ILLINOIS INSTITUTE OF TECHNOLOGY
COLLEGE OF SCIENCE
⊢ Ph.D. Physics ⊣

Interval Scheduling on Multiple Resources

CS 430 - Introduction to Algorithms

Professor: Peng-Jun Wan, Ph.D.

Pedro Rivero Ramírez†

Chicago, November 2018

†priveroramirez@hawk.iit.edu

Table of Contents

0. Statement and Rules

Problem Description: Consider m machines and $n \geq m$ jobs, each of which is specified by a start time and a finish time. Each job can be assigned to any machine, but each machine can serve at most one job at any time. The objective is is to schedule a largest number of given jobs to the m machines. Please develop a polynomial time algorithm and write program to implement it.

You may use any language (*e.g.* C/C++/JAVA) to implement; and if the language you use is not supported by the TA's computer, you must use your own computer to demo your program. Your program should be able to accept a file input (*e.g.* TXT file) and you may choose the format of the input file associated with the problem.

Project Report: You are required to submit a project report by the due date to Blackboard which includes:

- algorithm design and pseudocode, a proof of correctness, an analysis of the running time;
- a well commented source code;
- test source data and output;
- a separate README file describing the compiling and the execution of your program.

Project Demo: You are required to demonstrate your program to the TA and answer the questions raised by the TA.

0.1. Project Input and Output Format

Your program should be able to accept an input file called **"input.txt"** and produce an output file called **"output.txt"**. The content of the files will be specified below.

"input.txt" contains $n + 1$ lines. The first line contains the two numbers n and m. The next n lines each contain a starting time s_i and an ending time t_i for the i^{th} job.

"output.txt" contains $m + 1$ lines. The first line give the number k which is the maximum number of jobs you are able to schedule. The next m lines gives the sequences of jobs scheduled on each machine.

1. Introduction

As it was explained in the statement of the project, the main goal for this algorithm is to schedule as many jobs as possible from a list containing n jobs, in m different machines – being $n \geq m$. Furthermore, this algorithm must run in **polynomial time** with the size of the input.

Bearing in mind the required time complexity, it seems like a good idea to try and expand the **greedy** method that was developed in [2]. This technique will give out a nice and simple algorithm which will accomplish all regarded objectives.

In section §2 two different **pseudocodes** for this algorithm will be introduced; and section §3 will develop a **proof of correctness** for them. The analysis of their **time complexity** will be performed in section §4 along with an empirical time-performance test to check the behavior of the algorithm. Finally, section §5 will provide a **Java sourcecode** fully implementing the algorithm, and section §6 will show some **results** and point out the **conclusions**.

Notice that, in principle, there might be **different optimal solutions** for this problem. The focus of this algorithm will not be to find them all or to find an specific one, but rather to obtain any such solution: any solution scheduling the maximum number of jobs possible, regardless of how or to which machines they are assigned.

2. Pseudocode

2.1. A first approach

In comparison to the classical problem covered in chapter #4 from reference [2], an extra greedy strategy is needed in order to find to which machine – in the event of several possibilities – should the current job be

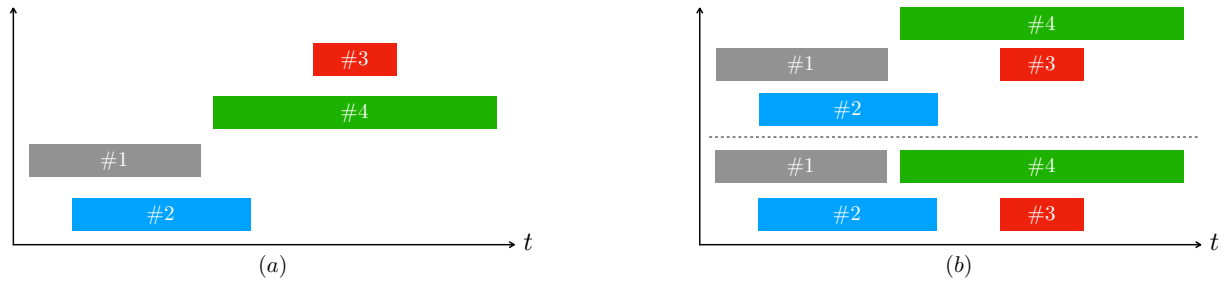

Figure 1: Example of interval scheduling on multiple resources for $n = 4$ and $m = 2$. (a) Jobs before scheduling. (b) Jobs after scheduling: lower picture is optimal, upper picture is not.

assigned. A natural thought here is to assign the job to the compatible machine which will become available **later**; this way, the amount of wasted time between jobs is minimized, and the earliest time some machine is available stays as low as possible for the amount of jobs scheduled over a collection them: this property will be called here on ***minimum earliest availability***. In section §3 it is shown that this method is indeed correct and finds an optimal solution to the problem.

