Este é o CS50

Introdução à Ciência da Computação (CS50)

OpenCourseWare

Doar (https://cs50.harvard.edu/donate)

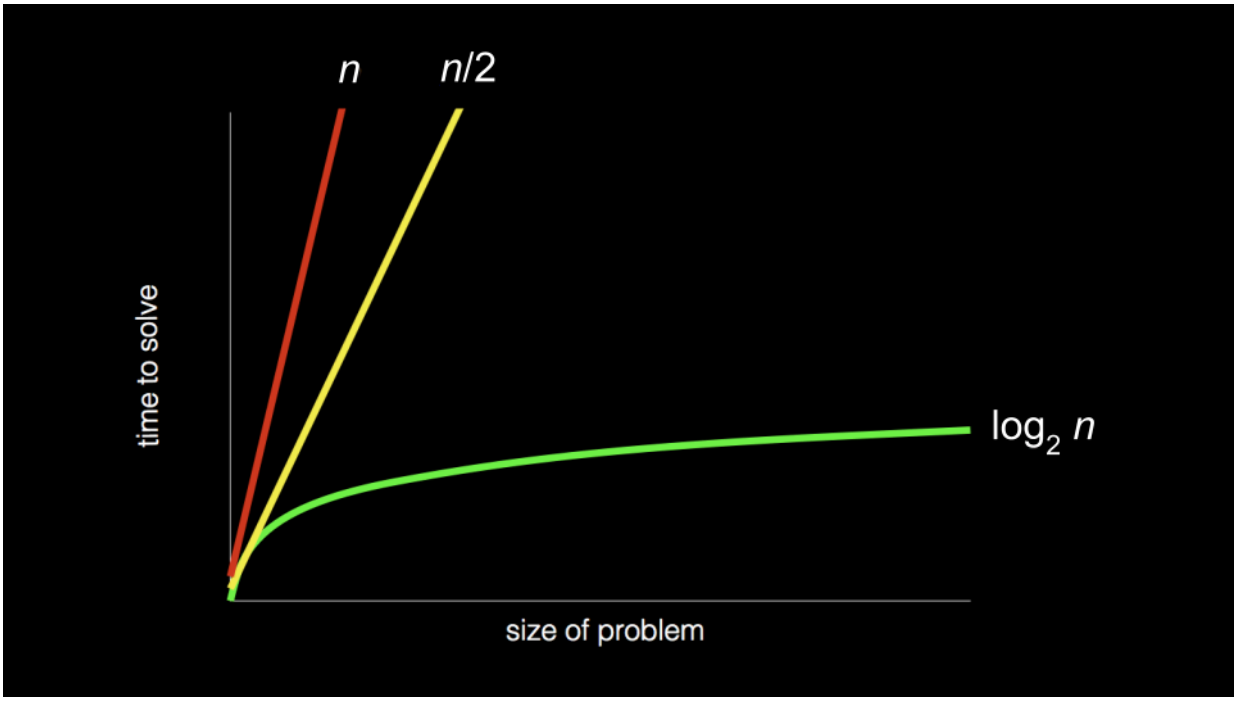
David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/) (https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Aula 3

Bem-vindo!

- Na semana zero, introduzimos a ideia de um *algoritmo* : uma caixa preta que pode receber uma entrada e gerar uma saída.
- Esta semana, vamos aprofundar nossa compreensão de algoritmos por meio de pseudocódigo e, posteriormente, com o código em si.

