

Pesquisa em Computação: Conceitos e Aplicações

Na computação, **pesquisa** ou **busca** é o processo de **procurar um item específico** dentro de uma estrutura de dados, como vetores (arrays), listas, árvores, arquivos ou bancos de dados. O objetivo é verificar **se o item existe** e, caso exista, **descobrir sua localização**.

Essa operação está presente em quase todos os sistemas: desde um aplicativo de mensagens que precisa encontrar uma conversa, até sistemas complexos que buscam registros em grandes bancos de dados ou na internet.

Conceitos Fundamentais

1. Elemento-alvo (ou chave de busca)

É o valor ou informação que se deseja encontrar. Pode ser um número, uma palavra, uma estrutura ou até uma combinação de atributos.

2. Espaço de busca

É o conjunto de dados onde o item será procurado. Pode ser um vetor simples, uma árvore binária, uma tabela hash, ou até um banco de dados com milhões de registros.

3. Critério de comparação

Define como os elementos serão comparados com o alvo. Por exemplo, se estamos buscando o número 5, comparamos cada item com 5 até encontrar ou esgotar as possibilidades.

Tipos de Algoritmos de Busca

◆ Busca Linear (ou Sequencial)

Percorre os elementos um a um até encontrar o item desejado. Simples e eficiente para listas pequenas ou desordenadas.

◆ Busca Binária

Divide o espaço de busca pela metade a cada passo. É muito mais rápida, mas **exige que os dados estejam ordenados** previamente.

◆ Busca em Tabelas Hash

Usa funções matemáticas (funções de hash) para calcular diretamente a posição onde um elemento deveria estar. É uma das buscas mais rápidas em média, com complexidade $O(1)$.

◆ Busca em Árvores

Em estruturas como árvores binárias de busca (BST), o item é encontrado descendo recursivamente pelas ramificações, comparando valores com os nós.

Importância da Pesquisa em Computação

A operação de busca está por trás de **diversas funcionalidades** essenciais, como:

- **Motores de busca** (Google, Bing)
 - **Consultas em bancos de dados**
 - **Sistemas de arquivos (buscar por nomes, datas, tipos de arquivos)**
 - **Autocompletar e sugestões em interfaces**
 - **IA e aprendizado de máquina (busca de padrões e vizinhos mais próximos)**
-

Complexidade e Desempenho

Um aspecto fundamental ao estudar algoritmos de busca é a **eficiência**. Alguns algoritmos são rápidos mesmo com milhões de dados (como a busca binária ou em hash), enquanto outros se tornam lentos conforme os dados crescem (como a busca linear).

A **escolha do algoritmo certo depende da estrutura dos dados e das restrições do problema**.

O que é pesquisar?

No cotidiano, **pesquisar** é procurar algo — uma informação na internet, uma palavra num livro, ou um número em uma lista. Em **computação**, **pesquisa** ou **busca** representa essa mesma ideia, mas aplicada a estruturas de dados e algoritmos. É a base de inúmeras tarefas computacionais: localizar, filtrar, acessar ou manipular dados.

Fundamentos da Busca

Para entender a operação de busca, devemos considerar alguns elementos fundamentais:

1. Conjunto de dados

É o local onde procuramos. Pode ser uma lista simples, uma matriz, um grafo, um banco de dados relacional ou um sistema distribuído na nuvem.

2. Critério de pesquisa

É o valor ou padrão que queremos encontrar: um nome, um número, um par chave-valor, ou até uma correspondência por aproximação (como em buscas com erros de digitação).

3. Estrutura de dados

A forma como os dados estão organizados influencia diretamente na eficiência da busca. Alguns exemplos:

- Vetores e listas (estrutura linear)
 - Árvores e heaps (estrutura hierárquica)
 - Tabelas de hash (acesso direto)
 - Grafos (estrutura de conexões)
 - Índices de banco de dados (estruturas otimizadas como B-trees)
-

Classificação dos algoritmos de busca

◆ Busca exata

Procura um valor específico. Ex: encontrar o número 42 em um vetor.

◆ Busca por faixa ou critério

Busca baseada em uma condição. Ex: retornar todos os valores entre 10 e 20.

