What is performance?

“How well a design meets all the often conflicting requirements.”

• Metrics?
  – Time & space: Response time, time loading, memory use/loading, etc.
  – Power, cost, ...

• How to optimize performance?
  – Hardware/Software (processor(s), parallelism/speed)
  – Data structures
  – Language & compiler
  – Algorithm (complexity)
Simple limitations on performance improvement
A variation of Amdahl’s Law:

\[ S = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{(T_{old} - T_{component}) + \frac{T_{component}}{n}} \]

\[ = \frac{1}{(1 - f) + \frac{f}{n}} \]

Fun questions to ask:
What if \( n \to \infty \)?
What if \( f \to 1 \)?

\( S \) ... Improvement (Speedup)
\( T \) ... Metric value (\( old \) – prior; \( new \) – after improvement; \( component \) - contribution of component to be improved)
\( n \) ... enhancement, amount improved
\( f = \frac{T_{comp}}{T_{old}} \) ... fraction of the component’s contribution
Example 1:
Some software consists of L components. One of them takes 40 time units to run. The whole system takes 100 time units to run.
How much must that component be improved if the total time should be reduced to 80 units?

\[ S = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{T_{\text{old}}}{(T_{\text{old}} - T_{\text{component}}) + \frac{T_{\text{component}}}{n}} \]

\[ = \frac{1}{(1 - f) + \frac{f}{n}} \]
Example 2:
Suppose that we are considering an enhancement that runs 10 times faster than the original machine but is usable only 40% of the time. What is the overall speedup gained by incorporating the enhancement?

\[
S = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{(T_{old} - T_{component}) + \frac{T_{component}}{n}} = \frac{1}{(1 - f) + \frac{f}{n}}
\]
Example 3:

We want to speed up floating point square root operation. Two designs are proposed. Which one is better?

**Design 1**: The operation uses FPSQR hardware. FPSQR is responsible for 20% of the square root execution time. Speedup this component by a factor of 10.

**Design 2**: Make all the floating point (FP) instructions run 2 times faster. FP instructions are responsible for 50% of the square root execution time.

\[
S = \frac{T_{old}}{T_{new}} = \frac{T_{old}}{(T_{old} - T_{component}) + \frac{T_{component}}{n}} = \frac{1}{(1-f) + \frac{f}{n}}
\]
“My program has an execution time of 10 s”


Consider:

• # of instructions executed
• Each instruction takes a fixed (unitary) amount of time.
• “Steps” of the algorithm.
• Ex. (Pseudo code):
  “Add two numbers”
  “Add all numbers”

Goal: How time (complexity) varies with the size of the input?
Example 4:

```c
int summation(myArray[], n) {
    int sum = 0
    int i = 0
    for(i = 0 to n) {
        sum = sum + myArray[i]
    }
    return sum
}
```

What is the size of the input? $\rightarrow n$

Complexity? # of steps

Total time: $5n + 6$

- if $n = 10$ Time: 56
- if $n = 100$ Time: 506
- if $n = 10000$ Time: 50006
Complexity Analysis

Procedure

1. Write basic operations.

2. Count # times each step runs as a function of n.

3. Derive and solve recurrent formula if necessary.

4. Use the magnitude estimation to assess behavior.
Assume: CPU technology improves one billion times every year.
Algorithm: \(2^N\), \(N = 1000\) \(\Rightarrow\) \(2^{1000} \approx 10^{200}\) steps
Assume each step takes 1 ms to run.
Improvement after a year: from \(10^{200}\) to \(10^{191}\) ms = \(10^{188}\) s
A year is \(\approx 3.1 \times 10^7\) seconds...

How about an algorithm proportional to \(n!\) ?
Assignment

• Read Sections 14.0 – 14.6