Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice

C++ONLINE

MIKE SHAH

THE STRATEGY DESIGN PATTERN

2024

The Strategy Design Pattern

-- Design Patterns

with Mike Shah

18:00 - 19:00 UTC Sat. Mar 2, 2024

60 minutes + 15 minute Q&A After Introductory Audience

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

Your Tour Guide for Today

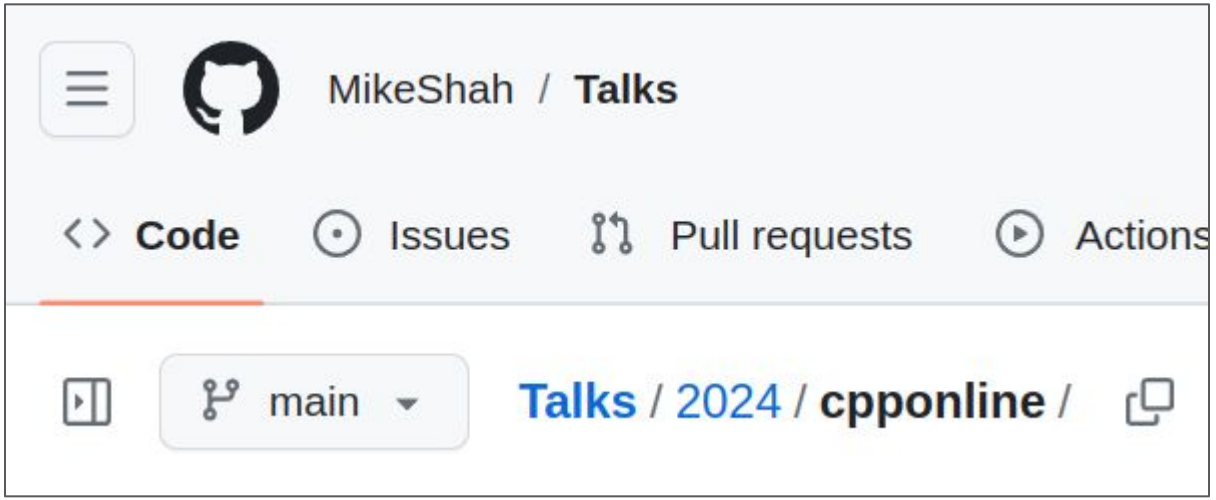
by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
  - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
  - My research is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do consulting and technical training on modern C++, DLang, Concurrency, and Graphics Programming
  - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2024/cpponline



C++ONLINE

Abstract

The abstract that you read and enticed you to join me is here!

The strategy design pattern is a fundamental behavioral design pattern allowing clients to control the behavior of an object at run-time. The strategy pattern can be seen in action in the C++ STL for example applying 'policies' to how to execute operations (e.g. std::par or std::seq as one example) -- thus strategy is often known as the 'policy pattern' as well. The strategy design pattern is commonly used to choose the algorithm at run-time to best solve a problem -- separating the algorithmic behavior from the object itself to help make our software more maintainable, extensible, and flexible. In this talk, we will look at a series of C++ examples (starting from scratch) of using the strategy pattern to deploy different algorithms at run-time. We will also look at examples of code in libraries deploying the strategy pattern, and talk about best practices for using the strategy pattern in Modern C++. The strategy pattern will also be compared to the Template Method Pattern, which may be an alternative choice. Attendees will leave this talk with the knowledge to go forward and implement the strategy pattern, as well as how to spot the strategy design pattern in projects they may already be working on!

The Software Interview

The Software Interview Process (1/3)

- Question to audience:
  - How many of you have ever described the software interview process to a non-software engineering friend?
- Likely, then you've explained how a typical interview involves:
  - Answering some questions (phone-screen)
  - One or more rounds of coding questions
  - And perhaps other components (take-home test, presentation, system design exercise, etc.)
- My personal experience has involved a lot of the whiteboard shown to the right.

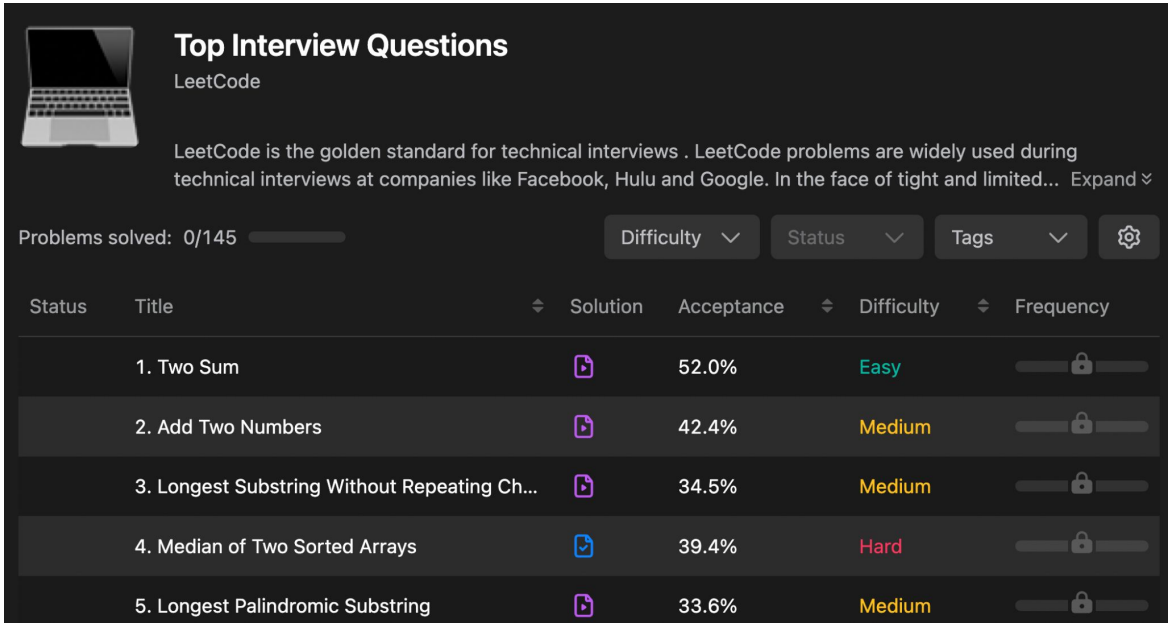
The Software Interview Process (2/3)

- Question to audience:
  - How many of you have ever described the software interview process to a non-software engineering friend?
- Likely, then you've explained how a typical interview involves:
  - Answering some questions (phone-screen)
  - One or more rounds of coding questions
  - And perhaps other components (take-home test, presentation, system design exercise, etc.)
- My personal experience has involved a lot of the whiteboard shown to the right.

The Software Interview Process (3/3)

- Question to audience:
  - How many of you have ever described the software interview process to a non-software engineering friend?
- Likely, then you've explained how a typical interview involves:
  - Answering some questions (phone-screen)
  - One or more rounds of coding questions
  - And perhaps other components (take-home test, presentation, system design exercise, etc.)
- My personal experience has involved a lot of the whiteboard shown to the right.

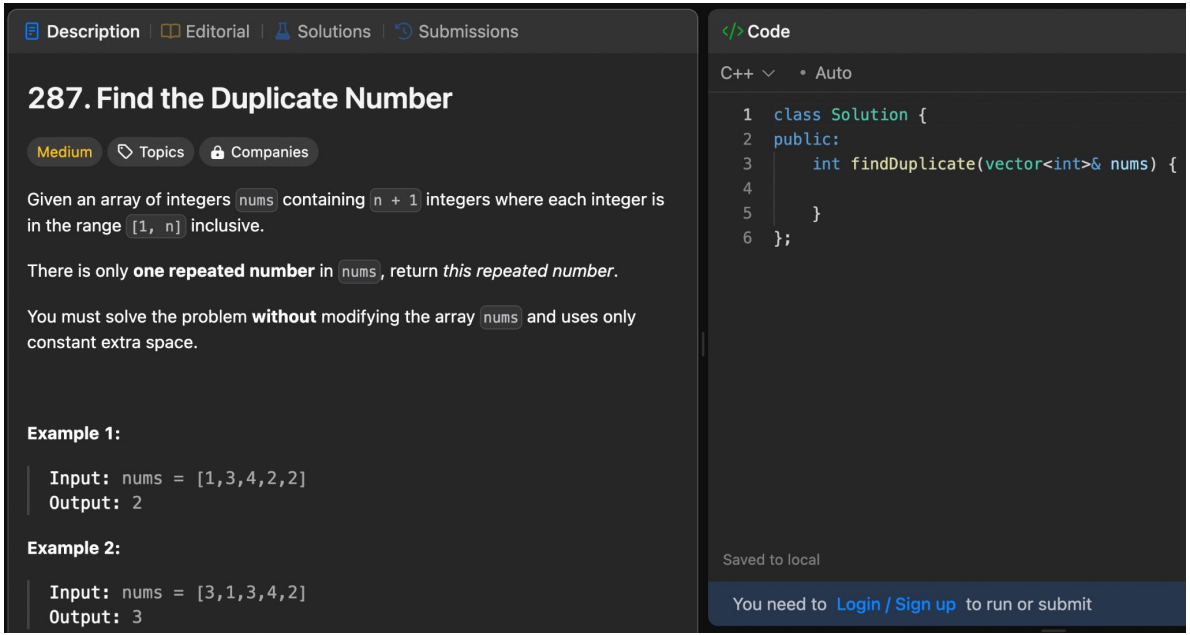
Let’s Look at a Top Interview Question



https://leetcode.com/problem-list/top-interview-questions/

Sample Problem (1/2)

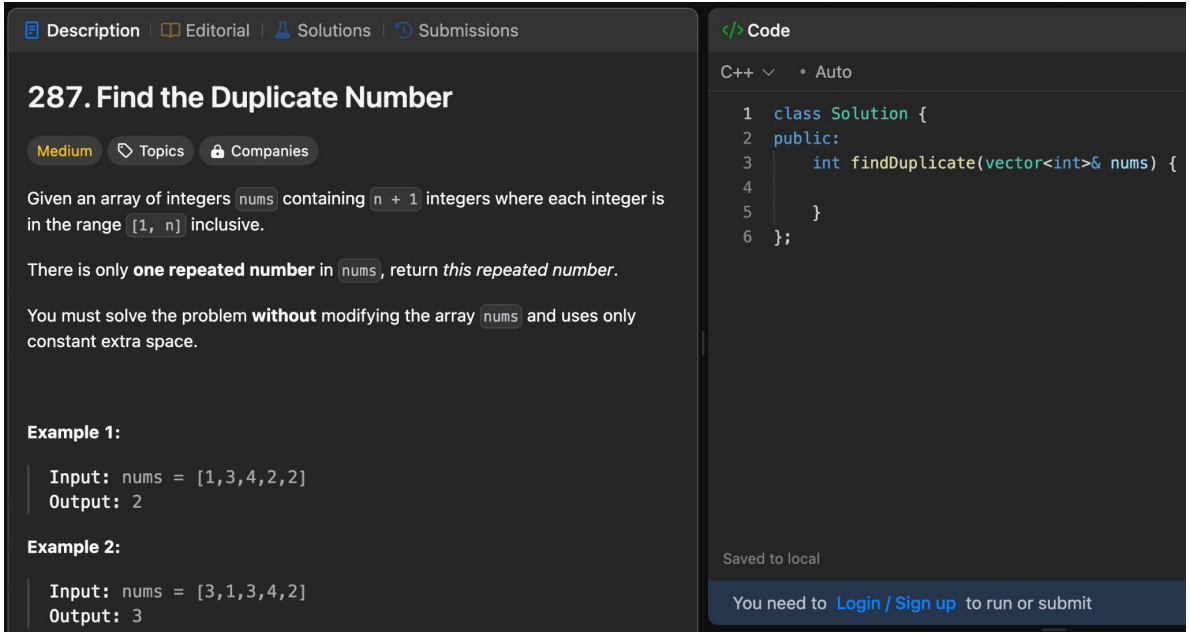
- So here's a sample problem
  - Finding duplicate numbers in a given vector.