Notice in the example of Figure 1 that the third assigned job – the one labeled #3 – must be properly assigned to the machine performing job #2 for the schedule to be able to perform the four jobs. If the algorithm chooses to assign job #3 to the machine performing job #1, then job #4 could not be done and so the result would not be optimal.

A first simple pseudocode is provided next. The initial goal was to obtain an algorithm linear in time with the number of input intervals as well as with the number of resources: $\mathcal{O}(nm)$.

Listing 1: A first Pseudocode for the *Interval Scheduling on Multiple Resources* problem

```
Read "input.txt" file
  Save variables n and m
  Build an array of jobs holding the start and finish times as well as a job ID number
Sort all jobs by their finish times, breaking ties arbitrarily

Create a variable "scheduled" for holding the number of scheduled jobs (initialized to zero)
Create an array of machines of size m
  Each machine will consist on a list of scheduled jobs and the time at which
  it becomes available (initialized to zero for all machines)

for i=1 to n
  boolean discard = true
  for j=1 to m
    boolean condition = Start(job i) >= Availability(machine j) AND
            { Availability(machine k)<Availability(machine j) OR discard }
    if (condition)
      k=j
      discard = false
    end if
  end for
  if (not discard)
    Add job "i" to the list of machine "k" and update its availability
    scheduled++
  end if
end for

Write "output.txt" file
```

2.2. Improvements

Making use of a fancier data structure some improvements over the previous method could be introduced. Instead of checking all possible machines, build a **binary search tree** ordering them by the time they become available `A[m]`. That way, the search for the machine with minimum gap between the starting time of the current job and the moment at which such machine becomes available can be done in $\mathcal{O}(\log(m))$ *average time* instead of $\mathcal{O}(m)$. Also, it will cost $\mathcal{O}(\log(m))$ *average time* in maintenance to fix the binary tree structure everytime an interval is scheduled and `A[m]` updated – see chapter #10 in reference [1] for binary tree operations. All in all, provided – as it will be shown – that the pseudocode in section §2.1 was right, the *average time complexity* will be improved to $\mathcal{O}(n\log(m))$, which is clearly better than what we previously had. More detail on this will be provided in section §4.

Listing 2: Improved Pseudocode for the *Interval Scheduling on Multiple Resources* problem

```
{...} Repeat lines 1-10

Create a binary search tree with all machines ordered according to their availability

for i=1 to n
  Use the tree to find "k" such that Start(job i) >= Availability(machine k) AND
           Start(job i) - Availability(machine k) is minimum
  if (it finds some "k")
    Add job "i" to the list of machine "k" and update its availability
    scheduled++
    Update the tree (remove old node for machine "k" and add new one)
  end if
end for

Write "output.txt" file
```

3. Proof of Correctness

The first thing to show is that both algorithms return a **valid solution**; in other words, that all jobs scheduled are compatible with one another giving the specific number of machines. This is easy to see for section §2.1, as the algorithm will check all machines and will only assigned the current job pending scheduling if it can be scheduled without conflicting with some machine's already scheduled jobs.

In the case of section §2.2, it is essentially the same thing; the only difference being in how it finds such a suitable machine. We will now prove how this is done and why the result is the one we are looking for.

Procedure:

Listing 3: Pseudocode for the Binary Search Tree procedure

```
Function: Use the tree to find "k" such that Start(job i) >= Availability(machine k) AND
          Start(job i) - Availability(machine k) is minimum

  while (it exists next element in the tree: starting in the root)
    if Start(job i) >= Availability(machine at current position in the tree)
      k = machine at current position in the tree
      move to the closest node in the higher value side of the tree
    else
      move to the closest node in the lower value side of the tree
    end if
  end while
  return" k" (or "null"  if nothing was found)
end function
```

Proof:
By means of contradiction, suppose the presented strategy fails to find the optimal "k". Then it will have to be one of two options: either it looks in the higher side when it should look in the lower or vice versa. Denote the starting time of the i^{th} job $S(i)$, and the time at which the j^{th} machine becomes available $A(j)$. When if it looks in the higher side, it must be that it already has "k" such that $S(i) \geq A(k)$ and by looking into lower values $A(j)$ it will only be $S(i) - A(k) < S(i) - A(j)$, and so any of these $A(j)$ could not correspond to an optimal solution: a contradiction. On the other hand, if it looks in the lower side, it must be that $S(i) - A(\texttt{machine at current position in the tree}) < 0$; therefore any element in the higher half could not be $S(i) - A(j) > 0$ as the hypothesis said: again, a contradiction.

