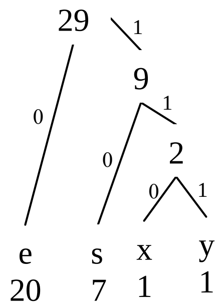
Practical Session 10 - Huffman code, Sort properties, QuickSort algorithm

Huffman Code

	Huffman coding is an encoding algorithm used for lossless data compression, using a priority queue.
Algorithm Description	Given data comprised of symbols from the set C (C can be the English alphabet, for example), Huffman code uses a priority queue (Minimum Heap based on symbol frequency) to assign encodings to the different symbols in the. The algorithm builds a binary tree (the Huffman tree) whose leafs are the elements of C. Every symbol in C is associated with a leaf in the Huffman tree. The binary encoding of a symbol is as long as the depth of the leaf associated with it, and contains a *0* bit for every left move and a *1* bit for every right move on the path from the root to that leaf.
Example	Example Huffman tree with 4 symbols (C={e,s,x,y}) Numbers signify symbol frequency. Encoding: e: 0 s: 10 x:110 y: 111 

	Huffman (C)
	    n ← |C|
	    Q ← { new priority queue for the letters in C }
	    for i ← 1 to n-1
	        z ← allocate new node
	        x ← Extract_Min(Q)
	        y ← Extract_Min(Q)
	        z.left ← x
	        z.right ← y
	        frequency (z) ← frequency (x) + frequency (y)
	        Insert(Q, z)

Question 1

A. What is the optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

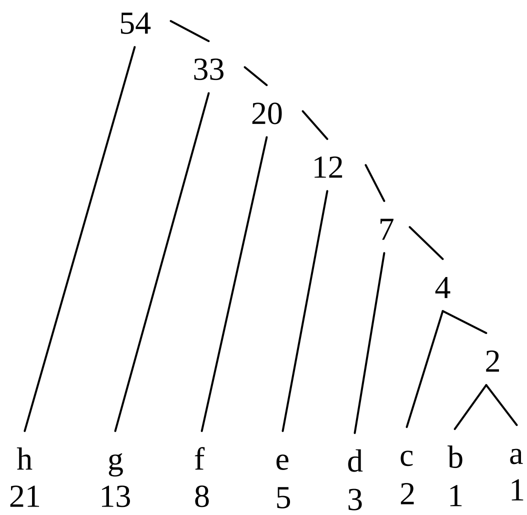
a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

B. Generalize your answer to find the optimal code when the frequencies are the first *n* Fibonacci numbers, for a general *n*.

Solution:

A. Since there are 8 letters in the alphabet, the initial queue size is n = 8, and 7 merge steps are required to build the tree. The final tree represents the optimal prefix code. The codeword for a letter is the sequence of the edge labels on the path from the root to the letter. Thus, the optimal Huffman code is as follows:

h :	0						
g :	1	0					
f :	1	1	0				
e :	1	1	1	0			
d :	1	1	1	1	0		
c :	1	1	1	1	1	0	
b :	1	1	1	1	1	1	0
a :	1	1	1	1	1	1	1



B. As we can see the tree is one long limb with leaves n=hanging off. This is true for Fibonacci weights in general, because the Fibonacci the recurrence is implies that

$$F_{n+2} = \sum_{i=0}^{n} F_i + 1$$

We can prove this by induction. The numbers 1,1,2,3 provide a sufficient base.
We assume the equality holds for all Fibonacci numbers smaller than F_{n+2}.
Step: We prove correctness for F_{n+2}:

$$F_{n+2} = F_{n+1} + F_n = \sum_{i=0}^{n-1} F_i + 1 + F_n = \sum_{i=0}^{n} F_i + 1$$

Therefore $F_{n+2} < \sum_{i=0}^{n} F_i + 1$ and clearly $F_{n+2} < F_{n+1}$ so F_{n+2} is chosen after all smaller Fibonacci numbers have been merged into a single tree.

