Pesquisa em Projeto de Algoritmos

Gerencie o tempo
e tome decisões
de forma eficaz.

Dividir para conquistar

"Dividir para conquistar (“Divide et impera” ou “Divide et Vinces”) é um clássico nas estratégias de guerra para enfraquecer e subjugar os povos. O termo, embora já era conhecida na Antiguidade, foi cunhado por Júlio César em seu livro ”De Bello Gallico” (Guerra das Gálias). Era essencialmente uma política de “dividir” seus inimigos, aliar com tribos individuais durante suas disputas com adversários locais.

Basicamente, os elementos dessa técnica envolvem:

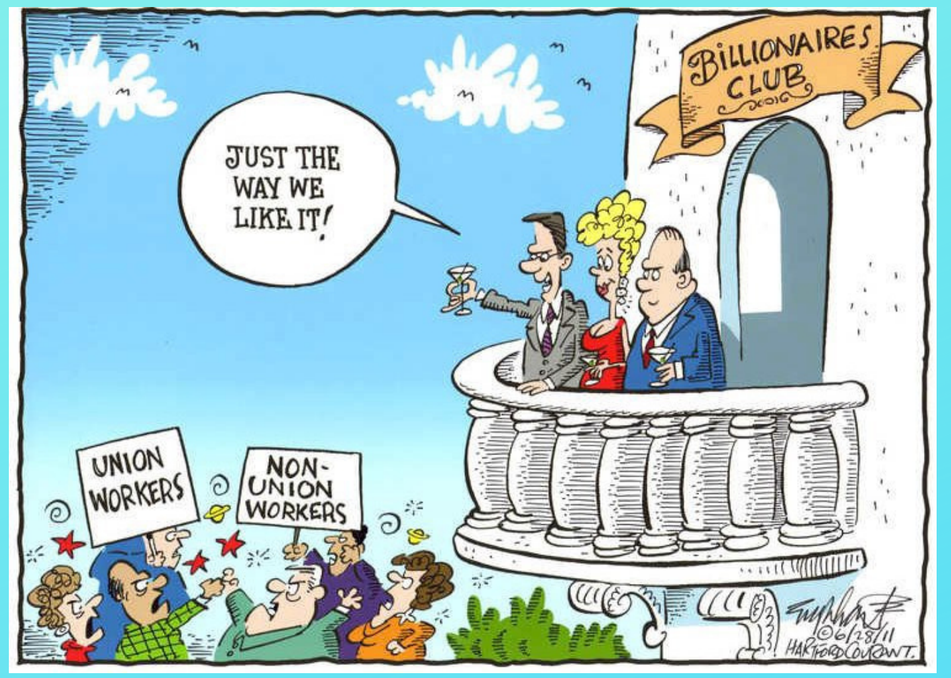
-Criar ou estimular divisões entre os indivíduos com o objetivo de evitar alianças que poderiam desafiar o soberano;

-Auxiliar, promover e dar poder político para aqueles que estão dispostos a cooperar com o soberano;

-Fomentar a inimizade e desconfiança entre os governantes locais;

-Incentivar gastos sem sentido que reduzam a capacidade de gastos políticos e militares.

Dividir para Conquistar



O projeto de muitos algoritmos eficientes é baseado no método da divisão e conquista. Esse método (ou estratégia de projeto de algoritmos) consiste no seguinte:

1. **a instância dada do problema é dividida em duas ou mais instâncias menores,**
2. **cada instância menor é resolvida usando o próprio algoritmo que está sendo definido,**
3. **as soluções das instâncias menores são combinadas para produzir uma solução da instância original.**

