



# *Problema do empacotamento (Bin Packing)*

**Disciplina:** Análise e Projeto de Algoritmos

**Alunos:**

- Arthur Moura Bernardo - 201905523
- Felipe Ramos Kafuri - 201905528
- Gabriel Ferreira Silva - 202206970
- Guilherme Faleiros de Siqueira - 201905532
- Pietro Niero Roque - 201905550

# Introdução

Este trabalho visa explorar detalhadamente o problema do Bin Packing, uma questão pertinente na área de otimização combinatória. Ele aborda as características únicas deste problema, analisando detalhes e qual sua função no mundo real. Aprofundamos nas diversas soluções propostas, analisando comparativamente as estratégias implementadas em termos de complexidade, tanto em tempo quanto em espaço. Além disso, o estudo inclui uma avaliação experimental, apresentando dados de desempenho baseados no tempo de execução para cada implementação, permitindo uma compreensão mais aprofundada das eficiências e limitações práticas de cada abordagem.

## Descrição do problema

O problema do empacotamento consiste em resolver um problema clássico de otimização combinatória: em um cenário onde possuímos uma sequência de diversos objetos de tamanhos diferentes e uma quantidade finita de recipientes (bins) de igual capacidade, qual a quantidade mínima de recipientes podemos utilizar para empacotar todos os objetos.

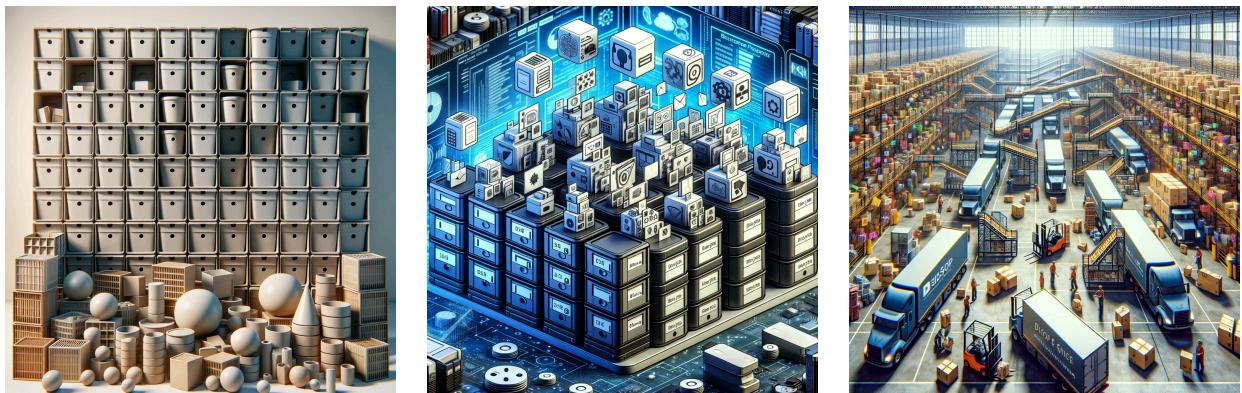
## Variações do problema

O problema é bastante abrangente e amplamente discutido na ciência da computação desde a década de 50, de forma que várias variações foram mapeadas ao longo do tempo, vejamos algumas principais:

- **Empacotamento Linear ou 1D:** essa se trata da forma mais simples do problema, onde cada objeto possui apenas uma dimensão e precisa ser armazenado em diversos recipientes de tamanho igual, de forma que uma simples representação de cada objeto poderia ser números diversos que devem ser distribuídos em recipientes nos quais a soma desses números não deve exceder a capacidade máxima.
- **Empacotamento 2D/3D:** diferentemente do empacotamento linear, neste caso os objetos possuem duas ou três dimensões, bem como os recipientes que também possuem duas ou três dimensões. Essa variação adiciona uma certa complexidade, pois a orientação e disposição desses itens dentro do recipiente se torna relevante.
- **Empacotamento variável:** nesta variação, os recipientes possuem capacidades diferentes, de forma que isso traz uma análise um pouco mais realista, onde muitas vezes esse tipo de otimização ocorre com recipientes não uniformes.

Para fins didáticos, utilizaremos como base nesse trabalho a variação do **problema de empacotamento linear** para demonstrar e comparar os algoritmos propostos e as diferenças entre eles.

## Aplicabilidade do problema



Quando analisamos as imagens acima, obtemos uma compreensão mais concreta da aplicação prática do problema de Bin Packing. Imaginemos, por exemplo, sua aplicação na indústria de transportes: uma empresa logística enfrenta o desafio de carregar um número variado de caixas, cada uma com dimensões distintas, em um conjunto limitado de caminhões de capacidade idêntica. O objetivo é minimizar o número de caminhões necessários para esta tarefa. A implementação de algoritmos eficientes para resolver este problema de otimização traz vantagens significativas:

- **Redução de Custos Operacionais:** A utilização ótima do espaço de carga permite a redução do número de viagens necessárias, impactando diretamente os custos com combustível, tempo de entrega e mão-de-obra.
- **Eficiência na Manutenção de Veículos:** Com menos caminhões em rotação e uma distribuição mais equilibrada de carga, há uma diminuição na frequência e no custo de manutenção dos veículos.
- **Segurança da Carga:** Uma alocação estratégica das caixas dentro do veículo minimiza o movimento durante o transporte, reduzindo o risco de danos aos itens transportados e assegurando a integridade da carga.
- **Otimização Logística e Conformidade Regulatória:** Um planejamento eficaz das rotas de entrega, guiado por soluções de Bin Packing, facilita o cumprimento de regulamentações, como limites de peso e dimensionais, otimizando a eficiência das rotas e garantindo a conformidade com as normas fiscais.

Outro caso de uso interessante para o problema de Bin Packing é o gerenciamento de recursos computacionais em ambientes de Cloud Computing, especificamente, por exemplo, na alocação de espaço em discos rígidos e na otimização de recursos como memória e CPU. Vejamos:

- Discos rígidos têm capacidade limitada e arquivos podem ter tamanhos diversos, dessa forma, o problema de Bin Packing é aplicado na otimização da distribuição desses arquivos em discos rígidos de tamanho limitado, garantindo que o máximo

possível de dados seja armazenado sem a necessidade de adquirir armazenamento adicional, gerando redução de custos e melhorando a eficiência do sistema.

- Em um ambiente de Cloud Computing, normalmente recursos como CPU, memória e armazenamento (disco rígido) são tratados como commodities. Dessa forma, esses recursos disponíveis são distribuídos entre diversas Virtual Machines, que por sua vez são servidas para vários clientes e aplicações ao redor do mundo. O problema de Bin Packing surge para otimizar a distribuição desses recursos de forma eficiente, de modo a minimizar recursos alocados desnecessariamente, consequentemente diminuindo o custo ao evitar manter recursos ociosos.

## Classe do problema

De forma geral, o problema de Bin Packing pode ser enxergado por duas perspectivas: problema de decisão ou problema de otimização, a diferença primordial entre essas duas formas é a maneira como o problema é formulado e o resultado obtido:

- **Decisão:** dado um conjunto de bins de tamanho fixo e um conjunto de objetos, é possível acomodar todos esses objetos sem que a capacidade máxima dos bins seja ultrapassada?
- **Otimização:** dado um conjunto de bins de tamanho fixo e um conjunto de objetos, qual o número mínimo de bins para que todos os objetos sejam acomodados sem ultrapassar a quantidade máxima de bins.

No geral, apesar de exemplificarmos essa distinção com o problema de Bin Packing, boa parte de problemas de otimização possuem um problema de decisão relacionado, e isso será importante adiante para conseguirmos provar que a classe do problema de otimização de Bin Packing.

Na prática, o problema de otimização acaba sendo mais explorado, uma vez que tal abordagem é comumente mais utilizada em aplicações reais, como por exemplo na indústria de transportes, citada anteriormente.

