Solution to Practice Exam

Problem 1. (15 points) Rank the following functions by order of growth; that is, find an arrangement \( f_1, f_2, ..., f_n \) of the functions satisfying \( f_1 = \Omega(f_2), f_2 = \Omega(f_3), ..., f_{n-1} = \Omega(f_n) \).

\[
\begin{align*}
n! & \quad 4^{\log n} & \quad n \cdot 2^n & \quad \sqrt{n} \\
e^n & \quad 3^n & \quad 2^n & \quad n^2 \\
\log^2 n & \quad n^3 & \quad 2^{\log n} & \quad n \log n \\
\log \log n & \quad \lg n & \quad n^{1/\lg n} & \quad 2
\end{align*}
\]

Solution:

\[
\begin{align*}
n! &= \Omega(3^n), \\
3^n &= \Omega(e^n), \\
e^n &= \Omega(n \cdot 2^n), \\
n \cdot 2^n &= \Omega(2^n), \\
2^n &= \Omega(n^3), \\
n^3 &= \Omega(n^2), \\
4^{\log n} &= n^2 = \Omega(n \log n), \\
2^{\log n} &= n = \Omega(\sqrt{n}), \\
\sqrt{n} &= \Omega(\log^2 n), \\
\log^2 n &= \Omega(\log n), \\
\lg n &= \Omega(\log \log n), \\
\log \log n &= \Omega(2), \\
n^{1/\lg n} &= 2.
\end{align*}
\]

Problem 2. (20 points) The integer square root problem is to determine the integer portion \( p \) of the square root of integer \( n \); that is, find \( p = \lfloor \sqrt{n} \rfloor \).

Solution:

(a) Give a linear-time algorithm to solve the integer square root problem. Prove your algorithm correct using a loop invariant.

\begin{verbatim}
INTEGER-SQRT-LINEAR(n)
1  assert n ≥ 0
2  p = 1
3  while p * p ≤ n
4      p = p + 1
5  return p - 1
\end{verbatim}
We prove the algorithm correct using the following loop invariant:

At the start of each iteration of the **while** loop on lines 6-7, \( p \leq \sqrt{n} + 1 \).

**Initialization:** Before the first iteration, \( n \geq 0 \) and \( p = 1 \), so the invariant holds.

**Maintenance:** Upon entering the loop body, \( p \leq \sqrt{n} \) due to the loop entry condition and \( \sqrt{n} \leq \sqrt{n} + 1 \), so the invariant holds. Before leaving the loop body, \( p \) is increased to \( p + 1 \). Adding 1 to both sides of the loop entry condition gives \( p + 1 \leq \sqrt{n} + 1 \), so the invariant holds again.

**Termination:** Upon termination, we must have \( p > \sqrt{n} \) due to the loop entry condition. Combining with \( p \leq \sqrt{n} + 1 \) from maintenance gives:

\[
\sqrt{n} < p \leq \sqrt{n} + 1, \\
\sqrt{n} - 1 < p - 1 \leq \sqrt{n}, \\
\sqrt{n} - 1 < \lfloor p \rfloor = \lfloor \sqrt{n} \rfloor.
\]

Therefore, the algorithm correctly returns \( \lfloor \sqrt{n} \rfloor \).

The algorithm runs in \( O(p) \) or, equivalently, \( O(\sqrt{n}) \) time.

(b) Give a logarithmic-time algorithm to solve the integer square root problem. Prove your algorithm correct using a loop invariant.

**INTEGER-SQRT-LOGARITHMIC**

```plaintext
1  assert n \geq 0
2  lo = 0
3  hi = n + 1
4  while lo < hi
5      mid = \lfloor (lo + hi)/2 \rfloor
6      square = mid * mid
7      if square == n
8          return mid
9      elseif square < n
10         lo = mid + 1
11      elseif square > n
12         hi = mid
13      return lo - 1
```

Note that we are less worried about overflow on \( lo + hi \) as we already have a more dangerous \( mid * mid \).

