Sets & Dictionaries

Omar Abdul-Latif

More Data Structures

- We have seen the list data structure and what it can be used for.
- We will examine here to more advanced data structures, the set and the dictionary.
- In particular, the dictionary is an important, very useful part of python, as well as generally useful to solve many problems.

Sets

- In mathematics, a set is a collection of objects, potentially of many different types.
- In a set, no two elements are identical, that is a set consists of elements each of which is unique compared to the other elements.
- There is no order to the elements of a set.
- A set with no elements is the empty set.

Creating a set

mySet = set(“abcd”)

- The “set” keyword creates a set.
- The single argument that follows must be something that is iterable, that it can be walked through one at a time with a for.
- The result is a set data structure:

print mySet

set([‘a’, ‘c’, ‘b’, ‘d’])

Diverse elements

- A set can consist of a mixture of different types of elements.

  mySet = set([‘a’, 1, 3.14159, True])

- As long as the single argument can be iterated through, you can make a set of it.

No Duplicates

- Duplicates are automatically removed:

```
mySet = set(“aabbccdd”)
print mySet
set([‘a’, ‘c’, ‘b’, ‘d’])
```

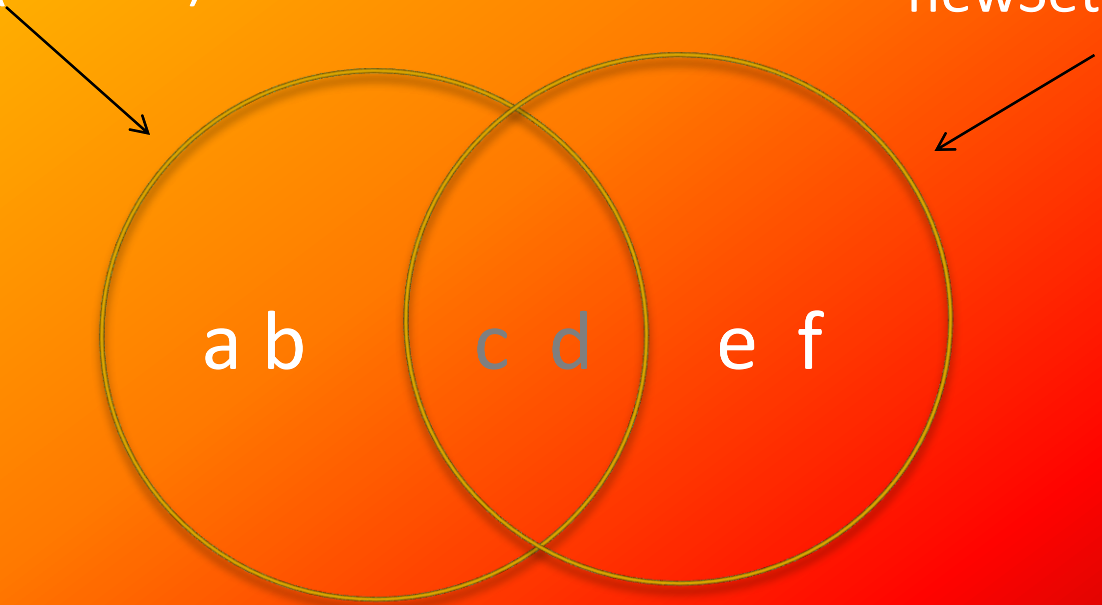
Common Operators

- Most data structures respond to these:
  - len(mySet):
    - the number of elements in a set.
  - element in mySet:
    - boolean indicating whether element is in the set.
  - for element in mySet:
    - iterate through the elements in the set

Sets Intersection

mySet = set("abcd")

newSet= set("cdef")



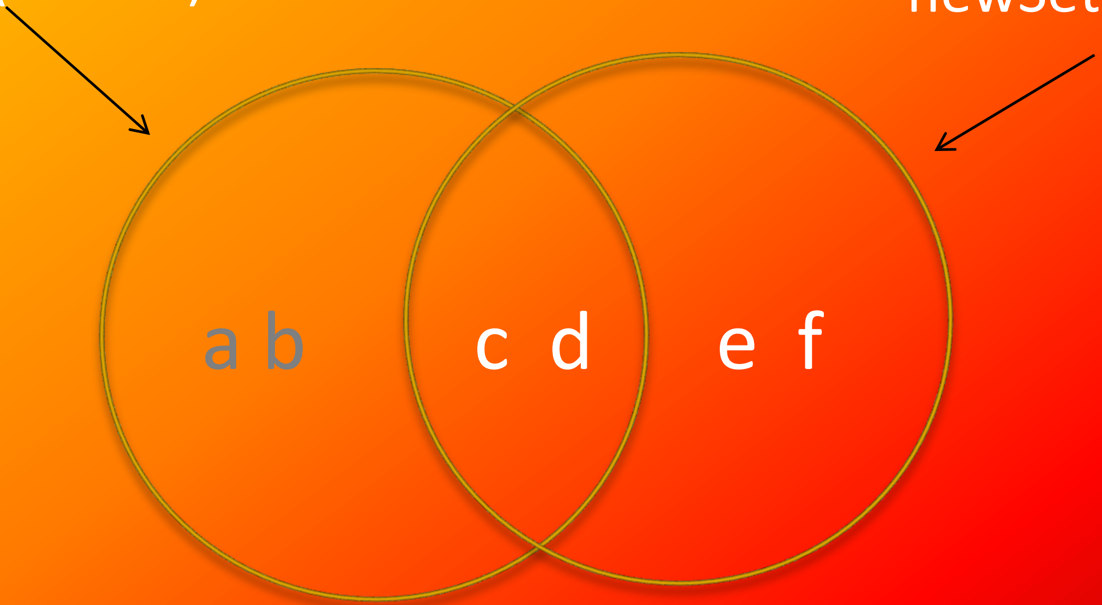
mySet.intersection(newSet)

returns: set(['c', 'd'])

Set Difference

mySet = set("abcd")

newSet= set("cdef")



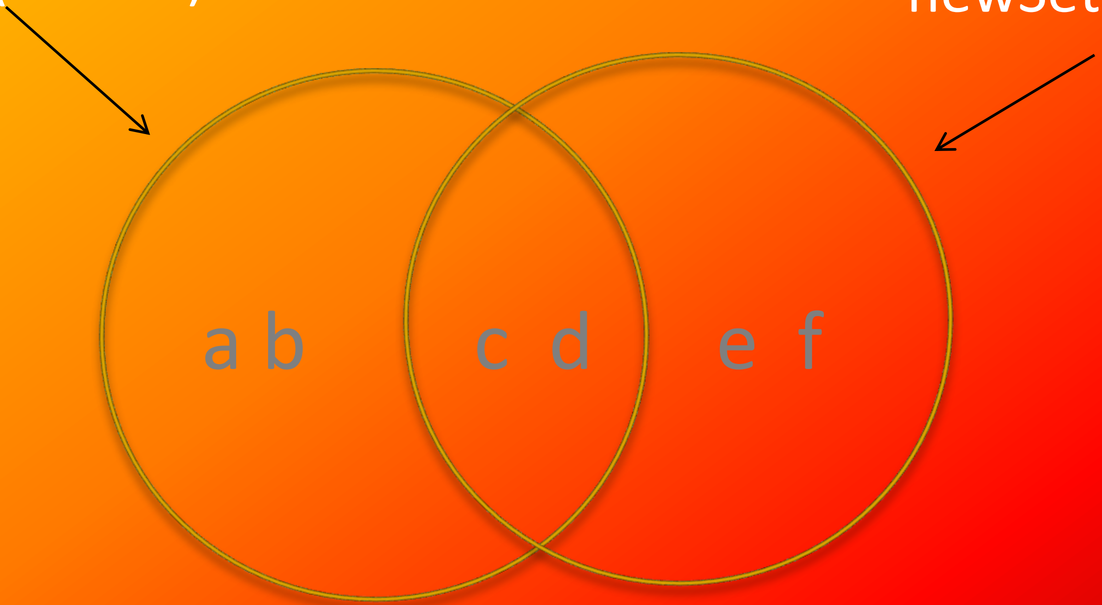
mySet.difference(newSet)

returns: set(['a', 'b'])

Sets Union

mySet = set("abcd")

newSet= set("cdef")



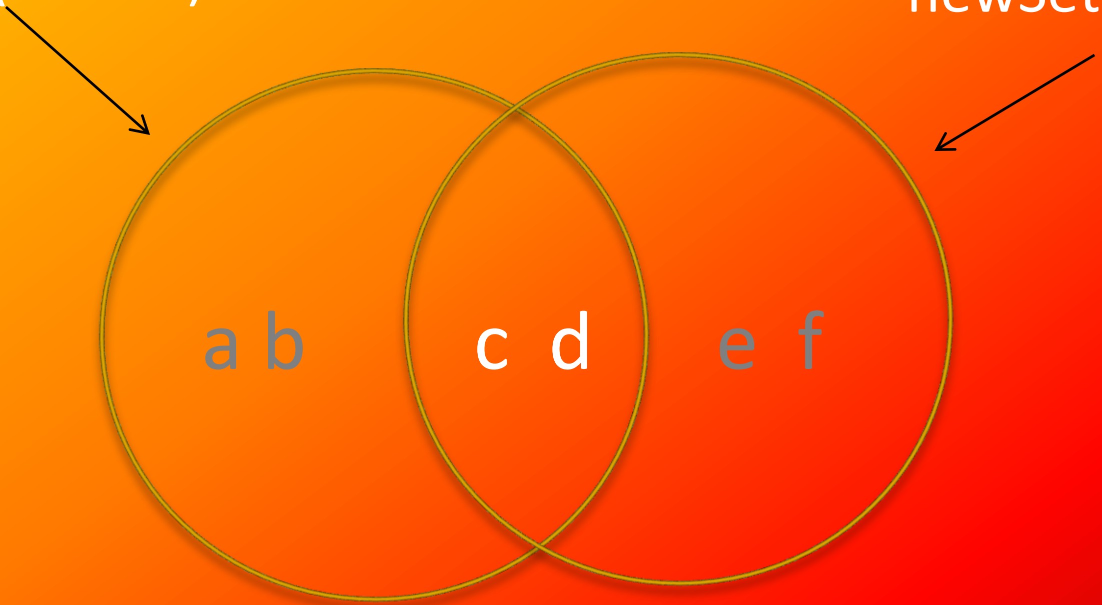
mySet.union(newSet)

returns: set(['a', 'b', 'c', 'd', 'e', 'f'])

Sets Symmetric Difference

mySet = set("abcd")

newSet= set("cdef")

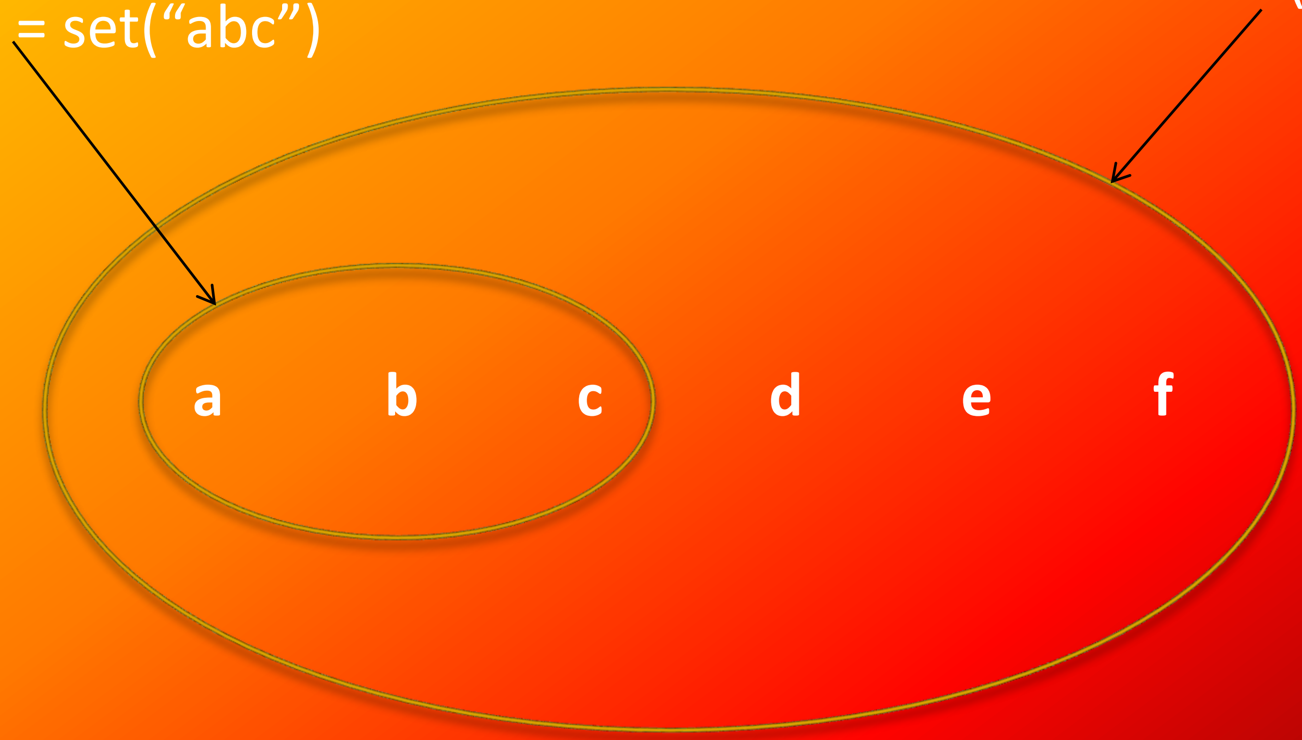


mySet.symmetric_difference(newSet) returns: set(['a', 'b', 'e', 'f'])

Super Set

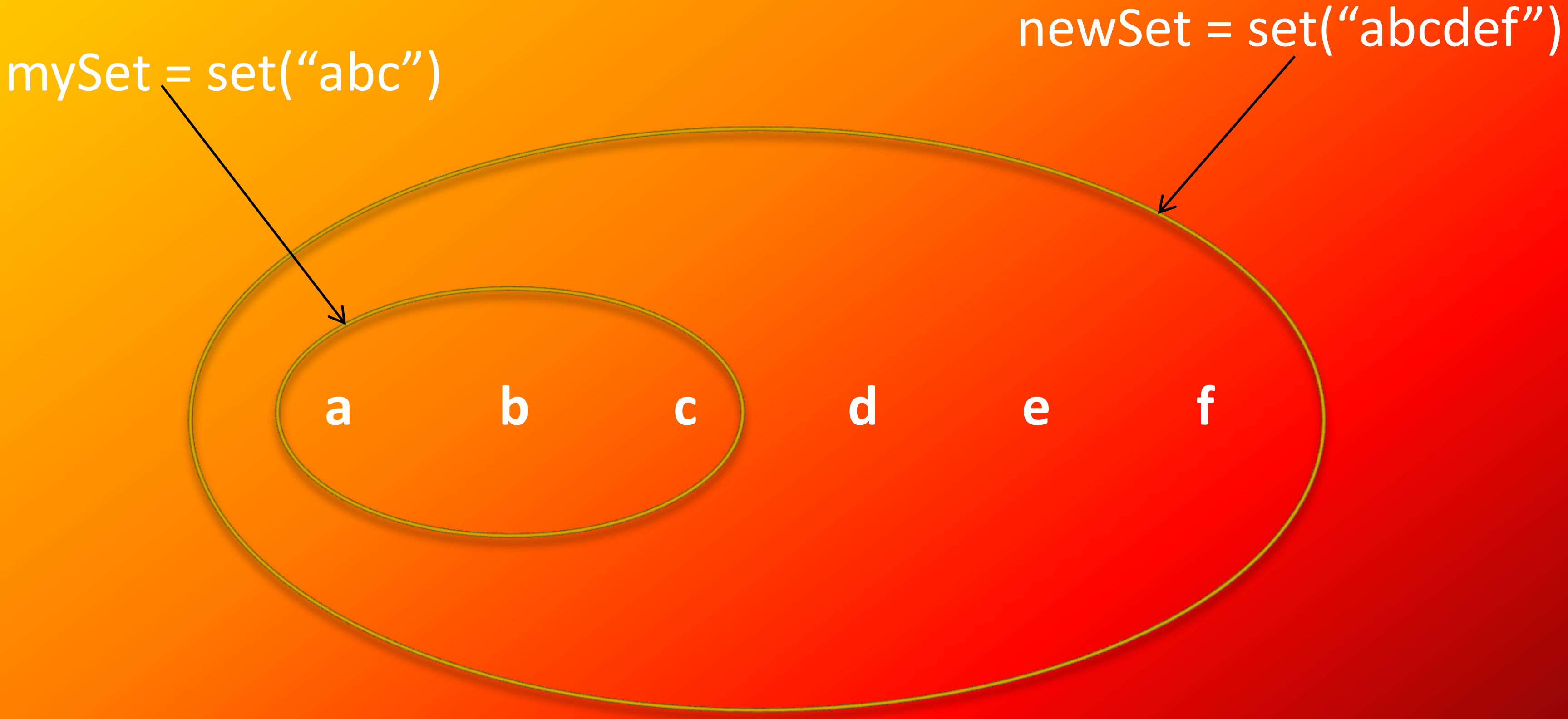
newSet = set(“abcdef”)

mySet = set(“abc”)



```
newSet.issuperset(mySet)      returns:   True
```

Sub Set



mySet.issubset(newSet)   returns: True

Other Set Operations

- mySet.add(“g”)
  - Adds to the set, no effect if item is in set already
- mySet.clear()
  - Emptys the set
- mySet.remove(“g”) vs. mySet.discard(“g”)
  - Remove throws an error if “g” isn’t there. Discard doesn’t care. Both remove “g” from the set.
- mySet.copy()
  - Returns a shallow copy of mySet

Example

- SimpleSets

Dictionaries

- In data structure terms, a dictionary is better termed an associative array or associative list.
- You can think of it as a list of pairs, where the first element of the pair, the key, is associated with the second element of the pair, the value.

