Answers for Homework Problems on Ch 4 & 8

Hanan Ayaad.

9 in 4.1

- Idea: First, we divide the array into two halves, $B = A[1 \dots \lceil n/2 \rceil]$ and $C = A[\lceil n/2 \rceil \dots n]$. Second, we count the number of inversions in each half. Third, we need to count the number of inversions $(A[i], A[j])$ where a_i & a_j belong to different halves. We must do the third step in $O(n)$ to achieve $O(n \log n)$ for the overall algorithm.

It is noted that the pairs $(A[i], A[j])$ are such that $A[i]$ belongs to left half (B) and $A[j]$ belongs to the right half (C) and $A[i] > A[j]$.

By merge-sorting the two halves, these inversions can be counted during the merging step (which takes $\Theta(n)$ time), as follows:

If $A[i] < A[j]$, we output $A[i]$ to the merged (sorted) list without incrementing the number of inversions, because $A[i]$ can't be part of an inversion with any of the elems of C, since they are all greater than $A[j]$.

If $A[i] > A[j]$, we output $A[j]$ to the sorted list and increment the number of inversions by the size of $B - i + 1$, the number of elements in B that form an inversion with $A[j]$.

The total number of inversions is the sum of the number of inversions in the right half (I_{left}), the number of inversions in the right half (I_{right}) and the number of inversions in the merging (I_{merge})

- Pseudo-Code: Same as mergesort with the additional instructions for computing the number of inversions.
- Analysis $t(n) = 2\,t(\frac{n}{2}) + \Theta(n)$

Then Master Theorem $\Rightarrow$ $t(n) = \Theta(n \log n)$

8 in Ex. 4.2

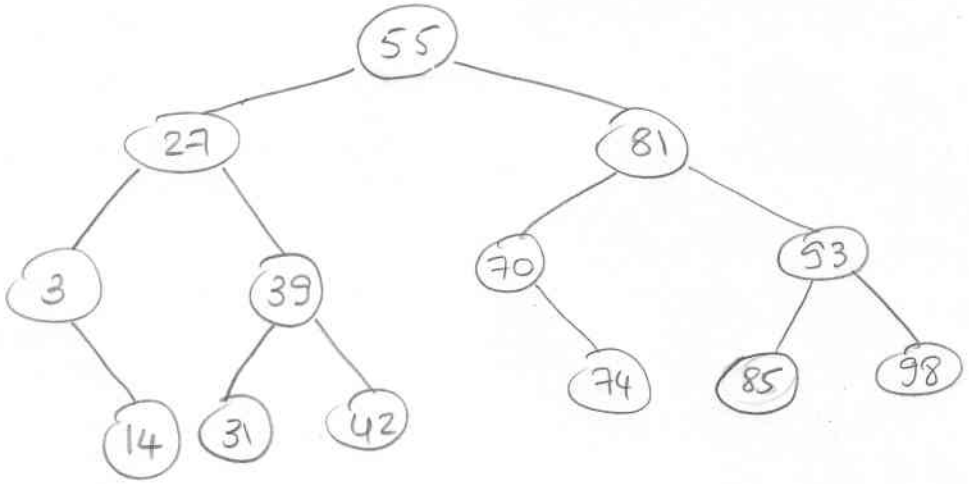
The algorithm scans the array once from both ends, left and right, simultaneously, moving +ve numbers to the right and negative numbers to the left parts of the array. The algorithm is in-place.

```
Alg. NegPreceedPositive (A[1...n])
    i = 0 , j = n
    While i < j do
        If A[i] < 0
            i = i+1
        else
            swap (A[i], A[j])
            j = j-1
        endif
    endwhile.
```

Any alg for this problem must visit each entry at least once, and hence, $\Theta(n)$. is the most efficient one can get.

1 in Ex. 4.3

(a) Given the sorted array of 13 numbers, the corresponding binary search tree is as shown below. The largest number of key comparisons in this tree is 4. It can also be computed as $\lceil \log_2 13 \rceil + 1 = 4$.



1 in Ex. 4.3 (Con't)

(b) The largest # of key comparisons is required for each of the elements at the last level (level 3), which are 14, 31, 42, 74, 85, and 98

(c) Av. No. of key comparison in a succesful search $= \frac{1}{13}*1*1 + \frac{1}{13}*2*2 + \frac{1}{13}*3*4 + \frac{1}{13}*4*6$

$= \frac{41}{13} \simeq 3.15.$

(d) Av. No. of key comparisons in an unsuccesful search $= \frac{1}{14}*3*2 + \frac{1}{14}*4*12 = \frac{54}{14} \approx 3.85$

---

6 in Ex. 8.1

See Lecture.

---

7. in EX 8.1

(a) Using $C(n,k) = \frac{n!}{k!(n-k)!}$

requires $n+k+n-k = 2n$ multiplications & one division.

(b) Using $C(n,k) = \frac{n(n-1)\cdots(n-k+1)}{k!}$

requires $k+k = 2k$ multiplications & one division.

(c) $C(n,k) = C(n-1,k-1) + C(n-1,k)$ for $n>k>0$, $C(n,0) = C(n,n) = 1$,

requires $C(n,k)-1$ additions.

