Today’s Outline

• Announcements
  – Assignment #1, due Thurs, Oct 4 at 11pm
  – Assignment #2, posted later this week, due Fri Oct 12 at BEGINNING of lecture

• Algorithm Analysis
  – Big-Oh
  – Analyzing code

Ignoring constant factors

• So binary search is $O(\log n)$ and linear search is $O(n)$
  – But which is faster?

• Could depend on constant factors:
  – How many assignments, additions, etc. for each $n$
    • E.g. $T(n) = 5,000,000n$ vs. $T(n) = 5n^2$
    – And could depend on size of $n$ (if $n$ is small then constant additive factors could be more important)
    • E.g. $T(n) = 5,000,000 \log n$ vs. $T(n) = 10 + n$

• But there exists some $n_0$ such that for all $n > n_0$ binary search wins
• Let’s play with a couple plots to get some intuition…

Linear Search vs. Binary Search

Let’s try to “help” linear search:
• Run it on a computer 100x as fast (say 2010 model vs. 1990)
• Use a new compiler/language that is 3x as fast
• Be a clever programmer to eliminate half the work
• So doing each iteration is 600x as fast as in binary search

For small $n$, linear search is faster! But eventually binary search wins.

Asymptotic notation

About to show formal definition of Big-O, which amounts to saying:
1. Eliminate low order terms
2. Eliminate coefficients

Examples:

- $4n + 5$
- $0.5n \log n + 2n + 7$
- $n^2 + 2^n + 3n$
- $n \log (10n^2)$

Examples

True or false?
1. $4+3n$ is $O(n)$
2. $n-2\log n$ is $O(\log n)$
3. $\log n+2$ is $O(1)$
4. $n^{5}$ is $O(1.1^{n})$
### Examples

**True or false?**

1. $4+3n$ is $O(n)$  \( \text{True} \)
2. $n+2\log n$ is $O(\log n)$  \( \text{False} \)
3. $\log n + 2$ is $O(1)$  \( \text{False} \)
4. $n^{50}$ is $O(1.1^n)$  \( \text{True} \)

---

### Big-Oh relates functions

We use $O$ on a function $f(n)$ (for example $n^2$) to mean the set of functions with asymptotic behavior less than or equal to $f(n)$.

So $(3n^3+17)$ is in $O(n^3)$

- $3n^3+17$ and $n^3$ have the same asymptotic behavior.

Confusingly, we also say/write:

- $(3n^3+17)$ is $O(n^3)$
- $(3n^3+17) \in O(n^3)$
- $(3n^3+17) = O(n^3)$

But we would never say $O(n^2) = (3n^3+17)$.

### Formally Big-Oh

**Definition:** $g(n)$ is in $O(f(n))$ if there exist positive constants $c$ and $n_0$ such that $g(n) \leq c f(n)$ for all $n \geq n_0$

To show $g(n)$ is in $O(f(n))$, pick a $c$ large enough to “cover the constant factors” and $n_0$ large enough to “cover the lower-order terms.”

- Example: Let $g(n) = 3n^3+17$ and $f(n) = n^3$.
  - $c = 5$ and $n_0 = 10$ is more than good enough.

This is “less than or equal to”

- So $3n^3+17$ is also $O(n^3)$ and $O(2n^3)$ etc.

### Using the definition of Big-Oh (Example 1)

Given: $g(n) = 1000n$ and $f(n) = n^2$.

Prove: $g(n)$ is in $O(f(n))$.

- A valid proof is to find valid $c$ and $n_0$.
- Try: $n_0 = 1000$, $c = 1$.
- Also: $n_0 = 1$, $c = 1000$.

### Using the definition of Big-Oh (Example 2)

Given: $g(n) = 4n$ and $f(n) = n^2$.

Prove: $g(n)$ is in $O(f(n))$.

- A valid proof is to find valid $c$ and $n_0$.
- When $n=4$, $g(n) = 16$ & $f(n) = 16$; this is the crossing over point.
- So we can choose $n_0 = 4$, and $c = 1$.
- Note: There are many possible choices: ex: $n_0 = 78$, and $c = 42$ works fine.

### Using the definition of Big-Oh (Example 3)

Given: $g(n) = n^4$ and $f(n) = 2^n$.

Prove: $g(n)$ is in $O(f(n))$.

- A valid proof is to find valid $c$ and $n_0$.
- One possible answer: $n_0 = 20$, and $c = 1$.
\[ f(n) = \begin{cases} \Theta(n^2) & \text{if } g(n) \in \Theta(f(n)) \text{ and } g(n) \in \Omega(f(n)) \\ \Omega(n^2) & \text{if } g(n) \in \Omega(f(n)) \text{ and } g(n) \in \omega(f(n)) \\ \omega(n^2) & \text{if } g(n) \in \omega(f(n)) \text{ and } g(n) \in \Omega(f(n)) \\ \omega & \text{if } g(n) \in \omega(f(n)) \text{ and } g(n) \in \omega(f(n)) \\ \end{cases} \]
Which Function Grows Faster?

\[ n^3 + 2n^2 \quad \text{vs.} \quad 100n^2 + 1000 \]

Which Function Grows Faster?

\[ n^0.1 \quad \text{vs.} \quad \log n \]

Which Function Grows Faster?

\[ 5n^5 \quad \text{vs.} \quad n! \]
Nested Loops

for i = 1 to n do
    for j = 1 to n do
        sum = sum + 1
    for i = 1 to n do
        for j = 1 to n do
            sum = sum + 1

More Nested Loops

for i = 1 to n do
    for j = 1 to n do
        if (cond) {
            do_stuff(sum)
        } else {
            for k = 1 to n*n
                sum += 1

Big-Oh Caveats

- Asymptotic complexity (Big-Oh) focuses on behavior for large n and is independent of any computer / coding trick
  - But you can "abuse" it to be misled about trade-offs
    - Example: $n^{1/10}$ vs. $\log n$
    - Asymptotically $n^{1/10}$ grows more quickly
    - But the 'cross-over' point is around $5 \times 10^{17}$
    - So if you have input size less than $2^{58}$, prefer $n^{1/10}$
- Comparing O() for small n values can be misleading
  - Quicksort: $O(n \log n)$ (expected)
  - Insertion Sort: $O(n^2)$ (expected)
  - Yet in reality Insertion Sort is faster for small n's
  - We'll learn about these sorts later

Addendum: Timing vs. Big-Oh?

- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of n
  - Be able to mathematically prove things about performance
  - Yet, timing has its place
    - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
    - Ex: Benchmarking graphics cards
  - We will do some timing in our homeworks
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software? Timing can be useful