Sorting, Searching, and Analysis of Algorithm

Topics

- Introduction to Sorting Algorithms
- Introduction to Searching Algorithms
- Analysis of Algorithms

Introduction

- Common problem: sort a list of values, starting from lowest to highest.
  - List of exam scores
  - Words of dictionary in alphabetical order
  - Students names listed alphabetically
  - Student records sorted by ID#
  - Optimizing the use of other algorithms: search & merge (required inputs to be sorted)
  - Generally, we are given a list of records that have *keys.* These keys are used to define an ordering of the items in the list.

What is Sorting?

- An array *A*[ ] of *N* numbers is sorted in ascending order if the array entries increase (or never decrease) as indices increase:

$$A[0] \leq A[1] \leq \cdots \leq A[N-2] \leq A[N-1]$$

What is Sorting?

Example of an array sorted in ascending order:

20 25 30 40 55

(20 < 25 < 30 < 40 < 55)

Example of an array that is not sorted in ascending order:

20 25 15 40 12

Note: Array[1] = 25 > Array[2] = 15

Sorting Algorithms

There are many strategies for sorting arrays.

Among them:

- Bubble sort
- Selection sort
- Insertion sort
- Quicksort

Quadratic Sorting Algorithms

- We are given *n* records to sort.
- There are a number of simple sorting algorithms whose **worst** and **average** case performance is quadratic O(n^2):
  - Selection sort
  - Insertion sort
  - Bubble sort
- Quicksort ?

Notation

- Let A[ ] be an array of length N
- Let last be an index in the range of the array:

$$0 \le \text{last} \le N-1$$

- A[0..last] denotes the portion of the array consisting of A[0], A[1], …, A[last]

A Simple Sorting Strategy

```
for (last = N -1; last >= 1; last --)
{
   Move the largest entry in A[0…last] to A[last]
}
```

A Simple Sorting Strategy

How it works on 15 35 20 10 25.

Portion of array already sorted in red.

Largest value in unsorted portion of array in bright blue.

15 35 20 10 25

- Move largest of A[0..4] to A[4]: 15 20 10 25 35
- Move largest of A[0..3] to A[3]: 15 20 10 25 35
- Move largest of A[0..2] to A[2]: 15 10 20 25 35
- Move largest of A[0..1] to A[1]: 10 15 20 25 35

```
for (last = 4; last >= 1; last--)
{
   Move largest of A[0..last] to A[last]
}
```

Selection and Bubble Sort

- Selection and Bubble Sort are similar: both use the Simple Sorting Strategy of the previous slide.

- Selection and Bubble Sort differ in how they implement the step:

  Move the largest of *A*[0..last] to *A*[last]

Bubble Sort

Bubble Sort moves the largest entry to the end of A[0..last] by comparing and swapping adjacent elements as an index sweeps through the unsorted portion of the array:

15 35 20 10 25	//Compare A[0], A[1], no swap
15 35 20 10 25	//Compare A[1], A[2], swap
15 20 35 10 25	
15 20 35 10 25	//Compare A[2], A[3], swap
15 20 10 35 25	
15 20 10 35 25	//Compare A[3], A[4], swap
15 20 10 25 35	//Largest is at A[4]

Bubble Sort (Cont – 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]
```

Bubble Sort

To accomplish the step:

move largest of A[0..last] to A[last]

Bubble Sort sweeps an index from 0 to last-1, swapping adjacent entries to put the largest element seen so far at index + 1:

```
for (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 (Cont – Multiple Passes)

- Original Unsorted Array:

  15 35 20 10 25

- Bubble sorted with multiple passes:

  15 20 10 25 35 - 1st Pass (Previous slide)

