Data Structures

Augustin Cosse.

Spring 2021

February 24, 2021

Equivalence testing

- When working with reference types, there are many different notions of what it means for one type to be equal to another
- At the lowest level, if a and b are reference variables, $a == b$ tests whether a and b refer to the same object
- For many types, there is however a higher level notion of two variables being considered equivalent, even if they do not refer to the same instance of the class
- For example, we typically want two strings to be considered equivalent to each other if they represent identical sequences of characters

Equivalence testing

- To support such a broader notion of equivalence, all object types support a method named **equals**
- Users of reference types should rely on the syntax **a.equals(b)** unless they have a more specific need to test the more narrow notion of identity
- The **equals** method is defined in the Object class (which is a superclass for all object instances) but that implementation reverts to return the value of the expression $a == b$
- Defining a more precise notion of equivalence requires knowledge about a class and its representation

Equivalence testing

- The author of each class has a responsability to provide an implementation of the **equals** method which overrides the one inherited from Object if there is a more relevant definition of the equivalence of two instances.
- For example, The Java `String` class redefines **equals** to test character-for-character equivalence
- Great care must be taken when overriding the notion of equality as the consistency of Java's library depends upon the **equals** method defining what is known as an equivalence relation

Equivalence testing

- An equivalence relation should satisfy the following properties:

  - Treatment of Null: For any nonnull reference variable x, the call x.equals(**null**) should return **false** (that is nothing equals **null**) except **null**
  - Reflexivity For any nonnull reference variable x, the call x.equals(x) should return **true** (that is an object should equal itself)
  - Symmetry: For any nonnull reference variables x and y, the calls x.equals(y) and y.equals(x) should return the same value.
  - Transitivity: For any nonull reference variables x, y and z, if both calls x.equals(y) and y.equals(z) return **true**, then x.equals(z) must return **true** as well.

Equivalence testing

- While these properties might seem intuitive, it might be challenging to make sure they are implemented for some data structures, especially in an object oriented framework.
- Arrays are considered a reference type in Java (although not a class). However, the java.util.Arrays class can be used to provide additional methods when working with arrays

Equivalence testing

- In particular, the following methods are supported:
  - a==b Tests if a and b refer to the same underlying array instance
  - a.equals(b): identical to a==b. Arrays are not a true class type and do not override the Object.equals method
  - Arrays.equals(a, b): Returns true if the arrays have the same length and and all pairs of corresponding elements are "equals" to each other. If the array elements are primitive, then this uses the traditional == to compare values. if elements of the arrays are reference types it makes pairwise comparisons a[k].equals(b[k]).

Equivalence testing

- An additional difficulty with arrays arises due to the fact that they compound objects (i.e. one two dimensional arrays in Java really are one dimensional arrays nested inside a common one dimensional array)

- If we have two different two dimensional arrays in Java that have the same entries, we probably want to think of them as being equal. However since the row of a and the row of b are stored in different memory locations, a call to the method java.util.Arrays.equals(a, b) will return **false** in this case because it tests a[k].equals(b[k]) which uses the object definition

- To support a more natural definition of equivalence, the Arrays class provides the additional method Arrays.deepEquals(a, b) which is identical to Arrays.equals(a, b) except when the elements of a and b are themselves arrays in which case it calls Arrays.deepEquals(a[k], b[k]) rather than a[k].equals(b[k])

Equivalence testing

- Another important class for which equivalence is of prime importance are linked lists

- One approach is to consider two lists as equivalent if they have the same length and contents that are element by element equivalent.

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;
  // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head; // traverse the primary list
  Node walkB = other.head; // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( )))
       return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );}
  return true; // if we reach this, everything matched successfully}
```

Equivalence testing

- An implementation of this idea is given below
- You can see that although we focus on the comparing between two Singly Linked Lists, we make it possible for the method to take an arbitrary Object as an argument

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;
  // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head; // traverse the primary list
  Node walkB = other.head; // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( )))
       return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );}
  return true; // if we reach this, everything matched successfully}
```

Equivalence testing

- However, if the Object class is not of type SinglyLinked (this.getClass), the objects are considered different
- At line 4 we know that the argument is a singly Linked List and we can thus safely cast our Object into a List

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;
  // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head; // traverse the primary list
  Node walkB = other.head; // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( )))
       return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );}
  return true; // if we reach this, everything matched successfully}
```

Equivalence testing

- Note that there is a catch to using generics here. If we use generic types <E>, we cannot determine at runtime whether the two lists have matching types.

- So instead, we turn to a more classic approach, relying on the getElement and equals methods.

```
public boolean equals(Object o) {
  if (o == null) return false;
  if (getClass( ) != o.getClass( )) return false;
  SinglyLinkedList other = (SinglyLinkedList) o;
  // use nonparameterized type
  if (size != other.size) return false;
  Node walkA = head; // traverse the primary list
  Node walkB = other.head; // traverse the secondary list
  while (walkA != null) {
    if (!walkA.getElement( ).equals(walkB.getElement( )))
       return false; //mismatch
    walkA = walkA.getNext( );
    walkB = walkB.getNext( );}
  return true; // if we reach this, everything matched successfully}
```

Cloning data structures

- When cloning data structures, a common expectation is that a copy of an object has its own state and that once made, the copy is independent of the original (i.e changes made to one do not directly affect the other)

- However, when an object has fields that are reference variables pointing to auxiliary objects, it is not always obvious whether a copy should have a corresponding field that refers to the same auxiliary object, or to a new copy of that auxiliary object.

- For example, if a hypothetical `AddressBook` generates instances that represent electronic address books, with contact information (such as phone numbers and email addresses) for friends and acquaintances, How should we envison a copy of this address book?

Cloning data structures

- Should an entry added into one book appear in its copy ? If we change a person's phone number in one book, would we expect that change to be synchronized in the other?
- There is no one-size-fits all answer to questions like this. Instead, each class in Java is responsible for defining whether its instances can be copied, and if so, how the copy is constructed.
- The universal `Object` superclass defines a method named `clone` which can be used to produce what is known as a shallow copy of an object
- Such a copy uses the standard assignment semantics to assign the value of each field of the new object equal to the corresponding field of the existing object that is being copied

Cloning data structures

- A shallow copy is not always appropriate for all classes and therefore Java intentionally disables use of the `clone()` method by declaring it as **protected** and by having it throw a `CloneNotSupportedException` when called.
- The author of a class must explicitely declare support for cloning by formally declaring that the class implements the **Cloneable** interface and by declaring a public version of the **clone()** method.

Cloning data structures

- The public method can simply call the protected one to do the field by field assignment that results in a shallow copy if appropriate.
- However for many classes, the class may choose to implement a deeper version of cloning in which some of the referenced objects are themselves cloned
- In either case however, to define a custom class that implements the Cloneable interface, the class must override the **clone()** method in the Object class.

Cloning data structures

- Consider the following example

```
public class House implements Cloneable, Comparable<House>{
  private int id;
  private double area;
  private java.util.Date WhenBuilt;

 //   ...

