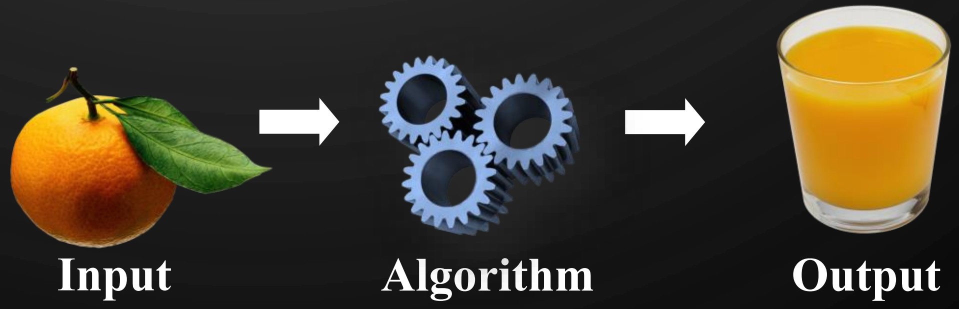
CH 4
ALGORITHM ANALYSIS

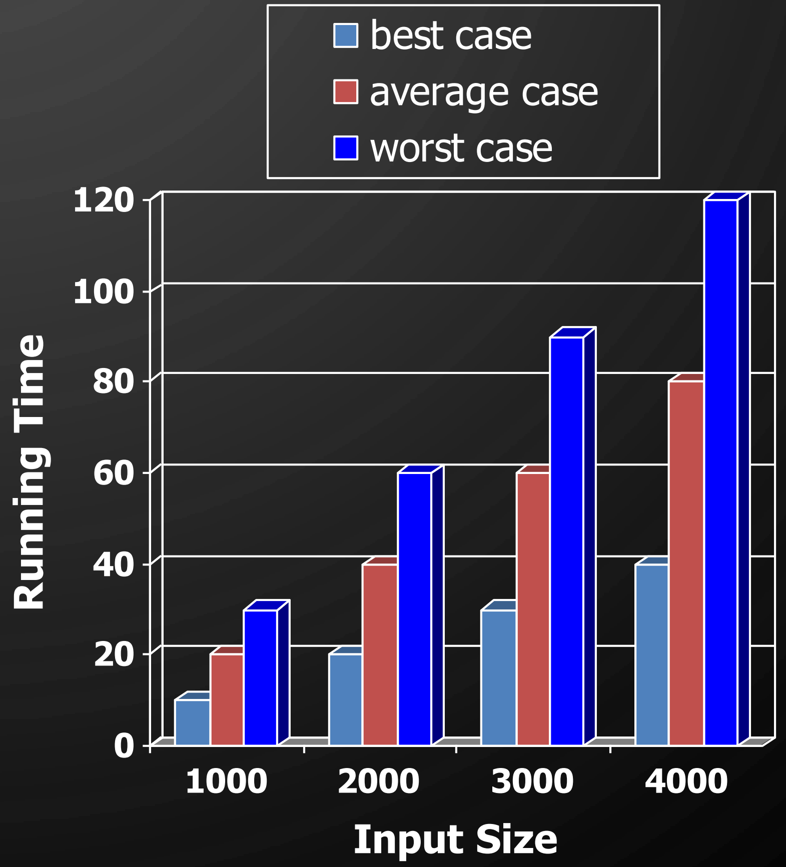
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN JAVA, GOODRICH, TAMASSIA AND GOLDWASSER (WILEY 2016)

ANALYSIS OF ALGORITHMS (CH 4.2-4.3)



RUNNING TIME

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance, and robotics



LIMITATIONS OF EXPERIMENTS

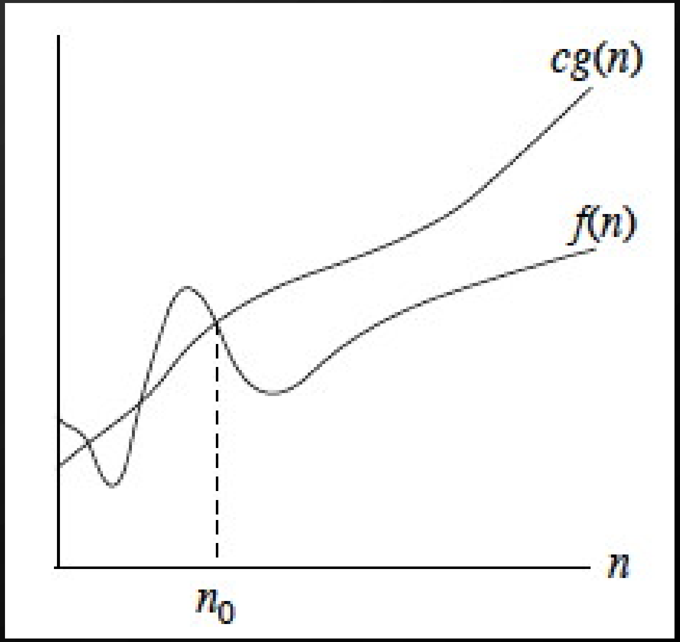
- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

