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 + … + 1 (n times)
  - equal to n
- for i = 1 to n, $\sum i$
  - 1 + 2 + 3 + … + 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	great
• $O(\log n)$	log	
• $O(n)$	linear	
• $O(n^2)$	quadratic	
• $O(2^n)$	exponential	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 that is O(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!

ARRAY AND NODE EFFICIENCIES

Arrays and Linked Nodes

Algorithm	Array	Linked Nodes (with head pointer only)
Traversal (e.g., find frequency, contains method, create and fill a list, etc.)		
Retrieve an element at a specific position		
Insert at the beginning		
Insert at the end		
Insert in the middle		
Delete from the beginning		
Delete from end		
Delete from the middle		

Arrays and Linked Nodes

Algorithm	Array	Linked Nodes (with head pointer only)
Traversal (e.g., find frequency, contains method, create and fill a list, etc.)	O(n)	O(n)
Retrieve an element at a specific position	O(1)	O(n)
Insert at the beginning	O(n)	O(1)
Insert at the end	O(1)	O(n)
Insert in the middle	O(n)	O(n)
Delete from the beginning	O(n)	O(1)
Delete from end	O(1)	O(n)
Delete from the middle	O(n)	O(n)

Linked Nodes with Tail

- How does this change efficiency?

BagInterface Implementations

Method	ArrayBag	LinkedBag
add(T)		
remove()		
remove(T)		
clear()		
getFrequencyOf()		
contains(T)		
toArray()		
getCurrentSize()		
isEmpty()		

BagInterface Implementations

Method	ArrayBag	LinkedBag
add(T)	O(1)	O(1)
remove()	O(1)	O(1)
remove(T)	O(n)	O(n)
clear()	O(n)	O(n)
getFrequencyOf()	O(n)	O(n)
contains(T)	O(n)	O(n)
toArray()	O(n)	O(n)
getCurrentSize()	O(1)	O(1)
isEmpty()	O(1)	O(1)

ListInterface Implementations

Method	AList	LList
add(T)		
add(int, T)		
remove(int)		
getEntry(int)		
replace(T,T)		
contains(T)		
toArray()		
getLength()		
isEmpty()		
clear()		

ListInterface Implementations

Method	AList	LList
add(T)	O(1)	O(n)
add(int, T)	O(n)	O(n)
remove(int)	O(n)	O(n)
getEntry(int)	O(1)	O(n)
replace(int,T)	O(1)	O(n)
contains(T)	O(n)	O(n)
toArray()	O(n)	O(n)
getLength()	O(1)	O(1)
isEmpty()	O(1)	O(1)
clear()	O(n)	O(1)

List Implementations

Action	ArrayList	LinkedList
Direct access- get(i)	O(1)	O(n)
Adding/removing beginning	O(n)	O(1)
Adding/removing middle	O(n)	O(n) O(1) with iterator
Adding/removing end	O(1)	O(1)

ESTIMATION

Orders of Magnitude

- Measuring efficiency requires you to think about orders of magnitude.
  - $O(n)$, $O(n^2)$, etc.
- *Estimating* also relates to this idea.
- When you are asked to estimate a value, think about getting it right *within an order of magnitude.*
  - This often means within a power of 10.

Estimating Tasks

- These are classic interview questions.
  - How many piano tuners are in Chicago?
  - How much should you charge to wash all the windows in Seattle?
  - How much does the Empire State Building weight?
- No one really wants to know the answer!
  - They want to see your thought process!
  - How do you break down and solve a problem?

How to Estimate

- Break the problem into smaller parts.
  - Often you can estimate to some other quantity to help you along the way.
- Make (and state) your assumptions.
    - Be clear about what you are assuming but don’t know!
- Use your general knowledge of the world.

Example

- How many piano tuners are there in Chicago?

What do I need to know?

- How many pianos are there in Chicago?
- How often do people tune their piano?
- How long does it take to tune a piano?

- This would answer how many are needed. Then we can estimate how many there are.

Example

- How many pianos are there in Chicago ?
  - How many people are there in Chicago?
  - What percentage of people have Pianos?
- How often do people tune their piano?
- How long does it take to tune a piano?

Example

- How many pianos are there in Chicago ?
  - How many people are there in Chicago?
    - Keep in mind orders of magnitude!
    - 100,000? 1,000,000? 10,000,000?
    - Think about the word population, then the US, then consider that Chicago is a large city in the US.
  - What percentage of people have Pianos?
- How often do people tune their piano?
- How long does it take to tune a piano?

Example

- How many pianos are there in Chicago ?
  - How many people are there in Chicago?
  - What percentage of people have Pianos?
    - Orders of magnitude: 1%, 10%, 100%?
- How often do people tune their piano?
- How long does it take to tune a piano?

Example

- How many pianos are there in Chicago ?
  - How many people are there in Chicago?
  - What percentage of people have Pianos?
- How often do people tune their piano?
  - Orders of magnitude: Hourly? Daily? Weekly? Monthly? Yearly?
- How long does it take to tune a piano?

Example

- How many pianos are there in Chicago ?
  - How many people are there in Chicago?
  - What percentage of people have Pianos?
- How often do people tune their piano?
- How long does it take to tune a piano?
  - Orders of magnitude: An hour? A day? A week?

Example

- How many pianos are there in Chicago?
  - How many people are there in Chicago?
  - What percentage of people have Pianos?
- How often do people tune their piano?
- How long does it take to tune a piano?

- Once we have answers to these, we can just do the math as to how many are needed.
  - How many are there?
  - Orders of magnitude: would there be 1%, 10% fewer tuners than needed? would there be 1%, 10% more tuners than needed?

Example

- How many piano tuners are there in Chicago?

- The answer doesn't matter!
  - If you've estimated well and used good assumptions, your estimate would likely be within an order of magnitude.
  - But, again, this doesn't really matter.
- What matters is reasonableness of your assumptions you made, how you reasoned and broke down the process.