Algorithm Analysis*

Xiaofeng Gao

Department of Computer Science and Engineering
Shanghai Jiao Tong University, P.R.China

Algorithm Course @ Shanghai Jiao Tong University

* Special thanks is given to Prof. Yuxi Fu for sharing his teaching materials, and also given to Mr. Shuodian Yu from CS2016@SJTU for producing pictures.

Outline

1. Computational Complexity
   - Theory of Computation
   - Time Complexity
   - Space Complexity
2. Complexity Analysis
   - Estimating Time Complexity
   - Basic Operation and Input Size
   - Best/Worst/Average/Amortized Analysis
3. Searching and Sorting
   - Searching Algorithms
   - Linear Sorting Algorithms
   - Recursive Sorting Algorithms

Theory of Computation

Theory of Computation is to understand the notion of computation in a formal framework.

- *Computability Theory* studies what problems can be solved by computers.
- *Computational Complexity* studies how much resource is necessary in order to solve a problem.
- *Theory of Algorithm* studies how problems can be solved.

In 1936 Alonzo Church published the first precise definition of a calculable function, regarded as the beginning of a systematic development of the Theory of Computation.

Computability vs Complexity

Computability Theory starts from mathematical logic, and discusses the ability to solve a problem in an effective manner.

Famous Computation Models:

- Church (1936): λ-Calculus.
- Gödel-Kleene (1936): Recursive Functions.
- Turing (1936): Turing Machines.
- Post (1943): Post Systems.
- Shepherdson-Sturgis (1963): Unlimited Register Machine.

Church-Turing Thesis: The intuitively and informally defined class of effectively computable functions coincides exactly with the same class $\mathscr{C}$ of computable functions.

Computational Complexity

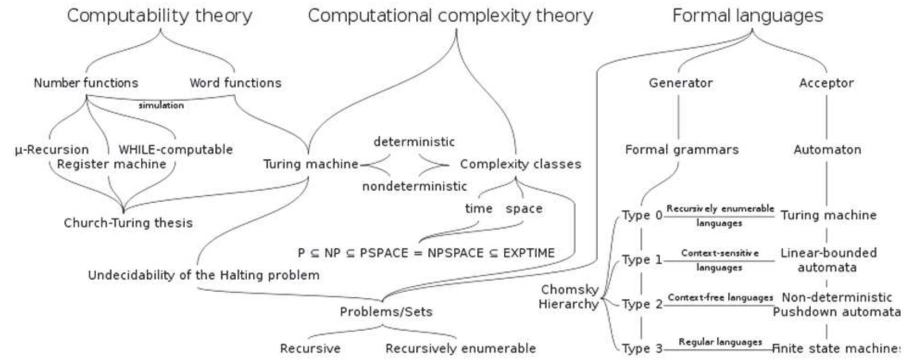
Computational Complexity is to classify and compare the practical difficulty of solving problems about finite combinatorial objects.

- Efficiency is the most important factor.
- Evolved from 1960's, flourished in 1970's and 1980's.

Important Phases:

- *Decision Problem* vs *Search Problem*.
- *Time Complexity* vs *Space Complexity*.
- *Deterministic* vs *Nondeterministic* Turing Machine.
- $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME$.

Relationship Diagram



Halting Problem asks, given a computer program and an input, will the program terminate or will it run forever?

A formal language is defined by means of a formal grammar. Formal language theory studies the syntactical aspects of such languages – that is, their internal structural patterns.

Theory of Algorithm

An algorithm is a procedure that consists of a finite set of *instructions* which, given an *input* from some set of possible inputs, enables us to obtain an *output* through a systematic execution of the instructions that *terminates* in a finite number of steps.

Blackbox: input $\longrightarrow \blacksquare \longrightarrow$ output

Theory of Algorithm includes:

- Algorithmic Thinking: the ability to think in terms of such algorithms as a way of solving problems. It is a core skill people develop when they learn to write their own computer programs.
- Applicability of Algorithm: the domain of objects to which an algorithm is applicable (correctness proof, resource estimation, and theoretical analysis).

