Entendendo algoritmo.

# Conceito de algoritmo

**O que é algoritmo?** É uma sequência finita e não ambígua de instruções bem definidas, que especifica uma sequência de operações a serem realizadas para resolver um problema ou executar uma tarefa.

**O que é uma lista binária?** É uma estrutura de dados onde cada nó pode ter até dois filhos. Cada nó contém uma chave e os nós da esquerda tem chaves menores e os nó da direita de chaves maiores. É frequentemente usada em algoritmos de pesquisa e ordenação, permitindo operação eficiente de busca, inserção e exclusão.

### Pesquisa Simples X Pesquisa Binária

**Pesquisa Simples** → Conhecida como pesquisa sequencial, envolve percorrer cada elemento da lista até encontrar o item desejado. Possui eficiência para pequenas listas desordenadas, mas sua eficiência diminui com listas maiores.

**Pesquisa Binária →** Aplicável apenas para listas ordenadas e opera dividindo repetidamente a lista ao meio até encontrar o elemento desejado. É muito eficiente em listas ordenadas, pois reduz o número de comparações necessárias tornando-a ideal para listas maiores.

### Um pouco de logaritmo

**Entenda a definição**: O logaritmo de um número \( x \) na base \( b \), denotado como \( \log\_b(x) \), é o expoente ao qual \( b \) deve ser elevado para produzir \( x \). Em outras palavras, \( \log b(x) = y \) significa que \( b^y = x \).

**Identifique a base e o número:** Você precisa saber o número \( x \) é a base \( b \) para calcular o logaritmo. Por exemplo, se você quer calcular \( \log\_{10}(100) \), a base é 10 e o número é 100.

**Aplique a definição:** Se não for possível simplificar usando as propriedades dos logaritmos, você pode aplicar a definição diretamente. Isso geralmente envolve experimentar diferentes expoentes para encontrar o valor correto.

Por exemplo, para calcular \( \log\_{10}(100) \), você pode saber intuitivamente que \( 10^2 = 100 \), então \( \log\_{10}(100) = 2 \).

Ou seja, na pesquisa binária precisa verificar log n elementos para o pior dos casos log(8) é igual a 3 por que 2³ é igual a 8 , ou seja, temos 3 tentativos para acertarmos uma condição de uma lista ordenada.

### Exercício 1.1 e 1.2

***1.1 Suponha que você tenha uma lista com 128 nomes e esteja fazendo uma pesquisa binária. Qual seria o número máximo de etapas que você levaria para encontrar o nome* desejado?**

1. O número máximo de de etapas seria 7, seguindo a seguinte lógica: usando log(128) = 2 ^ 7 = 128, ou seja, vai possuir 7 etapas.

***1.2 Suponha que você duplique o tamanho da lista. Qual seria o número máximo de etapas agora?***

Caso duplique basta multiplicar o número da lista inicial por 2 e fazer log(256) que vai precisar de 8 etapas.

Tempo de execução

### Tempo Linear e tempo logaritmo

* **Tempo Linear:**
  + Em um algoritmo com tempo de execução linear, o tempo necessário para processar os dados aumenta linearmente com o tamanho dos dados de entrada. Por exemplo, se um algoritmo leva 1 segundo para processar uma lista de 100 elementos, ele provavelmente levará cerca de 10 segundos para processar uma lista com 1000 elementos, assumindo que a relação é linear.
* **Tempo Logarítmico:**
  + Em contraste, um algoritmo com tempo de execução logarítmico aumenta de forma muito mais eficiente com o tamanho dos dados. Isso significa que, mesmo quando o tamanho dos dados de entrada aumenta consideravelmente, o tempo de execução do algoritmo cresce de forma mais moderada. Em geral, algoritmos com tempo logarítmico são considerados muito eficientes.

### O que é notação big 0 ?

A notação big O descreve como funciona a taxa de crescimento do tempo de execução de um algoritmo em relação ao tamanho da entrada , fornecendo a ideia do seu desempenho em escala. É uma forma de medir a eficiência e a complexidade de algoritmos.

**O(1) – Tempo linear →** Isso significa que a execução do algoritmo é constante,independente do tamanho da entrada. Um exemplo: Pesquisa simples

**O(log)n – Tempo logaritmo →** Isso significa que tempo de execução do algoritmo aumenta de forma logarítmica com o aumento do tamanho da entrada. Exemplo: Pesquisa binária onde a pesquisa é dívida pela metade de cada iteração.

**O(log\*n) – Tempo quase linear →** Isso significa que o fator de crescimento é um pouco maior que o linear. É comum em algoritmos de ordenação eficiente como Quicksort. Aqui o tempo de execução aumenta proporcionalmente ao tamanho da entrada multiplicado pelo logaritmo do tamanho da entrada.