  /** Override the protected clone method defined
 in the Object class, and strengthen its accessibility*/

  public Object clone() {
    try{
        return super.clone();
      }
    catch (CloneNotSupportedException ex){
       return null;
}}}
```

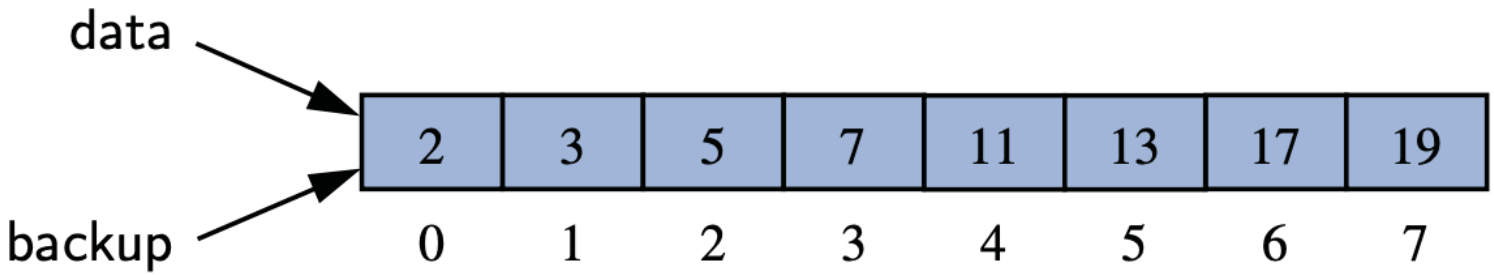
Cloning data structures

- Although arrays support some special syntaxes such as `a[k]`, and `a.length`, it is important to remember that they are objects and that array variables are reference variables. This has important consequences
- To illustrate one of them, consider the code below

```
int[] data = {2,3,5,7,11,13,17,19}
int[] backup;
backup = data;
```

- In this example, the assignment of variable `backup` to `data` does not create any new array. It simply creates a new alias for the same array

Cloning data structures



Cloning data structures

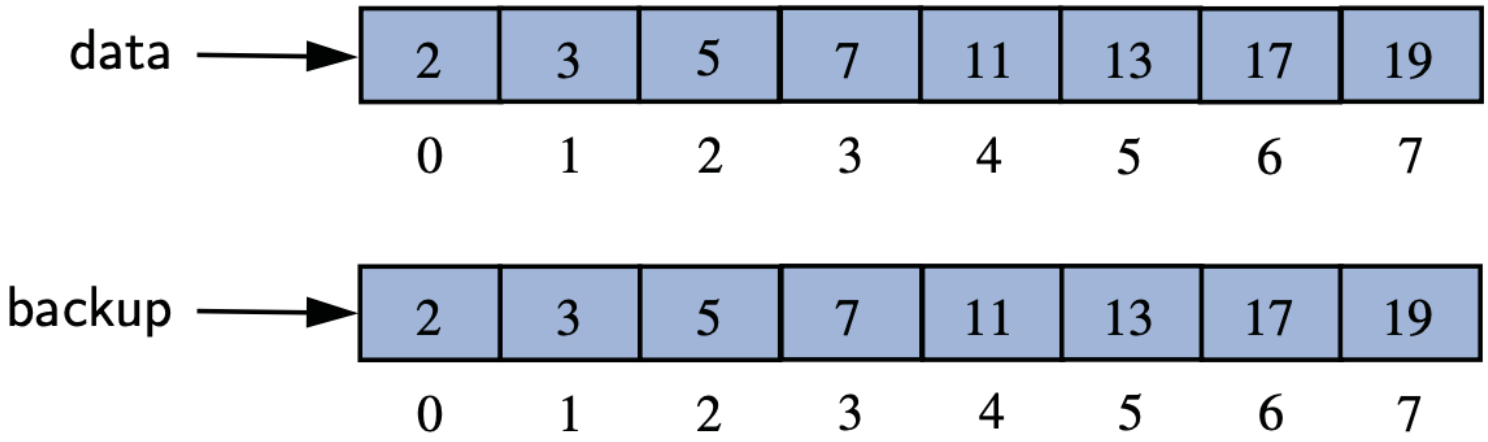
- Instead, if we want to make a copy of the array and assign a reference to the new array to a variable, we should write