◆ Busca por padrão

Busca baseada em similaridade ou correspondência. Ex: encontrar nomes que começam com "Lu" ou buscas fuzzy com tolerância a erros.

Tipos mais comuns de algoritmos de busca

Busca Linear (Sequencial)

- Verifica elemento por elemento até encontrar.
- Simples e universal.
- Funciona para listas ordenadas ou não.
- Custo: $O(n)$

Busca Binária

- Só funciona com dados ordenados.
- Divide a lista em duas partes a cada passo.
- Muito eficiente: $O(\log n)$

Hashing (Tabelas de Hash)

- Usa uma função hash para calcular diretamente o índice.
- Extremamente rápida para buscas exatas: $O(1)$ em média.
- Não serve para buscas ordenadas ou por faixa.

Busca em Árvores (BST, AVL, Red-Black Tree)

- Boa para dados com inserção, remoção e busca frequente.
- Complexidade: $O(\log n)$ se balanceada.

Busca em Grafos (BFS e DFS)

- Usada para explorar redes e conexões.
- BFS (Busca em Largura) e DFS (Busca em Profundidade) têm aplicações em mapas, redes sociais, IA etc.

Exemplos práticos de onde a busca é usada

| Contexto | Tipo de Busca Usada |
|---|--|
| Google | Busca por padrão com relevância |
| Banco de Dados SQL | Índices com árvores B e B+ |
| Arquivos em HDs | Busca sequencial com tabelas de alocação |
| Reconhecimento de voz | Busca aproximada (fuzzy search) |
| IA (como este chat!) | Busca em grafos e árvores de decisão |
| Aplicativos (e-commerce, contatos, etc) | Busca binária, hashing ou busca de texto |

Eficiência e escolha do algoritmo

A escolha do algoritmo certo depende de vários fatores:

- **Tamanho da base de dados**
- **Frequência de acesso**
- **Formato da estrutura**
- **Requisitos de tempo de resposta**
- **Recursos disponíveis (memória, CPU)**

Muitas vezes, **criar um bom índice** ou **escolher a estrutura certa** vale mais do que apenas mudar o algoritmo.

A evolução da busca

Hoje, a busca vai além do exato ou do rápido:

- **Full-text search:** pesquisa inteligente em grandes volumes de texto (ex: Elasticsearch).
- **Busca semântica:** entende o significado e contexto das palavras.
- **Busca vetorial:** usada em IA para encontrar vetores semelhantes (ex: embeddings de palavras e imagens).
- **Busca distribuída:** como o Google, que usa milhares de servidores para buscar em petabytes de dados.

A busca é uma das operações mais importantes e frequentes da computação. Entender **como ela funciona, quando usá-la e quais algoritmos estão disponíveis** é essencial para todo programador, cientista de dados ou engenheiro de software.

Seja em um simples vetor ou em um banco de dados distribuído, **pesquisar de forma eficiente** é o que permite que sistemas modernos sejam rápidos, responsivos e inteligentes.

1. BUSCA LINEAR (ou SEQUENCIAL)

Definição

A **busca linear** é o método mais direto para encontrar um elemento em uma estrutura de dados, especialmente um **vetor não ordenado**.

Consiste em **verificar cada elemento**, um por um, até encontrar o valor desejado ou terminar o vetor.

Formulação Matemática

Seja:

- $A = [a_0, a_1, a_2, \dots, a_{n-1}]$: vetor de tamanho n ,
- x : valor que estamos procurando.

A função de busca linear pode ser modelada como:

$$f(i) = \begin{cases} i, & \text{se } a_i = x \\ -1, & \text{se } x \notin A \end{cases}$$

Procuramos o **menor i** tal que $a_i = x$. A busca segue do índice 0 até $n-1$.

