How to Measure Program Efficiency? Surveying and Timing

- **Survey**
  - Hire several test subjects to use your program
  - Survey: How fast is it? Rate from 1 to 10
  - Collect and analyze survey results
  - Obviously not very reliable
- **Write program and time it**
  - Record running times
  - Do this several experiments and get average

Timing Is Not Enough

- Need to write code first before knowing if it will work
  - Maybe waste of time
- Gives no insight why program is slow, how to fix
  - Bad way: repeatedly make some random tweaks and time the program
- Gives no insight how to design fast programs in the first place
- What we need: a *scientific* way of measuring efficiency

Why Not Just Buy a Faster Computer?

- Well… How to make faster computers in the first place?
- Just buy faster parts!
- Well… How to make faster parts in the first place?
- Just buy faster materials!
- Well… How to make faster materials in the first place?
- Just buy… a faster universe!

How the Science of Computing Works

- Efficiency is not *absolute* but *relative*
- Assume at some level, the operations have a fixed, unchangeable cost
  - **Model of computation**: what operations we assume are available, and at what fixed costs
- Focus on how to make the higher-level operations require as few of these lower-level operations as possible to minimize the cost
- Efficiency of higher-level operations depends on **algorithm**
  - How the lower-level operations are structured to achieve the goal

How to Measure Program Efficiency? Counting Operations

- Choose a set of operations we consider fixed
  - Example: primitive operations available in programming language
  - More generally, any resource relevant to problem we are working on
- Measure the average cost for each of those operations
- Count number of times we perform each type of operation
- Total cost = sum of number of times $\times$ cost per type (for all types)

How to Measure Program Efficiency? Counting Operations

Operation	**Cost (in microseconds)**
Add two integers	0.5
Multiply two integers	2.5
Change value of a variable	1.0
Print an integer	3.0

How to Measure Program Efficiency? Counting Operations

- Even better:
  - Don’t measure in seconds/bytes/etc.
    - Actual number depends on factors (hardware) which we have no control over
    - Temporarily treat real number as an irrelevant detail
  - Choose set of operations such that each has roughly the same average cost
- Total cost = total number of operations
- Later, to get the real total cost, just multiply by the real average cost
- But because at first, we only need an *estimate* we can ignore this

How to Measure Program Efficiency? Counting Operations

Operation	Cost (in number of operations)
Add two integers	1
Multiply two integers	1
Change value of a variable	1
Print an integer	1

- Average time per operation = 2 microseconds

Let's Try It

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- **Model of computation**: the following operations have cost 1
  - Declaring a variable
  - Assigning a number to variable
  - Reading the value of a variable
  - Performing one addition
  - Performing one comparison
  - Doing a condition check

How Many Times Did We Do Each Operation?

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Declaration: 2
- Assignment:
  - 2 for initialization
  - 2 per iteration $\Rightarrow 2n$
- Variable read:
  - 4 per iteration $\Rightarrow 4n$
  - 2 for last loop condition check
- Addition: 2 per iteration $\Rightarrow 2n$
- Comparison
  - 1 per iteration $\Rightarrow n$
  - 1 for last loop condition check
- Condition check: same as above
- Total: $10n + 8$

Notice

- Number of operations can grow with input size: write cost as $T(n)$
  - T for "time" (but note we are really counting number of operations)
- This still requires writing the code before knowing if it will work
- Counting all the operations is too difficult for larger programs
- Minor implementation details can change answer
  - Minor: doesn't change the "essence" of the algorithm

“Minor Implementation Details” Example

Significantly different from earlier code

```
int S = n * (n + 1) / 2;
```

Not significantly different

```
int S = 0;
for(int i = 0; i < n; i++)
    S += i + 1;
```

Big Idea in Computer Science: Analyze Large-Scale Behavior of Cost Function

- Efficiency only matters when the problem is large
- As problem size grows, lower-order terms of the cost function grow insignificant compared to the highest-order term
- Generally also true: the constant factor in front of the highest-order term is irrelevant to the “essence” of the algorithm
  - In practice, this can matter, and it matters more than lower order terms, but matters less than the “degree” of the highest-order term

Interactive Demonstration

- https://www.desmos.com/calculator/k4p7sewkq2
- When zoomed in, the four graphs all look very different
- When zoomed out, $T(n)$ looks basically the same as $f(n)$, but still quite different from $g(n)$ and $h(n)$

How to Measure Program Efficiency? Large-Scale Analysis

- Write $T(n) = \Theta(f(n))$ if, by dropping the lower-order terms and constant factors from $T(n)$, we get $f(n)$
  - There is a more formal definition, but we won't need it for now
- Let's call $\Theta(f(n))$ the **rough efficiency**
- Append what we are measuring at the end
  - Example: $\Theta(f(n))$ operations, $\Theta(f(n))$ additions, $\Theta(f(n))$ multiplications

