I. Algorithms

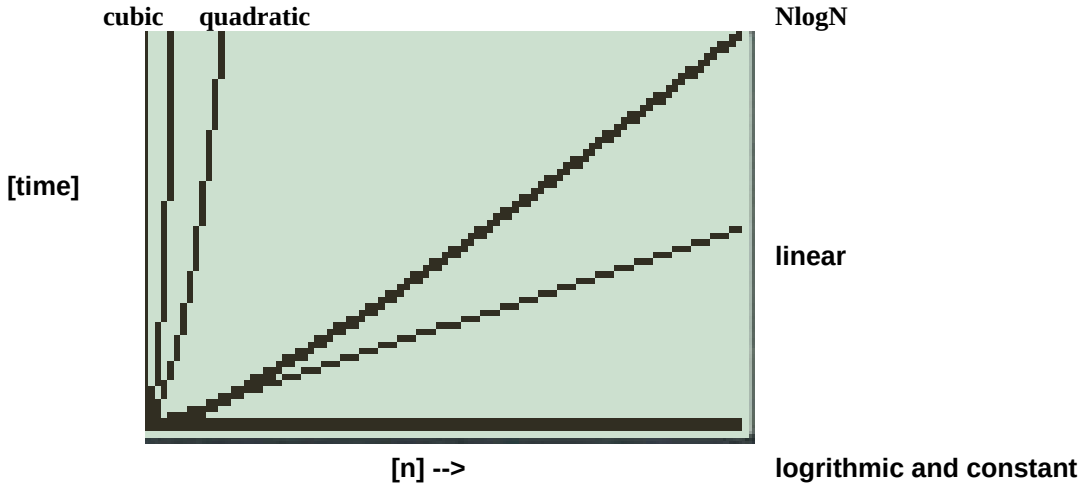
A. Intro to Analysis - Big-Oh

Basically, algorithm analysis is the amount of time any program or algorithm should be expected to take for any given size of input. It is based on the number of elements, n, that must be "processed". For example, in order to Bubble sort an array of 10 elements, the algorithm (bubble sort) must "scan through" all 10 elements and put the largest at the end. This takes at least 10 "calculations". This process must be repeated for 10 times (bubbling up each biggest number). Therefore there is a at least a total of 10 * 10 = 100 "calculations" to be done. If the array is 100 elements, it would take 100*100 = 10000 calculations. If there are n elements, it would take n^2 calculations. This is "Big-Oh" of the Bubble Sort algorithm, written $O(n) = n^2$.

Common Big-Oh functions for various algorithms are:

a) $O(n) = c$, $O(1)$	constant	(Algorithm: finding the nth element of an array takes only 1 calculation)
b) $O(n) = \log(n)$	logarithmic	(Algorithm: searching using the binary search)
c) $O(n) = n$	linear	(Algorithm: searching through each element of an array)
d) $O(n) = n * \log(n)$	N log N	(Algorithm: sorting using quicksort)
e) $O(n) = n^2$	quadratic	(Algorithm: Bubble sort)
f) $O(n) = n^3$	cubic	
g) $O(n) = 2^n$	exponential	(worst = very slow)

examples of cubic, quadratic, NlogN, linear, logarithmic, and constant



1. Rules for calculating Big-Oh

a. Coefficients and constants are not needed.
   a. $O(n) = 3n+5$ should just be $O(n) = n$
b. Throw away any "lower order terms"
   a. $O(n) = n^3 + n^2$ should just be $O(n) = n^3$
c. Always assume n is large enough to be bigger than any constant
   a. if one algorithm takes 10000 calculations ($O(n) = c$) and another algorithm is $O(n) = n$, we can assume the 2nd algorithm is "slower" because we will assume n is much larger than 10000.
d. *for* Loops tend to generate $O(n) = n$ but *while* loops are more tricky
   a. nested for loops (i.e. loop within loop) will multiply O(n)'s
   b. nested statements will multiply O(n) with outer statement's O(n)
   c. consecutive loops do not multiply, they add. Therefore, O(n) is equal to the larger of the two loop's O(n).

e. any time we "cut in half" the size of the array or data structure, the Big-Oh tends to be logarithmic i.e. O(n) = log(n)
f. Big-Oh is an approximation and tends to be "worst-case" scenario which may not happen that often in actual use. It is useful for "large n" only.