Running Time

Running time of a program is determined by:

- input size
- quality of the code
- quality of the computer system
- time complexity of the algorithm

We are mostly concerned with the behavior of the algorithm under investigation on large input instances.

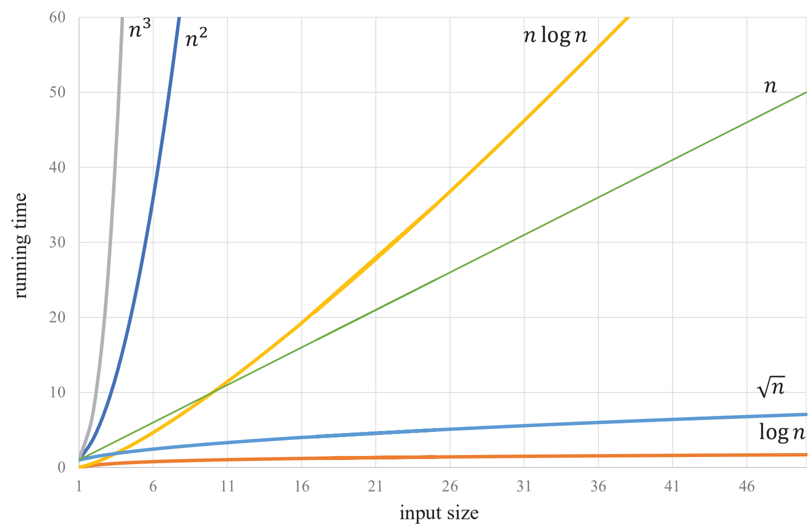
Thus, we may talk about the rate of growth or the order of growth of the running time.

Running Time vs Input Size

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

1s (second) =1,000 ms (millisecond) =10^6 μs (microsecond) = 10^9 ns (nanosecond)

Growth of Typical Functions



Order of Growth

Our main concern is about the order of growth.

- Our estimates of time are relative rather than absolute.
- Our estimates of time are machine independent.
- Our estimates of time are about the behavior of the algorithm under investigation on large input instances.

So we are measuring the *asymptotic running time* of the algorithms.

The O-Notation

The O-notation provides an *upper bound* of the running time; it may not be indicative of the actual running time of an algorithm.

Definition (O-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $O(g(n))$, written $f(n) = O(g(n))$, if

$$\exists c. \exists n_0. \forall n \geq n_0. f(n) \leq cg(n)$$

Intuitively, f grows no faster than some constant times g.

The Ω-Notation

The Ω-notation provides a *lower bound* of the running time; it may not be indicative of the actual running time of an algorithm.

Definition (Ω-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Omega(g(n))$, written $f(n) = \Omega(g(n))$, if

$$\exists c.\exists n_0.\forall n \geq n_0.f(n) \geq cg(n)$$

Clearly $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

The Θ-Notation

The Θ-notation provides an exact picture of the growth rate of the running time of an algorithm.

Definition (Θ-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\Theta(g(n))$, written $f(n) = \Theta(g(n))$, if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Clearly $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

The o-Notation

Definition (o-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $o(g(n))$, written $f(n) = o(g(n))$, if

$$\forall c.\exists n_0.\forall n \geq n_0.f(n) < cg(n)$$

The ω-Notation

Definition (ω-Notation)

Let $f(n)$ and $g(n)$ be functions from the set of natural numbers to the set of nonnegative real numbers. $f(n)$ is said to be $\omega(g(n))$, written $f(n) = \omega(g(n))$, if

$$\forall c.\exists n_0.\forall n \geq n_0.f(n) > cg(n)$$

Definition in Terms of Limits

Suppose $\lim_{n\to\infty} f(n)/g(n)$ **exists**.

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$ implies $f(n) = O(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$ implies $f(n) = \Omega(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ implies $f(n) = \Theta(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ implies $f(n) = o(g(n))$.
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ implies $f(n) = \omega(g(n))$.