```
backup = data.clone();
```

- The **clone()** method when executed on an array initializes each cell of the new array to the value that is stored in the corresponding cell of the original array.

- If we subsequently make an assignment such as `data[4] = 23` in this configuration, the backup array is unaffected.

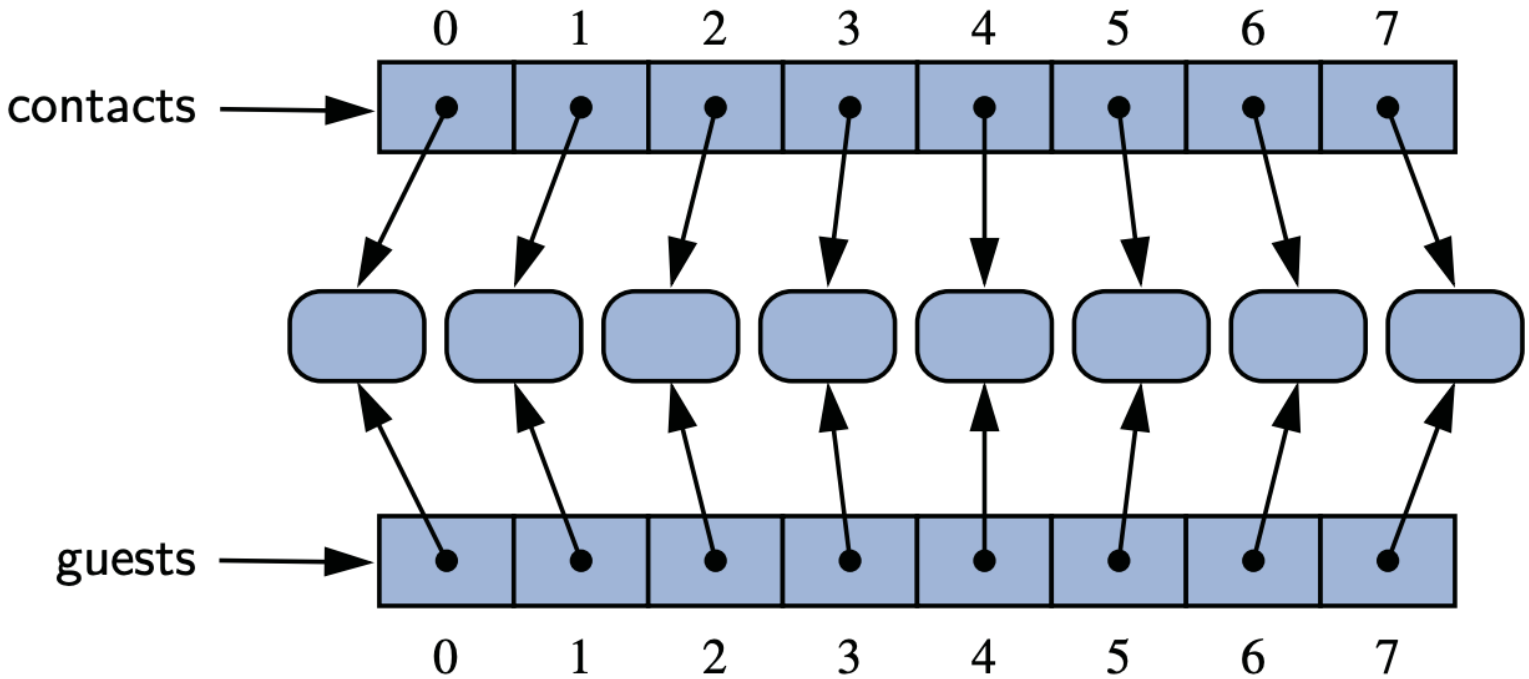
Cloning data structures



Cloning data structures

- There are more considerations when copying an array that stores reference types rather than primitive types
- In this case, the **clone()** method produces a Shallow copy of the array, leading to a new array whose cells refer to the same objects referenced by the first array

Cloning data structures



Cloning data structures

- A deep copy of the **contact** list can be created by iteratively cloning the individual elements
- This is however possible only if the **Person** class is declared as clonable

```
Person[] guests = new Person[contacts.length];
for (int k = 0; k< contacts.length; k++){
    guests[k] = (Person) contacts[k].clone();
// returns Object type
}
```

Cloning data structures

- Because a two-dimensional array is really a one-dimensional array storing other one dimensional arrays, the same distinction between shallow and deep copy exists
- Unfortunately the **java.util.Arrays** class does not provide any "deepClone" method. However, you can always implement your own method by cloning the individual rows of an array

```
public static int[][] deepClone(int[][]original){
  int[] [] backup = new int[original.length][];
  for(int k=0; k< original.length; k++){
    backup[k] = original[k].clone();
    return backup;
}}
```

Cloning data structures

- Just as we did it for arrays, it is possible to implement a **Clone** method for lists
- As we saw before, the first step to make a class cloneable in Java is to have it implement the Cloneable interface. We therefore need the first line

```
public class SinglyLinkedList<E> implements Cloneable{}
```

- As a second step we then need to implement a public version of the **clone()** method of the class. By convention, the method should begin by creating a new instance using a call to **super**.clone() which invokes the method from the Object class
- Since the inherited version returns an Object, we perform a narrowing cast to the `SingledLinkedLIst<E>` type.

Cloning data structures

- Through the use of the `super.clone()` reference, our list can be created as a shallow copy of the original.

- Since the original list has two fields, `size` and `head`, the assignments **other.size = this.size** and **other.head = this.head** have been made

```
public SinglyLinkedList<E> clone()
                          throws CloneNotSupportedException{
  // always use inherited Object.clone() to create initial copy
  SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone();
  if (size > 0){
    other.head = new Node<>(head.getElement(), null);
    Node<E> walk = head.getNext();
    Node<E> otherTail = other.head;
    while (walk != null){
      Note<E> newest = new Node<> (walk.getElement(), null);
      otherTail.setNext(newest);
      otherTail = newest;
      walk = walk.getNext();
}}}
```

Cloning data structures

- While the assignment of the size variable is correct, we cannot allow the new list to share the same head value (otherwise it remains a shallow copy)

- For a nonemtpy list to have an independent state, it must have an entirely new chain of nodes, each storing a reference to the corresponding element from the original list.

```
public SinglyLinkedList<E> clone()
                          throws CloneNotSupportedException{
  // always use inherited Object.clone() to create initial copy
  SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone();
  if (size > 0){
    other.head = new Node<>(head.getElement(), null);
    Node<E> walk = head.getNext();
    Node<E> otherTail = other.head;
    while (walk != null){
      Note<E> newest = new Node<> (walk.getElement(), null);
      otherTail.setNext(newest);
      otherTail = newest;
      walk = walk.getNext();}}}
```

Cloning data structures

- We can therefore create a new head node (line 5) and then perform a walk through the remainder of the original list while creating and linking new nodes for the new list

