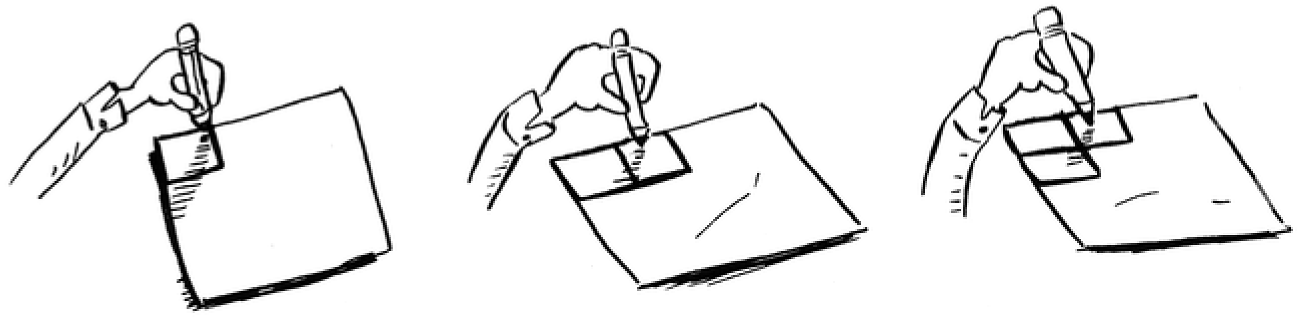
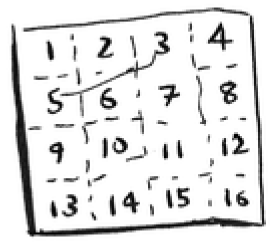
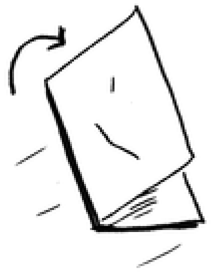
## Introduction

- **Visualizing different Big O run times**.
- Here’s a practical example you can follow at home with a few pieces of paper and a pencil. Suppose you have to draw a grid of 16 boxes.
- **Algorithm 1**
- One way to do it is to draw 16 boxes, one at a time. Remember, Big O notation counts the number of operations. In this example, drawing one box is one operation. You have to draw 16 boxes. How many operations will it take, drawing one box at a time?

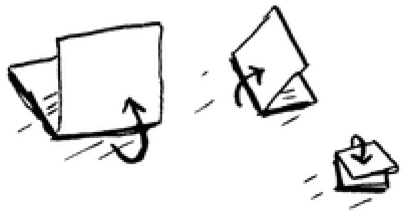



# Introduction

- It takes 16 steps to draw 16 boxes. What's the running time for this algorithm?
- **Algorithm 2**
- Try this algorithm instead. Fold the paper.

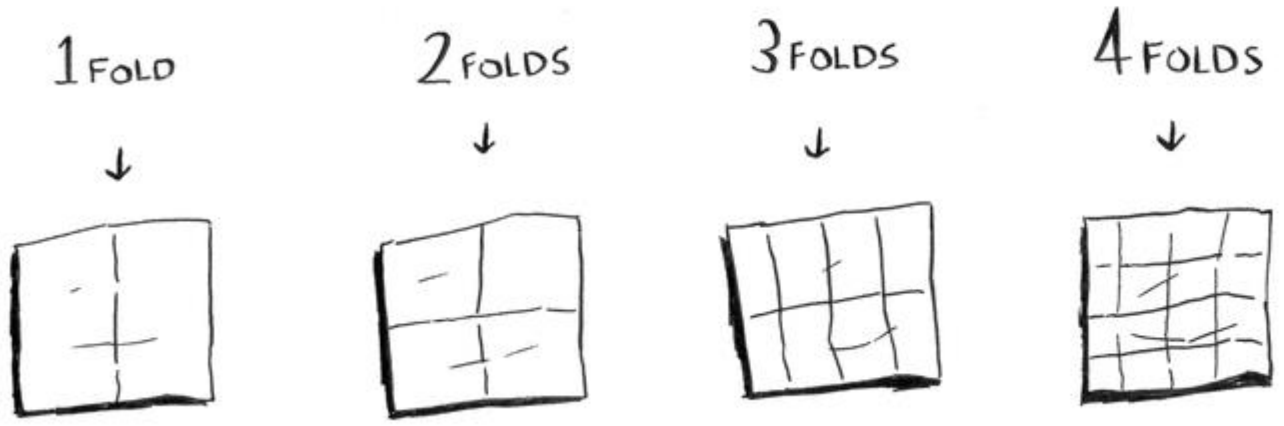


- In this example, folding the paper once is an operation. You just made two boxes with that operation!
- Fold the paper again, and again, and again



# Introduction

- Unfold it after four folds, and you'll have a beautiful grid! Every fold doubles the number of boxes. You made 16 boxes with 4 operations!



- You can "draw" twice as many boxes with every fold, so you can draw 16 boxes in 4 steps. What's the running time for this algorithm? Come up with running times for both algorithms before moving on. Answers: Algorithm 1 takes O(n) time, and algorithm 2 takes O(log n) time.

# Introduction

- **Big O establishes a worst-case run time**
- Suppose you're using simple search to look for a person in the phone book. You know that simple search takes O(n) time to run, which means in the worst case, you'll have to look through every single entry in your phone book.

