- Counting primitive operations:

Assigning a value to a variable
following an object reference
performing an arithmetic operations
comparing two numbers
Accessing a single element of an array by idx
calling a method
returning from a method.

The seven functions -

1) constant function

$$f(n) = c$$

* Note, in any other constant function $f(n) = c$ can be written as

$$f(n) = c\,(g(n))$$

↓

the most fundamental constant function

$$g(n) = 1$$

2) Logarithm function –

$f(n) = \log_b n^c$ for some constant $b > 1$.

(c → power, b → base)

it is inverse of power –

$$x = \log_b n \text{ iff } b^x = n$$

* computing the logarithm function –

Using "ceiling" of a real number, x is the smallest integer $\geq x$, denotated $\lceil x \rceil$

⇒ The cieling of x can viewed as an integer approximation of x since we have:

$$x \leq \lceil x \rceil < x + 1$$

⇒ For a positive integer, n, we repeatedly divide n by b and stop when we get a real number ≤ 1.

→ The number of divisions performed is equal to $\lceil \log_b n \rceil$

examples -

$\lceil \log_3 27 \rceil = 3$ bcz $(27/3)/3/3 = 1$

$\lceil \log_4 64 \rceil = 3$, bcz $(64/4)/4/4 = 1$

The following propositions describes several important identities involve logarithms for any base > 1.

Proposition 4.1 (Logarithm Rules):
Given real numbers $a > 0$, $b > 1$, $c > 0$, and $d > 1$ we have:

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. $b^{\log_d a} = a^{\log_d b}$

Examples –

Example 4.2: *We demonstrate below some interesting applications of the logarithm rules from Proposition 4.1 (using the usual convention that the base of a logarithm is 2 if it is omitted).*

- $\log(2n) = \log 2 + \log n = 1 + \log n$, *by rule 1*
- $\log(n/2) = \log n - \log 2 = \log n - 1$, *by rule 2*
- $\log n^3 = 3 \log n$, *by rule 3*
- $\log 2^n = n \log 2 = n \cdot 1 = n$, *by rule 3*
- $\log_4 n = (\log n)/\log 4 = (\log n)/2$, *by rule 4*
- $2^{\log n} = n^{\log 2} = n^1 = n$, *by rule 5.*

3) Linear function:

$$f(n) = n$$

This is given an input value n, the linear function f assigns the value itself.

4) The N-log-N function -

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

that is, the function that assigns to an input n the n times the logorithm base-two of n.

5) The Quadratic function: nested loops

$f(n) = n^2$

That is, given an input value n, the function f assigns the product of n with itself ('n squared')

⇒ Nested loops and 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 ----

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

Proposition 4.3 — For any integer $n \geq 1$:

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

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

6) The cubic function & other Polynomials.

$f(n) = n^3$

which assigns to an input value n the product of n with itself three times

- Polynomials - The linear, quadratic and cubic functions can each be viewed as being part of a larger class of functions, the polynomials.

The polynomial function has the form:

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

are constants — coefficients

and $a_d \neq 0$. Integer d, which indicates the highest power in the polynomial is called degree of polynomial.

For example, the following functions are all polynomials.

- $f(n) = 2 + 5n + n^2$
- $f(n) = 1 + n^3$
- $f(n) = 1$
- $f(n) = n$
- $f(n) = n^2$

Summations

A notation that appears again and again in the analysis of data structures and algorithms is the ***summation***, which is defined as follows:

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

where a and b are integers and $a \leq b$. Summations arise in data structure and algorithm analysis because the running times of loops naturally give rise to summations.

Using a summation, we can rewrite the formula of Proposition 4.3 as

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

Likewise, we can write a polynomial $f(n)$ of degree d with coefficients $a_0, \ldots, a_d$ as

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

Thus, the summation notation gives us a shorthand way of expressing sums of increasing terms that have a regular structure.

