EECS 660: FUNDAMENTALS OF COMPUTER ALGORITHMS

MODULE II: PRELIMINARIES

DISCLAIMER

ACKNOWLEDGEMENT

- Many of the figures, unless otherwise stated, come from ***Algorithm Design*** 4th edition, by Jon Kleinberg and Eva Tardos.

SUMMARY

- Basic notation conventions
- Time complexity and space complexity analysis
- Recap of data structures
  - array and list
  - hash table
  - queue and stack
  - priority queue
  - disjoint set
  - tree and graph

NOTATION CONVENTION

- While it is free to use any letter or label to represent a variable, everything will be more understandable if we follow the convention when writing.
  - similar to programming, you can use any style in naming your variables
  - but some are more clear

NOTATION CONVENTION

- Before we start, here are some general guide lines for notation convention
  - use notation that is consistent with the meaning
    - we often use U (Universe) to represent all elements in a given domain)
    - we use i (index) to represent an index
  - do not use the same notation for different things
  - do not overcomplicate the definition
  - follow the convention as much as possible

NOTATION CONVENTION

- A set of items (ordered or unordered) are often represented by an uppercase letter
  - say we have a set of different fruits F

NOTATION CONVENTION

- We shall use superscript to distinguish between sets if we have to use the same uppercase letter to represent them (which make things clearer sometimes)
  - we will define a set of red fruits as F^R and a set of green fruits as F^G

NOTATION CONVENTION

- We shall use subscript to represent a specific item in an ordered set
  - we will define a set of fruits ranked by their prices as F
  - the ith fruit would be written as F_i
- If we are working on an underorder set, we will simply use a lowercase letter to represent an item
  - we will define a set of fruits as F
  - we define a single type of fruit $f \in F$

NOTATION CONVENTION

- It is possible to combine the use of superscript and subscripts
  - we will define a set of red fruits sorted by their prices as F^R
  - the red fruit that ranked at the ith place can be represented as F_i^R

NOTATION CONVENTION

- Brackets:
  - the curly brackets are often used to explicitly represent the specific items within a set
    - $F = \{\text{apple}, \text{pear}, \text{grape}, \dots\}$
  - the square brackets are often used to indicate an element within a matrix
    - Let M be an matrix, and $M[i, j]$ is the element that locates at the ith row and jth column
  - the parenthesis are often used to represent the input arguments for a function
    - Let I be an identity function, $I(a, b) = 1$ if $a = b$, and $I(a, b) = 0$ otherwise

NOTATION CONVENTION

- The above rules are general guidelines, and are not always applicable
  - in statistics, the expected value of a random variable x is often expressed as $E[x]$ and the variance of x is often expressed as $Var(x)$
  - it is a convention in the field to use square brackets for expected value, although it is in fact a function just like the variance

- **<u>If there exists a convention for you to follow, just follow it.</u>**

NOTATION CONVENTION

- If you couldn't remember the rules, don't worry
  - we will clearly define the meaning of each notation before proceeding with the algorithm
  - and **<u>you should do that too in the homework and exams</u>**

COMPLEXITY ANALYSIS

- For complexity analysis we have time and space complexity, but the way we measure them are very similar. So, let's just focus on time complexity here.

- Time complexity measures how fast the running time grows with the input size. Basically it makes six fundamental assumptions (or focuses):
  - the measure of input size
  - the relationship between the input size and the running time as a function
  - the ignorance of the constants in the function
  - focusing on the fastest-growing term
  - focusing on asymptotic behavior
  - focusing on the worst-case scenario

COMPLEXITY ANALYSIS

- The measure of the input size:
  - the input size is measured as the amount of space to hold the inputs
  - for example, if we were asked to find the most pricy fruit from a set of fruits F, then we expect the input of each fruit's price. If there are n types of fruits, then the amount of space that is required to hold them is thus cn, assuming holding each price takes a space of c

COMPLEXITY ANALYSIS

- The relationship between the input size and the running time as a function:
  - the amount of time required by the algorithm, t, is measured as a function that depends on the input size n; that is, $t = T(n)$