- In this case, you're looking for AHMED. This guy is the first entry in your phone book. So you didn't have to look at every entry—you found it on the first try. Did this algorithm take O(n) time? Or did it take O(1) time because you found the person on the first try?

- Simple search still takes O(n) time. In this case, you found what you were looking for instantly. That's the best-case scenario. But Big O notation is about the worst-case scenario. So you can say that, in the worst case, you'll have to look at every entry in the phone book once. That's O(n) time. It's a reassurance—you know that simple search will never be slower than O(n) time.

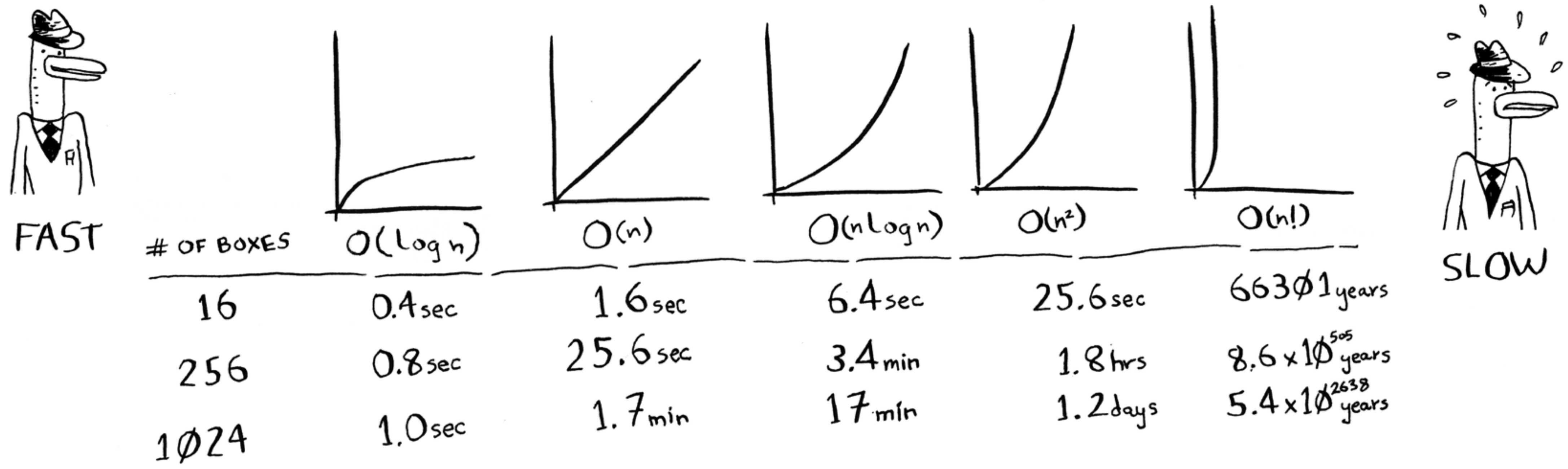
## Introduction

- **Some common Big O run times**
  Here are five Big O run times that you'll encounter a lot, sorted from fastest to slowest:
- O(log *n*), also known as *log time.* Example: Binary search.
- O(*n*), also known as *linear time*. Example: Simple search.
- O(*n* * log *n*). Example: A fast sorting algorithm, like quicksort
- O(*n*2). Example: A slow sorting algorithm, like selection sort
- O(*n*!). Example: A really slow algorithm, like the traveling salesperson (coming up next!).

## Introduction

- Suppose you're drawing a grid of 16 boxes again, and you can choose from 5 different algorithms to do so. If you use the first algorithm, it will take you O(log *n*) time to draw the grid.
- You can do 10 operations per second. With O(log *n*) time, it will take you 4 operations to draw a grid of 16 boxes (log 16 is 4). So it will take you 0.4 seconds to draw the grid. What if you have to draw 1,024 boxes? It will take you log 1,024 = 10 operations, or 1 second to draw a grid of 1,024 boxes. These numbers are using the first algorithm.
- The second algorithm is slower: it takes O(*n*) time. It will take 16 operations to draw 16 boxes, and it will take 1,024 operations to draw 1,024 boxes. How much time is that in seconds?
- Here's how long it would take to draw a grid for the rest of the algorithms, from fastest to slowest:

## Introduction



| # OF BOXES | O(Log n) | O(n) | O(n Log n) | O($n^2$) | O(n!) |
|---|---|---|---|---|---|
| 16 | 0.4 sec | 1.6 sec | 6.4 sec | 25.6 sec | 663Ø1 years |
| 256 | 0.8 sec | 25.6 sec | 3.4 min | 1.8 hrs | $8.6 \times 1Ø^{505}$ years |
| 1Ø24 | 1.0 sec | 1.7 min | 17 min | 1.2 days | $5.4 \times 1Ø^{2638}$ years |

# Introduction

- In reality you can't convert from a Big O run time to a number of operations this neatly, but this is good enough for now.
- Algorithm speed isn't measured in seconds, but in growth of the number of operations.
- Instead, we talk about how quickly the run time of an algorithm increases as the size of the input increases.
- Run time of algorithms is expressed in Big O notation.
- O(log *n*) is faster than O(*n*), but it gets a lot faster as the list of items you're searching grows.