7) Exponential function -

$f(n) = b^n$ → exponent

↓

positive constant, base

Proposition 4.4 (Exponent Rules): Given positive integers a, b, c, we have

1. $(b^a)^c = b^{ac}$
2. $b^a b^c = b^{a+c}$
3. $b^a / b^c = b^{a-c}$

For example, we have the following:

- $256 = 16^2 = (2^4)^2 = 2^{4\cdot 2} = 2^8 = 256$ (Exponent Rule 1)
- $243 = 3^5 = 3^{2+3} = 3^2 3^3 = 9 \cdot 27 = 243$ (Exponent Rule 2)
- $16 = 1024/64 = 2^{10}/2^6 = 2^{10-6} = 2^4 = 16$ (Exponent Rule 3)

...nents that are fractio...

Geometric Sums

Suppose we have a loop for which each iteration takes a multiplicative factor longer than the previous one. This loop can be analyzed using the following proposition.

Proposition 4.5: *For any integer $n \geq 0$ and any real number a such that $a > 0$ and $a \neq 1$, consider the summation*

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

(remembering that $a^0 = 1$ if $a > 0$). This summation is equal to

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

Summations as shown in Proposition 4.5 are called ***geometric*** summations, because each term is geometrically larger than the previous one if $a > 1$. For example, everyone working in computing should know that

$$1 + 2 + 4 + 8 + \cdots + 2^{n-1} = 2^n - 1,$$

for this is the largest unsigned integer that can be represented in binary notation using n bits.

Comparing the Growth Rate:.

4.2.1 Comparing Growth Rates

To sum up, Table 4.2 shows, in order, each of the seven common functions used in algorithm 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.

Ideally, we would like data structure operations to run in times proportional to the constant or logarithm function, and we would like our algorithms to run in linear or n-log-n time. Algorithms with quadratic or cubic running times are less practical, and algorithms with exponential running times are infeasible for all but the smallest sized inputs. Plots of the seven functions are shown in Figure 4.4.

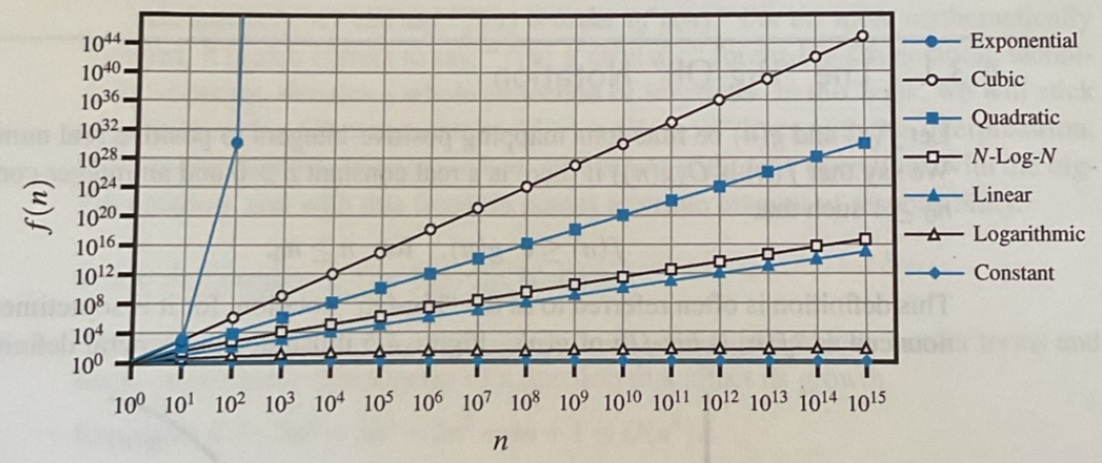


Figure 4.4: Growth rates for the seven fundamental functions used in algorithm analysis. We use base $a = 2$ for the exponential function. The functions are plotted on a log-log chart to compare the growth rates primarily as slopes. Even so, the exponential function grows too fast to display all its values on the chart.