  15 10 20 25 35 - 2nd Pass

  10 15 20 25 35 - 3rd Pass

  10 15 20 25 35 - 4th Pass

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;
       }
  }
}
```

Selection Sort

Selection sort, like Bubble sort, is based on a strategy that repeatedly executes the step:

Move the largest of *A*[0..last] to *A*[last]

Selection Sort

Selection Sort moves the largest entry to the end of A[0..last] by determining the position maxIndex of the largest entry, and then swapping A[maxIndex] with A[last]:

Selection Sort uses 2 variables:

1. index: keeps track of the portion A[0..index] that has already been examined: index will range from 0 to last.
2. maxIndex: keeps track of the position of the largest entry in A[0..index]. When index equals last, the variable maxIndex will be the position of the largest entry in A[0..last].

Determining the position of the largest entry

The portion of A[0..last] already examined while looking for the largest entry is in red.

The largest entry found in the portion already examined is in bright blue.

20 35 25 15 50 40 60 45 30	index = 0, maxIndex = 0
20 35 25 15 50 40 60 45 30	index = 1, maxIndex = 1
20 35 25 15 50 40 60 45 30	index = 2, maxIndex = 1
20 35 25 15 50 40 60 45 30	index = 3, maxIndex = 1
20 35 25 15 50 40 60 45 30	index = 4, maxIndex = 4
20 35 25 15 50 40 60 45 30	index = 5, maxIndex = 4
20 35 25 15 50 40 60 45 30	index = 6, maxIndex = 6
20 35 25 15 50 40 60 45 30	index = 7, maxIndex = 6
20 35 25 15 50 40 60 45 30	index = 8, maxIndex = 6

Selection Sort:

Determining the Position of the Largest Entry

```
int maxIndex = 0;
for (int index = 1; index <= last;
   index++)
{
    if (A[index] > A[maxIndex])
        maxIndex = index;
}
// maxIndex is position of largest in
   A[0..last]
```

The Simple Sorting Strategy
Adapted for Selection Sort

```
for (last = N -1; last >= 1; last --)
{
   //Move the largest entry in A[0…last] to A[last]
   //Determine Position of the Largest element
   int maxIndex =  Pos. of the Largest entry in A[0..last]
   swap A[maxIndex] with A[last]
}
```

Selection Sort

```
for (last = N -1; last >= 1; last --)
{
     //Move the largest entry in A[0…last] to A[last]
     //Determine position of largest and store in maxIndex
     int maxIndex = 0;
     for (int index = 1; index <= last; index++)
     {
         if (A[index] > A[maxIndex])
             maxIndex = index;
     }

     // maxIndex is position of largest in A[0..last]
    // swap A[maxIndex] with A[last]
    int temp = A[maxIndex];
    A[maxIndex] = A[last];
    A[last] = temp;
}
```

Selection Sort – Sample Outputs

```
Original order:
20 35 25 15 50 40 60 45 30
largest entry in A[0…8]:60
20 35 25 15 50 40 30 45 60
largest entry in A[0…7]:50
20 35 25 15 45 40 30 50 60
largest entry in A[0…6]:45
20 35 25 15 30 40 45 50 60
largest entry in A[0…5]:40
20 35 25 15 30 40 45 50 60
largest entry in A[0…4]:35
20 30 25 15 35 40 45 50 60
largest entry in A[0…3]:30
20 15 25 30 35 40 45 50 60
largest entry in A[0…2]:25
20 15 25 30 35 40 45 50 60
largest entry in A[0…1]:20
15 20 25 30 35 40 45 50 60
```

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 <u>20</u> 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
```

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

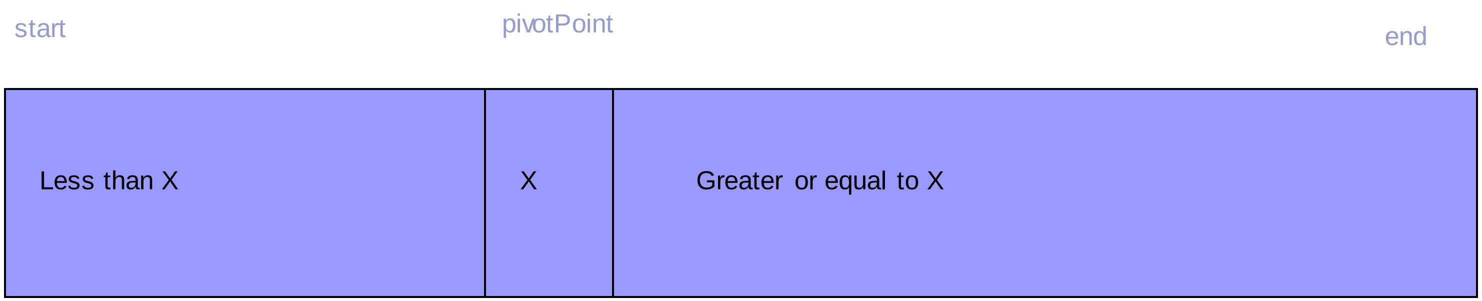
Quicksort

```
QuickSort(A, p, q)
if p < q
   then r <- Partition(A, p, q)
        QuickSort( A, p, r-1)
        QuickSort( A, r+1, q)

Initial Call : QuickSort( A, 0 , n)
```

