Analysis of Algorithms

Introduction

In this lecture we will study various ways to analyze the performance of algorithms. Performance concerns the amount of resources that an algorithm uses to solve a problem of a certain size: typically, we will speak about solving a problem using an array of size **N** or a linked list with **N** nodes in it. In addition, we will mostly be concerned with "worst-case" performance; other possibilities are "best-case" (simple, but not much useful information here), and "average-case" (too complicated mathematically to pursue in an introductory course). Finally, we will mostly be concerned with the speed (time, as a resource) of algorithms, although we will sometimes discuss the amount of storage that they require too (space, as a resource: we can also talk of worst-case, best-case, and average case).

We want to be able to analyze algorithms, not just the methods that implement them. That means we should be able to say something interesting about the performance of an algorithm independent from having a version of it (written in some programming language) that a machine can execute. Once we examine machine-executable versions, there is a lot of technology details to deal with: what language we write the code in, which compiler we use for that language, what speed processor we run it on, how fast memory is (and even how much caching is involved). While this information is important to predict running times, it is not fundamental to analyzing the algorithms themselves. So, we will analyze algoritms independent of technology, making this subject more scientific.

Instead, we will analyze an algorithm by predicting how many steps it takes, and then go through a series of simplifications leading to characterizing an algorithm by its complexity class. Although it initially may seem that we have thrown out useful information, we will learn how to predict running times of methods on actual machines, using an algorithm's complexity class, and timing it on the machine that it will be run on.

Analyzing Algorithms: From Machine Language to Big O Notation

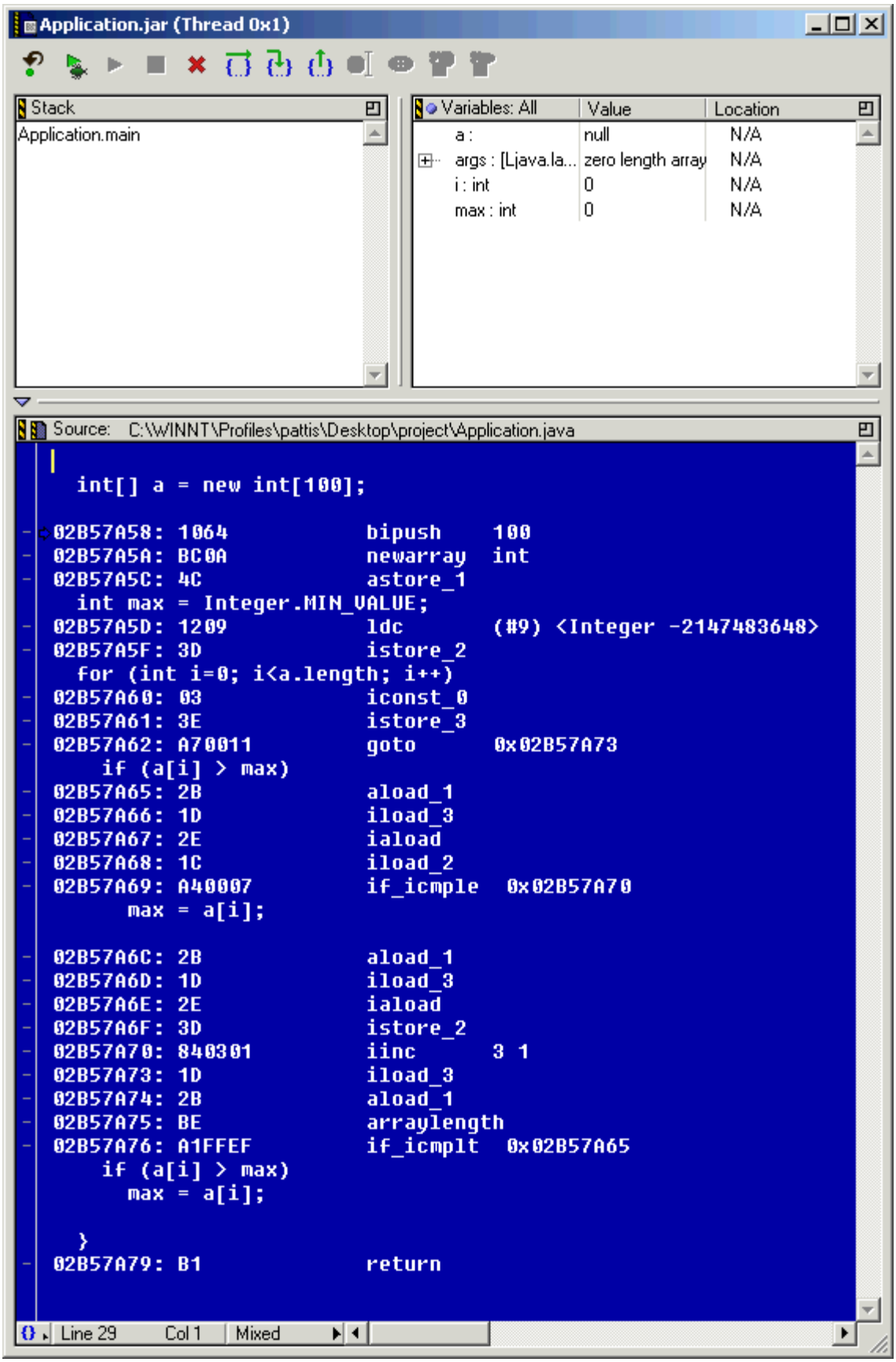
In this section we will start with a very concrete and technological approach to analyzing algorithms -by looking at how Java compiles such code to machine language- and then generalize to a science that is independent of such technology. First, suppose that we invent a mathematical function **Iaw(N)** that computes the number of machine code instructions executed by algorithm **a** when run on the **w**orst-case problem of size **N**. Such a function takes an integer as a parameter (**N**, the problem size) and returns an integer as a result (the number of machine language instructions executed).

Maximum

For example, the following code computes the maximum value in an array of integers.

```
int max = Integer.MIN_VALUE;
for (int i=0; i<a.length; i++)
  if (a[i] > max)
    max = a[i];
```

We can easily examine the machine language instructions that Java produces for this code by using the debugger. If we specify **Mixed** on the bottom control of its window, Java shows us the Java code interspersed with the machine language instructions.

