Solutions to ADS practice questions: Sorting and heaps

Question 1.

How many key comparisons does Mergesort do when the array is sorted at the start?

Solution

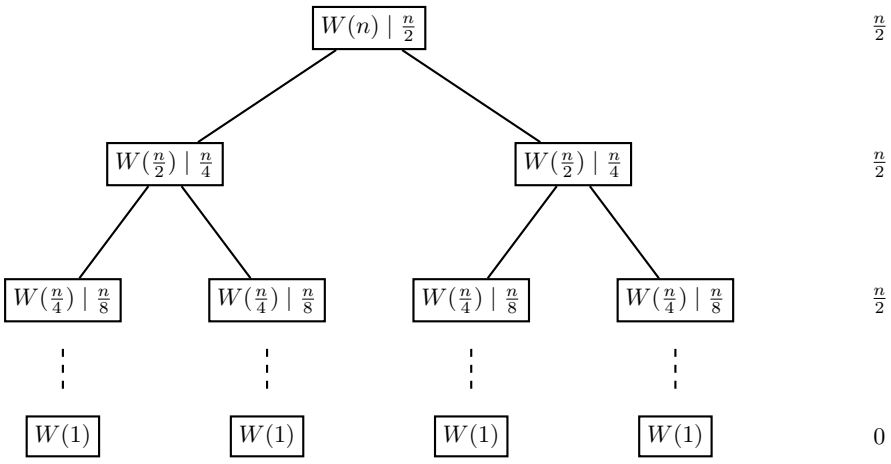
Dividing the array makes no change, but merging two sorted lists is faster when the left list only contains elements that are smaller than all elements in the right list.

This is because two lists are generally merged as follows: take two variables that initially point to the first element of the left list and the first element of the right list. These two elements are compared and the smallest element is placed first. The pointer of the list that contained the smallest element is moved one position to the right. Continue until one of the two lists has been completed: the other list can then be pasted under the result.

When the array is initially already sorted then on the merge all elements in the left list will be smaller than all elements in the right list. The entire left list is then sorted after just $\frac{n}{2}$ comparisons when n is even, or $\frac{n+1}{2}$ comparisons when n is uneven. So the number of comparisons for an array of n elements is:

$$W(n) = \begin{cases} 0 & \text{als } n = 1 \\ W(\lceil \frac{n}{2} \rceil) + W(\lfloor \frac{n}{2} \rfloor) + \lceil \frac{n}{2} \rceil & \text{als } n > 1 \end{cases}$$

When it is assumed that n is a power of 2 then the recursive tree is as follows:



As there are ${}^2\log n$ evels (excluding leaves), each with cost $\frac{n}{2}$, the total is:

$$W(n) = \frac{n}{2} \cdot {}^2\log n$$

(Note than the cost of an unsorted list can increase to a maximum of $n \cdot {}^2\log n$.)

Question 2.

(Dijkstra's 'Dutch Flag' Problem:) Each of the n elements of an array can have the value *red*, *white* or *blue*. Give an algorithm that sorts the array so that it lists all red, all white and then all blue elements (it is possible that a colour is not included). The only operations allowed are check colour and swap two colours (on the basis of their indices). What is the worst case complexity? There is a linear solution!

Solution

The trick lies in the division of the array into four sections, beginning with a section with red keys followed by sections with white, as yet unknown (random) and, finally, blue keys.

The algorithm is:

```
def sortVlag(E):

    n = E.length         # The length of the array
    r = -1               # The index of the last red key
    o = 0                # The index of the first unknown key
    b = n                # The index of the first blue key

    while o < b:
        if E[o] == red:               # When the following unknown key is red,
            E[r+1],E[o]=E[o],E[r+1]   # move E[o] to the red area
            r=r+1                     # and adjust r and o accordingly
            o=o+1
        else: if E[o] == white:       # OK, E[o] is already in the right place
                  o=o+1
              else:                   # The key is blue, so move it to the bottom
                  E[b-1],E[o]=E[o],E[b-1]
                  b=b-1               # and adjust b accordingly.
```

As either `o` is increased or `b` is decreased in each iteration, the number of iterations is equal to n. As a maximum of 3 perations are performed in each iteration, the worst case time complexity of this algorithm is $\Theta(n)$. The spatial complexity is constant, as no extra array is required (all operations are performed in place).

Question 3.

Given an unsorted array of n integers.

1. Give an algorithm that checks whether the array contains two equal elements. What is the worst case complexity?
2. Suppose that it is known that the elements lie in the range 1 to $2n$. Give an algorithm with linear worst case complexity.

Solution

1. The most obvious algorithm simply performs a check for each index to determine whether another larger index contains the same number:

```
def hasDuplicates(A):
    n = len(A)
    for i in range(0,n):
        for j in range(i+1,n):
            if A[i] == A[j]:
                return True;
    return False
```

The complexity of this algorithm is $O(n^2)$. This can readily be concluded from the observation that the outside for loop is called a maximum of $n-1$ times and the inside loop $n-i-1$ times. In each iteration of the inside loop 1 comparison is done. This

yields:

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

However, it would be smarter to begin by sorting the array (which can be performed, for example, with merge sort in $O(n \log n)$), and then comparing each element with the element to its immediate right (which is $O(n)$). So this is $O(n \log n)$, and, consequently, is better than the above solution.

2. In this case it is possible to maintain an array that keeps count of the number of appearances of each potential number. The algorithm is:

```
def hasDuplicates(A):

    n = len(A)
    count = [0]*(2*n)   #initialize array count

    for i in range(0,n):
        count[A[i]] += 1
        if count[A[i]] > 1:
            return True
    return False
```

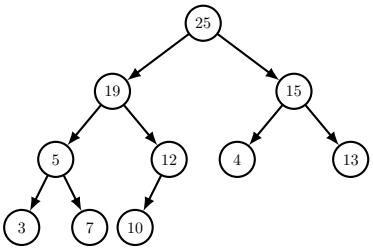
The worst case time complexity of this algorithm is $O(n)$ (which is also the spatial complexity).