# Time Complexities

- **Time Complexities**
- **Constant Time — O(1)**
- An algorithm is said to have a constant time when it is not dependent on the input data (*n*). No matter the size of the input data, the running time will always be the same. For example:

- if a > b:
-     return True
- else:
-     return False

## Time Complexities

- Now, let’s take a look at the function **get_first** which returns the first element of a list:
- def get_first(data):
-     return data[0]
- 
- data = [1, 2, 9, 8, 3, 4, 7, 6, 5]
-     print(get_first(data))

## Time Complexities

- Independently of the input data size, it will always have the same running time since it only gets the first value from the list.
- An algorithm with constant time complexity is excellent since we don’t need to worry about the input size.
- **Logarithmic Time — O(log n)**
- An algorithm is said to have a logarithmic time complexity when it reduces the size of the input data in each step (it don’t need to look at all values of the input data), for example:

## Time Complexities

- for index in range(0, len(data), 3):
-     print(data[index])

- Algorithms with logarithmic time complexity are commonly found in operations on binary trees or when using binary search. Let’s take a look at the example of a binary search, where we need to find the position of an element in a sorted list:

# Time Complexities

```
• def binary_search(data, value):
•     n = len(data)
•     left = 0
•     right = n - 1
•     while left <= right:
•         middle = (left + right) // 2
•         if value < data[middle]:
•             right = middle - 1
•         elif value > data[middle]:
•             left = middle + 1
•         else:
•             return middle
•     raise ValueError('Value is not in the list')
•     data = [1, 2, 3, 4, 5, 6, 7, 8, 9]
•     print(binary_search(data, 8))
```

## Time Complexities

- Steps of the binary search:
- Calculate the middle of the list.
- If the searched value is lower than the value in the middle of the list, set a new right bounder.
- If the searched value is higher than the value in the middle of the list, set a new left bounder.
- If the search value is equal to the value in the middle of the list, return the middle (the index).
- Repeat the steps above until the value is found or the left bounder is equal or higher the right bounder.

## Time Complexities

- It is important to understand that an algorithm that must access all elements of its input data cannot take logarithmic time, as the time taken for reading input of size *n* is of the order of *n*.
- **Linear Time — O(n)**
- An algorithm is said to have a linear time complexity when the running time increases at most linearly with the size of the input data. This is the best possible time complexity when the algorithm must examine all values in the input data. For example:

## Time Complexities

- for value in data:
-     print(value)
- Let’s take a look at the example of a linear search, where we need to find the position of an element in an unsorted list:
- def linear_search(data, value):
-     for index in range(len(data)):
-         if value == data[index]:
-             return index
-     raise ValueError('Value not found in the list')
-     if __name__ == '__main__':
-     data = [1, 2, 9, 8, 3, 4, 7, 6, 5]
-     print(linear_search(data, 7))
- Note that in this example, we need to look at all values in the list to find the value we are looking for.

## Linear Search

- **Algorithm and Code Example**
- The algorithm for linear search can be described in the following steps:

1. Start at the beginning of the array.
2. Compare the current element with the target value.
3. If the current element matches the target value, return the index of the current element.
4. If the current element does not match the target value, move to the next element in the array.
5. Repeat steps 2-4 until the end of the array is reached or the target value is found.

- Here's a Python code example that demonstrates a simple linear search:

# Linear Search

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
array = [4, 2, 1, 8, 5, 7, 6, 3]
target = 8
result = linear_search(array, target)
if result != -1:
    print("Element found at index:", result)
else:
    print("Element not found in the array")
```

## Linear Search

- **Time Complexity**
- The time complexity of linear search is O(n), where n is the number of elements in the array. In the worst-case scenario, the algorithm has to check all elements before finding the target value, making it inefficient for large datasets.

## Binary Search

- **Introduction to Binary Search**
- Binary search is a more efficient searching algorithm compared to linear search. It works by taking advantage of a sorted array or list, repeatedly dividing the search interval in half. The algorithm compares the middle element of the interval to the target value, narrowing down the search space based on the comparison result. Binary search is significantly faster than linear search for large datasets, as it eliminates a large portion of the data with each comparison.

# Binary Search

- Algorithm and Code Example
- The algorithm for binary search can be described in the following steps:
- Set the lower bound low to the first index of the array and the upper bound high to the last index.
- While low is less than or equal to high:
- Calculate the middle index mid as the average of low and high.
- Compare the element at index mid with the target value.
- If the element at index mid matches the target value, return the index mid.
- If the element at index mid is less than the target value, set low to mid + 1.
- If the element at index mid is greater than the target value, set high to mid - 1.
- If the target value is not found, return -1.
- Here's a Python code example that demonstrates a simple binary search:

## Binary Search

```
•  def binary_search(arr, target):
•      low, high = 0, len(arr) - 1
•      while low <= high:
•          mid = (low+ high) // 2
•          if arr[mid] == target:
•              return mid
•          elif arr[mid] < target:
•              low = mid + 1
•          else:
•              high = mid - 1
•      return -1
•  array = [1, 2, 3, 4, 5, 6, 7, 8]
•  target = 8
•  result = binary_search(array, target)
•  if result != -1:
•      print("Element found at index:", result)
•  else:
•      print("Element not found in the array")
```

## Binary Search

- **Time Complexity**
- The time complexity of binary search is O(log n), where n is the number of elements in the array. With each comparison, the search space is reduced by half, making it a highly efficient algorithm for searching large datasets.

## Binary Search vs. Linear Search

- Efficiency: Binary search is more efficient than linear search, especially for large datasets. Binary search has a time complexity of O(log n), while linear search has a time complexity of O(n).
- Algorithm Complexity: Linear search is a simpler algorithm compared to binary search, making it easier to understand and implement.
- Number of Comparisons: Linear search may require up to n comparisons (where n is the number of elements in the array), while binary search requires at most log2(n+1) comparisons.

## Linear Search