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Sample Problem (2/2)

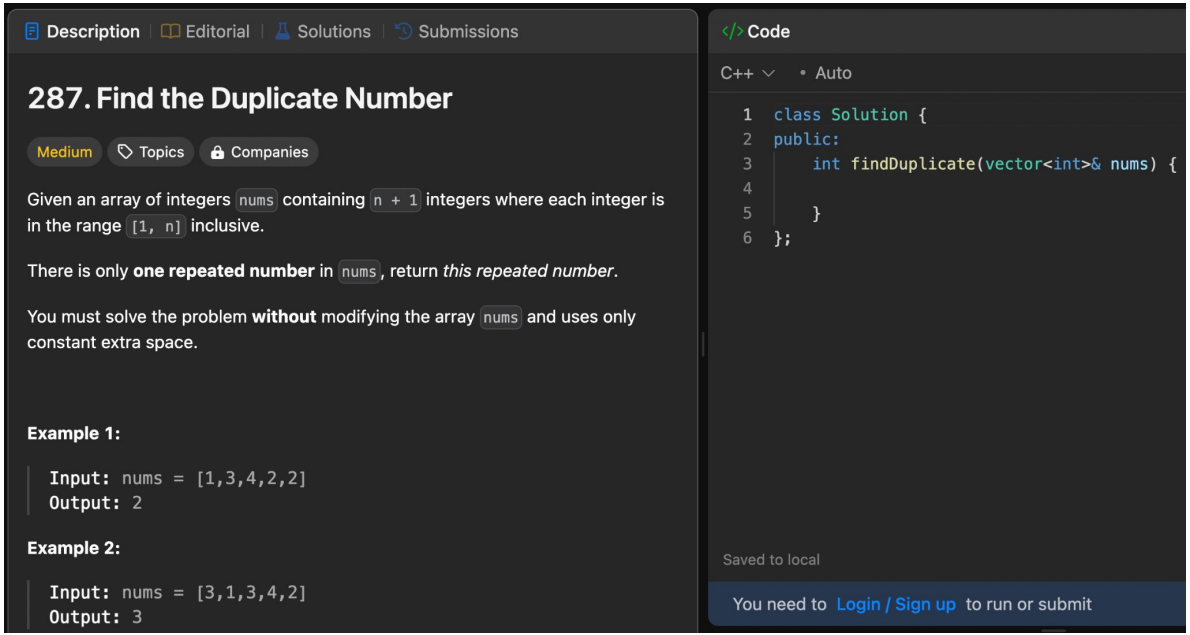
- Question to audience:
  - Can you think of at least one way to solve this?
  - Or maybe two ways?



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

(Aside) Interview Tip (1/2)

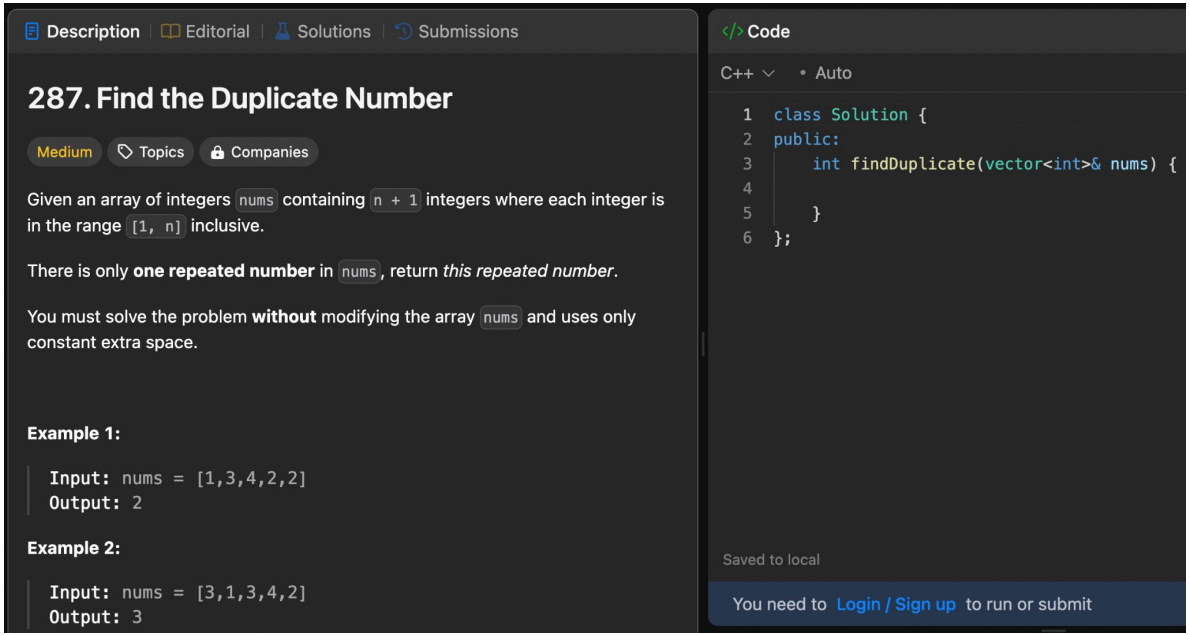
- Something I was advised was to make sure you get at least one solution during the interview (if you cannot be optimal)
- So let's start with 'brute force' strategy



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

(Aside) Interview Tip (2/2)

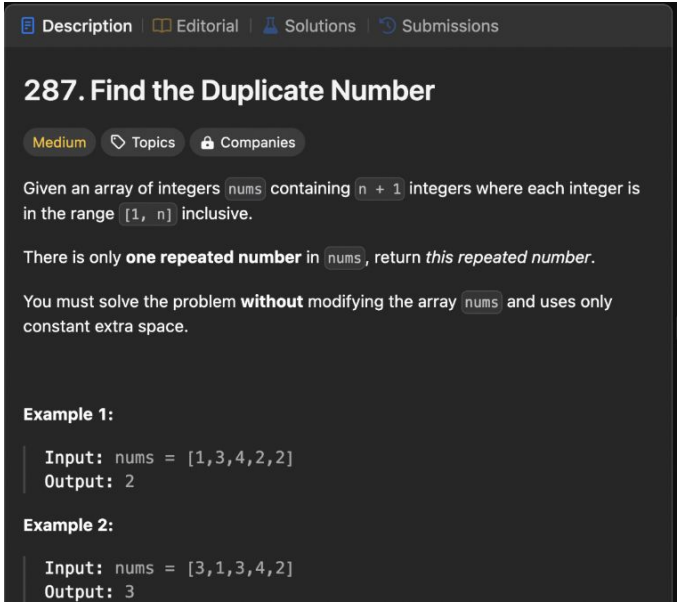
- My second tip, is to ask probing questions:
  - e.g. What is the size of the array
  - Are the integers 4-bytes, or are they 1 byte
  - etc.



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Brute Force Solution (1/4)

- Copy vector
- use ‘sort’ on copied vector
- Iterate through copied vector, until the ‘n+1’ element is equal to the nth element.
- Done?



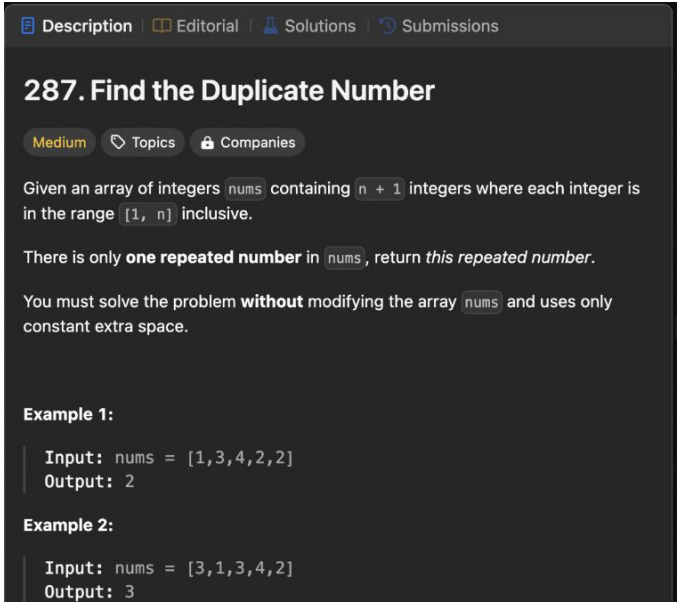
https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Brute Force Solution (2/4)

- Copy vector
- use ‘sort’ on copied vector
- Iterate through copied vector, until the ‘n+1’ element is equal to the nth element.
- Done?

Follow up:

- How can we prove that at least one duplicate number must exist in `nums`?
- Can you solve the problem in linear runtime complexity?



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

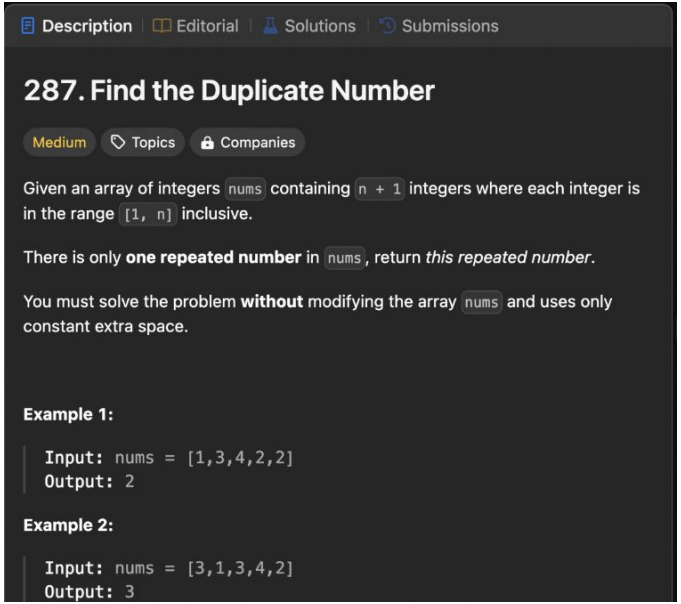
Brute Force Solution (3/4)

- Copy vector
- use ‘sort’ on copied vector
- Iterate through copied vector, until the ‘n+1’ element is equal to the nth element.
- Done?