**O(N^2)** –**Tempo quadrático** → Isso significa que o tempo de execução é quadrática, onde o tempo de execução do algoritmo aumenta quadraticamente como um tamanho de entrada.Um exemplo é a ordernação por seleção, onde é necessário comparar cada elemento com todos os outros.

**O(n!) – Tempo Fatoria**l → Isso significa que o tempo de execução cresce muito rapidamente com o tamanho da entrada.O exemplo: é o problema do caixeiro viajante, onde todas as possíveis todas as possíveis rotas devem ser exploradas.

### Principais pontos:

* A rapidez de um algoritmo não é medido por segundos,mas pelo crescimento do números operações.
* Em vez disso, discutimos como o tempo de execução do algoritmo aumenta conforme o número de elementos aumenta.
* O tempo de execução do algoritmo é expresso pela notação Big 0.
* Log(n) é mais rápido que o O(n) e o O(n) fixa mais rápido quando a lista aumenta.

### Árvore binária

Uma **árvore binária de busca (BST - Binary Search Tree)** é uma estrutura de dados onde cada nó tem no máximo dois filhos, com uma propriedade especial: Para cada nó , todos os nós na subárvore esquerda tem valores menores do que o valor do nó, da subárvore da direita que tem valores maiores.Isso significa que ao, inserir um novo valor na árvore ,podemos percorrer rapidamente a árvore comparando valores e decidindo qual direção seguir em base a comparação.

**Árvore de inserção:** Ao inserir um novo valor, começamos pela raiz e seguimos comparando o valor novo valor com valor atual. Se for menor segmentos pela subárvore esquerda, se for maior segmentos pela subárvore direita.Seguimentos esse processo até encontrar um lugar vazio para inserir.

Árvore de busca.

**Árvore de busca:** Ao procurar por um valor na árvore, começamos pela raiz e comparamos o valor desejado com valor do nó atual. se forem iguais encontramos o valor desejado, se for menos procuramos na subárvore esquerda e se for maior procuramos a subárvore direita.Repetimos esse processo até encontrar nó desejado ou encontrar o valor nulo, indicando que o valor não está na árvore

**Árvore Remoção:** Encontre o nó a ser removido e substitua-o por seu sucessor (o nó mais à direita na subárvore esquerda ou o nó mais à esquerda na subárvore direita), mantendo a propriedade da árvore binária de busca.

**Árvore Travessia**: Em ordem - visite primeiro a subárvore esquerda, depois o nó atual e, por fim, a subárvore direita; Pré-ordem - visite primeiro o nó atual, depois a subárvore esquerda e, por fim, a subárvore direita; Pós-ordem - visite primeiro a subárvore esquerda, depois a subárvore direita e, por fim, o nó atual.

Capítulo 2 → Ordenação por seleção

### **Arrays:**

É uma estrutura de dados que armazena uma coleção fixa de elementos do mesmo tipo de dados,acessíveis através do índice. Cada elemento do array é armazenado em uma posição específica, e o acesso a esses elementos é feito através do seu índice, que geralmente começa por zero. **Possui uma leitura rápida, mas inserções e remoções lentas.**

* **Características**:
* **Tamanho fixo:** O tamanho de um array é fixo, o valor é atribuído no momento de sua criação e não pode ser alterado posteriormente. Isso significa que o número de elementos de um array pode conter é fixo e conhecido antecipadamente.
* **Acesso direto aos elementos:** Todos os elementos de um array são acessados diretamente usando o seu índice. Isso permite acessar qualquer elemento em tempo constante,desde que o índice seja conhecido.
* **Elementos do mesmo tipo:** Todos os elementos de um array podem conter o mesmo tipo de dados. Por exemplo: Um array de inteiros(int) só pode conter números inteiros e string só pode conter string.
* **Armazenamento contíguo de memória:** Os elementos de um array são armazenados em posições de memória contíguas, o que significa que o endereço de memória do elemento i+1 é adjacente ao endereço do elemento i,ou seja, ele sempre incrementa mais um.
* **Desvantagens:**
* **Tamanho fixo:** Você pode não precisar dos espaços extras que reservou; então a memória será desperdiçada. Você não está utilizando a memória, mas ninguém mais pode usá-la também.
* **Ineficiência em inserções e remoções:** Você pode precisar adicionar mais de dez itens à sua lista de tarefas, então você terá de mover seus itens de qualquer maneira.
* I**nflexibilidade em relação ao tipo de dados:** Todos os elementos de um array devem ser do mesmo tipo de dados. Isso pode ser uma limitação em situações onde diferentes tipos de dados precisam ser armazenados na mesma estrutura.
* **Vantagens**:
* **Acesso direto aos elementos:** Os elementos em um array podem ser acessados diretamente usando seus índices, o que permite um acesso rápido aos dados.
* **Armazenamento contíguo de memória:** Os elementos de um array são armazenados em posições de memória contíguas, o que facilita a manipulação eficiente de grandes conjuntos de dados.
* **Eficiência em termos de espaço**: Os arrays geralmente consomem menos memória do que outras estruturas de dados dinâmicas, pois não têm sobrecarga associada a ponteiros extras ou estruturas adicionais.
* **Simplicidade**: Arrays são simples e fáceis de usar. Eles são uma estrutura de dados fundamental, com suporte nativo em muitas linguagens de programação.

