Esto es CS50x

OpenCourseWare

Donar (https://cs50.harvard.edu/donate) (https://community.alumni.harvard.edu/give/59206872)

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://orcid.org/0000-0001-5338-2522) (https://www.quora.com/profile/David-J-Malan) (https://www.reddit.com/user/davidjmalan) (https://twitter.com/davidjmalan)

Lección 3

La semana pasada

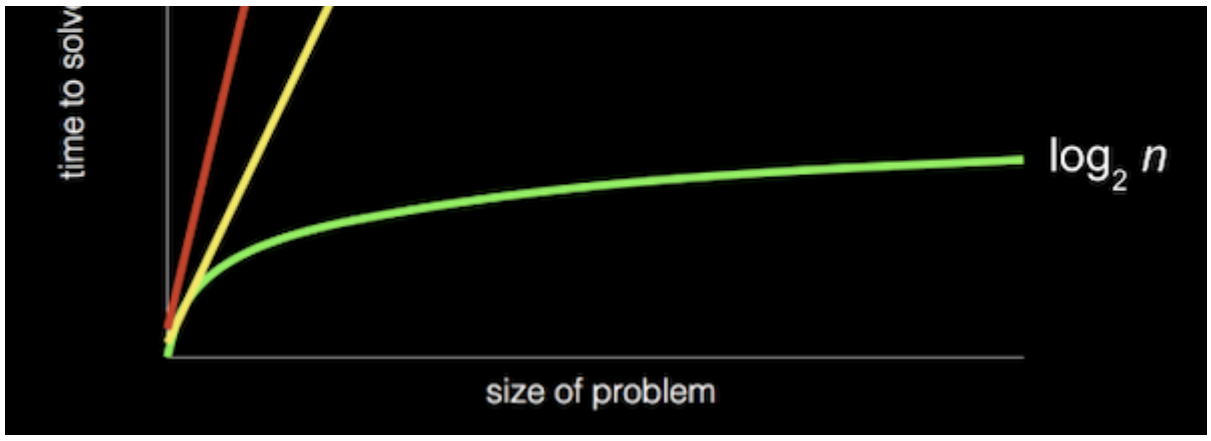
- Aprendimos sobre herramientas para resolver problemas o errores en nuestro código. En particular, descubrimos cómo usar un depurador, una herramienta que nos permite recorrer lentamente nuestro código y mirar los valores en la memoria mientras nuestro programa se está ejecutando.
- Otra herramienta poderosa, aunque menos técnica, es la depuración de patitos de goma, donde tratamos de explicar lo que estamos tratando de hacer con un patito de goma (o algún otro objeto) y, en el proceso, nos damos cuenta del problema (¡y con suerte la solución!) En nuestra propia.
- Observamos la memoria, visualizamos bytes en una cuadrícula y almacenamos valores en cada cuadro, o byte, con variables y matrices.

buscando

- Resulta que, con las matrices, una computadora no puede ver todos los elementos a la vez. En cambio, una computadora solo puede verlos uno a la vez, aunque el orden puede ser arbitrario. (Recuerde que en la semana 0, David solo podía mirar una página a la vez en una guía telefónica, ya sea que las hojeara en orden o de una manera más sofisticada).
- **La búsqueda** es la forma en que resolvemos el problema de encontrar un valor en particular. Un caso simple puede tener una entrada de alguna matriz de valores, y la salida puede ser simplemente a `bool`, ya sea que haya o no un valor particular en la matriz.
- Hoy veremos los algoritmos de búsqueda. Para discutirlos, consideraremos **el tiempo de ejecución** o cuánto tarda un algoritmo en ejecutarse dado cierto tamaño de entrada.

Grande O

- En la semana 0, vimos diferentes tipos de algoritmos y sus tiempos de ejecución:



  - Recuerde que la línea roja busca linealmente, una página a la vez; la línea amarilla busca dos páginas a la vez; y la línea verde busca logarítmicamente, dividiendo el problema a la mitad cada vez.
  - Y estos tiempos de ejecución son para el peor de los casos, o el caso en el que el valor tarda más en encontrar (en la última página, a diferencia de la primera página).