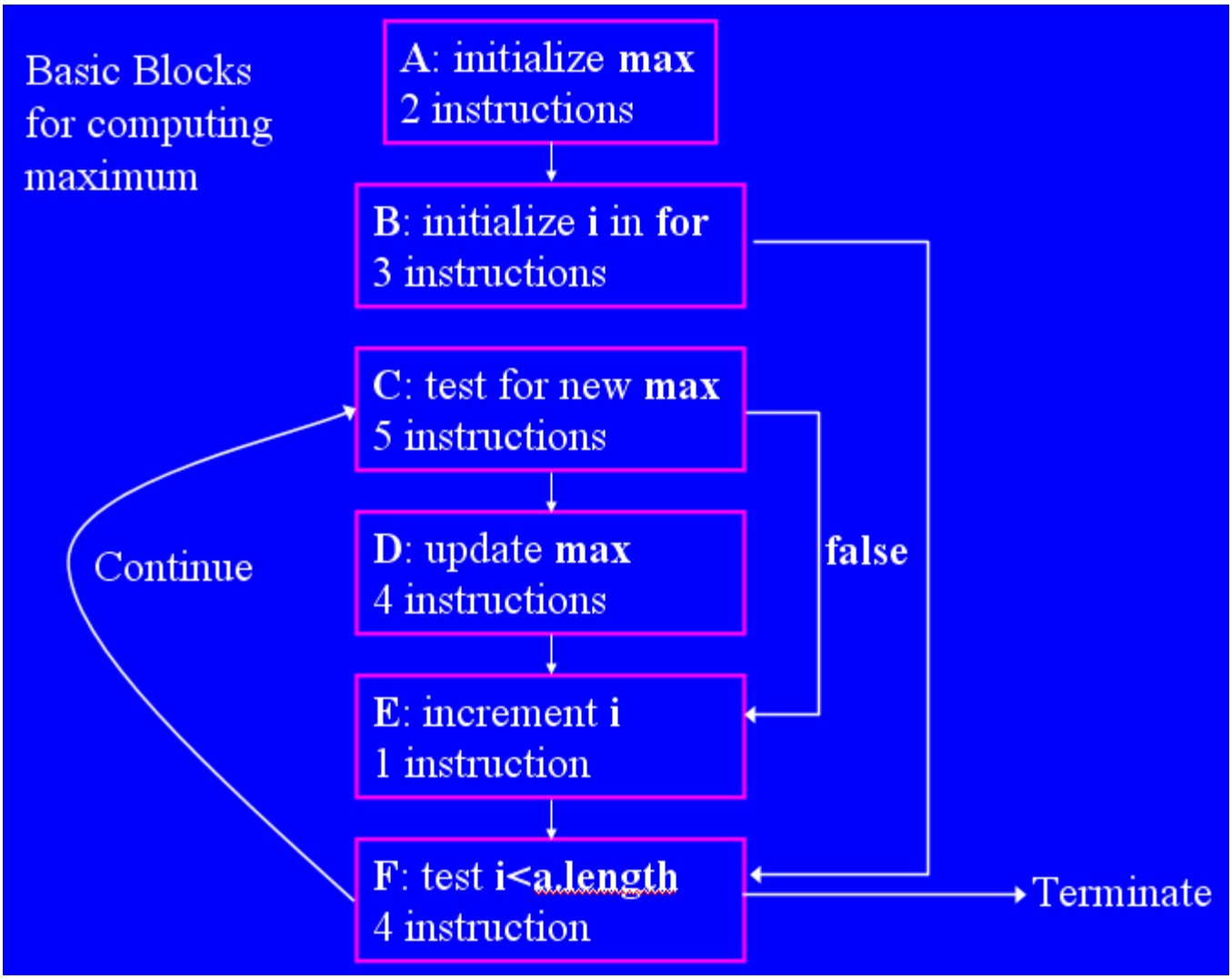


Below, I have duplicated this information, but reformatted it to be more understandable. I have put comments at the end of every line and put blank lines between what compiler folks call **basic blocks**.

```
          int max = Integer.MIN_VALUE;                          : A
```

```
051E6B3D: 1209           ldc       (#9)
051E6B3F: 3D             istore_2

          for (int i=0; i<a.length; i++)
051E6B40: 04       iconst_0               //get 0                : B
051E6B41: 3E       istore_3               //  store it in i
051E6B42: A70011   goto      0x051E6B53   //go to test near bottom

          if (a[i] > max)
051E6B45: 2B       aload_1                //get a's base address : C
051E6B46: 1D       iload_3                //  get i
051E6B47: 2E       iaload                 //  get a[i]
051E6B48: 1C       iload_2                //get max
051E6B49: A40007   if_icmple  0x051E6B50  //go to test near bottom if <=

          max = a[i];
051E6B4C: 2B       aload_1                //get a's base address : D
051E6B4D: 1D       iload_3                //  get i
051E6B4E: 2E       iaload                 //  get a[i]
051E6B4F: 3D       istore_2               //store it in max

051E6B50: 840301   iinc      3 1          //increment i          : E

051E6B53: 1D       iload_3                //get i                : F
051E6B54: 2B       aload_1                //get a's
051E6B55: BE       arraylength            //  length
051E6B56: A1FFEF   if_icmplt  0x051E6B45  //go to if above if <
```

Each basic block can be entered only at the top and exited only at the bottom. once a basic block is entered, all its instructions are executed. There may be multiple ways to enter and exit blocks. This code is a bit tortuous to read and hand simulate, but you can trace through it. An easier way to visualize the code in these basics blocks is as a graph, which I will annotate with all the information needed to understand computing instruction counts from it.



Block **A** initializes **max**. Block **B** initializes **i** in the **for** loop; not the branch leading to testing for termination, which is near the bottom. Block **C** compare **a[i]** to **max**, either falling through to Block **D** which updates **max** or skipping this block. Block **E** increments **i**. Block **F** tests whether the loop should terminate or execute the body (again). We can compute the exact number of instructions that are to be executed for any inputs. For simplicity, let us assume that all the array values are bigger than the smallest integer (used to initialize **max**.

If the array contains 0 values, 9 instructions are executed: blocks A, B, F. If the array contains 1 value, 23 instructions are executed: blocks A, B, F, C, D, E, F. If the array contains 2 values, either 33 instructions are executed (the first value is bigger than the second: blocks A, B, F, C, D, E, F, **C, E, F**) or 37 instructions are executed (the second value is bigger than the first - the worst case: A, B, F, C, D, E, F, **C, D, E, F**). Assuming the worst case from now on, if the array contains 3 values, 51 instructions are executed. ...

Thus, for this example -the code to compute the maximum value in an array- we can write the formula. **Iaw(N) = 14N + 9**. At most there are **14** instructions executed during each loop iteration (blocks C, D, E, F); the housekeeping to initialize **max** and initialize **i** and check the first loop iteration requires **9** instructions (blocks A, B, F). In fact, computing **Iab** (the number of steps in the **best** case, where the **if** test is true only on the first iteration, is

```
Iab(N) = 14N + 9 - 4(N-1) = 10N + 13
```

because the **4** instructions updating the **max** (block D) are never executed AFTER the first update; this formula works only when **N>0** Thus, the actual number of instructions has a lower bound of **10N + 13** and an upper bound **14N + 9** (when **N>0**).

Here **N** is **a.length** (the number of values stored in the array), and the **worst-case** run will be on an array of strictly increasing values: the **if** test executed during each iteration of the loop is repeatedly **true**, so the machine instructions to copy that value into **max** (block D) are always executed. Determining the **average case** is a problem in discrete math: given a random distribution (there are many; say the numbers are distributed **uniformly**) of values, how many times (on average) do we expect to execute block B, meaning the next value to be bigger than all the prior ones; this is not a simple problem but we can write programs to help understand it.

Let's return our focus to **Iaw(N) = 14N + 9**. Although this formula is simple, we want to make it even simpler if we can. Note that as **N** gets large (and most algorithmic analysis is asymptotic: it is concerned with what happens as the problem size **N** gets very large), the lower order term (**9**) can be dropped from this function to simplify it (less precision), without losing much accuracy.

For example, if **N** is 100, **Iaw(N) = 1,409**: if we drop the **9** term, the simplified answer is just 1,400 which is 99.3% of the correct answer; if we increase **N** to 1,000, Iaw(N) = 14,009: if we drop the **9** term, the simplfied answer is just 14,000 which is 99.94% of the correct answer; if we increase **N** to 10,000, Iaw(N) = 140,009: if we drop the **9** term, the simplfied answer is just 140,000 which is 99.994% of the correct answer.

