CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Analyzing the Time Complexity of Algorithms
- Introduction to Sorting
- Iterative Sorting Algorithms:
    - Selection Sort

Next Time:

- Insertion Sort
- Analysis of Iterative Sorts
- Recursive Sorting:
    - Mergesort
- Analysis of Recursive Sorts

Computer Science

Analyzing the Time Complexity of an Algorithm

Computer scientists study algorithms using mathematical and empirical tools; in CS 112 we begin this study by analyzing the programs we write. The MOST important kind of behavior we want to understand is its Time Complexity:

How long does the algorithm take to finish?

We will investigate this in two ways:

(1) Time the code using a clock;

(1) Count the number of instructions (lines of code) executed.

For the present, we will concentrate on (2) and compare it with (1) at this end of this week.

Analyzing the Time Complexity of an Algorithm

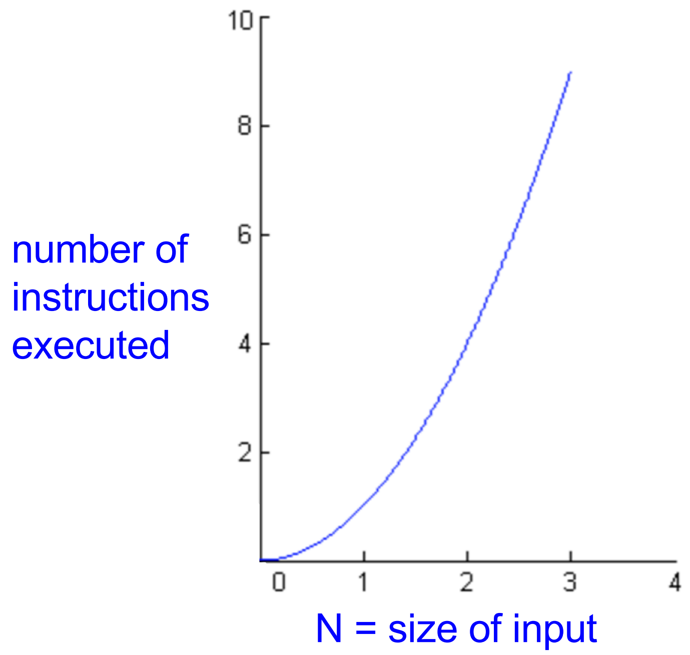
To characterize an algorithm's general behavior, we will study its

Asymptotic Time Complexity

that is, how many instructions it takes to process an input of size N, as N goes from 0 to ∞, by considering a function

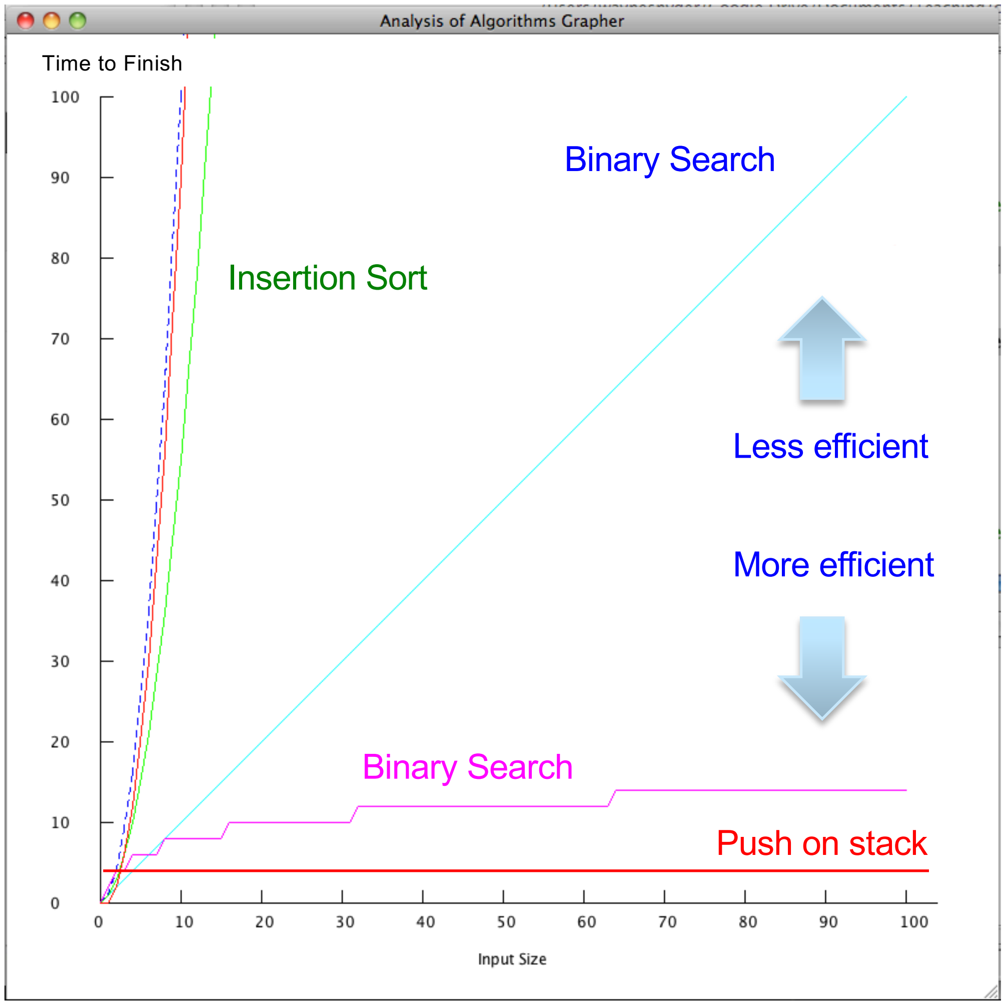
f(N) = the number of instructions necessary to process an input of size N

This gives us a sense for the overall behavior of the algorithm over all inputs.



Analyzing the Time Complexity of an Algorithm

Computer Science



The function f(N) is usually some kind of polynomial, possibly with $\log_2(N)$ factors:

Example 1: Stack or Queue operations:

$f(N) = C$ (constant)

Example 2: Searching an unordered list

$f(N) = A*N + C$

Example 3: Insertion Sort:

$f(N) = A*N^2 + B*N + C$

Example 4: Binary search:

$f(N) = A*\text{Log}_2(N) + B$

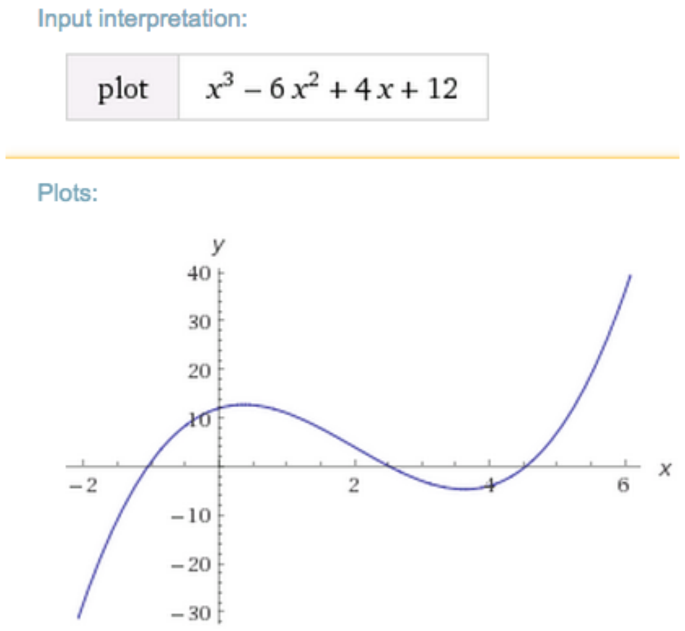
Analyzing the Time Complexity of an Algorithm

Computer Science

However, it is usual in science to provide approximations of results to focus on the important characteristics of the result, without getting overwhelmed by details.

