Algorithms Middle Examination

Dec. 9, 2020 (10:10 ~ 12:20)

(Note that, if you design an algorithm, you must have pseudo-code to represent your algorithm. You can put comments after your pseudo-code to clarify your presentation.)

1. (12% Ch. 15) Design an O(N) algorithm to solve the following problem. Given an array A of length N, which consists of positive integers. A subset of A is called valid if any two elements in the subset are not adjacent in A. Among all valid subsets of A, find the one with the maximum sum. The algorithm only needs to return the sum, not the actual subset. For example, if $N = 7$ and $A = [3, 5, 3, 1, 2, 5, 1]$, the algorithm should return 11 since among all valid subsets, $\{A_1, A_3, A_6\}$ and $\{A_2, A_4, A_6\}$ both have the maximum sum 11. (Hint: This problem can be solved using dynamic programming. Define the state $dp[i]$ as the answer for the subarray $A[1] \ldots A[i]$.)

Answer: To compute $dp[i]$, observe that any valid subset in the subarray $A[1]$ ... $A[i]$ falls into exactly one of the following two categories:

(1) If $A[i]$ is selected, $A[i-1]$ must not be selected. $A[i]$ can be inserted into any valid subset in the subarray $A[1]$ ... $A[i-2]$ to form a new valid subset, so the answer is $dp[i-2] + A[i]$.

(2) If $A[i]$ is not selected, the answer is same as the maximum sum of any valid subset in the subarray $A[1]..A[i-1]$, which is $dp[i-1]$.

The subset with the maximum sum must belong to one of the two categories, so the larger one is the answer for subarray $A[1]..A[i]$. Store it into $dp[i]$. The algorithm is as follows.

```
int solve(int n, int a[]) {
    int d[n + 1];
    d[0] = 0;
    d[1] = a[1];
    for (int i = 2; i <= n; i++)
        d[i] = max(d[i - 2] + a[i], d[i - 1]);
    return d[n];
}
```

2. (10% Ch. 15) Design an O($N + M$) algorithm to solve the following problem. Given an array A of length N, an array B of length M, and both consist of positive integers. Determine if B is a subsequence of A. A subsequence of an array is some array that can be formed by deleting zero or more elements from the array while maintaining the relative position of the remaining elements. For example, if A = [3, 1, 4, 1] and B = [3, 4, 1], the algorithm should return true. If A = [3, 1, 4, 1] and B = [4, 3], the algorithm should return false.

Answer: Consider the subarray $A[1]$ ... $A[i-1]$. Let c be the maximum number such that $B[1]$ ... $B[c]$ is a subsequence of $A[1]$ ... $A[i-1]$. If $A[i] = B[c+1]$, c can be increased by 1 since $B[1]$ ... $B[c+1]$ must be a subsequence of $A[1]..A[i]$, and c is still maximal. B is a subsequence of A if and only if $c = M$ ($B[1]..B[M]$ is a subsequence of $A[1]..A[N]$).

```
bool solve(int n, int m, int a[], int b[]) {
    int c = 0;
    for (int i = 1; i <= n; i++)
        if (c < m && a[i] == b[c + 1]) c++;
    if (c == m) return true;
    else return false;
}
```

3. (10% Ch. 15) Consider a chessboard of size $k \times n$. How many ways are there to cover the chessboard completely with n rectangular bars, each of size $k \times 1$ or $1 \times k$? For instance, when $k = 2$, $n = 3$, there are three different ways: (a) cover the leftmost column by a vertical bar and the remaining region by two horizontal bars; (b) cover the rightmost column with a vertical bar, and the remaining region by two horizontal bars; or (c) cover each column with a vertical bar. Design an O(n)-time algorithm to compute the desired answer for any input k and n.

Answer: The size of chessboard is $k \times n$. When a horizontal bar $1 \times k$ is used, the only way to fill the space below is to use another $(k - 1)$ bars of size $1 \times k$. Therefore, we can simplify the question into a $1 \times n$ chessboard covered completely with 1×1 bar or $1 \times k$ bar. This problem is similar to the staircase problem in Ch15-1. Assume a staircase with n steps. We can walk up either 1 or k steps in one time. The step before reaching n is either (n-1) or (n-k). Therefore, the number of ways is $f(n) = f(n\text{-}1) + f(n\text{-}k)$. The pseudo code of the chessboard problem is below.

```
Solve (n, k):
   Let dp[] be an array of length n+1
   Let dp[0] = 0, dp[1] = 1, and dp[k] = 0 for all k < 0
   For i = 2 upto n:
     dp[i] = dp[i-1] + dp[i-k]
   return dp[n]
```

The loop executes O(n) times with O(1) operations, rendering the total time complexity O(n).