Thus as **N** gets large (and 10,000 is not even a very large problem for computers) the lower term is not significant so we will drop it to simplify the formula to **Iaw(N) = 14N**.

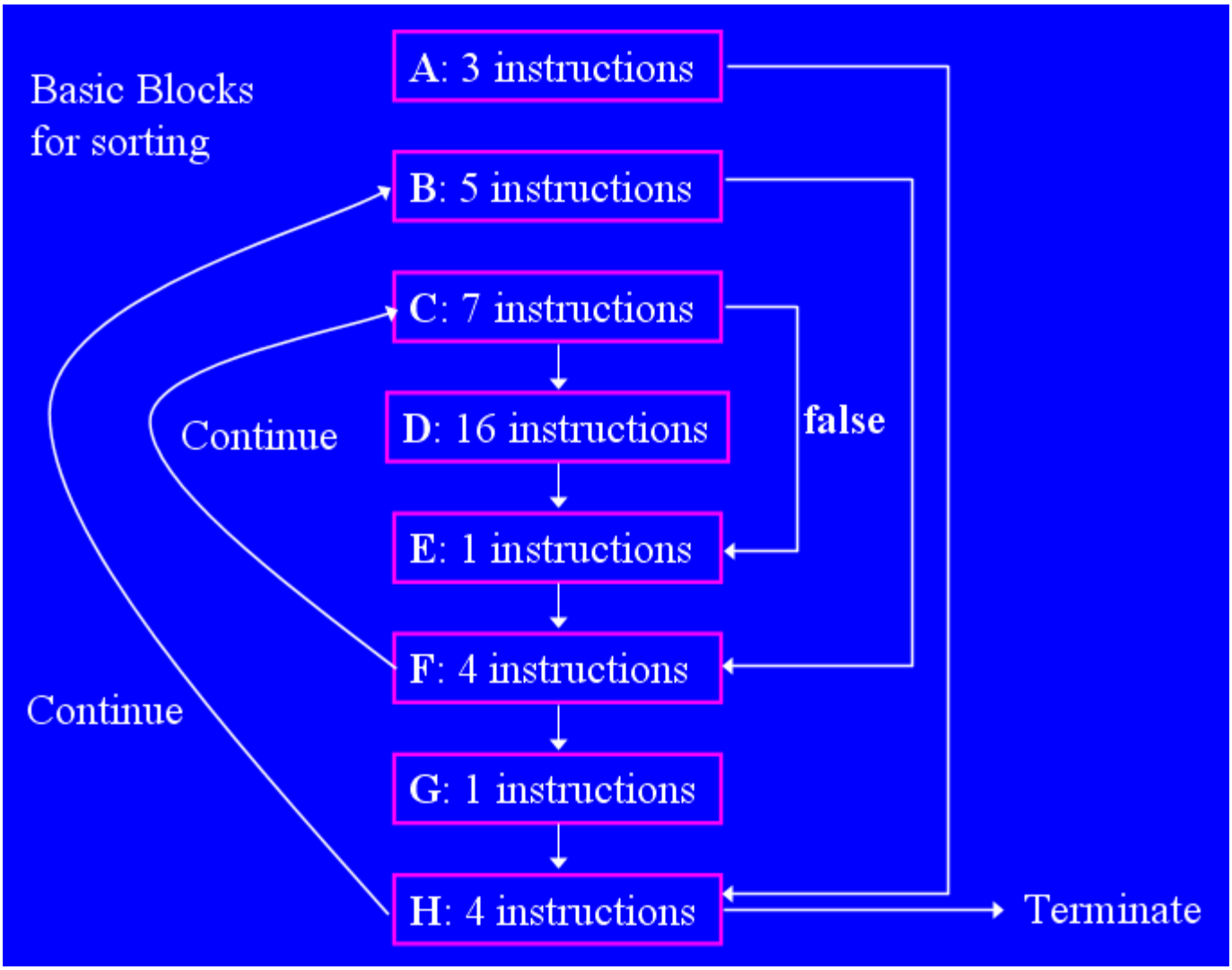
Mathematically, if Td(N) is the dominant term (here **14N**, we can drop any term T(N) if T(N)/Td(N) -> 0 as **N** -> infinity: note that 9/14N -> 0 as **N** -> inifinity, so that term can be dropped.

Sort

For another example, think about sorting an array. We can use the following simple to code but inefficient algorithm.

```
for (int base=0; base<N; base++)
  for (int check=base+1; check<N; check++)
    if (a[base]>a[check]) {
      int temp = a[base];
      a[base]  = a[check];
      a[check] = temp;
    }
```

The code for this example leads to the following basic blocks



Assume that for the worst-case input, every time two values in the array are compared, they are found to be in the wrong order and must be swapped. The following right side of an EBNF rule models the correct order of execution of basic blocks: **AH{BF{CDEF}GH}**, with the restriction that he inner repetition happens one fewer times than the outer repetition.

If the array contains 0 values, 7 instructions are executed: blocks A, H. If the array contains 1 value, 21 instructions are executed: blocks A, H, B, F, G, H. If the array contains 2 values, 63 instructions are executed: blocks A, H, B, F, C, D, E, F, G, H, B, F, I, H. If the array contains 3 values, 133 instructions are executed The resulting instruction-counting function is

$$\mathbf{Iaw(N) = 28N(N-1)/2 + 14N +7 = 14N^2 + 7}$$

For example, if **N** is 100, **Iaw(N) = 140,007**: if we drop the **7** term, the simplified answer is just 140,000 which is 99.995% of the correct answer. Thus as **N** gets large (and 100 is a tiny problem for computers) the lower term is not significant, so we will drop it to simplify the formula to **Iaw(N) = 14N^2**.

Recall our terms T(N) can be dropped if the limit T(N)/Td(N) -> 0 as **N** -> infinity: note that 7/14N^2 -> 0 as **N** -> inifinity, so that term can be dropped.

Now, let's take a look at the constant in front of the dominant term; it's value doesn't really matter for three important reasons. And, by getting rid of it we can simplify the formula again.

1. First, when computing the run time we are just going to multiply this constant by another constant relating to the machine, so we can discard this constant if we just use a different constant for the machine. Let **Taw(N)** denote the time taken by algorithm by algorithm **a** when run on the **w**orst-case problem of size **N**. If our machine executes 2 billion instructions/second, then the formula in the second case is **Taw(N) = $14N^2/2x10^9$**, or **Taw(N) = $.000000014N^2$**.

2. Second, a major question that we want answered is how much extra work does an algorithm do if the size of a problem is doubled. Note that for the second simplified version of **Iaw** (the one with only the dominant term multiplied by a constant), **Iaw(2N)/Iaw(N) = $14(2N)^2$ / $14N^2$ = 4** (meaning doubling the problem size quadruples the number of instructions that this algorithm executes), so the constant is actually irrelevant to the computation of this ratio.

3. Third, a major question that we want answered is how the speed of two algorithms compare: specifically, we want to know whether Iaw(N)/Ibw(N) -> 0 as **N** -> infinity, which would mean that algorithm **a** gets faster and faster compared to algorithm **b**. Again, the constant is irrelevant to this calculation.