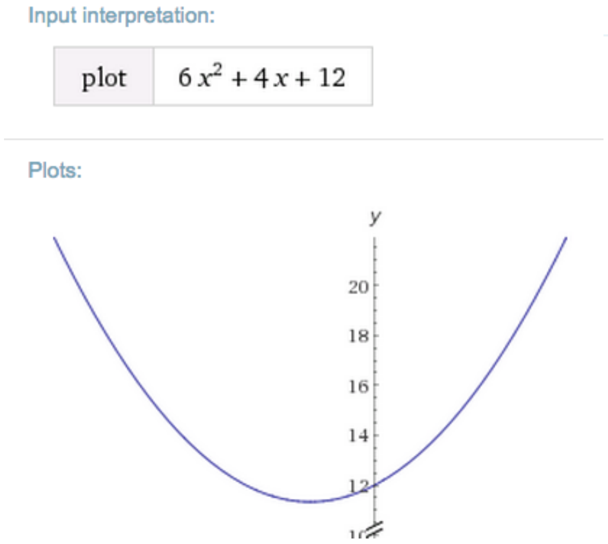
Thus in your physics class, you might report the area of a circle 1 meter square as "3.14 square meters," and not "3.141592653589793238462643 square meters"!

The same is true when characterizing the time complexity of an algorithm: we are interested in the general shape of the function f(N), which is given by the fastest-growing term (the "leading term")):


Input interpretation:

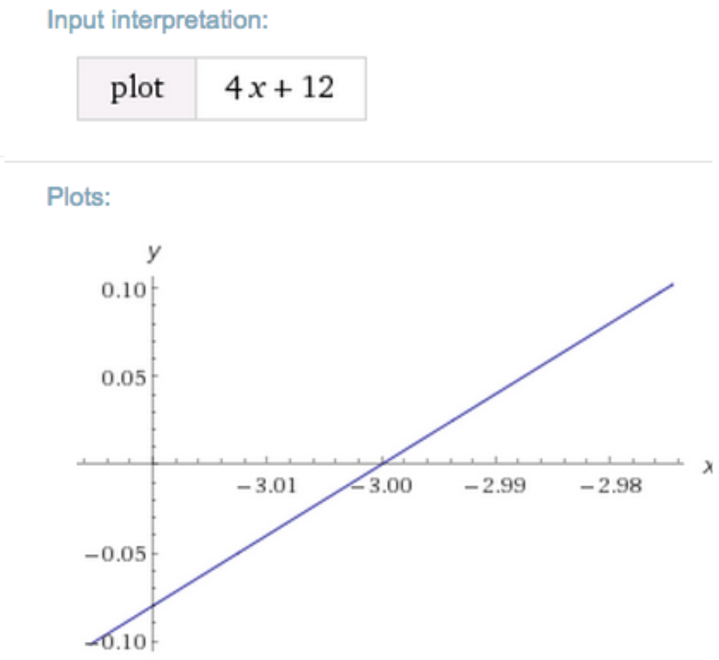
plot $x^3 - 6x^2 + 4x + 12$

Plots:


Input interpretation:

plot $6x^2 + 4x + 12$

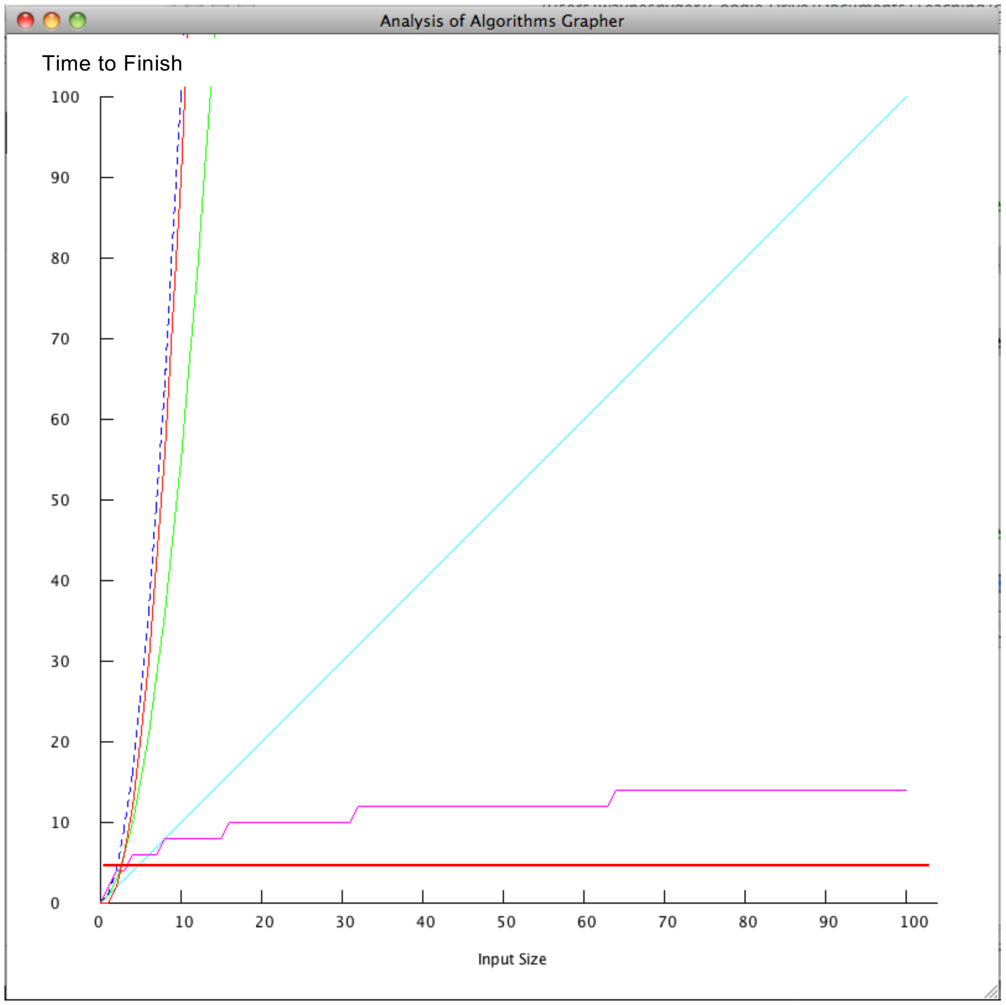
Plots:


Input interpretation:

plot $4x + 12$

Plots:

Analyzing the Time Complexity of an Algorithm



We will characterize our functions f(N) by "erasing" certain information from the formula:

If P is a polynomial (possibly with log terms), then we will find a simpler polynomial Q, which is the leading (fastest-growing) term without its coefficient.

