Anotações da Aula de Algoritmos

Nesta aula, vimos sobre...

- Revisão ao Módulo Anterior
- Busca
- Big O
- Pesquisa linear, pesquisa binária
- Realizando a busca em código
- Structs
- Ordenação
- Selection sort
- Bubble sort
- Recursão
- Merge sort

Quero compartilhar meu aprendizado e/ou minha dúvida...

Ir para o Discord

Recomendamos que você leia as anotações da aula, isso pode te ajudar!

Módulo Anterior

Aprendemos sobre ferramentas para resolver problemas, ou bugs, em nosso código. Em particular, descobrimos como usar um depurador, uma ferramenta que nos permite percorrer lentamente nosso código e examinar os valores na memória enquanto nosso programa está em execução.

Outra ferramenta poderosa, embora menos técnica, é a duck debugging (”depuração de pato de borracha”), onde tentamos explicar o que estamos tentando fazer com um pato de borracha (ou algum outro objeto) e, no processo, encontramos o problema (e esperamos, a solução!) sozinhos.

Olhamos para a memória, visualizando bytes em uma grade e armazenando valores em cada caixa, ou byte, com variáveis e matrizes.

Busca/Searching

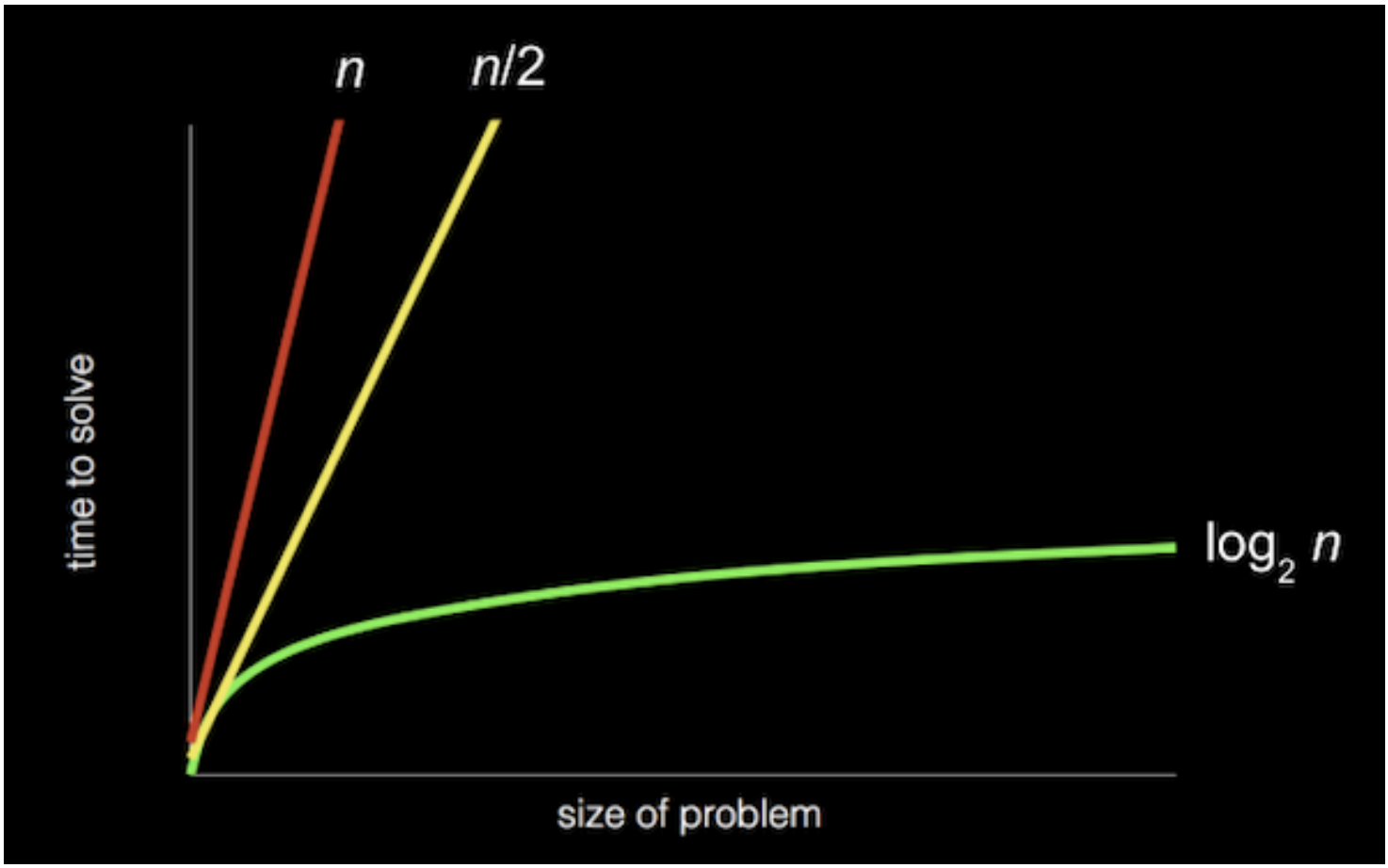
Acontece que, com matrizes, um computador não pode olhar para todos os elementos de uma vez. Em vez disso, um computador só pode olhar para eles um de cada vez, embora a ordem possa ser arbitrária. (Lembre-se de que, na semana 0, David só conseguia olhar uma página de cada vez na lista telefônica, quer folheasse em ordem ou de maneira mais sofisticada.)