A Helpful Analogy

- $f(n) = O(g(n))$ is similar to $f(n) \leq g(n)$.
- $f(n) = o(g(n))$ is similar to $f(n) < g(n)$.
- $f(n) = \Theta(g(n))$ is similar to $f(n) = g(n)$.
- $f(n) = \Omega(g(n))$ is similar to $f(n) \geq g(n)$.
- $f(n) = \omega(g(n))$ is similar to $f(n) > g(n)$.

Complexity Classes

An equivalence relation $\mathcal{R}$ on the set of complexity functions is defined as follows: $f\mathcal{R}g$ if and only if $f(n) = \Theta(g(n))$.

A complexity class is an equivalence class of $\mathcal{R}$.

The equivalence classes can be ordered by $\prec$ defined as follows: $f \prec g$ iff $f(n) = o(g(n))$.

$1 \prec \log\log n \prec \log n \prec \sqrt{n} \prec n^{\frac{3}{4}} \prec n \prec n\log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2}$

Space Complexity

The space complexity is defined to be the number of cells (*work space*) needed to carry out an algorithm, *excluding the space allocated to hold the input*.

The exclusion of the input space is to make sense the sublinear space complexity.

It is clear that the work space of an algorithm can not exceed the running time of the algorithm. That is $S(n) = O(T(n))$.

Trade-off between time complexity and space complexity.

Optimal Algorithm

In general, if we can prove that any algorithm to solve problem Π must be $\Omega(f(n))$, then we call any algorithm to solve problem Π in time $O(f(n))$ an *optimal algorithm* for problem Π.

How to estimate time complexity? Counting the Iterations

Algorithm 1: Count1

Input: $n = 2^k$, for some positive integer k.
Output: *count* = number of times Step 4 is executed.

1 $count \leftarrow 0$;
2 **while** $n \geq 1$ **do**
3   **for** $j \leftarrow 1$ **to** n **do**
4         $count \leftarrow count + 1$;
5     $n \leftarrow n/2$;
6 **return** *count*;

while is executed $k+1$ times; **for** is executed $n, n/2, \ldots, 1$ times

$$\sum_{j=0}^{k} \frac{n}{2^j} = n \sum_{j=0}^{k} \frac{1}{2^j} = n(2 - \frac{1}{2^k}) = 2n - 1 = \Theta(n)$$

Counting the Iterations

Algorithm 2: Count2

Input: A positive integer n.
Output: *count* = number of times Step 5 is executed.

1 $count \leftarrow 0$;
2 **for** $i \leftarrow 1$ **to** n **do**
3     $m \leftarrow \lfloor n/i \rfloor$;
4   **for** $j \leftarrow 1$ **to** m **do**
5         $count \leftarrow count + 1$;
6 **return** *count*;

The inner **for** is executed $n, \lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \ldots, \lfloor n/n \rfloor$ times

$$\Theta(n \log n) = \sum_{i=1}^{n} (\frac{n}{i} - 1) \leq \sum_{i=1}^{n} \lfloor \frac{n}{i} \rfloor \leq \sum_{i=1}^{n} \frac{n}{i} = \Theta(n \log n)$$

Counting the Iterations

Algorithm 3: Count3

Input: $n = 2^{2^k}$, k is a positive integer.
Output: *count* = number of times Step 6 is executed.

1 $count \leftarrow 0$;
2 **for** $i \leftarrow 1$ **to** n **do**
3     $j \leftarrow 2$;
4   **while** $j \leq n$ **do**
5         $j \leftarrow j^2$;
6         $count \leftarrow count + 1$;
7 **return** *count*;

Counting the Iterations

For each value of i, the **while** loop will be executed when $j = 2, 2^2, 2^4, \cdots, 2^{2^k}$.

That is, it will be executed when $j = 2^{2^0}, 2^{2^1}, 2^{2^2}, \cdots, 2^{2^k}$.

Thus, the number of iterations for **while** loop is $k + 1 = \log\log n + 1$ for each iteration of **for** loop.