Then we say that "P is Big Oh of Q":

$$P = O(Q)$$

Examples

$$f(N) = 3*N + 7 = O(N)$$

$$f(N) = 2*N^2 - 3*N + 1$$
$$= O(N^2)$$

Analyzing the Time Complexity of an Algorithm

Computer Science

This techique leads to “complexity classes” of algorithms:

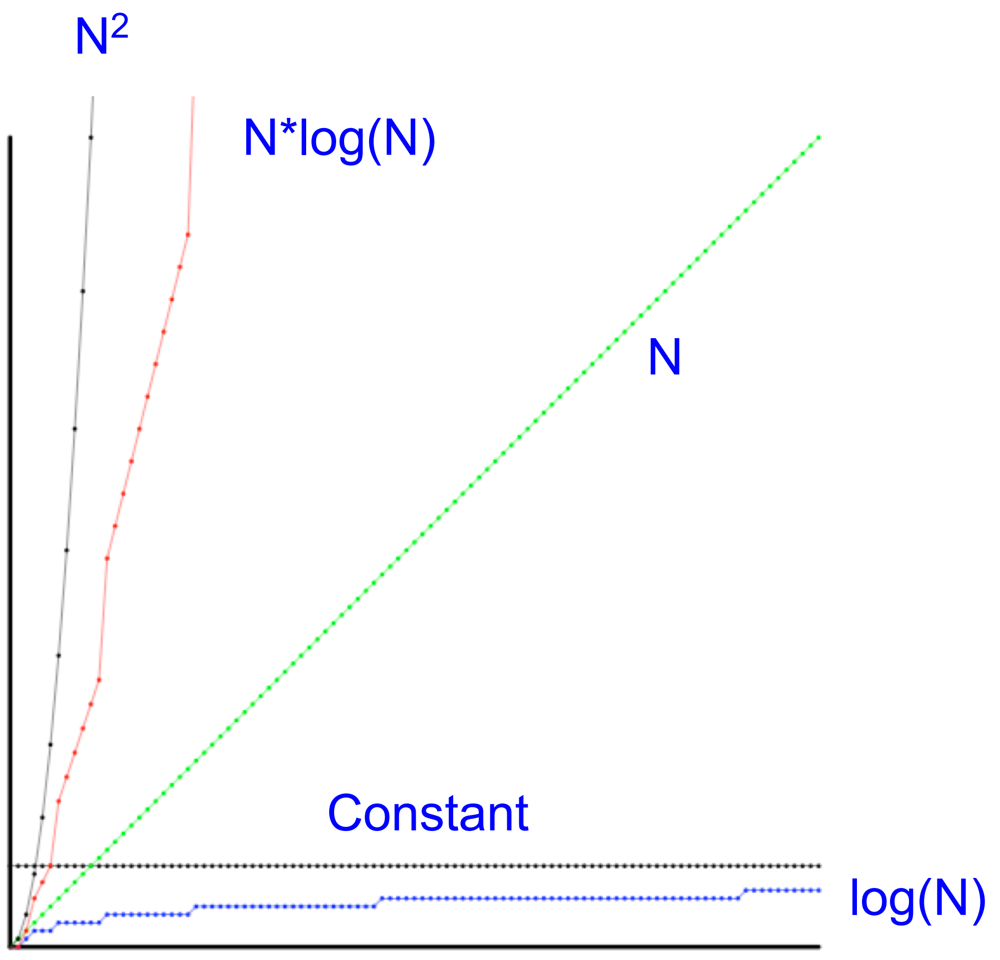
Name	Formula	Examples:
Constant Time:	$f(N) = O(1)$	Stack push, pop
Logarithmic Time:	$f(N) = O(\mathrm{Log}_2(N))$	Binary Search in Ordered List
Linear Time:	$f(N) = O(N)$	Sequential Search in Unordered List
Linearithmic Time:	$f(n) = O(N*\mathrm{Log}_2(N)$	Mergesort (we’ll look at this later)
Quadratic Time:	$f(N) = O(N^2)$	Selection Sort, Insertion Sort (later this lecture)
Cubic Time:	$f(N) = O(N^3)$	[All Pairs Shortest Path in Graph]
Exponential Time:	$f(N) = O(2^N)$	[Traveling Salesman Problem]

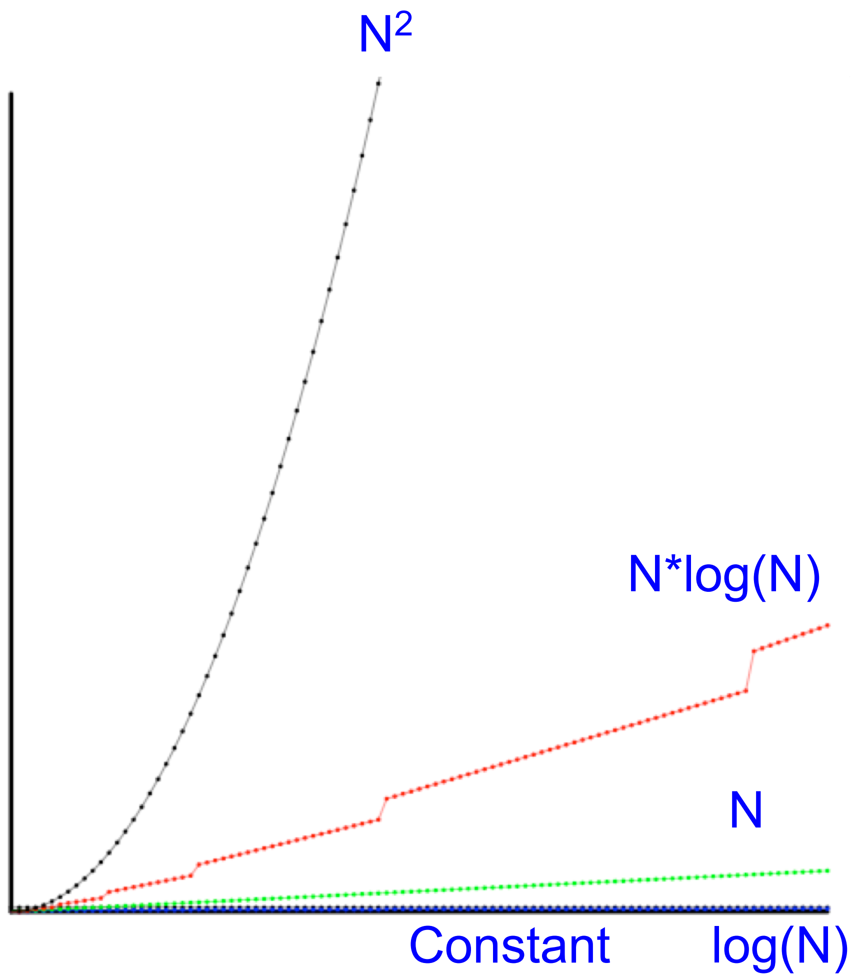
We will study examples of algorithms in this class of all but last two….