Searching ("busca") é como resolvemos o problema de encontrar um valor específico. Um caso simples pode ter como input algum array de valores, e a saída pode ser simplesmente um bool, esteja ou não um determinado valor no array.

Hoje veremos algoritmos de pesquisa. Para discuti-los, consideraremos o **tempo de execução**, ou quanto tempo um algoritmo leva para ser executado dado algum tamanho de input.

Big *O*

Na semana 0, vimos diferentes tipos de algoritmos e seus tempos de execução:



Lembre-se de que a linha vermelha está pesquisando linearmente, uma página por vez; a linha amarela está pesquisando duas páginas por vez; e a linha verde está pesquisando logaritmicamente, dividindo o problema pela metade a cada vez.

E esses tempos de execução são para o pior caso, ou o caso em que o valor leva mais tempo para ser encontrado (na última página, ao contrário da primeira página).

A maneira mais formal de descrever cada um desses tempos de execução é com a notação ***Big O("grande O")***, que podemos pensar como "na ordem de". Por exemplo, se nosso algoritmo for uma pesquisa linear, ele levará aproximadamente O (n) etapas, lidas como "grande O de n" ou "na ordem de n". Na verdade, mesmo um algoritmo que analisa dois itens por vez e executa n/2 etapas tem O (n). Isso ocorre porque, à medida que n fica cada vez maior, apenas o fator dominante, ou o maior termo, n, importa. No gráfico acima, se afastássemos o zoom e mudássemos as unidades em nossos eixos, veríamos as linhas vermelha e amarela ficando muito próximas.

Um tempo de execução logarítmico é O (log n), não importa qual seja a base, pois isso é apenas uma aproximação do que acontece fundamentalmente com o tempo de execução se n for muito grande.

Existem alguns tempos de execução comuns:

- *O(n^2)*
- *O (n log n)*
- *O(n)*
  - *(pesquisando uma página por vez, em ordem)*
- *O (log n)*
  - *(dividindo a lista telefônica pela metade a cada vez)*
- *O (1)*
  - Um algoritmo que executa um número **constante** de etapas, independentemente do tamanho do problema.

Os cientistas da computação também podem usar a notação *big **Ω**, grande Omega,* que é o limite inferior do número de etapas de nosso algoritmo. *Big O* é o limite superior do número de etapas ou o pior caso.

E temos um conjunto semelhante de tempos de execução mais comuns para big Ω:

- *Ω (n^2)*
- *Ω (nlog n)*
- *Ω (n)*
- *Ω (log n)*
- *Ω (1)*
  - (pesquisar em uma lista telefônica, pois podemos encontrar nosso nome na primeira página que verificarmos)

Pesquisa linear, pesquisa binária

No palco, temos algumas portas de mentira, com números escondidos atrás delas. Como um computador só pode olhar para um elemento de cada vez em um array, só podemos abrir uma porta de cada vez.

