Algorithm Performance

(the Big-*O*)

Lecture 6

Today:

- Worst-case Behaviour
- Counting Operations
- Performance Considerations
- Time measurements
- Order Notation (the Big-*O*)

Pessimistic Performance Measure

- Often consider the *worst-case* behaviour as a benchmark.
  - make guarantees about code performance under all circumstances

- Can predict performance by counting the number of “elementary” steps required by algorithm in the worst case
  - derive total steps (T) as a function of input size (N)

Analysis of `dup_chk()`

Q. What is N?

- The number of elements in the array

```
      int dup_chk(int a[], int length) {
1       int i = length;
N + 1   while (i > 0) {
N           i--;
N           int j = i - 1;
i + 1       while (j >= 0) {
i               if (a[i] == a[j]) {
                    return 1;
                }
i               j--;
            }
        }
1       return 0;
      }
```

Outside of loop: 2 (steps)

Outer loop: $3N + 1$

Inner loop: 3`i`$+ 1$ for all possible `i` from 0 to $N - 1$.

$= 3/2\ N^2 - 1/2\ N$

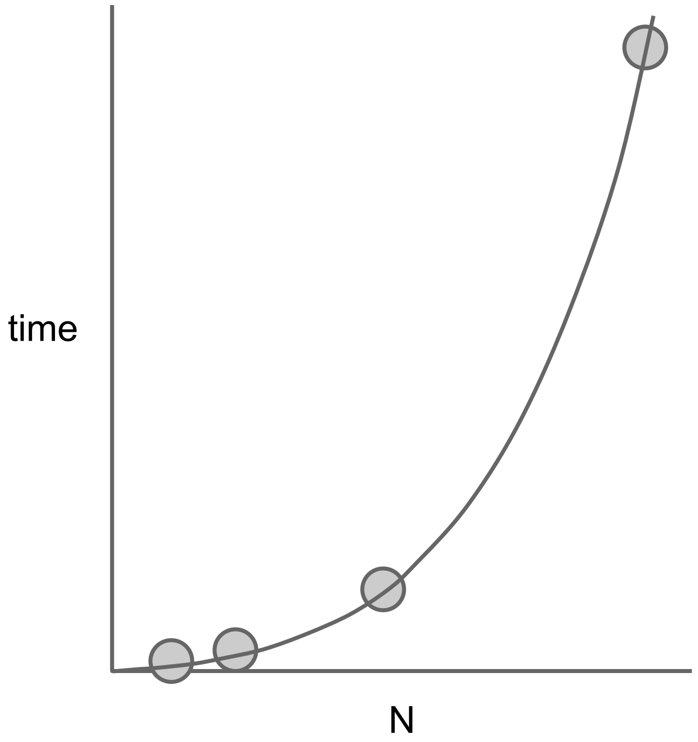
Grand total $= 3/2\ N^2 + 5/2\ N + 3$

A *quadratic* function!

Empirical Measurement

- Another graph - a quadratic this time!
- Confirms predictions: *doubling* (x2) the input size leads to **quadrupling** (x4) the running time

N	time (in ms)
10,000	89
20,000	365
40,000	1,424
100,000	9,011



2D Maximum Density Problem

Problem: Given a 2-dimensional array (NxN) of integers, find the 10x10 swatch that yields the largest sum.

Applications:

- Resource management and optimization
- Finding brightest areas of photos

Algorithm / Code?

- Simple approach: Try all possible positions for the upper left corner
    - $(N-10)$x$(N-10)$ of them
    - use a nested loop
- Total each swatch using a 10x10 nested loop
- A *brute-force* approach!
    - Generate a possible solution [naively]
    - Test it [naively]

In C

Precise accounting:

$348N^2 - 6956N + 34762$ operations

```
          int max10by10(int a[N][N]) {
             int best = 0;
             for (int u_row = 0; u_row < N-10; u_row++) {
x(N-10)         for (int u_col = 0; u_col < N-10; u_col++) {
     x(N-10)       int total = 0;
                   for (int row = u_row; row < u_row+10; row++) {
           x10        for (int col = u_col; col < u_col+10; col++) {
                         total += a[row][col];      11           10
                      }                  10
                   }
                   best = max(best, total);
                }
             }
             return best;
          }
```

Approximate Method:

Count the *barometer instructions*, the instructions executed most frequently. Usually, in the innermost loop.