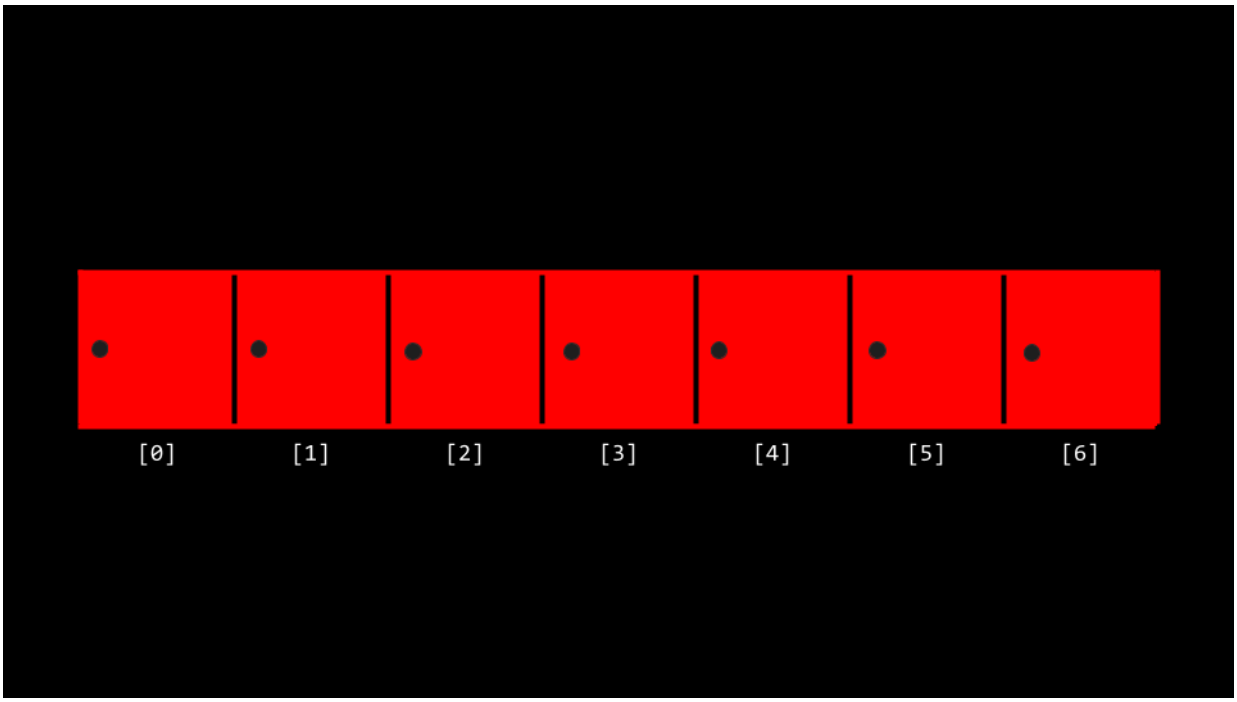
- Além disso, vamos considerar a eficiência desses algoritmos. De fato, vamos aprofundar nossa compreensão de como usar alguns dos conceitos que discutimos na semana passada na construção de algoritmos.
- Relembre o que vimos anteriormente no curso, quando apresentamos o seguinte gráfico:



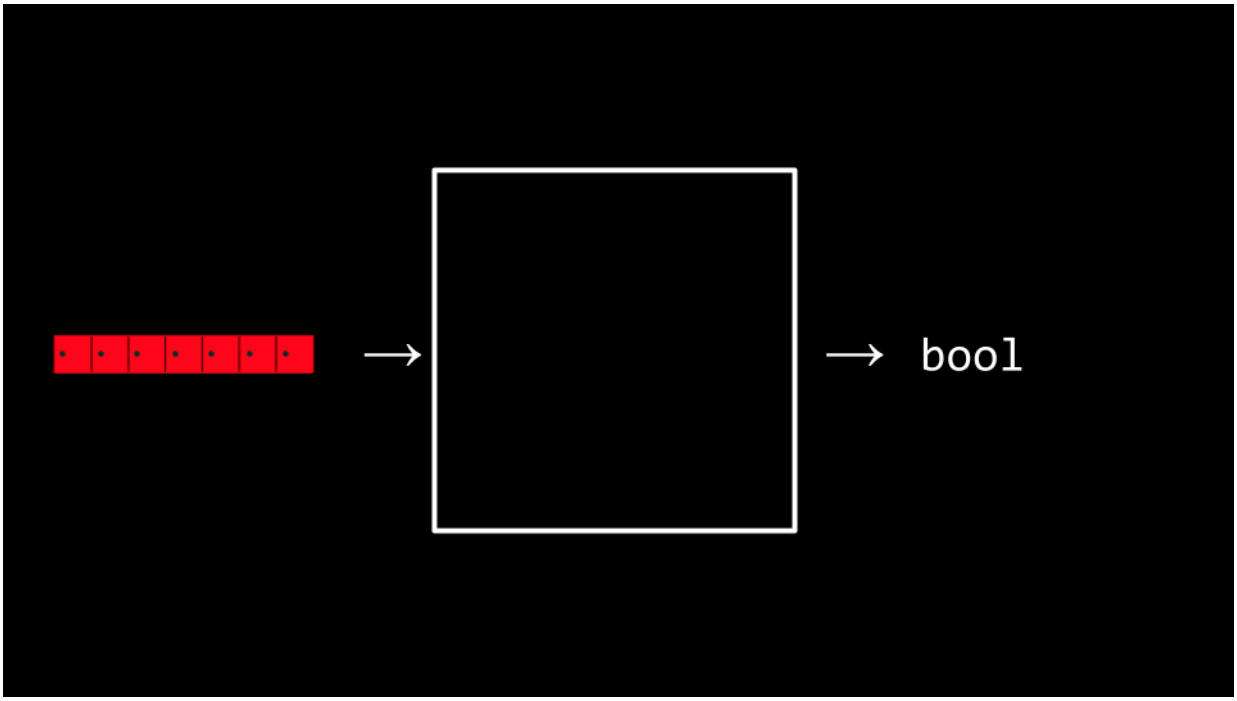
- Ao iniciarmos esta semana, você deve considerar como a forma como um algoritmo lida com um problema pode determinar o tempo necessário para resolvê-lo! Os algoritmos podem ser projetados para serem cada vez mais eficientes, até um certo limite.
- Hoje, vamos nos concentrar no projeto de algoritmos e em como medir sua eficiência.

Busca Linear

- Lembre-se que na semana passada você foi apresentado ao conceito de *array* , blocos de memória que são consecutivos: lado a lado.
- Você pode imaginar metaforicamente uma matriz como uma série de sete armários vermelhos, da seguinte forma:



- A posição mais à esquerda é chamada de *posição 0* ou *início da matriz* . A posição mais à direita é *a posição 6* ou *o fim da matriz* .
- Podemos imaginar que temos um problema fundamental: "O número 50 está dentro de uma matriz?". Um computador precisa examinar cada armário para verificar se o número 50 está presente. Chamamos esse processo de encontrar tal número, caractere, sequência ou outro item *de busca* .
- Podemos potencialmente entregar nosso array a um algoritmo, no qual nosso algoritmo procurará em nossos armários para ver se o número 50 está atrás de uma das portas, retornando o valor `true` ou `false` .



- Podemos imaginar várias instruções que poderíamos fornecer ao nosso algoritmo para realizar essa tarefa, como segue:

```
For each door from left to right
    If 50 is behind door
```

```
        Return true
Return false
```

Note que as instruções acima são chamadas de *pseudocódigo* : uma versão legível por humanos das instruções que poderíamos fornecer ao computador.

- Um cientista da computação poderia traduzir esse pseudocódigo da seguinte forma:

```
For i from 0 to n-1
    If 50 is behind doors[i]
        Return true
Return false
```

Note que o texto acima ainda não é código, mas é uma aproximação bastante precisa de como o código final poderá ser.

Busca Binária

- *A busca binária* é outro *algoritmo de busca* que poderia ser empregado em nossa tarefa de encontrar os 50.
- Supondo que os valores dentro dos armários estejam organizados do menor para o maior, o pseudocódigo para a busca binária seria o seguinte:

```
If no doors left
    Return false
If 50 is behind middle door
    Return true
Else if 50 < middle door
    Search left half
Else if 50 > middle door
    Search right half
```

- Utilizando a nomenclatura de código, podemos modificar ainda mais nosso algoritmo da seguinte forma:

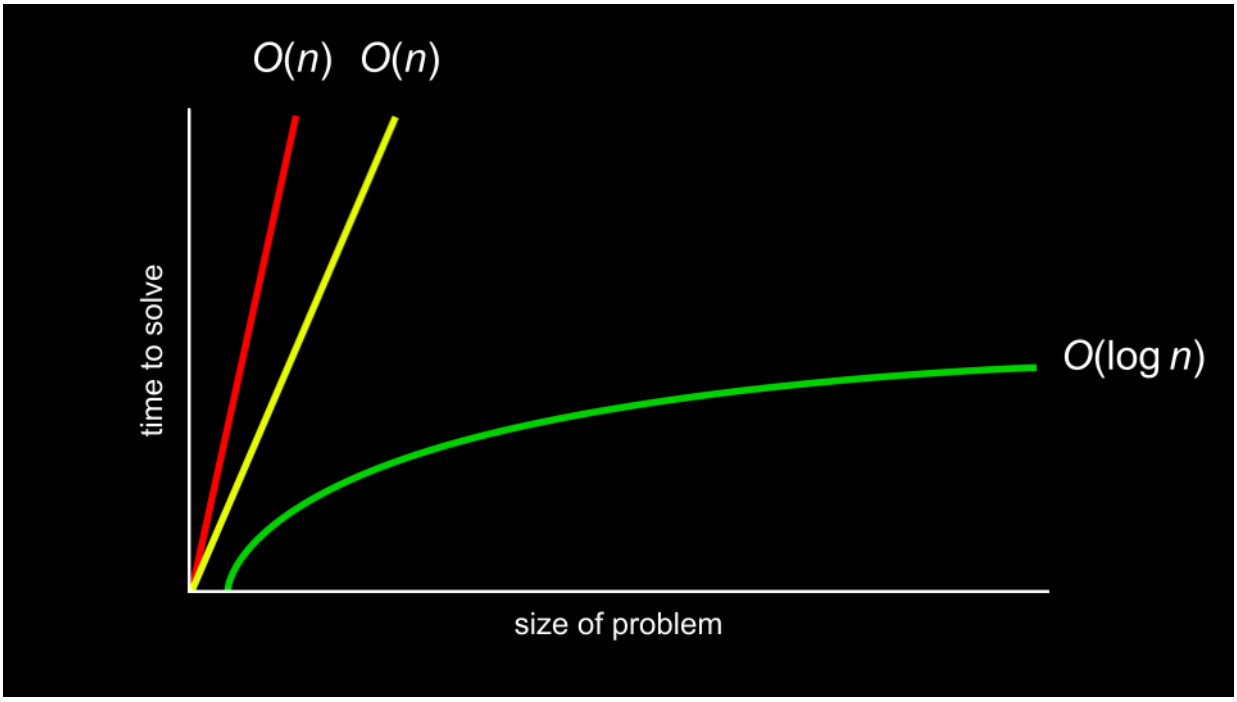
```
If no doors left
    Return false
If 50 is behind doors[middle]
    Return true
Else if 50 < doors[middle]
    Search doors[0] through doors[middle - 1]
Else if 50 > doors[middle]
    Search doors[middle + 1] through doors[n - 1]
```

Note que, ao observar essa aproximação do código, você consegue quase imaginar como ele seria em um código real.

Duração

- Você pode considerar quanto tempo um algoritmo leva para resolver um problema.

- *O tempo de execução* envolve uma análise usando a notação *Big O*. Veja o gráfico a seguir:



- Em vez de serem extremamente específicos sobre a eficiência matemática de um algoritmo, os cientistas da computação discutem a eficiência em termos da *ordem de* grandeza de seus tempos de execução.
- No gráfico acima, o primeiro algoritmo é$\mathrm{O(n)}$ou *na ordem de n* . O segundo está em $\mathrm{O(n)}$também. O terceiro está em$\mathrm{O(\log n)}$.
- É o formato da curva que demonstra a eficiência de um algoritmo. Alguns tempos de execução comuns que podemos observar são:
  - $\mathrm{O(n^2)}$
  - $\mathrm{O(n \log n)}$
  - $\mathrm{O(n)}$
  - $\mathrm{O(\log n)}$
  - $\mathrm{O(1)}$
- Dos tempos de execução acima,$\mathrm{O(n^2)}$é considerado o tempo de execução mais lento. $\mathrm{O(1)}$é o mais rápido.
- A busca linear era de ordem$\mathrm{O(n)}$porque , no pior dos casos, a execução pode levar *n passos*.
- A busca binária era de ordem$\mathrm{O(\log n)}$Porque seriam necessários cada vez menos passos para correr, mesmo no pior cenário.
- Os programadores estão interessados tanto no pior caso, ou *limite superior* , quanto no melhor caso, ou *limite inferior* .
- OΩO símbolo é usado para denotar o melhor caso de um algoritmo, como por exemplo: $\Omega(\log \mathrm{n})$.

