Analysis of Algorithm Continue

Computing the Complexity Function of an Algorithm

- We do not need to count all basic operations performed by the algorithm

- For example, if a loop executes *N* times and each loop iteration executes a constant number of basic operations, the entire loop executes *N* basic operations

- For most algorithms, we can pick one type of basic operation and count just that to determine the complexity of the algorithm.

Selecting the Basic Operations to Count

- For most algorithms we can count just one or two types of basic operations.
- Select operations that are germane to the problem: for example, sorting and searching algorithms should count comparisons between array entries.
- Select at least one operation in every loop.

Analysis of Bubble Sort

To sort an array *A*[0..*N*−1]:

```
for (int last = N -1; last >= 1; last --)
{
  // Move the largest entry in A[0…last] to A[last]
  for (int index = 0; index <= last-1; index++)
  {
       //swap adjacent elements if necessary
        if (A[index] > A[index+1])
        {
            int temp = A[index];
            A[index] = A[index+1];
            A[index + 1] = temp;
        }
  }
}
```

Bubble Sort (One Pass)

Bubble sort uses an index to keep track of which pair of adjacent elements should be swapped during a sweep through A[0..last].

```
15 35 20 10 25     // index = 0, Compare A[0], A[1]
15 35 20 10 25     // index = 1, Compare A[1], A[2], swap
15 20 35 10 25
15 20 35 10 25     // index = 2, Compare A[2], A[3], swap
15 20 10 35 25
15 20 10 35 25     //index = 3, Compare A[3], A[4], swap
15 20 10 25 35     //Largest is at A[4]
```

4 Comparisons and 3 swaps for 1 pass

Analysis of Bubble Sort with 3 printlns

```
for (lastPos = array.length - 1; lastPos >= 0; lastPos--)
    {
      for (index = 0; index <= lastPos - 1; index++)
      {
        // Compare an element with its neighbor.
        System.out.println("Before comparing");
        if (array[index] > array[index+1])
        {
          // Swap the two elements.
          System.out.println("Swapping");
          temp = array[index];
          array[index] = array[index + 1];
          array[index + 1] = temp;
        }
      }
      System.out.println("---------------");
    }
```

3 printlns's Result

```
 ----jGRASP exec: java ObjectBubbleSortTest

Original order:
15 35 20 10 25
Before comparing
Before comparing
Swapping
Before comparing
Swapping
Before comparing
Swapping
-------------
Before comparing
Before comparing
Swapping
Before comparing
-------------
Before comparing
Swapping
Before comparing
-------------
Before comparing
-------------
-------------

Sorted order:
10 15 20 25 35

 ----jGRASP: operation complete.
```

4 compares

3 compares

2 compares

1 compare

Bubble Sort's Analysis

Let the number of comparisons for bubble sort, C, is given by:

$C = (n - 1) + (n - 2) + \ldots\ldots + \quad 2 + \quad 1 \qquad (1)$

Ex: $C = 4 + 3 + 2 + 1$

Writing the terms in reverse order, we also have

$C(n) = \quad 1 \quad + \quad 2 \quad + \ldots\ldots + (n - 2) + (n - 1) \qquad (2)$

Adding the 2 equations (1) and (2):

$2C(n) = (n + n + n + n + \ldots + n)$

$C(n) = n(n - 1) / 2$ (3) (NOTE that there are (n - 1) terms dividing by 2)

$C(n) = n^2/2 – n/2$

$C(n) = O(n^2/2)$ -- Term $n^2/2$ dominated $n/2$ when n get large

$C(n) = O(n^2)$ – Constant ½ absorbed into the Big O

Analysis of Insertion Sort

Insertion Sort

- Note that for any array A[0..N-1], the portion A[0..0] consisting of the single entry *A*[0] is already sorted.

- Insertion Sort works by extending the length of the sorted portion one step at a time:
  - A[0] is sorted
  - A[0..1] is sorted
  - A[0..2] is sorted
  - A[0..3] is sorted, and so on, until *A*[0..N-1] is sorted.

Insertion Sort

The strategy for Insertion Sort:

```
//A[0..0] is sorted
 for (index = 1; index <= N -1; index ++)
 {
     // A[0..index-1] is sorted
    insert A[index] at the right place in A[0..index]
     // Now A[0..index] is sorted
 }
 // Now A[0..N -1] is sorted, so entire array is
sorted
```

How Insertion Sort Works

15 10 55 35 30 20 index = 1, Insert A[1] = 10 into A[0..1]:

10 15 55 35 30 20

10 15 55 35 30 20 index = 2, Insert A[2] = 55 into A[0..2]:

10 15 55 35 30 20

10 15 55 35 30 20 index = 3, Insert A[3] = 35 into A[0..3]:

10 15 35 55 30 20

10 15 35 55 30 20 index = 4, Insert A[4] = 30 into A[0..4]:

10 15 30 35 55 20

10 15 30 35 55 20 index = 5, Insert A[5] = 20 into A[0..5]:

10 15 20 30 35 55

10 15 20 30 35 55 Array is now sorted

The portion of A[0..last] already examined

The entry to be inserted is in bright blue.