4. (10% Ch. 16) Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

Answer:
Keep taking the lightest item until the bag is full.
Greedy choice property: Let *A* be the array of *n* items. Assume A[*i*].weight < A[*j*].weight. if $i < j$. Let set O be the optimal solution for array A[1..*N*]. The optimal solution O must contain item A[1].
Proof: If the optimal solution does not contain A[1], we can always replace A[1] with any item A[*j*] in O and still be an optimal solution because A[1].weight < A[*j*] and A[1].value > A[j].value.

Optimal Substructure: O – A[1] must be the optimal solution for sub-array A[2..N].
Proof: If O - A[1] is not the optimal solution of sub-array A[2..*N*], there exists an optimal solution O' is better than O - A[1]. It implies that O' + A[1] is an optimal solution for array A[1..*N*], which is better than O. It Conflicts!

```
modified_01_knapsack(I, W)
    sort(I)             \\ sort it in weight increasing order

    ANS = empty array
    total_weight = 0
    for item in I
        total_weight += item.weight
        if total_weight <= W
            ANS.push(item)
        else
            break

    return ANS
```

5. (10% Ch. 16) What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers? *a*:1 *b*:1 *c*:2 *d*:3 *e*:5 *f*:8 *g*:13 *h*:21. Can you generalize your answer to find the optimal code when the frequencies are the first *n* Fibonacci numbers?

Answer: (a) 1111111, 1111110, 111110, 11110, 1110, 110, 10, 0, respectively.
(b) Assuming there are N Fibonacci numbers. Then (0-indexed):
code(i) = "0", if i = N - 1
"1" + code(i + 1), if 0 < i
"1" * (N – 1), if i = 0

6. (10% Ch. 17) Suppose we perform a sequence of *n* operations on a data structure where the *i*th operation costs *i* if *i* is an exact power of 2, and 1 otherwise.

a. (4%) Use aggregate analysis to determine the total amortized cost in Big-Oh notation.

b. (4%) Using the potential method, show that the potential function $\Phi(D_i) = i - 2^{\lfloor \lg i \rfloor}$ cannot yield useful result.

c. (2%) Show how to modify $\Phi(D_i)$ so that the potential method analysis gives the exact result.

Answer:

a. The amortized cost is $\sum_{k=0}^{\text{floor}(\lg n)} 2^k + \sum_{i=1}^{n} 1 - \sum_{k=0}^{\text{floor}(\lg n)} 1$

$$\leq \sum_{k=0}^{\text{floor}(\lg n)} 2^k + n \leq 2n + n = 3n = O(n)$$

b. If i is not an exact power of 2, then the potential difference is $\Phi(D_i) - \Phi(D_{i-1}) = i - 2^{\lfloor \lg i \rfloor} - i + 1 + 2^{\lfloor \lg i-1 \rfloor} = 1$.
Otherwise, it is $\Phi(D_i) - \Phi(D_{i-1}) = i - 2^{\lfloor \lg i \rfloor} - i + 1 + 2^{\lfloor \lg i-1 \rfloor}$
$= i - i - i + 1 + \frac{i}{2} = 1 - \frac{i}{2}$
Then the amortized operation per cost is either $1 + 1 = 2 = O(1)$ or $i + 1 - \frac{i}{2} = 1 + \frac{i}{2}$. The presence of i halts further useful analysis.

c. Let $\Phi'(D_i) = 2(i - 2^{\lfloor \lg i \rfloor})$, then $\Phi'(D_i) - \Phi'(D_{i-1}) = 2$ or $2 - i$.
The amortized operation per cost is either $1 + 2 = 3 = O(1)$ or $i + 2 - i = 2 = O(1)$. The total amortized cost is $nO(1) = O(n)$.