THEORETICAL ANALYSIS

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

BIG-OH NOTATION

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O\big(g(n)\big)$ if there are positive constants c and n_0 such that $f(n) \le cg(n)$ for $n \ge n_0$
  - $f(n)$ - might represent real computation time (measured time, if you will)
  - $g(n)$ - approximation function
- Example: 2n + 10 is O(n)
  - $2n + 10 \le cn$
  - $\frac{10}{c-2} \le n$
  - Pick $c = 3$ and $n_0 = 10$
- To reduce: Strip constants, and take highest order terms
  - Constants do no matter because of limits as n goes to infinity



PRACTICE WITH BIG-OH

- Determine the big-oh approximation for the following functions:

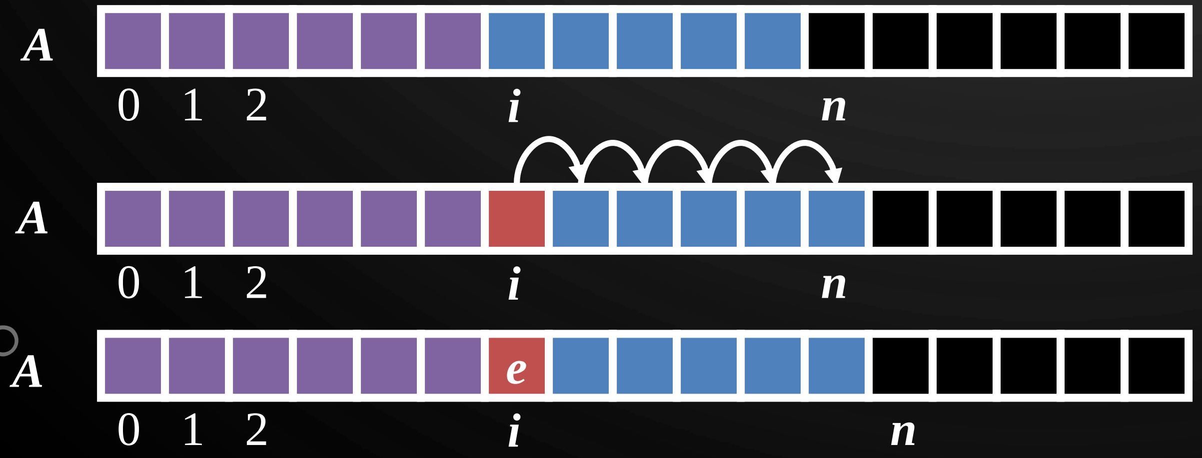
1. 2^{100}
2. $4n^2 + 3n - 10$
3. $n \log n + 100n$
4. $3 * 2^n + 400n^2$
5. $2^{\log n}$
6. $46n^2 + m$
7. $n\sqrt{n} + 23m \log n$
8. $\cos x$

BIG-OH NOTATION FOR ALGORITHMS

- In comparison of algorithms, $f(n)$ is the real (measurable) time an algorithm takes to compute on hardware (tied to an implementation)
  - Again, hard to compare to other algorithms
- To determine big-oh approximation we count the maximum number of steps required by our algorithm
  - Unary and binary math operations, (e.g., $+$, -, $*$, $/$) and single memory accesses are $O(1)$
  - Loops or math operations like summation/product are $O(k)$ where k is the number of iterations performed
- Essentially, we don't care about constants or exact times, we are reasoning about a general trend of n vs $f(n)$

EXAMPLE
ADDING TO AN ARRAY

- To add an entry e into array A at index i, we need to make room for it by shifting forward the $n - i$ entries $A[i], \dots, A[n-1]$



Algorithm Add
Input: **Array** A,
index i, **element** e
1. **for** $k \leftarrow n$ **to** $i + 1$ **do**
2. $\quad A[k] \leftarrow A[k-1]$
3. $A[i] \leftarrow e$
4. $n \leftarrow n + 1$

EXAMPLE
ADDING TO AN ARRAY

- Best case
  - Add at the end of the array
  - One comparison, one copy, one increment
  - $3 = O(1)$, by removal of constants
- Worst case
  - Add at the beginning of the array
  - n comparisons, n copies, $2n$ increments
  - $4n = O(n)$, by removal of constants
- Average case?