```
def linearSearch(array, n, x):
    for i in range(0, n):
        if (array[i] == x):
            return i
    return -1
array = [24, 41, 31, 11, 9]
x = 11
n = len(array)
result = linearSearch(array, n, x)
if(result == -1):
    print("Element not found")
else:
    print("Element is Present at Index: ", result)
```

# Binary Search

```
•   def binarySearch(array, x, low, high):
•           while low <= high:
•                   mid = low + (high - low)//2
•                   if array[mid] == x:
•                           return mid
•                   elif array[mid] < x:
•                           low = mid + 1
•                   else:
•                           high = mid - 1
•           return -1
•   array = [2, 4, 5, 7, 14, 17, 19, 22]
•   x = 22
•   result = binarySearch(array, x, 0, len(array)-1)
•   if result != -1:
•           print(str(result))
•   else:
•           print("Not found")
```

## Linear Search

- Let’s say our **query point (q)** is 5 and **size of list (n)** is 15.
- While doing **linear search** to find query point 5 in the given list, three cases may occur:

1. *If 5 exists in the list at index **k**, program does **k + 1** comparisons.*
2. *If 5 exists in the list at index **(n — 1),** program does **n** comparisons.*
3. *If 5 does not exist in the list, program does **n** comparisons.*

## Linear Search

- As is seen, to find the query point, at the worst case, program needs to check sequentially **all elements** of the list (n comparisons). As the size of the list increases the number of the comparisons will increase proportionally. Let’s show things practically on **Python**;

## Linear Search

- # Imports
- import numpy as np
- import random # Create the shuffled List of numbers from 0 to 15
- l = list(range(15))
- random.shuffle(l)# Display the shuffled list
- l
- [12, 3, 4, 2, 14, 9, 11, 6, 1, 10, 0, 7, 13, 5, 8]

## Linear Search

- Let’s write **the program** that does linear search to find the query point from the list.
- # Search for an element q in the list: O(n) where n is the length of the listq = 5 # 1 unit of time
- isFound = False # 1 unit of time
- for x in l: # n times
-     if x == q: # 1 unit of time
-         print('Found') # 1 unit of time
-         isFound == True # 1 unit of time
-         break # 1 unit of time
- if isFound == False: # 1 unit of time
-     print('Not Found') # 1 unit of time

## Linear Search

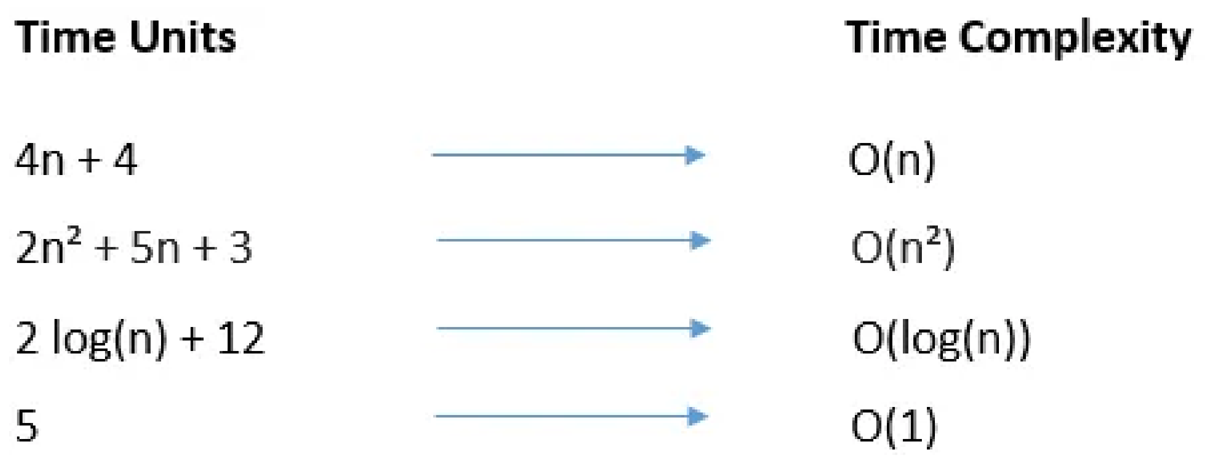
- I’ve commented in front of each row of the program to show how many time units were needed for each row to run. Let’s calculate total time units needed for our model to run:

- 1 + 1 + n*( 1 + 1 + 1 + 1 ) + 1+ 1 = 4n + 4 ( n is the size of list)

## Linear Search

- It is seen that time complexity of our program is proportionate to **n**, so **time complexity** of the program is **order of n** (i.e. O(n)).
