KEY

Big-O Analysis

Big-O Notation in 5 Minutes: https://www.youtube.com/watch?v=__vX2sjlpXU

Big-O is a notation that expresses the efficiency of algorithms, especially how well the algorithm performs as the number of items *n* increases. The expression O(n), or "Big-O of n" means the time the algorithm takes to process all items varies linearly with the number of items. Similarly, $O(n^2)$ means that the time varies with the square of the number of items.

Let's analyze these methods:

`public static double sum(double[] array)`
`{`
`double sum = 0.0;`
`for(int k = 0; k < size(array); k++)`
`sum = sum + get(array, k);`
`return sum;`
`}`
	Because the for-loop in `sum` visits each element, the Big O is **O(*n*)**. Doubling the number of items would double the work.

`public static double get(double[] array, int k)`
`{`
`return array[k];`
`}`
`public static int size(double[] array)`
`{`
`return array.length;`
`}`
	`get` and `size` are both **O(1)** or **constant**. The time it takes to compute the value of the k^{th} item or the size of the data set is independent of the number of items. Any change to the size of the data set would not affect these operations.

`public static double average(double[] array)`
`{`
`return sum(array) / size(array);`
`}`
	`average` calls `sum`, which is O(*n*) and `size`, which is O(1).
	It looks like the Big-O would be O(*n**1). However, since Big-O theory always considers *n* to be as large as possible, the constants, coefficients, and smaller terms are ignored. O(*n**1) becomes just **O(*n*).**

Here is an inefficient implementation called `foo`:

```
public static void foo(double[] array)
 {
    for( int k = 0; k < size(array); k++ )
    if( get(array, k) > average(array) )
       System.out.println( "Hello " + get(array, k) + "." );
 }
```

Since `average` is recalculated each time in the for-loop (i.e, the two for-loops are **nested**), `foo` runs in **$O(n^2)$** or **quadratic** time.

If we rewrite `foo` so that the for-loops are **side-by-side**, we get O(*n*+*n*), which reduces by the Big O rules to O(*n*). Here is the new and improved `foo`:

```
public static void foo(double[] array){
   double avg = average(array);
   for( int k = 0; k < size(array); k++ )
        if( get(array, k) > avg )
         System.out.println( "Hello " + get(array, k) + "." );
 }
```

We can easily get worse efficiency by changing the implementation of helper methods. For instance, suppose we reimplement `get` and `size` so that they use loops:

```
public static double get(double[] array, int k)
{
  int j = 0;
  while(j < k)
     j = j + 1;
  return array[j];
}
```

O(n)

```
public static int size(double[] array)
{
  int j = 0;
  while(j < array.length)
     j = j + 1;
  return j;
}
```

O(n)

Now both `get` and `size` have become linear or O(n). This means that our original `sum` and `average` are now O(n^2), and our original `foo` is now **3 nested loops**, or O(n^3). Ouch!

Here is the AP exam's favorite Big O question, which looks like one for-loop, but has a trick to it. Using the original, O(1) `get` method, what is Big O value for `trickfoo`?

```
    public static void trickfoo(double[] array)
{
  for(int k = 1; k < array.length; k*=2)
     System.out.print(get(array, k) + ", ");
}
```

k *= 2 cuts it in half each time ∴ O(log n)

Most algorithms we deal with in APCS will be either O(1), O(log n), O(n), O(n*log n), or O(n^2), where logs are understood to be taken base two (e.g., log 1024 = 10).

An example of an O(log n) algorithm is the binary search, as we have already seen.

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(a^n)$

Exercises

As the n increases, how do these Big O efficiencies change?

Big O	***n = 1***	***n = 2***	***n = 10***	***n = 100***	***n*** ≈ 1000
1	1	1	1	1	1
log n	0	1	≈ 3.3	≈ 7	≈ 10
n	1	2	10	100	1000
n log n	0	2	≈ 33	≈ 700	≈ 10000
n^2	1	4	100	10,000	1,000,000
2^n	2	4	1024	1.267E30	1.07E301

