02. Algorithm Fundamentals

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Problems

Computational problem: definition of input and desired output

Each is a mathematical object that could be stored in a computer data structure.

Sorting problem
input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.
output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Algorithms

Instance (of a problem): Concrete input datum

Example

$\langle 71, 14, 31, 2, 82 \rangle$

Algorithm: well-defined computational procedure that unerringly transforms input to output

Motivation

Why do we care about algorithms, or algorithmic efficiency?

- Algorithms are automating major parts of the economy: operations research, high frequency trading, machine learning, etc.
- Efficiency can mean the difference between computations being viable, sustainable, for human use versus impractical.
- The principle of not wasting product.
- Intriguing mathematical questions are worth studying in their own right.

Data Structures

Data Structure: method for storing, organizing data

- Data members e.g. head pointer, tail pointer
- *Invariant(s)* defining how the structure must be organized to remain valid, e.g. head points to first node, tail points to last node
- Defined *operations*, each operation is an algorithm that operates on the structure.

Pseudocode

Pseudocode: code-like notation for conveying algorithms

- Goal is clear communication to a human audience
- Not compiled, so no need to be syntactically perfect
- Not software engineering; no need for error checking, modularity, encapsulation, etc.

Algorithm implementer is a specific role and skill set, bridging the gap between scholarly pseudocode and industrial coding practices.

Insertion Sort

```
function INSERTION-SORT(A)
    for j = 2 to A.length do
        key = A[j]
        // Insert A[j] into the sorted sequence A[1 ... j-1].
        i = j - 1
        while i > 0 and A[i] > key do
            A[i+1] = A[i]
            i = i-1
        end while
        A[i+1] = key
    end for
end function
```

Pseudocode Observations

- Algorithm is a function/procedure, input is argument(s)
- No global variables
- Code-like but not compile-able code
- Arrays start at 1
- No error checking or modularity
- Translatable into practically any programming language

Analysis

Analysis: establish how efficient an algorithm is

- Usually a mathematical proof (alternatively empirical evidence)
- Usually analyze for time spent (or disk I/O, space, energy, randomness, etc.)
- Usually summarize resource use by *order of growth* in asymptotic notation; $O(n), \Theta(n^2)$, etc.

RAM model

Computational model: defines how a computer executes an algorithm, specifically enough to measure time (or other resources)

Random Access Machine (RAM):

- "default" computational model, approximates a generic real-world CPU and memory
- CPU has instructions for integer arithmetic, floating point arithmetic, control (jump, call, return, if), logic (or, and, not), data copying.
- one step $\equiv O(1)$ instructions; each pseudocode statement counts as 1 step (except function calls)
- CPU has some $O(1)$ word size, e.g. 32 or 64 bit; one instruction is limited to writing that many bits
- cannot "cheat" by packing $\Theta(n)$ information in one word

Worst-Case Analysis

- In a time analysis, we need to prove how much time insertion sort takes when run
- depends on the type of input, e.g. pre-sorted, completely jumbled, in between
- convention: analyze the *worst case* for the algorithm at hand
- generous to skeptics, conservative for software engineers
- as an exception, sometimes analyze *average case* of deliberately randomized algorithms

Claim: The worst-case time complexity of insertion sort is $\Theta(n^2)$.

Divide-and-conquer

1. **divide** input into several smaller instances of the same problem (often, divide input in half)
2. **"conquer"** by recursively solving all the sub-problems; may involve a simple base case
3. **combine** the many solutions into one coherent solution for the original problem

Merge sort

Merge sort: classical sorting algorithm using divide-and-conquer

divide: chop list of n unsorted elements into two lists of $n/2$ elements each
conquer: merge-sort each unsorted list; if $n \leq 1$, nothing to do
combine: *merge* two sorted lists of $n/2$ elements, into one sorted list of n elements

Merge pseudocode

```
Ensure: A[p ... r] is sorted
function MERGE-SORT(A, p, r)
    if p < r then
        q = ⌊(p + r)/2⌋
        MERGE-SORT(A, p, q)
        MERGE-SORT(A, q + 1, r)
        MERGE(A, p, q, r)
    end if
end function
```

```
Require: p ≤ q < r, A[p ... q] is sorted, A[q + 1 ... r] is sorted
Ensure: A[p ... r] is sorted
function MERGE(A, p, q, r)
    n1 = (q − p + 1), n2 = (r − q)
    let L[1 ... n1 + 1] and R[1 ... n2 + 1] be new arrays
    L[1 ... n1] = A[p ... q]
    R[1 ... n2] = A[q + 1 ... r]
    L[n1 + 1] = R[n2 + 1] = ∞
    i = j = 1
    for k = p to r do
        if L[i] ≤ R[j] then
            A[k] = L[i]
            i = i + 1
        else
            A[k] = R[j]
            j = j + 1
        end if
    end for
end function
```

Merge sort analysis

The worst-case time complexity of merge sort is given by the *recurrence relation*

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Claim: $T \in \Theta(n \log n)$

Claim: Merge sort uses $\Theta(n)$ extra space for L and R. (Observe that at most one merge is happening at any moment, and the largest merge uses $n + 2$ extra array elements.)