The total output is $n(\log\log n + 1) = \Theta(n \log\log n)$.

Elementary Operation

Definition: We denote by an "elementary operation" any computational step whose cost is always upperbounded by a constant amount of time regardless of the input data or the algorithm used.

Example:

- Arithmetic operations: addition, subtraction, multiplication and division
- Comparisons and logical operations
- Assignments, including assignments of pointers when, say, traversing a list or a tree

Counting the Frequency of Basic Operations

Definition: An elementary operation in an algorithm is called a *basic operation* if it is of highest frequency to within a constant factor among all other elementary operations.

- When analyzing searching and sorting algorithms, we may choose the element comparison operation if it is an elementary operation.
- In matrix multiplication algorithms, we select the operation of scalar multiplication.
- In traversing a linked list, we may select the "operation" of setting or updating a pointer.
- In graph traversals, we may choose the "action" of visiting a node, and count the number of nodes visited.

Input Size and Problem Instance

Suppose that the following integer

$$2^{1024} - 1$$

is a legitimate input of an algorithm. What is the *size* of the input?

Input Size and Problem Instance

Algorithm 4: Summation1

Input: A positive integer n and an array $A[1, \cdots, n]$ with $A[j] = j$ for $1 \le j \le n$.
Output: $\sum_{j=1}^{n} A[j]$.

$sum \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do**
  $sum \leftarrow sum + A[j]$;
return sum;

The input size is n. The time complexity is $O(n)$. It is linear time.

Input Size and Problem Instance

Algorithm 5: Summation2

Input: A positive integer n.
Output: $\sum_{j=1}^{n} j$.

$sum \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do**
  $sum \leftarrow sum + j$;
return sum;

The input size is $k = \lfloor \log n \rfloor + 1$. The time complexity is $O(2^k)$. It is exponential time.

Commonly Used Measures

- In sorting and searching problems, we use the number of entries in the array or list as the input size.
- In graph algorithms, the input size usually refers to the number of vertices or edges in the graph, or both.
- In computational geometry, the size of input is usually expressed in terms of the number of points, vertices, edges, line segments, polygons, etc.
- In matrix operations, the input size is commonly taken to be the dimensions of the input matrices.
- In number theory algorithms and cryptography, the number of bits in the input is usually chosen to denote its length. The number of words used to represent a single number may also be chosen as well, as each word consists of a fixed number of bits.

Best, Worst, Average Case Analysis

In **best case analysis**, we calculate lower bound on running time of an algorithm. Such case causes minimum number of operations to be executed.

In **worst case analysis**, we calculate upper bound on running time of an algorithm. Such case causes maximum number of operations to be executed.

In **average case analysis**, we take all possible inputs and calculate the expected computing time for all of the inputs.

Note: By default, usually we provide *worst case* running time for an algorithm without specification.

Amortized Analysis

In **amortized analysis**, we average out the time taken by the operation throughout the execution of the algorithm, and refer to this average as the *amortized running time* of that operation.

Amortized analysis guarantees the average cost of the operation, and thus the algorithm, *in the worst case*.

This is to be contrasted with the average time analysis in which the average is taken over all instances of the same size. Moreover, unlike the average case analysis, no assumptions about the probability distribution of the input are needed.

Linear Search

Linear search scan an array sequentially from the very beginning to check whether the key exists, as shown in Alg. 6.

Algorithm 6: LinearSearch($A[\cdot]$, x)

Input : An array $A[1, \cdots, n]$ of n elements, an integer key x
Output: First index of key x in A, -1 if not found

index $\leftarrow -1$;
for $i \leftarrow 1$ ***to*** n **do**
  if $A[i] = x$ **then**
    index $\leftarrow i$;
    break;
return *index;*

Algorithm Analysis for LinearSearch

Best Case: $\Omega(1)$.

- Appears when the key exists in the first slot of the array.
- Example: $A = [1, 2, 7, 3, 6, 0, 9]$, $x = 1$.

Worst Case: $O(n)$.

