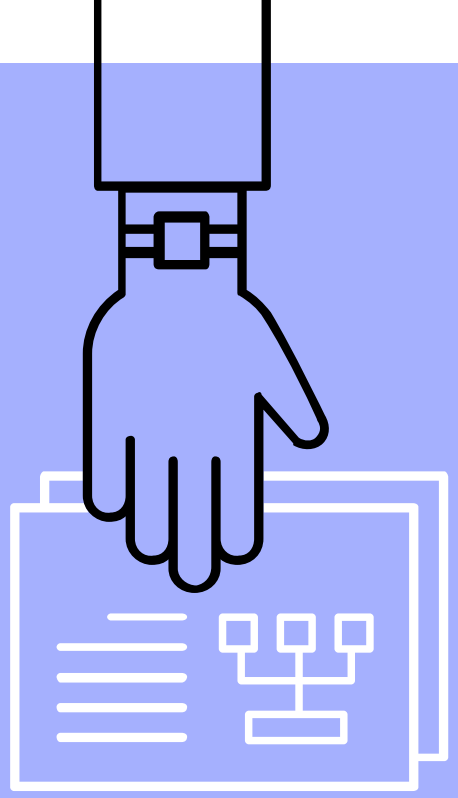
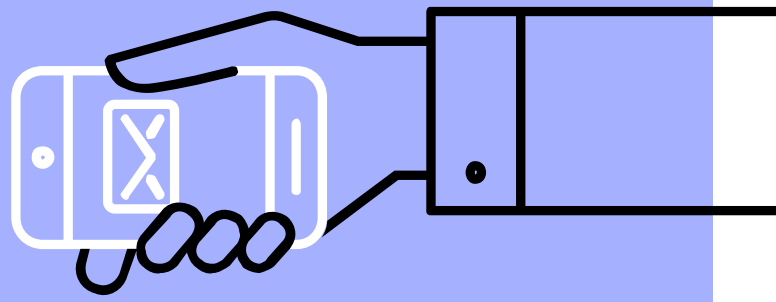
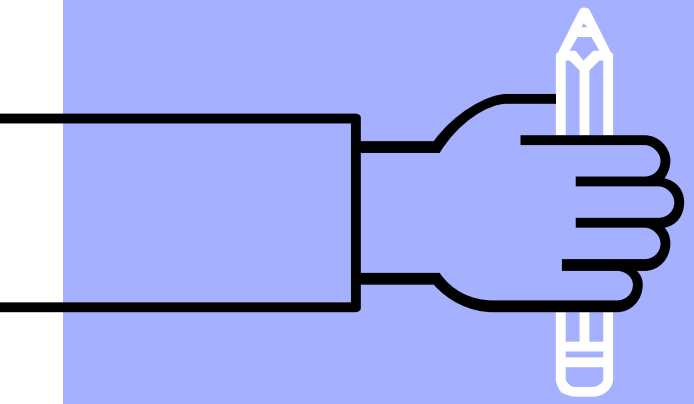
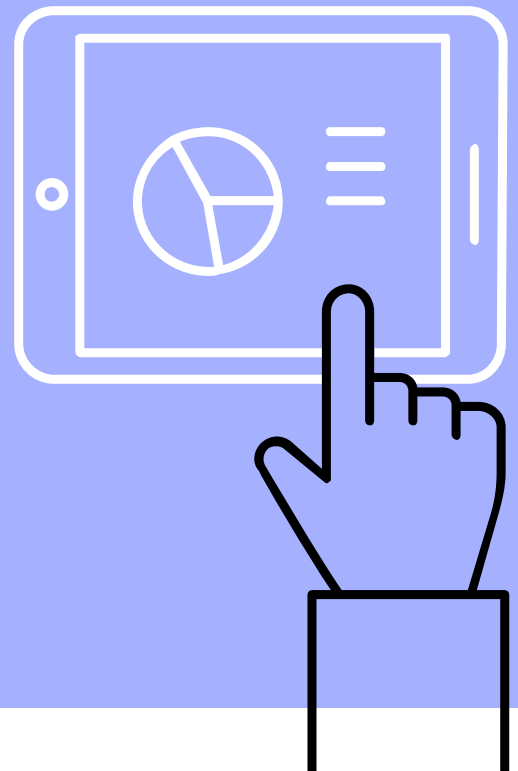




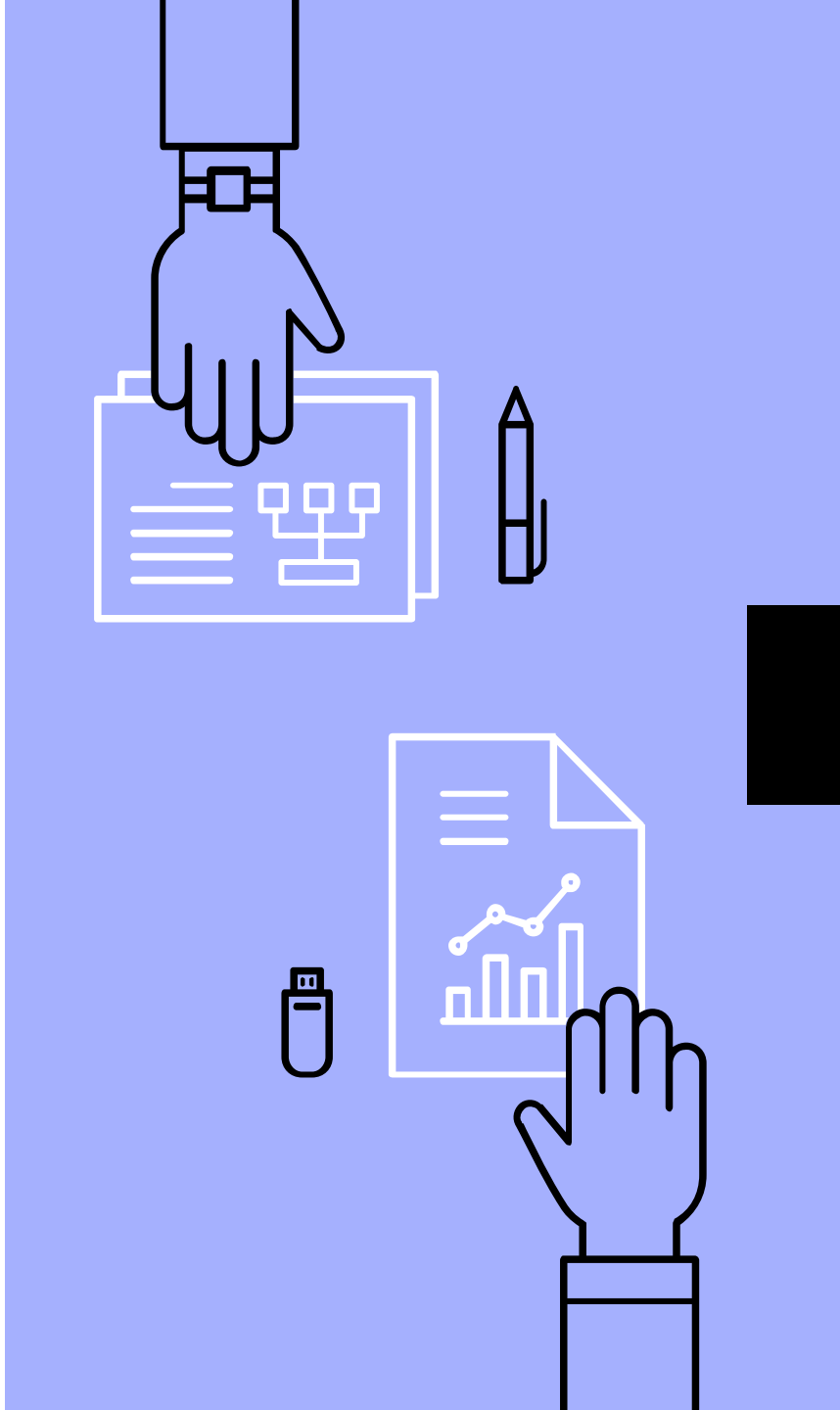
Classificação e Pesquisa





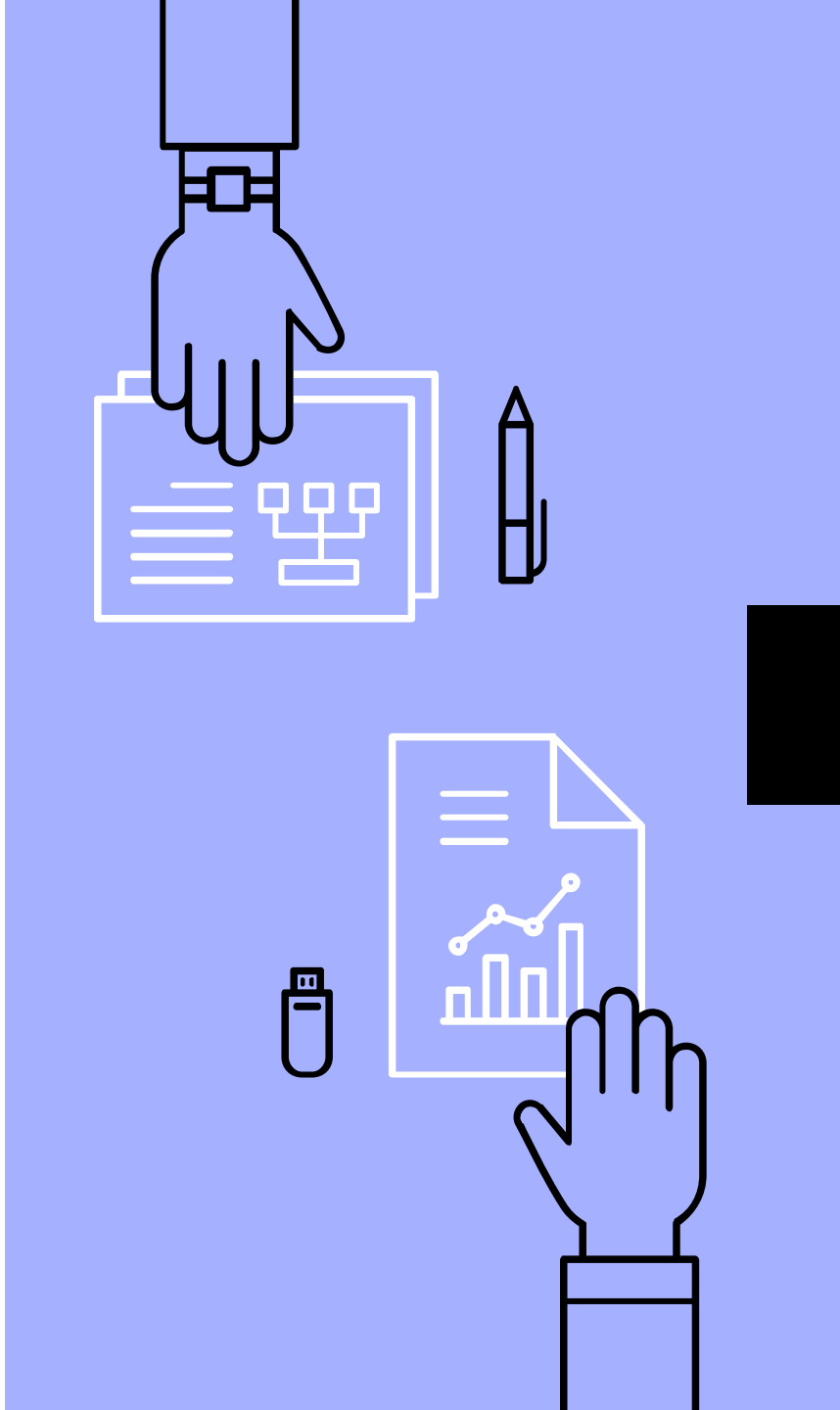
Objetivos

- Ser capaz de explicar e implementar a pesquisa sequencial e a pesquisa binária.
- Ser capaz de explicar e implementar classificação por seleção, classificação por bolha, classificação por mesclagem, classificação rápida, classificação por inserção e classificação por shell.
- Compreender a ideia de hashing como técnica de pesquisa.
- Para inserir o tipo de dados abstratos do mapa.
- Para implementar o tipo de dados abstratos do mapa usando hashing.



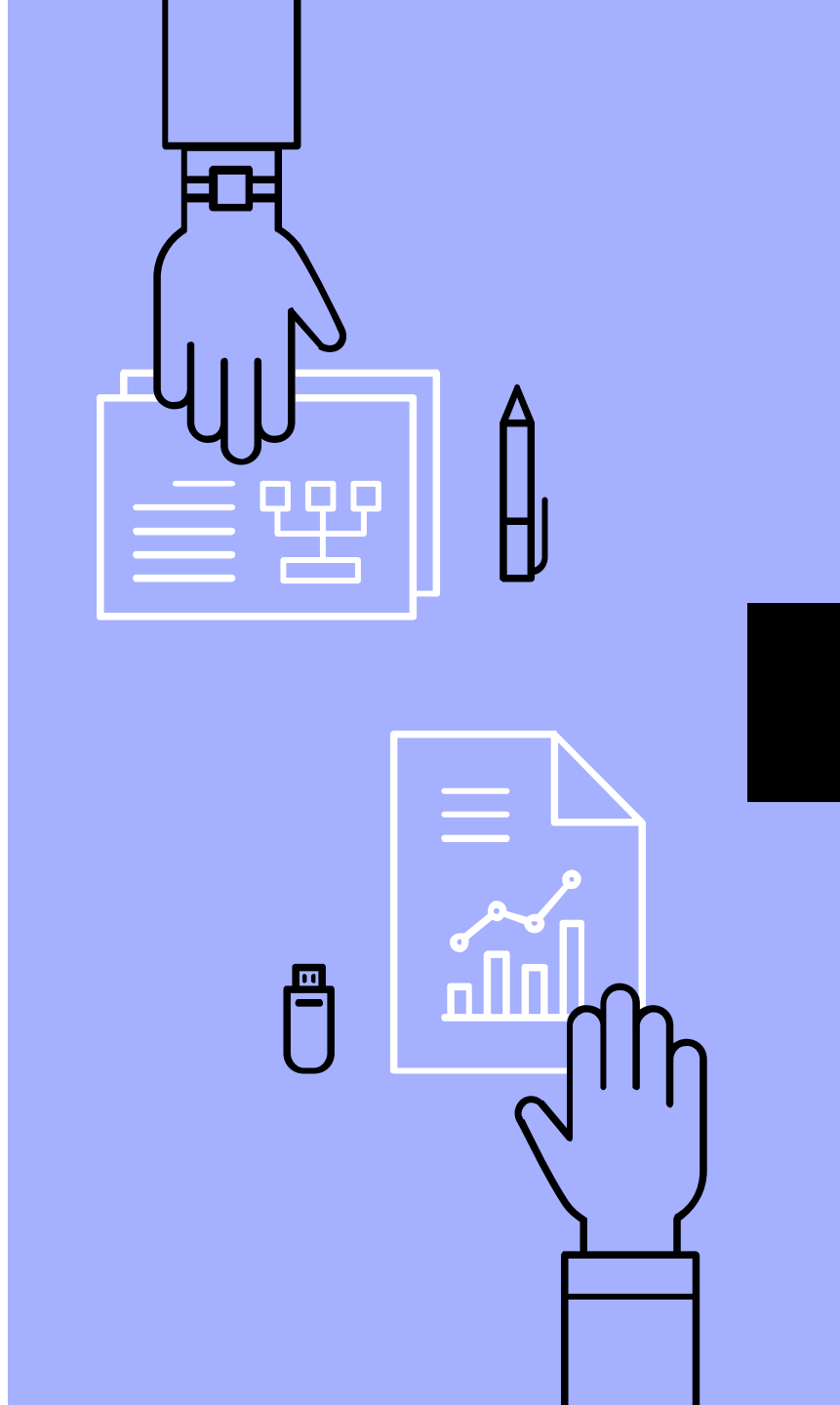
Searching

- Agora voltaremos nossa atenção para alguns dos problemas mais comuns que surgem na computação, os de busca e classificação.
- Pesquisar é o processo algorítmico de localizar um item específico em uma coleção de itens. Uma pesquisa normalmente responde como Verdadeiro ou Falso para saber se o item está presente. Ocasionalmente, ele pode ser modificado para retornar ao local onde o item foi encontrado.
- Em Python, há uma maneira muito fácil de perguntar se um item está em uma lista de itens.



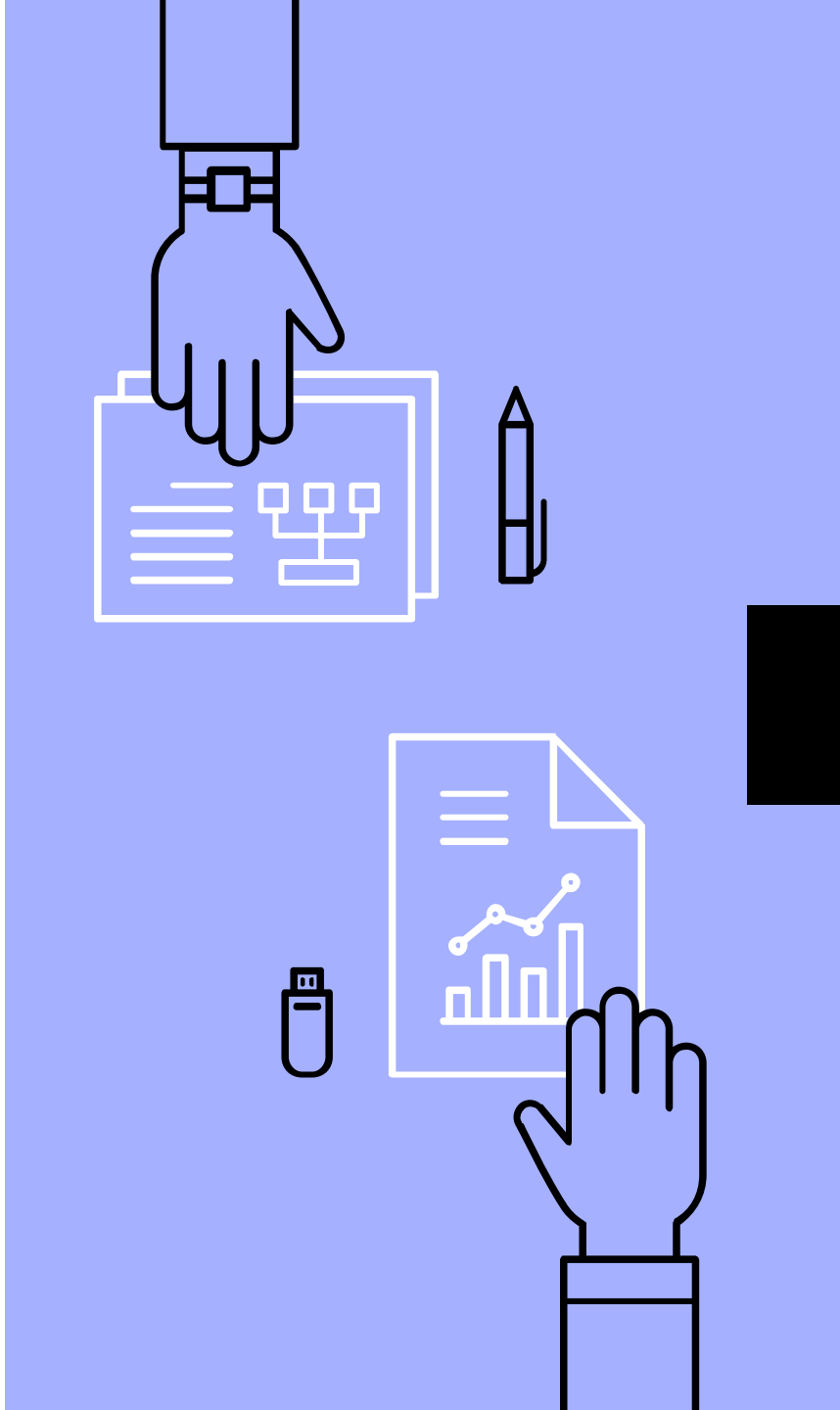
```
>>> 15 in [3,5,2,4,1]
False
>>> 3 in [3,5,2,4,1]
True
>>>
```

- Mesmo que seja fácil de escrever, um processo subjacente deve ser executado para responder à pergunta.
- Acontece que existem muitas maneiras diferentes de pesquisar o item. O que nos interessa aqui é como esses algoritmos funcionam e como eles se comparam.

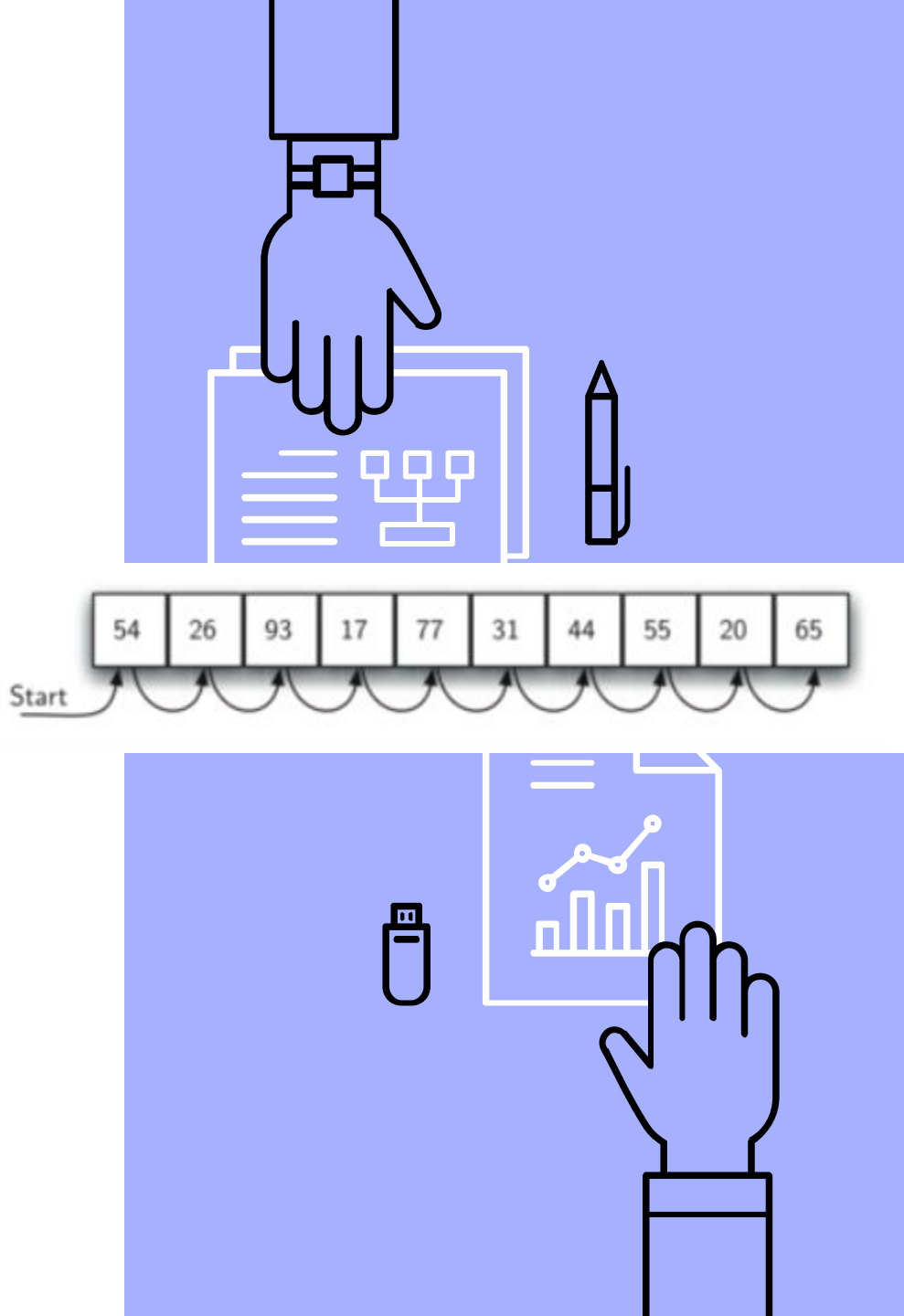


Pesquisa Sequencial

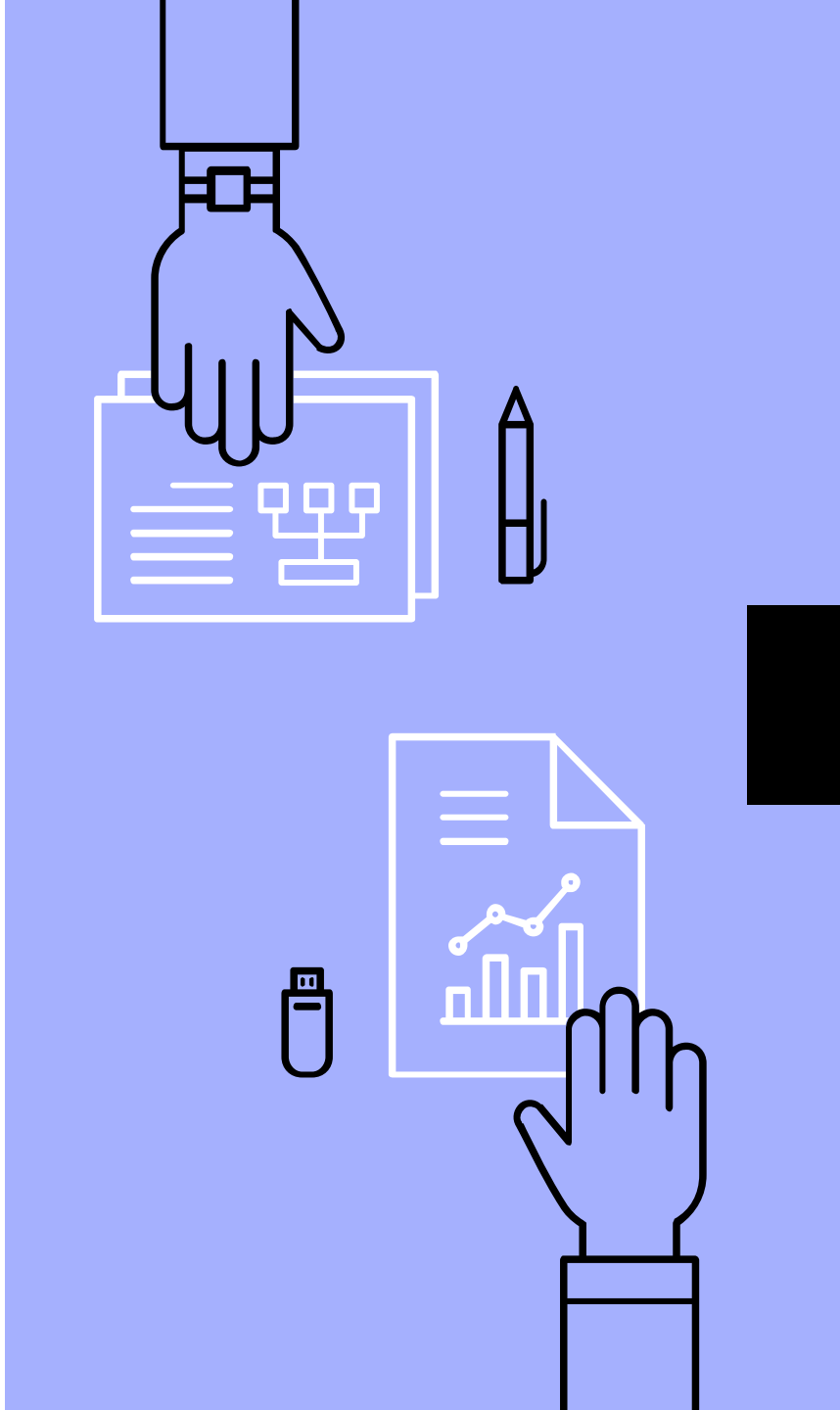
- Quando os itens de dados são armazenados em uma coleção, como uma lista, dizemos que eles têm uma relação linear ou sequencial. Cada item de dados é armazenado em uma posição em relação aos outros.
- Nas listas Python, essas posições relativas são os valores de índice dos itens individuais. Como esses valores de índice são ordenados, é possível visitá-los em sequência. Este processo dá origem à nossa primeira técnica de busca, a busca sequencial.