É preciso deixar claro a diferença entre essas duas abordagens pois isso irá interferir em como o problema de Bin Packing é classificado, caso seja uma abordagem de **decisão**, a classe do problema será **NP-completo**, caso seja um problema de **otimização**, a classe do problema será **NP-difícil**. Logo adiante, iremos provar como obtivemos essa classificação, mas antes é preciso esclarecer e trazer a diferença entre essas classificações.

## Classes de problema (P vs NP)

De maneira geral, podemos definir problemas de decisão com as classes P ou NP, porém, como visto anteriormente, existem algumas variações da classe NP que também serão abordadas aqui. Vejamos:

- **P:** se trata de uma classe de problemas que podem ser resolvidas em tempo polinomial por uma máquina de turing determinística. De maneira geral, são problemas que podem ser resolvidos com algoritmos eficientes, como por exemplo, algum algoritmo que possui complexidade  $n$ , onde  $n$  é a entrada do algoritmo.

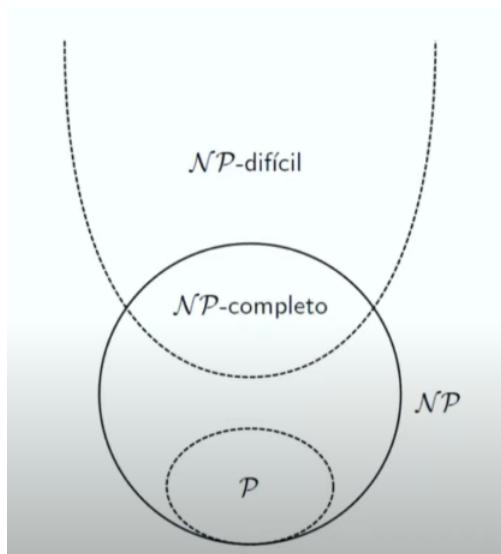
- **NP:** é uma classe de problemas que não possuem um algoritmo capaz de resolver o problema de forma polinomial, ou seja, eficiente. Porém, é possível que, dada uma solução, podemos verificar em tempo polinomial que a resposta está correta ou não.

Dessa forma, podemos concluir que os problemas P também são NP, pois uma vez que é possível resolver um problema em tempo polinomial, também é possível verificar que sua resposta em tempo polinomial. A recíproca ainda não é verdadeira, se alguém conseguir encontrar algum algoritmo que resolva um problema NP em tempo polinomial, haverão indícios que qualquer problema NP pode ser resolvido em tempo polinomial, portanto, esse é um dos grandes desafios da computação nos dias de hoje.

Por outro lado, temos classes que possuem características específicas, vejamos:

- **NP-completo:** um problema X é dito NP-completo se houver um problema sabidamente NP-completo Y que seja redutível em tempo polinomial ao problema X. Ou seja, é possível “transformar” o problema Y no problema X, de forma que a resolução de ambas, dada uma entrada equivalente, tenha a mesma resposta.
- **NP-difícil:** um problema de otimização X é dito NP-difícil se houver um problema de decisão NP-completo Y que seja redutível em tempo polinomial à uma versão de problema de decisão do problema X. É importante ressaltar que, diferentemente das classes NP e NP-completo, a classe NP-difícil não se aplica somente a problemas de decisão, mas engloba também problemas de otimização.

Podemos enxergar a relação de todas as classes explicitadas anteriormente através deste diagrama:



## Prova da classe de problema de Bin Packing

Como visto por definição anteriormente, chegamos à conclusão de que a melhor forma de provar que o problema de Bin Packing é NP-difícil seria escolhendo algum sabidamente NP-completo e fazendo a redução de seu problema em tempo polinomial.

## 1. Premissas

Para realizar essa prova, escolhemos o [Subset Sum Problem \(SSP\)](#), que é um problema conhecidamente da classe NP-completo e é comumente utilizado para realizar esta prova do problema de Bin Packing. Vejamos a estrutura do SSP:

**Descrição:** dado um conjunto de números inteiros  $S$  de tamanho  $n$  e um valor alvo  $T$ , é possível que exista algum subconjunto de  $S$  em que a soma desse subconjunto seja igual ao valor de  $T$ ?

**Exemplo:** dado um conjunto  $S = \{1, 2, 3, 4, 5\}$  e  $T = 7$ , existe um subconjunto cuja soma é  $T$ ? Sim, existe o subconjunto  $R = \{2, 5\}$  cuja a soma é 7.

Para essa prova, utilizaremos o **problema de decisão** de Bin Packing, recapitulando:

**Descrição:** dado um número fixo de bins  $n$  de tamanho  $T$  e um conjunto de objetos  $S$ , é possível acomodar todos esses objetos sem que a capacidade máxima dos bins seja ultrapassada?

## 2. Redução dos problemas

Com a descrição desses problemas em mente, precisamos reduzir o SSP a um problema de Bin Packing, vejamos os passos:

- Ambos possuem um conjunto de números inteiros, naturalmente no SSP o conjunto  $S$  de números inteiros, por outro lado, no problema de Bin Packing existe um conjunto de itens que, utilizando a sua versão linear, podemos abstrair como também um conjunto de números inteiros onde cada número representa a dimensão única de um determinado objeto. Portanto, podemos equiparar ambos os conjuntos.
- Ambos os problemas (SSP e Bin Packing) possuem um número inteiro fixo em que, no caso do SSP se trata do valor alvo  $T$ , e no caso do Bin Packing, se trata do tamanho de cada bin. Onde em ambos os casos devemos encontrar/acomodar algum subconjunto do conjunto  $S$  em que a soma de seus elementos não ultrapasse o valor inteiro mencionado anteriormente.
- Nesse caso, podemos assumir que no problema de Bin Packing o número de bins será igual a 1.

## 3. Exemplificando a redução

No contexto do SSP, consideramos um conjunto  $S = \{1, 3, 5, 7, 11\}$  com um valor alvo  $T = 12$ . Ao aplicar este cenário ao problema de Bin Packing, transformamos  $S$  nos tamanhos dos itens a serem empacotados e definimos um único bin com capacidade igual a  $T$ . A solução do problema de Bin Packing neste caso implica diretamente em uma solução

para o SSP. Se conseguirmos alocar um subconjunto de S dentro do bin de capacidade 12 de tal modo que ele fique completamente preenchido, isso significa que a soma dos tamanhos dos itens (elementos do subconjunto) é exatamente igual a 12.

Consequentemente, isso prova a existência de um subconjunto em S cuja soma é igual ao valor alvo T do SSP. Assim, uma solução para o problema de Bin Packing neste contexto específico não apenas resolve o problema de acomodação, mas também valida a condição do SSP, estabelecendo uma conexão direta entre a resolução dos dois problemas.

## 4. Conclusão

Através da redução do SSP para o problema de decisão de Bin Packing, estabelecemos uma relação direta entre a resolução dos dois problemas. Esta redução demonstra que, se pudermos resolver o problema de Bin Packing, também seremos capazes de resolver o SSP. Considerando que o SSP é reconhecido como um problema NP-completo, a capacidade de transformá-lo no problema de decisão de Bin Packing sugere fortemente que este último compartilha a mesma complexidade computacional. Portanto, concluímos que o problema de decisão de Bin Packing também é **NP-completo**.

Além disso, é importante destacar a distinção entre problemas de decisão e de otimização neste caso, especificamente. Enquanto o problema de decisão de Bin Packing é classificado como NP-completo, a versão de otimização do Bin Packing, cujo objetivo é minimizar o número de bins necessários para acomodar todos os itens, é considerada **NP-difícil**. Isso se deve ao fato de que os problemas de otimização, em geral, não se enquadram diretamente nas categorias de complexidade NP, como NP-completo ou NP, mas são classificados como NP-difíceis quando sua versão de decisão associada é NP-completa. Essa distinção é crucial, pois ressalta que, embora a verificação de uma solução ótima possa ser complexa, o processo de encontrar essa solução ótima em si apresenta desafios computacionais ainda mais significativos.

