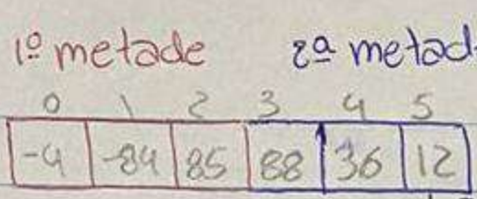
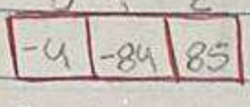
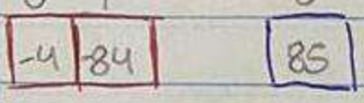
- A parte vermelha é feita 1º e a azul fica em standby

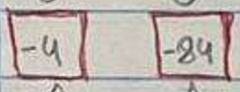
1)



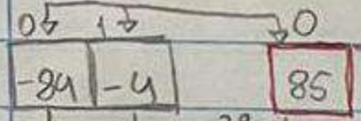
1ª etapa

I- 

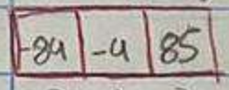
II- 

III- 

caso base
* início = fim em ambos
* compara-os e volta da recursão ordenando

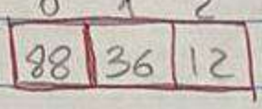
II- 

* início = fim
* compara com o outro array e volto ordenando

I- 

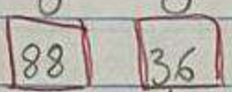
* o 1º array ficou vazio, adiciona o restante do outro

2ª etapa



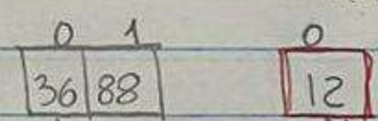
-I

-II



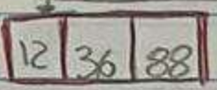
caso base
* início = fim em ambos
* compara-os e volta da recursão ordenando

-III



* início = fim
* compara com o outro array e volta ordenando.