- La forma más formal de describir cada uno de estos tiempos de ejecución es con **grandes$O$notación** , que podemos considerar como "del orden de". Por ejemplo, si nuestro algoritmo es una búsqueda lineal, tomará aproximadamente$O(n)$ pasos, lea como "grande O de n"O" en el orden de n". De hecho, incluso un algoritmo que analiza dos elementos a la vez y toma$n/2$ pasos tiene $O(n)$. Esto se debe a que, como$n$ se hace más y más grande, solo el factor dominante, o el término más grande, n, asuntos. En el gráfico de arriba, si alejáramos y cambiamos las unidades en nuestros ejes, veríamos que las líneas roja y amarilla terminan muy juntas.
- Un tiempo de ejecución logarítmico es $O(\log n)$, no importa cuál sea la base, ya que esto es solo una aproximación de lo que sucede fundamentalmente con el tiempo de ejecución si n es muy grande.
- Hay algunos tiempos de ejecución habituales:
  - $O(n^2)$
  - $O(n \log n)$
  - $O(n)$
    - (buscando una página a la vez, en orden)
  - $O(\log n)$
    - (dividiendo la guía telefónica a la mitad cada vez)
  - $O(1)$
    - Un algoritmo que toma un número **constante** de pasos, independientemente de la magnitud del problema.
- Los informáticos también pueden utilizar grandes Ω, notación Omega grande, que es el límite inferior del número de pasos de nuestro algoritmo. Grande$O$ es el límite superior del número de pasos, o el peor de los casos.
- Y tenemos un conjunto similar de los tiempos de ejecución de Ω grandes más comunes:
  - $\Omega(n^2)$
  - $\Omega(n \log n)$
  - $\Omega(n)$
  - $\Omega(\log n)$
  - $\Omega(1)$
    - (buscando en una guía telefónica, ya que podríamos encontrar nuestro nombre en la primera página que revisamos)

Búsqueda lineal, búsqueda binaria

- En el escenario, tenemos algunas puertas de utilería, con números escondidos detrás de ellas. Dado que una computadora solo puede mirar un elemento en una matriz a la vez, solo podemos abrir una puerta a la vez.
- Si queremos buscar el número cero, por ejemplo, tendríamos que abrir una puerta a la vez, y si no supiéramos nada sobre los números detrás de las puertas, el algoritmo más simple sería ir de izquierda a derecha.
- Entonces, podríamos escribir pseudocódigo para **búsqueda lineal** con:

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

  - Etiquetamos cada una de las `n` puertas de `0` a `n–1`, y verificamos cada una de ellas en orden.
  - "Devolver falso" está *fuera* del ciclo for, ya que solo queremos hacer eso después de haber mirado detrás de *todas* las puertas.
  - El gran O El tiempo de ejecución de este algoritmo sería $O(n)$, y el límite inferior, gran Omega, sería $\Omega(1)$.
- Si sabemos que los números detrás de las puertas están ordenados, entonces podemos comenzar en el medio y encontrar nuestro valor de manera más eficiente.
- Para la búsqueda binaria, nuestro algoritmo podría verse así:

```
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
```

- El límite superior para la búsqueda binaria es $O(\log n)$, y el límite inferior también $\Omega(1)$, si el número que estamos buscando está en el medio, por dónde empezamos.

- Con 64 bombillas, notamos que la búsqueda lineal lleva mucho más tiempo que la búsqueda binaria, que solo requiere unos pocos pasos.
- Apagamos las bombillas a una frecuencia de un **hercio** , o ciclo por segundo, y la velocidad de un procesador podría medirse en gigahercios, o miles de millones de operaciones por segundo.