### Listas Encadeadas:

Uma lista encadeada é uma estrutura de dados na qual os elementos são armazenados individualmente. Cada nó contém um valor de referência ( ou ponteiro) para o próximo nó na sequência. A estrutura é chamada de “ Encadeada” por que os nós estão conectados sequencialmente através das referências.**Possui inserções e remoções rápidas, e leituras lentas.**

Existem dois tipos de listas:

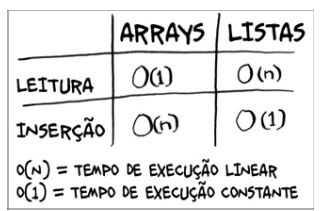
**A lista encadeada simples:** Cada nó contém um único ponteiro para próximo nò na sequência. O último nó geralmente aponta para “ null ”, indicando o final da lista.

**A lista encadeada dupla:** Cada nó contém dois ponteiros: Um para o próximo nó na sequência e outro para o anterior. Isso permite a travessia em ambas direção.

* **Características:**
* **Alocação da memória:** Ao contrário dos arrays, as listas encadeadas não exigem um tamanho fixo e podem crescer ou diminuir conforme necessário,pois a alocação de memória para cada nó é feita dinamicamente.
* **Inserção e remoção** → Inserir ou remover elementos em uma lista encadeada pode ser eficiente, especialmente se a inserção ou remoção for no início da lista. Não é necessário deslocar todos os elementos, apenas ajustar os ponteiros.
* **Acesso sequencial** → Para acessar um elemento específico em uma lista encadeada, geralmente é necessário percorrer a lista do início até o elemento desejado. Isso pode ser menos eficiente do que o acesso direto aos elementos do array.
* **Uso de mais memória:** Devido à necessidade de armazenar ponteiros adicionais para cada elemento, as listas encadeadas podem consumir mais memória do que arrays para armazenar a mesma quantidade de dados.

### Terminologia

Este capítulo está explicando como funciona a numeração de índices de um array ou lista. Inicialmente, eles começam com 0 para indicar sua posição, utilizando o termo "índice". Ou seja, para acessar um elemento em uma lista, devemos dizer que ele está na posição de índice 2.



*Perguntas:*

* ***Por que é necessário tempo de execução O(n) para inserir um elemento em um array?***

Por conta que é necessário mover todos os elementos subsequentes uma posição para frente, a fim de mover espaço para um novo elemento. Isso é necessário porque os arrays têm tamanho fixo e elementos contíguos na memória. Por esse motivo, quando você deseja inserir um elemento em uma posição arbitrária em uma array de tamanho (n), pode ser necessário deslocar (n) elementos. Isso resulta um tamanho proporcional da array resultando ao O(n)

* ***Suponha que você queira inserir um elemento no começo de um array. Como faria isso?***

Para inserir um elemento no início do array , você precisa deslocar todos os elementos atuais para direita , e em seguida inserir um novo elemento na primeira opção. Isso garante um novo elemento no array.

* **Quanto tempo levaria?**

O(n)

* ***2.1 Suponha que você esteja criando um aplicativo para acompanhar as suas finanças. Todos os dias você anotará tudo o que gastou e onde gastou. No final do mês, você deverá revisar os seus gastos e resumir o quanto gastou. Logo, você terá um monte de inserções e poucas leituras. Você deverá usar um array ou uma lista para implementar este aplicativo***

O indicado seria a implementação de uma lista que possuísse inserção e remoção sem precisar deslocar todos os elementos da posição.

### Respostas das perguntas do capítulo 2

**2.2** Eu utilizaria uma lista encadeada, pois para acessar cada pedido o garçom precisa percorrer um por um. Sua inserção é mais rápida.

**2.3** Uma Array, pois com array a gente possui o acesso de memória aleatória, ou seja, seu acesso é mais rápido.

**2.4** Uma das principais desvantagens de utilizar uma array para inserção é por conta que é necessário mover todos os elementos subsequentes para uma posição para frente, a fim de mover espaço para novo elemento. Por conta que as Array são lentas

**2.5** Lenta

Capítulo 3 → Recursão.