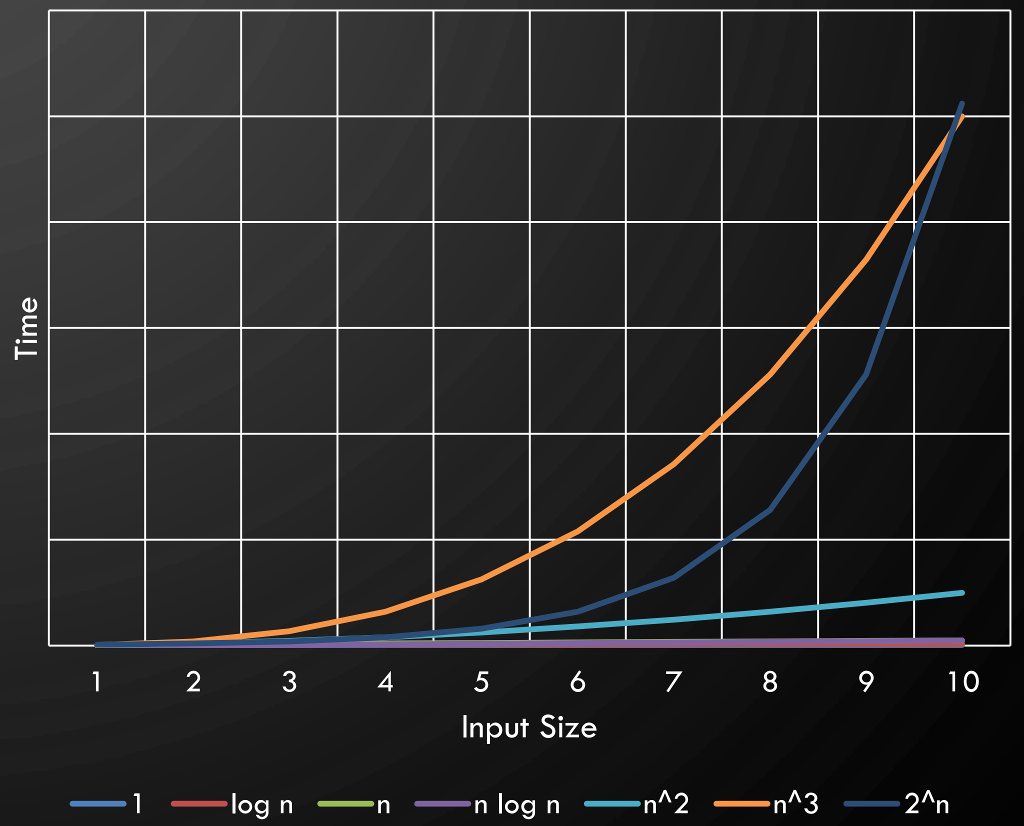
EXERCISES

- Removing from an array
  - Best, average, worst cases
- Inserting at head or tail of linked list
- Removing head of tail of doubly-linked list
- Removing head of singly-linked list
- Removing tail of singly-linked list

SEVEN IMPORTANT FUNCTIONS

- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic $\approx \log n$
  - Linear $\approx n$
  - Linearithmic $\approx n \log n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
  - Exponential $\approx 2^n$

- In a log-log chart, the slope of the line corresponds to the growth rate



BIG-OH ANALYSIS APPLIES TO TIME AND MEMORY

- How about recursion?
  - Each function call uses memory!
- Practice: How much memory does a recursive binary search use?

BIG-OMEGA AND BIG-THETA

- Big-oh describes an upper bound. Similar constructs exist for lower bounds (Big-omega $\Omega\big(g(n)\big)$), "tight" bounds (Big-theta $\Theta\big(g(n)\big)$), strict upper bounds (little-oh $o\big(g(n)\big)$), and strict lower bounds (little-omega $\omega\big(g(n)\big)$)
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Omega\big(g(n)\big)$ if there are positive constants c and n_0 such that $f(n) \geq cg(n)$ for $n \geq n_0$
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $\Theta\big(g(n)\big)$ if there are positive constants c', c'', and n_0 such that $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$
  - To prove: You must show upper and lower bounds hold. Because of this, in CS we often just say big-oh, but really big-theta is more accurate.

BIG-OH VS "WORST" CASE

- Despite common belief, big-oh does not always mean worst case
- Big-oh is an upper bound. So worst-case, average-case, and best case can each have a unique upper bound. It depends what we are describing.
- Similarly, big-omega does not mean best case and big-theta definitely does not mean average case

COMMON PROOF TECHNIQUES FOR THIS CLASS

- Direct proof – using knowledge of axioms and definitions
  - Used for determining theoretical complexity
  - ***Loose*** example
    - Copying takes one operation. My loop runs n times and performs n copies. Therefore the total runtime is $O(n)$
- Contradiction – assume the opposite and reach an impossibility
  - We will see this later in the course, in proving properties of structures
  - **Loose** example
    - Prove: if ab is odd, then a is odd and b is odd. Proof: Assume a is even, then $a = 2j$ for some integer j. Thus $ab = 2(jb)$, implying ab is even. This is a contradiction to our original assumption, thus a cannot be even.
- Induction – not on a test or homework, only for my lectures
- Counterproof by example