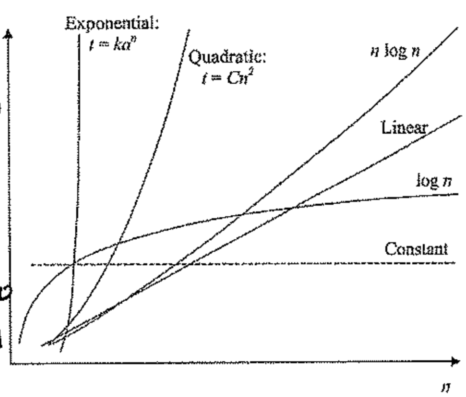

Figure 8-2. Rates of growth

Big-O in a Program

DIRECTIONS: Classify the time efficiency of this program by writing the order of magnitude, in Big-O notation, of each statement in the blanks on the right. Then write the time efficiency in Big-O notation for each method.

```
import javax.swing.JOptionPane;
public class statpkg
{
   public static final int MAX = 10000;                          // O(1)
   public static void main (String[] args)                       //main( ) O(n^3)
   {
      double [] list = new double[MAX];                          // O(1)
      list = readArray(list);                                    // O(n^2)
      System.out.println("Mean = " + mean(list));                // O(n)
      System.out.println("Standard deviation = " + stDev(list)); // O(n^2)
      print(list);                                               // O(n^3)
   }
   public static double[] readArray(double[] list)               //readArray( ) O(n^2)
   {
      int n = 0;                                                 // O(1)
      double height = 0;                                         // O(1)
      height = Double.parseDouble
            (JOptionPane.showInputDialog("Enter height:   -1 to stop")); // O(1)
       while(height > -1)                                        // O(n)
       {
          if ( n >= list.length )                                // O(1)
             list = resize(list, 2);                             // O(n)
          list[n] = height;                                      // O(1)      O(n^2)
          n++;                                                   // O(1)
          height = Double.parseDouble
               (JOptionPane.showInputDialog("Enter height:   -1 to stop")); // O(1)
       }
     list = resize(list, n);                                     // O(n)
     return list;                                                // O(1)
   }
```

```
public static double[] resize (double[] list, int n)            //resize() O(n)
{
  if (n == 2)                                                    //O(1)
     n = 2 * list.length;                                        //O(1)
  double [] newList = new double[n];                             //O(1)
  for (int i = 0; i < Math.min(n, list.length); i++)             //O(n)
       newList[i] = list[i];                                     //O(1)    O(n)
  return newList;                                                //O(1)
}

public static double mean(double[] list)                         //mean( ) O(n)
{
  double sum = 0;                                                //O(1)
  int n  = list.length;                                          //O(1)
  for(int i=0; i<n; i++)                                         //O(n)    O(n)
       sum += list[i];                                           //O(1)
  return (sum / n);                                              //O(1)
}

public static double stDev(double[] list)                        // stDev( ) O(n^2)
{
  double diff, sum = 0;                                          //O(1)
  int n = list.length;                                           //O(1)
  for(int i=0; i<n; i++)                                         //O(n)
  {                                  mean is recalculated
     diff = list[i]-mean(list);      for each i                  //O(n)    O(n^2)
     sum  = sum + diff*diff;                                     //O(1)
  }
  return Math.sqrt(sum / (n - 1));                               //O(1)
}

public static void print(double[] list)                          //print( ) O(n^3)
{
  int n = list.length;                                           //O(1)
  for(int i=0; i<n; i++)                   O(1)                  //O(n)
    System.out.println ("[  " + list[i] + " ]  "                           O(n^3)
          + (list[i] – mean(list)) / stDev(list) );              //O(n^2)
             O(1)      O(n)          O(n^2)
}
}
```

Searches: Linear and Binary

A **linear search** is usually implemented iteratively, meaning that it looks at each word in turn, starting from the beginning of the list.

A **binary search** can be implemented either iteratively or recursively. Each recursive call calculates the middle index and compares that value to the target. If they are equal, return the index and you're done. If the value is less than the target, then search the upper half. Else, search the lower half. Recur until either a) the index finds the target, or b) *first* is greater than *last*. Try it out below. Search the array below for "quiet". Then search the array for "if".

"A"	"The"	"a"	"an"	"and"	"be"	"between"	"car"	"quiet"	"quit"	"the"	"zoo"

↑ *first* ↑ *last*

Big-O and Linear Search

5	2	8	10	2	4	0	2	1	8	5	3	5	9

1. Given an array, what is the Big-O of the best case in the Linear Search?
   O(1) ex: search for 5

2. What is the Big-O of the average case in the Linear Search? $O(n/2) \therefore \rightarrow O(n)$

3. What is the Big-O of the worst case in the Linear Search? O(n)
   ex: search for 100 → not found!

Big-O and Binary Search

4	6	8	10	12	14	20	22	31	38	45

4. Given a sorted array, what is the Big-O of the best case in the Binary Search?
   O(1) ex: search for 14

5. What is the Big-O of the average case in the Binary Search?
   O(log n) ex: search for 10

6. What is the Big-O of the worst case in the Binary Search? ex: search for 100 → not found!
   O(log n)

The Big-O efficiency of the binary search (in the average case) is **O(log n)**. If n were doubled, then the work to search an item is, on average, increased by 1. Thinking about it a different way, the binary search discards half the data at each check. (There is a precondition: the data set must be sorted—assume it is).

7. In the Binary Search, the depth of recursion k is related to n elements by the equation: $n = 2^k$. Solve for k. $k = \log_2 n$

$$n = 2^k$$
$$\log_2 n = \log_2(2^k)$$
$$k = \log_2(n) = \frac{\log_{10}(n)}{\log_{10}(2)} = \frac{\ln(n)}{\ln(2)}$$

8. $\log_2(8) =$ 3 $\log_2(32) =$ 5 $\log_2(100) \approx$ 6.6 $\log_2(260) \approx$ 8.1

9. In general, given n elements, how many comparisons, on average, does the **binary search** require?

$\log_2 n$ (cs: $\log n$)

100 elements	6.6
3000 elements	11.6
1,000,000 elements	19.93

10. Calculate the average case Big-O values for binary searches ~~of~~

$O(\log n)$

11. In practical, not theoretical, terms, which is faster for small sets, linear search or binary search?

linear

recursion needs extra memory + method calls, which is expensive for small sets

12. In practical, not theoretical, terms, which is faster for large sets, linear search or binary search?

binary

$O(\log n) < O(n)$ if n is "infinitely large"

Selection Sort

Go to http://math.hws.edu/eck/js/sorting/xSortLab.html to watch the sorts in action.

