Pesquisa e Ordenação de Dados

Unidade 1:

Contextualização

Pesquisa e Ordenação de Dados

- Ementa (PPC):
  - Métodos de ordenação. Busca linear e binária. Organização de arquivos. Persistência de dados. Ordenação externa. Índices (árvores B+ e hashing). Compactação de dados. Implementações com linguagem imperativa estruturada.
- Objetivo geral (PPC):
  - Utilizar estruturas de dados avançadas para ordenação e pesquisa de informações. Construir algoritmos para persistir dados e tratar dados persistidos. Analisar comparativamente os métodos de ordenação quadráticos e os métodos MergeSort, HeapSort e QuickSort.

Por que estudar pesquisa?

search

- Uma das funcionalidades mais úteis de um computador é sua capacidade de armazenar e recuperar dados
- Dados podem ser armazenados:
  - Na memória principal
  - Na memória secundária
    - Arquivos de texto, arquivos binários ou por intermédio de um SGBD
- Uma **pesquisa** ou (busca) visa **recuperar** um ou mais itens de um arquivo ou conjunto de dados
- É interessante que a busca não precise percorrer exaustivamente todo o conjunto de dados
  - Para isto, diversos algoritmos e estruturas de dados foram desenvolvidos, conforme veremos adiante.

Por que estudar pesquisa?

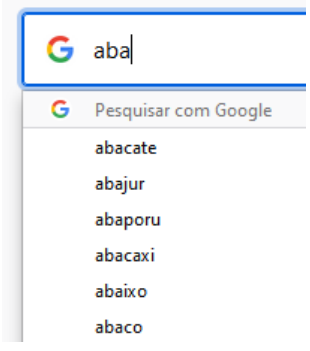
- Descobrir se um item existe em um conjunto de dados

existe o login asebben123 no cadastro de usuários?

- Recuperar dados associados a uma chave de busca única

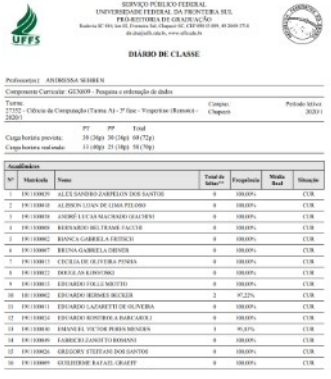
qual a descrição e o preço do produto com código de barras 123456789?