Quicksort

To sort a segment A[start..end] of an array, Quicksort partitions it into three parts:



Quicksort

For example, in the array

50 67 93 83 90 32 68 13 75 57

You can select A[0] = 50 to be the pivot value and then partition as follows:

32 13 50 67 93 83 90 68 75 57

Result of the Partition Step

32 13 50 67 93 83 90 68 75 57

Notice:

1. The pivot value, 50, is in the right place relative to all the other elements! It is where it would be if the array were sorted.
2. The lists on the left and right side of the pivot point are not sorted, but they are shorter!

Quicksort

32 13 50 67 93 83 90 68 75 57

- Every time we partition, the pivot value is in the right place relative to all the other elements of the list.

- If we recursively carry out the same procedure on the sublists to the left and right of the pivot point, we will keep placing the pivot values in the right position while shortening the remaining sublists. Eventually the sublists get down to a length of 1 or zero, then the whole array is sorted!

Quicksort

- Suppose we have a method to do the partitioning of an array segment A[start..end]:

  int partition(int A[ ], int start, int end)

- The method partitions the segment and returns the pivot point.

Quicksort

- This recursive procedure will repeatedly partition the sublists until A[start..end] is sorted:

```
void doQuicksort(int A[ ], int start, int end)
  {
      if (start < end)
      {
            // partition A[start..end] and get the pivot point
            int pivotPoint = partition(A, start, end);

            // recursively do the first sublist
            doQuicksort(A, start, pivotPoint-1);

            // recursively do the second sublist
            doQuicksort(A, pivotPoint+1, end);
      }
  }
```

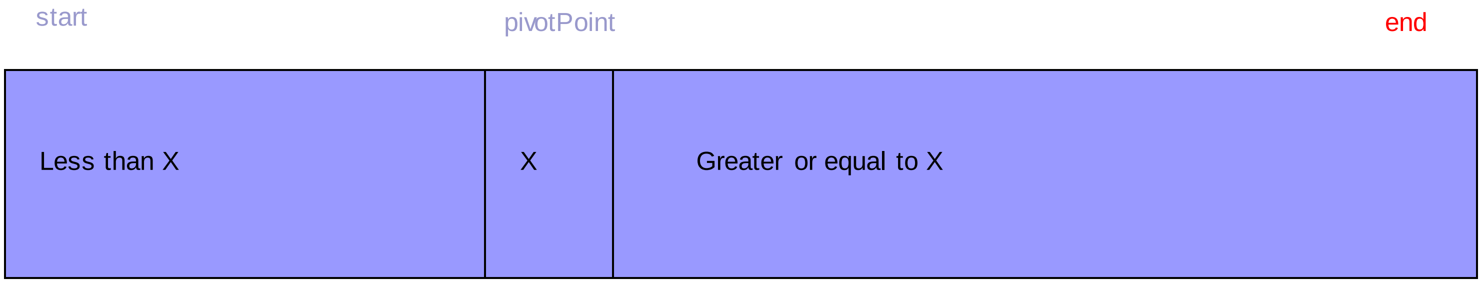
Quicksort

To sort the entire array A[0..N -1], simply call doQuicksort and pass it 0 and N-1 for start and end:

```
void Quicksort(int A[ ])
  {
      doQuicksort(A, 0, N – 1);
  }
```