The goal in sorting is to put a list of items in order. "To put in order" means to be able to compare any two items and determine before-and-after, or less-than-greater-than. For primitive types we just use the built-in less-than operator (<). For objects we either use `compareTo()`, if the class *implements the Comparable interface*, or `compare()` if the class *implements the Comparator interface*. If none of these apply, then the objects don't have any order, and it makes no sense to sort the list.

Selection Sort

Find the largest item in the list and swap it to the end. That item is now in its correct position and is no longer considered. Now look at the sublist containing the first N–1 items. On that sublist, find the largest item and swap it to the end (the next to last position of the overall list). That item is now in its correct position and is no longer considered. Then move to the sublist containing the first N–2 items, and repeat. A list with N items needs N-1 passes.

begin	**3**	**1**	**4**	**1**	**5**	**9**	**2**	**6**
pass 1	3	1	4	1	5	6	2	9
pass 2	3	1	4	1	5	2	6	9
pass 3	3	1	4	1	2	5	6	9
pass 4	3	1	2	1	4	5	6	9
pass 5	1	1	2	3	4	5	6	9
pass 6	1	1	2	3	4	5	6	9
pass 7	1	1	2	3	4	5	6	9

Exercises

1. Suppose you are finding the largest and swapping to the end. What does this array look like after 3 passes?

4	3	7	3	1	8	5	6

4	3	5	3	1	6	7	8

2. Suppose you are finding the smallest and swapping to the front. What does this array look like after 3 passes?

4	3	7	3	1	8	5	6

1	3	3	7	4	8	5	6

3. Let's think about the Big-O of the Selection Sort in its best case, average case, and worst case. We will focus on the number of comparisons made by the algorithm. Here is an array with 6 items in random order:

3	6	2	7	2	1

5 + 4 + 3 + 2 + 1 = 15 $\frac{n(n-1)}{2} = \frac{6 \cdot 5}{2} = 15$

How many comparisons are made to sort this array in ascending order? 15

4. Here is a 6-item array already in sorted order (ascending order):

1	2	3	4	5	6

How many comparisons are made to sort this array in ascending order? 15

5. Here is a 6-item array in reverse order (descending order):

6	5	4	3	2	1

How many comparisons are made to sort the array in ascending order? 15

6. What can you conclude from above? Order of data does not affect efficiency of Selection Sort

7. Know thy Big-O: the Selection Sort is $O(n^2)$ in the best case, $O(n^2)$ in the average case, and $O(n^2)$ in the worst case.

Insertion Sort

Go to http://math.hws.edu/eck/js/sorting/xSortLab.html to watch the sorts in action.

The Insertion Sort is an entirely different algorithm for sorting. It is not just a different way to code the Selection Sort! Think of picking up a hand of cards, one at a time. The first card is automatically in order. Pick up the second card. **Insert** it in order with the first card, **sliding** over if necessary. Pick up the third card, and repeat. At each step we produce a sorted hand of cards, then get the next item, until we have inserted each item in its proper place.

The first item of data is automatically a sorted sublist of length 1. Now look at the second item in the list. If the second item is smaller then we **slide** the first item over, and put the second item in its place. (Be careful! As the length of the sublists grow, swapping items is not part of the general solution.) Now look at the third item. If the third item is larger than the second then it must be larger than the first also, and the third item is in its correct location. On the other hand, if it's smaller than the second then we move the second over, and compare it to the first. It might now be in the correct location, but if it is smaller than the first, we move the first over, and so on. Eventually we insert the third item in its correct location. And so on. A list with N items needs N-1 passes.

begin	3	1	4	1	5	9	2	6
pass 1	1	3	4	1	5	9	2	6
pass 2	1	3	4	1	5	9	2	6
pass 3	1	1	3	4	5	9	2	6
pass 4	1	1	3	4	5	9	2	6
pass 5	1	1	3	4	5	9	2	6
pass 6	1	1	2	3	4	5	9	6
pass 7	1	1	2	3	4	5	6	9

Exercises

1. Using the Insertion Sort algorithm, what does this array look like after 3 passes?

4	3	7	1	3	8	5	6

1	3	4	7	3	8	5	6

2. Let's think about the Big-O of the Insertion Sort in its best case, average case, and worst case. We will focus on the number of comparisons made by the algorithm. Here is a 6-item array in random order:

3	2	1	4	6	5

1 + 2 + 1 + 1 + 2 = 7

How many comparisons are made to sort this array in ascending order? 7

3.Here is a 6-item array already in sorted order (ascending order):

1	2	3	4	5	6

How many comparisons are made to sort this array in ascending order? 5

4. Here is a 6-item array in reverse order (descending order):

6	5	4	3	2	1

1 + 2 + 3 + 4 + 5

How many comparisons are made to sort the array in ascending order? 15

5. Conclusion: best case for insertion sort : already or almost sorted

6. Know thy Big-O: the Insertion Sort is $O(n)$ in the best case, $O(n^2)$ in the average case, and $O(n^2)$ in the worst case. * uses 2 nested loops *

Merge Sort

Go to http://math.hws.edu/eck/js/sorting/xSortLab.html to watch the sorts in action.

