Algorithms Analysis

- **What is an algorithm?**
- **Algorithm Analysis**
- **Running time**
- **Experimental Study**
- **Moving Beyond experimental analysis**

Outline

- What is an algorithm?
- Algorithm Analysis
- Running time
- Experimental Study
- Moving Beyond experimental analysis

What is an algorithm?

An **algorithm** is a **step by step procedure** for **performing some task** in a **finite amount of time**. Typically, an algorithm takes input data and produces an output based upon it.

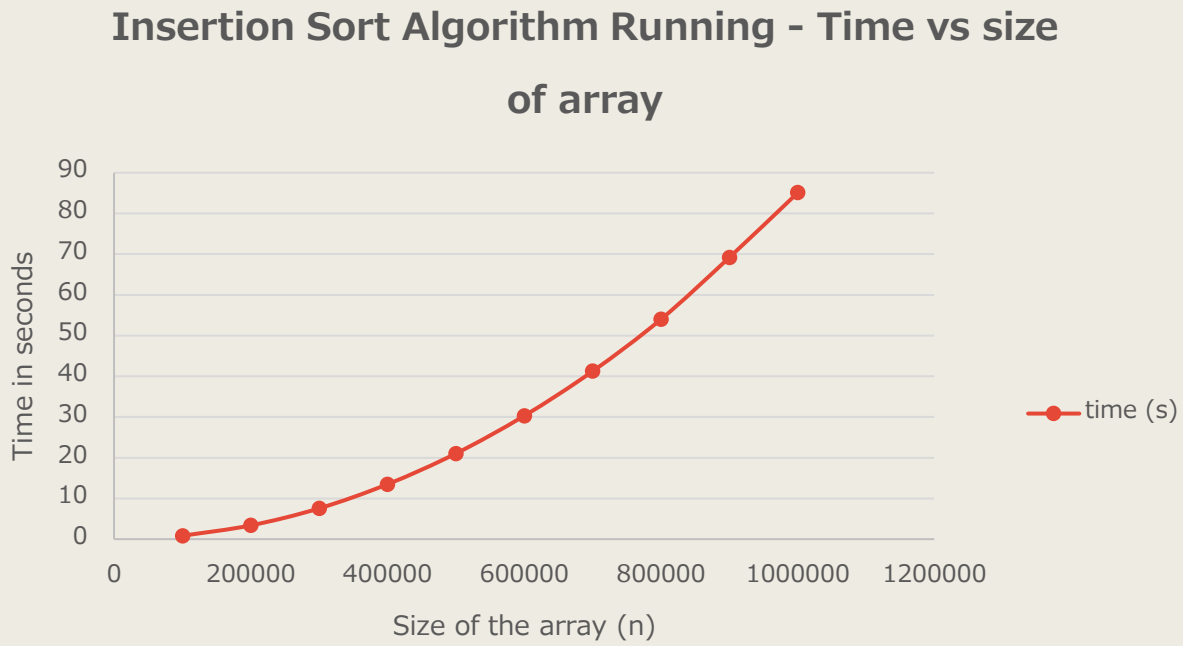
A **data structure** is a systematic way of organizing and accessing data. And therefore, the steps we follow to perform a task depend on that data structure and it affects the amount of time to finish the task.

Algorithm Analysis

- How can we classify data structures operations and algorithms as "**good**" using precise analysis tools?
- Natural measures of "goodness" in algorithms or data structures' operations:
- **Running time** (which is a precious resource – computer solutions should run fast)
- **Space or amount of memory** (ideally computer solutions should consume less memory)

Running Time

- **How long** does it take for an algorithm to finish execution?
- Generally speaking, running time increase with the size of the input. F***or example, the size of the array***
- It could vary for the different inputs of the same size. ***We could have several arrays of the same size, however they are of different datatypes (int, double, objects)***



What affects the running time of an algorithm?

Run time is affected by:

- Size of the input
- Different types of the input of the same size
- Number of primitive operations and their times (we will see what these are later)
- Hardware of the machine (e.g. CPU Speed & clock rate, memory, disk etc..)
- Software Environment (e.g. Operating system, programming language, compiler, interpreter, Running processes/applications, garbage collector)

How to measure running time?

In spite of the possible variations that come from different environmental factors, we would like to focus on **the relationship between the running time of an algorithm and the size of its input**.

