Timing
and Efficiency

ALGORITHM EFFICIENCY

Comparing Algorithms

- You can compare algorithms based on storage or execution time.
- Storage
  - Less memory is better
  - But memory is cheap these days!
  - Unless you are working in a low-memory environment, you can often safely ignore storage when comparing algorithms.
- Timing is key!

Measuring Empirically

- Empirical measurement is based on observation and data
  - This means actually using a timer to time your program!
- To compare two methods, you would implement them, select random inputs, then time both. You could do this multiple times and find the average for comparison.
- This isn’t common and often isn’t practical, but keep in mind that it’s always a possibility!

Running Time

- Let’s start with discussing running time.
- This is a fine-grained approach to figuring out the exact number of operations required by an algorithm.
- This is **not** how we will compare algorithms, but it is a helpful first step in learning about efficiency.

Running Time Example

- Algorithm A:

```
int sum = 0;
int i=0;
while(i < n) {
    sum += i;
    i++;
}
```

Running Time Example

- Algorithm A Rewritten:

```
int sum = 0;
int i=0;
while(i < n) {
    sum = sum + i;
    i = i + 1;
}
```

Running Time Example

```
int sum = 0; 1 assignment
int i=0; 1 assignment
while(i < n) { loop runs n times
      sum = sum + i; 1 addition, 1 assignment
      i = i + 1; 1 addition, 1 assignment
}
```

Running Time Example

```
int sum = 0; 1 assignment
int i=0; 1 assignment
n+1 comparisons
while(i < n) { loop runs n times
      sum = sum + i; 1 addition, 1 assignment
      i = i + 1; 1 addition, 1 assignment
}
```

Running Time Example

```
int sum = 0; 1 assignment
int i=0; 1 assignment
while(i < n) { loop runs n times
      n+1 comparisons
      sum = sum + i; 1 addition, 1 assignment
      i = i + 1; 1 addition, 1 assignment
}
```

- 2 assignments +
- (n+1) comparisons +
- n (2 addition + 2 assignments)

Running Time Example

2 assignments + (n+1) comparisons +

n (2 addition + 2 assignments)

2 assignments + (n+1) comparisons + 2n additions + 2n assignments

(2n + 2) assignments + (n+1) comparisons + 2n additions

Running Time Example

T(n) = (2n + 2) assignments + (n+1) comparisons + 2n additions

- Let's assume that all simple statements like assignment, addition, and comparison require the same amount of time.

T(n) = (2n + 2) * TIME + (n+1) * TIME + 2n * TIME

Running Time Example

$T(n) = (2n + 2) * TIME + (n+1) * TIME + 2n * TIME$

- Let's assume that TIME is one *unit of time* (using whatever arbitrary *unit* we want!)

$T(n) = (2n + 2) + (n+1) + 2n$

$T(n) = 5n + 3$

Running Time Example

```
statement1
int i = 1;
while( i <= n) {
    if(condition1) {
        statement2
    }
    statement3;
    i++;
}
```

Running Time Example

```
statement1 1
int i = 1;  1
n+1 conditional
while( i <= n) { n times
     if(condition1) { 1
          statement2 1 (worst case)
     }
     statement3; 1
     i++; 2
}

T(n)= 2 + (n+1) + n(5) = 6n + 3
```

Formulas used in Running Time

- for i = 1 to n, $\sum 1$
  - $1 + 1 + 1 + \ldots + 1$ (n times)
  - equal to n
- for i = 1 to n, $\sum i$
  - $1 + 2 + 3 + \ldots + n$
  - equal to $\frac{n(n+1)}{2}$
  - Gauss Formula

Comparing Running Times

- We could calculate and compare running times for algorithms.
- But it’s clear that figuring out the running time for a complex algorithm will get very complex and tedious!
- Do we really need this?

Order of Magnitude

- When thinking about time and efficiency, we often only care about order of magnitude.
- Based on powers of 10: 1, 10, 100, 1000, etc.

Example: Comparing Running Times

- Let's pretend we were comparing Algorithm A to some Algorithm B that had a running time of $T(n) = 4n + 12$. We want to know which is more efficient.

Example

n	A: 5n + 3	B: 4n + 12
1	8	16
10	53	52
100	503	412
1000	5003	4012
10,000	50,003	40,012
100,000	500,003	400,012
1,000,000	5,000,003	4,000,012

- These are within the same order of magnitude.
- They are equally efficient.