Key-Value pairs

- The key acts as a “lookup” to find the associated value.
- Just like a dictionary, you look up a word by its spelling to find the associated definition
- A dictionary can be searched to locate the value associated with a key.

Python Dictionary

- Use the { } marker to create a dictionary
- Use the : marker to separate pairs
- Example:

```
myDictionary = {‘e’:2.7183, ‘pi’:3.1416,
‘h’:6.6261e-34}
print myDictionary
{    ‘h’: 6.6261399999999998e-034,
     ‘pi’: 3.1415999999999999,
     ‘e’: 2.7183000000000002}
```

Keys and values

- Key can be anything as long as it is immutable:
  - Strings, integers, tuples are fine.
  - Lists are NOT.

- Value can be anything

Access Dictionary Elements

- Access requires [ ], but the key is the index!
- *myDictionary = { }*
  - *An empty dictionary*
- *myDictionary[‘bill’] = 25*
  - *Added the pair ‘bill’:25*
- *Print myDictionary[‘bill’]*
  - *Prints 25*

Some Common Operators

- Just like “others”, dictionaries respond to these:
  - len(myDictionary)
    - Number of key:value pairs in the dictionary
  - Element in myDictionary
    - boolean, is element as a key in the dictionary
  - For key in myDictionary:
    - Iterates through the keys of a dictionary

Some Dictionary Methods

- myDictionary.items(): all the key/value pairs
- myDictionary.keys():   all the keys
- myDictionary.values(): all the values.
- myDictionary.has_key(key) or key in myDictionary: does the key exist in the dictionary.
- myDictionary.clear(): empty the dictionary.
- myDictionary.update(yourDictionary): for each key in yourDictionary, updates myDictionary with key/value pair.

Dictionaries are Iterable

```
for key in myDictionary:
    print key
```

- Prints all the keys

```
for key,value in myDictionary.items():
    print key,value
```

- Prints all the key/value pairs

```
for value in myDictionary.values():
    print value
```

- Prints all the values

Building Dictionaries faster

- *zip* creates pairs from two parallel lists
  - *zip (“abc”, [1, 2, 3]) yields:*
    [(‘a’, 1), (‘b’, 2), (‘c’, 3)]
- That’s good for building dictionaries, we call the *dict* function which takes a list of pairs to make a dictionary:
  - dict (zip (“abc”, [1, 2, 3]))
  - { ‘a’:1 , ‘c’:3 , ‘b’:2 }

Examples

- Simple Dicts
- Word Frequency
- PhoneBook

Storing functions

- Once a function is defined, it is an object just like any other variable.
- It can be stored as a value in a dictionary (indexed however you like)
- You can call the function by appending the () to the end of it.
- Very convenient!

Lookup of functions

- Look at the following:

```
commandSet = { “init”:initFromFile ,
“lookup”:lookup , “add”:addPair, “del”:delPair ,
“print”:printBook }


response = raw_input(“Command:”).strip()


commandSet[response]()
```

Any Questions?

Algorithm Design

Search and Sort Algorithms

Objectives

- To understand the basic techniques for analyzing the efficiency of algorithms.
- To know what searching is and understand the algorithms for linear and binary search.
- To understand sorting in depth and know the algorithms for selection sort and merge sort.

Searching

- ***Searching*** is the process of looking for a particular value in a collection.
- For example, a program that maintains a membership list for a club might need to look up information for a particular member – this involves some sort of search process.

A simple Searching Problem

- Here is the specification of a simple searching function:

```
def search(x, nums):
# nums is a list of numbers and x is a number
# Returns the position in the list where x occurs
# or -1 if x is not in the list.
```

- Here are some sample interactions:

```
>>> search(4, [3, 1, 4, 2, 5])
2
>>> search(7, [3, 1, 4, 2, 5])
-1
```

Built-in Python methods

- We can test to see if a value appears in a sequence using `in`.

```
if x in nums:
    # do something
```

- If we want to know the position of `x` in a list, the `index` method can be used.

```
>>> nums = [3, 1, 4, 2, 5]
>>> nums.index(4)
2
```

A Simple Searching Problem

- The only difference between our `search` function and `index` is that `index` raises an exception if the target value does not appear in the list.

- We could implement `search` using `index` by simply catching the exception and returning -1 for that case.

A Simple Searching Problem with Exception (more about this later)

-
  ```
  def search(x, nums):
      try:
          return nums.index(x)
      except:
          return -1
  ```

- Sure, this will work, but we are really interested in the algorithm used to actually search the list in Python!

Strategy 1: Linear Search

- **linear search**: you search through the list of items one by one until the target value is found.

```
def search(x, nums):
    for i in range(len(nums)):
        if nums[i] == x: # item found, return the index value
            return i
    return -1           # loop finished, item was not in list
```

- This algorithm wasn’t hard to develop, and works well for modest-sized lists.

Strategy 1: Linear Search

- The Python `in` and `index` operations both implement linear searching algorithms.
- If the collection of data is very large, it makes sense to organize the data somehow so that each data value doesn’t need to be examined.

Strategy 1: Linear Search

- If the data is sorted in ascending order (lowest to highest), we can skip checking some of the data.
- As soon as a value is encountered that is greater than the target value, the linear search can be stopped without looking at the rest of the data.
- On average, this will save us about half the work.

Strategy 2: Binary Search

- If the data is sorted, there is an even better searching strategy – one you probably already know!
- Have you ever played the number guessing game, where I pick a number between 1 and 100 and you try to guess it? Each time you guess, I'll tell you whether your guess is correct, too high, or too low. What strategy do you use?

Strategy 2: Binary Search

- Young children might simply guess numbers at random.
- Older children may be more systematic, using a linear search of 1, 2, 3, 4, … until the value is found.
- Most adults will first guess 50. If told the value is higher, it is in the range 51-100. The next logical guess is 75.

Strategy 2: Binary Search

- Each time we guess the middle of the remaining numbers to try to narrow down the range.
- This strategy is called ***binary search***.
- Binary means two, and at each step we are diving the remaining group of numbers into two parts.

Strategy 2: Binary Search

- We can use the same approach in our binary search algorithm! We can use two variables to keep track of the endpoints of the range in the sorted list where the number could be.

- Since the target could be anywhere in the list, initially `low` is set to the first location in the list, and `high` is set to the last.

Strategy 2: Binary Search

- The heart of the algorithm is a loop that looks at the middle element of the range, comparing it to the value `x`.
- If `x` is smaller than the middle item, `high` is moved so that the search is confined to the lower half.
- If `x` is larger than the middle item, `low` is moved to narrow the search to the upper half.

Strategy 2: Binary Search

- The loop terminates when either
  - `x` is found
  - There are no more places to look (`low` > `high`)

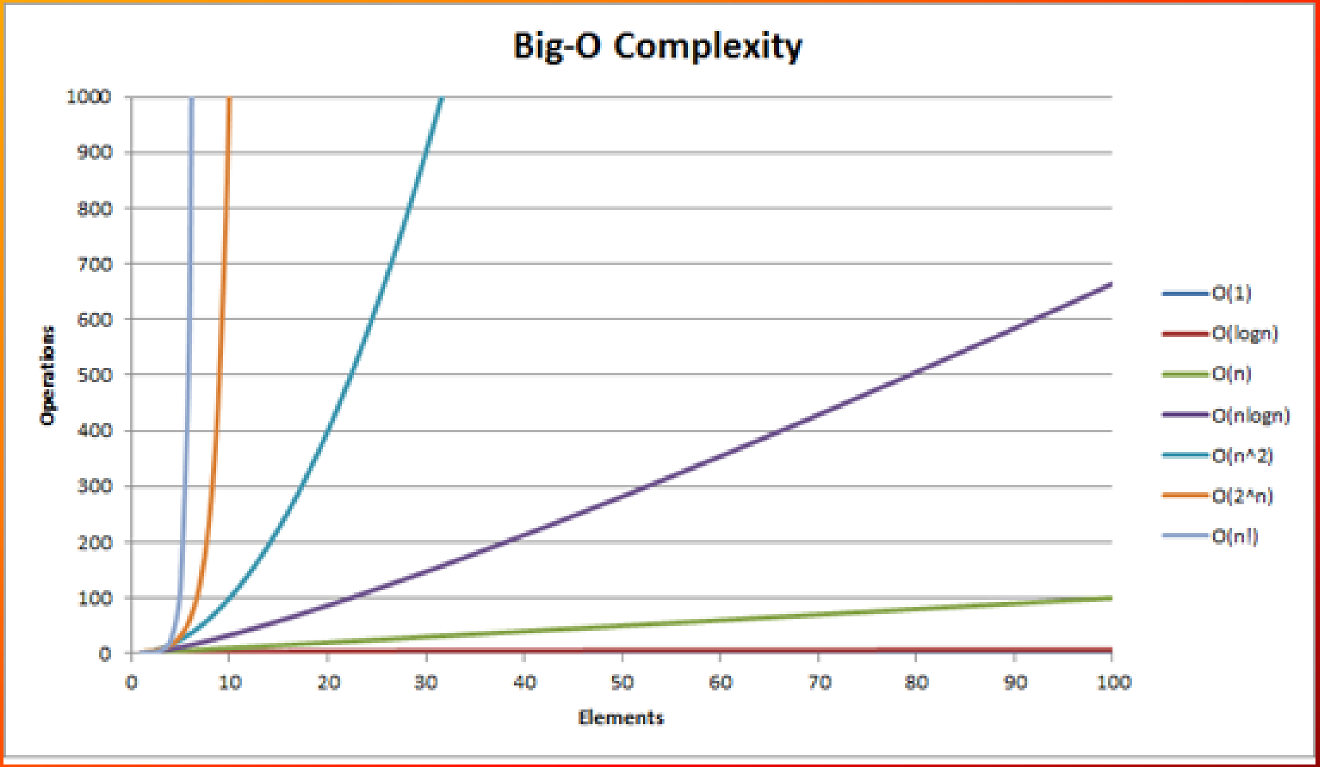
```
def search(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:          # There is a range to search
        mid = (low + high)//2 # Position of middle item
        item = nums[mid]
        if x == item:           # Found it! Return the index
            return mid
        elif x < item:          # x is in lower half of range
            high = mid - 1      #  move top marker down
        else:                   # x is in upper half of range
            low = mid + 1       #  move bottom marker up
    return -1                   # No range left to search,
                                # x is not there
```

Comparing Algorithms

- Which search algorithm is better, linear or binary?
  - The linear search is easier to understand and implement
  - The binary search is more efficient since it doesn’t need to look at each element in the list
- Intuitively, we might expect the linear search to work better for small lists, and binary search for longer lists. But how can we be sure?

Big-O notations (remember!)

- Linear Search: O(n)
- Binary Search: O(log(n))



Sorting Algorithms

- The basic sorting problem is to take a list and rearrange it so that the values are in increasing (or non-decreasing) order.

Naive Sorting: Selection Sort

- To start out, pretend you’re the computer, and you’re given a shuffled stack of index cards, each with a number. How would you put the cards back in order?

Naive Sorting: Selection Sort

- One simple method is to look through the card deck to find the smallest value and place that value at the front of the stack.
- Then go through, find the next smallest number in the remaining cards, place it behind the smallest card at the front.
- Rinse, lather, repeat, until the stack is in sorted order!

Naive Sorting: Selection Sort

- The algorithm has a loop, and each time through the loop the smallest remaining element is selected and moved into its proper position.
  - For *n* elements, we find the smallest value and put it in the *0th* position.
  - Then we find the smallest value from position 1 –> (*n*-1) and put it into position 1.
  - The smallest value from position 2 –> (*n*-1) goes in position 2.
  - … etc.

```
def selSort(nums):
  # sort nums into ascending order
    n = len(nums)
  # For each position in the list (except the very last)
    for bottom in range(n-1):
  # find the smallest item in nums[bottom]..nums[n-1]
        mp = bottom       # bottom is smallest initially
        for i in range(bottom+1, n): # look at each position
            if nums[i] < nums[mp]:   # this one is smaller
                mp = i               # remember its index
        # swap smallest item to the bottom
        nums[bottom], nums[mp] = nums[mp], nums[bottom]
```

Naive Sorting: Selection Sort

- The selection sort is easy to write and works well for moderate-sized lists, but is not terribly efficient.

Divide and Conquer: Merge Sort

- We’ve seen how divide and conquer works in other types of problems. How could we apply it to sorting?
- Say you and your friend have a deck of shuffled cards you’d like to sort. Each of you could take half the cards and sort them. Then all you’d need is a way to recombine the two sorted stacks!

Divide and Conquer: Merge Sort

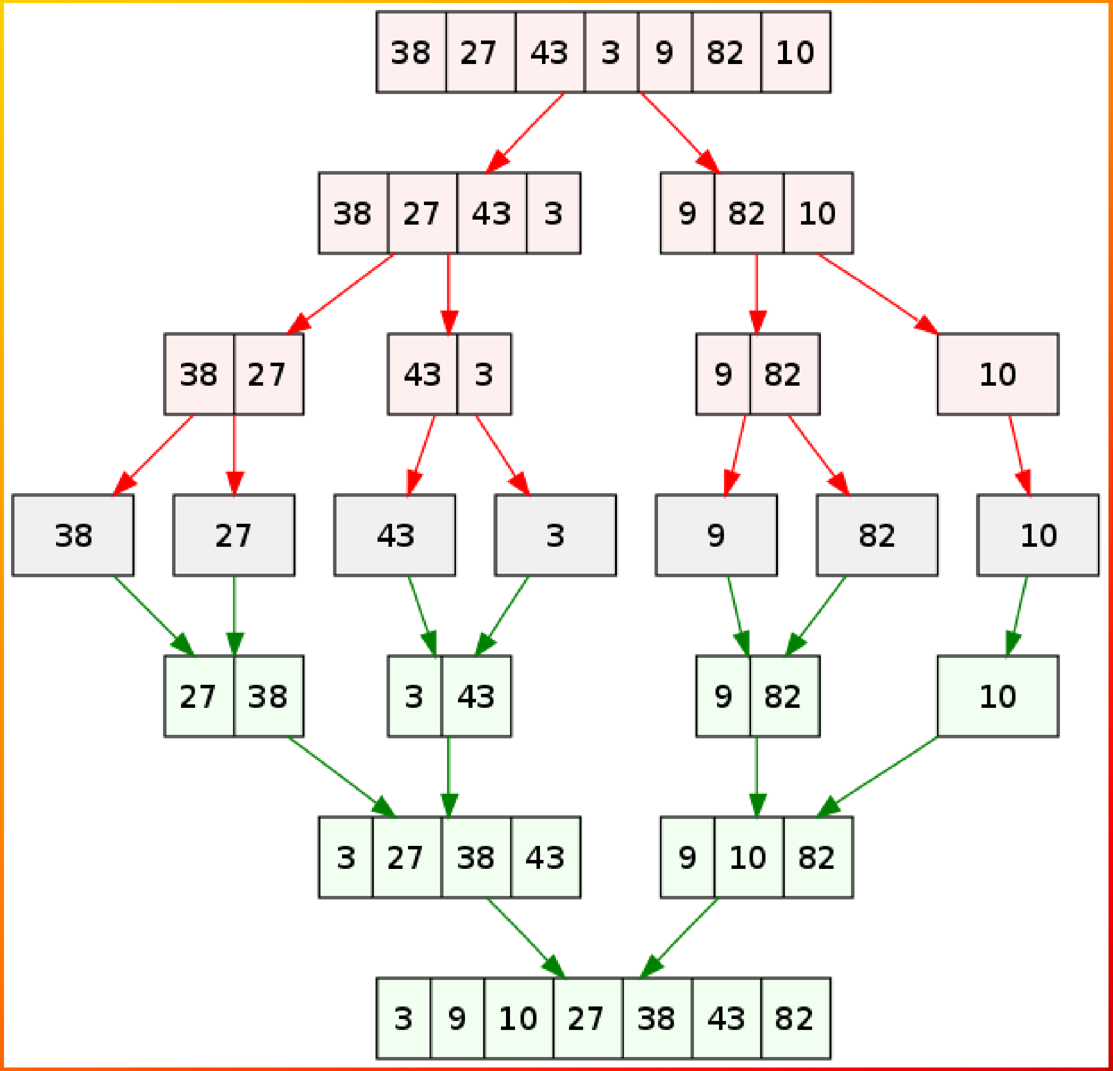
- This process of combining two sorted lists into a single sorted list is called *merging*.

