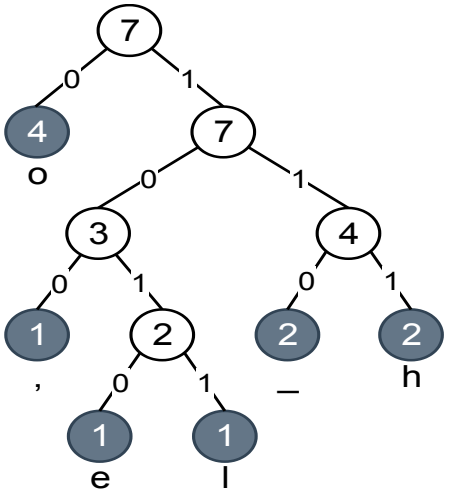
Lecture 2 exercise solutions

1.

Frequency table:

o	h	_	,	l	e
4	2	2	1	1	1

An optimal coding trie (the result of the Huffmans algorithm):



The “oho_ho,_ole” is coded as: 0111011011101001100101111010

CLRS3 16.3-3.

The coding trie, when the frequencies follow the Fibonacci sequence, has a “linear” structure: one child of each internal node is a leaf, except for the deepest internal node, which has two leaf children. Thus the lengths of codes of n characters are as follows: n-1, n-1, n-2, n-3, …, 1. In the given example:

Total length of text: 54.

Bits in fixed length coding: 54*3 = 162.

Bits in Huffman coding: 7 + 7 + 2*6 + 3*5 + 5*4 + 8*3 + 13*2 + 21*1 = 132

2.

Structure of **Dijkstra**'s:

Problem: A weighted graph $G = (V, E)$ and a starting vertex s. Find a shortest path tree of G starting at s.

Greedy choice: Among incident edges of s, choose an edge (s, u) with a minimum weight (but see below how these wights are defined in subproblems!)

Remaining *subproblem*:

$G' = (V', E')$, where $V' = V \setminus \{s, u\} \cup \{s'\}$, $E' = E \setminus \{(s, u)\}$, but with all the edges incident on s or u made

incident on s' (see the figure on the last slide). The weights of these new edges (s', v) in E' is $w(s', v) = \min(w(s, v), w(s, u) + w(u, v))$. In this formula, if either (s, v) or (u, v) is not in E, then we consider its weight infinite.

The starting vertex s'. Find a shortest path tree of G' starting at s'.

The idea is that we merged the two vertexes s and u into one pseudo-vertex s'. This vertex corresponds to the already computed shortest-path tree to which we keep merging new vertices.

3. CLRS3 16.2-5

First, we sort the points (having an order is usually a good idea in solving most problems). The greedy choice is then to make the first interval to start at the *leftmost point*. The intuition is clear. We have to cover this point anyway. So while we are doing it, we try to cover as many points to the right as possible. Then, repeat starting from the next not covered point. The main part of the algorithm runs in $\Theta(n)$ time. The total running time is dominated by sorting: $\Theta(n \lg n)$.

Proof of the greedy choice property: Let us assume S is an optimal solution that *does not* include an interval starting at x_l – the leftmost point. Then, let a be *the interval in S with the leftmost starting point.* The starting point of a will be smaller than x_l, because x_l have to be covered in any solution. We can modify S, by shifting a to start at x_l ($S' = S \setminus \{a\} \cup \{[x_l, x_l+1]\}$). This is still a valid solution ($[x_l, x_l+1]$ covers all the points that a was covering plus possibly some more) and has the same number of intervals. Thus we have shown that the greedy choice belongs to an optimal solution.

We also have to show the optimal substructure to complete the proof of the correctness of the algorithm, but that is quite obvious for this problem (Try to formulate a cut-and-paste argument. It is analogous to the activity selection problem).

4. CLRS3 16.1-4

Start with zero lecture halls allocated and take activities one by one.

The greedy choice is to consider as the next activity the one with the smallest starting time. If one of the allocated lecture halls is free, we take it and schedule that activity in it, otherwise we allocate an additional hall and schedule the activity there. The algorithm is intuitive: we consider where to schedule an activity only *when* it has to start.

To prove that the algorithm allocates optimally minimal number of lecture halls, let us say that the algorithm allocates k lecture halls. Now consider the first activity (x) scheduled on the k-th lecture hall. At the starting time x_s of this activity x, there are k-1 other activities that have started but have not yet finished (otherwise, one of the other lecture halls would have been free at s and we would have taken it instead of allocating a new lecture hall). Together with x, these activities form a set of k overlapping activities, which means that it is impossible to schedule them in fewer than k lecture halls.

By the way, the time can be reversed, running "the movie" backwards: we would take activities sorted on the decreasing finishing time, instead of increasing starting time.

```
// Array A is sorted on A[i].s. Lecture hall # will be saved in A[i].h
ActivityScheduler(A[1..n])
01 d.f = 0      // dummy activity that starts before all activities
02 d.h = 1      // ...and is in lecture hall 1
03 Q.insert(d) // Q: a min-priority queue on finish times
04 k = 1        // lecture halls used
05 for i =1 to n do
06    a = Q.Min()        // without extracting
07    if a.f < A[i].s then
08       Q.extractMin()  // a is removed from Q
09       A[i].h = a.h    // use the same lecture hall as a
10    else
11       k = k + 1       // allocate a new lecture hall
12       A[i].h = k
13    Q.insert(A[i])
14 return k
```

The running time of the algorithm is dominated by sorting on starting times before calling the alg: $\Theta(n \lg n)$. The main loop takes activities one by one in this order and schedules them. It runs in $\Theta(n \lg n)$ by maintaining a min-priority queue on the ending times of the currently running activities. In an iteration of the main loop, when we need to start a new activity *A[i]* with a starting time *A[i].s*, we check the activity *a* at the top of the queue with the lowest ending time *a.f*. If $a.f < A[i].s$, we can schedule *A[i]* in the same lecture hall as *a*, else we need to allocate a new lecture hall. In both cases, we push *A[i]* to the queue.