Example: Comparing Running Times

- Let’s now compare Algorithm C with a running time of $T(n) = n + 10{,}000$ to Algorithm D with a running time of $T(n) = n^2$

Example

n	C: n + 10,000	D: n^2
1	10,001	1
10	10,010	10,100
100	10,100	20,000
1000	11,000	1,010,000
10,000	20,000	100,010,000
100,000	110,000	10,000,010,000
1,000,000	1,010,000	1,000,000,010,000

- These quickly stop being the same order of magnitude.
- Algorithm C is more efficient.

What’s going on?

- What is driving which algorithm is more efficient?

n

- n is the size of the problem data: the number of inputs, the size of the data set, the size of the array, the number of elements in a list, etc.

What’s going on?

- It’s all about n!
- It does matter what n’s coefficient is.
- It doesn’t matter what other values are added to n.
- n drives everything.

BIG-O

Measuring Efficiency

- We can calculate the actual running time… but we don’t really need it.
- All we really care about is the order of magnitude.
- We don’t need the complete running time to get that!
- We can use *order of growth* instead.

Big O (Order of Growth)

- The efficiency of an algorithm can be described by Big O, which stands for the *order of growth*.
- Big O is described as a function of n, which is the size of the data set.

Big O

- Big O doesn't measure how long an algorithm takes.
- Big O is a measure of how the time required **changes** as the size of the data set **changes**.
  - The *rate of increase*
  - How the running time **changes** as the problem size **changes**
- It's not: "How long does the problem take to execute on size n?" It's: If I increase the size of n, how much longer will the problem take now?"

Big O

- The order of growth (Big O) is based on the dominant factor in the running time.
  - The highest power of n.
  - We ignore coefficients.
  - We ignore other parts of the running time.

Big O

- Drop the constants!
  - There is no $O(2n)$. This is $O(n)$.
- Drop the lower-order terms!
  - There is no $O(n^2 + n)$. This is $O(n^2)$.
  - When combining growth functions, higher order wins.
- Examples
  - $T(n) = 999n + n^2 \rightarrow O(n^2)$
  - $T(n) = 6n^3 + 45n \rightarrow O(n^3)$

Common Orders of Growth

- $O(1)$ constant
- $O(\log n)$ log
- $O(n)$ linear
- $O(n^2)$ quadratic
- $O(2^n)$ exponential

great

terrible

Common Orders of Growth

Order of Growth / Data Size	1	log n	n	n^2	2^n
10	1	4	10	100	1024
100	1	7	100	10,000	10^{30}
1000	1	10	1000	1,000,000	10^{301}

Common Orders of Growth

Order of Growth / Data Size	1	log n	n	n^2	2^n
10	1	4	10	100	1024
100	1	7	100	1000	10^{30}
1000	1	10	1000	1000000	10^{301}

Problem size multiplied by 10… Running time multiplied by 10.

Common Orders of Growth

Order of Growth / Data Size	1	log n	n	n^2	2^n
10	1	3	10	100	1024
100	1	7	100	10,000	10^{30}
1000	1	10	1000	1,000,000	10^{301}

Problem size multiplied by 10… Running time multiplied by 100.

Common Orders of Growth

Order of Growth / Data Size	1	log n	n	n^2	2^n
10	1	4	10	100	1024
100	1	7	100	10,000	10^{30}
1000	1	10	1000	1,000,000	10^{301}

Problem size multiplied by 10… Running time multiplied by… A LOT.

Common Orders of Growth

Order of Growth / Data Size	1	log n	n	n^2	2^n
10	1	4	10	100	1024
100	1	7	100	10,000	10^{30}
1000	1	10	1000	1,000,000	10^{301}

Still growth! Just less growth! The growth is *slower*.

