Assignment 2: Corrections

Q1c:
void strcopy2(char *s1, char *s2) {
}
int main () {
    char s2[10] = "copy this";
    char s1[10];
    strcopy2(s1, s2);
}

Q2, first expression: \( 0.001 \log_4 n + \log_2(\log_2 n) \)
- Course website has corrected version
- Due next Wednesday (Jan. 23) in CSIL assignment boxes before class
Algorithm Performance

(the Big-O)
Lecture 7

Today:

- Barometer instructions
- Manipulating Big-O expressions
- Growth rates of common functions
The Story So Far . . .

- Often consider the *worst-case* behaviour as a benchmark
- Derive total steps ($T$) as a function of input size ($N$)
  - use `time` command to measure for various $N$
  - OR . . . count the elementary operations
- Use Big-O to express the growth rate
  - compares algorithms’ behaviour as $N$ gets large
  - leading constants are removed
  - a hardware-independent analysis
Leading Constants (Review)

Leading constants are affected by:

- CPU speed
- other tasks in the system
- characteristics of memory
- program optimization

Regardless of leading constants, a $O(N \log N)$ algorithm will outperform a $O(N^2)$ algorithm as $N$ gets large.
As $N$ Gets Large, The Algorithm is Most Important

A carefully crafted algorithm can make the difference between software that is usable and useless

- e.g., if it costs a $O(N)$ algorithm 0.5s to search 1 billion bank records, but a $O(\log N)$ algorithm 0.005s
- e.g., real-time computing - where a nearly instant response is required
Optimizing Algorithms

If you find yourself trying all sorts of clever implementation tricks to speed up an algorithm, you should:

- Step back and ask if you’re trying to improve a fundamentally inefficient algorithm
- Consider if there might be a better one

...but also realize that there might not be!

It’s more important to reduce your running time by a factor of $N$, than by a factor of 10

- both are important, but not equally important
Big-O and Barometer Instructions

Problem: Given an algorithm, how do you determine its Big-O growth rate?

- Rule of Thumb: the frequency of the algorithm’s barometer instructions will be proportional to its Big-O running time

So, find the most frequent operation(s) and count them!
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
Loops ⇒ Multiply

int max10by10(int a[N][N]) {
    int best = 0;
    for (int u_row = 0; u_row < N-10; u_row++) {
        for (int u_col = 0; u_col < N-10; 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;
}

\[ f(N) = 3 \times 10 \times 10 \times (N-10) \times (N-10) = O(N^2) \]
Loops ⟷ Multiply

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

```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--;
        }
        i--;
    }
    return 0;
}
```

- 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!

\[ = O(N^2) \]
Inner Loops that Depend on Outer Loop

```java
int count = 0;
int N = 1000000;

for (int i = 0; i < N; i++) {
    for (int j = 0; j < i; j++) {
        count++;
    }
}
```

**Thought process**
- i goes from 0 to N-1, so N iterations
- j goes from 0 to i-1, so i iterations
- But i keeps changing!
- 1 iteration, 2 iterations, 3 iterations, ..., N iterations
- Average: N/2 iterations

**Overall:**
- Loops → multiply!
- O(N^2)
Inner Loops that Depend on Outer Loop

```c
int count = 0;
int N = 1000000;

for (int i = 0; i < N; i++) {
    for (int j = 5; j < i; j = j + 3) {
        count++;
    }
}
```

Thought process

- `i` goes from 0 to N-1, so N iterations
- `j` goes from 5 to i-1
  - But `i` keeps changing!
  - 5 iterations, 8 iterations, 11 iterations, ..., approx. N iterations
  - Average: N/2 iterations

Overall:
- Loops → multiply!
- \(O(N^2)\)
**Inner Loops that Depend on Outer Loop**

```java
int count = 0;
int N = 1000000;

for (int i = 1; i < N; i = i*2) {
    for (int j = 1; j < i; j++) {
        count++;
    }
}
```

**Thought process**
- i goes like 1, 2, 4, ... approx. N
  - So, log N
- j goes from 0 to i-1, so i iterations
  - But i keeps changing!
  - 1 iteration, 2 iterations, 4 iterations, ..., approx. N iterations
  - Average is not N/2!

Sometimes you need to do the math, at least partially
- Arithmetic series
- Geometric series
- Logarithms
Function Calls ➡ Substitute

Function calls are not elementary operations
• substitute their Big-O running times

```c
int range(int A[], int n) {
    int lo = min(A, n); // O(N)
    int hi = max(A, n); // O(N)
    return hi - lo; // O(1)
}
```

\[ T(N) = O(N) + O(N) + O(1) = O(N) \]
int search(int A[], int n, int key) {
    if (!sorted(A, n)) {
        return lsearch(A, n, key); // O(N)
    } else {
        return bsearch(A, n, key); // O(logN)
    }
}

T(N) = O(N) + \max(O(N), O(logN))
= O(N) + O(N)
= O(N)

if / else is not an elementary operation

- pick the largest of the two running times
  - remember this is worst case analysis
Rules of the Big-O (Review)

Usually, take the dominant term, remove the leading constant, and put $O(\ldots)$ around it

- Properties:
  - constant factors don’t matter
  - low-order terms don’t matter
Rules about Polynomials

1. 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 \( O(N^2) \)

2. A logarithm grows more slowly than any positive power of \( N \)
   - e.g., \( \log_2 N = O(N^{1/2}) \)

For most functions, can apply L’Hôpital’s Rule:
   - Theorem: If \( \lim_{N \to \infty} \frac{f(N)}{g(N)} \) exists then \( f(N) = O(g(N)) \)
More Rules

3. Transitivity: if \( f(N) = O(g(N)) \) and \( g(N) = O(h(N)) \) then \( f(N) = O(h(N)) \)

4. Addition: \( f(N) + g(N) = O(\max(f(N), g(N))) \)

5. Multiplication: if \( f_1(N) = O(g_1(N)) \) and \( f_2(N) = O(g_2(N)) \) then \( f_1(N) * f_2(N) = O(g_1(N) * g_2(N)) \)

\[
\text{e.g., } (10 + 5N^2)(10\log_2N + 1) + (5N + \log_2N)(10N + 2N \log_2N) \\
\text{Ignore lower order terms} \\
O(N^2 \log N) + O(N^2 \log N) \\
= O(N^2 \log N)
\]
Typical Growth Rates

- $O(1)$ – *constant* time
  - The time is independent of $N$, e.g., array 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!
Some Plots to Convince You

courtesy of [fooplot.com](http://fooplot.com)

- yellow - $O(\log N)$
- purple - $O(N)$
- green - $O(N \log N)$
- red - $O(N^2)$
- black - $O(N^3)$
Some Plots to Convince You

- yellow - $O(\log N)$
- purple - $O(N)$
- green - $O(N \log N)$
- red - $O(N^2)$
- black - $O(N^3)$

Courtesy of fooplot.com
Acknowledgement

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