Follow up:

- How can we prove that at least one duplicate number must exist in `nums`?
- Can you solve the problem in linear runtime complexity?

This suggests we can have a more optimal strategy, or rather should think carefully about our algorithms for things like ‘copy’ and ‘sort’



https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Brute Force Solution (4/4)

- Copy vector
- use ‘sort’ on copied vector
- Iterate through copied vector, until the ‘n+1’ element is equal to the nth element.
- Done?

Description | Editorial | Solutions | Submissions

287. Find the Duplicate Number

Medium | Topics | Companies

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

Follow up:

- How can we prove that at least one duplicate number must exist in `nums`?
- Can you solve the problem in linear runtime complexity?

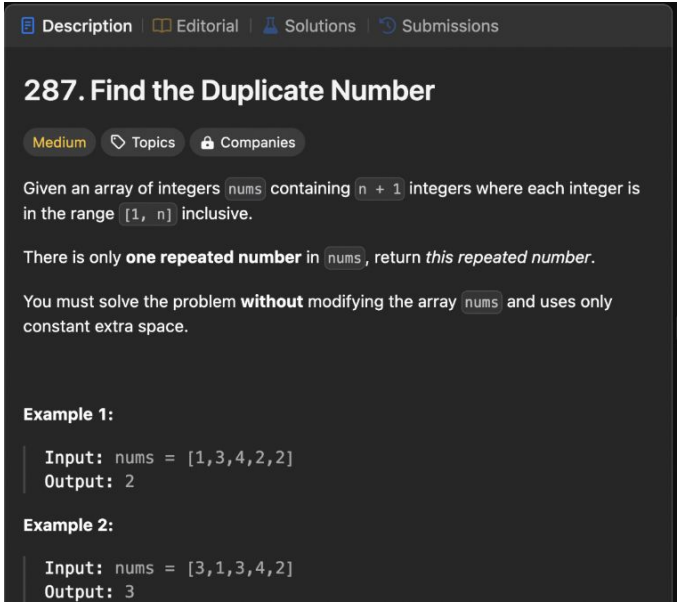
ured-list&envId=top-inte

https://miro.medium.com/v2/resize:fit:807/1*3KU5GNKBpUbe7qnWLHHmuQ.png

This suggests we can have a more optimal strategy, or rather should think carefully about our algorithms for things like ‘copy’ and ‘sort’

Sample Problem - More Solutions

- Some more possible algorithms:
  - iterate through nums and store each value in an unordered_set
    - Test for membership before adding, and if '.contains' returns true, you found your duplicate
  - (Same as above, but with multiset)
    - iterate through nums and store each value in unordered_multiset, if count > 1, then you found a duplicate
  - Copy vector, Radix Sort then use adjacent_find to return iterator to first pair.
  - Floyd's Cycle Detection Algorithm (Tortoise and Hare) -- **Optimal solution** [link]



Description | Editorial | Solutions | Submissions

287. Find the Duplicate Number

Medium | Topics | Companies

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

```
Input: nums = [1,3,4,2,2]
Output: 2
```

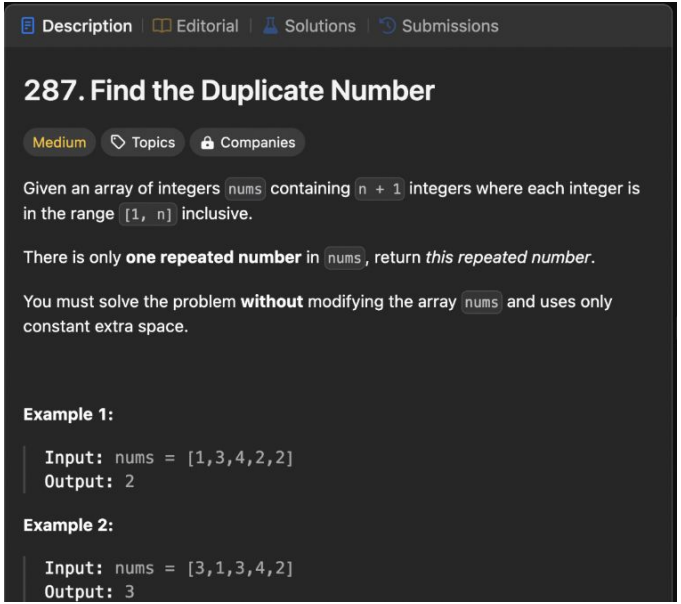
Example 2:

```
Input: nums = [3,1,3,4,2]
Output: 3
```

https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Constrained Problem, One Optimal Solution

- So for these types of constrained problems, we likely will strive to find an optimal solution for our task.
  - Ultimately we can arrive at a linear time complexity and constant space solution.
    - Note: We could maybe ask more probing questions (e.g. how many integers, size of integer, do we run this code frequently, etc.) to explore perhaps even more optimizations.
- Software development however often comes with trade-offs -- the solutions are not always so clean cut
  - Let's explore more in the next slide!



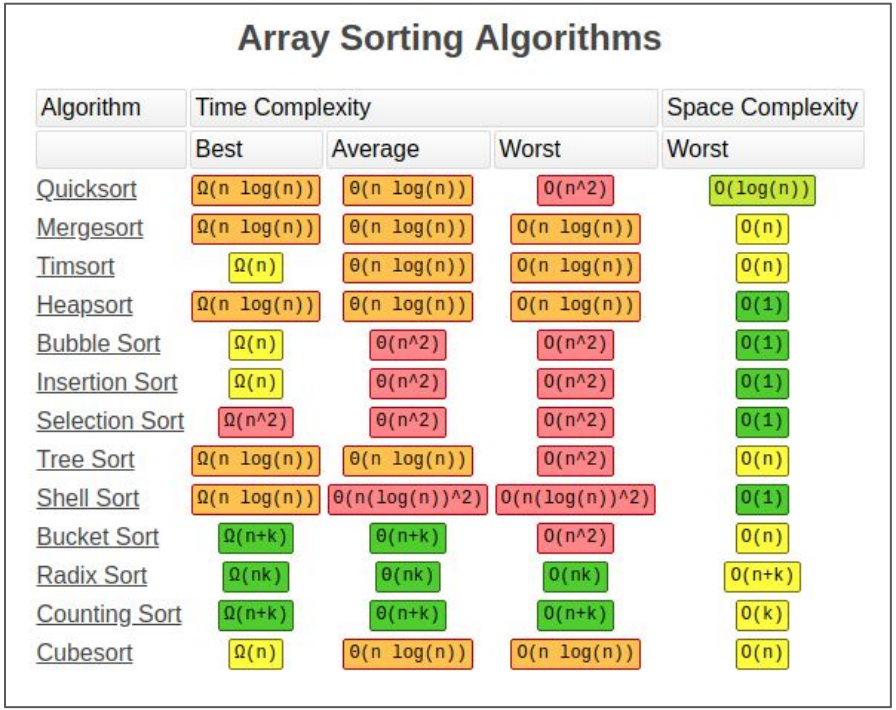
https://leetcode.com/problems/find-the-duplicate-number/description/?envType=featured-list&envId=top-interview-questions?envType=featured-list&envId=top-interview-questions

Sort

The standard problem of most algorithms textbooks

Sorting Algorithms

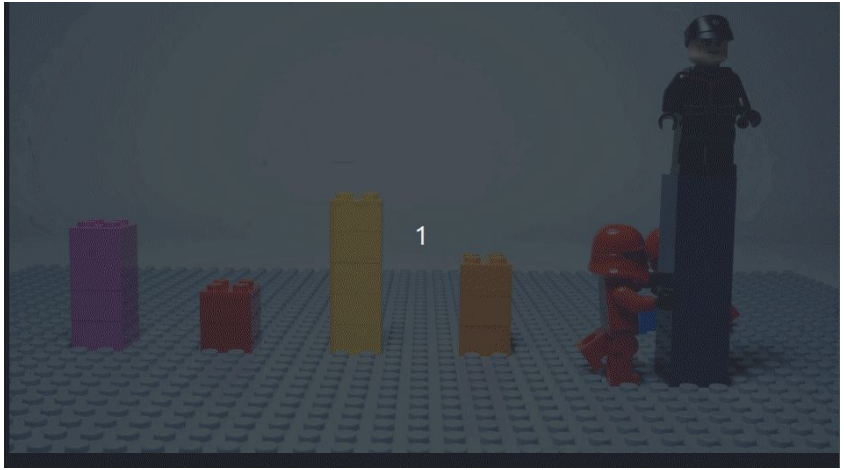
- The truth is, when you learn sorting algorithms in university (or a textbook, or after doing enough leetcode) -- a reason they are taught is because you have to start thinking about the trade-offs.
  - They are ‘interesting’ because there are many strategies to solving the problem, and we can practice thinking about working with data in different ways.



https://lamfo-unb.github.io/img/Sorting-algorithms/Complexity.png

Sorting (1/3)

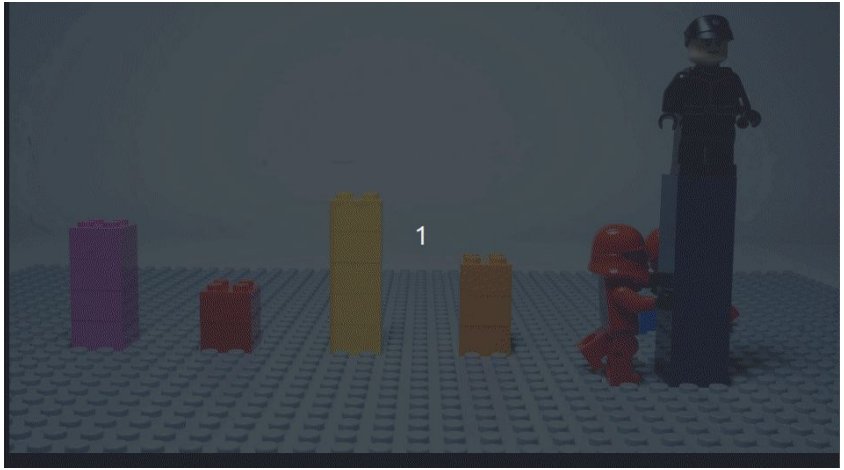
- Given:
  - N integers
  - sort them as fast as possible
- Question to Audience:
  - What solutions might you come up with?