- Our *merge sort* algorithm looks like:

```
split nums into two halves
sort the first half
sort the second half
merge the two sorted halves back into nums
```

Divide and Conquer: Merge Sort

- Once the smaller value is removed, examine both top cards. Whichever is smaller will be the next item in the list.
- Continue this process of placing the smaller of the top two cards until one of the stacks runs out, in which case the list is finished with the cards from the remaining stack.
- In the following code, `lst1` and `lst2` are the smaller lists and `lst3` is the larger list for the results. The length of `lst3` *must* be equal to the sum of the lengths of `lst1` and `lst2`.


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

```
def merge(lst1, lst2, lst3):
 # merge sorted lists lst1 and lst2 into lst3
 # these indexes keep track of current position in each list
    i1, i2, i3 = 0, 0, 0  # all start at the front
    n1, n2 = len(lst1), len(lst2)
 # Loop while both lst1 and lst2 have more items
    while i1 < n1 and i2 < n2:
        if lst1[i1] < lst2[i2]: # top of lst1 is smaller
             lst3[i3] = lst1[i1] #  copy it into current spot
             i1 = i1 + 1
        else:                     # top of lst2 is smaller
             lst3[i3] = lst2[i2] #  copy it into current spot
             i2 = i2 + 1
        i3 = i3 + 1              # item added to lst3 update
                                 # position
 # Here either lst1 or lst2 is done.
 # One of the following loops
 # will execute to finish up the merge.

# Copy remaining items (if any) from lst1
  while i1 < n1:
      lst3[i3] = lst1[i1]
      i1 = i1 + 1
      i3 = i3 + 1

# Copy remaining items (if any) from lst2
  while i2 < n2:
      lst3[i3] = lst2[i2]
      i2 = i2 + 1
      i3 = i3 + 1
```

Divide and Conquer: Merge Sort

- We can slice a list in two, and we can merge these new sorted lists back into a single list. How are we going to sort the smaller lists?

- We are trying to sort a list, and the algorithm requires two smaller sorted lists… this sounds like a job for recursion!

Divide and Conquer: Merge Sort

- We need to find at least one base case that does not require a recursive call, and we also need to ensure that recursive calls are always made on smaller versions of the original problem.
- For the latter, we know this is true since each time we are working on halves of the previous list.

Divide and Conquer: Merge Sort

- Eventually, the lists will be halved into lists with a single element each. What do we know about a list with a single item?
- It’s already sorted!! We have our base case! When the length of the list is less than 2, we do nothing.

Divide and Conquer: Merge Sort (pseudocode)

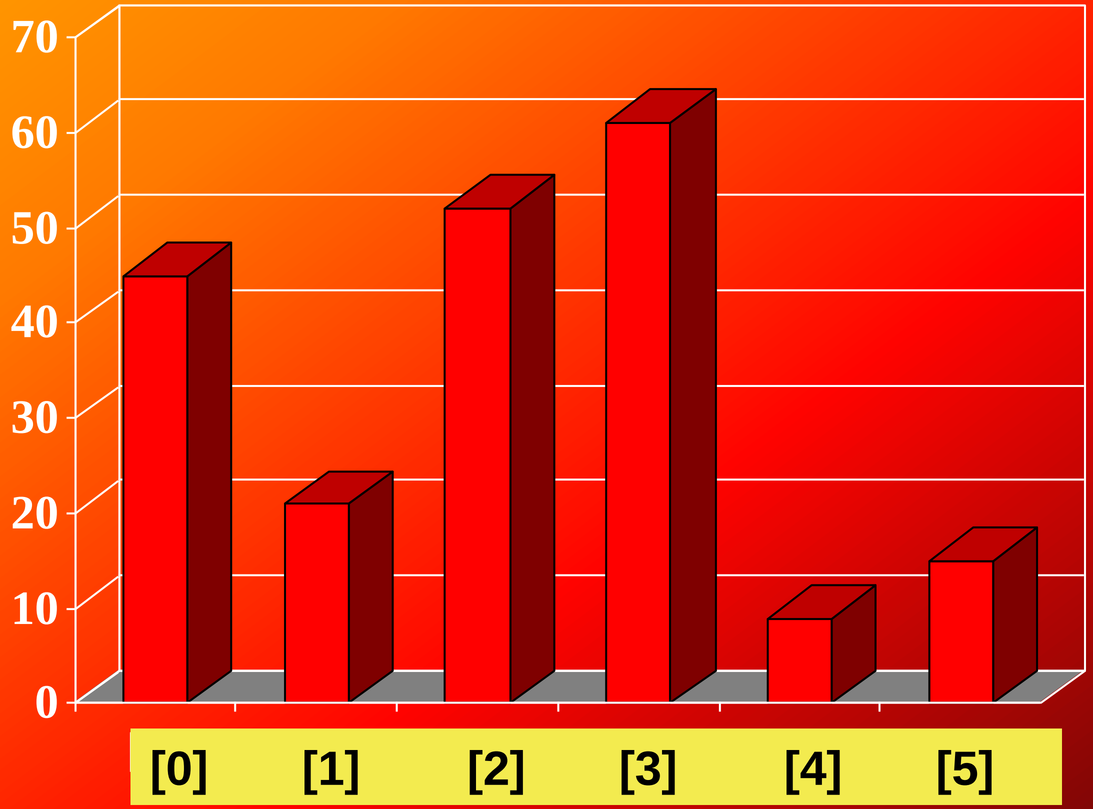
```
if len(nums) > 1:
    split nums into two halves
    mergeSort the first half
    mergeSort the second half
    merge the two sorted halves back into nums
```

Divide and Conquer: Merge Sort

```
def mergeSort(nums):
    # Put items of nums into ascending order
    n = len(nums)
    # Do nothing if nums contains 0 or 1 items
    if n > 1:
        # split the two sublists
        m = n/2
        nums1, nums2 = nums[:m], nums[m:]
        # recursively sort each piece
        mergeSort(nums1)
        mergeSort(nums2)
        # merge the sorted pieces back into original list
        merge(nums1, nums2, nums)
```

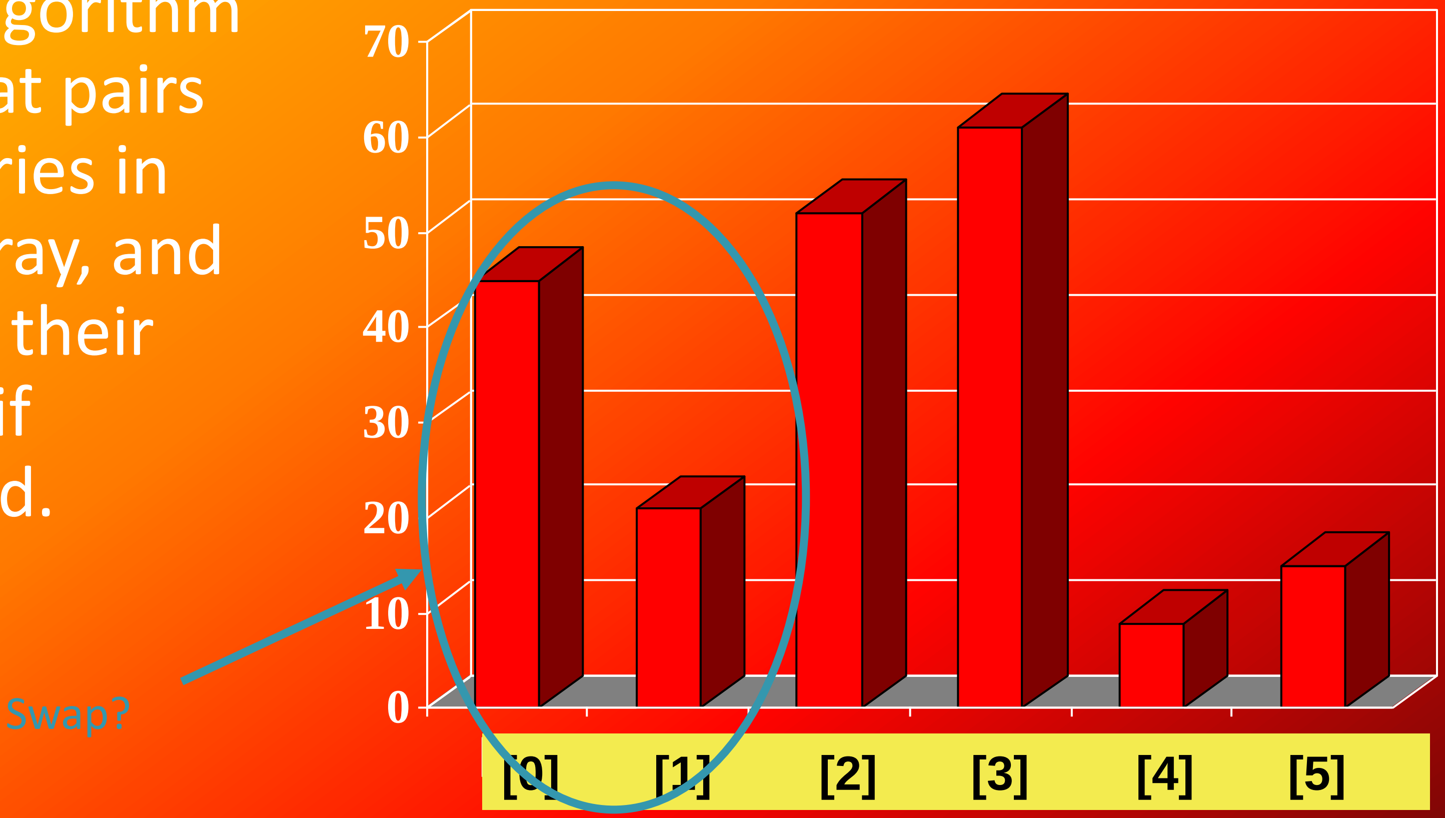
Bubble Sort

- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



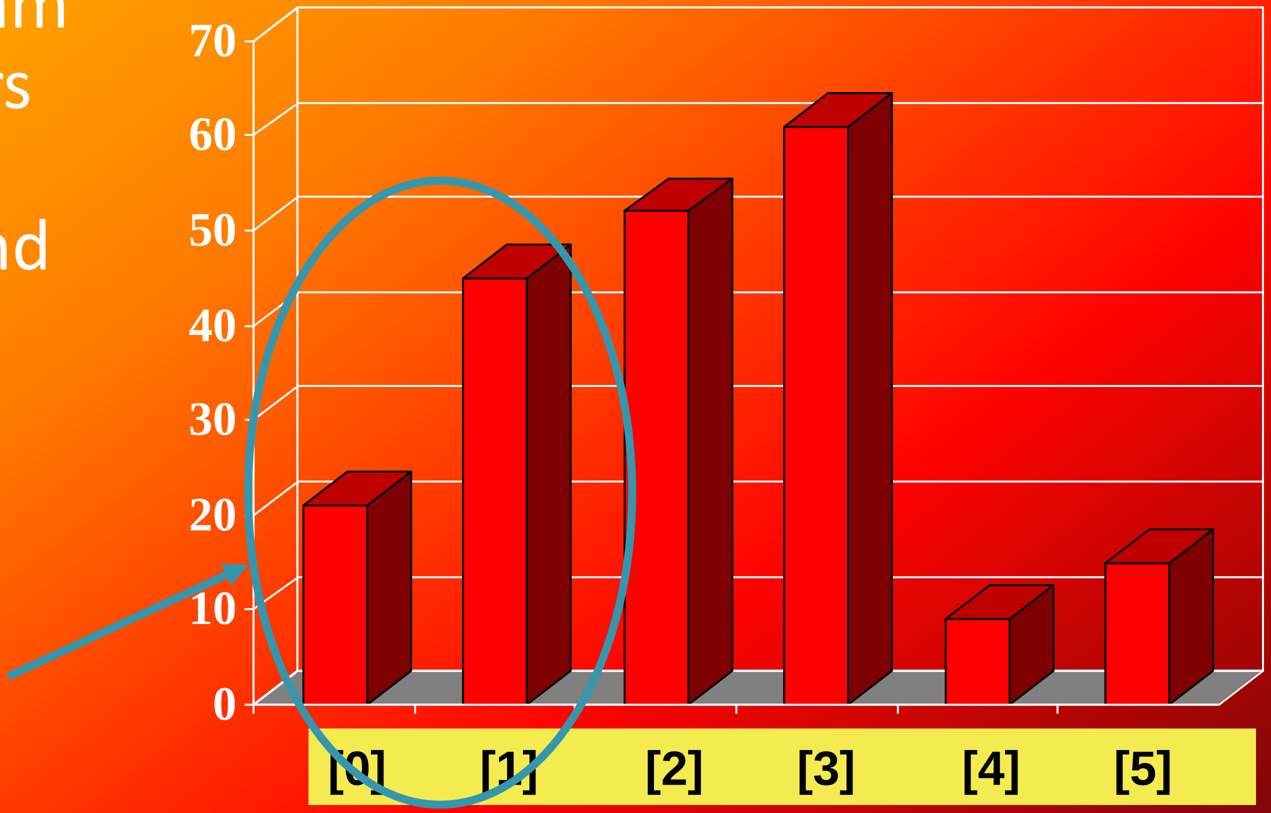
Bubble Sort

- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Bubble Sort

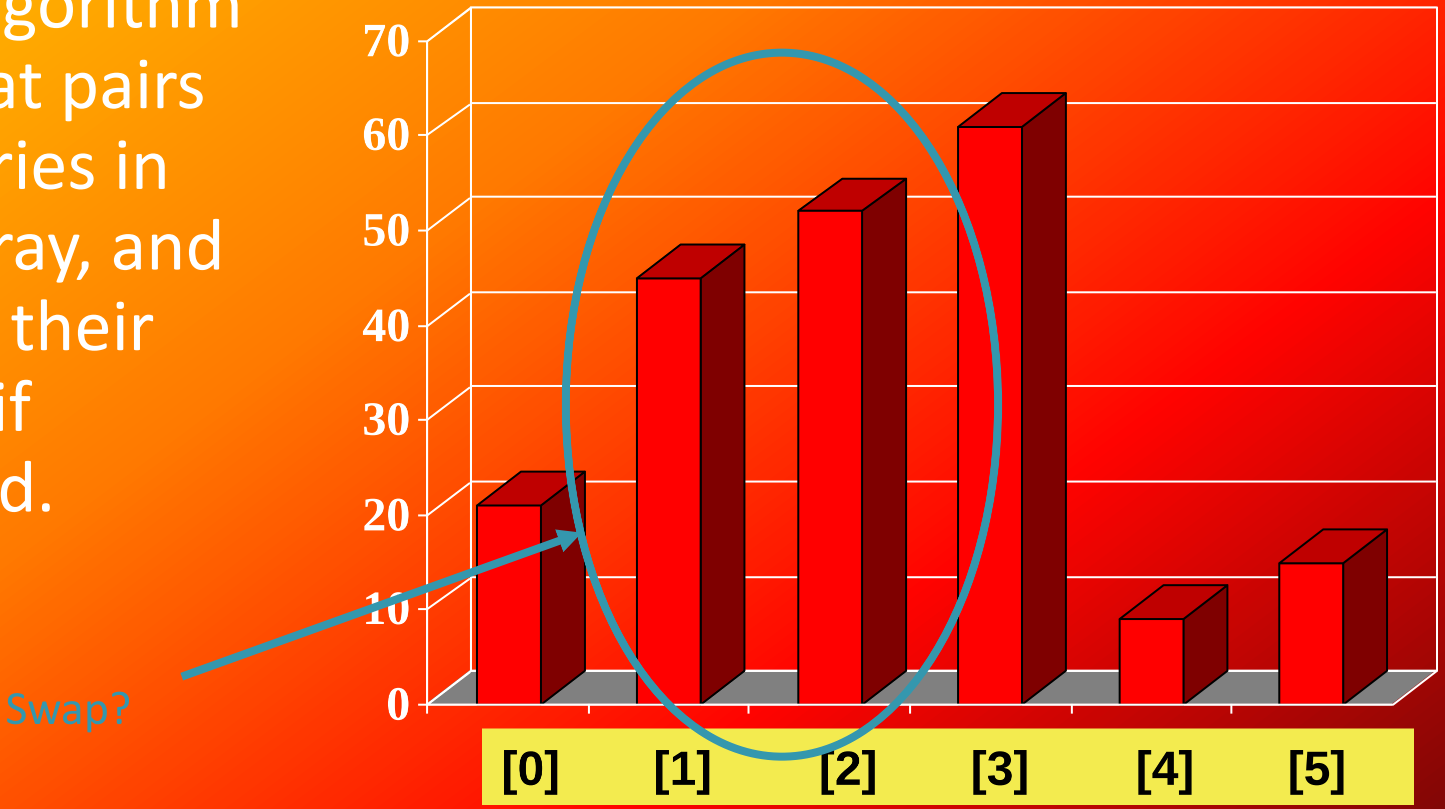
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Yes!