How to Measure Program Efficiency? Large-Scale Analysis

- Measure efficiency by figuring out the rough efficiency
- This is not always easy, but easier than counting individual operations

Example

```
int S = n * (n + 1) / 2;
```

- Actual number of operations
$$T(n) = 6$$
- Rough efficiency: $\Theta(1)$

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Actual number of operations
$$T(n) = 10n + 8$$
- Rough efficiency: $\Theta(n)$

Example

```
int S = n * (n + 1) / 2;
```

- Without counting actual number of operations first: by “rough feel,” since we are doing only a fixed number of basic operations, it is $\Theta(1)$

```
int S = 0;
for(int i = 1; i <= n; i++)
    S += i;
```

- Without counting actual number of operations first: by “rough feel,” due the loop, we are doing $\Theta(n)$ operations

For Convenience, We Use the Following Shortcuts/Synonyms

- Rough efficiency of number of operations as function of input = running time
  - Since, number of operations is directly proportional to time, informally, we just use these interchangeably

For Convenience, We Use the Following Shortcuts/Synonyms

$\Theta(1)$	constant
$\Theta(\log n)$	logarithmic
$\Theta(n)$	linear
$\Theta(n \log n)$	linearithmic (less frequently used)
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
$\Theta(n^c)$ where c is some constant > 0	polynomial
$\Theta(2^n)$	exponential

Upper Bound Analysis

- Sometimes, it's hard to determine even the highest-order term $f(n)$ such that the cost is $\Theta(f(n))$
- Determine the **upper bound** $O(g(n))$ instead
  - $T(n)$ is $O(g(n))$ if $g(n)$ is $\geq$ highest order term of $T(n)$
  - There is a more formal definition, but we won't need it for now
- Why this is ok
  - If we can show that our algorithm is "good enough," then we can go ahead and implement it
  - Actual rough efficiency may be better, but all we care about is that it doesn't exceed a certain limit

Upper Bound Analysis: Example

```
for(int i = 0, j = n - 1; i < n; i++)
    for(; j >= 0 and a[i] + a[j] >= k; j--)
        if(a[i] + a[j] == k)
            cout << i << " " << j << endl;
```

- It is quite easy to see by the "shape" of this program that it is $O(n^2)$
- However, it is actually $\Theta(n)$
- Seeing it is $\Theta(n)$ is harder, but if $O(n^2)$ is acceptable, then we don't need to analyze further

Worst Case Analysis

- Efficiency depends not only on input size, but also what kind of input
- On certain kinds of cases, algorithm may perform significantly better/worse than on others
- Since we want algorithm to work for all possible valid cases, when we analyze, we should always assume the **worst case** and try to ensure worst case behavior is good enough

Worst Case Analysis: Example

```
bool found = false;
for(int i = 0; i < n and not found; i++)
    if(a[i] == target)
        found = true;
```

- Loop will stop as soon as target is found
- Best case: target is at the beginning of array $\Rightarrow O(1)$ time
  - Not really useful for writing fast or understanding/fixing slow algorithms
- Worst case: target is at the end of array $\Rightarrow O(n)$ time

Average Case Analysis

```
bool found = false;
for(int i = 0; i < n and not found; i++)
    if(a[i] == target)
        found = true;
```

- Target is somewhere in the middle of the loop $\Rightarrow O\left(\frac{n}{2}\right) = O(n)$
- Sometimes, worst case analysis is too pessimistic, but much simpler than average case analysis, so we will mostly stick to worst case
  - Ok for the same reason that upper bound analysis is ok

Large-Scale Analysis Gives Insight into Why Programs Are Slow and How to Fix

- Why so slow?
  - Should not work according to analysis ⇒ algorithm is not good enough
  - Should work according to analysis ⇒ implementation is not good enough
  - Computing the rough efficiency gives insight into the structure of the algorithm and understanding where the slowness is coming from
- How to fix
  - If algorithm not good enough, need to completely rethink our approach
    - **If you don't do this, then no matter how much you micro-optimize, your program will still be too slow!**
  - Fix the part of the program with worst efficiency, this is the bottleneck

Applying Large-Scale Analysis in Practice to Estimate Running Time

1. Figure out the rough efficiency $O(f(n))$ of the algorithm's running time as function of input size
2. Take expected input size N and compute $f(N)$
3. Multiply by actual number of seconds it takes per operation
   - Modern computers can do $\sim 10^9$ operations per second
   - In practice, because each primitive operation in modern programming languages translates to several CPU operations, we should take this number to be $\sim 10^8$ operations per second instead
   - 1 operation takes 10^{-8} seconds