## Implementação de Algoritmos

Nesta seção, iremos demonstrar a implementação de algoritmos que resolvem o Bin Packing utilizando diversas técnicas de programação, bem como a complexidade de cada um desses algoritmos.

É importante salientar que como o problema de Bin Packing se trata de um problema da classe NP, os algoritmos a seguir apresentam soluções aproximadas para o problema, dependendo da entrada e das técnicas apresentadas, um terá resultado melhor que o outro e alguns casos o algoritmo implementado dará a solução ótima do problema. De forma que para cada algoritmo fornecemos duas versões do código: uma no contexto do problema de otimização e outra no contexto de decisão.

Para a implementação dos algoritmos escolhemos a linguagem de programação Python, pelo amplo material disponível sobre implementação de algoritmos com a linguagem, sua simplicidade sintática e facilidade de configuração e execução dos scripts implementados.

O código fonte dos algoritmos está presente nesta documentação, como anexo juntamente com esse trabalho e disponível através de um repositório do GitHub, no qual o link será fornecido em uma seção ao final desta documentação.

## First-Fit Decreasing

Nesta seção, iremos apresentar o algoritmo de First-Fit Decreasing, que é classificado como um algoritmo guloso e utiliza heurísticas devido à sua abordagem de tomar decisões ótimas locais com a esperança de encontrar uma solução global ótima ou próxima do ótimo. Especificamente, ele ordena os itens em ordem decrescente de tamanho e, em seguida, itera sobre cada item, colocando-o no primeiro recipiente que tem espaço suficiente para acomodá-lo. Essa abordagem é gulosa porque toma a decisão imediata de colocar o item no primeiro recipiente adequado, sem considerar as consequências dessa decisão para as situações futuras. A heurística de ordenar os itens em ordem decrescente antes da alocação é baseada na intuição de que acomodar os itens maiores primeiro pode levar a uma utilização mais eficiente do espaço dos recipientes, embora não haja garantia de que essa abordagem leve à solução ótima em todos os casos. Portanto, o algoritmo é eficaz para muitas situações do problema, mas como se baseia em decisões gulosas e heurísticas, não pode garantir a solução ótima em todos os cenários.

### Problema de otimização

#### Implementação

A seguir, veja a implementação em Python do algoritmo de otimização utilizando a heurística de First-Fit Decreasing:

```
import time

# Recebe como parâmetro uma lista de itens e a capacidade de cada bin
def bin_packing(items, bin_capacity):
    # Ordena os itens em ordem decrescente
    sorted_items = sorted(items, reverse=True)

    bins = []

    # Passa por todos os itens da lista e tenta acomodar ele em um bin
    # já existente ou cria um novo bin caso seja possível
    for item in sorted_items:
        placed = False
        for bin in bins:
            if sum(bin) + item <= bin_capacity:
                bin.append(item)
                placed = True
        if not placed:
            bins.append([item])
```

```

        placed = True
        break

    if not placed:
        bins.append([item])

    # Retorna o número de bins necessários
    return len(bins)

```

## Complexidade de tempo

Para analisar a complexidade desse algoritmo, vejamos a tabela com a contagem de instruções em termos de  $n$ , onde  $n$  é o tamanho da entrada do conjunto de itens fornecido:

Linhas de Código	Número de instruções
sorted_items = sorted(items, reverse=True)	$n \log n$
bins = []	1
for item in sorted_items:	$n$
placed = False	1
for bin in bins:	$n$
if sum(bin) + item <= bin_capacity:	1
bin.append(item)	1
placed = True	1
break	1
if not placed:	1
bins.append([item])	1
return len(bins)	1
<b>Total de instruções:</b>	$n^2 + n \log n + 9$
<b>Complexidade:</b>	$O(n^2)$

## Complexidade de espaço

Para analisarmos a complexidade de espaço, vejamos que naturalmente o espaço para a variável `sorted_items` requer  $O(n)$  de espaço adicional, uma vez que ela terá o mesmo tamanho da entrada fornecida. Se analisarmos a variável `bins`, podemos perceber

que no pior caso, cada item estará em um bin diferente, portanto, requer também  $O(n)$  de espaço adicional. Portanto, podemos concluir que a complexidade de espaço desse algoritmo é  $O(n)$ .

## Problema de decisão

### Implementação

A seguir, veja a implementação em Python do algoritmo de decisão utilizando a heurística de First-Fit Decreasing:

```
import time

# Recebe como parâmetro uma lista de itens, a capacidade de cada bin e o
# número de bins
def bin_packing(items, bin_capacity, number_of_bins):
    # Preenche todos os bins existentes peso 0
    bins = [0] * number_of_bins

    # Ordena os itens em ordem decrescente
    sorted_items = sorted(items, reverse=True)

    for item in sorted_items:
        placed = False
        for i in range(number_of_bins):
            if bins[i] + item <= bin_capacity:
                bins[i] += item
                placed = True
                break
        if not placed:
            return False

    # Retorna verdadeiro caso todos os itens tenham sido acomodados
    return True
```

### Complexidade de tempo

Para analisar a complexidade desse algoritmo, vejamos a tabela com a contagem de instruções em termos de  $n$ , onde  $n$  é o tamanho da entrada do conjunto de itens fornecido:

Linhas de Código	Número de instruções
<code>bins = [0] * number_of_bins</code>	1
<code>sorted_items = sorted(items, reverse=True)</code>	$n \log n$

for item in sorted_items:	n
placed = False	1
for i in range(number_of_bins):	n
if bins[i] + item <= bin_capacity:	1
bins[i] += item	1
placed = True	1
break	1
if not placed:	1
return False	1
return True	1
<b>Total de instruções:</b>	<b><math>6n^2 + n \log n + 2</math></b>
<b>Complexidade:</b>	<b><math>O(n^2)</math></b>

### Complexidade de espaço

Ao analisar a complexidade de espaço do algoritmo de empacotamento de bin (bin packing), encontramos que é  $O(m + n)$ . O algoritmo utiliza duas listas principais: uma para os bins ( $m$ ) e outra para os itens ordenados ( $n$ ). A lista de bins tem uma complexidade de espaço  $O(m)$  e a lista de itens ordenados tem complexidade  $O(n)$ . Mesmo no pior caso, a complexidade de espaço do algoritmo permanece a soma dessas duas, ou seja,  $O(m + n)$ .

## First-Fit Decreasing com divisão e conquista

Nesta seção, faremos uma implementação um pouco diferente do algoritmo de First-Fit Decreasing apresentada anteriormente, para isso utilizaremos divisão e conquista para exemplificar como é possível utilizar essa estratégia no contexto de bin packing. Porém é importante salientar: abordagem de divisão e conquista pode não ser ideal para o problema de Bin Packing devido à sua natureza NP-difícil, onde a divisão em subproblemas independentes não é trivial e a combinação de soluções de cada subconjunto pode ser tão complexa quanto o problema original. No Bin Packing, a interdependência dos subproblemas e a dificuldade em otimizar a utilização do espaço nos recipientes de forma global tornam a combinação de soluções de subconjuntos um problema em potencial. Além disso, a divisão e a combinação de itens podem introduzir overhead significativo sem garantir melhorias na eficiência, fazendo com que métodos heurísticos, como o First-Fit tradicional, sejam mais utilizados por oferecerem soluções boas e práticas em tempo razoável, apesar de não garantirem a solução ótima. Portanto, o foco dessa seção é aprofundar um pouco mais na estratégia de divisão e conquista utilizando o Bin Packing como caso de uso.