- OΘO símbolo é usado para indicar onde o limite superior e o limite inferior são iguais: onde os tempos de execução no melhor caso e no pior caso são iguais.
- *A notação assintótica* é a medida de quão bem os algoritmos se comportam à medida que a entrada se torna cada vez maior.
- À medida que você continuar a desenvolver seus conhecimentos em ciência da computação, explorará esses tópicos com mais detalhes em cursos futuros.

pesquisa.c

- Você pode implementar a busca linear digitando `code search.c` no seu terminal e escrevendo o código da seguinte forma:

```c
// Implements linear search for integers

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

int main(void)
{
    // An array of integers
    int numbers[] = {20, 500, 10, 5, 100, 1, 50};

    // Search for number
    int n = get_int("Number: ");
    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == n)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  Observe que a linha que começa com `if` `int numbers[]` nos permite definir os valores de cada elemento da matriz à medida que a criamos. Em seguida, no `for` laço, temos uma implementação de busca linear. `if` `return 0` é usado para indicar sucesso e encerrar o programa. `if` `return 1` é usado para encerrar o programa com um erro (falha).
- Implementamos agora a busca linear em C!
- E se quiséssemos procurar uma string dentro de um array? Modifique seu código da seguinte forma:

```c
// Implements linear search for strings

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

int main(void)
```

```
{
    // An array of strings
    string strings[] = {"battleship", "boot", "cannon", "iron", "thimble", "to

    // Search for string
    string s = get_string("String: ");
    for (int i = 0; i < 6; i++)
    {
        if (strcmp(strings[i], s) == 0)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Observe que não podemos utilizar ` `==`as` na iteração anterior deste programa. Em vez disso, usamos `` `strcmp`, que vem da `string.h` biblioteca. ` `strcmp` ` retornará `0` se as strings forem iguais. Observe também que o comprimento da string em `` `6` está codificado diretamente no código, o que não é uma boa prática de programação.

- De fato, executar este código nos permite iterar sobre este array de strings para verificar se uma determinada string está presente. No entanto, se você encontrar uma *falha de segmentação* , onde uma parte da memória foi acessada pelo seu programa sem que ele devesse ter acesso, certifique-se de ter `i < 6` notado acima em vez de `i < 7`.
- Você pode aprender mais sobre isso `strcmp` nas páginas do manual do CS50 (https://manual.cs50.io/3/strcmp) .