Bubble Sort

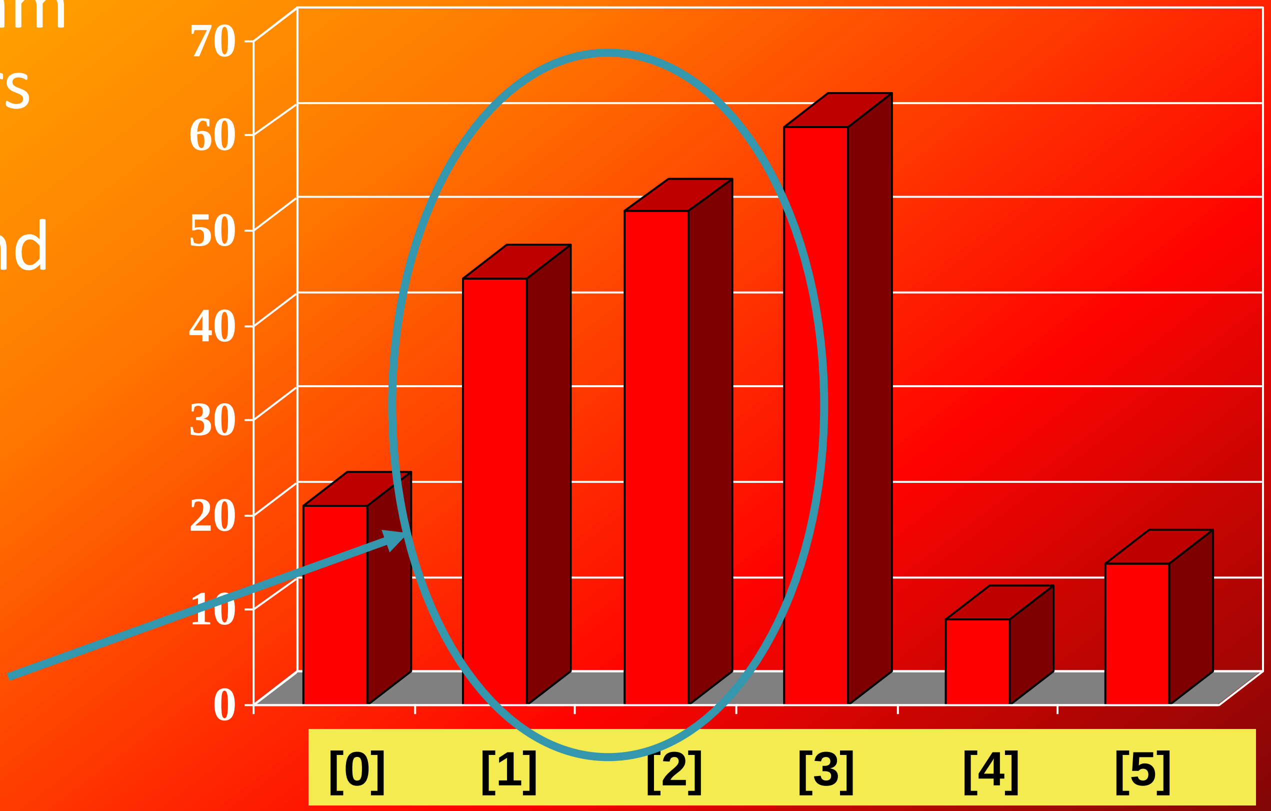
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Bubble Sort

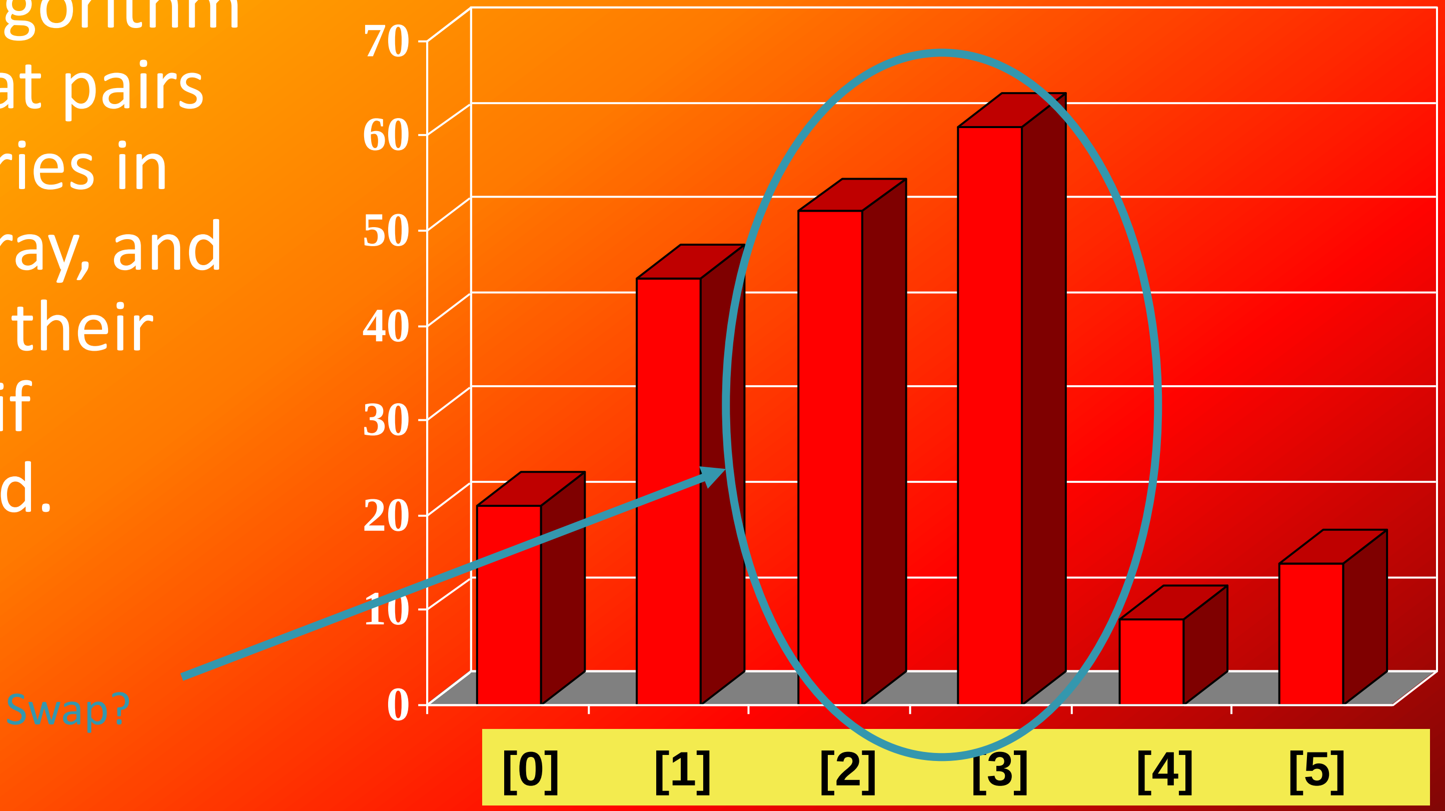
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

No.



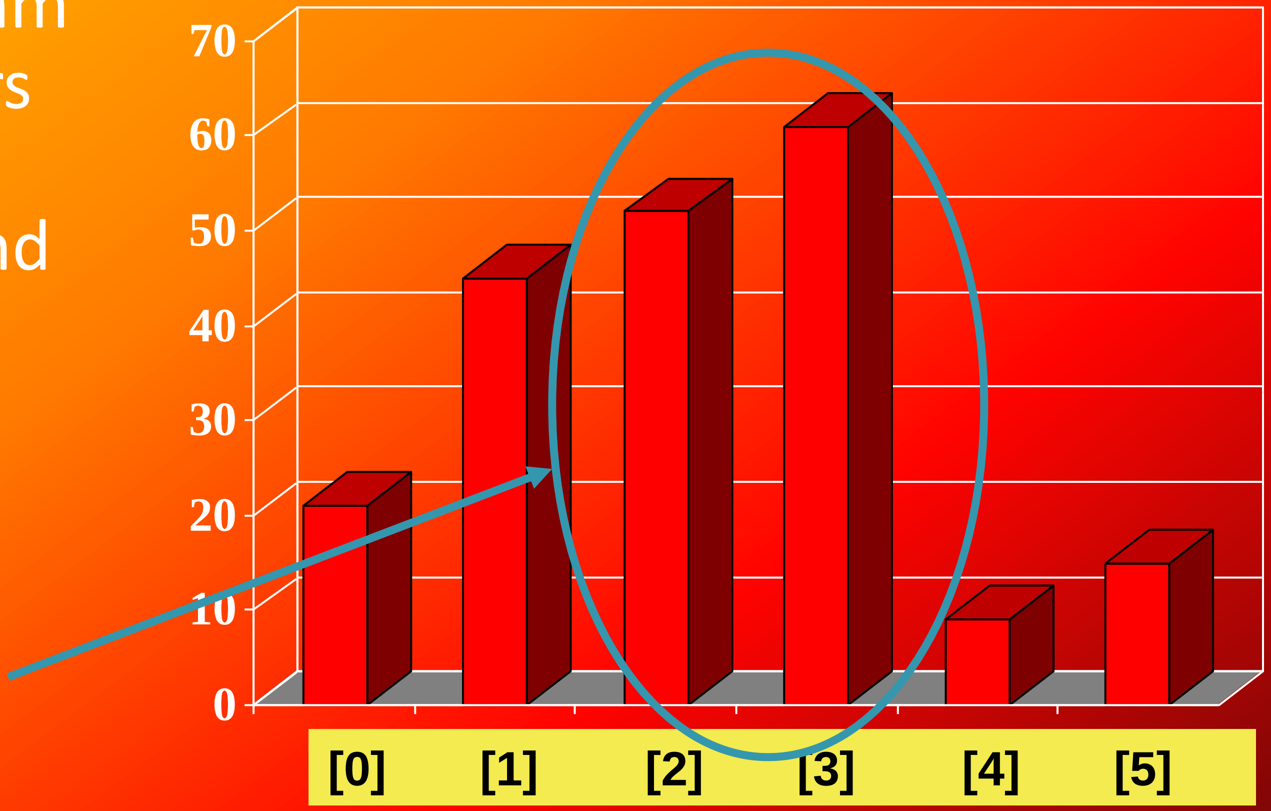
Bubble Sort

- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Bubble Sort

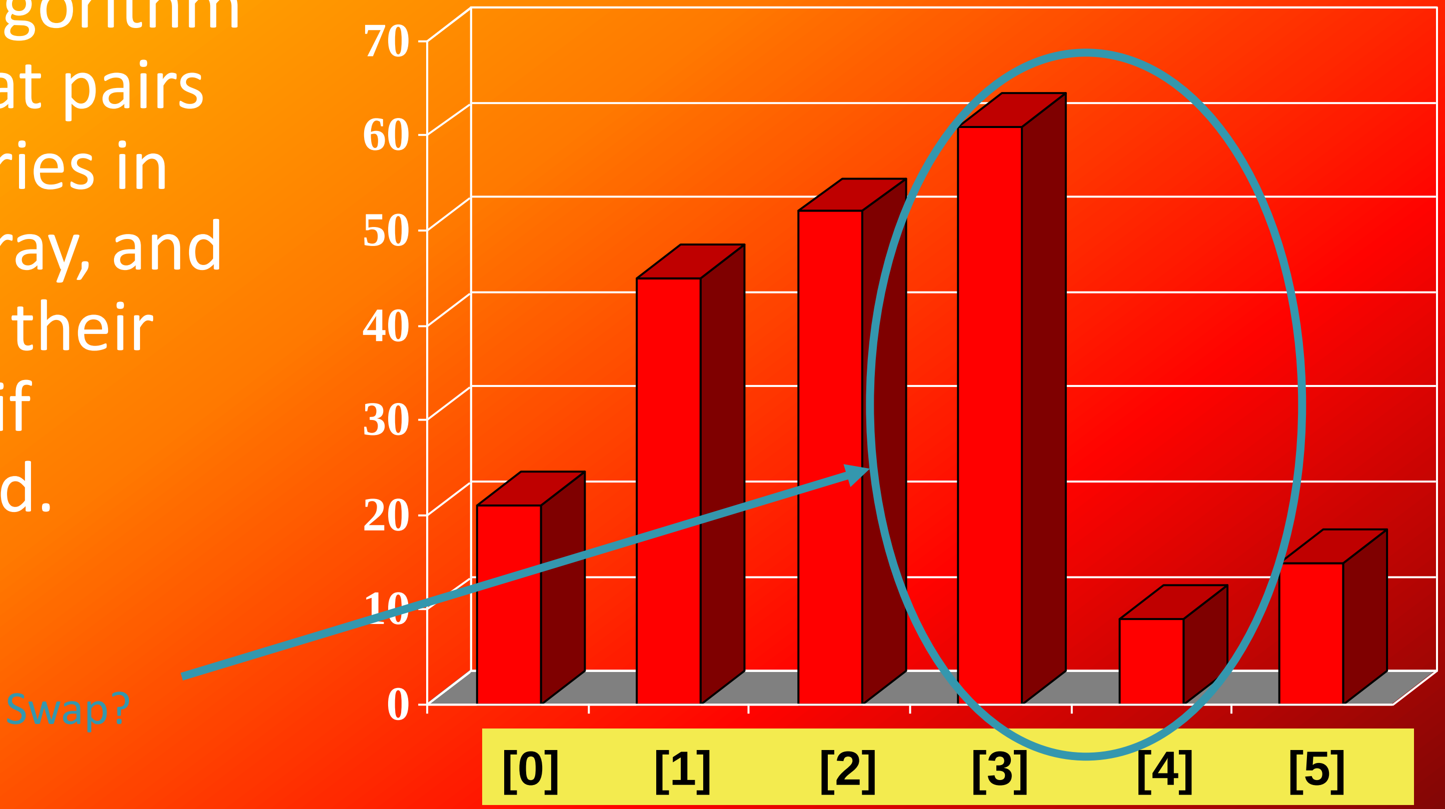
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



No.

Bubble Sort

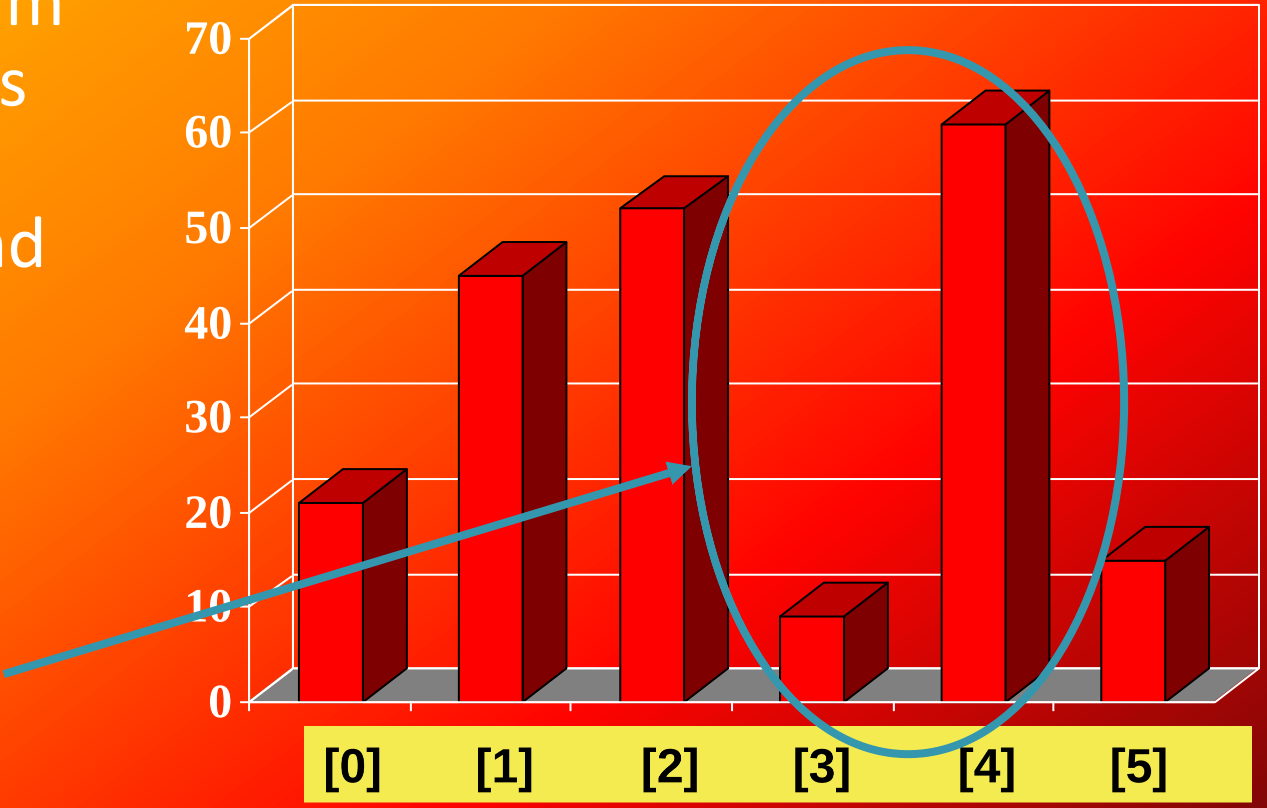
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Bubble Sort