- A figura mostra como funciona essa pesquisa. Começando no primeiro item da lista, simplesmente passamos de um item para outro, seguindo a ordem sequencial subjacente até encontrarmos o que estamos procurando ou ficarem sem itens.
- Se ficarmos sem itens, descobrimos que o item que procurávamos não estava presente.
- A implementação do Python para este algoritmo mais a frente.
- A função precisa da lista e do item que estamos procurando e retorna um valor booleano para saber se ele está presente. A variável booleana encontrada é inicializada como False e recebe o valor True se descobrirmos o item na lista.

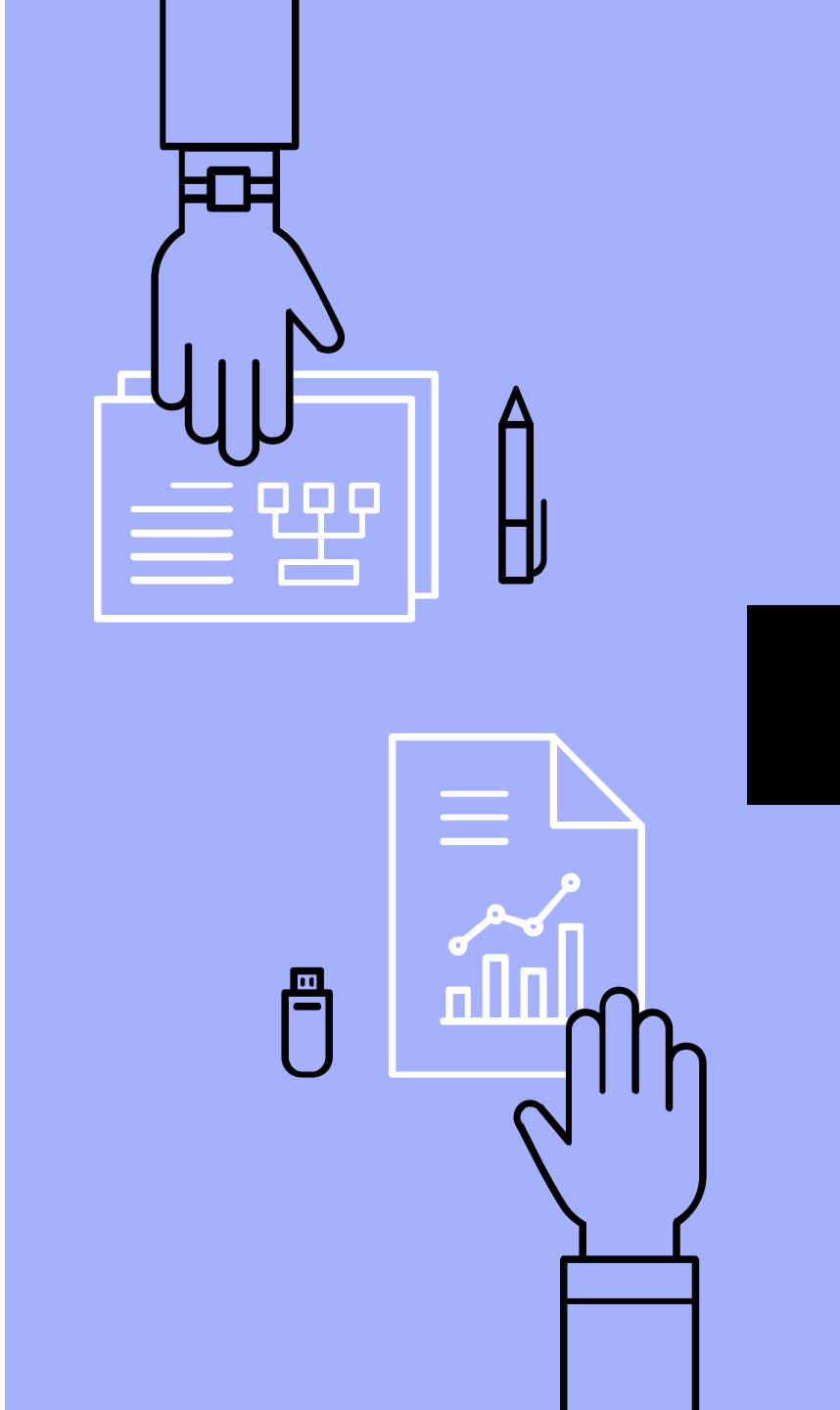


```python
def sequential_search(a_list, item):
    pos = 0
    found = False
    while pos < len(a_list) and not found:
        if a_list[pos] == item:
            found = True
        else:
            pos = pos + 1

    return found


test_list = [1, 2, 32, 8, 17, 19, 42, 13, 0]
print(sequential_search(test_list, 3))
print(sequential_search(test_list, 13))
```

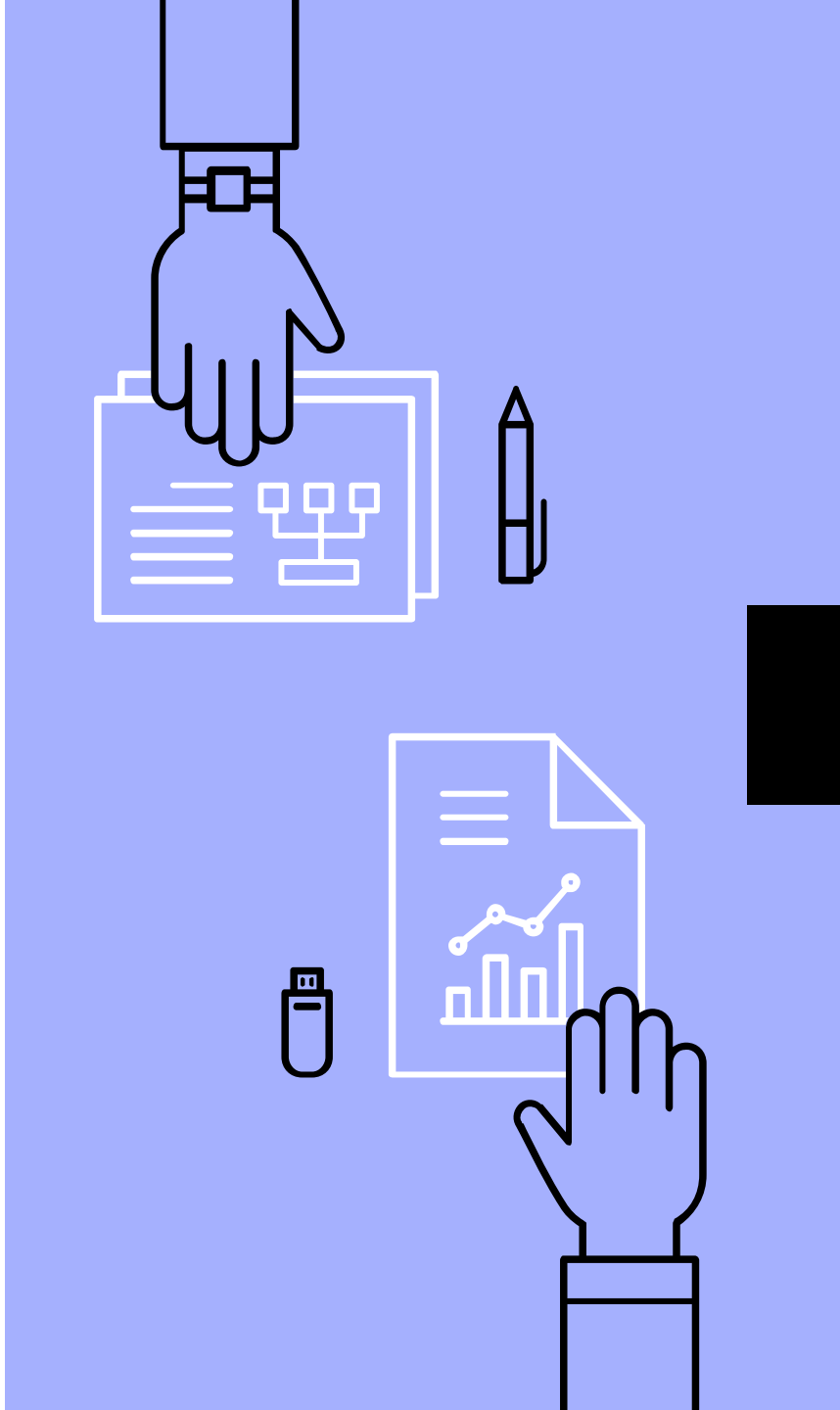


Análise de busca sequencial

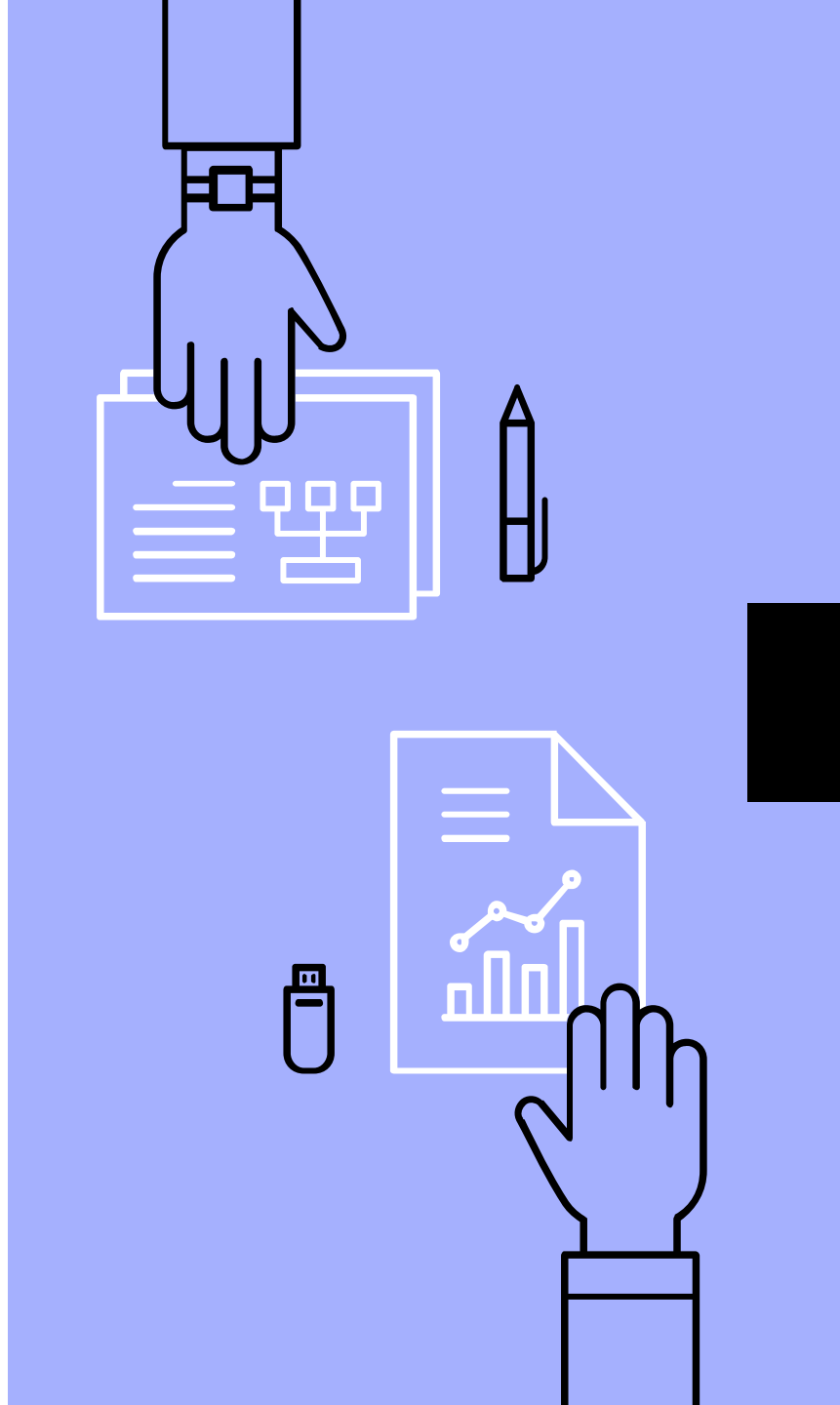
- Para analisar algoritmos de busca, precisamos decidir sobre uma unidade básica de computação. Lembre-se de que essa é normalmente a etapa comum que deve ser repetida para resolver o problema.
- Para pesquisar, faz sentido contar o número de comparações realizadas. Cada comparação pode ou não descobrir o item que procuramos.
- Além disso, fazemos outra suposição aqui. A lista de itens não é ordenada de forma alguma. Os itens foram colocados aleatoriamente na lista.
- Em outras palavras, a probabilidade de que o item que procuramos esteja em qualquer posição específica é exatamente a mesma para cada posição da lista.



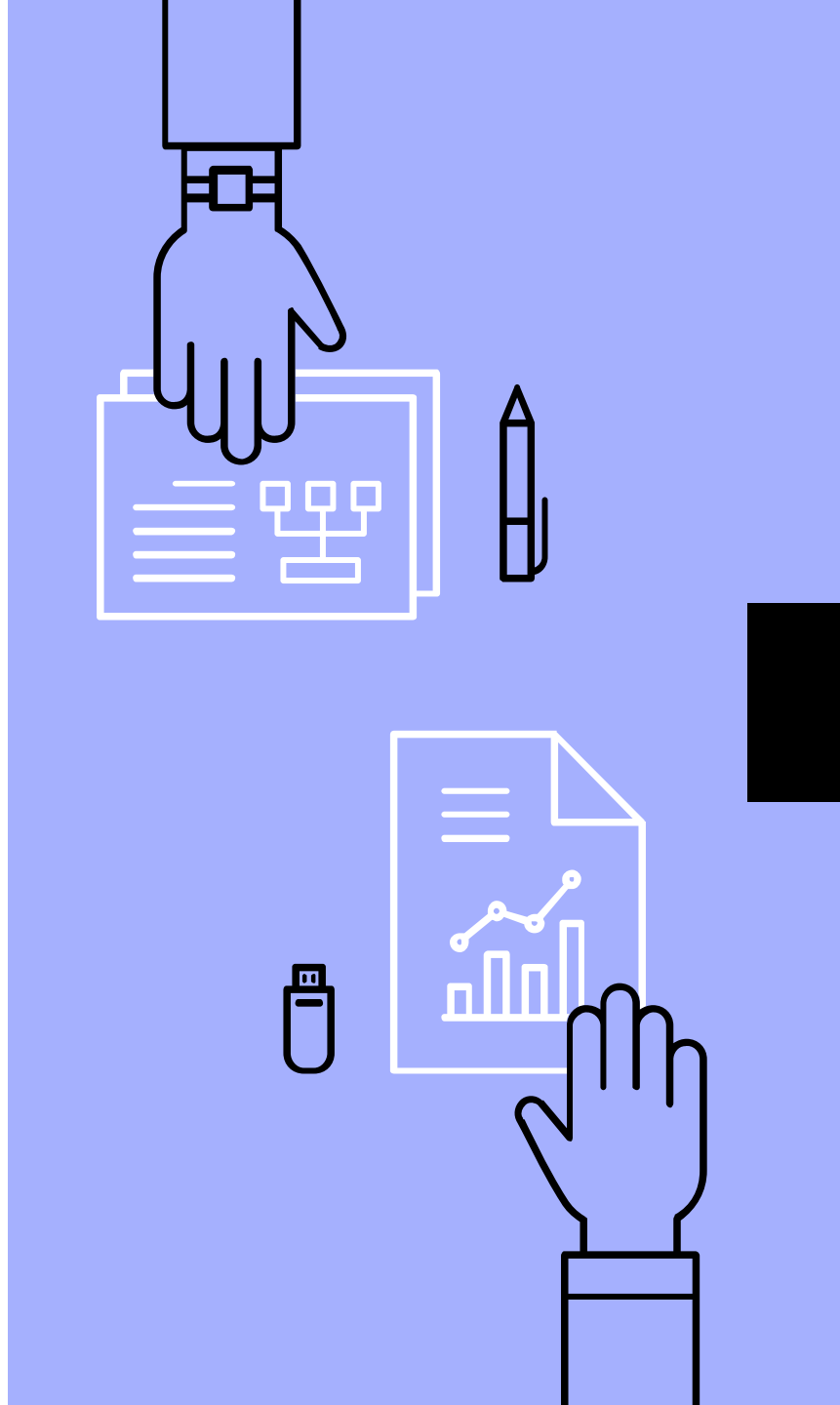
- Suponha que a lista de itens foi construída de forma que os itens estivessem em ordem crescente, de baixo para cima.
- Nesse caso, o algoritmo não precisa continuar examinando todos os itens para relatar que o item não foi encontrado. Isso pode parar imediatamente.



```python
def ordered_sequential_search(a_list, item):
    pos = 0
    found = False
    stop = False
    while pos < len(a_list) and not found and not stop:
        if a_list[pos] == item:
            found = True
        else:
            if a_list[pos] > item:
                stop = True
            else:
                pos = pos + 1

    return found


test_list = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(ordered_sequential_search(test_list, 3))
print(ordered_sequential_search(test_list, 13))
```



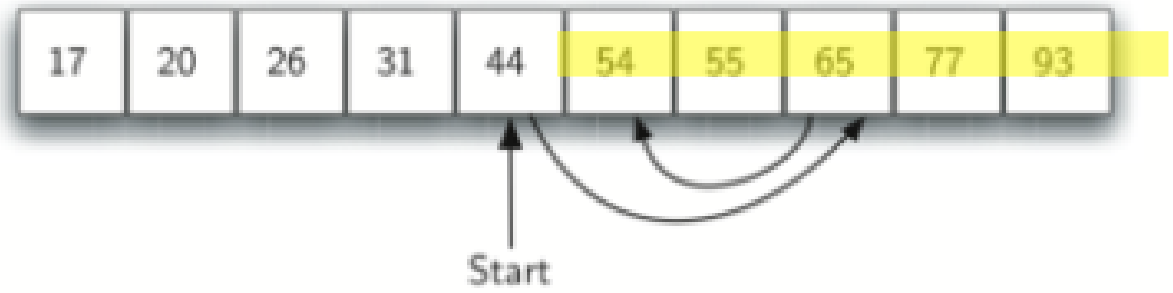
- Observe que, na melhor das hipóteses, podemos descobrir que o item não está na lista olhando apenas para um item. Em média, saberemos depois de examinar apenas $n/2$ itens. No entanto, essa técnica ainda é $O(n)$.
- Em resumo, uma busca sequencial é melhorada ordenando a lista apenas no caso em que não encontramos o item.

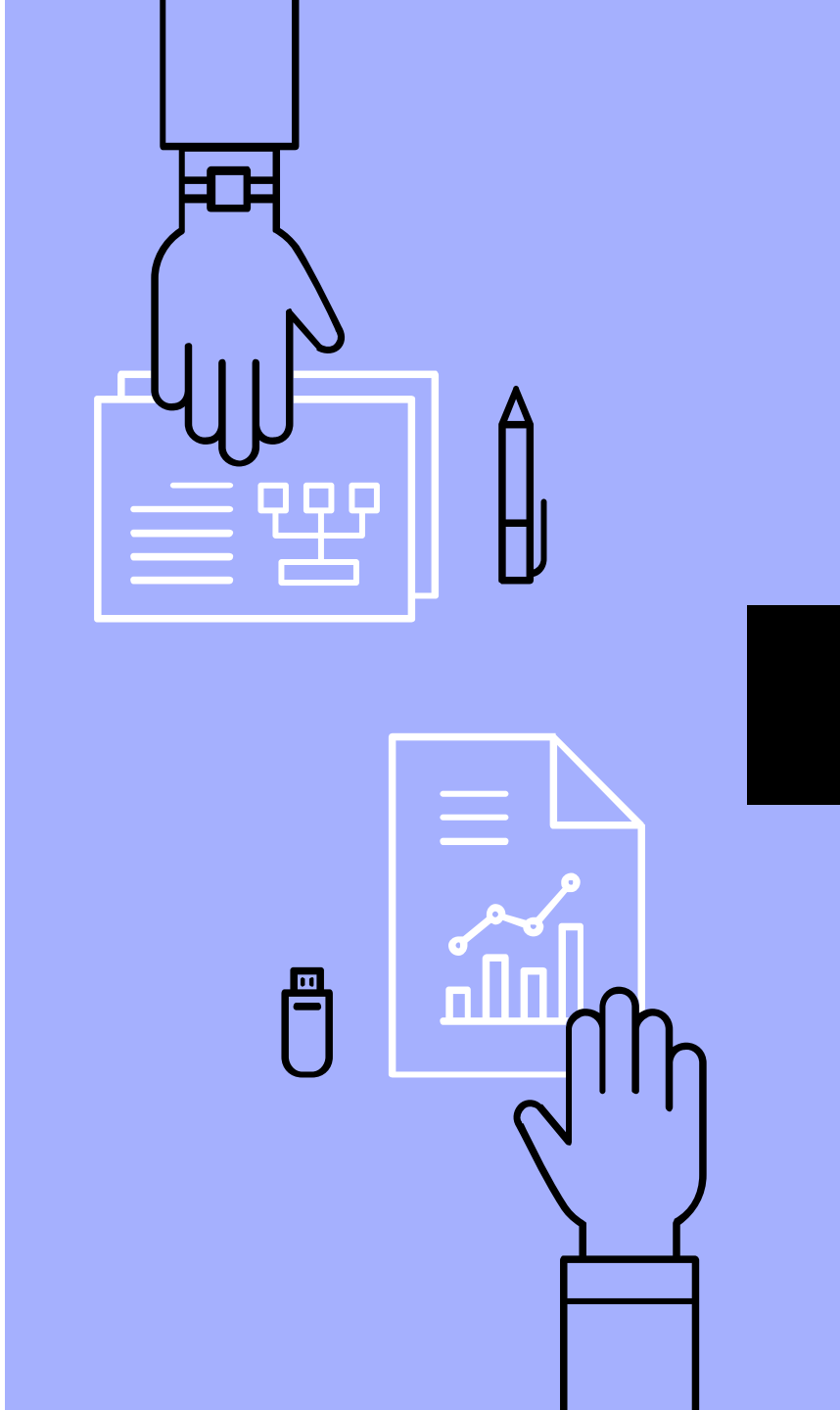


Pesquisa Binária

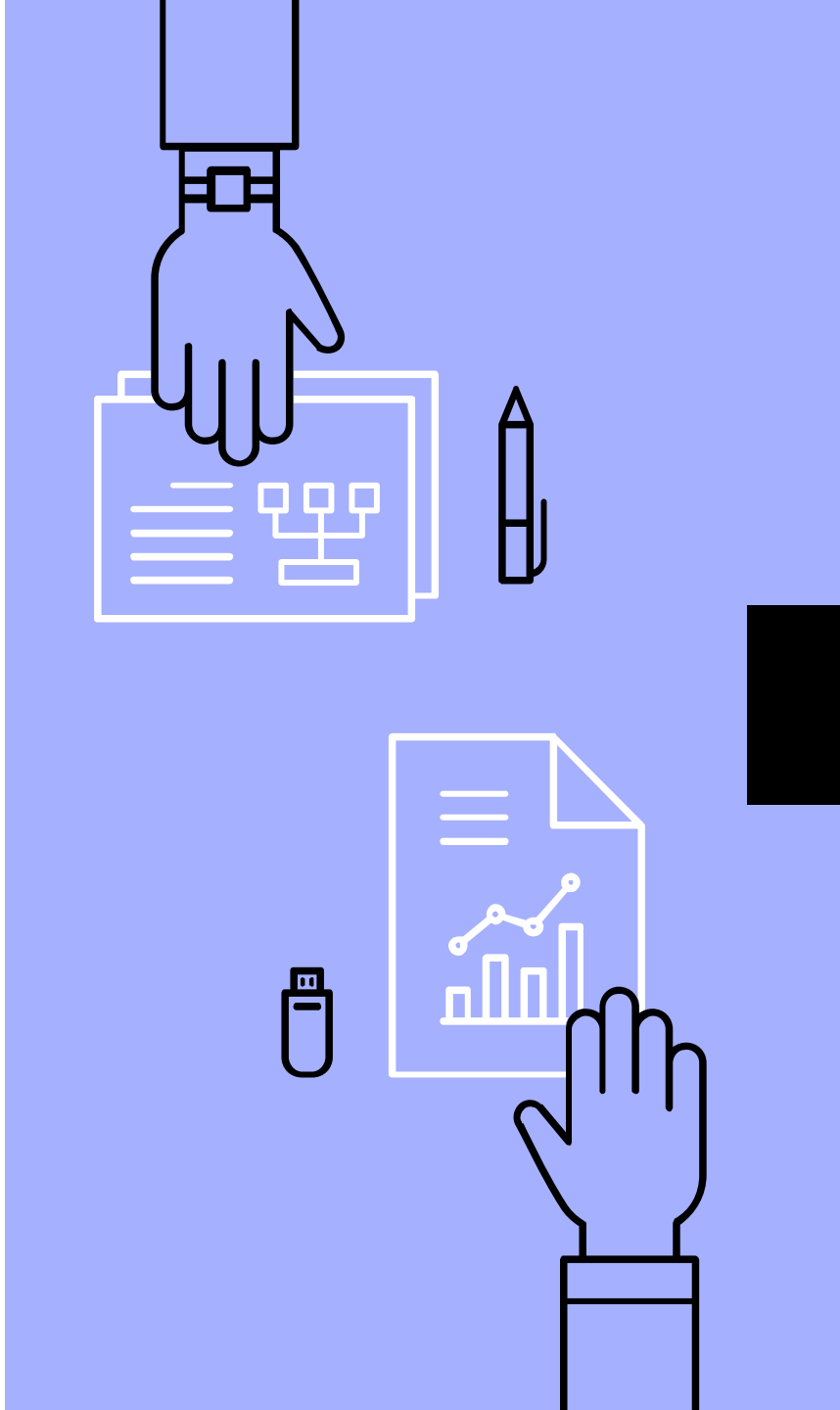
- É possível tirar maior proveito da lista ordenada se formos hábeis em nossas comparações. Na pesquisa sequencial, quando comparamos com o primeiro item, há no máximo n - 1 item a mais para examinar se o primeiro item não for o que estamos procurando.
- Em vez de pesquisar a lista em sequência, uma pesquisa binária começará examinando o item do meio.
- Se esse item for o que estamos procurando, estamos prontos.
- Se não for o item correto, podemos usar a natureza ordenada da lista para eliminar metade dos itens restantes.
- Se o item que procuramos for maior do que o item do meio, sabemos que toda a metade inferior da lista, bem como o item do meio, podem ser eliminados de uma análise posterior.
- O item, se estiver na lista, deve estar na metade superior.