- Appears when the key does not exist in the array (or as the last item).
- Example: $A = [3, 1, 0, 5, 4, 7, 2]$, $x = 6$.

Space Complexity: $O(1)$.

Algorithm Analysis for LinearSearch

Average Case: $O(n)$.

We consider the cases that x is found (otherwise all cases that x is not found should have n comparisons).

Assume the probability that x appears at $A[i]$ is equal for all i (Note that $i = n$ means $x = A[n]$ or x is not found).

The expected number of comparisons should be:

$$
\begin{aligned}
& E[\text{total comparison}] \\
= & \sum_{i=1}^{n} Pr(x \text{ appears at } A[i]) \cdot (\text{no. of comparisons in this case}) \\
= & \sum_{i=1}^{n} \frac{i}{n} = \frac{n+1}{2}
\end{aligned}
$$

Binary Search (In Sorted Array)

Algorithm 7: BinarySearch($A[\cdot]$, x)

Input : A sorted array $A[1 \ldots n]$ of n elements, an integer key x
Output: First index of key x in A, -1 if not found

$low \leftarrow 1$; $high \leftarrow n$; $index \leftarrow -1$;
while $low \leq high$ **do**
    $mid \leftarrow low + ((high - low)/2)$;
    if $A[mid] > x$ **then**
        $high \leftarrow mid - 1$;
    else if $A[mid] < x$ **then**
        $low \leftarrow mid + 1$;
    else
        $index \leftarrow mid$; **Break**;
return $index$;

Algorithm Analysis for BinarySearch

Best Case: $\Omega(1)$.

- Appears when the key exists in the middle slot of the array.
- Example: $A = [1, 2, 3, 6, 7]$, $x = 3$.

Worst Case: $O(\log n)$.

- Appears when the key does not exist in the array (or as the last or first item).
- $A = [0, 1, 3, 4, 5, 7, 9]$, $x = 2$.

Space Complexity: $O(1)$.

Algorithm Analysis for BinarySearch

Average Case: $O(\log n)$.
To simplify the calculation, let $n = 2^k - 1$ so that $k = \log(n+1)$.

$$
\begin{aligned}
& E[\text{comparison}] \\
= & \sum_{i=1}^{n} Pr(x \text{ appears at } A[i]) \cdot (\text{no. of comparisons in this case}) \\
= & \frac{1}{n}\sum_{i=1}^{k} (\text{no. of iterations in case } i) \cdot (\text{no. of nodes in case } i) \\
= & \frac{1}{n}\sum_{i=1}^{k} i \times 2^{i-1}
\end{aligned}
$$

Algorithm Analysis for BinarySearch

Arithmetico-Geometric Progression (A.G.P.):

$$
\begin{cases} c_n = (a_1 + (n-1)\cdot d)\cdot q^{n-1}, \\ S_n = (A \cdot n + B)\cdot q^n - B, \qquad A = \frac{d}{q-1}, B = \frac{a_1 - d - A}{q-1} \end{cases}
$$

$$E[\text{comparison}] = \frac{1}{n}\sum_{i=1}^{k} i \times 2^{i-1} = \frac{1}{n}(k \cdot 2^k - 2^k + 1) = \frac{n+1}{n}\log(n+1) - 1.$$

Average Case: $O(\log n)$.

Example: Take an array of 15 elements, the average cost is:

$$E = (4 \times 8 + 3 \times 4 + 2 \times 2 + 1 \times 1)/15 = 3.26 \text{ (or } \log n).$$

Selection Sort

Every iteration, select the ith smallest number and locates it at ith slot.

Algorithm 8: SelectionSort($A[\cdot]$)

Input : An array $A[1, \cdots, n]$ of n elements.
Output: $A[1, \cdots, n]$ in nondecreasing order.

for $i \leftarrow 1$ ***to*** $n-1$ **do**
  for $j \leftarrow i+1$ ***to*** n **do**
    if $A[i] > A[j]$ **then**
      swap $A[i]$ and $A[j]$;
return $A[1, \cdots, n]$;

Algorithm Analysis for SelectionSort

Best Case, **Average Case**, **Worst Case**: $\Theta(n^2)$.