## Problema de otimização

De maneira resumida, iremos utilizar a estratégia de First-Fit para a resolução de três subconjuntos que são divididos de acordo com seu tamanho, portanto, primeiramente categorizamos itens em pequeno, médio e grande, respectivamente. Após isso, aplicamos o algoritmo de First-Fit em cada subconjunto e executamos uma função para combinar todos os resultados e produzir o resultado final do algoritmo. Vejamos:

### Implementação

```
import time
import random

def bin_packing(items, bin_capacity):

    # Caso base: se há apenas um item ou nenhum, retorna esse número
    if len(items) <= 1:
        return len(items)

    # Ordena os itens em ordem decrescente
    items.sort(reverse=True)

    # Define os limites para classificar os itens como grandes, médios
    # ou pequenos
    large_threshold = bin_capacity * 0.7
    medium_threshold = bin_capacity * 0.4

    # Divide os itens em três subconjuntos baseados nos limites
    # definidos
    large_items = [item for item in items if item > large_threshold]
    medium_items = [item for item in items if medium_threshold < item <=
large_threshold]
    small_items = [item for item in items if item <= medium_threshold]

    # Executa o algoritmo de first fit decreasing para cada subconjunto
    small_items_bins = first_fit_decreasing(small_items, bin_capacity)
    medium_items_bins = first_fit_decreasing(medium_items, bin_capacity)
    large_items_bins = first_fit_decreasing(large_items, bin_capacity)

    # Combina os recipientes dos três subconjuntos
    combined_bins = combine_bins_variable(small_items_bins,
                                           medium_items_bins, large_items_bins, bin_capacity)

    # Retorna o número total de recipientes necessários
    return len(combined_bins)
```

```

# Define a função que implementa o algoritmo First Fit Decreasing
def first_fit_decreasing(items, bin_capacity):
    # Ordena os itens em ordem decrescente
    items.sort(reverse=True)
    bins = []
    # Aloca os itens nos recipientes
    for item in items:
        allocated = False
        for bin in bins:
            # Verifica se o item cabe no recipiente
            if sum(bin) + item <= bin_capacity:
                bin.append(item)
                allocated = True
                break
        # Se o item não couber em nenhum recipiente existente, cria um novo
        if not allocated:
            bins.append([item])
    # Retorna os recipientes com os itens alocados
    return bins

# Define a função para combinar os recipiente dos três subconjuntos
def combine_bins_variable(bins1, bins2, bins3, bin_capacity):
    combined_bins = bins1[:]
    # Combina os recipientes dos subconjuntos
    for bin in bins2 + bins3:
        allocated = False
        for combined_bin in combined_bins:
            # Tenta adicionar os itens do recipiente atual em um recipiente combinado
            if sum(combined_bin) + sum(bin) <= bin_capacity:
                combined_bin.extend(bin)
                allocated = True
                break
        # Se os itens não couberem em nenhum recipiente combinado, cria um novo
        if not allocated:
            combined_bins.append(bin)
    # Retorna os recipientes combinados
    return combined_bins

```

## Complexidade de tempo

Vamos analisar a complexidade de tempo contando as instruções de cada função do algoritmo:

### **first\_fit\_decreasing**

Linhas de Código	Número de instruções
items.sort(reverse=True)	$n \log n$
bins = []	1
for item in items:	n
allocated = False	1
for bin in bins:	n
if sum(bin) + item <= bin_capacity:	n
bin.append(item)	1
allocated = True	1
break	1
if not allocated:	1
bins.append([item])	1
return bins	1
<b>Total de instruções:</b>	$n \log n + n^3 + 8$
<b>Complexidade:</b>	$O(n^3)$

### **combine\_bins\_variable**

Linhas de Código	Número de instruções
combined_bins = bins1[:]	n
for bin in bins2 + bins3:	n
allocated = False	1
if sum(combined_bin) + sum(bin) <= bin_capacity:	n
combined_bin.extend(bin)	n
allocated = True	1
break	1

if not allocated:	1
combined_bins.append(bin)	1
return combined_bins	1
<b>Total de instruções:</b>	$n \log n + n^3 + n + 6$
<b>Complexidade:</b>	$O(n^3)$

## bin\_packing

Linhas de Código	Número de instruções
if len(items) <= 1:	1
return len(items)	1
items.sort(reverse=True)	$n \log n$
large_threshold = bin_capacity * 0.7	1
medium_threshold = bin_capacity * 0.4	1
large_items = [item for item in items if item > large_threshold]	$n$
medium_items = [item for item in items if medium_threshold < item <= large_threshold]	$n$
small_items = [item for item in items if item <= medium_threshold]	$n$
small_items_bins = first_fit_decreasing(small_items, bin_capacity)	$n^3$
medium_items_bins = first_fit_decreasing(medium_items, bin_capacity)	$n^3$
large_items_bins = first_fit_decreasing(large_items, bin_capacity)	$n^3$
combined_bins = combine_bins_variable(small_items_bins, medium_items_bins, large_items_bins, bin_capacity)	$n^3$
return len(combined_bins)	1
<b>Total de instruções:</b>	$3n + 3n^3 + 3 + n \log n$
<b>Complexidade:</b>	$O(n^3)$

## Complexidade de espaço

O algoritmo de bin packing analisado tem uma complexidade de espaço principalmente influenciada pela armazenagem dos itens em recipientes e pela combinação desses recipientes. Durante a execução, cada item e cada recipiente ocupam espaço na memória, e a quantidade de espaço necessário cresce com o número de itens,  $n$ . A função *first\_fit\_decreasing* cria uma lista de recipientes onde cada item é alocado, e a função *combine\_bins\_variable* combina esses recipientes em uma estrutura única. No pior caso, cada item pode estar em um recipiente separado, levando a uma complexidade de espaço de  $O(n)$  para a lista de recipientes. Além disso, as listas intermediárias usadas para dividir os itens em categorias (grande, médio, pequeno) também contribuem para a complexidade de espaço, mas não aumentam a complexidade além de  $O(n)$ . Portanto, a complexidade de espaço total do algoritmo é  $O(n)$ , onde  $n$  é o número total de itens, refletindo a necessidade de armazenar cada item e a estrutura de recipientes utilizada para a organização dos itens.

## Problema de decisão

Extremamente semelhante à implementação do algoritmo de otimização, a única diferença nesta implementação é a forma como produzimos o resultado do algoritmo, mantendo inalterada sua complexidade de tempo e espaço.

## Implementação

```
def bin_packing(items, bin_capacity, k):
    # Caso base: se há apenas um item ou nenhum, retorna se é possível
    # acomodar no número de recipiente permitido
    if len(items) <= 1:
        return len(items) <= k

    # Ordena os itens em ordem decrescente
    items.sort(reverse=True)

    # Define os limites para classificar os itens como grandes, médios
    # ou pequenos
    large_threshold = bin_capacity * 0.7
    medium_threshold = bin_capacity * 0.4

    # Divide os itens em três subconjuntos baseados nos limites
    # definidos
    large_items = [item for item in items if item > large_threshold]
    medium_items = [item for item in items if medium_threshold < item <=
large_threshold]
    small_items = [item for item in items if item <= medium_threshold]
```

```

# Executa o algoritmo de first fit decreasing para cada subconjunto
small_items_bins = first_fit_decreasing(small_items, bin_capacity)
medium_items_bins = first_fit_decreasing(medium_items, bin_capacity)
large_items_bins = first_fit_decreasing(large_items, bin_capacity)

# Combina os recipientes dos três subconjuntos
combined_bins = combine_bins_variable(small_items_bins,
medium_items_bins, large_items_bins, bin_capacity)

# Retorna se o número total de recipientes necessários é menor ou
igual a k
return len(combined_bins) <= k

# Define a função que implementa o algoritmo First Fit Decreasing
def first_fit_decreasing(items, bin_capacity):
    # Ordena os itens em ordem decrescente
    items.sort(reverse=True)
    bins = []
    # Aloca os itens nos recipientes
    for item in items:
        allocated = False
        for bin in bins:
            # Verifica se o item cabe no recipiente
            if sum(bin) + item <= bin_capacity:
                bin.append(item)
                allocated = True
                break
        # Se o item não couber em nenhum recipiente existente, cria um
novo
        if not allocated:
            bins.append([item])
    # Retorna os recipientes com os itens alocados
    return bins

# Define a função para combinar os recipientes dos três subconjuntos
def combine_bins_variable(bins1, bins2, bins3, bin_capacity):
    combined_bins = bins1[:]
    # Combina os recipientes dos subconjuntos
    for bin in bins2 + bins3:
        allocated = False
        for combined_bin in combined_bins:
            # Tenta adicionar os itens do recipiente atual em um
recipiente combinado
            if sum(combined_bin) + sum(bin) <= bin_capacity:
                combined_bin.extend(bin)

```

```

        allocated = True
        break
    # Se os itens não couberem em nenhum recipiente combinado, cria
    # um novo
    if not allocated:
        combined_bins.append(bin)
    # Retorna os recipientes combinados
return combined_bins

```

Note que a única diferença neste algoritmo é o retorno da função **bin\_packing**, enquanto na versão de otimização retornamos **len(combined\_bins)** para retornarmos a quantidade otimizada de bins necessários para o empacotamento, o algoritmo decisão retorna **len(combined\_bins) <= k**, de forma a responder se o número de bins fornecido como entrada atende à demanda de objetos fornecidos.