Big O Notation

By ignoring all the lower-order terms and constants, we would say that algorithm **a** is **$O(N^2)$**, which means that the growth rate of the work performed by algorithm **a** (the number of instructions it executes) is **on the order** of N^2. This is called **big O** notation, and we use it to specify the **complexity class** of an algorithm.

Big O notation doesn't tell us everything that we need to know about the running time of an algorithm. For example, if two algorithms are **$O(N^2)$**, we don't know which will eventually become faster). And, if one algorithm is **O(N)** and another is **$O(N^2)$**, we don't know which will be faster for samll **N**. But, it does economically tell us quite a bit about the performance of an algorithms (see the three important questions above). We can compute the complexity class of an algorithm by the process shown above, or by doing something much simpler: determining how often its most frequently executed statement is executed as a function of **N**.

Returning to our first example,

```
int max = Integer.MIN_VALUE;
for (int i=0; i<a.length; i++)
  if (a[i] > max)
    max = a[i];
```

the **if** statement is executed **N** times, where **N** is the length of the array: **a.length**.

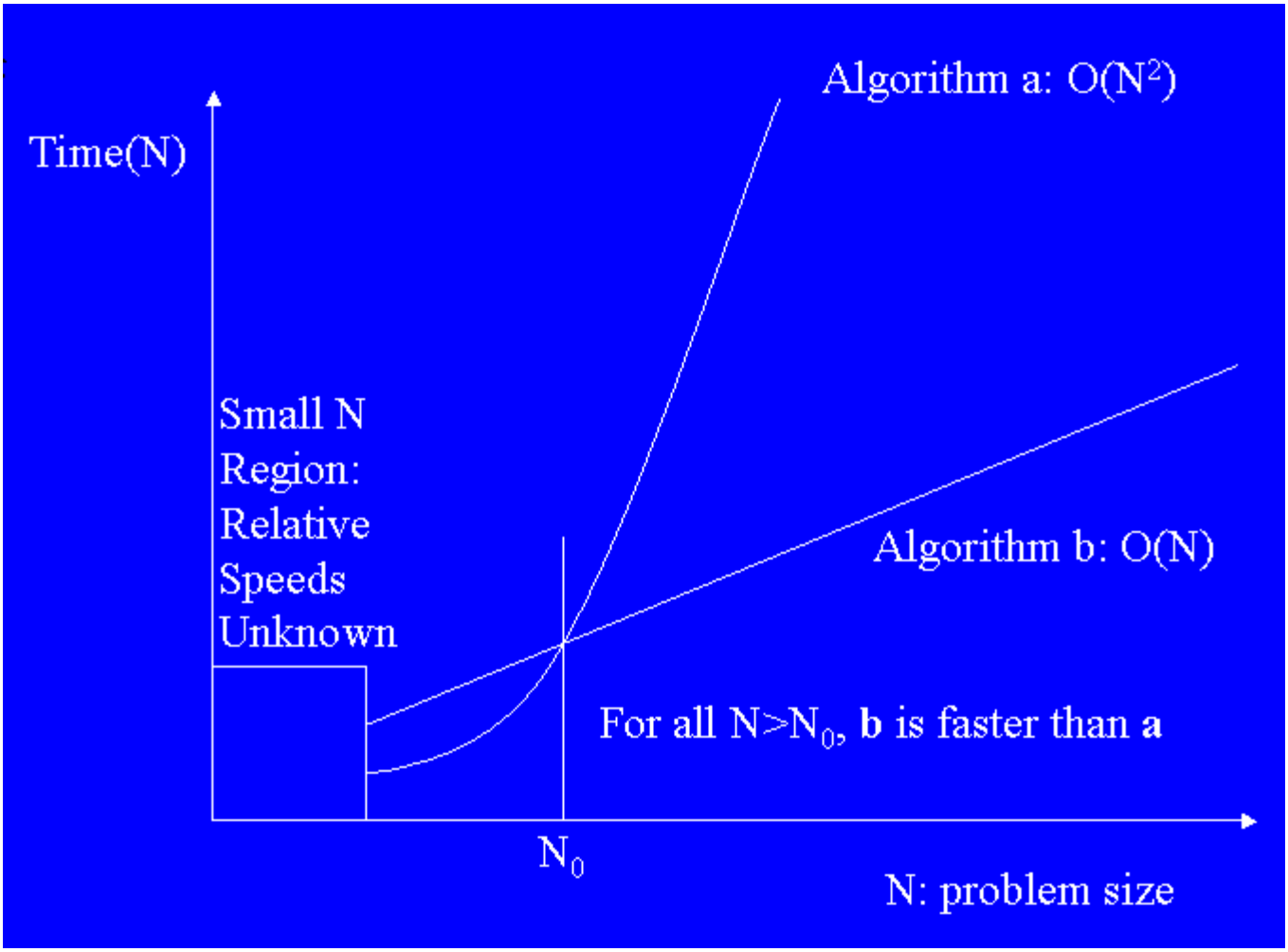
Returning to our second example,

```
for (int base=0; base<N; base++)
  for (int check=base+1; check<N; check++)
    if (a[base]>a[check]) {
      int temp = a[base];
      a[base]  = a[check];
      a[check] = temp;
    }
```

the **if** statement is executed about **N^2** times, where **N** is the length of the array: **a.length**. It is actually executed exactly **N(N-1)/2** times: for the first outer loop iteration it is executed N-1 times; for the second N-2 times, ... for the last 0 times. Know that **1+2+3+...+ N = N(N+1)/2**, so **1+2+3+...+N-1 = N(N-1)/2** or **$N^2/2$ - N/2**. Dropping the lower terms an constant yields

N^2.

Finally, note that comparing algorithms by their complexity classes is useful only for large N. We cannot state authoritatively whether an **O(N)** algorithm or an **O(N^2)** algorithm is faster for small N; but we can state that once we pass some threshold for N (call it N_0), the **O(N)** algorithm will always be faster than the **O(N^2)** algorithm. This ignorance is illustrated by the picture below.



In this example, the O(N) algorithm takes more time for small **N**. Of course, by adjusting constants and lower-order terms, it could also be the case that the O(N) algorithm is always faster; we cannot tell this information solely from the complexity class.

Technically, an algorithm **a** is O(f(n)) if and only iff there exist a number **M** and $\mathbf{N_0}$ such that **Iaw(n) <= Mf(n)** for all **n>N_0** This means for example that any O(N) algorithm is also O(N^2) too (or O(f(n)) for any f(n) that grows faster than linearly). Technically Ω is the symbol to use when you know a tight bound on both sides: if there exists $\mathbf{M_1}$, $\mathbf{M_2}$, and $\mathbf{N_0}$ such that $\mathbf{M_1}$**f(n) <= Iaw(n) <=** **Mf(n)** for all **n>N_0**, we say that algorithm **a** is Ω(f(n)). We will use just Big O notation, often pretending it is Ω. See Big O Notation in the online Wikipedia for more details.

Complexity Classes

Using big O notation, we can broadly categorize algorithms by their complexity classes. This categorization supplies one kind of excellent information: given the time it takes a method (implementing an algorithm) to solve a problem of size **N**, we can easily compute how long it would take to solve a problem of size **2N**.