The Merge sort is the first of our O(n log n) sorts. It creates smaller and smaller sublists by recurring on successive halves of the array, then zipper-merging each successive pair of sublists. The last zipper-merge produces the sorted list.

```
private static void mergeSortHelper(double[] array,
                          double[]copyBuffer, int low, int high) {
   if (low < high) {
      int middle = (low + high) / 2;
      mergeSortHelper(array, copyBuffer, low, middle);
      mergeSortHelper(array, copyBuffer, middle + 1, high);
      merge(array, copyBuffer, low, middle, high);
   }
}
```

Since `low` and `high` are moving towards each other, the sublists are successively cut in half, until we reach sublists of length one. By definition, sublists of length one are already sorted. As the recursion unwinds, the zipper-merge methods on each level merge and order the items of successively larger pairs of sublists. The zipper-merge method does the actual work of ordering the items in the array.

Trace the Merge Sort

the method calls	the data after the call
mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() mergeSortHelper() etc.	[3,1,4,1,5,9,2,6] 3, 1, 4, 1 5, 9, 2, 6 3, 1 4, 1 5, 9 2, 6 5 9 2 6 3 1 4 1 · base case: length == 1 · after split, then merge
merge [3, 1]	[1,3,4,1,5,9,2,6]
merge [4, 1]	[1,3,1,4,5,9,2,6]
merge [1, 3, 1, 4]	[1,1,3,4,5,9,2,6]
merge [5, 9]	[1,1,3,4,5,9,2,6]
merge [2, 6]	[1,1,3,4,5,9,2,6]
merge [5, 9, 2, 6]	[1,1,3,4,2,5,6,9]
merge [1, 1, 3, 4, 2, 5, 6, 9]	[1,1,2,3,4,5,6,9]

Important! Merging two lists **must** be done using a single pass. That's why it is called a "zipper merge." Do **not** use nested loops! The `merge` method should use one loop, either `for` or `while`, running between `low` and `high` inclusive. Since you know the sub-lists are already sorted, you need **one loop** to compare values at index `i1` and `i2` and copy the smaller value into the index in the `copybuffer`. The index of the smaller item is then updated. It can be tricky! There are 4 cases to consider: `array[i1]<array[i2]`, `array[i2]<array[i1]`, `i1>mid`, `i2>high`. Use this diagram to practice the `merge` method algorithm:

`array`									1	1	3	4	2	5	6	9
									`low` `mid`				`mid+1` `high`			
									`i1`				`i2`			
`copybuffer`									1	1	2	3	4	5	6	9
`index`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The algorithm diagrammed above uses a temporary array `copybuffer`, which each time has to be copied (using a second, but not a nested loop) from `low` to `high` back to `array`.

4	5	8	2
4·	5·	8	2
4	5	2·	8·
2	4	5	8

Exercises

1. Suppose all the recursive calls of the Merge Sort have been completed on this array. None of the zipper-merge methods have run. What does the array look like after **one, two,** and **three** of the zipper-merge methods have run?

4	5	3	2	9	7	5	1
4	5	3	2	9	7	5	1
4	5	2	3	9	7	5	1
2	3	4	5	9	7	5	1
2	3	4	5	7	9	5	1

2. In the Merge Sort, 8 items of data takes 3 levels of recursion. For *n* items of data (in random order), how many levels of recursion do you need? $\log_2 n$ What is the Big-O of each zipper-merge? $O(n)$ Therefore, what is the Big-O of the Merge Sort (in the average case)? $O(n \log n)$

3. Here is an array already in sorted order (ascending order):

7 merge methods

1	2	3	4	5	6	7	8

$(1+1+1+1)+(2+2)+4$

How many comparisons does the Merge Sort make to sort this array in ascending order? 12

4. Here is an array in reverse order (descending order):

8	7	6	5	4	3	2	1

$(1+1+1+1)+(2+2)+4$

How many comparisons does the Merge Sort make to sort this array in ascending order? 12

5. Know thy Big-O: the Merge Sort is $O(n \log n)$ in the best case, $O(n \log n)$ in the average case, and $O(n \log n)$ in the worst case. ∴ The mergesort is stable

QuickSort

Go to http://math.hws.edu/eck/js/sorting/xSortLab.html to watch the sorts in action.

Quick Sort is another O(n log n) sort. Its general strategy is to select a *pivot* (or a *partition*) and move every smaller item to the left of the pivot and every larger item to its right. Then we call Quick Sort on the left side and the right side, recurring until we are done.

```
private static void sort(double[] array, int first, int last) {
    int splitPt;
    if (first < last) {            // General case
      splitPt = rearrange(array, first, last);
      sort(array, first, splitPt - 1);   // sort left side
      sort(array, splitPt + 1, last);    // sort right side
    }
}
```

Normally the code generates a tree structure, with the `rearrange` method at each branch running in linear Or O(n) time. That general arrangement should remind you of the MergeSort.