Note that a complete proof would require showing that the loop terminates. Informally, the half-open range \([lo, hi)\) the loop iterates over is reduced by at least one on each iteration, so eventually \( lo \geq hi \). See problem 3 in homework 2 for a more detailed way to show termination.

We prove the algorithm correct using the following loop invariant:
At the start of each iteration of the while loop on lines 4-12, $lo - 1 < \sqrt{n} \leq hi$.

Initialization: Before the first iteration, $n \geq 0$, $lo = 0$, $hi = n + 1$, so $-1 < \sqrt{n} \leq n + 1$, and the invariant holds.

Maintenance: If $mid = \sqrt{n}$, the algorithm correctly returns $mid$ at line 8, the loop terminates, and we don’t need to show maintenance.

If $mid < \sqrt{n}$, line 10 sets $lo = mid + 1$, $hi$ remains unchanged, and the invariant reads $mid < \sqrt{n} \leq hi$, which holds by the entry condition $mid < \sqrt{n}$ and unchanged $hi$.

If $mid > \sqrt{n}$, line 12 sets $hi = mid$, $lo$ remains unchanged, and the invariant reads $lo - 1 < \sqrt{n} \leq mid$, which holds by the entry condition $mid > \sqrt{n}$ and unchanged $lo$.

Termination: Upon termination, we must have $lo \geq hi$ due to the loop entry condition. Combining with $lo - 1 < \sqrt{n} \leq hi$ from maintenance gives:

\[
lo - 1 < \sqrt{n} \leq hi \leq lo,
\]
\[
lo - 1 < \sqrt{n} \leq lo,
\]
\[
lo = \left\lfloor \sqrt{n} \right\rfloor.
\]

$n$ is not a perfect square, as otherwise the loop would have terminated with the return at line 8. Therefore, $\sqrt{n}$ is not an integer and $lo - 1 = \left\lfloor \sqrt{n} \right\rfloor$.

The algorithm runs in $O(\lg n)$ time.

Problem 3. (10 points) Suppose that we have a hash table with $m$ slots.

Solution:

(a) Describe two ways to resolve collisions.

Two common ways to resolve collisions are chaining and open addressing. In chaining, all elements hashing to the same table slot are put into a linked list. In open addressing, the table slots are repeatedly examined until a free slot (if inserting) or the desired element (if searching) is found.

(b) If collisions are resolved by chaining and $n$ keys are inserted into the table, assuming simple uniform hashing, what is the expected number of collisions?

The probability of two keys hashing to the same location is $\frac{1}{m}$, and there are $\binom{n}{2}$ ways to pick two keys, leading to the expected number of collisions $N_c = \frac{1}{m} \binom{n}{2}$.

(c) Under the same assumptions, what is the probability that exactly $k$ keys hash to a particular slot?

The probability of $k$ keys hashing to the same location is $\left( \frac{1}{m} \right)^k$, the probability of remaining $n-k$ keys hashing somewhere else is $\left( 1 - \frac{1}{m} \right)^{n-k}$, and there are $\binom{n}{k}$ ways to pick $k$ keys out of $n$, making the probability that exactly $k$ keys hash to a particular slot $P_k = \left( \frac{1}{m} \right)^k \left( 1 - \frac{1}{m} \right)^{n-k} \binom{n}{k}$. 

3
Problem 4. (25 points) Give an algorithm to determine whether a given node is a root of a valid binary search tree. Analyze the running time of your algorithm.

Solution: One possible approach is to recurse on both subtrees, checking the values encountered for being in the allowed range \([\text{min}, \text{max}]\) in the algorithm below.

\[
\begin{align*}
\text{IS-BST}(\text{node}, \text{min}, \text{max}) \\
1 & \text{ if } \text{node} = \text{NIL} \\
2 & \quad \text{return TRUE} \\
3 & \text{ if } \text{node.key} < \text{min} \text{ or } \text{node.key} \geq \text{max} \\
4 & \quad \text{return FALSE} \\
5 & \text{return IS-BST(\text{node.left}, \text{min}, \text{node.key}) \text{ and IS-BST(\text{node.right}, \text{node.key}, \text{max})}}
\end{align*}
\]

The algorithm assumes the initial call is made as \(\text{IS-BST(\text{node}, -\infty, \infty)}\).