### Complexidade de tempo

Computacionalmente, o algoritmo de decisão possui o mesmo número de instruções do algoritmo de otimização, de forma que a mesma contagem desse algoritmo seria redundante nessa seção, de forma que podemos concluir que a complexidade de tempo deste algoritmo também é **O(n^2)**.

### Complexidade de espaço

Da mesma forma explicada pela seção de complexidade de tempo, esse algoritmo possui a mesma complexidade de espaço do algoritmo de otimização, então podemos concluir que a complexidade de espaço da versão de decisão é **O(n)**.

## Backtracking

Nesta seção, abordaremos implementações do algoritmo de Bin Packing usando backtracking. Esta técnica de programação consiste em resolver problemas avaliando todas as combinações possíveis de decisões. No Bin Packing, o objetivo é alocar itens de diversos tamanhos em um número mínimo de recipientes de capacidade limitada. O backtracking aloca itens nos recipientes e faz o “backtrack” se uma alocação não for eficaz, buscando continuamente a solução mais eficiente. Embora preciso, o backtracking pode ser computacionalmente exigente, sendo mais aplicável a problemas menores de Bin Packing.

### Problema de otimização

Nesta implementação, o objetivo utilizando a técnica de backtracking é encontrar através de recursão, as combinações possíveis de resultado de forma a buscar sempre a combinação mais otimizada possível. Também note que da mesma forma do algoritmo “First-Fit Decreasing”, é utilizada uma heurística que consiste em ordenar os itens de forma decrescente de modo a tentar alocar itens maiores primeiros e na sequência tentar alocar os menores, de forma que a tendência é que os itens maiores sejam difíceis de serem

combinados com outros, portanto, eliminá-los primeiro se torna uma boa estratégia de aproximação.

## Implementação

```
import time

def bin_packing(items, bin_capacity):
    # Função de backtracking para encontrar o número mínimo de recipientes
    def backtrack(index, bins):
        # Se todos os itens foram alocados, retorna o número atual de recipientes
        if index == len(items):
            return len(bins)

        # Item atual a ser alocado
        item = items[index]
        # Inicializa o número mínimo de recipientes como infinito
        min_bins = float('inf')

        # Tenta colocar o item em cada um dos recipientes existentes
        for i in range(len(bins)):
            if bins[i] + item <= bin_capacity:
                # Se o item couber, adicione ao recipiente
                bins[i] += item
                # Recursivamente tenta alocar os próximos itens e atualiza o mínimo de recipientes necessário
                min_bins = min(min_bins, backtrack(index + 1, bins))
                # Desfaz a adição do item ao recipiente (backtracking)
                bins[i] -= item

            # Tenta criar um novo recipiente para o item
            bins.append(item)
            # Recursivamente tenta alocar os próximos itens com o novo recipiente
            min_bins = min(min_bins, backtrack(index + 1, bins))
            # Remove o recipiente criado (backtracking)
            bins.pop()

        return min_bins

    # Ordena os itens em ordem decrescente para otimização
    items.sort(reverse=True)
    # Inicia o backtracking com o primeiro item e sem recipientes
    return backtrack(0, [])
```

## Complexidade de tempo

Devido à complexidade desse algoritmo devido a sua recursão, na explicação da complexidade desse algoritmo optamos por usar uma forma mais explicativa ao invés da utilização do método tradicional de contagem de instruções para ambas as abordagens (otimização e decisão).

O algoritmo começa ordenando os itens, que tem uma complexidade de tempo de  $O(n \log n)$ , onde  $n$  é o número de itens. Esta é uma etapa inicial única e, embora relevante, não é a parte mais crítica em termos de tempo computacional.

A parte essencial do algoritmo é a função de backtracking, que é uma função recursiva. A cada chamada dessa função, o algoritmo tenta colocar um item em um dos recipientes existentes ou abrir um novo recipiente. O número de chamadas recursivas cresce exponencialmente com o número de itens, porque para cada item, o algoritmo explora várias possibilidades - cada recipiente existente mais a opção de criar um novo recipiente.

Teoricamente, no pior caso, a complexidade de tempo pode se aproximar de  $O(n!)$ , o fatorial de  $n$ . Isso ocorre porque cada item pode ser colocado de  $n$  maneiras diferentes (considerando todos os recipientes possíveis até esse ponto), e essa escolha é independente para cada um dos  $n$  itens.

No entanto, na prática, várias dessas possibilidades são podadas porque não levam a soluções viáveis (por exemplo, quando a adição de um item excede a capacidade do recipiente). Além disso, a ordenação inicial dos itens em ordem decrescente também ajuda a reduzir o número de combinações inviáveis que são exploradas. Apesar disso, a natureza exponencial/fatorial da complexidade de tempo permanece, tornando o algoritmo impraticável para um número grande de itens.

Portanto, a complexidade de tempo do algoritmo de Bin Packing com backtracking é dominada pelo crescimento exponencial do número de possíveis maneiras de alocar os itens nos recipientes, resultando em uma complexidade de tempo que é, no pior caso,  $O(n!)$ .

## Complexidade de espaço

A complexidade de espaço do algoritmo de bin packing apresentado é dominada pela profundidade da árvore de recursão e pelo armazenamento dos estados dos recipientes em cada nível da recursão. No pior caso, a complexidade de espaço é  $O(n^2)$ , onde  $n$  é o número de itens. Isso se deve ao fato de que a árvore de recursão pode atingir uma profundidade de  $n$ , e em cada nível da árvore, o estado de até  $n$  recipientes pode ser mantido. Embora a ordenação inicial dos itens tenha sua própria complexidade de espaço, ela é tipicamente  $O(n)$  e não altera a complexidade geral de espaço do algoritmo, que é majoritariamente influenciada pela estrutura da árvore de recursão e pelo armazenamento dos estados dos recipientes.

## Problema de decisão

Semelhante ao problema de otimização, o problema de decisão segue os mesmos princípios e técnicas abordadas, mas com o objetivo de responder se é possível empacotar todos os itens dados em uma determinada quantidade de recipientes.

### Implementação

```
import time

def bin_packing(items, bin_capacity, k):
    # Função de backtracking para verificar se é possível alocar todos os itens em k recipientes
    def backtrack(index, bins, k):
        # Se todos os itens foram alocados, retorna True
        if index == len(items):
            return True

        # Item atual a ser alocado
        item = items[index]

        # Tenta colocar o item em cada um dos recipientes existentes
        for i in range(len(bins)):
            if bins[i] + item <= bin_capacity:
                # Se o item couber, adiciona ao recipiente
                bins[i] += item
                # Recursivamente tenta alocar os próximos itens
                if backtrack(index + 1, bins, k):
                    return True
                # Desfaz a adição do item ao recipiente (backtracking)
                bins[i] -= item

        # Se ainda houver espaço para criar um novo recipiente
        if len(bins) < k:
            # Tenta criar um novo recipiente para o item
            bins.append(item)
            # Recursivamente tenta alocar os próximos itens com o novo recipiente
            if backtrack(index + 1, bins, k):
                return True
            # Remove o recipiente criado (backtracking)
            bins.pop()

        # Se não foi possível alocar o item em nenhum recipiente,
        return False
    return backtrack(0, [0] * k, k)
```

```
# Ordena os itens em ordem decrescente para otimização
items.sort(reverse=True)
# Inicia o backtracking com o primeiro item e sem recipientes
return backtrack(0, [], k)
```

## Complexidade de tempo

O algoritmo de decisão para o problema de empacotamento é semelhante à versão de otimização em abordagem e complexidade de tempo, mas há diferenças significativas na intenção e no comportamento de execução. Semelhante à versão otimizada, o algoritmo de decisão começa com a ordem dos elementos e possui uma complexidade de tempo de  $O(n \log n)$ , onde  $n$  é o número de elementos. Esta etapa preliminar torna mais fácil alocar primeiro os elementos maiores, reduzindo potencialmente o número de chamadas recursivas necessárias.

