CSE 373

OCTOBER 4TH – ALGORITHM ANALYSIS

TODAY'S LECTURE

- **Algorithm Analysis**
  - Asymptotic analysis
  - bigO notation

PROJECT 1

- Checkpoint 1 due at 11:30 pm
- Submit only the files listed in the deliverables section
- If you submit as a group, make sure all files have both team names
- Helpful if you could add a comment on your canvas submission indicating your partner

REVIEW

- **Algorithm Analysis**
  - Testing is for implementations
  - Analysis is for algorithms

REVIEW

- **Algorithm Analysis**
  - Testing is for implementations
  - Analysis is for algorithms
  - Runtime, memory and correctness

REVIEW

- **Algorithm Analysis**
  - Testing is for implementations
  - Analysis is for algorithms
  - Runtime, memory and correctness
  - Best case, average case, worst case

REVIEW

- **Algorithm Analysis**
  - Testing is for implementations
  - Analysis is for algorithms
  - Runtime, memory and correctness
  - Best case, average case, worst case
  - Over groups of inputs, not just one

ALGORITHM ANALYSIS

- Principles of analysis

ALGORITHM ANALYSIS

- **Principles of analysis**
  - Determining performance behavior
  - How does an algorithm react to new data or changes?
  - Independent of language or implementation

ALGORITHM ANALYSIS

- **Example: find()**
  - Sorted v Unsorted
    - How is insert impacted?

ALGORITHM ANALYSIS

- **Example: find()**
  - Sorted v Unsorted
    - How is insert impacted?
  - A sorted array gives us faster find because we can use binary search

ALGORITHM ANALYSIS

- **Example: find()**
  - Sorted v Unsorted
    - How is insert impacted?
  - A sorted array gives us faster find because we can use binary search
  - Can we **prove** that this is the case?

ALGORITHM ANALYSIS

- **Example: find()**
  - Sorted v Unsorted
    - How is insert impacted?
  - A sorted array gives us faster find because we can use binary search
  - Can we ***prove*** that this is the case?

BINARY SEARCH

- Analyzing binary search.
- What is the worst case?

BINARY SEARCH

- **Analyzing binary search.**
- **What is the worst case?**
  - When the item is not in the list

BINARY SEARCH

- **Analyzing binary search.**
- **What is the worst case?**
  - When the item is not in the list
- **How long does this take to run?**

BINARY SEARCH

- **Consider the algorithm**

```
public int binarySearch(int[] data, int toFind){
int low = 0; int high = data.length-1;
while(low <= high){
        int mid = (low+high)/2;
        if(toFind>mid) low = mid+1; continue;
        else if(toFind<mid) high = mid-1; continue;
        else return mid;
}
return -1;
}
```

BINARY SEARCH

- **What is important here?**

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain
  - At second, N/4 elements remain

BINARY SEARCH

- **What is important here?**
  - At each iteration, we eliminate half of the remaining elements.
- **How long will it take to reach the end?**
  - At first iteration, N/2 elements remain
  - At second, N/4 elements remain
  - At the kth iteration?

BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**

BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**
  - When $N/2^k = 1$

BINARY SEARCH

- **At the kth iteration:**
  - $N/2^k$ elements remain.
- **When does this terminate?**
  - When $N/2^k = 1$
- **How many iterations then? Solve for k.**

BINARY SEARCH

- **Solve for k.**

$$N / 2^k = 1$$

BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**

BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**

BINARY SEARCH

- **Solve for k.**

$N \ / \ 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**
  - N can be things other than powers of two

BINARY SEARCH

- **Solve for k.**

$N / 2^k = 1$

$N = 2^k$

$\log_2 N = k$

- **Is this exact?**
- **Where was the error introduced?**
  - N can be things other than powers of two
  - Ceiling and floor rounding

ANALYSIS

- If this isn’t exact, is it still correct?

ANALYSIS

- If this isn’t exact, is it still correct?
- Yes. We care about asymptotic growth.

ANALYSIS

- **If this isn't exact, is it still correct?**
- **Yes. We care about asymptotic growth.**
  - How a the runtime of an algorithm grows with big data

ANALYSIS

- **If this isn't exact, is it still correct?**
- **Yes. We care about asymptotic growth.**
  - How a the runtime of an algorithm grows with big data
- **To incorporate this perspective, we use bigO notation**

BIG-O NOTATION

- **Informally: bigO notation denotes an upper bound for an algorithms asymptotic runtime**

BIG-O NOTATION

- Informally: bigO notation denotes an upper bound for an algorithms asymptotic runtime
- For example, if an algorithm `A` is `O(log n)`, that means some logarithmic function upper bounds `A`.

BIG-O NOTATION

- Formally, a function $f(n)$ is $O(g(n))$ if there exists a c and n_0 such that:
- For all $n \geq n_0$, $f(n) < c*g(n)$
- To prove a function is O(g(n)), simply find the c and n_0

BIG-O NOTATION

- Example: is $5n^3 + 2n$ in $O(n^4)$?
- Can we find a c, n_0 such that:
- $5n^3 + 2n \leq c*n^4$ for all $n \geq n_0$

BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

- Is $5n^3 + 2n$ in $O(n^3)$?

BIG-O NOTATION

- This is an upper bound, so if

$5n^3 + 2n$ is in $O(n^4)$, then

$5n^3 + 2n$ is in $O(n^5)$ and $O(n^n)$

- Is $5n^3 + 2n$ in $O(n3)$?
- Yes, let c be 7 and n > 1

BIG-O NOTATION

- Big-O is for upper bounds.

BIG-O NOTATION

- Big-O is for upper bounds.
- Its equivalent for lower bounds is big Omega

BIG-O NOTATION

- Big-O is for upper bounds.
- Its equivalent for lower bounds is big Omega

Formally, a function $f(n)$ is $\Omega(g(n))$ if there exists a c and $n_0 > 0$ such that:

- For all $n \geq n_0$, $f(n) > c*g(n)$

BIG-O NOTATION

- If a function `f(n)` is in `O(g(n))` and `Ω(g(n))`, then g(n) is a tight bound on f(n), we call this big theta.

BIG-O NOTATION

- If a function `f(n)` is in O`(g(n))` and Ω`(g(n))`, then g(n) is a tight bound on f(n), we call this big theta.
- Formally, iff `f(n)` is in O`(g(n))` and Ω`(g(n))`, then `f(n)` is in θ `(g(n))`
- Note that the two will have different c and n_0

BIG O NOTATION

- **What does this help us with?**
  - Sort algorithms into families

BIG O NOTATION

- **What does this help us with?**
  - Sort algorithms into families
    - O(1): constant
    - O(log n): logarithmic
    - O(n) : linear
    - O(n^2): quadratic
    - O(n^k): polynomial
    - O(k^n): exponential

BIG O NOTATION

- **What does this help us with?**
  - The constant multiple c lets us organize similar algorithms together.
  - Remember that $\log_a k$ and $\log_b k$ differ by a constant factor?

BIG O NOTATION

- **What does this help us with?**
  - The constant multiple c lets us organize similar algorithms together.
  - Remember that $\log_a k$ and $\log_b k$ differ by a constant factor?
  - That makes all logs in the same family

NEXT CLASS

- **Recurrence Relations**
  - How to analyze recursively defined functions
- **Analyzing the naïve dictionary implementations**