Asymptotics and Disjoint Sets

Abhinav Ashar
CS 61B: Data Structures and Algorithms

March 13, 2019

1 Asymptotics

Asymptotics describes the runtime of an algorithm and how it changes as the size of the input approaches to infinity. For example, if the input was an array, how does the runtime get affected if the array is twice as long? Does the runtime stay the same? Does it double? Does it quadruple? By analyzing our algorithms to determine runtime, we can understand how efficient the algorithm is. There are 3 different ways we can describe the runtime efficiency of our algorithm: Big-O, Big-Ω, and Big-Θ.

Big-O

a) Gives an upper bound on the runtime

b) Consider the worst case scenario

Big-Ω

a) Gives a lower bound on the runtime

b) Consider the best case scenario. Best case does not mean you make the size of your input very small. It means that despite have very large input (approaching infinity), there exists a condition (maybe an if statement) that causes the function to complete faster than it normally does.

Big-Θ

a) Gives a tight bound on the runtime

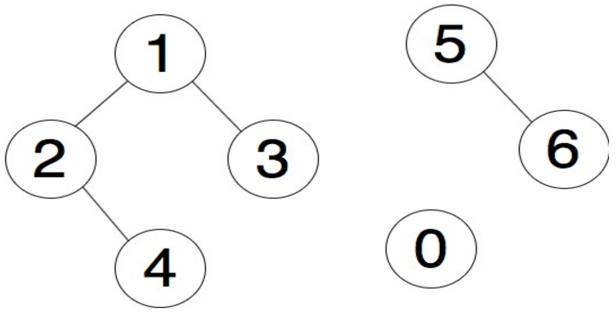
b) If Big-O and Big-Ω have the same runtime, then you can describe the method's runtime in Big-Θ. Otherwise, describe it in Big-O and Big-Ω.

There is also a general ordering of runtimes that you should remember (this ordering is true for Θ and Ω as well):
$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(c^n) < O(n!)$ as $n \to \infty$

2 Disjoint Sets

Disjoint sets are a powerful way for programmers to create groupings amongst different pieces of data. They help us efficiently add to a group/set and determine if two elements are part of the same group.

Let this be the disjoint set used for reference:



Quick Find

a) *connect:* $\Theta(N)$ and *isConnected:* $\Theta(1)$

b) Useful for helping us determine whether two items are in the same set

c) Maintains an array that stores the value of the root/boss node of the set in its index

d) The array for the disjoint set above is $[0, 1, 1, 1, 1, 5, 5]$.

e) Not very efficient for connecting

f) Further explanation on Quick Find: Josh Hug's Slides

Quick Union

a) *connect:* $\Theta(N)$ and *isConnected:* $\Theta(N)$

b) Sets up the design for Weighted Quick Union, but Quick Union by itself is not very useful

c) Maintains an array that stores the value of the parent node of in its index. If parent doesn't exist (root node of the set), put a -1 in its index.

d) The array for the disjoint set above is $[-1, -1, 1, 1, 2, -1, 5]$.

e) Trees can get too tall, which makes it very inefficient

f) Further explanation on Quick Union: Josh Hug's Slides

Weighted Quick Union

a) *connect:* $\Theta(\log N)$ and *isConnected:* $\Theta(\log N)$

b) Uses the design of Quick Union, but also tracks tree size (this is what makes it a "weighted" quick union)

c) How to connect elements in WQU: look at set A and B and determine which one is bigger (has more elements). Make the root of the smaller set become the child of the root of the larger set. For example, $connect(1, 6)$ would compare the size of the two sets (4 vs. 2), makes 5 as a child of 1, and changes the array accordingly. The resulting array would be $[-1, -1, 1, 1, 2, 1, 5]$.

d) No matter how large the trees are, connecting two trees A and B will only produce a resulting tree whose height is 1 greater than the height of the "heavier" tree between A and B. **TLDR: *connect* always only increases the height of the tree by 1 in WQU**.

e) Guarantees that a spindly/tall tree cannot be formed

f) Further explanation on Weighted Quick Union: Josh Hug's Slides

Path Compression

a) Technique that is used to optimize the *isConnected* method for future uses

b) When initially checking whether two nodes are connected, determine the path between the two nodes and begin traversing the path. For each node seen along the path, connect this node to the root of the tree to make *isConnected* much more efficient later on.

c) Further explanation on Path Compression: Josh Hug's Slides