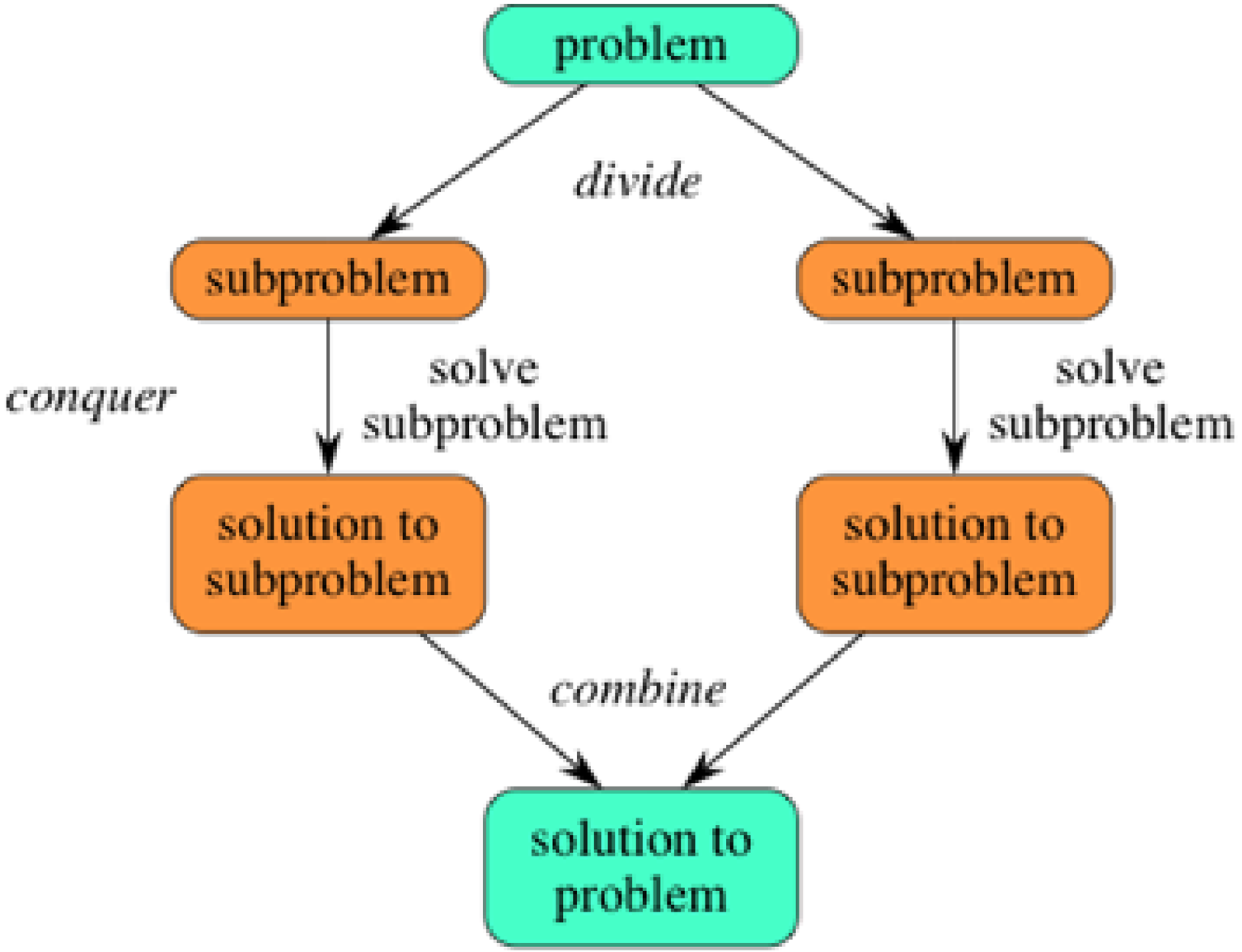
A segunda fase é implementada por uma chamada recursiva. Essa é a fase da conquista.

Problema: Ordenar um vetor de n posições através do MergeSort.

Algoritmo Merge Sort

A idéia básica do Merge Sort é criar uma sequência ordenada a partir de duas outras também ordenadas. Para isso, o algoritmo Merge Sort divide a sequência original em pares de dados, agrupa estes pares na ordem desejada; depois as agrupa as sequências de pares já ordenados, formando uma nova sequência ordenada de quatro elementos, e assim por diante, até ter toda a sequência ordenada.

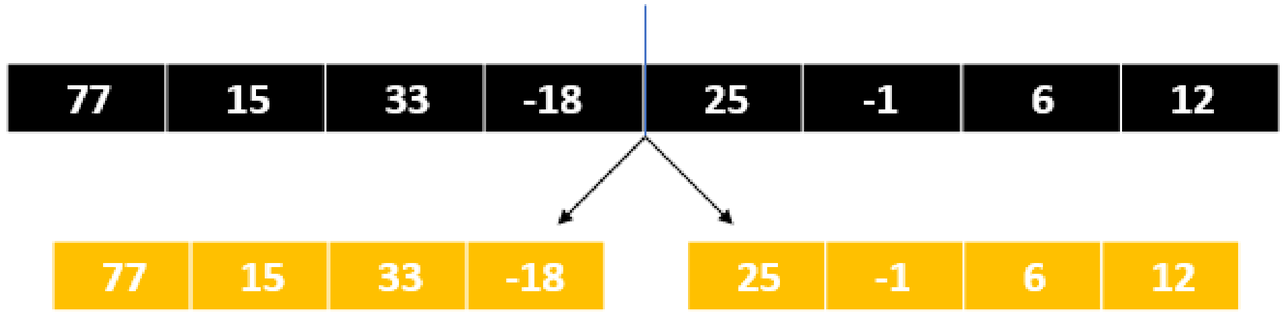
A desvantagem do Merge Sort é que requer o dobro de memória, ou seja, precisa de um vetor com as mesmas dimensões do vetor que está sendo classificado.