1) Let's run **one** pass of the `rearrange` method on the toy array below. For this teaching example, we will let the pivot be 0 and already in the middle, as shown. We want to rearrange the array so that all negative numbers appear to the left of 0 and the positive numbers appear to the right of 0. We keep track of two indexes, one starting at the far left and the other at the far right. Compare the left value to 0. If smaller, move over. Compare the right value to 0. If larger, move over. Keep comparing and moving until you find two values that are each on the "wrong side." Then swap that pair. Repeat. Show the contents of the array after one pass:

-1	2	1	-4	0	-6	3	-2

	1	3			2		1
-1	-2	-6	-4	0	1	3	2

Look at the result of one pass of the `rearrange` method. All negatives are to the left and all positives are to the right. The 0 is in its correct place, and never has to be compared or moved again. Fill in the blanks in this code, which is part of the `rearrange` method.

```
while (first <= last)
 {
   if (array[first] <= pivot)            //if it's on the correct side,
       first ++;                          //   move right
   else if (array[last] >= pivot)        //if it's on the correct side,
       last --;                           //   move left
   else                                   //if both on the wrong side,
   {
       swap(array, first, last);          //   then swap them,
       first ++;                          //   update both right and left
       last --;
   }
 }
```

2) Let's go on to the full Quick Sort, in which the pivot is not 0, but changes with every recursive call. Watch this http://www.cs.armstrong.edu/liang/animation/web/QuickSortNew.html We simply choose the new pivot as the *first value* in the array. We look for pairs of values that each are "on the wrong side" and swap them. In the example below, the pivot is 9. You have to imagine that the 9 is going to be in the middle somewhere, and all values that are smaller than 9 will be on the left and all values that are greater than 9 will be on the right. After all the "on the wrong side" pairs have been swapped, and the indices meet each other, one last swap puts the pivot 9 at that index-meeting place.

9	20	3	5	60	6	14	11
first = pivot	↑						↑

5 2	6 1	3	9 2	60	20 1	14	11

3) Then recur on the left side of the array and recur on the right side of the array, running the `rearrange` method on each side:

5	6	3	9	60	20	14	11
3	5 3	6 6	9	11	20	14	60
	↑	↑			↑		↑

4) Assuming that the `rearrange` method uses the *first item* as the pivot, display the contents of array `Arr` after the call `rearrange(Arr, first, last);`

12	18	8	4	11	7	6	3	10	1	5	20

1 2	5 1	8	4	11	7	6	3	10	12 2	18 1	20

`Arr`

5) The pivot does not have to be the first. The pivot could be `(first+last)/2`. If that is the case, show one pass of the `rearrange` method on this data:

5	12	3	7*	8	5	2
5	2	3	5	7	8	12

6) We can reason intuitively about the Big-O for Quick Sort. If we have large arrays of random data, each pivot (the first item) will probably cut the array near the middle. (You have seen that before in the Binary Search.) Therefore, for *n* items of data (in random order), how many levels of recursion do you need? $\log_2 n$ What is the Big-O of each `rearrange` method (see above)? $O(n)$
Therefore, what is the Big-O of the Quick Sort for random data? $O(n\log n)$ · same as MergeSort · slightly faster on large, random data sets

7) The Quick Sort has a worst case in which the Big-O degenerates to $O(n^2)$. Here is an array already in sorted order (ascending order).

1	2	3	4	5	6	7	8

How many levels (recursive calls) does the Quick Sort make to sort this array in ascending order? 7
How many comparisons does it do in each level and in total? 35
What is the Big-O in this case? $O(n^2)$ $8+7+6+5+4+3+2$

8) For similar reasons, the Big-O for an array in reverse order (descending order) is also $O(n^2)$.

8	7	6	5	4	3	2	1

How many levels (recursive calls) does the Quick Sort make to sort this array in ascending order? 8
How many comparisons does it do in each level and in total? $8*8=64$
What is the Big-O in this case? $O(n^2)$
Indeed, the QuickSort in this case behaves just like a Selection Sort. Try it and see.

9) Know thy Big-O: the Quick Sort is $O(n\log n)$ in the best case, $O(n\log n)$ in the average case, and $O(n^2)$ in the worst case, i.e., the case of the "bad pivot."

- QuickSort is "unstable"
- MergeSort is "stable"

if n is even:

$$(n-1)+(n-1)+(n-3)+(n-3)+\cdots+3+3+1 = 2*n/2*n/2-1$$
$$= n^2/2 - 1 \quad \therefore O(n^2)$$

if n is odd:

$$(n-1)+(n-1)+(n-3)+(n-3)+\cdots+2+2 = 2*\left(\frac{n+1}{2}\right)*\left(\frac{n-1}{2}\right) \therefore O(n^2)$$

KNOW THY Big-O

Fill in the Big-O notation for the two search and four sorting algorithms covered to date. Keep in mind:

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$$

Type of Sort/Search	Best Case, what condition?	Average Case, what condition?	Worst Case, what condition?
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n)$ - already sorted	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$ - already sorted w/ "bad" pivot
Linear Search	$O(1)$ - first element	$O(n)$	$O(n)$
Binary Search array must be sorted	$O(1)$ - middle element	$O(\log n)$	$O(\log n)$

Big-O Worksheet 1 *Name*________________________

Instructions: The following algorithms will display a different number of stars, depending on the value in `nNum`. Classify the Big-O growth rate, and justify your answer with a calculation.