- Listar valores que correspondem a um determinado critério de busca



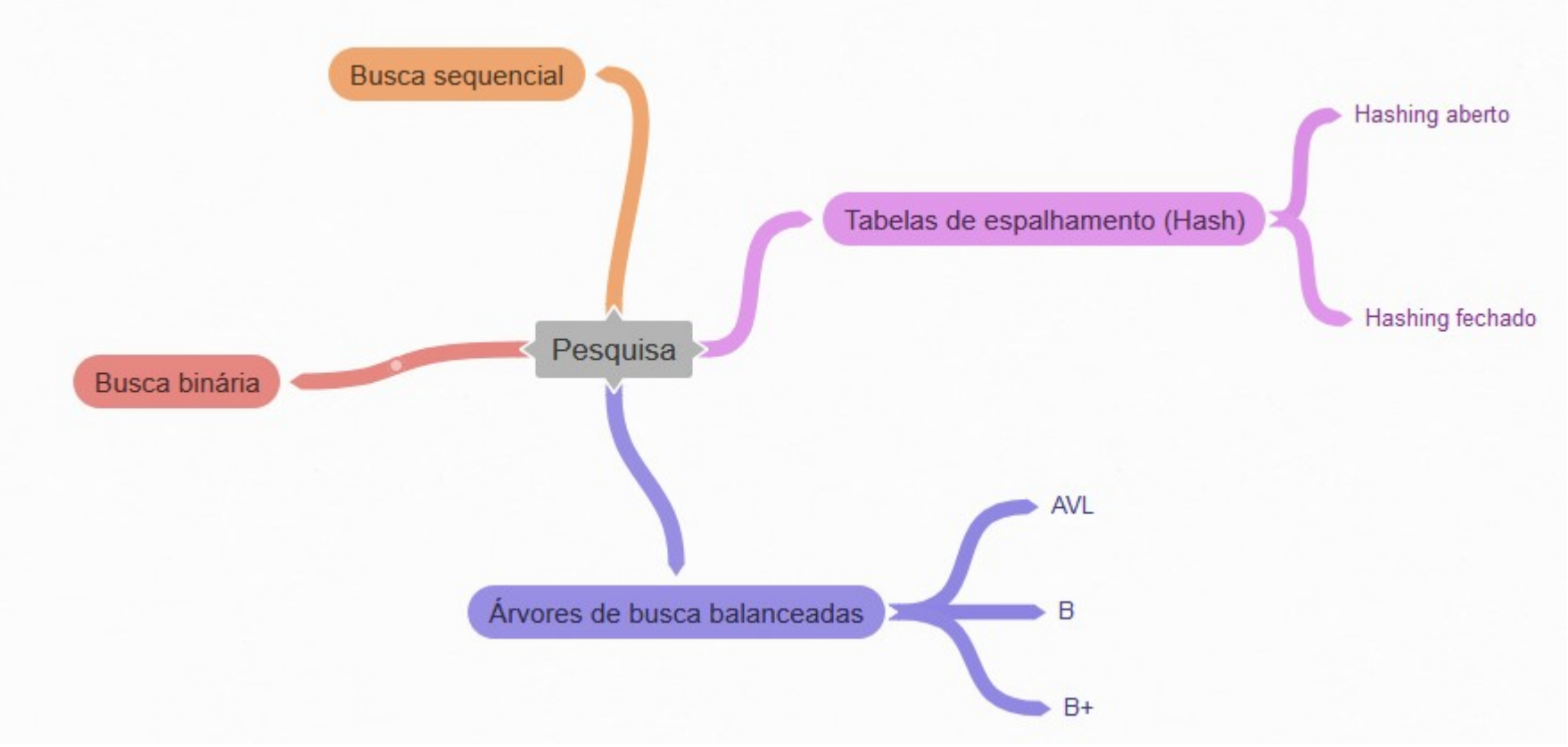
quais termos de busca começam com a string aba?

- Listar todos os itens de um conjunto



quem são os alunos matriculados na turma 27352?

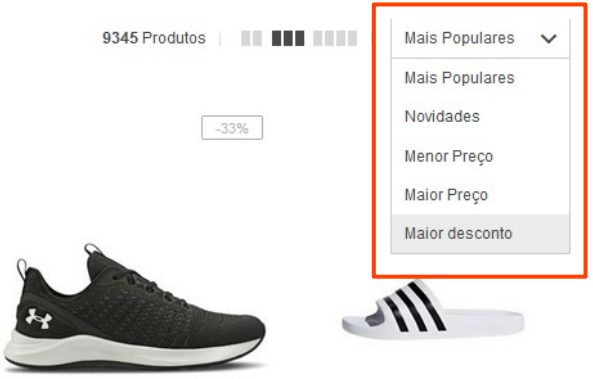
Pesquisa


Busca sequencial
Hashing aberto
Tabelas de espalhamento (Hash)
Hashing fechado
Busca binária
Pesquisa
AVL
Árvores de busca balanceadas
B
B+

Por que estudar ordenação?

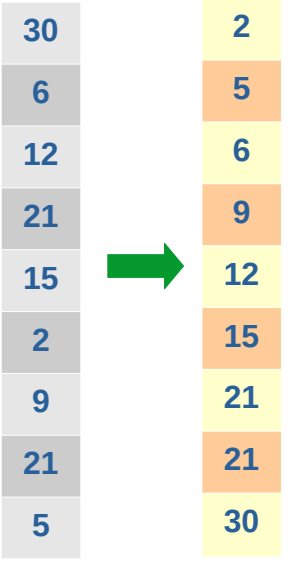
- **Rearranjar um conjunto** de itens em ordem (ascendente ou descendente) para agilizar sua recuperação posterior é uma tarefa bastante comum em nosso cotidiano
- Esta atividade de colocar as coisas em ordem está presente na maioria das aplicações nas quais objetos armazenados precisam ser pesquisados e recuperados, tais como dicionários, índices de livros, tabelas e arquivos (ZIVIANI, 2004).

