CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Asymptotic complexity of algorithms
- The Sorting Problem
- Iterative sorting: Selection sort and Insertion sort

Next Time:

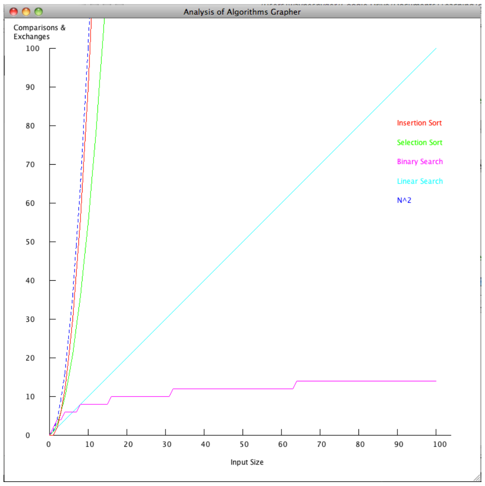
- Recursive sorting: Mergesort and Quicksort

Computer Science

Algorithm Analysis

Computer Science

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: How long does the algorithm take to finish?



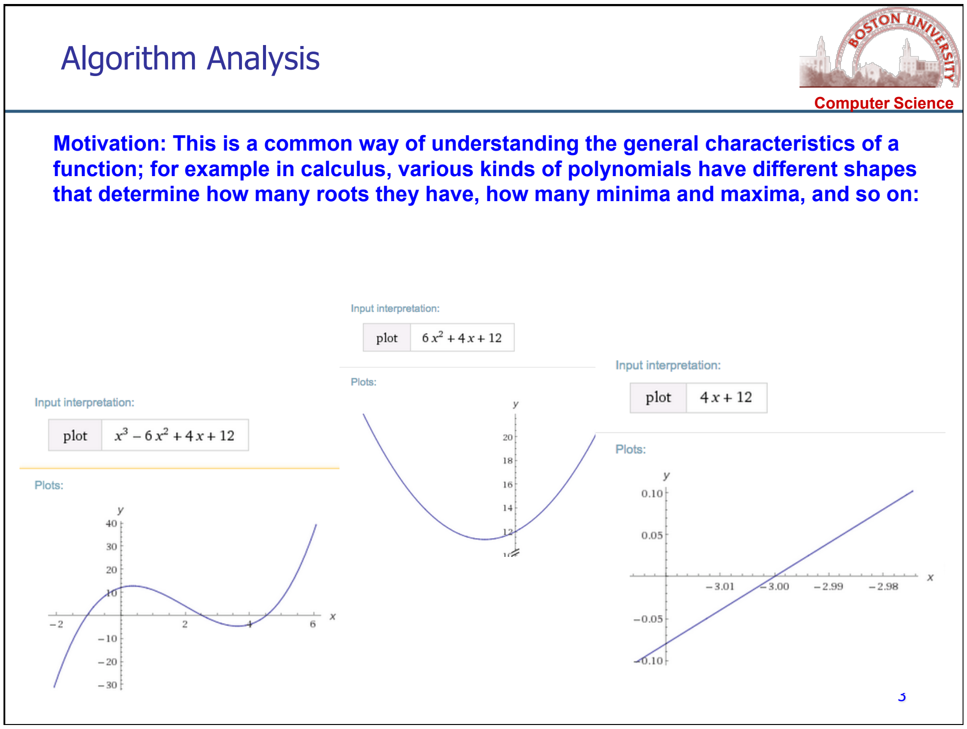
We express this as a function

f(N) = # steps to process an input of size N

But in the first place we only want to know the approximate “shape” of the function, so we count, not every step, but only “important steps” without which we can not do the algorithm.

E.g., in searching, we count the number of times we compare two keys.

2

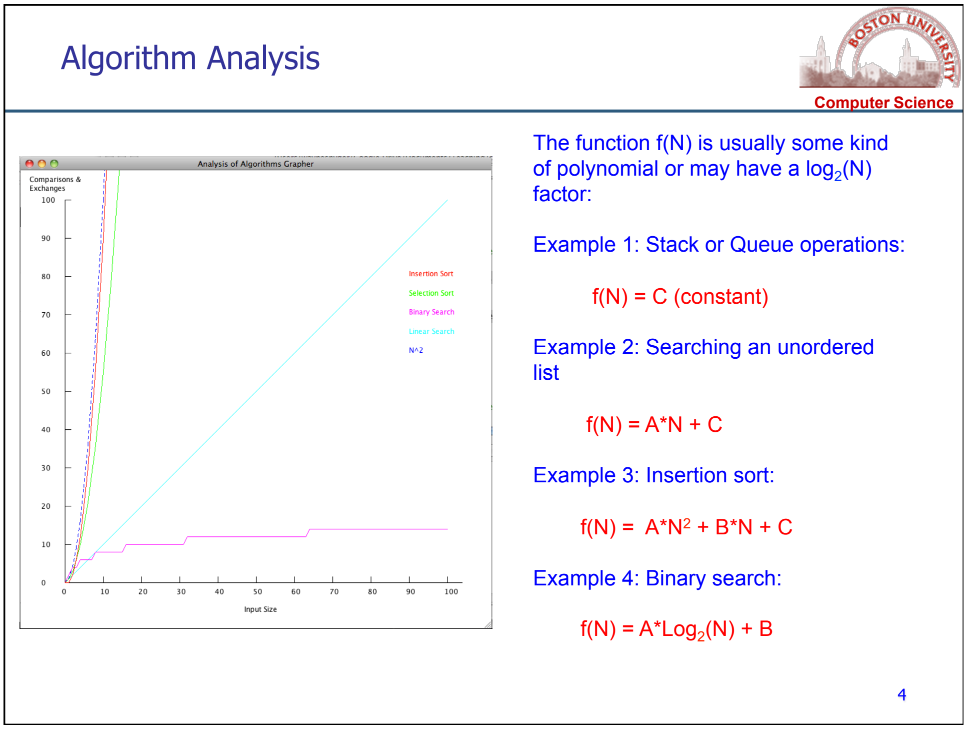


Algorithm Analysis

Computer Science

Motivation: This is a common way of understanding the general characteristics of a function; for example in calculus, various kinds of polynomials have different shapes that determine how many roots they have, how many minima and maxima, and so on:

3



Algorithm Analysis

Computer Science

The function f(N) is usually some kind of polynomial or may have a $\log_2(N)$ factor:

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:

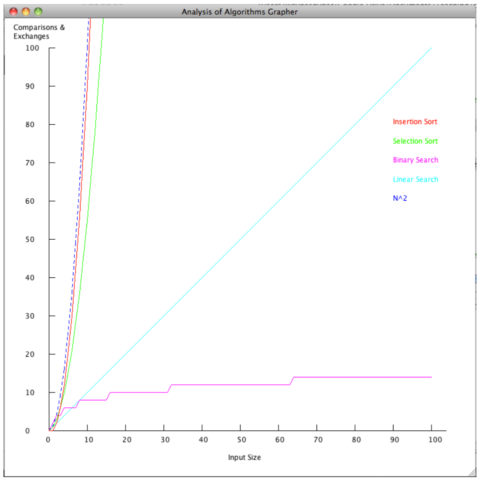
f(N) = A*N^2 + B*N + C

Example 4: Binary search:

f(N) = A*Log_2(N) + B

4

Algorithm Analysis



And we use various ways of approximating these functions, so we generally don't need so many details:

If P is a polynomial (possibly with log terms added), then

P = ~T where T is the leading term of P (or you can write "P ~ T")