For example, if a method implementing a certain sorting algorithm is in the complexity class $O(N^2)$, and it takes about 1 second to sort 10,000 values, it will take about 4 seconds to sort 20,000 values. That is, for complexity class $O(N^2)$, doubling the size of the problem quadruples the amount of time taken executing a method. The algebra to prove this fact is simple. Assuming $Taw(N) = c*N^2$ (where **c** is some technology constant related to the compiler used, the speed of the computer and its memory, etc.) the ratio of the time taken to solve a problem of size **2N** to the time take to solve a problem of size **N** is.

$$\mathbf{Taw(2N)/Taw(N) = c*(2N)^2 / c*(N)^2 = 4cN^2 / cN^2 = 4}$$

As we saw before, the constants are irrelevant: they all disappear no matter what the complexity class.

Likewise, using this method to sort 1,000,000 values (100 times more data) would take about 2.8 hours (that is 10,000 times longer, which is 100^2).

Here is a short characterization of common complexity classes. We will discuss some of these algorithms in more detail later in this handout, and use these complexity class to characterize many methods throughout the semester.

Complexity Class	**Class Name**	**Example**	**T(2N)**
$O(1)$	Constant	Insertion at the rear of an array Insertion at front of linked list Parameter passing	**T(N)**
$O(\log_2 N)$	Logarithmic	Binary Search	**T(N)+constant**
$O(N)$	Linear	Linear Search (arrays or linked lists)	**2T(N)**
$O(N \log_2 N)$	Log-Linear or Linearithmic	Fast Sorting	**2T(N)+constant**
$O(N^2)$	Quadratic	Slow/Simple Sorting	**4T(N)**
$O(N^3)$	Cubic	NxN Matrix Multiplication	**8T(N)**
$O(N^C)$	Polynomial or Geometric		$\mathbf{2^cT(N)}$
...	...	...	...
$O(C^N)$	Exponential	Proving Boolean Equivalences of **N** variables	$\mathbf{C^NT(N)}$

We can compute $\log_2 N = (\ln N)/(\ln 2) = 1.4427 \ln N$. Since log base 2 and log base e are linearly related, it really makes no difference which we use when using Big O notation, because only the constants (which we ignore) are different. You should also memorize that $\log_2 1000$ is about 10 (actually $\log_2 1024$ is exactly 10), and that $\log_2 N^a = a\log_2 N$. From this fact we can easily compute $\log_2 1{,}000{,}000$ as $\log_2 1{,}000^2$ which is $2\log_2 1000$ which is about 20. Do this for $\log_2 1{,}000{,}000{,}000$ (one billion).

Again, we should understand that these simple formulas work only when **N** gets large. This is the core of asymptotic algorithmic analysis Note that complexity classes before (and including) log-linear are considered "fast": their running time does not increase much faster than the size of the problem increases. The later complexity classes $O(N^2)$, $O(N^3)$, etc. are slow but "tractable". The final complexity class $O(2^N)$ grows so fast that it is called "intractable": only small problems in this complexity class can ever be solved.

For example, assume that $\mathbf{Ia_1w(N) = 10}$ (constant), and $\mathbf{Ia_2w(N) = 10log_2N}$ (logrithmic), and $\mathbf{Ia_3w(N) = 10N}$ (linear), etc. Assume further that we are running code on a machine executing 1 billion (10^9) operations per second. Then the following table gives us an intuitive idea of how running times for algorithms in different complexity classes changes with problem size.

Complexity Class	**N = 10**	**N = 100**	**N = 1,000**	**...**	**N = 1,000,000**
O(1)	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	$1x10^{-7}$ seconds	...	$1x10^{-7}$ seconds
$O(log_2N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-7}$ seconds	$10x10^{-7}$ seconds	...	$20x10^{-7}$ seconds
O(N)	$1x10^{-7}$ seconds	$1x10^{-6}$ seconds	$1x10^{-5}$ seconds	...	$1x10^{-3}$ seconds
$O(Nlog_2N)$	$3.3x10^{-7}$ seconds	$6.6x10^{-6}$ seconds	$10x10^{-5}$ seconds	...	$20x10^{-3}$ seconds
$O(N^2)$	$1x10^{-6}$ seconds	$1x10^{-4}$ seconds	$1x10^{-2}$ seconds	...	2.7 hours
$O(N^3)$	$1x10^{-5}$ seconds	$1x10^{-2}$ seconds	10 seconds	...	$3x10^3$ years
$O(2^N)$	$1x10^{-5}$ seconds	$4x10^{21}$ centuries	fuggidaboutit	...	fuggidaboutit

---

Time Estimation Based on Complexity Class

Up until this point we have continually simplified information about algorithms to make our analysis of them easier. Have we strayed so far from reality that our information is useless. No! In this section we will learn how we can easily and accurately (say, within 10%) predict how long it will take a method to solve a large problem size, if we know the complexity class for the method, and have measured how long the method takes to execute for some large problem size. Notice both the measured and predicted problem sizes must be reasonably large, otherwise the simplifications used to compute the complexity class will not be accurate: the lower order terms will have a real effect on the answer.

For a first example, we will measure, and then predict, the running time of a simple, quadratic sorting method. We will use a driver program (discussed below, in the Sorting section) to repeatedly sort an array containing 1,000 random values, and then predict how long it will take this method to sort an array containing of 10,000 random values (and actually compare this prediction to the measured running time for this problem size).

1. Because this sorting method is in the $\mathbf{O(N^2)}$ complexity class, we simply assume that

we can write $\mathbf{T(N) = cN^2}$ where we do not know the value of **c** yet.
2. We run the sorting method five times on an array containing 1,000 random values and measure the average running time: it is **.022** seconds.

Now we solve for **c**. Using **N = 1000** we have

$$\begin{aligned} T(1000) &= c\ 1000^2 \\ .022 &= c\ 10^6 \\ c &= .022/10^6 \\ c &= 2.2 \times 10^{-8} \end{aligned}$$

Thus for large **N**, $\mathbf{T(N) = 2.2x10^{-8}\ N^2}$ seconds. Using this formula, we can predict that using this method to sort an array of 10,000 random values would take about 2.2 seconds. The actually amount of time is about 2.7 seconds. The prediction is **100[1-(2.6-2.2)/2.6]** or 85% accurate (so, we barely missed our goal of 90% accuracy). It would be more accurate if we measured this sort on a 10,000 value array and predicted the time to sort a 100,000 value array.

For a second example, we wil measure, and then predict, the running time of a more complicated log-linear sorting method (this algorithm is in the lowest complexity class for all those that accomplish sorting). We will use a driver program to repeatedly sort an array containing 100,000 random values, and then predict how long it will take this method to sort an array containing of 1,000,000 random values (and actually compare this prediction to the measured running time for this problem size, which is small enough to measure).

1. Because this sorting method is in the $\mathbf{O(N \log_2 N)}$ complexity class, we simply assume that we can write $\mathbf{T(N) = c(N \log_2 N)}$ where we do not know the value of **c** yet.
2. We run the sorting method five times on an array containing 100,000 random values and measure the average running time: it is **.15** seconds (notice that this method sorts 10 times as many values over 10 times faster than the simple quadratic sorting method on the same amount of data).