Selecionar	Foto do usuário	Nome / Sobrenome	Endereço de email	Cidade/Município	País	Último acesso ao curso
☐		ANDRESSA SEBBEN	asebben@uffs.edu.br	CHAPECÓ/SC	Brasil	4 segundos
☐		[redacted]	[redacted]	CHAPECÓ/SC	Brasil	1 dia 7 horas
☐		[redacted]	[redacted]	CHAPECÓ/SC	Brasil	4 dias 6 horas



- A ordenação é um problema largamente estudado na Ciência da Computação.
  - Veremos diversos algoritmos, com diferentes características.

Por que estudar ordenação?

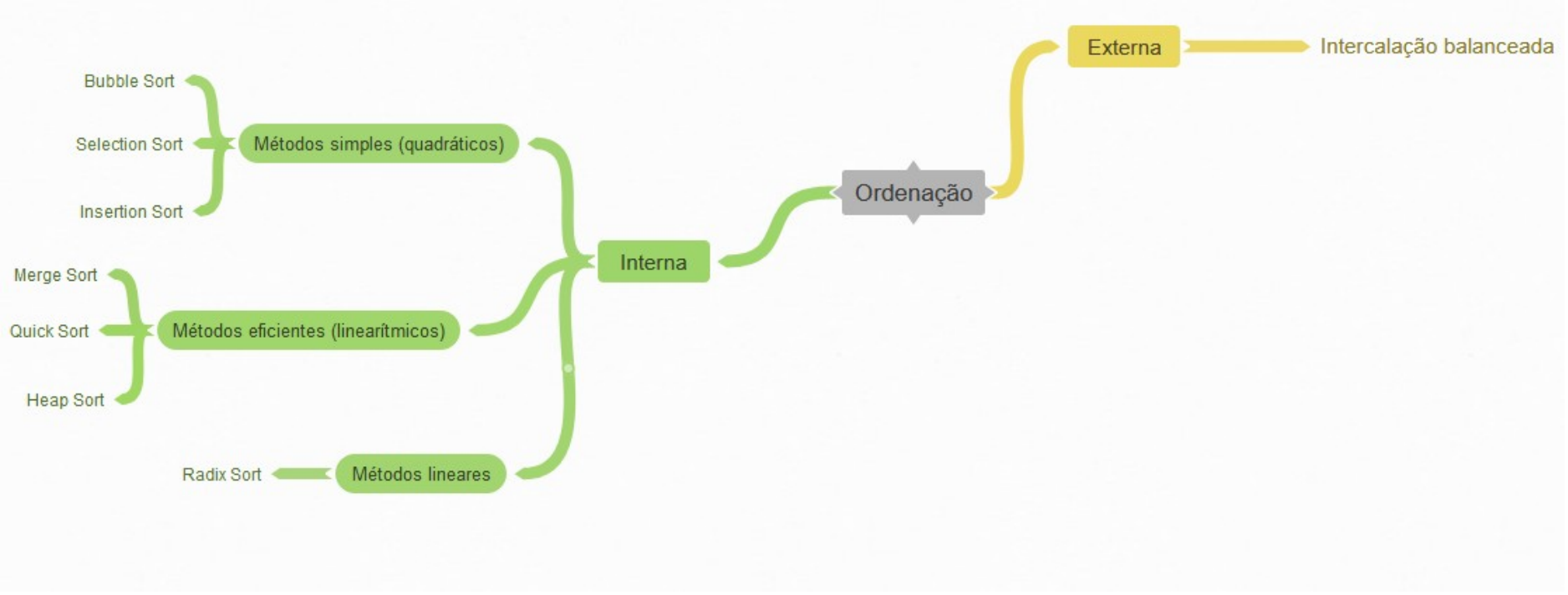


- Quando uma lista de valores está ordenada, diversos problemas tornam-se fáceis - ou, pelo menos, mais fáceis (VISUALGO.NET):
  - Buscar por um valor específico na lista (busca binária)
  - Encontrar o valor mínimo, o valor máximo ou o k-ésimo menor/maior valor da lista
  - Verificar a unicidade de um valor e deletar duplicatas
  - Contar quantas vezes um valor específico aparece na lista
  - Calcular a mediana, bem como identificar os valores discrepantes ou "*outliers*"
  - Obter a intersecção/união entre a lista ordenada A e outra lista ordenada B
  - Encontrar valores x e y pertencentes à lista tal que x+y resulta em um valor alvo z
- Aplicações menos óbvias incluem: compressão de dados, computação gráfica, biologia computacional, balanceamento de carga em sistemas paralelos, etc (SEDGEWICK, 2016).

Ordenação (*Sorting*)

- Ordenação interna:
  - quando o conjunto de dados a ser ordenado encontra-se integralmente na memória principal;
  - qualquer registro pode ser imediatamente acessado;
- Ordenação externa:
  - quando o conjunto a ser ordenado não cabe na memória principal e, por isso, precisa ser armazenado em um arquivo em memória secundária (fita ou disco);
  - os registros são acessados sequencialmente ou em grandes blocos.

Ordenação – O que veremos



Análise de Algoritmos

- Dada a relevância da pesquisa e da ordenação, é altamente desejável que os métodos utilizados sejam **eficientes**
  - Não basta resolver o problema, é necessário ter um desempenho aceitável na prática!
  - Um algoritmo simples e rápido para um conjunto pequeno de dados pode se tornar inviável para um conjunto com milhões de registros
- Analisar a **complexidade** de um algoritmo = estimar os recursos necessários para que a tarefa seja completada