P = Θ(Q) where Q is the leading term of P without its coefficient

Examples

$$f(N) = A*N + C = \sim(A*N) = \Theta(N)$$

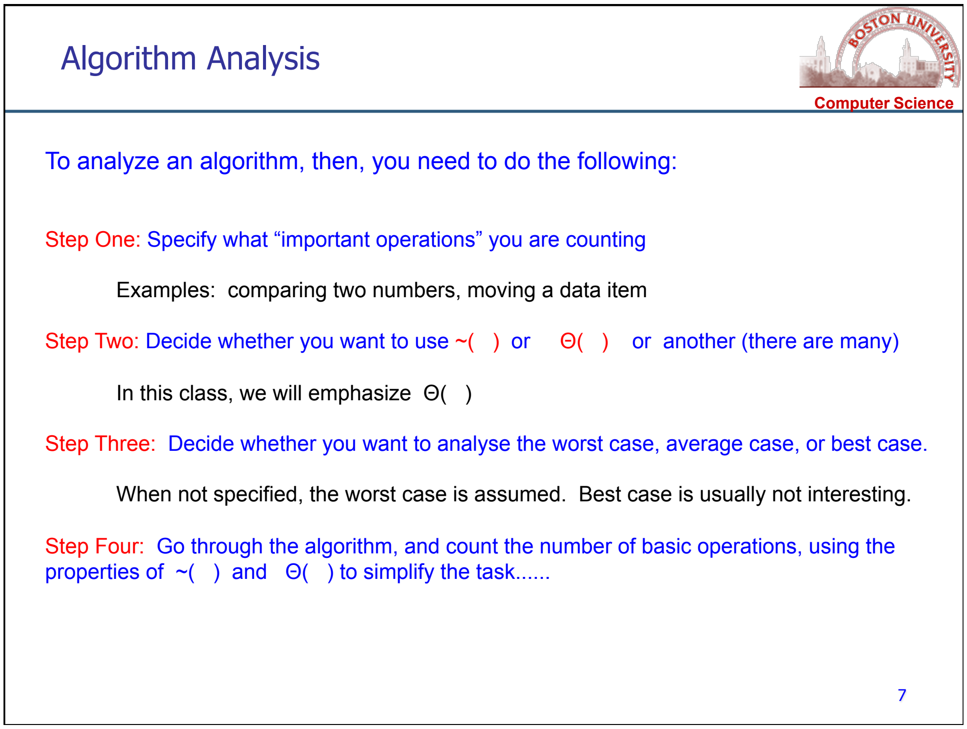
$$f(N) = A*N^2 + B*N + C = \sim A*N^2 = \Theta(N^2)$$

Algorithm Analysis

We categorize these by the degree of the leading term:

Name	Formula	Examples:
Constant Time:	f(N) = ~(C) = Θ(1)	Stack push, pop
Logarithmic Time:	f(N) = ~(A*Log$_2$(N) = Θ(Log$_2$(N))	Binary Search in Ordered List
Linear Time:	f(N) = ~(A*N) = Θ(N)	Sequential Search in Unordered List
Linearithmic Time:	f(n) = ~(A*N*Log$_2$(N)) = Θ(N*Log$_2$(N))	Mergesort (later in the lecture)
Quadratic Time:	f(N) = ~(A*N^2) = Θ(N^2)	Selection Sort, Insertion Sort
Cubic Time:	f(N) = ~(A*N^3) = Θ(N^3)	All Pairs Shortest Path in Graph
Exponential Time:	f(N) = Θ(2^N)	Traveling Salesman Problem

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



Algorithm Analysis

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

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

Examples: comparing two numbers, moving a data item

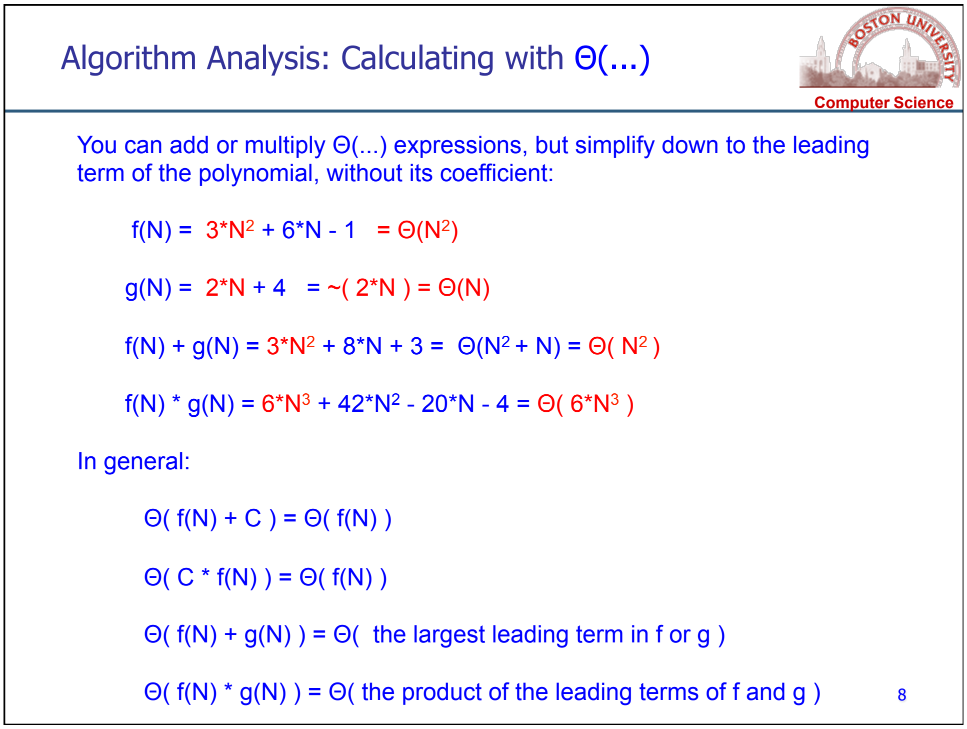
Step Two: Decide whether you want to use ~( ) or Θ( ) or another (there are many)

In this class, we will emphasize Θ( )

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



Algorithm Analysis: Calculating with Θ(...)

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

$f(N) = 3*N^2 + 6*N - 1 = \Theta(N^2)$

$g(N) = 2*N + 4 = \sim(2*N) = \Theta(N)$

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

$f(N) * g(N) = 6*N^3 + 42*N^2 - 20*N - 4 = \Theta(6*N^3)$

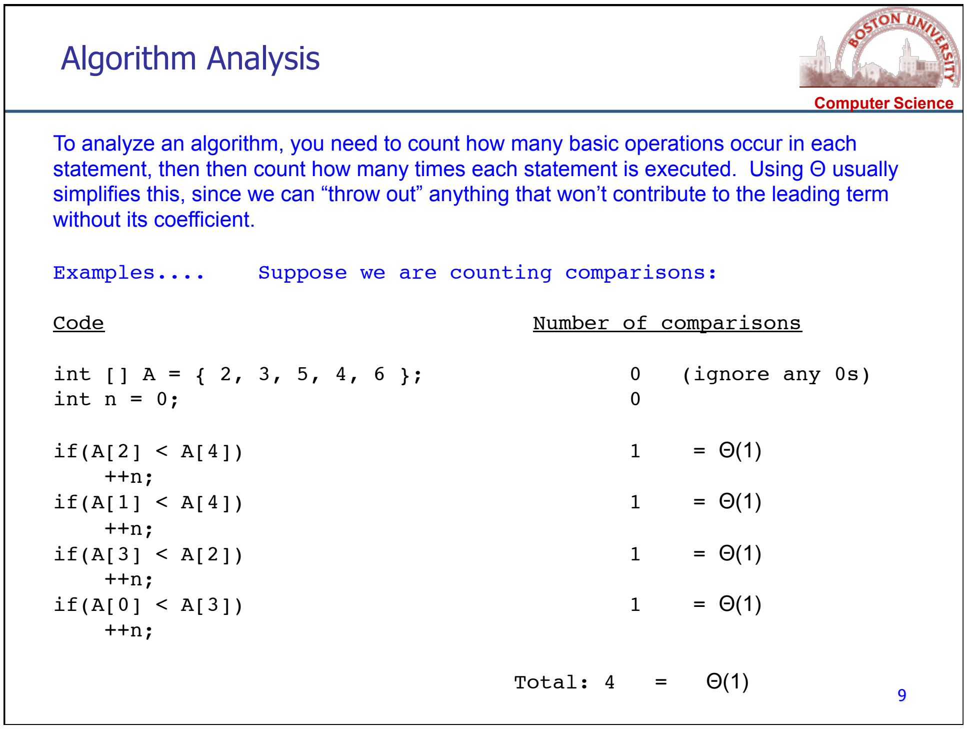
In general:

$\Theta(f(N) + C) = \Theta(f(N))$

$\Theta(C * f(N)) = \Theta(f(N))$

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

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



Algorithm Analysis

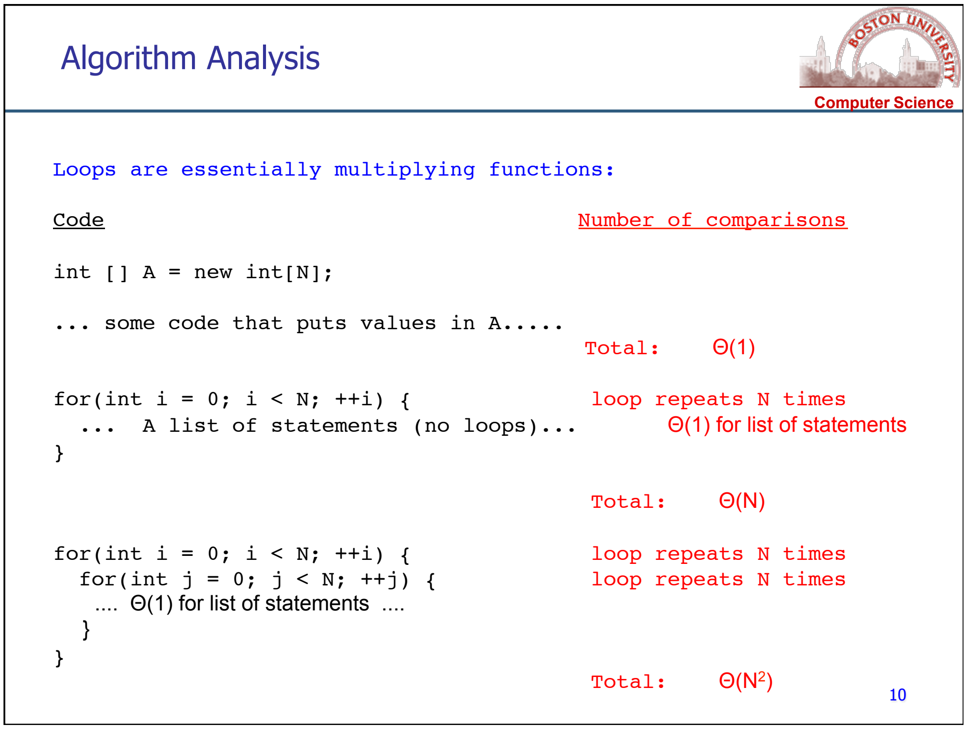
Computer Science

To analyze an algorithm, you need to count how many basic operations occur in each statement, then then count how many times each statement is executed. Using Θ usually simplifies this, since we can “throw out” anything that won’t contribute to the leading term without its coefficient.

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

                                    Total: 4    =    Θ(1)
```

9



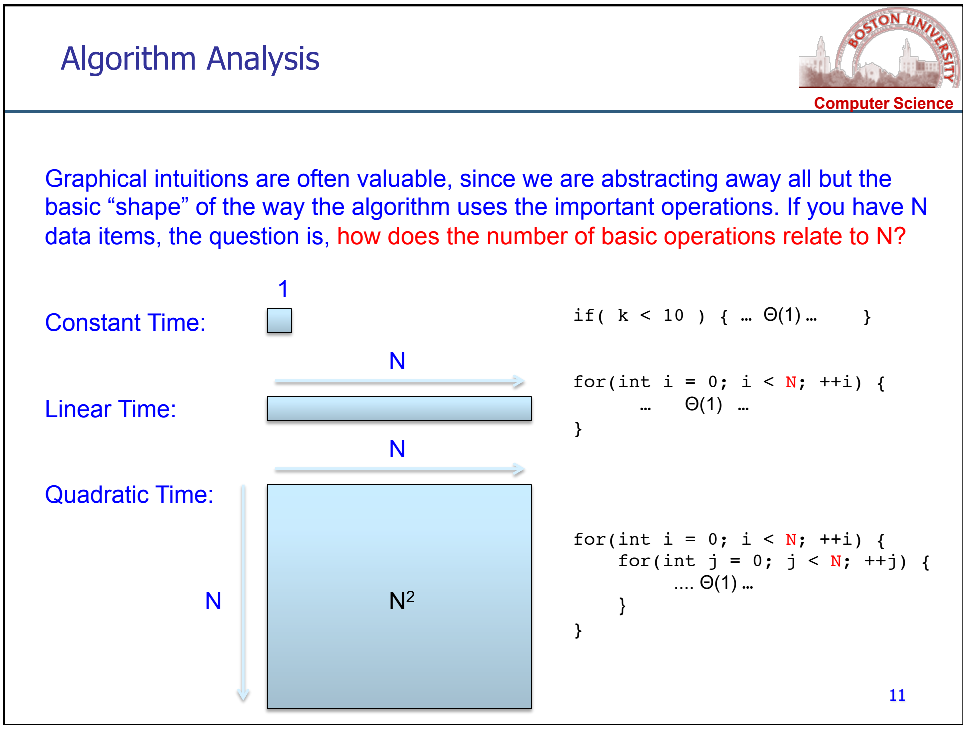
Algorithm Analysis

Computer Science

Loops are essentially multiplying functions:

```
Code                                      Number of comparisons

int [] A = new int[N];

... some code that puts values in A.....
                                           Total:    Θ(1)

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

                                           Total:    Θ(N)

