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.

Outline

Outline

Time Complexity

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.

Computational Complexity evolved from 1960's, flourished in 1970's and 1980's.

- Time is the most precious resource.
- Important to human.

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

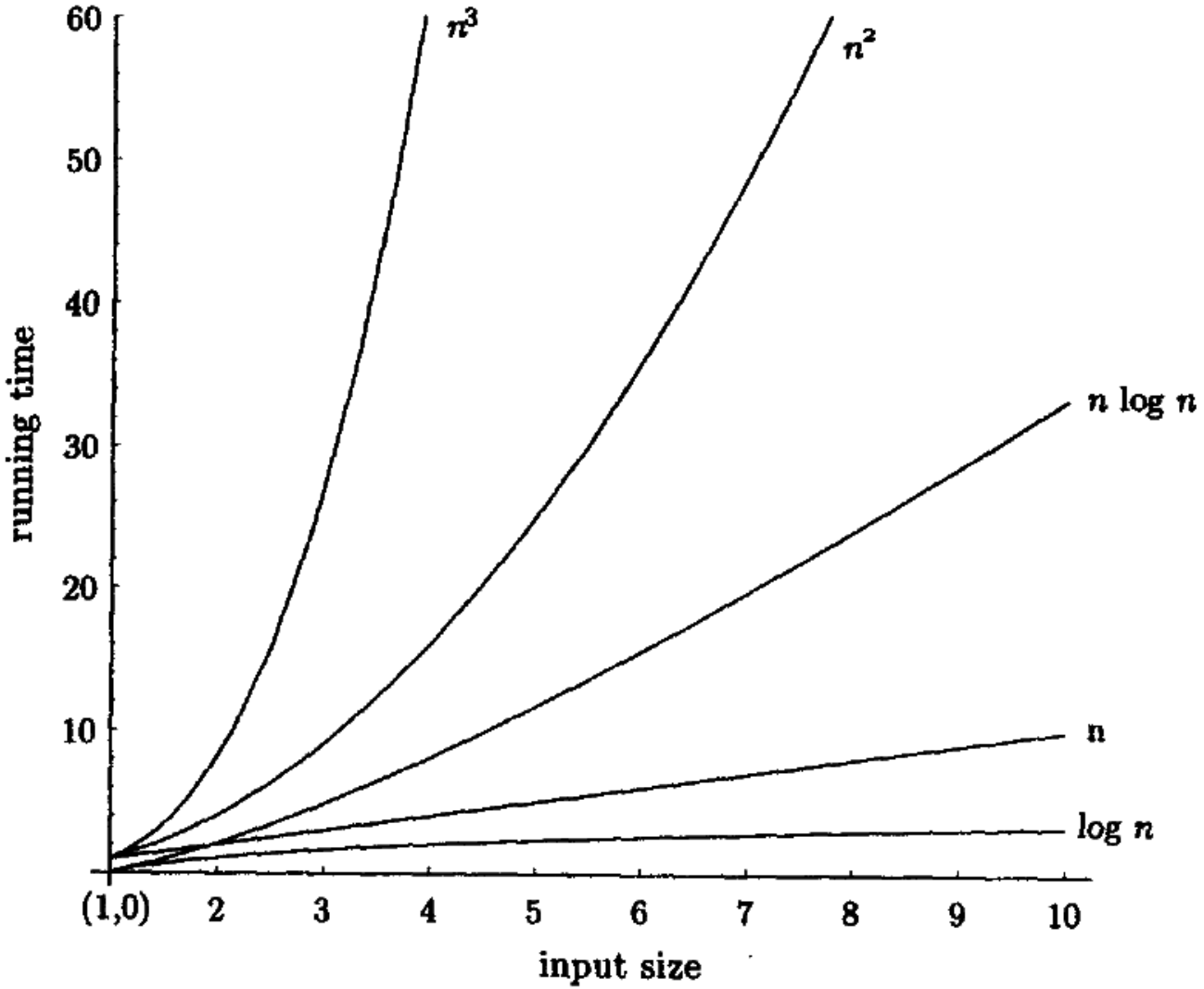
We are mostly concerned with the behavior of the algorithm under investigation on large input instances.

