esta implementação a primeira vista não parece ter nenhum problema. Porém, é quando tentamos executar esse código que o problema se mostra. O código consegue calcular o 6°, 7° e 8° número da sequência, porém o código não consegue calcular o 50° número.

Para entender a natureza desse problema precisamos entender a complexidade desse código, logo, precisamos da árvore de recursão gerada pelo código:

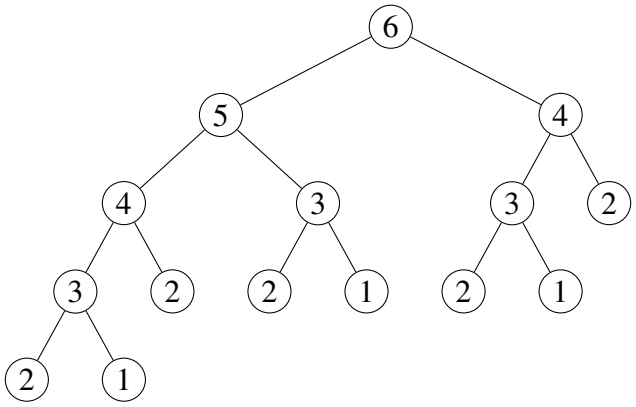

Figura 8. encontrando o 6° número da sequência Fibonacci

A primeira coisa a se notar é a complexidade do algoritmo. A árvore formada possui uma altura N(o número que queremos achar da sequência), e a cada novo nível da árvore, formamos no máximo, dois novos nós, logo podemos concluir que a complexidade é de $O(2^n)$, justificando também, porque não conseguimos calcular o 50° número, já que este levaria um tempo muito alto.

Então agora a pergunta fica: como podemos otimizar essa árvore, de forma que o algoritmo consiga ter um tempo linear? Voltando à árvore, podemos observar um padrão recorrente.

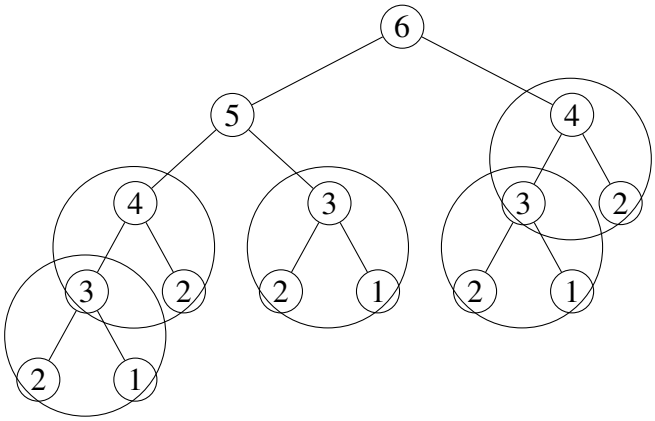

Figura 9. Árvore de recursão, com os sub-problemas repetidos circulados

Com esse segundo desenho, fica mais evidente que existe passos da recursão que estão repetidos, ou seja, o programa acaba resolvendo mais de uma vez os mesmos problemas de forma desnecessária. Em conclusão, para resolver esses problemas, o ideal seria uma estrutura que possa guardar resultados de recursões já feitas para usar depois quando o mesmo problema for encontrado, resultando em uma árvore de recursão da seguinte forma:

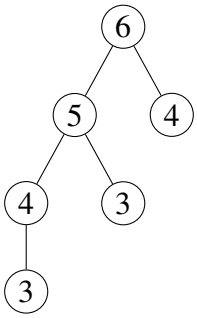

Figura 10. Árvore de recursão otimizada

Com a otimização percebemos que agora a complexidade do código melhorou drasticamente, tendo um resultado de $O(2n)$, visto que é resultado da altura da árvore multiplicado por 2, que pode ser simplificado para apenas $O(n)$.

Eis então agora o código otimizado:

Algorithm 6 Optimized Fibonacci

```
1: procedure FIB(n,memo)
2:     if n ≥ 2 then
3:         return n
4:     if n in memo then
5:         return memo[n]
6:     memo ← fib(n-1) + fib(n-2)
7:     return memo[n]
```

5.3.2. Programação dinâmica Vs Backtracking

- Programação dinâmica procura a solução ótima para o problema, enquanto que backtracking tenta sistemáticamente satisfazer um critério, podendo ou não varrer por toda a árvore, assim como pode ou não encontrar a solução ótima.
- Ambos algoritmos possuem conceitos que tem natureza recursiva. Entretanto programação dinâmica ainda pode ser implementado de forma iterativa, usando bottom-up (tabulation).

5.3.3. Programação dinâmica Vs Branch and Bound

- Programação dinâmica tenta minimizar os passos para resolver o problema, evitando subproblemas já resolvidos. Enquanto isso, branch-and-bound tenta sistemáticamente enumerar todos os candidatos possíveis na árvore de soluções possíveis,
- Branch and Bound permite backtracking, para selecionar as melhores soluções dentro da árvore de estados, enquanto programação dinâmica não utiliza esse tipo de recursão. Como resultado, Branch and Bound pode se demonstrar mais lento que programação dinâmica em alguns casos.

6. Conclusão

No fim das contas, *Backtracking* e *Branch and Bound* são estratégias bem semelhantes, porém, como foi demonstrado neste artigo, eles não substituem um ao outro. Devido a isso, cabe ao programador analisar na qual este quer fazer a implementação de um algoritmo de otimização, pensando qual são as necessidades do contexto atual.

Isso também vale para as outras estratégias mencionadas, todas elas possuem suas aplicações específicas, por mais que possuem propriedades semelhantes umas as outras.

Referências

Datta, S. (2020). Branch and bound algorithm. https://www.baeldung.com/cs/branch-and-bound. Acessado em 28 de Novembro de 2021.

Desconhecido (2020). Divide and conquer algorithm. https://www.programiz.com/dsa/divide-and-conquer. Acessado em 28 de Novembro de 2021.

Desconhecido (2021). Closest pair of points using divide and conquer algorithm. https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/?ref=lbp. Acessado em 28 de Novembro de 2021.

freeCodeCamp.org (2020). Dynamic Programming - Learn to Solve Algorithmic Problems & Coding Challenges. https://www.youtube.com/watch?v=oBt53YbR9Kk. Acessado em 28 de Novembro de 2021.

Toffolo, T. and Carvalho, M. A. (2021). Backtracking. `http://www3.decom.ufop.br/toffolo/site_media/uploads/2011-1/bcc402/slides/10._backtracking.pdf`. Acessado em 28 de Novembro de 2021.

Walker, A. (2021). Greedy Algorithm with Example: What is, Method and Approach. https://www.guru99.com/greedy-algorithm.html. Acessado em 28 de Novembro de 2021.

Yoshioka, H. (2018). Knapsacker (python version). https://github.com/irohiroki/knapsacker-py. Acessado em 28 de Novembro de 2021.