1. Introduction

• Computational operations involve some computational process
• Computational processes have some expense or uses some resource
• Expense in terms of:
  o CPU cycles
  o Memory usage, reads/writes
  o Bandwidth
  o Power consumption
  o Size of a circuit (number of gates, embedded systems)
  o Idleness (wasted resources) or utilization
  o Some programs run continuously (operating systems, application servers); performance is measured in load

• Example 1: index-find operation in array-based list vs linked list (constant versus size of list)
  o Array based: random access enables a single memory address lookup
  o Linked list: May require up to n or n/2 “next” operations

• Example 2: insert at start of list operation in array-based list vs linked list
  o Array based: needed to potentially copy entire n sized list
  o Linked list: simple matter of a few memory pointer swaps

• Similar operations on different data structures have a relative cost
  o If constant: scales well with $n$
  o If proportional to $n$: twice as big, twice as long
  o If proportional to $n^2$:
    ▪ Twice as big: 4x as long
    ▪ 10x as big: 100x as long

• Illustrative example:
  o Compute $a^n$ for “large” $n$ ($n \sim 2^{128}$)
  o Naïve algorithm: requires $\sim n$ multiplications
  o Folding@Home: 8 PetaFLOPS = $8 \times 10^{15}$ operations per second
  o Naïve algorithm:
    $$(2^{128} / 8 \times 10^{15}) \times (1 / (60*60*24*365.25)) \sim 1.3478 \times 10^{15}$$
    (1.34 quadrillion years)

• Example 3: Sum of numbers 1…$n$ demo:
  o Constant
2. Algorithm Analysis
   • Given 2 or more algorithms how can you tell which one is “better”?
   • May depend on the data structure used (list operations) or other considerations (particular resource constraint)
   • Empirical tests are good, but not sufficient:
     i. Only provides evidence, not proof
     ii. Testing may be insufficient (may pertain only to the testing conditions, may not test scalability, etc.)
     iii. Building tests may be expensive
   • Algorithm analysis provides a theoretical “proof” of the relative complexity of algorithms

3. Algorithms
   • Definition: An algorithm is a sequence of unambiguous instructions for solving a problem; algorithms must be:
     o Correct – they always give a correct solution
     o Finite – must eventually terminate
   • Algorithms are not code (which are implementations or realizations of algorithms)
   • Algorithms are abstract and invariant (complexity doesn’t change with time, hardware, platforms, etc.); Euclid’s algorithm still takes a linear number of operations to perform even after 3000 years
   • Algorithm: named for Muhammad ibn Musa al-Khwarizmi (Persian Mathematician 780-850)
   • To be useful, an algorithm must also be feasible
     o Executes using a “reasonable” amount of resources
     o Depending on application requirements, milliseconds may be too long, or “days” may be acceptable

4. Algorithm description: Pseudocode
   a. Examples
      • Find maximum element in an array
      • Find average of an array
   b. Good Pseudocode
      • Clear, not too terse nor verbose
      • Does not hide essential details (obfuscation that gives a false measure of its complexity)
      • Abstracts the algorithm
      • Uses short-hand and mathematical notation if applicable
      • Easy to read
   c. Bad Pseudocode
      • Too many details
5. Analysis
   a. Designing an algorithm
      i. Understand the problem
      ii. Choose an approach
      iii. Choose appropriate data structures
      iv. Outline a strategy (algorithm)
      v. Prove correctness
      vi. Evaluate Complexity
      vii. Test it
   b. Resource: must be independent of CPU/memory resources
      • Computers get faster, better (Moore’s law)
      • An algorithm is a (mathematical) process, should be invariant
      • Method does not become more/less efficient with different technology, only more or less practical
      • Empirical run times good for benchmarking, but not for algorithm analysis
      • Small inputs can be executed quite fast
      • Need an objective measure for how an algorithm performs
      • Interested in performance for larger inputs
      • Difference in runtime becomes much more apparent with larger inputs
      • Want to study how it performs as an order of magnitude
      • Efficient algorithms and data structures guide our design decisions
      • Trade-offs may determine our design decisions!
   c. General process for analysis
      1. Identify the input
      2. Identify the input size
      3. Identify an elementary operation (basic operation)
      4. Determine how many times elementary operation gets executed with respect to the input size
      5. Characterize the algorithm using Big-O analysis
   d. Elementary operation
      • Comparisons
      • Addition, divisions, or some other operation
      • Swaps
      • Node traversal (linked list, trees, graphs)
      • Collections: size of individual elements not as important as number (size is constant with respect to number; elementary operation operated on collection, not relevant to individual elements
      • Items related to algorithm constructs (increments in a for-loop, assignment operators) are not relevant: they are out-weighed by, or proportional-to the elementary operation
e. Input size
   • Performance is a function of input (and input size)
   • Finding the maximal element in an array of 10 elements vs 10 million elements
   • How “big” is the input and/or number of inputs
     Example: list: size of list
       o A linear find operation on a list will take time proportional to the list size: double the list size, double the time
       o A constant find operation on a list will take the same amount!
   • Example: graphs: number of vertices
f. Identify how many times executed with respect to the input size
   • May require a summation
   • Requires a careful analysis
   • For this course: only interested in worst-case running time
   • Examples:
     o Finding the mode
     o Finding sum of natural numbers
     o Max subsequence?

6. Asymptotic Analysis
   a. Big-O Notation
      i. Definition: f(n) is O(g(n)) if there are positive constants c, n_0 such that f(n) <= cg(n) for all n >= n_0
      ii. Examples
         1. \(100n^2 + 5n\) is O(n^3)
            • Plot
            • Cross over at \(n = 50 + 5\sqrt{102} \approx 100.4975\)
         c=1, n_0 = 101, g(n) > f(n)
      iii. Constants and lower order terms don’t matter
         1. Identities
            • O(kf(n)) = O(f(n))
            • O(f(n)) + O(g(n)) = O(f(n) + g(n))
            • O(f(n)) * O(g(n)) = O(f(n)*g(n))
• \( O(f_1(n) + f_2(n) + \ldots + f_k(n)) = O(\max(\ldots)) \)

iv. Logarithms
   • If \( x = b^y \), \( \log_b(x) = y \)
   • \( B \) is the base, \( \log \) is the exponent, bases: 2, e, 10
   • Change of base formula: \( \log_b(n) = \frac{\log_c(n)}{\log_c(b)} \)
   • Robustness of analysis (binary versus decimal)
   • \( \log(n), n \log(n) \) algorithms: important because they are \( \text{near-linear} \)

v. Common function classes: polynomials, exponentials, factorials
   \[ 1 < \log(\log(n)) < \log(n) < \log^k(n) < n < n \log(n) < n^2 < n^3 < 2^n < 3^n < n! \]

b. Other notations
   • Omega: asymptotic lower-bound
   • Theta: asymptotic equivalence
   • Little asymptotics (limit definitions, omit)

7. Practical Considerations
   a. Small input sizes may have better performance with worse algorithms (nlog(n) vs 1000n)
   b. Best/Average case analysis
   c. Amortized running time (over the long run of the algorithm—Heap sort)

8. Additional Examples & Demonstrations
   a. Mode finding
      • Given a list of integers, the \textit{mode} is the most common element
      • Quadratic algorithm
      • Linear algorithm
   b. Maximum contiguous subsequence sum problem (book)
      • Problem description:
        Given an array of \( n \) elements, what is the largest sum of a contiguous subsequence
      • Cubic algorithm vs quadratic vs linear
      • Demonstration