Greedy Algorithms Practice problems

1. Determine the time complexity of the following iterative function:

```
int f ( int A[SIZE][SIZE] , int n )
   {
      int i, j, sum = 0;

      for (i=0; i<n; ++i) {
         if (i % 2 == 0)
            for (j=0; j<=i; j=j+1) sum = sum + A[i][j];
         else
            for (j=n-1; j>=i; j=j-1) sum = sum - A[i][j];
      }
   }
```

2. You are given n real intervals (ai ,bi) standing for the running times of n processes. That is, (ai ,bi) stands for a process that starts at time ai and finishes at time bi . Assume that ai < bi for all i. Your objective is to schedule all the processes, using as few processors as possible. You are not allowed to schedule two or more conflicting processes (that is, processes having overlapping running times) on the same processor. A process running in a processor is allowed to continue until it finishes. Propose an efficient greedy algorithm to solve this problem. Supply an optimality proof for your greedy algorithm, and deduce its running time.

Solution We first sort the intervals with respect to their left endpoints. We then try to schedule the intervals one by one in this sorted order. A processor with earliest finish time is chosen for each scheduling. If no existing processor can accommodate a new job, a new processor is introduced. We use a min-priority queue Q to store the right endpoints of the intervals currently scheduled—only one entry per processor. Processors are numbered $1,2,3,\ldots$. Each entry (p,f) in Q stores a processor number p and the finish time f of the last process assigned to p. The heap-ordering is with respect to the second component f.

Sort the given intervals with respect to their left endpoints.
Let the sorted list be $(a_0,b_0),(a_1,b_1),(a_2,b_2),\ldots,(a_{n-1},b_{n-1})$.
Initialize nproc $= 0$, the min-priority queue Q to empty, and $f = a_0 - 1$.
For $i = 0,1,2,\ldots,n-1$ {
    If $(i > 0)$, set $(p,f) = \min(Q)$. /* Q is empty only for $i = 0$ */
    If $(f > a_i)$ {
        Use a new processor: nproc++.
        Set $p =$ nproc.
    } else {
        Make a deleteMin in Q.
    }
    Schedule the i-th process (a_i,b_i) to processor number p.
    Insert (p,b_i) in Q.
}

For the correctness, let x be a real number. The number of intervals (a_i,b_i) to which x belongs is denoted by $N(x)$. Let $N = \max_{x \in \mathbb{R}} N(x)$. We cannot schedule all the processes with less than N processors. The above algorithm clearly uses exactly N processors and is therefore optimal.

The initial sorting of the intervals can be done in $O(n \log n)$ time. Subsequently, there are n deleteMin and insert operations in Q. The maximum size of Q is N, since Q stores only one entry for each processor. So the total time for preparing the schedule (after the sorting phase) is $O(n \log N)$. Finally, $N \leqslant n$, so the overall running time of this greedy algorithm is $O(n \log n)$.

3. Ms. Rotunda is making a long train journey. She can stay without food for four hours. The train does not have a pantry car, so Ms. Rotunda can eat only when the train stops at stations. Given the complete timetable for the train, design an efficient algorithm to identify the stations where Ms. Rotunda would eat so that she never feels hungry throughout the journey, and the number of meals is as small as possible. Assume that she takes her first meal just before the train leaves its source station. Assume also that the train halts at least once in any period of four hours (otherwise, there is no solution to Ms. Rotunda's problem). Neglect the halting times of the train at stations. Prove the correctness of your algorithm, and deduce its running time.

Solution **The algorithm:** Let $S_0, S_1, S_2, \ldots, S_n$ be the stations where the train stops (in that sequence), where S_0 is the source, and S_n the destination. The times t_i (in hours) for the train to go from Station S_{i-1} to Station S_i are also given for $i = 1, 2, \ldots, n$. Neglecting halting times at stations, the time taken by the train to travel from station S_i to S_j (with $j \geqslant i$) is then $t_{i+1} + t_{i+2} + \cdots + t_j$. The following greedy algorithm solves Ms. Rotunda's minimization problem.

```
Print "Take meal at Station 0".
Set lastmeal = 0, i = 0, and fasttime = 0.
While (i ⩽ n) {
    Set fasttime = fasttime + t_{i+1}.
    If (fasttime > 4) {
        Print "Take meal at Station i".
        Set lastmeal = i and fasttime = 0.
    } else {
        Increment i by 1.
    }
}
```

Running time: Under the assumption that each $t_i \leqslant 4$, the loop of the above program runs for at most $2n$ times. Each iteration of the loop takes constant time. So the running time of this greedy algorithm is $\Theta(n)$.