LEGO stop motion sorting algorithm
https://www.youtube.com/watch?v=XN0saL9ujLE

Sorting (2/3)

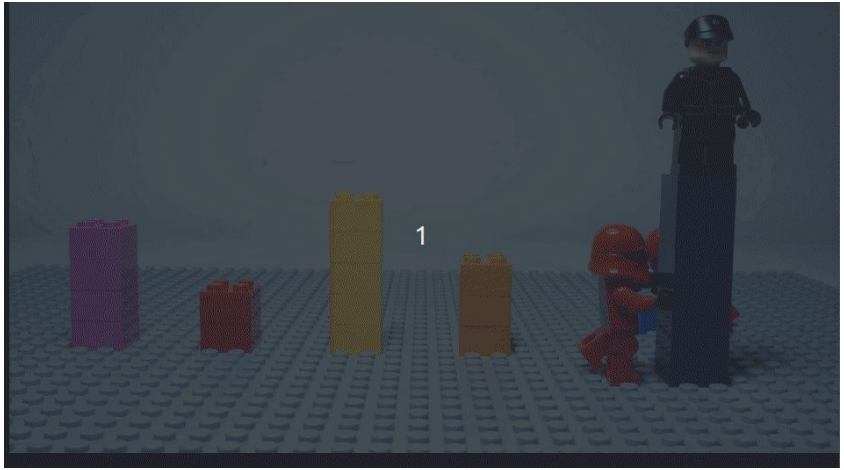
- Given:
  - N integers
  - sort them as fast as possible
- Question to Audience:
  - What solutions might you come up with?
    - Quicksort?
    - Radix sort again? (Or counting sort)
    - Anyone say insertion sort?



LEGO stop motion sorting algorithm
https://www.youtube.com/watch?v=XN0saL9ujLE

Sorting (3/3)

- Given:
  - N integers
  - sort them as fast as possible
- Question to Audience:
  - What solutions might you come up with?
    - Quicksort?
    - Radix sort again? (Or counting sort)
    - Anyone say insertion sort?
  - What probing questions did you come up with this time?



LEGO stop motion sorting algorithm
https://www.youtube.com/watch?v=XN0saL9ujLE

(Aside) -- A Java Perspective (Back to C++ shortly) (1/3)

- If we look at some standard libraries, we find some important implementation details!
- Looks like there's a special overload for 'byte' type that uses qsort.
  - In fact, that's the case for char[], short[], etc. types.
- So a probing questions you might ask when designing a sort is:
  - What is the data type?

```
/**
 * Performs a stable sort on the elements, arranging them according to their
 * natural order.
 *
 * @param a the byte array to sort
 * @param fromIndex the first index to sort (inclusive)
 * @param toIndex the last index to sort (exclusive)
 * @throws IllegalArgumentException if fromIndex &gt; toIndex
 * @throws ArrayIndexOutOfBoundsException if fromIndex &lt; 0
 *         || toIndex &gt; a.length
 */
public static void sort(byte[] a, int fromIndex, int toIndex)
{
  if (fromIndex > toIndex)
    throw new IllegalArgumentException();
  if (fromIndex < 0)
    throw new ArrayIndexOutOfBoundsException();
  qsort(a, fromIndex, toIndex - fromIndex);
}
```

https://developer.classpath.org/doc/java/util/Arrays-source.html

(Aside) -- A Java Perspective (Back to C++ shortly) (2/3)

- Now if we dig in a bit further, there's something really interesting here.
  - A special branch for small arrays using not quicksort, but insertion sort

```
/**
 * Performs a recursive modified quicksort.
 *
 * @param array the array to sort
 * @param from the start index (inclusive)
 * @param count the number of elements to sort
 */
private static void qsort(byte[] array, int from, int count)
{
  // Use an insertion sort on small arrays.
  if (count <= 7)
    {
      for (int i = from + 1; i < from + count; i++)
        for (int j = i; j > from && array[j - 1] > array[j]; j--)
          swap(j, j - 1, array);
      return;
    }
```

```
// Use an insertion sort on small arrays.
if (count <= 7)
  {
    for (int i = from + 1; i < from + count; i++)
      for (int j = i; j > from && array[j - 1] > array[j]; j--)
        swap(j, j - 1, array);
    return;
  }
```

https://developer.classpath.org/doc/java/util/Arrays-source.html

(Aside) -- A Java Perspective (Back to C++ shortly) (3/3)

- Perhaps better locality empirically measured?
  - So a probing questions you might ask when designing a sort is:
    - What is the collection size?

```
// Use an insertion sort on small arrays.
if (count <= 7)
  {
    for (int i = from + 1; i < from + count; i++)
      for (int j = i; j > from && array[j - 1] > array[j]; j--)
        swap(j, j - 1, array);
    return;
  }
```

```
/**
 * Performs a recursive modified quicksort.
 *
 * @param array the array to sort
 * @param from the start index (inclusive)
 * @param count the number of elements to sort
 */
private static void qsort(byte[] array, int from, int count)
{
  // Use an insertion sort on small arrays.
  if (count <= 7)
    {
      for (int i = from + 1; i < from + count; i++)
        for (int j = i; j > from && array[j - 1] > array[j]; j--)
          swap(j, j - 1, array);
      return;
    }
```

```
9

s, array);
s, array);
i, array);

he fray, and use it as a pivot.
```

https://developer.classpath.org/doc/java/util/Arrays-source.html

Back to C++

- In fact -- we can see this same logic in C++'s 'sort' routines.
  - Where if we are sorting less than some '`_S_threshold`' for the container size
  - Based on that threshold value, we branch to a different part of code for perhaps a different algorithm
- Note:
  - These interesting decisions may appear in other places you may not expect (e.g. linear search may prove faster than binary search under various conditions, despite the difference in Big-Oh notation)

```
enum { _S_threshold = 16 };

/// This is a helper function for the sort routine.
template<typename _RandomAccessIterator>
  void
  __final_insertion_sort(_RandomAccessIterator __first,
                         _RandomAccessIterator __last)
  {
    if (__last - __first > int(_S_threshold))
  {
    std::__insertion_sort(__first, __first + int(_S_threshold));
    std::__unguarded_insertion_sort(__first + int(_S_threshold), __last);
  }
    else
  std::__insertion_sort(__first, __last);
  }
```

https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html

Note: C++ standard library may commonly use 'introsort'

Changing Algorithmic Behavior With Branching

`if`, `else if`, `else` (or a `switch`)

Changing Algorithmic Behavior (1/2)

- When we execute programs, we may want to change the behavior for many reasons:
  - Perhaps the **workload has changed**
    - (e.g. number of elements to sort)
  - Perhaps some **state change in the application requires adaptation**
    - (e.g. In a game the AI changes their pathfinding algorithm to be more passive based on user)
  - Perhaps our **system has more or less cpu resources** available
  - Perhaps we have entered **some error state**

```
enum { _S_threshold = 16 };

/// This is a helper function for the sort routine.
template<typename _RandomAccessIterator>
  void
  __final_insertion_sort(_RandomAccessIterator __first,
                         _RandomAccessIterator __last)
  {
    if (__last - __first > int(_S_threshold))
  {
    std::__insertion_sort(__first, __first + int(_S_threshold));
    std::__unguarded_insertion_sort(__first + int(_S_threshold), __last);
  }
    else
  std::__insertion_sort(__first, __last);
  }
```

Changing Algorithmic Behavior (2/2)

- Of course as we have previously seen, we can change behavior based on conditional statements to branch and change the control flow.
  - So let's see how this works at scale with a task of writing some 'sort' function that could be called, and choose an optimal algorithm.

```
enum { _S_threshold = 16 };

/// This is a helper function for the sort routine.
template<typename _RandomAccessIterator>
  void
  __final_insertion_sort(_RandomAccessIterator __first,
             _RandomAccessIterator __last)
  {
    if (__last - __first > int(_S_threshold))
  {
    std::__insertion_sort(__first, __first + int(_S_threshold));
    std::__unguarded_insertion_sort(__first + int(_S_threshold), __last);
  }
    else
  std::__insertion_sort(__first, __last);
  }
```

Sorting Algorithm (1/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

What do folks think?

- Could this be a reasonable sort?
- We choose different sorts based on empirical data and cache sizes to inform these values?

Note: The thresholds here are completely arbitrary

Sorting Algorithm (2/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- How maintainable is this code?
  - Is it reasonable?

Note: The thresholds here are completely arbitrary

Sorting Algorithm (3/11)

```
if (threshold < 16)
       insertion_sort(...);
else if (threshold >= 16 && thresho
              tim_sort (
else if(threshold >=64 &&
              merge_sort(...);
else if(threshold >=128 && threshold < 256)
              some_other_sort2(...)
else if (threshold >= 256 && threshold < 512)
              some_other_sort3 (...);
else if(threshold >=512 && threshold < 1024)
              some_other_sort4(...);
else if(threshold >=1024 && threshold < 2048)
              some_other_sort5(...)
else if (threshold >= 2048 && threshold < 4096)
              some_other_sort6 (...);
else if(threshold >=4096 && threshold < 8192)
              some_other_sort7(...);
else if(threshold >=8192 && threshold < 16384)
              some_other_sort8(...)
else{
              quick_sort(...)
}
```

- How maintainable is this code?
  - Is it reasonable?
  - How about now?
    - The project has evolved over the last few years...
    - (And I'm being nice putting everything into a one-line function)

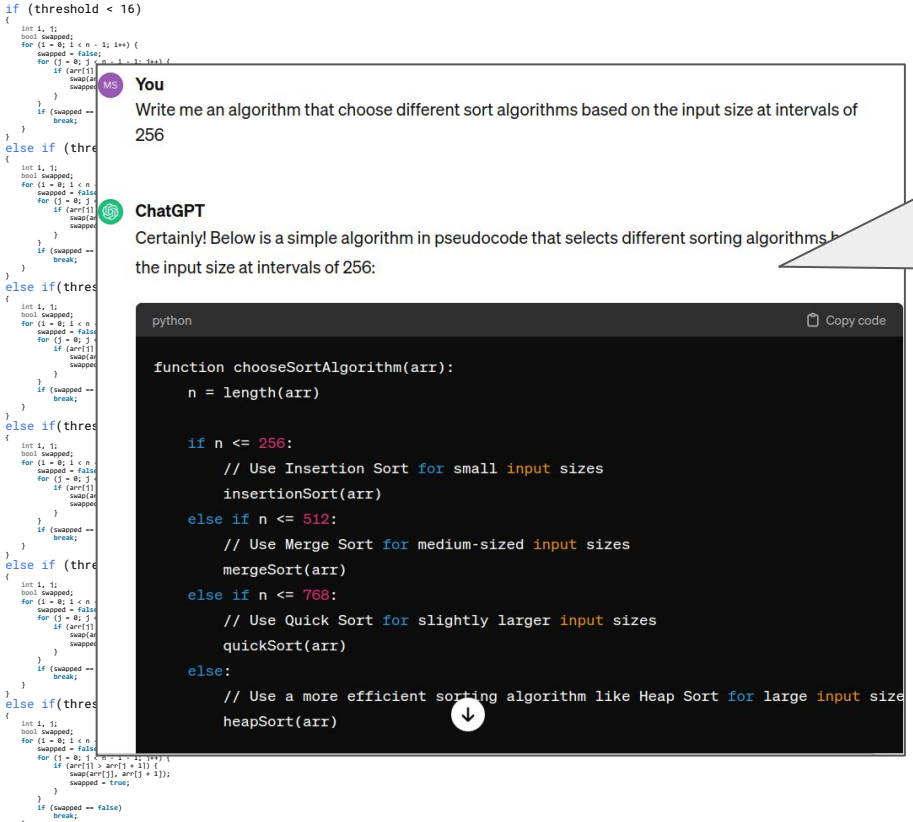
Note: The thresholds here are completely arbitrary

Sorting Algorithm (4/11)

```
if (threshold < 16)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if (threshold >= 16 && threshold < 64)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=64 && threshold < 128)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=128 && threshold < 256)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if (threshold >= 256 && threshold < 512)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=512 && threshold < 1024)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
```

- How maintainable is this code?
  - Is it reasonable?
  - How about now?
    - The project has evolved over the last few years...
    - (And I'm being nice putting everything into a one-line function)
  - How about where I am not so nice and do not use functions.
    - Maybe I need some local state, or find this more performant?