The Ceiling and Floor Functions

When discussing logarithms, we noted that the value is generally not an integer, yet the running time of an algorithm is usually expressed by means of an integer quantity, such as the number of operations performed. Thus, the analysis of an algorithm may sometimes involve the use of the ***floor function*** and ***ceiling function***, which are defined respectively as follows:

- $\lfloor x \rfloor$ = the largest integer less than or equal to x. (e.g., $\lfloor 3.7 \rfloor = 3$.)
- $\lceil x \rceil$ = the smallest integer greater than or equal to x. (e.g., $\lceil 5.2 \rceil = 6$.)

- Recursive Algorithm : The template

→ To solve a problem recursively

- break into smaller problems
- solve sub-problems recursively
- assemble sub-solutions

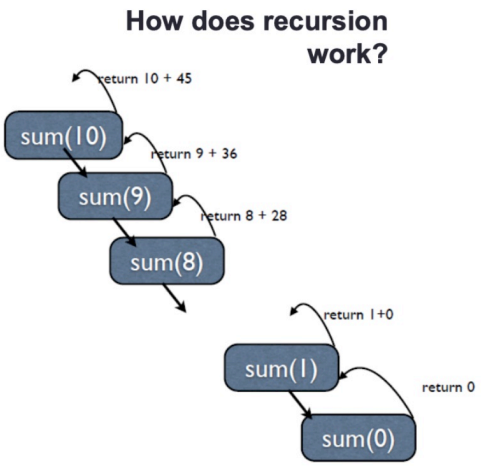
```
recursive-algorithm(input) {
// base case
if (isSmallEnough(input))
   compute the solution and return it
else
// recursive case
   break input into simpler instances input1, input 2,...
   solution1 = recursive-algorithm(input1)
   solution2 = recursive-algorithm(input2)
   ...
   figure out solution to this problem from solution1, solution2,...
   return solution
}
```

Recursion versus iterative approach: Example

Write a function that computes the sum of numbers from 0 to n (i) using a loop; (ii) recursively.

```
// with a loop
int sum (int n) {
  int s = 0;
  for (int i=0; i<=n; i++)
       s+= i;
  return s;
}
```

```
// recursively
int sum (int n) {
  // base case
  if (n == 0) return 0;
  // else
  return n + sum(n-1);
}
```



Recursion versus iterative approach: Example

Recursive is not always better!

```
// Fibonacci: recursive version
int Fibonacci_R(int n) {
   if(n<=0) return 0;
   else if(n==1) return 1;
   else return Fibonacci_R(n-1)+Fibonacci_R(n-2);
}
```

This takes $O(2^n)$ steps! Impractical for large n.

```
// Fibonacci: iterative version
int Fibonacci_I(int n) {
   int fib[] = {0,1,1};
   for(int i=2; i<=n; i++) {
      fib[i%3] = fib[(i-1)%3] + fib[(i-2)%3];
   }
   return fib[n%3];
}
```

This iterative approach is "linear"; it takes $O(n)$ steps.

Merge Sort: Idea

- **Divide:** Divide the unsorted list into two sub-lists of about half the size.
- **Conquer:** Sort each of the two sub-lists recursively until we have list sizes of length 1, in which case the list itself is returned.
- **Merge:** Merge the two sorted sub-lists back into one sorted list.