Correctness: Let $0, i_1, i_2, \ldots, i_k$ be an optimal solution to Ms. Rotunda's problem, whereas $0, j_1, j_2, \ldots, j_l$ be the solution produced by the greedy algorithm. Clearly, $i_1 \leqslant j_1$, so $0, j_1, i_2, i_3, \ldots, i_k$ continues to remain an optimal solution. We must have $i_2 \leqslant j_2$, so $0, j_1, j_2, i_3, i_4, \ldots, i_k$ is again an optimal solution. Proceeding in this way, we can convert the optimal solution to the greedy solution without increasing the number of meals. Thus, $k \geqslant l$. But since $0, i_1, i_2, \ldots, i_k$ is an optimal solution, we must have $k \leqslant l$. Therefore, $k = l$, that is, the greedy solution too is optimal.

4. Let G = (V,E) be a directed graph. A vertex s in G is called a source if its in-degree is zero. Likewise, a vertex t in G is called a target (or sink) if its out-degree is zero.

(a) Propose an O(|V|+|E|)-time algorithm to locate all the sources and all the targets in G.
Solution: We assume the adjacency-list representation of the graph. We use two arrays S and T indexed by V to mark whether a vertex can be a source or a target (respectively). Initially, we mark each vertex as a potential source and a potential target. For each (directed) edge (u, v) ∈ E, we unmark u in the target array T, and unmark v in the source array S. After all the edges are considered, those vertices that are still marked in S are the sources, and those vertices that are still marked in T are the targets.

(b) Prove that a directed acyclic graph must contain at least one source and at least one target.
Solution: Assume that a DAG does not contain a source. This means that for every vertex v, there is (at least) an edge (u, v) ∈ E. Let n = |V|. We start with any arbitrary vertex v0, and obtain a sequence of vertices v1, v2, v3,..., vn such that (vi , vi−1) ∈ E for all i = 1,2,3,...,n. Since G contains only n vertices, there must be a repetition in v0, v1, v2,..., vn. Let vi = vj with 0 6 i < j 6 n. By

construction, vj , vj–1, vj–2,..., vi+1, vi is a cycle in G, a contradiction. Analogously, the existence of a target in G can be proved.

(c) Let G = (V,E) be a directed acyclic graph. Propose an O(|V|+|E|)-time algorithm to count the total number of paths from all the sources in G to all the targets in G.

Solution Let $s_1, s_2, \ldots, s_k$ be all the sources and $t_1, t_2, \ldots, t_l$ be all the targets in G (these can be identified in $O(|V|+|E|)$ time by Part (a)). We convert G to a new DAG G' whose vertex set contains two additional vertices s and t. We add the edges (s, s_i) for all $i = 1, 2, \ldots, k$ and also the edges (t_j, t) for all $j = 1, 2, \ldots, l$. G' is a DAG with a unique source s and a unique target t. Moreover, the count of all (s_i, t_j) paths (for all i, j) in G is the same as the count of all (s, t) paths in G'. The size of G' continues to remain $O(|V|+|E|)$.

We make a topological sorting of the vertices in G'. This can be done in $O(|V|+|E|)$ time. Let the listing be $s = v_0, v_1, v_2, \ldots, v_n, t = v_{n+1}$. We use an array C indexed by the vertices in G' to store the count of paths from s to the vertices.

```
Initialize C[v_0] = 1 and C[v_i] = 0 for all i = 1,2,3,...,n+1.
For i = 0,1,2,...,n {
    For all edges (v_i, v_j) in G', set C[v_j] = C[v_j] + C[v_i].
}
Return C[v_{n+1}].
```

Since there are no back edges (that is, edges (v_i, v_j) with $i > j$), the for loop does not miss a path from s to t. With the adjacency list representation of G', this phase can again be finished in $O(|V|+|E|)$ time.

The introduction of the new vertices s, t could have been avoided. In that case, we start by setting $C[s_i] = 1$ for all the sources s_i in G. At the end, we return $C[t_1] + C[t_2] + \cdots + C[t_l]$. However, a topological sorting of G is necessary for the correctness of this algorithm.

Define two functions, f and g, for which $f \notin O(g)$ and $g \notin O(f)$.

SOLUTION

We need only define two function which never dominate each other. For example, define

$$f(n) = \begin{cases} 0 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases} \qquad g(n) = \begin{cases} n & \text{if } n \text{ is odd} \\ 0 & \text{if } n \text{ is even} \end{cases}$$

There are no N and C such that, for all $n \geq N$, $g(n) \leq C \cdot f(n)$ or $f(n) \leq C \cdot g(n)$.

Prove: *If* $f(n) \in O(h(n))$ *and* $g(n) \in O(h(n))$ *then* $f(n) + g(n) \in O(h(n))$.