Question 4.

Given an array 25, 19, 15, 5, 12, 4, 13, 3, 7, 10. Does this array represent a heap?

Solution

Drawing the associated binary tree for this array yields the following diagram:



As $25 > 19$, this could only be a max heap. However, as the heap property is not satisfied because 5 is above 7, the array does not represent a heap.

Question 5.

Suppose that an array initially contains the following sequence of letters: C O M P L E X I T Y. Show the array that results after it has been built into a heap (using the *buildHeap* algorithm).

Solution

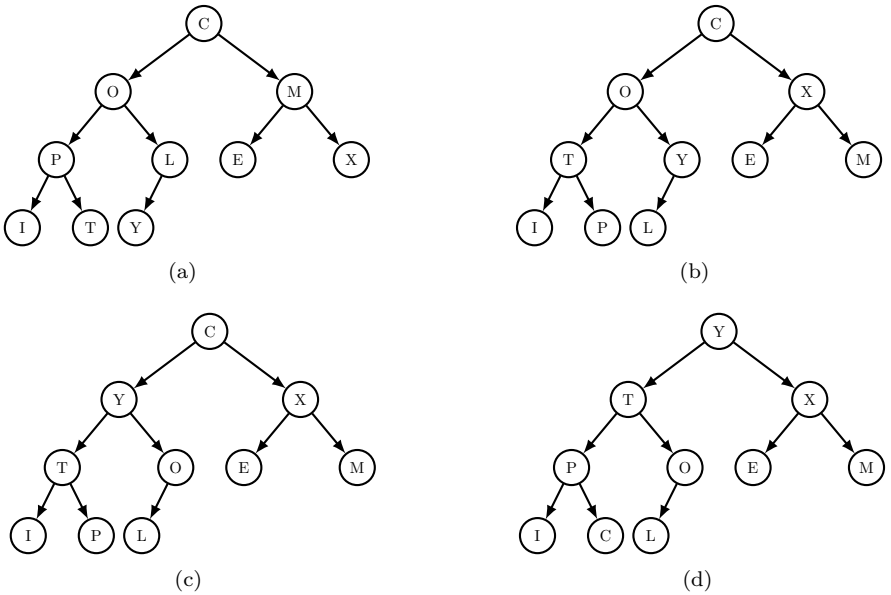
The initial heap and the final heap, as well as two intermediate heaps, are shown in Figure 1.

Question 6.

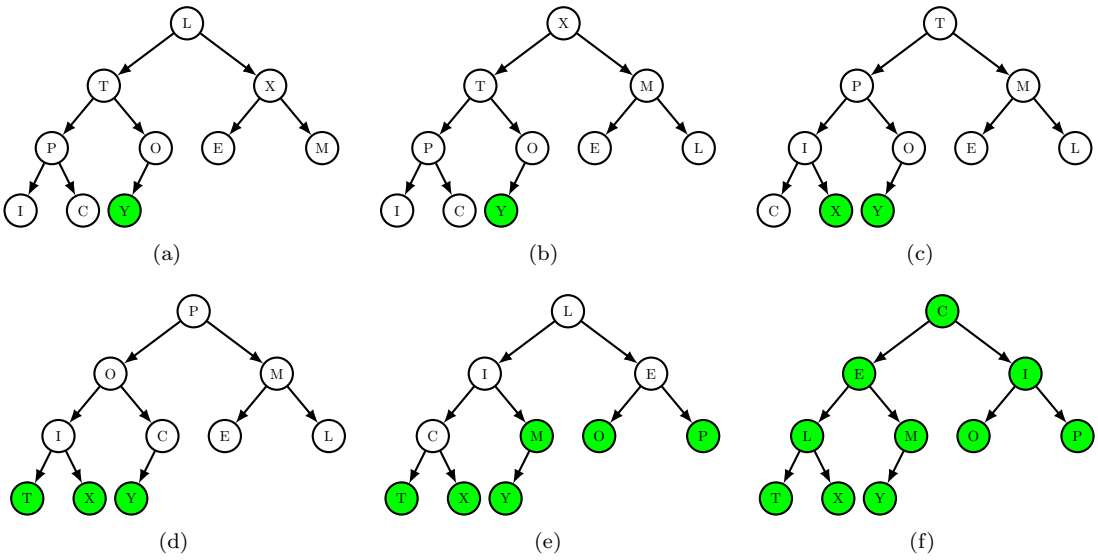
Use the result from the above question as input for the *heapSort* algorithm.

Solution

The initial heap and the final heap, as well as four intermediate heaps, are shown in Figure 2. Obviously, it would have been easy to forecast the result, as the elements are in alphabetical order.



Figuur 1: A number of intermediate heaps from buildHeap.



Figuur 2: A number of intermediate heaps from heapSort.

Question 7.

An array containing 10 different elements is sorted in decreasing order (so the first element is the largest).

1. How many comparisons need to be done when building a heap from this array?
2. How many comparisons need to be done when this array contains n elements?
3. Is an array of this nature a worst case, best case or average case, or none of these cases?

Solution

1. Each key in the tree is checked 1 time to determine whether it needs to be swapped with its parent. As there are 10 nodes and every node except the root has a parent, this will require 9 comparisons.
2. From the above argument it follows that $n - 1$ comparisons will be required.
3. As none of the keys needs to be swapped, this is the best case.