Announcement

- Assignment 2 will be posted tonight
  - Due Jan. 23.
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()`

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

Q. What is $N$?
- The number of elements in the array

Outside of loop: 2 (steps)

Outer loop: $3N + 1$

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

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

A quadratic function!

$1 + 4 + 7 + ... + 3(N-1) + 1$
Some Math

\[ 1 + 4 + 7 + \ldots + 3(N-1) + 1 \]
Some Math

\[ 1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \times \frac{N}{2} \]
Some Math

\[ 1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \times \frac{N}{2} \]
\[ = \frac{1}{2} \times (3N-1) \times N \]
Some Math

\[1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \times \frac{N}{2}\]

\[= \frac{1}{2} \times (3N-1) \times N\]

\[= \frac{1}{2} \times (3N^2 - N)\]
1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \times N/2 \\
= 1/2 \times (3N-1) \times N \\
= 1/2 \times (3N^2 - N) \\
= 3/2 \times N^2 - 1/2 \times N
Some Math

\[1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \times N/2\]
\[= 1/2 \times (3N-1) \times N\]
\[= 1/2 \times (3N^2 - N)\]
\[= 3/2 \times N^2 - 1/2 \times N\]
Some Math

\[1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) * \frac{N}{2}\]
\[= \frac{1}{2} * (3N-1) * N\]
\[= \frac{1}{2} * (3N^2 - N)\]
\[= \frac{3}{2} * N^2 - \frac{1}{2} * N\]

Observation 1: The \(\frac{1}{2} * N\) term doesn’t matter very much
Some Math

$$1 + 4 + 7 + \ldots + 3(N-1) + 1 = (3N-3+1 + 1) \cdot \frac{N}{2}$$

$$= \frac{1}{2} \cdot (3N-1) \cdot N$$

$$= \frac{1}{2} \cdot (3N^2 - N)$$

$$= \frac{3}{2} \cdot N^2 - \frac{1}{2} \cdot N$$

Observation 1: The $1/2 \cdot N$ term doesn’t matter very much

Observation 2: Arithmetic series have $N^2$ leading terms
  1. As $N$ gets bigger, the last number that we add is bigger
  2. The number of pairs of numbers is bigger
Analysis of `dup_chk()`

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

Q. What is \( N \)?
- The number of elements in the array

Outside of loop: 2 (steps)

Outer loop: \( 3N + 1 \)

Inner loop: \( 3i + 1 \) for all possible \( i \) from 0 to \( N - 1 \).

\[ = \frac{3}{2} N^2 - \frac{1}{2} N \]

Grand total = \( \frac{3}{2} N^2 + \frac{5}{2} N + 3 \)

A quadratic function!

Observation: The \( \frac{5}{2} \times N \) and 3 terms don't matter very much
Empirical Measurement

- Another graph - a quadratic this time!
- Confirms predictions: **doubling** (2x) the input size leads to a **quadrupling** (4x) of the running time.

<table>
<thead>
<tr>
<th>$N$</th>
<th>time (in ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>89</td>
</tr>
<tr>
<td>20000</td>
<td>365</td>
</tr>
<tr>
<td>40000</td>
<td>1424</td>
</tr>
<tr>
<td>100000</td>
<td>9011</td>
</tr>
</tbody>
</table>
2D Maximum Density Problem