A Closer Look at the Logic of the Insertion Step

A[0..4] is already sorted, insert A[5] into A[0..5]:

10 15 30 35 55 20   index = 5, Insert A[5] = 20 into A[0..5]

----

unsortedValue = 20, will scan for the right place to put it.

Use a variable scan to find the place where A[scan-1] is less or equal to unsortedValue:

10 15 30 35 55 __   // scan = 5

10 15 30 35 __ 55   // scan = 4

10 15 30 __ 35 55   // scan = 3

10 15 __ 30 35 55   // scan = 2

Drop in the unsorted value:

10 15 20 30 35 55   // A[scan] = unSortedValue

Insertion Sort:
insert *A*[index] at the right place in *A*[0..index]

```
//A[0..index-1] is already sorted
int unSortedValue = A[index];
scan = index;

while (scan > 0 && A[scan-1] > unSortedValue)
{
    A[scan] = A[scan-1];
    scan --;
}

// Drop in the unsorted value
A[scan] = unSortedValue;
```

Insertion Sort

```
//A[0..0] is sorted
for (index = 1; index <= N -1; index ++)
{
     // A[0..index-1] is sorted
     // insert A[index] at the right place in A[0..index]
     int unSortedValue = A[index];
     scan = index;

     while (scan > 0 && A[scan-1] > unSortedValue)
     {
        A[scan] = A[scan-1];
        scan --;
     }

     // Drop in the unsorted value
     A[scan] = unSortedValue;
     // Now A[0..index] is sorted
}
// Now A[0..N -1] is sorted, so entire array is sorted
```

outer **loop**

outer **times**

inner **loop**

inner **times**

Insertion Sort: Number of Comparisons

# of Sorted Elements	Best case	Worst case
0	0	0
1	1	1
2	1	2
…	…	…
n-1	1	n-1
	n-1	**n(n-1)/2**

Remark: we only count comparisons of elements in the array.

Recursive Binary Search

- The logic of binary search has a natural recursive implementation:
- If lower > upper, then return -1 (base case).
- Compare X to A[middle], where middle is the midpoint between lower and upper:

  middle = (lower + upper)/2

- If X == A[middle], return middle (we found it!)
- If X < A[middle], then continue search in A[lower..middle-1]
- If X > A[middle], then continue search in A[middle+1..upper]

Analysis Of Recursive Binary Search

Without loss of generality, assume n, the problem size, is a multiple of 2, i.e., $n = 2^k$

Expanding:

$T(1) = a$ (1)

$T(n) = T(n / 2) + b$ (2)

And we know $T(n / 2) = T(n/4) + b$

$= [T(n / 2^2) + b] + b = T(n / 2^2) + 2b$ by substituting $T(n/2)$ in (2)

$= [T(n / 2^3) + b] + 2b = T(n / 2^3) + 3b$ by substituting $T(n/2^2)$ in (2)

= ……..

$= T(n / 2^k) + kb$

The base case is reached when $n / 2^k = 1$ ➔ $n = 2^k$ ➔ $k = \log_2 n$, we then have:

$T(n) = T(1) + b \log_2 n$

$= a + b \log_2 n$

Therefore, Recursive Binary Search is **O(log n)**

In Summary: Recurrence Relations to Remember

- Recognizing Common Recurrences
- Below are some algorithms and recurrence relation encountered
- Solve once, re-use in new contexts

Recurrence	Algorithm	Big-O Solution
$T(n) = T(n/2) + O(1)$	Binary Search	$O(\log n)$
$T(n) = T(n-1) + O(1)$	Sequential Search	$O(n)$
$T(n) = 2\,T(n/2) + O(1)$	tree traversal	$O(n)$
$T(n) = T(n-1) + O(n)$	Selection Sort (other n^2 sorts)	$O(n^2)$

Orders of common functions

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a constant-size lookup table or hash table
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.
$O(n^c),\ 0 < c < 1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two n-bit integers.
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two *n*-digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), shell sort, quicksort (worst case), selection sort or insertion sort
$O(n^c),\ c > 1$	polynomial or algebraic	Tree-adjoining grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha, c],\ 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n),\ c > 1$	exponential	Finding the (exact) solution to the traveling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the traveling salesman problem via brute-force search; finding the determinant with expansion by minors.

Classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case, c is a constant and n increases without bound. The slower-growing functions are generally listed first. (Source: http://en.wikipedia.org/wiki/Big_O_notation)