notation	*name*	*effect of doubling data*
O(1)	constant	no effect
O(log n)	logarithmic	increases work by 1
O(n)	linear	doubles work
O(n · log n)	sometimes called "linearithmic" or "supralinear"	2x < work < 4x
$O(n^2)$	quadratic	quadruples work
$O(2^n)$	exponential	if data increases by 1,work doubles
O(n!)	factorial	if data increases by 1, work > 2x

1. How many stars would be displayed by the following algorithm? 100
   How many stars would be displayed if `nNum` were doubled? 400

```
int nNum = 10;
for(int nOuter = 0; nOuter < nNum; nOuter++)
   for(int nInner = 0; nInner < nNum; nInner++)
   System.out.print("*");
```

Big-O notation: $O(n^2)$ Justify your answer: 2 nested loops

2. How many stars would be displayed by the following algorithm? 10
   How many stars would be displayed if `nNum` were doubled? 20

```
int nNum = 10;
for(int nOuter = 0; nOuter < nNum; nOuter++)
  System.out.print("*");
```

Big-O notation: O(n) Justify your answer: one loop

3. How many stars would be displayed by the following algorithm? 4
   How many starts would be displayed if `nNum` were doubled? 5

```
int nNum = 10;
for(int nOuter = 1; nOuter <= nNum; nOuter*=2)
  System.out.print("*");
```

Big-O notation: O(log n) Justify your answer: one loop w/ counter doubled each time

4. How many stars would be displayed by the following algorithm? 1
   How many stars would be displayed if `nNum` was doubled? 1

```
int nNum = 10;
char[] caStars = new char[100];
for(int nI = 0; nI < 100; nI++)
caStars[nI] = '*';
System.out.print("" + caStars[nNum]);
```

Big-O notation: O(1) Justify your answer: The loop doesn't depend on nNum. The for-loop runs 100 times every time. Just one access into the array.

5. How many stars would be displayed by the following algorithm? $2^5 = 32$
How many stars would be displayed if nNum were increased by 1? $2^6 = 64$

```
int nNum = 5;
int nStars = 1;
while(nNum > 0)
{
   nStars*=2;
   nNum--;
}
for(int nI = 0; nI < nStars; nI++)
System.out.print("*");
```

Big-O notation: $O(2^n)$ Justify your answer: exponential growth

6. How many stars would be displayed by the following algorithm? 40
How many stars would be displayed if nNum were doubled? 100

```
int nNum = 10;
for(int nOuter = 1; nOuter <= nNum; nOuter++)
   for(int nInner = 1; nInner <= nNum; nInner*=2)
      System.out.print("*");
```

Big-O notation: $O(n \log n)$ Justify your answer: 2 nested for-loops linear * logarithmic

7. How many stars would be displayed by the following algorithm? 10
How many stars would be displayed if nNum were doubled? 20

```
public static void main(String[] args) {
  int nNum = 10;
  printStar(nNum);
}
public static void printStar(int nNum){
   if(nNum > 0){
      System.out.print('*');
      printStar(nNum - 1);   ← recursion
   }
}
```

Big-O notation: $O(n)$ Justify your answer: Simple counting

8. How many stars would be displayed by the following algorithm? 4
How many stars would be displayed if nNum were doubled? 5

```
public static void main(String[] args) {
  int nNum = 10;
  printStar(nNum);
}
public static void printStar(int nNum){
   if(nNum > 0){
      System.out.print('*');
      printStar(nNum/2);
   }
}
```

Big-O notation: $O(\log n)$ Justify your answer: cut in half each time

Big-O Worksheet 2 *Name*______________________

Instructions: The following algorithms will display a different number of stars, depending on the value in nNum. Classify the Big-O growth rate, and justify your answer with a calculation.

notation	*name*	*effect of doubling data*
O(1)	constant	no effect
O(log n)	logarithmic	increases work by 1
O(n)	linear	doubles work
O(n · log n)	sometimes called "linearithmic" or "supralinear"	2x < work < 4x
$O(n^2)$	quadratic	quadruples work
$O(2^n)$	exponential	if data increases by 1,work doubles
O(n!)	factorial	if data increases by 1, work > 2x

1. **How many stars would be displayed by the following algorithm?** 1000
   How many stars would be displayed if nNum was doubled? 8000

```
int nNum = 10;
for(int nA = 0; nA < nNum; nA++)
    for(int nB = 0; nB < nNum; nB++)
        for(int nC = 0; nC < nNum; nC++)
            System.out.print("*");
```

Big-O notation: $O(n^3)$ **Justify your answer:** 3 nested for-loops

2. **How many stars would be displayed by the following algorithm?** 1000
   How many stars would be displayed if nNum was doubled? 4000

```
int nNum = 10;
for(int nA = 0; nA < nNum; nA++)
    for(int nB = 0; nB < nNum; nB++)
        for(int nC = 0; nC < 10; nC++)
            System.out.print("*");
```

Big-O notation: $O(n^2)$ **Justify your answer:** the last loop does not depend on nNum
↳ $O(10n^2) \to O(n^2)$