* **O que é recursão?** É uma função que resolve um problema dividindo- o em casos menores e chamando-si mesma repetidamente até alcançar um caso base, onde a solução é conhecida e direta. Esse processo de dividir problemas em partes menores é chamado de decomposição recursiva. Cada chamada recursiva resolve uma parte menor do problema, e as soluções dessas partes são combinadas para formar solução do problema original.
* **Caso-base X Caso-Recursivo**

**Caso-base** → É a condição que determina quando a recursão deve parar.Quando a função alcança o caso-base,ela não chama a si mesma novamente e a recursão termina.Caso-base é essencial para evitar que a função entre em um loop infinito.

**Caso-Recursivo →** É uma parte da função que continua chamando a si mesmo até que o caso-base seja alcançado. Essa parte contém a lógica recursiva, ou seja,a chamada recursiva para a própria função com parâmetros diferentes.

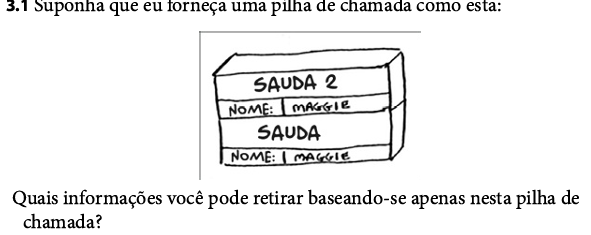
### Pilha:

Uma pilha é uma estrutura de dados linear que segue o princípio “ o último a entrar é o primeiro a sair “ (LIFO - last in, First out). Ela funciona de forma semelhante a uma pilha de pratos onde você pode empilhar os pratos**(no topo da pilha)** e remover **(Desempilhar)** o prato mais recente adicionado.

**Características da pilha:**

* Inserção e Remoção: Os elementos são adicionados (empilhados) ou removidos (desempilhados) apenas no topo da pilha.
* Operações básicas: As operações básicas de uma pilha são:
  + push(item): Adiciona um item ao topo da pilha.
  + pop(): Remove e retorna o item do topo da pilha.
  + peek(): Retorna o item no topo da pilha sem removê-lo.
  + is\_empty(): Verifica se a pilha está vazia.
* **Estrutura interna:** Internamente, uma pilha pode ser implementada usando uma lista ou uma lista encadeada. Na implementação com lista, o topo da pilha é o último elemento da lista

### Exercício 3.1:



**Resposta:** *Quando a função sauda() é chamada, a variável nome é armazenada na memória, guardando a informação. Durante a execução de sauda(), se a função sauda2(nome) for chamada, uma nova caixa de memória é criada e aloca a informação nome, enquanto o estado da função sauda() é salvo na pilha de chamada.*

*Quando a função sauda2() termina sua execução, ela é removida da pilha, e o controle retorna para a função sauda(). A partir desse ponto, sauda() continua sua execução de onde parou. Se sauda() não tiver mais instruções a serem executadas, ela também é removida da pilha.*

### Exercício 3.24

**3.2 Suponha que você acidentalmente escreva uma função recursiva que está executando infinitamente. Como você viu, seu computador aloca memória na pilha para cada chamada de função. O que acontece com a pilha quando a função recursiva cai executando infinitamente?**

*Quando a pilha recursiva é adicionada novos dados no topo da pilha, e quando não tem uma condição definida para parada, essas chamadas de adição ao ponto continha indefinidamente. Por conta disso, a pilha continua a crescer a medidas que novos quadros são adicionados e eventualmente a pilha atingirá o seu limite máximo de memória assim ocorre um stack Overflor.*

Capítulo 4 → Quicksort

### **Dividir em conquistar** → É uma técnica de resolução de problemas que envolve dividir um problema complexos em resoluções menores e mais simples. Essa técnica é frequentemente usadas em algoritmos recursivos e pode ser aplicados a uma ampla variedades de problemas

**Caso-base** → Identificar o caso mais simples possível , que pode ser resolvido diretamente sem divisão.

**Exemplo:**

If x == 20

console.log(x)

**Divisão do problema** → Dividir o problema maior em subproblemas menores que se aproxima do caso base.

**Exemplo.**

else x = x\*3

console.log(x)

### Quicksort

* **O que é?**

É um algoritmo de ordenação. Este algoritmo é muito mais rápido que a ordenação por seleção e é muito utilizado na prática.

* **Como funciona?**

O quicksort utiliza a estratégia DO(dividir para conquistar) ele divide um problema maior em casos menores.

### Quicksort com Array

* Array que estão vazias e que só possui um elemento não precisam ser ordenadas, ou seja, elas são consideradas caso base que significa ser o caso mais possível que não precisa da necessidade de dividir.
* Agora o processo para ordenar uma array com mais de 2 elementos funciona da seguinte maneira.