Se quisermos procurar o número zero, por exemplo, teríamos que abrir uma porta por vez, e se não soubéssemos nada sobre os números atrás das portas, o algoritmo mais simples seria ir da esquerda para a direita.

Então, podemos escrever pseudocódigo para **busca linear** com:

```
For i from 0 to n-1
    If number behind i'th door
         Return true
Return false
```

- Rotulamos cada uma das **n** portas de 0 a **n – 1** e verificamos cada uma delas em ordem.
- "Return false" está fora do for loop, já que só queremos fazer isso depois de olharmos por trás de todas as portas.
- O ***big O*** para este algoritmo seria *O(n)*, e o limite inferior, ***big Ω***, seria *Ω(1)*.

Se soubermos que os números atrás das portas estão classificados, podemos começar pelo meio e encontrar nosso valor com mais eficiência.

Para busca binária, nosso algoritmo pode ser semelhante a:

```
If no doors
     Return false
If number behind middle door
     Return true
Else if number < middle door
     Search left half
Else if number > middle door
     Search right half
```

- O limite superior da pesquisa binária é ***O(log n)***, e o limite inferior também ***Ω(1)***, se o número que procuramos estiver no meio, onde começamos.

Com 64 lâmpadas, notamos que a pesquisa linear leva muito mais tempo do que a pesquisa binária, que leva apenas alguns passos.

Desligamos as lâmpadas na frequência de um **hertz**, ou ciclo por segundo, e a velocidade de um processador pode ser medida em gigahertz, ou bilhões de operações por segundo.

Realizando a busca em código

Vamos dar uma olhada em numbers.c :

```
#include <cs50.h>
#include <stdio.h>

int main(void)
```

```
{
    int numbers[] = {4, 6, 8, 2, 7, 5, 0};
    for (int i = 0; i < 7; i++)
    {
         if (numbers[i] == 0)
         {
               printf("Found\n");
               return 0;
         }
    }
    printf("Not found\n");
    return 1;
}
```

- Aqui, inicializamos uma matriz com alguns valores entre chaves e verificamos os itens na matriz, um de cada vez, para ver se eles são iguais a zero (o que estávamos originalmente procurando atrás das portas no palco).
- Se encontrarmos o valor zero, retornamos um código de saída 0 (para indicar sucesso). Caso contrário, após nosso loop for, retornamos 1 (para indicar falha).

Podemos fazer o mesmo com os nomes:

```
· #include · #include · #include · · int main(void) · { · string names[] = {"Bill", "Charlie", "Fred", "Ge
orge", "Ginny", "Percy", "Ron"}; · · for (int i = 0; i < 7; i++) · { · if (strcmp(names[i], "Ron") == 0) ·
{ · printf("Found\n"); · return 0; · } · } · printf("Not found\n"); · return 1; · }
```

- Observe que os **names** é uma matriz ordenada de strings.
- Não podemos comparar strings diretamente em C, uma vez que não são um tipo de dados simples, mas sim um array de muitos caracteres. Felizmente, a biblioteca de **string** tem uma função **strcmp** (“string compare”) que compara strings para nós, um caractere por vez, e retorna 0 se forem iguais.
- Se checarmos apenas strcmp (nomes [i], "Ron") e não strcmp (nomes [i], "Ron") == 0 , então imprimiremos Encontrado mesmo se o nome não for encontrado. Isso ocorre porque strcmp retorna um valor que não é 0 se duas strings não corresponderem, e qualquer valor diferente de zero é equivalente a verdadeiro em uma condição.

Structs

Se quisermos implementar um programa que pesquisa uma lista telefônica, podemos querer um tipo de dado para uma “pessoa”, com seu nome e número de telefone.