Merge Sort: Pseudocode

Input: sequence S with n elements, comparator C

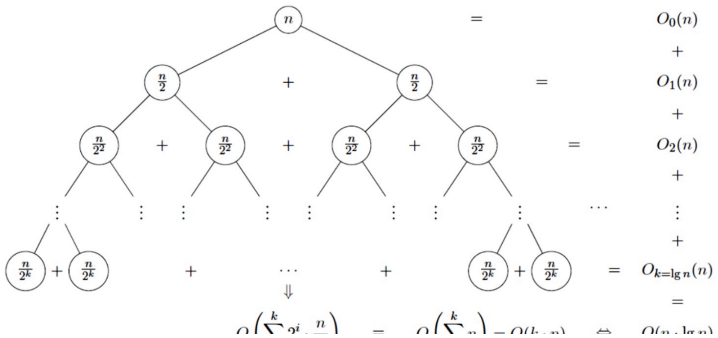
Output: sequence S sorted according to C

```
void MergeSort( S, C )
    if ( Size(S) > 1 ) {
        (S1, S2) ← Partition( S, n/2 )
        MergeSort( S1, C )
        MergeSort( S2, C )
        S ← Merge( S1, S2 )
    }
```

Merge Sort on an input sequence S with n elements consists of three steps:

- **Divide:** partition S into two sequences S_1 and S_2 of about $\frac{n}{2}$ elements each
- **Recur:** recursively sort S_1 and S_2
- **Conquer:** merge S_1 and S_2 into a unique sorted sequence

Merge Sort: Complexity



Merge Sort: Complexity and recurrence relation

The running time of merge sort for a list of length n is

$$T(n) = \begin{cases} b, & \text{if } n = 1 \\ 2 \cdot T\left(\frac{n}{2}\right) + n, & \text{if } n > 1 \end{cases}$$

apply the algorithm to two lists of half the size of the original list

and add the n steps taken to merge the resulting two lists

Merge Sort: Properties and complexity

- Provides **stable sort**, which means that the implementation preserves the input order of equal elements in the sorted output.
- Consistent speed in all type of data sets. Good performance for huge inputs.
- Most common implementation does not sort in place; therefore, requires additional memory space to store the auxiliary arrays.

- Worst case: T(n) = $O(n \log n)$ comparisons
- Best case: T(n) = $O(n \log n)$ comparisons
- Average case: T(n) = $O(n \log n)$ comparisons
- Worst-case space complexity $O(n)$ auxiliary

Quick Sort: Idea

Quick Sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- **Divide**: pick a random element x (called **pivot**) and partition array A into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- **Recur**: sort L and G
- **Conquer**: join L, E and G

Quick Sort: Complexity of partition step

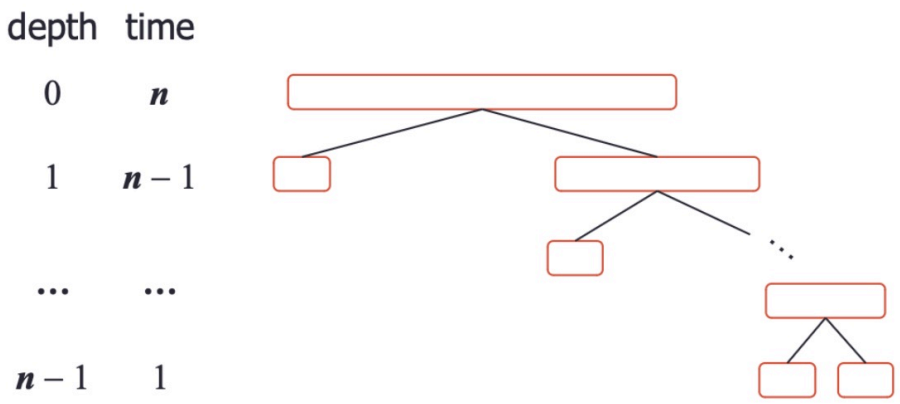
- We partition an input sequence as follows:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x.
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time.
- Thus, the partition step of Quick Sort takes $O(n)$ time.

Quick Sort: The worst-case complexity

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of L and G has size $n-1$ and the other has size L.
- The running time is proportional to the sum

$$n + (n-1) + \cdots + 2 + 1$$

- Thus, the worst-case running time of Quick Sort is $O(n^2)$.