1. Você escolher um pivô [ Pivô é o elemento escolhido que vai ser utilizado para particionar a array em duas partes; menor e maior]
2. Agora você criar duas subarrays:

* Com elementos menores

**[Pivô]**

* Com elementos maiores

1. Aplique a recursão nas duas subarray, combinando seus resultados ordenados os valores com pivô no meio.

*Exemplo:*

*Array Original:[33, 15, 10]*

1. ***Escolha do Pivô:*** 33
2. **Particionamento:**

* Menores que 33: `[15, 10]`
* Maiores que 33: `[]` (vazio)

3. **Ordenar Subarrays:**

* Ordenando `[15, 10]`:
* Pivô: `15`
* Menores que 15: [10]
* Maiores que 15: []
* Combine: `quicksort([10]) + [15] + quicksort([]) => [10, 15]`
* Ordenando `[]`: Já está ordenado.

4**. Combinar Tudo:**

* Combine `[10, 15]` (subarray ordenado) com `33` (pivô) e `[]` (subarray vazio):
* Resultado Final: `[10, 15, 33]`

### Notação big 0 (Revisada)

* **Exemplo 01: Ordenação por seleção**

A ordenação por seleção tem o tempo de execução O(n²) . Isso quer dizer que tivemos um array com (n) elementos, o tempo de execução será proporcional ao quadrado do número de elementos.

*Exemplo:*

* Para \( n = 10 \), o tempo de execução é proporcional a \( 10² = 100 \) operações.
* Se o computador realizasse 10 operações por segundo, levaria 10 segundos para ordenar
* 10 elementos.
* **Exemplo 02: Merge sort**

O merge sort, tem execução O(n log n), o que é muito mais eficiente.

Exemplo:

* Para \( n = 10 \), o tempo de execução é proporcional a \( 10 \log(10) \) operações.
* Se log(10) é aproximadamente 3,3, então \( 10 \log(10) \approx 33 \) operações.
* Levaria aproximadamente 3,3 segundos para ordenar 10 elementos.
* **Exemplo 03: Quicksort**

O quicksort pode ter dois casos :

* Pior caso = O(n²)
* Médio caso = O( n log n)

### Pior caso vs Caso médio

* **O que é pior caso?**

O pior o caso para o quicksort ocorre quando o pivô escolhido é o menor ou maior elemento em cada divisão, resultando subarrays muito desbalanceados.

* O tempo de execução é O(n²)
* Exemplos: Ordenar listas já ordenadas ou uma lista reversa.
* **O que é o caso médio?**

No caso médio, o quicksort divide a array em duas partes aproximadamente iguais em cada iteração.

* O tempo de execução: O(n log n)
* Esse caso é quando o pivô é bem escolhido, levando as divisões mais balanceadas.

### Marge sort vs Quicksort

* Marge sort
* É consistemente O(n log n) , mas usa mais memória.
* Sua vantagem é que garante todos os casos
* Utiliza muita memória adicional.
* Quicksort
* Geralmente mais rápidos na prática devido ao menor overhead constante.
* Pode ser O(n²) no pior caso, embora isso possa ser mitigado com boas estratégias de escolha de pivô.
* Geralmente é mais rápido na prática (O(n log n) no caso médio), mas pode ser mais lento no pior caso (O(n²)).

### Aplicando os Princípios de Análise de Complexidade

Quando estudamos Quicksort, aprendemos a analisar como o número de operações cresce com o tamanho da entrada (n). Aplicamos esse conhecimento para analisar outros algoritmos e operações, usando os seguintes princípios:

1. **Iteração Simples** →Percorrer um array ou lista uma vez resulta em \(O(n)\).

2. **Operação Constante** →: Uma operação única, independente do tamanho da entrada, resulta em \(O(1)\).

3. **Aninhamento de Loops** → Loops aninhados resultam na multiplicação de complexidades internas, por exemplo, um loop dentro de outro loop resulta em \(O(n^2)\).

*EXERCÍCIOS*

Quanto tempo levaria, em notação Big O, para completar cada uma destas operações? 4.5 Imprimir o valor de cada elemento em um array.

* **4.5 Imprimir o valor de cada elemento em um array.**
* O(n)
* **4.6 Duplicar o valor de cada elemento em um array.**

O(n)

* **4.7 Duplicar o valor apenas do primeiro elemento do array.**

O(1)

* **4.8 Criar uma tabela de multiplicação com todos os elementos do array. Assim, caso o seu array seja [2, 3, 7, 8, 10], você primeiro multiplicará cada elemento por 2. Depois, multiplicar cada elemento por 3 e então por 7, e assim por diante**

O(n2).

Capítulo 5 → Tabela Hash

### Conceitos:

* **O que é tabela Hash?**

É uma estrutura de dados eficiente usada para armazenar e recuperar dados de maneira rápida. A principal característica é a capacidade de acessar os dados de maneira constante(1) , maioria dos casos, graças ao uso de uma função hash.