3. **How many stars would be displayed by the following algorithm?** 1000
   How many stars would be displayed if nNum was doubled? 2000

```
int nNum = 10;
for(int nA = 0; nA < nNum; nA++)
    for(int nB = 0; nB < 10; nB++)
        for(int nC = 0; nC < 10; nC++)
            System.out.print("*");
```

Big-O notation: $O(n)$ **Justify your answer:** only one loop depends on n

4. **How many stars would be displayed by the following algorithm?** 25
   How many stars would be displayed if nNum was doubled? 100

```
int nNum = 10;
for(int nA = 0; nA < nNum/2; nA++)
    for(int nB = 0; nB < nNum/2; nB++)
        System.out.print("*");
```

Big-O notation: $O(n^2)$ **Justify your answer:** two nested loops

5. **How many stars would be displayed by the following algorithm?** 6
How many stars would be displayed if nNum was increased by 1? 24

```
public static void main(String[] args) {
  int nNum = 3;
           int nLimit = mystery(nNum);
  for(int nA = 0; nA < nLimit; nA++)
       System.out.print("*");
}
public static int mystery(int nNum)
{
  if(nNum == 0)
     return 1;
  else
     return nNum * mystery(nNum-1);
}
```

Big-O notation: $O(n!)$ **Justify your answer:** recursive part: $O(n)$
loop $O(n!)$
↳ $O(n+n!) \rightarrow O(n!)$

6. How many stars would be displayed in the *Worst Case* (the longest run-time) by the following algorithm? $6*5*4=120$
How many stars would be displayed in the *Worst Case* if nNum was doubled? $6*15*14$
$= 3360$

```
static int rand = (int)(Math.random() *10);
public static void main(String[] args) {
  if(rand < 5)
  {
     int nNum = 10;
     for(int nA = 0; nA < nNum-4; nA++)
        for(int nB = 0; nB < nNum-5; nB++)
           for(int nC = 0; nC < nNum-6; nC++)
                        System.out.print("*");
  }
  else
  {
     System.out.println("*");
  }
}
```

Big-O notation: $O(n^3)$ **Justify your answer:** 3 nested loops

7. How many stars would be displayed by #6 in the *Best Case* (the shortest run-time)? 1
How many stars would be displayed by #6 in the *Best Case* if nNum was doubled? 1

Big-O notation: $O(1)$ **Justify your answer:** nNum irrelevant

Exercises **BIG-OH 1** Name______________

Find the Big-Oh efficiency of the following: Period________

1)
```
public static void loops1(int n)
{
        for (int a=0; a<n; a++)
          for (int b=0; b<n; b++)
            for(int c=0; c<n; c++)
              System.out.println("**" + n + "**");
}
```
$O(n^3)$

2)
```
public static void loops2(int n)
{
        for (int a=0; a<n; a++)
          for (int b=0; b<n; b++)
            for(int c=0; c<5; c++)
              System.out.println("**" + n + "**");
}
```
$O(n^2)$

3)
```
public static void loops3(int n)
{
        for (int a=0; a<n; a++)
          for (int b=0; b<7; b++)
            for(int c=0; c<5; c++)
              System.out.println("**" + n + "**");
}
```
$O(n)$

4)
```
public static void loops4(int n)
{
        for (int a=0; a<2; a++)
          for (int b=0; b<7; b++)
            for(int c=0; c<5; c++)
              System.out.println("**" + n + "**");
}
```
$O(1)$

5)
```
public static void loops5(int n)
{
        for (int a=0; a<n; a++)
          System.out.println("**" + n + "**");   O(n)
        for (int b=0; b<n; b++)
          System.out.println("$$" + n + "$$");   O(n)
        for(int c=0; c<n; c++)
          System.out.println("~~" + n + "~~");   O(n)
}
```
$O(n)$

$(O(3n) \Rightarrow O(n))$

6)
```
public static void loops6(int n)
{
        for (int a=0; a<n; a++)
          System.out.println("**" + n + "**");
        for (int b=0; b<n; b++)
          for(int c=0; c<n; c++)
            System.out.println("$$" + n + "$$");
}
```
$O(n)$

$O(n^2)$

$O(n^2)$

7)
```
public static void loops7(int n)
{
        for (int a=0; a<n; a++)
          for (int b=0; b<a; b++)
            for(int c=0; c<5; c++)
              System.out.println("**" + n + "**");
}
```
$O(n^2)$

8)
```
public static void loops8(int n)
{
        for(int c=0; c<5; c++)
        {
            int i=0;
            while(i<c)
            {
                System.out.println("**" + n + "**");
                i++;
            }
        }
}
```
$O(1)$

9)
```
public static void loops9(int []list)
{
        for (int a=0; a<10; a++)
          for (int b=0; b< list.length; b++)
            System.out.println(list[b]*a);
}
```
$O(n)$

10)
```
public static void loops10(int [][]list)
{
        for (int a=0; a<list.length; a++)
          for (int b=0; b<list[0].length; b++)
            for(int c=0; c<20; c++)
              System.out.println(list[a][b] * c);
}
```
$O(n^2)$

Exercises **BIG-OH 2** Name________________________

Find the Big-Oh efficiency of the following: Period________

1) int size=list.length;

```
...
for(int j=0; j<size; j++)
        System.out.print(list[j] + " ");
```

O(n)