Example: O(1) Constant

- Problem: print the capacity of an array.
- Solution:

```
System.out.println(arr.length);
```

- For an array of size 10 (n=10), this problem requires a single statement.
  - For n=100, same thing.
  - For n = 1,000, same thing.
- As the problem size changes, the solution time remains the same.
- O(1) solutions are *constant.*
  - As the data set grows, the time required to solve the problem does not change.
- Excellent execution time! But you can't do anything too exciting…

Example: O(n) Linear

- Problem: print all elements in an array.
- Solution:

```
for(int i=0; i<arr.length; i++)
    System.out.println(arr[i]);
```

- For an array of size 10 (n=10), this problem requires looping through the array and printing each element. This is *essentially* 10 statements.
  - For n=100, this requires 100 statements.
  - For n = 1,000, this requires 1,000 statements.
- As the problem size is *10, the solution time is *10.
- O(n) solutions are *linear.*
  - As the data set grows, the time required to solve the problem grows *at the same rate.*

– Linear is considered very good efficiency!

Example: $O(n^2)$ Quadratic

- Problem: print all elements in a square two-dimensional array (a matrix). There are n rows and n columns.
- Solution:

```
for(int i=0; i<arr.length; i++)
    for(int j=0; j<arr[0].length; j++)
        System.out.println(arr[i][j]);
```

- For a matrix of size 2x2, this problem requires a nested loop that will invoke 4 print statements.
    - For a 4x4 matrix, this requires 16 statements.
    - For am 8x8 matrix, this requires 64 statements.
- As the problem doubles, the solution time is multiplied by 4.
- $O(n^2)$ solutions are *quadratic*
    - As the data set grows, the time required to solve the problem grows ***faster***.
- Quadratic is not a good efficiency for large data sets.

Determining Big O

- The order of growth (Big O) is based on the dominant factor in the running time.
- But wait… we just said we're not going to calculate running time. So how can we know what the dominant factor is in the running time?
- We can examine code to look for certain constructs that affect running time.
- The biggest culprit: LOOPS!

Determining Big O

- Consecutive blocks of code (blocks that follow each other) are considered separately.
  - Evaluate each block on its own and added.
  - Then the highest order will “win out”
  - Do one thing, finish, then do something else. → in these cases, add together the run times.

Determining Big O

- Loops
  - Loops are often the driving factor in growth rate
  - Loops are often dependent on the size of the dataset (meaning based on n, or dataArray.length, or dataList.size())
- Nested Loops
  1. Figure out how many times the inside loop runs.
  2. Figure out how many times the outside loop runs.
  3. Total is inside * outside
  - Do something one full time for every single time you do something else → In these cases, multiple together the run times.

Determining Big O

- Methods Inside of Loops
  - Same rules apply as nested loops: inside * outside
  - If a method is O(n) and it is called inside of an O(n) loop, the whole loop is $O(n^2)$!
- Carefully consider the efficiency of methods you didn't write!
  - Good documentation will describe the efficiency of non-constant methods.
  - Example: The ArrayList API page.

Big O- The Worst Case

- We can evaluate the best, worst, or average/expected case for efficiency.
  - The efficiency can be different depending on the input.
- We usually use the worst case.
  - This is often the same as the average/expected case.
  - If they are different, that will usually be specified.

Examples

```
for(int i=0; i<array.length; i++) {
    for(int j=0; j<array.length; j++) {
        // code independent of n (such as:)
        System.out.println(array[i]);
        System.out.println(“something else”);
        constantMethodCall();
}
```

Examples

```
for(int i=0; i<array.length; i++) {
   for(int j=0; j<array.length; j++) {
      // code independent of n (such as:)
      System.out.println(array[i]);
      System.out.println(“something else”);
      constantMethodCall();
}
```

$$O(n^2)$$

Examples

```
for(int i=0; i<n; i++) {
   // code independent of n
}
for(int i=0; i<n; i++) {
   // code independent of n
}
```

Examples

```
for(int i=0; i<n; i++) {
   // code independent of n
}
for(int i=0; i<n; i++) {
   // code independent of n
}
```

O(n)

Examples

```
for(int i=0; i<array.length; i++) {
    for(int j=0; j<array.length; j++) {
        for(int k=0; k<100; k++) {
            // code independent of n
}
```

Examples

```
for(int i=0; i<array.length; i++) {
    for(int j=0; j<array.length; j++) {
        for(int k=0; k<100; k++) {
            // code independent of n
}
```

$O(n^2)$

Examples

```
for(int i=0; i<array.length; i++) {
    for(int j=0; j<array.length; j++) {
        for(int k=0; k<100; k++) {
            // code that is O(n)
}
```