- As you see **time units** and **time complexity** are not the same, we need to do some conversions.
- *The main thing to find time complexity is to ignore the constants from calculated time units and choose the highest term that the algorithm run time is proportionate to.* Here are the some examples:

## Linear Search



| Time Units | | Time Complexity |
|---|---|---|
| $4n + 4$ | $\rightarrow$ | $O(n)$ |
| $2n^2 + 5n + 3$ | $\rightarrow$ | $O(n^2)$ |
| $2 \log(n) + 12$ | $\rightarrow$ | $O(\log(n))$ |
| $5$ | $\rightarrow$ | $O(1)$ |

## Linear Search

- Linear Search when done sequentially requires us to read all the elements in the list one by one and compare it with the required result. Since we don’t require any extra space, except the iterator, i.e. x, we require constant extra space for linear search, i.e irrespective of the size of the list, we require the same memory. Thus, **space complexity** is O(1).

## Linear Search

- We found that our program has time complexity of O(n) and space complexity of O(1). In some cases, there is a trade-off between time complexity and space complexity. Let’s say that my system has not any space problem and I need to lower the time complexity. Then, what is the possible way to lower the time complexity?

## Binary Search

- We saw that with the program using linear search, it is possible find q (query point) from list of size n with time complexity of O(n) and space complexity of O(1). The time complexity can be lowered to **O(log(n))** using binary search. Let’s see how the **binary search** works.
- Let’s see the shuffled list again:
- [12, 3, 4, 2, 14, 9, 11, 6, 1, 10, 0, 7, 13, 5, 8]

## Binary Search

- In order to be able to do binary search, our list must be sorted. Then let’s sort our list in Python:
- # Sort the list
- l.sort()# Display the sorted list
- l
- The new sorted list is:
- [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

- While program does Binary Search, it searches the sorted list by repeatedly dividing the search interval in half. Let’s see how the program divides the search interval internally until it finds the query point.

## Binary Search

- [0,1,2,3,4,5,6,7,8,9,10,11,12,13,14]    # Before comparison
- [0,1,2,3,4,5,6]    # After 1st comparison
- [3,4,5,6]    # After 2nd comparison
- [4,5,6]    # After 3rd comparison
- [5,6]    # After 4th comparison
- [5]    # After 5th comparison

## Binary Search

- In order to divide search interval, the program calculates the middle point of each interval. After finding the middle point, there can be three cases:

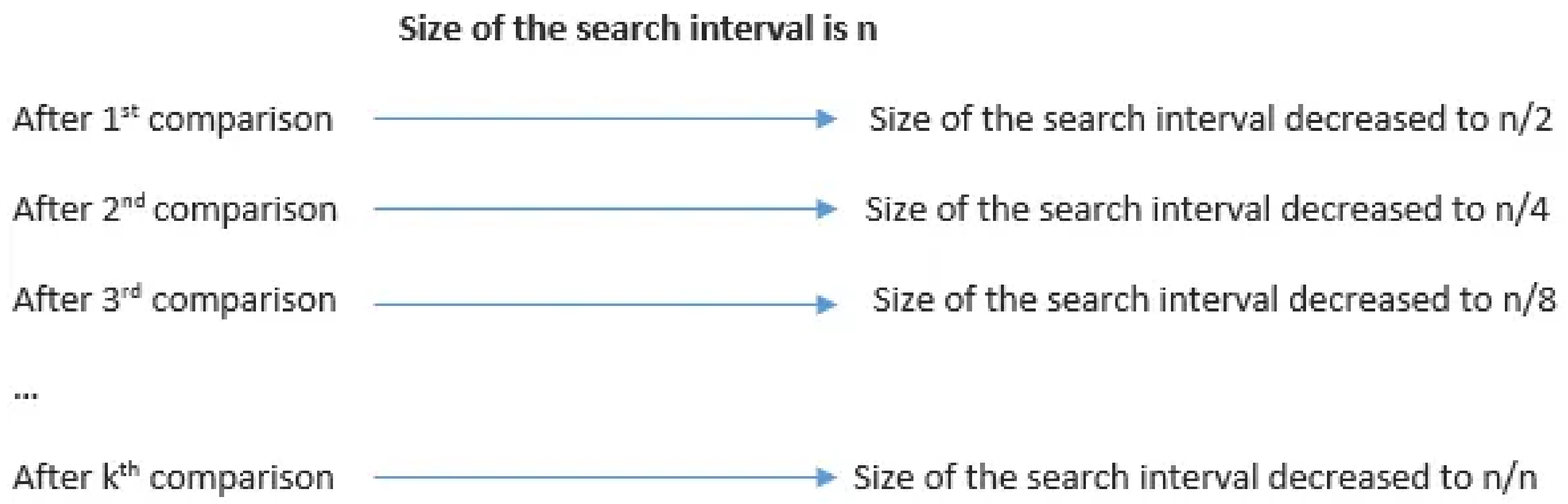
- If query point presents at the middle, then the search finishes.
- If query point is smaller than mid-point, then the program continues with the left sublist.
- If query point is higher than mid-point, then the program continues with the right sublist.

