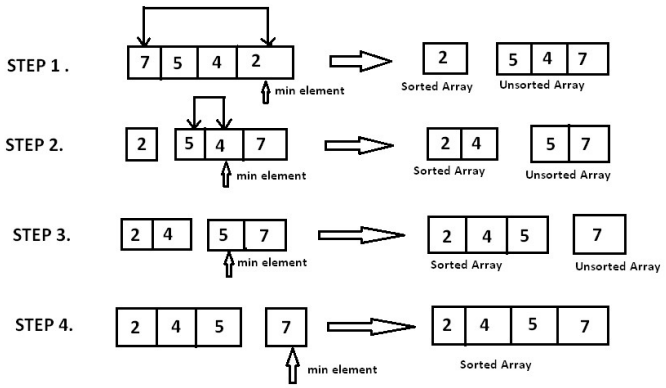
Selection Sort

Como funciona:
A cada iteração, encontra o menor elemento do restante da lista e coloca na posição certa.

Pontos fortes:
- Simples de implementar
- Número fixo de comparações (independente da ordem da lista)
- Boa escolha se trocas forem mais caras que comparações

Pontos fracos:
- Lento: sempre faz O(n²) comparações e trocas
- Não adaptativo: não melhora se a lista já estiver ordenada
- Pouco eficiente para listas grandes

Ideal para:
Aprender lógica de ordenação e ensinar algoritmos básicos. Pouco usado na prática.



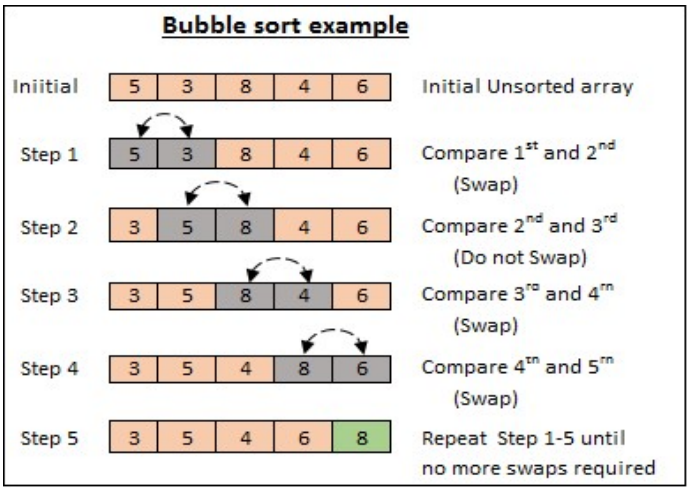
Bubble Sort

Como funciona:
Percorre a lista várias vezes, trocando vizinhos fora de ordem. Os maiores "borbulham" para o final.

Pontos fortes:
- Muito fácil de entender e implementar
- Pode ser otimizado para parar se a lista já estiver ordenada (versão adaptativa)

Pontos fracos:
- Lento: O(n²) no pior caso
- Mesmo com otimizações, continua ineficiente em listas médias ou grandes
- Muitas trocas desnecessárias

Ideal para:
Ensino básico e quando a lista é muito pequena. Nunca é usado em produção.



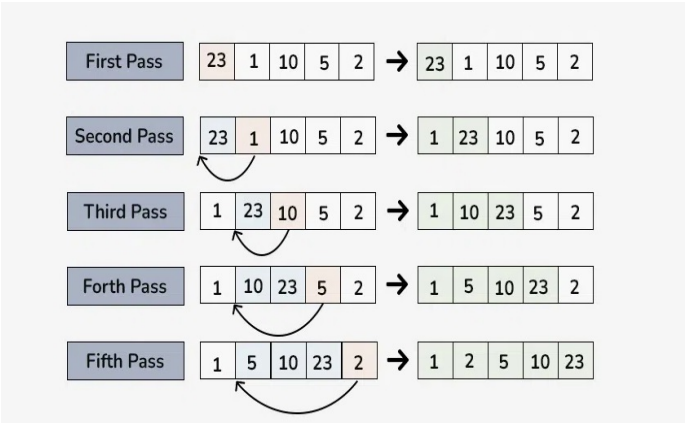
Insertion Sort

Como funciona:
Pega cada elemento da lista e o insere na posição correta da parte que já está ordenada.

Pontos fortes:
- Excelente para listas quase ordenadas
- Poucas trocas: só movimenta o necessário
- Funciona bem com listas pequenas
- Simples e estável (mantém ordem dos iguais)

Pontos fracos:
- O(n²) no pior caso (lista invertida)
- Ainda lento para listas grandes

Ideal para:
Listas pequenas ou quase ordenadas. Muito usado como parte de algoritmos maiores (ex: no final de um Quick Sort).



