HW2

Seyed Armin Vakil Ghahani
PSU ID: 914017982
CSE-565 Fall 2018
Collaboration with: Sara Mahdizadeh Shahri, Soheil Khadirsharbiyani, Muhammad Talha Imran

September 5, 2018

Problem 1. Problem 6, Chapter 2 on page 68 of the Textbook. (Efficient computations of sums)

Solution

- a) Both of for loops iterate n times, at most. Moreover, the add operation is the sum of at most n numbers. Hence, its time complexity is $O(n)$. As a result, the time complexity of the algorithm is $O(n * n * n) = O(n^3)$ and $f = n^3$.
- b) Suppose the first iteration of i loop ($i = 1$). In this iteration, j iterates from 2 through n. If j equals 2, the count of numbers that should sum is 1. If j equals 3, the count of numbers that should sum is 2. ... If j equals n, the count of numbers that should sum is $n - 1$. Hence, the number of computations is the sum of 1 to $n - 1$ which equals to $n(n - 1)/2$.

  At the second iteration of i loop ($i = 2$), if j equals 3, the count of numbers that should sum is 1, and so on. As a result, the number of computations is the sum of 1 to $n - 2$ which equals to $(n - 1)(n - 2)/2$.

  The same happens for the rest and for the last iteration of the i loop ($i = n - 1$), j can be just n and there is just one sum computation which equals to $2 * 1/2$.

  Thus, the number of sum computations in this algorithm is

$$\sum_{i=1}^{n-1} \frac{i(i+1)}{2} = \frac{1}{2}(\sum_{i=1}^{n-1} i^2 + i) =$$

$$\frac{1}{2}(\sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i) = \frac{1}{2}(\frac{(n-1)n(2n-1)}{6} + \frac{n(n-1)}{2}) =$$

$$\frac{n^3 - n}{6}.$$

  It means that the algorithm time complexity is $\Omega(n^3)$, which results that the algorithm's time complexity is $\Theta(n^3)$.
- c) We can change the algorithm which is given by the question in such a way that instead of adding the corresponding entries always, use the previous computed values like dynamic programming. In every iteration of this algorithm, instead of add all the entries, add $A[j]$ with $B[i, j - 1]$ that produces the same result, however, it is $O(1)$. With this change, the time complexity of the algorithm changes to $O(n^2)$.

Problem 2. Solve Problem 8, Chapter 2 on page 69 of the Textbook. (Combining two strategies)

Solution

- (a) At first, I want to mention that whenever there is only one jar, it should going step by step to figure out the highest safe rung.

  For simplicity, suppose that n is a square number. The algorithm start from $\sqrt{n}$ and the jar brokes at this rung, it should going from the first rung to $\sqrt{n}-1$ rung step by step and the number of drops would be $\sqrt{n}$. However, if the jar does not broke at $\sqrt{n}$, it should try the $2\sqrt{n}$ rung and if it brokes, the same procedure should repeat and the number of drops would be $\sqrt{n}+1$. This procedure will continue and it the jar brokes at the last rung n, the number of drops till now is $\sqrt{n}$ and it should check jars from $n-\sqrt{n}+1$ through $n-1$ step by step, which results $2\sqrt{n}$ drops. As a result, $f(n) \in O(\sqrt{n})$.

- (b) I am going to prove that $f_k(n) = \sqrt[k]{n}$ by induction.

  Initial Step. The initial step is for k = 2 that is proven in part (a).

  Inductive Step. Our inductive assumption is: For each k, greater than or equal to two and less than k, $f_k \in O(\sqrt[k]{n})$. We must prove the formula is true for $n = k+1$. We drop the jar at the multiplys of $\sqrt[k]{n}$ until the jar brokes. Whenever the jar brokes, we realize that the question is with the following parameters:

  $$n \to \sqrt[k]{n}, k \to k-1$$

  It is known by the induction assumption that this problem could be solved by $f_{k-1}(\sqrt[k]{n})$ drops which is $\sqrt[k(k-1)]{n}$. As a result $f_k(n) = \sqrt[k]{n} + \sqrt[k(k-1)]{n}$. Because $O(\sqrt[k(k-1)]{n}) \in O(\sqrt[k]{n})$, then it is figured out that $f_k(n) \in O(\sqrt[k]{n})$.

  The limit equation is also true for the $f_k(n)$ because $O(\sqrt[k]{n}) \in O(\sqrt[k-1]{n})$.

Problem 3. Solve Problem 3, Chapter 3 on page 107 of the Textbook. (DAG or cycle)

Solution If we run the same algorithm in the textbook for the arbitrary graph, it may give us a topological order which means that the graph is DAG. On the other hand, if the algorithm finishes without the topological order and some vertexes remain still active, we have a non-zero graph which all of its vertexes has an incoming edge. Thus, realize a vertex v and start from it and go to its incoming vertex until the algorithm reaches a repeating vertex. In this situation, we have a cycle in G because the algorithm always see a new vertex except the last vertex and the path from the first appearance of this vertex until the last visit is a cycle in G.

Problem 4. (Types of edges in DFS)

Solution

- (a)
  - **Tree edge.** $A[v], B[v] = \infty$
  - **Forward edge.** $A[v], B[v] \neq \infty, A[u] < A[v]$
  - **Back edge.** $A[u] > A[v], A[v] \neq \infty, B[v] = \infty$
  - **Cross edge.** $B[v] < A[u]$

- (b) Whenever a back edge exists in the DFS algorithm, it means that there is a directed cycle in the graph. If there is not any back edge in the graph, and we look at the DFS tree, the tree edges and forward edges are all from up to down and they can not create a directed cycle. Moreover, the cross edges can not create a directed cycle like the following graph.