Sorting Algorithm (5/11)



- Let's prefer whatever chatGPT tells people to do these days....and at least have functions for our different sorts
  - Note: A good reason for studying software design will be to validate GPT and other tools.

Sorting Algorithm (6/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- So looking back at the following example -- another line of thought:

Sorting Algorithm (7/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- So looking back at the following example -- another line of thought:
  - How testable is this code?
  - One test per function? or branch?
  - Performance tests at every different thresholds?

Sorting Algorithm (8/11)

```
if (threshold < 16)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if (threshold >= 16 && threshold < 64)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=64 && threshold < 128)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=128 && threshold < 256)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if (threshold >= 256 && threshold < 512)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
    }
}
else if(threshold >=512 && threshold < 1024)
{
    int i, j;
    bool swapped;
    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        if (swapped == false)
            break;
```

- How testable is our code in the not-so-nice instance?
  - Probably a bit more difficult without abstracting away our algorithms
  - i.e.
    - If there is local or global state, this becomes hard to test!

Sorting Algorithm (9/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- We might be able to get away with this type of code in our programs.
- But a few key observations.

Sorting Algorithm (10/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- We might be able to get away with this type of code in our programs.
- But a few key observations.

We want to be able to **change behavior**
- In this case **at run-time**
- But also **perhaps at compile-time**

Sorting Algorithm (11/11)

```
if (threshold < 16)
        insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
        tim_sort (...);
else if(threshold >=64 && threshold < 128)
        merge_sort(...);
else if ...
        ....
```

- We might be able to get away with this type of code in our programs.
- But a few key observations.

Observe we have a family of related algorithms, operating on the same data

Goals for this talk

We want to design code (and discover a design pattern) that helps us:

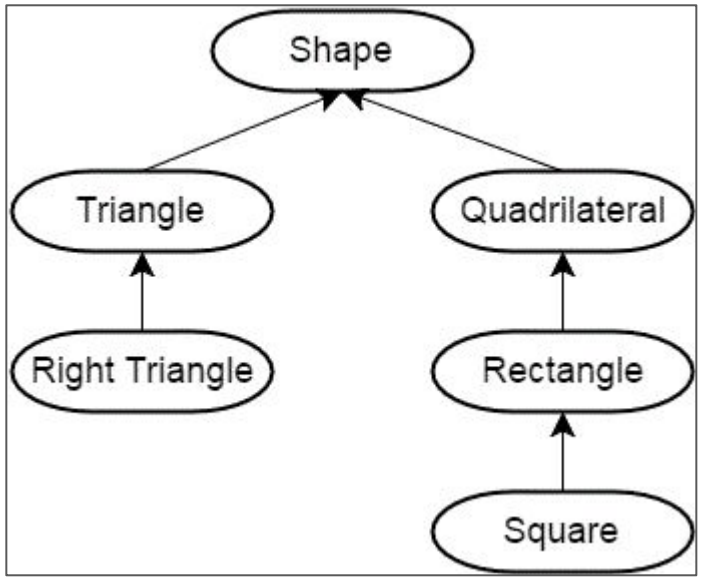
1. We want to **maintain flexibility to choose algorithms**:
   a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
   a. Perhaps a way to organize related algorithms
   b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

Changing Algorithmic Behavior With Inheritance

1. We want to **maintain flexibility to choose algorithms**:
    a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
    a. Perhaps a way to organize related algorithms
    b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

Inheritance (1/3)

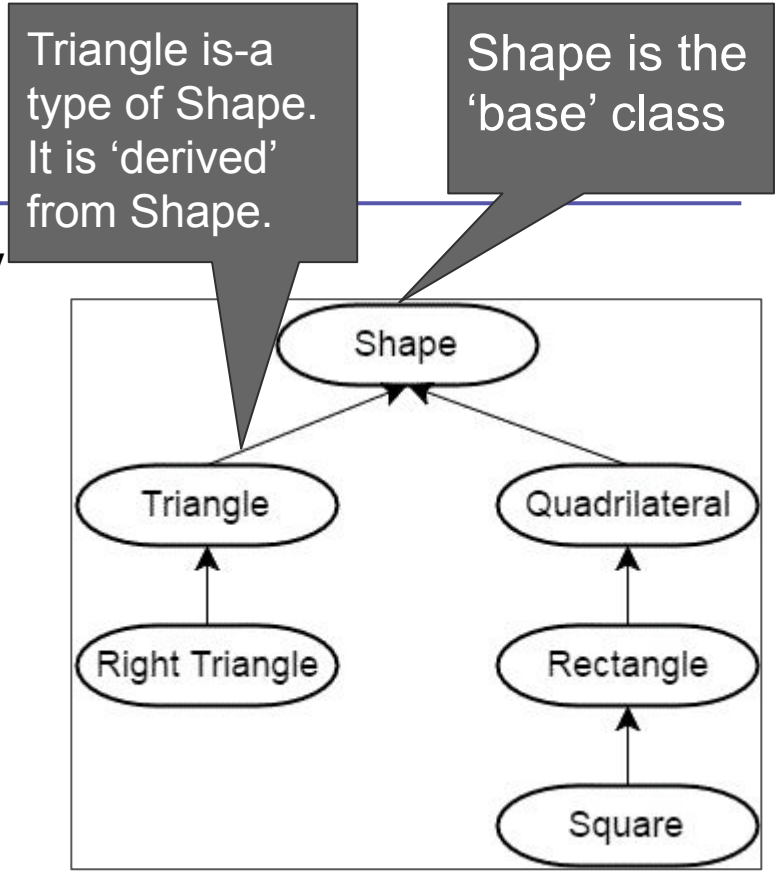
- One mechanism to group related types is by using ‘inheritance.
  - New classes which are derived from existing classes (i.e. inherited from) form an ‘**is-a**’ relationship.



https://learncpp.com/images/CppTutorial/Section11/ShapesInheritance.gif

Inheritance (2/3)

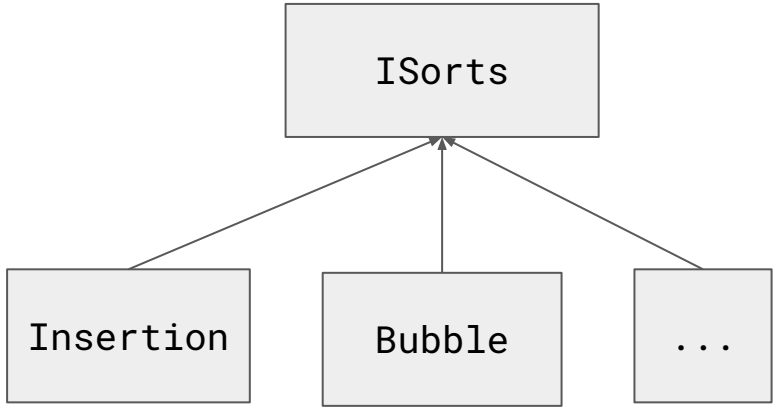
- One mechanism to group related types is by using ‘inheritance.
  - New classes which are derived from existing classes (i.e. inherited from) form an ‘**is-a**’ relationship.



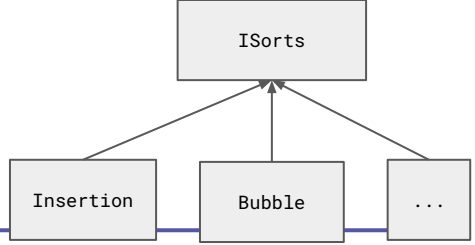
https://learncpp.com/images/CppTutorial/Section11/ShapesInheritance.gif

Inheritance (3/3)

- So in the context of our sorting problem
  - We could have a common interface (`ISorts`) where we derive new types of sorts.
  - What does this give us?
    - (next slide)



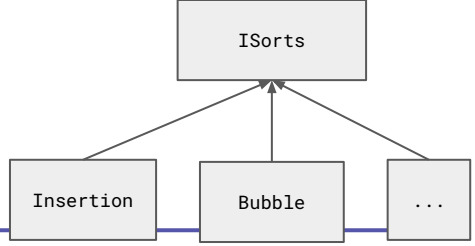
Run-time polymorphism (1/5)



- We now have multiple Sorts implementing from a common interface
- Now we can use **run-time** polymorphism to choose the algorithm.

```cpp
// isorts.cpp
#include <iostream>

struct ISorts{
    ~ISorts(){}

    virtual void Sort(/* container, params */)=0;
};

struct InsertionSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "InsertionSort::Sort\n";
    }
};

struct BubbleSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "BubbleSort::Sort\n";
    }
};

int main(){

    ISorts* sorting_algo = new InsertionSort;

    sorting_algo->Sort();

    delete sorting_algo;

    return 0;
}
```

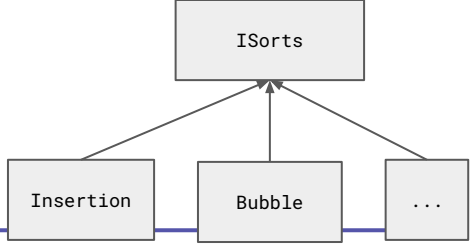
Run-time polymorphism (2/5)



- We now have multiple Sorts implementing from a common interface
- Now we can use **run-time** polymorphism to choose the algorithm.

```cpp
// isorts.cpp
#include <iostream>

struct ISorts{
    ~ISorts(){}

    virtual void Sort(/* container, params */)=0;
};

struct InsertionSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "InsertionSort::Sort\n";
    }
};

struct BubbleSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "BubbleSort::Sort\n";
    }
};

int main(){

    ISorts* sorting_algo = new InsertionSort;

    sorting_algo->Sort();

    delete sorting_algo;

    return 0;
}
```