Whatever the input array is, Selection Sort will always go through the whole array.

$$\text{total comparisons} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Example: $A = [5, 8, 5, 2, 9]$, when we go through the whole array, we will interchange the position of the first 5 and the 2, then interchange the position of the second 5 and the 8, ..., until $A = [2, 5, 5, 8, 9]$.

Space Complexity: $O(1)$.

Bubble Sort

BubbleSort repeatedly swaps the adjacent elements if they are in wrong order.

Algorithm 9: BubbleSort($A[\cdot]$)

Input: An array $A[1 \ldots n]$ of n elements.
Output: $A[1 \ldots n]$ in nondecreasing order.

$i \leftarrow 1$;
while $i \leq n-1$ **do**
  for $j \leftarrow n$ ***downto*** $i+1$ **do**
    if $A[j] < A[j-1]$ **then**
      interchange $A[j]$ and $A[j-1]$;
  $i \leftarrow i+1$;
return $A[1, \cdots, n]$;

Algorithm Analysis for BubbleSort

Best Case, **Average Case**, **Worst Case**: $\Theta(n^2)$.

The bubble sort always goes through the whole array. Notice that even the original array is already sorted, the bubble sort will also go through the whole process. Thus,

$$\text{total comparisons} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Space Complexity: $O(1)$.

Insertion Sort

Each time takes first element in the unsorted part and inserts it to the right place of the sorted one.

Algorithm 10: InsertionSort

Input: An array $A[1, \cdots, n]$ of n elements.
Output: $A[1, \cdots, n]$ sorted in nondecreasing order.

```
for i ← 2 to n do
    x ← A[i];
    j ← i − 1;
    while j > 0 and A[j] > x do
        A[j + 1] ← A[j];
        j ← j − 1;
    A[j + 1] ← x;
return A[1, ··· , n];
```


Analysis of InsertionSort

Best Case: $\Omega(n)$.
The best case happens when the array is already sorted. Then for each element in the array, it enters the loop and exits at once. The total amount of comparison will be $n - 1$.

Worst Case: $O(n^2)$.
The worst case happens when the array is reverse ordered. Then for each element in the array, it will always be moved to the top of the array. Thus the total amount of comparison will be

$$\text{total comparison} = \sum_{i=2}^{n}(i-1) = \frac{n(n-1)}{2}$$

Space Complexity: $O(1)$.

Average Case Analysis

Take Algorithm InsertionSort for instance. Two assumptions:

- $A[1, \cdots, n]$ contains the numbers 1 through n.
- All $n!$ permutations are equally likely.

Suppose $A[i]$ should be inserted at position j $(1 \leq j \leq i)$.

- When $j = 1$, we need $i - 1$ comparisons to insert $A[i]$.
- Otherwise, we need $i - j + 1$ comparisons. (Note when $j = 2$, we still need $i - 1$ comparisons to determine its proper position.)

Since any integer in $[1, \cdots, i]$ is equally likely to be taken by j, i.e.,

$$P(j=1) = P(j=2) = \cdots = P(j=i) = \frac{1}{i},$$

Average Case Analysis

The expectation number of comparisons for inserting element $A[i]$ in its proper position, is