Now we solve for **c**. Using **N = 10,000** we have

$$\begin{aligned} T(100{,}000) &= c\ (1{,}000{,}000 \log_2 1{,}000{,}000) \\ .15 &= c\ 1{,}660{,}964 \\ c &= .15/1{,}660{,}964 \\ c &= 9.0 \times 10^{-9} \end{aligned}$$

Thus for large **N**, $\mathbf{T(N) = 7.8x10^{-8}(N \log_2 N)}$ seconds. Using this formula, we can predict that using this method to sort an array of 1,000,000 random values would take 1.6 seconds. The actually amount of time is about 1.8 seconds. The prediction is **100[1-(1.8-1.6)/1.6]** or 87% accurate (so, we again missed our goal of 90% accuracy, but only barely).

Here is a final word on the accuracy of our predictions. If we sort the exact same array a few times (the sort testing driver easily does this) we will see variations of 10%-20%; likewise we get a slightly greater spread if we sort different arrays (but all of the same size). Our model predicts that these would all take the same amount of time. So all kinds of things (operating system, what programs it is running, what network connections are open, etc.) influence the actually amount of time taken to sort an array. In this light, the accuracy of our "naive" predictions is actually quite good.

Determining Complexity Classes Empirically

We have seen that it is fairly simple, given an algorithm, to determine its complexity class: determin how often its most frequently executed statement is executed as a function of **N**. But what if even that is too hard: it is too big or convoluted. Well, if we have a method

implementing the algorithm, we can actually time it on a few different-sized problems and infer the complexity class from the data.

First, be aware that the standard timer in Java is accurate to only .001 second (1 millisecond). Call this one **tick** So, to get any kind of accuracy, you should run the method on large enough data to take tens to hundreds of ticks (milliseconds).

So, run the method on some data of size **N**, enough for the required number of ticks, then of size **2N**, then of size **4N**, then of size **8N**. For algorithms in simple complexity classes, you should be able to recognize a pattern (which will be approximate by not exact). If the sequence of values is 1.0 seconds, 2.03 seconds, 3.98 seconds, and 8.2 seconds: the method seems O(N). Here each doubling approximately doubled the time the method ran. If the sequence of values is 1.0 seconds, 3.8 second, 17.3 seconds, and 70.3 seconds: the method seems $O(N^2)$. Here each doubling approximately quadrupled the time the method ran.

Of course, things get a bit subtle for a complexity class like $O(N\log_2 N)$, but you'll see it always a bit worse than linear, but nowhere near quadratic. Of course $O(N\log^2_2 N)$ would behave simlarly, so you must apply this process with a bit of skepticism that you are computing perfect answers.

Searching: O(N) and $O(\log_2 N)$ Algorithms

Linear seaching, whether in an array or in a linked list, is O(N); in the worst case (where the value being searched for is not in the data structure), each value in the data structure must be examined (the inner **if** statement must be executed **N** times).

```
public static int linearSearch (int[] a, int value)
{
  for (int i; i<a.length; i++)
    if (a[i] == value)
      return i;
  return -1;
}
```

Linear searching in an ordered array is no better: it is still O(N). Again, in the worst case (where the value being searched for is bigger than any value in the data structure), each value in the data structure must be examined. But, there is a way to search an ordered array that is much faster. This algorithm, for reasons that will become clear soon, is called **binary searching**.

Let's explore this algorithm first in a more physical context. Suppose that we have 1,000,000 names in alphabetical (sorted) order in a phone book, one name and its phone number per page (only on the front of a page, not the back). Here is an algorithm to find a person's name (and their related phone number) in such a phone book

1. Find the middle page in the (remaining) book
2. If it contains the name that we are looking for, we are done
3. Otherwise, we rip out that page and tear the remaining phone book in half
    1. If the name we are looking for comes before the page we that ripped out, we throw away the second half of the phone book
    2. If the name we are looking for comes after the page that we ripped out, we throw away the first half of the phone book
4. Repeat this process until we find the name or there are no pages left in the phone book

This method is called binary search because each "iteration" divides the problem size (the phone book) in half (bi means two: e.g., bicycle).

If the original phone book had 1,000,000 pages, after the first iteration (assuming the name we are looking for isn't right in the middle) the remaining book would have about 500,000 pages (actually it would have 499,999). In this algorithm, the first comparison eliminates about 500,000 pages! After the second comparison, we are down to a phone book containing about 250,000 pages. Here one more comparison eliminates about 250,000 pages; not as good as the first comparison, but still much better than linear searching, where each comparison eliminates just one page! If we keep going, we'll either find the name or after about 20 comparisons the phone book will be reduced to have no pages. Critical to this method is the fact that the phone book is alphabetized (ordered); it is also critical to be able to find the middle of the phone book quickly (which is why this method doesn't work on linked lists).

To determine the complexity class of this algorithm (operating on a sorted array), notice that each comparison cuts the remaining array size in half (actually, because the midpoint is also eliminated with the comparison, the size is cut by a bit more than a half). For an **N** page book, the maximum number of iterations $\log_2$ **N** (the number of times we can divide **N** by 2 before it is reduced to 1; or, the number of times that we can double 1 before reaching N). Notice in this algorithm if the array size doubles, the number of iterations increases by just 1: the first comparison would cut the doubled array size back to the original array size.

Again, here are some important facts about logarithms that you should memorize.

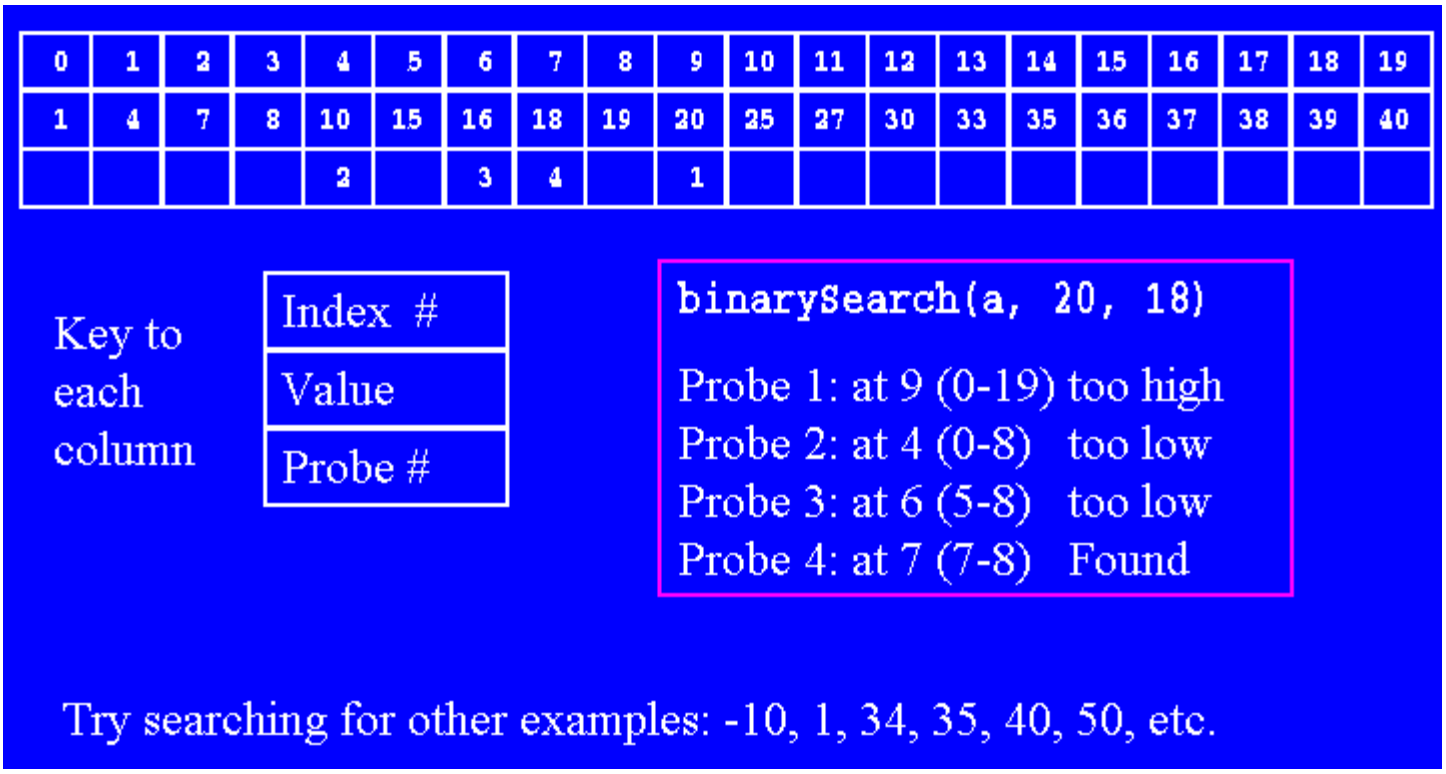
1. $2^{10} = 1{,}024$ so $\log_2 1{,}000$ is about 10
2. $\log_2 X^2 = 2 \log_2 X$; so $\log_2 1{,}000{,}000 = \log_2 1000^2 = 2 \log_2 1000$ is about 20
3. On a calculator, compute $\log_2$ **N** = ln **N** / ln 2 (where ln is logarithm base e, provided on most calculators -and by the **Math.log** method).