```
• # Import math library
• import math# Returns index of x in list if present, else -1
• def binarySearch(list, l, r, x)
•     # @list the sorted list
•     # @l the left border
•     # @r the right border
•     # @x the query point    # Check base case
•     if r >= l:
•         # Find the mid point
•         mid = l + math.floor((r - l)/2)       # If element is present at the middle itself
•         if list[mid] == x:
•             return mid
•               # if element is smaller than mid, then it can only
•         # be present in left sublist
•         elif list[mid] > x:
•             return binarySearch(list, l, mid-1, x)       # Else the element can only be present in right sublist
•         else:
•             return binarySearch(list, mid+1, r, x)    else:
•         # Element is not present in the list
•         # return -1list = l
• q = 5# Call the function
• binarySearch(list, 0, len(list)-1, q)
```

## Binary Search

- As it is seen, the program doesn’t check all the list elements to find the query point, what it does is just dividing the search interval into half intervals until the interval has only one element.
- Now, it is time to understand how the time complexity decreases from O(n) to O(log(n)). Let’s say we have the list of size n. At the first loop, the program does only 1 comparison and the size of the search interval decreases to n/2. As the number of comparisons increases, the size of the search interval decreases. Let’s see the process:

# Binary Search



**Size of the search interval is n**

After 1st comparison $\longrightarrow$ Size of the search interval decreased to n/2

After 2nd comparison $\longrightarrow$ Size of the search interval decreased to n/4

After 3rd comparison $\longrightarrow$ Size of the search interval decreased to n/8

...

After kth comparison $\longrightarrow$ Size of the search interval decreased to n/n

## Binary Search

- As it is seen the comparisons will continue until there is only 1 (n/n) element in the search interval. Then, let’s ask the questions to get the insights from the process table. If n = 8, how many comparisons are needed to find the query point? What if the size of the list is 16?

- If n is 8, then the program will do 3 comparisons. If the size is 16 then the amount of comparisons will increase only one unit and it will be 4. Is there any relationship? Yes, there is a logarithmic relationship. It was required to do n (size of the list) comparisons when the program did linear search. Now, it will be required to do log(n) comparisons when the program does binary search. So we see that time complexity dropped to O(log(n)) when the binary search was applied.

- When using both searching methods, the runtime of the program was less than 1 second. In this case, it seems that we didn’t achieved so much difference by using binary search, but we shouldn’t forget that this was just one of the simplest examples to explain the time and space complexity and showing the possibility of reducing them.

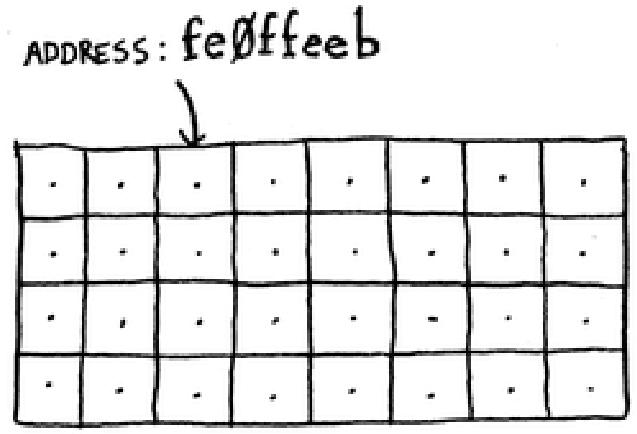
# Introduction

- **How memory works**

  Each drawer can hold one element. You want to store two things, so you ask for two drawers.

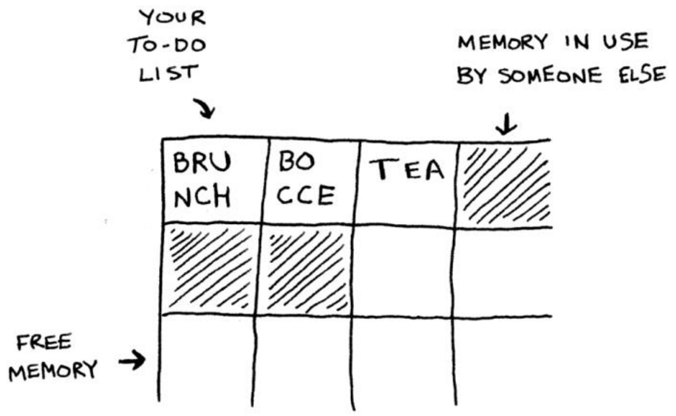