- Podemos então repetir o processo com a metade superior. Comece pelo item do meio e compare-o com o que estamos procurando. Novamente, nós o encontramos ou dividimos a lista ao meio, eliminando assim outra grande parte de nosso possível espaço de busca.



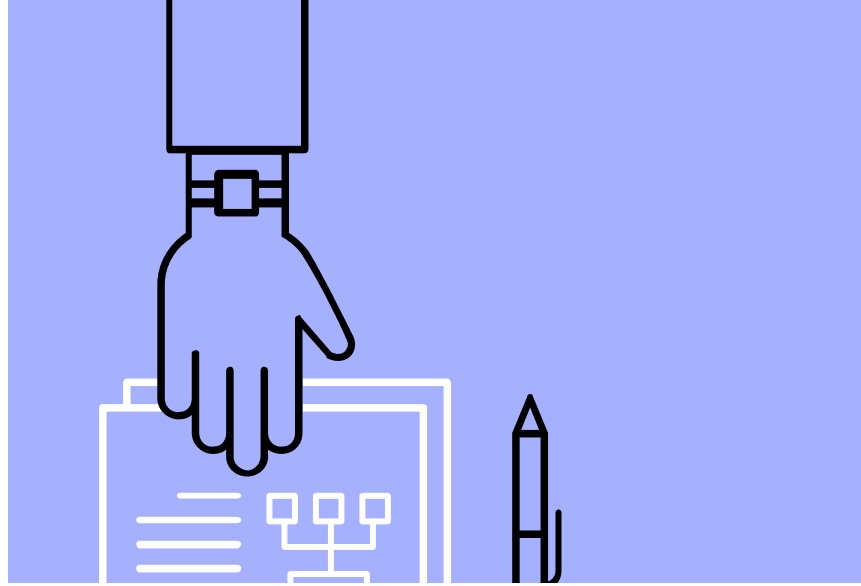


- Antes de prosseguirmos para a análise, devemos observar que esse algoritmo é um ótimo exemplo de estratégia de dividir para conquistar. Dividir e conquistar significa que dividimos o problema em partes menores, resolvemos as partes menores de alguma forma e, em seguida, remontamos todo o problema para obter o resultado.
- Quando realizamos uma pesquisa binária de uma lista, primeiro verificamos o item do meio. Se o item que estamos procurando for menor que o item do meio, podemos simplesmente realizar uma pesquisa binária da metade esquerda da lista original. Da mesma forma, se o item for maior, podemos realizar uma pesquisa binária da metade direita.
- De qualquer forma, esta é uma chamada recursiva para a função de pesquisa binária passando uma lista menor.



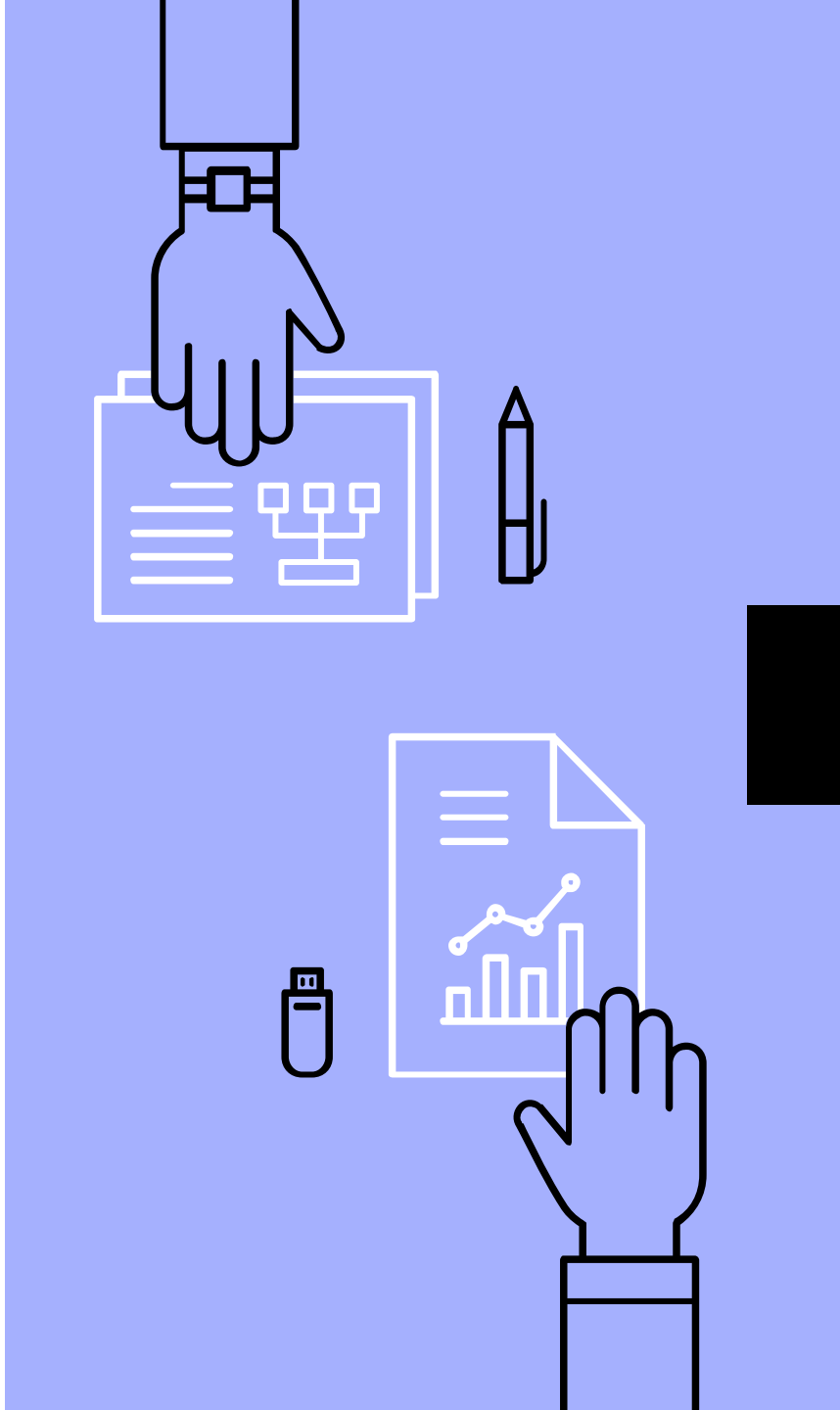
Análise da Pesquisa Binária

- Para analisar o algoritmo de pesquisa binária, precisamos lembrar que cada comparação elimina cerca da metade dos itens restantes da consideração.
- Qual é o número máximo de comparações que esse algoritmo exigirá para verificar a lista inteira?
- Se começarmos com n itens, cerca de $n/2$ itens sobrarão após a primeira comparação. Após a segunda comparação, haverá cerca de $n/4$. Então $n/8$, $n/16$ e assim por diante. Quantas vezes podemos dividir a lista?

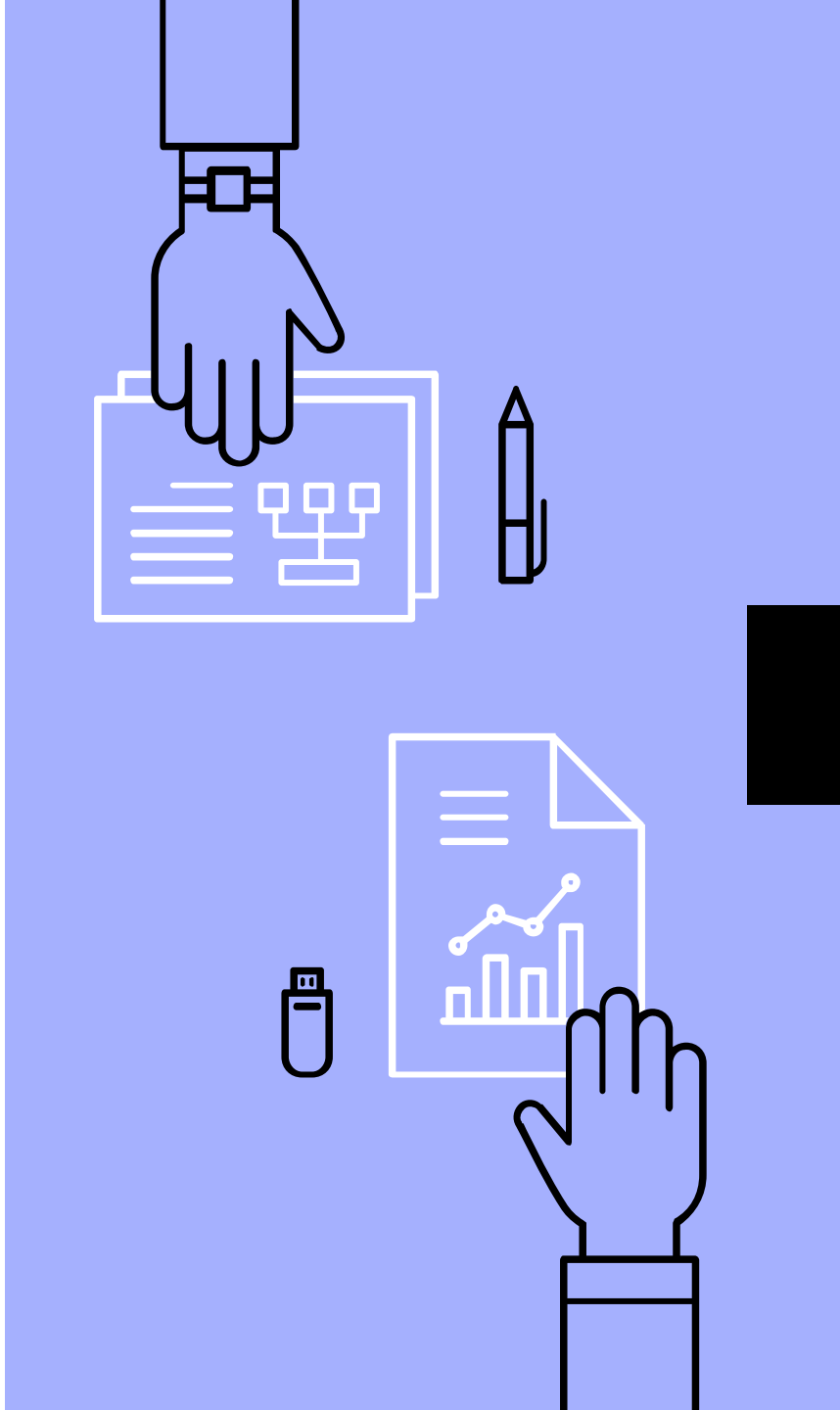


Comparisons	Approximate Number Of Items Left
1	$\frac{n}{2}$
2	$\frac{n}{4}$
3	$\frac{n}{8}$
...	...
i	$\frac{n}{2^i}$

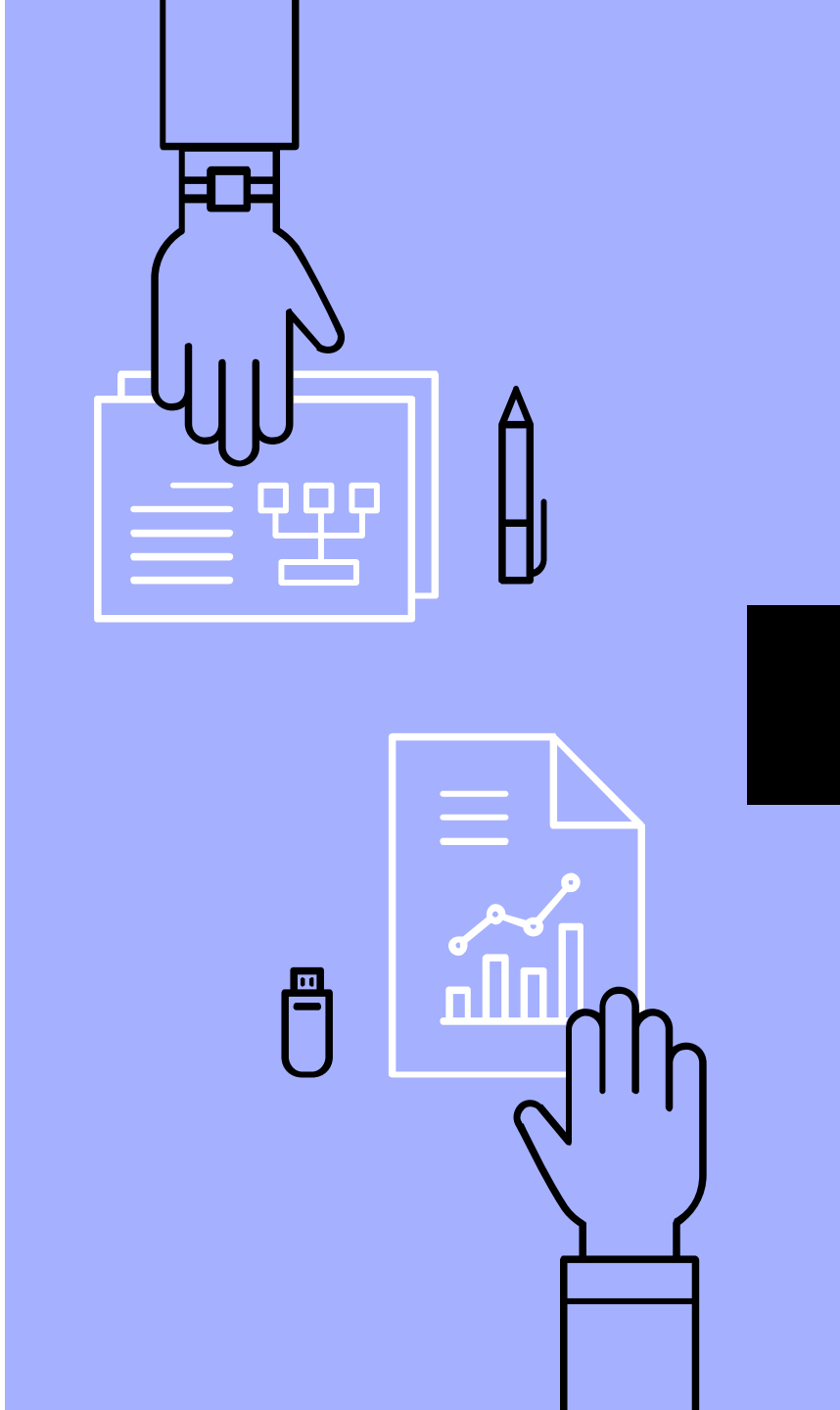
- Quando dividimos a lista várias vezes, terminamos com uma lista que tem apenas um item. Ou esse é o item que procuramos ou não.
- De qualquer maneira, terminamos. O número de comparações necessárias para chegar a este ponto é i onde $n/2^i = 1$.
- Resolver i nos dá $i = \log n$. O número máximo de comparações é logarítmico em relação ao número de itens na lista. Portanto, a busca binária é $O(\log n)$.



- A análise que fizemos acima assumiu que o operador de fatia leva um tempo constante. No entanto, sabemos que o operador ***slice*** em Python é na verdade $O(k)$. Isso significa que a pesquisa binária usando fatia não será executada em tempo logarítmico estrito.
- Felizmente, isso pode ser remediado passando a lista junto com os índices inicial e final.

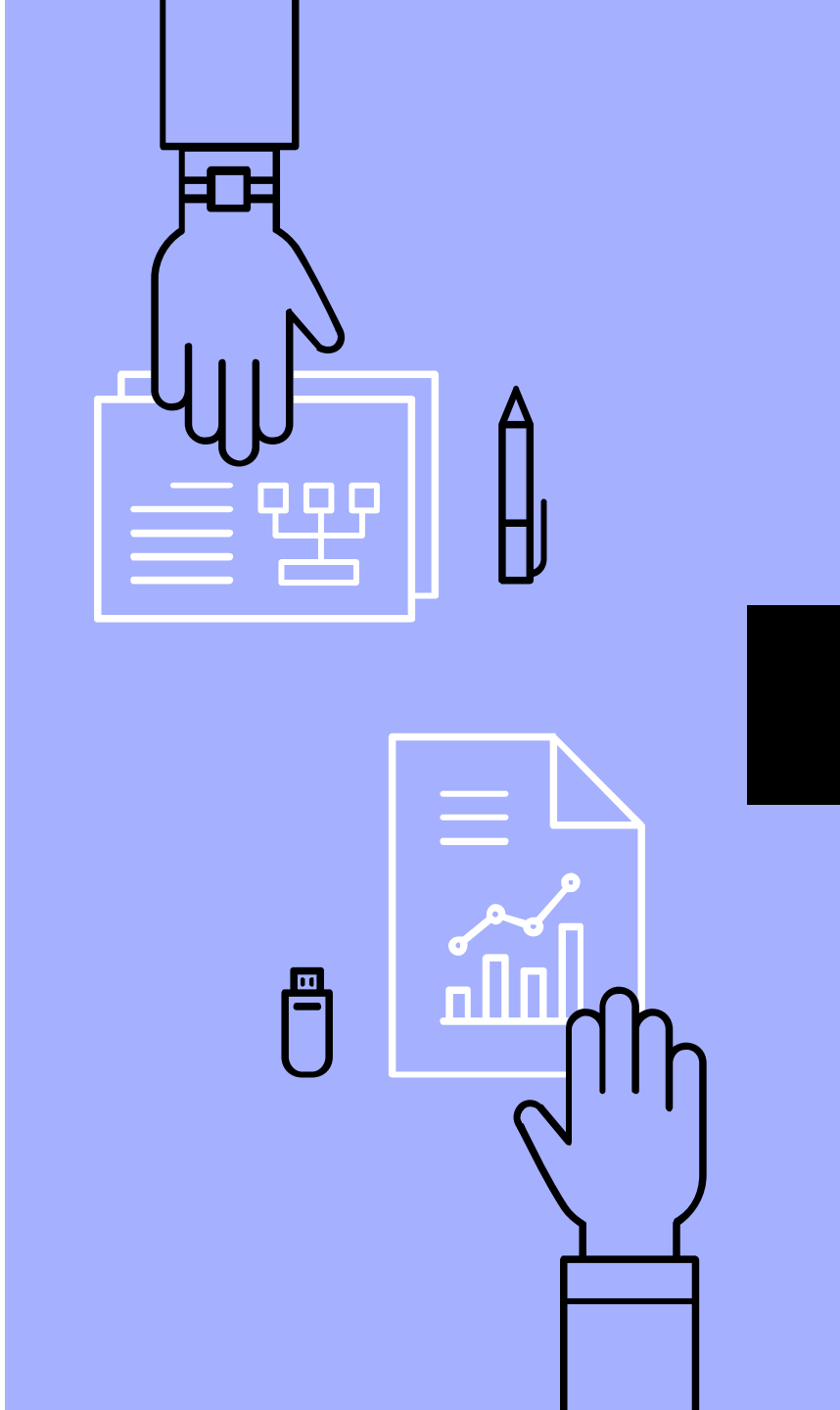


- Embora uma pesquisa binária geralmente seja melhor do que uma pesquisa sequencial, é importante observar que, para valores pequenos de n, o custo adicional de classificação provavelmente não vale a pena.
- Na verdade, devemos sempre considerar se é rentável assumir o trabalho extra de classificação para obter benefícios de pesquisa. Se pudermos classificar uma vez e pesquisar várias vezes, o custo da classificação não será tão significativo.
- No entanto, para listas grandes, classificar apenas uma vez pode ser tão caro que simplesmente realizar uma pesquisa sequencial desde o início pode ser a melhor escolha.