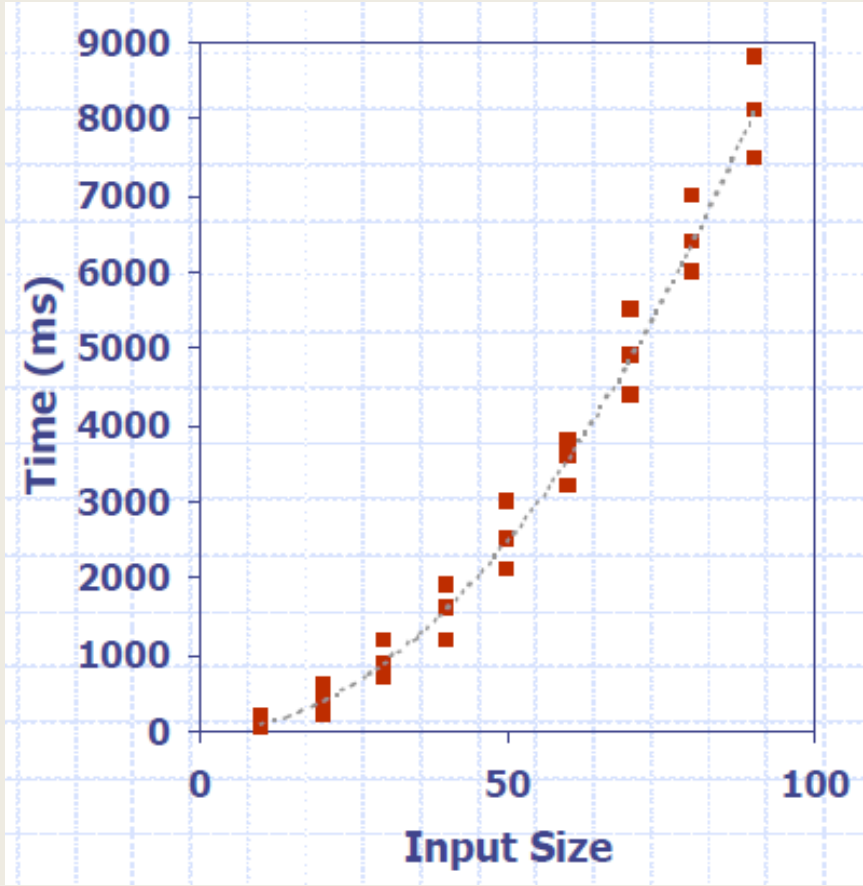
In the end, we would like to characterize the running time of an algorithm or data structure operation as a function $f(n)$ of the input size n.

But what is the proper way of measuring running time?

- **Experimental analysis** (known also as empirical analysis)
- **Theoretical analysis** (known also as mathematical or asymptotic analysis)

Experimental Study

- Write a program implementing a certain algorithm, ***for example sorting the elements of an array.***
- Run the program with inputs of varying size and composition, noting the time needed for each trial/execution. ***Run the sort method with arrays of different sizes, 10, 1000, 10000, 100000,etc. Save the time before the method starts and save it when it finishes, compute the difference, you can tell now how long it took.***
- Plot the results
- Bad news. Difficult to get precise measurements.
- Good news. Much easier and cheaper than other sciences



Measuring the running time experimentally in Java?

- Using the Java **System.currentTimeMillis()** → reports the number of milliseconds that has passed since the epoch (January 1st 1970). 1 sec = 1000 ms
- Using the Java **System.nanoTime()** -> reports the number of nanoseconds that has passed since the epoch (January 1st 1970). nanoTime() can be used for extremely fast algorithms.

Java techniques to measure running times

```
long startTime = System.currentTimeMillis( );          // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis( );            // record the ending time
long elapsed = endTime − startTime;                    // compute the elapsed time
```

Typical approach for timing in milliseconds an algorithm in Java.

```
//1 second is 1e+9 NanoSeconds or 10^9 NanoSeconds
long startTime = System.nanoTime();    //Record the starting time in nanoseconds
/*(run the algorithm here)*/
long endTime = System.nanoTime();      //Record the ending time in nanoseconds
long elapsed = endTime - startTime;
```

Typical approach for timing in nanosecond an algorithm in Java.

Caution

Bare in mind **System.currentTimeMillis() & System.nanoTime()** numbers vary greatly from machine to machine and from trial to trial. Why?

- Faster CPUs runs algorithms faster (less running time)
- Elapsed time will also depend on what other processes (programs) are running during the experiment on your machine & the amount of CPU or memory being consumed which is variable and dynamic.