7. (14% Ch. 17) Suppose that we wish to implement a dynamic table T with insertion and deletion operations. Let α be the load factor of the table T. Each insertion (deletion) operation will insert (delete) an item from the table T. If T is full before insertion of an item, we expand T by doubling its size. If T is bellow (1/4)-full after deletion, we contract T by halving its size. Let α_i denote the load factor of table T after the ith operation. Please answer the following question:

a. (5%) What is the amortized cost of the ith operation if the ith operation is an insertion, $\alpha_{i-1} < 1/2$, and $\alpha_i < 1/2$?

b. (5%) What is the amortized cost of the ith operation if the ith operation is a deletion and $\alpha_{i-1} > 1/2$?

c. (4% Bonus) What is the amortized cost of the ith operation if $\alpha_{i-1} = 1/2$ and $\alpha_i < 1/2$?

Answer:

Define a potential function s.t.,

$$\begin{cases} \Phi(T) = 2 \cdot \text{num}(T) - \text{size}(T), & \text{if the table is at least half full} \\ \Phi(T) = \dfrac{\text{size}(T)}{2} - \text{num}(T), & \text{if the table is less than half full} \end{cases}$$

a. The insertion does not cause an expansion. The amortized cost is $c_i + \Phi_i - \Phi_{i-1}$

$$= 1 + \left(\frac{\text{size}_i}{2} - \text{num}_i\right) - \left(\frac{\text{size}_{i-1}}{2} - \text{num}_{i-1}\right)$$

$$= 1 + (-1) = 0$$

b. No contraction happens. The amortized cost is $c_i + \Phi_i - \Phi_{i-1}$

$$= c_i + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1})$$

$$= 1 + (-2) = -1$$

c. The statement implies a deletion operation.

The amortized cost is $c_i + \Phi_i - \Phi_{i-1}$

$$= c_i + \left(\frac{\text{size}_i}{2} - \text{num}_i\right) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1})$$

$$= c_i + \left((\text{num}_i + 1) - \text{num}_i\right) - 0$$

$$= 1 + 1 = 2$$

8. (10%, Ch. 22) Given an undirected graph $G = (V, E)$, check if there is a way to color the nodes in red or blue such that no two adjacent nodes have the same color. The algorithm should run in $O(V + E)$-time. You have to specify the termination condition.

Answer:

A graph is said to be a bipartite graph if its vertices may be colored in two colors in such a way that no two adjacent nodes have same color.
First determine the root color, and let all of its children another color. Use DFS or BFS to keep explore and color the nodes along the way. If we attempt to color *v* into some color but it is already the other color, the graph is not a bipartite graph. If all of the nodes are successfully colored and no conflict occurs, the graph is a bipartite graph.

```
isBipartite(G)

    for v in V
        color[i] = -1

    start = 1
    color[start] = 1
    q.push(start)

    while (!q.empty())

        u = q.pop()

        for v in u.children
            if color[v] == -1
                color[v] = another color from color[u];
                q.push(v)
            else if color[v] == color[u]
                return false
return true
```

9. (10% Ch. 22) Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a simple cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

Answer:
Simply do DFS on G and see if we find a back edge. An undirected graph without a cycle cannot have more than |V|-1 edges, so the time complexity of the algorithm is O(V)

```
cycle_detect(v)
    visit(v)
    for u in v.neighbor
        if u is visited && u is not v's parent
            return true
        else if cycle_detect(u)
            return true
```

10. (10% Ch. 22) Prove that for any directed graph G, we have $((G^T)^{SCC})^T = G^{SCC}$. That is, the transpose of the component graph of G^T is the same as the component graph of G.

Answer:

$(U,V) \in G^{SCC} \Rightarrow (U,V) \in ((G^T)^{SCC})^T$
Let U and V be vertices in G^{SCC}, and $(U,V) \in G^{SCC}$. There must be $u \in U, v \in V$ such that $(u,v) \in G$, and so $(v,u) \in G^T$ We know that G^{SCC} and $(G^T)^{SCC}$ has same vertices, so $(V,U) \in (G^T)^{SCC}$, and so $(U,V) \in ((G^T)^{SCC})^T$.

$(U,V) \in G^{SCC} \Leftarrow (U,V) \in ((G^T)^{SCC})^T$
Let U and V be vertices in $((G^T)^{SCC})^T$, and $(U,V) \in ((G^T)^{SCC})^T \Rightarrow (V,U) \in (G^T)^{SCC}$. There must be $u \in U, v \in V$ such that $(v,u) \in G^T$, and so $(u,v) \in G$
We know that G^{SCC} and $(G^T)^{SCC}$ has same vertices, and so $(U,V) \in G^{SCC}$.