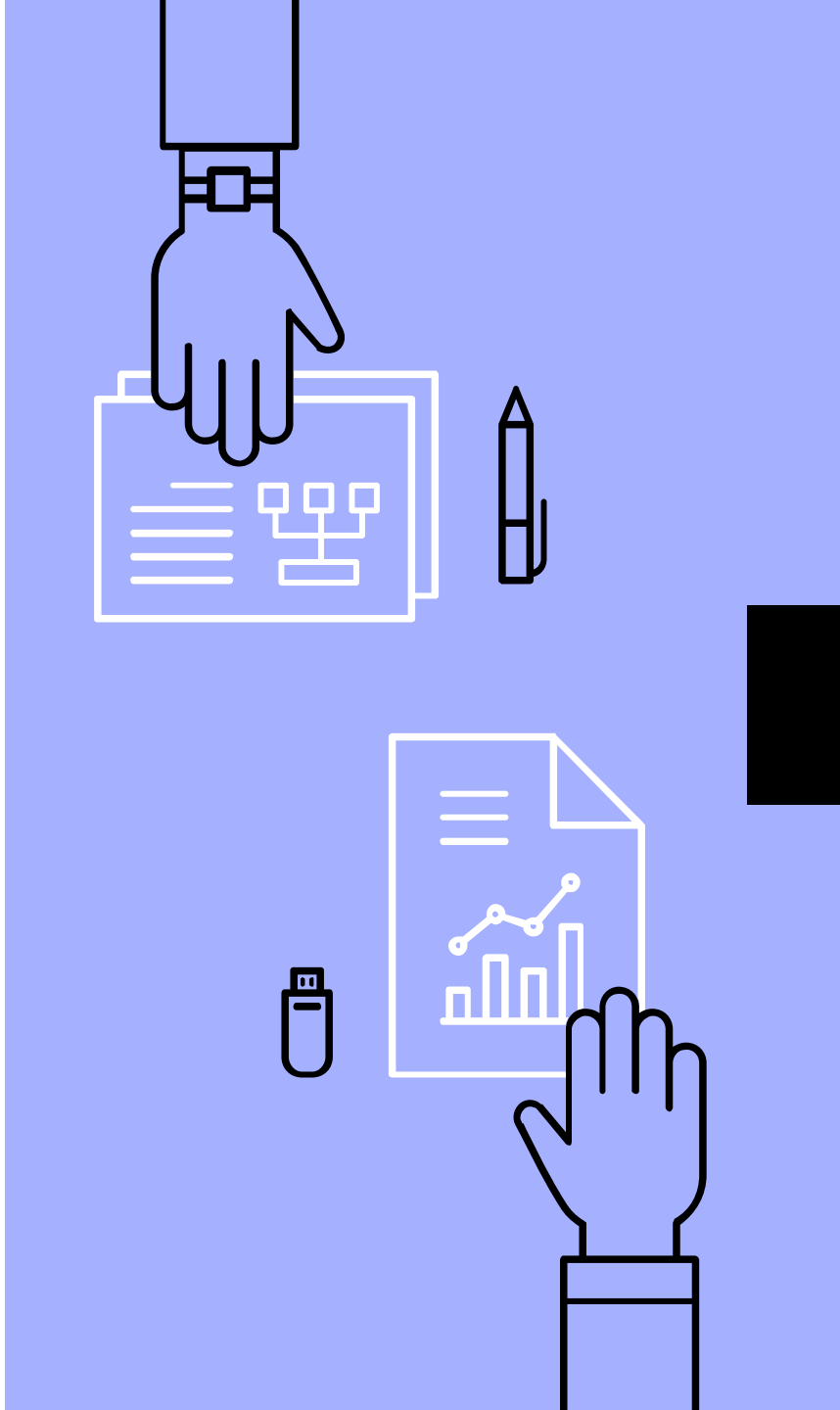


Ordenação (classificação)

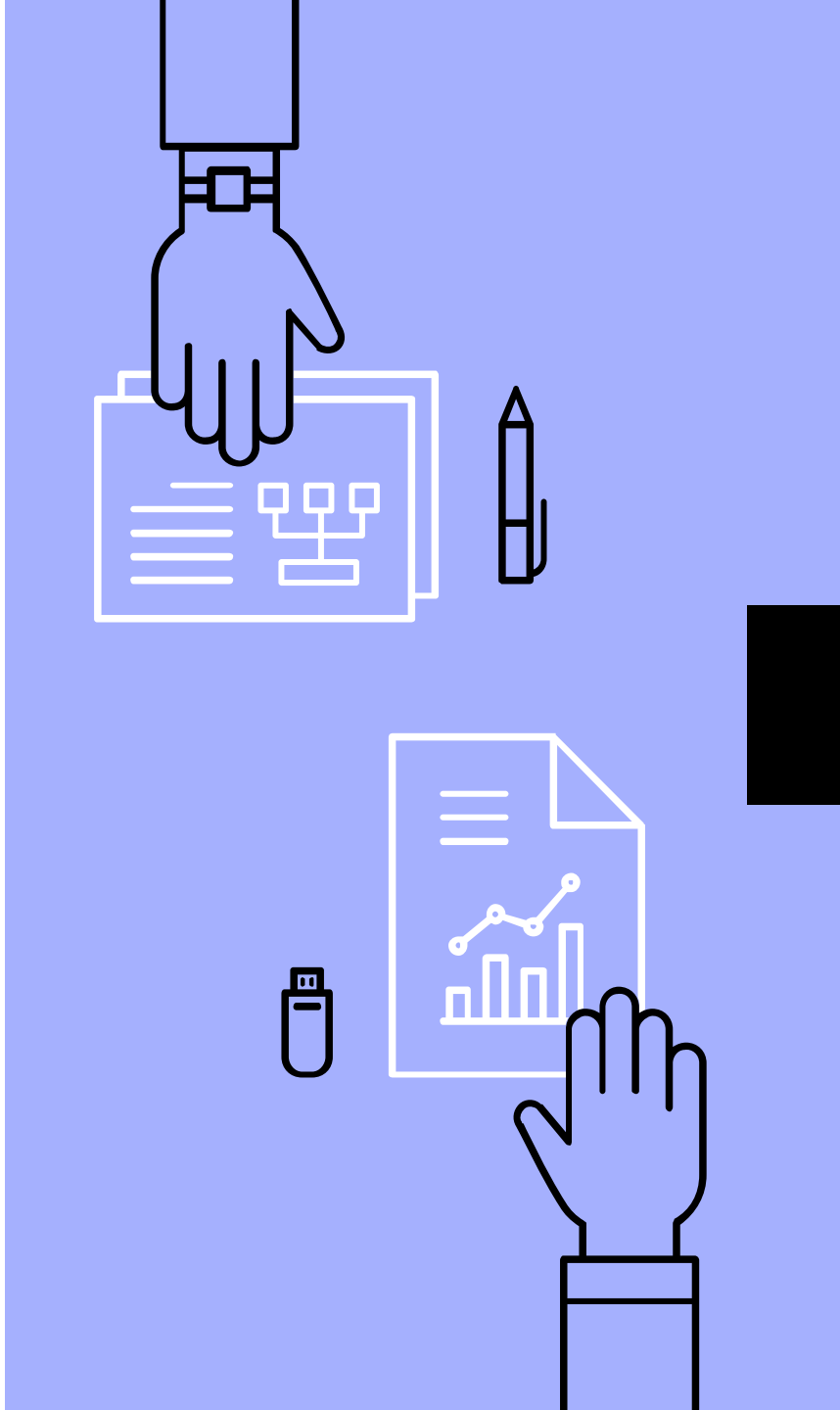
- A classificação é o processo de colocar os elementos de uma coleção em algum tipo de ordem.
- Por exemplo, uma lista de palavras pode ser classificada em ordem alfabética ou por comprimento. Uma lista de cidades pode ser classificada por população, área ou código postal.
- Já vimos alguns algoritmos que foram capazes de se beneficiar de uma lista ordenada.



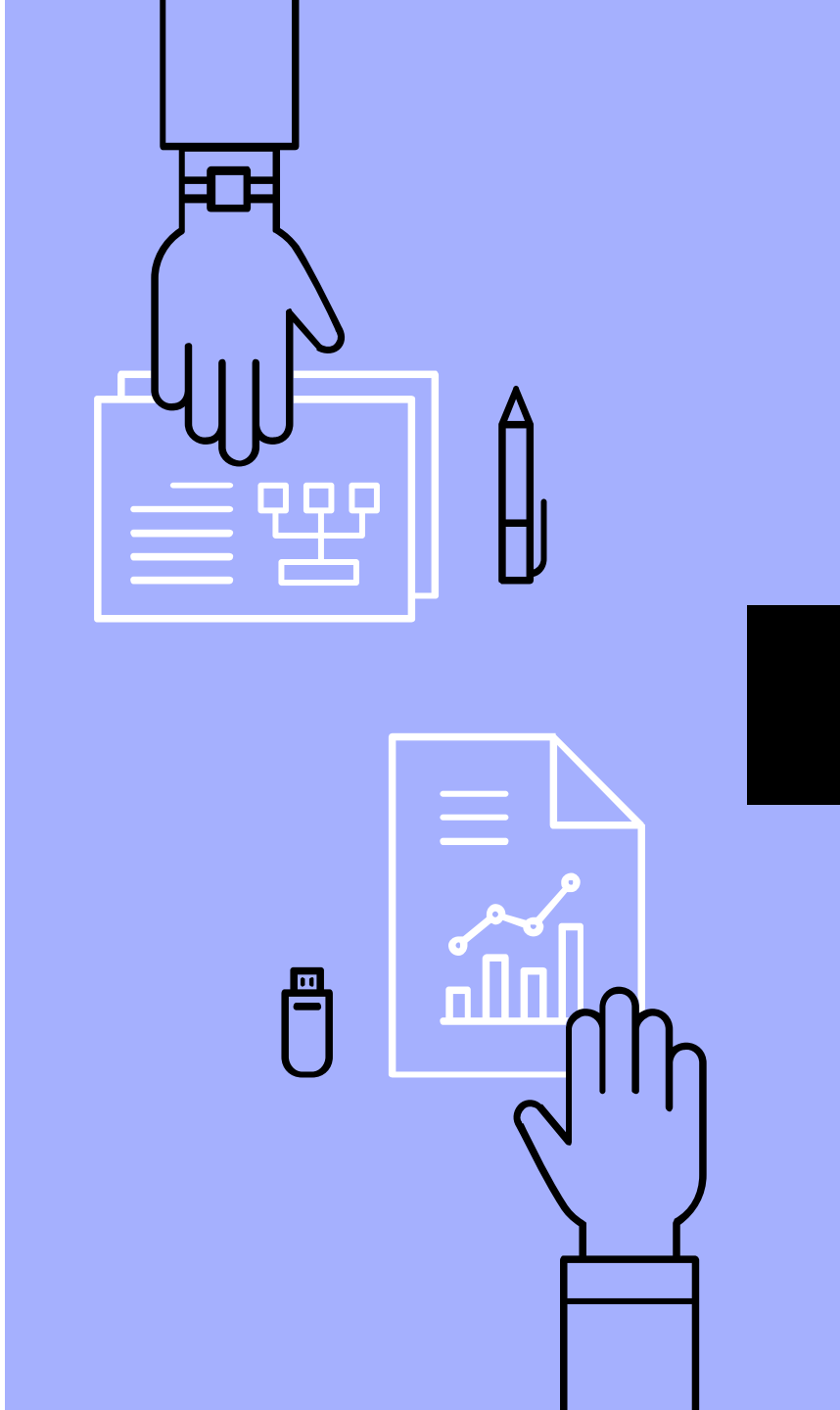
- Existem muitos, muitos algoritmos de classificação que foram desenvolvidos e analisados. Isso sugere que a classificação é uma importante área de estudo na ciência da computação.
- Classificar um grande número de itens pode exigir uma quantidade substancial de recursos de computação. Assim como na pesquisa, a eficiência de um algoritmo de classificação está relacionada ao número de itens sendo processados.
- Para pequenas coleções, um método de classificação complexo pode ser mais problemático do que compensador. A sobrecarga pode ser muito alta.



- Por outro lado, para coleções maiores, queremos aproveitar o máximo possível de melhorias. Aqui veremos e discutiremos várias técnicas de classificação e as compararemos em relação ao seu tempo de execução.
- Antes de entrar em algoritmos específicos, devemos pensar sobre as operações que podem ser usadas para analisar um processo de classificação.

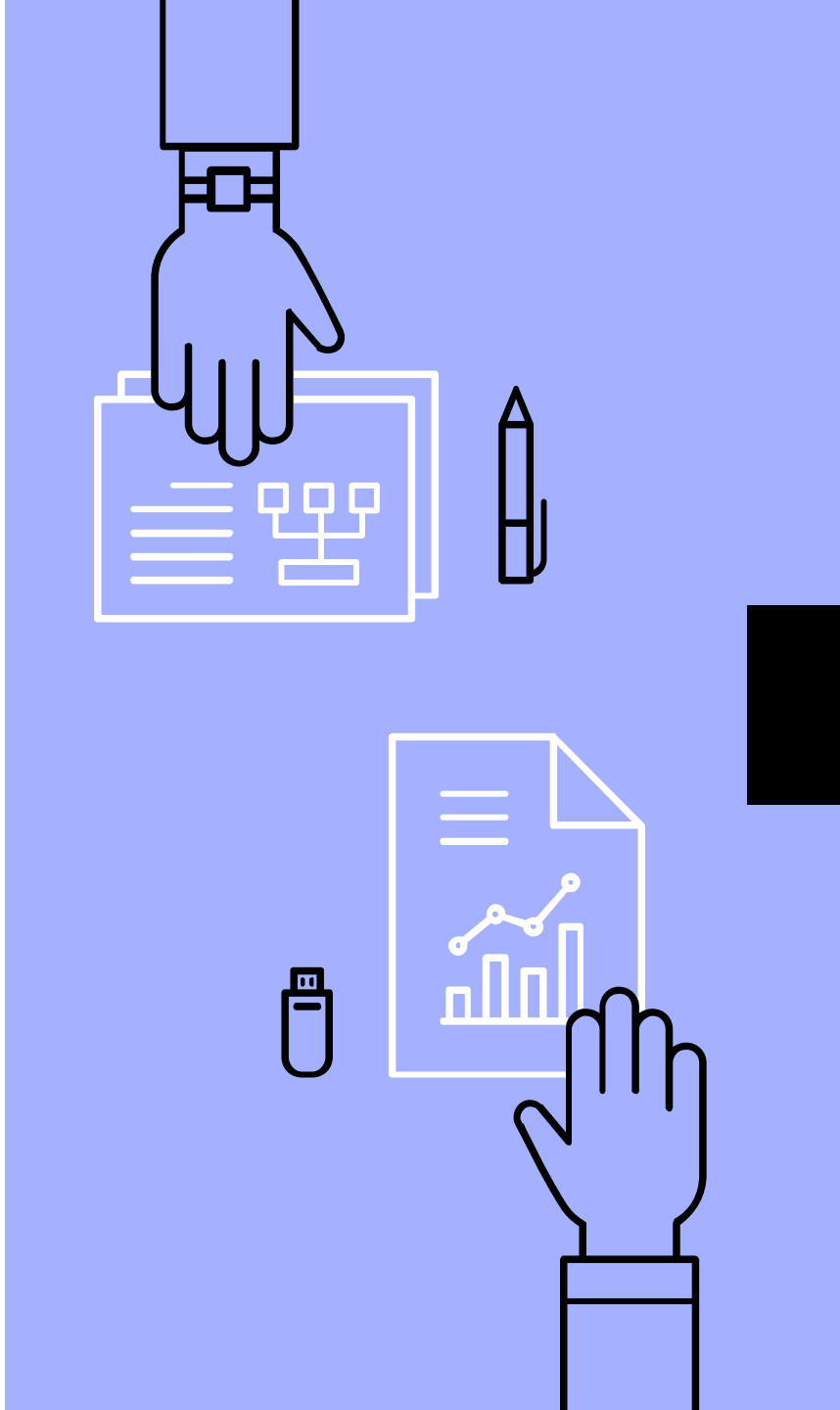


- Primeiro, será necessário comparar dois valores para ver qual é o menor (ou o maior). Para classificar uma coleção, será necessário ter uma maneira sistemática de comparar valores para ver se eles estão fora de ordem. O número total de comparações será a forma mais comum de medir um procedimento de classificação.
- Em segundo lugar, quando os valores não estão na posição correta uns com os outros, pode ser necessário trocá-los. Essa troca é uma operação cara e o número total de trocas também será importante para avaliar a eficiência geral do algoritmo.

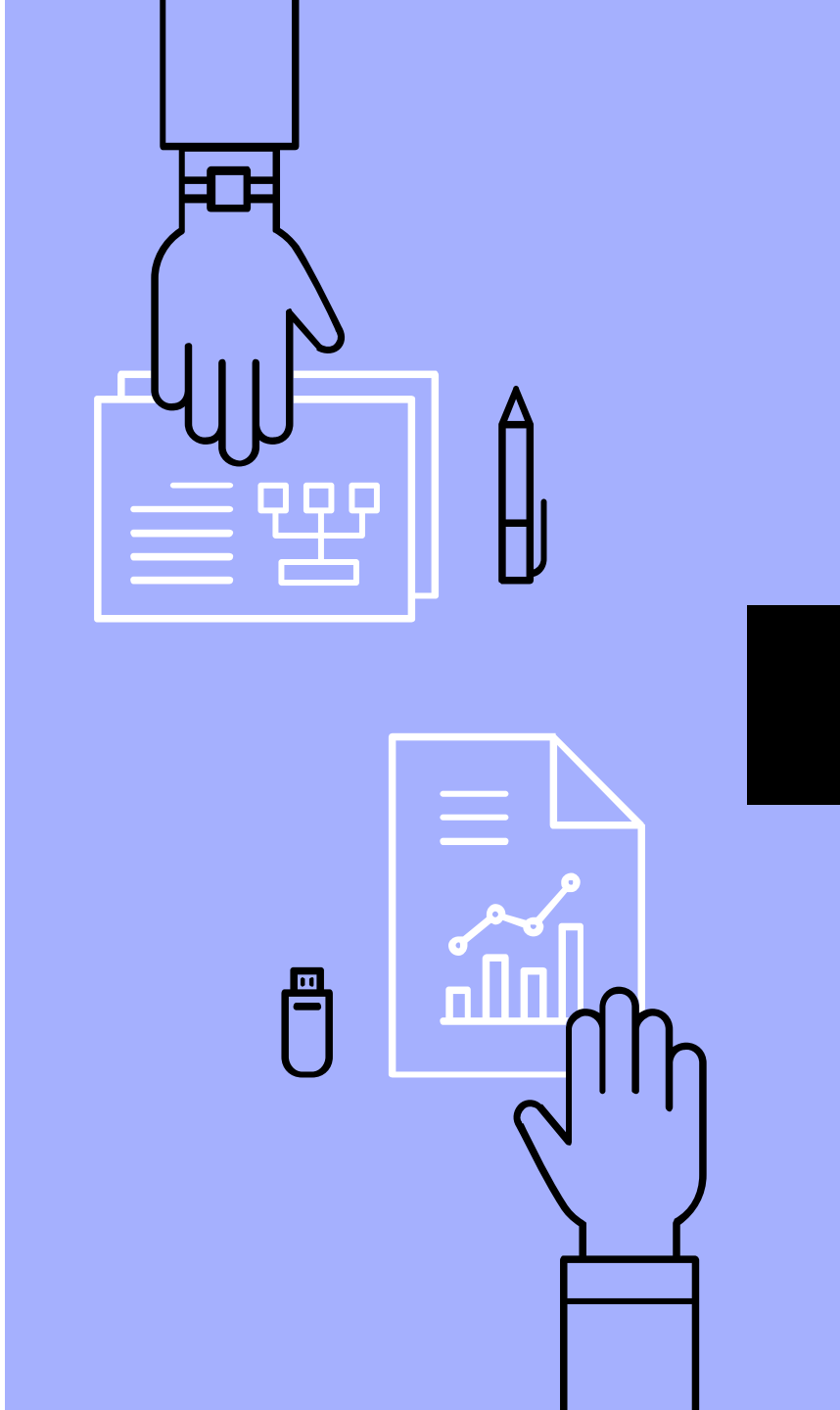


Bubble Sort (Ordenação Bolha)

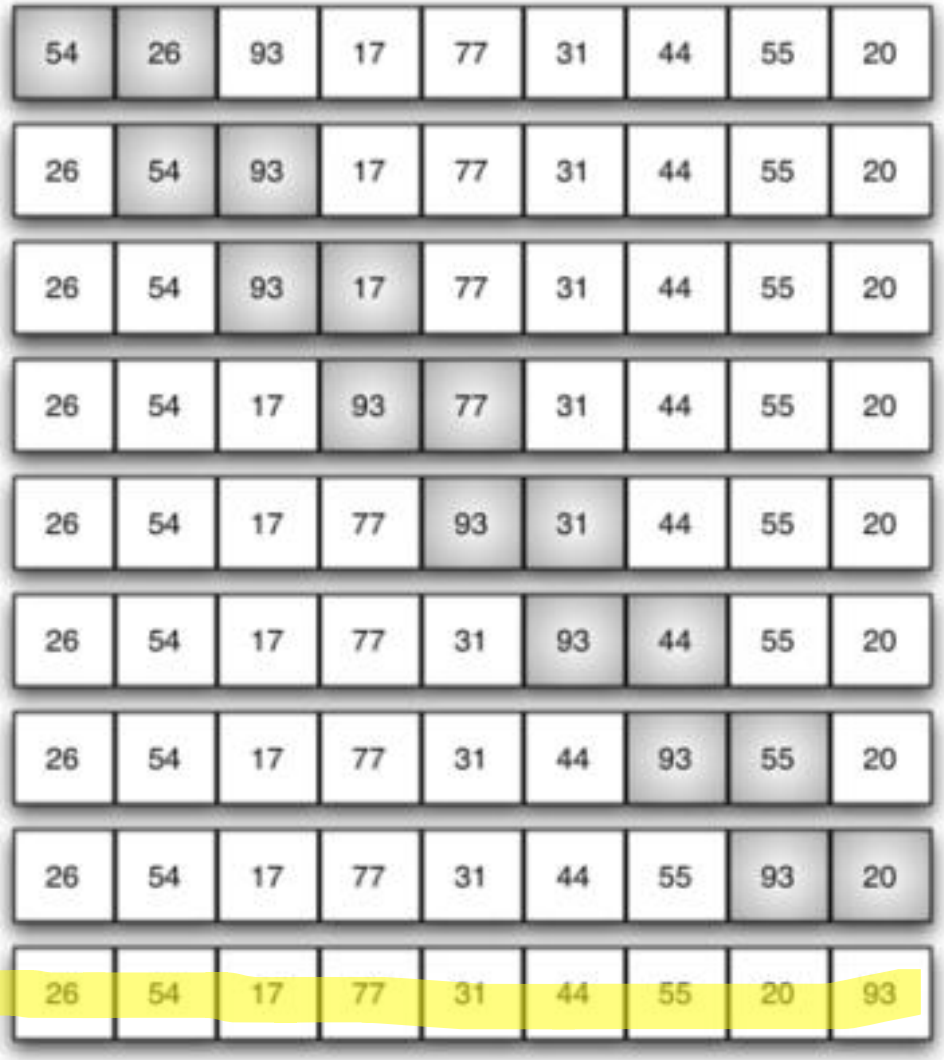
- A classificação por bolha faz várias passagens por uma lista.
- Ele compara itens adjacentes e troca aqueles que estão fora de ordem esperada. Cada passagem pela lista coloca o próximo maior valor em seu lugar apropriado.
- Em essência, cada item “borbulha” até o local ao qual pertence.



- Os itens sombreados estão sendo comparados para ver se estão fora de ordem.
- Se houver n itens na lista, então haverá n - 1 pares de itens que precisam ser comparados na primeira passagem.
- É importante observar que, uma vez que o maior valor da lista faça parte de um par, ele será movido continuamente até que a passagem seja concluída.



First pass



Exchange

No Exchange

Exchange

Exchange

Exchange

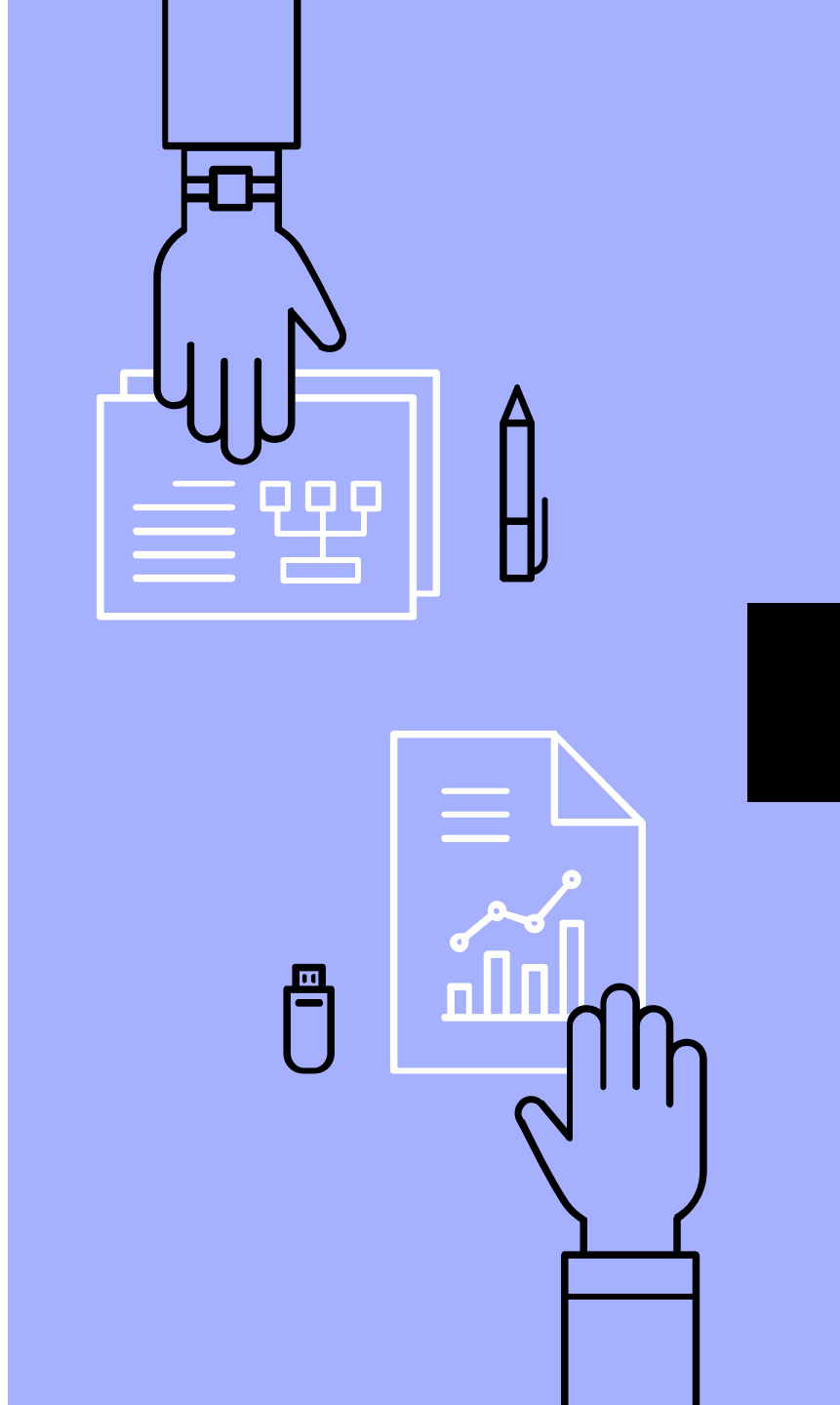
Exchange

Exchange

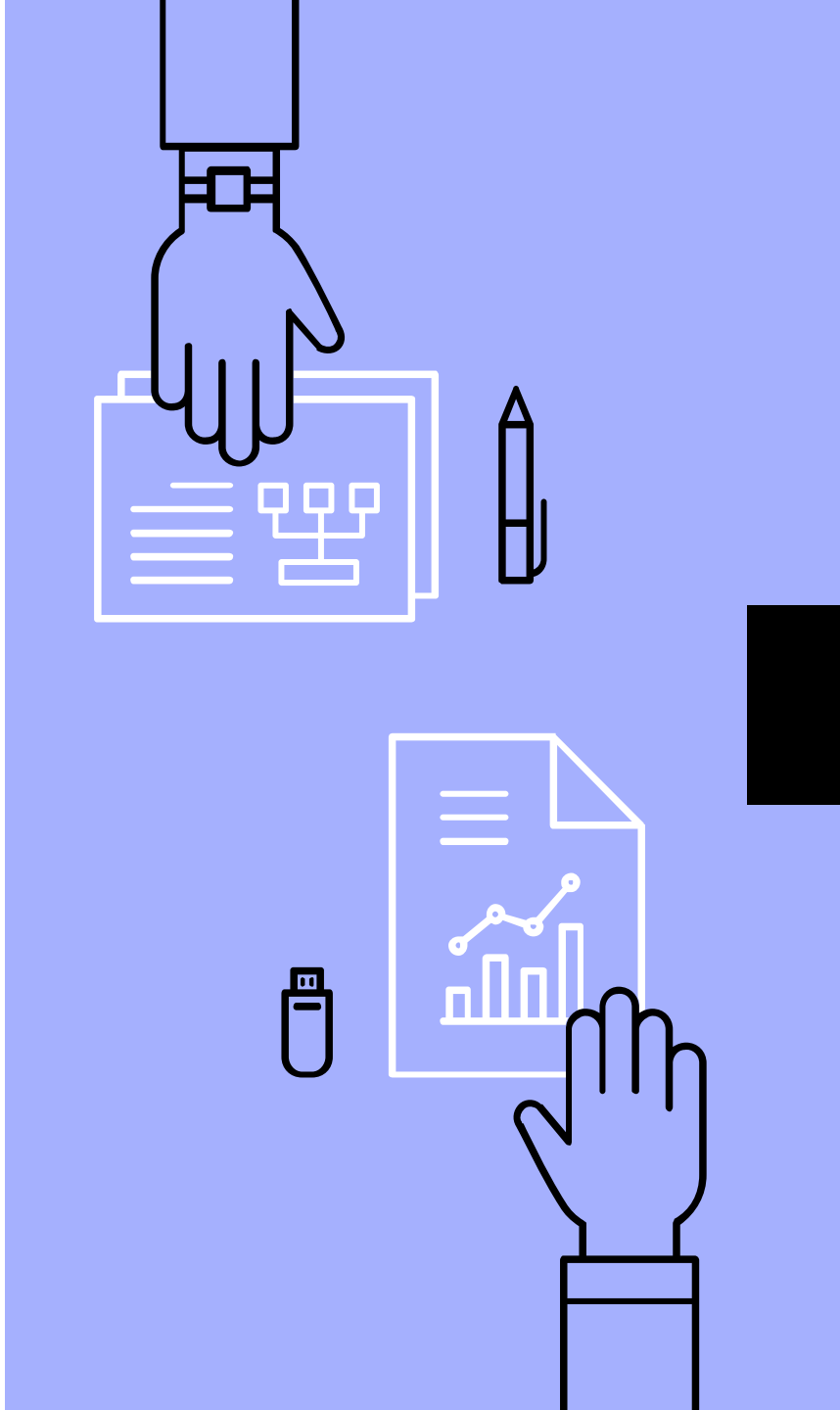
Exchange

93 in place after first pass

- No início da segunda passagem, o maior valor está agora em vigor. Restam n - 1 itens para classificar, o que significa que haverá n - 2 pares.
- Uma vez que cada passagem coloca o próximo maior valor no lugar, o número total de passagens necessárias será n - 1.
- Depois de concluir as n - 1 passagens, o menor item deve estar na posição correta sem a necessidade de processamento adicional.