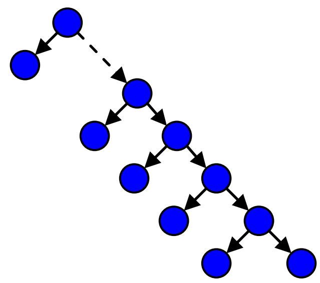
Question 2

A. Given the frequency series for a Huffman code as follows:

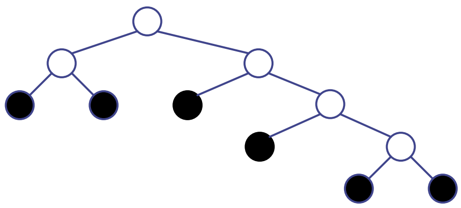
$$f_i = \begin{cases} 4 & i = 1 \\ 2^i & i \in \{2..n\} \end{cases}$$

Draw the structure of the Huffman Tree that describes this series.

Solution A:

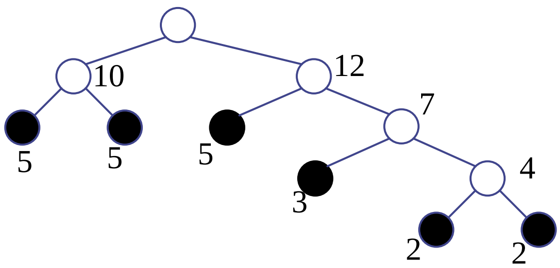
Explanation	tree diagram
on each level of the tree, f_j can be written as: $f_j = f_1 + f_2 + \ldots + f_{j-1}$ Therefore, on each level we will choose the node with the root of the subtree of $f_1 --- f_{i-1}$ created before, and we will get the tree in the diagram	

B. Write a frequency list that the Huffman code of this frequency would deterministically create the following structure.

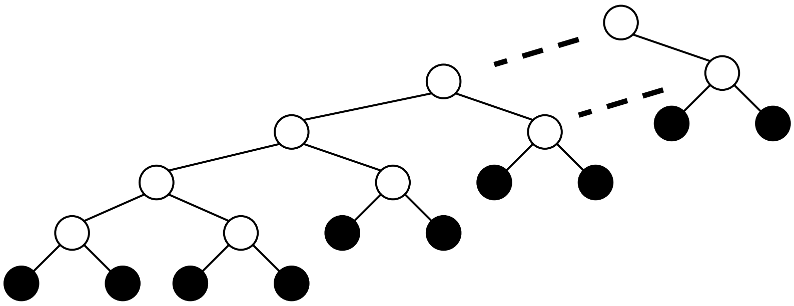


Solution B:

Frequencies: 2,2,3,5,5,5



C. Write a frequency formula the Huffman code of this frequency would deterministically create the following structure.



Solution C:

In order to create this structure, we want that the next two elements on the series will be chosen before the unification of the existent subtree. The pattern of the series is based on the principle that on each level the frequency of each of the next two elements is smaller than the sum of the frequencies till now.

The following recurrence formula that satisfy this quality:

$$f_i = \begin{cases} \text{some constant } c \geq 1 & i \leq 4 \\ f_{i-1} & i \text{ is even} \\ \left(\sum_{1 \leq j \leq i-1} f_j \right) - 1 & \textit{otherwise} \end{cases}$$

An example of the function f that creates the series:

$= \langle 1,1,1,1,3,3,9,9,27,27,81,81,\ldots \rangle$ is:

$$f_i = \begin{cases} \text{some constant } c \geq 1 & i \leq 4 \\ f_{i-1} & i \text{ is even} \\ 3^{\left(\frac{i-3}{2}\right)} & \textit{otherwise} \end{cases}$$

Quicksort

```
quickSort( A, low, high )
    if( high > low )
        pivot ← partition( A, low, high )   //
        quickSort( A, low, pivot-1 )
        quickSort( A, pivot+1, high )
```

```
int partition( A, low, high )
    pivot_value ← A[low]
    left ← low
    pivot ← left
    right ← high
    while ( left <  right )

        // Move left while item <  pivot
        while( left < high && A[left] ≤ pivot_value)
            left++

        // Move right while item >  pivot
        while( A[right] > pivot_value)
            right--

        if( left < right )  Make sure right has not passed left
            SWAP(A,left,right)

    // right is final position for the pivot
    A[low] ← A[right]
    A[right] ← pivot_item
    return right
```

quickSort(A,0,length(A)-1)

- **stable sorting algorithms**: maintain the relative order of records with equal keys
- **in place algorithms**: need only O(log N) extra memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored
- **QuickSort version above is not stable.**

Question 3

Given a multi-set S of n integer elements and an index k ($1 \leq k \leq n$), we define the k-smallest element to be the k-th element when the elements are sorted from the smallest to the largest.