for(int i = 0; i < N; ++i) {               loop repeats N times
  for(int j = 0; j < N; ++j) {             loop repeats N times
    .... Θ(1) for list of statements ....
  }
}
                                           Total:    Θ(N^2)
```

10



Algorithm Analysis

Computer Science

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 number of basic operations relate to N?

Constant Time:

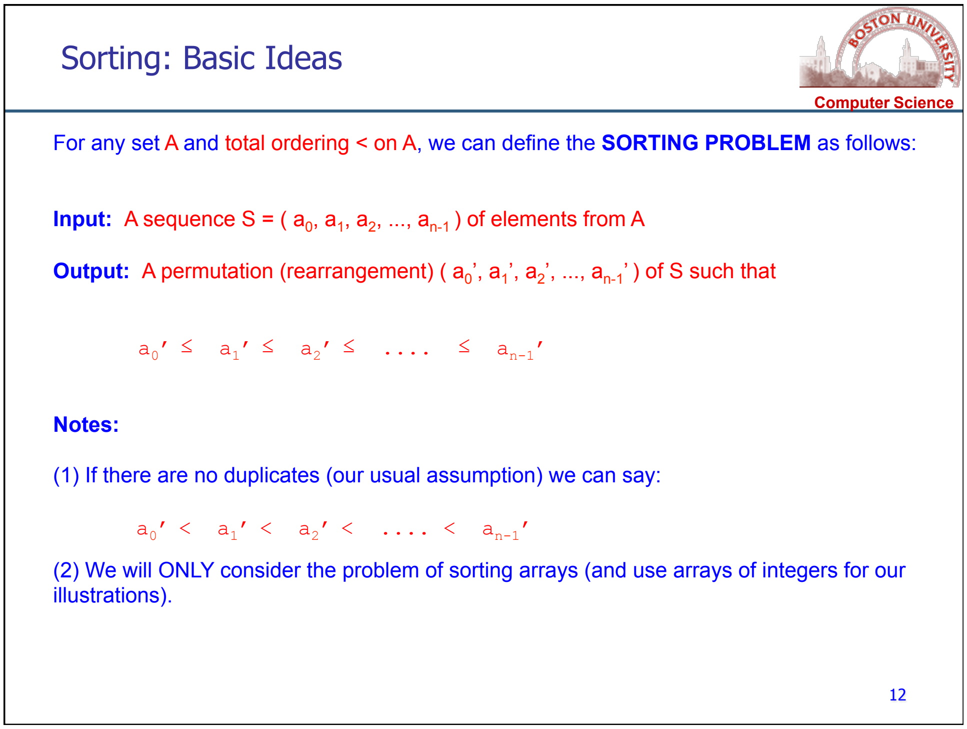
```
if( k < 10 ) { … Θ(1) …     }
```

Linear Time:

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

Quadratic Time:

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



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

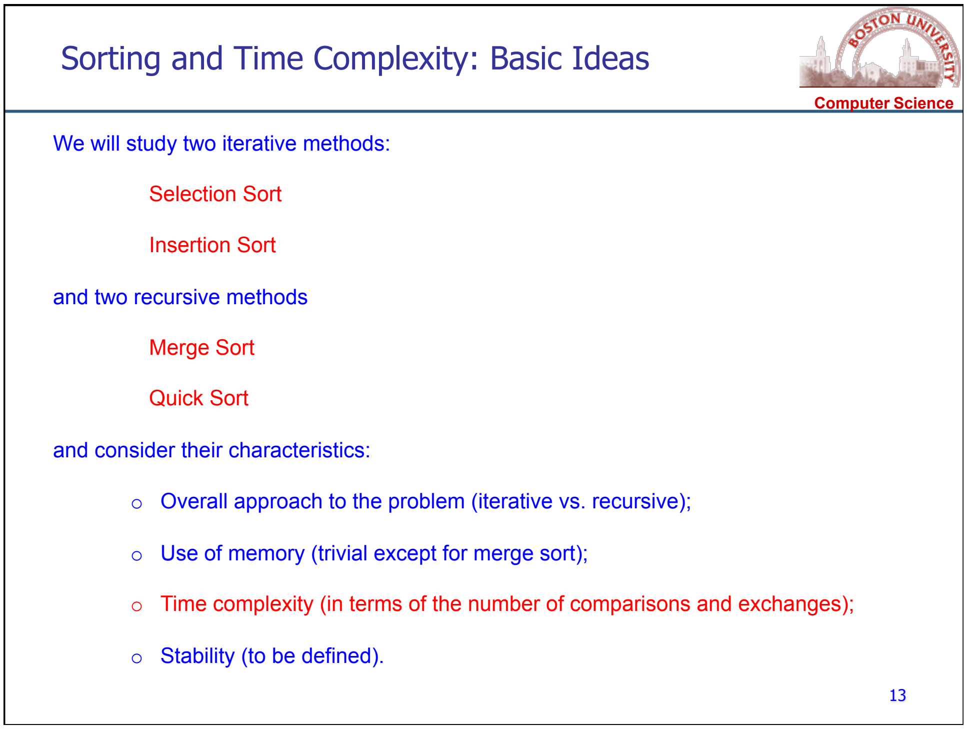
$$a_0' \leq a_1' \leq a_2' \leq \ldots. \leq a_{n-1}'$$

Notes:

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

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

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



Sorting and Time Complexity: Basic Ideas

Computer Science

We will study two iterative methods:

Selection Sort

Insertion Sort

and two recursive methods

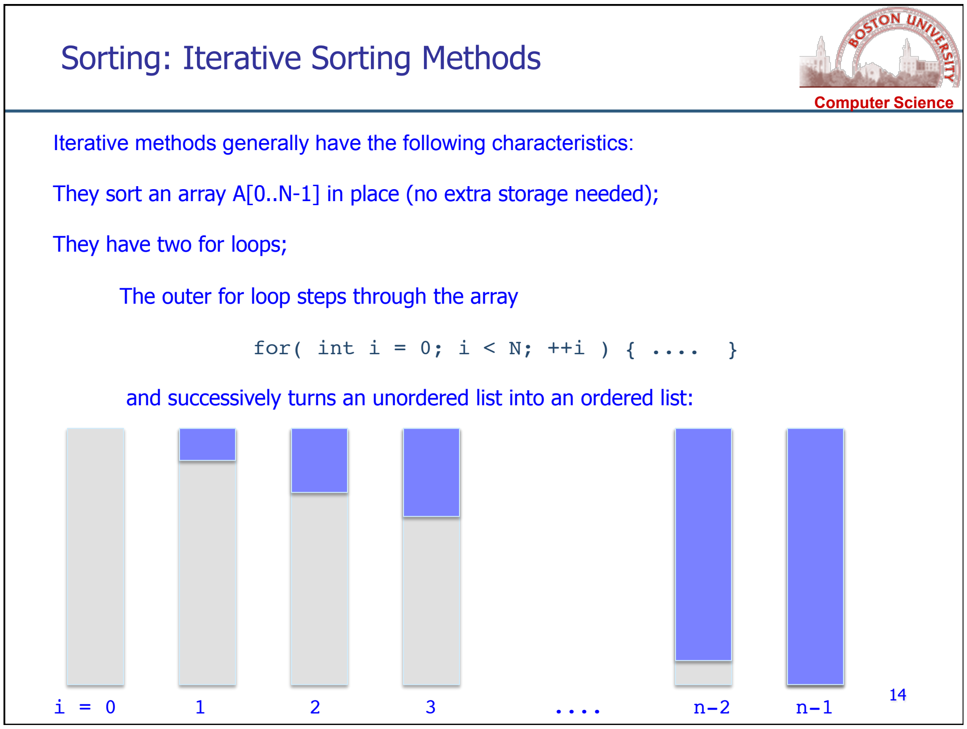
Merge Sort

Quick Sort

and consider their characteristics:

- ○ Overall approach to the problem (iterative vs. recursive);
- ○ Use of memory (trivial except for merge sort);
- ○ Time complexity (in terms of the number of comparisons and exchanges);
- ○ Stability (to be defined).

13



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);

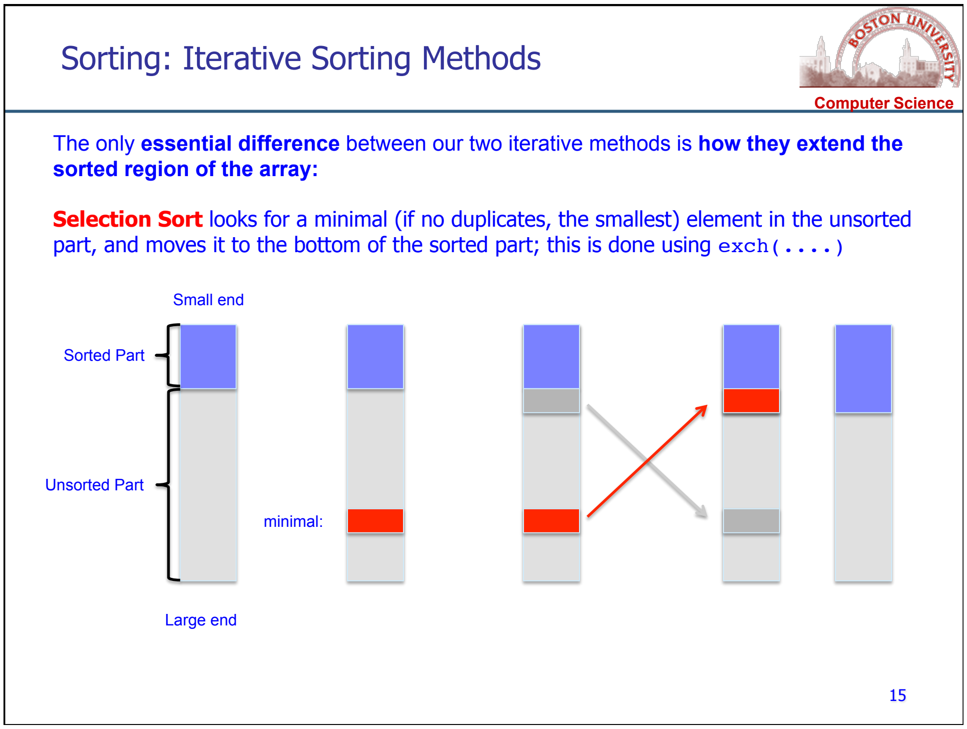
They have two for loops;

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:

14

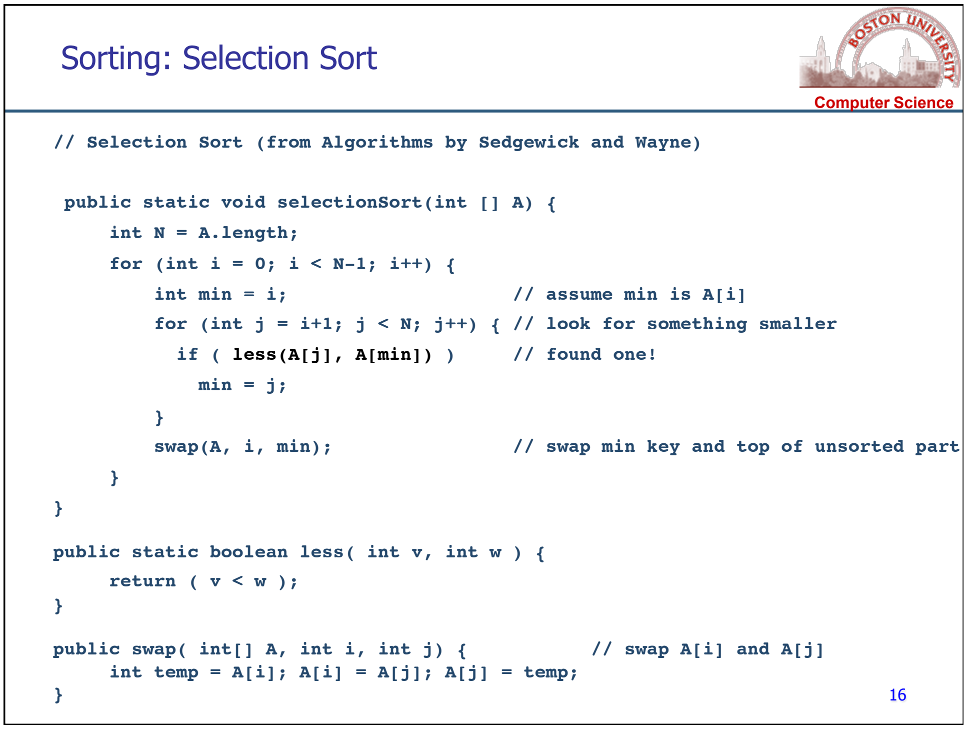


Sorting: Iterative Sorting Methods

Computer Science

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

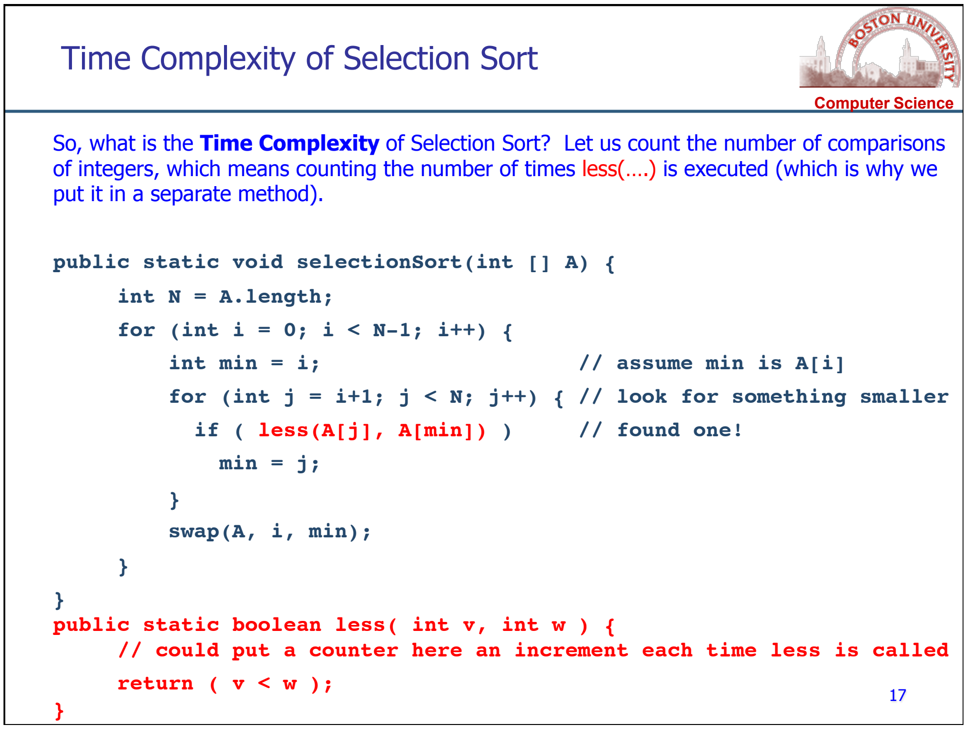
Selection Sort looks for a minimal (if no duplicates, the smallest) element in the unsorted part, and moves it to the bottom of the sorted part; this is done using `exch(....)`



Sorting: Selection Sort

Computer Science