* **Conceitos básicos**
* **Chave e valor:** Em uma tabela hash, os dados são armazenados como pares de chave e essa chave é usada para identificar de forma única cada entrada da tabela.
* **Função hash:**A função hash é um componente fundamental da tabela hash. Ela recebe uma chave como entrada e retorna um índice no array (ou tabela) onde o valor associado àquela chave será armazenado. O principal objetivo da função hash é distribuir uniformemente as chaves para minimizar colisões, ou seja, evitar que múltiplas chaves diferentes sejam mapeadas para o mesmo índice. Quando usamos uma tabela hash para armazenar informações, a chave é mapeada para um índice específico usando a função hash. Assim, ao invés de realizar uma pesquisa simples (que percorre toda a lista até encontrar o valor), a tabela hash usa a função hash para acessar diretamente o índice onde a informação foi armazenada, tornando a busca muito mais rápida.
* **Função Hash** recebe uma chave e retorna um índice no array onde o valor será armazenado.
* **Distribuição Uniforme:** A função hash deve distribuir as chaves de maneira uniforme para evitar colisões
* **Eficiência:** A tabela hash permite acessar dados de maneira rápida, geralmente em tempo constante \(O(1)\), ao contrário da pesquisa simples que tem tempo linear \(O(n)\).

### Tabela hash com cache

* **Como Funciona a Solicitação ao Servidor?**

Quando um usuário envia uma solicitação para um servidor, o processo funciona da seguinte forma:

**Envio da Solicitação:** O usuário faz uma requisição ao servidor.

**Processamento da Informação:** O servidor processa essa solicitação, o que pode envolver várias etapas, como coletar dados, processar informações, analisar os resultados e preparar a resposta**.**

**Resposta ao Usuário:** Após o processamento, o servidor envia a resposta de volta ao usuário.

* Para tornar as respostas dos servidores mais rápidas, utilizamos uma técnica chamada **cache.**
* **O Que é Cache?**

Cache é uma técnica que armazena temporariamente dados que são frequentemente acessados. Isso permite acelerar significativamente o tempo de resposta das solicitações subsequentes.

* **Tabelas Hash e Cache**

As tabelas hash são extremamente eficazes para implementar cache por causa de sua capacidade de acessar dados rapidamente.

* **- Inserção e Busca Rápidas:** As operações de inserção e busca em uma tabela hash têm tempo constante (O(1)), o que significa que são quase instantâneas.

### **Benefícios do Cache**

* **Redução do Tempo de Resposta**: Dados no cache são retornados quase instantaneamente.
* **Menor Carga no Servidor**: Com menos processamento necessário para solicitações repetidas, o servidor pode lidar com mais usuários ao mesmo tempo.
* **Melhoria na Experiência do Usuário**: Respostas rápidas resultam em uma navegação mais fluida e agradável.

.

### Colisões

* Em uma tabela hash ocorre quando duas ou mais chaves diferentes produzem o mesmo índice em uma tabela. Isso ocorre porque cada chave deve idealmente

armazenar uma única entrada para garantir eficiência na recuperação de dados.

* Ocorrem quando uma nova chave sobrescreve a antiga,levando a perda de dados

A solução para colisões é o uso das listas encadeadas, criar uma lista encadeada no espaço onde ocorre a colisão , armazenar todas as chaves que mapeiam para o mesmo espaço nessa lista. O acesso aos dados pode ser mais lento, mas eficiente se a lista for curta.

### Exercício

**5.1**

Essa função é consistente.

**5.2**

Essa função é inconsistente

**5.3**

Essa função é inconsistente

**5.4**

Função Consistente.

**5.5** Lista telefônica (nomes: Esther, Ben, Bob, Dan)

Função Hash Ideal: (d)

Justificativa: A função (d) proporcionar uma boa distribuição, reduzindo colisões mesmo com nomes que podem ter comprimentos ou primeiros caracteres semelhantes.

**5.6** Tamanho de baterias e potência (A, AA, AAA, AAAA)

Função Hash Ideal: (b)

Justificativa: A função (b) pode ser eficaz aqui, pois os comprimentos das strings são diferentes. No entanto, (d) também é uma boa escolha se uma distribuição mais uniforme for necessária.

**5.7** Títulos de livros e autores (Maus, Fun Home, Watchmen)

Função Hash Ideal: (d)

Justificativa: A função (d) é novamente ideal, pois os títulos podem variar em comprimento e composição de caracteres, necessitando de uma distribuição que minimize colisões.

Capítulo 6 → Grafos

### O que são Grafos e pesquisa de largura(breadth-rst search, BFS)?

* O que são os Grafos?

