CSC 212: Data Structures and Abstractions

03: Introduction to Analysis of Algorithms (part 1)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

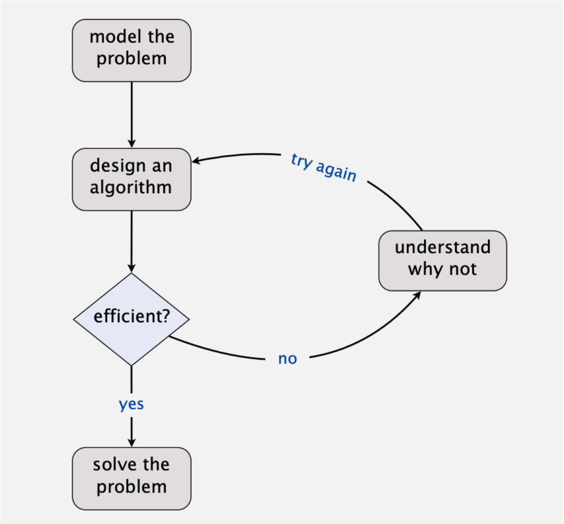
THINK BIG WE DO℠

Analysis of algorithms

Problems, algorithms, programs

- Problem
  - ✓ a computational problem represents a formalized task requiring a solution
  - ✓ well-defined input specifications, expected output requirements, and formal constraints and conditions
- Algorithm
  - ✓ precise, unambiguous sequence of computational steps
  - ✓ essential properties:
    - correctness (produces valid output for all valid inputs), finiteness (terminates after a finite number of steps), determinism (produces consistent results), feasibility (each step must be executable)
- Program implementation
  - ✓ concrete realization of an algorithm using a programming language

Developing a usable algorithm



[COS 226 lectures, Princeton University]

Analysis of algorithms

- Definition
  - branch of CS that studies the **efficiency of algorithms** in terms of time and space complexity
- Objectives
  - **performance prediction**: estimate running time growth with input size (asymptotic analysis)
  - **resource optimization**: identify bottlenecks and minimize CPU time, memory, and other resources
  - **comparative evaluation**: provide standardized metrics (e.g., Big-O) to compare algorithms solving the same problem

Importance of analysis of algorithms

- Scientific classification of algorithms
  - groups algorithms by computational characteristics; complexity classes guide design and approximation strategies
- Performance prediction and resource utilization
  - anticipates behavior before implementation to avoid costly bottlenecks
- Time and space complexity
  - critical in environments where processing power and memory are constrained
- Theoretical guarantees
  - provides confidence bounds for reliable and predictable system design

Approaches

- Empirical analysis
  - **implement** the program using a programming language
  - systematic **testing** with varied input sizes
  - statistical analysis of collected performance (**hypothesis** formation)
  - **validation** through prediction models
- Theoretical analysis
  - **mathematical modeling** of time and space complexity

Empirical analysis

Example: the e number

An irrational constant that is the base of the natural logarithm. It is approximately equal to 2.71828

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

$$e = \sum_{n=0}^{\infty}\frac{1}{n!}$$

can be expressed as an infinite sum

$$= \frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\ldots$$

Algorithm 1

Solution 1

```
double euler1(int n) {
    double e = 1.0;
    for (int i = 1; i <= n; i++) {
        double fact = 1.0;
        for (int j = 1; j <= i; j++) {
            fact *= j;
        }
        e += (1.0 / fact);
    }
    return e;
}
```

check redundant computation with n=4

Example: the e number

An irrational constant that is the base of the natural logarithm. It is approximately equal to 2.71828

$$e = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^n$$

$$e = \sum_{n=0}^{\infty}\frac{1}{n!}$$

can be expressed as an infinite sum

$$= \frac{1}{0!}+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\ldots$$

Algorithm 2

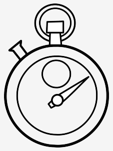
Solution 2

```
double euler2(int n) {
    double e = 1.0;
    double fact = 1.0;
    for (int i = 1; i <= n; i++) {
        fact *= i;
        e += (1.0 / fact);
    }
    return e;
}
```

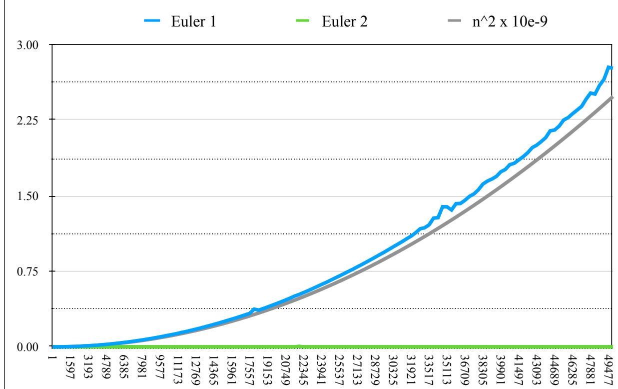
Timing

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

```cpp
#include <iostream>
#include <iomanip>

int main(int argc, char *argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " <n> <alg>\n";
        return 1;
    }

    double e;
    int n = std::stoi(argv[1]);
    int alg = std::stoi(argv[2]);

    auto start = std::chrono::high_resolution_clock::now();
    if (alg == 1) e = euler1(n);
    if (alg == 2) e = euler2(n);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << std::fixed << std::setprecision(10);
    std::cout << e << " " << (double) elapsed.count() << std::endl;

    return 0;
}
```



Hypothesis, Prediction, Validation



Hypothesis: observe running times and formulate specific hypothesis about performance characteristics, for example, we observe that Euler1's time is $T(n) = cn^2$.

Prediction: make concrete predictions that can be tested empirically, for example, predict running times for different values of n.

Validation: designing and implementing controlled experiments and comparing actual timing results against predictions.

```
$ seq 1 399 50000 | while read n; do echo -n "$n\t"; ./prog $n 1; done > e1.txt
$ seq 1 399 50000 | while read n; do echo -n "$n\t"; ./prog $n 2; done > e2.txt
```


Impact of compiler optimization

```cpp
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;
}
```

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


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)

Limitations of empirical analysis

- Implementation challenges
  - ✓ multiple algorithm implementations required, variations in code quality
- Experimentation challenges
  - ✓ designing comprehensive test cases, time-intensive execution cycles, input distribution sensitivity
- Environmental dependencies
  - ✓ compiler optimizations and flags (-O1, -O2, -O3), HW variations (AVX, branch prediction, cache sizes), OS scheduling, runtime environment fluctuations, networking
- Reproducibility
  - ✓ timing/memory measurements fluctuate, exact conditions hard to replicate, experiments may not scale or generalize