Quick Sort: Properties and complexity

- Not stable because of long distance swapping.
- High probability of choosing the right pivot
  $O(n^2)$ runtime complexity, but typically $O(n\log n)$ time.
- Requires little space and exhibits good cache locality
  Can run **in-place** using only $O(\log n)$ additional storage space to perform sorting.

- Worst case: $\mathrm{T(n)} = O(n^2)$ comparisons
- Best case: $\mathrm{T(n)} = O(n\log n)$ comparisons
- Average case: $\mathrm{T(n)} = O(n\log n)$ comparisons
- Worst-case space complexity $O(n)$ auxiliary
  $O(\log n)$ auxiliary using bounded recursion

Sorting algorithm in C#

List<T>.Sort Method

This method uses the **Introspective Sort** (Intro Sort) algorithm for an array of n elements as follows:

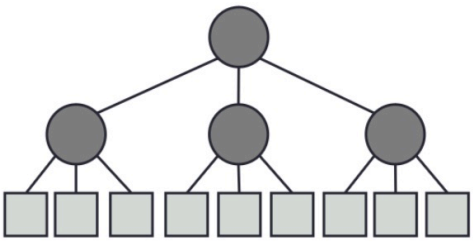
- If the partition size is fewer than 16 elements, it uses Insertion Sort.
- If the number of partitions exceeds $2 \log n$, it uses a Heap Sort algorithm.
- Otherwise, it uses a Quick Sort algorithm.

- This implementation performs an **unstable** sort; that is, if two elements are equal, their order might not be preserved.
- On average, this method is an $O(n \log n)$ operation; in the worst case it is an $O(n^2)$ operation.

– Binary Search –

Divide-and-Conquer Paradigm

- Divide-and-conquer is a general algorithm design paradigm:
  - Divide: divide the input data S in two or more disjoint subsets $S_1, S_2, \ldots$
  - Recur: solve the sub-problems recursively
  - Conquer: combine the solutions for $S_1, S_2, \ldots$ into a solution for S
- The base case for the recursion are sub-problems of constant size



Binary Search

- There is a strategy you can use to limit the number of guesses that you have to make.
- It only works if the array of elements where you are searching for an item is sorted.
- It is an example of *Divide-and-Conquer*.

Binary Search: Let's play a game

$a = \{1, \ldots, 15\}$ consists of all integers from $1, \ldots, 15$.

Player 1 picks a secret number x of a.

Player 2 has to guess x querying in each step a number y.

Answer of Player 1 is either

- found if $x = y$
- x is greater than y
- x is smaller than y

If you are Player 2:
What is your strategy to use the smallest number of queries to reveal x?

Binary Search: Strategy

1. Start with a sorted list
2. Find the midpoint of the part your are searching
3. Compare our search value to the midpoint
   - if the midpoint is our search value we stop!
   - if our number is smaller we now only look between the start of the list and this point
   - if our number is bigger we now only look between this point and the end of the list.
4. If the part we are searching is not empty, go back to step 2
5. If the part we are searching is empty, the value is not in the list

Binary Search: Program

```
1.  // return the position of value in the array A or -1 if it is not present
2.  int binary_search(int A[], int length, int value)
3.  {
4.      // work out the first and last array indexes to search
5.      int imin = 0 ; int imax = length - 1;

6.      // search while [imin,imax] is not empty
7.      while (imax >= imin)
8.      {
9.          // calculate the midpoint using integer division
10.         int midpt = (imin + imax) / 2;
11.         if(A[midpt] == value)
12.         {
13.             return midpt;  // We found it!
14.         }
15.         // Now have to deal with subarrays.
16.         if (A[midpt] < value)
17.         {
18.             imin = midpt + 1;  // change min index to search upper subarray
19.         } else
20.         {
21.             imax = midpt - 1;  // change max index to search lower subarray
22.         }
23.     }
24.     // value was not found - why does this work?
25.     return -1;
26. }
```