- Each drawer can hold one element. You want to store two things, so you ask for two drawers.

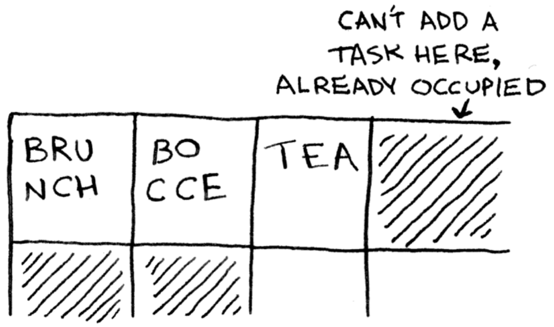
- You store your two things here.
- And you're ready for the show! This is basically how your computer's memory works. Your computer looks like a giant set of drawers, and each drawer has an address.
- fe0ffeeb is the address of a slot in memory.
- Each time you want to store an item in memory, you ask the computer for some space, and it gives you an address where you can store your item. If you want to store multiple items, there are two basic ways to do so: arrays and lists.
- There isn't one right way to store items for every use case, so it's important to know the differences.



# Arrays and linked lists

- Sometimes you need to store a list of elements in memory. Suppose you're writing an app to manage your todos. You'll want to store the todos as a list in memory.
- Should you use an array, or a linked list? Let's store the todos in an array first, because it's easier to grasp. Using an array means all your tasks are stored contiguously (right next to each other) in memory.



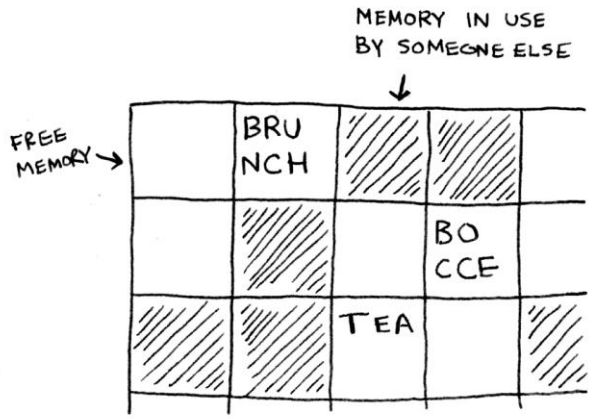


- Now suppose you want to add a fourth task. But the next drawer is taken up by someone else's stuff!

# Arrays and linked lists

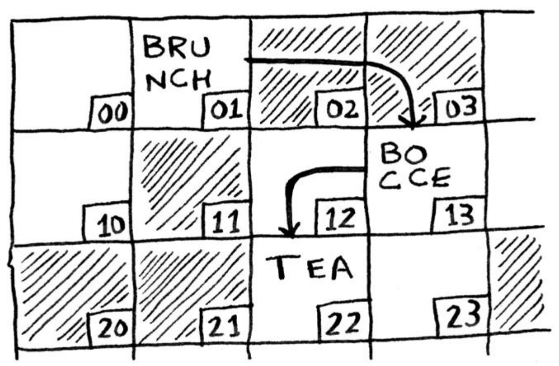
- you can ask the computer for 10 slots, just in case. Then you can add 10 items to your task list without having to move. This is a good workaround, but you should be aware of a couple of downsides:
- You may not need the extra slots that you asked for, and then that memory will be wasted. You aren't using it, but no one else can use it either.
- You may add more than 10 items to your task list and have to move anyway.
- So it's a good workaround, but it's not a perfect solution. Linked lists solve this problem of adding items.

  With linked lists, your items can be anywhere in memory



## Arrays and linked lists

- Each item stores the address of the next item in the list. A bunch of random memory addresses are linked together.



- You go to the first address, and it says, “The next item can be found at address 123.” So you go to address 123, and it says, “The next item can be found at address 847,” and so on. Adding an item to a linked list is easy: you stick it anywhere in memory and store the address with the previous item.

# Arrays and linked lists

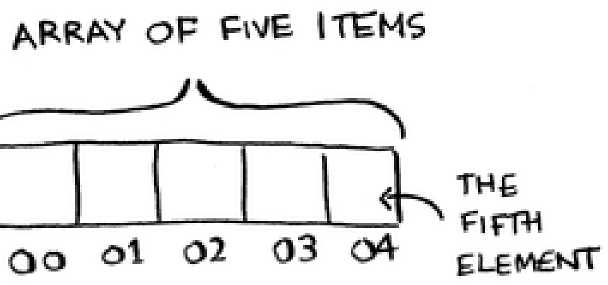
- With linked lists, you never have to move your items. You also avoid another problem. Let’s say you go to a popular movie with five of your friends. The six of you are trying to find a place to sit, but the theater is packed. There aren’t six seats together. Well, sometimes this happens with arrays. Let’s say you’re trying to find 10,000 slots for an array. Your memory has 10,000 slots, but it doesn’t have 10,000 slots together. You can’t get space for your array! A linked list is like saying, “Let’s split up and watch the movie.” If there’s space in memory, you have space for your linked list.
- If linked lists are so much better at inserts, what are arrays good for?