```
// ...
    Node<E> walk = head.getNext();
    Node<E> otherTail = other.head;
    while (walk != null){
      // make new node storing the same element
      Note<E> newest = new Node<> (walk.getElement(), null);
      // link previous node to this one
      otherTail.setNext(newest);
      otherTail = newest;
      walk = walk.getNext();}}}
```

Part III: Algorithm Analysis

Algorithm Analysis

- In this fourth part, we are interested in the design of "good" data structures and algorithms
- SImply put, a data structure is a systematic way of organizing and accessing data and an algorithm is a step-by-step procedure for performing some task in a finite amount of time
- These concepts are central to good computing
- To be able to classify data structures and algorihtms as good, we must have precise ways of analyzing them

Algorithm Analysis

- The primary analysis tool we will use in this book involves characterizing the running time of algorithms and data structures operations, with space usage also being of interest
- Running time is a natural measure of "goodness" as time is a precious resource. Computer solutions should run as fast as possible.
- In general the running time of an algorithm or data structure operations, increases with the input size, although it may also vary for different inputs of the same size
- The running time is also affected by the harware environment (processor, clock rate, memory, disk,..) and the software environment (operating system, programming language,..)

Algorithm Analysis

- Obviously, all other factors being equal, the running time of the same algorithm on the same input data will be smaller if the computer has for example a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation run on a virtual machine.
- Focusing on the running time as a primary measure of "goodness" requires that we be able to use a few mathematical tools
- We will be interested in characterizing an algorithm's running time as a function of the input size.

Algorithm Analysis

- One way to study the efficiency of an algorithm is to implement it and experiment by running the program on various test inputs while recording the spent on each execution.
- A simple mechanism for collecting such running times in Java is baed on use of the `currentTimeMillis` method of the System class.
- That method reports the number of milliseconds that have passed since a benchmark time known as the epoch
- By recording the time immediately before executing the algorithm and then immediately after, we can measure the elapsed time of an algorithm execution by computing the difference of those times.

Algorithm Analysis

- A typical way to automate this process is given below

```
long startTime = System.currentTimeMillis();
/*runs the algorithm*/
long endTime = System.currentTimeMillis();
long elapsed = endTime - startTime;
```

- For extremely quick operations, Java provides a method `nanoTime` that measures in nanoseconds rather than in milliseconds.

- Because we are interested in the general dependence of the running time on the size and structure of the input, we should perform independent experiments on many different inputs of various sizes

Algorithm Analysis

- We can then visualize the results by plotting the performance of each run of the algorithm as a point with x-coordinate equal to the input size n and y-coordinate equal to the running time t
- To be meaningful, such an analysis would require that we choose good sample inputs and test enough of them to be able to make sound statistical claims about the algorithm's running time

Algorithm Analysis

- The measured times reported by `currentTimeMillis` and `nanoTime` will vary greatly from machine to machine though and may even vary from trial to trail on the same machine
- I.e many processes share use of the computer's central processing unit (CPU) and memory system hence the elapsed time will depend on what other processes are running on the computer when a test is performed
- While a precise running time might not be dependable, experiments are quite useful when comparing the efficiency of two or more algorithms, so long as they are gathered under similar circumstances

Algorithm Analysis

- As an illustration of experimental analysis, consider the following two algorithms for constructing long strings

```
public static String repeat1(char c, int n){
  String answer = "";
  for(int j=0; j<n; j++){
    answer + = c;
  return answer;
}}
```