```
// 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;
}
```

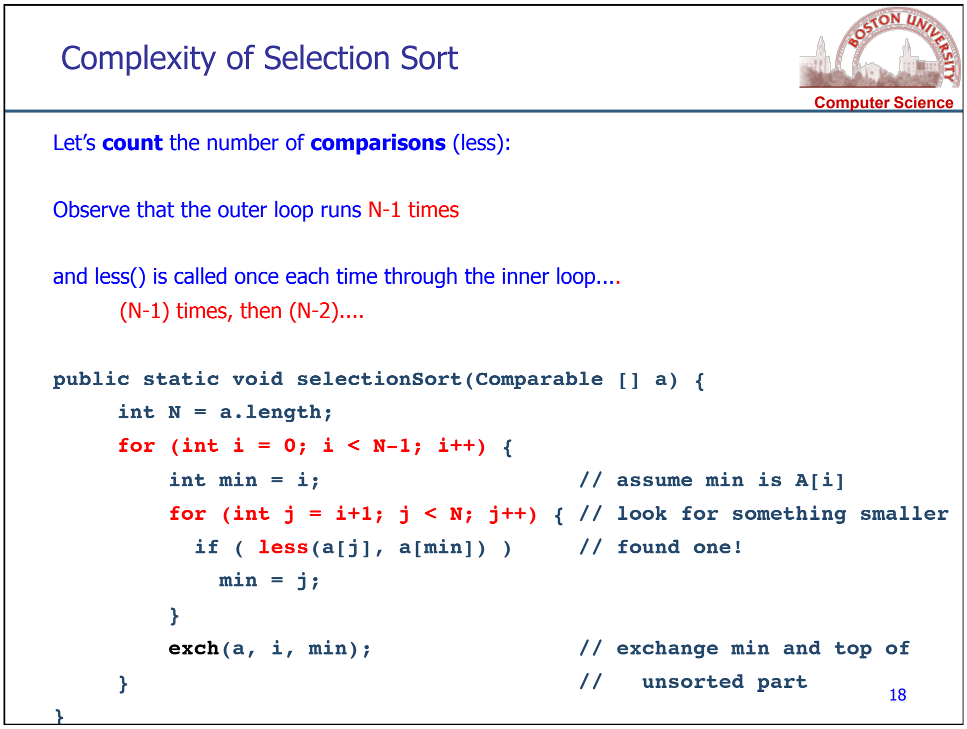


Time Complexity of Selection Sort

Computer Science

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

Computer Science

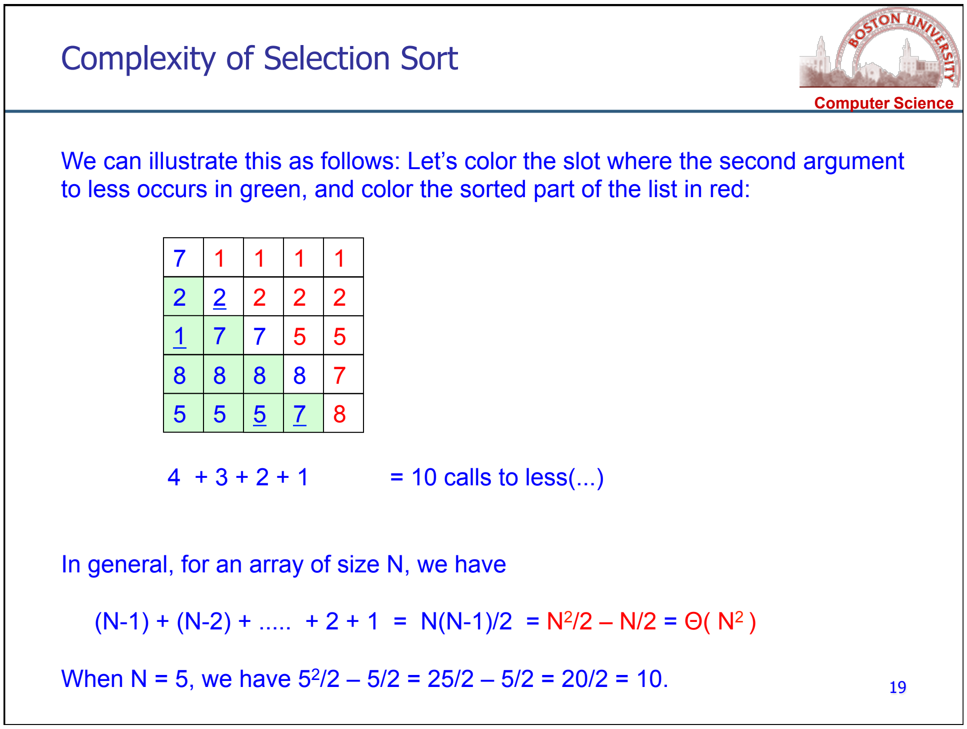
Let's **count** the number of **comparisons** (less):

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;
         }
         exch(a, i, min);                    // exchange min and top of
     }                                       //   unsorted part
}
```