problem
divide
subproblem
subproblem
conquer
solve
subproblem
solve
subproblem
solution to
subproblem
solution to
subproblem
combine
solution to
problem

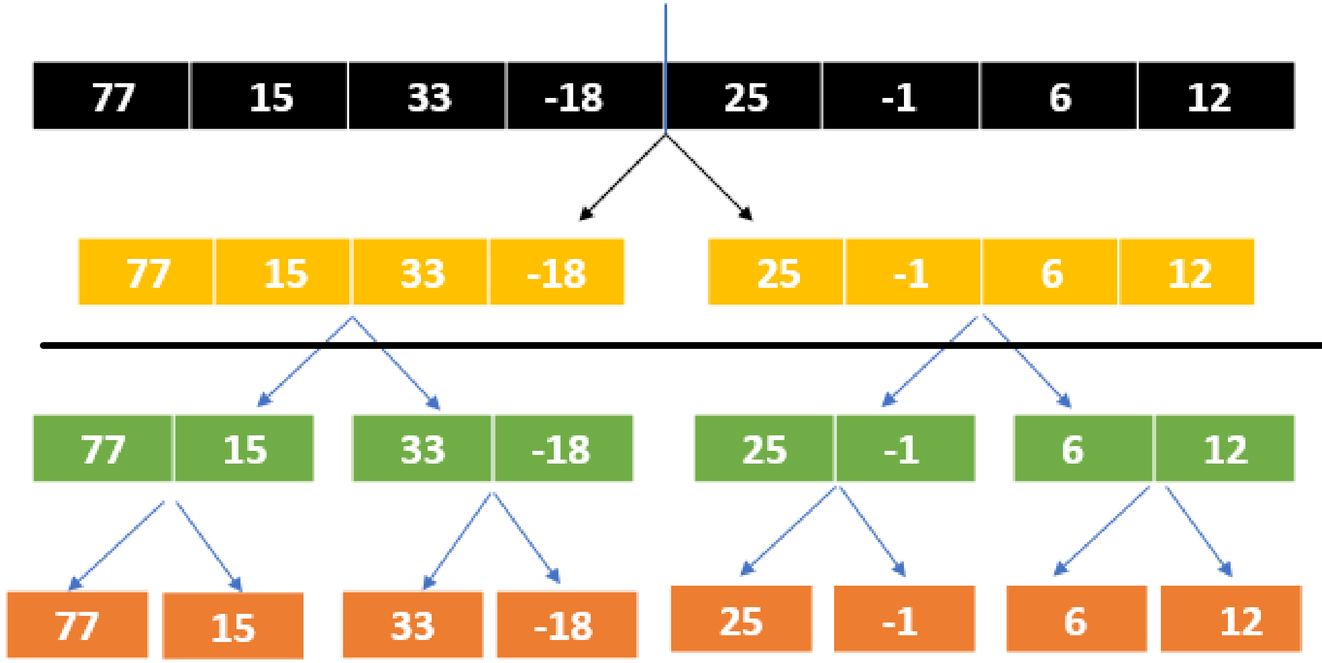

problem
divide
subproblem
subproblem
divide
divide
subproblem
subproblem
subproblem
subproblem
divide
divide
divide
divide
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
subproblem
conquer
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solve subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
combine
combine
solution to subproblem
solution to subproblem
solution to subproblem
solution to subproblem
combine
combine
solution to subproblem
solution to subproblem
combine
solution to problem

Dividir:

Supondo que temos um vetor[8] = { 77, 15, 33, -18, 25, -1, 6, 12 }

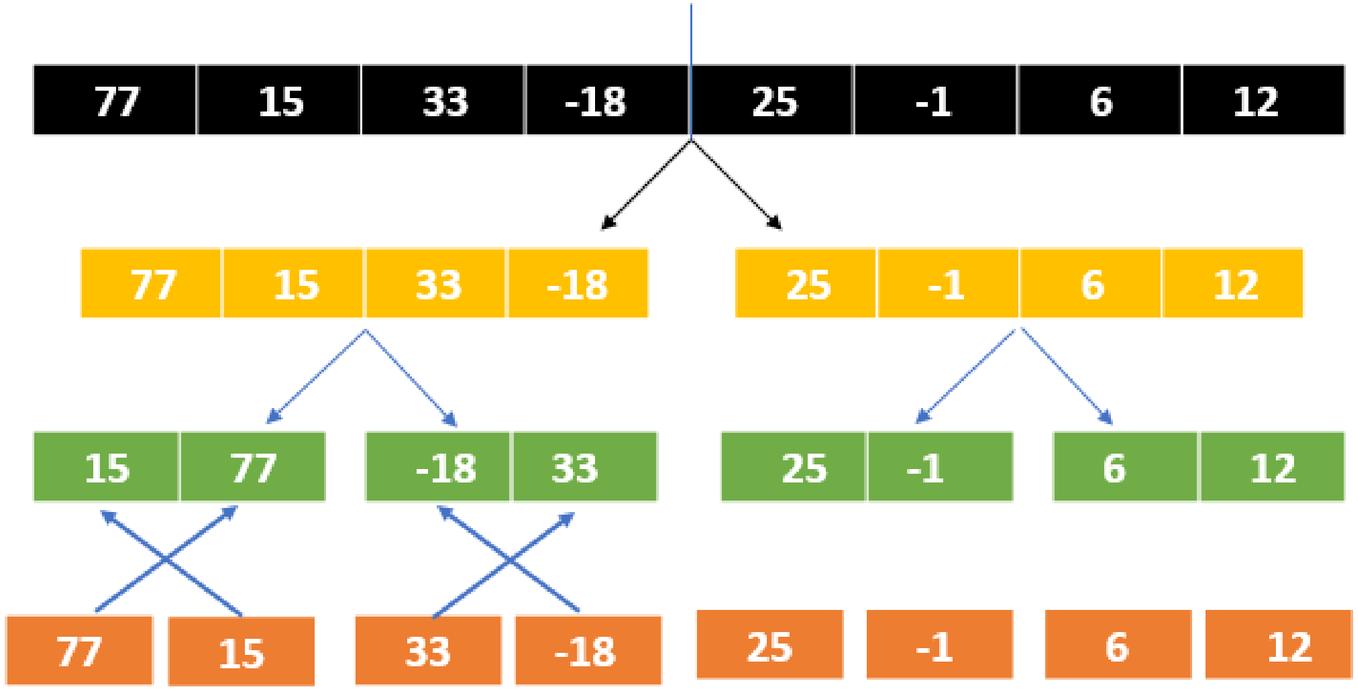



```
void sort(int *vetor, int *aux, int i, int f)
{
    if (i >= f)
        return;

    int m = (i + f) / 2;

    sort(vetor, aux, i, m);
    sort(vetor, aux, m + 1, f);

    if (vetor[m] <= vetor[m + 1])
        return;

    merge(vetor, aux, i, m, f);
}

void mergesort(int *vetor, int n)
{
    int *aux = (int*) malloc(sizeof(int) * n);
    sort(vetor, aux, 0, n - 1);
    free(aux);
}
```