Buscando con código

- Echemos un vistazo a `numbers.c`:

```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;
}
```

- Aquí inicializamos una matriz con algunos valores entre llaves, y verificamos los elementos de la matriz uno a la vez, en orden, para ver si son iguales a cero (lo que originalmente estábamos buscando detrás de las puertas en el escenario). .
- Si encontramos el valor de cero, devolvemos un código de salida de 0 (para indicar éxito). De lo contrario, *después de* nuestro ciclo for, devolvemos 1 (para indicar falla).

- Podemos hacer lo mismo con los nombres:

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

int main(void)
{
    string names[] = {"Bill", "Charlie", "Fred", "George", "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;
}
```

- Tenga en cuenta que `names` es una matriz ordenada de cadenas.
- No podemos comparar cadenas directamente en C, ya que no son un tipo de datos simple sino una matriz de muchos caracteres. Afortunadamente, la `string` biblioteca tiene una función `strcmp` ("comparar cadenas") que compara cadenas para nosotros, un

carácter a la vez, y devuelve `0` si son iguales.

- Si solo buscamos `strcmp(names[i], "Ron")` y no `strcmp(names[i], "Ron") == 0`, imprimiremos `Found` incluso si no se encuentra el nombre. Esto se debe a que `strcmp` devuelve un valor que no lo es `0` si dos cadenas *no* coinciden, y cualquier valor distinto de cero es equivalente a verdadero en una condición.

Estructuras

- Si quisiéramos implementar un programa que busca en una guía telefónica, podríamos querer un tipo de datos para una "persona", con su nombre y número de teléfono.
- Resulta que en C podemos definir nuestro propio tipo de datos, o *estructura de* datos , con una **estructura** en la siguiente sintaxis:

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

  - Usamos `string` para `number`, ya que queremos incluir símbolos y formato, como signos más o guiones.
  - Nuestra estructura contiene otros tipos de datos dentro de ella.
- Primero intentemos implementar nuestra guía telefónica sin estructuras:

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

  - Tendremos que tener cuidado de asegurarnos de que `names` el primer nombre en coincide con el primer número en `numbers`, y así sucesivamente.
  - Si el nombre en un determinado índice `i` en la `names` matriz coincide con lo que estamos buscando, podemos devolver el número de teléfono en la `numbers` matriz en el mismo índice.
- Con las estructuras, podemos estar un poco más seguros de que no tendremos errores humanos en nuestro 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("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

- Creamos una matriz del `person` tipo de estructura y la nombramos `people` (como en `int numbers[]`, aunque podríamos nombrarla arbitrariamente, como cualquier otra variable). Hemos establecido los valores para cada campo, o variable, dentro de cada `person` estructura, utilizando el operador punto, `.`.
- En nuestro ciclo, ahora podemos estar más seguros de que `number` corresponde a, `name` ya que son de la misma `person` estructura.
- También podemos mejorar el diseño de nuestro programa con una constante, como `const int NUMBER = 10;`, y almacenar nuestros valores no en nuestro código sino en un archivo separado o incluso en una base de datos, que veremos pronto.

- Pronto también, escribiremos nuestros propios archivos de encabezado con definiciones para estructuras, para que puedan compartirse en diferentes archivos para nuestro programa.

Clasificación

- Si nuestra entrada es una lista de números sin clasificar, hay muchos algoritmos que podríamos usar para producir una salida de una lista ordenada, donde todos los elementos están en orden.
- Con una lista ordenada, podemos usar la búsqueda binaria para la eficiencia, pero puede llevar más tiempo escribir un algoritmo de clasificación para esa eficiencia, por lo que a veces nos encontraremos con la compensación del tiempo que le toma a un humano escribir un programa en comparación con el tiempo. se necesita una computadora para ejecutar algún algoritmo. Otras compensaciones que veremos podrían ser el tiempo y la complejidad, o el tiempo y el uso de la memoria.

veremos podrían ser el tiempo y la complejidad, o el tiempo y el uso de la memoria.

Orden de selección

- Brian está detrás del escenario con un conjunto de números en un estante, sin clasificar:

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

- Tomando algunos números y moviéndolos a su lugar correcto, Brian ordena los números con bastante rapidez.
- Yendo paso a paso, Brian mira cada número de la lista, recordando el más pequeño que hemos visto hasta ahora. Llega al final y ve que 1 es el más pequeño, y sabe que debe ir al principio, así que lo cambiará por el número al principio, 6:

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

- Ahora Brian sabe que al menos el primer número está en el lugar correcto, por lo que puede buscar el número más pequeño entre los demás e intercambiarlo con el siguiente número sin clasificar (ahora el segundo número):

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

- Y repite esto de nuevo, intercambiando el siguiente más pequeño, 3, con el 8:

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

- Después de algunos intercambios más, terminamos con una lista ordenada.
- Este algoritmo se llama **ordenamiento por selección** , y podemos ser un poco más específicos con algún pseudocódigo:

```
For i from 0 to n–1
    Find smallest item between i'th item and last item
    Swap smallest item with i'th item
```

  - El primer paso en el ciclo es buscar el elemento más pequeño en la parte no ordenada de la lista, que estará entre el elemento i y el último elemento, ya que sabemos que lo hemos ordenado hasta el "i-1". " Articulo.
  - Luego, intercambiamos el elemento más pequeño con el elemento i'th, lo que hace que todo llegue al elemento que he ordenado.
- Observamos una visualización en línea (https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html) con animaciones de cómo se mueven los elementos para la ordenación por inserción.
- Para este algoritmo, analizamos aproximadamente todos n elementos para encontrar el más pequeño, y hacer n pasa para ordenar todos los elementos.
- Más formalmente, podemos usar algunas fórmulas matemáticas para mostrar que el factor más importante es de hecho n^2. Empezamos por tener que mirar todo$n$ elementos, entonces solo $n - 1$, entonces $n - 2$:
$n + (n-1) + (n-2) + \ldots + 1$
$n(n + 1)/2$
$(n^2 + n)/2$
$n^2/2 + n/2$
$O(n^2)$
  - Ya que n^2 es el factor más grande o dominante, podemos decir que el algoritmo tiene un tiempo de ejecución de $O(n^2)$.

Ordenamiento de burbuja

- Podemos probar un algoritmo diferente, uno en el que intercambiamos pares de números repetidamente, llamado **clasificación de burbujas** .
- Brian mirará los dos primeros números y los intercambiará para que estén en orden:

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

- El siguiente par, `6` y `8`, están en orden, por lo que no es necesario intercambiarlos.
- El siguiente par, `8` y `5`, deben intercambiarse:

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

- Brian continúa hasta que llega al final de la 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
            –
```

- Nuestra lista aún no está ordenada, pero estamos un poco más cerca de la solución porque el valor más grande `8`, se ha desplazado completamente hacia la derecha. Y otros números más grandes también se han movido hacia la derecha, o "han aumentado".
- Brian hará otra pasada por la lista:

```
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
            – –
```

  - Tenga en cuenta que no necesitamos intercambiar el 3 y el 6, o el 6 y el 7.
  - Pero ahora, el siguiente valor más grande `7`, se movió completamente hacia la derecha.
- Brian repetirá este proceso unas cuantas veces más, y más y más partes de la lista se ordenarán, hasta que tengamos una lista completamente ordenada.
- Con la ordenación por selección, el mejor caso con una lista ordenada aún requeriría tantos pasos como el peor de los casos, ya que solo verificamos el número más pequeño con cada pasada.
- El pseudocódigo para la clasificación de burbujas podría verse así:

```
Repeat until sorted
    For i from 0 to n–2
        If i'th and i+1'th elements out of order
            Swap them
```