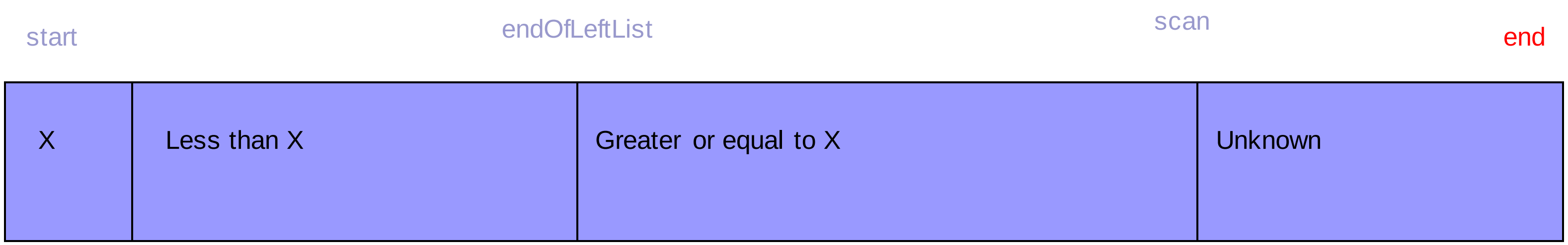
How to Partition

Given an array segment *A[start..end]*, we want to partition it and return the pivot point:



How to Partition A[start..end]

- Arbitrarily choose X= A[start] as the pivotValue, so start is the initial pivotPoint.
- Use a variable endOfLeftList to mark the end of the segment of values smaller than X, and the beginning of the segment of values larger than or equal to X
- Use a variable scan to mark the end of the segment that is larger than or equal to X
- Initially: endOfLeftList = start and scan = start+1.



The End Stage of Partition

At the end, endOfLeftList may still be equal to start.

If not, then *A*[endOfLeftList] < X.

Swap *A*[start] with *A*[endOfList] to get X between the two sublists.

How to Partition A[start..end]

```
int partition(int A[ ], int start, int end)
{
    int pivotValue = A[start];
    endOfLeftList = start;

     // At this point A[endOfLeftList] == pivotValue
    for (int scan = start + 1; scan <= end; scan ++)
    {
         if (A[scan] < pivotValue)
         {
              endOfLeftList ++;
              swap(A, endOfLeftList, scan);
              // At this point A[endOfLeftList] < pivotValue
         }
    }

   // Move the pivotValue between the left and right sublists
   swap(A, start, endOfLeftList);
   return endOfLeftList;
}
```

How to Partition A[start..end]

	Pivot =5	Start=0	End=5	Scan=1				
	EoII=0							
	Scan=1							
Array	5	1	3	6	4	2		
	EoII=1							
	Scan=1							
Array	5	1	3	6	4	2		Swapped A,1, 1 (Swapped value 1 of array to itself)
	EoII=2							
	Scan=2							
Array	5	1	3	6	4	2		Swapped A,2,2
	EoII=2							
	Scan=3							
Array	5	1	3	6	4	2		No swapping: 6 > 5
	EoII=3							
	Scan=4							
Array	5	1	3	4	6	2		Swapped A,3,4
	EoII=4							
	Scan = 5							
Array	5	1	3	4	2	6		Swapped A,4,5
	EoII=4							
Array	2	1	3	4	5	6		Final swapped A,0,4