Suggest an O(n) on average time algorithm for finding the k-smallest element.

Example:
For the given set of numbers: $\{6, 3, 2, 4, 1, 1, 2, 6\}$
The 4-smallest element is 2 since in the 2 is the 4'th element in the sorted set $\{1, 1, 2, 2, 3, 4, 6, 6\}$.

Solution:

The algorithm is based on the Quick-Sort algorithm.

```
Quick-Sort :  //Reminder
quicksort(A,p, r)
  If (p<r)
     q ← partition(A,p,r)       // Partition into two parts in Θ(r − p) time.
     quicksort(A,p,q-1)
     quicksort(A,q+1,r)
```

```
Select(k, S) // returns k-th element in S.
   pick x in S
   partition S into: // Slightly different variant of partition()
      max(L) < x, E = {x}, x < min(G)
   if k ≤ length(L) // Searching for item ≤ x.
      return Select(k, L)
   else if k ≤ length(L) + length(E) // Found
      return x
   else  // Searching for item ≥ x.
      return Select(k - length(L) - length(E), G)
```

In the worst case: the chosen pivot x is the maximal element in the current array and there is only one such element. G is empty $length(E) = 1$ and $length(L) = length(S) - 1$

$$T(l) = \begin{cases} T(l-1) + O(l) & l < k \\ k & l = k \end{cases}$$

The solution of the recursive equation: $T(n) = O(n^2)$

In the average case: similar to quick-sort, half of the elements in S are good pivots, for which the size of L and G are each less than $3n/4$.
Therefore, $T(n) \leq T(3n/4) + O(n) = O(n)$, (master theorem, case c).

Question 4

Given an array of n numbers, suggest an $\Theta(n)$ **expected time** algorithm to determine whether there is a number in A that appears more than $\boldsymbol{n/2}$ times.

Solution:

If *x* is a number that appears more than $\boldsymbol{n/2}$ times in A, then *x* is the $(\lfloor \boldsymbol{n/2} \rfloor + \mathbf{1})$-smallest in the array A.

```
Frequent (A,n)
   x ← Select ((⌊n/2⌋ + 1), A)   // find middle element
   count ← 0
   for  i ← 1 to n do:  // count appearances of middle element
       if (A[i] = x) count ++
   if  count > n/2
   then return TRUE
   else return FALSE
```

Time Complexity:
In the mean case, *Select* algorithm runs in $\Theta(n)$.
Computing *count* takes $\Theta(n)$ as well.
Total run time in the mean case: $\Theta(n)$

Question 5

n records are stored in an array A of size n.
Suggest an algorithm to sort the records in O(n) (time) and **no additional space** in each of the following cases:

I. All the keys are 0 or 1
II. All the keys are in the range [1..k], k is **constant**

Solution:

- **I.** Use Quicksort's partition method as we did in question 4 with pivot 0. After the completion of the partition function, the array is sorted (L={}, E will have all elements with key 0, G will have all elements with key 1). Time complexity is $\Theta(n)$ – the cost of one partition.
- **II.** First, partition method on A[1..n] with pivot 1, this way all the records with key 1 will be on the first x_1 indices of the array.
  Second, partition method on A[x_1+1,..,n] with pivot 2
  ...
  After k-1 steps A is sorted
  Time complexity is O(kn)=O(n) – the cost of k partitions.

Question 6

Given the following algorithm to sort an array A of size n:

1. Sort recursively the first 2/3 of A (A[1..2n/3])
2. Sort recursively the last 2/3 of A (A[n/3+1..n])
3. Sort recursively the first 2/3 of A (A[1..2n/3])

* If (2/3*n) is not a natural number, round it up.

Prove the above algorithm sorts A and find a recurrence T(n), expressing it's running time.