Insertion sort versus merge sort

Insertion sort: $\Theta(n^2)$ time, $\Theta(1)$ space

Merge sort: $\Theta(n \log n)$ time, $\Theta(n)$ space

Merge sort is subjectively more complicated

Space vs. time tradeoff (typical)

Efficiency vs. complication tradeoff (typical)

Refactoring complication into algorithm design is usually a win

Asymptotic notation

Asymptotic notation: $\Theta(n^2), O(n^2), \Omega(n^2)$, etc.

Solves some problems with algorithm analysis

- Whole point is how algorithms perform on very large n
- $\implies$ ignore small n
- We count abstract "steps" so constant factors are not meaningful
- i.e. $4n^2$ and $5n^2$ should count as essentially the same
- Want to prioritize speeding up the part of the algorithm that actually accounts for the most time
- Running time functions usually include multiple terms; the asymptotically fastest-growing term *dominates* the running time

Example: insertion sort

Suppose insertion sort's run time is

$$\begin{aligned} T(n) &= (\text{inner loop steps}) + (\text{outer loop steps}) + (\text{outside loop}) \\ &= 3n^2 + 4n + 1 \end{aligned}$$

Let $n = 2$,

$$T(n) = 3(2)^2 + 4(2) + 1 = 12 + 8 + 1 = 21.$$

Let $n = 1,000$,

$$T(n) = 3(1,000)^2 + 4(1,000) + 1 = 3,000,000 + 4,000 + 1.$$

$\implies$ inner loop dominates time, focus on speeding that up

Θ notation

"big-theta": intuitively,

$$\Theta(g(n)) = \{f(n) \ : \ f(n) \text{ grows asymptotically the same as } g(n)\}$$

Formally,

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1, c_2, n_0 \\ \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}.$$

This is set notation, so e.g.

$$3n^2 + 4n + 1 \in \Theta(n^2).$$

Θ-notation indicates a *tight bound:* set of functions upper- **and** lower-bounded by $g(n)$

$O, \Omega, o,$ and ω

Intuitively,

- "big-oh":

$$O(g(n)) = \{f(n) : f(n) \text{ grows asymptotically } \leq g(n)\}$$

- "big-omega":

$$\Omega(g(n)) = \{f(n) : f(n) \text{ grows asymptotically } \geq g(n)\}$$

- "little-oh":

$$o(g(n)) = \{f(n) : f(n) \text{ grows asymptotically } < g(n)\}$$

- "little-omega":

$$\omega(g(n)) = \{f(n) : f(n) \text{ grows asymptotically } > g(n)\}$$

Knuth-style notation abuse

An asymptotic notation expression such a $O(n^2)$ is, technically, a **set** of function objects

$\implies$ a run-time function may be an **element** of one of these sets

Mathematically precise:

$$3n^2 + 4n + 1 \in O(n^2).$$

Some others (notably Knuth) abuse notation to use $=$, which is technically incorrect since the operands are different types, but is arguably more readable:

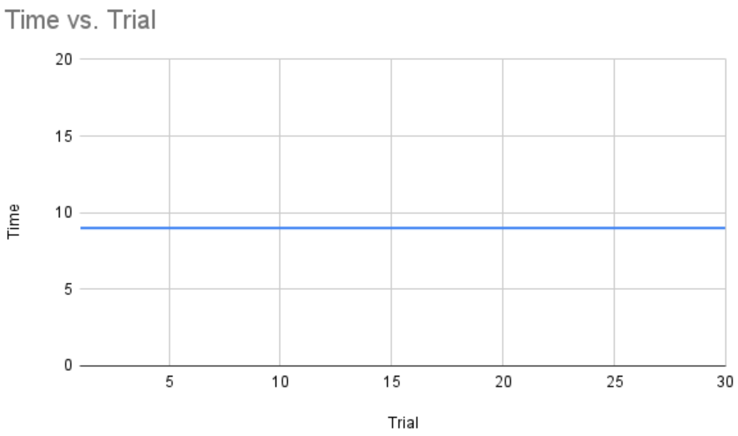
$$3n^2 + 4n + 1 = O(n^2).$$

Kinds of Time Bounds

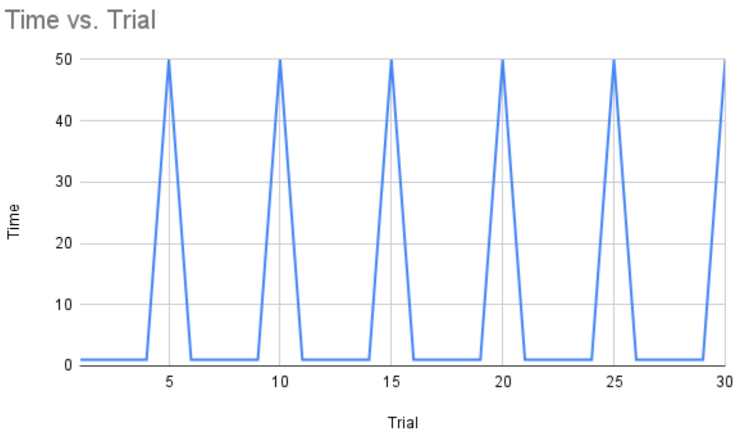
Suppose algorithm A takes...