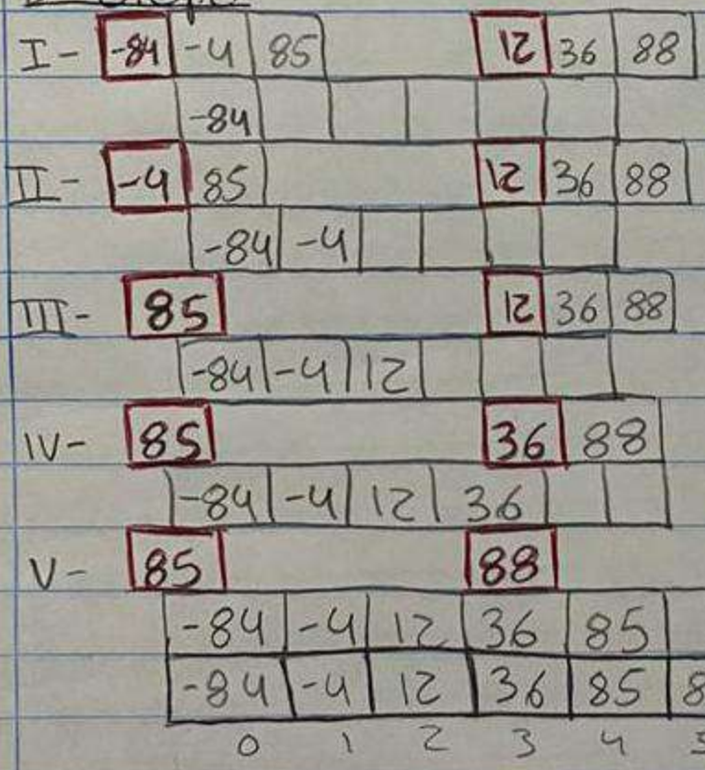
-II



* o 2º array fica vazio, adiciona o resto do outro

-I

3ª etapa

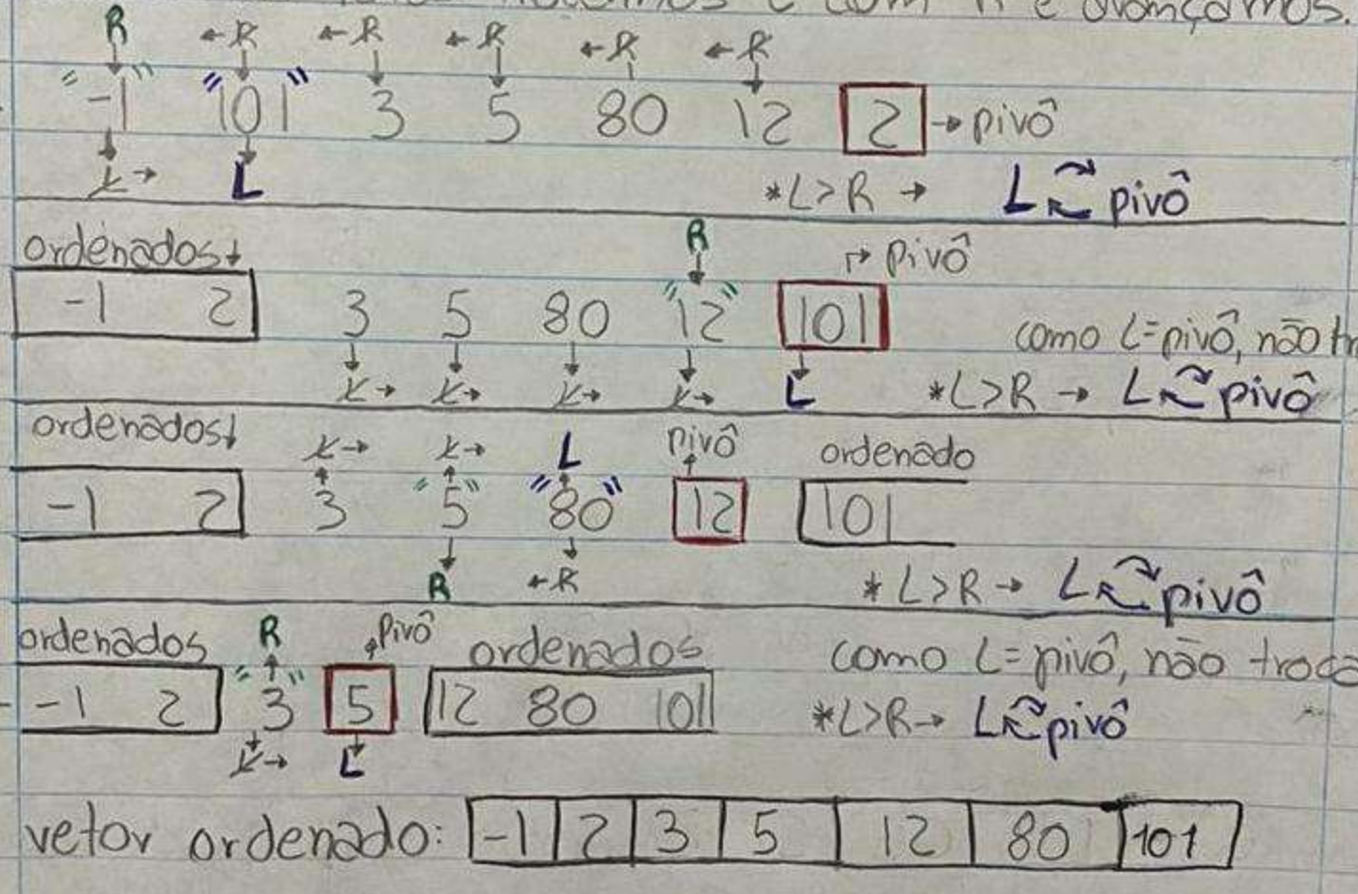


* Seguiremos comparando os destacados e adicionaremos o menor voltando na recursão

* Após a última comparação o 1º vetor ficará vazio e adicionamos o resto do 2º

1) O merge sort usa a abordagem dividir para conquistar seguindo as etapas: 1º) dividir o problema em sub-problemas; 2º) resolver recursivamente as novas instâncias; 3º) intercalar os resultados dos sub-problemas para solucionar o problema principal. A etapa onde comparamos os elementos para intercalá-los é da ordem de O(n) só que isso será feito "log n" vezes. Dando ao algoritmo a complexidade O(n log n) no melhor e no pior caso.