- Note: estimate will be off by some constant factor, but this is ok

Applying Large-Scale Analysis in Practice to Check if Algorithm Works in Time Limit

1. Estimate running time
2. Check if it fits within time limit

- This can be reversed (easier)

1. Find number of operations we can do in the given amount of time
   - 1 second = 10^8 operations
2. Check if number of operations we are doing fits this limit

Exercise

- $O(n)$ for $n = 10^5$
- $O(1)$ for $n = 10^{10}$
- $O(n^2)$ for $n = 10^5$
- $O(n \log n)$ for $n = 10^5$
- $O(n \log n)$ for $n = 100$
- $O(n^2 \log n)$ for $n = 10^4$
- $O(n^2)$ for $n = 10^4$
- $O(n^2 2^n)$ for $n = 18$

- How long will a program with the following running times run on the following input sizes?
- Will they run under 3 seconds?

Exercise

- $O(n)$ for $n = 10^5$ ⇒ Yes
- $O(1)$ for $n = 10^{10}$ ⇒ Yes
- $O(n^2)$ for $n = 10^5$ ⇒ No
- $O(n \log n)$ for $n = 10^5$ ⇒ Yes
- $O(n \log n)$ for $n = 100$ ⇒ Yes
- $O(n^2 \log n)$ for $n = 10^4$ ⇒ No
- $O(n^2)$ for $n = 10^4$ ⇒ Risky
- $O(n^2 2^n)$ for $n = 18$ ⇒ Yes

Large-Scale Analysis Gives Insight into How to Design Efficient Programs

- Given constraints of the problem, we can *reverse engineer* to find out what target efficiency an acceptable solution should have
- Eliminate solutions that are “clearly too inefficient”
- Target efficiency sometimes gives a hint on what the structure of the solution should be
- Helps us narrow down to the correct solution

Applying Large-Scale Analysis in Practice to Guide Program Design

1. Given the expected input size and time limit, figure out highest possible efficiency an acceptable solution must have
2. Given the target efficiency, come up with an algorithm

Exercise

- $n = 10^5$, 3 seconds
- $n = 100$, 3 seconds
- $n = 1000$, 3 seconds
- $n = 10^4$, 6 seconds
- $n = 10^{18}$, 60 seconds
- $n = 10^9$, 3 seconds
- $n = 10^7$, 3 seconds
- $n = 54$, 3 seconds

- Given the following maximum input sizes and time limit, what target efficiency should your algorithm have?

Exercise

- $n = 10^5$, 3 seconds $\Rightarrow O(n \log n)$
- $n = 100$, 3 seconds $\Rightarrow O(n^4)$ risky, $O(n^3)$ safe
- $n = 1000$, 3 seconds $\Rightarrow O(n^2)$
- $n = 10^4$, 6 seconds $\Rightarrow O(n^2)$ risky
- $n = 10^{18}$, 60 seconds $\Rightarrow O(\log n)$
- $n = 10^9$, 3 seconds $\Rightarrow O(\sqrt{n})$
- $n = 10^7$, 3 seconds $\Rightarrow O(n)$
- $n = 54$, 3 seconds $\Rightarrow O(2^{\frac{n}{2}})$

Exercise

- $O(n)$
- $O(n^2)$
- $O(n^3)$
- $O(n^4)$
- $O(2^n)$

- How much bigger of an input can a program with the following running times process in the same time, given a computer twice as fast?

Exercise

- $O(n)$ $\Rightarrow 2$ times
- $O(n^2)$ $\Rightarrow \sqrt{2} \approx 1.414$ times
- $O(n^3)$ $\Rightarrow \sqrt[3]{2} \approx 1.2599$ times
- $O(n^4)$ $\Rightarrow \sqrt[4]{2} \approx 1.1892$ times
- $O(2^n)$ $\Rightarrow +1$

The Lesson

- Buying a faster computer is not enough
- All the effort put into making computers faster is wasted with bad algorithms
- Algorithm analysis is important!
- Having good algorithms is important!

The Model of Computation We Will Be Using

- Assume all of the following operations take $O(1)$ time
  - Variable declaration/read/write
  - Add/subtract/multiply/divide/compare two `int`s (or variants thereof)
  - Call/return from a function
  - Read/write to a single cell of an array
- Assume all of the following operations take $O(n)$ time
  - Declare an array of size n
  - Read a string of length n
  - Print a string of length n

Scarier But Official Words

Not-Scary Words	**Official Words**
Large-Scale Analysis	Asymptotic Analysis
Rough Efficiency	Asymptotic Complexity
Running Time	Time Complexity

Practice Problems

- https://progvar.fun/problemsets/sums-and-asymptotics