Acontece que em C podemos definir nosso próprio tipo de dados, ou *data structure* (“estrutura de dados”), com um **struct** na seguinte sintaxe:

```
typedef struct
{
      string name;
      string number;
}
person;
```

- Usamos **string** para o **number**, pois queremos incluir símbolos e formatação, como sinais de mais ou hifens.
- Nossa estrutura contém outros tipos de dados dentro dela.

Vamos tentar implementar nossa lista telefônica sem structs primeiro:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string names[] = {"Brian", "David"};
    string numbers[] = {"+1-617-495-1000", "+1-949-468-2750"};
    for (int i = 0; i < 2; i++)
    {
          if (strcmp(names[i], "David") == 0)
          {
               printf("Encontrado %s\n", numbers[i]);
               return 0;
          }
    }
    printf("Nao encontrado\n");
    return 1;
}
```

- Precisamos ter cuidado para garantir que firstname nos **names** corresponda ao primeiro número em **numbers** e assim por diante.
- Se o nome em um determinado índice **i** na matriz de **names** corresponder ao que estamos procurando, podemos retornar o número de telefone em **numbers** no mesmo índice.

Com structs, podemos ter um pouco mais de certeza de que não teremos erros humanos em nosso programa:

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

typedef struct
{
     string name;
     string number;
}
person;

int main(void)
{
     person people[2];
     people[0].name = "Brian";
     people[0].number = "+1-617-495-1000";
     people[1].name = "David";
     people[1].number = "+1-949-468-2750";

     for (int i = 0; i < 2; i++)
     {
         if (strcmp(people[i].name, "David") == 0)
         {
              printf("Encontrado %s\n", people[i].number); return 0;
         }
     }
     printf("Não encontrado\n");
     return 1;
}
```

- Criamos uma matriz do tipo struct **person** e a damos o nome **people** (como em int numbers[] , embora pudéssemos nomeá-la arbitrariamente, como qualquer outra variável). Definimos os valores para cada campo, ou variável, dentro de cada estrutura de **person**, usando o operador ponto, **.** .
- Em nosso loop, agora podemos ter mais certeza de que o **number(número)** corresponde ao **name(nome)**, pois eles são da mesma estrutura de **person**.
- Também podemos melhorar o design do nosso programa com uma constante, como const int NUMBER = 10; , e armazenar nossos valores não em nosso código, mas em um arquivo separado ou mesmo em um banco de dados, que veremos em breve.

Em breve, também escreveremos nossos próprios arquivos de cabeçalho com definições para structs, para que possam ser compartilhados entre diferentes arquivos para nosso programa.

Ordenação

Se nossa entrada for uma lista não ordenada de números, existem muitos algoritmos que podemos usar para produzir um output de uma lista classificada, onde todos os elementos estão em ordem.

Com uma lista classificada, podemos usar a pesquisa binária para eficiência, mas pode levar mais tempo para escrever um algoritmo de classificação para essa eficiência, então às vezes encontraremos a compensação de tempo que leva para um ser humano escrever um programa em comparação com o tempo é preciso um computador para executar algum algoritmo. Outras compensações que veremos podem ser tempo e complexidade ou uso de tempo e memória.

Ordenação de seleção

Brian está nos bastidores com um conjunto de números em uma prateleira, em ordem não classificada:

```
6 3 8 5 2 7 4 1
```

Pegando alguns números e colocando-os no lugar certo, Brian os classifica rapidamente.

Indo passo a passo, Brian olha cada número da lista, lembrando-se do menor que vimos até agora. Ele chega ao final e vê que 1 é o menor, e ele sabe que deve ir no início, então ele vai apenas trocá-lo pelo número do início, 6:

```
6 3 8 5 2 7 4 1
-             -
1 3 8 5 2 7 4 6
```

Agora, Brian sabe que pelo menos o primeiro número está no lugar certo, então ele pode procurar o menor número entre os demais e trocá-lo pelo próximo número não ordenado (agora o segundo número):

```
1 3 8 5 2 7 4 6
  -     -
1 2 8 5 3 7 4 6
```

E ele repete isso, trocando o próximo menor, 3, pelo 8:

```
1 2 8 5 3 7 4 6
    -   -
1 2 3 5 8 7 4 6
```

Depois de mais algumas trocas, terminamos com uma lista ordenada.

