Summation Formulas, Sigma Notation & Algorithm Complexity

SCS1308 – Foundations of Algorithms

Contents

1 Introduction

Summations are fundamental mathematical constructs used to add sequences of numbers. In computer science and algorithm analysis, summations offer a concise way to describe how many basic operations an algorithm performs as a function of its input size. The Greek letter Σ (sigma) is used to denote these sums. Understanding the properties of *sigma notation*, common summation formulas, and how to bound or approximate sums is vital for analysing the time and space complexity of algorithms. This document presents a comprehensive treatment of these topics for programmers studying the foundations of algorithms.

2 Sigma Notation and Summation Basics

2.1 The components of a summation

Summation notation concisely represents the addition of a sequence of terms. A general summation has the form

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + a_{m+2} + \cdots + a_{n-1} + a_n.$$

Here i is the *index of summation*, m is the lower bound, n is the upper bound, and a_i denotes the i-th term in the sequence. When $n < m$ the sum is defined to be zero. This compact notation is far less cumbersome than writing out long sums of terms one by one.

2.2 Properties of summations

Several algebraic properties make manipulating summations easier. Let c be a constant and a_i, b_i be sequences. Then

1. **Linearity:** A constant factor can be pulled out of a sum and sums of sequences can be split:

$$\sum_{i=m}^{n} c\, a_i = c \sum_{i=m}^{n} a_i,$$
$$\sum_{i=m}^{n} (a_i \pm b_i) = \sum_{i=m}^{n} a_i \pm \sum_{i=m}^{n} b_i.$$

   These identities show that summation is a linear operator.

2. **Index shifting:** Changing the index variable or shifting the bounds does not change the value of the sum as long as the terms are adjusted accordingly. For example, setting $j = i - 1$ gives

$$\sum_{i=1}^{n} f(i) = \sum_{j=0}^{n-1} f(j+1).$$

3. **Splitting at an intermediate index:** For any integer k with $m \le k < n$, the sum can be split as

$$\sum_{i=m}^{n} f(i) = \sum_{i=m}^{k} f(i) + \sum_{i=k+1}^{n} f(i).$$

4. **Caveat for products:** In general, the product of sums is not equal to the sum of products:
$$\sum_{i=m}^{n} a_i b_i \neq \Big(\sum_{i=m}^{n} a_i\Big)\Big(\sum_{i=m}^{n} b_i\Big),$$
and similarly $\sum_{i=m}^{n} a_i b_i \neq \sum_{i=m}^{n} a_i \sum_{i=m}^{n} b_i$.

2.3 Common summation formulas

Closed-form formulas exist for many frequently occurring sums. Knowing these formulas allows us to replace summations by polynomial or logarithmic expressions, which in turn makes it easier to analyse algorithm complexity. The following results assume the lower limit of summation is 1:

- **Constant series:**
$$\sum_{i=1}^{n} c = c\,n.$$

- **Arithmetic series:**
$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$
The arithmetic series is sometimes called the *triangular number*. Its exact formula was famously derived by Gauss.

- **Quadratic series:**
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}.$$

- **Cubic series:**
$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2.$$

- **Geometric series:** For a real number $r \neq 1$,
$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}.$$

- **Harmonic series:**
$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) + \gamma,$$
where γ is the Euler–Mascheroni constant. The harmonic series diverges, but its divergence is very slow; bounding by the natural logarithm is often useful.

These formulas follow directly from the properties of summations and can be proved by induction or other methods. For example, to evaluate $\sum_{i=1}^{n}(3-2i)^2$ one expands the square and then applies the linearity property and the formulas above.

2.4 Bounding summations

Not all sums have simple closed forms. When an exact expression is difficult to obtain, bounding the sum by integrals provides useful approximations. For a monotonically increasing function f,

$$\int_{m-1}^{n} f(x)\,dx \;\le\; \sum_{i=m}^{n} f(i) \;\le\; \int_{m}^{n+1} f(x)\,dx,$$

and for a monotonically decreasing function the inequalities reverse. This technique is invaluable when estimating the growth rate of complex series such as $\sum_{i=1}^{n} i^3$. Evaluating the integral $\int_1^{n+1} x^3\,dx$ yields $\frac{(n+1)^4-1}{4}$, from which we conclude $\sum_{i=1}^{n} i^3 \in O(n^4)$.

3 Time Complexity and Asymptotic Notation

3.1 Time complexity of algorithms

The *time complexity* of an algorithm quantifies the amount of time the algorithm takes to run as a function of the size of its input. It does not depend on machine-dependent constants but instead reflects how the running time grows asymptotically. To determine the time complexity, we count the number of *basic instructions* (elementary operations such as arithmetic, logical comparisons and memory accesses). Summations arise naturally because loops perform the same basic instruction multiple times.