For **N** = 10, binary search does about 3 iterations, but each iteration is more complicated than linear search. For **N** = 1,000,000 binary search does 20 iterations, or 50,000 times fewer iterations than the worst case for linear search! Practically, even for arrays of size 1,000,000 both algorithms run quickly, but binary search runs 50,000 times faster so when repeatedly searching such a big array, binary search would be much much better (seconds vs. hours).

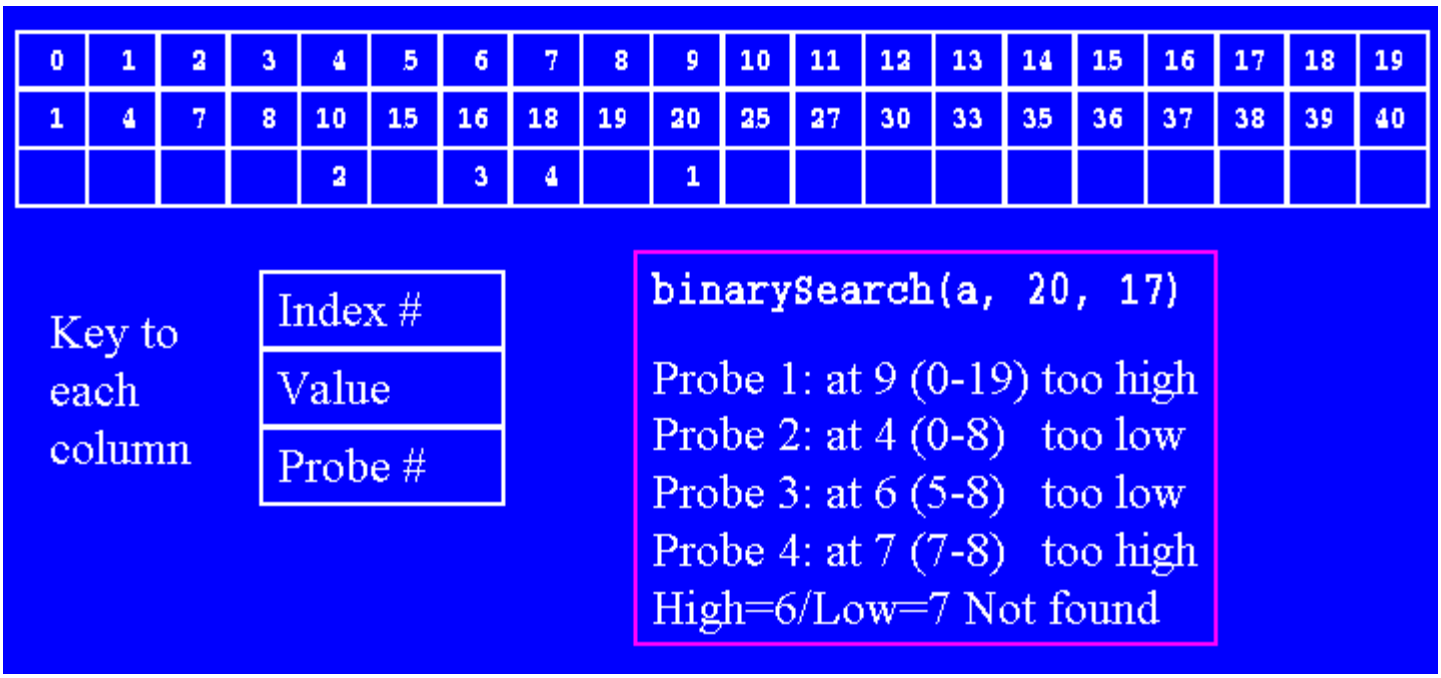
Here a method for implementing the binary search algorithm on arrays.

```
public static int binarySearch (int[] a, int value)
{
  int low  = 0;
  int high = a.length-1
  for(;;) {
    if (low > high)           //low/high bounds inverted, so
      return -1;              //  the value is not in the array

    int mid = (low+high)/2;   //Find middle of the array

    if (a[mid] == value)      //Found value looking for, so
      return mid;             //  return its index; otherwise
    else if (value < a[mid])  //determine which half of the
      high = mid-1;           //  array potential stores the
    else                      //  value and continue seraching
      low  = mid+1;           //  only that part of the array
}
```

The following illustration shows how this method executes in a situation where it finds the value it is searching for. Notice how it converges on those indexes in the array that might store the searched for value.



The following illustration shows how this method executes in a situation where it does not find the value it is searching for.



Again, each iteration of the loop reduces the part of the array being looked at by a factor of two. How many times can we reduce a size **N** array before we are left with a single value? $\log_2 N$ (the same number of times we can double the size of an array from 1 value to N).

Finally, note that we cannot perform binary searching efficiently on linked lists, because we cannot quickly find the middle of a linked list. In fact, another self-referential data structure, trees, can be used to perform efficient searches.

Sorting: $O(N^2)$ and $O(N \log_2 N)$

Sorting is one of the most common operations performed on an array of data. We saw in the previous section how sorting an array allows it to be searched much more efficiently. Sorting

Algorithms

algorithms are often divided into two complexity classes: simple to understand algorithms whose complexity class is $O(N^2)$ and more complicated algorithms whose complexity class is $O(N \log_2 N)$. The latter are much faster than the former for large arrays. The fast one was the **Arrays.sort** method which sorts any array of objects efficiently: it implements an $O(N\log_2 N)$ algorithm with a small constant.

Here is a brief description of three $O(N^2)$ sorting algorithms.

1. In bubble sort, a next position to fill is compared with all later positions, swapping out-of-order values.
2. In selection sort, the smallest value in the remaining positions is computed and swapped with the value in the next position.
3. In insertion sort, the next value is moved backward (swapped with the value in the previous position in the region of sorted values) until it reaches its correct position.