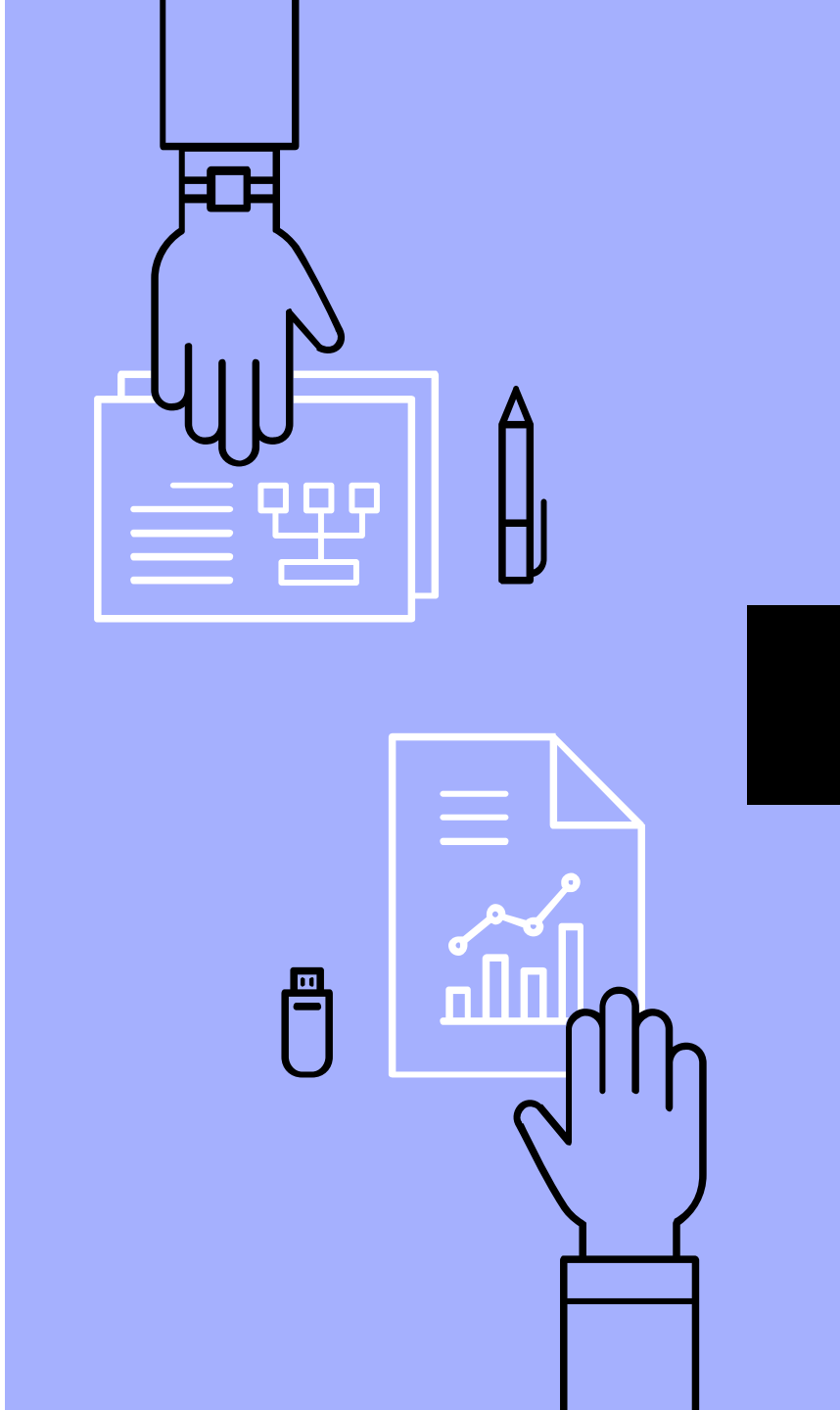


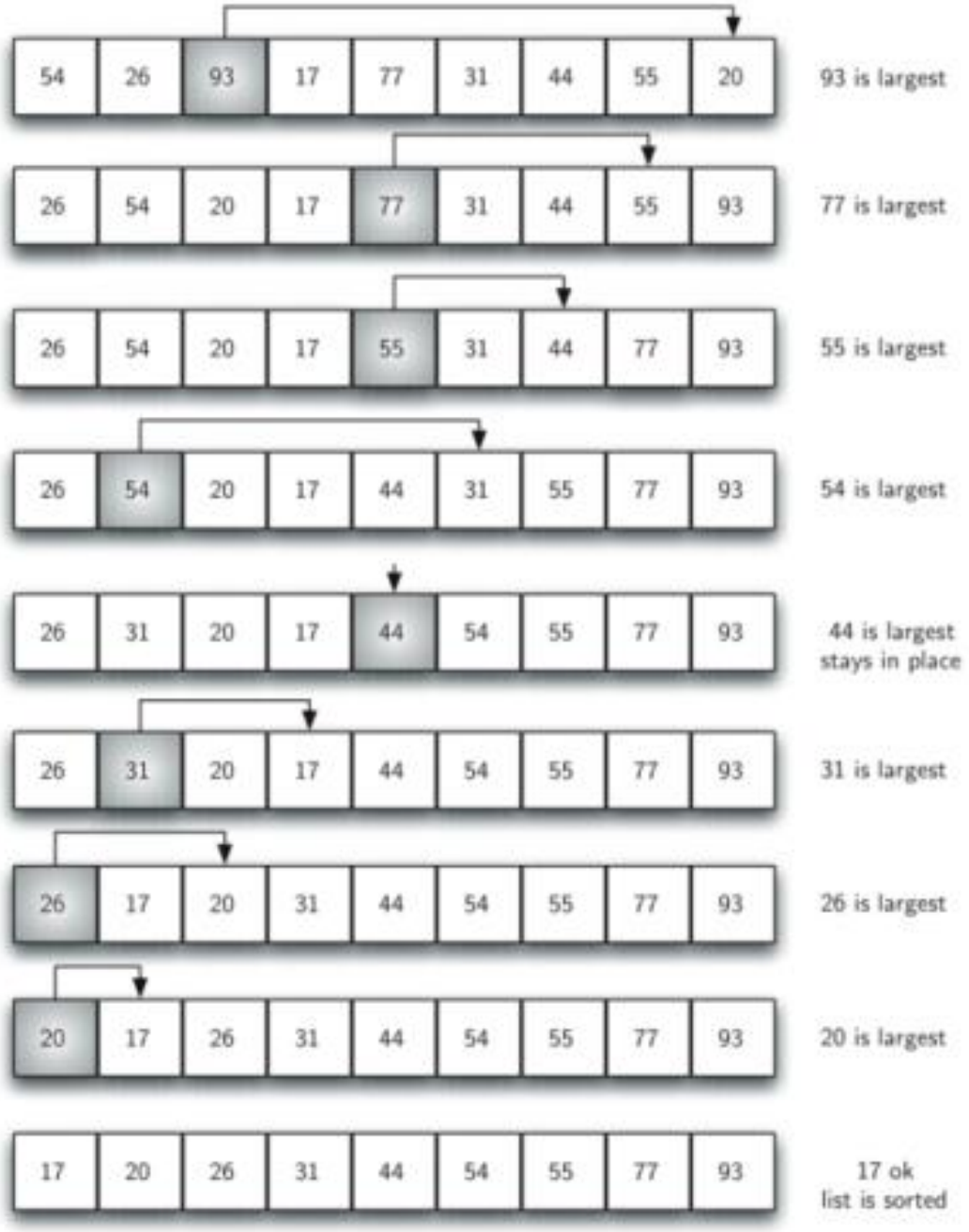
- Em particular, se durante um passe não houver trocas, sabemos que a lista deve ser ordenada.
- Uma classificação por bolha pode ser modificada para parar mais cedo se descobrir que a lista foi classificada.
- Isso significa que, para listas que exigem apenas algumas passagens, uma classificação por bolha pode ter a vantagem de reconhecer a lista classificada e parar.

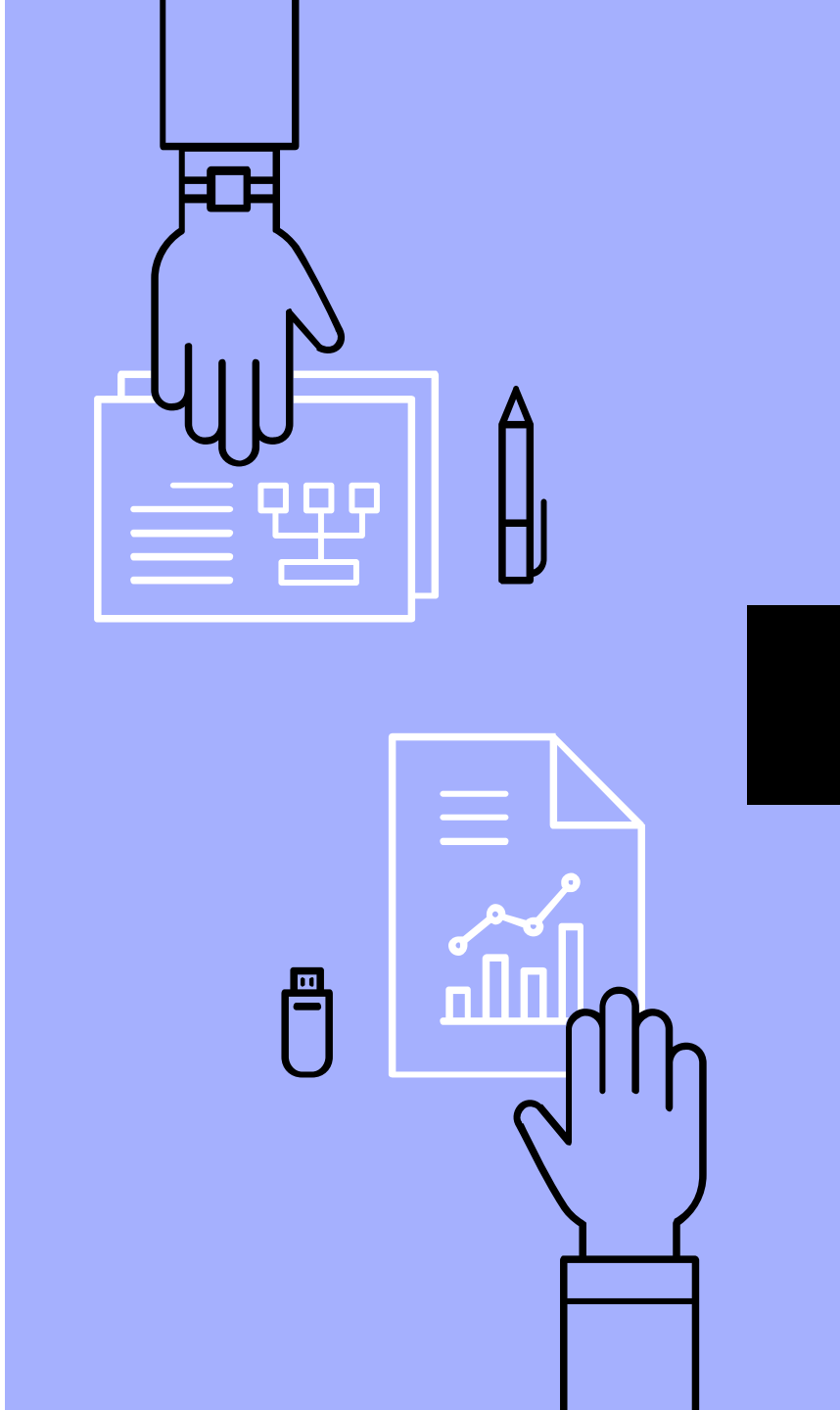


Ordenação por Seleção (Selection Sort)

- A classificação por seleção melhora a classificação por bolha, fazendo apenas uma troca para cada passagem pela lista.
- Para fazer isso, uma ordenação de seleção procura o maior valor à medida que faz uma passagem e, após concluir a passagem, coloca-o no local adequado.
- Tal como acontece com uma classificação por bolha, após a primeira passagem, o maior item está no lugar correto.
- Após a segunda passagem, a próxima maior está no lugar. Este processo continua e requer n - 1 passagens para classificar n itens, uma vez que o item final deve estar no lugar após a passagem (n - 1).

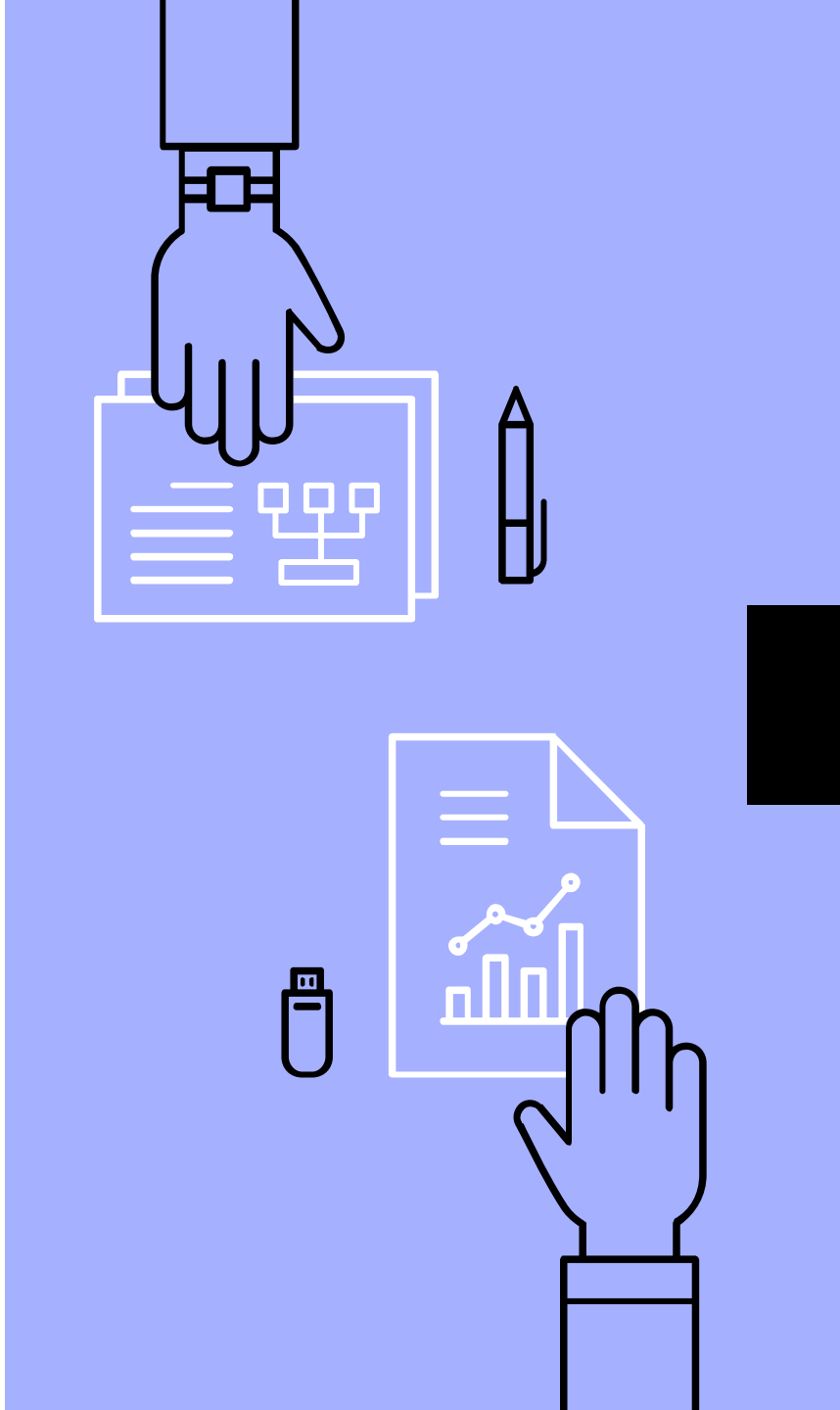



54 26 93 17 77 31 44 55 20
93 is largest
26 54 20 17 77 31 44 55 93
77 is largest
26 54 20 17 55 31 44 77 93
55 is largest
26 54 20 17 44 31 55 77 93
54 is largest
26 31 20 17 44 54 55 77 93
44 is largest
stays in place
26 31 20 17 44 54 55 77 93
31 is largest
26 17 20 31 44 54 55 77 93
26 is largest
20 17 26 31 44 54 55 77 93
20 is largest
17 20 26 31 44 54 55 77 93
17 ok
list is sorted

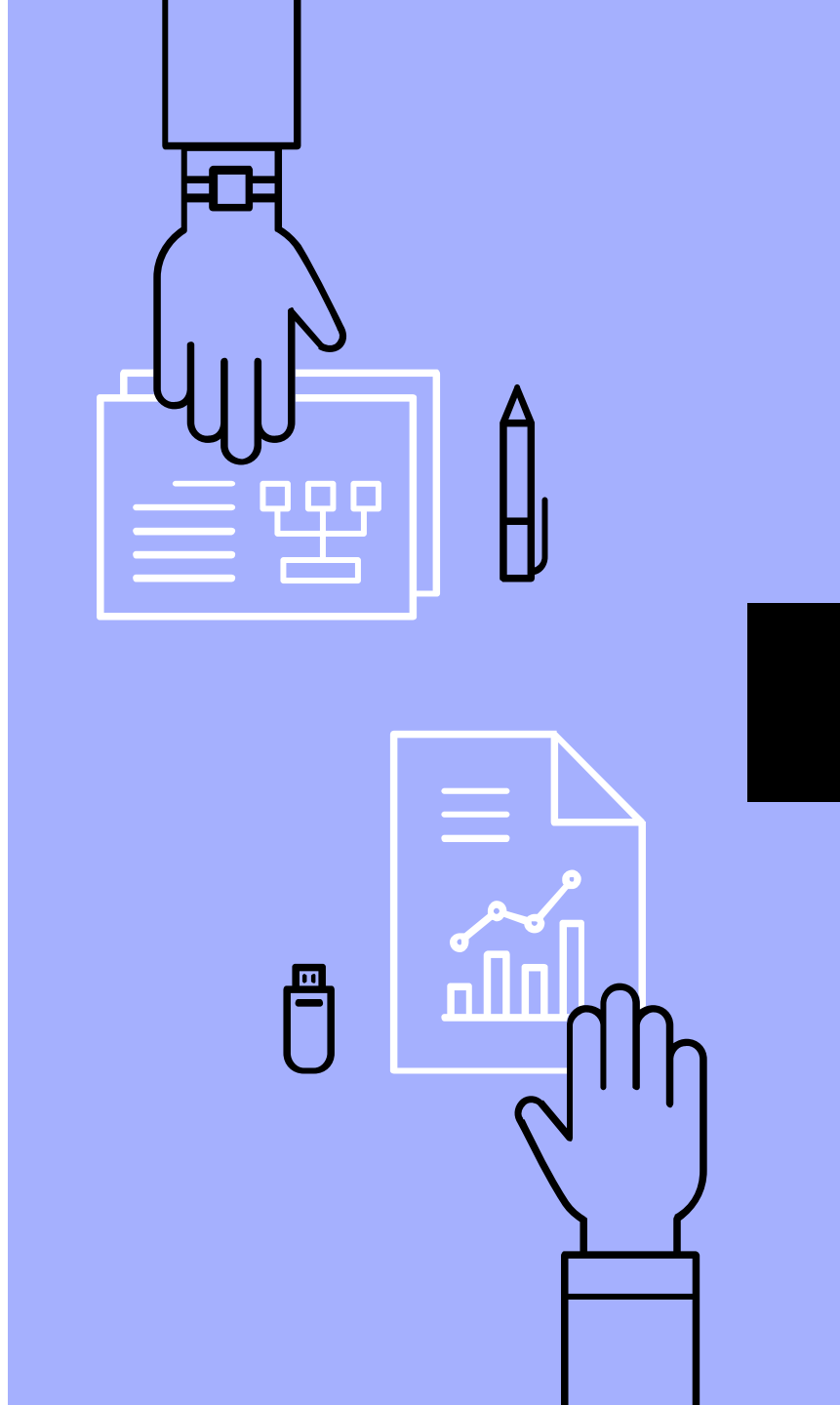


Ordenação por Inserção (Insertion Sort)

- A classificação de inserção, embora ainda $O(n^2)$, funciona de maneira um pouco diferente.
- Ele sempre mantém uma sublista classificada nas posições inferiores da lista.
- Cada novo item é então "inserido" de volta na sublista anterior de forma que a sublista classificada seja um item maior.
- Os itens sombreados representam as sublistas ordenadas conforme o algoritmo faz cada passagem.