agenda telefônica.c

- Podemos combinar essas ideias de números e strings em um único programa. Digite `code phonebook.c` o seguinte código na janela do terminal:

```
// Implements a phone book without structs

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

int main(void)
{
    // Arrays of strings
    string names[] = {"Yuliia", "David", "John"};
    string numbers[] = {"+1-617-495-1000", "+1-617-495-1000", "+1-949-468-2750

    // Search for name
    string name = get_string("Name: ");
    for (int i = 0; i < 3; i++)
    {
        if (strcmp(names[i], name) == 0)
        {
```

```
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Observe que o número de Yuliia começa com `+1-617`, o número de telefone de David começa com `+1-617` e o número de John começa com `+1-949`. Portanto, `names[0]` é o número de Yuliia e `numbers[0]` é o número de Yuliia. Esse código nos permitirá pesquisar na agenda telefônica o número específico de uma pessoa.

- Embora esse código funcione, ele apresenta diversas ineficiências. De fato, existe a possibilidade de que nomes e números de telefone não correspondam. Não seria ótimo se pudéssemos criar nosso próprio tipo de dado onde pudéssemos associar uma pessoa ao seu número de telefone?

Estruturas

- Descobrimos que a linguagem C nos permite criar nossos próprios tipos de dados por meio de um `struct`...
- Não seria útil criar nosso próprio tipo de dados chamado `a` `person` que contenha um `a` `name` e um `b` `number`? Considere o seguinte:

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

Observe como isso representa nosso próprio tipo de dados chamado `a`, `person` que possui uma string chamada `a` `name` e outra string chamada `b` `number`.

- Podemos melhorar nosso código anterior modificando nosso programa de agenda telefônica da seguinte forma:

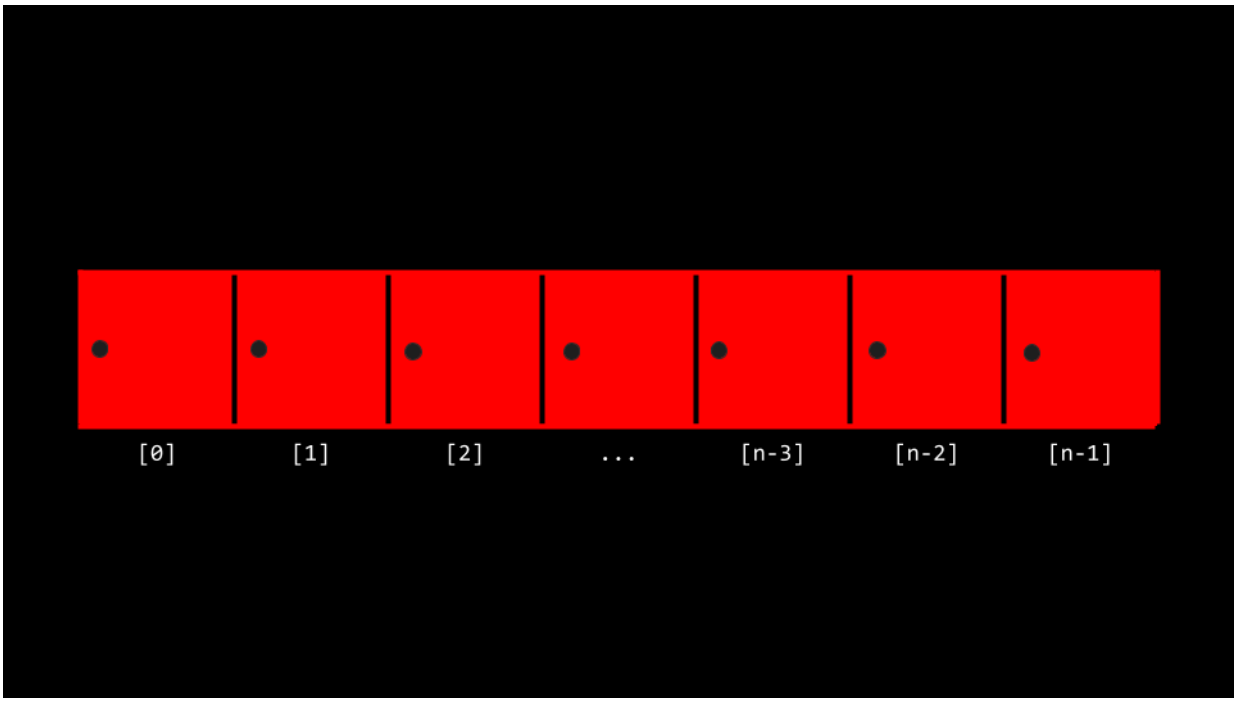
```
// Implements a phone book with structs

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

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

int main(void)
{
```

```
    person people[3];

    people[0].name = "Yuliia";
    people[0].number = "+1-617-495-1000";

    people[1].name = "David";
    people[1].number = "+1-617-495-1000";

    people[2].name = "John";
    people[2].number = "+1-949-468-2750";