Pivot Point

Quicksort - Recap

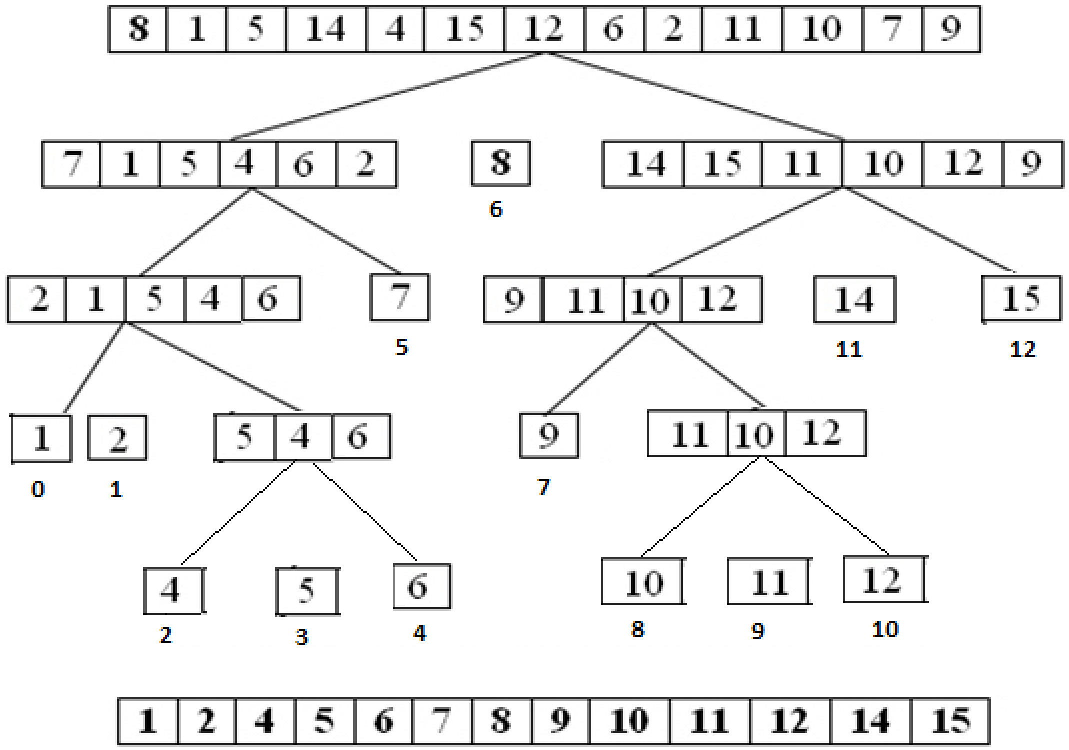
- This recursive procedure will repeatedly partition the sublists until A[start..end] is sorted:

```
void doQuicksort(int A[ ], int start, int end)
  {
      if (start < end)
      {
            // partition A[start..end] and get the pivot point
            int pivotPoint = partition(A, start, end);

            // recursively do the first sublist
            doQuicksort(A, start, pivotPoint-1);

            // recursively do the second sublist
            doQuicksort(A, pivotPoint+1, end);
      }
  }
```

Quicksort example (Recursion Tree)



Array Searching Algorithms

Array Searching Algorithms

Two methods for searching an array for a given item:

1. The Sequential Search method can be used with any array.
2. The Binary Search method can only be used with arrays that are known to be sorted, but is much faster than Sequential Search.

Sequential Search

- To search an array A[0..N-1] for a value X, start an index at one end of the array, say 0.

- Step index through the array, examining each A[index] to see if it is equal to X.

- Stop if you find X and return index. Otherwise you get to the end of the array and return -1.

Sequential Search
Search an array A[0..N-1] for X

```
int search(int A[ ], int X)
 {
     // Default assumption is X won't be found
     int position  = -1;
     boolean found = false;
     int index = 0;

     while (!found && index < N)
      {
          // check A[index]
          if (A[index] == X)
          {
               found = true;
               position = index;
          }
          index ++;
     }
     return position;
 }
```

Efficiency of Sequential Search

- In the worst case, you search the entire array, peforming *N* comparisons
- If you are lucky, you find *X* the first place you look, requiring only one comparison
- On average, you perform *N*/2 comparisons

Binary Search

Binary Search

- Works on a sorted portion *A*[lower..upper]:

- 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]

- Search terminates if **X** is found, or when we try to search an empty segment.

