Exercise 15.1-5

15.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i. The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, the goal is to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Sol)

We can no longer use the greedy algorithm to solve this problem. However, as we show, the problem still has an optimal substructure which allows us to formulate a dynamic programming solution. The analysis here follows closely the analysis of Section 15.1 in the book. We define the value of a set of compatible events as the sum of values of events in that set. Let S_{ij} be defined as in Section 15.1. An *optimal solution* to S_{ij} is a subset of mutually compatible events of S_{ij} that has maximum value. Let A_{ij} be an optimal solution to S_{ij}. Suppose A_{ij} includes an event a_k. Let A_{ik} and A_{kj} be defined as in Section 15.1. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the value of maximum-value set A_{ij} is equal to the value of A_{ik} plus the value of A_{kj} plus v_k.

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where the value of A'_{kj} is greater than the value of A_{kj}, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. We would have constructed a set of mutually compatible activities with greater value than that of A_{ij}, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

Let us denote the value of an optimal solution for the set S_{ij} by $value[i, j]$. Then, we would have the recurrence

$$value[i, j] = value[i, k] + value[k, j] + v_k \ .$$

Of course, since we do not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$value[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \ , \\ \max \{value[i, k] + value[k, j] + v_k : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset \ . \end{cases}$$

While implementing the recurrence, the tricky part is determining which activities are in the set S_{ij}. If activity k is in S_{ij}, then we must have $i < k < j$, which means that $j - i \geq 2$, but we must also have that $f_i \leq s_k$ and $f_k \leq s_j$. If we start k at $j - 1$ and decrement k, we can stop once k reaches i, but we can also stop once we find that $f_k \leq f_i$, since then activities $i + 1$ through k cannot be compatible with activity i.

We create two fictitious activities, a_0 with $f_0 = 0$ and a_{n+1} with $s_{n+1} = \infty$. We are interested in a maximum-size set $A_{0,n+1}$ of mutually compatible activities in $S_{0,n+1}$. We'll use tables $value[0 : n + 1, 0 : n + 1]$, as in the recurrence, and $activity[0 : n + 1, 0 : n + 1]$, where $activity[i, j]$ is the activity k that we choose to put into A_{ij}.

We fill the tables in according to increasing difference $j - i$, which we denote by l in the pseudocode. Since $S_{ij} = \emptyset$ if $j - i < 2$, we initialize $value[i, i] = 0$ for all i and $value[i, i + 1] = 0$ for $0 \leq i \leq n$. As in RECURSIVE-ACTIVITY-SELECTOR and GREEDY-ACTIVITY-SELECTOR, the start and finish times are given as arrays s and f, where we assume that the arrays already include the two fictitious activities

and that the activities are sorted by monotonically increasing finish time. The array v specifies the value of each activity.

```
MAX-VALUE-ACTIVITY-SELECTOR(s, f, v, n)
  let value[0:n+1, 0:n+1] and activity[0:n+1, 0:n+1] be new tables
  for i = 0 to n
      value[i, i] = 0
      value[i, i+1] = 0
  value[n+1, n+1] = 0
  for l = 2 to n+1
      for i = 0 to n-l+1
          j = i+l
          value[i, j] = 0
          k = j-1
          while f[i] < f[k]
              if f[i] ≤ s[k] and f[k] ≤ s[j] and
                        value[i, k] + value[k, j] + v_k > value[i, j]
                  value[i, j] = value[i, k] + value[k, j] + v_k
                  activity[i, j] = k
              k = k-1
  print "A maximum-value set of mutually compatible activities has value "
          value[0, n+1]
  print "The set contains "
  PRINT-ACTIVITIES(value, activity, 0, n+1)

PRINT-ACTIVITIES(value, activity, i, j)
  if value[i, j] > 0
      k = activity[i, j]
      print k
      PRINT-ACTIVITIES(value, activity, i, k)
      PRINT-ACTIVITIES(value, activity, k, j)
```

The PRINT-ACTIVITIES procedure recursively prints the set of activities placed into the optimal solution A_{ij}. It first prints the activity k that achieved the maximum value of $value[i, j]$, and then it recurses to print the activities in A_{ik} and A_{kj}. The recursion bottoms out when $value[i, j] = 0$, so that $A_{ij} = \emptyset$.

Whereas GREEDY-ACTIVITY-SELECTOR runs in $\Theta(n)$ time, the MAX-VALUE-ACTIVITY-SELECTOR procedure runs in $O(n^3)$ time.