    // Search for name
    string name = get_string("Name: ");
    for (int i = 0; i < 3; i++)
    {
        if (strcmp(people[i].name, name) == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Observe que o código começa com a definição de `typedef struct` um novo tipo de dados chamado `names` . Dentro de `names` , temos uma string chamada `name` e um número chamado `number`. Na função `update_number`, criamos um array chamado `number` do tipo ` number` com tamanho 3. Em seguida, atualizamos os nomes e números de telefone das duas pessoas no array. O mais importante é observar como a *notação de ponto* , como `number`, nos permite acessar o ` name` na posição 0 e atribuir um nome a essa pessoa. `person` `person` `name` `string` `main` `people` `person` `people` `people[0].name` `person`

Classificação

- *Ordenar* é o ato de pegar uma lista de valores não ordenada e transformá-la em uma lista ordenada.
- Quando uma lista está ordenada, a busca nessa lista exige muito menos processamento do computador. Lembre-se de que podemos usar a busca binária em uma lista ordenada, mas não em uma lista não ordenada.
- Descobriu-se que existem muitos tipos diferentes de algoritmos de ordenação.
- *A ordenação por seleção* é um desses algoritmos de ordenação.
- Podemos representar um array da seguinte forma:



- O algoritmo de ordenação por seleção em pseudocódigo é:

```
For i from 0 to n-1
    Find smallest number between numbers[i] and numbers[n-1]
    Swap smallest number with numbers[i]
```

- Resumindo esses passos, a primeira iteração pela lista levou `n - 1` passos. A segunda vez, levou `n - 2` passos. Seguindo essa lógica, os passos necessários poderiam ser representados da seguinte forma:

```
(n - 1) + (n - 2) + (n - 3) + ... + 1
```

- Isso poderia ser simplificado para n(n-1)/2 ou, mais simplesmente, $O(n^2)$No pior caso ou limite superior, a ordenação por seleção está na ordem de$O(n^2)$No melhor caso, ou limite inferior, a ordenação por seleção está na ordem de$\Omega(n^2)$.

Classificação de bolhas

- *O Bubble Sort* é outro algoritmo de ordenação que funciona trocando elementos repetidamente para "promover" os elementos maiores para o final.
- O pseudocódigo para o algoritmo de ordenação por bolha é:

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
    If no swaps
        Quit
```

- À medida que ordenamos o array, percebemos que cada vez mais partes dele ficam ordenadas, então precisamos analisar apenas os pares de números que ainda não foram ordenados.

- O algoritmo de ordenação por bolha pode ser analisado da seguinte forma:

  ```
  (n - 1) × (n - 1)
  n2 - 1n - 1n + 1
  n2 - 2n + 1
  ```

  ou, mais simplesmente$O(n^2)$.
- No pior caso, ou limite superior, o algoritmo de ordenação por bolha está na ordem de $O(n^2)$No melhor caso, ou limite inferior, o algoritmo de ordenação por bolha está na ordem de$\Omega(n)$.
- Você pode visualizar (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) uma comparação desses algoritmos.

Recursão

- Como podemos melhorar a eficiência da nossa triagem?
- *Recursão* é um conceito em programação onde uma função chama a si mesma. Vimos isso anteriormente quando vimos...

  ```
  If no doors left
      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
  ```

  Observe que estamos recorrendo `search` a iterações cada vez menores deste problema.
- Da mesma forma, em nosso pseudocódigo da Semana 0, você pode ver onde a recursão foi implementada:

  ```
  1  Pick up phone book
  2  Open to middle of phone book
  3  Look at page
  4  If person is on page
  5      Call person
  6  Else if person is earlier in book
  7      Open to middle of left half of book
  8      Go back to line 3
  9  Else if person is later in book
  10     Open to middle of right half of book
  11     Go back to line 3
  12 Else
  13     Quit
  ```

- Este código poderia ter sido simplificado para destacar suas propriedades recursivas da seguinte forma:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Search left half of book
9  Else if person is later in book
10     Search right half of book
12 Else
13     Quit
```

- Considere como, na Semana 1, queríamos criar uma estrutura piramidal da seguinte forma:

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

- Digite `code iteration.c` o seguinte código na janela do terminal:

```c
// Draws a pyramid using iteration

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

void draw(int n);

int main(void)
{
    // Get height of pyramid
    int height = get_int("Height: ");

    // Draw pyramid
    draw(height);
}

void draw(int n)
{
    // Draw pyramid of height n
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

Observe que este código constrói a pirâmide por meio de um loop.

- Para implementar isso usando recursão, `code iteration.c` digite o seguinte código na janela do terminal:

```c
// Draws a pyramid using recursion

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

void draw(int n);

int main(void)
{
    // Get height of pyramid
    int height = get_int("Height: ");

    // Draw pyramid
    draw(height);
}

void draw(int n)
{
    // If nothing to draw
    if (n <= 0)
    {
        return;
    }

    // Draw pyramid of height n - 1
    draw(n - 1);

    // Draw one more row of width n
    for (int i = 0; i < n; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

Observe que o *caso base* garantirá que o código não seja executado indefinidamente. A linha `if (n <= 0)` encerra a recursão porque o problema foi resolvido. Cada vez `draw` que `f` chama a si mesma, ela chama a si mesma por `f` `n-1`. Em algum momento, ` `n-1` f` será igual a `f` `0`, resultando no `draw` retorno da função e, consequentemente, no fim do programa.

Classificação por intercalação

- Agora podemos aproveitar a recursão em nossa busca por um algoritmo de ordenação mais eficiente e implementar o que é chamado de *ordenação por intercalação (merge sort)* , um algoritmo de ordenação muito eficiente.
- O pseudocódigo para o algoritmo de ordenação por intercalação (merge sort) é bastante curto:

```
If only one number
    Quit
Else
    Sort left half of number
```

```
    Sort right half of number
    Merge sorted halves
```

- Considere a seguinte lista de números:

```
6341
```

- Primeiro, o algoritmo de ordenação por intercalação (merge sort) pergunta: "Este é um único número?". A resposta é "não", então o algoritmo continua.

```
6341
```

- Em segundo lugar, o merge sort irá dividir os números ao meio (ou o mais próximo possível disso) e ordenar a metade esquerda dos números.

```
63|41
```

- Em terceiro lugar, o algoritmo de ordenação por intercalação (merge sort) analisaria esses números à esquerda e perguntaria: "Este número é um só?". Como a resposta é não, ele dividiria os números à esquerda ao meio.

```
6|3
```

- Em quarto lugar, o algoritmo de ordenação por intercalação (merge sort) perguntará novamente: "Este é um único número?". A resposta desta vez é sim! Portanto, ele encerrará esta tarefa e retornará à última tarefa que estava executando neste ponto:

```
63|41
```

- Em quinto lugar, o algoritmo de ordenação por intercalação (merge sort) irá ordenar os números à esquerda.

```
36|41
```

- Agora, retornamos ao ponto onde paramos no pseudocódigo, uma vez que o lado esquerdo foi ordenado. Um processo semelhante, seguindo os passos 3 a 5, ocorrerá com os números do lado direito. Isso resultará em:

```
36|14
```

- As duas metades agora estão ordenadas. Finalmente, o algoritmo irá mesclar os dois lados. Ele analisará o primeiro número à esquerda e o primeiro número à direita. Colocará o menor número em primeiro lugar e, em seguida, o segundo menor. O algoritmo repetirá esse processo para todos os números, resultando em:

```
1346
```

- A ordenação por intercalação foi concluída e o programa foi encerrado.
- O Merge Sort é um algoritmo de ordenação muito eficiente com um pior caso de $O(n \log n)$O melhor cenário ainda é$\Omega(n \log n)$porque o algoritmo ainda precisa visitar

cada posição na lista. Portanto, o merge sort também é $\Theta(n \log n)$ já que o melhor cenário e o pior cenário são iguais.

- Uma visualização (https://www.youtube.com/watch?v=ZZuD6iUe3Pc) final foi compartilhada.

Resumindo

Nesta lição, você aprendeu sobre pensamento algorítmico e como criar seus próprios tipos de dados. Especificamente, você aprendeu...

- Algoritmos.
- Notação Big *O*.
- Busca binária e busca linear.
- Diversos algoritmos de ordenação, incluindo ordenação por bolha, ordenação por seleção e ordenação por intercalação.
- Recursão.

Até a próxima!