Advanced Analysis of Algorithms - Practice Midterm (Solutions)

L. Kovalchick
LCSEE,
West Virginia University,
Morgantown, WV
{lynn@csee.wvu.edu}

1 Problems

1. Show that

$$\sum_{i=1}^{n} i^3 = [\frac{n \cdot (n+1)}{2}]^2.$$

Proof:
Base case $P(1)$:

$$\begin{aligned} LHS &= \sum_{i=1}^{1} i^3 \\ &= 1^3 \\ &= 1 \\ RHS &= [\frac{1 \cdot (1+1)}{2}]^2 \\ &= [\frac{1 \cdot (2)}{2}]^2 \\ &= [\frac{2}{2}]^2 \\ &= [1]^2 \\ &= 1 \end{aligned}$$

Thus, $LHS = RHS$ and $P(1)$ is true.
Let us assume that $P(k)$ is true, i.e.

$$\sum_{i=1}^{k} i^3 = [\frac{k \cdot (k+1)}{2}]^2$$

We need to show that $P(k+1)$ is true.

$$\begin{aligned} LHS &= \sum_{i=1}^{k+1} i^3 \\ &= 1^3 + 2^3 + 3^3 + \ldots + k^3 + (k+1)^3 \end{aligned}$$

$$
\begin{aligned}
&= [\frac{k\cdot(k+1)}{2}]^2 + (k+1)^3 \ \ (\textit{using the inductive hypothesis})\\
&= \frac{k^2\cdot(k+1)^2}{4} + (k+1)^3\\
&= \frac{k^2\cdot(k+1)^2 + 4\cdot(k+1)^3}{4}\\
&= \frac{(k+1)^2\cdot[k^2+4\cdot(k+1)]}{4}\\
&= \frac{(k+1)^2\cdot[k^2+4k+4)]}{4}\\
&= \frac{(k+1)^2\cdot(k+2)^2}{4}\\
&= [\frac{(k+1)\cdot(k+2)}{2}]^2\\
RHS &= [\frac{(k+1)\cdot((k+1)+1)}{2}]^2\\
&= [\frac{(k+1)\cdot(k+2)}{2}]^2
\end{aligned}
$$

LHS=RHS. Thus, we have shown that $P(k) \rightarrow P(k+1)$; applying the principle of mathematical induction, we conclude that the conjecture is true. □

2. Consider the recurrence relation:

$$
\begin{aligned}
T(n) &= 1, \ \textit{if} \ \ n = 1\\
&= T(n-1) + 2^n, \ \ \textit{otherwise}
\end{aligned}
$$

Show that $T(n) = 2^{n+1} - 3$.

Proof:
Base Case:

$$T(1) = 1$$

Using expansion:

$$
\begin{aligned}
T(n) &= T(n-1) + 2^n\\
&= T(n-2) + 2^{n-1} + 2^n\\
&= T(n-3) + 2^{n-2} + 2^{n-1} + 2^n\\
&\vdots\\
&= T(n-(n-1)) + 2^{n-(n-2)} + 2^{n-(n-3)} + \ldots + 2^{n-1} + 2^n\\
&= T(1) + 2^2 + 2^3 + \ldots + 2^{n-1} + 2^n\\
&= T(1) + \sum_{i=2}^{n} 2^i\\
&= 1 + \sum_{i=0}^{n} 2^i - 2^0 - 2^1
\end{aligned}
$$

$$
\begin{aligned}
&= 1 + \frac{2^{n+1}-1}{2-1} - 1 - 2 \\
&= 1 + \frac{2^{n+1}-1}{1} - 1 - 2 \\
&= 1 + 2^{n+1} - 1 - 1 - 2 \\
&= 2^{n+1} - 3
\end{aligned}
$$

□

3. Give an algorithm which will return the third largest element of a heap with $heap - size \geq 3$, in $O(1)$ time. Assume that all elements are unique.

   Solution:
   Observe that a heap is a complete binary tree with all levels filled, except maybe the last. A heap of size at least 3, therefore must have at least the first two levels filled. This means that the root must have both a left and right child. Observe that, by definition, a heap must satisfy the heap property which states that for every node i other than the root, $A[PARENT(i)] \geq A[i]$. Since we have assumed that all elements are unique, we can restate the heap property as: for every node i other than the root, $A[PARENT(i)] > A[i]$. Thus, the subtrees rooted at the right child of the root and the left child of the root are themselves heaps and the only possible values for the third largest element are located at levels 1 or 2. Therefore, we only need to check these two levels in order to determine the third largest element. Observe that this requires only $O(1)$ time. □

4. Describe an algorithm that, given n integers in the range from 1 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range from $[a \cdots b]$ in $O(1)$ time. Your algorithm should use $O(n+k)$ preprocessing time.

   Solution:
   Assume that our integers are stored in array $A[1 \cdots n]$ and thus, $length[A] = n$. We need two additional arrays $B[1 \cdots k]$ and $C[1 \cdots k]$. Our algorithm will first initialize all elements of array B to 0. This step requires $O(k)$ time. Next, for each element of array A (i.e., $A[i]$ for $i = 1, 2, \cdots, n$), we will increment $B[A[i]]$. Observe that this step will require $O(n)$ time and $B[j]$ for $j = 1, 2, \cdots, k$, now contains the number of elements of A having value j. Finally, make $C[1] = B[1]$ and for each element l of array C where $l = 2, 3, \cdots, k$, we will compute $C[l] = B[l] + B[l-1]$. This step takes $O(k)$ time.
   Now, to answer any query about how many of n integers fall into a range $[a \cdots b]$, we simply compute $C[b] - C[a] + B[a]$. Observe that this computation requires only $O(1)$ time after the $O(n+k)$ preprocessing time. □

5. Suppose a biker is about to go on a ride on a bike trail carrying a single knapsack. Suppose further that she knows the maximum total weight W that she can carry, and she has a set S of n different useful items that she can potentially take with her. Let us assume that each item j has an integer weight w_j and a benefit value b_j, which is a value that the biker assigns to item j. Her problem is to optimize the value of the items that she places into her knapsack, without going over the weight limit W (i.e., maximize $\sum_{j \in T} b_j$ subject to $\sum_{j \in T} w_j \leq W$). Give a recursive definition for the profit value of the optimal solution to this problem.

   Solution:
   Let us number the items in S as $1, 2, \cdots, n$ and define, for each $k \in \{1, 2, \cdots, n\}$, the subset
   $S_k = \{$ items in S labeled $1, 2, \cdots, k\}$. Now, let $B[k, w]$ be the maximum total value of a subset of S_k from among all those subsets having total weight exactly w. We can then define the following recursive definition for $B[k, w]$:

$$
\begin{aligned}
B[k,w] &= B[k-1,w], && \text{if } w_k > w \\
&= max\{B[k-1,w], B[k-1, w-w_k] + b_k\}, && \text{otherwise}
\end{aligned}
$$

What the above recurrence is saying is as follows: When a decision has to be made on the k^{th} object, there are two possibilities, viz., the weight of the k^{th} object is greater than the knapsack capacity or its weight is less than or equal to the knapsack capacity. In the former case, $B[k, w] = B[k-1, w]$, since the k^{th} object has to be excluded. In the latter case, we solve the subproblem resulting from including the k^{th} object, i.e., $B[k-1, w-w_k] + b_k$ and the subproblem resulting from excluding it, i.e., $B[k-1, w]$ and take the maximum of the two! I leave it as an exercise to argue that the principle of optimality applies in the case of the 0/1 knapsack problem. □