Analyzing the Time Complexity of an Algorithm

Computer Science

Each of these corresponds to a particular shape for f(N) for a given algorithm:





Analyzing the Time Complexity of an Algorithm

Computer Science

Important Note: Since we are interested only in an O(f(N)), we do not need to count every instruction, but only "basic operations" that are used at critical steps of the algorithm.

```
private static boolean member(int num, int[] A) {

    int lo = 0, hi = A.length - 1;
    while (lo <= hi) {

        int mid = (lo + hi) / 2;
        if(A[mid] == num) {
            return true;
        }
        else if(num < A[mid]) {
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return false;
}
```

To count the exact number of instructions executed to process an array of size A would be tedious and useless: we only want to know the leading term of f(N) without its constant, and this is just the number of times through the while loop:

O(f(N)) = number of iterations of while loop

= number of equality comparisons of two numbers

Thus, it is sufficient (and easier!) to just count the number of "basic operations" necessary to perform the algorithm.

Analyzing the Time Complexity of an Algorithm

To analyze an algorithm, then, you need to do the following:

Step One: Specify what "basic operations" you are counting

Example 1: When doing binary search, each stage involves comparing two numbers, so count the number of comparisons.

Example 2: When resizing an array, count the number of times you move a number.

Step Two: Decide whether you want to analyse the worst case, average case, or best case.

When not specified, the worst case is assumed. Best case is usually not interesting.

Step Three: Go through the algorithm, and count the number of basic operations, using the properties of O( ) to simplify the task......

Analyzing the Time Complexity of an Algorithm

Computer Science

You can add or multiply O(...) expressions, but simplify down to the leading term of the polynomial, without its coefficient:

$f(N) = 3*N^2 + 6*N - 1 = O(N^2)$ $g(N) = 2*N + 4 = O(N)$

$f(N) + g(N) = 3*N^2 + 8*N + 3 = O(N^2 + N) = O(N^2)$

$f(N) * g(N) = 6*N^3 + 24*N^2 + 20*N - 4 = O(N^3)$

In general:

$O(f(N) + C) = O(f(N))$

$O(C * f(N)) = O(f(N))$ = ignore all constants

O(f(N) + g(N)) = O(the largest leading term in f or g) = maximum

O(f(N) * g(N)) = O(the product of the leading terms of f and g) = multiply

Follow the leading term without constants and always throw out anything else....

Analyzing the Time Complexity of an Algorithm

Computer Science

This makes analysis of algorithms relatively easy: just pay attention to loops (and recursion) and consider everything else to be Constant Time:

`Examples....     Suppose we are counting comparisons:`

```
Code                                              Number of comparisons

int [] A = { 2, 3, 5, 4, 6 };                             0    (ignore any 0s)
int n = 0;                                                0

if(A[2] < A[4])                                           1     = O(1)
    ++n;
if(A[1] < A[4])                                           1     = O(1)
    ++n;
if(A[3] < A[2])                                           1     = O(1)
    ++n;
if(A[0] < A[3])                                           1     = O(1)
    ++n;

                                              Total: 4    =    O(1)
```

Analyzing the Time Complexity of an Algorithm

Computer Science

A series of loops is like adding O(...) espressions, and nested loops are like multiplying:

Code   Number of comparisons

```
for(int i = 0; i < N-2; ++i) {                   loop repeats N-2 times
  ...  A list of statements (no loops)...            O(1) for list of statements
}

for(int j = 0; j < N/2; ++i) {                   loop repeats N/2 times
  ...  A list of statements (no loops)...            O(1) for list of statements
}
```

Total: O(N) + O(N) = O(N)

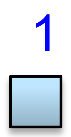
```
for(int i = 0; i < N; ++i) {                   loop repeats N times
  for(int j = 0; j < N; ++j) {                 loop repeats N times
     .... O(1) for list of statements ....
  }
}
```

Total: O(N^2)

Analyzing the Time Complexity of an Algorithm

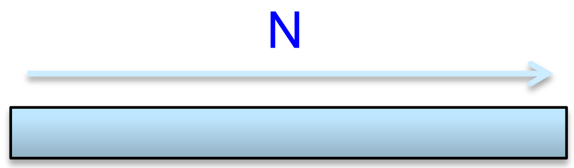
Graphical intuitions are often valuable, since we are abstracting away all but the basic "shape" of the way the algorithm uses the important operations. If you have N data items, the question is, how does the time complexity relate to N?

Constant Time:



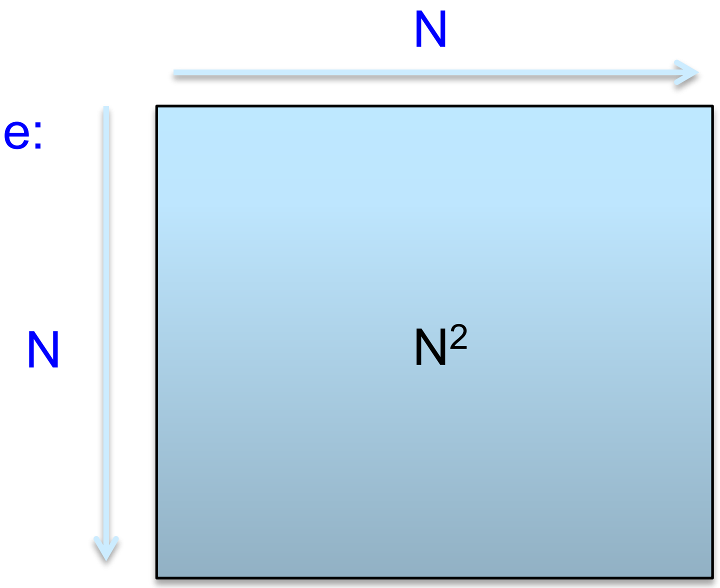
```
if( k < 10 ) { … O(1) …     }
```

Linear Time:



```
for(int i = 0; i < N; ++i) {
        …    O(1)  …
}
```

Quadratic Time:



```
for(int i = 0; i < N; ++i) {
    for(int j = 0; j < N; ++j) {
         .... O(1) …
    }
}
```

Analyzing the Time Complexity of an Algorithm

Computer Science

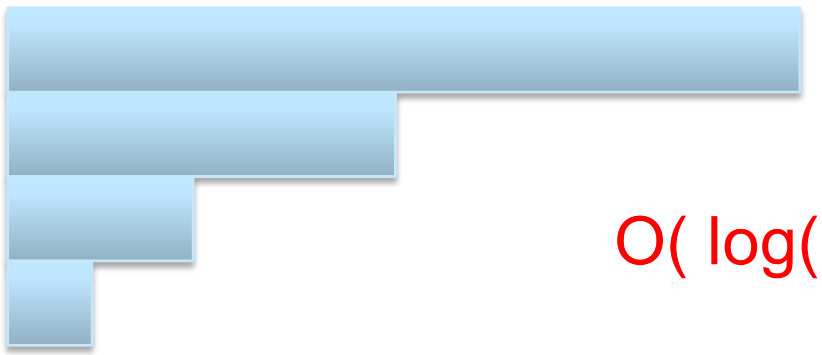
Logarithmic Time is almost always the result of dividing a problem (like searching an ordered list) in half repeatedly:

```java
private static boolean member(int num, int[] A, int lo, int hi) {

    if( lo > hi) {
        return false;
    }
    int mid = (lo + hi) / 2;
    if(A[mid] == num) {
        return true;
    }
    else if(num < A[mid]) {
        return member(num, A, lo, (mid - 1));
    } else {
        return member(num, A, (mid + 1), hi);
    }
}
```

```
member(60,A)
  2  5  7  9 12 15 18 19 20 25 26 31 32 35 38 42 45 54 57 59 60 64 73 75 78 83 85 88 94 96 99
                                                 45 54 57 59 60 64 73 75 78 83 85 88 94 96 99
                                                 45 54 57 59 60 64 73
                                                             60 64 73
                                                             60
true
```

The question is: How many times can we divide N in half before we get ≤ 1 ?