These algorithms are arranged in both simplest-to-most-complicated order, as well as slowest-to-fastest order for large N.

Here is a brief description of three $O(N \log_2 N)$ sorting algorithms.

1. In merge sort, pairs of small, adjacent ordered arrays (the smallest are 1 member arrays) are merged repeated into larger ordered arrays until the result is just one ordered array containing all the values.
2. In heap sort, values are added to and then removed from a special kind of tree data structure called a heap (which we will study later: its add and remove operations are both $O(\log_2 N)$, so adding and then removing **N** values is $NxO(\log_2 N) + NxO(\log_2 N) = O(N \log_2 N)$ total.
3. In quick sort, a pivot value is chosen and then the array is partitioned into three regions: on the left are those values less than the pivot, in the middle are those equal to the pivot, and on the right are those values greater than the pivot; then this process is repeated in the left and right regions (if they contain more than one value).

Heap sort is slower than merge sort, but it takes no extra space (merge sort requires another array that is as big as the array being sorted). Technically, Quick sort is $O(N^2)$. But on most arrays itis the fastest (and requires no extra space), but on pathologicallyh bad arrays, which are rare, it can take much longer to execute than the other methods.

All these sorting algorithms are defined as **static** methods in the **Sort** class. All method have exactly the same prototype (so they can be easily interchanged)

```
public static void bubble (Object[] a, int size, Comparator c)
```

which includes

1. An array of **Object** references to be sorted.
2. An **int** specifying how many references are stored in the array; this can be **a.length** if the array is filled.
3. An object from a class implementing **Comparator**, which decides which objects belong before which others in the sorted array.

Finally, it has been proven that when using comparisons to sort values, all algorithms require at least $O(N \log_2 N)$ comparisons. Thus, there are no general sorting algorithms in any complexity class smaller than log-linear (although better algorithms -ones with smaller

constants- may exist).

Analyzing Collection Classes

Analyzing a collection class is a bit of an art, because to do it accurately we need to understand how often each of its methods is called. We can, however, make one reasonable simplifying assumption for most simple collection classes: we assume that **N** values are added to the collection and then those **N** values are removed from the collection. This doesn't always happen, but it is reasonable.

So, in the case of simple array implementations of a stack or queue, both "add" methods (**push** and **enqueue**) are O(1) (assuming no new memory must be allocated); but the **pop** remove method is O(1) while the **dequeue**) remove method is O(N). Because NxO(1) is O(N), and O(N)+O(N) is O(N), adding and then removing **N** values from the stack collection classes is O(N). Because NxO(N) is $O(N^2)$, and O(N)+$O(N^2)$ is $O(N^2)$, adding and then removing **N** values from the queue collection classes is $O(N^2)$.

As another example, look at the array implementation of a simple priority queue, keeping the array sorted. There, the **enqueue** operation is O(N) because this method scans the array trying to find the correct position (based on its priority; highest priority is at the rear) for the added value. In the worst case, it has a priority lower than any other value, so the entire array must be moved backward to put that value at the front. The **dequeue** operation is just O(1), because it just removes the value at the rear of the array, requiring no other data movement. Because NxO(N) is $O(N^2)$, and NxO(1) is O(N), and $O(N^2)$+O(N) is $O(N^2)$, adding and then removing **N** values from this implementation of a priority queue also has a complexity class $O(N^2)$.

If we instead enqueued the value on the rear and dequeued by searching through the array for the highest priority value, we would still have one O(N) term and one $O(N^2)$ term, leading to $O(N^2)$ as the overall complexity class. But, later we will learn how implement priority queues with heaps. Both **enqueue** and **dequeue** are $O(\log_2 N)$: worse than O(1) but better than O(N). Thus, adding and then removing **N** values from this implementation of a priority queue is $NxO(\log_2 N)$ + $NxO(\log_2 N)$ which is $O(N\log_2 N)$ + $O(N\log_2 N)$ which is $O(N\log_2 N)$. So, "balancing" the add and remove operations yields a lower complexity class when both operations occur **N** times.

Finally, when we use an array to store a collection, each time that we double the array we must copy its **N** values. By doubling the size, we do this only $\log_2$ **N** times when adding **N** values, for a total of $N\log_2 N$ copies; therefore, we can think of each addition as requiring $\log_2 N$ copies (this is called the "amortized cost" of this operation: it really doesn't occur on every add, but when averaged over all the adds it is correct). So, in the case of an array implementation of a stack or queue, both "add" methods are actually $O(\log_2 N)$. Because $NxO(\log_2 N)$ is $O(N \log_2 N)$, NxO(1) is O(N), and NxO(N) is $O(N^2)$, The stack class is actually $O(N \log_2 N)$ for **N** pushes and pops, while the stack class is actually $O(N^2)$ for **N** enqueues and dequeues. By this calculation, the array implementations have a slightly higher complexity class than those using linked lists. But, because linked lists allocated a new object for every value put in the linked list, the running time of collections using linked lists can actually be higher. We will address this problem again when we cover linked lists.

Efficiency Pragmatics

Generally programmers address efficiency concerns after a program has been written clearly and correctly: "First get it right, then make it fast." Sometimes (see below) there is no need to

make a program run any faster; other times a program must be made to run faster just to test it (if tests cannot be performed quickly enough when debugging the program).

Programs should run as fast as necessary; making a program run faster often requires making it more complicated, more difficult to generalize, etc. For example, many scientific programs run in just a few seconds. Is there a pragmatic reason to work on them to run faster? No, because it typically takes a few days to collect the data for the program. As a component in the entire task, the program part is already fast enough. Likewise, programs that require lots of user interaction don't need to be made more efficient: if the computer spends less than a tenth of a second between the user entering his/her data and the program prompting for more, it is fast enough. Finally, in the pinball animation program, if the model can update and show itself in less than a tenth of a second, there is no reason to make it run faster; if it cannot, then the animation will be slowed down, and there is a reason to improve its performance.

The most famous of all rules of thumb for efficiency is the rule of 90/10. It states that 90% of the time a program takes to run is a result of executing just 10% of its code. That is, most time in a program's execution is spent in a small amount of its code. Modifying this code is the only way to achieve any significant speedup.

For example, suppose a 10,000 line program runs in 1 minute. By the rule of 90/10, executing 1,000 lines in this program accounts for 54 seconds, while executing the remaining 9,000 lines account for only 6 seconds. So, if we could locate and study those 1,000 lines (a small part of the program) and get them to execute in half the time, the total program would run in 27+6 = 33 seconds, which reduces the execution time for the entire program by almost 50%. If instead we could study the other 9,000 lines and get them to execute instantaneously (admittedly, a very difficult feat!), the total program would run in 54+0 = 54 seconds, which reduced the execution time for the entire program by ony 10%. Note that if you randomly change code to improve its efficiency, 90% of the time you will not be making any changes resulting in a significant improvement.

Thus, a corollary of the 90/10 rule is that for 90% of the code in a program, if we make it clearer but less efficient, it will not affect the total execution time of the program by much. In the above program, if we rewrote the 9,000 lines to make them as clear and simple as possible (with no regard for their efficiency) and increased their running time by 50% (from 6 to 9 seconds), the total program would run in 54+9 = 63 seconds, which is only a 5% increase in total execution time.