$\mathbb{Q}.\mathbb{E}.\mathbb{D}.$ ■

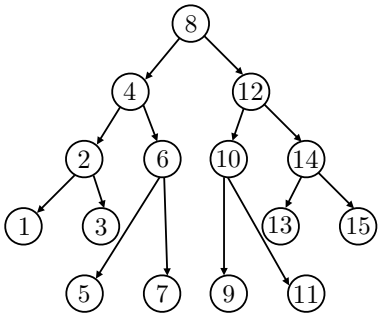

Figure 2: Example of a Binary Search Tree.

Call $\mathcal{A}$ the solution given by the algorithm. Next, it has to be shown that any of these algorithms provides with an optimal solution. This means that the number of scheduled jobs $|\mathcal{A}|$ is the same as in any other optimal solution $\mathcal{O}$ that could be proposed; which can be proven by showing that the algorithm always **stays ahead** in the search for such an optimal solution.

As mentioned in section §2, the implemented greedy strategy works in such way as to provide availability – for the amount of jobs scheduled over a collection them – as early as possible: *minimum earliest availability*. Denoting $f_{\mathcal{O}/\mathcal{A}}(i)$ the earliest available time in any one machine, for solutions $\mathcal{O}$ and $\mathcal{A}$ respectively, when the first i jobs have been considered in order of growing finishing times; the correctness of the strategy can be proven by **induction** showing that for all partial solutions that only considered the first i jobs $\{\mathcal{A}_i, \mathcal{O}_i\}$:

$$|\mathcal{A}_{i-1}| \geq |\mathcal{O}_{i-1}| \quad \Rightarrow \quad |\mathcal{A}_i| \geq |\mathcal{O}_i| \quad (\forall i) \tag{3.1}$$

Therefore, as $|\mathcal{A}_0| \geq |\mathcal{O}_0|$ for obvious reasons, it is the case that the algorithm stays ahead $\forall i$, and therefore produces an optimal solution when $i = n$.

Proof:

When evaluating the i^{th} job several things can happen:

1. Both $\mathcal{A}$ and $\mathcal{O}$ schedule/discard the job $\Rightarrow$ Eq. 3.1
2. $\mathcal{A}$ schedules the job and $\mathcal{O}$ discards it $\Rightarrow$ Eq. 3.1
3. $\mathcal{O}$ schedules the job and $\mathcal{A}$ discards it $\Rightarrow$ $|\mathcal{A}_{i-1}| \neq |\mathcal{O}_{i-1}| \Rightarrow |\mathcal{A}_{i-1}| > |\mathcal{O}_{i-1}| \Rightarrow |\mathcal{A}_i| \geq |\mathcal{O}_i|$

To show that the first implication in the last statement is correct, assume by means of contradiction that it is not. Then it will be $|\mathcal{A}_{i-1}| = |\mathcal{O}_{i-1}|$, and because for the same number of scheduled jobs the algorithm produces a solution with *minimum earliest availability*, it must also be $f_{\mathcal{A}}(i-1) \leq f_{\mathcal{O}}(i-1)$. But the hypothesis was that $\mathcal{O}$ schedules the job and $\mathcal{A}$ discards it, implying $f_{\mathcal{A}}(i-1) > S(i) \geq f_{\mathcal{O}}(i-1)$, where $S(i)$ is the starting time of the i^{th} job in order of growing finishing time. Therefore $f_{\mathcal{A}}(i-1) > f_{\mathcal{O}}(i-1)$: a contradiction.

$\mathbb{Q}.\mathbb{E}.\mathbb{D}.$ ■

The only thing left to prove is that indeed the algorithm produces a solution which, for the amount of jobs scheduled over a collection them, presents ***minimum earliest availability***. To do so, notice that the greedy strategy always schedules a job when $f_{\mathcal{A}}(i-1) \leq S(i)$, so it increases $|\mathcal{A}_i|$ as fast as possible following the low-to-high finishing times order for the jobs. Also, this strategy implies $f_{\mathcal{A}}(i-1) \neq f_{\mathcal{A}}(i)$ when only one machine is available to schedule the i^{th} job: the machine with earliest availability.