O(log(N))

Suppose N = 2^k , i.e., k = log(N)

N / 1	N / 2^0
N / 2	N / 2^1
N / 4	N / 2^2
…	
N / (N/2)	N / $2^{(k-1)}$
N / N	N / 2^k

Sorting: Basic Ideas

Computer Science

For any set A and total ordering < on A, we can define the **SORTING PROBLEM** as follows:

Input: A sequence S = (a_0, a_1, a_2, ..., a_{n-1}) of elements from A

Output: A permutation (rearrangement) (a_0', a_1', a_2', ..., a_{n-1}') of S such that

$$a_0' \leq a_1' \leq a_2' \leq \ldots\ldots \leq a_{n-1}'$$

Notes:

(1) If there are no duplicates (our usual assumption) we can say:

$$a_0' < a_1' < a_2' < \ldots\ldots < a_{n-1}'$$

(2) We will ONLY consider the problem of sorting arrays (and use arrays of integers for our illustrations).

Sorting and Time Complexity: Basic Ideas

We will study two iterative methods:

Selection Sort

Insertion Sort

and a recursive method

Merge Sort

and consider their characteristics:

- Overall approach to the problem (iterative vs. recursive);
- Use of memory (can it be done “in place” in the same array the data is stored in, or do you needan auxiliary array?);
- Time complexity (in terms of the number of comparisons);

Sorting: Iterative Sorting Methods

Computer Science

Iterative methods generally have the following characteristics:

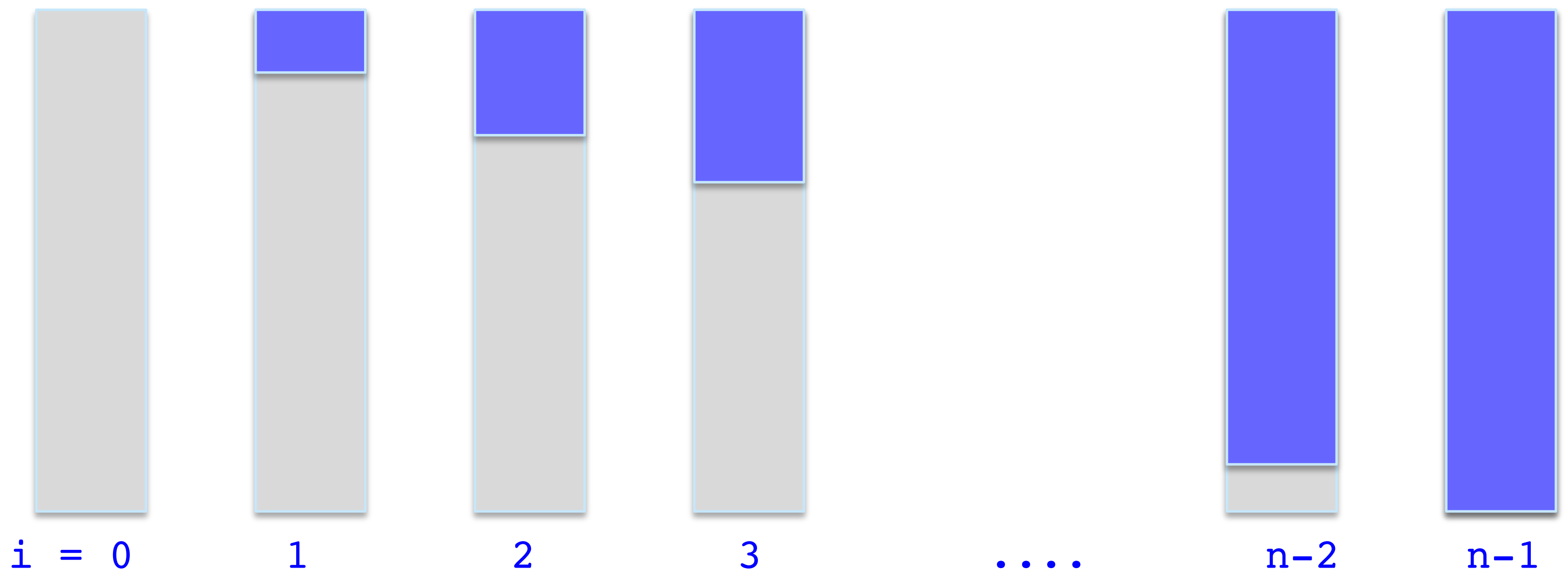
They sort an array A[0..N-1] in place (no extra storage needed);

They have two for loops;

The outer for loop steps through the array

```
for( int i = 0; i < N; ++i ) { ....  }
```

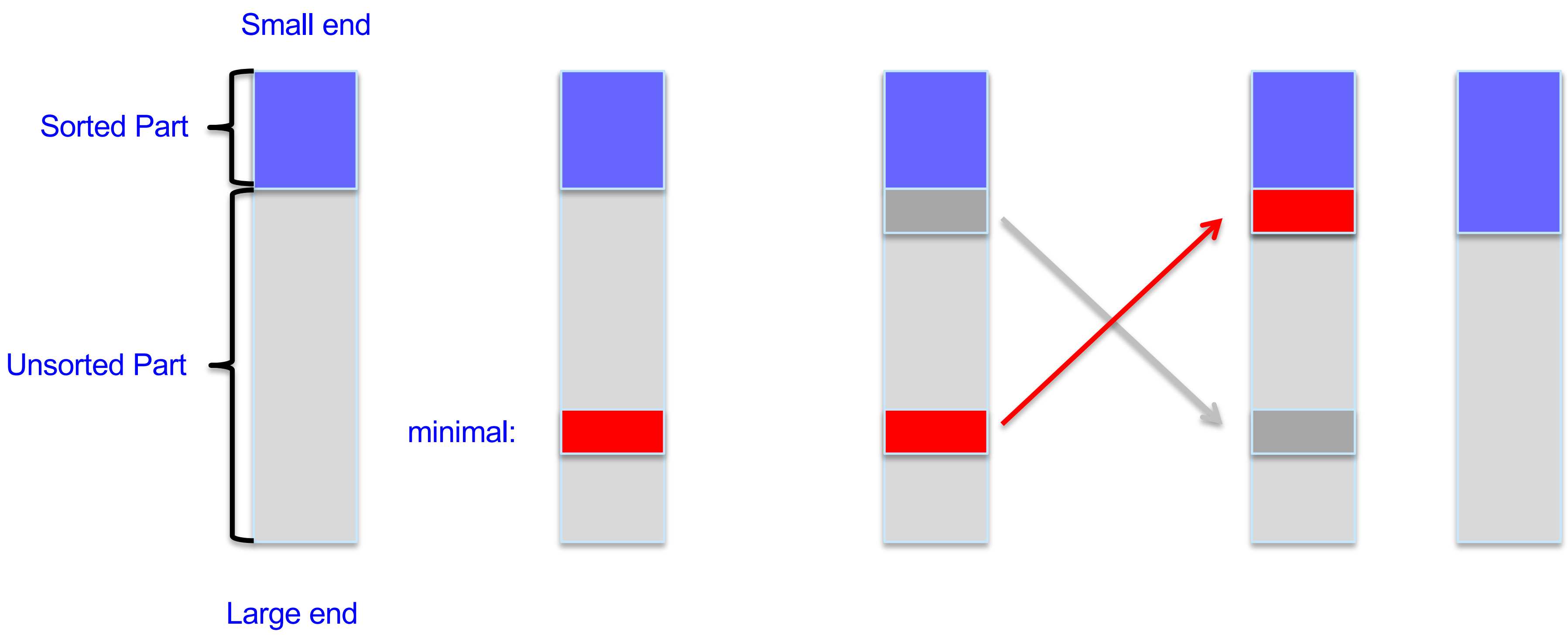
and successively turns an unordered list into an ordered list:



Sorting: Iterative Sorting Methods

The only **essential difference** between our two iterative methods is **how they extend the sorted region of the array:**

Selection Sort looks for a minimal (if no duplicates, the smallest) element in the unsorted part, and moves it to the bottom of the sorted part; this is done using `swap(....)`



Sorting: Selection Sort

```
// Selection Sort (from Algorithms by Sedgewick and Wayne)

 public static void selectionSort(int [] A) {
     int N = A.length;
     for (int i = 0; i < N-1; i++) {
         int min = i;                       // assume min is A[i]
         for (int j = i+1; j < N; j++) { // look for something smaller
           if ( less(A[j], A[min]) )     // found one!
             min = j;
         }
         swap(A, i, min);                   // swap min key and top of unsorted part
     }
}

public static boolean less( int v, int w ) {
     return ( v < w );
}

public swap( int[] A, int i, int j) {             // swap A[i] and A[j]
     int temp = A[i]; A[i] = A[j]; A[j] = temp;
}
```

Time Complexity of Selection Sort

So, what is the **Time Complexity** of Selection Sort? Let us count the number of comparisons of integers, which means counting the number of times less(….) is executed (which is why we put it in a separate method).

```
public static void selectionSort(int [] A) {
     int N = A.length;
     for (int i = 0; i < N-1; i++) {
         int min = i;                        // assume min is A[i]
         for (int j = i+1; j < N; j++) { // look for something smaller
           if ( less(A[j], A[min]) )     // found one!
             min = j;
         }
         swap(A, i, min);
     }
}
public static boolean less( int v, int w ) {
     // could put a counter here an increment each time less is called
     return ( v < w );
}
```

Complexity of Selection Sort

Let's **count** the number of **comparisons** (less):

Observe that the outer loop runs N-1 times

and less() is called once each time through the inner loop....

(N-1) times, then (N-2)....

```
public static void selectionSort(Comparable [] a) {
     int N = a.length;
     for (int i = 0; i < N-1; i++) {
         int min = i;                           // assume min is A[i]
         for (int j = i+1; j < N; j++) { // look for something smaller
           if ( less(a[j], a[min]) )     // found one!
             min = j;
         }
         swap(a, i, min);                       // swap min and top of
     }                                          //   unsorted part
}
```

Complexity of Selection Sort

We can illustrate this as follows: Let's color the slot where the second argument to less occurs in green, and color the sorted part of the list in red:

7	1	1	1	1
2	2	2	2	2
1	7	7	5	5
8	8	8	8	7
5	5	5	7	8

4 + 3 + 2 + 1 = 10 calls to less(...)

In general, for an array of size N, we have

$(N\text{-}1) + (N\text{-}2) + \ldots\ldots + 2 + 1 = N(N\text{-}1)/2 = N^2/2 - N/2$ calls to less(....)

When N = 5, we have $5^2/2 - 5/2 = 25/2 - 5/2 = 20/2 = 10$.

Complexity of Selection Sort

Computer Science

Let's **count** the number of **comparisons** (less):

Observe that the outer loop runs N times

and less() is called once each time through the inner loop: $(N-1) + ... + 1 = N(N-1)/2$

$= N^2/2 – N/2$

Total: $N^2/2 - N/2$

$= O(N^2)$

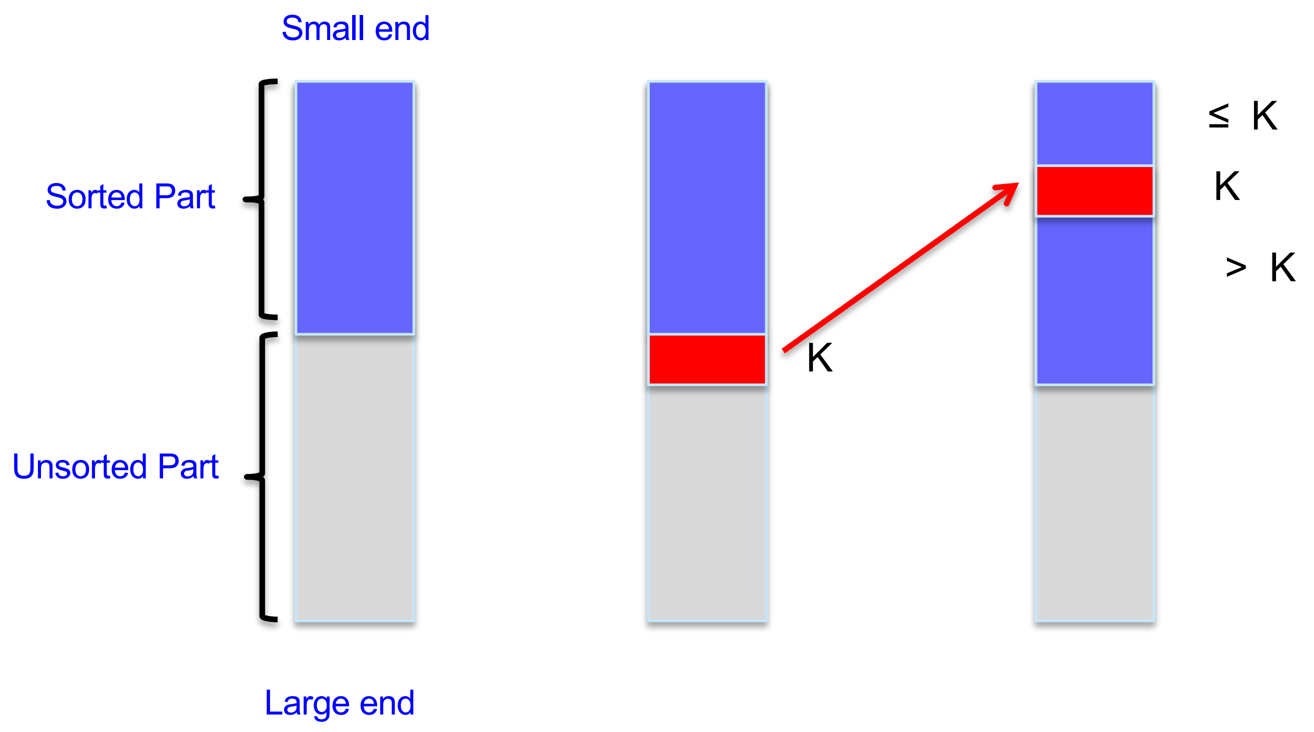
```
public static void selectionSort(Comparable [] a) {
    int N = a.length;
      for (int i = 0; i < N-1; i++) {
        int min = i;
          for (int j = i+1; j < N; j++) {
            if ( less(a[j], a[min]) )
              min = j;
          }
          swap(a, i, min);                 // swap min and top of
      }                                    //   unsorted part
}
```