Challenges of experimental analysis

- It is necessary to implement the algorithm to do the experimental analysis, which may be difficult. In many instances, we need to know the running times in an approximate way without implementing the algorithms (We will see how to do this in another lecture).
- Experimental running times of 2 algorithms are difficult to directly compare unless the experiments are performed in the *exact* same hardware and software environments.
- Results may not be indicative of the running time on other inputs not included in the experiment.

Moving Beyond Experimental Analysis

We are aiming to develop an approach to analyze efficiency of algorithms that:

- Evaluate the relative efficiency of any algorithm ***independent*** of the hardware and software environment.
- Studies algorithms at high level description ***without need to implement them***.
- Take into account ***all possible inputs***.

Counting Primitive Operations

- To analyze the running time of an algorithm without doing experiments, **we analyze the high level description of an algorithm** (could be as a pseudocode or as a code fragment).
- How? We count **primitive operations** *(identified in the next slide)* which are low-level instructions with execution times that are **constant.**
- We use this count as a measure of the running time.

What can be considered as primitive operations?

- Assigning a value to a variable → ***x = 5;***
- Following an object reference → ***Car c = new Car();***
- Performing an arithmetic operation (for example, adding two numbers) → ***x = 2 + b;***
- Comparing two numbers → ***2 > 3***
- Accessing a single element of an array by index → ***a[0]***
- Calling a method → **sum(a,b)**
- Returning from a method → ***return true;***

Measuring Operations as a function of Input Size

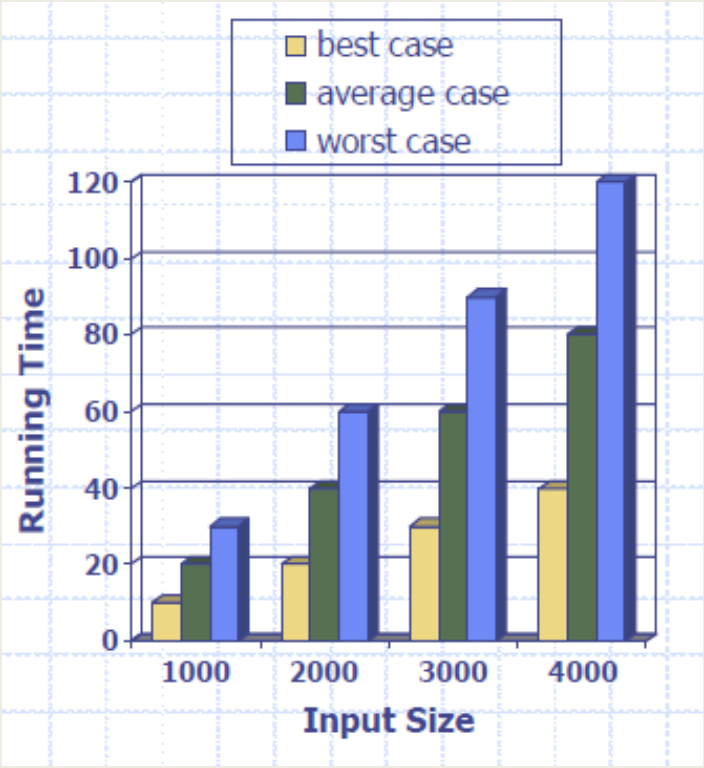
To capture **the order of growth of an algorithm's running time – *how fast or slow the algorithm runs with respect to the input size***, we will associate, with each algorithm, **a function *f* (*n*)** that characterizes **the number of primitive operations** that are performed as **a function of the input size *n***.

Best, Worst, and Average Case

- The reality is that *an algorithm can run faster on some input and slower on other type of input of the same size*.

For a particular problem of size n, we can find:

- **Best case**: the input that can be solved the fastest
- **Worst case**: the input that will take the longest time
- **Average case**: average time for all inputs of the same size.