  - Complexidade de tempo → medida do tempo de execução
  - Complexidade de espaço → medida da quantidade de memória

Medida mais comum

Análise de Algoritmos

- Formas de computar:
  - **Análise empírica** (*benchmark*)
    - Implementar o algoritmo e realizar **experimentos** com diversos conjuntos de dados, de diferentes tamanhos e características, **medindo** o tempo de execução/consumo de memória;
    - Os resultados são dependentes da linguagem de programação utilizada, do compilador, da capacidade do hardware, da quantidade de processos sendo executados, etc;
    - Computa custos não aparentes (como alocação de memória);
    - Permite comparar computadores, linguagens, etc;
    - Depende da habilidade do programador;
    - Cenários de teste limitados.

Análise de Algoritmos

- Formas de computar:
  - **Análise matemática** (modelo teórico)
    - Estudo **formal** das propriedades do algoritmo;
    - Considera um computador idealizado onde cada operação executa em tempo constante e de forma sequencial;
      - Operações simples (comparações, atribuições, cálculos, incrementos, acesso a um elemento em um array) possuem mesmo custo
    - Realiza simplificações buscando considerar apenas o custo dominante;
    - Objetivo: **estimar como o algoritmo se comporta em função do tamanho do conjunto de dados** (entrada);
    - Independe de aspectos específicos de hardware, linguagem, compilador e ambiente de execução;

Complexidade

- Ao olhar para complexidade dos algoritmos, podemos:
  - predizer seu desempenho;
  - comparar diferentes algoritmos que resolvem o mesmo problema;
  - avaliar sua viabilidade, provendo algumas garantias de que sua execução será completada no tempo esperado.
- Ao invés de olhar para o tempo exato de execução (o qual depende de fatores relacionados ao ambiente de execução), a noção de complexidade possibilita analisar e entender o **comportamento** do algoritmo, isto é,
  - **como o tempo de execução cresce à medida que o tamanho da entrada cresce.**

Análise de Complexidade

Exemplo

```
int menor(int *A, int n) {
  int i, menor;
  menor = A[0];
  for(i = 1; i < n; i++) {
    if(A[i] < menor)
      menor = A[i];
  }
  return menor;
}
```