```
public static String repeat2(char c, int n){
  StringBuilder sb = new StringBuilder();
  for(int j=0; j<n; j++){
    sb.append(c);
  return sb.toString();
}}
```

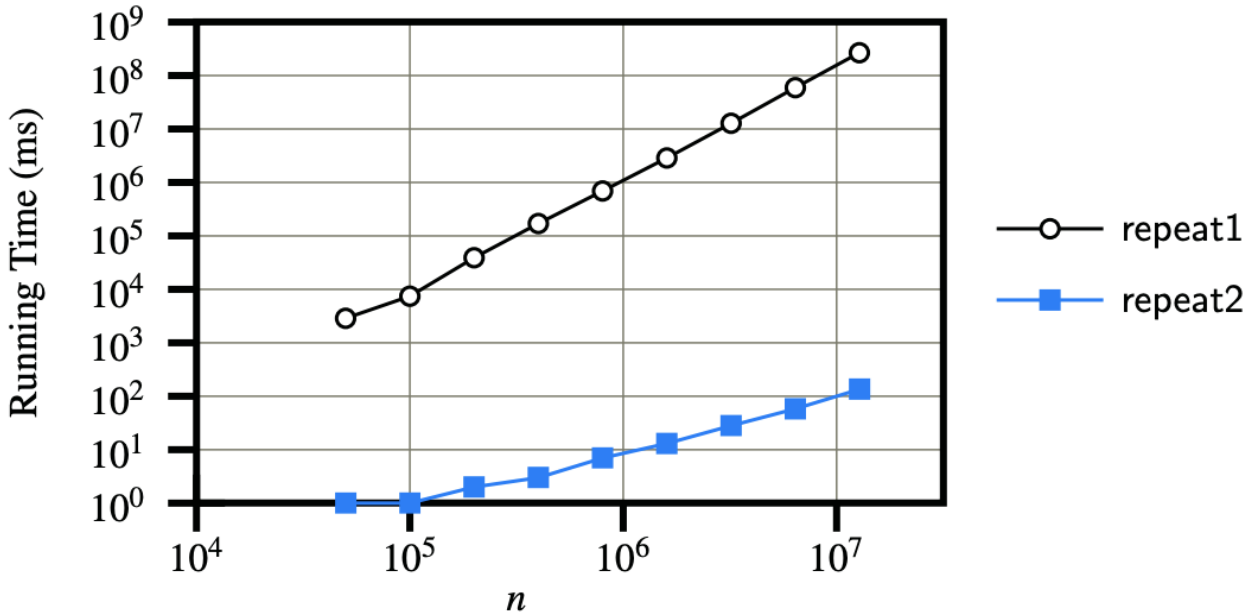
Algorithm Analysis

- As an experiment, we can use `System.currentTimeMillis()` to measure the efficiency of both `repeat1` and `repeat2` for very large strings. The results of the experiments are gathered in the Table below

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

Algorithm Analysis

- The most impressive part of the experiments is how faster the `repeart2` algorithm is relative to the `repeat1` algorithm.
- While `repeat1` is already taking more than 3 days to compose a string of 12.8 million characters, `repeat2` is able to do the same in a fraction of second.



Algorithm Analysis

- While experimental studies of running times are valuable, especially when fine tuning production quality code, there are three major limitations to their use for algorithm analysis:
  - As we saw earlier, experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same harware and software environments
  - Experiments can be done only on a limited set of test inputs; hence they leave out the running times of inputs not included in the experiment (and these inputs might be important)
  - An algorithm must be fully implemented in order to execute it to study its running time experimentally

Algorithm Analysis

- The last requirement is the most serious drawback to the use of those experimental studies
- At an early stage of the design, when considering a particular choice of data structures or algorithms, it would be foolish to spend a significant amount of time implementing an approach that could easily be deepmed inferior by a higher level analysis

Algorithm Analysis

- Our goal will be to develop an approach to analyzing the efficiency of algorithms that:
  - Allows us to evaluate the relative efficiency for any two algorithms in a way that is independent of the hardware and software environments
  - Is performed by studying a high level description of the algorithm without need for any implementation
  - Takes into account all possible inputs

Algorithm Analysis

- To analyze the running time of an algorithm without performing experiments, we perform an analysis directly on a high level description of the algorithm (code fragment or pseudo-code)
- We define a set of primitive operations including
  - Assigning a value to a variable
  - Performing an arithmetic operation
  - Following an object reference
  - Calling a method
  - ...

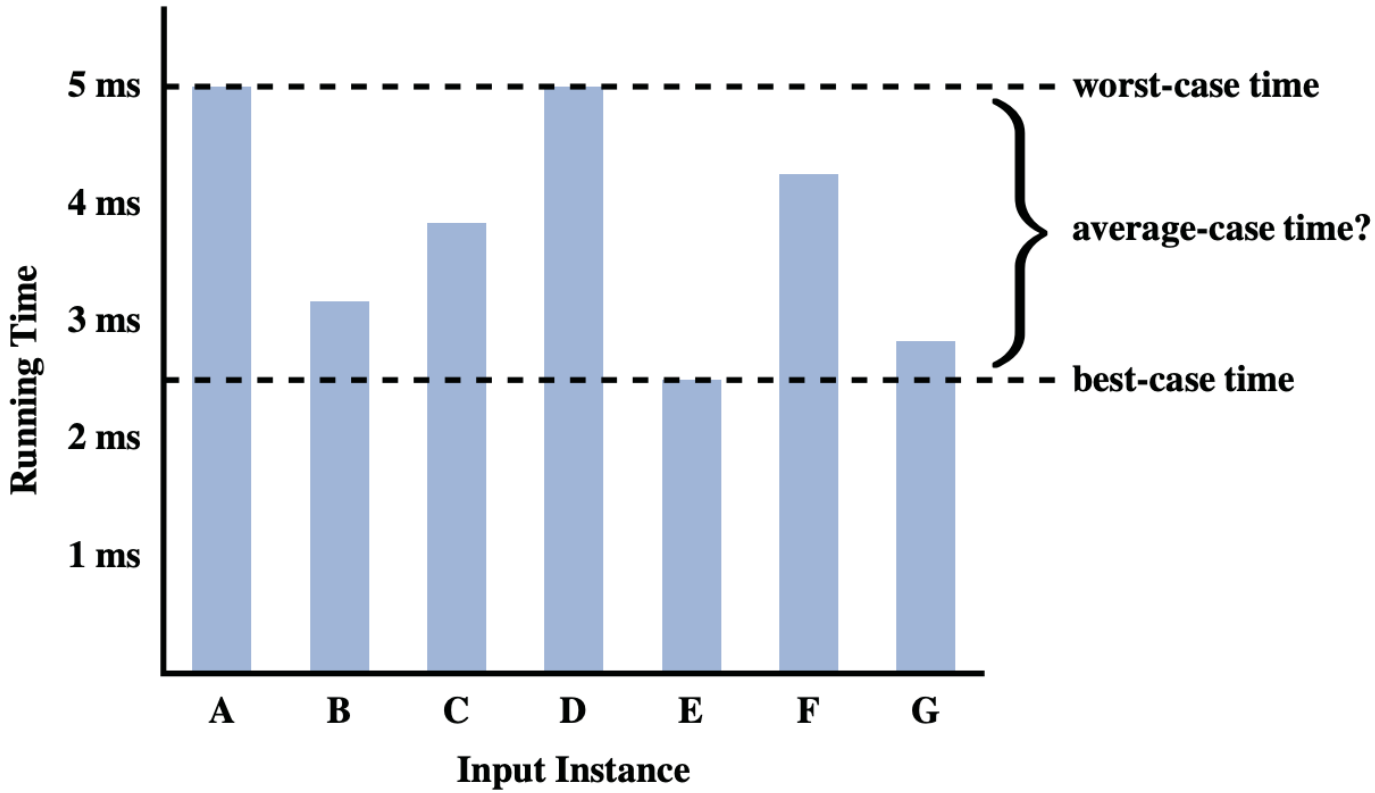
Algorithm Analysis

- Formally, a primitive operation corresponds to a low level instruction with an execution time that is constant
- Ideally, this might be the type of basic operations that is executed by the harware (although many of the primitive operations may be translated to a small number of instructions)
- Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed and use this number t as the running time of the algorithm
- This operation count will correlate to the actual running time on a specific computer

Algorithm Analysis

- To capture the order of growth of an algorithm's running time, we will associate with each algorithm, a function $f(n)$ that characterizes the number of primitive operations that cna be performed as a function of the input size n
- An algorithm may run faster on some inputs, than it does on others of the same size
- For this reason, we may wish to express the running time of an algorithm as a function of the input size obtained by taking the average over all possible inputs of the same size
- Unfortunately such an average case is challenging in practice as it requires to define a probability distribubtion on the set of inputs

Algorithm Analysis

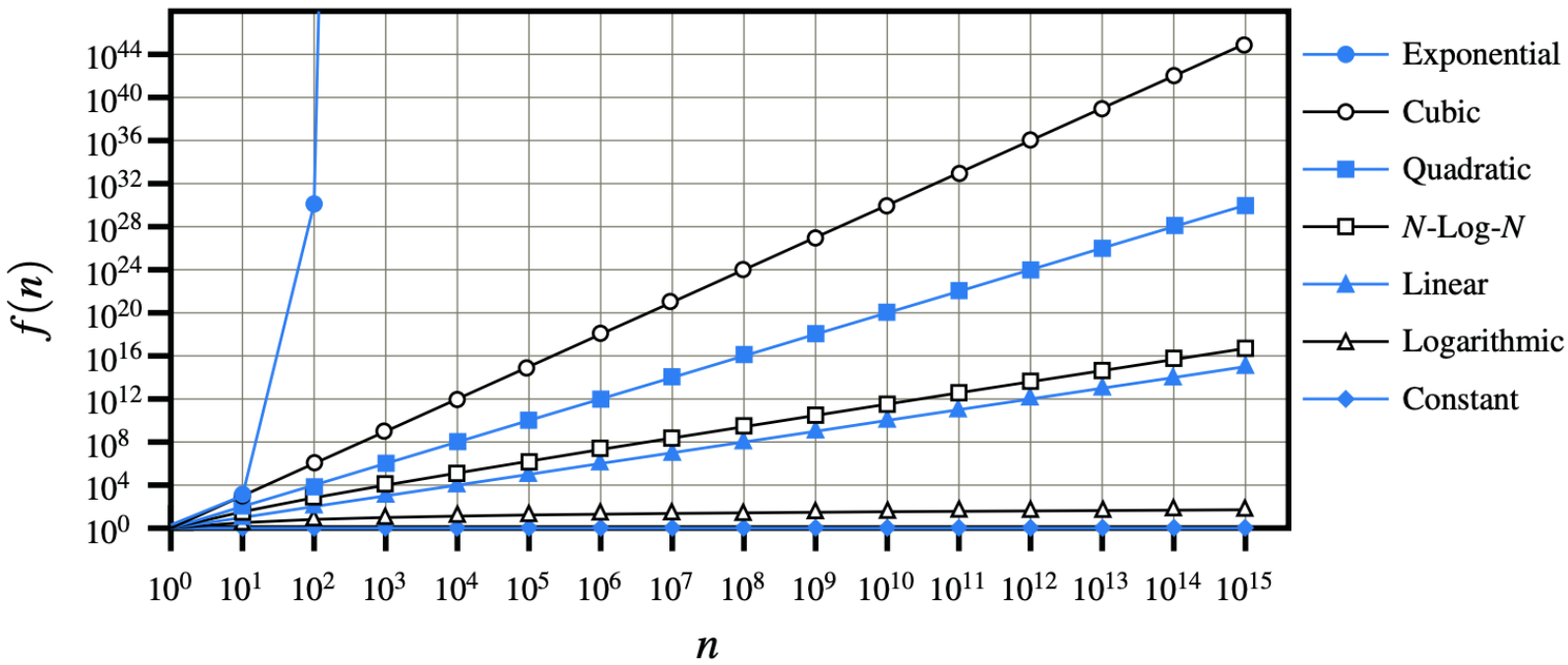


Algorithm Analysis