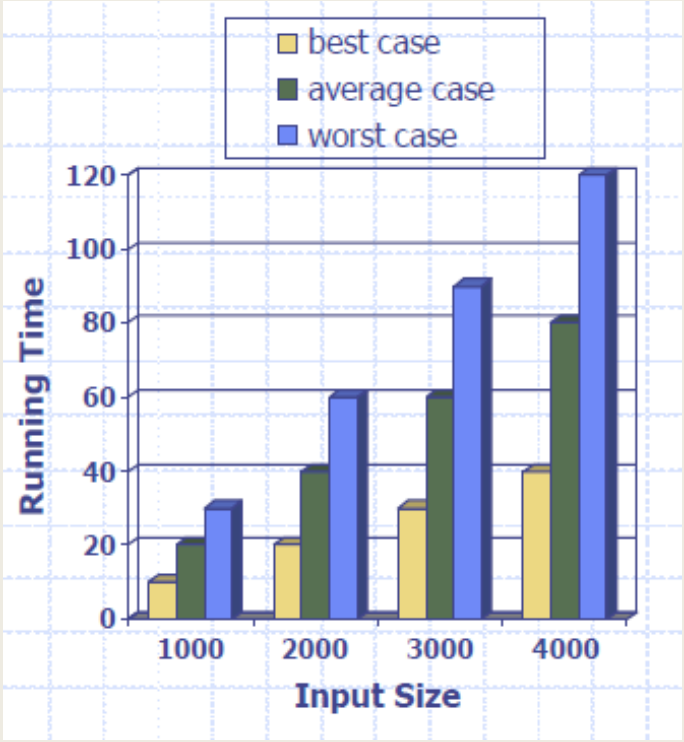


Best, Worst, and Average Case (cont.)

- **Average case analysis** is quite challenging (we may study the probability distribution & probability theory on the set of inputs)
- **Best case** is often trivial and misleading.

We'll focus **on worst case running** time analysis.

- **Easier** to analyze
- **Sufficient** for common applications
- Provides a **minimum performance guarantee** (a **standard of success**) – if an algorithm performs well here, it performs well on any input



Best, Worst, and Average Case (cont.)

If we go back to the example of sorting an array:

Can you define what could be the Best, Average and Worst case?

Best case: array is already sorted

Average case: some elements are in place

Worst case: elements are sorted in reverse order

The Seven Functions Used in the analysis of algorithms

- There are many functions used in the analysis of algorithms.
- We explore the seven most important ones.
- These functions are:
  - Constant ≈ 1
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - N-Log-N $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$

The Constant Function

- The simplest function we can think of is the ***constant function***, that is, $f(n) = c$ for some fixed constant c
  - such as $c = 5$, $c = 27$, or $c = 210$.
- Which means, that for any **argument *n* *(input size)***, the constant function $f(n)$ assigns the value c.
- → It does not matter what the value of n is; $f(n)$ will always be equal to the constant value c.
- **Independent of the input size**

The Constant Function

- Given the following example:

```
public void method (int[] a){
  int i = a[0];
  System.out.println(i);
}
```

- If we count the primitive operations, we have 2:
  - assigning a value to i
  - printing the value of i
- If the array “a” was of size of 10 or 100 or 1000, will the count change?
- **No! → the count is independent of the size → Constant function.**

The Constant Function

- Because it doesn't really matter if c is 5 or 10 or 25 when we try to estimate the running time, the most fundamental constant function is $\boldsymbol{g(n) = 1}$
- Note that any other constant function, $f(n) = c$, can be written as a constant c times $g(n)$. That is, $f(n) = cg(n)$ in this case.

The Logarithm Function

- One of the interesting and surprising functions is the **logarithm function, $f(n) = \log_b n$**, for some constant $b > 1$.
- This function is defined as the inverse of a power, as follows:

  $x = \log_b n$ if and only if $b^x = n$.

- The value b is known as the ***base*** of the logarithm.
- For any base $b > 0$, we have that $\log_b 1 = 0$.
- The most common **base** for the logarithm function in computer science is **2** as computers store integers in **binary**.
- This base is so common that we will typically omit it from the notation when it is 2. That is, $\log n = \log_2 n$.

The Logarithm Function

- Give the following code, how many times will the loop execute?

```
int n = 16;
for(int i = n ; i > 1 ; i /=2){
   //some code
}
```

- The answer is: **4 times**: i = 16, 8, 4, 2
- What if we change n to 8, 32?
  - The answer is log n ⇔ $\log_2 n$

The Linear Function

- Another simple yet important function is the ***linear function***, ***f*(*n*) = *n*.**
- Given an input value *n*, the linear function *f* assigns the value *n* itself.
- This function arises in algorithm analysis any time we have to do a single basic operation for each of *n* elements.
- For example, **comparing a number x to each element of an array of size n** will require n comparisons.