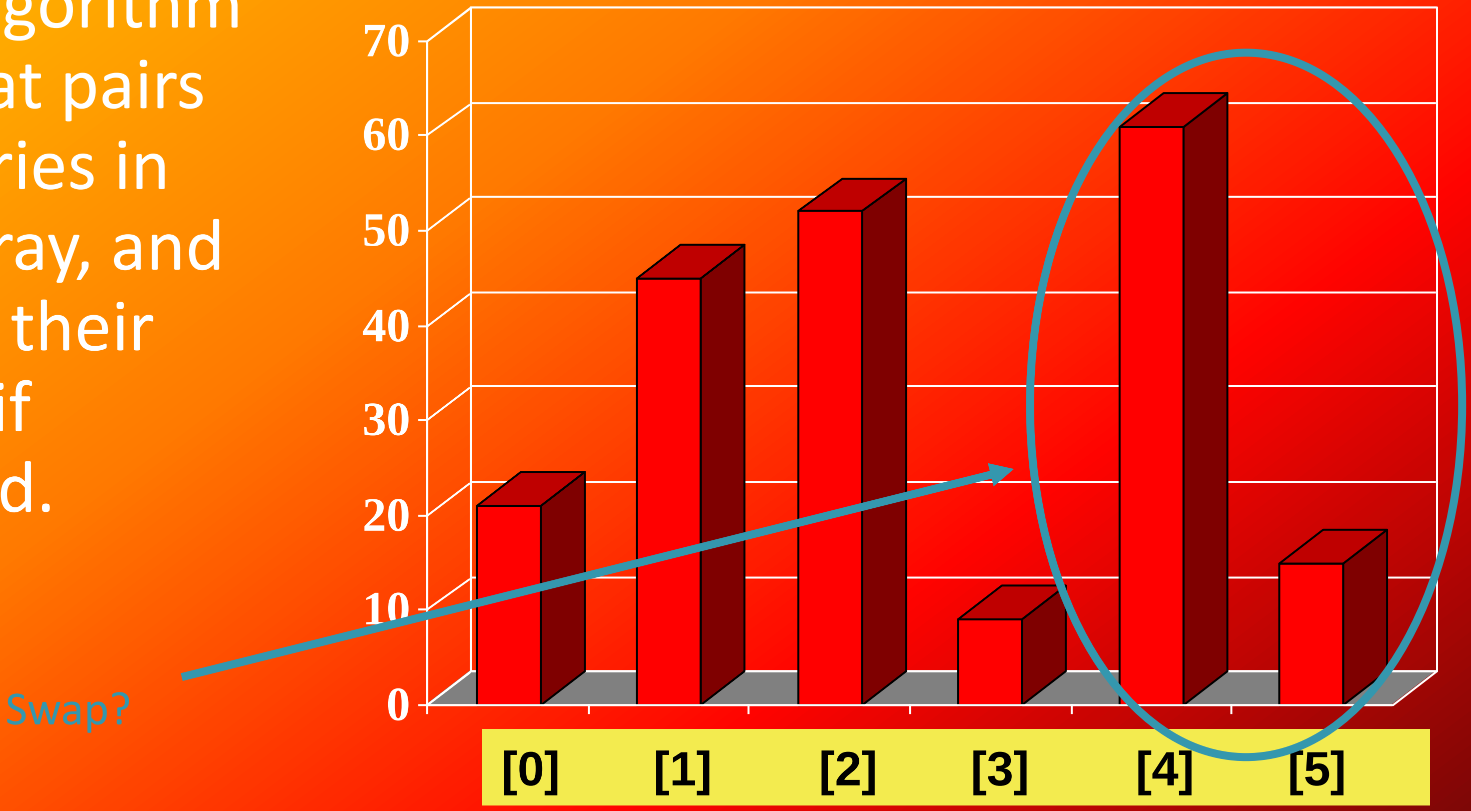
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



Bubble Sort

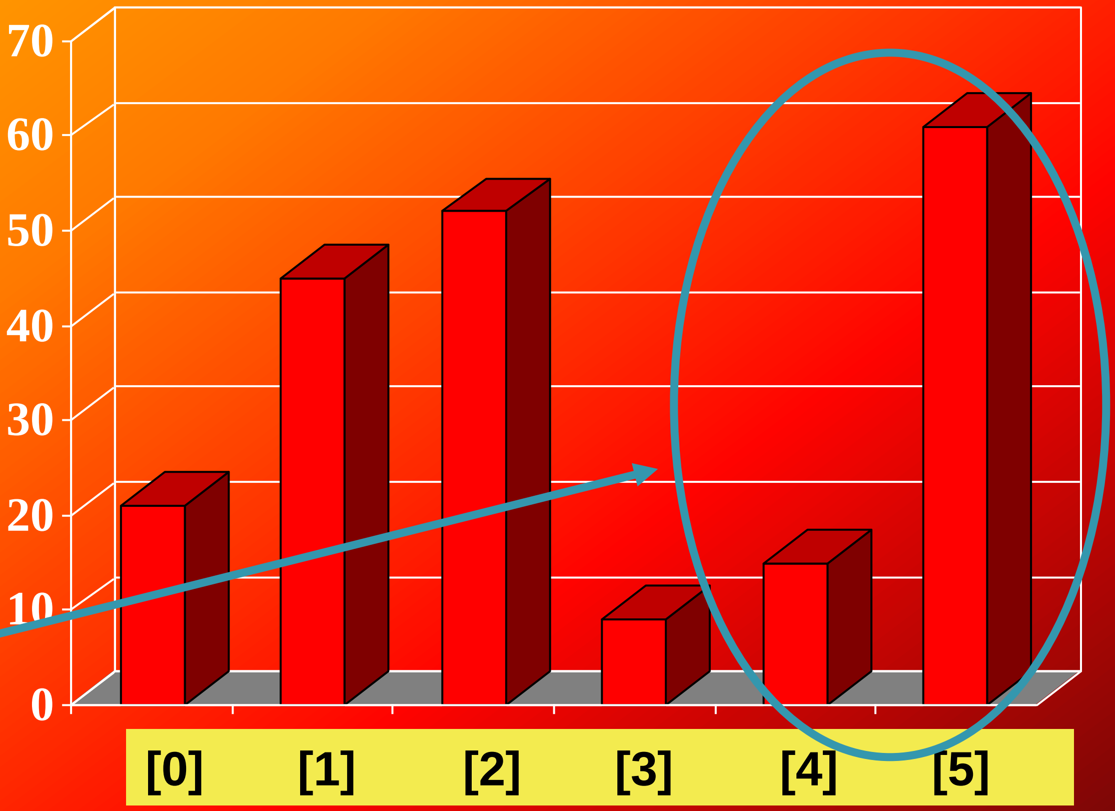
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.



Bubble Sort

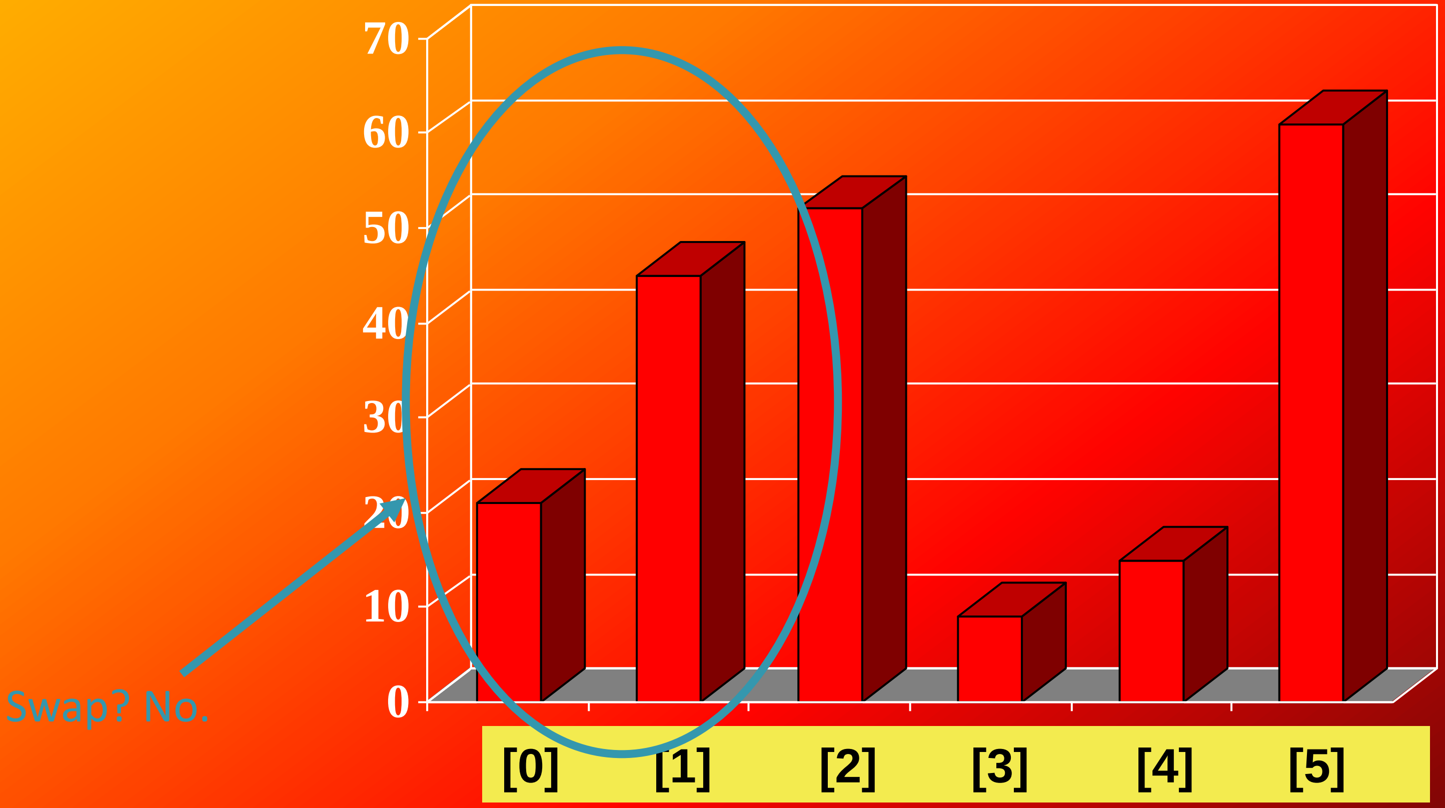
- The Bubble Sort algorithm looks at pairs of entries in the array, and swaps their order if needed.

Yes!



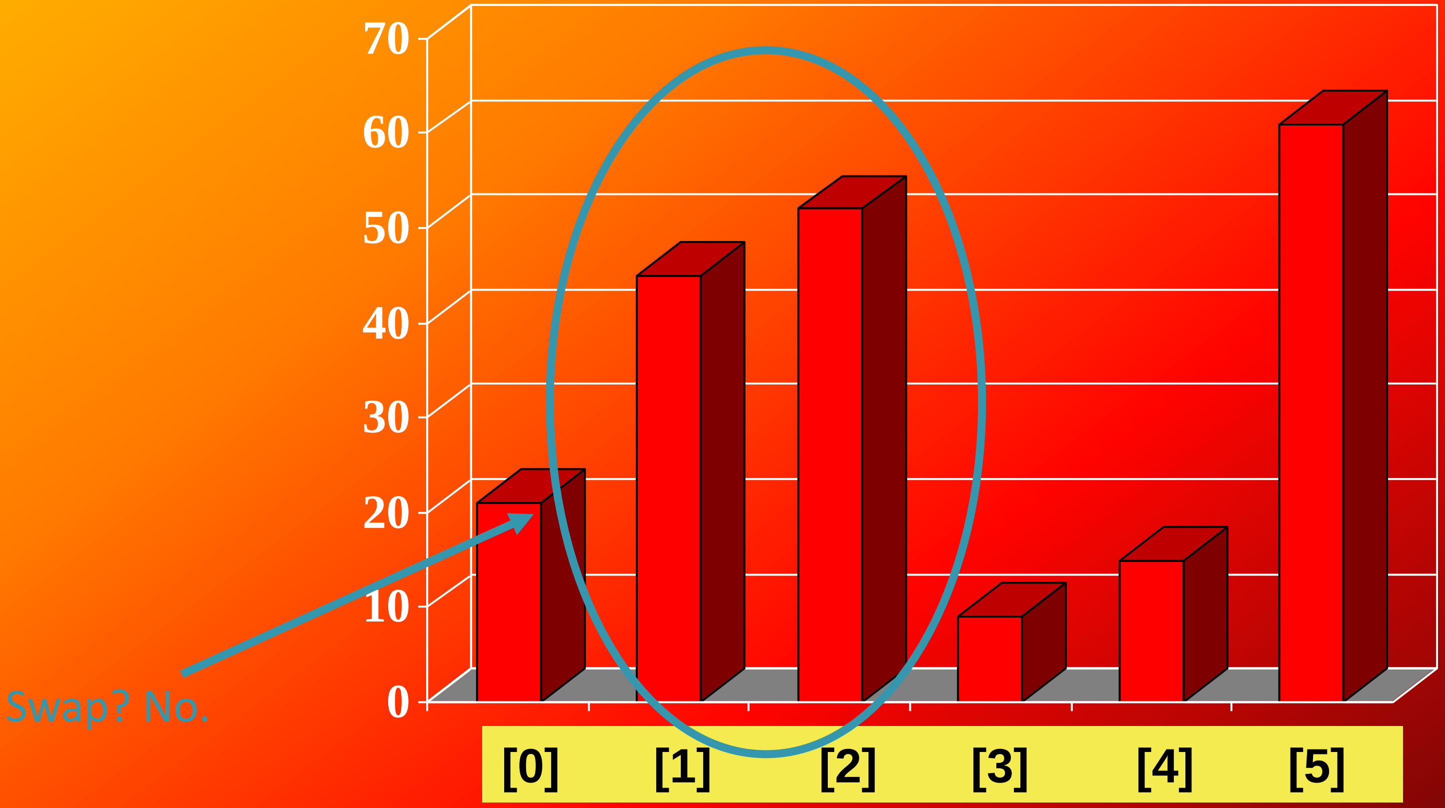
Bubble Sort

- Repeat.



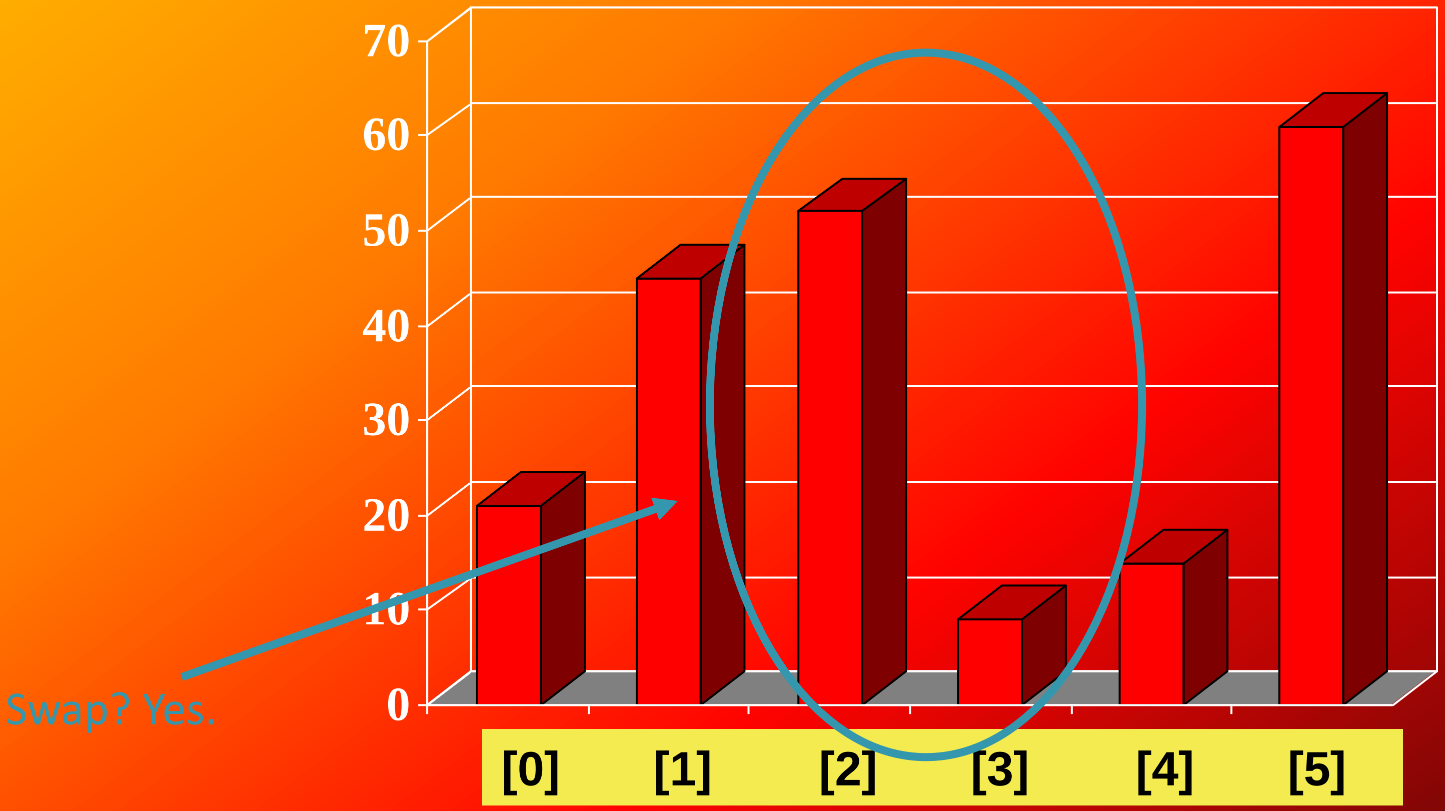
Bubble Sort

- Repeat.