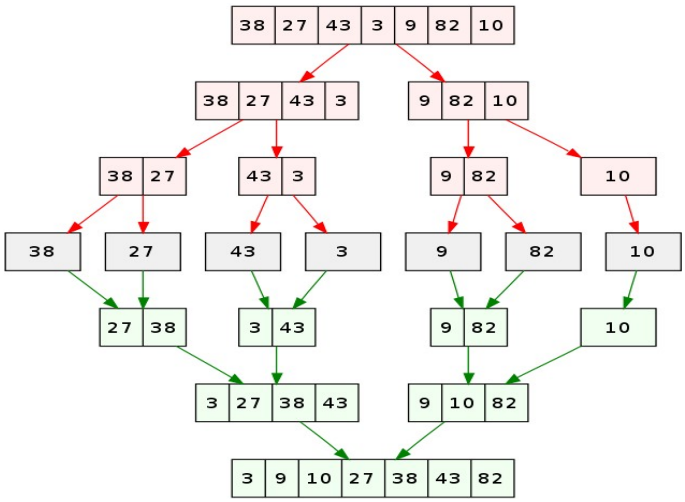
Merge Sort

Como funciona:
Divide a lista em duas até ter só elementos únicos e depois mescla tudo em ordem.

Pontos fortes:
- Complexidade garantida: O(n log n) mesmo no pior caso
- Muito estável
- Excelente para listas grandes
- Usado quando é necessário manter a estabilidade da ordenação

Pontos fracos:
- Usa mais memória: cria muitas listas temporárias
- Mais lento que Quick Sort na média (mas mais seguro)

Ideal para:
Listas muito grandes ou quando a estabilidade da ordenação importa (ex: ordenar por nome sem bagunçar a ordem anterior).



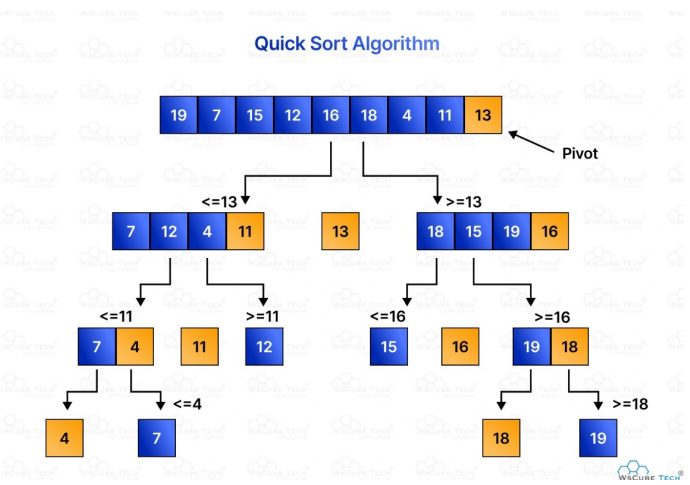
Quick Sort

Como funciona:
Escolhe um pivô, divide em menores e maiores, e ordena recursivamente.

Pontos fortes:
- Muito rápido na prática
- Complexidade média: O(n log n)
- Usa pouca memória (em versões in-place)
- Funciona bem mesmo com listas grandes

Pontos fracos:
- Pior caso é $O(n^2)$ (quando o pivô é mal escolhido)
- Não é estável
- Recursivo: pode estourar pilha se mal implementado

Ideal para:
Ordenação geral de grandes volumes de dados, quando não precisa de estabilidade. É o queridinho de muitos sistemas por ser rápido e enxuto.



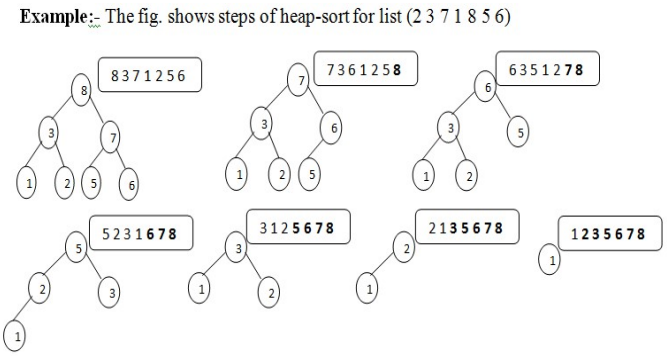
Heap Sort

Como funciona:
Transforma a lista em um heap máximo (uma árvore binária onde cada pai é maior que seus filhos).
Depois, vai extraindo o maior elemento (a raiz) e colocando no final da lista, reconstruindo o heap a cada passo.

Pontos fortes:
- Complexidade garantida: O(n log n) em todos os casos
- Não usa memória extra significativa (é feito in-place)
- Ótimo para situações onde o desempenho precisa ser estável e previsível
- Útil quando não pode haver surpresas no tempo de execução

Pontos fracos:
- Não é estável (pode alterar a ordem de elementos iguais)
- Mais complexo de implementar que os algoritmos básicos
- Pode ser mais lento que o Quick Sort na prática, apesar da mesma complexidade

Ideal para:
Sistemas que precisam de desempenho consistente e que não podem gastar memória com listas auxiliares. É usado em compiladores, engines de jogos, sistemas embarcados.