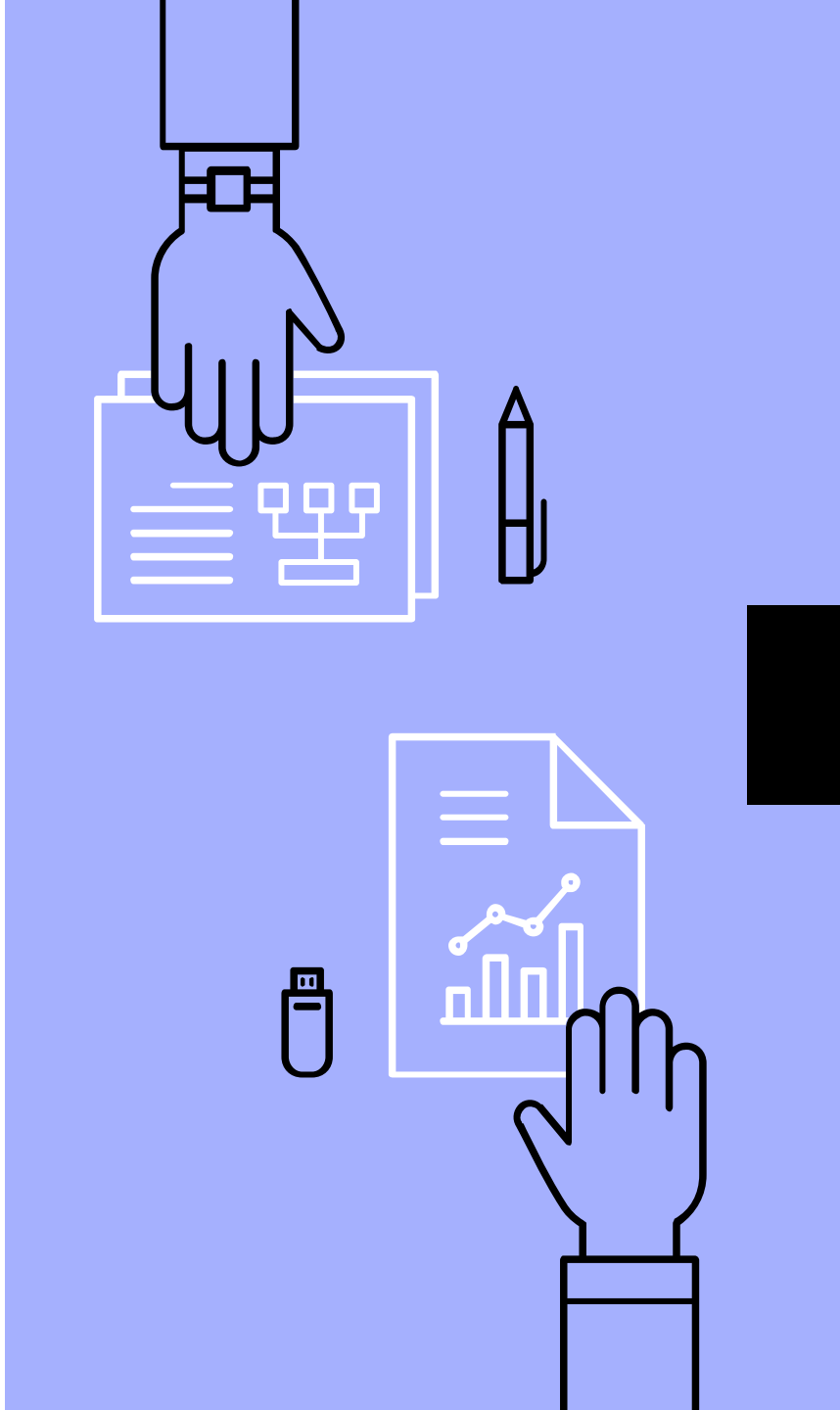
- A implementação de insertion_sort mostra que há novamente n - 1 passagens para classificar n itens.
- A iteração começa na posição 1 e se move até a posição n - 1, pois esses são os itens que precisam ser inseridos de volta nas sublistas classificadas.
- Lembre-se de que esta não é uma troca completa como era realizada nos algoritmos anteriores.
- O número máximo de comparações para uma classificação por inserção é a soma dos primeiros n – 1 inteiros.



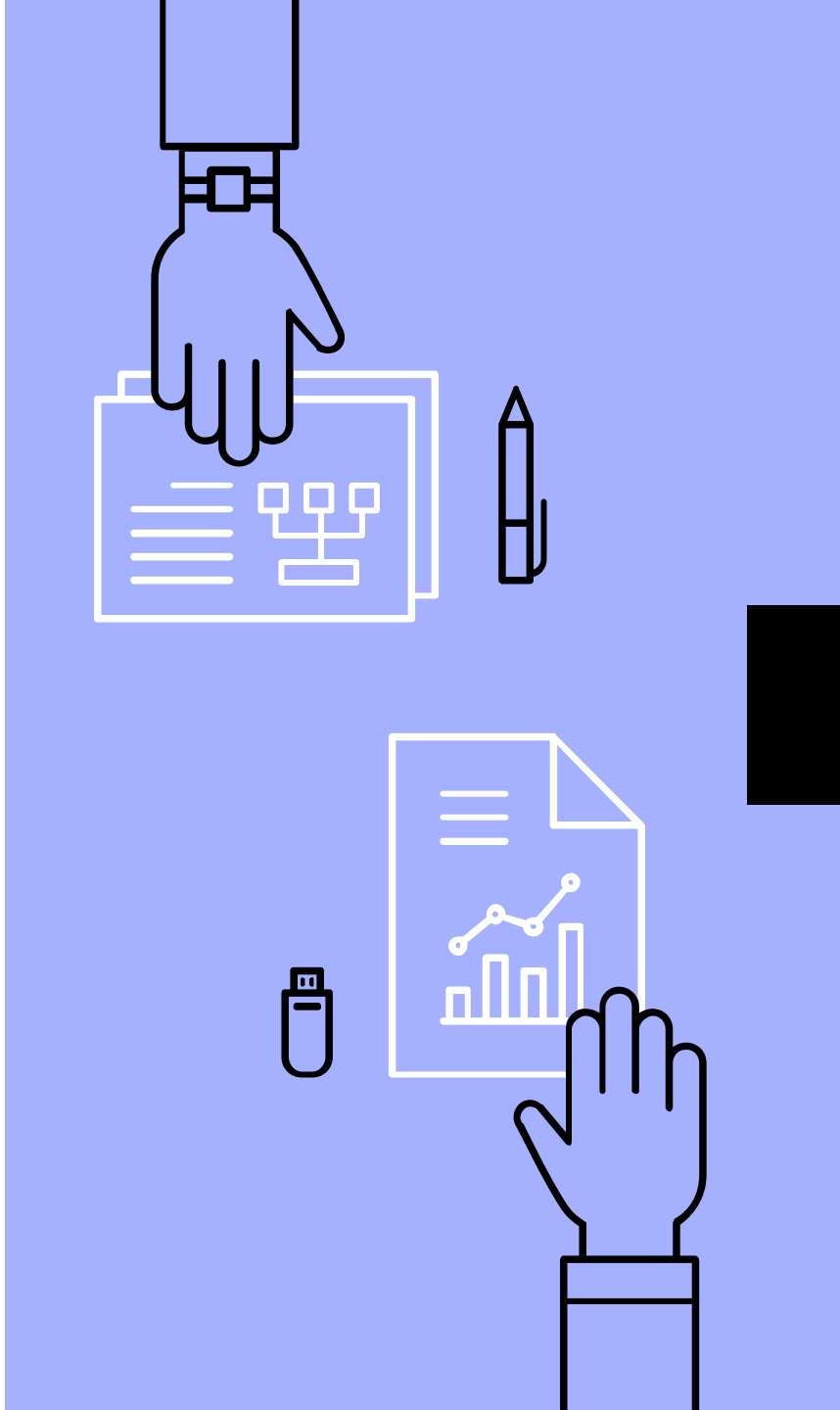
54	26	93	17	77	31	44	55	20	Assume 54 is a sorted list of 1 item
26	54	93	17	77	31	44	55	20	inserted 26
26	54	93	17	77	31	44	55	20	inserted 93
17	26	54	93	77	31	44	55	20	inserted 17
17	26	54	77	93	31	44	55	20	inserted 77
17	26	31	54	77	93	44	55	20	inserted 31
17	26	31	44	54	77	93	55	20	inserted 44
17	26	31	44	54	55	77	93	20	inserted 55
17	20	26	31	44	54	55	77	93	inserted 20

Ordenação Merge Sort

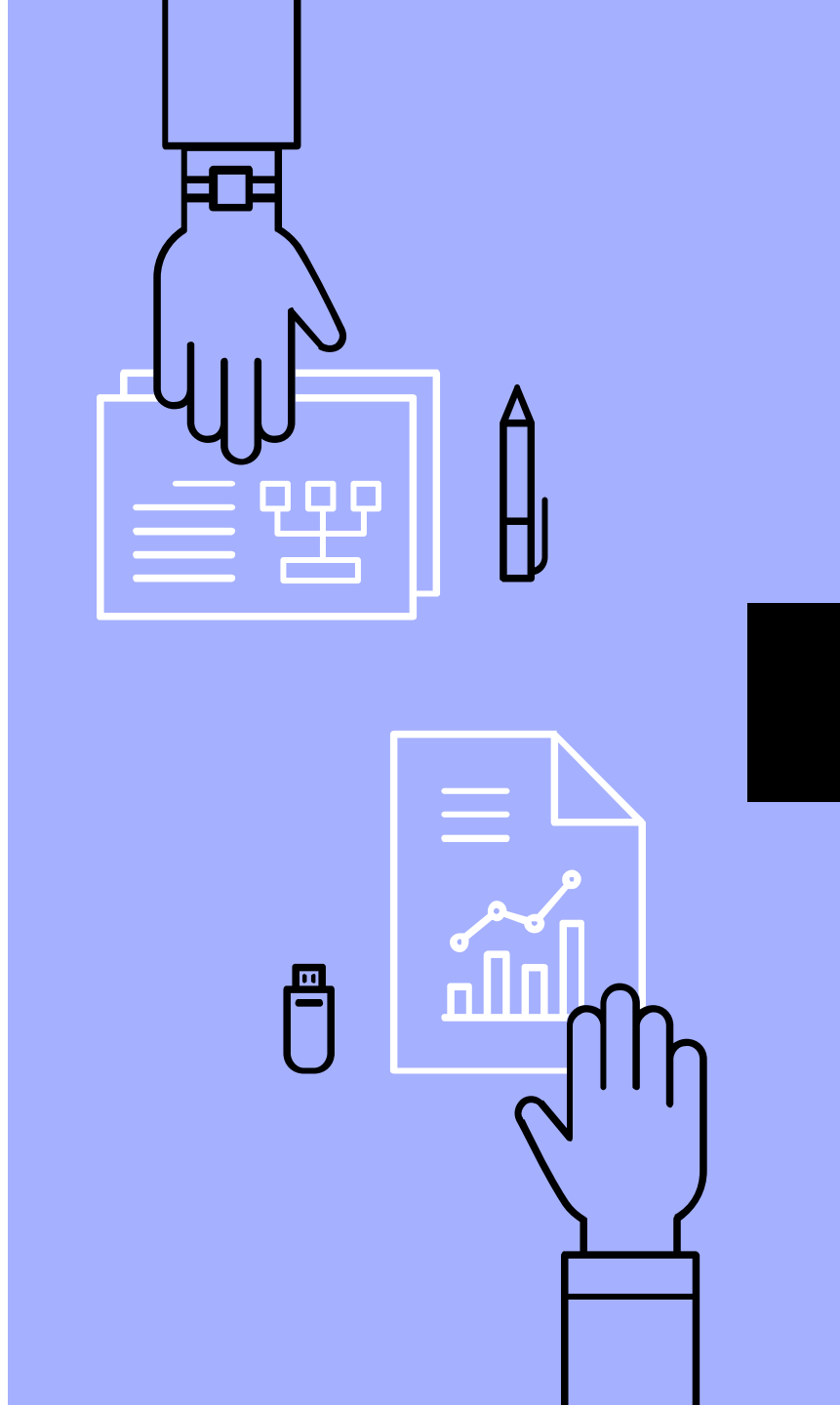
- Voltamos para o uso de uma estratégia de dividir e conquistar como uma forma de melhorar o desempenho dos algoritmos de classificação.
- O primeiro algoritmo que estudaremos é o **merge sort**.



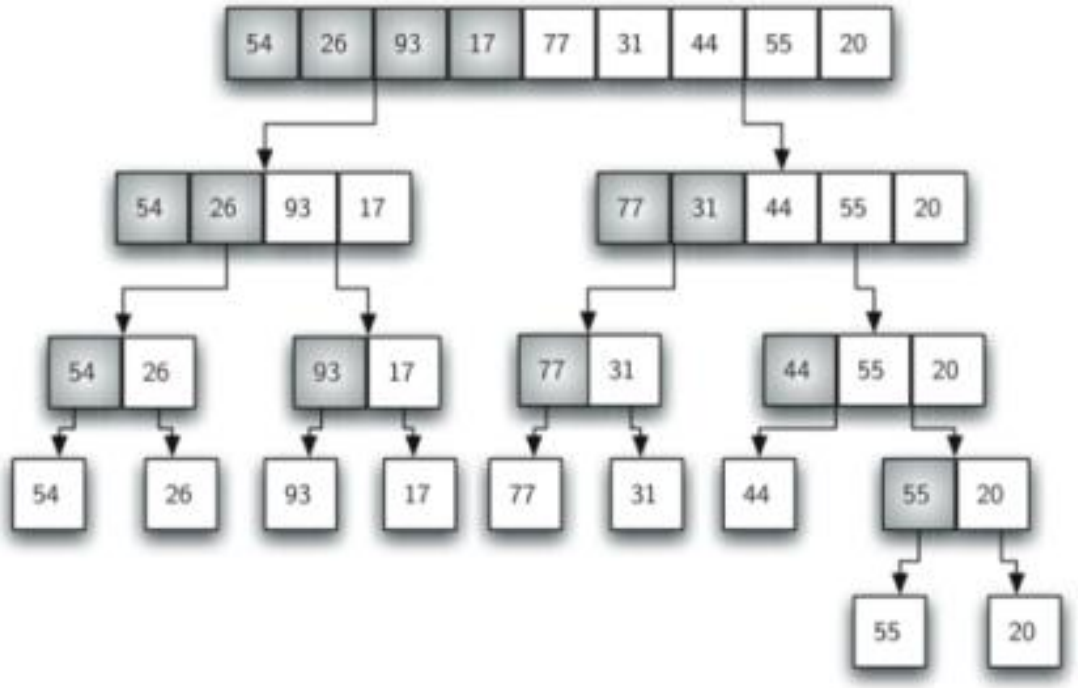
- A classificação por mesclagem é um algoritmo recursivo que divide continuamente uma lista pela metade.
- Se a lista estiver vazia ou tiver um item, ela será classificada por definição (o caso base).
- Se a lista tiver mais de um item, dividimos a lista e invocamos recursivamente uma classificação por mesclagem em ambas as metades.
- Depois que as duas metades são classificadas, a operação fundamental, chamada de mesclagem, é executada.
- Mesclar é o processo de pegar duas listas menores classificadas e combiná-las em uma única lista nova e classificada.

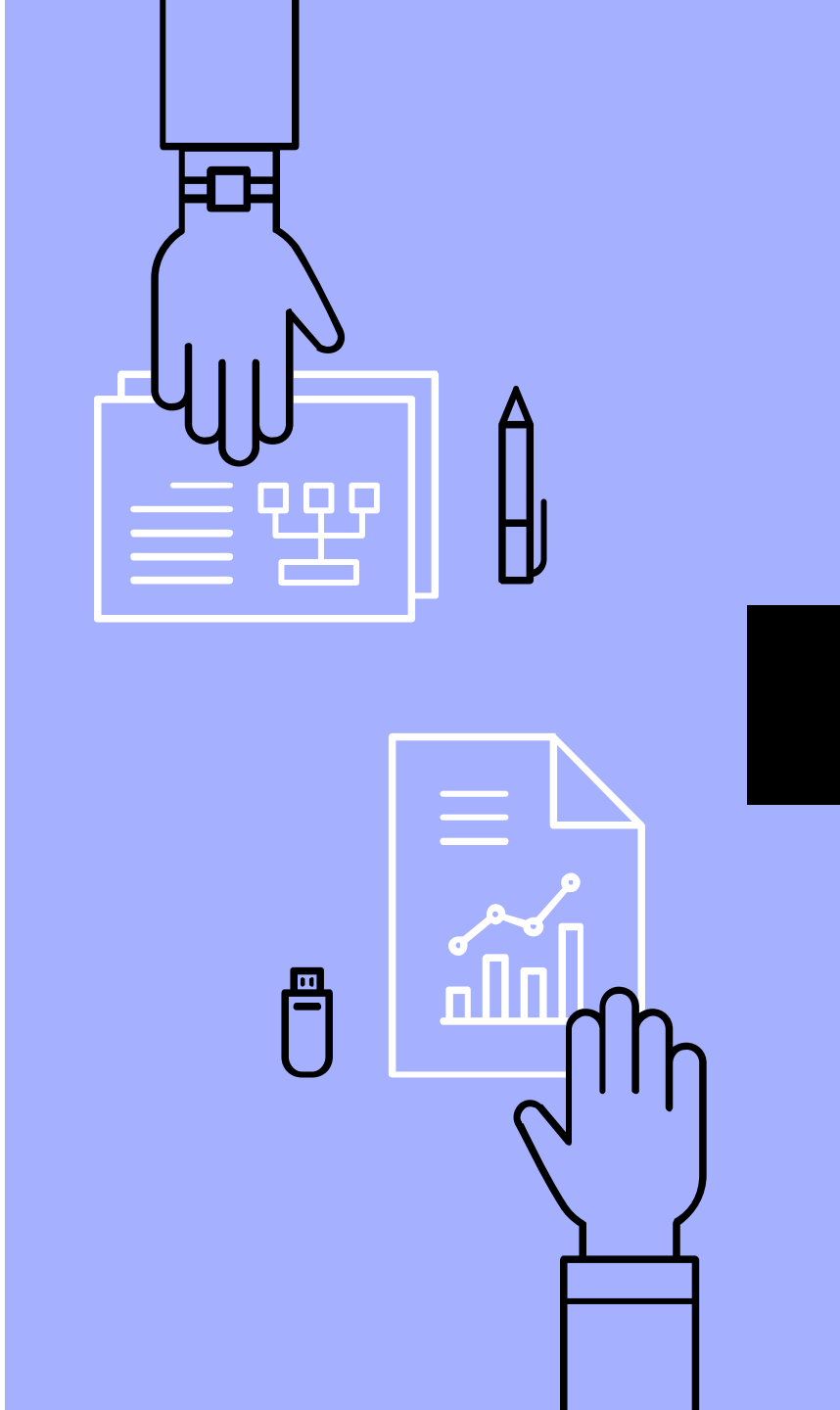


- A função **merge_sort** começa fazendo a pergunta do caso base.
- Se o comprimento da lista for menor ou igual a um, então já temos uma lista classificada e não é necessário mais processamento.
- Se, por outro lado, o comprimento for maior do que um, usamos a operação de fatia do Python para extrair as metades esquerda e direita.
- É importante observar que a lista pode não ter um número par de itens. Isso não importa, pois os comprimentos serão diferentes em no máximo um.

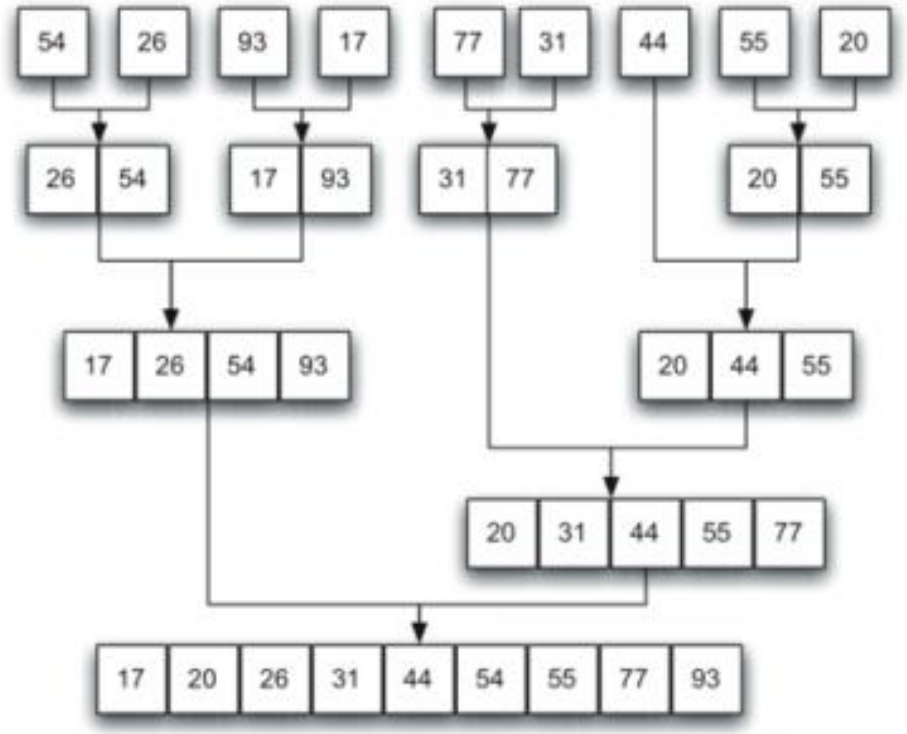


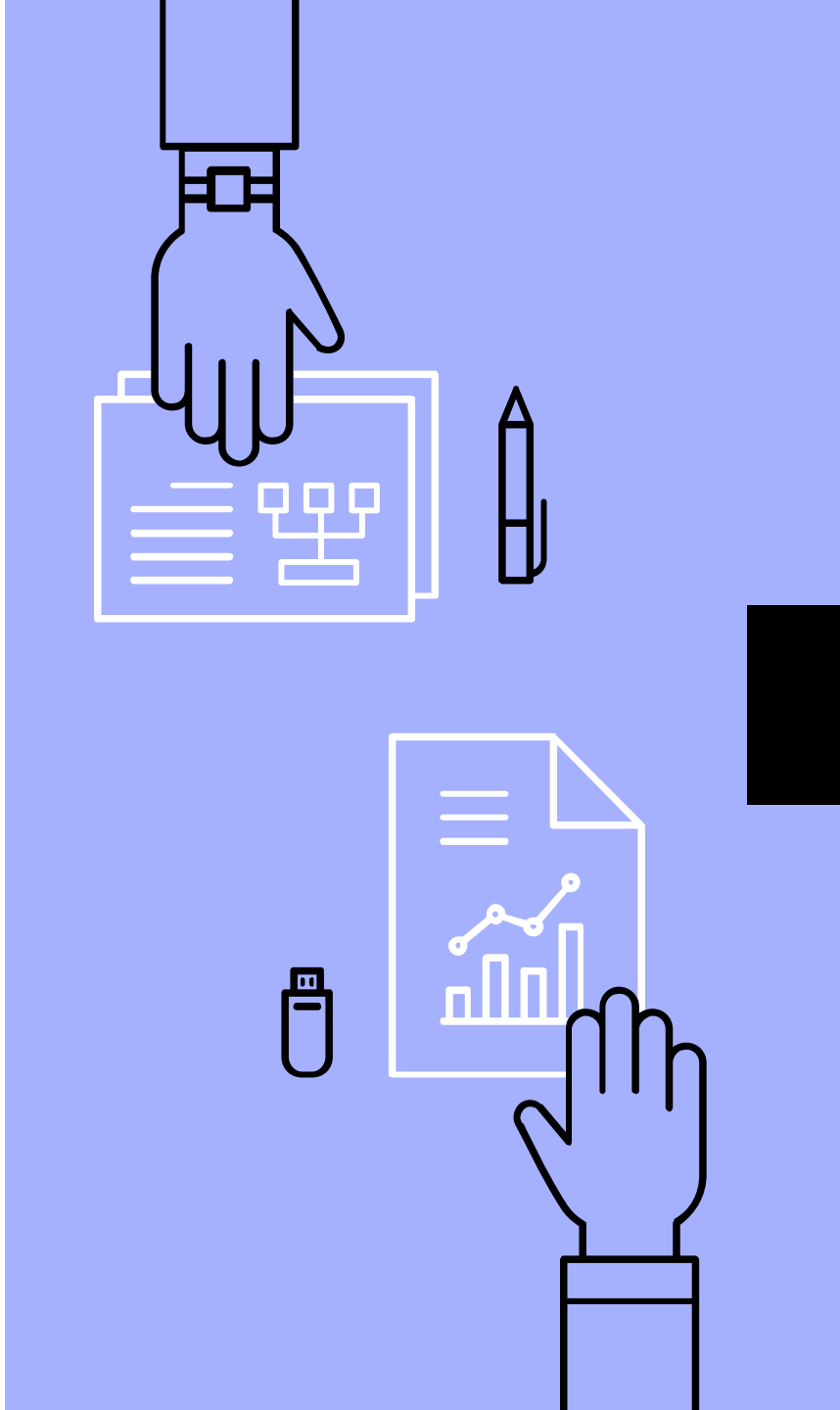
splitting


54 26 93 17 77 31 44 55 20
54 26 93 17
77 31 44 55 20
54 26
93 17
77 31
44 55 20
54
26
93
17
77
31
44
55 20
55
20