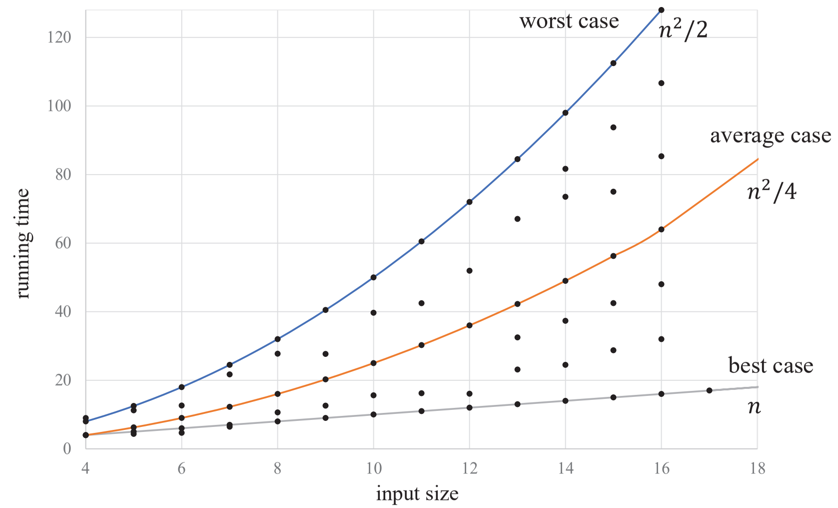
$$\frac{i-1}{i} + \sum_{j=2}^{i}\frac{i-j+1}{i} = \frac{i-1}{i} + \sum_{j=1}^{i-1}\frac{j}{i} = \frac{i}{2} - \frac{1}{i} + \frac{1}{2}$$

The *average* number of comparisons performed by Algorithm InsertionSort is

$$\sum_{i=2}^{n}\left(\frac{i}{2} - \frac{1}{i} + \frac{1}{2}\right) = \frac{n^2}{4} + \frac{3n}{4} - \sum_{i=1}^{n}\frac{1}{i}$$

Thus, the **average case** complexity is $O(n^2)$.

Performance of InsertionSort



Merging Two Sorted Lists

Algorithm 11: Merge($A[\cdot], p, q, r$)

Input: $A[1, \cdots, m]$, p, q and r with $1 \leq p \leq q < r \leq m$.
Output: $A[p, \cdots, r]$ (merging $A[p, \cdots, q]$, $A[q+1, \cdots, r]$).

```
s ← p; t ← q + 1; k ← p;
while s ≤ q and t ≤ r do
    if A[s] ≤ A[t] then
        B[k] ← A[s]; s ← s + 1; (B[p, ··· , r] is an auxiliary array)
    else B[k] ← A[t]; t ← t + 1;
    k ← k + 1;
if s = q + 1 then
    B[k, ··· , r] ← A[t, ··· , r];
else B[k, ··· , r] ← A[s, ··· , q];
return A[p, ··· , r] ← B[p, ··· , r];
```


Analysis of Merge

Suppose $A[p, \cdots, q]$ has m elements and $A[q+1, \cdots, r]$ has n elements. The number of comparisons done by Algorithm Merge is

- at least $\min\{m, n\}$;

  E.g. | 2 | 3 | 6 | and | 7 | 11 | 13 | 45 | 57 |

- at most $m + n - 1$.

  E.g. | 2 | 3 | 66 | and | 7 | 11 | 13 | 45 | 57 |

If the two array sizes are $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, the number of comparisons is between $\lfloor n/2 \rfloor$ and $n - 1$.

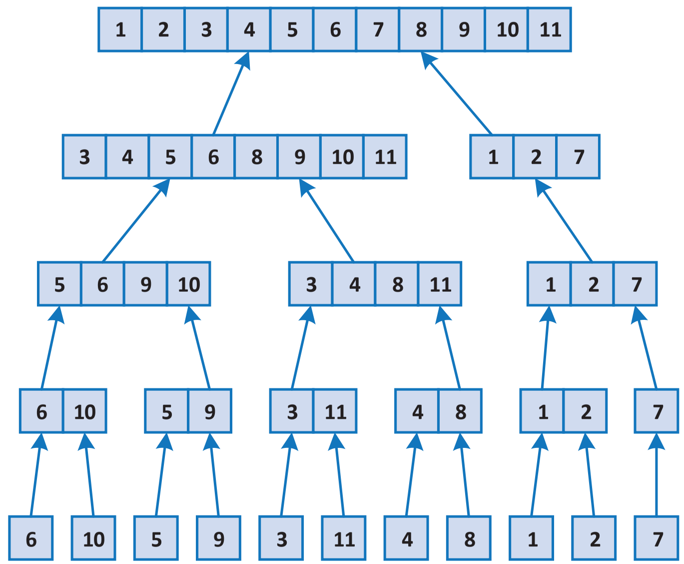
Bottom-Up MergeSort Algorithm

Algorithm 12: MergeSort($A[\cdot]$)

