<http://cse.iitkgp.ac.in/~abhij/course/theory/Algo1/Autumn13/lab/A2.html>

**Greedy Algorithms**

Your institute has a huge event room V0. 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 V0. He has a list of forthcoming events that request the room V0. Since multiple events cannot run simultaneously in V0, 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 V0 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 (*ai*, *bi*), where *ai* and *bi* are respectively the start and end times of the *i*-th event. Two intervals (*ai*, *bi*) and (*aj*, *bj*) are called *compatible* if they do not overlap, that is, if either *ai* ≥ *bj* or *aj* ≥ *bi*. 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.

Write a C/C++ program to do the following.

* 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 V0—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 Θ(*n*) time. The other two may require as bad as Θ(*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