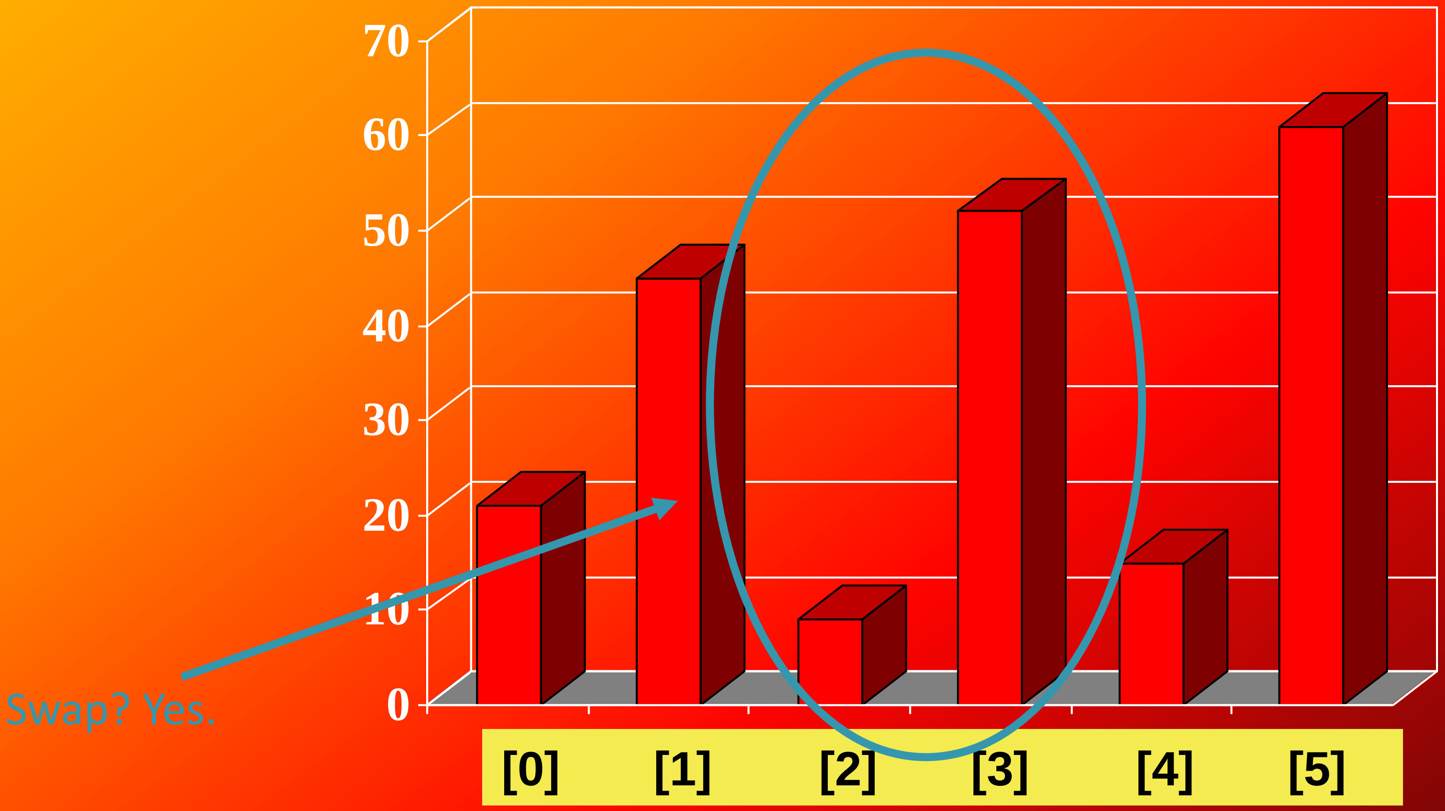
Bubble Sort

- Repeat.



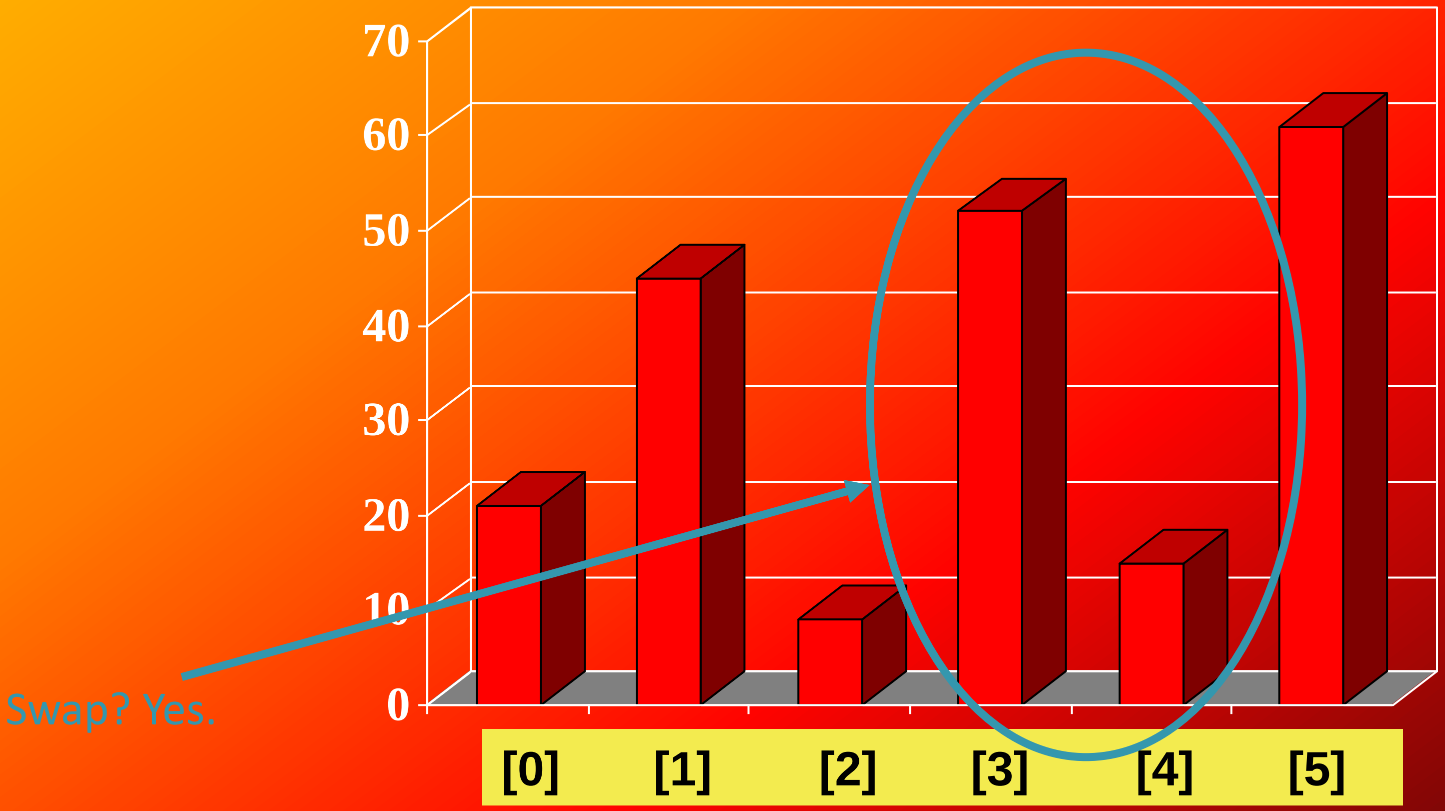
Bubble Sort

- Repeat.



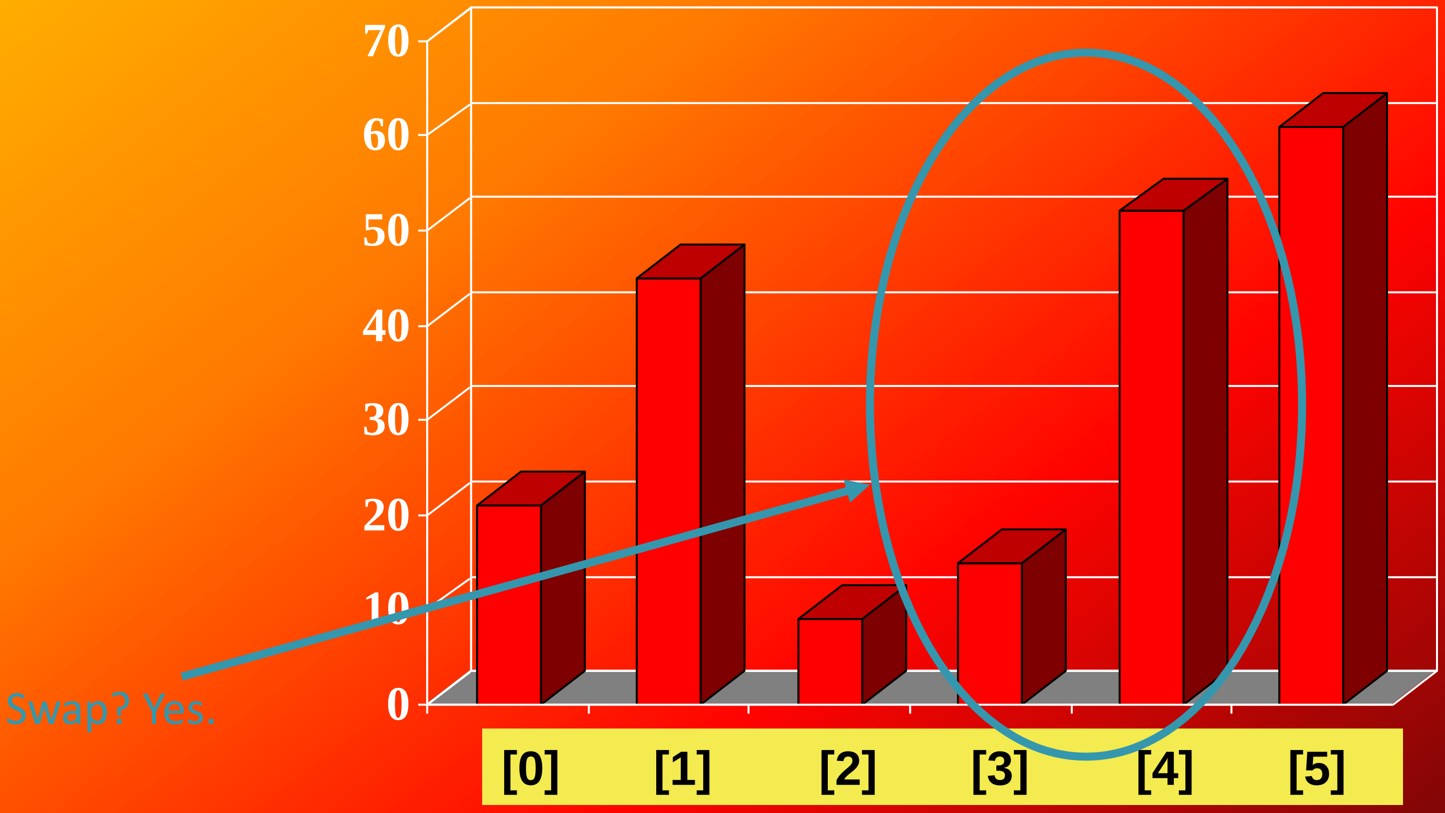
Bubble Sort

- Repeat.



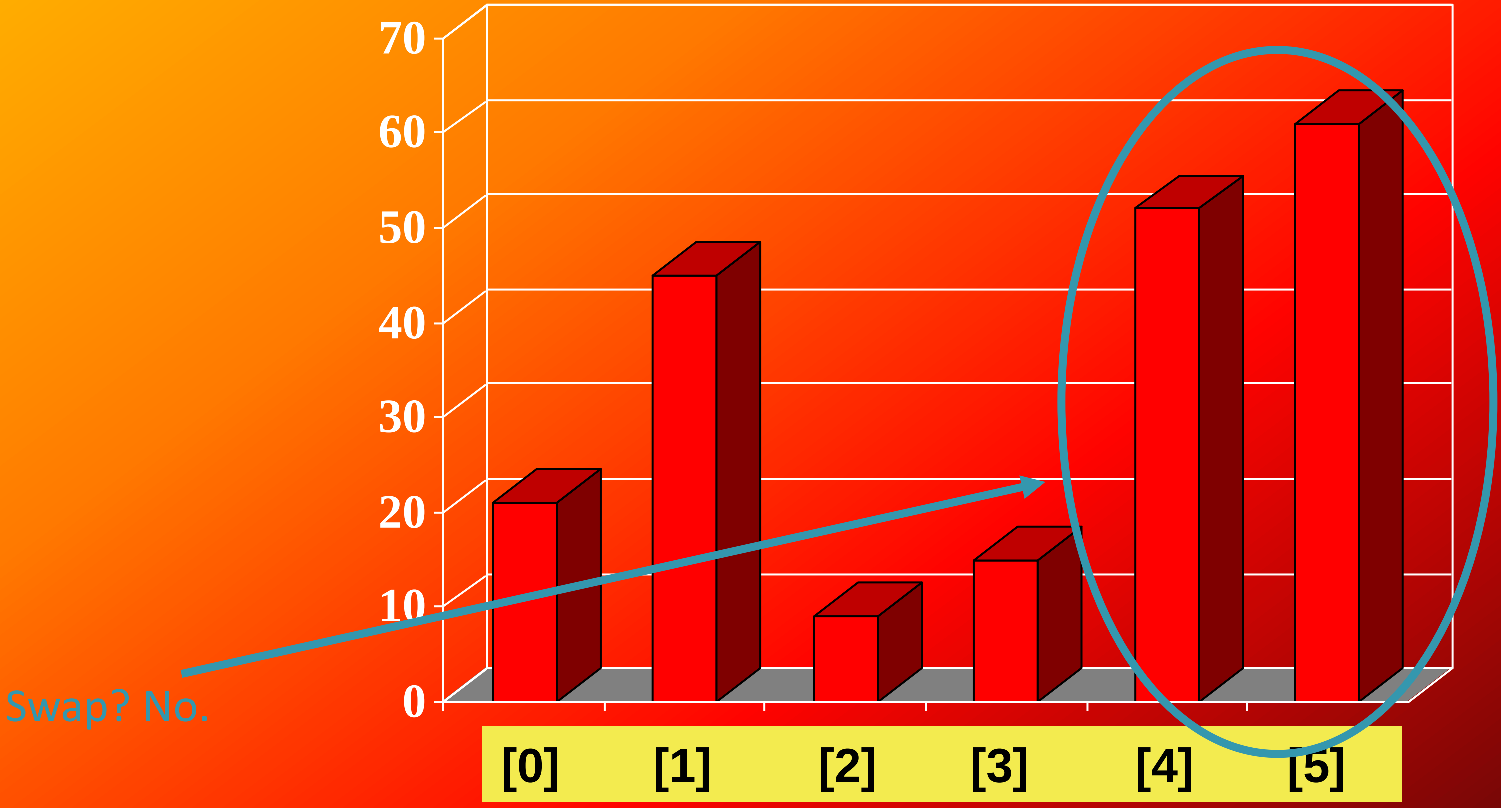
Bubble Sort

- Repeat.



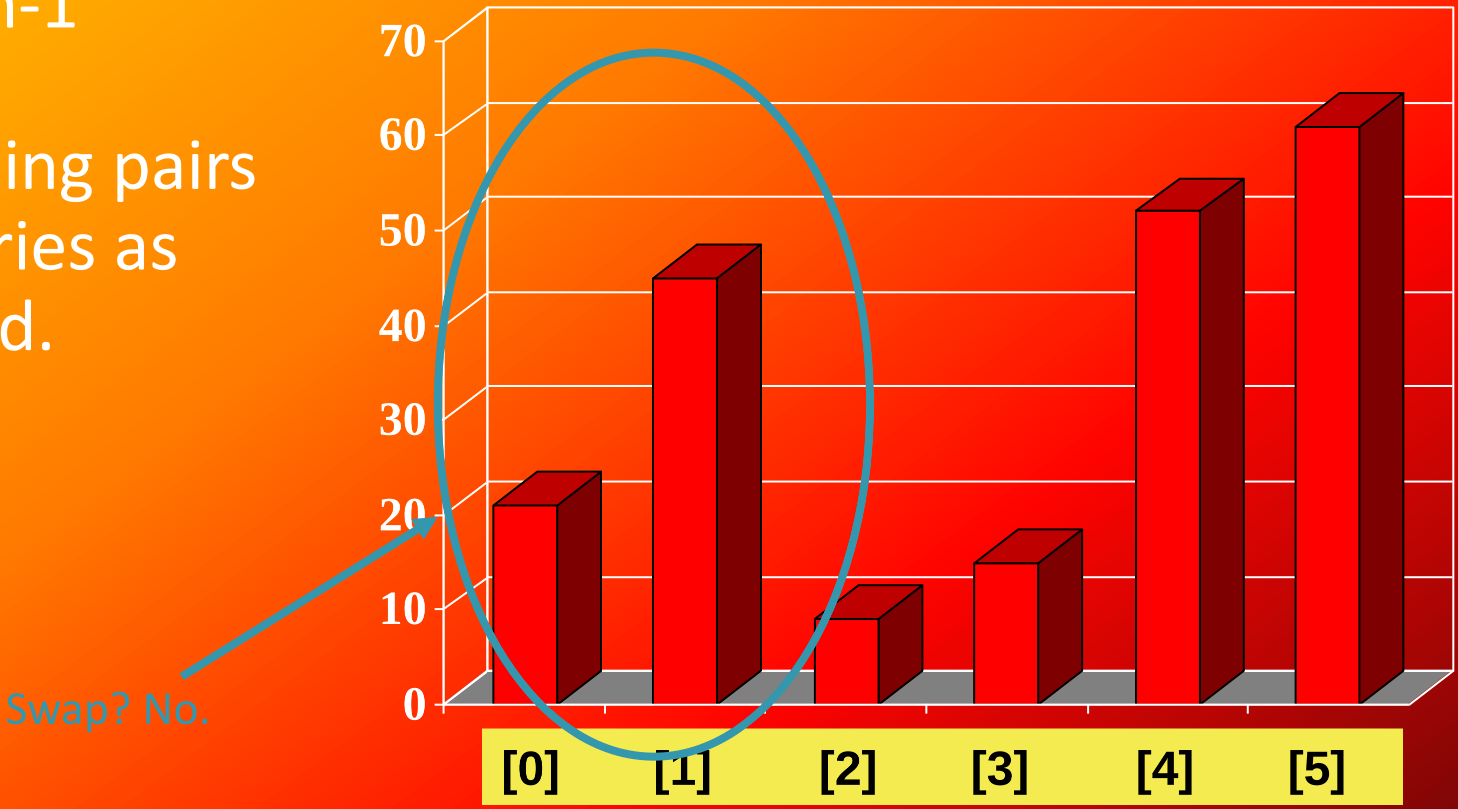
Bubble Sort

- Repeat.