The N-Log-N Function

- The next function we discuss in this section is the ***n-log-n function (linearithmic)***, $f(n) = n\log(n)$
- The function that assigns to an input n the value of n times the logarithm base-two of n.
- This function **grows a little more rapidly than the linear** function and **a lot less rapidly than the quadratic** function; therefore, we would greatly prefer an algorithm with a running time that is proportional to $n\log(n)$

The N-Log-N Function

- Given the code snippet we explained for the log function. What happens if we embed that “for loop” in another like below?

```
for(int j = 1; j <= n ; j++){
   for(int i = n ; i > 1 ; i /= 2){
          //some code
   }
 ]
```

- How many iterations in total we will have?
- We know the inner loop will execute **log(n)** times
- The outer loop will execute **n** times
- Since these are nested loops → **total number of iterations is nlog(n)**

The Quadratic Function

- Another function that appears often in algorithm analysis is the ***quadratic function***, $f(n) = n^2$.
- The main reason why the quadratic function appears in the analysis of algorithms is that there are many algorithms that have **nested loops**, where the **inner loop performs a linear** number of operations and the **outer loop is performed a linear** number of times.
- Thus, in such cases, the algorithm performs $\boldsymbol{n * n = n^2}$ operations.

The Quadratic Function

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, the third uses three operations, and so on. That is, the number of operations is

Proposition 4.3: *For any integer $n \geq 1$, we have:*

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

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

The Quadratic Function

- If we simply change the previous example, so that the inner loop runs "n" times instead of **log(n)** times.

```
for(int j = 1; j <= n ; j++){
   for(int i = n ; i > 1 ; i -= 1){
          //some code
   }
}
```

- We can see that the inner loop will execute from i=n down to 1, that's n times and that will be repeated for every iteration of j

```
j=1 → i=n, n-2,n-2, …  ,2,1
j=2 → i=n, n-2,n-2, …  ,2,1
…
j=n→ i=n, n-2,n-2, …  ,2,1
```

The Cubic Function and Other Polynomials

- Continuing our discussion of functions that are powers of the input, we consider the ***cubic function***, $f(n) = n^3$, which assigns to an input value n the product of n with itself three times.
- The cubic function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned

The Cubic Function and Other Polynomials

- The linear, quadratic and cubic functions can each be viewed as being part of a larger class of functions, the ***polynomials***.
- A ***polynomial*** function has the form

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

- where *a*0,*a*1, . . . ,*ad* are constants, called the ***coefficients*** of the polynomial, and *a* != 0.
- Integer *d*, which indicates the highest power in the polynomial, is called the ***degree*** of the polynomial

The Exponential Function

- Another function used in the analysis of algorithms is the ***exponential function***, $f(n) = b^n$, where b is a positive constant, called the ***base***, and the argument n is the ***exponent***.
- The function $f(n)$ assigns to the input argument n the value obtained by multiplying the base b by itself n times
- As was the case with the logarithm function, the most ***common base for the exponential function in algorithm analysis is $b = 2$***.