SOLUTION

If $f(n) \in O(h(n))$ then there exist witnesses N_f and C_f such that

$$\text{For all } n \geq N_f,\ f(n) \leq C_f \cdot h(n)$$

And if $g(n) \in O(h(n))$ then there exist witnesses N_g and C_g such that

$$\text{For all } n \geq N_g,\ g(n) \leq C_g \cdot h(n)$$

Let $N = \max(N_f, N_g)$ and $C = \max(C_f, C_g)$. Then if $n \geq N$,

$$\begin{array}{rcll} f(n) + g(n) & \leq & C_f \cdot h(n) + g(n) & (f \in O(g) \text{ with witnesses } N_f,\ C_f) \\ & \leq & C_f \cdot h(n) + C_g \cdot h(n) & (g \in O(g) \text{ with witnesses } N_g,\ C_g) \\ & \leq & C \cdot h(n) + C \cdot h(n) & (C = \max(C_f, C_g)) \\ & = & 2C \cdot h(n) & \end{array}$$

Thus $f + g \in O(h)$ with witnesses N and $2C$.

Greedy Algorithms

Your institute has a huge event room V_0. Several events are organized there, including classes, tutorials, tests, dramas, chess competitions, sarod recitals, robot design contests, food festivals and social gatherings. Each such event has a start time and an end time. Mr. Sched is in charge of scheduling events in V_0. He has a list of forthcoming events that request the room V_0. Since multiple events cannot run simultaneously in V_0, Mr. Sched's task is to select a non-overlapping set of events. He has two objectives in mind.

1. Schedule as many (non-overlapping) events as possible.
2. Schedule (non-overlapping) events in such a way that V_0 is utilized for the maximum duration.

In this assignment, you write greedy algorithms to solve Mr. Sched's first problem.

More concretely, suppose that there are n intervals (a_i, b_i), where a_i and b_i are respectively the start and end times of the i-th event. Two intervals (a_i, b_i) and (a_j, b_j) are called *compatible* if they do not overlap, that is, if either $a_i \geq b_j$ or $a_j \geq b_i$. If two intervals overlap, we call them *conflicting*. Your task is to select a maximal collection of intervals which are compatible to one another. This is called the **interval scheduling problem**. The following four greedy strategies can be used to solve this problem.

1. Earliest start time first.
2. Earliest end time first.
3. Shortest first.
4. Least conflict first.

Among these, only one is guaranteed to give an optimal solution. The other three may produce sub-optimal solutions. A theoretical proof of optimality will be covered in a tutorial session. In this lab, you experimentally determine which one is the optimal strategy.

- Read the number n of intervals from the user.

- Generate n random intervals. A strategy to generate a random interval is to choose a random left end (start time) in the range 0 to MAX_INTERVAL_LEFT. You then select a random length of the interval in the range 1 to MAX_INTERVAL_LENGTH. The right end (end time) of the interval is calculated by adding the interval length to the left end. For simplicity, you may assume that the left and right ends of intervals (and so their lengths too) are integers. Do not allow intervals of zero length.

- Implement each of the four greedy algorithms. First, sort the list of intervals as needed by the algorithm (with respect to left ends, right ends, interval lengths or numbers of conflicts). Use the merge sort algorithm. Then, make a pass through the sorted list from beginning to end. Select the next interval if and only if it is compatible with the intervals chosen so far. When the sorted list is exhausted, print the intervals selected. Also print the number of intervals selected (Mr. Sched's first problem) and the total length covered by the chosen intervals (the utilization time of V_0—Mr. Sched's second problem—although you are not maximizing it here).

- You may write four different sets of merge sort, merge and scheduler functions. Another alternative is to write only one set of functions each accepting a flag indicating the greedy strategy. You may also use function pointers or templates. But do not worry about these language-specific details. We are interested only in efficient implementations. Your sorting function should run in $O(n \log n)$ time. Scheduling for the first two strategies can be done in $\Theta(n)$ time. The other two may require as bad as $\Theta(n^2)$ time. Calculating the conflicts for all of the n intervals can be finished in $O(n \log n)$ time using a sweep algorithm. For this assignment, a quadratic algorithm is acceptable.

The interval scheduling problem has many applications (other than Mr. Sched's first problem). For example, process scheduling in real-time operating systems has to do with some variants of the interval scheduling problem. There, each process has an arrival time, a deadline and a running time. The task is to schedule a process no earlier than it arrives, in such a way that it finishes no later than the deadline. The objective is to schedule as many processes as possible. In many situations, such scheduling has to be done on-line. There may be additional dependency constraints on the processes, like scheduling some processes requiring some other processes to have been completed. The processes may have priorities. Moreover, the exact running times may be unavailable *a priori*. The scheduling may or may not be preemptive. The processor may need some preparation time between two processes (for example, to run the on-line scheduling algorithm and for task switching). In another variant, one schedules all the processes, but using as few processors as possible. Some of these variants are easy, others potentially difficult.

