

Performance: theory and practice

General observations

- Performance usually matters
 - Small improvements are less important
 - Sometimes, huge differences are possible
- Measuring performance accurately is hard
- So is predicting it without measuring it

Performance matters

- Usually two components
 - fixed overhead
 - related to size of input
- Usually two dimensions
 - Time
 - Space
- You can often trade one for another

How do we characterize performance?

- Usually, we express execution properties (time, space, etc) in terms of properties of the input (length, etc.)
- Relative measurements are often more useful than absolute ones
- We might give either average or worst case, possibly amortized
- Many degrees of rigor are possible

Asymptotic representations

- We often want to know approximately “how good (bad) it is” even if we don’t (and can’t) know exactly
 - Machines and compilers differ
 - We may wish to disregard fixed overhead...
 - ...or constant multiples
- One way to get the right amount of imprecision is the $O(f(n))$ notation

The $O(f(n))$ notation

- Introduced by Paul Bachmann in 1892
- Loosely speaking, $O(f(n))$ means “asymptotically no larger than a suitable multiple of $f(n)$,” where $n > 0$
- More precisely, $g(n) = O(f(n))$ means that there are constants K and N such that $|g(n)| \leq K|f(n)|$ whenever $n \geq N$.

Examples of O-notation

- $42 = O(1)$
- $3n + 42 = O(n)$
- $5n^2 - 3n + 7 = O(n^2)$
- $1^2 + 2^2 + \dots + n^2 = n^3/3 + n^2/2 + n/6 = O(n^3)$
- Loosely: Pick the fastest growing term and discard constant multiples

Related notations

- O-notation refers only to upper bounds
- To express a similar lower bound, we use Ω (omega) instead of O
- If a function is simultaneously an upper and lower bound, we use Θ (theta), so that saying that $g(n) = \Theta(f(n))$ says that $g(n)$ gets arbitrarily close to a multiple of $f(n)$ when n is large enough

The importance of these notations

- It usually doesn't matter how a program performs on small inputs
- For large inputs, these notations show what dominates performance
- Practical calibration, for input size n :
 - $O(1)$: Ideal, but usually impossible
 - $O(n)$: Usually the best possible, often unattainable
 - $O(n \log n)$: Almost as good as $O(n)$
 - $O(n^2)$: OK in toy programs but not for serious purposes
 - $O(n^3)$: Hopeless even for toy programs

Sometimes algorithms vary

- Algorithms sometimes perform poorly
 - Quicksort is usually $O(n \log n)$ but can be $O(n^2)$ if the input is unfortunate
 - Self-adjusting data structures may pause from time to time to adjust themselves
- We might therefore talk about
 - Worst-case performance
 - Average performance
 - Amortized performance

What are we measuring? (harder than it sounds)

- Theory
 - Do we assume that adding two integers takes constant time?
 - Even if they are of unbounded precision?
- Practice
 - How do we account for system interference?
 - What about caching?

Real computers have bounded memory

- On a machine with unbounded memory
 - integers would need unbounded precision
 - $m+n$ would take $O(\log(|m|+|n|))$ time
 - claiming $O(n)$ would be problematic
- Once we fix a word size, we can treat addition as taking $O(1)$ time
- Therefore, distinguishing between $O(n)$ and $O(n \log n)$ can be tricky

A concrete example (HW1, part 2)

- Assume that we have a string package in which concatenating two strings takes $O(\text{length}(\text{result}))$ time.
- How long does the following loop take?

```
s = "";  
while (--n >= 0)  
    s = s + x;
```

Analyzing the loop

```
s = "";  
while (--n >= 0)  
    s = s + x;
```

$O(1)$ (for `s = ""`)
Each iteration is $O(1)$ (for `--n >= 0`)
Each iteration is $O(\text{length}(x) \cdot n)$ (for `s = s + x`)

$O(\text{length}(x) \cdot (1 + 2 + \dots + n))$
 $= O(1 + 2 + \dots + n)$
 $= O(n^2)$

File-system directories have similar problems

- Typically linear search, for reasons of
 - reliability
 - laziness
- Inserting an entry into a directory with n entries takes $O(n)$ time
- Creating a directory with n entries takes $O(n^2)$ time (Ouch!)

Fast string duplication

- Preallocate memory for the result

```
s = "";  
s.reserve(x.length() * n);  
while (--n >= 0)  
    s += x;
```
- Advantage: $O(n)$ time instead of $O(n^2)$
- Disadvantage: requires cooperation with string class

Another approach

```
string dupl(string x, unsigned n)  
{  
    string r;    // null by default  
    if (n) {  
        r = dupl(x, n/2);  
        r += x;  
        if (n % 2)  
            r += x;  
    }  
    return r;  
}
```

Measuring performance in practice

- Computers are faster than stopwatches
- Sources of interference:
 - operating systems
 - caches and other buffers
 - optimizers
 - hardware oddities
 - bugs
- Accurate measurement is *hard*!

A measurement example

- How long do subroutine calls take?

```
void churn(int n) {  
    if (n > 0)  
        churn(n-1);  
}
```
- Timings for $n=0, 1, \dots$: 0.2, 0.4, 0.7, 0.9, 1.1, 1.3, 4.2, 7.0, 10.0, 12.8, ...
- With optimization, it is nicely linear!

What is going on here?

- This particular machine has a stack cache in the processor chip
- When recursively nested calls get too deep, the code must flush the cache
- When optimization is turned on, the compiler turns the recursion into iteration

Another example: memory allocation

- Ideally, allocating a block of memory should take $O(1)$
- If n blocks are already allocated in memory, many implementations take $O(n)$ to allocate one more (worst case)
- Allocating n blocks therefore takes $O(n^2)$ in the worst case

Another timing example

- Here is a program fragment

```
int x[1000000];  
for (int i = 0; i < 1000000; ++i)  
    x[i] = i;
```
- What does it cost to replace

```
int x[1000000];
```


by

```
vector<int> x(1000000);
```


?
Expected a factor of 2; got nearly 5
because of default "debug mode"

Benchmark detectors

- Compiler vendors care about reported performance
- Reviewers tend to use widely known benchmarks
- Therefore, some compilers check whether they are running a known benchmark, and cheat if they are!

Other hazards

- Memory fragmentation may inflate space usage
- Garbage collection may introduce unpredictable delays
- A virtual-memory operating system may interact with timing in weird ways

Advice

- Think about overall performance as early as possible, especially to avoid $O(n^2)$ or worse in space or time
- Don't worry too much about detailed performance until you can measure it
- Expect the measurements to be surprising
- Good performance is hard to obtain

Homework assignment #3 (due Monday)

- What is the asymptotic performance of the `dup1` program? Prove it.
- Experiment with the computer you normally use to find an aspect of its performance that could be dramatically improved. Use `malloc` or file-system performance only as a last resort.