So 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

Growth of Typical Functions



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

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.

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

大O符号

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 $f(n) = \Theta(g(n))$.

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 $\mathcal{R}$.

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 $f(n) = o(g(n))$.

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 $f(n) = o(g(n))$.

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

Outline

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*.

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.

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 $S(n) = O(T(n))$.

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.

Outline

How to estimate time complexity? Counting the Iterations

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 ← 0;
2 while n ≥ 1 do
3     for j ← 1 to n do
4         count ← count + 1;
5     n ← n/2;
6 return count;
```

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 ← 0;
2 while n ≥ 1 do
3     for j ← 1 to n do
4         count ← count + 1;
5     n ← 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 ← 0;
2 for i ← 1 to n do
3 |   m ← ⌊n/i⌋;
4 |   for j ← 1 to m do
5 |   |   count ← count + 1;
6 return count;
```

Counting the Iterations

Algorithm 2: Count2

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

```
1 count ← 0;
2 for i ← 1 to n do
3 |   m ← ⌊n/i⌋;
4 |   for j ← 1 to m do
5 |   |   count ← 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

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

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 ← 0;
2 for i ← 1 to n do
3 |   j ← 2;
4 |   while j ≤ n do
5 |   |   j ← j^2;
6 |   |   count ← 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 $n(\log\log n + 1) = \Theta(n \log\log n)$.

Counting the Iterations

Algorithm 4: PSUM

Input: $n = k^2$, k is a positive integer.

Output: $\sum_{i=1}^{j} i$ for each perfect square j between 1 and n.

```
1 k ← √n;
2 for j ← 1 to k do
3 |   sum[j] ← 0;
4 |   for i ← 1 to j^2 do
5 |   |   sum[j] ← sum[j] + i;
6 return sum[1 · · · k];
```

Counting the Iterations

Assume that $\sqrt{n}$ can be computed in $O(1)$ time.

The outer and inner **for** loop are executed $k = \sqrt{n}$ and j^2 times respectively.

Thus, the number of iterations for inner **for** loop is

$$\sum_{j=1}^{k}\sum_{i=1}^{j^2} 1 = \sum_{j=1}^{k} j^2 = \frac{k(k+1)(2k+1)}{6} = \Theta(k^3) = \Theta(n^{1.5}).$$

The total output is $\Theta(n^{1.5})$.

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.

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.

Outline

Insertion Sort

Algorithm 5: InsertionSort

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