For a true understanding of what this implies consider this table

	n	$Log_2(n)$	n $Log_2(n)$	n^2	n^3	2^n
	1	0	1	1	1	2
	2	1	2	4	8	4
	4	2	8	16	64	16
	8	3	24	64	512	256
	16	4	64	256	4096	65536
	32	5	160	1024	32768	2147483648
	256	8	2048	65536	16777216	Oh My Gosh
	1 million	20	20 Million	1 Trillion	1 Trillion Millions	Forget It
ex.	linear searrch	binary search	quicksort	bubble sort	weather simulartions	population simulations

Notes

1) The Oh My Gosh would take approximately 12 weeks on a current machine .

2) For 1 Million member array -
The Bubble sort would take 50,000 times as long as the Quicksort .

B. Counter

```
for (int j = 0; j < 100; j++)
      System.out.println(j);
```

Analysis: O(n) = n

C. Accumulation

```
int sum = 0;
for (int j = 0; j < 100; j++)
      sum += j;                    // sum = 0+1+2+3+...+99+100
```

Analysis: O(n) = n

D. Swap

Analysis: O(n) = c

1. **simple - local variables, local scope**

```
int temp = x;
x = y;
y = temp;
```

2. **method on field variables**

```
public void swap()
{
      int temp   = this.x;
          this.x = this.y;
          this.y = temp;
}
```

3. **method call to swap local variables - naive attempt...won't work**

```
public void localSwap()
{
      int x = 5, y = 6;       // local variables
      swap2(x,y);             // works???       ...not possible
}
```

4. **swap method on primitive array**

```
public void swap(int a[], int index1, int index2)
{
      int temp  = a[index1];
      a[index1] = a[index2];
      a[index2] = temp;
}
```

5. **swap method on ArrayList (or List)**

```
public void swap(List a, int index1, int index2)
{
      Object temp = a.getAt(index1);
      a.set(index1, a.getAt(index2));
      a.set(index2, temp);
}
```

E. Search

1. Linear Search - no assumptions other than can check for equality [== or equals()]

a) Approach

Start at the beginning and check each element one by one to see if the target is an element of the array

b) Algorithm

```
for (j = 0 to j < length of the array, j++)
      if (target == a[j]) then found target and return
after loop is done -- haven't found target so return false
```

c) Code

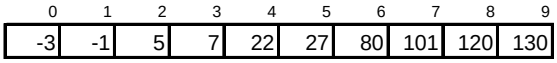
```
public boolean searchPrimitiveArray(int [] a, int target)
{
      for (int j = 0; j < a.length; j++)
            if a[j] == target then return true;
      return false;
}

public boolean searchObjectArray(ArrayList a, Object target)
{
      for (int j = 0; j < a.size(); j++)
            if a.getAt(j).equals(target) then return true;
      return false;
}
```

d) Analysis: each takes one loop through each element of the array so $O(n) = n$

2. Binary Search - *assume the data structure is sorted in ascending order*

a) diagram

leftIndex middleIndex rightIndex



a) target = 27
b) target = 6

b) Approach

1. find middle element of array.
2. if middle element is target, then return true
3. if target is not at middle then it may be in either first half of array or 2nd half of array
   3a. if in first half, then "cut" array into the "lower half" and repeat steps 1 - 3 on only lower half
   3b. if in 2nd half, then "cut" array into "upper half" and repeat steps 1 - 3 on only upper half

c) Algorithm

1. need leftIndex, rightIndex, and middleIndex
2. while ((haven't found target) && (not finished cutting array in half))
   2a. if (middle element == target) return true
   2b. if (target < middle element) then move rightIndex down to middle
   2c. else move leftIndex up to middle
   2d. recalculate middleIndex

Issues:

1. need to keep track of whether target is found .... boolean found = false
2. how to calculate middleIndex ... middle = first + last / 2 !!!! logic error
   middle = (first + last) / 2
3. Boundary Conditions:
   a. how to know when we are done cutting array in half? (target not in array) ---> when leftIndex > rightIndex
   b. why don't we want to move leftIndex up to middleIndex????
   ---> leftIndex = middleIndex + 1 // so don't have an infinite loop

d) Code

```
public boolean binarySearchPrimitives(int [] a, int target)
{
	int left = 0, right = a.length, middle = (left+right)/2;
	boolean found = false;
	while ((!found) && (left <= right))
	{
		if (a[middle] == target)
			found = true;		// or return true ????
		else
			if (target < a[middle])
				right = middle - 1;
			else
				left = middle + 1;
		middle = (left + right) / 2;
	}
	return found;
}
```

e) Analysis **Running Total**

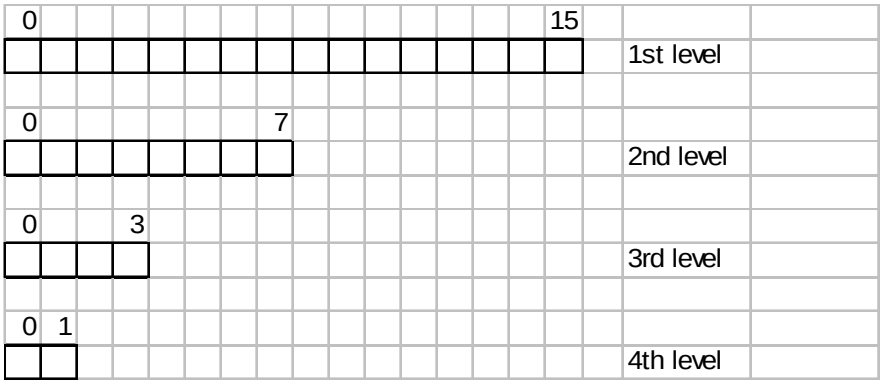
1 statement	1		1
1 statement	1		2
loop (if not found, cut array in half) --->	***$log_2(n)$!!!!***		2+ $\log_2(n)$
1/2 if statement	0.5		
1 statement		1	2+0.5$\log_2(n)$
1/2 else	0.5		
1 if statement		.5	
1 statement			
1 else statement	.5		
1 statement			2+(1)$\log_2(n)$
1 statement	1		2+(2)$\log_2(n)$
1 statement	1		3+(2)$\log_2(n)$

$O(n) = \log_2(n)$

WHY ***$log_2(n)$!!!!***

Example: suppose n = 16, the important question becomes
"how many times will the body of loop be executed?"

If n = 16, then we <u>may</u> need to divide it into two "halves" repeatedly



1. The number of levels will determine how many times the loop body must be executed.
2. $2^{\#\text{ of levels}} = 2^4 = 16 = n$
3. solving for the "# of levels" by taking $\log_2$ of both sides gives:

$$\log_2 (2^{\#\text{ of levels}}) = \#\text{ of levels} = \log_2 (16) = \log_2 (n)$$

What would happen if we could divide the array into 3rds rather than halves?
Answer: the log would be base 3 rather than base 2 (i.e. $\log_3$)

F. Divisibility (modulus)

1. Greatest Common Factors of two integers, x and y.

a) Approach -- try out on actual numbers. Let x = 54 and y = 24 GCF = ??? (6)

1. Start with a one of the numbers (24) and test to see if it is a divisor of BOTH numbers. If it is, then done
2. Else check next number smaller (23) and test to see if it is divisor of BOTH numbers. If it is, then done
3. Repeat process until down to 1 (which must be a divisor of both numbers)

b) Algorithm

```
for (int j = x; j > 1; j--)
        if (j is divisor of x) and (j is divisor of y) then return j
return 1;
```

Issues:

1. how do we tell if "j is divisor of x"? ---> `(x % j) == 0`

c) Code

```
public int greatestCommonFactor (int x, int y)
{       for (int j = x; j > 1; j--)
                if ((x%j)==0) && ((y%j == 0)) return j;
        return 1;
}
```

d) Analysis

one for loop that goes from n down to 1. Therefore O(n) = n

2. Least Common Multiple of two integers, x and y.

a) Approach -- try out on actual numbers. Let x = 54 and y = 24 LCM = ??? (216 = 2x2x2x3x3x3)

1. start with one of the numbers
2. generate and check every multiple to see if y is a divisor of the multiple.

b) Algorithm

1. int possibleLCM = x;
2. loop until done
   2a. if (possibleLCM % y = = 0) then return possibleLCM
   2b. update possibleLCM;

Issues:

1. when are we "done"? ---> when possibleLCM => x*y
2. how do we update possibleLCM? - --> possibleLCM += x;
3. what kind of loop (for or while)? ---> while is more flexible

c) Code

```
public int leastCommonMultiple (int x, int y)
{
        int possibleLCM = x;
        while (possibleLCM < x * y)
        {
                if ((possibleLCM%y) == 0) return j;
                possibleLCM += x;
        }
        return x*y;
}
```

d) Analysis

one *while* loop whose worst case is x*y or n*m or n*n --> O(n) = n^2

best case is O(n) = 1 (find the LCM right away)

G. Sorts (assume the data structures are "comparable")

Purpose: Most computers are used for sorting and searching for data.
Examples: Phone book, databases for credit card records, scheduling, SSN & tax forms?
Big Problem: The Internet suffers from a lack of comprehensive way to sort and search through all the data in a "reasonable amount of time" -- Big-Oh analysis. XML might help in this regard. Does Google do a good job????

1. Selection Sort

a) diagram

scan through to find largest (15 at index 2)

0	1	2	3	4	5	6
7	15	-2	6	3	15	11

swap

0	1	2	3	4	5	6
7	*11*	-2	6	3	15	*15*

scan through on shortened array (15 at index 5)
swap with itself

0	1	2	3	4	5	6
7	*11*	-2	6	3	*15*	*15*

scan through on shortened array (11 at index 1)

0	1	2	3	4	5	6
7	*3*	-2	6	*11*	*15*	*15*

swap

scan through on shortened array (7 at index 0)

0	1	2	3	4	5	6
6	*3*	-2	*7*	*11*	*15*	*15*

swap

scan through on shortened array (6 at index 0)

0	1	2	3	4	5	6
-2	*3*	*6*	*7*	*11*	*15*	*15*

swap

b) Approach

1. Scan through entire array and select the largest element.
2. Put this element in last slot and "shorten" array by one element
3. Repeat step 1 on "shortened" array.

c) Algorithm

{Precondition: We have a List/Array of "comparable" objects called "A[]" of size "n"}
{Postcondition: The entire List/Array is sorted in "ascending" order}

1. Do steps a & b *for endOfArray = n - 1 down to 1* // shorten the array each pass
   a. Scan through "shortened" array and record the largest element, max_i and its location, maxIndex.
   b. Swap max_i and last element of shortened array (i.e. A[maxIndex] and A[endOfArray]

d) Code

```
public boolean selectionSort(Comparable A[])
{
      for (int endOfArray = A.size(); endOfArray > 0; endOfArray--)
      {
            int max = A[0];   // start max at first element
            int maxIndex = 0;
            for (int i = 0; i < endOfArray; i++)
                  if (A[i] > max) {max = A[i]; maxIndex = i}
            swap(A,maxIndex,endOfArray);
      }
}
```

e) Analysis

Ignoring the "individual" statements (O(n) = c). We can focus on the loops.
Two loops: outer loop is O(n) = n and inner loop is O(n) = n/2 (on average)
Since nested, multiply to get O(n) = n(n/2) = n^2

2. Insertion Sort

a) General Method –
Like putting cards in order
i.e. assuming all the cards already picked are in order, pick up next card, determine where it fits in, insert it

b) Example

new item to be inserted

	0	1	2	3	4	5	6	
A	***-8***	***0***	***5***	***17***	-2	21	4	...

in order (move over) pos
move each over 4
insert where it belongs

a) diagram

0	1	2	3	4	5	6
10	17	3	5	-2	16	3

arraysize = 1

10

arraysize = 2

10	17

move 17 into place

arraysize = 3

3	10	17

move 3 into place

arraysize = 4

3	5	10	17

move 5 into place

arraysize = 5

-2	3	5	10	17

move -2 into place

arraysize = 6

-2	3	5	10	16	17

move 16 into place

arraysize = 7

-2	3	3	5	10	16	17

move 3 into place

b) Approach

1. Assume the first element in array is sorted.
2. Add each element by assuming the previous elements are sorted.
    a. When adding the next element, it will be at the "end" of the partially sorted array.
    b. temporarily store this next element. Move consecutive prior elements to the right until the "next element" can be inserted into its "proper spot"
3. Repeat step 2 until all the elements have been "inserted" into their correct order.