O foco do algoritmo de decisão continua sendo os métodos de backtracking. No entanto, diferentemente da versão de otimização, que visa o número mínimo de recipientes necessários para acomodar todos os itens, a versão de decisão visa se é possível encaixar todos os itens em um número fixo de recipientes, especificado por  $k$ . O foco está em testar se a entrada é válida ou não. Essa diferença altera a natureza da árvore de recursão e como a pesquisa é realizada.

Na versão de decisão, a função de backtracking explora as possibilidades de atribuir cada elemento aos recipientes disponíveis até que todos os elementos sejam atribuídos ou o número de recipientes exceda  $k$ . A complexidade do tempo aumenta exponencialmente com o número de elementos, à medida que o algoritmo verifica se cada elemento precisa ser atribuído a cada um dos recipientes existentes. Porém, o crescimento é limitado pelo número fixo de recipientes  $k$ .

Teoricamente, a complexidade de tempo do pior caso é limitada por  $O(k^n)$ , e cada um dos  $n$  elementos pode ser colocado em no máximo  $k$  contêineres. Esta é uma melhoria significativa em relação à complexidade  $O(n!)$  da versão otimizada, especialmente quando  $k$  é muito menor que  $n$ . Porém, semelhante à versão otimizada, algumas possibilidades são rapidamente descartadas devido à incompatibilidade com a capacidade do recipiente, e o espaço de busca pode ser reduzido ordenando primeiro os itens em ordem decrescente. Contudo, é importante notar que embora a complexidade teórica seja menor, o algoritmo de decisão ainda enfrenta desafios práticos devido à sua natureza exponencial, especialmente para grandes valores de  $n$  e  $k$ . Em resumo, a versão otimizada do problema de empacotamento concentra-se em encontrar o número mínimo de recipientes, de modo que a complexidade do tempo possa se aproximar de  $O(n!)$ .

## Complexidade de espaço

A mesma explicação da complexidade de espaço do algoritmo de otimização citada anteriormente pode ser aplicada ao algoritmo de decisão, portanto, podemos concluir que a complexidade de espaço no pior caso deste algoritmo é  $O(n^2)$ .

# Força Bruta

Nesta seção, iremos explorar a implementação do problema de Bin Packing utilizando a técnica de força bruta.

## Problema de otimização

O método de Força Bruta busca encontrar a menor quantidade de recipientes necessária para acomodar todos os itens, testando todas as combinações possíveis de itens nos recipientes.

## Implementação

Essa implementação busca a menor quantidade de recipientes necessários, testando todas as permutações possíveis de acomodação dos itens nos recipientes. Note que este código é altamente ineficiente para grandes conjuntos de dados.

```
from itertools import permutations

# Define a função bin_packing, que recebe uma lista de itens e a
capacidade dos recipientes
def bin_packing(items, bin_capacity):
    # Calcula o número total de itens
    n = len(items)
    # Ordena os itens em ordem decrescente
    items = sorted(items, reverse=True)

    # Inicialmente, assume-se que cada item vai para um recipiente
    # diferente
    best_bin_count = n

    # Gera todas as permutações possíveis para organizar os itens
    for bins in permutations(range(n)):
        # Inicializa a capacidade de cada recipientes com a capacidade
        # máxima
        bin_capacities = [bin_capacity] * n
        # Inicializa o contador de recipientes usados
        bin_count = 0

        # Itera sobre cada índice de item na permutação atual
        for item_index in bins:
            # Obtém o item correspondente ao índice
            item = items[item_index]
            # Verifica se o item cabe em algum dos recipientes já
            # abertos
```

```

        for i in range(bin_count):
            # Se o item couber no recipiente, subtrai a capacidade
            do recipiente
                if bin_capacities[i] >= item:
                    bin_capacities[i] -= item
                    break
                else:
                    # Se o item não couber em nenhum recipiente, abre um
                    novo recipiente
                    bin_count += 1
                    bin_capacities[bin_count - 1] -= item

            # Atualiza o melhor número de recipientes, se a permutação atual
            usar menos recipientes
            best_bin_count = min(best_bin_count, bin_count)

        # Retorna o menor número de recipientes necessário para acomodar
        todos os itens
    return best_bin_count

```

## Complexidade de tempo

Linhas de Código	Número de instruções
<code>n = len(items)</code>	1
<code>items = sorted(items, reverse=True)</code>	$n \log n$
<code>best_bin_count = n</code>	1
<code>for bins in permutations(range(n)):</code>	$n!$
<code>bin_capacities = [bin_capacity] * n</code>	$n$
<code>bin_count = 0</code>	1
<code>for item_index in bins:</code>	$n$
<code>item = items[item_index]</code>	1
<code>for i in range(bin_count):</code>	$n$

if bin_capacities[i] >= item:	1
bin_capacities[i] -= item	1
break	1
else:	
bin_count += 1	1
bin_capacities[bin_count - 1] -= item	1
best_bin_count = min(best_bin_count, bin_count)	1
return best_bin_count	1
<b>Total de instruções:</b>	<b><math>n \log n + n! * (1 + n + n^2)</math></b>
<b>Complexidade:</b>	<b><math>O(n!)</math></b>

## Complexidade de espaço

A complexidade de espaço do algoritmo de Bin Packing com força bruta é principalmente influenciada pela armazenagem das permutações dos itens e pela manutenção de estruturas de dados para os  $n$  recipientes. Para cada permutação, o algoritmo cria um array de capacidades de recipiente e um conjunto de  $n$ , cada um podendo conter até  $n$  itens. Embora o número de permutações seja  $n!$ , a memória para as permutações, não é armazenada simultaneamente, pois são geradas e processadas sequencialmente. Portanto, a complexidade de espaço é dominada pelo armazenamento dos  $n$  e capacidades por permutação, resultando em uma complexidade de  $O(n^2)$  no pior caso, uma vez que cada recipiente pode conter até  $n$  itens e há  $n$  recipientes.

## Problema de decisão

Em sua forma de problema de decisão, o método de força bruta pergunta: "É possível acomodar todos os itens em um número específico de recipientes?". Esta versão do problema ainda exige verificar todas as combinações possíveis até encontrar uma solução viável.