```
1 for i ← 2 to n do
2     x ← A[i];
3     j ← i − 1;
4     while j > 0 and A[j] > x do
5         A[j + 1] ← A[j];
6         j ← j − 1;
7     A[j + 1] ← x;
8 return A[1..n];
```

Analysis of InsertionSort

The number of comparisons carried out by Algorithm InsertionSort is at least

$$n-1$$

and at most

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

最坏每个都要移动

Average Case Analysis

Take Algorithm InsertionSort for instance. Two assumptions:

- $A[1..n]$ 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 $[1,\ i]$ 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

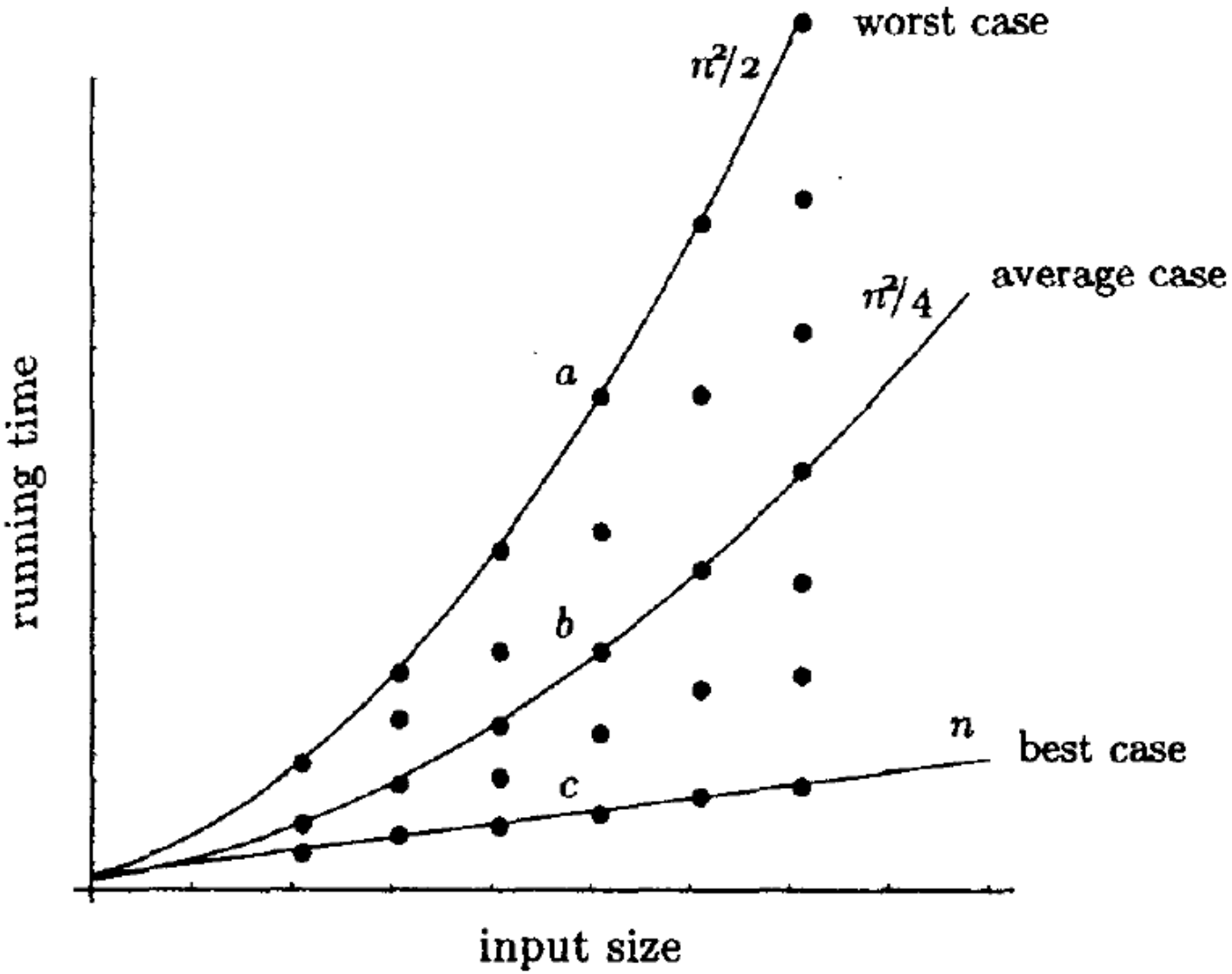
The expectation number of comparisons for inserting element $A[i]$ in its proper position, is

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

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

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

Performance of INSERTIONSORT



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.

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 6: FIRST

Input: A positive integer n and an array $A[1..n]$ with $A[j] = j$ for $1 \leq j \leq n$.

Output: $\sum_{j=1}^{n} A[j]$.

```
1 sum ← 0;
2 for j ← 1 to n do
3     sum ← sum + A[j];
4 return sum;
```

Input Size and Problem Instance

Algorithm 6: FIRST

Input: A positive integer n and an array $A[1..n]$ with $A[j] = j$ for $1 \leq j \leq n$.

Output: $\sum_{j=1}^{n} A[j]$.

```
1 sum ← 0;
2 for j ← 1 to n do
3     sum ← sum + A[j];
4 return sum;
```

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

Input Size and Problem Instance

Algorithm 7: SECOND

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

```
1 sum ← 0;
2 for j ← 1 to n do
3     sum ← sum + j;
4 return sum;
```

Input Size and Problem Instance

Algorithm 7: SECOND

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

```
1 sum ← 0;
2 for j ← 1 to n do
3     sum ← sum + j;
4 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.