CMP_SC 3050: Homework 2 answers

1. Consider the following computational problem.

   Input: A sequence $a_1, \ldots, a_n$ of n integers stored in an array A.
   Output: A doubly-indexed array B of size $n \times n$ such that

   $$B[i,j] = \begin{cases} 0 & \text{if } j < i \\ \max\{a_k | i \le k \le j\} & \text{otherwise.} \end{cases}$$

   Give an algorithm that solves the above computational problem in $O(n^2)$ running time.

   Answer. Observe that we can write

   $$B[i,j] = \begin{cases} 0 & \text{if } j < i \\ a_i & \text{if } j = i \\ \max(B[i,j-1], a_j) & \text{otherwise.} \end{cases}$$

   So, this gives the following algorithm:

   ```
   PARTIAL-MAXS(A)
     n = A.size
     for i = 1 to n
         for j = 1 to n
             if j < i
                 B[i, j] = 0
             elseif i == j
                 B[i, j] = A[i]
             elseif B[i, j - 1] > A[j]
                 B[i, j] = B[i, j - 1]
             else B[i, j] = A[j]
   ```

   It is easy to see that because of two nested loops, each line is executed at most n^2 times giving us an $O(n^2)$ algorithm.

2. Consider the following computational problem.

Input:	A *sorted* sequence $a_1, \ldots, a_n$ of n integers stored in an array A and a number v.
Output:	The number of times v occurs in A.

Give an algorithm that solves the above computational problem in $O(\log n)$ running time.

Answer. Since the array is sorted, v occurs contiguously in the array. So if v occurs the first time at position i in the array and for the last time at position j, all occurrences of A happen between i and j. So, the number of occurrences of A is $j - 1 + 1$. We can compute i and j using the algorithms BinSearchLow and BinSearchHigh respectively below. Note these functions are minor modifications of Binary Search.

BinSearchLow($A, v, low, high$)

- **if** $high < low$
  - **return** -1
- **if** $high == low$ and $A[low] \neq v$
  - **return** -1
- **if** $A[low] == v$
  - **return** low
- $mid = \lfloor \frac{low+high}{2} \rfloor$
- **if** $A[mid] == v$
  - **if** $A[mid\text{-}1] < v$
    - **return** mid
- **elseif** $A[mid] \geq v$
  - **return** BinSearch($A, v, low + 1, mid - 1$)
- **else return** BinSearch($A, v, mid + 1, high$)

BinSearchHigh($A, v, low, high$)

- **if** $high < low$
  - **return** -1
- **if** $high == low$ and $A[high] \neq v$
  - **return** -1
- **if** $A[high] == v$
  - **return** $high$
- $mid = \lfloor \frac{low+high}{2} \rfloor$
- **if** $A[mid] == v$ and $A[mid + 1] > v$
  - **return** mid
- **elseif** $A[mid] > v$
  - **return** BinSearch($A, v, low, mid - 1$)
- **else return** BinSearch($A, v, mid + 1, high - 1$)

So, now the algorithm is just one call each to the above algorithms.

COUNTOCCURRENCES(A, v)

$n = A.size$
$i =$ BINSEARCHLOW$(A, v, 0, n)$
if $i == -1$
  return 0
$j =$ BINSEARCHHIGH(A, v, i, n)
return $j - i + 1$

BINSEARCHHIGH and BINSEARCHLOW are $O(\log n)$ algorithms because they give the same recurrence as binary search. From this it is easy to see that COUNTOCCURRENCES is $O(2 \log n)$ which is $O(\log n)$.

3. We want to solve the following computational problem.

Input:	A sequence $a_1, \ldots, a_n$ of n integers stored in an array A and a number $k \leq n$.
Output:	A one-dimensional array B such that

$$B[i] = \begin{cases} 0 & \text{if } i < k \\ k\text{-th smallest number of } A[1 \ldots i] & \text{otherwise.} \end{cases}$$

Give an algorithm that solves the above problem in $O(n \log k)$ running time.

Answer. The key idea in this problem is the following:

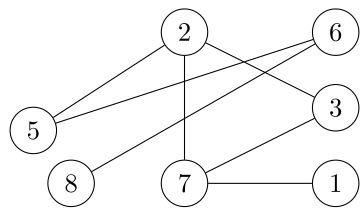
- We can traverse the array A from 1 to n. As we keep going from 1 to n, we can remember the smallest k values of $A[1 \ldots i]$ in an array C.
- Now, the k-th smallest value is the maximum value of C.
- We learnt a great structure in class which is useful to maintain the maximum value of C, namely a max-heap. Therefore, we will maintain C as a max-heap. The maximum value of C, thus, resides at index 1.
- When reading $A[i + 1]$, we will compare this new value with $C[1]$. If this number is smaller than $C[1]$ then we will replace $C[1]$ by $A[i+1]$ and rebuild the heap. We need only use MAX-HEAPIFY as there is at most error when we replace $C[1]$ by $A[i + 1]$. Otherwise we will continue.

This yields the following algorithm

```
kthSmallest(A, k, n)
1  n = A.size
2  if k ≥ n
3        error "k must be smaller than n"
4
5  for i = 1 to k // Copy first k elements of A into C
6        C[i] = A[i]
7        B[i] = 0
8
9  Build-Max-Heap(C, k) // Now make C a heap
10 B[k] = C[1] // Copy the kth smallest element seen thus far to B[k]
11
12 for i = k + 1 to n // Now fill the rest of the array B.
13       if C[1] > A[i] // New element is smaller than the highest element of C
14             C[1] = A[i]
15             Max-Heapify(C, 1, k)
16       B[i] = C[1]
```

Now, in the above algorithm, lines 5–10 take $O(k)$ time which is smaller than $O(n \log k)$ time. The loop in lines 12–16 runs $n-k$ times and spends at most $O(\log k)$ time each time it runs. Therefore the loop in lines 12–16 takes $O((n-k)\log k)$ time which is smaller that $O(n \log k)$ time. Thus, the total running time $O(n \log k)$ time.

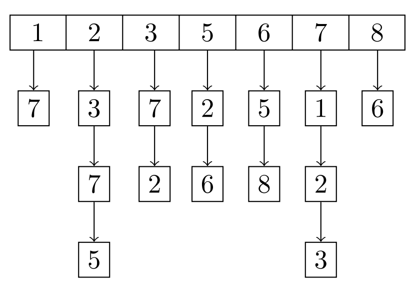
4. Consider the following undirected graph G:



(a) Give the adjacency matrix representation of G.

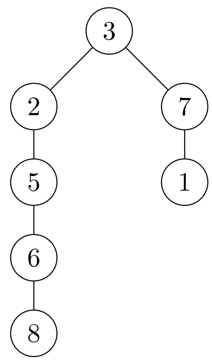
	1	2	3	5	6	7	8
1	0	0	0	0	0	1	0
2	0	0	1	1	0	1	0
3	0	1	0	0	0	1	0
5	0	1	0	0	1	0	0
6	0	0	0	1	0	0	1
7	1	1	1	0	0	0	0
8	0	0	0	0	1	0	0

(b) Give the adjacency list representation of G.



(c) Give the distances and the BFS tree generated by running the Breadth First Search algorithm on G with 3 as the source vertex.

Tree:



Distances:

$$
\begin{aligned}
3.d &= 0\\
2.d &= 1\\
7.d &= 1\\
1.d &= 2\\
5.d &= 2\\
6.d &= 3\\
8.d &= 4
\end{aligned}
$$