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.

Theorem proving is in general not algorithmic.
Theorem verification is often algorithmic.

Quotation from Donald E. Knuth

“Computer Science is the study of algorithms.”

——Donald E. Knuth

What is Computer Science?

Computer Science is the study of problem solving using computing machines. The computing machines must be physically feasible.

Donald E. Knuth
(1938 – )
Stanford University
Remark on Algorithm

The word ‘algorithm’ is derived from the name of Muḥammad ibn Mūsā al-Khwārizmī (780?-850?), a Muslim mathematician whose works introduced Arabic numerals and algebraic concepts to Western mathematics. The word ‘algebra’ stems from the title of his book *Kitab al jahr wa'l-muqābala*.

(American Heritage Dictionary)

Algorithm vs. Program

A program is an implementation of an algorithm, or algorithms. A program does not necessarily terminate.

What is Computer Science?

I. **Theory of Computation**
   is to understand the notion of computation in a formal framework.
   - Some well known models are: the general recursive function model of Gödel and Church, Church’s λ-calculus, Post system model, Turing machine model, RAM, etc.

II. **Computability Theory**
    studies what problems can be solved by computers.

III. **Computational Complexity**
     studies how much resource is necessary in order to solve a problem.

IV. **Theory of Algorithm**
    studies how problems can be solved.

Linear Search, First Example of an Algorithm

The problem to start with: Search and Ordering.

**Algorithm 1.1 LinearSearch**

**Input:** An array $A[1..n]$ of $n$ elements and an element $x$.

**Output:** $j$ if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

1. $j \leftarrow 1$
2. while $j < n$ and $x \neq A[j]$
3. \hspace{1em} $j \leftarrow j + 1$
4. end while
5. if $x = A[j]$ then return $j$ else return $0$
### Algorithm 1.2 BinarySearch

**Input:** An array $A[1..n]$ of $n$ elements sorted in nondecreasing order and an element $x$.

**Output:** $j$ if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

1. $low \leftarrow 1$; $high \leftarrow n$; $j \leftarrow 0$
2. while $low \leq high$ and $j = 0$
3. $mid \leftarrow \lfloor (low + high)/2 \rfloor$
4. if $x = A[mid]$ then $j \leftarrow mid$ break
5. else if $x < A[mid]$ then $high \leftarrow mid - 1$
6. else $low \leftarrow mid + 1$
7. end while
8. return $j$

### Analysis of BinarySearch

The complexity of the algorithm is the number of comparison.

The number of comparison is maximum if $x \geq A[n]$.

The number of comparisons is the same as the number of iterations.

In the second iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2 \rfloor$.

In the $j$-th iteration, the number of elements in $A[mid + 1..n]$ is exactly $\lfloor n/2^{j-1} \rfloor$.

The maximum number of iteration is the $j$ such that $\lfloor n/2^{j-1} \rfloor = 1$, which is equivalent to $j - 1 \leq \log n < j$.

Hence $j = \lfloor \log n \rfloor + 1$.

### Algorithm 1.3 Merge

**Input:** An array $A[1..m]$ of elements and three indices $p$, $q$ and $r$. with $1 \leq p \leq q < r \leq m$, such that both the subarray $A[p..q]$ and $A[q+1..r]$ are sorted individually in nondecreasing order.

**Output:** $A[p..r]$ contains the result of merging the two subarrays $A[p..q]$ and $A[q+1..r]$.

**Comment:** $B[p..r]$ is an auxiliary array.
### Merging Two Sorted Lists

1. $s \leftarrow p$; $t \leftarrow q + 1$; $k \leftarrow p$
2. while $s \leq q$ and $t \leq r$
3. if $A[s] \leq A[t]$ then
4. $B[k] \leftarrow A[s]$
5. $s \leftarrow s + 1$
6. else
7. $B[k] \leftarrow A[t]$
8. $t \leftarrow t + 1$
9. endif
10. $k \leftarrow k + 1$
11. endwhile
12. if $s = q + 1$ then
13. $B[k..r] \leftarrow A[t..r]$
14. else $B[k..r] \leftarrow A[s..q]$
15. endif

### Analysis of Merge

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

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

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

- at most $m + n − 1$.

  E.g. $2\ 3\ 66\ 7\ 11\ 13\ 45\ 57$

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 $n − 1$.