3.2 Asymptotic notations

Asymptotic notations describe how a function grows as the input size tends to infinity. Three common notations are Big O, Big Ω and Big Θ. Let f and g be non-negative functions defined on the natural numbers.

Definition

[Big O notation] We say $f(n)$ is $O(g(n))$ if there exist positive constants C and n_0 such that

$$0 \le f(n) \le C\,g(n) \quad \text{for all } n \ge n_0.$$

Big O notation defines an *upper bound* on the growth of f.

Definition

[Big Ω notation] We say $f(n)$ is $\Omega(g(n))$ if there exist positive constants C and n_0 such that

$$0 \le C\,g(n) \le f(n) \quad \text{for all } n \ge n_0.$$

Big Ω notation defines a *lower bound* on the growth of f.

Definition

[Big Θ notation] We say $f(n)$ is $\Theta(g(n))$ if there exist positive constants c_1, c_2 and n_0 such that

$$0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \text{for all } n \ge n_0.$$

Big Θ provides a *tight bound*: f grows neither faster nor slower than g up to constant factors.

These notations are hierarchical: if $f(n) = \Theta(g(n))$, then $f(n)$ is also $O(g(n))$ and $\Omega(g(n))$. When analysing algorithms, Big O is often used to describe the worst-case running time, Big Ω for best-case running time, and Big Θ when the upper and lower bounds coincide.

3.3 Best, worst and average case analysis

To obtain a meaningful measure of an algorithm's performance, one must consider how it behaves on different inputs. The *worst-case* running time is the maximum number of operations over all inputs of a given size. The *best-case* running time is the minimum number of operations, and the *average-case* considers the average over all inputs (weighted by their probability). In practice, worst-case analysis is most common because it provides an upper bound and avoids making assumptions about input distributions. Best-case analysis is seldom used because guaranteeing only a lower bound tells us little about worst-case behaviour, while average-case analysis requires knowing the distribution of inputs and can be complex.

4 Using Summations in Algorithm Analysis

4.1 Guidelines for deriving time complexity

When an algorithm contains loops, the total number of basic operations is often the sum of the operations performed in each iteration. The key steps are:

1. Identify the basic operations within the loop body. These are operations that take constant time, denoted $O(1)$.
2. Determine how many times each loop executes as a function of the input size n. If the number of iterations changes during the algorithm, express it symbolically and replace the innermost loop by a summation.
3. Apply the properties and formulas of summations to simplify the expression. When the exact sum is complicated, bound it by integral approximations or identify the dominant term.
4. Finally, apply the rules of Big O notation: drop lower-order terms and constant factors.

4.2 Simple examples

Example

[Summing constant operations] Consider a loop that executes a constant number of operations once per element of an array:

$$\text{for } i = 1 \text{ to } n \text{ do constant work.}$$

Each iteration performs c basic operations. The total number of operations is

$$\sum_{i=1}^{n} c = c\,n,$$

which simplifies to $O(n)$. This reflects linear time complexity.

Example

[Arithmetic series in a loop] Suppose the inner loop runs i times within an outer loop that runs n times:

$$\text{for } i = 1 \text{ to } n \quad \text{do } \big(\text{for } j = 1 \text{ to } i \text{ do constant work}\big).$$

The total cost is the sum of the inner loop counts:

$$\sum_{i=1}^{n} \sum_{j=1}^{i} c = c \sum_{i=1}^{n} i = c\, \frac{n(n+1)}{2},$$

which grows on the order of n^2. This example shows how a triangular sum arises in nested loops. Such a pattern appears in algorithms like insertion sort.

Example

[Triple nested loops] Consider three nested loops where the limits depend on the upper loop counters:

$$\text{for } i = 1 \text{ to } n \quad \text{do } \big(\text{for } j = 1 \text{ to } i \text{ do } \big(\text{for } k = 1 \text{ to } j \text{ do constant work}\big)\big).$$

The number of operations is

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

Evaluating the outer sum yields a cubic term proportional to n^3. Hence the algorithm runs in $\Theta(n^3)$ time. These kinds of nested summations are common when analysing naive matrix multiplication or similar algorithms.

4.3 Binary search and geometric series

Not all loops iterate linearly through the input. Divide-and-conquer algorithms, such as binary search, reduce the problem size by a constant factor at each step. For example, a binary search on a sorted array of length n repeatedly halves the range. The number of iterations is the smallest integer k such that $n/2^k \leq 1$, which implies $k = \lceil \log_2 n \rceil$. Thus binary search runs in $O(\log n)$ time. More generally, repeatedly dividing by r leads to a geometric series; solving $\sum_{i=0}^{k} 1 = k + 1$ for $n = r^k$ gives $k = \log_r n$.