If the case is such, then $f_{\mathcal{A}}(i-1) < f_{\mathcal{A}}(i) \leq T(i)$; where $T(i)$ is the finishing time for the i^{th} job in order of growing finishing time. Therefore, if the earliest availability is only incremented when strictly necessary and it is incremented to, at most, the lowest possible value – the one that shows up first in order of growing finishing times – it must be the case that for any number of scheduled jobs, the algorithm keeps producing partial solutions with *minimum earliest availability*.

Proof:

From the partial solution $\mathcal{A}_i$ it is possible to construct any other partial solution not achieved by the algorithm – and with the same number of jobs scheduled – by performing a number of changes to it. If none of these changes can result in an improvement over the ***earliest available time***, no sequence made out of many changes of the same kind will. Notice that, by definition, the *available earliest time* must be the finishing time of one of the scheduled jobs.

Any of these changes must maintain the total number of scheduled jobs and fall into one of the following categories:

1. ***Switch*** *one sequence of consecutively scheduled jobs in a machine for a compatible same-length sequence of discarded ones:*

   For this kind of changes to have an effect on the *earliest available time*, it must switch one of the final sequences of scheduled jobs in any machine, as changes in some intermediate or initial sequence of jobs will not have an effect on the time at which the machine in question becomes available. Nevertheless, in general, if some job has not been scheduled by the algorithm it means that there was some conflict with a previously scheduled one. Because the scheduling takes place in order of finishing times, even if this change is compatible, it will mean changing an specific number of jobs for others with later finishing times: there cannot be any of these substituting jobs with an earlier finishing time than its counterpart in the substituted sequence, because in such case that job would not have been discarded by the algorithm. Therefore, by definition, this kind of change cannot improve the *earliest available time*.

2. ***Swap*** *a sequence of scheduled jobs to a compatible position in a different machine.*

   Again, for this kind of changes to have an effect on the *earliest available time*, one of the final sequences of scheduled jobs in any machine must be involved. Also, in general, swapping positions between two sequences of jobs – not necessarily of the same length – from different machines will not have any effect over the *earliest available time*, as the two finishing times of those sequences will remain the same. For a swap to make a difference, it will have to consist on sequence of jobs moving to an empty slot in a different machine. Because the algorithm assigns each job to the machine with latest compatible availability, any such swap will have to be done so that the first job in the sequence occupies a spot of earlier availability, and frees one of later availability. Doing so will at best maintain the current *earliest available time*, so this kind of changes cannot improve the result given by the algorithm.

3. ***Combinations*** *of the previous two.*

   It is not so clear though, that through some clever switch and swap combination this *earliest available time* could not be improved. To show this it is good to focus only on the final sequences of jobs in every machine. As it has been previously shown, switching a sequence of jobs for another which was discarded only results on a later finishing time, but something has to be said about the starting times of the different discarded jobs, as they could potential leave gaps for some scheduled jobs to swap positions. Nevertheless, if those new jobs were discarded to begin with, it must mean that their starting times were earlier than the earliest availability, making such swaps incompatible. Therefore, even through this mechanism, a favorable swap to increase the *earliest available time* cannot be found.

All in all, if there is no possible change that will improve that earliest available time, then it must be that the algorithm produces a solution with ***minimum earliest availability***; and so, both of the proposed algorithms work for finding an optimal solution to the problem.

$\mathbb{Q}.\mathbb{E}.\mathbb{D}.$ ■

4. Analysis and Computational Complexity

4.1. Complexity Analysis

Time Complexity:

In the case of the algorithm provided in section §2.1, it can be broken up in the following input-size-dependent parts:

- Reading the **"input.txt"** file and building the array of jobs takes $\mathcal{O}(n)$ time.
- Ordering such array takes $\mathcal{O}(n\log n)$ time.
- Creating the array of machines and initializing it takes $\mathcal{O}(m)$ time.
- Outer loop takes $\mathcal{O}(n)$ time.
- Each step of the outer loop is an entire inner loop which takes $\mathcal{O}(m)$ time.
- Writing the **"output.txt"** file takes $\mathcal{O}(n)$ time.

Therefore, this algorithm will show a time-complexity of $\mathcal{O}(n\log n + nm)$:

$$\mathcal{O}(n + n\log n + m + nm + n) = \mathcal{O}(2n + m + n\log n + nm) = \mathcal{O}(n\log n + nm) \tag{4.1}$$

On the other hand, the algorithm provided in section §2.1 can be broken up in the following input-size-dependent parts:

- Reading the **"input.txt"** file and building the array of jobs takes $\mathcal{O}(n)$ time.
- Ordering such array takes $\mathcal{O}(n\log n)$ time.