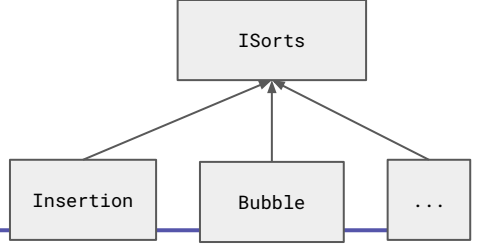
Run-time polymorphism (3/5)



- Same code as previous slide, but this time with `std::unique_ptr`
  - (I'll try to keep things modern! :))

```cpp
// isorts_modern.cpp
#include <iostream>
#include <memory>

struct ISorts{
    ~ISorts(){}

    virtual void Sort(/* container, params */)=0;
};

struct InsertionSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "InsertionSort::Sort\n";
    }
};

struct BubbleSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "BubbleSort::Sort\n";
    }
};

int main(){

    std::unique_ptr<ISorts> sorting_algo = std::make_unique<InsertionSort>();

    sorting_algo->Sort();

    return 0;
}
```

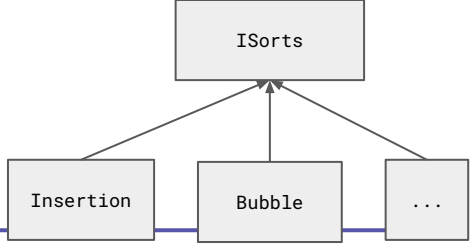
Run-time polymorphism (4/5)



- Observe that we have encapsulated the behavior (i.e. the act of ‘sorting’) into an object as well.
  - Note:
    - If you are anti-OOP, we could have free-functions and change behaviors using function pointers or std::function
      - A functional approach using a std::variant holding different function pointers may also work
- The advantage we get with inheritance here, is we have some type checking
  - i.e. I cannot assigning ‘sorting_algo’ to anything outside of the ISorts hierarchy.
- The other advantage with inheritance is we can more easily enforce the interface with pure virtual functions.

```cpp
// isorts_modern.cpp
#include <iostream>
#include <memory>

struct ISorts{
    ~ISorts(){}

    virtual void Sort(/* container, params */)=0;
};

struct InsertionSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "InsertionSort::Sort\n";
    }
};

struct BubbleSort : public ISorts{
    void Sort(/* container, params */) override{
        std::cout << "BubbleSort::Sort\n";
    }
};

int main(){

    std::unique_ptr<ISorts> sorting_algo = std::make_unique<InsertionSort>();

    sorting_algo->Sort();

    return 0;
}
```

Run-time polymorphism (5/5)



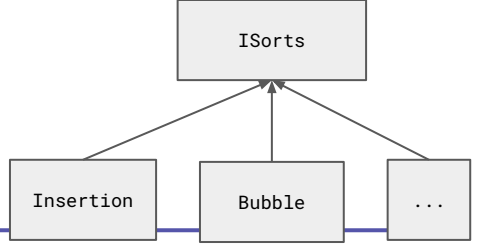
- Observe we can further use our sorts to implement a generic_sort function
  - I would say this is quite maintainable, and the details of implementation are given to each individual struct of sorting.
    - It becomes nice to also have these building blocks (i.e. the different sorts) -- in this case following the Single Responsibility Principle (SRP)
- So you now have this idea of using inheritance for a family of similar algorithms

```
1  // functions.cpp
2  #include <iostream>
3  #include <memory>
4
5  +-- 17 lines: struct ISorts{----------------------------------------------
22
23 void generic_sort(/*container, params*/){
24     std::unique_ptr<ISorts> sorting_algo = std::make_unique<BubbleSort>();
25
26     int collectionSize = 52;
27     if(collectionSize< 16){
28         sorting_algo = std::make_unique<BubbleSort>();
29     }else if (collectionSize>= 16 && collectionSize < 64){
30         sorting_algo = std::make_unique<InsertionSort>();
31     }
32
33     sorting_algo->Sort();
34 }
35
36 int main(){
37
38     generic_sort(/* container, params */);
39
40     return 0;
41 }
```

Managing Complexity

1. We want to **maintain flexibility to choose algorithms**:
    a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
    a. Perhaps a way to organize related algorithms
    b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

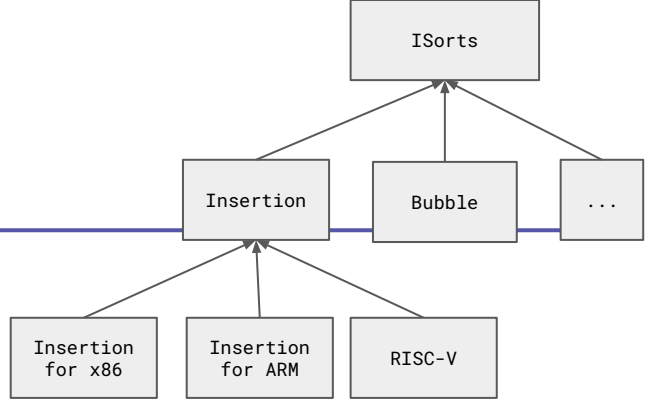
Managing Complexity (1/4)



- So far we have improved our code to some degree
  - We can simply re-assign ‘sorting_algo’ based on some conditional.
- It also becomes ‘easy’ to add new sorting algorithms
  - Adding a new type to our hierarchy implementing the `ISorts` interface is easy
- Something else interesting that we have done, is isolated our ‘sorting’ its our own hierarchy
  - (next slide)

```
1 // functions.cpp
2 #include <iostream>
3 #include <memory>
4
5 +-- 17 lines: struct ISorts{----------------------------------------------
22
23 void generic_sort(/*container, params*/){
24     std::unique_ptr<ISorts> sorting_algo = std::make_unique<BubbleSort>();
25
26     int collectionSize = 52;
27     if(collectionSize< 16){
28         sorting_algo = std::make_unique<BubbleSort>();
29     }else if (collectionSize>= 16 && collectionSize < 64){
30         sorting_algo = std::make_unique<InsertionSort>();
31     }
32
33     sorting_algo->Sort();
34 }
35
36 int main(){
37
38     generic_sort(/* container, params */);
39
40     return 0;
41 }
```

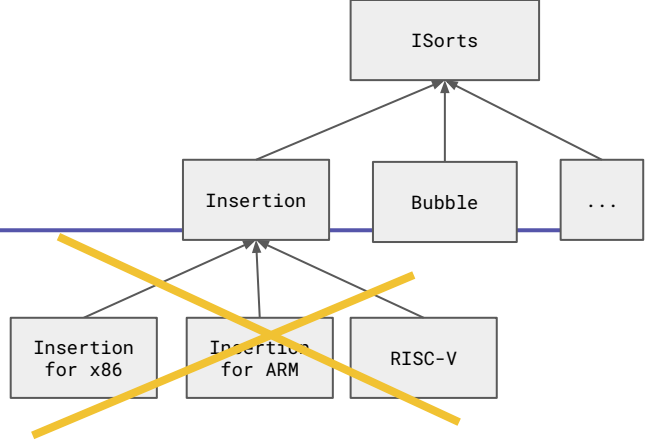
Managing Complexity (2/4)



- It’s important that we do not get too carried away with our inheritance
  - Perhaps we start learning clever tricks for different platforms for instance to optimize some sorts
    - (maybe you find some new hardware instructions to cleverly swizzle bits in parallel)
  - Beware, that as soon as we start using inheritance to solve all of our problems, or our hierarchy gets too deep -- it becomes difficult to maintain code (or choose the right algorithm!)

Managing Complexity (3/4)

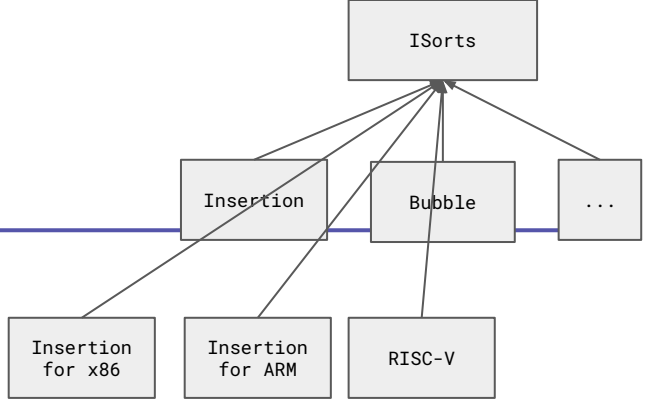
- It's important that we do not get too carried away with our inheritance
  - Perhaps we start learning clever tricks for different platforms for instance to optimize some sorts
    - (maybe you find some new hardware instructions to cleverly swizzle bits in parallel)
  - Beware, that as soon as we start using inheritance to solve all of our problems, or our hierarchy gets too deep -- it becomes difficult to maintain code (or choose the right algorithm!)



Careful! *may* increase complexity -- first ask if hierarchy could remain 1-level deep.

Managing Complexity (4/4)

- It's important that we do not get too carried away with our inheritance
  - Perhaps we start learning clever tricks for different platforms for instance to optimize some sorts
    - (maybe you find some new hardware instructions to cleverly swizzle bits in parallel)
  - Beware, that as soon as we start using inheritance to solve all of our problems, or our hierarchy gets too deep -- it becomes difficult to maintain code (or choose the right algorithm!)
    - We *might* consider a separate hierarchy for x86,ARM, RISC-V, platforms, or perhaps moving them up one level.
      - This allows a separation of concerns to some degree if you move to a different hierarchy or sub-tree



Maybe okay -- slightly better

Testability

1. We want to **maintain flexibility to choose algorithms**:
    a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
    a. Perhaps a way to organize related algorithms
    b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

Testing

- With each implementation of our sort, we can test it relatively easily.
  - Using a test-driven design, we would setup a test for every derived class and set our expectation for sorted (or unsorted) data.
  - Note:
    - Or even better -- you could parametrize your unit tests with the different strategy types!

```
1 // testable.cpp
2 #include <algorithm>
3 #include <iostream>
4 #include <memory>
5
6 +-- 30 lines: struct ISorts{----------------------------------------------
36
37 bool unittest1(){
38     std::vector some_vector{1,3,6,4,2,5};
39
40     std::unique_ptr<ISorts> sorting_algo = std::make_unique<BubbleSort>();
41     sorting_algo->Sort();
42
43     return std::is_sorted(some_vector.begin(),some_vector.end());
44 }
45
46 bool unittest2(){
47     std::vector some_vector{1,3,6,4,2,5};
48
49     std::unique_ptr<ISorts> sorting_algo = std::make_unique<InsertionSort>();
50     sorting_algo->Sort();
51
52     return std::is_sorted(some_vector.begin(),some_vector.end());
53 }
54
55 int main(){
56
57     unittest1();
58     unittest2();
59
60     return 0;
61 }
```

Goal Review

1. We want to **maintain flexibility to choose algorithms**:
    a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
    a. Perhaps a way to organize related algorithms
    b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

Goals

- So it looks like it's as simple as that for writing better code, huh?
  - Move related algorithms into an inheritance structure
  - And otherwise be careful about extending the structure too much
- And there is actually a pattern that largely leverages this idea.