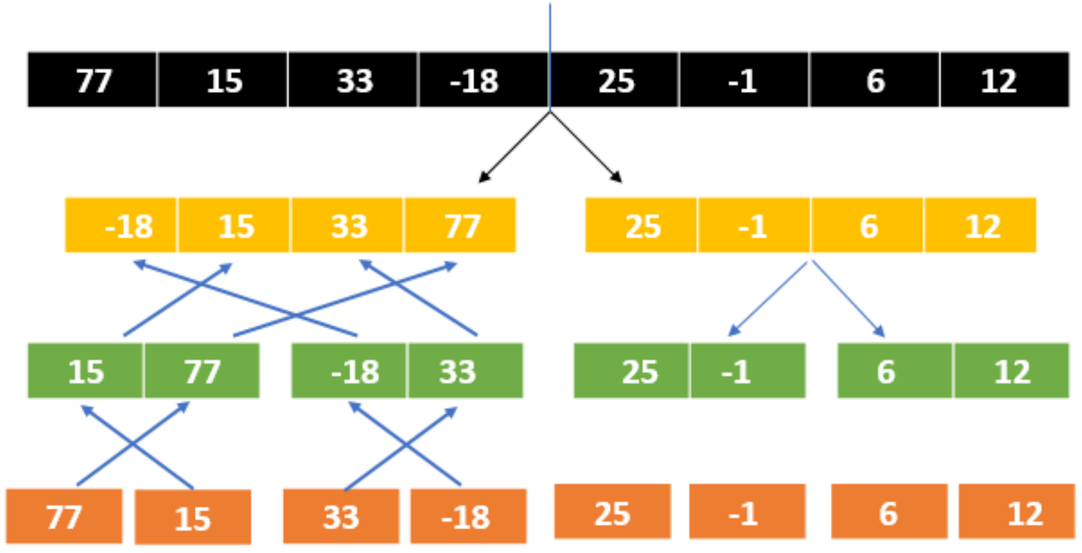


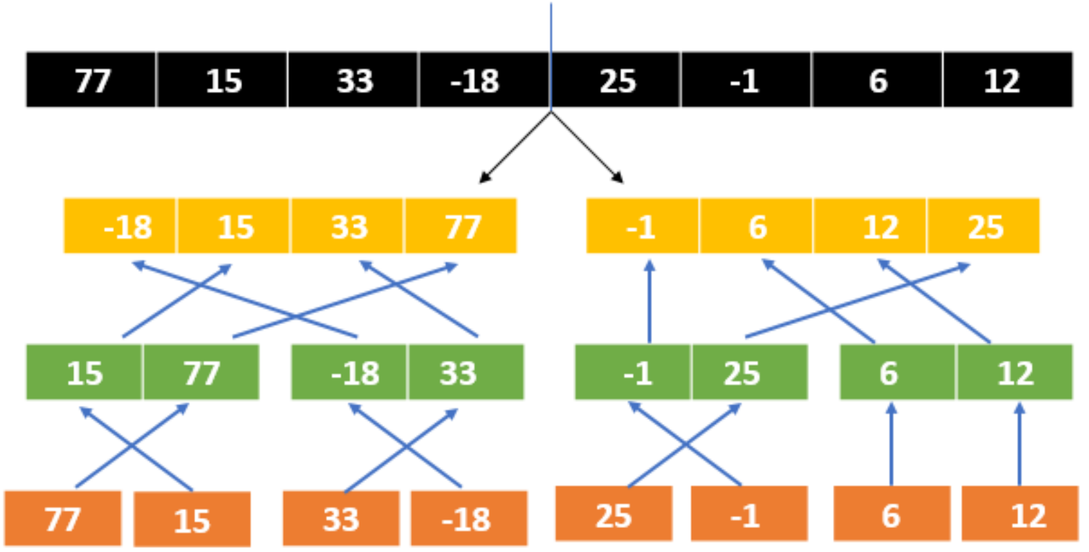

```c
void sort(int *vetor, int *aux, int i, int f)
{
    if (i >= f)
        return;

    int m = (i + f) / 2;

    sort(vetor, aux, i, m);
    sort(vetor, aux, m + 1, f);

    if (vetor[m] <= vetor[m + 1])
        return;

    merge(vetor, aux, i, m, f);
}

void mergesort(int *vetor, int n)
{
    int *aux = (int*) malloc(sizeof(int) * n);
    sort(vetor, aux, 0, n - 1);
    free(aux);
}
```

Conquistar:

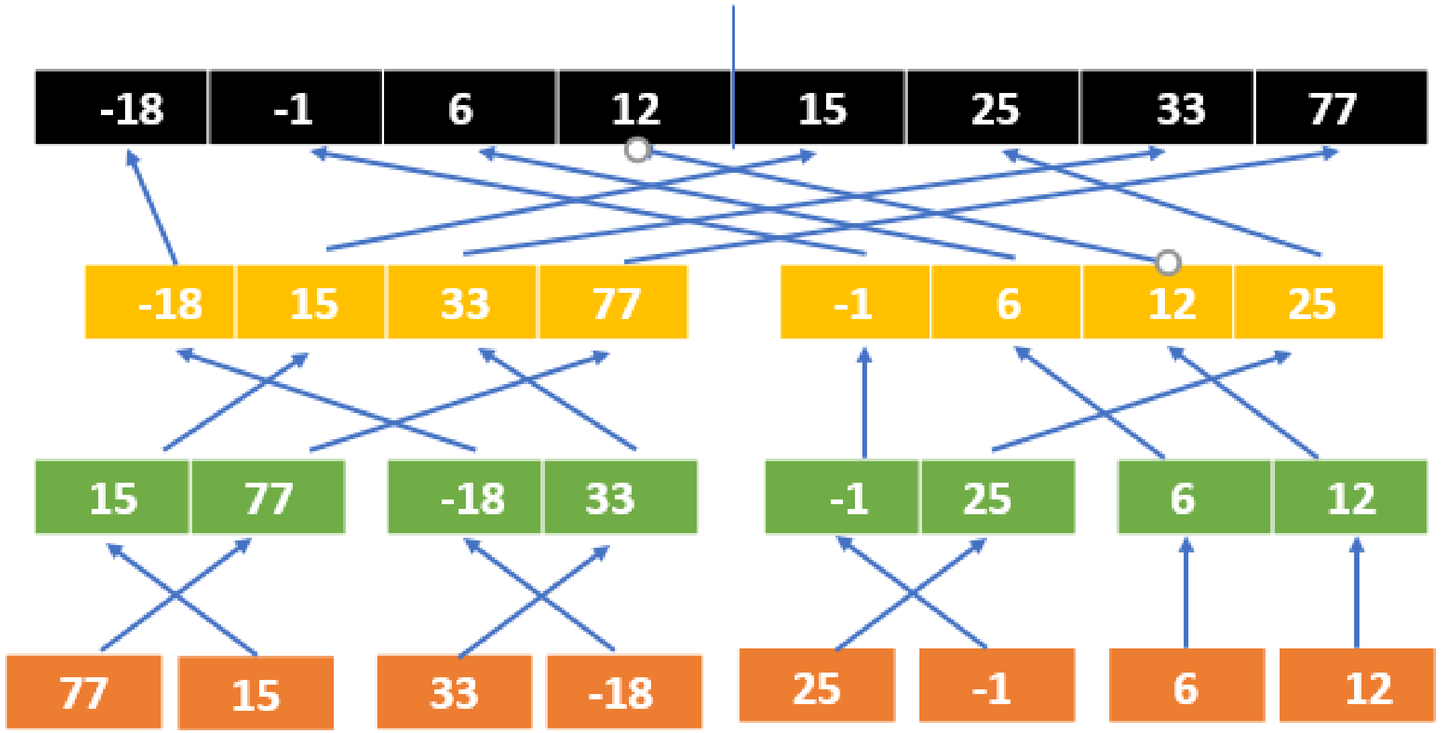



```
void merge(int *vetor, int *aux, int i, int m, int f)
{
   int z,
      ndxEsq = i, ndxDir = m + 1;

   for (z = i; z <= f; z++)
      aux[z] = vetor[z];

   z = i;

   while (ndxEsq <= m && ndxDir <= f)
   {
      if (aux[ndxEsq] <= aux[ndxDir])
         vetor[z++] = aux[ndxEsq++];
      else
         vetor[z++] = aux[ndxDir++];
   }

   while (ndxEsq <= m)
      vetor[z++] = aux[ndxEsq++];

   while (ndxDir <= f)
      vetor[z++] = aux[ndxDir++];
}
```


77 15 33 -18 25 -1 6 12
-18 15 33 77
25 -1 6 12
15 77
-18 33
25 -1
6 12
77
15
33
-18
25
-1
6
12


77 15 33 -18 25 -1 6 12
-18 15 33 77
-1 6 12 25
15 77
-18 33
-1 25
6 12
77
15
33
-18
25
-1
6
12

Conquistado!




```
void merge(int *vetor, int *aux, int i, int m, int f)
{
   int z,
      ndxEsq = i, ndxDir = m + 1;

   for (z = i; z <= f; z++)
      aux[z] = vetor[z];

   z = i;

   while (ndxEsq <= m && ndxDir <= f)
   {
      if (aux[ndxEsq] <= aux[ndxDir])
         vetor[z++] = aux[ndxEsq++];
      else
         vetor[z++] = aux[ndxDir++];
   }

   while (ndxEsq <= m)
      vetor[z++] = aux[ndxEsq++];

   while (ndxDir <= f)
      vetor[z++] = aux[ndxDir++];
}
```

Desempenho:

Algoritmo	*Bubble Sort*	*Insertion Sort*	*Selection Sort*	***Merge Sort***	*Quick Sort*
Pior caso	O (n^2)	O (n^2)	O (n^2)	**O (n log n)**	O (n^2)
Melhor caso	O (n)	O (n)	O (n^2)	**O (n log n)**	O (n log n)
Caso médio	O (n^2)	O (n^2)	O (n^2)	**O (n log n)**	O (n log n)
Memória aux.	O (1)	O (1)	O (1)	**O (n)**	O (log n)