Examples

```
for(int i=0; i<array.length; i++) {
    for(int j=0; j<array.length; j++) {
        for(int k=0; k<100; k++) {
            // code independent of n
}
```

$O(n^3)$

Examples

```
// myList is type ArrayList

for(int i=0; i<myList.size(); i++) {
   System.out.println(myList.get(i));
}
```

Examples

```
// myList is type ArrayList

for(int i=0; i<myList.size(); i++) {
   System.out.println(myList.get(i));
}
```

O(n)

Examples

```
// myList is type ArrayList

for(int i=0; i<myList.size(); i++) {
   boolean contains =
     myList.contains(Integer.valueOf(i));

}
```

Examples

```
// myList is type ArrayList

for(int i=0; i<myList.size(); i++) {
   boolean contains =
     myList.contains(Integer.valueOf(i));

}
```

$O(n^2)$

Big O in the Real World

- Small data sets
  - Inefficient algorithms are not a problem with small data sets (but are a problem with large data sets)
  - For some data sets, an O(n) algorithm can be faster than an O(1) algorithm!
- Constants matter
  - Although the same order of growth, if you can make your code run in 5n this is realistically better than 100n!

Search

Searching

- Searching is the process of finding a target element within a group of items (called a *search pool*).
- The target may or may not be in the search pool.
- We want to perform searching efficiently, minimizing the number of comparisons we make.

Searching

- Search requires a way to compare items.
- Typically, this will be the overridden equals method.

LINEAR SEARCH

Linear Search

- A linear search begins at one end of a list and examines each element in order.
- Either the item is found or we reach the end of the list.
- Linear search is O(n).

Linear Search

- Linear search can be performed on sorted or unsorted data.
  - If performed on sorted data, we can be more efficient!
- Linear search can be implemented iteratively or recursively.
- Linear search can work for arrays or linked node implementations.

Linear Search- Iterative

```
boolean found=false;
for (int i=0; i<data.length; i++) {
   if (target.equals(data[i]) ) {
      found = true;
   }
}
return found;
```

Linear Search- Improved Iterative

```
boolean found=false;
for (int i=0; i<data.length && !found; i++) {
    if (target.equals(data[i]) ) {
        found = true;
    }
}
return found;
```

- Still O(n), but more efficient in the real world.
- We could also use a break or return inside the conditional.

Linear Search- Improved Iterative for Sorted Lists Only

```
boolean found=false;
boolean pastIt = false;
for (int i=0; i<data.length && !found && !pastIt; i++) {
   if (target.equals(data[i]) ) {
      found = true;
   } else if(target.compareTo(data[i]) < 0)) {
      // target is smaller than the data- so we should
      // have seen it by now- it’s not in the data
      pastIt = true;
   }
}
return found;
```

- Still O(n), but more efficient in the real world.
- What must be true of the type of objects in the array?

Linear Search- Recursive

```
boolean linearSearch(int first, int last,
                      Object[] data, Object target) {
   if(first > last) {
      return false; // indices cross over
   } else if(target.equals(data[first])) {
      return true;  // we found it!
   } else {
      return linearSearch(first+1, last, target, data);
      // keep looking
   }
}
```

Linear Search- Recursive

```
boolean linearSearch(int first, int last,
                     Object[] data, Object target) {
   if(first > last) {
      return false; // indices cross over
   } else if(target.equals(data[first])) {
      return true;  // we found it!
   } else {
      return linearSearch(first+1, last, target, data);
      // keep looking
   }
}
```

- Can you modify this to be more efficient for a sorted list?

Example

- Review the search code and examples.

Linear Search ERROR!

- This is a common mistake!

```
boolean found;
for (int i=0; i<length; i++)
    if (searchValue.equals(entry[i]) ) {
        found = true;
    } else {
        found = false;
    }
}
return found;
```

BINARY SEARCH

Binary Search

- A *binary search* assumes the list of items in the search pool is already sorted.
- Binary search eliminates a large part of the search pool with a single comparison.
  – Each comparison eliminates about half of the remaining data.
- Binary search is O(log n).

Binary Search

- Binary search can be implemented iteratively or recursively.
- Binary search does not make sense for linked node implementations!

Binary Search

- A binary search first examines the middle element of the list.
  - If it matches the target, the search is over.
  - If it doesn’t match the target, we only need to search half of the remaining elements (since they are sorted).
- This process continues by comparing the middle element of the remaining viable candidates.
- Eventually, we find the target or exhaust the data.