### Selection Sort

**Algorithm 1.4 SelectionSort**

**Input:** An array $A[1..n]$ of $n$ elements.

**Output:** $A[1..n]$ sorted in nondecreasing order.

1. for $i \leftarrow 1$ to $n − 1$
2. $k \leftarrow i$
3. for $j \leftarrow i + 1$ to $n$
4. if $A[j] < A[k]$ then $k \leftarrow j$
5. end for
6. if $k \neq i$ then interchange $A[i]$ and $A[k]$
7. end for

### Analysis of SelectionSort

The number of comparisons carried out by Algorithm SelectionSort is precisely

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

**Algorithm 1.5** InsertionSort

**Input:** An array $A[1..n]$ of $n$ elements.

**Output:** $A[1..n]$ sorted in nondecreasing order.

1. for $i ← 2$ to $n$
2.    $x ← A[i]$
3.    $j ← i − 1$
4.    while $j > 0$ and $A[j] > x$
6.        $j ← j − 1$
7.    endwhile
8.    $A[j + 1] ← x$
9. endfor

Analysis of InsertionSort

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

$$n - 1$$

and at most

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

Bottom-Up Merge Sort

**Algorithm 1.6** BottomUpSort

**Input:** An array $A[1..n]$ of $n$ elements.

**Output:** $A[1..n]$ sorted in nondecreasing order.

1. $t ← 1$
2. while $t < n$
3.    $s ← t; t ← 2s; i ← 0$
4.    while $i + t \leq n$
5.        $\text{Merge}(A, i + 1, i + s, i + t)$
6.        $i ← i + t$
7.    endwhile
8. if $i + s < n$ then $\text{Merge}(A, i + 1, i + s, n)$
9. end while

An Example
Analysis of BottomUpSort

Suppose that $n$ is a power of 2, say $n = 2^k$.

- The outer while loop is executed $k = \log n$ times.
- Step 8 is never invoked.
- In the $j$-th iteration of the outer while loop, 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 $2^j - 1$.
- The number of comparisons in BottomUpSort 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 BottomUpSort is at most
  \[
  \sum_{j=1}^{k} \left( \frac{n}{2^j} \right) (2^j - 1) = \sum_{j=1}^{k} \left( n - \frac{n}{2^j} \right) = n \log n - n + 1
  \]
Basic Concepts in Algorithmic Analysis
Search and Ordering
Computational Complexity
Complexity Analysis

Growth of Typical Functions

![Graph showing the growth of typical functions](image)

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.

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 $\Omega$-Notation

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

**Definition (\(\Omega\)-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 $\Theta$-Notation

The $\Theta$-notation provides an exact picture of the growth rate of the running time of an algorithm.

**Definition (\(\Theta\)-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 \(g(n) = \Theta(f(n))\).

**Example**

\[ f(n) = 10n^2 + 20n. \]

- Since \(\forall n \geq 1, f(n) \leq 30n^2, f(n) = O(n^2); \]
- Since \(\forall n \geq 1, f(n) \geq n^2, f(n) = \Omega(n^2); \]
- Since \(\forall n \geq 1, n^2 \leq f(n) \leq 30n^2, f(n) = \Theta(n^2); \)

**Examples**

\[ a_kn^k + a_{k-1}n^{k-1} + \cdots + a_1n + a_0 = O(n^k). \]

\[ \log n^2 = O(n). \]

\[ \log n^k = \Omega(\log n). \]

\[ n! = O((n+1)!). \]
Examples

Consider the series \( \sum_{j=1}^{n} \log j \). Clearly,

\[
\sum_{j=1}^{n} \log j \leq \sum_{j=1}^{n} \log n = n \log n.
\]

Thus \( \sum_{j=1}^{n} \log j = O(n \log n) \)

On the other hand,

\[
\sum_{j=1}^{n} \log j \geq \sum_{j=1}^{\lfloor n/2 \rfloor} \log \left( \frac{n}{2} \right) = \lfloor n/2 \rfloor \log \left( \frac{n}{2} \right) = \lfloor n/2 \rfloor \log n - \lfloor n/2 \rfloor
\]

That is

\[
\sum_{j=1}^{n} \log j = \Omega(n \log n)
\]

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 \( \omega \)-Notation

Definition (\( \omega \)-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 \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \) 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 \( \prec \) \( \log \log n \prec \log n \prec \sqrt{n} \prec n \frac{1}{2} \prec n \prec n \log n \prec n^2 \prec 2^n \prec n! \prec 2^{n^2} \)

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.
Space Complexity

It is clear that the work space of an algorithm cannot 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 \( \Pi \) must be \( \Omega(f(n)) \), then we call any algorithm to solve problem \( \Pi \) in time \( O(f(n)) \) an *optimal algorithm* for problem \( \Pi \).

Summary

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEARSEARCH</td>
<td>( O(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>BINARYSEARCH</td>
<td>( O(\log n) ), ( \Omega(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>MERGE</td>
<td>( O(n) ), ( \Omega(n^1) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>SELECTIONSORT</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>INSERTIONSORT</td>
<td>( O(n^2) ), ( \Omega(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>BOTTOMUPSORT</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n) )</td>
</tr>
</tbody>
</table>

HOW do we estimate time complexity?
### Counting the Iterations

#### Algorithm 1.7 Count1

**Input:** \( n = 2^k \), for some positive integer \( k \).

**Output:** \( \text{count} = \text{number of times Step 4 is executed} \).

1. \( \text{count} \leftarrow 0; \)
2. while \( n \geq 1 \)
3. for \( j \leftarrow 1 \) to \( n \)
4. \( \text{count} \leftarrow \text{count} + 1 \)
5. end for
6. \( n \leftarrow n / 2 \)
7. end while
8. return \( \text{count} \)