Complexidade

- **Melhor caso:** $O(1)$ (x está na primeira posição).
- **Pior caso:** $O(n)$ (x está na última posição ou não está presente).
- **Caso médio:** $O(n)$

Para o caso médio:

Se cada elemento tem a mesma probabilidade de ser o alvo, o número esperado de comparações é:

$$\frac{1 + 2 + 3 + \dots + n}{n} = \frac{n + 1}{2}$$
$$\rightarrow O(n)$$

Particularidades em C

- Usa-se **for** ou **while** com comparação direta.
-

- Vetores são implementados como **arrays estáticos ou dinâmicos**.
- Baixo overhead, mas ineficiente em grandes volumes de dados.

2. BUSCA BINÁRIA

Definição

A **busca binária** é um algoritmo eficiente para localizar um elemento em um **vetor ORDENADO**.

A ideia é dividir repetidamente o vetor ao meio, descartando metade do espaço de busca a cada passo.

Formulação Matemática

Seja:

- $A = [a_0, a_1, \dots, a_{n-1}]$: vetor ordenado tal que $a_0 \leq a_1 \leq \dots \leq a_{n-1}$,
- x : elemento alvo.

A busca binária consiste na avaliação recursiva:

$$\text{Meio: } m = \left\lfloor \frac{l + r}{2} \right\rfloor$$

$$f(l, r) = \begin{cases} -1, & \text{se } l > r \\ m, & \text{se } A[m] = x \\ f(l, m - 1), & \text{se } x < A[m] \\ f(m + 1, r), & \text{se } x > A[m] \end{cases}$$

Complexidade

- **Melhor caso:** $O(1)$ (x está no meio).
- **Pior caso:** $O(\log_2 n)$
- **Caso médio:** $O(\log_2 n)$

Número máximo de comparações:

$$\text{Máx. de iterações} = \left\lfloor \log_2 n \right\rfloor + 1$$

Intuição Geométrica

A busca binária funciona como uma **árvore binária de decisão** com profundidade:

\$

$$h = \left\lfloor \log_2 n \right\rfloor$$

\$

Cada nó representa uma comparação. A busca "caminha" até encontrar uma folha ou o nó desejado.

Particularidades em C

- Vetores devem estar ordenados!
- Pode ser feita de forma iterativa ou recursiva.
- Cuidados com **overflow em $m = (l + r)/2$** — em C, o ideal é:

```
int m = l + (r - l)/2;
```

- É possível implementar usando **ponteiros** para maior flexibilidade.

Comparação entre Busca Linear e Binária

| Característica | Busca Linear | Busca Binária |
|-------------------------|--------------|----------------|
| Pré-requisito | Nenhum | Vetor ordenado |
| Complexidade (pior) | $O(n)$ | $O(\log n)$ |
| Melhor caso | $O(1)$ | $O(1)$ |
| Caso médio | $O(n)$ | $O(\log n)$ |
| Implementação | Simples | Mais complexa |
| Eficiência em grandes n | Ruim | Excelente |

Custo de Memória

- Ambas usam **espaço constante $O(1)$** se implementadas iterativamente.
- Se a busca binária for **recursiva**, o custo de memória será:

\$

$$O(\log n) \quad \text{(para chamadas na pilha)}$$

\$

Aplicações e Reflexões

| Algoritmo | Usado quando? |
|-----------|---------------|
|-----------|---------------|

| Algoritmo | Usado quando? |
|---------------|--|
| Linear Search | Dados pequenos ou não ordenados |
| Binary Search | Dados grandes e ordenados, com busca frequente |

Exemplo real:

- A **função `bsearch` da `libc` em C** implementa busca binária genérica com ponteiros.
- Ferramentas como `std::binary_search` em C++ fazem isso com **templating**.



Formulações Adicionais

Complexidade de Comparações — Análise de Informação

No modelo de comparação, a busca binária é **ótima**:

- Para n elementos, o número mínimo de comparações requeridas para localizar um item é:
 $\Omega(\log n)$

Nenhum algoritmo baseado apenas em comparação pode fazer melhor do que isso — isso é provado usando **teoremas de complexidade da informação**.