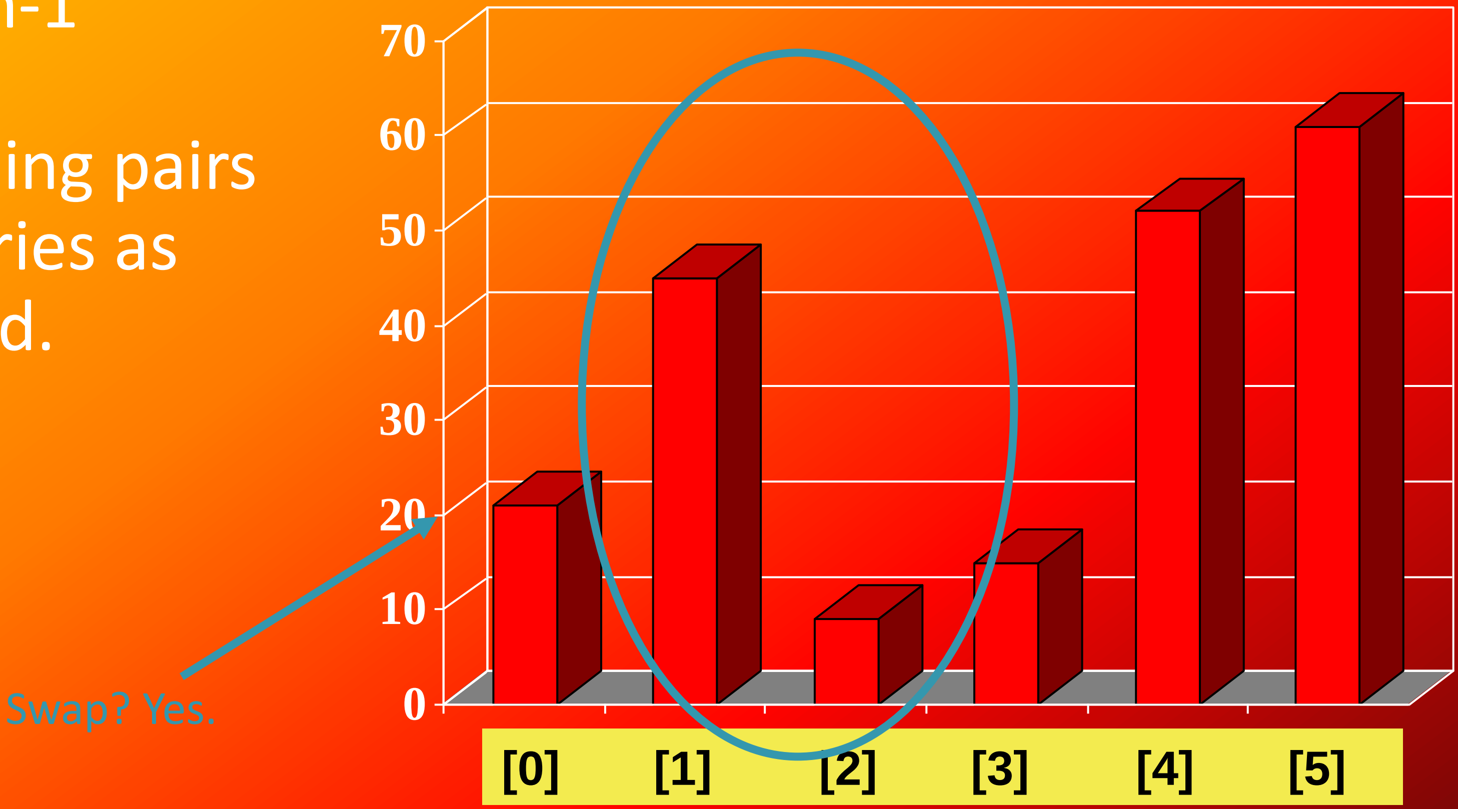
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



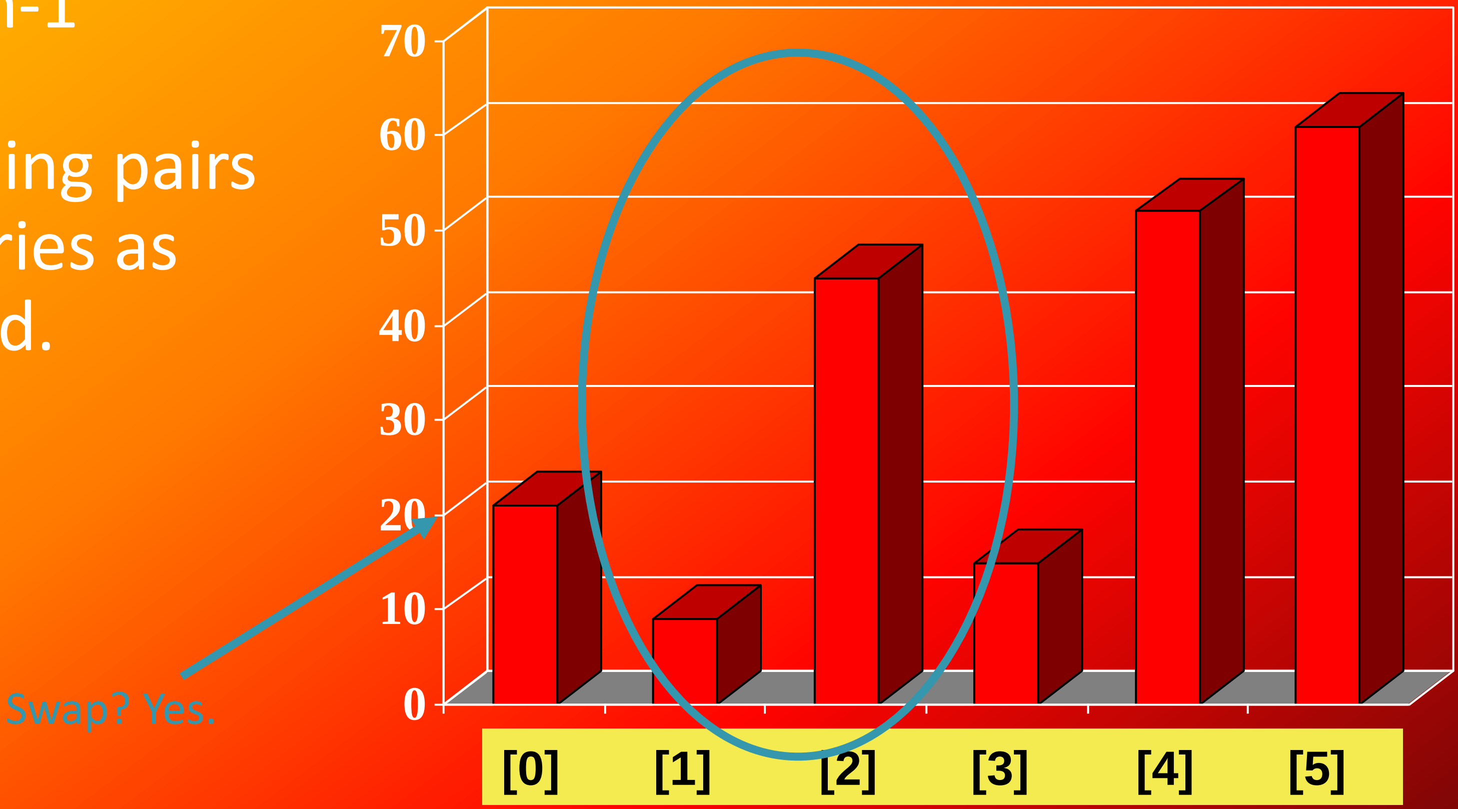
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



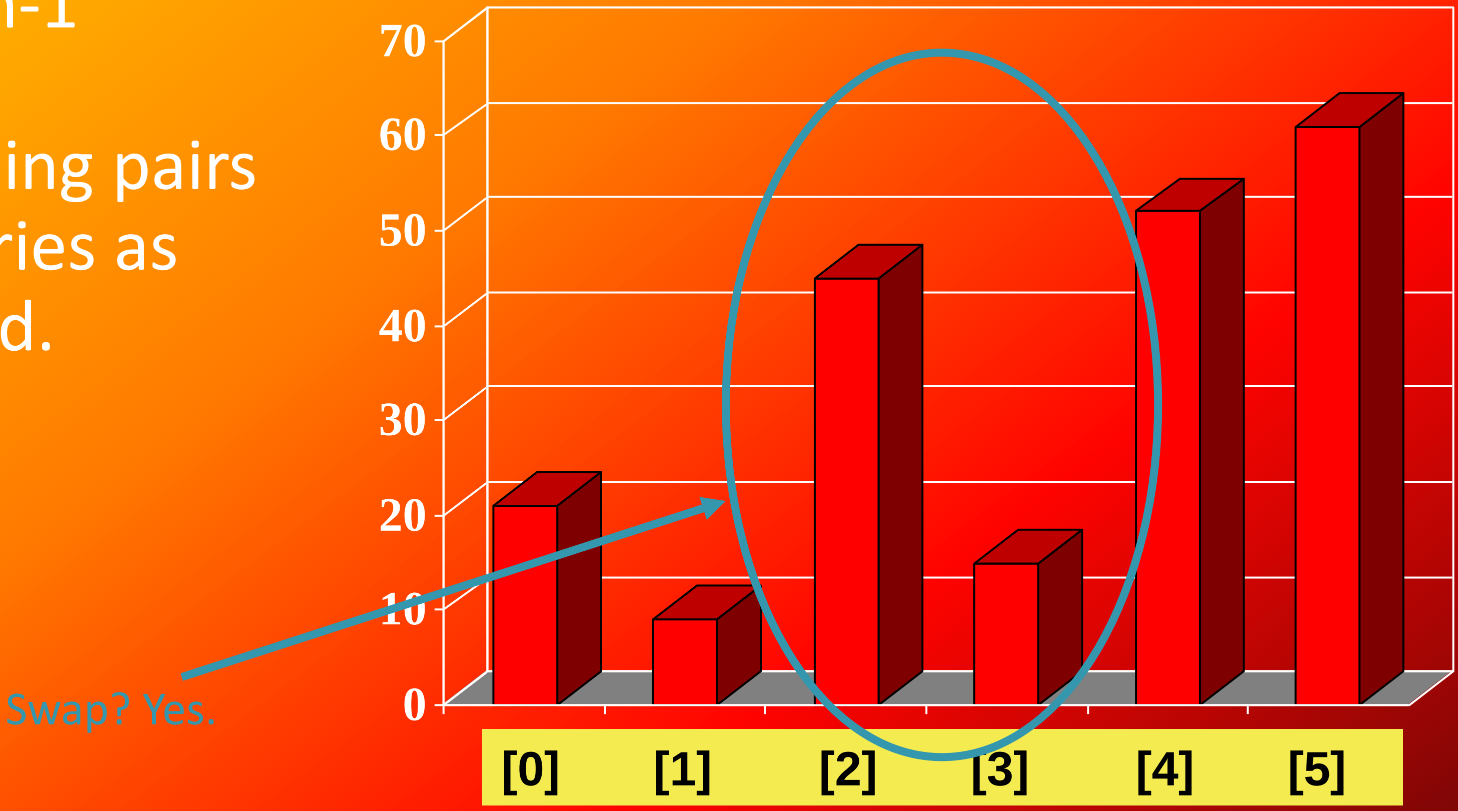
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



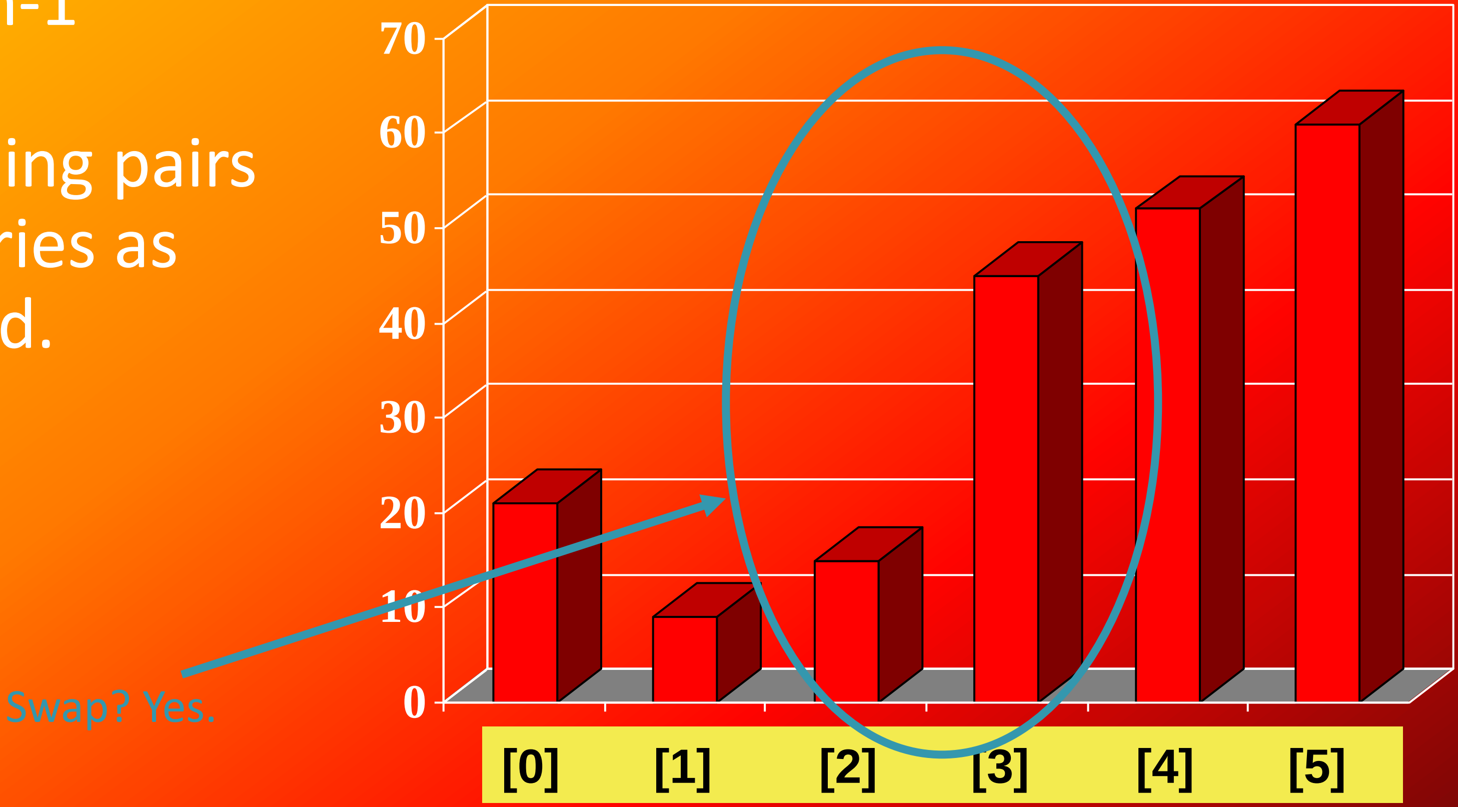
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