É uma estrutura matemática usada para modelar relações e objetos, ou seja, são capazes de modelar eventos diferentes que estão conectados entre si. Eles são formados por:

* Vértices: Representados por V,são pontos e entidades do grafo.
* Arestas: Representadas por E, são conexão ou relação entre vértices.
* O que é pesquisa de largura?

É um algoritmo de travessia de grafos que explora todos os vértices de um grafo ou dígrafos por nível.Começa pela raiz e explora todos os seus vizinhos antes de ir para o próximo nível.

### Filas

O que são as filas:

É uma estrutura de dados que opera em FIFO(- First In, First Out) isto significa que o primeiro elemento adicionado na fila é o primeiro a ser removido, ou seja, pode ser visualizado como uma coleção de elementos disposto em uma linha, onde as operações principais são:

* **Enqueue**: Adicionar um elemento ao final da fila.
* **Dequeue**: Remover um elemento do início da fila.

### Exercício

* 6.1 O menor caminho possui 2 aresta
* 6.2 O menor caminho possui 2 aresta

### Grafos Direcionados e Grafos não direcionados.

* **Grafos direcionados:** É um grafo em que as arestas têm uma direção associada a elas. Isso significa que se existe uma aresta de um vértice **A** para um vértice **B,** essa relação não necessariamente indica que existe uma relação de **B** para **A.** Mostrado como setas.
* **Exemplo:** Imagine uma rede social onde você segue outras pessoas, mas elas não necessariamente te seguem de volta. Aqui está como isso pode ser representado:
* Vértices: Usuários da rede social.
* Arestas: Relações de "segue".

**Grafo não direcionado:** É um grafo onde as arestas não tem direção. Isso significa que existe uma aresta entre A e B,então A vizinho de B e B é vizinho de A. mostrado com linhas simples (sem setas).

Exemplo:Imagine um mapa de estradas entre cidades. Se existe uma estrada entre duas cidades, você pode viajar de uma cidade para outra e vice-versa.

* Vértices: Cidades.
* Arestas: Estradas entre as cidades.

### Tempo de execução:

Também se mantém uma lista das pessoas verificadas, onde adicionar uma pessoa leva tempo constante: O(1). Fazer isso para todas as pessoas leva tempo de execução(número de pessoas). Portanto, a pesquisa em largura tem um tempo de execução O(número de pessoas + número de arestas), frequentemente descrito como O(V + A) (V para número de vértices, A para número de arestas).

### Exercícios

* **6.3**

B

* **6.4**

Acordar

Praticar Exercício

Tomar banho

Escovar os dentes

Trocar de roupa

embrulhar o lanche

tomar Café da manhã

* **6.5**

A e C

Capitulo 07 → Algoritmo de Dijkstra

### Conceitos iniciais

* **Grafos ponderados** → é um grafo no qual cada aresta tem um peso associado.Esses pesos podem representar várias coisas, como distância,custos,tempos ou qualquer outra métrica que faça sentido no contexto do problema que estamos tentando resolver.
* **Algoritmo de Dijkstra** → Utilizado para encontrar o caminho mais curto de vértice de origem para todos os outros vértices em um grafo ponderado, onde os pesos das arestas não são negativas. Esse algoritmo é útil em muitas aplicações,como determinar a rota mais curta em redes de transporte ou redes de comunicação.
* Os passos para encontrar o algoritmo de Dijkstra :

1. Encontra o vértice que você chegar com menor tempo, conhecido como vértice “barato”.
2. Verificar se há um caminho mais barato para os vizinhos desse vértice caso exista, atualize os custos deles.
3. Repita até que você tenha feito isso para cada vértice do grafo
4. Calcule o caminho final

### Termologia

* Os grafos possuem valores que são chamados de pesos.
* Os grafos com pesos são chamados de grafos ponderados e são calculados pelo o **algoritmo de Dijkstra**. Enquanto,os algoritmos sem pesos são pesquisas pela **pesquisa em largura.**
* Grafos não direcionados **têm arestas bidirecionais,** permitindo a viagem entre vértices em ambas as direções. Grafos direcionados possuem arestas com direção, permitindo o fluxo apenas de um vértice de origem para um de destino. Aplicações incluem redes sociais para grafos não direcionados e fluxos de dados para grafos direcionados.

Capítulo 08 →Algoritmo Gulosos

### Resumo

**Problema do Cronograma da Sala de Aula**

* **Objetivo:**Maximizar o número de aulas em uma sala de aula sem sobreposição de horários.
* Algoritmo Guloso:

1. Escolha a aula que termina mais cedo.
2. Escolha a próxima aula que comece após a aula escolhida e que termine mais cedo.
3. Repita até não haver mais aulas possíveis.

* *Exemplo*

- Escolha a aula de Artes (termina às 10h).

- Escolha a aula de Matemática (começa após 10h e termina mais cedo que as demais).