Perfeito! Vamos aprofundar ainda mais nos fundamentos matemáticos, computacionais e teóricos das **buscas linear e binária**, entrando agora em tópicos como:

1. 🌲 **Árvores de decisão e limites inferiores teóricos**
2. 📏 **Análise detalhada de complexidade com constantes ocultas**
3. 🔄 **Variações da busca binária (primeira, última ocorrência, etc.)**
4. 🏠 **Busca em domínios contínuos (float, double)**
5. 🗝️ **Relação com entropia e teoria da informação**

1. 🌲 Árvore de Decisão — Modelo Teórico de Comparações

Um dos modos clássicos de provar o **limite inferior da busca binária** é com **árvore de decisão por comparações**.

🧠 Conceito

- Cada **nó** da árvore representa uma comparação (ex: $x == A[m]$?).
- Cada **ramo** representa o resultado da comparação ($<$, $>$, $==$).
- As **folhas** representam a posição onde o elemento foi encontrado ou a ausência dele.

Se temos n elementos distintos, então precisamos de $n+1$ folhas (para cada posição ou ausência).



Altura mínima de uma árvore binária com k folhas:

$\$$
 $h \geq \lceil \log_2(k) \rceil$
 $\rightarrow \text{Logo, } \Omega(\log_2(n)) \text{ comparações}$
 $\$$

📌 Conclusão teórica importante:

A busca binária é assintoticamente ótima em termos de comparações quando se trabalha com vetores ordenados.

2. 🧱 Constantes Ocultas nas Complexidades

Embora $O(n)$ vs. $O(\log n)$ seja teórico, em prática devemos considerar:

| Tipo de busca | Custo por comparação | Cache locality | Overhead |
|---------------|----------------------|---------------------------|----------|
| Linear | Baixo (~1 ciclo) | Ótimo (acesso sequencial) | Nenhum |
| Binária | Médio (~10 ciclos) | Ruim (acessos saltados) | Maior |

Ou seja: **em vetores pequenos**, a busca linear pode ser mais rápida **na prática** que a binária, **apesar da sua complexidade maior**, devido a:

- Pré-fetching da CPU,
- Cache de memória,
- Menor número de saltos (branches).

3. 🔄 Variações da Busca Binária

Em C (ou qualquer linguagem), é comum **modificar a busca binária** para atender a diferentes situações:

📌 a) Encontrar a primeira ocorrência de x :

```
// Continua buscando à esquerda mesmo quando encontra x
```

📌 b) Encontrar a última ocorrência:

```
// Continua buscando à direita mesmo quando encontra x
```

📌 c) Encontrar o menor índice tal que $A[i] \geq x$ (lower bound)

Usado em problemas como agendamento, alocação, etc.

📌 d) Buscar valor em função contínua

Se temos uma função $f(x)$ monótona contínua (ex: decrescente), podemos usar **busca binária nos reais** para encontrar um valor com erro tolerado ϵ :

\$
 $\text{Enquanto } |r - l| > \epsilon$:
 $m = \frac{l + r}{2}$
 $\text{chegar se } f(m) \leq k$
\$

Isso é muito comum em:

- Geometria computacional
- Problemas de otimização
- Física computacional

4. 🏰 Busca em Domínios Contínuos (float, double)

✅ Problemas

1. Comparações diretas com `==` podem falhar com `float/double`.
2. É necessário usar **intervalo de tolerância** ϵ :

```
if (fabs(f(m) - x) < epsilon)
```

3. Loop de parada precisa garantir:

\$
 $|l - r| < \epsilon$
 $\Rightarrow \text{convergência numérica}$
\$

5. 🔑 Teoria da Informação

A **busca binária é o algoritmo ótimo** quando se mede o **custo em bits de informação** necessário para isolar um item entre n opções.

Informação necessária:

\$
 $I = \log_2(n) \text{ bits}$
\$

A cada comparação binária, ganhamos no máximo **1 bit de informação**:

- "x está à esquerda ou à direita?"