- In this course, except if specified otherwise, we will characterize the running time in terms of the Worst case complexity.
- Worst case analysis is much easier than average case as it requires only the ability to identify the worst case input
- Moreover, studying the worst case generally leads to the improvement of algorithms as performing well in the worst case means doing well on every input

Algorithm Analysis: complexity functions

- For almost all the analysis we will do during the course, we will restrict to 7 main complexity functions:
  - The constant function
  - The logarithm function
  - The linear function
  - The $n \log n$ function
  - The quadratic function
  - The cubic function and other polynomials
  - The exponential function


10^{44}
10^{40}
10^{36}
10^{32}
10^{28}
10^{24}
10^{20}
10^{16}
10^{12}
10^{8}
10^{4}
10^{0}
$f(n)$
10^{0} 10^{1} 10^{2} 10^{3} 10^{4} 10^{5} 10^{6} 10^{7} 10^{8} 10^{9} 10^{10} 10^{11} 10^{12} 10^{13} 10^{14} 10^{15}
n
Exponential
Cubic
Quadratic
N-Log-*N*
Linear
Logarithmic
Constant

The constant function

- The simplest function we can think of is the constant function,

$$f(n) = c$$

- For any argument n, the function $f(n)$ assigns the value c where c is any constant.
- In particular, it does not matter what the value of n is, $f(n)$ will always be equal to the value c
- The most fundamental constant function $g(n) = 1$. From this function we can thus write $f(n) = c \cdot g(n)$
- The constant function characterizes the number of operations needed to do a basic operation on a computer, like adding two numbers, assigning a value to a variable, or comparing two numbers

The logarithm function

- AAn interesting aspect of the analysis of data structures and algorithms is the presence of the logarithm function,

$$f(n) = \log_b(n)$$

for some $b > 1$. This function is defined as the inverse of a number

$$x = \log_b n, \quad \text{if and only if} \quad b^x = n$$

- The value b is known as the base of the logarithm. The most common base in computer science is 2 as computers store integers in binary. This base is so common that we will typically ommit it from the notations when it is 2. That is to say $\log n = \log_2 n$.

The logarithm function

- The ceiling of a real number x is the smallest integer greater than or equal to x denoted by $\lceil x \rceil$.
- The ceiling of x can be viewed as an integer approximation of x since we have $x \leq \lceil x \rceil \leq x + 1$.
- For an integer n, if we repeatedly divide n by $b$and stop when we get a number less than or equal to 1. the number of divisions is equal to $\lceil \log_b(n) \rceil$.
- Consider the examples below
  - $\lceil \log_3(27) \rceil = 3$ because $((27/3)/3)/3 = 1$
  - $\lceil \log_2 12 \rceil = 4$ because $((((12/2)/2)/2)/2)/2 = 0.75 < 1$
- Note that one can also define the largest integer less than or equal to x using the floor notation, $\lfloor x \rfloor$

The logarithm function

- Together with the logarithm function come several important identities which are recalled below

1. $\log_b(ac) = \log_b(a) + \log_b(c)$
2. $\log_b(a/c) = \log_b(a) - \log_b(c)$
3. $\log_b(a^c) = c \log_b(a)$
4. $\log_b a = (\log_d a)/(\log_d(b))$
5. $b^{\log_d a} = a^{\log_d b}$

- In particular, you can see that the fourth item makes is easy to compute a base two logarithm on a calculator that only has base 10 logarithm.

The linear function

- Another important function is the linear function

$$f(n) = n$$

- The linear function arises in the analysis of algorithms and data structures each time we have do repeat a simple operation for each of the elements in the structure. For example, comparing a number x to each of the elements in an array of size n will require n comparisons

- The linear function also represents the best runnning time we can hope to achieve for any algorithm that processes each of n objects that are not already in the computer's memory (as reading in the n elements in itself already requires n operations)