## Arrays and linked lists

- Suppose you want to read the last item in a linked list. You can’t just read it, because you don’t know what address it’s at. Instead, you have to go to item #1 to get the address for item #2. Then you have to go to item #2 to get the address for item #3. And so on, until you get to the last item. Linked lists are great if you’re going to read all the items one at a time: you can read one item, follow the address to the next item, and so on. But if you’re going to keep jumping around, linked lists are terrible.

# Arrays and linked lists

- Arrays are different. You know the address for every item in your array. For example, suppose your array contains five items, and you know it starts at address 00. What is the address of item #5?

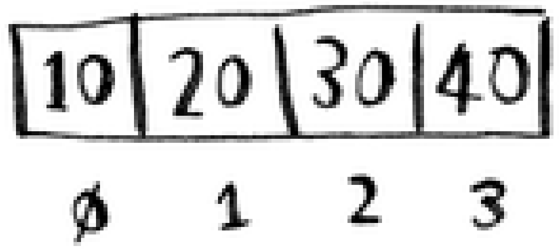


## Arrays and linked lists

- Simple math tells you: it's 04. Arrays are great if you want to read random elements, because you can look up any element in your array instantly.
- With a linked list, the elements aren't next to each other, so you can't instantly calculate the position of the fifth element in memory—you have to go to the first element to get the address to the second element, then go to the second element to get the address of the third element, and so on until you get to the fifth element.

## Arrays and linked lists

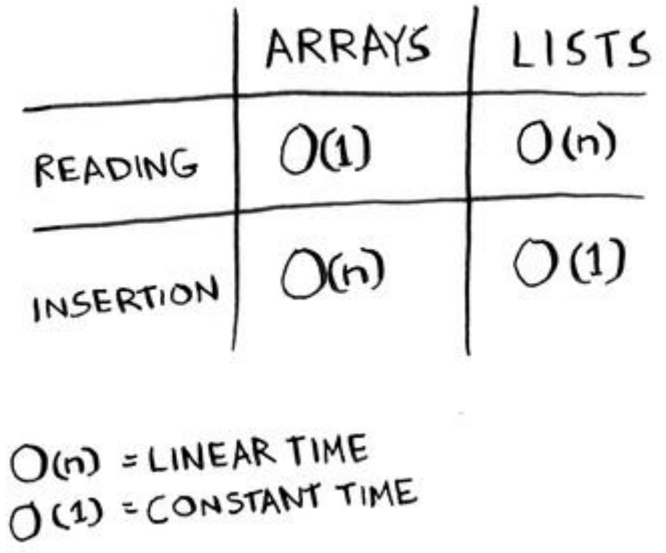
- The elements in an array are numbered. This numbering starts from 0, not 1. For example, in this array, 20 is at position 1.



- Starting at 0 makes all kinds of array-based code easier to write, so programmers have stuck with it. Almost every programming language you use will number array elements starting at 0. You'll soon get used to it.

# Arrays and linked lists

- The position of an element is called its index. So instead of saying, “20 is at position 1,” the correct terminology is, “20 is at index 1.
- Here are the run times for common operations on arrays and lists.

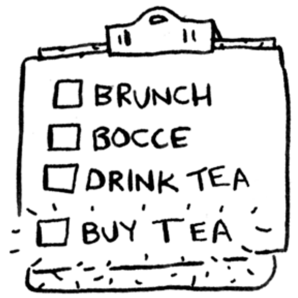


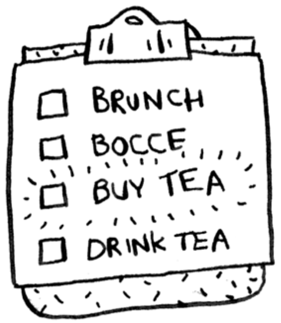
| | ARRAYS | LISTS |
|---|---|---|
| READING | O(1) | O(n) |
| INSERTION | O(n) | O(1) |

O(n) = LINEAR TIME
O(1) = CONSTANT TIME

## Arrays and linked lists

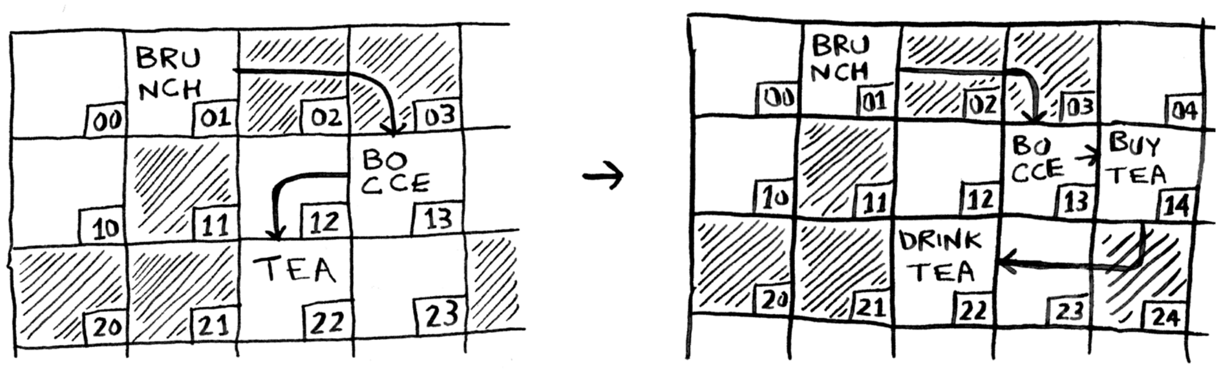
- Question: Why does it take O(n) time to insert an element into an array? Suppose you wanted to insert an element at the beginning of an array. How would you do it? How long would it take?
- **Inserting into the middle of a list**
- Suppose you want your todo list to work more like a calendar.
- Earlier, you were adding things to the end of the list.
- Now you want to add them in the order in which they should be done.





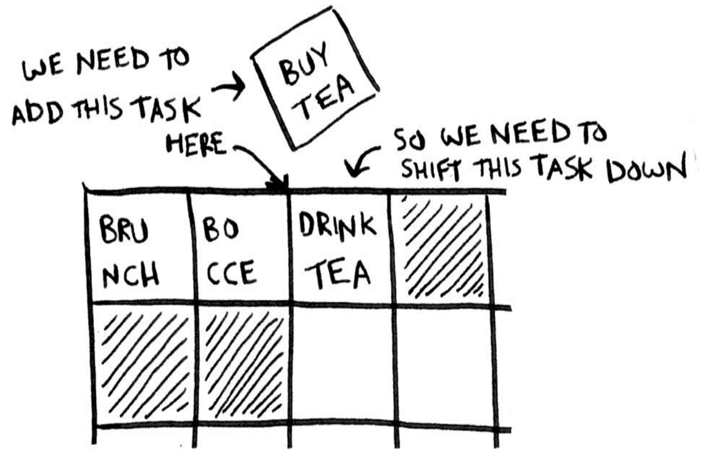
# Arrays and linked lists

- What's better if you want to insert elements in the middle: arrays or lists? With lists, it's as easy as changing what the previous element points to.



## Arrays and linked lists

- But for arrays, you have to shift all the rest of the elements down.


WE NEED TO ADD THIS TASK HERE
BUY TEA
SO WE NEED TO SHIFT THIS TASK DOWN
BRU NCH
BO CCE
DRINK TEA

- And if there’s no space, you might have to copy everything to a new location! Lists are better if you want to insert elements into the middle.