Então, o **número mínimo teórico de comparações** para localizar algo em uma lista de n elementos é:

$$\boxed{\log_2(n)} \text{ comparações}$$

Resumo das Diferenças Essenciais

| Aspecto | Busca Linear | Busca Binária |
|--------------------------|---------------------------------|-------------------------------------|
| Pré-requisito | Nenhum | Vetor ordenado |
| Complexidade (pior) | $O(n)$ | $O(\log n)$ |
| Base teórica | Percurso sequencial | Árvores binárias / entropia |
| Ideal para | Pequenas listas, qualquer ordem | Grandes listas ordenadas |
| Implementação em C | Simples | Exige cuidado (overflow, ponteiros) |
| Cache e performance real | Alta em listas pequenas | Baixa em listas pequenas |
| Possível paralelismo | Sim (SIMD) | Difícil de paralelizar |

OTIMIZAÇÕES PARA BUSCA LINEAR — EM DETALHES

✓ 1. Busca com Sentinela (Sentinel Search)

Objetivo:

Eliminar a comparação $i < n$ em cada iteração do loop, o que reduz o número de verificações por ciclo.

Como funciona:

Você adiciona o elemento x (valor buscado) ao final da lista, de modo que **a busca sempre irá encontrar x eventualmente**, evitando a verificação de fim.

Implementação:

```
int linear_search_sentinel(int A[], int n, int x) {
    A[n] = x; // Coloca sentinela (x) no final (A deve ter espaço extra!)
    int i = 0;
    while (A[i] != x) {
        i++;
    }
}
```

```
    return (i < n) ? i : -1;
}
```

✅ Vantagens:

- Elimina a verificação `i < n`, ganhando **até 30% de desempenho em algumas arquiteturas**.
- Especialmente útil em **sistemas embarcados** ou arquiteturas simples (como AVR/ARM Cortex-M).

⚠ Cuidados:

- Você precisa garantir **espaço extra no array** (ou trabalhar com buffers maiores).
- O array original é **modificado** temporariamente.

✅ 2. Loop Unrolling (Desenrolamento de Laço)

📌 Objetivo:

Reduzir a quantidade de instruções de controle de loop, processando **vários elementos por iteração**.

🧠 Como funciona:

O compilador ou programador manualmente escreve múltiplas comparações por ciclo:

```
int linear_search_unrolled(int A[], int n, int x) {
    int i = 0;
    while (i <= n - 4) {
        if (A[i] == x) return i;
        if (A[i+1] == x) return i + 1;
        if (A[i+2] == x) return i + 2;
        if (A[i+3] == x) return i + 3;
        i += 4;
    }
    while (i < n) { // Últimos elementos restantes
        if (A[i] == x) return i;
        i++;
    }
    return -1;
}
```

✅ Vantagens:

- Reduz o número de saltos (**jump**) e comparações de índice.
- Melhora o desempenho por uso mais eficiente do **pipeline** da CPU.

⚠ Cuidados:

- Código menos legível.
- Pode ser ineficiente para vetores muito pequenos (overhead fixo).
- O compilador moderno (como GCC/Clang) pode fazer isso automaticamente com `-O3`.

✓ 3. Busca Paralela com SIMD

📌 Objetivo:

Comparar **vários elementos em paralelo** usando instruções de vetor (ex: AVX2 ou NEON).

🧠 Como funciona:

Com **SIMD**, você pode comparar por exemplo 8 inteiros de 32 bits de uma só vez usando registradores de 256 bits (AVX2):

- Usa instruções como `_mm256_cmpeq_epi32` (AVX)
- Ou bibliotecas como `emmintrin.h` ou `immintrin.h`

🧠 Exemplo (pseudo):

```
__m256i target = _mm256_set1_epi32(x);
for (i = 0; i < n; i += 8) {
    __m256i values = _mm256_loadu_si256((__m256i*)&A[i]);
    __m256i cmp = _mm256_cmpeq_epi32(values, target);
    int mask = _mm256_movemask_epi8(cmp);
    if (mask != 0) return i + pos_in_mask(mask);
}
```

✓ Vantagens:

- Pode acelerar até **4x a 8x** em comparação à busca linear sequencial.
- Ideal para vetores grandes, com suporte SIMD.

⚠ Cuidados:

- Requer **compiladores modernos** e CPUs com suporte SIMD.
- Mais difícil de portar e manter.
- Funciona melhor quando o array está **alinhado na memória**.