Exercise 15.3-5

15.3-5

Given an optimal prefix-free code on a set C of n characters, you wish to transmit the code itself using as few bits as possible. Show how to represent any optimal prefix-free code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits. (*Hint:* Use $2n - 1$ bits to specify the structure of the tree, as discovered by a walk of the tree.)

Sol)

A full binary tree with n leaves has $n - 1$ internal nodes, for a total of $2n - 1$ nodes. You can specify the structure of the tree by a preorder walk, with a 1 at a node indicating that it's an internal node and a 0 meaning that the node is a leaf. Because the code is on n characters, $\lceil \lg n \rceil$ bits are needed to represent each character, so that if each leaf represents a character, $n \lceil \lg n \rceil$ bits represent all the characters. Store the characters in the order in which the preorder walk visits the leaves. The total number of bits is then $2n - 1 + n \lceil \lg n \rceil$.

Exercise 15.3-7

15.3-7

A data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Sol)

Let f be the minimum frequency among the 256 characters, so that the maximum character frequency is less than $2f$. The first merge creates an internal node with frequency at least $2f$, so that this internal node won't be selected for merging until all the characters have been merged. The same holds for each of the other 127 merges of characters, so that the first 128 merges create internal nodes with frequencies at least $2f$ and less than $4f$. The next merge merges two such internal nodes, creating an internal node with frequency at least $4f$ and less than $8f$. This internal node won't be selected for merging until all the other internal nodes with frequencies less than $4f$ have been merged. This process continues, always merging internal nodes of the same height, until a single complete binary tree has emerged. The leaves of this tree will all have depth 8, so that all the Huffman codes comprise exactly 8 bits.

Problem 15-2

15-2 Scheduling to minimize average completion time

You are given a set $S = \{a_1, a_2, \ldots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete. Let C_i be the ***completion time*** of task a_i, that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n)\sum_{i=1}^{n} C_i$. For example, suppose that there are two tasks a_1 and a_2 with $p_1 = 3$ and $p_2 = 5$, and consider the schedule

in which a_2 runs first, followed by a_1. Then we have $C_2 = 5$, $C_1 = 8$, and the average completion time is $(5+8)/2 = 6.5$. If task a_1 runs first, however, then we have $C_1 = 3$, $C_2 = 8$, and the average completion time is $(3+8)/2 = 5.5$.

a. Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run nonpreemptively, that is, once task a_i starts, it must run continuously for p_i units of time until it is done. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

b. Suppose now that the tasks are not all available at once. That is, each task cannot start until its ***release time*** b_i. Suppose also that tasks may be ***preempted***, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $b_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and analyze the running time of your algorithm.

Sol)

a. To minimize the average completion time, run the tasks in monotonically increasing order of their processing times.

Suppose the tasks run in the order $a_1, a_2, \ldots, a_n$. Then task a_1 has completion time $c_1 = p_1$, task a_2 has completion time $c_2 = p_1 + p_2$, task a_3 has completion time $c_3 = p_1 + p_2 + p_3$, and so on, so that task a_k has completion time $\sum_{i=1}^{k} p_i$. The average completion time is minimized by minimizing $\sum_{i=1}^{n} c_i$. Noting that

$$\sum_{i=1}^{n} c_i = np_1 + (n-1)p_2 + (n-2)p_3 + \cdots + p_n \ ,$$

we see that this sum is minimized when $p_1, p_2, \ldots, p_n$ are in monotonically increasing order.

b. To minimize the average completion time, run the task that has been released and not yet completed with the shortest time remaining. When a task is released, if its processing time is less than the time remaining for the running task, preempt the running task by the new task, keeping track of the time remaining in the preempted task. A priority queue of ready tasks can determine which task to run next.

One way to think about this situation is that if a task is running while a new task is released, then break the running task into the portion already run and the portion yet to be run. This situation reduces to the situation in part (a).

A little more formally, suppose that at time t, task a_1 has time remaining r_1 and task a_2 has time remaining r_2, where $r_1 < r_2$. The greedy choice is to run a_1 before a_2. If a_1 runs before a_2, then the average completion time for the two tasks is $((t+r_1)+(t+r_1+r_2))/2 = (2t+2r_1+r_2)/2$. If a_2 runs before a_1, then the average completion time for the two tasks is $((t+r_2)+(t+r_2+r_1))/2 = (2t+2r_2+r_1)/2$. Since $r_1 < r_2$, the first way gives a lower average completion time.