- Creating the array of machines and initializing it takes $\mathcal{O}(m)$ time.
- Outer loop takes $\mathcal{O}(n)$ time.
- Each step of the outer loop is takes $\mathcal{O}(\log m)$ ***average time*** and $\mathcal{O}(m)$ ***worst-case time*** to search and update the tree. Check chapter #10 in Reference [1] for more information.
- Writing the **"output.txt"** file takes $\mathcal{O}(n)$ time.

Therefore, this algorithm will show an average, and worst-case time-complexities of $\mathcal{O}(n\log n)$ and $\mathcal{O}(n\log n + nm)$ respectively:

$$\mathcal{O}(n + n\log n + m + n\log m + n) = \mathcal{O}(n\log n + n\log m) = \mathcal{O}(n\log n) \tag{4.2}$$
$$\mathcal{O}(n + n\log n + m + nm + n) = \mathcal{O}(n\log n + nm) \tag{4.3}$$

Space Complexity:

The space Complexity is even simpler in these cases. The only difference between the two is the binary tree, which will not worsen the asymptotic behavior. The corresponding input-size-dependent parts are:

- The array of jobs takes $\mathcal{O}(n)$ space.
- The array of machines takes $\mathcal{O}(m)$ space to save each machine's availability.
- The sum of all lists of schedule jobs in the machines takes $\mathcal{O}(n)$ space.
- The binary search tree takes $\mathcal{O}(m)$ space.

All in all, the resulting space-complexity in both cases is $\mathcal{O}(n)$:

$$\mathcal{O}(n + m + n + m) = \mathcal{O}(2n + 2m) = \mathcal{O}(n + m) = \mathcal{O}(n) \tag{4.4}$$

Notice that this algorithms return the well-known solutions of the *interval scheduling problem with a single resource* when $m = 1$, so they can be considered a **generalization**.

4.2. Runtime Tests

Using the implementation of the pseudocode from section §2.1 that will be shown in section §5, a modest run-time test was performed.

The main goal for this, was to show how the **time complexity truly behaves as predicted** both for the number of jobs n, and for the number of machines m: in other words that it is dominated by $\mathcal{O}(n\log n + nm)$. This will be generally true. Notice that it will always have to be $n > m$ in order to see actual growth. The ranges that were picked are:

- For the test on the number of jobs: $n \in [400, 40000]$ and $m = 200$.
- For the test on the number of machines: $n = 4000$ and $m \in [0, 2000]$.

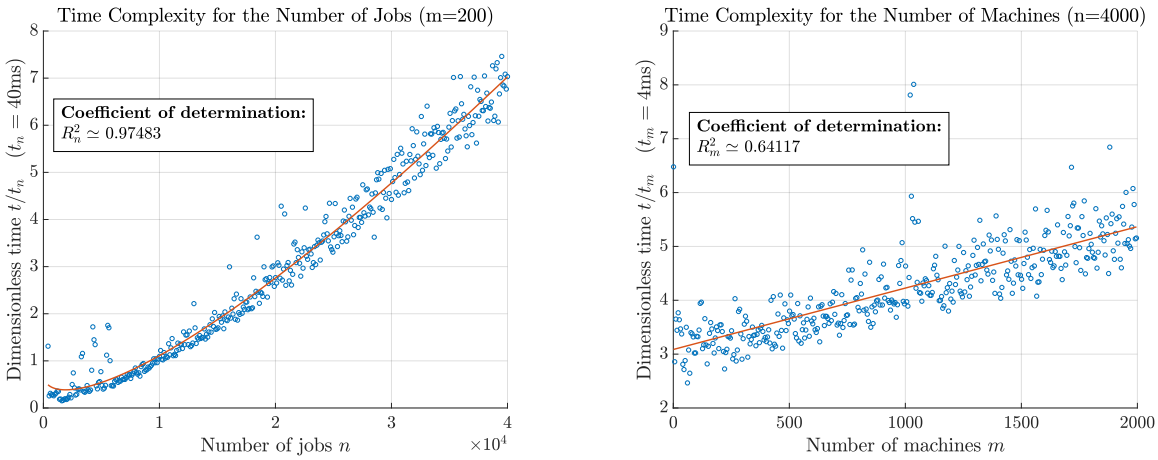

Figure 3: Experimental tests for the runtime of one implementation of the algorithm in section §2.1 **(Left)** Dimensionless time as a function of the number of jobs in the input and for $m = 200$ machines. **(Right)** Dimensionless time as a function of the number of machines available and for $n = 4000$ jobs.