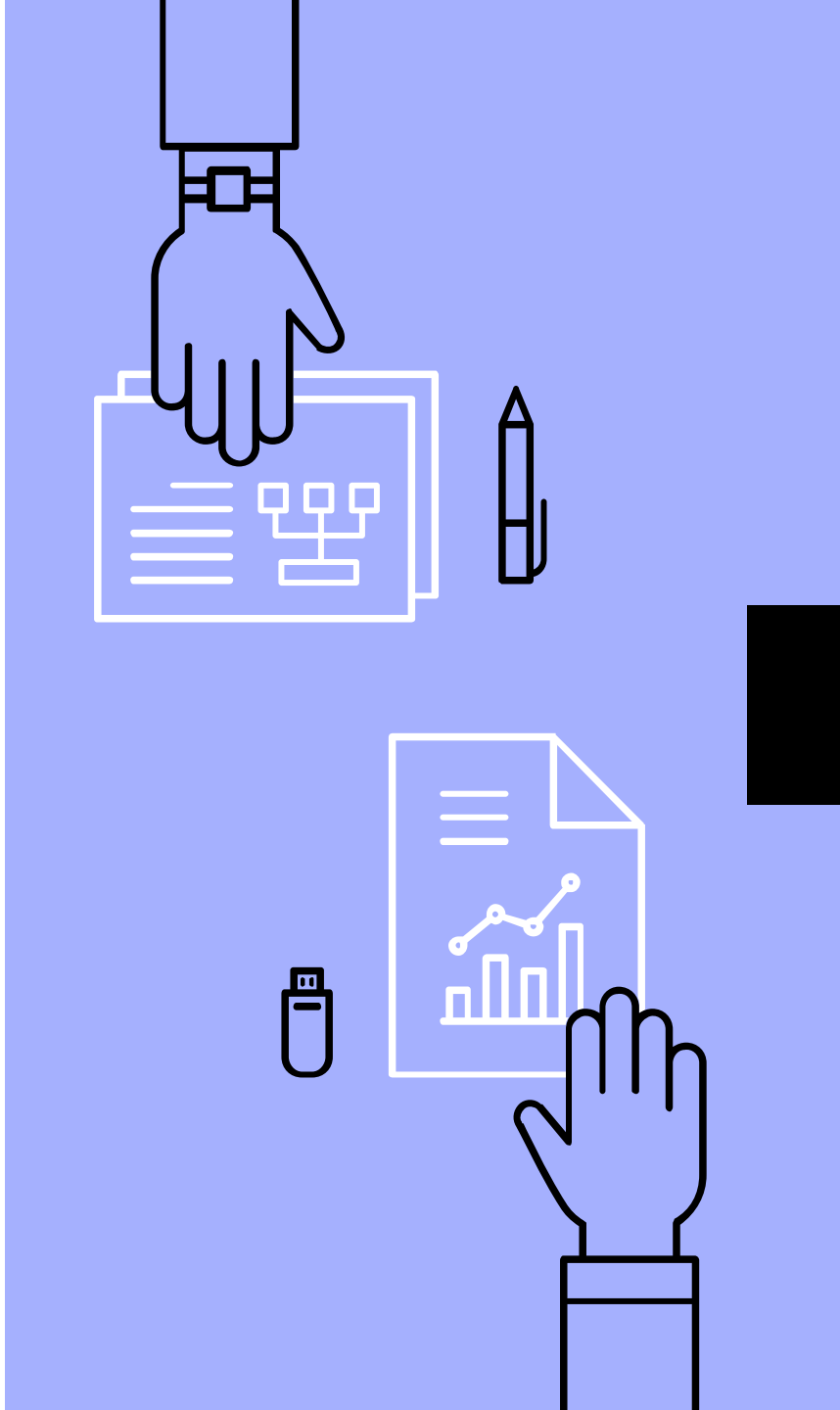


merging

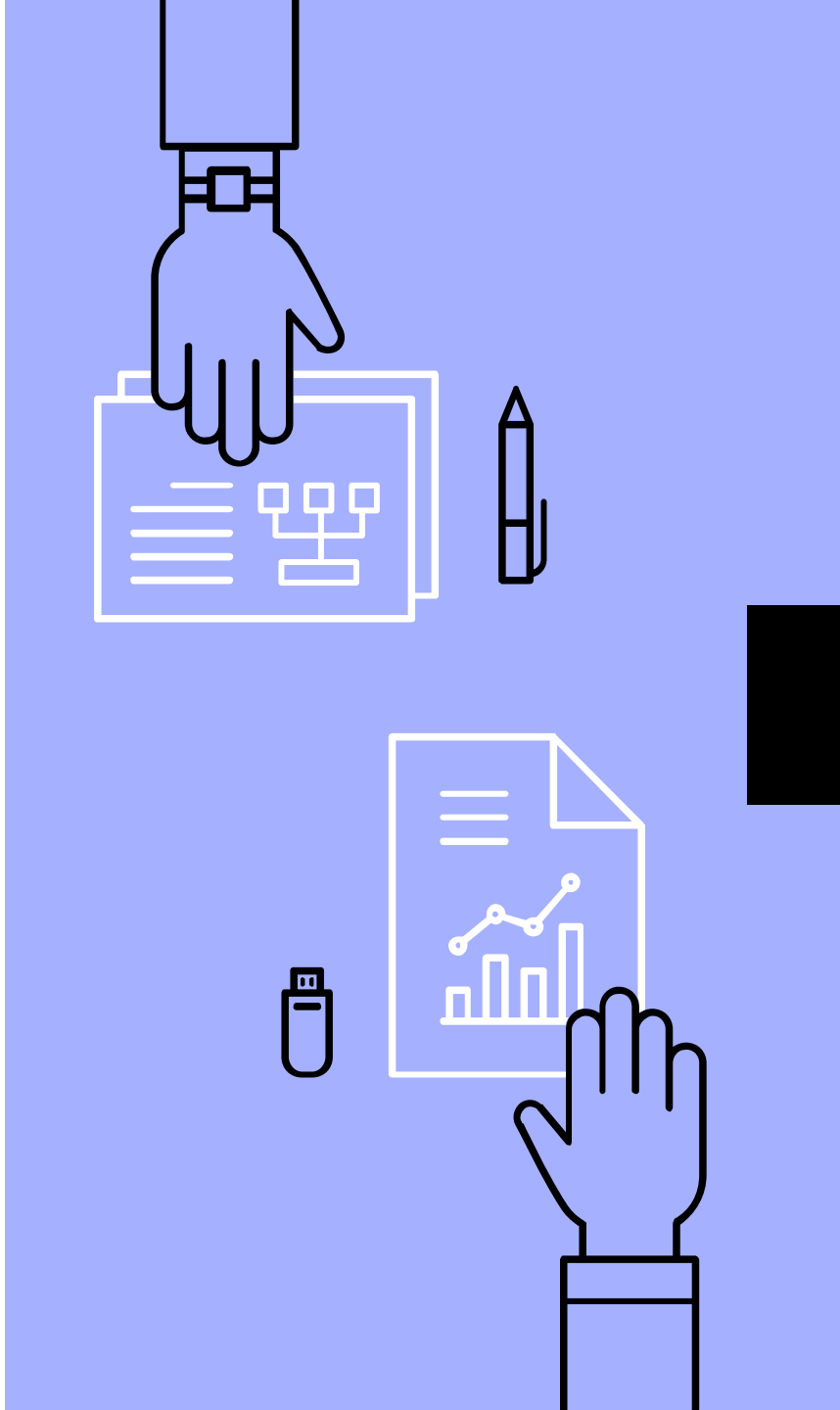

54 26 93 17 77 31 44 55 20
26 54 17 93 31 77 20 55
17 26 54 93 20 44 55
20 31 44 55 77
17 20 26 31 44 54 55 77 93



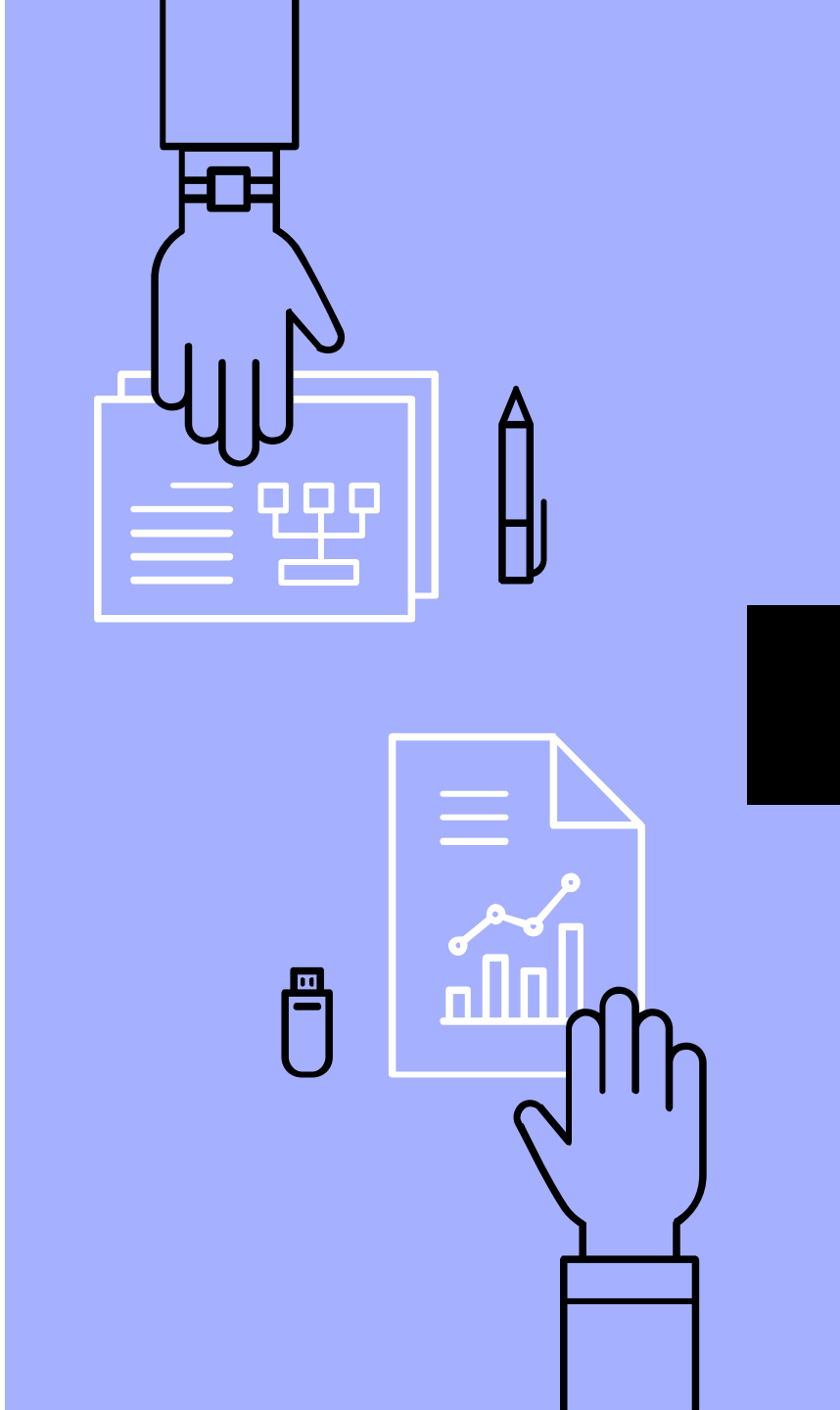
- Para analisar a função **merge_sort**, precisamos considerar os dois processos distintos que compõem sua implementação.
- Primeiro, a lista é dividida em duas metades. Calculamos (em uma busca binária) que podemos dividir uma lista ao meio log n vezes, onde n é o comprimento da lista.
- O segundo processo é a fusão. Cada item da lista será processado e colocado na lista classificada.
- Portanto, a operação de fusão que resulta em uma lista de tamanho n requer n operações.
- O resultado dessa análise é que log n se divide, cada um custando n para um total de $n\log n$ operações.



- Uma classificação por mesclagem é um algoritmo $O(n \log n)$.
- Lembre-se de que o operador de fatiamento é $O(k)$ onde k é o tamanho da fatia.
- Para garantir que **merge_sort** será $O(n \log n)$, precisaremos remover o operador de fatia.
- Novamente, isso é possível se simplesmente passarmos os índices inicial e final junto com a lista quando fizermos a chamada recursiva.

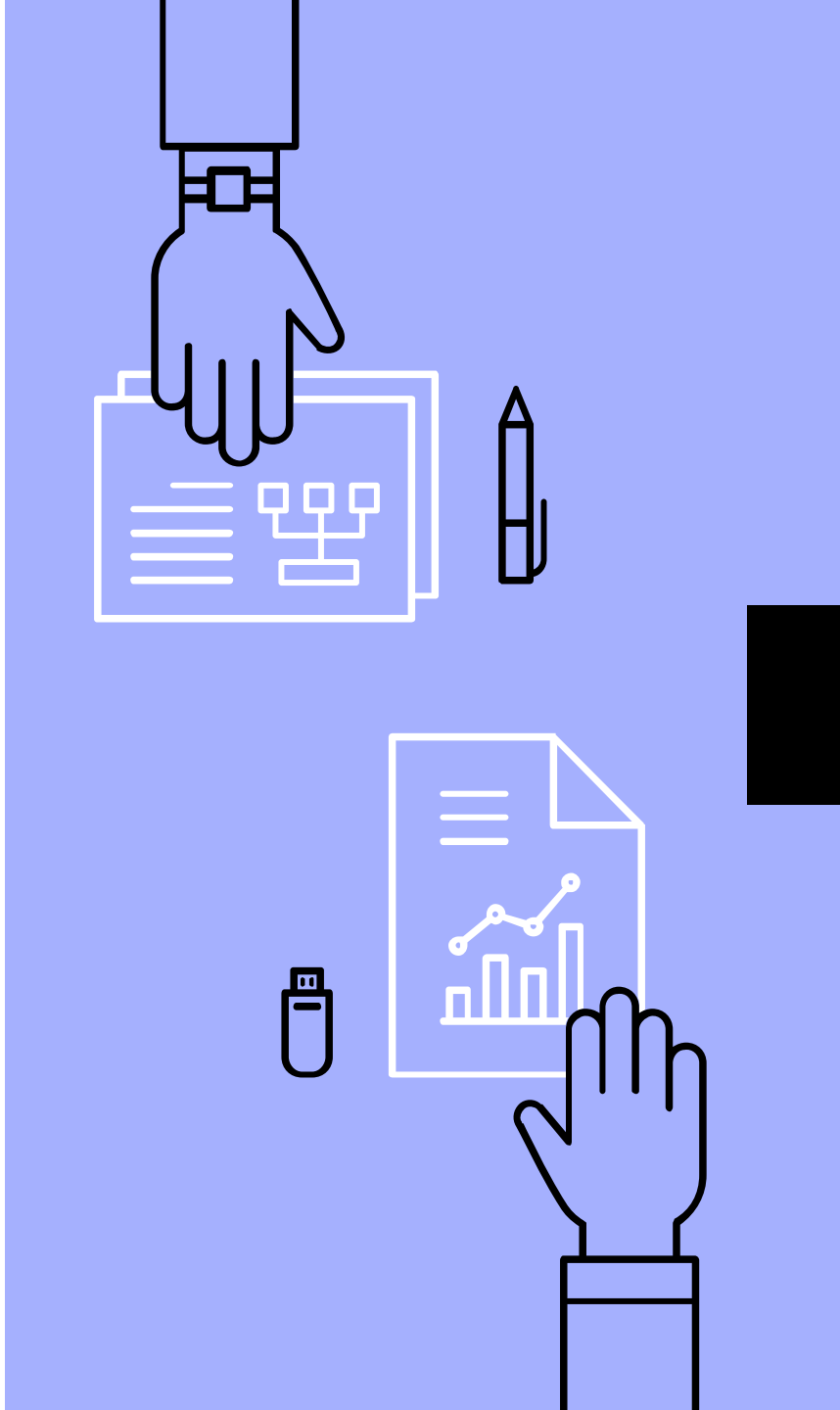


- É importante notar que a função **merge_sort** requer espaço extra para conter as duas metades à medida que são extraídas com as operações de fatiamento.
- Esse espaço adicional pode ser um fator crítico se a lista for grande e pode tornar essa classificação problemática ao trabalhar com grandes conjuntos de dados.

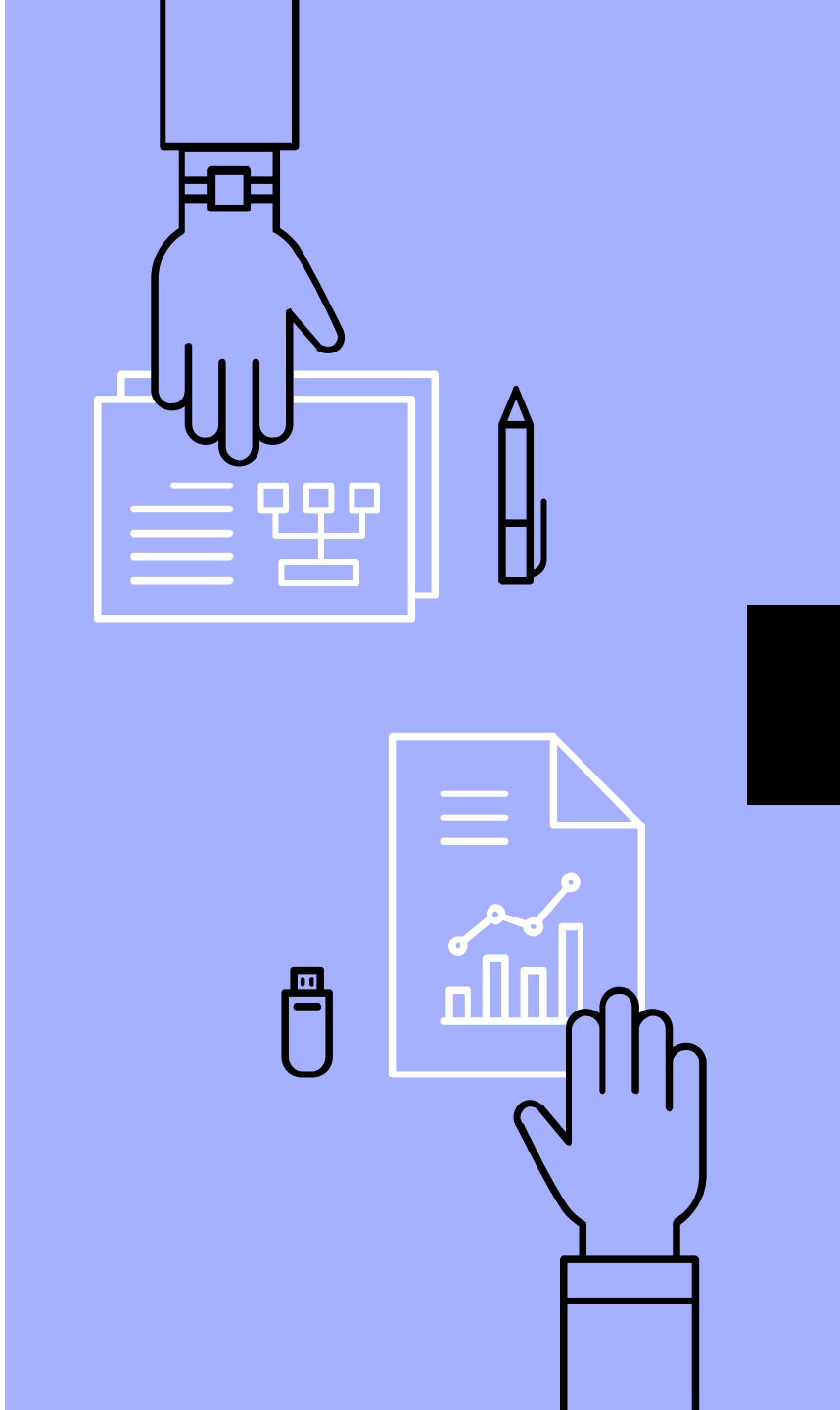


Ordenação Quick Sort

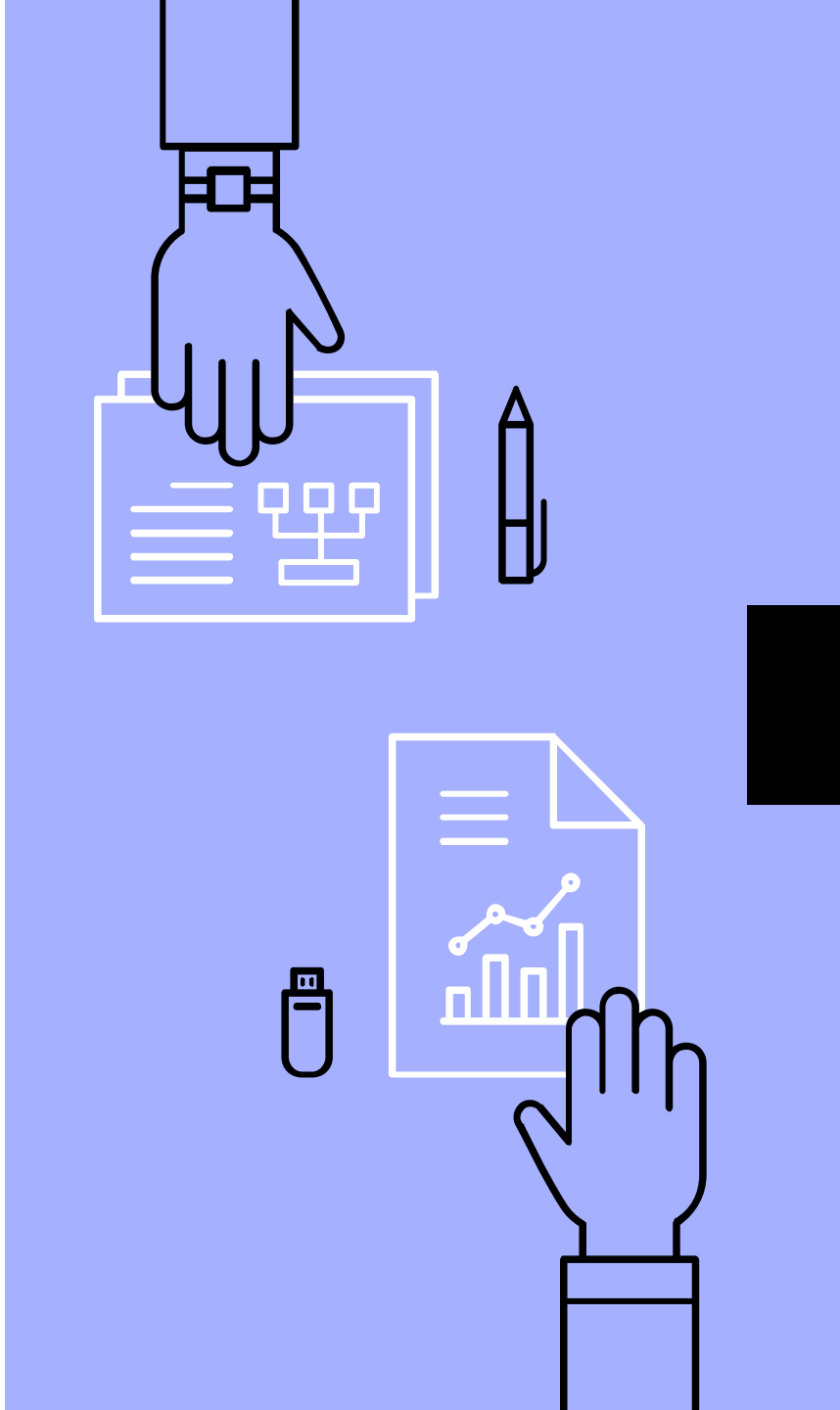
- A classificação rápida usa dividir e conquistar para obter as mesmas vantagens da classificação por mesclagem, embora não use armazenamento adicional.
- Como compensação, no entanto, é possível que a lista não seja dividida pela metade. Quando isso acontecer, veremos que o desempenho diminui.



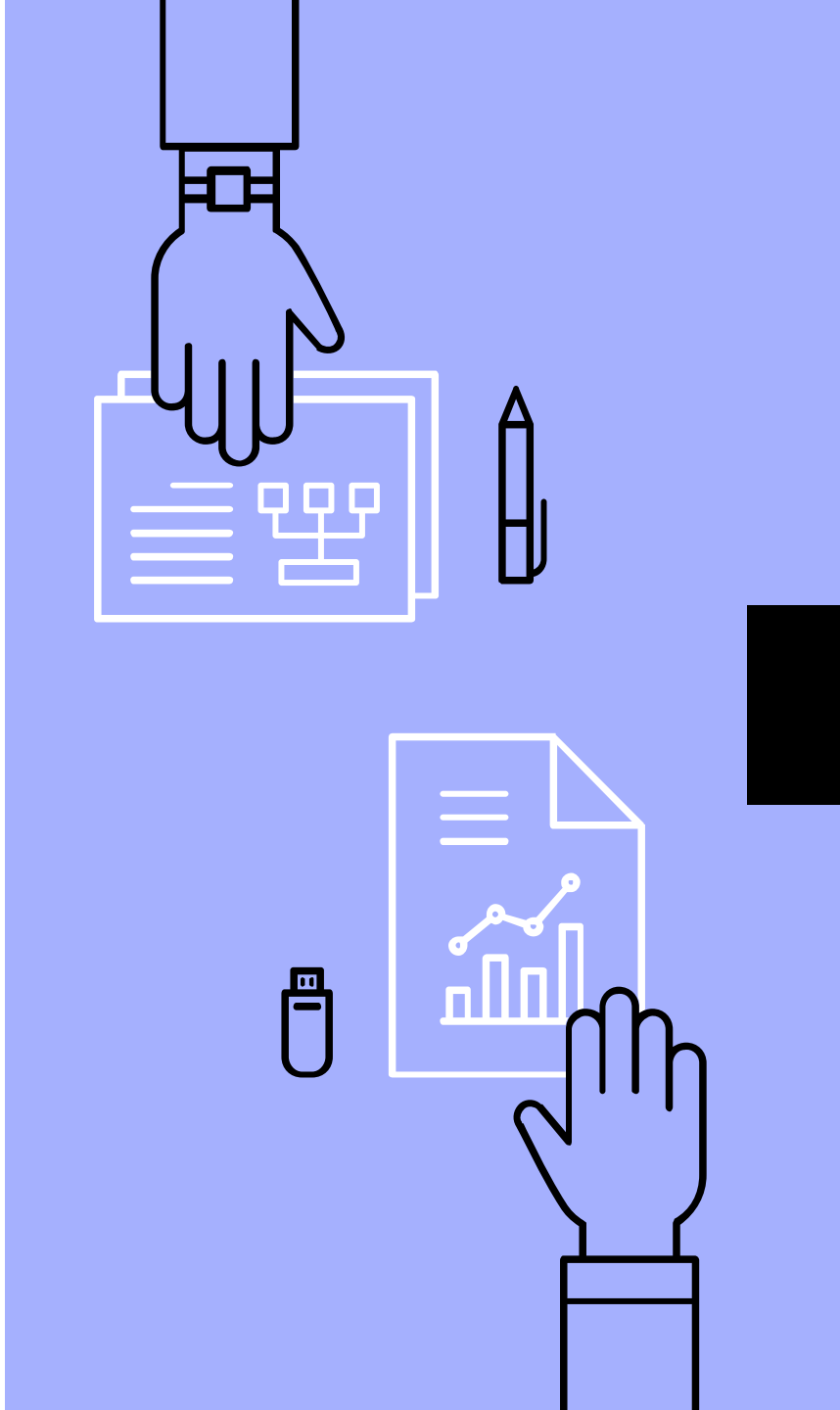
- Uma classificação rápida primeiro seleciona um valor, que é chamado de valor pivô.
- Embora existam muitas maneiras diferentes de escolher o valor pivô, usaremos simplesmente o primeiro item da lista.
- A função do valor pivô é ajudar na divisão da lista.
- A posição real onde o valor pivô pertence na lista classificada final, comumente chamada de ponto de divisão, será usada para dividir a lista para chamadas subsequentes para a classificação rápida.