Complexity of Selection Sort

Computer Science

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:

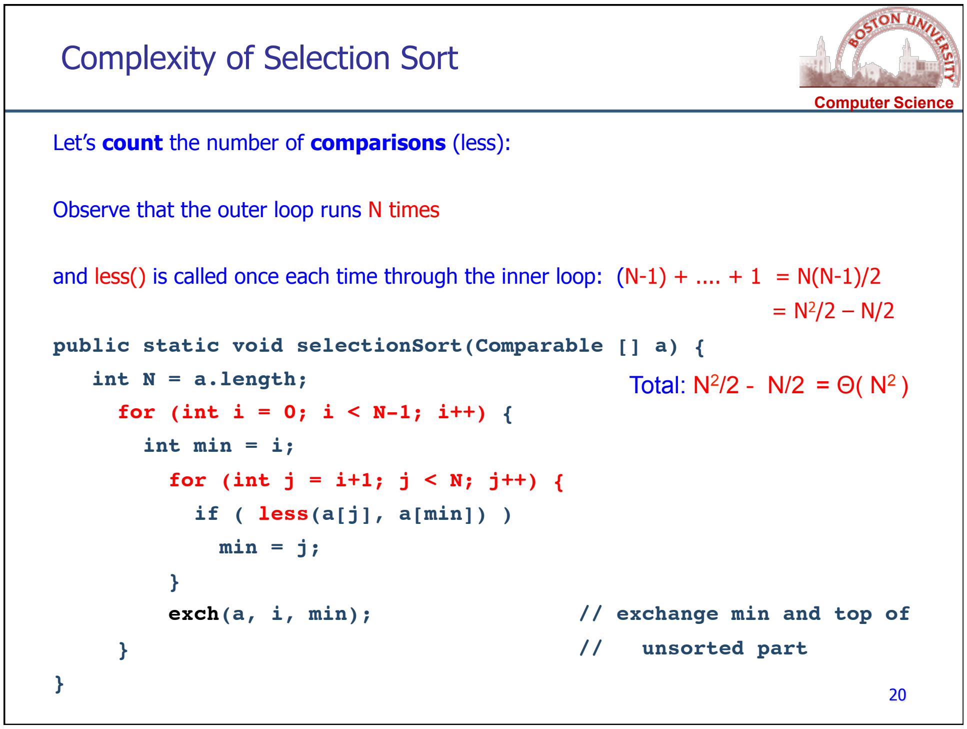
7	1	1	1	1
2	2	2	2	2
1	7	7	5	5
8	8	8	8	7
5	5	5	7	8

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

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

$$(N-1) + (N-2) + ... + 2 + 1 = N(N-1)/2 = N^2/2 – N/2 = \Theta(N^2)$$

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



Complexity of Selection Sort

Computer Science

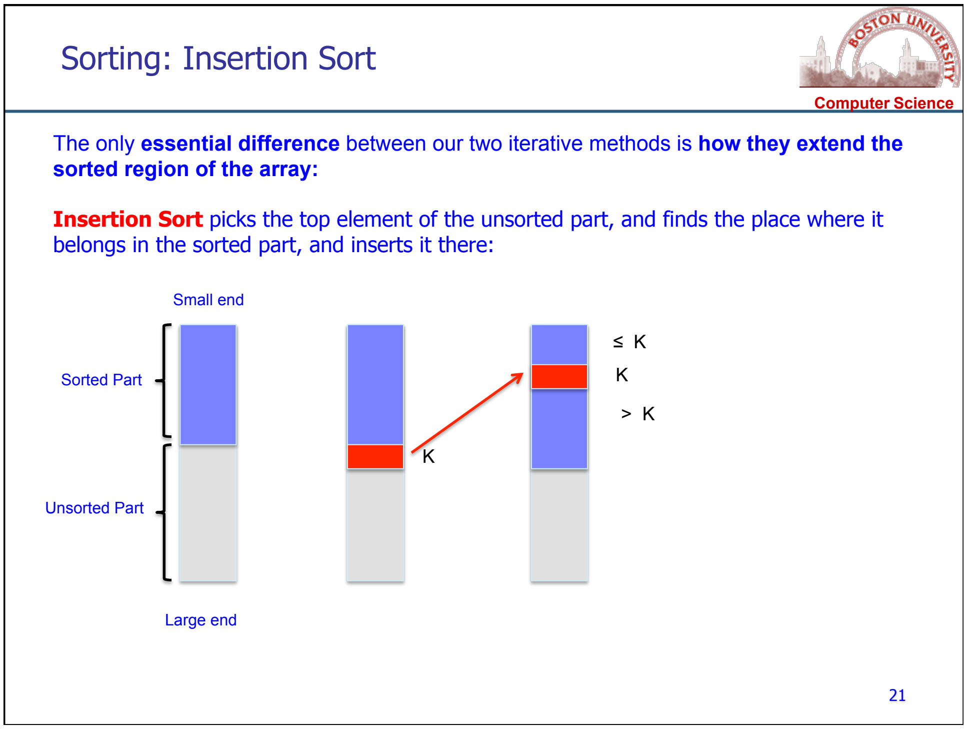
Let's **count** the number of **comparisons** (less):