c) Algorithm

{Precondition: We have a List/Array of "comparable" objects called "A[]" of size "n"}
{Postcondition: The entire List/Array is sorted in "ascending" order}

1. for currentSorted = 1 to n // scan through the entire array to insert each element
    a. Set nextElement = A[currentSorted]
    b. compareIndex = currentSorted - 1
    c. while (compareIndex > 0 && A[compareIndex] > nextElement)
        i. A[compareIndex] = A[compareIndex-1] // move each "bigger" element to the right
        ii. decrease compareIndex by one
    d. insert nextElement into A[compareIndex]

```
for (Pos = 1; Pos < Nbrterms, Pos++)     // for each item to be inserted
  {
     Temp = A[Pos];                             // hold new number temporarily
     J = Pos – 1;                          // look at the # right before it
     while (J >= 0 && (A[J] > Temp))     // while # not in right slot
       {
          A[J+1] = A[J];                 // copy # in slot to right
          J--;                                  // move down array
       }
     A[J+1] = Temp;                             // put new # in correct slot
  }
```

d) Code

```
public boolean insertionSort(Comparable A[])
{
      for (int currentSorted = 1; currentSorted < A.size(); currentSorted++)
      {
            int nextElement = A[currentSorted];
            int compareI = currentSorted-1;
            while ((compareI > 0) && (A[compareI] > nextElement))
            {
                  A]compareI] = A[compareI-1];
                  compareI--;
            }
            A[compareI] = nextElement;
      }
}
```

e) Analysis

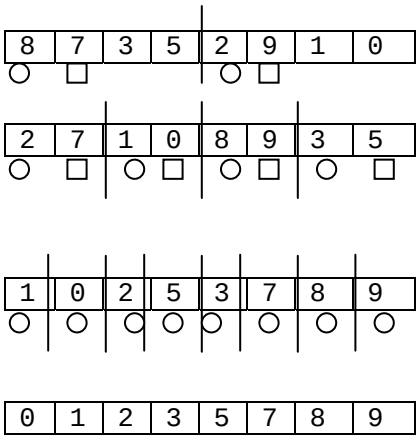
Outer loop is O(n) = n; Inner Loop will, on average, be O(n) =n/2
Nested loops multiply so O(n) = n^2

3. Bubble - already done in Programming in Java course

4. Shell Sort - Donald Shell 1958

a) Example using NumDiv = 2 (i.e. cutting the array in divisions of 2)

Nterms = 8



Insertion sort circles then the boxes
Then subdivide the array into NumDiv=4 parts

Insertion sort circles then boxes
Then subdivide the array into NumDiv=8 parts

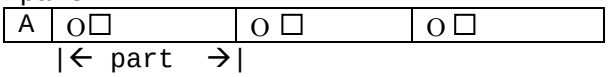
Insertion sort the circles

b) General Algorithm

```
numDiv = 3;
p = nbrTerms / numDiv;                  // divide array into p "parts"

1)in a loop for J = 1 to p              // scan through the items in single part
     Insertion Sort (A[J], A[p + J], A [2p + J], etc.)
                                        // insertion sort every "part-th" number

2) numDiv = 3 * numDiv;                 // make more divisions
3) p = Nbrterms / NumDiv;               // there are fewer #'s in each "part"

4) Repeat 1) while numDiv > nbrTerms    // there are more than 1 # in each part
```

c) Ergo

J =|1 2 3… p |(p+1)(p+2)… |(2p+1) (2p+2)…| p = Nbrterms/NumDiv = #s in each "part"