  - Como estamos comparando el elemento `i'th` y `i+1'th`, solo necesitamos subir a$n-2$para `i`. Luego, intercambiamos los dos elementos si están fuera de servicio.
  - Y podemos detenernos tan pronto como la lista esté ordenada, ya que solo podemos recordar si hicimos cambios. De lo contrario, la lista ya debe estar ordenada.
- Para determinar el tiempo de ejecución de la clasificación de burbujas, tenemos $n-1$ comparaciones en el bucle, y como máximo $n-1$ bucles, así que obtenemos n^2-2n+2pasos en total. Pero el factor más importante, o término dominante, es nuevamenten^2a medida que se `n` hace más y más grande, podemos decir que el tipo de burbuja tiene$O(n^2)$. Entonces, resulta que, fundamentalmente, la ordenación por inserción y la ordenación por burbujas tienen el mismo límite superior para el tiempo de ejecución.
- El límite inferior para el tiempo de ejecución aquí sería $\Omega(n)$, una vez que miramos todos los elementos una vez.
- Entonces, nuestros límites superiores para el tiempo de ejecución que hemos visto son:
  - $O(n^2)$
    - clasificación de selección, clasificación de burbujas
  - $O(n \log n)$
  - $O(n)$
    - búsqueda lineal
  - $O(\log n)$

        - búsqueda binaria
    - $O(1)$
- Y para límites inferiores:
    - $\Omega(n^2)$
        - orden de selección
    - $\Omega(n \log n)$
    - $\Omega(n)$
        - ordenamiento de burbuja
    - $\Omega(\log n)$
    - $\Omega(1)$
        - búsqueda lineal, búsqueda binaria

Recursividad

- **La recursividad** es la capacidad de una función de llamarse a sí misma. No hemos visto esto en código todavía, pero hemos visto algo en pseudocódigo en la semana 0 que podríamos convertir:

```
1 Recoger directorio telefónico
2 Abrir hasta la mitad de la guía telefónica
3 Mira la página
4 Si Smith está en la página
5 Llama a Mike
6 De lo contrario, si Smith es anterior en el libro
7 Abierto hasta la mitad de la mitad izquierda del libro
8       Regrese a la línea 3
9 De lo contrario, si Smith está más adelante en el libro
10 Abierto hasta la mitad de la mitad derecha del libro
11      Regrese a la línea 3
12 más
13 Salir
```

    - Aquí, estamos usando una instrucción en forma de bucle para volver a una línea en particular.
- En cambio, podríamos simplemente repetir todo nuestro algoritmo en la mitad del libro que nos queda:

```
1 Recoger directorio telefónico
2 Abrir hasta la mitad de la guía telefónica
3 Mira la página
4 Si Smith está en la página
5 Llama a Mike
6 De lo contrario, si Smith es anterior en el libro
7       Buscar en la mitad izquierda del libro
8
9 De lo contrario, si Smith está más adelante en el libro
10      Buscar en la mitad derecha del libro
11
12 más
13 Salir
```

    - Esto parece un proceso cíclico que nunca terminará, pero en realidad estamos cambiando la entrada a la función y dividiendo el problema por la mitad cada vez, deteniéndonos una vez que no quede más libro.
- En la semana 1, también implementamos una "pirámide" de bloques con la siguiente forma:

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

- Pero observe que una pirámide de altura 4 es en realidad una pirámide de altura 3, con una fila adicional de 4 bloques agregados. Y una pirámide de altura 3 es una pirámide de altura 2, con una fila adicional de 3 bloques. Una pirámide de altura 2 es una pirámide de altura 1, con una fila adicional de 2 bloques. Y finalmente, una pirámide de altura 1 es solo un bloque.
- Con esta idea en mente, podemos escribir una función recursiva para dibujar una pirámide, una función que se llama a sí misma para dibujar una pirámide más pequeña antes de agregar otra fila.

