Reading 21: Cost Models

# Exercise 1: Summarize

Cost models help programmers understand the relative costs of operations in programming languages, guiding efficient code writing by considering factors like list operations, function calls, Prolog search mechanisms, array memory layouts, and compiler optimizations.

# Exercise 2: Demonstrate & Explain

Instead of a revision, I opted to add both versions together with timing to get an average execution time. Both inline and function are looped 100 times and I ran this program multiple times with the same results.

#include <stdio.h>

#include <time.h>

// Simple function to sum numbers from 1 to n

int sum(int n) {

int total = 0;

for (int i = 1; i <= n; i++) {

total += i;

}

return total;

}

int main() {

clock\_t start\_inline, end\_inline, start\_function, end\_function;

double total\_inline\_time = 0, total\_function\_time = 0;

int runs = 100; // Number of times to run the main logic

int n = 1000000;

for (int i = 0; i < runs; i++) {

start\_inline = clock();

int total = 0;

for (int ii = 1; ii <= n; ii++) {

total += ii;

}

end\_inline = clock();

total\_inline\_time += ((double) (end\_inline - start\_inline)) / CLOCKS\_PER\_SEC;

// Function call version

start\_function = clock();

int function\_sum = sum(n);

end\_function = clock();

total\_function\_time += ((double) (end\_function - start\_function)) / CLOCKS\_PER\_SEC;

}

double avg\_inline\_time = total\_inline\_time / runs;

double avg\_function\_time = total\_function\_time / runs;

printf("Average inline code time: %f seconds\n", avg\_inline\_time);

printf("Average function call time: %f seconds\n", avg\_function\_time);

if (avg\_inline\_time < avg\_function\_time) {

printf("Inline code is faster.\n");

} else {

printf("Function call is faster.\n");

}

return 0;

}

// Average inline code time: 0.001053 seconds

// Average function call time: 0.001034 seconds

// Function call is faster.

As you can see from the results, the function call is on average faster than the inline code, which supports the claims in the notes. The reason for this is most likely due to compiler optimizations and smart caching. While the inline code has its loop directly inside the main function, it may have used more cache space than the function version, which is more flexible in terms of where it can be placed and what resources it uses.

Modern compilers are highly optimized and can automatically inline functions when it is beneficial for the program. By explicitly inlining, it interferes with the compiler's optimization strategy in favor of keeping the programmers original intent. Additionally, inlining can increase the size of the binary, leading to more cache misses and potentially slower performance.

# Exercise 3: Inquire

Explain the key differences between cost models and Big O notation.