The algorithm visits each node of the tree at most once, so the running time is \(O(n)\).

Problem 5. (15 points) The transpose of a directed graph \(G = (V, E)\) is the graph \(G^T = (V, E^T)\), where \(E^T = \{(u, v) \in V \times V : (v, u) \in E\}\). Thus, \(G^T\) is \(G\) with all its edges reversed. Give efficient algorithms for computing \(G^T\) from \(G\), for:

Solution:

(a) Adjacency-list representation of \(G\).

\[
\begin{align*}
\text{TRANSPOSE-LIST}(G) \\
1 & \text{ for each } u \in G.\text{adj} \\
2 & \quad G^T.\text{adj} = \emptyset \\
3 & \text{ for each } u \in G.\text{adj} \\
4 & \quad \text{ for each } v \in G.\text{adj}[u] \\
5 & \quad \quad \text{INSERT}(G^T.\text{adj}[v], u) \\
6 & \quad \text{return } G^T
\end{align*}
\]

The algorithm runs in \(O(V + E)\) time.

(b) Adjacency-matrix representation of \(G\).

\[
\begin{align*}
\text{TRANSPOSE-MATRIX}(G) \\
1 & G^T.\text{adj} = \emptyset \\
2 & \text{ for } i = 1 \text{ to } |G.V| \\
3 & \quad \text{ for } j = 1 \text{ to } |G.V| \\
4 & \quad \quad G^T.\text{adj}[j, i] = G.\text{adj}[i, j] \\
5 & \text{return } G^T
\end{align*}
\]

The algorithm runs in \(O(V^2)\) time.

Problem 6. (25 points) The knapsack problem is the following. A thief robbing a store finds \(n\) items. The \(i\)-th item is worth \(v_i\) dollars and weighs \(w_i\) pounds, where \(v_i\) and \(w_i\) are integers.
The thief wants to take as valuable a load as possible, but they can carry at most $W$ pounds in their knapsack, for some integer $W$. Which items should they take?

Give a dynamic-programming solution to the knapsack problem.

**Solution:** We define $mv[i,w]$ to be the maximum value obtainable by considering items 1 through $i$ with total weight no greater than $w$.

$mv[0,w] = 0$ and $mv[i,0] = 0$ for all $i$ and $w$, as no value can be obtained with zero items or with weight no greater than zero.

If the $i$-th item doesn't fit, the maximum value with $i$ items $mv[i,w]$ is the same as the maximum value with $i-1$ items $mv[i-1,w]$.

If the $i$-th item fits, we need to pick the greater of two values: the one that includes the $i$-th item, and the one that doesn't.

We can provide a recursive definition for $mv[i,w]$, and associated dynamic-programming algorithms:

$$mv[i,w] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } w = 0, \\
mv[i-1,w] & \text{if } w_i > w, \\
\max(mv[i-1,w],mv[i-1,w-w_i] + v_i) & \text{otherwise.}
\end{cases}$$

```pseudocode
KNAPSACK(W, n, v, w)
1 let mv[0..n,0..W] be a new array
2 for k = 0 to W
3    mv[0,k] = 0
4 for k = 0 to n
5    mv[k,0] = 0
6 for i = 1 to n
7    for j = 1 to W
8        if w_i > j
9            mv[i,j] = mv[i-1,j]
10       else
11           mv[i,j] = max(mv[i-1,j],mv[i-1,j-w_i] + v_i)
12    return mv

PRINT-KNAPSACK(W, n, w, mv)
1 j = W
2 for i = n to 1
3     if mv[i,j] ≠ mv[i-1,j]
4         print "Taking item " + i
5     j = j - w_i
```

The algorithm fills an $n \times W$ table, spending constant time on each cell, so the running time and space are both $\Theta(nW)$. 

The algorithm fills an $n \times W$ table, spending constant time on each cell, so the running time and space are both $\Theta(nW)$. 

5