Sorting: Insertion Sort

Computer Science

The only **essential difference** between our two iterative methods is **how they extend the sorted region of the array:**

Insertion Sort picks the top element of the unsorted part, and finds the place where it belongs in the sorted part, and inserts it there:

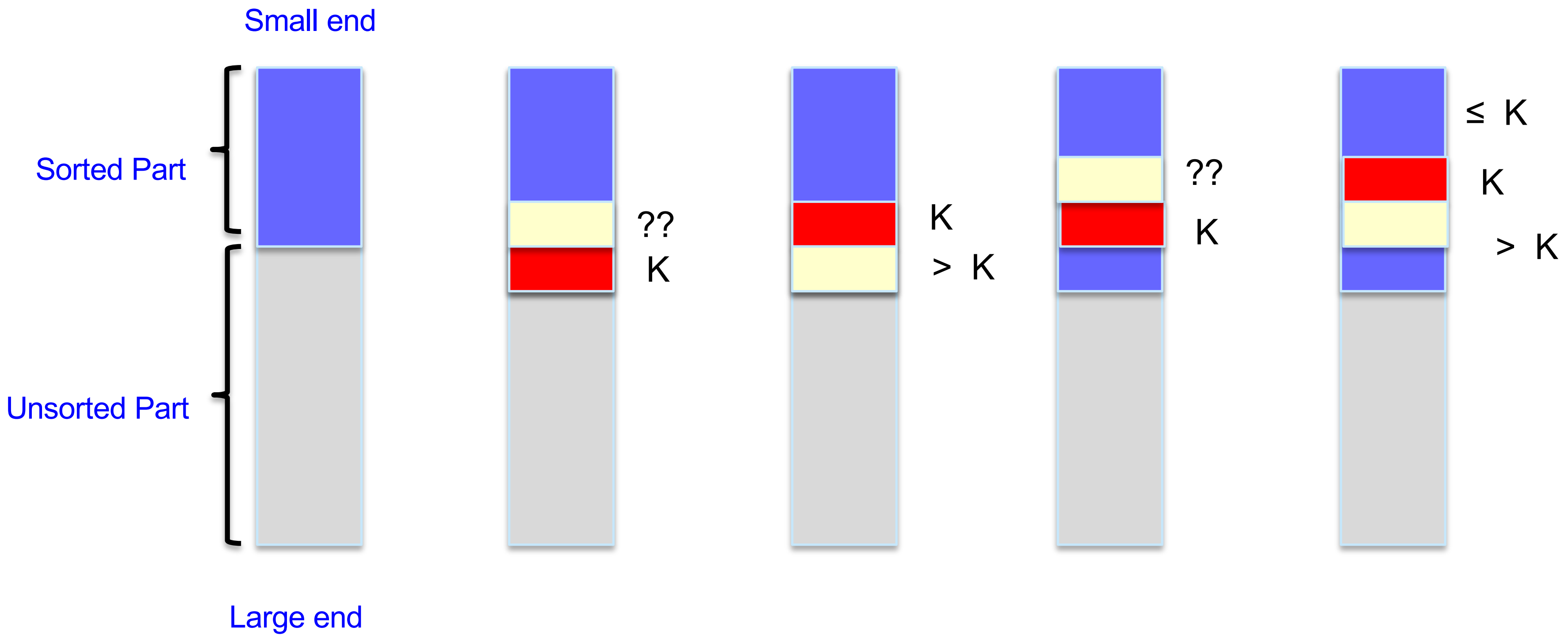


Sorting: Insertion Sort

Computer Science

The only **essential difference** between our two iterative methods is **how they extend the sorted region of the array:**

Insertion Sort picks the top element of the unsorted part, and finds the place where it belongs in the sorted part, and inserts it there; it actually does this by exchanging K with its upper neighbor if that neighbor is greater:



Sorting: Insertion Sort

Let's **count** the number of **calls to less(...)** in worst case, which in fact is a reverse sorted list....

Observe that the outer loop runs N-1 times, and less is called

1 time, then 2 times, ... then (N-2) times, then finally (N-1) times.

```
public static void insertionSort(int[] a) {
  int N = a.length;
    for (int i = 1; i < N; i++) {
      for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
        swap(a, j, j-1);
      }
    }
}
```

Complexity of Insertion Sort

Now let’s look at the diagram, coloring a slot blue if it was compared with the new key being inserted:

8	7	5	2	1
7	8	7	3	2
5	5	8	7	5
2	2	2	8	7
1	1	1	1	8

1 + 2 + 3 + 4 = 10 calls to less(...)

It is the same as for Selection Sort: $N^2/2 - N/2$ calls to less(....)

This is for a reverse sorted list! What about an already sorted list?

Complexity of Insertion Sort

Computer Science

For an already sorted list, Insertion Sort does something very smart: it just checks to see that each key is not less than the one above it, and doesn't go any further!

1	1	1	1	1
2	2	2	2	2
5	5	5	5	5
7	7	7	7	7
8	8	8	8	8

1 + 1 + 1 + 1 = 4 calls to less(...)

In this case, Insertion Sort checks that the list is already sorted in N-1 calls to less(...) = O(N)

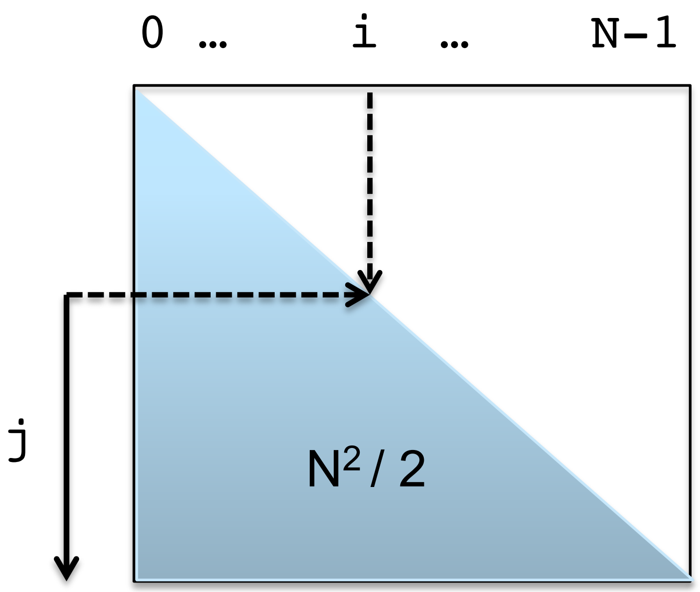
Punchline: Insertion Sort adapts to its input, and does less work than Selection Sort, except in the case of a reverse-sorted list, where they both do the same!