Hi-Lo Guessing Game

- You think of a number between 1 and 100 and I try to guess it. You tell me if I am too high or low.
- If we play this *smartly*, what would the first guess be? If you make smart guesses, how many guesses will it take (in the worst case)?

Hi-Lo Guessing Game

- The smart first guess is the halfway point, so 50. Then, if 50 is too low, you should guess the new halfway point (75), and so on.

Hi-Lo Guessing Game

Range	Half-Way Value (The Guess)	Value is...	Guess Number
1-100	50	too low	1
51-100	75	too high	2
51-74	62	too low	3
63-74	68	too high	4
63-67	65	too low	5
66-67	66	too low	6
67-67	67	equal!	7

- If the number was 67, it took 7 (smart) guesses to find it.
- This is because the number of times that we guessed the halfway value was log(n) and log(100) = 7.
- We should always be able to guess the number in 7 or less guesses!

Binary Search- Iterative

```
boolean binarysearch(int[] numbers, int target) {
   boolean found = false;
   int first = 0;
   int last = numbers.length - 1;
   while (first <= last && !found) {
      int mid = (first + last) / 2;
      if (numbers[mid] == target) {
         found = true;
      } else if (numbers[mid] < target) {
         first = mid + 1;
      } else { // numbers[mid] > target
         last = mid - 1;
      }
   }
   return found;
}
```

Binary Search- Iterative

```
boolean binarysearch(int[] numbers, int target) {
   boolean found = false;
   int first = 0;
   int last = numbers.length - 1;
   while (first <= last && !found) {
      int mid = (first + last) / 2;
      if (numbers[mid] == target) {
         found = true;
      } else if (numbers[mid] < target) {
         first = mid + 1;
      } else { // numbers[mid] > target
         last = mid - 1;
      }
   }
   return found;
}
```

Binary Search- Recursive

```
boolean binarySearch(int first, int last, int[] data, int target
{
   int mid = ( (last - first) / 2 ) + first;
   if(first > last) {
      return false; // indices cross over
   } else if(target==data[mid]) {
      return true;  // we found it!
   } else if (target < data[mid]) {
      return binarySearch(first, mid-1, target, data);
      // keep looking in left half
   } else { // target > data[mid]
      return binarySearch(mid+1, last, target, data);
      // keep looking in right half
   }
}
```

Binary Search- Recursive

```
boolean binarySearch(int first, int last, int[] data, int target
{
    int mid = ( (last - first) / 2 ) + first;
    if(first > last) {
        return false; // indices cross over
    } else if(target==data[mid]) {
        return true;  // we found it!
    } else if (target < data[mid]) {
        return binarySearch(first, mid-1, target, data);
        // keep looking in left half
    } else { // target > data[mid]
        return binarySearch(mid+1, last, target, data);
        // keep looking in right half
    }
}
```

Binary Search

- Two ways to choose mid:
  - mid = ((last - first) / 2) + first;
  - mid = (first + last) / 2
- The second is simpler. It will work unless large numbers- overflow
- With the algorithms shown, what would be passed in as first and last?

Example

- Review the search code and examples.

Efficiencies of Searches

- Linear search is O(n)
- Binary search is O(log n)

- Binary is much more efficient than linear!
- But binary requires sorted data!
  - Even “fast” sorting algorithms are O(n log n).

- You have to know your data and the task required!

Choosing a Search

- Do you have linked nodes? yes!
  - Use linear (sequential) search!

Choosing a Search

- Do you have an array? yes!
- Is it sorted? no
  - Use linear (sequential) search!
  - Or consider sorting! This depends… How often do you plan to search? (If only a few times, probably not worth the sort.) How often will you have to sort? What is the size of your data?

Choosing a Search

- Do you have an array? yes!
- Is it sorted? yes
  - (This assumes items implement Comparable!)
  - Use binary search (but linear is okay too)

Search in the Java Standard Library

- List interface (implemented by ArrayList and LinkedList) has the indexOf method and the lastIndexOf method. These use linear search.

- Arrays class has a collection of static binary search methods.
  - Throws a runtime exception if the objects are not Comparable!
  - Invoke as: Arrays.binarySearch(data, target);