Binary Search of A[lower..upper]

- To continue search in A[lower..middle-1], keep lower the same and replace upper with middle-1:

    upper = middle -1

- To continue search in A[middle+1..upper], replace lower with middle+1 and keep upper the same:

    lower = middle+1

Binary Search of A[0..N-1]

```
// returns index of X if found, -1 otherwise
int binSearch(int A[ ], int X)
 {
     int lower = 0, upper = N-1;
     int position = -1;              // index of X to be returned
     boolean found = false;   // assumption is X will not be found

    // if X is there, it must be in A[lower..upper]
     while (!found && lower <= upper)
     {
           int middle = (lower + upper)/2;
           if (A[middle] == X)
           {
               found = true;   position = middle;
           }
          else if (A[middle] > X)
                 {  // if X is there, it is in A[lower..middle-1]
                     upper = middle -1;
                 }
                else
                {  //  if X is there, it is in A[middle+1, upper]
                    lower = middle +1;
                }
     }
     return position;
 }
```

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]

Recursive Binary Search of A[lower..upper]

```
int  binSearch(int A[ ], int lower, int upper, int X)
{
     // check base case for missing X
     if (lower > upper)
        return -1;

     // check if X is at the middle
     int middle = (lower + upper)/2;

      if (A[middle] == X)
        return middle;

     if (A[middle] < X)
        return binSearch(A, middle+1, upper, X);
     else
        return binSearch(A, lower, middle-1, X);
}
```

Efficiency of Binary Search

Binary Search is very efficient: large increases in the size of the array require very small increases in the number of basic steps, approximately:

size of array	# steps needed
500	8
1 thousand	10
1 million	20

Efficiency of Binary Search

- A basic step in binary search is to split the array, compare *X* to the middle element, and then select the half of the array in which to continue the search

- Each basic step reduces the size of the array to half the previous size

- If the array has size *N*, binary search will require no more than $\log N$ basic steps in the worst case

Analysis of Algorithms

Efficiency of Algorithms

- Usually there is more than one algorithm for solving a given problem.

- One algorithm may be more efficient than another, that is, it may need less time, or less memory, to solve a problem of a given size

Criteria for Measuring Efficiency

- Time: the time efficiency of an algorithm is measured by the time complexity function of the algorithm.

- Space: the space efficiency of an algorithm is measured by its space complexity function.

Computational Problems

- A computational problem is a problem that is meant to be solved by an algorithm.

- A computational problem is described by specifying the data to be input to the algorithm, and the output that should be produced by the algorithm

- Each possible input is called an instance of the problem

Instances of Problems

- Each instance is characterized by its size, the amount of memory occupied by the input data that describes the instance.

- The size of an instance is the number of bits occupied by the input, but is usually specified by giving an integer that allows us to deduce the actual size in bits.

A Typical Description of a Computational Problem

The problem of summing an array:

INPUT: an array of size *N.*

SIZE OF INPUT: *N* is the number of entries in the array.

OUTPUT: an integer representing the sum of the array entries.

The Time Complexity Function

- The time complexity $f(N)$ of an algorithm is determined by counting the number of basic steps executed by the algorithm when solving an instance of size N.

- A basic step is an operation that is executed in constant time, that is, the time to execute the operation does not increase even if the size of the input increases.

- A basic step is a theoretical approximation for an operation that could be built into the hardware of any reasonable computer.

Basic Operations

- A basic step is also called a basic operation.

- Basic steps do not specify the constant time in which they execute: we do not differentiate between a basic step that executes in 1msec and one that executes in 1000 msec.
  - No hardware dependent.

- Ignoring constant factors in this manner allows the theory to be applicable to computers with different built-in hardware operations and different technologies.

- Ignoring constant factors also means we do not differentiate between 1 basic operation, or 10, or even 100 basic operations.

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.

Average and Worst Case Complexity

- An algorithm may require a different number of basic steps in solving two different instances of the same size: When searching for *X* in an array of size *N*, Sequential search may find *X* after only 1 comparison (Best case), or may require *N* comparisons (Worst case).

- The average case complexity function averages the number of basic steps required over all instances of size *N.*

- The worst case complexity function is the number of basic steps required for those instances of size *N* that require the most work to solve.

<u>Average Case Complexity</u>

- Is a good measure to use when you want to know how an algorithm is likely to perform in <u>practice.</u>

- Requires a knowledge of the frequency distribution of problem instances: how often each instance is likely to appear in practice.