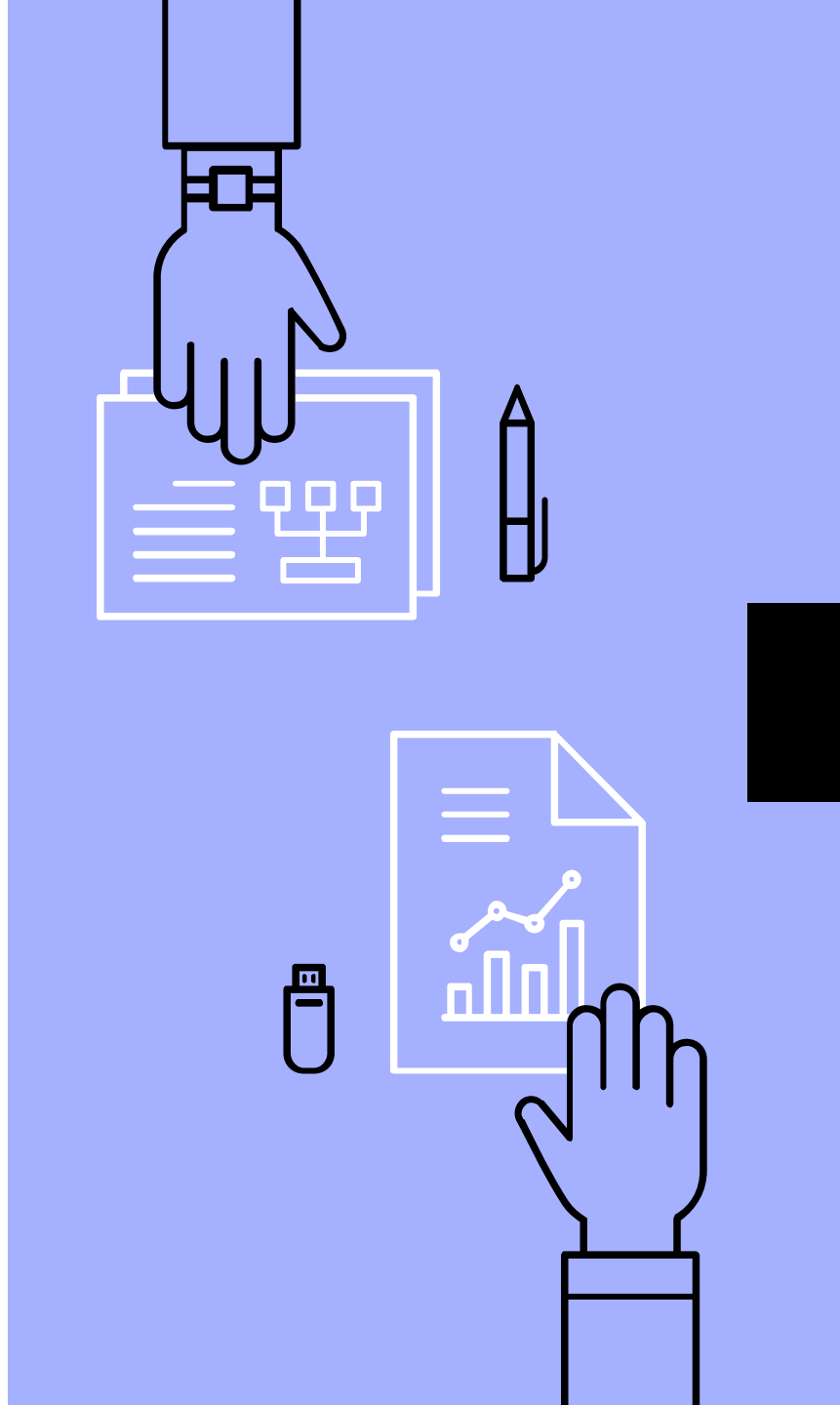
- O particionamento começa localizando dois marcadores de posição – chamados de **left_mark** e **right_mark** - no início e no final dos itens restantes na lista.
- O objetivo do processo de partição é mover os itens que estão do lado errado em relação ao valor de pivô, ao mesmo tempo que convergem no ponto de divisão.



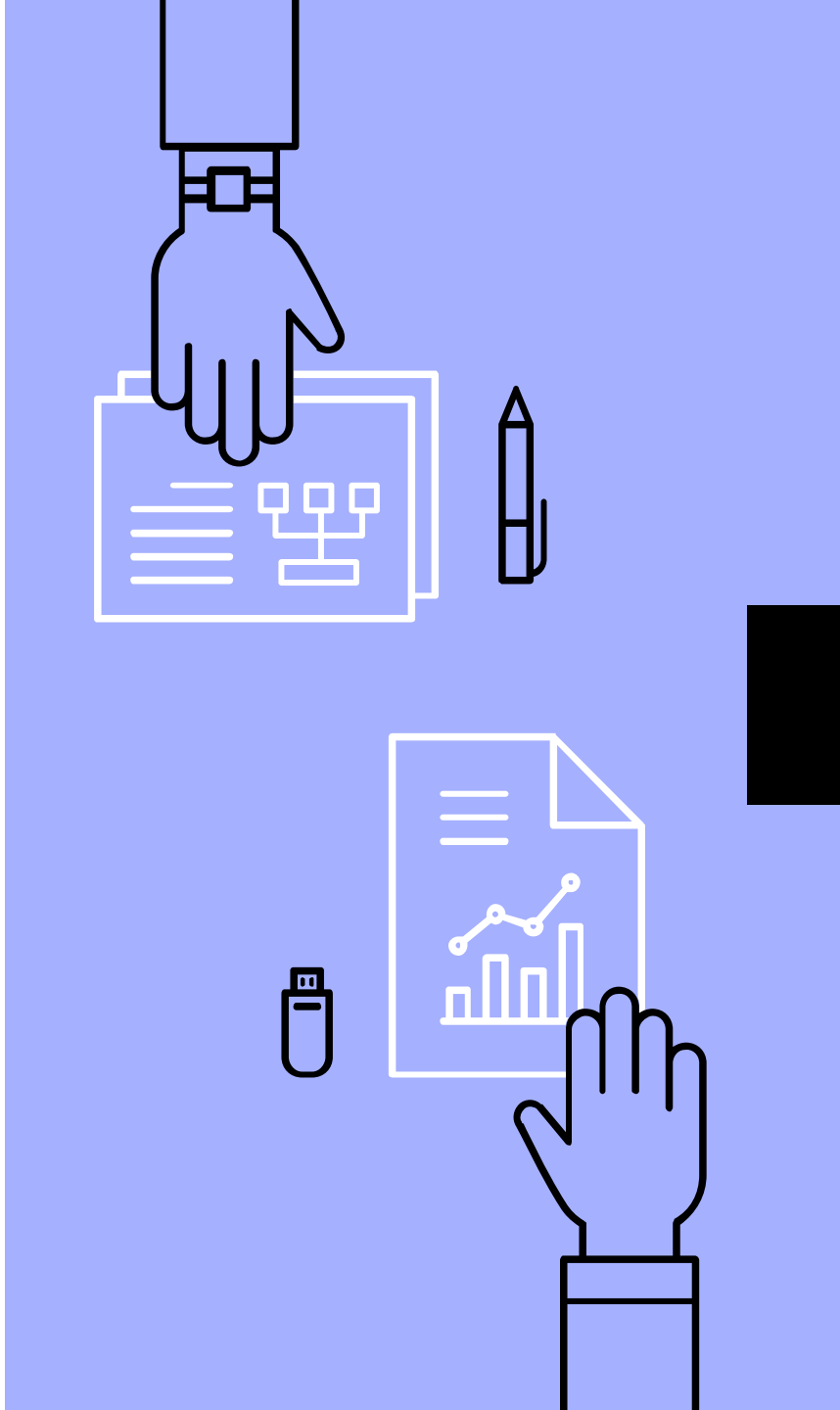
- Começamos incrementando **left_mark** até localizarmos um valor maior que o valor pivô.
- Em seguida, decrementamos **right_mark** até encontrarmos um valor menor que o valor pivô.
- Neste ponto, descobrimos dois itens que estão fora do lugar em relação ao eventual ponto de divisão. Agora podemos trocar esses dois itens e repetir o processo novamente.



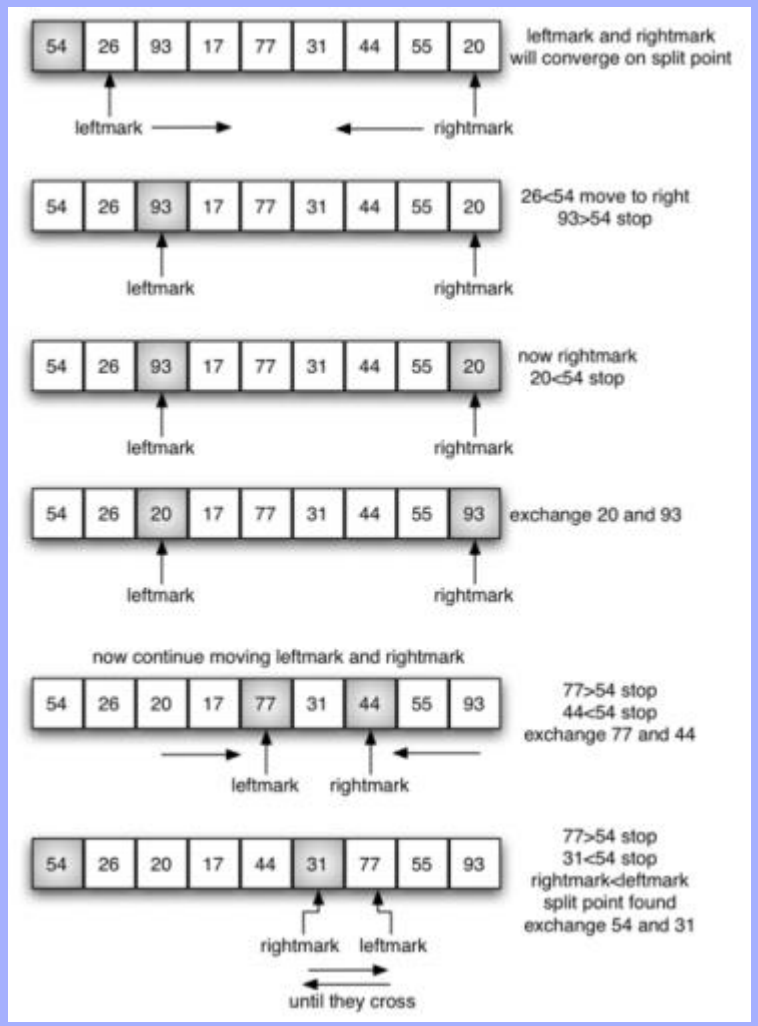
- No ponto em que **right_mark** se torna menor que **left_mark**, paramos. A posição de **right_mark** agora é o ponto de divisão.
- O valor do pivô pode ser trocado pelo conteúdo do ponto de divisão e o valor do pivô agora está no lugar. Além disso, todos os itens à esquerda do ponto de divisão são menores que o valor de pivô e todos os itens à direita do ponto de divisão são maiores que o valor de pivô.
- A lista agora pode ser dividida no ponto de divisão e a classificação rápida pode ser chamada recursivamente nas duas metades.

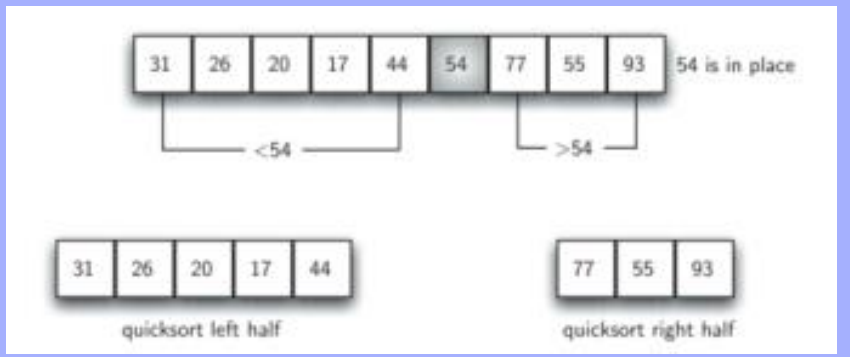


- A função **quick_sort** invoca uma função recursiva, **quick_sort_helper**. Esta começa com o mesmo caso base da classificação por mesclagem.
- Se o comprimento da lista for menor ou igual a um, ela já está classificada. Se for maior, ele pode ser particionado e classificado recursivamente.

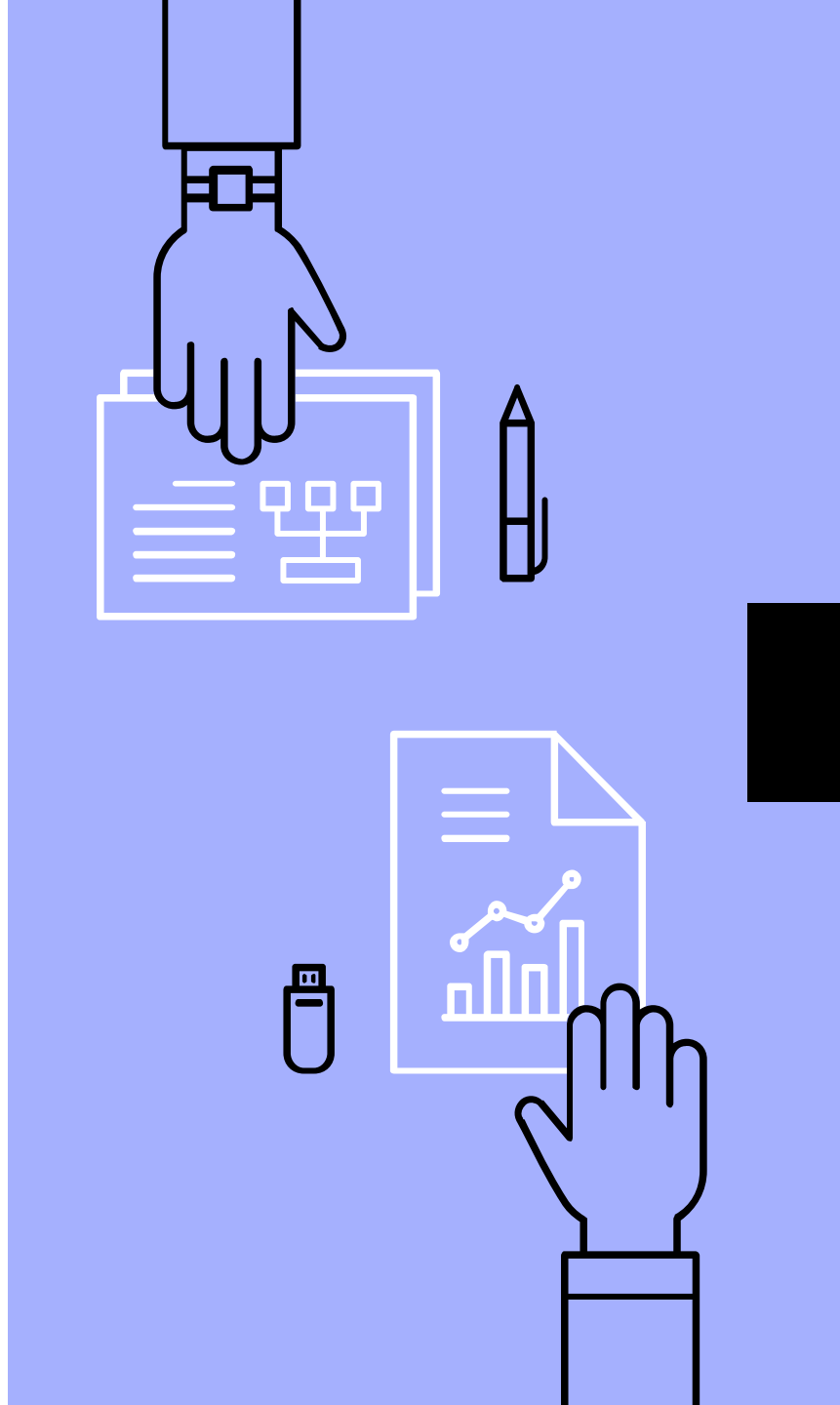



54 26 93 17 77 31 44 55 20
54 will be the first pivot value

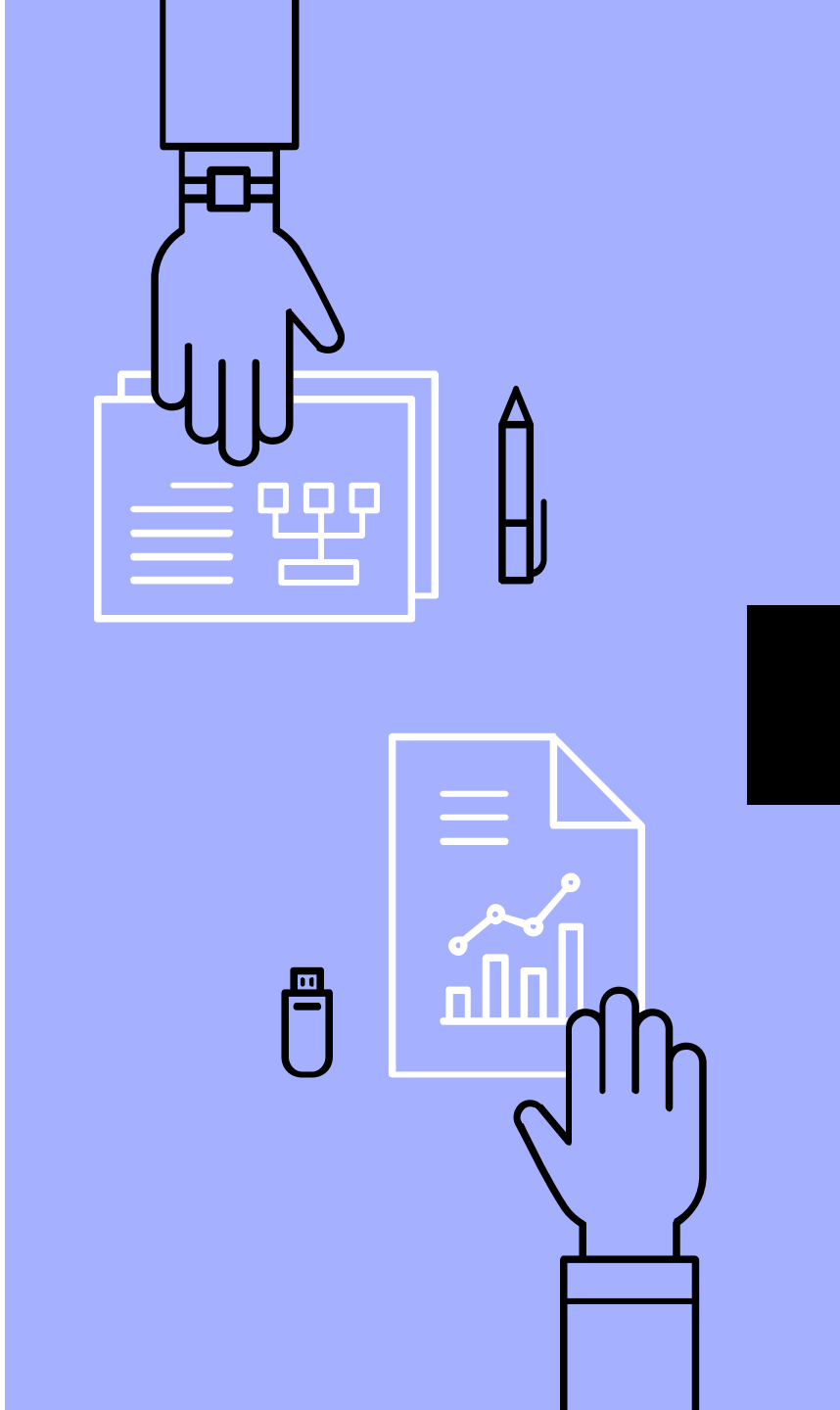

54 26 93 17 77 31 44 55 20
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right
93>54 stop
now rightmark
20<54 stop
exchange 20 and 93
now continue moving leftmark and rightmark
77>54 stop
44<54 stop
exchange 77 and 44
77>54 stop
31<54 stop
rightmark<leftmark
split point found
exchange 54 and 31
until they cross


31 26 20 17 44 54 77 55 93
54 is in place
<54
>54
31 26 20 17 44
quicksort left half
77 55 93
quicksort right half

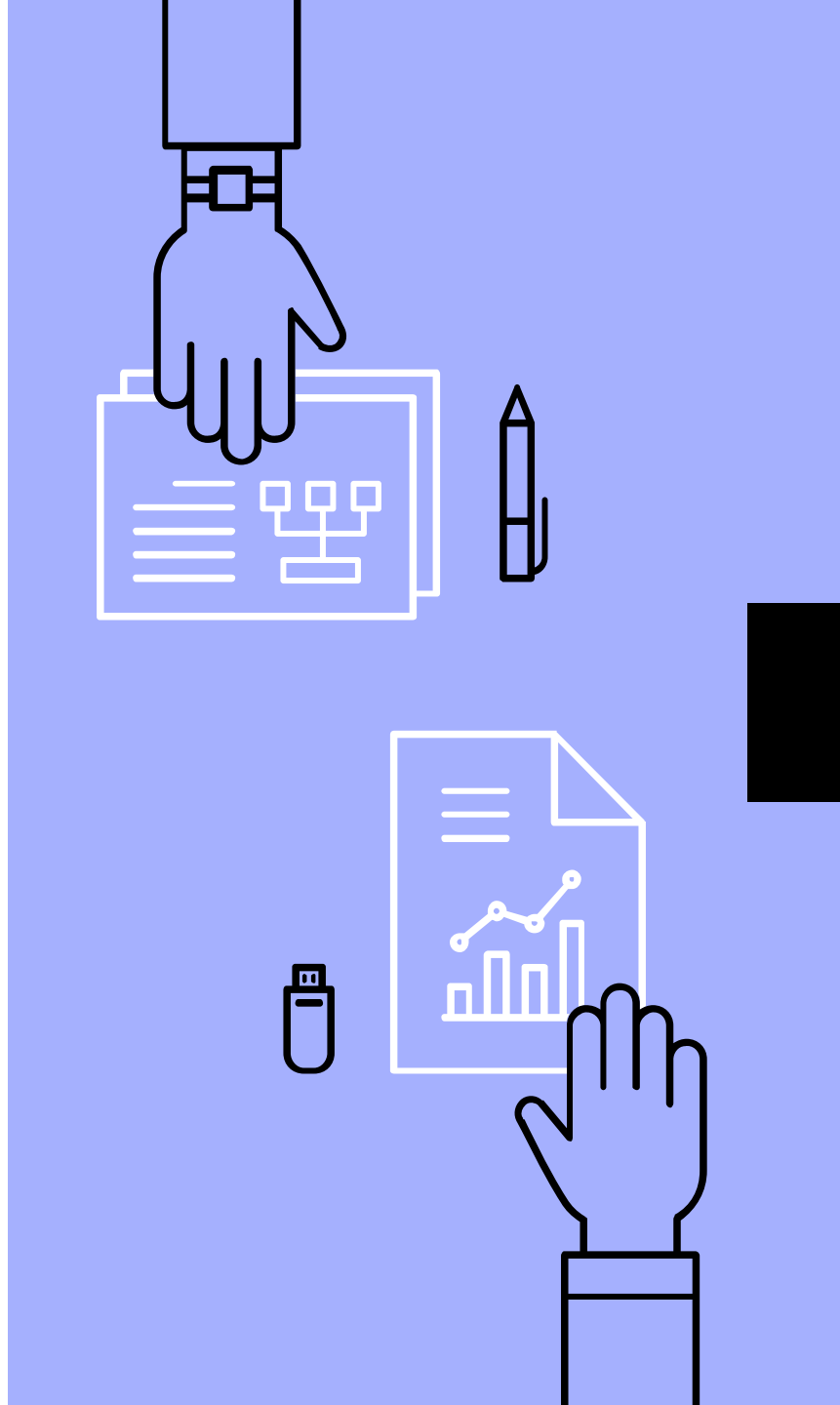
- Para analisar a função **quick_sort**, observe que para uma lista de comprimento n, se a partição sempre ocorrer no meio da lista, haverá novamente $\log n$ divisões.
- Para encontrar o ponto de divisão, cada um dos n itens precisa ser verificado em relação ao valor de pivô. O resultado é $n \log n$.
- Além disso, não há necessidade de memória adicional como no processo de classificação por mesclagem.



- Infelizmente, no pior caso, os pontos de divisão podem não estar no meio e podem ser muito inclinados para a esquerda ou para a direita, deixando uma divisão muito desigual.
- Nesse caso, classificar uma lista de n itens divide em classificar uma lista de 0 itens e uma lista de n - 1 itens.
- Em seguida, classificar uma lista de n - 1 divide em uma lista de tamanho 0 e uma lista de tamanho n - 2 e assim por diante.
- O resultado é uma classificação $O\left(n^2\right)$ com toda a sobrecarga que a recursão requer.



- Existem maneiras diferentes de escolher o valor do pivô.
- Em particular, podemos tentar aliviar parte do potencial de uma divisão desigual usando uma técnica chamada mediana de três.
- Para escolher o valor pivô, consideraremos o primeiro, o meio e o último elemento da lista. Agora escolha o valor mediano e use-o para o valor de pivô (é claro, esse foi o valor de pivô que usamos originalmente).



- A ideia é que no caso em que o primeiro item da lista não pertença ao meio da lista, a mediana de três escolherá um valor “intermediário” melhor.
- Isso será particularmente útil quando a lista original estiver um tanto ordenada para começar.