- $\Theta(n)$ **worst-case time**: for *every* input, A takes $\Theta(n)$ time
- $\Theta(n)$ **amortized time**: for any sequence of k executions, the total time is $\Theta(k \cdot n)$, so the average time of a single execution is $\Theta(n)$
- $\Theta(n)$ **expected time**: A's run time is a random variable whose expected value is $\Theta(n)$
- $\Theta(n)$: no explicit qualifier; by default, this means $\Theta(n)$ worst-case time

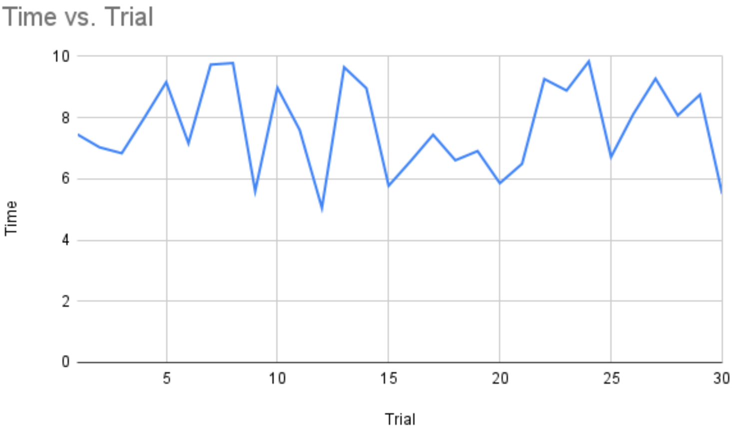
Repeated Trials of $O(1)$ Worst-Case Time



Repeated Trials of $O(1)$ Amortized Time



Repeated Trials of $O(1)$ Expected Time



Kinds of Time Bounds and Goal Setting

Predictability is desireable, so in general, our preference is

1. worst-case
2. amortized
3. expected

Improving the kinds of time bounds is a research area
Example:

- Conventional chained hash tables take $O(1)$ expected time to search, insert, or delete
- *Cuckoo hash tables* take $O(1)$ worst-case time to search or delete, and $O(1)$ expected time to insert

Combining Different Kinds of Time Bounds

In general:

an "amortized" or "expected" qualifier is "contagious"

Examples:

- $\Theta(n)$ worst-case $+$ $\Theta(n)$ amortized $=$ $\Theta(n)$ amortized
- $\Theta(n)$ worst-case $+$ $\Theta(n)$ expected $=$ $\Theta(n)$ expected
- $\Theta(n)$ amortized $+$ $\Theta(n)$ expected $=$ $\Theta(n)$ amortized expected
- $\Theta(n)$ worst-case $\times$ $\Theta(n)$ expected $=$ $\Theta(n^2)$ expected

Exception: Dropping "Amortized"

- Recall: $\Theta(n)$ **amortized time**: for any sequence of k executions, the total time is $\Theta(k \cdot n)$, so the average time of a single execution is $\Theta(n)$
- Observe: the total time is worst-case
- So **when an amortized operation is repeated** $\omega(1)$ **times, the total time drops "amortized" to become worst-case**
- Examples:
  - n repetitions of $O(1)$ amortized $= O(n)$ worst-case
  - n repetitions of $O(\log n)$ amortized $= O(n \log n)$ worst-case
  - 2 repetitions of $O(\log n)$ amortized $= O(\log n)$ amortized

Exception: Dropping "Expected"

- A randomzied algo. has both
  - a faster expected-time bound, and
  - a slower worst-case time bound
- Example: quick sort takes $\Theta(n \log n)$ expected time and $\Theta(n^2)$ worst-case time
- We may choose to use either one
- When adding an expected time bound to a non-expected one,

choose the randomized algo's worst-case bound if that wouldn't be the dominating term

Example: Dropping "Expected"

Algo.: insertion sort then quick sort
Choosing quick sort's expected time bound:

$$\Theta(n^2) + \Theta(n \log n) \text{ expected} = \Theta(n^2) \text{ expected}$$

Choosing quick sort's worst-case time bound:

$$\Theta(n^2) + \Theta(n^2) = \Theta(n^2)$$

Second choice is more favorable