Finally, the step in between each sample was chosen as to collect a total of 400 samples for each test.

Another thing that was taken into account was the fact that the actual absolute results will vary depending on factors such as the kind of processor used, how busy it is at the moment of the tests, or the programming language in which the algorithm is implemented. As a simple way to filter out all these discrepancies the time presented in the results has been made dimensionless according to certain **characteristic times** which in this case happened to be $t_n = 40\text{ms}$ and $t_m = 4\text{ms}$. The choice of this times is to some extent random, but as far as the decision is made in a consistent fashion, the results should be cross-comparable with any other machine running the algorithm. The major points that were considered to take this decision are:

1. The **range of the results** varies from zero to ten.
2. The **global behavior** of the regression curves can be seen clearly.

It is apparent from Figure 3, how the tests match the theoretical results predicting the time complexity for this algorithm, both with respect to the number of jobs and with respect to the number of available machines. The regression curves correspond to the following functions:

$$\frac{t}{t_n} \simeq 0.91577 - 0.06133\log n - 0.00054n + 0.00006n\log n = \mathcal{O}(n\log n) \tag{4.5}$$

$$\frac{t}{t_m} \simeq 3.08640 + 0.00114m = \mathcal{O}(m) \tag{4.6}$$

These functions present **coefficients of determination** $R_n^2 \simeq 0.97483 \simeq 97\%$ and $R_m^2 \simeq 0.64117 \simeq 64\%$ respectively. These are pretty good results for linear regressions, specially in the first case. The reason why the second result is appreciably lower is that its absolute time scales are smaller by an order of magnitude: $t_m = t_n/10 = 4(\text{ms})$. This smaller timescale makes the experimental results more susceptible to measurements errors and noise. The variability due to this effect can be easily appreciated by looking at the dispersion of the experimental measurements in Figure 3. This means that the lower value of its *coefficient of determination* is not due to a mismatch in the shape of the curve, but rather to the scattering of the samples.

5. Implementation in Java

The following source code presents a possible implementation of the pseudocode from section §2.1. Due to time constraints when working on this project, implementation of the pseudocode from section §2.2 was not pursued. This turned out not to be an issue for producing results, as the time complexity was already fairly good for the simpler version of the algorithm, and practical implementation on a laptop showed typical times in the order of 0.01 seconds to schedule hundreds of jobs over tens of machines. This was shown in section §4.2.

Nevertheless, the process to adapt the current source code to the improved version is fairly straight forward thanks to the modularity achieved by adopting OOP techniques in the Java programming language. The only piece of code that would need to be partially updated is the `"buildSchedule()"` method in the `"Scheduler"` class, in order to include the binary search tree discussed in previous sections.

Listing 4: Main Class Source Code in Java

```
// import Interval_Scheduling.JobRandomizer;
import Interval_Scheduling.Scheduler;
import java.io.FileNotFoundException;
import java.io.IOException;

public class Main {

    /* ------------------------------------------------- */
    /* -------------- PEDRO RIVERO RAMIREZ -------------- */
    /* ------------------------------------------------- */

    public static void main(String[] args) {

        // int n = 400;                         // Number of jobs
        // int m = 20;                          // Number of resources
        double startTime, elapsedTime;

        String inputPath = "./input.txt";
        String outputPath = "./output.txt";

        try {

            // JobRandomizer.randomJobInput(n,m,inputPath);

            startTime = (double) System.nanoTime();

            Scheduler intervalScheduler = new Scheduler(inputPath);
            intervalScheduler.saveSchedule(outputPath);

            // Print elapsed time
            elapsedTime = ((double)System.nanoTime()-startTime)/1000000000;
            System.out.println( "Elapsed time: " + elapsedTime + " seconds" );
        }

        catch (FileNotFoundException fnf) {
            System.out.println("File 'input.txt' not found at location of execution!"); }
        catch (IOException io) { System.out.println("IOException!"); }
    }
}
```