1. We want to **maintain flexibility to choose algorithms**:
   a. Whether this is at run-time or compile-time
2. **Manage complexity of our code** (recall the not-so-nice and large code example)
   a. Perhaps a way to organize related algorithms
   b. This allows us to make intelligent decisions based on what is in our toolbox
3. It is reasonable that we want to be able to **test** our algorithms

Strategy Design Pattern

https://en.wikipedia.org/wiki/Software

Strategy Design Pattern [wiki] (1/3)

- In computer programming, the **strategy pattern** (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.[1]

Strategy Design Pattern [wiki] (2/3)

- In computer programming, the **strategy pattern** (also known as the policy pattern) is a behavioral software design pattern that **enables selecting an algorithm at runtime**. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.[1]

Strategy Design Pattern [wiki] (3/3)

- In computer programming, the **strategy pattern** (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. **Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.**[1]

When to use Strategy Design Pattern [wiki]

1. When you have a ‘family’ of related algorithms that are interchangeable within the ‘family’
   a. e.g. different sorts acting on integer data
   b. e.g. different compression algorithms (e.g. lossy vs non-lossy)
   c. e.g. exporting dataset to different text formats (e.g. raw, json, xml, yaml, etc.)
   d. e.g. different graph traversals
2. When you want to prevent yourself from redeploying software
   a. i.e. May be able to select another algorithm at run-time instead of a patch.
3. To *potentially* make testing easier
   a. I’ve found it works well in test-driven development when you have related algorithms.
4. You observe that a class has multiple conditional statements in its operations -- instead of many conditionals, move related conditional branches into their own strategy class.

> In computer programming, the **strategy pattern** (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use.[1]

Strategy Pattern in the Wild

Question to Audience: (1/2)

- Where is Strategy used in the C++ Standard Library?

Question to Audience: (2/2)

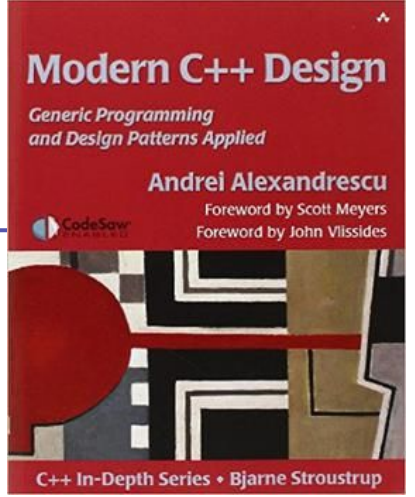
- Where is Strategy used in the C++ Standard Library?
  - Execution policies perhaps (https://en.cppreference.com/w/cpp/algorithm/reduce
  - Comparators (https://en.cppreference.com/w/cpp/algorithm/sort_heap)
  - Allocators (https://en.cppreference.com/w/cpp/memory/allocator)
  - More?
    - (Sometimes at run-time, sometime at compile-time!)

```
template< class ExecutionPolicy, class ForwardIt, class T, class BinaryOp >
T reduce( ExecutionPolicy&& policy,                                    (6)   (since C++17)
          ForwardIt first, ForwardIt last, T init, BinaryOp binary_op );
```

```
                                                                       (2)
template< class RandomIt, class Compare >
constexpr void sort_heap( RandomIt first, RandomIt last, Compare comp );   (since C++20)
```

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;                                                        (1)

namespace pmr {
    template< class T >
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;  (2)   (since C++17)
}
```

(Aside) Policy-based design



- I do want to mention that some of the previous examples use ‘compile-time polymorphism’ (i.e. templates) to select the ‘strategy’.
  - This is often known as ‘policy-based design’, which was popularized by the book in the top-right corner.

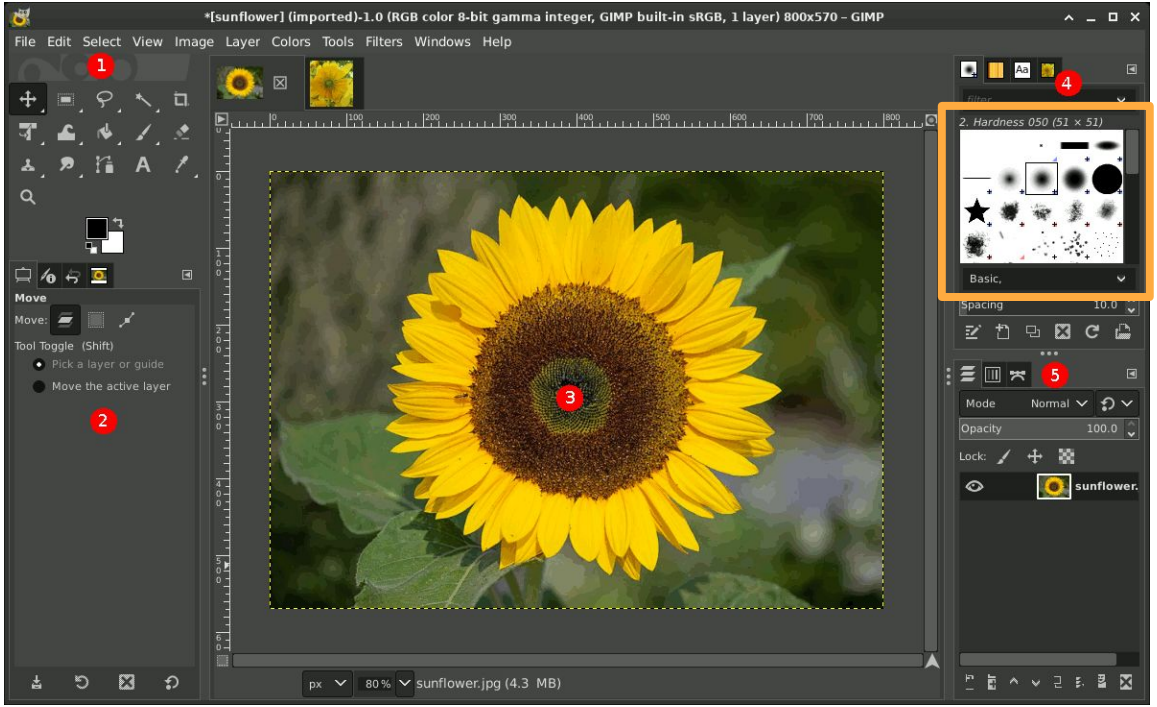
Policy-based design [edit]

Policy-based design, also known as **policy-based class design** or **policy-based programming**, is the term used in *Modern C++ Design* for a design approach based on an idiom for C++ known as **policies**. It has been described as a compile-time variant of the strategy pattern, and has connections with C++ template metaprogramming. It was first popularized in C++ by Andrei Alexandrescu with *Modern C++ Design* and with his column *Generic<Programming>* in the *C/C++ Users Journal*, and it is currently closely associated with C++ and D as it requires a compiler with highly robust support for templates, which was not common before about 2003.

https://en.wikipedia.org/wiki/Modern_C%2B%2B_Design#Policy-based_design

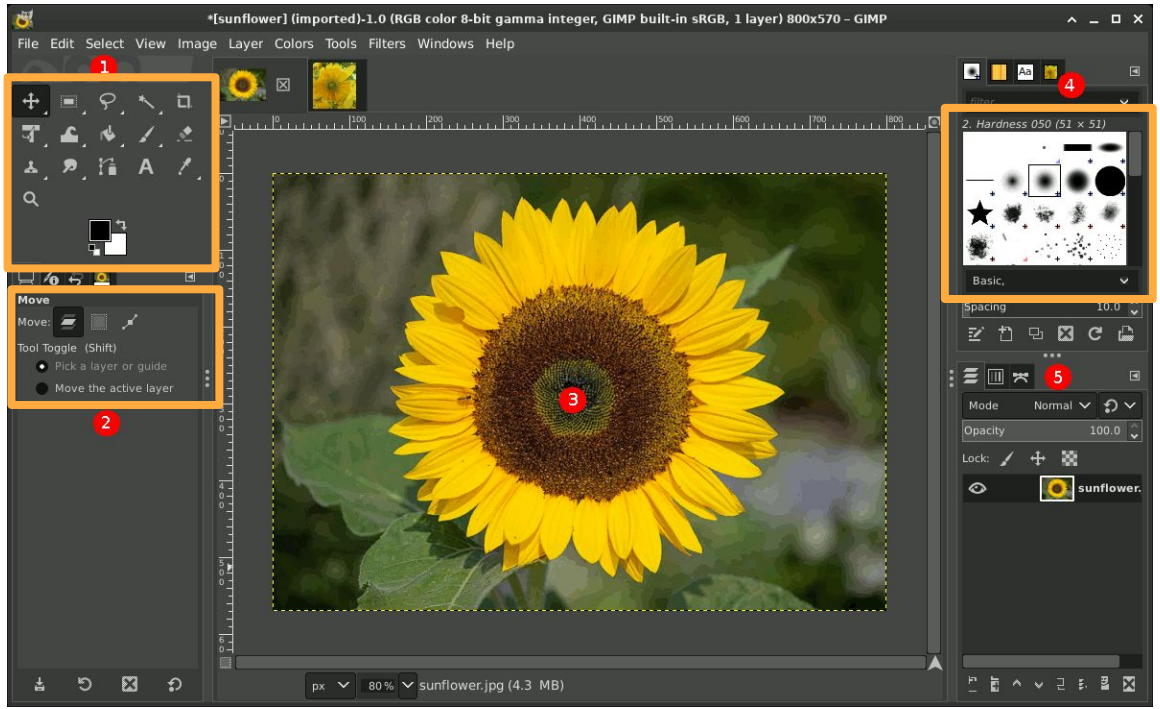
Question to Audience: Where is Strategy Pattern here?

- Thoughts?



Paint Application (GNU Gimp)

- Observe the different brushes that are highlighted to the right
  - There are at least 20 + brushes
  - How would we model the behaviors for them?
  - Or the different tools (spray can, lines, pencil drawing, etc.)



Command Line Applications

- Most command line applications, likely have opportunity for strategy
  - e.g. ffmpeg with different configurations/Options

complete list of ffmpeg flags / commands

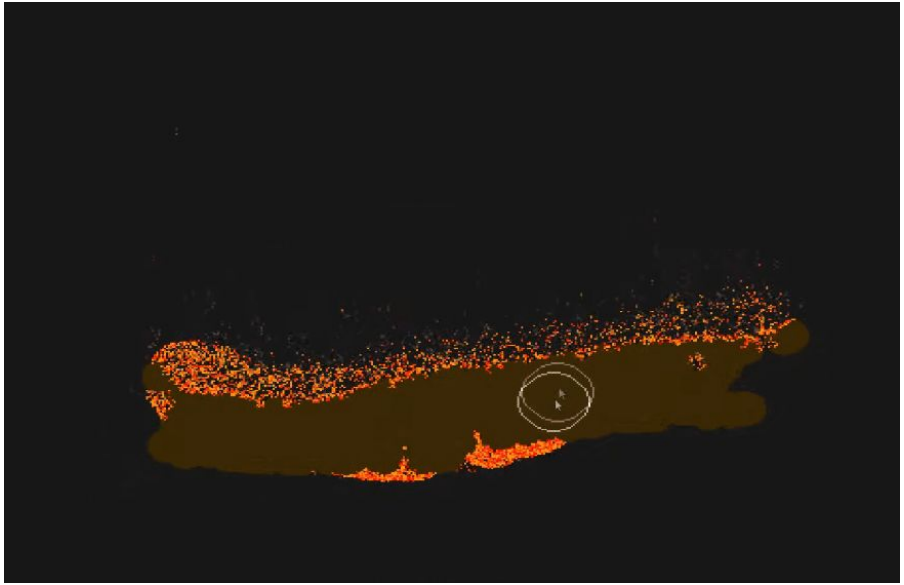
gistfile1.txt