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: $N^2/2 - N/2 = \Theta(N^2)$

```
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;
         }
         exch(a, i, min);                // exchange min and top of
     }                                   //   unsorted part
}
```

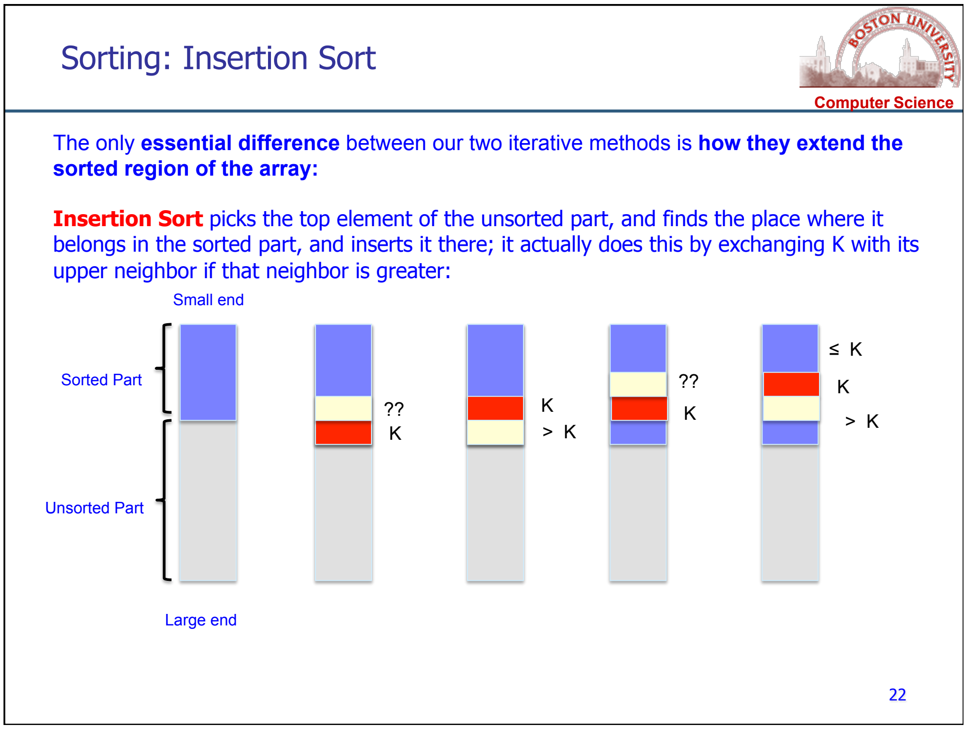


Sorting: Insertion Sort

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

Computer Science

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--) {
        exch(a, j, j-1);
      }
    }
}
```

Complexity of Selection Sort

Computer Science

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: $N^2/2 - N/2 = \Theta(N^2)$ calls to less(....)

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

Complexity of Selection 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(...), so it is = Θ(N) = "linear time."

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!

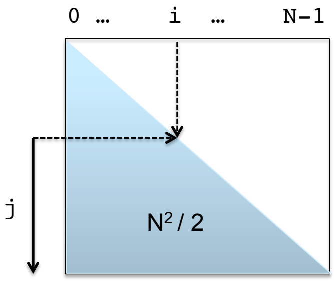
Algorithm Analysis

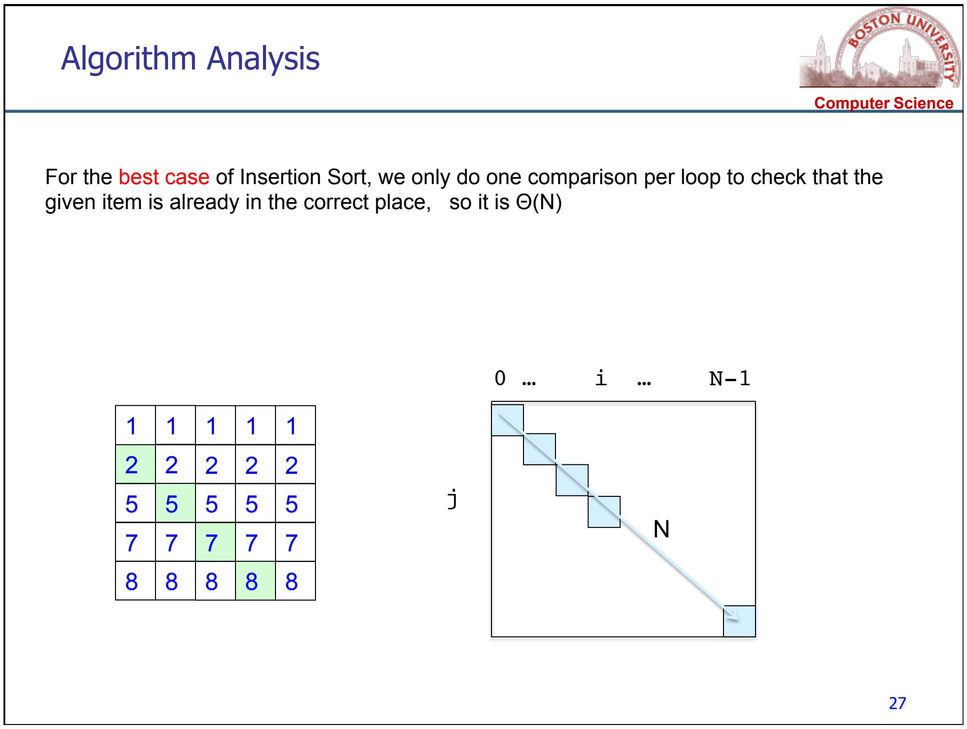
Computer Science

The graphical analogy helps us see why Selection sort (in all cases) is Θ(N^2) :

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





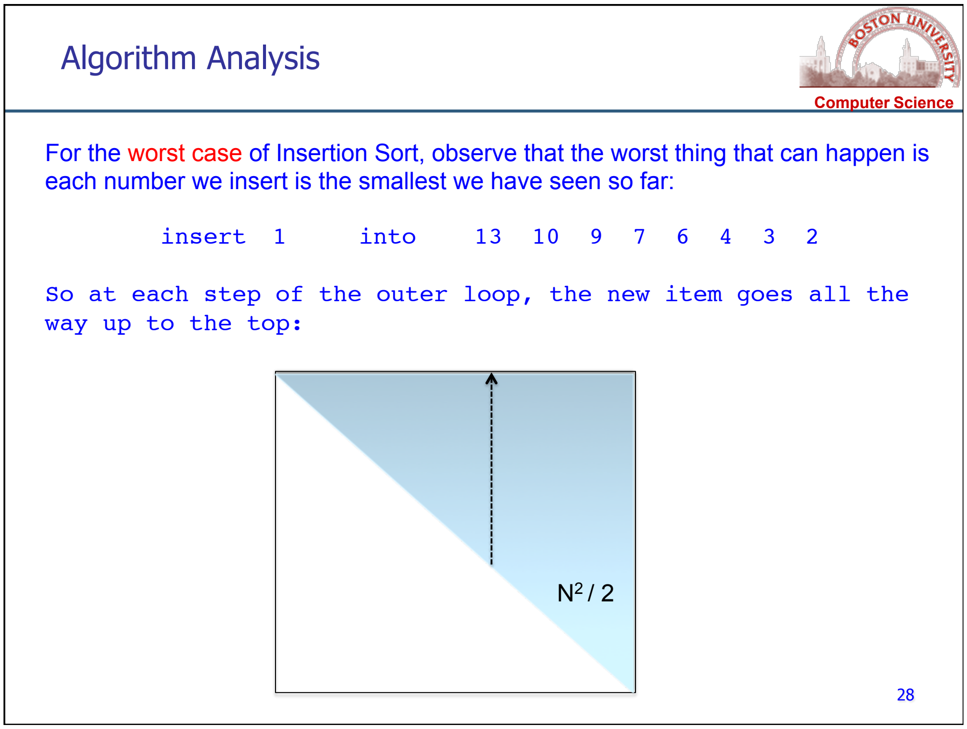
Algorithm Analysis

Computer Science

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 Θ(N)

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

27



Algorithm Analysis

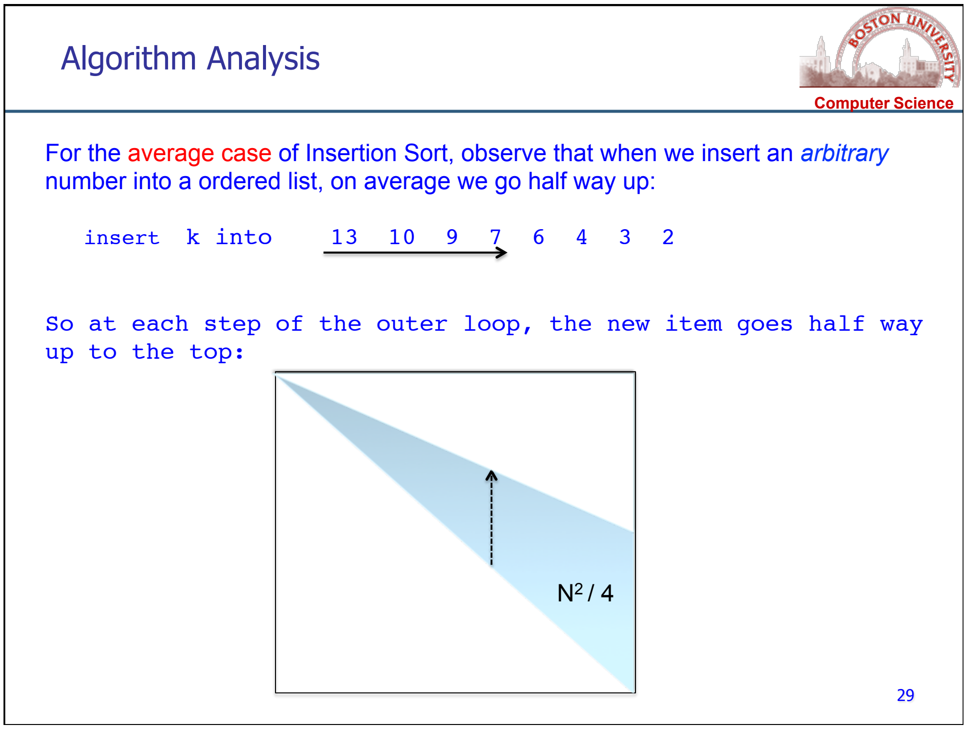
Computer Science

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:

28



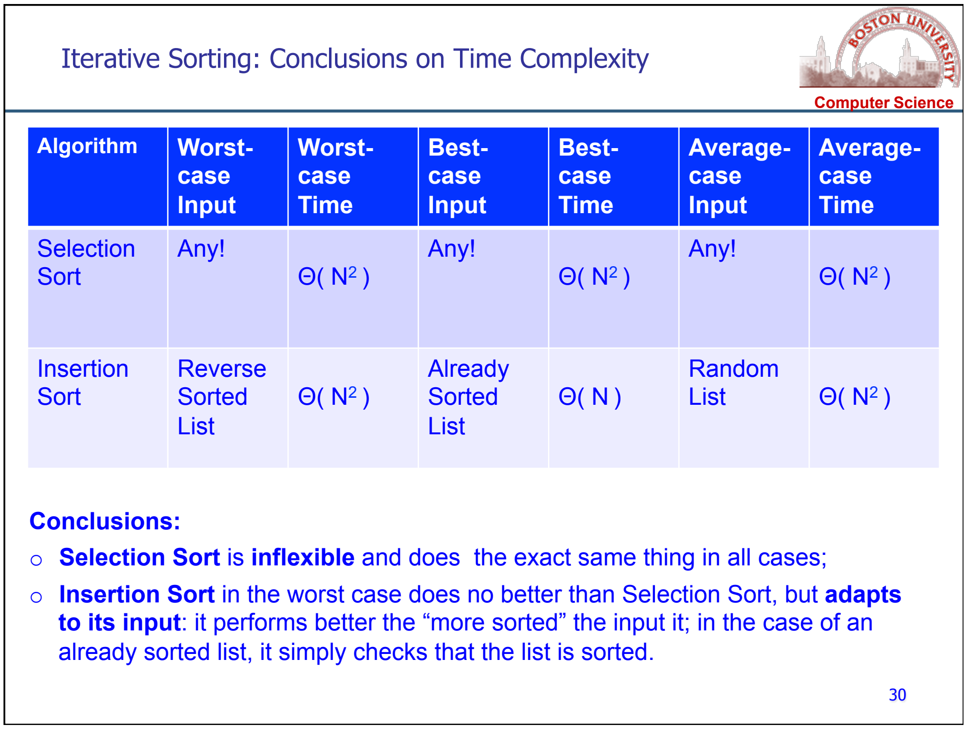
Algorithm Analysis

Computer Science

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
```

