Hudson Arney

CS3851 Algorithms

§ Problem Set 6 §

Test whether the input satisfies some special-case condition and if it does,

Problem 1:

How can we modify almost any algorithm to have a good best-case running time?

Modifying an algorithm to have a good best-case running time may require figuring out the most favorable input patterns. The algorithm can then be adapted to fit these patterns. Pre-processing the data can also be an important step to optimize algorithms. Early termination and using efficient data structures can also be employed to optimize the best-case performance. The most important step however, would be knowing when to use the correct algorithm in different cases.

Some examples can include using heap sort when working with large datasets where limiting memory allocation is a priority. Since heap sort is an in-place sorting algorithm, it quite memory-efficient.
Merge sort can be a better choice in situations where a consistent time complexity is a primary concern. Merge sort consistently offers a time complexity of O(n log n) regardless of the input data, providing predictability in performance.

Problem 2:

Describe a $\Theta(n \log n)$ time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Which value cannot be a solution?

Merge Sort is an Θ(n logn) algorithm that could solve this problem. You would first have to sort the set which is the most time-consuming part of the algorithm (Θ(n logn)).

After sorting, for each element s in S, perform a binary search in the sorted array for the element x - s. If the search is successful, it means there exists an element in S that, when added to s, results in the desired sum x. Repeat this process for all elements in S. This would have a time complexity of Θ(logn). In the worst case you have to do this n times (you only have to find one num)
Combined these steps converge to Θ(n logn).
The value that cannot be a solution would be the value x/2. This value cannot be part of a pair that sums to x, as it is either too small or too large to be combined with any other value to result in x. This assumes there are no duplicates within the set S.
Brute force would be to add every combination and see if it equals X. This would be O(n^2)

Problem 3:

Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

This undesirable outcome occurs because quicksort's choice of pivot element from having the decreasing order sorted input data, leads to unbalanced partitions during the partitioning step. When the pivot is selected as the smallest element (e.g., the first or last element in the array), all other elements end up in one partition, while the other remains empty. Consequently, quicksort requires Θ(n) recursive calls to process the array, resulting in a quadratic time complexity of Θ(n^2).

T(n) = T(left)+T(right) ==> Cant be in normal master method form because you cant predict the partitions

Problem 4:

Suppose that the splits at every level of quicksort are in the proportion $1-\alpha$ to α, where $0 < \alpha \leq \frac{1}{2}$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\frac{\log n}{\log \alpha}$ and the maximum depth is approximately $-\frac{\log n}{\log (1-\alpha)}$. (Don't worry about integer round-off.)

Minimum Depth: The minimum depth will be reached when we have subarrays of size 1 (base case), and this occurs when the array is divided into (1−α)^k * n = 1, where k represents the number of levels.
Solving for k:
(1−α)^k * n = 1
k = log(1/n) / log(1−α) = -log(n) / log(1−α)

Maximum Depth: The maximum depth occurs when we have subarrays of size n/2 (i.e., roughly half the size of the original array). To find the maximum depth, we need to find the number of recursive divisions required to reach this subarray size. When the array is divided into (1−α)^k * n = n/2.
Solving for k:
(1−α)^k * n = n/2
k = log(1/2) / log(1−α) = -log(2) / log(1−α).

Problem 5:

Given an adjacency-list representation, how long does it take to compute the out-degree of every vertex?

Steps for computing the out-degree of every vertex:
1. Iterate through each vertex in the graph (V iterations).
2. For each vertex, access its adjacency list (constant time or O(1) for each vertex).
3. Count the number of neighbors in the adjacency list (O(out-degree) for each vertex).

The total time complexity for computing the out-degrees of all vertices is O(V) for the iterations, and the cost for accessing the adjacency lists and counting neighbors is typically O(E), as you may have to visit every edge once in the worst case.

In total = O(V + E)

Problem 6:

Given an adjacency-list representation, how long does it take to compute the in-degree of every vertex?

To compute all in-degrees for every vertex we have to scan through all adjacency lists and keep counters for how many times each vertex has been pointed to. Thus, the time complexity is also O(|V|+|E|) because we'll visit all nodes and edges.

Problem 7:

The transpose of a graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all edges reversed.

(1) Describe an efficient algorithm for computing G^T from G, both for the adjacency-list and adjacency-matrix representations of G.

Adjacency-Matrix:

```
num_vertices = len(adj_matrix)
transposed_matrix = [[0] * num_vertices for x in range(num_vertices)]

for i in range(num_vertices):
    for j in range(num_vertices):
        transposed_matrix[j][i] = adj_matrix[i][j]
```

Adjacency-List:

```
num_vertices = len(adj_list)
reversed_adj_list = [[] for x in range(num_vertices)]

for u in range(num_vertices):
    for v in adj_list[u]:
        reversed_adj_list[v].append(u)
```

O(V) O(E)

(2) Analyze the running time of each algorithm.

The time complexity for both of these algorithms is O(V^2) which is determined by the double loop that iterates over the elements of the adjacency matrix/list.

Problem 8:

When an adjacency-matrix representation is used, most graph algorithms require time $\Omega(V^2)$, but there are some exceptions. A universal sink is a a vertex of in-degree $|V| - 1$ and out-degree 0.

(1) How many sinks could a graph have?

There can be at most one universal sink in a graph because if there were more than one universal sink, each of them would need to have in-degree |V| - 1, and there wouldn't be enough vertices left to have edges going into these sinks.

(2) How can we determine whether a given vertex u is a universal sink?

1. Calculate the number of incoming edges (in-degree) for vertex u by summing up the values in the column of the adjacency matrix corresponding to vertex u. Make sure this is |V| - 1.
2. Check the out-degree of vertex u: Calculate the number of outgoing edges (out-degree) for vertex u by summing up the values in the row of the adjacency matrix corresponding to vertex u. Make sure this is 0.

If both of these pass we can determine u to be a universal sink

(3) How long would it take to determine whether a given vertex u is a universal sink?

Calculating the in-degree takes O(V) time.
Calculating the out-degree also takes O(V) time.
The comparison and boolean operations have constant time complexity.

Calculating the overall time complexity for finding universal sink is O(V)

(4) How long would it take to determine whether a given graph contains a universal sink if you were to check every single vertex in the graph?

In the worst case, you would need to check every vertex. To check each vertex, you would use the code provided earlier, which has a time complexity of O(V) for each vertex. You would repeat this process for all V vertices in the graph.

This would result in a time complexity of O(V^2)

(5) Show how to design an algorithm that determines whether a directed graph G contains a universal sink in time $O(V)$.

```
num_vertices = len(adj_matrix)

for vertex in range(1, num_vertices):
        if adj_matrix[potential_sink][vertex] == 1:
            potential_sink = vertex

for vertex in range(num_vertices):
    if vertex != potential_sink and adj_matrix[vertex][potential_sink] != 1:
        return -1
```

Here we go through each vertex in the graph and check if it has any outgoing vertices (which it shouldn't) and if all the other vertices point to it. This has two comparisons of all vertices which should overall be a time complexity of O(V)