Comparing Growth Rates

- The following table sums up the seven common functions used in analysis

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

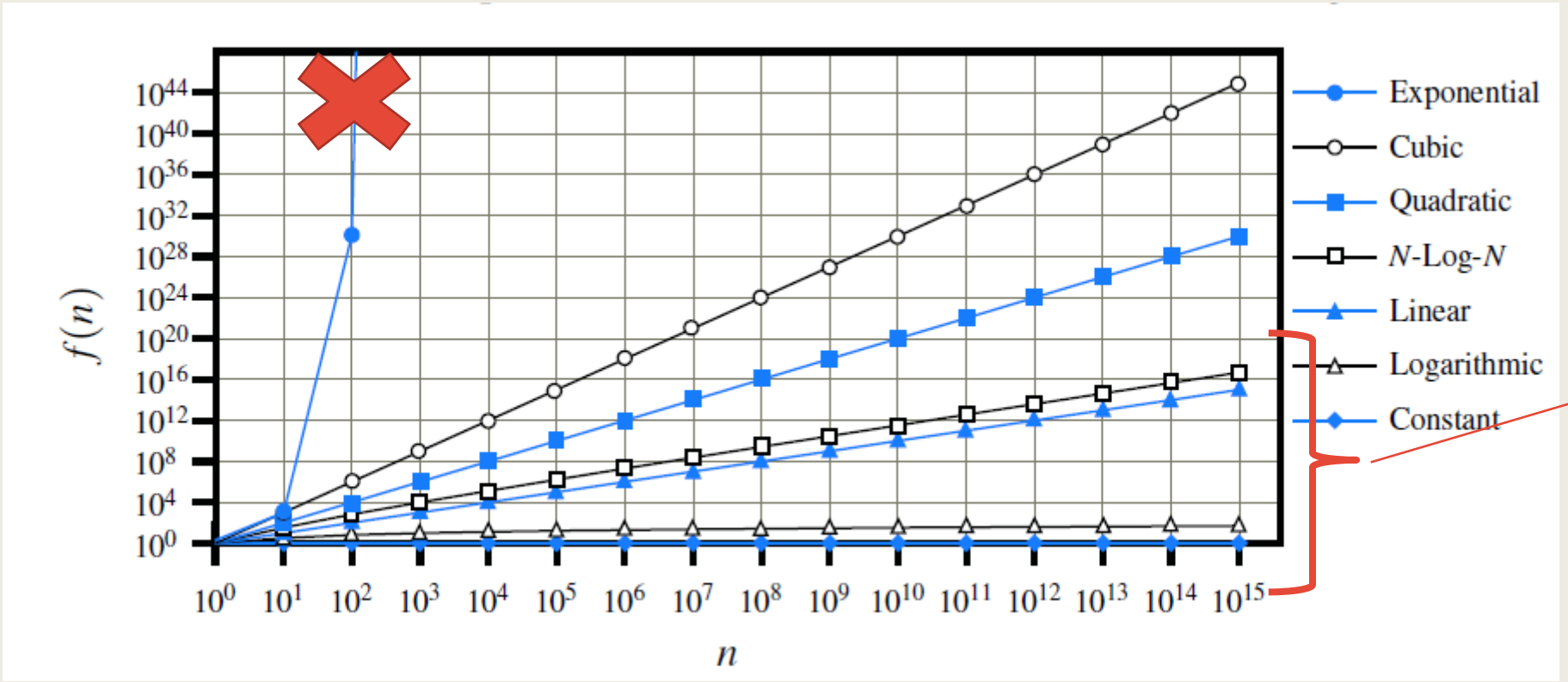
Table 4.2: Seven functions commonly used in the analysis of algorithms. We recall that $\log n = \log_2 n$. Also, we denote with a a constant greater than 1.

- Comparing between the "growth rate" of these functions means comparing how fast they grow; how slow the running time becomes when the input size increases largely.

Comparing Growth Rates

- Remember that these functions are used to measure the running time of an algorithm, so let's see what the curve look like for each of them

Algorithms with exponential running times are infeasible for all but the smallest sized inputs.



We would like data structure operations to run in times proportional to the constant, logarithmic, linear, or *n*-log-*n* time.

Asymptotic Analysis

- In algorithm analysis, we focus on the **growth rate of the running time** as a function of the **input size *n***
- Sometimes, it is often enough just to know that the running time of an algorithm ***grows proportionally to*** *n*.
- We analyze algorithms using a mathematical notation for functions that **disregards constant factors**.
- **So, if we have a running time f(n)=2n, we can still say that the growth rate is linear (n).**

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 $n0 \geq 1$ such that

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

- **We say f(n) is big-Oh of g(n) → g(n) is an upper-bound of f(n) after a certain input size (n0).**
- The big-Oh notation allows us to say that a function $f(n)$ is “less than or equal to” another function $g(n)$ up to a constant factor and in the ***asymptotic*** sense as n grows toward infinity.
- We can say “ $f(n)$ is ***order of*** $g(n)$.” or more formally $f(n) \in O(g(n))$

The “Big-Oh” Notation

- If we look at the figure, we can see that the running time of cg(n) is better or less than the one of f(n), but as of n0, the curve representing cg(n) is always above the curve of f(n), so after n0, cg(n) becomes an upper-bound for f(n)

