1. A random complete binary tree in an array format is given to you. Develop an algorithm to verify whether the array is a max-heap or not.

1. How would you decide if the array is a max-heap?

   The brute-force is to test the max-heap property by applying heap-sort without first applying the build-maxheap function to the array and seeing if it results in a sorted array. If not the array could not have been a maxheap prior to the next stage of the heap-sort of the algorithm. This is $O(n \cdot log(n))$.

   An optimization is to use a variation of the build-maxheap as a check to see of the max-heap property. Concretely we start from the last internal node and check if the node is greater than it's child and so on. If this holds through all the iterations then by definition the array is a max-heap. The worst cost is to do the checks until the root and root turns out out to be smaller then it's children.. This require $\lfloor \frac{n}{2} \rfloor$ iterations in the worst case and is thus $O(n)$. In the best case this algorithm is $\Omega(1)$ as when the last internal node does not satisfy the max-heap property.

2. Write the pseudocode for the best algorithm you came up with.

Algorithm 1 valid max-heap

prodedure(A) :

```
1: n = A.length
2: for(i = ⌊n/2⌋ ;i >= 0 ;i−−) :
3:     left_child = 2 · i + 1
4:     right_child = 2 · i + 2
5:     if left_child < n and A[left_child] > A[i] :
6:         return false
7:     if right_child < n and A[right_child] > A[i]:
8:         return false
9: return true
```

3. Implement your answer using any programming language you want to.

```
1  bool is_max_heapv1(const vector<int> &A){
2      for(int i = (A.size() - 1) / 2;i >= 0;i--){
3          int left_child_idx = 2 * i + 1;
4          if(left_child_idx < A.size() && A[i] < A[
               left_child_idx])
5              return false;
6          int right_child_idx = 2 * i + 2;
7          if(right_child_idx < A.size() && A[i] < A[
               right_child_idx])
8              return false;
9      }
10     return true;
11 }
```

4. What is the time complexity of you answer?

   We analyze the pseudo-code. Constant amount of work is done in every iteration of the for loop, we denote this constant c. In the worst-case all internal nodes are inspected for the max-heap property, this requires $\lfloor \frac{n}{2} \rfloor$ iterations. We obtain the following

$$\underbrace{c}_{work\ per\ iteration} \cdot \underbrace{\lfloor \frac{n}{2} \rfloor}_{worst\ case\ \#\ of\ iterations} = O(n)$$

   Which shows the following algorithm is $O(n)$. A similar analysis in the best case is to see if that the last internal node does not satisfy the max-heap property we immediately return false. This is $\Omega(1)$ time.

2. A sorted array of size n and a random array of size k are given to you where $k < n$. Develop an algorithm to decide whether the smaller random array is a subset of the larger sorted array.

1. How would you decide if b is a subset of a?

   By definition of subset, b is a subset of a if and only if all elements of b are in a. we can thus decompose this problem into checking if all elements of b are in a.
   The brute-force algorithm then is to do a linear search for all of b's element and check if they are in a. We record successful searches with a counter. if the counter is the same size as the size of b then every element of b was found and by definition is a subset of a. Since the size of b is k and the size of a is n and we do a linear search for all of $b's$ elements this is $O(n \cdot k)$. When $k = 1$ this reduces down to the problem of one linear search and is $O(n)$. In the worst case where $k = n - 1$ this is $O(n^2)$. We could optimize this algorithm further by using the sorted property of a to do binary searches instead of linear searches. This still requires k iterations and is thus $O(k \cdot log(n))$. A similiar analysis to the previous one would result in $O(log(n))$ when $k = 1$ and $O(n \cdot log(n))$ when $k = n - 1$.
   We further analyze and use a variation of the counting sort algorithm. We get the maximum value, call it q, in a in $\Theta(1)$ since a is sorted. We then allocate a boolean array, denoted c, of size $q + 1$ with values false, this is $\Theta(q)$. We iterate through the elements of b, denoted e, and set $q[e] = true$. This is $\Theta(k)$. We now iterate through the elements of a,denoted e, and increment a counter every time $q[e] == true$. This $\Theta(n)$. This algorithm running time is $\Theta(n + k + q)$. This algorithm is worst than the brute-force depending on the value of q.
   We could optimize this variation by using a hash-table instead of a boolean array. Concretely we insert all of $b's$ element into a hashtable, this is on average $O(k)$. We then mantain a counter and increment it every time one of $a's$ element is in the hashtable. This is also on average $O(n)$. Thus the

running times is $O(n+k)$ where since $k = n-1$ in the worst case reduces to $O(n)$. In the case when $k = 1$ the binary search algorithm is preferable while in the case $k = O(n)$ the hashtable is optimal. We conclude the hashtable algorithm is the most optimal given the problem constraints.

2. Write the pseudocode for the best algorithm you came up with.

Algorithm 2 is_subset

prodedure(A,B) :

```
1: set = {}
2: for e in B:
3:     set = set ∪ e
4: counter = 0
5: for e in A:
6:     if e in set:
7:         counter++
8: return counter == B.lengths
```

3. Implement your answer using any programming language you want to.

```
#include <iostream>
#include <deque>
#include <unordered_set>
#include <vector>
#include <algorithm>

using namespace std;

bool is_subsetv1(const vector<int>& A, const vector<int>& B){
    unordered_set<int> set(B.cbegin(), B.cend());
    int counter = 0;
    for(int e : A){
        if(set.count(e)){
            counter++;
        }
    }
    return counter == B.size();
}
```

4. What is the time complexity of your answer?

   We analyze the pseudo-code. Lines 2-3 insert all of $b's$ element into the hashtable. An insertion is on average $O(1)$ time and since we iterate through b this is overall $O(k)$. lines 5-7 iterate through a and check if an element of a is in the set if so increment the counter. The cost of the operations in the loop is $O(1)$ and since we iterate through all of a's element this is $O(n)$. we conclude that the total running time is $O(n+k)$. This reduces to $O(n)$ since $k = O(n)$.