Conclusions on Complexity of Iterative Sorts

So it is easy to see why Selection sort (in all cases) is O(N^2) :

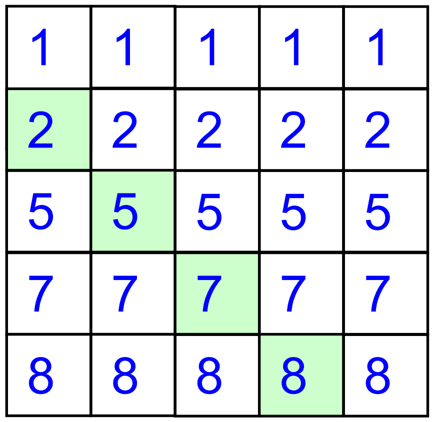
```
for (int i = 0; i < N; i++) {
   for (int j = i+1; j < N; j++) {
             ….   O(1)   ….
   }
}
```

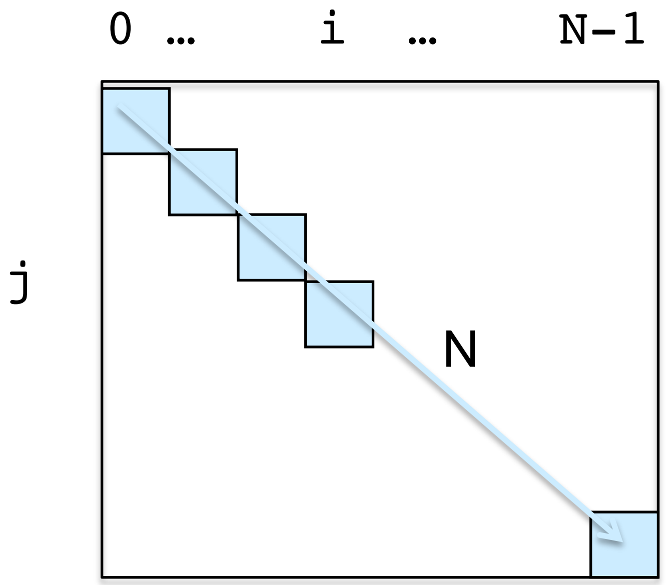
7	1	1	1	1
2	2	2	2	2
1	7	7	5	5
8	8	8	8	7
5	5	5	7	8



Conclusions on Complexity of Iterative Sorts

For the best case of Insertion Sort, we only do one comparison per loop to check that the given item is already in the correct place,   so it is O(N)



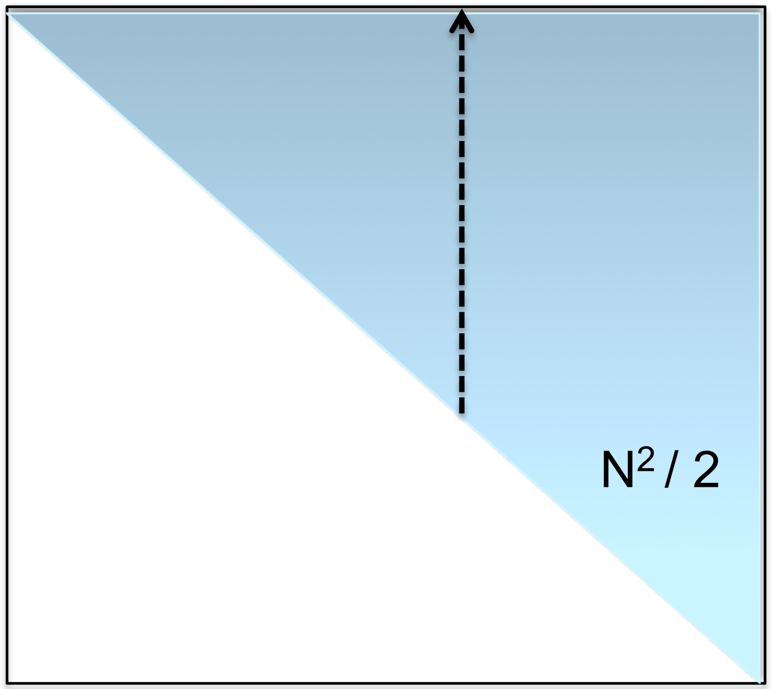


Conclusions on Complexity of Iterative Sorts

For the worst case of Insertion Sort, observe that the worst thing that can happen is each number we insert is the smallest we have seen so far:

```
insert  1     into    13  10  9  7  6  4  3  2
```

```
So at each step of the outer loop, the new item goes all the
way up to the top:
```



$N^2/2 - N/2 == O(N^2)$

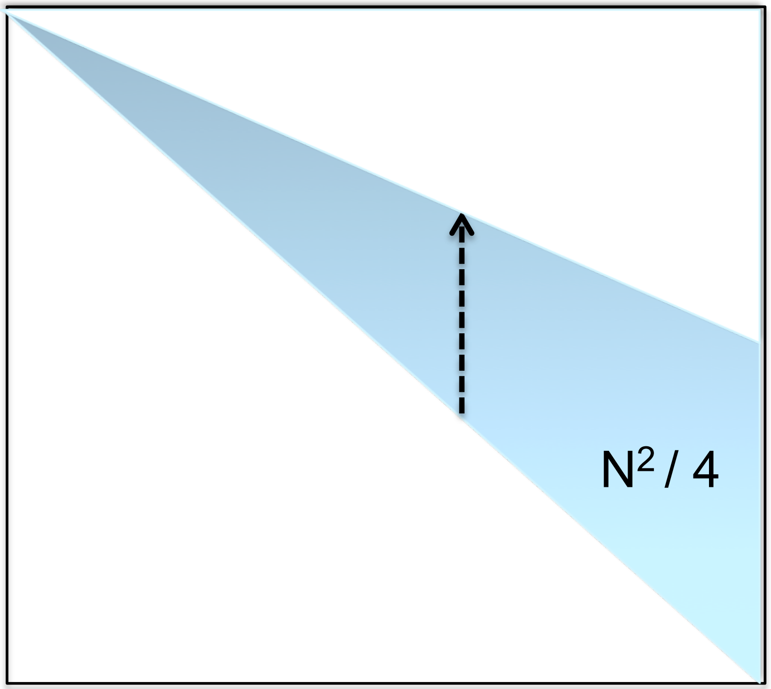
Conclusions on Complexity of Iterative Sorts

For the average case of Insertion Sort, observe that when we insert an *arbitrary* number into a ordered list, on average we go half way up:

```
insert  k into    13  10  9  7  6  4  3  2
```

$\xrightarrow{\hspace{4em}}$

`So at each step of the outer loop, the new item goes half way up to the top:`



$N^2/4 – N/2 = = O(N^2)$

Iterative Sorting: Conclusions on Time Complexity

Computer Science

Algorithm	Worst-case Input	Worst-case Time	Best-case Input	Best-case Time	Average-case Input	Average-case Time
Selection Sort	Any!	O(N^2)	Any!	O(N^2)	Any!	O(N^2)
Insertion Sort	Reverse Sorted List	O(N^2)	Already Sorted List	O(N)	Random List	O(N^2)

Conclusions:

- **Selection Sort** is **inflexible** and does O(N^2) comparisons in all cases;
- **Insertion Sort** in the worst case does no better than Selection Sort, but **adapts to its input**: it performs better the “more sorted” the input it; in the case of an already sorted list, it simply checks that the list is sorted.