Listing 5: Scheduler Class Source Code in Java

```
package Interval_Scheduling;

import java.io.*;
import java.util.Arrays;
import java.util.Scanner;
```

```
6
7   public class Scheduler {
8
9       /* ------------------------------------------------- */
10      /* -------------- PEDRO RIVERO RAMIREZ -------------- */
11      /* ------------------------------------------------- */
12      /* This class provides objects called Schedulers.    */
13      /* ------------------------------------------------- */
14      /*  Their function is to schedule a series of jobs   */
15      /* defined by starting and finishing times.          */
16      /*  Times will be assumed to be non-negative integer */
17      /* numbers.                                          */
18      /*  Jobs will be scheduled for a chosen number of    */
19      /* resources.                                        */
20      /* ------------------------------------------------- */
21      /* ------------------------------------------------- */
22
23      /* ------------- FIELDS ------------- */
24
25      private int n, m;                       // Number of jobs and resources respectively
26      private Job[] jobs;                     // Starting and finishing times for all jobs
27      private int scheduled;                  // Keeps track of the number of jobs scheduled
28      private Resource[] resources;           // Array holding all resources
29
30      /* ---------- CONSTRUCTORS ---------- */
31
32      public Scheduler (String inputPath) throws FileNotFoundException {
33          Scanner sc = new Scanner(new File(inputPath));
34          this.n = sc.nextInt();
35          this.m = sc.nextInt();
36
37          this.jobs = new Job[this.n];
38
39          for (int i=0; i<n; i++) {
40              this.jobs[i] = new Job();
41              this.jobs[i].setStart(sc.nextInt());
42              this.jobs[i].setFinish(sc.nextInt());
43              this.jobs[i].setJobID(i+1);
44          }
45          System.out.println();
46
47          JobComparator jc = new JobComparator(); // Creates comparator for Job objects
48          Arrays.sort(this.jobs, jc);                  // Sorts jobs array by finishing time
49
50          this.scheduled = 0;
51          this.resources = new Resource[this.m];
52
53          for (int j=0; j<m; j++) {
54              this.resources[j] = new Resource();
55          }
56
57          buildSchedule();                        // Assigns jobs to the resources optimally
58          sc.close();
59      }
60
61      /* ------------ METHODS ------------- */
62
63      private void buildSchedule () {         // Simple assignation of jobs to resources
64
65          for (int i=0; i<n; i++) {
66
67              int k = 0;
68              boolean discard = true;
69              boolean condition;
70
```

```
            for (int j=0; j<m; j++) {

                condition = jobs[i].getStart()>=resources[j].getAvailableTime() &&
                        (discard || resources[k].getAvailableTime()<resources[j].
    getAvailableTime());

                if (condition) {
                    k = j;
                    discard = false;
                }
            }

            if (!discard) {
                resources[k].addJob(jobs[i]);
                scheduled++;
            }
        }
    }


    public void saveSchedule (String outputPath) throws IOException {

        BufferedWriter bw = new BufferedWriter(new FileWriter(outputPath));

        bw.write(Integer.toString(scheduled));
        bw.newLine();

        for (int i=0; i<this.m; i++) {
            bw.write(resources[i].toString());
            bw.newLine();
        }

        bw.flush();
        bw.close();
    }
}
```

Listing 6: Job Class Source Code in Java

```
package Interval_Scheduling;

public class Job {

    /* ------------------------------------------------- */
    /* -------------- PEDRO RIVERO RAMIREZ -------------- */
    /* ------------------------------------------------- */

    /* ------------- FIELDS ------------- */

    private int start;                      // Starting time of the job
    private int finish;                     // Finishing time of the job
    private int jobID;                      // ID number of the job according to input file

    /* ---------- CONSTRUCTORS ---------- */

    public Job () {
        this.start = 0;
        this.finish = 0;
        this.jobID = 0;
    }

    /* ------------ METHODS ------------- */

    public int getStart() { return start; }
    public int getFinish() { return finish; }
```

```
27      public int getJobID() { return jobID; }
28
29      public void setStart(int start) { this.start = start; }
30      public void setFinish(int finish) { this.finish = finish; }
31      public void setJobID(int jobID) { this.jobID = jobID; }
32
33      @Override
34      public String toString() {
35          return "Job{" +
36                  "jobID=" + jobID +
37                  ", start=" + start +
38                  ", finish=" + finish +
39                  '}';
40      }
41  }
```