While is executed \( k + 1 \) times; for is executed \( n, n/2, \ldots, 1 \) times

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

#### Algorithm 1.8 Count2

**Input:** A positive integer \( n \).

**Output:** \( \text{count} = \text{number of times Step 5 is executed} \).

1. \( \text{count} \leftarrow 0; \)
2. for \( i \leftarrow 1 \) to \( n \)
3. \( m \leftarrow \lfloor n/i \rfloor \)
4. for \( j \leftarrow 1 \) to \( m \)
5. \( \text{count} \leftarrow \text{count} + 1 \)
6. end for
7. end for
8. return \( \text{count} \)

The inner for is executed \( n, \lfloor n/2 \rfloor, \lfloor n/3 \rfloor, \ldots, \lfloor n/n \rfloor \) times

\[
\Theta(n \log n) = \sum_{i=1}^{n} \left( \frac{n}{i} - 1 \right) \leq \sum_{i=1}^{n} \frac{n}{i} \leq \sum_{i=1}^{n} n = \Theta(n \log n)
\]

#### Algorithm 1.9 Count3

**Input:** \( n = 2^k \), \( k \) is a positive integer.

**Output:** \( \text{count} = \text{number of times Step 6 is executed} \).

1. \( \text{count} \leftarrow 0; \)
2. for \( i \leftarrow 1 \) to \( n \)
3. \( j \leftarrow 2; \)
4. while \( j \leq n \)
5. \( j \leftarrow j^2; \)
6. \( \text{count} \leftarrow \text{count} + 1 \)
7. end while
8. end for
9. return \( \text{count} \)

For each value of \( i \), the while loop will be executed when \( j = 2, 2^2, 2^4, \ldots, 2^k \).

That is, it will be executed when \( j = 2^n, 2^{2^n}, 2^{2^2}, \ldots, 2^{2^k} \).

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 \( n(\log \log n + 1) = \Theta(n \log \log n) \).
**Algorithm 1.10 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. \( k \leftarrow \sqrt{n} \);
2. for \( j \leftarrow 1 \) to \( k \)
3. \( \text{sum}[j] \leftarrow 0; \)
4. for \( i \leftarrow 1 \) to \( j^2 \)
5. \( \text{sum}[j] \leftarrow \text{sum}[j] + i; \)
6. end for
7. end for
8. return \( \text{sum}[1 \cdots k] \)

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 \( \Theta(n^{1.5}) \).

---

**Method of Choice**

- 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.
Master theorem

If

\[ T(n) = aT\left(\lceil n/b \rceil \right) + O(n^d) \]

for some constants \( a > 0, b > 1, \) and \( d \geq 0, \) then

\[ T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a.
\end{cases} \]

Analysis for Mergesort

The recurrence relation:

\[ T(n) = 2T(n/2) + O(n); \]

By Master Theorem

\[ T(n) = O(n \log n). \]

Performance of Insertionsort

Consider the following algorithm:

1. if \( n \) is odd then \( k \leftarrow \text{BinarySearch}(A, x) \)
2. else \( k \leftarrow \text{LinearSearch}(A, x) \)

In the worst case, the running time is \( \Omega(\log(n)) \) and \( O(n). \)
**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.

The number of comparisons for inserting element $A[i]$ in its proper position, say $j$, is on average the following:

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

The average number of comparisons performed by Algorithm InsertionSort is

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

**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.

**An Example**

Consider the following algorithm:

1. for $j \leftarrow 1$ to $n$
2. $x \leftarrow A[j]$
3. Append $x$ to the list
4. if $x$ is even then
5. while $pred(x)$ is odd do delete $pred(x)$
6. end if
7. end for
**Worst Case Analysis:** If no input numbers are even, or if all even numbers are at the beginning, then no elements are deleted, and hence each iteration of the for loop takes constant time. However, if the input has \( n - 1 \) odd integers followed by one even integer, then the number of deletions is \( n - 1 \), and the number of while loops is \( n - 1 \). The overall running time is \( O(n^2) \).

**Amortized Analysis:** The total number of elementary operations of insertions and deletions is between \( n \) and \( 2n - 1 \). So the time complexity is \( \Theta(n) \). It follows that the time used to delete each element is \( O(1) \) amortized time.

**Algorithm 1.9 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. \( \text{sum} \leftarrow 0 \);
2. for \( j \leftarrow 1 \) to \( n \)
3. \( \text{sum} \leftarrow \text{sum} + A[j] \)
4. end for
5. return \( \text{sum} \)

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

Suppose that the following integer

\[ 2^{1024} - 1 \]

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

**Algorithm 1.10 SECOND**
**Input:** A positive integer \( n \).
**Output:** \( \sum_{j=1}^{n} j \).

1. \( \text{sum} \leftarrow 0 \);
2. for \( j \leftarrow 1 \) to \( n \)
3. \( \text{sum} \leftarrow \text{sum} + j \)
4. end for
5. return \( \text{sum} \)

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