- Is difficult to use in practice because reliable estimates of frequency distributions are usually not available
  - Looks for similar problems with their instances (May give you something to get started)

Worst Case Complexity

- Measures the efficiency of an algorithm by how it does on the worst case inputs.

- Is a good measure to use when you want a performance guarantee.
  - If the worst case is predicted correctly, your program can not perform worst than that.

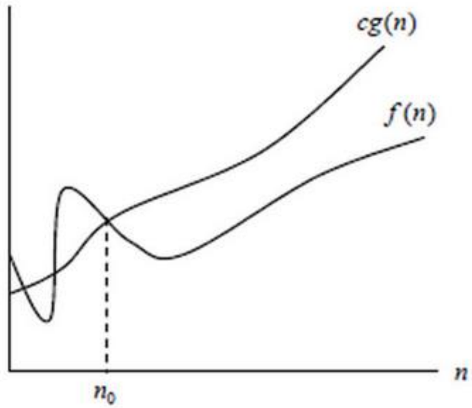
Worst Case Complexity

- The math involved in computing the worst case complexity is easier than the math in average case complexity.

- For these reasons, analysis of algorithms is usually based on worst case complexity.

The Big O Notation

The (Big) O Notation



$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

$g(n)$ is an asymptotic upper bound for $f(n)$.

Examples:

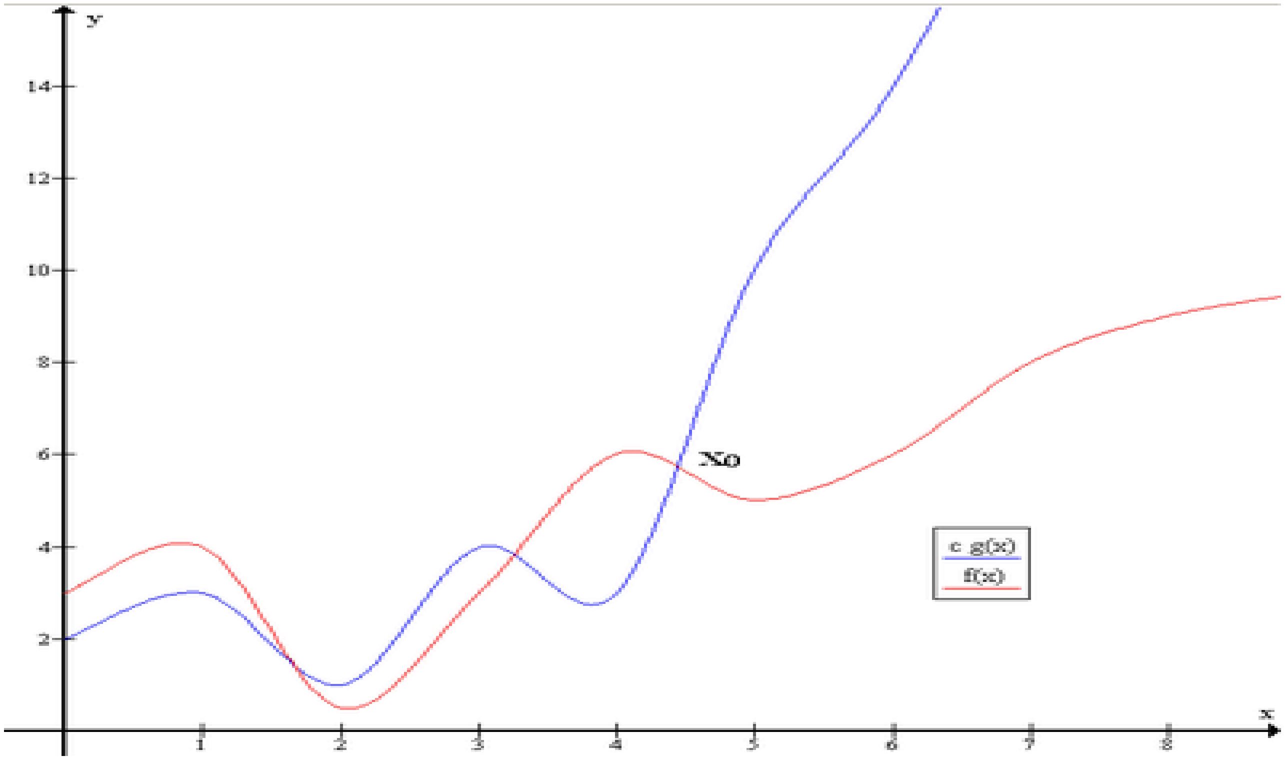
$$n^2 = O(n^2)$$
$$n^2 + n = O(n^2)$$
$$n^2 + 1000n = O(n^2)$$
$$5230n^2 + 1000n = O(n^2)$$