## Implementação

De forma semelhante ao algoritmo de otimização, neste caso também montamos todas as permutações possíveis e verificamos se dado um certo número de bins com uma capacidade especificada, é possível acomodar todos os itens. Vejamos:

```
from itertools import permutations

# Define a função que verifica se é possível acomodar os itens em um
número fixo de bins
def bin_packing(items, bin_capacity, k):
    # Calcula o número total de itens
    n = len(items)
    # Ordena os itens em ordem decrescente
    items = sorted(items, reverse=True)

    # Gera todas as permutações possíveis para organizar os itens
    for bins in permutations(range(n)):
        # Inicializa a capacidade de cada recipiente com a capacidade
        # máxima
        bin_capacities = [bin_capacity] * n
        # Inicializa o contador de recipientes usados
        bin_count = 0

        # Itera sobre cada índice de item na permutação atual
        for item_index in bins:
            # Obtém o item correspondente ao índice
            item = items[item_index]
            # Verifica se o item cabe em algum dos recipientes já
            # abertos
            for i in range(bin_count):
                # Se o item couber no recipiente, subtrai a capacidade
                # do recipiente
                if bin_capacities[i] >= item:
                    bin_capacities[i] -= item
                    break
            else:
                # Se o item não couber em nenhum recipiente, abre um
                # novo recipiente
                bin_count += 1
                bin_capacities[bin_count - 1] -= item
            # Se o número de recipientes exceder k, a resposta é não
            if bin_count > k:
                break

        # Se a permutação atual usar k recipientes ou menos, a resposta
        # é sim
        if bin_count <= k:
            return True

    # Se todas as permutações foram testadas e nenhuma coube em k
    # recipientes ou menos, a resposta é não
```

```
return False
```

## Complexidade de tempo

Linhas de Código	Número de instruções
<code>n = len(items)</code>	1
<code>items = sorted(items, reverse=True)</code>	$n \log n$
<code>for bins in permutations(range(n)):</code>	$n!$
<code>    bin_capacities = [bin_capacity] * n</code>	$n$
<code>    bin_count = 0</code>	1
<code>    for item_index in bins:</code>	$n$
<code>        item = items[item_index]</code>	1
<code>        for i in range(bin_count):</code>	$n$
<code>            if bin_capacities[i] &gt;= item:</code>	1
<code>                bin_capacities[i] -= item</code>	1
<code>                break</code>	1
<code>            else:</code>	
<code>                bin_count += 1</code>	1
<code>                bin_capacities[bin_count - 1] -= item</code>	1
<code>                if bin_count &gt; k:</code>	1
<code>                    break</code>	1
<code>                if bin_count &lt;= k:</code>	1
<code>                    return True</code>	1
<code>    return False</code>	1
<b>Total de instruções:</b>	$n \log n + n! * (1 + n + n^2)$
<b>Complexidade:</b>	$O(n!)$

## Complexidade de espaço

Para cada permutação de itens, o algoritmo mantém um array de capacidades de recipiente e um contador de recipientes usados. Embora o número de permutações seja factorial em relação ao número de itens, o algoritmo não armazena todas as permutações simultaneamente, mas processa cada uma sequencialmente. Assim, o espaço necessário é proporcional ao número de itens, resultando em uma complexidade de espaço da ordem de  $O(n)$ , onde  $n$  é o número de itens. No entanto, a utilização efetiva do espaço pode variar com a distribuição específica dos tamanhos dos itens e das capacidades dos recipientes.

## Programação Dinâmica

Nesta seção, iremos explorar a implementação do problema de Bin Packing utilizando a técnica de programação dinâmica.

### Problema de otimização

O algoritmo de Bin Packing apresentado utiliza princípios da programação dinâmica, uma técnica poderosa para resolver problemas de otimização. A abordagem do algoritmo foca na otimização incremental, armazenando e reutilizando soluções de subproblemas, característica fundamental da programação dinâmica. Ao iterar sobre cada item e buscar o primeiro recipiente com capacidade suficiente para acomodá-lo, o algoritmo efetivamente divide o problema em subproblemas menores, onde a decisão ótima em cada passo é baseada no estado atual das soluções dos subproblemas (a capacidade restante em cada recipiente). Essa abordagem permite que decisões locais ótimas levem a uma solução global eficiente, aproveitando os resultados de alocações anteriores para informar decisões futuras, minimizando assim o número total de recipientes necessários. Embora não garanta a solução ótima em todos os casos como de costume, o uso da programação dinâmica neste contexto permite uma solução eficiente para o problema de Bin Packing.

### Implementação

```
import time

# Define a função que implementa o algoritmo de bin packing
def bin_packing(items, bin_capacity):
    n = len(items) # Calcula o número total de itens
    items.sort(reverse=True) # Ordena os itens em ordem decrescente
    para otimizar a alocação

    # Inicializa o número de bins necessários
    bin_count = 0

    # Inicializa a lista que armazena a capacidade restante para cada
    bin
    bin_rem = [0] * n
```

```

# Itera por cada item na lista
for item in items:
    # Encontra o primeiro bin que pode acomodar o item
    j = 0
    while(j < bin_count):
        if (bin_rem[j] >= item): # Verifica se o bin atual tem
capacidade para o item
            bin_rem[j] = bin_rem[j] - item # Aloca o item no bin,
atualizando a capacidade restante
            break
        j += 1

    # Se nenhum bin existente puder acomodar o item, cria um novo
bin
    if (j == bin_count):
        bin_rem[j] = bin_capacity - item # Aloca o item no novo
bin, inicializando a capacidade restante
        bin_count += 1 # Incrementa o contador de bins

    # Retorna o número total de bins necessários para acomodar todos os
itens
return bin_count

```

Complexidade de tempo

Linhas de Código	Número de instruções
<code>n = len(items)</code>	1
<code>items.sort(reverse=True)</code>	$n \log n$
<code>bin_count = 0</code>	1
<code>bin_rem = [0] * n</code>	$n$
<code>for item in items:</code>	$n$
<code>    j = 0</code>	1
<code>    while(j &lt; bin_count):</code>	$n$
<code>        if (bin_rem[j] &gt;= item):</code>	1
<code>            bin_rem[j] = bin_rem[j] - item</code>	1
<code>        break</code>	1

j += 1	1
if (j == bin_count):	1
bin_rem[j] = bin_capacity - item	1
bin_count += 1	1
return bin_count	1
<b>Total de instruções:</b>	$n \log n + 3 + n + n * (1 + 4n + 3)$
<b>Complexidade:</b>	$O(n^2)$

### Complexidade de espaço

O algoritmo de bin packing aqui apresentado tem uma complexidade de espaço bastante gerenciável, primariamente influenciada pelo armazenamento da lista de itens e pela lista que rastreia a capacidade restante de cada bin. A lista de itens, uma vez ordenada, ocupa um espaço proporcional ao número de itens, ou  $O(n)$ . Paralelamente, a lista bin\_rem, que mantém a capacidade restante de cada bin, também tem um tamanho máximo de  $n$ , resultando em outra  $O(n)$  de uso de espaço. Não há crescimento exponencial ou factorial no uso de espaço, pois todas as operações e armazenamentos são feitos em estruturas de dados que escalam linearmente com o número de itens. Assim, a complexidade de espaço total do algoritmo é linear, marcada como  $O(n)$ , o que significa que o espaço necessário na memória para executar o algoritmo cresce linearmente com o número de itens a serem alocados.

### Problema de decisão

Mostraremos agora a mesma abordagem, mas implementando a adaptação para o problema de decisão.

### Implementação

```
import time

# Define a função que verifica se é possível acomodar os itens em um
# número fixo de bins
def bin_packing(items, bin_capacity, bin_count):
    items.sort(reverse=True) # Ordena os itens em ordem decrescente
    # para otimizar a alocação

    # Inicializa a lista que armazena a capacidade restante para cada
    # bin
```

```

bin_rem = [bin_capacity] * bin_count

# Itera por cada item na lista
for item in items:
    # Encontra o primeiro bin que pode acomodar o item
    j = 0
    while(j < bin_count):
        if (bin_rem[j] >= item): # Verifica se o bin atual tem
capacidade para o item
            bin_rem[j] = bin_rem[j] - item # Aloca o item no bin,
atualizando a capacidade restante
            break
        j += 1

    # Se o item não coube em nenhum bin existente, a alocação é
impossível
    if (j == bin_count):
        return False # Retorna False, pois não foi possível
acomodar o item

    # Se todos os itens foram acomodados, retorna True
return True

```

## Complexidade de tempo

Linhas de Código	Número de instruções
items.sort(reverse=True)	$n \log n$
bin_rem = [bin_capacity] * bin_count	1
for item in items:	$n$
j = 0	1
while(j < bin_count):	$n$
if (bin_rem[j] >= item):	1
bin_rem[j] = bin_rem[j] - item	1
break	1
j += 1	1
if (j == bin_count):	1

return False	1
return True	1
<b>Total de instruções:</b>	$n \log n + 1 + (n * 3 * n * 4)$
<b>Complexidade:</b>	$O(n^2)$

### Complexidade de espaço

A complexidade de espaço desse algoritmo segue a mesma lógica da implementação da versão de otimização, portanto, a complexidade espaço é linear, ou seja,  $O(n)$ .

