CSC 212: Data Structures and Abstractions

04: Introduction to Analysis of Algorithms (part 2)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2026

THINK BIG WE DO℠

Theoretical analysis

"mathematical models for analyzing time and space complexity"

Computational cost analysis

- Definition and importance
  - ✓ **computational cost**, expressed as $T(n)$, represents the resources (primarily **time**, sometimes memory) an algorithm requires to process input of a given size n
  - ✓ essential for algorithm comparison and optimization in real-world applications (without implementing/running a program)
- Mathematical framework (HW/SW independent)
  - ✓ based on counting (primitive/elementary) **operations**
    - arithmetic operations (additions, multiplications), comparisons, assignments, memory access operations, etc.
    - focuses on **asymptotic behavior**

Practice

- Count the total number of operations as a function of the input size n
  - ✓ arithmetic operations, comparisons, assignments, array indexing, memory accesses, etc.

depends on specific HW

counting all operations is tricky, repetitive, and time-consuming

```
// sum of all elements in the array
int sum(int *A, int n) {
    int sum = 0;
    for (int i = 0 ; i < n ; i++) {
        sum = sum + A[i];
    }
    return sum;
}
```

Operation	Count	Time (ps)
variable declaration	2	
assignment	2 + n	
comparison (less than)	n + 1	
addition	n	
array access	n	
increment	n	

Counting operations

- Computational cost $T(n)$
  - ✓ count **elementary operations** that are **relevant** to the problem
  - ✓ express the total number of operations as a function of input size
- Examples:
  - ✓ sum of all elements in an array of length n
    - count the total number of additions => $T(n) = n$
  - ✓ finding the maximum value in an array of length n
    - count the total number of comparisons => $T(n) = n - 1$
- Formal assumptions
  - ✓ each **elementary operation** takes one time unit
  - ✓ operations execute sequentially (ignores parallelism, pipelining, and other HW optimizations)
  - ✓ infinite memory available

Practice

- Count the elementary operations (multiplications)

```
for (int i = 0 ; i < n ; i ++) {
    sum = sum * i;
}
```

Practice

- Count the elementary operations (divisions)

```
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        sum = sum / j;
    }
}
```

Practice

- Count the elementary operations (additions)

```
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n ; j ++) {
        for (int k = 0 ; k < n ; k ++) {
            sum = sum + j;
        }
    }
}
```

Practice

- Count the elementary operations (multiplications)

```
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < n*n ; j ++) {
        sum = sum * j;
    }
}
```

Practice

- Count the elementary operations (multiplications)

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=0}^{n-1} i$$

$$\sum_{i=0}^{n-1} i = 0 + 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2}$$

```
for (int i = 0 ; i < n ; i ++) {
    for (int j = 0 ; j < i ; j ++) {
        sum = sum * j;
    }
}
```

Some rules …

- Single loops
  - ✓ typically equals the number of iterations × the number of operations at each iteration
  - ✓ **requires careful analysis** of range and step size
- Nested loops
  - ✓ count operations from the innermost loop outward, multiplying the number of iterations at each level
  - ✓ **dependent loops** often result in operation counts that are not simply the product of the loop ranges, but rather require summation formulas to determine the exact count
- Consecutive statements
  - ✓ just add the counts

Useful series

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

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

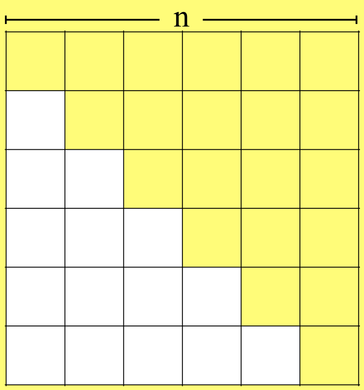
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

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

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$$

https://tug.org/texshowcase/cheat.pdf

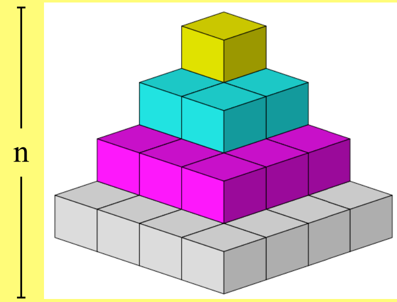
Practice



How many white squares as a function of n?

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^2(n+1)^2}{4}$$
$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$$

Practice



How many cubes as a function of n?

1 layer: 1
2 layers: 5
3 layers: 14
4 layers: 30
…
n layers: ?

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$
$$\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$
$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$
$$\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 = \frac{n^2(n+1)^2}{4}$$
$$\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}, c \neq 1$$

Image credit: Stanford's CS 106B lectures

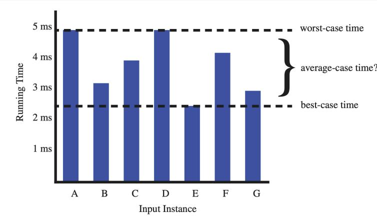
Case analysis

Practice

- Count the number of comparisons

```
// returns the index of the last occurrence
// of the minimum value in the array
int r_argmin(int *A, int n) {
    int idx = 0;
    int current = A[idx];
    for (int i = 1 ; i < n ; i ++) {
        if (A[i] <= current) {
            current = A[i];
            idx = i;
        }
    }
    return idx;
}
```

```
// returns the index of the first
// occurrence of k in the array
int l_argk(int *A, int n, int k) {
    for (int i = 0 ; i < n ; i ++) {
        if (A[i] == k) {
            return i;
        }
    }
    return -1;
}
```



An algorithm may run faster on some inputs than it does on others of the same size.

Case analysis

- **best-case analysis**:
  - algorithm behavior under optimal input conditions (easiest input)
  - useful for understanding algorithm behavior, but it provides insufficient information for real-world performance
  - e.g., searching for an element that happens to be at the first position
- **worst-case analysis**: (most commonly used)
  - algorithm behavior under the most unfavorable input conditions (hardest input)
  - critical for systems requiring performance guarantees
  - e.g., searching for an element that doesn't exist or is at the last position
- **average-case analysis**:
  - expected algorithm behavior across all possible inputs
  - requires understanding of input probability distribution, often mathematically complex

Image credit: Data Structures and Algorithms in C++, Goodrich, Tamassia, Mount

Average case analysis

- Consider the `l_argk` example
  - we assume k is in the array and is **equally likely** to be at any position (uniform distribution)
- Solution
  - each position has probability $1/n$
  - expected value E = sum of (probability $\times$ comparisons)

$$
\begin{aligned}
E\left[T(n)\right] &= \frac{1}{n}(1) + \frac{1}{n}(2) + \ldots + \frac{1}{n}(n) \\
&= \frac{1}{n}(1 + 2 + \ldots + n) \\
&= \frac{1}{n}\sum_{i=1}^{n} i \\
&= \frac{1}{n}\frac{n(n+1)}{2} \\
&= \frac{n+1}{2}
\end{aligned}
$$

Practice

- Provide T(n) for the worst-, average-, and best-case
  - find value in an unsorted sequence (return first occurrence)
  - finding the largest element in an unsorted sequence
  - finding the largest element in a sorted sequence
  - factorial of a number — iterative algorithm