Problem: Given a 2-dimensional array $(N \times N)$ 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
  - \((N-9) \times (N-9)\) of them
  - use a nested loop
- Total each swatch using a 10x10 nested loop
- A \textit{brute-force} approach!
  - Generate a possible solution [naively]
  - Test it [naively]
In C

```c
int max10by10(int a[N][N]) {
    int best = 0;
    for (int u_row = 0; u_row < N-9; u_row++) {
        for (int u_col = 0; u_col < N-9; u_col++) {
            int total = 0;
            for (int row = u_row; row < u_row+10; row++) {
                for (int col = u_col; col < u_col+10; col++) {
                    total += a[row][col];
                }
            }
            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 = $31 \times 10 \times (N-9) \times (N-9) = 310N^2$
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 $O(\ldots)$ around it.

- E.g., $f(N) = 348N^2 - 6956N + 34762 \rightarrow O(N^2)$
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 $f_2(N) = 3N^2$, 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.

<table>
<thead>
<tr>
<th>Cray Supercomputer</th>
<th>iMac Desktop Personal Computer (2011)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.6 x $10^{15}$ instructions per second runs optimized dup_chk( ) code from last time $f(N) = 3/2 N^2 + 5/2 N + 3$</td>
<td>40 x $10^{9}$ instructions per second runs an unoptimized, different dup_chk( ) $f(N) = 30N \log N + 5N + 4$</td>
</tr>
</tbody>
</table>
Experimental Results

<table>
<thead>
<tr>
<th>$N$</th>
<th>iMac</th>
<th>Cray</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>1.2 ms</td>
<td>850 ns</td>
</tr>
<tr>
<td>$10^6$</td>
<td>15 ms</td>
<td>85 μs</td>
</tr>
<tr>
<td>$10^7$</td>
<td>0.2 s</td>
<td>8.5 ms</td>
</tr>
<tr>
<td>$10^8$</td>
<td>2 s</td>
<td>0.85 s</td>
</tr>
<tr>
<td>$10^9$</td>
<td>22 s</td>
<td>1.75 min</td>
</tr>
<tr>
<td>$10^{10}$</td>
<td>4.2 min</td>
<td>2:22 hr</td>
</tr>
<tr>
<td>$10^{11}$</td>
<td>56 min</td>
<td>10 days</td>
</tr>
<tr>
<td>$10^{12}$</td>
<td>8:20 hr</td>
<td>2.7 years</td>
</tr>
</tbody>
</table>

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 $\log N$).
Some Plots

100\log(x) + 60 (yellow)
0.3x + 200 (purple)
0.18x\log(x) (green)
0.0001x^2 + 0.2x + 50 (red)
0.00000005x^3 (black)
Some Plots

- $100\log(x) + 60$ (yellow)
- $0.3x + 200$ (purple)
- $0.18x\log(x)$ (green)
- $0.0001x^2 + 0.2x + 50$ (red)
- $0.00000005x^3$ (black)
What Does It All Mean?

- A carefully crafted algorithm can make the difference between a usable and an useless piece of software.
- E.g., If it costs one algorithm 0.5s to search 1 billion bank records and another one 0.005s.
- E.g., Or, if $10^9$ isn’t “big” how about Google?
- E.g., Real-time systems - where a nearly instant response is required.
- “You can’t make a racehorse of a pig, but you can make a very fast pig.”
As N Gets Large, The Algorithm is Most Important

“You can’t make a racehorse of a pig, but you can make a very fast pig.”

“When you find yourself trying all sorts of clever implementation tricks to speed up an algorithm, you should step back for a moment and ask if you’re trying to make a racehorse out of a pig. It might be more productive to use your time to look for a racehorse.”

“It’s important to know whether an efficient algorithm is possible. There’s no sense spending time looking for a unicorn. In this situation, the best you can hope for is a fast pig and a short racecourse.”

- Lou Hafer, SFU CS
Big-O and Barometer Instructions

Rule of thumb:
The frequency of the barometer instructions will be proportional to the big-O running time

So, find the most frequent operation(s) and count them!
int max10by10(int a[N][N]) {
    int best = 0;
    for (int u_row = 0; u_row < N-9; u_row++) {
        for (int u_col = 0; u_col < N-9; u_col++) {
            int total = 0;
            for (int row = u_row; row < u_row+10; row++) {
                for (int col = u_col; col < u_col+10; col++) {
                    total += a[row][col];
                }
            }
            best = max(best, total);
        }
    }
    return best;
}

f(N) = 3 \times 10 \times 10 \times (N-9) \times (N-9) = O(N^2)
Polynomials

Rule:
The powers of N are ordered according to their exponents, i.e., $N^a = O(N^b)$ if and only if $a \leq b$
- E.g., $N^2 = O(N^3)$, but $N^3$ is not $O(N^2)$.

Why are lower-ordered terms not included?
- E.g., If your bank account followed $f(N) = N^2 + N + 1$, you would probably care a lot about the lower-ordered terms for small $N$, like $N=5$, as $f(5) = 5^2 + 5 + 1 = 31$. You’ll care about every dollar. But not for larger $N$, like $N=1000$, as $f(1000) = 1000^2 + 1000 + 1 = 1,001,001$. You care most that you have that million bucks, and not much about the $1000$ or the $1$. 
More Rules

3. A logarithm grows more slowly than any other positive power of $N$.
   ○ E.g., $\log_2 N = O(N^{1/2})$.

4. If $f(N) = O(g(N))$ and $g(N) = O(h(N))$ then $f(N) = O(h(N))$.

5. If both $f(N)$ and $g(N)$ are $O(h(N))$ then $f(N) + g(N) = O(h(N))$.

6. If $f_1(N) = O(g_1(N))$ and $f_2(N) = O(g_2(N))$ then $f_1(N) \times f_2(N) = O(g_1(N) \times g_2(N))$
   E.g., $(10 + 5N^2)(10\log_2 N + 1) + (5N + \log_2 N)(10N + 2N \log_2 N)$
Typical Growth Rate Functions

- $O(1)$ – constant time
  - The time is independent of $N$, E.g., list look-up
- $O(\log N)$ – logarithmic time
  - Usually the log is to the base 2, E.g., binary search
- $O(N)$ – linear time, E.g., linear search
- $O(N \log N)$ – E.g., quicksort, mergesort
- $O(N^2)$ – quadratic time, e.g. selection sort
- $O(N^k)$ – polynomial (where $k$ is a constant)
- $O(2^N)$ – exponential time, very slow!