Sample Output

In the following sample run, I have taken 20 intervals. In my code, MAX_INTERVAL_START and MAX_INTERVAL_LENGTH are `#defined` as 1000 and 100, respectively. Your program may, however, read these too (along with n) from the user. In this example, all of the four strategies give optimal solutions. In the interval listings below, the start and end times are shown along with the numbers of conflicts (within parentheses).

```
n = 20
[ 249, 320]( 2) [ 548, 578]( 3) [ 476, 526]( 2) [ 810, 895]( 3) [ 856, 901]( 3)
[  42,  77]( 2) [ 483, 582]( 5) [ 531, 587]( 4) [ 260, 276]( 2) [  45, 127]( 3)
[ 722, 778]( 1) [ 229, 315]( 3) [  65,  98]( 2) [ 771, 868]( 4) [ 119, 204]( 2)
[ 573, 642]( 3) [ 154, 177]( 1) [ 449, 533]( 3) [ 223, 235]( 1) [ 819, 899]( 3)
```

```
+++ Schedule 1: Earliest start time first
[  42,  77]( 2) [  45, 127]( 3) [  65,  98]( 2) [ 119, 204]( 2) [ 154, 177]( 1)
[ 223, 235]( 1) [ 229, 315]( 3) [ 249, 320]( 2) [ 260, 276]( 2) [ 449, 533]( 3)
[ 476, 526]( 2) [ 483, 582]( 5) [ 531, 587]( 4) [ 548, 578]( 3) [ 573, 642]( 3)
[ 722, 778]( 1) [ 771, 868]( 4) [ 810, 895]( 3) [ 819, 899]( 3) [ 856, 901]( 3)
--- Total number of intervals selected = 8
[  42,  77]( 2) [ 119, 204]( 2) [ 223, 235]( 1) [ 249, 320]( 2) [ 449, 533]( 3)
[ 548, 578]( 3) [ 722, 778]( 1) [ 810, 895]( 3)
Total utilization = 458

+++ Schedule 2: Earliest finish time first
[  42,  77]( 2) [  65,  98]( 2) [  45, 127]( 3) [ 154, 177]( 1) [ 119, 204]( 2)
[ 223, 235]( 1) [ 260, 276]( 2) [ 229, 315]( 3) [ 249, 320]( 2) [ 476, 526]( 2)
[ 449, 533]( 3) [ 548, 578]( 3) [ 483, 582]( 5) [ 531, 587]( 4) [ 573, 642]( 3)
[ 722, 778]( 1) [ 771, 868]( 4) [ 810, 895]( 3) [ 819, 899]( 3) [ 856, 901]( 3)
--- Total number of intervals selected = 8
[  42,  77]( 2) [ 154, 177]( 1) [ 223, 235]( 1) [ 260, 276]( 2) [ 476, 526]( 2)
[ 548, 578]( 3) [ 722, 778]( 1) [ 810, 895]( 3)
Total utilization = 307

+++ Schedule 3: Shortest first
[ 223, 235]( 1) [ 260, 276]( 2) [ 154, 177]( 1) [ 548, 578]( 3) [  65,  98]( 2)
[  42,  77]( 2) [ 856, 901]( 3) [ 476, 526]( 2) [ 531, 587]( 4) [ 722, 778]( 1)
[ 573, 642]( 3) [ 249, 320]( 2) [ 819, 899]( 3) [  45, 127]( 3) [ 449, 533]( 3)
[ 119, 204]( 2) [ 810, 895]( 3) [ 229, 315]( 3) [ 771, 868]( 4) [ 483, 582]( 5)
--- Total number of intervals selected = 8
[ 223, 235]( 1) [ 260, 276]( 2) [ 154, 177]( 1) [ 548, 578]( 3) [  65,  98]( 2)
[ 856, 901]( 3) [ 476, 526]( 2) [ 722, 778]( 1)
Total utilization = 265

+++ Schedule 4: Least conflict first
[ 223, 235]( 1) [ 154, 177]( 1) [ 722, 778]( 1) [ 260, 276]( 2) [  65,  98]( 2)
[  42,  77]( 2) [ 476, 526]( 2) [ 249, 320]( 2) [ 119, 204]( 2) [ 548, 578]( 3)
[ 856, 901]( 3) [ 573, 642]( 3) [ 819, 899]( 3) [  45, 127]( 3) [ 449, 533]( 3)
[ 810, 895]( 3) [ 229, 315]( 3) [ 531, 587]( 4) [ 771, 868]( 4) [ 483, 582]( 5)
--- Total number of intervals selected = 8
[ 223, 235]( 1) [ 154, 177]( 1) [ 722, 778]( 1) [ 260, 276]( 2) [  65,  98]( 2)
[ 476, 526]( 2) [ 548, 578]( 3) [ 856, 901]( 3)
Total utilization = 265
```