Esse algoritmo é chamado de **selection sort**, e podemos ser um pouco mais específicos com alguns pseudocódigos:

```
Para i de 0 a n-1
Encontre o menor item entre i-ésimo item e último
Troque o menor item com i-ésimo item
```

- A primeira etapa do loop é procurar o menor item na parte não ordenada da lista, que estará entre o i'ésimo item e o último item, pois sabemos que classificamos até o "i-1'º " item.
- Em seguida, trocamos o menor item pelo i'ésimo item, o que compõe tudo até o item que eu classifiquei.

Vemos uma **visualização online** com animações de como os elementos se movem durante um selection sort.

Para esse algoritmo, examinamos quase todos os **n** elementos para encontrar o menor e fizemos n passos para classificar todos os elementos.

Mais formalmente, podemos usar algumas fórmulas matemáticas para mostrar que o maior fator é de fato $\mathbf{n^2}$. Começamos tendo que olhar para todos os n elementos, então apenas ***n − 1***, então ***n − 2:***

n (n + 1) / 2
$(n^2 + n) / 2$
$n^2 / 2 + n / 2$
$O(n^2)$

- Como $\mathbf{n^2}$ é o maior, ou dominante, fator, podemos dizer que o algoritmo tem um tempo de execução de $\boldsymbol{O(n^2)}$.

Bubble sort

Podemos tentar um algoritmo diferente, em que trocamos pares de números repetidamente, chamado de **bubble sort.**

Brian vai olhar para os dois primeiros números e trocá-los para que fiquem em ordem:

```
6 3 8 5 2 7 4 1
- -
```

```
3 6 8 5 2 7 4 1
```

O próximo par, **6** e **8** , está em ordem, então não precisamos trocá-los.

O próximo par, **8** e **5** , precisa ser trocado:

```
3 6 8 5 2 7 4 1
    - -
3 6 5 8 2 7 4 1
```

Brian continua até chegar ao final da lista:

```
3 6 5 2 8 7 4 1
        - -
3 6 5 2 7 8 4 1
          - -
3 6 5 2 7 4 8 1
            - -
3 6 5 2 7 4 1 8
              -
```

Nossa lista ainda não foi ordenada, mas estamos um pouco mais próximos da solução porque o maior valor, **8** , foi deslocado totalmente para a direita. E outros números maiores também se moveram para a direita, ou "bubbled up".

Brian fará outra passagem pela lista:

```
3 6 5 2 7 4 1 8
- -
3 6 5 2 7 4 1 8
  - -
3 5 6 2 7 4 1 8
    - -
3 5 2 6 7 4 1 8
      - -
3 5 2 6 7 4 1 8
        - -
3 5 2 6 4 7 1 8
          - -
3 5 2 6 4 1 7 8
            - -
```

- Observe que não precisamos trocar o 3 e o 6, ou o 6 e o 7.
- Mas agora, o próximo maior valor, **7** , mudou totalmente para a direita.

Brian repetirá esse processo mais algumas vezes e cada vez mais a lista será ordenada, até que tenhamos uma lista totalmente ordenada.

Com selection sort, o melhor caso com uma lista ordenada ainda levaria tantos passos quanto o pior caso, uma vez que verificamos apenas o menor número em cada passagem.

O pseudocódigo para bubble sort pode ser parecido com:

```
Repita até estar ordenado
     Para i de 0 a n-2
```

```
Se iºésimo and i+1ºésimo elemento fora de ordem
    Troque eles
```

- Uma vez que estamos comparando o **i'ésimo** e **i + 1'ésimo** elemento, só precisamos ir até ***n – 2*** para **i** . Em seguida, trocamos os dois elementos se eles estiverem fora de ordem.
- E podemos parar assim que a lista for ordenada, já que podemos apenas lembrar se fizemos alguma troca. Caso não tenhamos feito, a lista já deve estar ordenada.