Input: An array $A[1, \cdots, n]$ of n elements.
Output: $A[1, \cdots, n]$ sorted in nondecreasing order.

```
t ← 1;
while t < n do
    s ← t; t ← 2s; i ← 0;
    while i + t ≤ n do
        Merge(A, i + 1, i + s, i + t);
        i ← i + t;
    if i + s < n then
        Merge(A, i + 1, i + s, n);
return A[1, ··· , n];
```

An Example



Algorithm Analysis for MergeSort

Suppose that n is a power of 2, say $n = 2^k$. The outer **while** loop is executed $k = \log n$ times. In the j-th iteration, there are $2^{k-j} = n/2^j$ pairs of arrays of size 2^{j-1}.

The number of comparisons needed in the merge of two sorted arrays in the j-th iteration is at least 2^{j-1} and at most $2^j - 1$.

Thus, the number of comparisons in MergeSort is at least

$$\sum_{j=1}^{k} \left(\frac{n}{2^j}\right) 2^{j-1} = \sum_{j=1}^{k} \frac{n}{2} = \frac{n \log n}{2}$$

The number of comparisons in MergeSort is at most

$$\sum_{j=1}^{k} \left(\frac{n}{2^j}\right)(2^j - 1) = \sum_{j=1}^{k} \left(n - \frac{n}{2^j}\right) = n \log n - n + 1$$

Best Case, **Worst Case**, **Average Case**: $\Theta(n \log n)$.

MergeSort: A Recursive Manner

Algorithm 13: MergeSort($A[\cdot]$)

Input: $A[1, \cdots, n]$ of n, first index *left*, last index *right*.
Output: $A[1, \cdots, n]$ in nondecreasing order.

```
if left ≥ right then
    return;
mid ← (left + right)/2;
MergeSort(A[left, ··· , mid]);
MergeSort(A[mid + 1, ··· , right]);
Merge(A, left, mid, right);
```

As a typical Divide-and-Conquer method, we can implement Master's Theorem to compute its complexity.

QuickSort

Randomly choose a *pivot* and partition the array by smaller and larger halves, locating the correct position of *pivot*.

Algorithm 14: QuickSort($A[\cdot]$)

Input: An array $A[1, \cdots, n]$
Output: $A[1, \cdots, n]$ sorted nonincreasingly

```
i ← 1; pivot ← A[n];
for j ← 1 to n − 1 do      (Partition A as smaller and larger parts)
    if A[j] < pivot then
        swap A[i] and A[j];
        i ← i + 1;
swap A[i] and A[n];
if i > 1 then QuickSort(A[1, ··· , i − 1]);
if i < n then QuickSort(A[i + 1, ··· , n]);
```

Algorithm Analysis for QuickSort

Best Case: $\Omega(n \log n)$.

Appears when every time the pivot separates the array into two equally-sized subarrays. Similar as MergeSort, QuickSort will separate the array approximately $\log n$ times.

$$T(n) = \sum_{j=1}^{\log n} \frac{n}{2^j} \times 2^j = n \log n.$$

Worst Case: $O(n^2)$.

Happens when every time the pivot always separates the array into 1 and $n - 1$ sized subarrays. In this situation, the divide-and-conquer concepts fails to perform well. Hence, generally the time complexity will go through something like the double loops.

Comparison

Algorithm	Best Case	Average Case	Worst Case	Space
Linear Search	$\Omega(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$\Omega(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Insertion Sort	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(n)$
Quick Sort	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

- Many of those sorting and searching algorithms can be optimized by different implementation manners.
- The complexity of MergeSort and QuickSort will be further discussed with *Divide-and-Conquer* and *Randomized Algorithm*.