1

2+(n-1)+(n-1)

n-1

n-1

1

2 da inicialização do laço
n-1 comparações
n-1 incrementos

no pior caso

= 1 + 2 + n - 1 + n - 1 + n - 1 + n - 1 + 1
= 4n

4n = **n**

Comportamento assintótico

60	50	40	30	20	10
0	1	2	3	4	5

Notação Assintótica

- Quando trabalhamos com **N** objetos, algumas operações tomarão tempo proporcional a **N**
  - Para valores pequenos de **N**, até mesmo um algoritmo ruim pode ter desempenho aceitável
  - Nos interessa saber a ordem de crescimento do algoritmo para **valores grandes** de **N**
  - À medida que **N** cresce, as constantes aditivas e multiplicativas têm cada vez menos impacto, podendo até mesmo se tornar irrelevantes

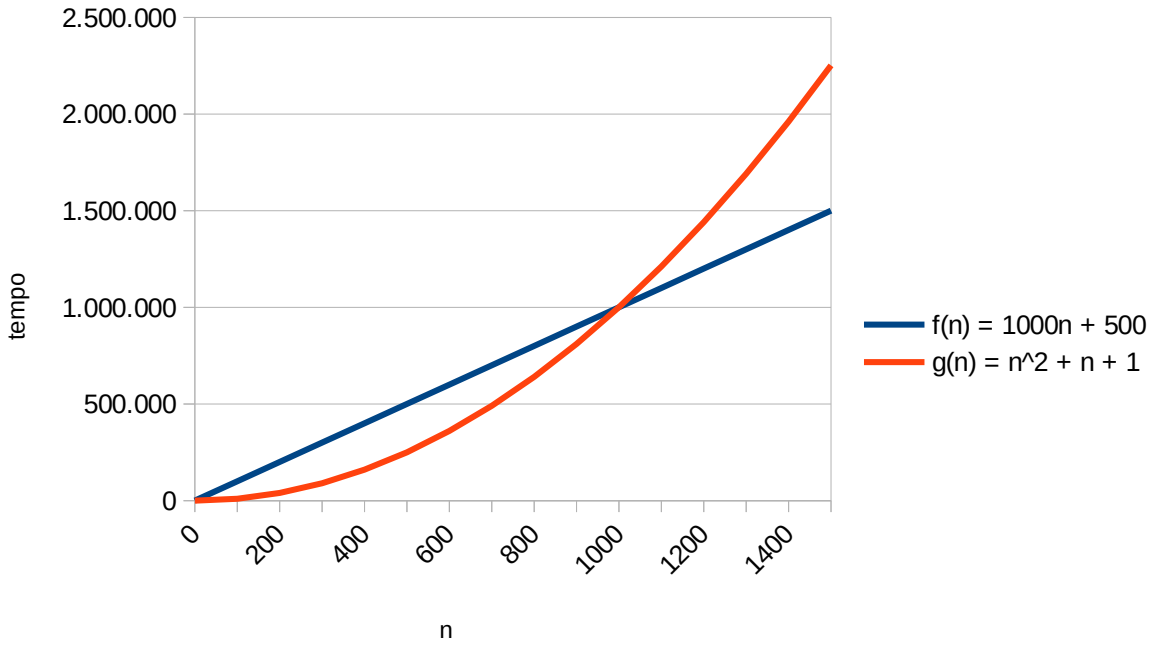
- Por exemplo, para valores de **N** suficientemente grandes, as funções

  n^2 $(3/2)n^2$ $9999n^2$ $n^2/1000$ n^2+100n

  têm todas a mesma taxa de crescimento e, portanto, são todas "equivalentes".

Notação Assintótica

- Ex: considere as funções:
  - **f(n) = 1000n + 500**
  - **g(n) = n^2 + n + 1**



  - Existe um valor de n a partir do qual **g(n)** é sempre maior do que **f(n)**, tornando os demais termos e constantes pouco significativos.