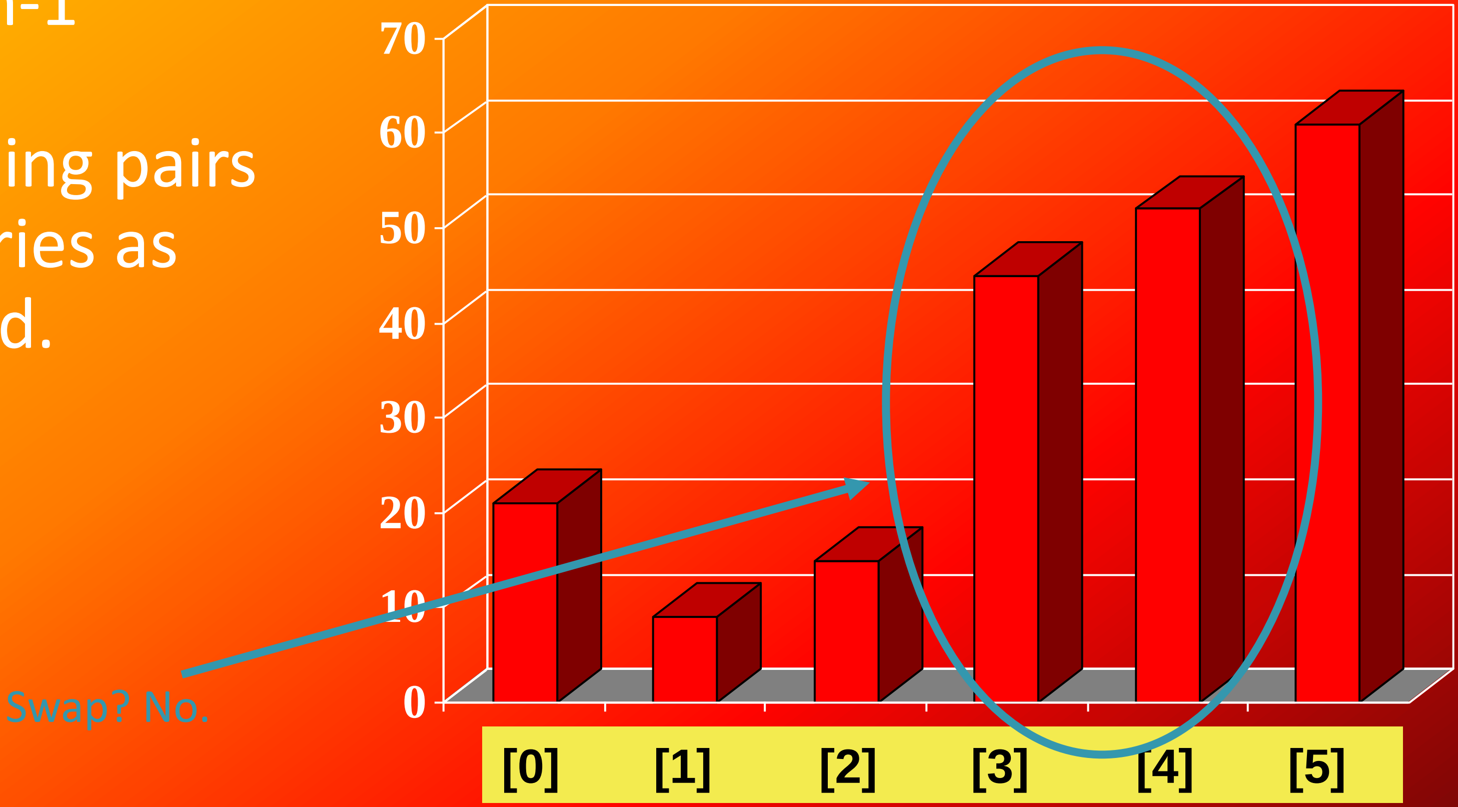
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



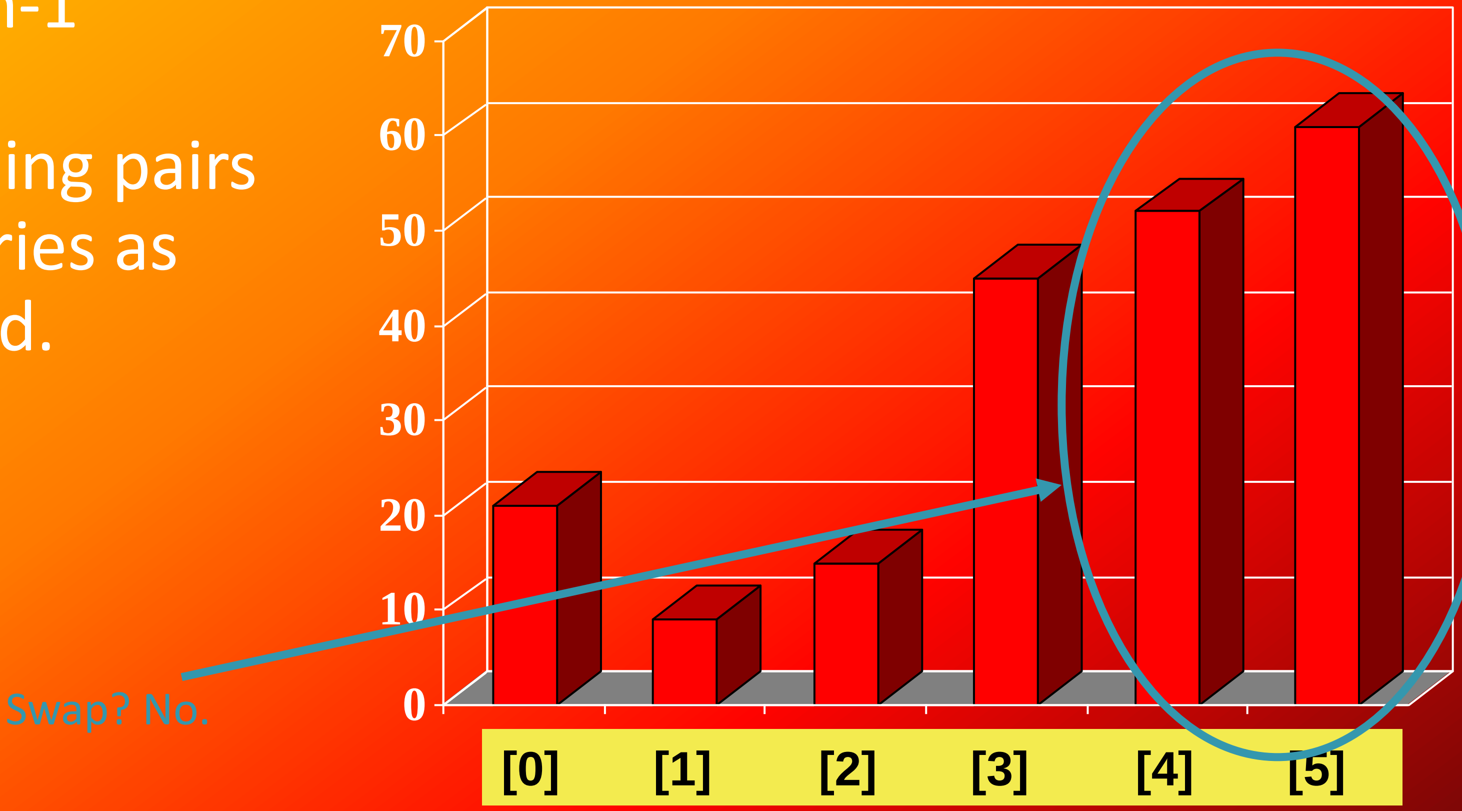
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.



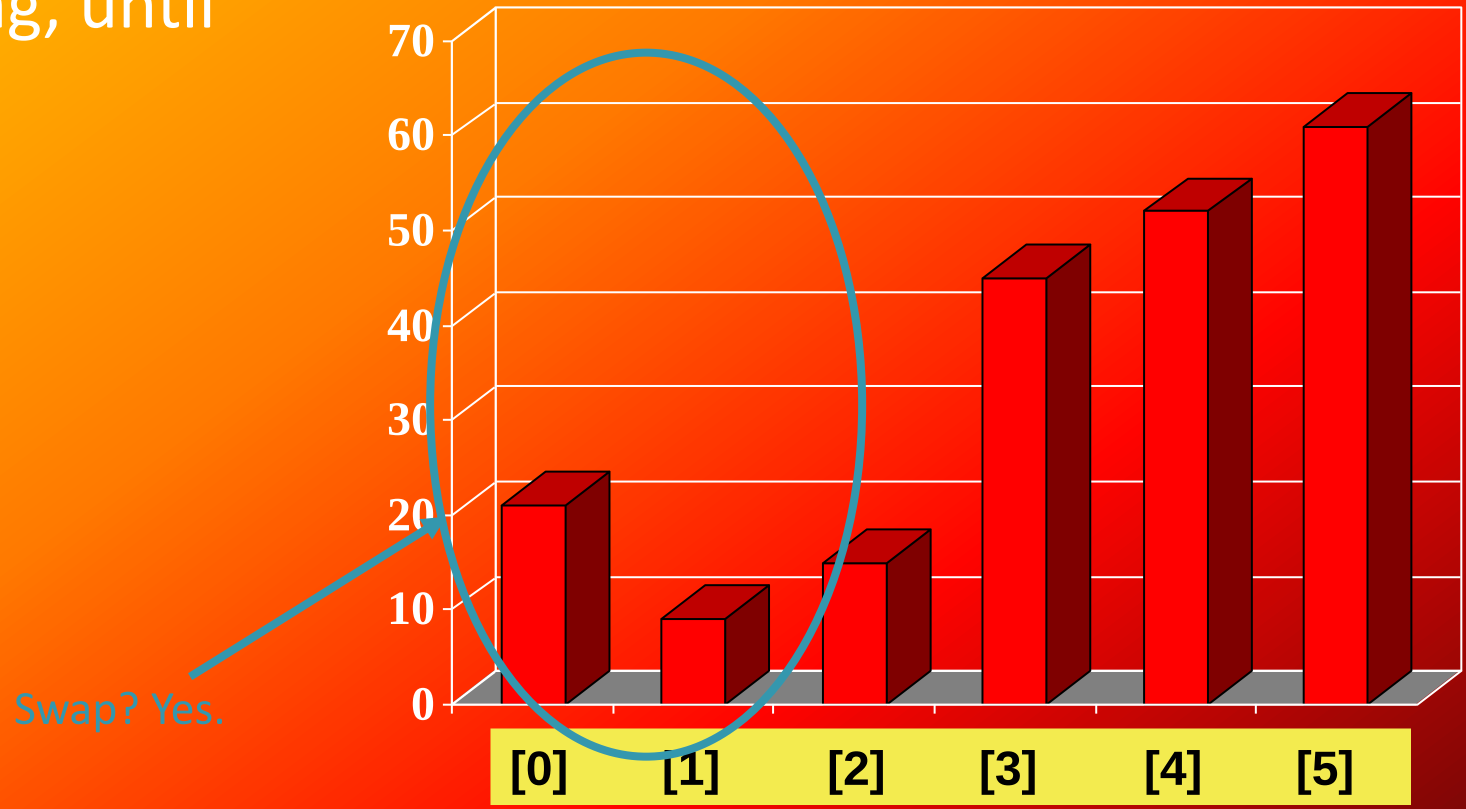
Bubble Sort

- Loop over array n-1 times, swapping pairs of entries as needed.

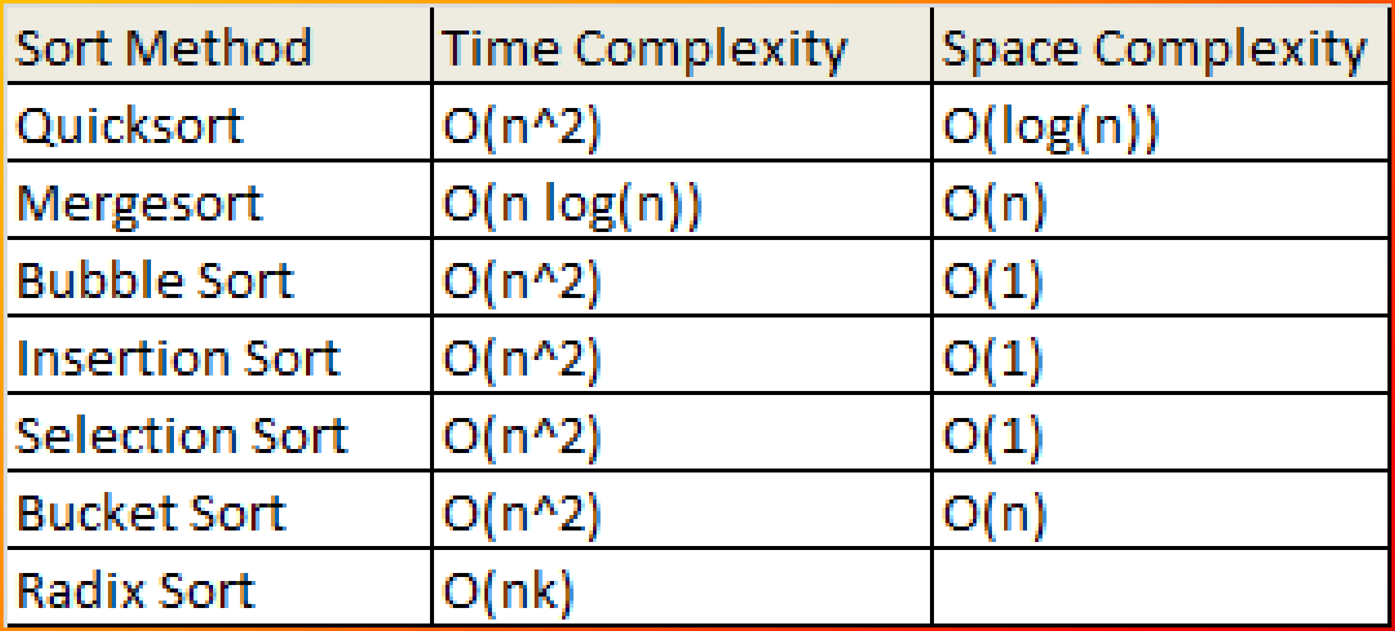


Bubble Sort

- Continue looping, until done.

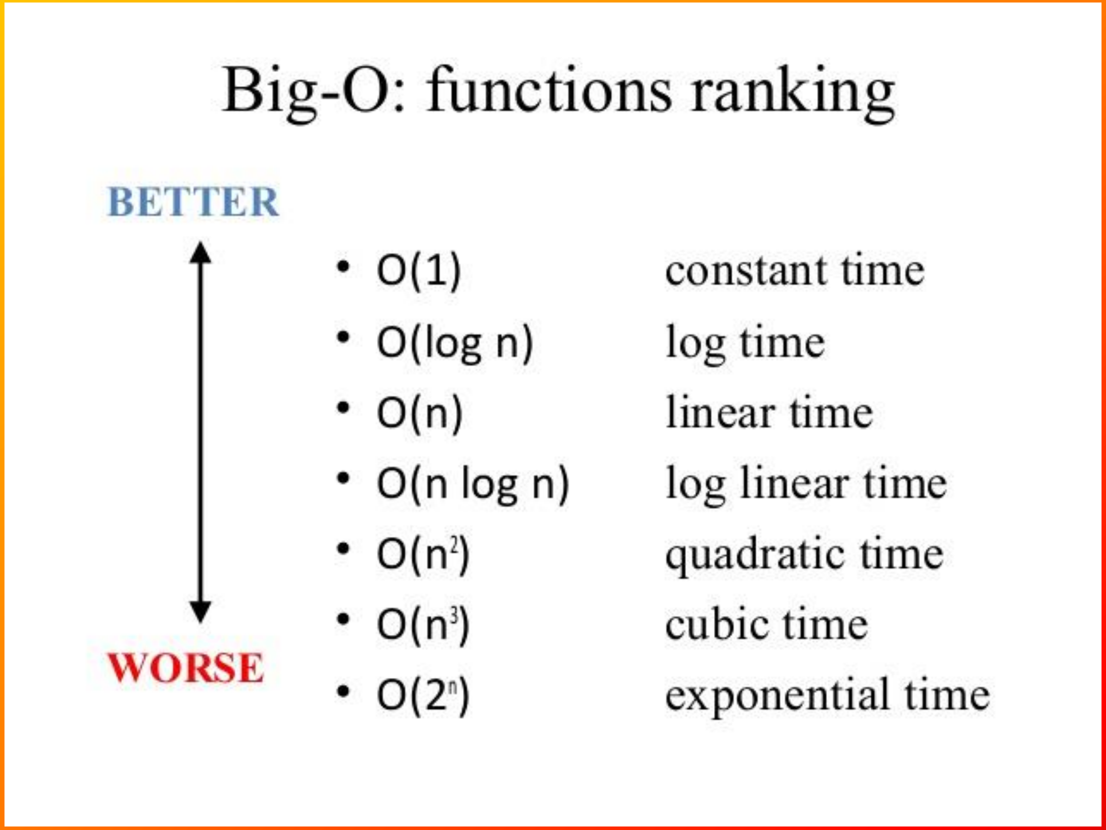


Comparing Sort Algorithms



Sort Method	Time Complexity	Space Complexity
Quicksort	O(n^2)	O(log(n))
Mergesort	O(n log(n))	O(n)
Bubble Sort	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(1)
Selection Sort	O(n^2)	O(1)
Bucket Sort	O(n^2)	O(n)
Radix Sort	O(nk)	

Bear in Mind:



Big-O: functions ranking

BETTER

WORSE

• O(1)	constant time
• O(log n)	log time
• O(n)	linear time
• O(n log n)	log linear time
• $O(n^2)$	quadratic time
• $O(n^3)$	cubic time
• $O(2^n)$	exponential time

Any Questions?