Listing 7: Resource Class Source Code in Java

```
1   package Interval_Scheduling;
2
3   import java.util.ArrayList;
4   import java.util.Iterator;
5
6   public class Resource {
7
8       /* ------------------------------------------------- */
9       /* -------------- PEDRO RIVERO RAMIREZ -------------- */
10      /* ------------------------------------------------- */
11
12      /* ------------- FIELDS ------------- */
13
14      private ArrayList<Job> schedule;
15      private int availableTime;
16
17      /* ---------- CONSTRUCTORS ---------- */
18
19      public Resource() {
20          this.schedule = new ArrayList();     // ArrayList of jobs scheduled for the resource
21          this.availableTime = 0;              // Time at which the resource becomes available
        again
22      }
23
24      /* ------------ METHODS ------------- */
25
26      public int getAvailableTime() { return availableTime; }
27
28      public void addJob(Job job) {
29          this.schedule.add(job);
30          if (this.availableTime < job.getFinish()) { this.availableTime = job.getFinish(); }
31      }
32
33      @Override
34      public String toString() {
35
36          String schedule = new String();              // Output variable
37          Iterator it = this.schedule.iterator();      // Iterator for ArrayList
38
39          while (it.hasNext()) {
40              schedule += Integer.toString( ((Job)it.next()).getJobID() ) + " ";
41          }
42
43          return schedule;                     // Outputs a sequence of scheduled jobIDs
44      }
45  }
```

Listing 8: JobComparator Class Source Code in Java

```java
package Interval_Scheduling;

import java.util.Comparator;

public class JobComparator implements Comparator<Job> {

    /* ------------------------------------------------- */
    /* -------------- PEDRO RIVERO RAMIREZ -------------- */
    /* ------------------------------------------------- */

    @Override
    public int compare(Job o1, Job o2) {
        if ( o1.getFinish() < o2.getFinish() ) return -1;
        else if ( o1.getFinish() == o2.getFinish() ) return 0;
        else return 1;
    }
}
```

Listing 9: JobRandomizer Abstract Class Source Code in Java

```java
package Interval_Scheduling;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Random;

public abstract class JobRandomizer {

    /* ------------------------------------------------- */
    /* -------------- PEDRO RIVERO RAMIREZ -------------- */
    /* ------------------------------------------------- */
    /* This abstract class provides randomization method  */
    /* ------------------------------------------------- */
    /*  Its function is to create the input files with    */
    /* the jobs that will have to be scheduled.           */
    /*  The longest time duration of any interval will be */
    /* of 20 units of time.                               */
    /*  Jobs will be scheduled along 100000 units of time.*/
    /*  The parameters n and m will be inputted as        */
    /* arguments in its method.
    /* ------------------------------------------------- */
    /* ------------------------------------------------- */

    /* ------------- FIELDS ------------- */

    private static int MAX_SPAN = 100000;    // Maximum time span
    private static int MAX_DURATION = 20;    // Maximum duration

    /* ------------ METHODS ------------- */

    public static void randomJobInput (int n, int m, String inputPath) throws IOException {

        Random rand = new Random();          // Random integer generator
        int start, duration;                 // Starting time and duration

        BufferedWriter bw = new BufferedWriter(new FileWriter(inputPath));

        bw.write(Integer.toString(n) + " " + Integer.toString(m));
        bw.newLine();

        for (int i=0; i<n; i++) {

```

```
44              start = rand.nextInt(MAX_SPAN - MAX_DURATION + 1);
45              duration = rand.nextInt(MAX_DURATION) + 1;
46
47              bw.write(Integer.toString(start) + " " + Integer.toString(start + duration));
48              bw.newLine();
49
50          }
51
52          bw.flush();
53          bw.close();
54      }
55  }
```

6. Results and Conclusions

A small and simple example of the algorithm is shown next for $n = 20$ and $m = 4$. The point of showing such a reduced example is so that the correctness of the solution can be checked more or less easily by direct inspection, without loosing a sense of how these files should look like. For this specific example the time it took to obtain the solutions shown is $t = 34$ milliseconds.

Listing 10: Example "input.txt" file

```
1  20 4
2  26 36
3  77 88
4  79 95
5  39 55
6  9 11
7  77 81
8  13 16
9  47 64
10 34 43
11 80 81
12 53 57
13 20 28
14 53 60
15 3 5
16 22 30
17 1 6
18 78 97
19 69 75
20 35 46
21 77 90
```

Listing 11: Example "output.txt" file

```
1  18
2  14 15 9 13 10
3  16 5 7 12 19 11 2
4  1 4 20
5  8 18 6
```

The format for these files is explained in section §0.1

Some **conclusions** to be drawn out of the project are:

- The theory predicts to really high precision the **asymptotic behavior** of the algorithm.
- For this kind of algorithms, **sorting** is usually the most restrictive process.

References

[1] Collins, W.J. *Data Structures and the Java Collections Framework*. 3rd Edition. John Wiley & Sons, 2011.

[2] Kleinberg, J. & Tardos, E. *Algorithm Design*. 1st Edition. Pearson Education, 2005.