✓ 4. Busca Paralela com Multi-threading

📌 Objetivo:

Dividir o vetor em blocos e buscar simultaneamente com múltiplas **threads** (ex: com OpenMP ou pthreads).

💡 Exemplo (OpenMP):

```
int result = -1;
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    if (A[i] == x) {
        #pragma omp critical
        {
            if (result == -1 || i < result)
                result = i;
        }
    }
}
```

✅ Vantagens:

- Aceleração em CPUs com múltiplos núcleos.
- Boa para vetores **muito grandes** (ex: milhões de elementos).

⚠️ Cuidados:

- Overhead de sincronização (**critical**) pode anular os ganhos em vetores pequenos.
- Não garante ordem — pode não retornar a **primeira ocorrência** corretamente sem controle.

✅ 5. Otimização Baseada em Heurística de Frequência

📌 Objetivo:

Se alguns elementos são **mais prováveis de serem buscados**, coloque-os no início.

💡 Exemplo:

```
// Se o elemento x = 5 é muito frequente:
int A[] = {5, 3, 4, 5, 7, 1, 6}; // Coloque 5 mais cedo
```

Ou usar **movimento ao topo (move-to-front)** após cada acerto:

```
if (A[i] == x) {
    if (i != 0) {
        int temp = A[i];
        A[i] = A[0];
        A[0] = temp;
    }
}
```

```
    return 0;  
}
```

✅ Vantagens:

- Efetivo quando há **acessos repetitivos e não uniformes**.
- Autoajustável dinamicamente.

⚠️ Cuidados:

- Modifica o vetor — não funciona bem em arrays imutáveis.
- Pode prejudicar algoritmos que dependem da ordem dos dados.

RESUMO FINAL — OTIMIZAÇÕES DE BUSCA LINEAR

| Otimização | Melhor Caso | Ganho esperado | Quando usar | Limitações |
|----------------------------|--------------|-------------------------|-----------------------------------|---------------------------|
| Sentinela | Qualquer | 10–30% | Vetores pequenos, tempo crítico | Requer espaço extra |
| Loop unrolling | Médio | 5–20% | Vetores médios, otimização manual | Código verboso |
| SIMD | Grande | 4x–8x (em hardware bom) | Vetores grandes, CPU moderna | Complexidade, alinhamento |
| Multi-threading | Muito grande | 2x–10x (multi-core) | Vetores muito grandes | Overhead de sincronização |
| Move-to-front / Heurístico | Frequente | Até $O(1)$ acesso | Dados com padrões repetitivos | Modifica vetor original |

Otimizacao busca binária

✅ 1. Prevenir Overflow

🔧 O que é:

Evita estouro de inteiros ao calcular o índice do meio.

💣 Problema:

A fórmula tradicional:

```
m = (l + r) / 2;
```

pode causar *overflow inteiro* se `l` e `r` forem grandes (ex: `INT_MAX - 1`).

✅ Solução correta:

```
m = l + (r - l) / 2;
```

🧠 Por que funciona:

Essa versão é equivalente matematicamente, mas evita que `l + r` ultrapasse o limite de `int`.

✅ 2. Branchless Binary Search

🔧 O que é:

Elimina instruções `if` dentro do loop para evitar *branch misprediction* pela CPU.

⚠️ Problema com `if`:

```
if (A[m] < x)
    l = m + 1;
else
    r = m;
```

Se o padrão de dados for imprevisível, a CPU pode errar a predição do `if`, desperdiçando ciclos.

✅ Solução sem ramificações:

```
l = (A[m] < x) ? m + 1 : l;
r = (A[m] >= x) ? m : r;
```

🧠 Por que funciona:

- Usa **operadores ternários** (condições embutidas).
- Mantém o pipeline da CPU mais eficiente.
- Ideal quando há **muitas buscas seguidas**.

✅ 3. Busca Interpolada

🔍 O que é:

Ajusta o índice do meio usando **proporcionalidade**, útil quando os dados são *uniformemente distribuídos*.