Combinar ordenación

- Podemos llevar la idea de recusión a la clasificación, con otro algoritmo llamado **clasificación por fusión** . El pseudocódigo podría verse así:

```
If only one number
  Return
Else
    Sort left half of number
    Sort right half of number
    Merge sorted halves
```

- Veremos mejor esto en la práctica con dos listas ordenadas:

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

- Vamos a *fusionar* las dos listas para una lista ordenada definitiva, tomando el elemento más pequeño en la parte delantera de cada lista, uno a la vez:

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

1
```

- El 1 en el lado derecho es el más pequeño entre 1 y 3, por lo que podemos comenzar nuestra lista ordenada con él.

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

1 2
```

- El siguiente número más pequeño, entre 2 y 3, es 2, por lo que usamos el 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
```

  - Ahora tenemos una lista completamente ordenada.
- Hemos visto cómo se puede implementar la última línea de nuestro pseudocódigo, y ahora veremos cómo funciona todo el algoritmo:

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

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

- Empezamos con otra lista sin clasificar:

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

- Para empezar, primero debemos ordenar la mitad izquierda:

```
6 3 8 5
```

- Bueno, para ordenar eso, primero tenemos que ordenar la mitad izquierda de la mitad izquierda:

```
6 3
```

- Ahora, estas dos mitades solo tienen un elemento cada una, por lo que ambas están ordenadas. Fusionamos estas dos listas juntas, para una lista ordenada:

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

- Volvemos a ordenar la mitad derecha de la mitad izquierda, fusionándolas:

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

- Ambas mitades de la *mitad izquierda* se han ordenado individualmente, por lo que ahora debemos fusionarlas:

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

- Haremos lo que acabamos de hacer, con la mitad correcta:

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

  - Primero, clasificamos ambas mitades de la mitad derecha.

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

  - Luego, los fusionamos para obtener una mitad derecha ordenada.
  - Finalmente, tenemos dos mitades ordenadas nuevamente, y podemos fusionarlas para obtener una lista completamente ordenada:

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

- Cada número se movió de un estante a otro tres veces (ya que la lista se dividió de 8 a 4, a 2 y a 1 antes de fusionarse nuevamente en listas ordenadas de 2, 4 y finalmente 8 nuevamente). Y cada estante requería que los 8 números se fusionaran, uno a la vez.
- Cada estante requerido n pasos, y solo quedaban $\log n$estantes necesarios, por lo que multiplicamos esos factores juntos. Nuestro tiempo de ejecución total para la búsqueda binaria es$O(\log n)$:
  - $O(n^2)$
    - clasificación de selección, clasificación de burbujas
  - $O(n \log n)$
    - fusionar ordenación
  - $O(n)$
    - búsqueda lineal
  - $O(\log n)$
    - búsqueda binaria

    - $O(1)$
- (Ya que $\log n$ es mayor que 1 pero menor que n, $n \log n$ está en el medio n (multiplicado por 1) y n^2.)
- El mejor caso Ω, es todavía $n \log n$, ya que todavía tenemos que ordenar cada mitad primero y luego fusionarlas:
    - $\Omega(n^2)$
        - orden de selección
    - $\Omega(n \log n)$
        - fusionar ordenación
    - $\Omega(n)$
        - ordenamiento de burbuja
    - $\Omega(\log n)$
    - $\Omega(1)$
        - búsqueda lineal, búsqueda binaria

- Aunque es probable que la ordenación por fusión sea más rápida que la ordenación por selección o la ordenación por burbujas, necesitábamos otro estante, o más memoria, para almacenar temporalmente nuestras listas combinadas en cada etapa. Nos enfrentamos a la compensación de incurrir en un costo más alto, otra matriz en la memoria, en beneficio de una clasificación más rápida.
- Finalmente, hay otra notación, Θ, Theta, que usamos para describir los tiempos de ejecución de los algoritmos si el límite superior y el límite inferior son iguales. Por ejemplo, la ordenación combinada tiene$\Theta(n \log n)$ya que el mejor y el peor de los casos requieren el mismo número de pasos. Y el ordenamiento por selección tiene$\Theta(n^2)$:
    - $\Theta(n^2)$
        - orden de selección
    - $\Theta(n \log n)$
        - fusionar ordenación
    - $\Theta(n)$
    - $\Theta(\log n)$
    - $\Theta(1)$
- Observamos una visualización final (https://www.youtube.com/watch?v=ZZuD6iUe3Pc) de los algoritmos de clasificación con un mayor número de entradas, ejecutándose al mismo tiempo.