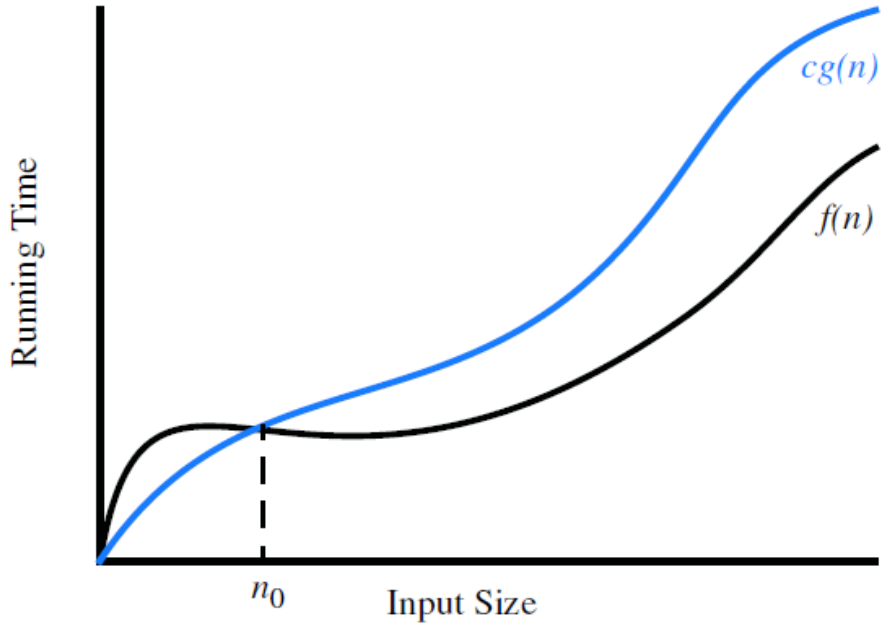


Figure 4.5: Illustrating the “big-Oh” notation. The function $f(n)$ is $O(g(n))$, since $f(n) \leq c \cdot g(n)$ when $n \geq n_0$.

The “Big-Oh” Notation

- To prove one function is of big-Oh of another, it is not always easy to draw the curves and find c and n0.
- So, we’d rather use a more formal approach, using the formula and finding c and n0.

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

The “Big-Oh” Notation

- Given a function f(n) = 8n + 5, we need to prove that f(n) is O(n)
- **What does it really mean?**
- We can say for example, that we have an algorithm, we counted the number of primitive operations, and we got 8n+5.
- We want to estimate the time complexity → estimate the growth rate of that algorithm:
    - Will it grow linearly to the problem size? Quadratically?
- It will be enough to use the order of growth function to express it.
- We can say it will grow linearly, so the time complexity is O(n)

The “Big-Oh” Notation

- To prove it more formally, we are effectively trying to prove that we can find an upper-bound of the form **cn**. Let’s see how:
- To prove that the function f(n) = 8n + 5 is O(n)
- We need to prove that:
- f(n) <= c. g(n) → 8n + 5 <= cn n>=n0 by finding c and n0
- If we are assuming that 8n + 5 <= cn n>=n0
- We can assume that:
- 8n + 5n <= cn n>=n0 → 13n <= cn n>=n0

→ when is this assumption true?

→ when c = 13 and when n0>=1

→if n0=1, they will be equal.

The “Big-Oh” Notation

- Because we were able to find a value for “c” and a value for “n0”, this means our initial assumption is true and that f(n) is of O(g(n))
- **A proof that when a function has some constants, we can disregard them and look at the highest term in function of n.**

The “Big-Oh” Notation

- Similarily, to prove that $5n^4 + 3n^3 + 2n^2 + 4n + 1$ is $O(n^4)$.
- We have to prove that f(n) <= cg(n) for n>=n0
- Where $f(n) = 5n^4 + 3n^3 + 2n^2 + 4n + 1$
- And $g(n) = n^4$
- **Note** that $5n^4 + 3n^3 + 2n^2 + 4n + 1 \leq (5+3+2+4+1)\, n^4 = cn^4$, for $c = 15$, when $n \geq n0 = 1$.
- ***This means again that the most important term is the leading term with the highest power: n^4***
- It might be possible to find one or more values for c and n0, it means that we are finding different upper-bound for the f(n).

Comparative Analysis

- The **big-Oh notation** is widely used to characterize running times and space bounds in terms of some parameter n, which is defined as a chosen measure of **the "size" of the problem**.
- Suppose two algorithms solving the same problem are available:
  - an algorithm A, which has a running time of $O(n)$,
  - and an algorithm B, which has a running time of $O(n^2)$.
- Which algorithm is better?
  - We know that n is $O(n^2)$, which implies that algorithm A is **asymptotically better** than algorithm B, **although for a small value of n, B may have a lower running time than A**.

Comparative Analysis

- We can use the big-Oh notation to **order classes of functions by asymptotic growth rate.**
- Our seven functions are ordered by **increasing growth rate** in the following sequence, such that ***f(n) is O(g(n)) if function f(n) precedes function g(n):***
    - $1, \log n, n, n\log n, n^2, n^3, 2^n$.

Comparative Analysis

Rapid increase of run time

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,096	262,144	1.84×10^{19}
128	7	128	896	16,384	2,097,152	3.40×10^{38}
256	8	256	2,048	65,536	16,777,216	1.15×10^{77}
512	9	512	4,608	262,144	134,217,728	1.34×10^{154}

Table 4.3: Selected values of fundamental functions in algorithm analysis.