The $n \log n$ function

- The $n \log n$ function assigns to an input n the value of n times the logarithm base two of n

$$f(n) = n \log n$$

- The function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function
- As a result we greatly prefer an algorithm whose complexity grows like $n \log n$
- Examples of algorithms exhibiting $n \log n$ complexity include the fastest possible algorithms for sorting n arbitrary values

The quadratic function

- Another function that frequently appears in algorithms analysis is the quadratic function

$$f(n) = n^2$$

- The main reason behind the use of the quadratic function in algorithm analysis is the presence of nested loops where the inner loop performs a linear number of operations and the outer loops is performed a linear number of times.

- The quadratic function can also arise in the context of nested loops where the first iteration of a loop uses one operation, the second uses two operations, and so on. In this case, the total number of operations is

$$1 + 2 + 3 + \ldots + (n-1) + n$$

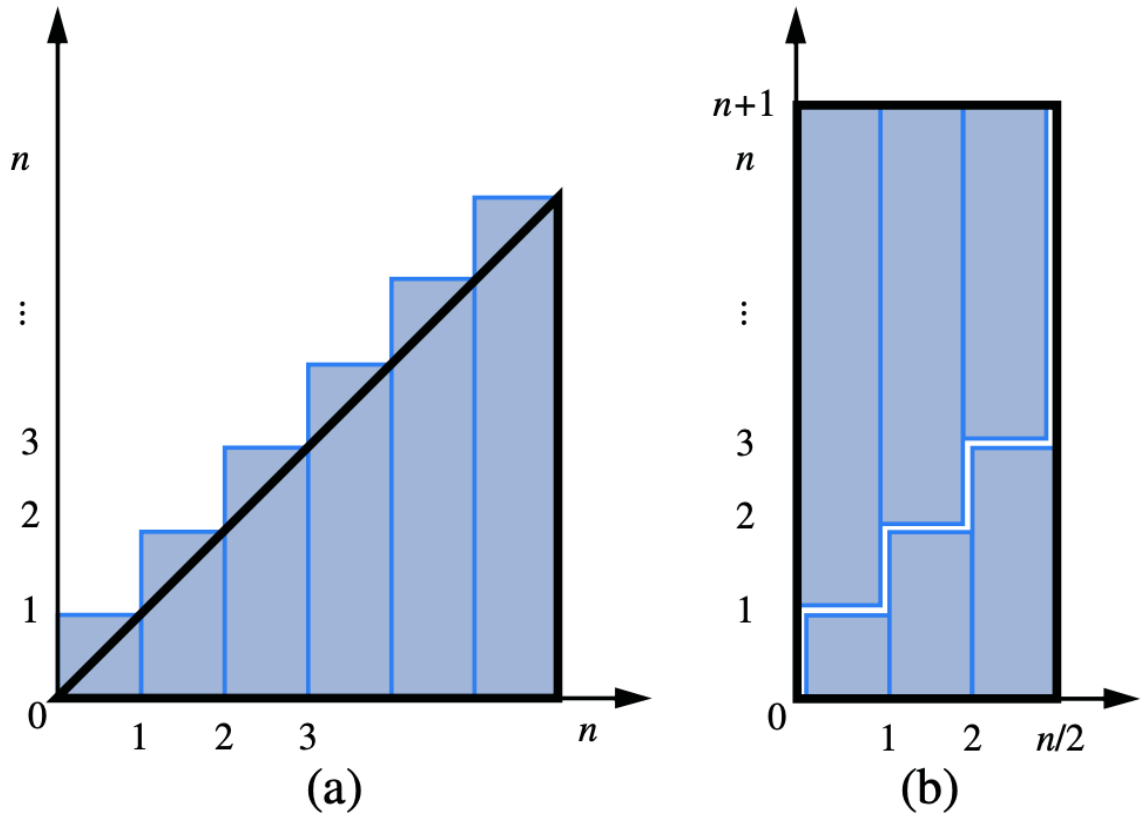
The quadratic function

- In 1787, a German schoolteacher decided to keep his 9 and 10 years old pupils occupied by getting them add up the integers from 1 to 100. Almost immediately however, one of the children claimed to have the answer. The schoolteacher was suspicious for the student only had the final answer written on his slate. But the answer 5050 was correct. The student, Carl Gauss grew up to be one of the greatest mathematician of his time

- Presumably Gauss used the following identity

$$1 + 2 + 3 + \ldots + (n-2) + (n-2) + n = \frac{n(n+1)}{2}$$

The quadratic function



The cubic function and other polynomials

- The cubic function $f(n) = n^3$ appears less frequently
- The linear, quadratic and cubic functions can be viewed as particular instances of the more general polynomial function

$$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

- where $a_0, a_1, \ldots, a_d$ and d (the degree) are constants.
- Obviously, when discussing complexity, polynomial with low degrees are better than polynomial with high degrees

The cubic function and other polynomials

- A notation that appears again and again in the analysis of data structures is the summation which is defined as follows

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + f(a+2) + \ldots + f(b)$$

- Using a summation we can write the sum of integers as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

- Likewise we can write a polynomial $f(n)$ of degree d as

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

The exponential function

- The last function that we will often use is the exponential function

$$f(n) = b^n$$

- b is a positive constant called the base and n is the exponent
- As with the logarithm, the most common base for the exponential function is the $b = 2$ (e.g. an integer word containing n bits can represent all the non negative integers less than 2^n)

The exponential function

- When characterizing the complexity of an algorithm we might want to use the following properties of the exponential function. For positive integers a, b and c, we have
  - $(b^a)^c = b^{ac}$
  - $b^a b^c = b^{a+c}$
  - $b^a / b^c = b^{a-c}$
- Obviously, the exponent extends to fractional values.

The exponential function

- For any integer $n \geq 0$ and any real number $a > 0, \neq 1$, another important relation when characterizing the complexity of an algorithm or data structure is the following:

$$\sum_{i=0}^{n} a^i = 1 + a + a^2 + \ldots + a^n$$

$$= \frac{a^{n+1} - 1}{a - 1}$$

- Summations such as the above are called geometric summations

- Note that the largest unsigned integer that can be represented using n bits is given by $1 + 2 + 4 + \ldots + 2^{n-1} = 2^n - 1$.