Innermost loop: 11 + 10 + 10 = 31 ops

Total = 31 **x10 x(N-10)x(N-10)=310N^2**

Which Performance Measurement?

- Empirical timings
  - run your code on a real machine with various input sizes
  - plot a graph to determine the relationship
- Operation counting
  - assumes all elementary instructions are created equal
- Actual performance can depend on much more than just your algorithm!

Running Time is Affected By . . .

- CPU speed
- Amount of main memory
- Specialized hardware (e.g., graphics card)
- Operating system
- System configuration (e.g., virtual memory)
- Programming Language
- Algorithm Implementation
- Other Programs
- . . .

Comparing Algorithm Performance

- There can be many ways to solve a problem, i.e., different algorithms that produce the same result
  - e.g., There are numerous sorting algorithms.
- Compare algorithms by their behaviour for large input sizes, i.e., as N gets large
  - On today's hardware, *most* algorithms perform quickly for small N
- Interested in growth rate as a function of N
  - e.g., Sum an array: *linear* growth $= O(N)$
  - e.g., Check for duplicates: *quadratic* growth $= O(N^2)$

Order Notation (the Big-*O*)

- Suppose we express the number of operations used in our algorithm as a function of N, the size of the problem
- Intuitively, take the dominant term, remove the leading constant, and put $O(\ ...\ )$ around it

- E.g., $f(N) = 348N^2 - 6956N + 34762 \longrightarrow \boxed{O(N^2)}$

Formalities of the Big-*O*

- Given a function $T(N)$, we say $T(N) = O(f(N))$ if $T(N)$ is at most a constant times $f(N)$, except perhaps for some small values of N
- Properties:
  - constant factors don't matter
  - low-order terms don't matter
- Rules:
  - For any k and any function $f(N)$, $k \cdot f(N) = O(f(N))$
    - e.g., $5N = O(N)$
    - e.g., $\log_a N = O(\log_b N)$ - why?
    - Q. Do leading constants really not matter?

Leading Constants - Experiment

Of course, constant factors affect performance

- e.g., If two different algorithms run in $f_1(N) = 20N^2$ and $f_2(N) = 2N^2$, respectively, you would expect Algorithm 2 to run 10 times faster
- e.g., Similarly, a 10x faster machine running Algorithm 1 would have the same running time
- Big-O hides leading constants - *a hardware independent analysis*

Cray Supercomputer

17.6×10^{15} instructions per second
runs optimized dup_chk( ) code from last time
$f(N) = 3/2\ N^2 + 5/2\ N + 3$

vs

iMac Desktop Personal Computer (2011)

40×10^9 instructions per second
runs an unoptimized, different dup_chk ( )
$f(N) = 30N \log N + 5N + 4$

Experimental Results

N	iMac	Cray
100,000	1.2 ms	850 ns
10^6	15 ms	85 µs
10^7	0.2 s	8.5 ms
10^8	2 s	0.85 s
10^9	22 s	1.75 min
10^{10}	4.2 min	2:22 hr
10^{11}	56 min	10 days
10^{12}	8:20 hr	2.7 years

Conclusions:

- Cray runs $O(N^2)$ algorithm
- iMac runs $O(N \log N)$ algorithm which runs faster than Cray for large N (10^9 and beyond)
- Thus slow computer + no opt + $O(N \log N)$ >> fast computer + optimization + $O(N^2)$ algorithm
- **Rule of Thumb: The slower the function grows, the faster the algorithm**
- For the $O(N^2)$ Cray, a 10x increase in N leads to roughly a 100x increase in running time
- For the $O(N \log N)$ iMac, a 10x increase in N leads to roughly a 10x increase in running time (for the N), plus a little (for the $\log N$)

Acknowledgement

These slides are the work of Brad Bart (with minor modifications)