4.4 Insertion sort: a detailed analysis

Insertion sort repeatedly inserts the next element into a growing sorted prefix. In pseudocode (using an imperative language),

```
for i <- 1 to n - 1 do
  j <- i
  while j > 0 and A[j] < A[j - 1] do
    swap A[j] and A[j - 1]
    j <- j - 1
  end while
```

```
end for
```

In the worst case (when the input array is in reverse order), the inner `while` loop compares and swaps j down to 1, taking j iterations. The outer loop runs for $n-1$ values of i. The total number of comparisons and swaps is

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

which is proportional to n^2. Therefore insertion sort has quadratic time complexity in the worst case. In the best case (an already sorted array), the inner loop performs no swaps and only one comparison per iteration; the algorithm then runs in $O(n)$ time.

4.5 Merge sort and logarithmic summations

Divide-and-conquer algorithms lead to recurrence relations rather than simple sums, but their solutions often involve logarithmic factors. Merge sort splits an array of length n into two halves, recursively sorts each half, and then merges the sorted halves in $O(n)$ time. Its running time satisfies the recurrence

$$T(n) = 2\,T\!\left(\frac{n}{2}\right) + c\,n.$$

Unfolding the recurrence yields a sum over the levels of the recursion tree. Each level performs cn work, and there are $\log_2 n$ levels, giving

$$T(n) = cn \sum_{i=0}^{\log_2 n - 1} 1 = c\,n \log_2 n,$$

so merge sort runs in $\Theta(n \log n)$ time. This example illustrates how geometric progressions and logarithms naturally arise when the input size is repeatedly halved.

5 Space Complexity

5.1 Definition and measurement

While time complexity measures how the running time grows, *space complexity* measures the amount of memory an algorithm uses. It encompasses memory for code, constants and variables (the fixed part) and memory whose size depends on the input (the variable part). Formally, the space complexity of an algorithm is the function $S(n)$ that gives the total space required as a function of the input size n. The fixed part includes program instructions and constant variables, while the variable part accounts for dynamic data structures, recursion stacks and temporary variables.

5.2 Examples of space complexity

Example

[Constant space] An algorithm that adds two scalar variables uses one extra memory location to store the result. Its space complexity is constant: $S(n) = O(1)$.

Example

[Linear space] Storing an array of n elements requires space proportional to n. Algorithms like counting frequency of elements create a frequency array or map whose size grows with n. Their space complexity is $O(n)$.

5.3 Auxiliary space vs. total space

Space complexity is sometimes broken down into *auxiliary space*, which measures the temporary or extra space required by an algorithm (excluding the space for the input), and total space, which includes both the input and auxiliary space. For example, merge sort requires $O(n)$ auxiliary space for the temporary array used during merging.

6 Additional Techniques and Remarks

6.1 Estimating sums by dominance

In practice, exact summations are often unnecessary. When summing several terms, the term with the highest order of growth (the *dominant* term) determines the asymptotic behaviour. Consider

$$f(n) = 4n^2 + 2n + 7.$$

As n grows large, the $4n^2$ term dominates, so we write $f(n) \in O(n^2)$. The contributions of lower-order terms and constant factors become insignificant. This rule allows us to simplify complex expressions obtained from summations.

6.2 Harmonic and logarithmic sums

The harmonic series and its variants occur in algorithms that shrink the input by a constant amount. For example, some data structures (such as heaps) perform $O(\log n)$ operations per insertion or deletion. Summing $1/i$ over i from 1 to n yields $\ln(n) + \gamma$, meaning that repeated logarithmic costs accumulate to a logarithmic factor.

6.3 Bounding factorials and exponential sums

Stirling's approximation and integral bounds help estimate sums involving factorial or exponential terms. For instance, the number of operations in algorithms that generate all permutations of n elements grows as $n!$. Using Stirling's formula $n! \approx \sqrt{2\pi n}\,(n/e)^n$ gives an exponential growth rate, which is much faster than polynomial or logarithmic growth.

7 Conclusion

Summations and sigma notation are more than mathematical curiosities; they are indispensable tools for analysing algorithms. By mastering the properties of summations, learning common closed-form formulas, and understanding how to bound or approximate sums, programmers can accurately characterise the time and space requirements of their algorithms. Asymptotic notations such as Big O, Big Ω and Big Θ provide a language for comparing growth rates and reasoning about performance at scale. Through examples like nested loops, insertion sort and merge sort, we see how summations translate code structure into mathematical expressions. A strong foundation in these concepts equips programmers to design efficient algorithms and to recognise when an algorithm's resource consumption may become a bottleneck.