Para determinar o tempo de execução para classificação por bolha, temos **n – 1** comparações no loop e no máximo **n – 1** loops, portanto, obtemos **n^2 - 2n + 2** etapas no total. Mas o maior fator, ou termo dominante, é novamente **n^2** à medida que **n** fica cada vez maior, então podemos dizer que a classificação por bolha tem ***$O(n^2)$***. Portanto, basicamente, a classificação por seleção e a classificação por bolha têm o mesmo limite superior para o tempo de execução.

O limite inferior para o tempo de execução aqui seria Ω(n), uma vez que examinamos todos os elementos uma vez.

Portanto, nossos limites superiores para o tempo de execução que vimos são:

- $O(n^2)$
  - selection sort, bubble sort
- O(n log n)
- O(n)
  - busca linear
- O(log n)
  - busca binária
- O(1)

E para limites inferiores:

- $\Omega(n^2)$
  - selection sort
- Ω(nlog n)
- Ω(n)
  - bubble sort
- Ω(log n)
- Ω(1)
  - pesquisa linear, pesquisa binária

Recursão

Recursão é a capacidade de uma função chamar a si mesma. Ainda não vimos isso no código, mas vimos algo em pseudocódigo na semana 0 que podemos converter:

```
Pegue a lista telefônica
Abra no meio da lista telefônica
Olhe para a página
Se Smith estiver na página
   Ligue para Mike
Caso contrário, se Smith estiver no início do livro
   Aberto no meio da metade esquerda do livro
   Volte para a linha 3
Caso contrário, se Smith estiver mais tarde no livro
   Aberto no meio da metade direita do livro
   Volte para a linha 3
Senão
   Desistir
```

- Aqui, estamos usando uma instrução semelhante a um loop para voltar a uma linha específica.

Em vez disso, poderíamos apenas repetir todo o nosso algoritmo na metade do livro que restou:

```
Pegue a lista telefônica
Abra no meio da lista telefônica
Olhe para a página
Se Smith estiver na página
   Ligue para Mike
Caso contrário, se Smith estiver no início do livro
   Pesquise na metade esquerda do livro

Caso contrário, se Smith estiver mais tarde no livro
   Pesquise a metade direita do livro

Senão
   Desistir
```

- Parece um processo cíclico que nunca termina, mas na verdade estamos mudando a entrada para a função e dividindo o problema pela metade a cada vez, parando assim que não houver mais livro sobrando.

Na semana 1, também implementamos uma "pirâmide" de blocos na seguinte forma:

```
#
##
###
####
```

Mas observe que uma pirâmide de altura 4 é na verdade uma pirâmide de altura 3, com uma linha extra de 4 blocos adicionados. E uma pirâmide de altura 3 é uma pirâmide de altura 2, com uma linha extra de 3 blocos. Uma pirâmide de altura 2 é uma pirâmide de altura 1, com uma linha extra de 2 blocos. E, finalmente, uma pirâmide de altura 1 é apenas um único bloco.

Com essa ideia em mente, podemos escrever uma função recursiva para desenhar uma pirâmide, uma função que se chama para desenhar uma pirâmide menor antes de adicionar outra linha.

Merge sort

Podemos levar a ideia de recursão para classificação, com outro algoritmo chamado merge sort . O pseudocódigo pode ser semelhante a:

```
Se apenas um número
    Retornar
Senão
     Ordenar a metade esquerda do número
     Ordenar a metade direita do número
     Mesclar metades classificadas
```

Veremos isso melhor na prática com duas listas classificadas:

```
3 5 6 8 | 1 2 4 7
```

Vamos mesclar as duas listas para uma lista final classificada, pegando o menor elemento na frente de cada lista, um de cada vez:

```
3 5 6 8 | _ 2 4 7
1
```

O 1 no lado direito é o menor entre 1 e 3, então podemos começar nossa lista ordenada com ele.

```
3 5 6 8 |  4 7
1 2
```

O próximo menor número, entre 2 e 3, é 2, então usamos o 2.

```
_ 5 6 8 |  _ _ 4 7
1 2 3

_ 5 6 8 |_ _ _ 7
1 2 3 4

_ _ 6 8 |_ _ _ 7
1 2 3 4 5

_ _ _ 8 | _ _ _ 7
1 2 3 4 5 6

_ _ _ 8 | _ _ _ _
1 2 3 4 5 6 7

_ _ _ _ | _ _ _ _
1 2 3 4 5 6 7 8
```

- Agora temos uma lista completamente ordenada.