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



Iterative Sorting: Conclusions on Time Complexity

Computer Science

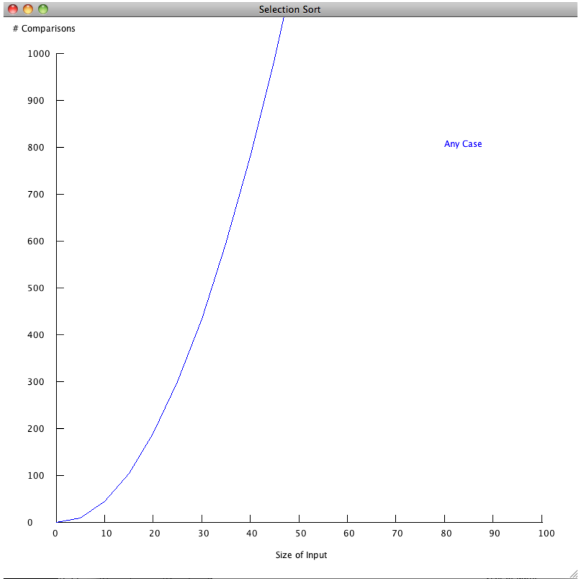
Algorithm	Worst-case Input	Worst-case Time	Best-case Input	Best-case Time	Average-case Input	Average-case Time
Selection Sort	Any!	Θ(N^2)	Any!	Θ(N^2)	Any!	Θ(N^2)
Insertion Sort	Reverse Sorted List	Θ(N^2)	Already Sorted List	Θ(N)	Random List	Θ(N^2)

Conclusions:

- **Selection Sort** is **inflexible** and does the exact same thing 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.

Experimental Results

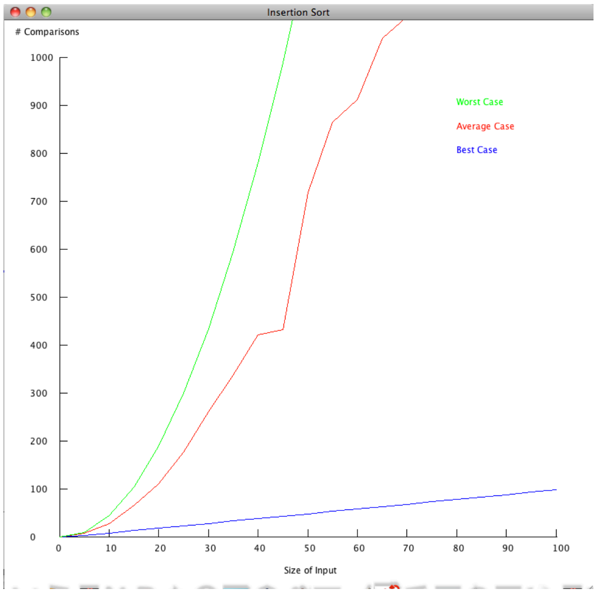
Computer Science



31

Comparing Sorting Algorithms

Computer Science



32