6.0001 Final Review

Complexity & Algorithms
What is an Algorithm?

- A method of solving a problem
- Lays out the steps to follow to derive the output (solution) from the input (problem)
- Can often be implemented in different languages, or in different ways in the same language
- Exists independently of any implementation
Time Complexity

- An algorithm might be useless if it takes too long to get an answer
- We need a notion to measure how long an algorithm takes
- We would like our notion to be independent of the machine it runs on and the details of the implementation
Big O Notation

- Describes the growth of the runtime of an algorithm as a function of its input size

- Typically describes the worst case runtime (also interesting is the average case)
Big O Notation - Mechanics

Fastest growing term dominates:

\[ n^2 + 100n + 1,000,000 \log(n) = O(n^2) \]

Constant factors don’t affect complexity:

\[ 1,000,000n = O(n) = .0000001n \]
Meaning of Complexity

- Describes how changing the size of a “large” input will affect the runtime
- If the input keeps increasing in size, eventually the algorithm with the lower complexity will be faster
- Makes no guarantee how big the input needs to get to make it faster
Complexities

$O(1)$ - Constant

$O(\log n)$ - Logarithmic

$O(n)$ - Linear

$O(n \log n)$ - Log-Linear

$O(n^k)$ - Polynomial

$O(k^n)$ - Exponential
Complexities: Graph

Big-O Complexity

Operations vs. Elements graph showing different time complexities:
- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(2^n)$
- $O(n!)$
Analyzing Complexity

Total Time = Time per Iteration * # of Iterations

- Beware of calling methods with non-constant complexity

- If the algorithm needs to look at the entire input, its complexity is at least $O(n)$
In constant \( \mathcal{O}(1) \) time on average, dictionaries can:

- Insert a new element
- Delete an element
- Find an element given a key
- Check if a key is in the dictionary
Hashing

- A dict is a list of lists of tuples internally
- Dict entries are represented as (key, value)
- Given a key, a hash function converts it to an index into the list
- The index is used to retrieve an inner list, which is searched for the tuple with the key
- On average, there is a constant number of entries per list
Hashing

1: []
2: [(k1, v1)]
3: []
4: [(k2, v2), (k3, v3)]
... 
n-1: []
n: [(k4, v4)]
Search Algorithms

**Problem**: Given some target value $t$, function $f$, and a range to search, find a value $x$ such that $f(x)$ is close to $t$

**Input Size**: Size of the range in which we search for $x$

- Linear Search (naive algorithm)
- Bisection Search (clever improvement)
Linear Search

- Start at bottom of range
- Check increasing values until one that works is found

Complexity: O(n)
At worst, checks the whole range before finding nothing works, or an answer near the end
Bisection Search

- Start in middle of range
- Check value, then rule out half the remaining range depending on whether value is too high or too low
- If input is a list, requires pre-sorted data

**Complexity: O(log n)**

Each iteration, we cut the remaining input size in half
Sort Algorithms

**Problem**: Given a list $l$, reorder its elements so that each element is no greater than the one after it.

**Input Size**: Length of the list $l$

- Selection Sort
- Merge Sort
Selection Sort

- Find the smallest element still in the unsorted side of the list
- Swap it with the end of the sorted side

Complexity: $O(n^2)$
We do $n$ iterations, each of which requires a scan of a sublist of average size $n/2$
Merge Sort

Key Observation: two small sorted lists can be combined into one big sorted list in $O(n)$ time
- Split list in half recursively until we have single element sublists
- Merge two lists at a time until they are all merged

Complexity: $O(n \log n)$
We do $\log n$ list merges, each with $O(n)$ elements in each list