A2 ICCS 312 Algorithms and Tractability

ALEXANDER OGAY 6380727

April 28, 2024

1 Problem 1

(a) $f(k)$ is the largest integer i such that 2^i divides k.

When k is a power of 2, i reaches a larger integer. We pay $i = log_2 k$, we can see that 2^1 operations cost $i-2, ..., 2^{i-1}$ operations that cost $(i-i)$. Therefore we place $\sum_{m=1}^{log_2 k} 2^{m-1} * (i-m)$ into our account, so there will be $log_2 k$ elements in the sum. What we place into the account when k is a power of 2 will pay for the k-1 operations that follow. Amortized cost per operation is $log_2 k + \sum_{m=1}^{log_2 k} 2^{m-1} * (i-m)$

(b) $f(k)$ is the largest power of 2 that divides k.

When k is a power of 2, we pay $f(k) = k$, there will be also be operations that cost $\frac{k}{2log_2 k}$, when k is a power of 2, we pay k and place $\sum_{m=1}^{log_2 k} 2^{m-1} * \frac{k}{2^m}$ into the account. There will be $log_2 k$ elements in the sum. Hence the amortized cost per operation is $log_2 k + \sum_{m=1}^{log_2 k} 2^{m-1} * \frac{k}{2^m}$

(c) $f(k) = n^2$ if k is a power of 2, and $f(k) = 1$ otherwise.

When k is a power of 2, we pay n^2 and place $k-1$ into our account, there will be $i-1$ operations that cost 1 after the k-th operation. Therefore, what is placed into the account at the k-th operation pays for the $k-1$ operations tha follow. The amortized cost per operation is $n^2 + k - 1$ which is $\mathcal{O}(n^2)$

(d) $f(k) = k$ if k is a Fibonacci number, and $f(k) = 1$ otherwise.

When k is a Fibonacci number, we pay k and place $k-1$ into our account because there will be $k-1$ operations that cost 1 after the next Fibonacci number. After this k-th operation, there will be a number of operations that cost 1 which will be paid for by the money we placed into the account. We place into our account at this k-th

operation will pay for the $k-1$ operations that follow after the next Fibonacci number. Amortized cost per operation is $k+k-1=2k$ which is $\mathcal{O}(n)$

(e) $f(k)$ is the largest integer whose square divides k.

We can see a pattern: every number divides 1, every 4th number divides 4. We can find the upper bound sum for the sum of $f(1)$ to $f(n)$ $\sum_{m=0}^{\sqrt{n}} \frac{n}{m} * m$. The amortized cost per operation will be slightly lower than $\frac{\sum_{m=0}^{\sqrt{n}} \frac{n}{m} * m}{n}$

2 Problem 2

OrderedPush, The actual cost $c_i = \mathcal{O}(n)$ since in the worst case the entire stack is removed, which is $\mathcal{O}(n)$, and if it's called again the stack is empty so the operation becomes $O(1)$. The change in potential is: $\phi(H') - \phi(H) = \mathcal{O}(n)$ since it can remove the entire stack in one operation. Therefore the amortized cost of OrderedPush() is: $c_i + \phi(H') - \phi(H) = \mathcal{O}(n) - \mathcal{O}(n) = 0$

Pop, The actual cost $c_i = 1$ since it removes an item from the stack. The change in potential is: $\phi(H') - \phi(H) = 1$ we lose an element from the stack between the two states. The amortized cost of Pop() is: $c_i + \phi(H')\phi(H) = 1 - 1 = 0$.

3 Problem 3

An undirected graph is a tree, we need to travel the tree and check if all vertices are reachable, and check if no adjacent vertex of the current vertex we are on, was already visited, except if that adjacent vertex is its parent. To travel the graph we use the Breadth-First Search algorithm which starts at a node and iteratively travels through its adjacent vertices not marked as visited, and then marking them as visited. The running time of a BFS algorithm is O(V+E) since the algorithm will iterate over every vertex and edge once. Checking for a cycle only requires an if statement so it's O(1), and checking for connectivity as well. Therefore the total running time of the algorithm is O(V+E).

4 Problem 4

A DAG is a directed acyclic graph. For a graph to be a DAG there must be a node with no incoming edges and another node with no

outgoing edges. Suppose every vertex in a graph had at least one incoming edge. That would mean that for each vertex we would be able to move back to another vertex. If there are n vertices and from a vertex we move backwards n times, we must at some point have reach a vertex we had already visited. If every vertex in a graph had at least one outgoing edge, then we would be able to move back n times and we would eventually visit a vertex we have already visited. That is why a DAG graph must have at least one vertex with no incoming edges and one vertex with no outgoing edges.

5 Problem 5

We will make a recursive algorithm. This algorithm will have 2 base cases: if the current vertex has no adjacent vertices, and if the current node is node t. In the first base case we return 0. In the second base case we return 1. The inductive step will start by initializing a variable called count which will count the number of paths that will reach t, and then will follow a for loop iterating through the adjacent nodes of the current node s and will be calling the function for each adjacent vertex with the input s being the adjacent vertex of that iteration. We will add the return value of each call to the count variable. The count variable will therefore represent the number of paths that reach vertex t from the current node s. The function returns the count variable.

6 Problem 6

Proof by contradiction. We have a graph G that is not connected and has two components. Now let's pick a vertex s from one component and a vertex t from the other. There must be no common adjacent vertices between s and t which we can describe as $A(s) \cap A(t) = 0$. We can also say that $A(s) \cup A(t) \leq n - 2$, since both vertices s and t have, in this case, at most $\frac{n}{2}$ adjacent vertices, and without counting each other. we can use the formula of $A(s) \cap A(t)$ to check if it's truly equal to 0, $A(s) \cap A(t) = A(s) + A(t) + A(s) \cup A(t) \geq \frac{n}{2} + \frac{n}{2}n + 2 = 2$. We can see that $A(s) \cap A(t) = 2$ which contradicts with the statement we made previously stating that $A(s) \cap A(t) = 0$ since vertices s and t are in different, non-connected components and should therefore have no adjacent vertices. We can conclude from our proof by contradiction that the claim is true.