Computer Science CSC263H | January 16, 2020
St. George Campus | University of Toronto

Solutions for Homework Assignment #1

Answer to Question 1. $T(n)$ is $\Theta(n)$. To prove this, we now show that $T(n)$ is both $O(n)$ and $\Omega(n)$:

1. $T(n)$ is $O(n)$.

   This is not obvious because the procedure has two loops, one nested inside the other, each loop with an index ranging from 1 to n. So it may seem that the inner statement of line 4 is executed n^2 times (once for each value of i and j where $1 \leq i \leq n$ and $1 \leq j \leq n$). As we show below, however, this can never happen.

   Claim: For every possible input array A, the ***outer*** loop of lines 2-5 is executed ***at most 2 times***.

   Proof: Let $n \geq 2$, and consider any input array $A[1..n]$.

   If $A[1] \neq n-1$ then $\textsc{nothing}(A)$ returns in line 5 of iteration $i = 1$ of the loop of lines 2-5.

   Now assume that $A[1] = n - 1$ (*).

   If $A[2] \neq n-2$ then $\textsc{nothing}(A)$ returns in line 5 of iteration $i = 2$ of the loop of lines 2-5.

   Now assume that $A[2] = n - 2$ (**).

   From (*) and (**), $A[1] + A[2] = 2n - 3$.

   So $\textsc{nothing}(A)$ returns in line 5 of iteration $i = 2$ of the outer loop of lines 2-5.

   Thus, in ***all*** possible cases, the ***outer*** loop of lines 2-5 is executed ***at most 2 times***. ***Q.E.D.***

   The above claim implies the following:

   For all $n \geq 2$, for *every* input A of size n, $\textsc{nothing}(A)$ executes *at most* $2n$ iterations of the *inner* loop of lines 3–4.

   Since each one of these inner loop iterations takes a **constant** time, it now clear that:

   There is a constant $c > 0$ such that for all $n \geq 2$,
   for *every* input A of size n, $\textsc{nothing}(A)$ takes *at most* $c \cdot n$ time.

   So $T(n)$ is $O(n)$.

2. $T(n)$ is $\Omega(n)$.

   This is not obvious because the procedure may return "early" (e.g., after executing only a constant number of inner loop iterations) because of the loop exit conditions in line 4 and 5. Thus, to show that $T(n)$ is $\Omega(n)$, we must show that there is at least one input array A such that the procedure executes a linear number of inner loop iterations on this input, ***despite the loop exit conditions of lines 4 and 5.*** We do so below.

   Let $n \geq 2$ and consider the input $A = [n, n-1, n-2, \ldots, 2, 1]$, i.e., $A[n-j+1] = j$ for all $1 \leq j \leq n$. Since $A[n-j+1] \neq j$ does ***not*** hold for ***any*** j such that $1 \leq j \leq n$, with ***this*** input A, the ***inner*** loop of lines 3–4 ***never*** returns because of the condition $A[n-j+1] \neq j$ in line 4.

   So for ***this*** input A, $\textsc{nothing}(A)$ executes ***at least*** n iterations of the ***inner*** loop of lines 3–4.

   Since each one of these iterations takes a **constant** time, we conclude that:

   There is a constant $c > 0$ such that for all $n \geq 2$,
   for *some* input A of size n, namely $A = [n, n-1, \ldots, 1]$, $\textsc{nothing}(A)$ takes *at least* $c \cdot n$ time.

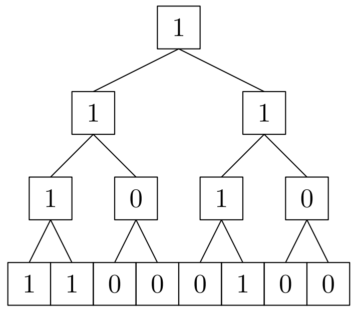
   Important note: For many arrays A of size n, for example all those where $A[n] \neq 1$, those where $A[n] = 1$ but $A[n-1] \neq 2$, etc..., the execution of procedure $\textsc{nothing}(A)$ takes only constant time! This is because the execution stops "early", in line 4 (with $i = 1$), on these arrays.