```
Originall From: Posted 2015-05-29 http://ubwg.net/b/full-list-of-ffmpeg-flags-and-options

This is the complete list that's outputted by ffmpeg when running ffmpeg -h full.

usage: ffmpeg [options] [[infile options] -i infile]… {[outfile options] outfile}…

Getting help:
    -h — print basic options
    -h long — print more options
    -h full — print all options (including all format and codec specific options, very long)
    See man ffmpeg for detailed description of the options.
```

https://gist.github.com/tayvano/6e2d456a9897f55025e25035478a3a50

Falling Sands Game

- These falling sands games with lots of different ‘particles’ seem like an excellent candidate for an `IParticle` or `IElement` class.

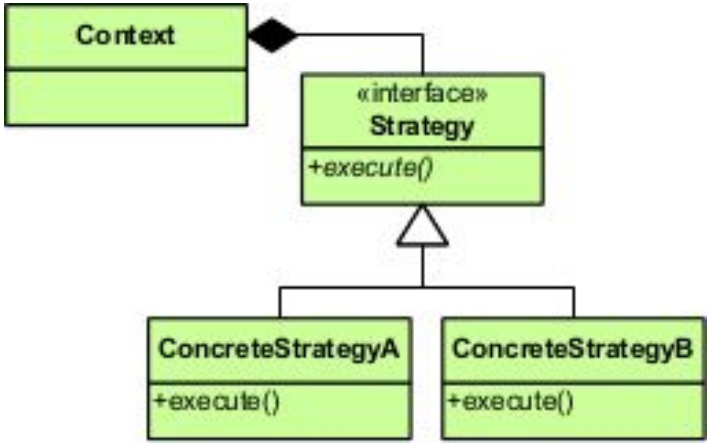


https://www.youtube.com/watch?v=VLZjd_Y1gJ8

More Formal Strategy Pattern

Strategy Pattern

- Formalizing the strategy pattern just a bit more -- the strategies are usually part of some ‘context’
  - Where the ‘context’ is some other object that uses the pattern.
  - Let’s look at an example.

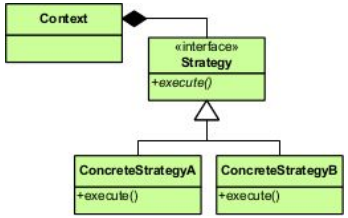


https://en.wikipedia.org/wiki/Strategy_pattern

(Hypothetical) Strategy Pattern in a Game

(Disclaimer, I did not work on this project)

Example of combat strategy (1/5)

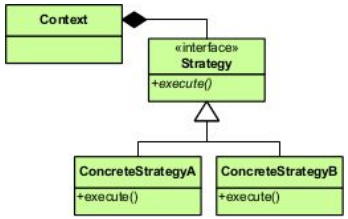


- Let’s say we’ve got an ‘Orc’ to the right here.
  - This ‘Orc’ represents the ‘Context’

https://www.kotaku.com.au/wp-content/uploads/2014/10/03/xq72wcq2noxwaroanfiu.gif?quality=75

```
struct Orc /* The Context */ {

    std::unique_ptr<ICombat> mCombatStrategy;

    Orc(){
        mCombatStrategy = std::make_unique<NoCombat>();
    }

    void SetStrategy(std::unique_ptr<ICombat>&& combat){
        mCombatStrategy = std::move(combat);
    }

    void DoActions(){
        mCombatStrategy->Execute();
    }
};
```

Example of combat strategy (2/5)

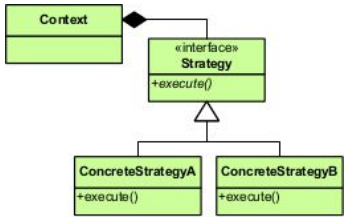


- Let's say we've got an 'Orc' to the right here.
  - This 'Orc' represents the 'Context'
- So now observe that we hold a 'strategy' for something that the Orc will do
  - Note: We delegate to the 'ICombat' to do the 'Execute' action.
  - Note: You may need to pass data from 'Orc' to the selected combat strategy
    - Component systems or other use of specialized pointers may also easy how strategy passes data between 'Orc'

```
struct Orc /* The Context */ {

    std::unique_ptr<ICombat> mCombatStrategy;

    Orc(){
        mCombatStrategy = std::make_unique<NoCombat>();
    }

    void SetStrategy(std::unique_ptr<ICombat>&& combat){
        mCombatStrategy = std::move(combat);
    }

    void DoActions(){
        mCombatStrategy->Execute();
    }
};
```

https://www.kotaku.com.au/wp-content/uploads/2014/10/03/xo72wco2poxwaroanfiu.gif?quality=75

Example of combat strategy (3/5)




```
struct Orc /* The Context */ {

    std::unique_ptr<ICombat> mCombatStrategy;

    Orc(){
        mCombatStrategy = std::make_unique<NoCombat>();
    }

    void SetStrategy(std::unique_ptr<ICombat>&& combat){
        mCombatStrategy = std::move(combat);
    }

    void DoActions(){
        mCombatStrategy->Execute();
    }
};
```

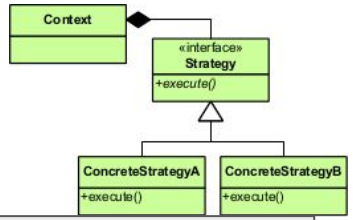
Context

And if you were curious -- here is an example family of related algorithms -- i.e. our Strategy

```
 6 struct ICombat{
 7     ~ICombat(){}
 8     virtual void Execute(/* container, params */)=0;
 9 };
10
11 struct NoCombat: public ICombat{
12     void Execute(/* params */) override{
13         std::cout << "Run away in fear!!\n";
14     }
15 };
16 struct MeleeCombat : public ICombat{
17     void Execute(/* params */) override{
18         std::cout << "MeleeCombat::Execute(...)\n";
19     }
20 };
21 struct RangedCombat : public ICombat{
22     void Execute(/* params */) override{
23         std::cout << "RangedCombat::Execute(...)\n";
24     }
25 };
```

Strategy

Example of combat strategy (4/5)




```
struct Orc /* The Context */ {

    std::unique_ptr<ICombat> mCombatStrategy;

    Orc(){
        mCombatStrategy = std::make_unique<NoCombat>();
    }

    void SetStrategy(std::unique_ptr<ICombat>&& combat){
        mCombatStrategy = std::move(combat);
    }

    void DoActions(){
        mCombatStrategy->Execute();
    }
};
```

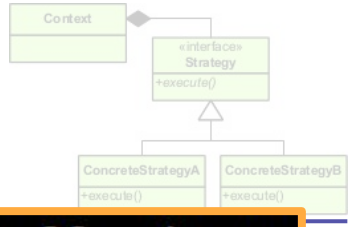
Context

- Note: We could consider always passing in a strategy in the constructor (construction-injection).
  - We have a setter otherwise to change strategy (setter-injection).
  - All are ‘dependency injection’ techniques for setting our strategy
- Note: Service Locator pattern may also be another way to ‘find’ strategies to control actions -- see https://gameprogrammingpatterns.com/service-locator.html

```
 6 struct ICombat{
 7     ~ICombat(){}
 8     virtual void Execute(/* container, params */)=0;
 9 };
10
11 struct NoCombat: public ICombat{
12     void Execute(/* params */) override{
13         std::cout << "Run away in fear!!\n";
14     }
15 };
16 struct MeleeCombat : public ICombat{
17     void Execute(/* params */) override{
18         std::cout << "MeleeCombat::Execute(...)\n";
19     }
20 };
21 struct RangedCombat : public ICombat{
22     void Execute(/* params */) override{
23         std::cout << "RangedCombat::Execute(...)\n";
24     }
25 };
```

Strategy

Example of combat strategy (5/5)




```
mike@mike-MS-7B17:cpponline$ g++ context.cpp -o prog && ./prog
Run away in fear!!
RangedCombat::Execute(...)
```

```
    void SetStrategy(std::unique
        mCombatStrategy = std::m
    }

    void DoActions(){
        mCombatStrategy->Execute
    }
};
```

Conte

```
44 int main(){
45
46     Orc orc;
47
48     orc.DoActions();
49
50     /* time passes */
51     orc.SetStrategy(std::make_unique<RangedCombat>());
52
53     orc.DoActions();
54
55     return 0;
56 }
```

Usage

```
{}
id Execute(/* container, params */)=0;

t: public ICombat{
ute(/* params */) override{
out << "Run away in fear!!\n";

mbat : public ICombat{
ute(/* params */) override{
    cout << "MeleeCombat::Execute(...)\n";
    }
};
struct RangedCombat : public ICombat{
    void Execute(/* params */) override{
        std::cout << "RangedCombat::Execute(...)\n";
    }
};
```

Strategy

Summary

Pros and Cons

- Pro
  - Provide choice of implementation at run-time
    - Based on workload/data-set that may fluctuate you may
  - Potentially more understandable code
    - As opposed to lots of conditional statements, or otherwise an API with many special cases
- Con
  - Need to measure performance
    - May have some performance implication passing around objects or a level of indirection
    - Fastest computation likely to ‘just do the thing’ that you need to do
  - May be adding inheritance hierarchies that are not needed

Use the Strategy Pattern When...

- You find yourself working with ‘families of algorithms’
- You have lots of ‘if-else’ or ‘switch’ statements
  - Simplifying your code with this abstraction may also allow you to create/explore/change your program more freely
- You need a way to change run-time behavior
- You want more testable code
  - (Whether that encourages testing in general, or makes it easier to write tests)

Thank you C++ONLINE 2024!

The Strategy Design Pattern

-- Design Patterns

with Mike Shah

18:00 - 19:00 UTC Sat. Mar 2, 2024

60 minutes + 15 minute Q&A After Introductory Audience

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube
www.youtube.com/c/MikeShah
http://tinyurl.com/mike-talks

Thank you!

Extras and Notes