2) Usaremos o ponteiro "pivô" começando na direita. O ponteiro "L" (left) avançará da esquerda para direita até encontrar um elemento maior que o pivô. O "R" (right) irá da direita para esquerda até encontrar um menor que o pivô. Quando L >= R saímos do laço e trocamos quem está na posição "L" com quem está como pivô, pois quem estiver à esquerda de "L" é menor que o pivô e quem estiver à direita maior. Caso a condição citada for falsa trocamos "L" com "R" e avançamos.



vetor ordenado: | -1 | 2 | 3 | 5 | 12 | 80 | 101 |

Com o auxílio do pivô ao "dividir" o vetor já ordenamos parcialmente os dados e isso o torna bastante eficiente pois evitará comparações desnecessárias, tendo uma complexidade $O(n \log n)$ no melhor caso. Porém, numa sequência ordenada inversamente haverá o pior caso, pois o pivô será comparado com os demais e o "left" e o "right" trocarão de lugar todas as vezes tendo complexidade $O(n^2)$, porém trata-se de um cenário difícil de ocorrer.

3) a) A abordagem recursiva é geralmente mais lenta e usa mais memória do que a iterativa pelo fato de que o método faz muitas chamadas a si. Porém, essa abordagem pode simplificar problemas em que a versão iterativa seria mais complexa.

A recursividade é bem vinda em algoritmos de busca visto que ao dividir o problema em problemas menores, obtendo assim complexidade O(log n) na busca binária e em contrapartida na busca sequencial, que é iterativa, por ter que percorrer o vetor acaba tendo complexidade O(n). Obs: Apesar da buscar binária ser naturalmente recursiva, é possivel obter a mesma complexidade com a busca binária iterativa, que seguirá a mesma lógica do dividir para conquistar.

A recursividade traz vantagens em algoritmos de ordenação também. Métodos como bubbe sort, selection sort e insertion sort, que não são recursivos acabam sendo de ordem quadrática; enquanto isso os métodos recursivos Quicksort e Merge Sort, por usar a abordagem dividir para conquistar têm uma complexidade O(n log n) o que é mais eficiente em tempo de execução. Obs: é importante analisar se existem saídas para quando for trabalhar com vetores muito grandes, pois o alto número de chamadas dos métodos recursivos pode causar estouro de pilha, trazendo um grande consumo de memória.

3) b) Ambos os algoritmos são recursivos e utilizam a abordagem dividir para conquistar que segue a lógica "se tenho um problema grande, quebro ele em pedaços menores. Resolvendo-os e unindo-os resolvo o maior". No Merge sort se "divide" o vetor recursivamente e a 2ª metade é congelada enquanto a 1ª é resolvida. O resultado dos subproblemas é intercalado resolvendo o principal. Já o Quicksort, com o auxilio de um ponteiro(pivô), ao fazer a divisão ordena parcialmente os dados e isso o torna mais eficiente que o Merge sort na maioria dos casos, pois evitará mais comparações. Porém, para uma sequencia inversamente ordenada o Quicksort demora mais. Esse é o pior caso do Quichsort, pois o pivô será comparado com todos os outros e o "left" e "right" trocarão de lugar todas as vezes. Apesar do Merge sort ter complexidade $O(n \log n)$ todas as vezes e o Quicksort $O(n \log n)$ no melhor e $O(n^2)$ no pior, na maioria das vezes o segundo é preferível pois uma sequencia inversamente ordenada é difícil de ocorrer e em geral se faz menos comparações com ele.

3)c) ArrayList:

*Pró: Acesso a qualquer elemento através do indice, fazendo com que o acesso tenha tempo constante "O(1)". Então é interessante usa-lo quando estiver trabalhando com os indices (como no método get).

*Contra: Por ter alocação estática ao criá-lo ele começa com um tamanho fixo, que pode crescer se preciso, mas demandaria um custo computacional por precisar criar uma cópia para um novo array maior. Porém seu ponto fraco é que, por ser sequencial, no método remove haveria casos em que seria preciso remanejar os demais elementos, tendo complexidade linear "O(n)".

LinkedList:

*Pró: Por ter elementos organizados de forma dispersa, não tem tamanho fixo. Outra vantagem é em inserir ou remover no início da lista: só precisa mudar a referencia dos ponteiros em vez de remanejar elementos. Complexidade O(1) nesses métodos.

*Contra: Precisa percorrer a lista para acessar os elementos, tendo complexidade O(n) nesses casos.

Ou seja, seria mais interessante o LinkedList em casos com muitas adições e remoções (principalmente no início da lista). E, apesar disso, o ArrayList é mais usado pois acessar os elementos pelo indice o faz ganhar em tempo de execução na maioria dos casos.