 Fórmula:

$$m = l + \frac{(x - A[l]) \cdot (r - l)}{A[r] - A[l]}$$


Exemplo em C:

```
while (l <= r && x >= A[l] && x <= A[r]) {  
    int m = l + ((double)(x - A[l]) * (r - l)) / (A[r] - A[l]);  
    if (A[m] == x) return m;  
    if (A[m] < x) l = m + 1;  
    else r = m - 1;  
}
```

 Por que é melhor:

- Em arrays com espaçamento regular (ex: [10, 20, 30, ..., 1000]), é **mais rápida** que a binária pura.
- Complexidade média: $O(\log \log n)$
- **Pior caso** ainda é $O(n)$, por isso deve-se testar a adequação.


✓ 4. Busca Binária Vetorial (SIMD)

 O que é:


Usa instruções SIMD (Single Instruction, Multiple Data) como SSE/AVX para comparar vários valores de uma vez só.

 Vantagem:

- Em vez de comparar x com um $A[m]$, você pode comparar com 4 ou 8 posições ao mesmo tempo.

 Como funciona:

- Carrega blocos de dados em registradores vetoriais.
- Executa comparações simultâneas.
- Usa intrínsecos como `_mm256_cmp_epi32`, `_mm256_movemask_ps`, etc.

 Exemplo (simplificado):

```
// Pseudocódigo SIMD
load A[m], A[m+1], ..., A[m+7] in SIMD register
compare all to x
get bitmask of matches
```

💡 Aplicação:

- Ideal para **bancos de dados in-memory**
- **Indexadores de texto, motores de busca, motores de inferência vetorial**

✓ 5. Busca Binária com Estimativa Inicial (Exponencial)

⚡ O que é:

Usa uma busca **exponencial** para encontrar um intervalo pequeno onde x pode estar, e depois aplica busca binária nesse intervalo.

💡 Quando usar:

- Quando não se conhece o tamanho total do array (ex: listas virtuais ou streams).
- Quando o elemento está **muito próximo do início**.

📅 Etapas:

```
int bound = 1;
while (bound < n && A[bound] < x) bound *= 2;
// Busca binária entre bound/2 e min(bound, n-1)
```

📊 Complexidade:

- Tempo total: $O(\log i)$, onde i é a posição do item desejado.
- Útil em **vetores ordenados com acesso incremental**.

✓ 6. Busca Ternária (Curiosidade)

🔍 O que é:

Divide o vetor em três partes ao invés de duas:

- Usa dois pontos médios: m_1 , m_2 .
- Verifica onde está x entre essas três regiões.

💡 Exemplo:

```
int m1 = l + (r - l) / 3;
int m2 = r - (r - l) / 3;
```

🧠 Quando usar:

- Pouco usada para busca em arrays.
- Mais comum em **otimização unimodal** (ex: encontrar valor mínimo de uma função convexa).

✅ 7. Busca Binária em Domínios Contínuos (com **double**)

📈 Problema:

Comparações com ponto flutuante podem ser imprecisas (ex: `x == A[m]` pode nunca ser verdadeiro).

✅ Solução:

Use um **ϵ (epsilon)** para comparar com tolerância:

```
#define EPSILON 1e-9
while (right - left > EPSILON) {
    double mid = (left + right) / 2.0;
    if (f(mid) < target) left = mid;
    else right = mid;
}
```

💡 Aplicações:

- Raízes de equações
- Otimização contínua
- Geometria computacional

📚 Resumo das Otimizações

| Otimização | Vantagem | Quando aplicar |
|-----------------------|----------------------------------|----------------------------------|
| Prevenir overflow | Segurança e portabilidade | Sempre |
| Branchless search | +10~40% em CPUs modernas | Listas grandes, muitas buscas |
| Interpolada | $O(\log \log n)$ média | Dados quase uniformes |
| Vetorial (SIMD) | Alto desempenho paralelo | Bancos de dados, vetores grandes |
| Exponencial + binária | $O(\log i)$, onde i é posição | Streams, listas encadeadas |
| Ternária | Otimização de funções unimodais | Não usada para busca clássica |

| Otimização | Vantagem | Quando aplicar |
|------------------|-----------------------------|--------------------------------------|
| Binária contínua | Precisão com floats/doubles | Domínios reais, algoritmos numéricos |