$$n = O(n^2)$$
$$\frac{n}{1200} = O(n^2)$$
$$n^{1.99999} = O(n^2)$$
$$\frac{n^2}{\log n} = O(n^2)$$

Note: Since changing the base of a log only changes the function by a constant factor, we usually don't worry about log bases in asymptotic notation.

Informally, saying some equation f(n) = O(g(n)) means it is less than some constant multiple of g(n). The notation is read, "f of n is big oh of g of n".

Example



Red => f(n), Blue => g(n)

Example of Big O notation: $f(x) \in O(g(x))$ as there exists $c > 0$ (e.g., $c = 1$) and x0 (e.g., x0 = 5) such that $f(x) < cg(x)$ whenever $x > x0$.

Example

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)

Comparing Algorithms by their Complexity Functions

- Let F and G be two algorithms for solving a problem, and let their complexity functions be $f(n)$ and $g(n)$.

- To see the relative performance of the two algorithms, look at the ratio $f(n)/g(n)$ as n gets large.

- We assume that $f(n) > 0$ and $g(n) > 0$ for all $n > 0$.

Comparing Complexity Functions

Simplest case in comparing two algorithms is when the limit $f(n)/g(n)$ exists as n goes to infinity. There are three possible cases where the limit exists:

- Limit of $f(n)/g(n)$ is a positive constant K
- Limit of $f(n)/g(n)$ is infinite
- Limit of $f(n)/g(n)$ is 0

Comparing Complexity Functions

Algorithm *F* has complexity function $f(n)$

Algorithm *G* has complexity function $g(n)$

If the limit of $f(n)/g(n)$ is infinite, then Algorithm *F* is taking a lot more time than *G* as the size of the problem gets bigger, so *G* is more efficient.

For example:

f(n)/g(n)= (3n^2 + 5n) / n = 3n + 5 -> infinitive (as n -> infinitive)

Conclude: G is better

Comparing Complexity Functions

Algorithm F has complexity function $f(n)$

Algorithm G has complexity function $g(n)$

If limit of $f(n)/g(n)$ is zero, then Algorithm G is taking a lot more time than F as the size of the problem gets bigger, so F is more efficient.

For example:

f(n)/g(n)= (3n^2 + 5n) / n^3 = 3/n + 5/n^2 -> 0 (as n -> infinitive)

Conclude: F is better

Comparing Complexity Functions

Algorithm *F* has complexity function $f(n)$

Algorithm *G* has complexity function $g(n)$

If the limit of $f(n)/g(n)$ is a positive constant *K*, then Algorithm *F* is performing *K* times as many basic operations as *G*. But constant factors are not significant, so *F* and *G* perform the same number of basic operations for really large problems sizes.

For example:

f(n)/g(n)= (3n^2 + 5n) / n^2 = 3 + 5/n -> 3 (as n -> infinitive)

Conclude: f and g are the same in efficiency.

The Big O Notation

- In general, the limit $f(n)/g(n)$ may not exist.
- Nevertheless, there may be a positive constant K such that $f(n)/g(n) \leq K$ when n gets large enough.
- If this happens, it means growth of $g(n)$ keeps pace with growth of $f(n)$ and keeps $f(n)/g(n)$ from going to infinity.
- This means that algorithm F is no worse than G, and we say f is in $O(g)$

Summary

- Introduction to Sorting Algorithms
- Introduction to Searching Algorithms
- Analysis of Algorithms