Vimos como a linha final em nosso pseudocódigo pode ser implementada e agora veremos como todo o algoritmo funciona:

```
Se apenas um número
   Retornar
Senão
   Ordenar a metade esquerda do número
```

```
   Ordenar a metade direita do número
Mesclar metades classificadas
```

Começamos com outra lista não classificada:

```
6 3 8 5 2 7 4 1
```

Para começar, precisamos classificar a metade esquerda primeiro:

```
6 3 8 5
```

Bem, para classificar isso, precisamos classificar a metade esquerda da metade esquerda primeiro:

```
6 3
```

Agora, ambas as metades têm apenas um item cada, portanto, estão classificadas. Nós mesclamos essas duas listas, para obter uma lista classificada:

```
_ _ 8 5 2 7 4 1
3 6
```

Estamos de volta à classificação da metade direita da metade esquerda, mesclando-as:

```
_ _ _ _ 2 7 4 1
3 6 5 8
```

As duas metades da *metade esquerda* foram classificadas individualmente, então agora precisamos mesclá-las:

```
_ _ _ _ 2 7 4 1
_ _ _ _
3 5 6 8
```

Faremos o que acabamos de fazer, com a metade certa:

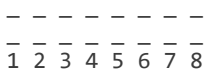
```
_ _ _ _ _ _ _ _
_ _ _ _ 2 7 4 1
3 5 6 8
```

- Primeiro, classificamos as duas metades da metade direita.

```
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
3 5 6 8 2 7 1 4
```

- Em seguida, nós os mesclamos para uma metade direita classificada.

Por fim, temos duas metades classificadas novamente e podemos mesclá-las para obter uma lista totalmente classificada:



Cada número foi movido de uma prateleira para outra três vezes (uma vez que a lista foi dividida de 8 para 4, para 2 e para 1 antes de ser mesclada novamente em listas ordenadas de 2, 4 e, finalmente, 8 novamente). E cada prateleira exigia que todos os 8 números fossem mesclados, um de cada vez.

Cada prateleira exigia n passos, e havia apenas **log n** prateleiras necessárias, então multiplicamos esses fatores juntos. Nosso tempo total de execução para pesquisa binária é ***O(log n)***:

- $O(n^2)$
  - selection sort, bubble sort
- O(nlog n)
  - merge sort
- O(n)
  - busca linear
- O(log n)
  - busca binária
- O(1)

(Uma vez que log n é maior que 1, mas menor que n, nlog n está entre n (vezes 1) e n^2.)

O melhor caso, Ω, ainda é n log n, uma vez que ainda temos que classificar cada metade primeiro e, em seguida, mesclá-los juntos:

- $\Omega(n^2)$
  - selection sort
- Ω(nlog n)
  - merge sort
- Ω(n)
  - bubble sort
- Ω(log n)
- Ω(1)
  - pesquisa linear, pesquisa binária

Embora a merge sort provavelmente seja mais rápida do que a selection sort ou bubble sort, precisamos de outra prateleira, ou mais memória, para armazenar temporariamente nossas listas mescladas em cada estágio. Enfrentamos a desvantagem de incorrer em um custo mais alto, outro array na memória, pelo benefício de uma classificação mais rápida.

Finalmente, há outra notação, Θ , Theta, que usamos para descrever os tempos de execução de algoritmos se o limite superior e o limite inferior forem iguais. Por exemplo, merge sort tem ***Θ (nlog n)***, uma vez que o melhor e o pior caso requerem o mesmo número de passos. E a classificação de seleção tem ***Θ(n^2)***:

- Θ(n^2)
  - selection sort
- Θ(nlog n)
  - merge sort
- Θ(n)
- Θ(log n)
- Θ(1)

Vemos uma **visualização final** dos algoritmos de classificação com um número maior de inputs, em execução ao mesmo tempo.