(d) Dynamic programming requires $nk - \frac{1}{2}k^2 - \frac{1}{2}k$ additions $\Rightarrow \Theta(nk)$.

The choice would be between (b) & (d), but they have different basic operations. Nevertheless, we can argue that (b) is more efficient than (d) since it is $\Theta(k)$ compared to $\Theta(nk)$ for alg (d). It is also more space efficient.

#7 in Ex. 8.2

Applying Floyd's alg. to the given weight matrix, we get the following sequence of matrices:

$$D^{(0)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & \infty \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & \infty & \infty & \infty & 0 \end{bmatrix} \qquad D^{(1)} = \begin{bmatrix} 0 & 2 & \infty & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & \infty & 4 & 0 \end{bmatrix}$$

$$D^{(2)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix} \qquad D^{(3)} = \begin{bmatrix} 0 & 2 & 5 & 1 & 8 \\ 6 & 0 & 3 & 2 & 14 \\ \infty & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 8 & 4 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ \infty & \infty & 0 & 4 & 7 \\ \infty & \infty & 2 & 0 & 3 \\ 3 & 5 & 6 & 4 & 0 \end{bmatrix} \qquad D^{(5)} = \begin{bmatrix} 0 & 2 & 3 & 1 & 4 \\ 6 & 0 & 3 & 2 & 5 \\ 10 & 12 & 0 & 4 & 7 \\ 6 & 8 & 2 & 0 & 3 \\ 3 & 5 & 6 & 4 & 0 \end{bmatrix} = D$$

#3 in Ex. 8.3

The entries of the root table R, $k = R[i,j]$, indicate that the root of the optimal subtree consisting of the list of ordered keys $a_i, \ldots, a_k, \ldots, a_j$ is the kth key, where $i \le k \le j$. The top right entry $k = R[1,n]$ indicates that the root of the optimal tree is the k-th key in $a_1, \ldots, a_n$. The roots of its left & right subtrees are indicated by $R[1,k-1]$ and $R[k+1,n]$, respectively.

The following alg. traces the roots of each subtree, hence visiting each node of the BST once. It may be set to generate an inorder, postorder or pre-order traversal of the BST. Thus it is $O(n)$.

- Call OptimalTree(1,n)

Alg OptimalTree(i,j)

```
if i ≤ j
    k = R[i,j]
    OptimalTree(i, k-1)
    print(k)
    OptimalTree(k+1, j)
```

The algorithm prints the indices of the nodes of the optimal BST in inorder (i.e left-root-right)

#8 in Ex. 8.3

The dynamic programming alg finds the root of an optimal BST for keys $a_i, \dots, a_j$ by minimizing $\{C[i,k-1] + C[k+1,j]\}$ for $i \le k \le j$.

It is noteworthy that the entries of the root table are always non-decreasing along each row & column. Thus, $R[i,j-1] \le R[i,j] \le R[i+1,j]$.

This observation can be used to limit the bounds of the innermost loop of the algorithm to $R[i,j-1]$ and $R[i+1,j]$, as lower & upper bounds, respectively.

The runtime of the algorithm is given by:

$$T(n) = \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=R[i,j-1]}^{R[i+1,j]} 1 = \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=R[i,i+d-1]}^{R[i+1,i+d]} 1$$

$$= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \left(R[i+1,i+d] - R[i,i+d-1] + 1 \right)$$

$$= \sum_{d=1}^{n-1} \left(\underbrace{\sum_{i=1}^{n-d} \left(R[i+1,i+d] - R[i,i+d-1] \right)}_{\text{Sum \#1}} + \underbrace{\sum_{i=1}^{n-d} 1}_{\text{Sum \#2.}} \right)$$

By "telescoping" Sum #1, we get the following where all the terms except two get cancelled:

$$\sum_{i=1}^{n-d} (R[i+1, i+d] - R[i, i+d-1])$$

$$= (R[2, 1+d] - R[1, d]) + (R[3, 2+d] - R[2, 1+d]) + (R[4, 3+d] - R[3, 2+d]) + \cdots + (R[n-d+1, n] - R[n-d, n-1]) = R[n-d+1, n] - R[1, d]$$

Sum #2 is equal to $n-d$.

Thus,

$$\sum_{i=1}^{n-d} (R[i+1, i+d] - R[i, i+d-1]) + \sum_{i=1}^{n-d} 1 = R[n-d+1, n] + R[1, d] + n - d < 2n$$

Hence,

$$\sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=R[i, j-1]}^{R[i+1, j]} 1 = \sum_{d=1}^{n-1} (R[n-d+1, n] - R[1, d] + n - d) < \sum_{d=1}^{n-1} 2n < 2n^2$$

Since $R[i+1, i+d] - R[i, i+d-1] \geq 0$

$$\sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=R[i, j-1]}^{k=R[i+1, j]} 1 = \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} (R[i+1, i+d] - R[i, i+d-1] + 1)$$

$$\geq \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} 1 = \sum_{d=1}^{n-1} (n-d) = \frac{(n-1)n}{2} \geq \frac{1}{4} n^2 \quad \text{for } n \geq 2$$

Therefore, the efficiency class of the algorithm for finding an optimal BST is $\Theta(n^2)$.