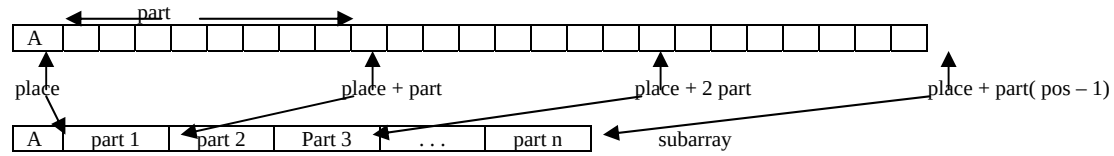


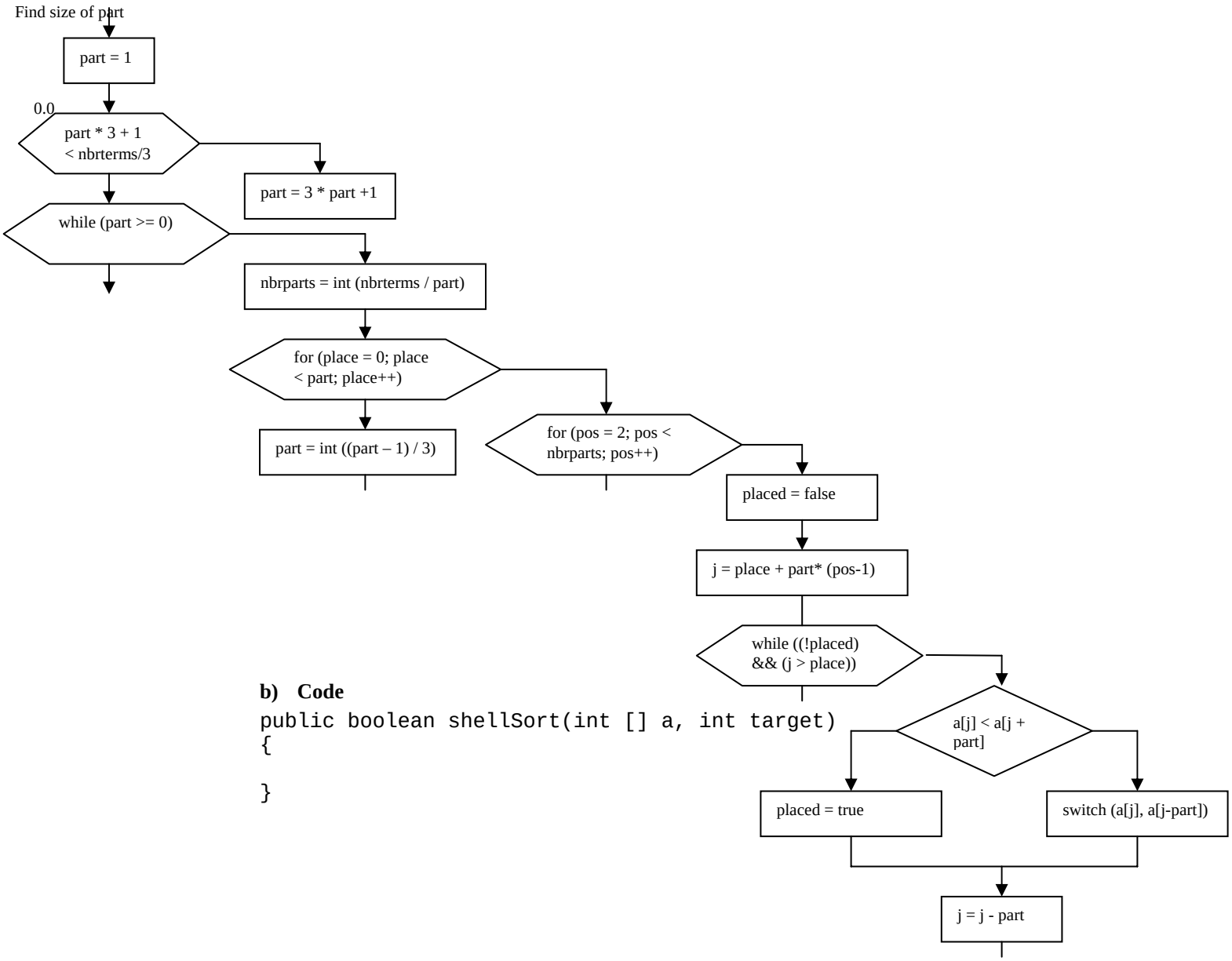
|← part →|

in p = parts sorts, first insertion sort the O's
A[1], A[p + 1], A[2p + 1], A[3p + 1], etc.
then insertion sort the □ 's
A[2], A[p + 2], A[2p + 2], etc.
etc.

then divide A into 3x as many parts and repeat the process

d) This is more efficient because members out of order are moved much farther initially than a simple insertion sort. The optimal value for numDiv turns out to be 3.452...





b) Code

```
public boolean shellSort(int [] a, int target)
{

}
```

5. Quicksort - C.A.R. Hoare 1962

a) diagram

b) Approach (÷ and conquer)