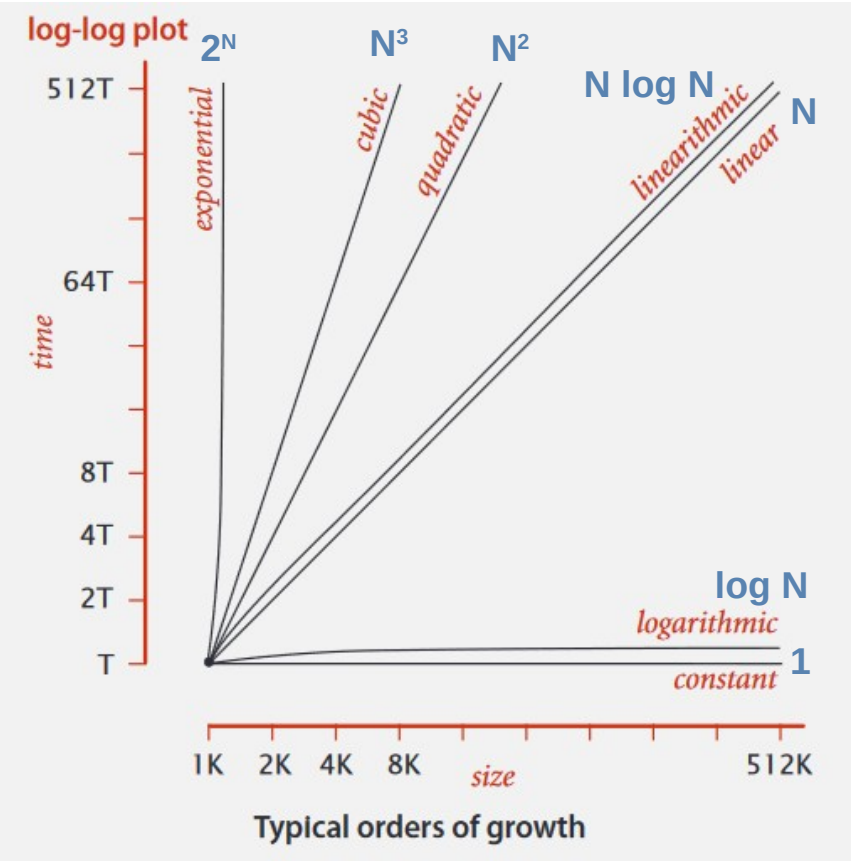
Notação Assintótica

- Na notação assintótica, considera-se apenas o **termo dominante** da equação (ou seja, o termo de maior grau). Os termos de grau menor e as constantes aditivas e multiplicativas são desconsiderados.
- Ex:

$f(n) = 75$	=	$f(n) = 1$
$f(n) = 2n + 1$	=	$f(n) = n$
$f(n) = n^2 + n$	=	$f(n) = n^2$
$f(n) = 10n^3 + 4n - 1$	=	$f(n) = n^3$

quando função não possui n, então o comportamento assintótico é 1 (constante)

Classes de Complexidade (*Order of growth*)



Fonte: (SEDGEWICK, 2016)

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Fonte: (SEDGEWICK, 2016)

Complexidade

- **Melhor caso** (*best case*)
  - Representado pela letra grega **Ω** (ômega)
  - Consiste na entrada mais “fácil” (e a que deve preferencialmente ser buscada)
  - Constitui o limite inferior de custo (não pode ser mais rápido)
- **Pior caso** (*worst case*)
  - Representado pela letra **O** (ômicron), também chamada de “**Big O**”
  - Representa a entrada mais difícil (a que faz o algoritmo executar o maior número de operações)
  - Constitui o limite superior de custo, servindo como uma garantia para as demais entradas (não pode ser mais lento)
- **Caso médio** (*average case*)
  - Representado pela letra grega **θ** (theta)
  - Custo esperado para uma entrada aleatória
  - Difícil de determinar na maioria dos casos

Notação “Big O”: a mais utilizada para expressar a complexidade do algoritmo

Exemplo

Busca sequencial em um vetor de tamanho n

- Melhor caso: **Ω(1)**
  - O valor procurado é o primeiro do vetor
- Pior caso: **O(n)**
  - O valor procurado é o último ou não faz parte do vetor
  - Será necessário visitar todos os **n** elementos do vetor até encontrar o valor procurado
- Caso médio: **θ(n)**
  - Será necessário visitar na média **n/2** elementos do vetor até encontrar o valor procurado

Dizemos que o algoritmo é O(n), ou seja, linear