Asymptotic Analysis

- In algorithm analysis, we focus on the growth rate as a function of the input size n, taking a "big picture" approach
- As an illustration of this, it is often enough just to know that the running time of an algorihthm grows proportionally to n
- As a consequence, when analyzing algorithms and data structures, we will often rely on mathematical functions that disregard constant factors

Asymptotic Analysis

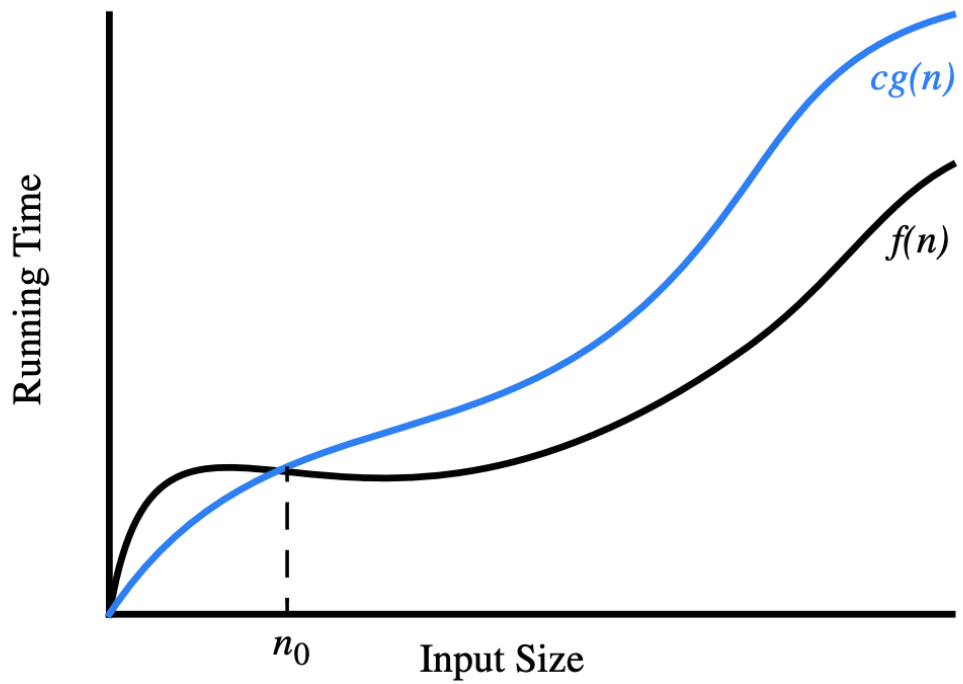
- More generally, we characterize the running times of algorithms by using functions that maps the size of the input, n to values to correspond to the main fators that determines the growth rate in terms of n
- This allows us to analyze algorithms by focusing on the number of primitive operations up to a constant factor rather than getting bogged down in language specific or hardware specific analysis of the exact number of operations executed by the computer

The "Big-Oh" Notation

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq c \cdot g(n), \quad \text{for } n \geq n_0$$

- This definition is often referred to as the "big-Oh" notation for it is sometimes pronounced as "$f(n)$ is big-Oh of $g(n)$".



The "Big-Oh" Notation

- As an example the function $8n + 5$ is $O(n)$. By the big-Oh notation, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $8n + 5 \leq cn$ for every integer $n \geq n_0$. In this case, this works for example for $c = 9$ and $n_0 = 5$

- As another example, the function $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$. Proof: note that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5 + 3 + 2 + 4 + 1)n^4 = cn^4$ for $c = 15$.

- In short, the "big-Oh" notation allows us to ignore constant factors and lower order terms

- Generally speaking if $f(n)$ is a degree d polynomial, $f(n) = a_0 + a_1 n + \ldots + a_d n^d$, and $a_d > 0$, then $f(n)$ is $O(n^d)$.

The "Big-Oh" Notation

- The highest degree term in a polynomial is the term that determines the asymptotic growth rate of that polynomial
- Note that $3\log n + 2$ is $O(\log n)$ and 2^{n+2} is $O(2^n)$.
- In general we should use the big-Oh notation to characterize a function as closely as possible. While it is true that the function $f(n) = 4n^3 + 3n^2$ is $O(n^5)$ or even $O(n^4)$, it is more accurate to say that $f(n)$ is $O(n^3)$.
- The seven functions we just listed are the most common functions used in conjunction with the "big-Oh" notation. We typically use the names of those functions to refer to the running time of the algorithms they charasterize. E.g we talk about quadratic or linear time algorithms.

Big Omega and Big Theta

- Just as the big-Oh notation provides an asymptotic way of saying that a function is "less than or equal to" another function, the Big Omega notation provides an asymptotic notation for saying that a function grows at a rate that is "greater than or equal to" that of another

- Let $f(n)$ and $g(n)$ be functions mapping positive integers to positive real numbers. We say that $f(n)$ is $\Omega(g(n))$ pronounced "$f(n)$ is big-Omega of $g(n)$" if $g(n)$ is $O(f(n))$, that is if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \geq c \cdot g(n), \quad \text{for } n \geq n_0$$

- As an example, $3n \log n - 2n$ is $\Omega(n \log n)$, taking $n_0 = 2$ and $c = 1$.

Big Omega and Big Theta

- The last notation will allow us to say that two functions grow at the same rate, up to constant factors.
- We say that $f(n)$ is $\Theta(g(n))$, pronounced "$f(n)$ is big-Theta of $g(n)$" if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$, that is there are real constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n), \quad \text{for } n \geq n_0.$$

- As an example, we have $3n \log n + 4n + 5 \log n$ is $\Theta(n \log n)$. Proof: $3n \log n \leq 3n \log n + 4n + 5 \log n \leq (3 + 4 + 5)n \log n$ for $n \geq 2$.