COMPLEXITY ANALYSIS

- The ignorance of the constants in the function:
  - for each term in the function $T(n)$, its associated constant is ignored
  - for example, let $T(n) = an^2 + bn + c$; we do not care about what the constants a, b, and c are
  - we can simplify the function as $T(n) = n^2 + n + 1$ by making all constants 1
  - this is because the constants depends on specific primitive operation, computer architecture, and CPU speed *etc.* that are hard to measure and consolidate

COMPLEXITY ANALYSIS

- Focusing on the fastest-growing term:
  - for the previous example where $T(n) = n^2 + n + 1$, we only care about the fastest-growing term n^2, and ignores the other slower-growing terms
  - this is because when n gets larger enough, the fastest growing term will dominate $T(n)$
  - in theoretical computer science, there is a very clear distinction between polynomial and exponential time complexity
  - it is often of less interest (at least in this course) in identifying the fastest-growing term among many exponential terms, as any of them will be too slow to be practical

COMPLEXITY ANALYSIS

- Focusing on asymptotic behavior:
  - we are more interested in how fast the algorithm runs when the input gets large, that is, when $n > n_0$ where n_0 is an arbitrary constant
  - this assumption is sufficient to make the constant-free fastest-growing term dominate the total running time
  - you can think in this way: if the algorithm can handle large inputs well, then it can handle small inputs too, but not the vice versa; so, we care more about its behavior when the input gets large.

COMPLEXITY ANALYSIS

- Focusing on the worst-case scenario:
  - we similar to the rationale of focusing on large inputs, we also focus on the worse-case scenario the algorithm may encounter
  - for example, we are asked to find the maximum among a set of random numbers
    - in the ideal case, the set of numbers are sorted in a descending order, so we simply take the first one and it requires a constant time
    - in the worst-case scenario, they could be sorted in an ascending order, so we will have to go through the entire set to find the maximum
  - in some cases we are interested in average-case analysis, which is similar to finding the expected value of a variable in statistics

COMPLEXITY ANALYSIS

- In complexity analysis, the mostly used measure is the upper bound:
  - upper bound, denoted using $O(.)$, and called big-O; substitute "$.$" with the fastest growing term in $T(n)$, for example we can write $O(n^2)$ for $T(n) = an^2 + bn + c$
  - it simply says that, we can assign a constant c_0, and multiply it with the complexity term, and the product will be larger than $T(n)$ no matter now large n will be
  - that is $c_0 * (.) \geq T(n)$ (note that $. = f(n)$)
  - for $T(n) = an^2 + bn + c$, we know that any $c_0 \geq a + b + c$ is going to satisfy the definition when we set its time complexity to be $O(n^2)$
  - we can also set the upper bound to be any higher polynomial time, say $O(n^3)$
  - but not any lower, for example, we know that no matter how large c_0 would be $c_0 * (n) < T(n)$ when $n > \frac{c_0}{a}$

COMPLEXITY ANALYSIS

- Upper bound continued:
  - using upper bound, we will have a guarantee of the running time
  - it says "we should be able to finish the job within that amount of time, no matter what will happen"

COMPLEXITY ANALYSIS

- The second type of bound is called lower bound:
  - we use $\Omega(.)$ to denote that, and call it the big-omega notation
  - it is defined similarly as the upper bound, but in a reverse way that for any constant $c_o{}', T(n) \geq c_o{}'$*(.) when n gets large enough
  - it says "no one can finish the job sooner than indicated, no matter what will happen"
  - it is often viewed as a termination of inventing a theoretically more efficient algorithm for the problem
  - of course, we can also plug in any lower-than-necessary term to fulfill the definition of lower bound

COMPLEXITY ANALYSIS

- As discussed previously, we can use arbitrarily fast-growing function to satisfy the upper bound definition, and arbitrarily slow-growing function to satisfy the lower bound
  - to address the issue, tight bound was introduced
  - if $T(n)$ is in both $O(.)$ and $\Omega(.)$, with the same function “$.$”, we say “$.$” is a tight bound for $T(n)$ and use $\Theta(.)$ to indicate it and call it the big-theta notation
  - tight bound is the best way to describe the time complexity of an algorithm, if one can establish

COMPLEXITY ANALYSIS

- The pessimistic nature of computer scientists:
  - we have encountered three pessimistic assumptions in the definitions and analysis of time complexity
    - large inputs
    - worst-case scenario
    - upper bound for running time
  - but we need to distinguish their meanings (e.g., the find maximum problem)
    - large inputs: we are asked to work on a huge amount of numbers
    - worst-case scenario: if the maximum number happens to locate at the last position
    - upper bound for running time: finding the maximum will not take longer than the indicated time

COMPLEXITY ANALYSIS

- The most straightforward way to perform time complexity analysis is to solve for the function $T(n)$.