   So to prove that the worst-case time complexity of the procedure $\textsc{nothing}()$ is $\Omega(n)$, a correct argument ***must explicitly describe*** some specific input array A of size n for which the execution of $\textsc{nothing}(A)$ does take time proportional to n.

Answer to Question 2. The basic idea is to combine an n-bit vector with a complete binary tree, and use a structure that represents this tree without pointers.

a. The n-bit vector that represents S is as follows: the i-th bit of the vector V is set to 1 if and only if $i \in S$. The bits of V are also the leafs of a complete binary tree. Each node v of that tree is a single bit such that $v = 1$ if and only if the subtree rooted at v contains at least one leaf that is set to 1, i.e., it contains a leaf i such that $i \in S$.

For example, the data structure representation $n = 8$ and $S = \{1, 2, 6\}$ is as follows:



Note that each node v in the tree is just the logical "OR" of its two children.

To avoid pointers in the above tree structure (and thus save space), we use the *array representation* of a complete binary tree, just like we do to represent the complete binary tree of a HEAP! So we use a binary array A where bits $A[2i]$ and $A[2i + 1]$ are the left and right children of bit $A[i]$, respectively.

For example, the array A for $n = 8$ and $S = \{1, 2, 6\}$ is as follows:

$$\underbrace{\boxed{1\,|\,1\,|\,1\,|\,1\,|\,0\,|\,1\,|\,0}}_{\text{binary tree}}\underbrace{\boxed{1\,|\,1\,|\,0\,|\,0\,|\,0\,|\,1\,|\,0\,|\,0}}_{\text{8-bit vector for } S=\{1,2,6\}}$$

This data structure takes only $2n - 1$ bits of space: n bits for the n-bit vector V representing S, and $n - 1$ bits for the nodes of the tree above V.

Note that the entire structure can be thought of a *single* complete binary tree with $2n - 1$ nodes: n leafs and $n-1$ internal nodes. The depth of this tree is $\log_2 n$. In our example, $n = 8$ and the depth is $\log_2 8 = 3$.

b. MAXIMUM: the maximum element of S is the index of the rightmost "1" in the bit vector V (which is the also the rightmost leaf that contains 1 in the complete binary tree). To find the rightmost "1" in V, start at the root of the complete binary tree, and follow the rightmost path that contains only 1's. In the example above, this path is $A[1], A[3], A[6], A[13]$, and it leads to the 6-th element in the binary vector, i.e., it leads to element 6 of S.

The *index* of the rightmost "1" in the n-bit vector is just its index in the array A minus $n - 1$. In the example above, the rightmost "1" in the 8-bit vector is $A[13]$, and its index in the n-bit vector is $13 - (n - 1) = 13 - 7 = 6$. Note that 6 is indeed the maximum element of the set S in our example.

The worst-case time complexity of the MAXIMUM operation is proportional to the depth of the tree, more precisely it is $\Theta(\log_2 n)$.

INSERT(j): Set bit j of the n-bit vector to 1, and set all its ancestors in the tree also to 1. More precisely, start from the bit A that represents element j of S, namely bit $A[j + (n - 1)]$, and set it to 1. Then go to its parent $A[\lfloor j + (n - 1)/2 \rfloor]$ and set it to 1, etc., and continue this way up to the root and set $A[1]$ to 1.

Alternatively, one can start by setting $A[1]$ to 1, and do the same for every element of A in the path from $A[1]$ to $A[j + (n - 1)]$ (think about how you would find this path...).

The worst-case time complexity of an INSERT(j) operation is $\Theta(\log_2 n)$.

c. MEMBER(j): Check whether bit j of the n-bit vector is 1. More precisely, MEMBER(j) = TRUE iff $A[j + (n - 1)] = 1$. The worst-case time complexity of this operation is clearly $\Theta(1)$.