Since for most sorts # of terms => time^2

2x as many elements => 4x as long
10x as man\y elements => 100x as long

or

2 sets of half => 2(1/4 as long) = ½ as long
4 sets of 4ths => 4(1/16 as long) = ¼ as long
etc.

c) Algorithm

Pick some element (to put in middle--easy to pick "first" element to put in middle)
put all larger above it
and all smaller below it
repeat the process on each half

Using 2 functions
Partition
QuickSort

NOTE: make sure you chose your variable names carefully! (i.e. does a variable refer to an element of the array or an index/pointer)

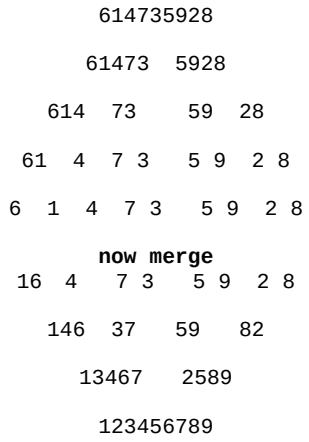
d) Code

```
public void quickSort (int A[]) {quicksort(A,0,A.length);}

private void   quicksort   (int A[maxsize],
                            int frontIndex,
                            int backIndex)
{
      int middleIndex;

      if (frontIndex < backIndex)
      {
            middle = partition (A, frontIndex, backIndex) ;
            Quicksort (A, frontIndex, middleIndex - 1) ;
            Quicksort (A, middleIndex + 1, backIndex);

      } // End if

} // End Fcn Quicksort



private int   partition   (int A[maxsize],
                           int frontIndex ,
                           int backIndex)

{
      int pivotElement ;
      int leftIndex, rightIndex ;

      pivotElement = A[frontIndex] ;
      leftIndex    = frontIndex ;
      rightIndex   = backIndex + 1 ;

      do
      {

            //  Move Left over until a value >= to Pivot is found
            do
               leftIndex++ ;
            while ((A[leftIndex]  < pivotElement) && (leftIndex < rightIndex)) ;

            //  Move Right over until a value <=  to Pivot is found
            do
               rightIndex-- ;
            while (A[rightIndex] > pivotElement) ;

            if (leftIndex < rightIndex)
               Switch (A[leftIndex], A[rightIndex]) ;
      }
      while (leftIndex < rightIndex) ;

      // put the pivotElement at frontIndex in correct place and return its position
      Switch (A[frontIndex], A[rightIndex]) ;
      return rightIndex ;

   } // End Fcn Partition
```

e) Analysis

6. MergeSort – similar to Quicksort but in reverse order

a) diagram



b) Approach

1. Divide the array into two halves (perfect halves)
2. Continue dividing each half into further halves until down to 1 element
3. Merge each of 2 elements and put into one part in order
4. Continue merging “halves” into wholes in order until re-constructing original array in order

c) Algorithm

```
mergeSort()
      if (nElements < 2) then stop
      else
            if (nElements > 1)
                  mergeSort the left half
                  mergeSort the right half
                  merge the two halves

How to merge two halves given parameters:  arrayA, lowIndex, highIndex

1. make a copy of arrayA and call it arrayB
2. initilize a Acounter = lowIndex
3. calculate the middleIndex = (lowIndex + highIndex) / 2
4. initialize leftIndex = lowIndex;  rightIndex = middleIndex
5. while (leftIndex < middleIndex) && (rightIndex < highIndex)
      if (B[leftIndex] > B[rightIndex] then
            A[Acounter] = B[rightIndex]         // put smaller into A
            rightIndex++                        // move right one over
      Else
            A[Acounter] = B[leftIndex]          // put smaller into A
            LeftIndex++                         // move left one over
      Acounter++                                // increment the A counter
6. if (leftIndex = middleIndex)                 // left side is finished
      Copy rest of right side of B into A
   Else                                         // right side is finished
      Copy rest of left side of B into A
                  .
```

d) Code

```
public boolean mergeSort(int [] a, int target)
{
}
```

e) Analysis

7. HeapSort - combination of arrays and trees

a) diagram

b) Approach

1.

c) Algorithm

1.

d) Code

```
public boolean heapSort(int [] a, int target)
{

}
```

e) Analysis