- But in most cases we do not have to do that:
  - we can analyze the algorithm to find the number of iterations or recursions
  - and multiply them with the amount of work we need to do in each iteration or recursion
  - solving recursion could be a bit more challenging, but we will discuss that in detail later in the class

COMPLEXITY ANALYSIS

- Note: **the analysis of space complexity is similar to the analysis of time complexity**.
  - except that we are working on the amount of space rather than the amount of time
  - we don’t care constants either (e.g., don’t care 32-bit for integer or 8-bit for character)
  - some upper bound, lower bound, and tight bound defined

DATA STRUCTURES

- Array and (linked) list
  - linear representation of a set of elements (both sorted and unsorted)
  - because their different implementation, they may differ in the time complexity for some operations:
    - find the kth element: $O(1)$ for array, $O(n)$ for linked list
    - insert/delete a specific element: $O(n)$ for array, $O(1)$ for linked list
    - merging two sets: $O(n)$ for array, $O(1)$ for linked list
    - enumeration of all elements: $O(n)$ for both array and linked list

DATA STRUCTURES

- Hash table:
  - a set of key-value pairs
  - recall the basic concepts of hash function, load factor, and table doubling/halving
  - time complexity for the associated operations:
    - find a specific key: $O(1)$
    - insert/delete a specific key: $O(1)$
    - merge two hash tables: $O(n)$
    - enumerate all elements: $O(n)$
  - note that hash table operations are often associated with large constants, so it is not often involved in theoretical algorithm development unless one can come up with a simple hash function with guaranteed bound (it is very frequently used in practice though)

DATA STRUCTURES

- Queue and stack:
  - queue and stack are sets with different insert/delete priorities
  - queue: first in first out
  - stack: first in last out
  - time complexity
    - find a specific element: prohibited
    - insert/delete: $O(1)$ (note that you cannot choose which element to delete, it has to comply with the defined behavior)
    - merge two queues/stacks: prohibited, or $O(1)$ or $O(n)$ depending on implementation
    - enumerate all elements: $O(n)$

DATA STRUCTURES

- Priority queue
  - allows ordering of elements according user-defined priority, rather then simply by their order of insertion
  - the element with the highest/lowest priority will be taken out the queue first
  - time complexity
    - find a specific element: prohibited
    - insert/delete: $O(\log n)$
    - merge two queues/stacks: $O(n)$ (better to reconstruct a new priority queue with both sets of elements)
    - enumerate all elements: $O(n)$

DATA STRUCTURE

- Disjoint set:
  - also known as the union-find data structure
  - time complexity
    - find the label of a specific element: $O(1)$
    - merge two disjoint sets: $O(\log n)$

DATA STRUCTURE

- Tree and graph:
  - non-linear data structures that are more powerful, especially in representing sophisticated relationships
  - tree is a graph without cycle
  - basic concepts:
    - nodes (or vertices)
    - edges
    - weight
    - degree
    - children and parent (as in tree); source and target (as in graph)
    - directed and undirected graphs
    - path and cycle

DATA STRUCTURE

- Tree and graph:
  - implementation (assuming $|V|$ nodes and an average degree of d)
    - matrix: $O(1)$ to determine whether a edge exists given two nodes, $O(|V|)$ to enumerate all neighbors, $O(|V|^2)$ storage
    - list: $O(d)$ to determine whether a edge exists given two nodes, $O(d)$ to enumerate all neighbors, $O(|V| + |E|)$ storage
  - fundamental algorithms in graph:
    - BFS and DFS: $O(|V| + |E|)$
    - Finding connected components: $O(|V| + |E|)$

SUMMARY

- Basic notation conventions
- Time complexity and space complexity analysis
- Recap of data structures
  - array and list
  - hash table
  - queue and stack
  - priority queue
  - disjoint set
  - tree and graph