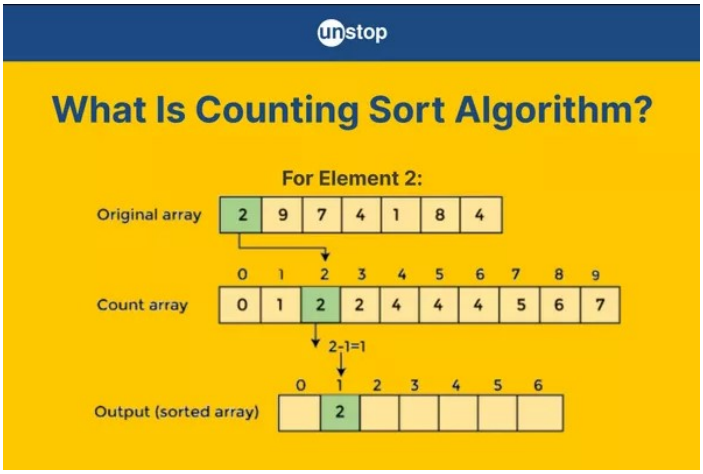
Counting Sort

Como funciona:
Conta quantas vezes cada valor aparece na lista e reconstrói a lista ordenada com base nessa contagem.
Não compara elementos entre si.

Pontos fortes:
- Complexidade O(n + k), onde k é o maior valor da lista — super eficiente quando k não é muito maior que n
- Muito rápido para listas com valores inteiros pequenos e não negativos
- Algoritmo estável

Pontos fracos:
- Só funciona com números inteiros não negativos (precisa de adaptação para negativos)
- Pode desperdiçar muita memória se o intervalo de valores for muito grande
- Pouco flexível: não funciona com números decimais ou tipos complexos

Ideal para:
Listas inteiras pequenas, como dados de frequência, idades, notas de provas, etc. Ótimo em situações onde se sabe de antemão o intervalo de valores.



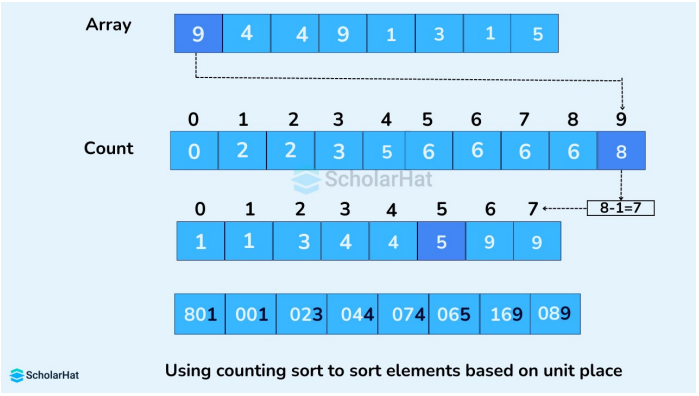
Radix Sort

Como funciona:
Ordena os números dígito por dígito, começando pelo dígito menos significativo (unidade).
Em cada passo, usa um algoritmo estável, geralmente o Counting Sort, para ordenar com base no dígito atual.

Pontos fortes:
- Complexidade O(d × (n + k)), onde d é o número de dígitos
- Extremamente rápido para listas de inteiros com poucos dígitos
- Estável
- Não faz comparações diretas entre os valores

Pontos fracos:
- Só funciona bem com números inteiros não negativos
- Pode consumir memória extra
- Não é ideal para números com muitos dígitos ou distribuições muito desiguais

Ideal para:
Grandes volumes de inteiros não negativos com dígitos limitados (ex: CPF, matrícula, código de produto). Também muito usado em hardware e sistemas de banco de dados.



Bucket Sort

Como funciona:
Distribui os elementos em “baldes” (listas) com base em uma função de faixa.
Depois ordena cada balde individualmente (geralmente com Insertion Sort) e junta tudo.

Pontos fortes:
- Pode atingir quase O(n) se os dados forem uniformemente distribuídos
- Excelente desempenho em conjuntos de dados distribuídos de forma contínua
- Permite ordenações paralelas (cada bucket pode ser tratado separadamente)

Pontos fracos:
- Depende muito da distribuição dos dados — pode virar $O(n^2)$ se tudo cair num balde só
- Precisa de uma boa função de distribuição e número adequado de buckets
- Não funciona bem com dados muito concentrados ou altamente desbalanceados

Ideal para:
Listas de números reais ou decimais entre 0 e 1, ou quando os dados são uniformemente distribuídos. Muito usado em gráficos, jogos, simulações, processamento de imagens.