- Escolha a aula de Música (sem conflito com Matemática).

**Problema da Mochila:**

* **Objetivo**: Maximizar o valor dos itens colocados em uma mochila com capacidade limitada.
* **Algoritmo Guloso:**

1. Pegue o item mais caro que cabe na mochila.
2. Repita com o próximo item mais caro.

* **Falha do Algoritmo**: Pode não resultar na melhor solução global. Exemplo: escolher o item mais caro pode deixar espaço insuficiente para itens menores que, juntos, teriam maior valor.

**Problema da Cobertura de Conjuntos:**

* Objetivo: Minimizar o número de estações de rádio necessárias para cobrir todos os estados dos EUA.
* Algoritmo Exato: Listar todos os subconjuntos possíveis e escolher o menor que cubra todos os estados. Impraticável devido ao tempo de execução exponencial.

**Algoritmo de Aproximação Guloso**

* Escolha a estação que cobre o maior número de estados ainda não cobertos.
* Exemplo de Código para Cobertura de Conjuntos

1. Use um conjunto para os estados e uma tabela hash para as estações.
2. Itere escolhendo a estação que cobre o maior número de estados não cobertos até que todos os estados estejam cobertos.

**problemas NP-completos:**

* **Exemplo Clássico:** Problema do Caixeiro-viajante - encontrar a rota mais curta que visita todas as cidades.
* **Característica**: Crescimento exponencial de soluções possíveis; difícil de resolver exatamente para grandes conjuntos de dados.
* **Algoritmo de Aproximação:** Escolher a cidade mais próxima não visitada para o caixeiro-viajante, resultando em uma solução que é "boa o suficiente".

**Moral da História:**

* - Algoritmos gulosos e de aproximação são simples e eficientes para muitos problemas práticos, mesmo que não ofereçam a solução perfeita.
* - Problemas NP-completos são difíceis de resolver exatamente, e algoritmos de aproximação são uma estratégia útil nesses casos.

Capítulo 09 → Programação Dinâmica

### Resumo do capítulo

* **O que é programação dinâmica?** Consiste em ser uma técnica de otimização utilizada para resolver problemas complexos divididos em subproblemas menores e simples.
* **Como funciona a programação dinâmica ?**
* **Top-Down (Memoization):** Nessa abordagem, você resolve o problema de cima para baixo armazenando ( ou memorizando) as soluções do subproblemas à medida que se resolvem.Se você encontra um subproblema mais simples usa a solução armazenada em vez de recalculá-la.
* **Bottom-Up (Tabulation)**: Nessa abordagem, você resolve os problema de baixo para cima construindo uma tabela que possui solução de todos os subproblemas menores. Você começa resolvendo os subproblemas mais simples e usa essas soluções para construir soluções para subproblemas maiores.

Capítulo 10 – K-Vizinhos mais próximos

### Resumo do Capítulo

* Principais Conceitos e Objetivos:
* Construir um sistema de classificação utilizando o algoritmo dos k-vizinhos mais próximos (k-NN).
* Entender a extração de características.
* Aprender sobre regressão e como prever números, como o valor da bolsa de valores ou a preferência de um usuário por um filme.
* - Identificar quando usar o algoritmo k-NN e suas limitações.
* Classificação: Laranja vs Toranja:
* Utilizando o exemplo de frutas, o capítulo explica como classificar uma fruta como laranja ou toranja com base nas características de tamanho e cor.
* Introduz a ideia de observar os vizinhos mais próximos para classificação, exemplificando com o k-NN.
* Sistema de Recomendações:
* Apresenta como criar um sistema de recomendações de filmes, semelhante ao exemplo de frutas.
* Agrupamento de usuários por similaridade e uso dos vizinhos mais próximos para recomendar filmes.
* Extração de Características:
* Explica como extrair características de objetos (como frutas ou usuários) e como usar essas características para medir similaridade.
* Introduz a fórmula da distância para calcular a similaridade entre usuários com base em suas avaliações.
* Regressão:
* Explica como usar k-NN para prever valores, como quantos pães fazer em uma padaria ou como um usuário avaliará um filme.
* Diferença entre classificação (agrupamento) e regressão (previsão de valores).
* Similaridade de Cosseno:
* Introduz a similaridade de cosseno como uma alternativa à fórmula da distância para medir a similaridade, útil quando usuários têm diferentes padrões de avaliação.
* Escolha de Boas Características:
* Destaca a importância de escolher características relevantes e imparciais para que o algoritmo k-NN funcione corretamente.
* Introdução ao Aprendizado de Máquina:
* O algoritmo k-NN é apresentado como uma introdução ao aprendizado de máquina.
* Exemplos de aplicação incluem OCR (reconhecimento óptico de caracteres), filtros de spam e previsão de bolsa de valores.