Comparative Analysis

- Even if we achieve a dramatic speedup in hardware, we still cannot overcome the handicap of an asymptotically slow algorithm.
- So, designing a faster algorithm is always a better choice than increasing resources.

Comparative Analysis - Some Words of Caution

- The use of the big-Oh and related notations can be misleading, should the **constant factors they "hide" be very large**.
- For example, while it is true that the function **$10^{100}n$ is O(n),** if this is the running time of an algorithm being compared to one whose running time is **10nlog(n)**, we should prefer the **O(nlog(n))**- time algorithm, even though the linear-time algorithm is asymptotically faster.

Comparative Analysis - Some Words of Caution

- What constitutes a “fast” algorithm?
- Any algorithm running in $O(n\log n)$ time (with a reasonable constant factor) should be considered efficient.
- Even an $O(n^2)$-time function may be fast enough in some contexts, that is, when n is small.
- But an algorithm whose running time is an exponential function, e.g., $O(2^n)$, should almost never be considered efficient.

Examples of Algorithm Analysis

- Now that we have the big-Oh notation for doing algorithm analysis, let us give some examples by characterizing the running time of some simple algorithms using this notation.

Examples of Algorithm Analysis- Constant-Time Operations

- *A* is an array of *n* elements

  → A.length is evaluated in constant time.

  → for any valid index j, the individual element, A[j], can be accessed in constant time. A[j] is evaluated in O(1) time for an array.

Examples of Algorithm Analysis- Finding the Maximum of an Array

- Consider the method that returns the maximum element in an array

The algorithm, arrayMax, **runs in $O(n)$ time.**

the loop executes $n-1$ times

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;                                   O(1)
4    double currentMax = data[0];                           O(1)   // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)                                     // consider all other entries
6      if (data[j] > currentMax)                            O(1)   // if data[j] is biggest thus far...
7        currentMax = data[j];                              O(1)   // record it as the current max
8    return currentMax;                                     O(1)
9  }
```

Code Fragment 4.3: A method that returns the maximum value of an array.

Examples of Algorithm Analysis- Finding the Maximum of an Array

- We account for the number of primitive operations being
  - $c' \cdot (n-1)+c''$
- for appropriate constants c' and c'' that reflect, respectively, the work performed inside and outside the loop body
- → arrayMax is $O(n)$.

Examples of Algorithm Analysis

- We will now revisit the algorithms we explained for the Linked List data structure and estimate the big-Oh complexity for these algorithms.

Examples of Algorithm Analysis

Let us examine now some of the operations of the Linked List data structure. The below 5 methods **are independent of the size** of the list **→O(1)**

```
public LinkedList() {
   head = null;
}
public Node getHead() {
   return head;
}
public boolean isEmpty() {
   return (head == null);
}
public void clear()     {
   head = null;
}
public void deleteHead() {
   if (head != null) // if not empty
       head = head.getNext();
}
```

Examples of Algorithm Analysis

The method addHead is also O(1)

The count for the primitive operations is constant no matter what the size of the list is.

```
public void addHead(NodeData item) {
    Node newNode = new Node(item);
    if (isEmpty()) // empty list
        head = newNode;
    else {
        newNode.setNext(head);
        head = newNode;
    }
} // end addHead
```

Examples of Algorithm Analysis

Whereas addTail is O(n) where n is the size of the list because we need to loop through all the elements to find the tail (last node).

```
public void addTail(NodeData item)  {
  Node newNode = new Node(item);
  if (isEmpty()) // empty list
      head = newNode;
  else {
      Node curr = head;
      while (curr.getNext() != null)
           curr = curr.getNext();
      curr.setNext(newNode);
  }
} // end addTail
```

Examples of Algorithm Analysis

- The methods countNodes and print are also O(n) since we need to traverse all n nodes (using the loop).
- In the worst case, the search method is also O(n) and that is the case when either the element we are searching for is at the end of the list or not found in it.
- What about the operations insert, delete? They are both O(n) since in the worst case we could be adding or deleting from the end of the list and this also requires traversing all n nodes.
- Think about the running (execution) time of the operation mergeSorted. What is then time complexity? Is it O(1)? O(n) where n is the size of list 1? O(m) where m is list 2? Or is it a function of m and n?

Examples of Algorithm Analysis

The running time of the operation mergeSorted is actually O(max(n,m)).

The following rules hold for Big Oh notation:

$O(kf(n)) = O(f(n))$ where k is a constant

$O(f(n)) + O(g(n)) = O(f(n) + g(n))$

$O(f(n))\ O(g(n)) = O(f(n)\ g(n))$