2) char[][] board=new char[size][size];

```
...
for(int r=0; r<size; r++)
{
        for(int c=0; c<size; c++)
                System.out.print(board[r][c]);
        System.out.println();
}
```

$O(n^2)$

3) $F(n) = 3n^3 + 200n^2 + 1024n$ (consider which part grows the fastest) $O(n^3)$

4) if (list.length > 0)

```
{
        for(int j=0; j<list.length; j++)
                System.out.print(list[j] + " ");
        System.out.print("There are " + list.length + " people in the list. ");
}
```

O(n)

5) $F(n) = (n^2 + 2n + 1) / (n + 1)$ //be careful O(n)

6) $F(n) = 3n^2 + 200n*\log_2 n^2 + 50n*\log_2 n + 342n^4 + 300n$ $O(n^4)$

7) int ran = (int)(Math.random()*100);
System.out.println(ran + " is your lucky number."); O(1)

8) String[] names = new String[size]; //be careful

```
...
for(int r = 0; r<10; r++)
        for(int c = 0; c< size; c++)
                System.out.println(names[s]);
```

O(n)

9) System.out.println("Hello World"); O(1)

10) $F(n) = 1{,}000{,}000n + n^{1{,}000{,}000} + 2^n$ $O(2^n)$

11)

```java
public static int recurA(int n)
{
        if (n == 1)
                return 1;
        return n + recurA(n – 1);
}
```

$O(n)$

12)

```java
public static int recurB(int n)
{
        if (n == 1)
                return 1;
        return n + recurB(n / 2);
}
```

$O(\log n)$

13)

```java
public static void doStuff(int n)
{
        for(int i=1; i<=n; i++)
                System.out.println(recurB(n));
}                                   //as defined above
```

$O(\log n)$ $O(n \log n)$

14)

```java
public static void repeatMore(int n)
{
        for(int i=n; i>0; i--)
                doStuff(n);             //as defined above
}
```

$O(n^2 \log n)$

15)

```java
public static int recurC(int n)
{
        if (n == 1)
                return 1;
        return n + recurC(n – 1) + recurC(n - 2);
}
```

$O(2^n)$

16)

```java
public static int sumRecurC(int n)
{       int sum=0;
        for(int i=100; i <= n+100; i++)
                sum += recurC(n);
        return sum;
}
```

$O(n)$ $O(n 2^n)$

$O(2^n)$

Effect of Doubling the Size on Running Time

name	*Big-O notation*	*if the data is doubled, then the work is:*	*if n = 500 and takes 3 ms, then n=1000 will take __ms*
constant	O(1)	unchanged	3
log n	O(log N)	increased by 1	$\frac{\log(500)}{\log(1000)} = \frac{3}{3.335*}$
linear	O(N)	doubled	$\frac{500}{1000} = \frac{3}{6*}$
n log n linearithmic	O(N log N)	2x < work < 4x	$\frac{500*\log(50)}{1000*\log(1000)} = \frac{3}{6.8*}$
	$O(N^{3/2})$	increased by a factor of $2\sqrt{2}$	8.48
quadratic	$O(N^2)$	increased by a factor of 2^2	12
	$O(N^3)$	increased by a factor of 2^3	24
exponential	$O(2^N)$	increased by a factor of 2^N	$\frac{3*2^{1000}}{2^{500}} = 3*2^{500}$

2. Give the Big-Oh of:

a. Selection Sort on random data n^2. On sorted data n^2

b. Insertion Sort, random data n^2. On sorted data n. Backwards data n^2

c. multiplying two NxN matrices n^3 (3 nested for loops)

d. finding Fibonacci numbers iteratively n. By recursion 2^n

e. generating all permutations of n symbols $n(n-1)(n-2)\cdots(1) = n!$

f. merging two sorted lists n

g. Merge Sort $n \log n$

h. Quick Sort, average case $n \log n$. Quick Sort, worst case (bad pivot) n^2

i. finding a max or a min among n unsorted elements (linear search) n

j. Binary Search on a sorted list of n elements $\log n$

k. finding the median value in a sorted array $O(1)$ - constant

l. returning the k^{th} element in a sorted array $O(1)$

m. printing an array of length n n

Multiple Choice Questions on Big-O

1. An efficient algorithm that must delete the last two elements in a long list of n elements stored in an array is

   (A) $O(n)$
   (B) $O(n^2)$
   (C) $O(1)$
   (D) $O(2)$
   (E) $O(\log n)$

2. An algorithm to remove all negative values from a list of n integers sequentially examines each element in the array. When a negative value is found, each element is moved down one position in the list. The algorithm is

   (A) $O(1)$
   (B) $O(\log n)$
   (C) $O(n)$
   (D) $O(n^2)$
   (E) $O(n^3)$

   always assume worst case
   unless told otherwise

3. A certain algorithm is $O(\log n)$. Which of the following will be closest to the number of computer operations required if the algorithm manipulates 1000 elements?

   (A) 10
   (B) 100
   (C) 1000
   (D) 10^6
   (E) 10^9

4. A certain algorithm examines a list of n random integers and outputs the number of times the value 5 appears in the list. Using Big O notation, this algorithm is

   (A) $O(1)$
   (B) $O(5)$
   (C) $O(n)$
   (D) $O(n^2)$
   (E) $O(\log n)$