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 2025

THINK BIG WE DO℠

Grow Your Technical Skills with Google

Whether you're new to computer science or an experienced coder, there's something for you here in Google's Tech Dev Guide.

We've carefully curated materials from various sources, including some made by Google, that you can use to grow your technical skills, supplement your coursework, and prepare for interviews.

Interested in pursuing a career in business? Check out Google's Business Dev Guide.

Data Structures & Algorithms

Familiarize yourself with common data structures and algorithms such as lists, trees, maps, graphs, Big-O analysis, and more!

Suggested prerequisites: Familiarity with basics programming concepts (e.g. if statements, loops, functions)

https://techdevguide.withgoogle.com/

From previous lecture

```
$ g++ -std=c++11 -O0 euler.cpp -o prog
```

```
void take_step(int n, double fn(int)) {
    auto start = std::chrono::high_resolution_clock::now();
    double e = fn(n);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << n << '\t';
    std::cout << std::fixed << std::setprecision(10);
    std::cout << e << " " << (double) elapsed.count() << '\t';
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: " << argv[0] << " <steps>\n";
        return 1;
    }

    int n = 1;
    int steps = std::stoi(argv[1]);

    for (int i = 0 ; i < steps ; i++) {
        take_step(n, &euler1);
        take_step(n, &euler2);
        std::cout << std::endl;
        n *= 2;
    }

    return 0;
}
```


Using -O0

1	2.0000000000 0.0000001340	1	2.0000000000 0.0000000680
2	2.5000000000 0.0000000840	2	2.5000000000 0.0000000500
4	2.7083333333 0.0000000890	4	2.7083333333 0.0000000610
8	2.7182787698 0.0000001640	8	2.7182787698 0.0000000900
16	2.7182818285 0.0000003460	16	2.7182818285 0.0000001450
32	2.7182818285 0.0000011100	32	2.7182818285 0.0000002580
64	2.7182818285 0.0000039970	64	2.7182818285 0.0000004830
128	2.7182818285 0.0000159510	128	2.7182818285 0.0000009450
256	2.7182818285 0.0001041730	256	2.7182818285 0.0000018210
512	2.7182818285 0.0003848000	512	2.7182818285 0.0000035790
1024	2.7182818285 0.0011147290	1024	2.7182818285 0.0000070770
2048	2.7182818285 0.0044453980	2048	2.7182818285 0.0000140490
4096	2.7182818285 0.0178193800	4096	2.7182818285 0.0000280730
8192	2.7182818285 0.0715642710	8192	2.7182818285 0.0000573730
16384	2.7182818285 0.2795817420	16384	2.7182818285 0.0001120670
32768	2.7182818285 1.0806353640	32768	2.7182818285 0.0002190680
65536	2.7182818285 4.5505467900	65536	2.7182818285 0.0004871680
131072	2.7182818285 18.0388929500	131072	2.7182818285 0.0008569540
262144	2.7182818285 73.0555476340	262144	2.7182818285 0.0017519060
524288	2.7182818285 285.4464698470	524288	2.7182818285 0.0035419900

3.2 GHz 6-Core Intel Core i7 (using a single core)

Using -O3

1	2.0000000000 0.0000001160	1	2.0000000000 0.0000000470
2	2.5000000000 0.0000000710	2	2.5000000000 0.0000000460
4	2.7083333333 0.0000000790	4	2.7083333333 0.0000000540
8	2.7182787698 0.0000001370	8	2.7182787698 0.0000000550
16	2.7182818285 0.0000002430	16	2.7182818285 0.0000000580
32	2.7182818285 0.0000005280	32	2.7182818285 0.0000000700
64	2.7182818285 0.0000015410	64	2.7182818285 0.0000001040
128	2.7182818285 0.0000057720	128	2.7182818285 0.0000001700
256	2.7182818285 0.0000252330	256	2.7182818285 0.0000002980
512	2.7182818285 0.0001099880	512	2.7182818285 0.0000005820
1024	2.7182818285 0.0004630170	1024	2.7182818285 0.0000011220
2048	2.7182818285 0.0019061790	2048	2.7182818285 0.0000020220
4096	2.7182818285 0.0077172340	4096	2.7182818285 0.0000039080
8192	2.7182818285 0.0311110110	8192	2.7182818285 0.0000078850
16384	2.7182818285 0.1249416530	16384	2.7182818285 0.0000153770
32768	2.7182818285 0.4870702990	32768	2.7182818285 0.0000290200
65536	2.7182818285 2.0076935130	65536	2.7182818285 0.0000612600
131072	2.7182818285 8.0763145900	131072	2.7182818285 0.0001146470
262144	2.7182818285 32.0460794660	262144	2.7182818285 0.0002447660
524288	2.7182818285 128.4794438710	524288	2.7182818285 0.0004891970

3.2 GHz 6-Core Intel Core i7 (using a single core)

Theoretical analysis

"mathematical models for analyzing running time"

Computational cost analysis

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

Example

- Count the total number of operations as a function of the input size n

```
// calculate the 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]; // do not worry about overflow
    }
    return sum;
}
```

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

depends on specific HW

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

Computational cost analysis

- Asymptotic behavior analysis
  - ✓ count **only elementary operations** that are **relevant** to the problem
    - approximating the total number of operations
  - ✓ ignore **constant factors** and l**ower-order terms**
  - ✓ examples:
    - calculate the sum of all elements in an array of length *n*
      - count the total number of array accesses
    - find max value in an array of length *n*
      - count the total number of comparisons
- Formal assumptions
  - ✓ each **elementary operation** requires one time unit
  - ✓ operations execute sequentially
  - ✓ 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)

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

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

Some rules …

- Single loops
  - ✓ typically the number of iterations × the number of operations at each iteration
  - ✓ loop range determination **requires careful analysis** of raage 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

Some 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

- What is T(n) for the following examples?

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


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
- **worst-case analysis**:
  - algorithm behavior under the most unfavorable input conditions (hardest input)
  - critical for systems requiring performance guarantees
  - most commonly used in practice
- **average-case analysis**:
  - expected algorithm behavior across all possible inputs
  - requires understanding of input probability distribution
  - relevant for practical applications but often mathematically complex

An algorithm may run faster on some inputs than it does on others of the same size e.g., sorting may run faster on already sorted data

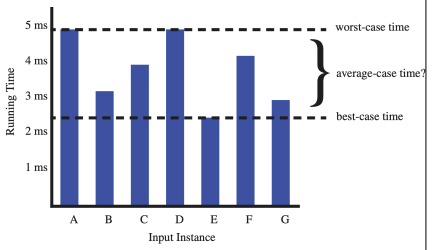

Image credit: Data Structures and Algorithms in C++, Goodrich, Tamassia, Mount

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 a sequence
    - does it matter if the sequence is sorted already?
  - factorial of a number — iterative algorithm