Solution:
The basic assumption is that after the first 2 steps, the n/3 largest number are in their places, sorted in the last third of A. In the last stage the algorithm sorts the left 2 thirds of A.

$$T(n) = 3T\left(\frac{2n}{3}\right) = 3\left(3T\left(\frac{4n}{9}\right)\right) = 3\left(3\left(3T\left(\frac{8n}{27}\right)\right)\right) = \ldots = 3^i T\left(\left(\frac{2}{3}\right)^i n\right)$$

after i=$\log_{\frac{3}{2}} n$ steps ...

$$\begin{aligned} T(n) &= 3^{\log_{3/2} n} \cdot T(1) = 3^{(\log_3 n)/\log_3(3/2)} \\ &= \left(3^{\log_3 n}\right)^{1/(\log_3(3/2))} = n^{1/(\log_3(3/2))} \\ &= n^{(\log_3 3)/(\log_3(3/2))} = n^{\log_{3/2} 3} \end{aligned}$$

T(n) = O($n^{\log_{3/2} 3}$) , (also according to the Master-Theorem)

Question 7

Given an array A of M+N elements, where the first N elements in A are sorted and the last M elements in A are unsorted.

1. Evaluate the run-time complexity in term of M and N in the worst case, of fully sorting the array using insertion sort on A?

2. For each of the following cases, which sort method (or methods combination) would you use and what would be the run-time complexity in the worst case?

   a) M = O(1)
   b) M = O(logN)
   c) M = O(N)

Solution:

1. O(M(M+N))
   The last M elements will be inserted to their right place and that requires N, N+1, N+2,...,N+M shifts (in the worst case), or $O(M^2 + N)$ if we apply insertion sort to the last M elements and then merge.
2.
   a. Insertion-Sort in O(N)
   b. Use any comparison based sort algorithm that has a runtime of O(MlogM) (Such as merge sort) on the M unsorted elements, and then merge the two sorted parts of the array in O(M + N). Total runtime: O(MlogM + N) = O(N)
   c. Use any efficient comparison based sort algorithm for a runtime of O((M+N)log(M+N))=O(NlogN). Quick-Sort is bad for this case, as its worst case analysis is $O(n^2)$.

Question 8

How can we use an unstable sorting (comparisons based) algorithm U (for example, quick-sort or heap-sort) to build a new stable sorting algorithm S with the same time complexity as the algorithm U?

Solution 1:

U is a comparisons based sorting algorithm, thus it's runtime $\boldsymbol{T_U(n) = \Omega(n \log n)}$.

1. Add a new field, ***index***, to each element. This new field holds the original index of that element in the unsorted array.
2. Change the comparison operator so that:
   $[key_1, index_1] < [key_2, index_2] \Leftrightarrow key_1 < key_2$ or
   $(key_1 = key_2$ and $index_1 < index_2)$
   $[key_1, index_1] > [key_2, index_2] \Leftrightarrow key_1 > key_2$ or
   $(key_1 = key_2$ and $index_1 > index_2)$
3. Execute the sorting algorithm U with the new comparison operation.

<u>Time complexity:</u>
adding an index field is O(n), the sorting time is the same as of the unstable algorithm, $T_U(n)$, total is $T(n) = T_U(n)$ (as $T_U(n) = \Omega(n \log n)$).

Solution 2:

1. Add a new field, ***index***, to each element in the input array A – O(n). This new field holds the original index of that element in the input.
2. Execute U on A to sort the elements by their key – $T_U(n)$
3. Execute U on each set of equal-key elements to sort them by the ***index*** field – $T_U(n)$

Time complexity of phase 3: assume we have *m* different keys in the input array($1 \leq m \leq n$), n_i is the number of elements with key k_i, where $\boldsymbol{1 \leq i \leq m}$ and $\boldsymbol{\sum_{i=1}^{m}(n_i) = n}$. That is, the time complexity of phase 3 is:

$$\boldsymbol{T(n) = \sum_{i=1}^{m} T_U(n_i) = \sum_{i=1}^{m} \Omega(n_i \log n_i)}$$

In the worst case all keys in the array are equal (i.e., m=1) and the phase 3 is in fact sorting of the array by index values: $\boldsymbol{T(n) = T_U(n) = \Omega(n \log n).}$

Time complexity (for entire algorithm): $\boldsymbol{T(n) = 2T_U(n) + O(n) = O(T_U(n))}$.