## Tempo de execução dos algoritmos

Nesta seção, iremos fazer um estudo sobre o tempo de execução de cada algoritmo apresentado anteriormente. Para isso, utilizaremos as seguintes restrições e método:

- A máquina em que os algoritmos serão executados será um **MacBook Pro Intel Core i7 6-Core 2.6 GHz com memória RAM 16GB 2667 MHz DDR4**.
- O tempo de execução de cada algoritmo será medido em **milissegundos**.
- Como abordaremos a versão de Bin Packing linear, a entrada de cada algoritmo será uma lista não ordenada de números aleatórios entre 1 e 10.
- Para cada algoritmo, faremos testes utilizando três tamanhos de entradas diferentes: 10, 100 e 1000, de modo a simular como o algoritmo se comporta à medida que a entrada aumenta.
- O tamanho de cada recipiente do trabalho será fixado em 10.
- Para cada tamanho de entrada diferente, executaremos o algoritmo 10 vezes e o resultado será a média aritmética do tempo de execução de cada teste.

### Problema de otimização

Algoritmo	Tamanho da entrada	Média do tempo de execução	Tempo de execução mais alto
First-Fit Decreasing	10	0.012ms	0.016ms
	100	0.635ms	0.714ms
	1000	58.765ms	61.067ms
First-Fit Decreasing com Divisão e Conquista	10	0.023ms	0.034ms
	100	1.069ms	1.242ms
	1000	104.835ms	127.240ms

Programação dinâmica	10	0.008ms	0.014ms
	100	0.452ms	0.657ms
	1000	35.696ms	38.464ms
Backtracking	10	0.567ms	1.528ms
	100	Não foi possível medir	Não foi possível medir
	1000	Não foi possível medir	Não foi possível medir
Força bruta	10	26571.503ms	29339.474ms
	100	Não foi possível medir	Não foi possível medir
	1000	Não foi possível medir	Não foi possível medir

Note que, nos algoritmos utilizando força bruta e backtracking, não foi possível medir em tempo hábil os tempos de execução para as entradas de tamanho 100 e 1000. Isso deve-se principalmente ao fato de que a complexidade de tempo desses algoritmos é  $O(n!)$ , ou seja, representa a pior ordem de complexidade para algoritmos. Assim, à medida que o tamanho da entrada aumenta, o tempo de execução cresce abruptamente, diferentemente dos outros algoritmos que mantêm uma complexidade de tempo na ordem de  $O(n^2)$ .

## Problema de decisão

Nesta seção, faremos a mesma comparação de desempenho, porém apenas dos algoritmos de decisão. Uma consideração importante para esta seção é que o valor de k (parâmetro relativo ao número de recipientes utilizados para realizar a decisão) será, para as entradas 10, 100, 1000, respectivamente: 6, 55 e 560. Esses valores foram obtidos através de uma média de "bins" suficientes, seguindo o algoritmo de First-Fit Decreasing, um dos mais precisos abordados neste estudo.

Algoritmo	Tamanho da entrada	Média do tempo de execução	Tempo de execução mais alto
First-Fit Decreasing	10	0.014ms	0.049ms
	100	0.185ms	0.272ms

	1000	21.918ms	54.982ms
First-Fit Decreasing com Divisão e Conquista	10	0.018ms	0.023ms
	100	0.793ms	1.139ms
	1000	69.913ms	73.199ms
Programação dinâmica	10	0.006ms	0.015ms
	100	0.188ms	0.248ms
	1000	20.278ms	26.582ms
Backtracking	10	0.015ms	0.034ms
	100	Não foi possível medir	Não foi possível medir
	1000	Não foi possível medir	Não foi possível medir
Força bruta	10	6937.104ms	20143.896ms
	100	Não foi possível medir	Não foi possível medir
	1000	Não foi possível medir	Não foi possível medir

Note que permaneceram os mesmos problemas com os algoritmos que utilizam backtracking e força bruta, como explicado anteriormente, isso se deve a péssima complexidade de tempo que ambos os algoritmos têm em comparação aos outros concorrentes.

## Análise dos tempos de execução

Analizando a comparação entre os algoritmos na seção anterior, podemos ver que todos se comportam como o previsto pelo cálculo de sua complexidade de tempo. Enquanto os algoritmos de *First-Fit Decreasing*, *First-Fit Decreasing com Divisão e Conquista* e *programação dinâmica* obtiveram resultados fáceis de serem obtidos na prática, com tempos de execução relativamente baixos, refletindo um crescimento do tempo de execução proporcional ao que concluímos pela suas complexidades que são polinomiais. Por outro lado, conseguimos enxergar nos algoritmos de força bruta e backtracking que tem complexidade factorial, para as entradas maiores, 100 e 1000 itens, não conseguimos executar em tempo hábil, indicando que dado o aumento da entrada, a quantidade de instruções necessárias para a resolução do problema torna o algoritmo muito ineficiente.

# Principais dificuldades

No geral, os principais desafios encontrados nesse trabalho foram:

- Dificuldade em adaptar estratégias de projeto de algoritmos não convencionais para resolução do problema de Bin Packing, como por exemplo, backtracking e força bruta. Em várias literaturas, é indicado que essas abordagens não são viáveis para o problema de Bin Packing e podemos comprovar através de experimentos que realmente não são praticáveis.
- Cálculo da complexidade: alguns algoritmos apresentados neste trabalho são bem complexos e com lógicas sofisticadas, o que levou a uma certa dificuldade para o cálculo de instruções e da conclusão sobre as complexidades de tempo e espaço.
- Abordagem de duas perspectivas do algoritmo de Bin Packing: uma vez que o algoritmo é possível de ser enxergado como otimização ou decisão, tivemos que considerar sempre essas duas abordagens em todas as seções do trabalho.

# Outros algoritmos

Existem grandes algoritmos consolidados para resolução aproximada do problema de Bin Packing, fica aqui como anexo alguns deles e uma referência sobre:

- [Algoritmo Best-Fit](#)
- [Algoritmo Next-Fit](#)
- [Algoritmo Harmonic Fit](#)
- [Almost Worst Fit](#)

# Conclusão

Através deste trabalho, conseguimos aprofundar-nos no problema de Bin Packing, observando diferentes implementações com estratégias variadas, isoladas ou combinadas, para produzir resultados aproximados. Uma vez que se trata de um problema da classe NP, não existe um algoritmo ótimo para sua resolução. Dessa forma, podemos também perceber como algumas estratégias de programação melhoram ou pioram a eficiência de algoritmos na resolução do Bin Packing e de problemas semelhantes. Além disso, entendemos como o problema é aplicado na vida real, exemplificando com o caso de uso na indústria de transportes.

# Observações

Todas as implementações feitas neste trabalho serão anexadas na entrega juntamente com esta documentação, bem como pode ser conferida neste [repositório](#) na pasta `source_code`.

# Bibliografia

- [https://pt.wikipedia.org/wiki/Problema\\_do\\_empacotamento](https://pt.wikipedia.org/wiki/Problema_do_empacotamento)
- [https://en.wikipedia.org/wiki/First-fit\\_bin\\_packing](https://en.wikipedia.org/wiki/First-fit_bin_packing)
-  Complexidade e Classes de Problemas em Otimização: P, NP, NP-completo, N...
- [https://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/NPcompleto.html](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/NPcompleto.html)
- <https://www.baeldung.com/cs/backtracking-algorithms>
- <https://lamfo.unb.br/wp-content/uploads/2021/03/Programa%C3%A7%C3%A3o-Din%C3%A2mica.pdf>
- <https://www.prp.unicamp.br/pibic/congressos/xxicongresso/paineis/135464.pdf>