1. **Scheduling algorithms** The following $n = 12$ jobs with given processing times have to be scheduled on $m = 3$ parallel and identical processors with the objective of minimizing the makespan. $C_j$ is the completion time of a job.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>4</td>
<td>9</td>
<td>12</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

(a) Draw the List-Scheduling-schedule as a Gantt-Chart. How much is $C_{\text{max}}$ and how much is $\sum C_j$?

(b) Draw the LPT-schedule as a Gantt-Chart. How much is $C_{\text{max}}$ and how much is $\sum C_j$?

(c) Draw the SPT-schedule as a Gantt-Chart. How much is $C_{\text{max}}$ and how much is $\sum C_j$?

(d) Find the optimal and the worst list concerning the objective of minimizing the makespan.

(e) Find the optimal and the worst list concerning the objective of minimizing the sum of the processing times.

(f) Draw McNaughton’s schedule (preemptions allowed).

**Solution:**

(a) The Gantt-Chart for List-Scheduling is:

```
C_{\text{max}} = \max\{12,29\} = 29 and \sum C_j = 78 + 56 + 62 = 196
```

(b) The Gantt-Chart for LPT is:

```
C_{\text{max}} = \max\{12,29\} = 29 and \sum C_j = 85 + 85 + 82 = 252
```

(c) The Gantt-Chart for SPT is:
\[ C_{\text{max}} = \max\{12,31\} = 31 \text{ and } \sum C_j = 45 + 59 + 64 = 168 \]

(d) Lower Bound, \( LB = \max\{12,84/3\} = 28 \) The optimal list concerning the objective of minimizing the makespan must have a makespan of 28. One such example is \( \{C,J,H,B,I,K,L,F,G,E,A,D\} \). The Gantt-Chart for this list using List-Scheduling is:

To get the worst list concerning the objective of minimizing the makespan, remove the job with max processing time \( C(=12) \), then apply LS to remaining jobs such that makespan = \((84-12)/3 = 24\). Then add the job \( C \) to one of the machines to get the makespan to be \( 24+12 = 36 \). One such example is \( \{J,H,I,K,B,G,L,F,E,A,D,C\} \). The Gantt-Chart for this list using List-Scheduling is:

(e) SPT minimizes the sum of completion times at 168 and LPT has the maximum at 252.

(f) The Gantt-Chart for McNaughton’s schedule is:

2. Upper and lower bounds for scheduling algorithms, NP-completeness
(a) Explain why $\sum_{j=1}^{n} \frac{p_j}{m}$ and $p_{\text{max}} = \max(p_k)$ (for any $k \in \{1, \ldots, n\}$) are Lower Bounds for the makespan of any algorithm solving $P||C_{\text{max}}$.

(b) Find Upper Bounds for the makespan of List Scheduling for $P||C_{\text{max}}$.

(c) Find a tight worst-case example for the makespan achieved by List Scheduling in comparison to the optimum makespan for $m = 5$ processors.

(d) Repeat in your own words the most important ideas behind Ron Graham’s proof for the worst-case behaviour of List Scheduling.

(e) Prove that the decision version of $P || C_{\text{max}}$ is NP-complete by reducing Partition.

Solution:

(a) $\sum_{j=1}^{n} \frac{p_j}{m}$ is the total amount of processing that must be done across all machines. Then for $m$ machines, $\sum_{j=1}^{n} \frac{p_j}{m}$ would be the average processing that needs to be done. By properties of the average, at least one machine then must process at least this average. Hence, $C_k \geq \sum_{j=1}^{n} \frac{p_j}{m}$.

Also, a processor can process only one job at a time. This means, a job can be completed no faster than it’s given processing time. If $p_{\text{max}} = \max\{p_k\}$, then at least one machine must have $p_{\text{max}}$ processing load.

Therefore, $\sum_{j=1}^{n} \frac{p_j}{m}$ and $p_{\text{max}}$ form the lower bounds for makespan of any algorithm solving $P||C_{\text{max}}$ and clearly $C_{\text{max}} \geq \max \{\sum_{j=1}^{n} \frac{p_j}{m}, p_{\text{max}}\}$.

(b) One upper bound is the sum of all processing times (e.g. if all jobs ran sequentially on a single machine). We can go further and lower this upper bound when we schedule the jobs according to the List Scheduling procedure: schedule the next job always on that processor that has done the least amount of work so far. Let $J_k$ denote the job that completes last across all machines, say on machine $P_i$. This job (by definition) was assigned when machine $P_i$ had the least load, so that:

$$C_{\text{LS max}} = C_k = s_k + p_k$$

where $s_k$ is the amount of load on machine $P_i$ when $J_k$ was assigned to it. But this, by definition, must have been the least load among the machines when this happened. Therefore:

$$s_k \leq \sum_{j=1}^{n} \frac{p_j}{m} = \sum_{j=1}^{n} \frac{p_j}{m} - \frac{p_k}{m}$$

So we come to:

$$C_{\text{LS max}} = C_k = s_k + p_k \leq \sum_{j=1}^{n} \frac{p_j}{m} - \frac{p_k}{m} + p_k$$

and so

$$C_{\text{LS max}} \leq \frac{\sum_{j=1}^{n} p_j}{m} + p_k \left(1 - \frac{1}{m}\right)$$

By part (a) it follows that:

$$C_{\text{LS max}} \leq C_{\text{max}} + C_{\text{max}} \left(1 - \frac{1}{m}\right) = (2 - \frac{1}{m}) C_{\text{max}}$$

(c) To achieve a tight worst-case for the makespan, by List-Scheduling for $m = 5$ processors, $C_{\text{max}} = (2 - \frac{1}{5}) C_{\text{max}} \Rightarrow C_{\text{max}} = \frac{9}{5} C_{\text{max}}$. One example can be a list with 20 jobs having processing times $p_j = 1$ and one job with time $p_j = 5$. The Gantt-Chart for this list using List-Scheduling in comparison to the optimum makespan is:
Using List-Scheduling $C_{\text{max}} = 9$, while the optimum makespan is $C^*_{\text{max}} = 5$. $C_{\text{max}} = \frac{9}{5}C^*_{\text{max}}$.

We can observe that one job $J_k$, the one that finishes last, determines the makespan $C_{\text{max}}$. So we have, for the makespan of List-Schedule

$$C_{\text{max}}^{\text{LS}} = p_k + s_k$$

where $s_k$ is the amount of load on machine $P_i$ when $J_k$ was assigned to it.

We now use “$\leq$” relations.

(i) $s_k$, the starting time of this job cannot be larger than the sum of the processing times of all the other jobs divided by the number of machines, $m$.

$$s_k \leq \frac{\sum_{j \neq k} p_j}{m}$$

(ii) Also, the optimum makespan cannot be smaller than the average processing time or processing time of any single job.

$$\frac{\sum_{j=1}^{n} p_j}{m} \leq C^*_\text{max} \quad \text{and} \quad p_k \leq C^*_\text{max}$$

(see solution to part b)).

(e) We are going to prove that $P_2 || C_{\text{max}}$ is NP-hard. (From this it follows that also $P_m || C_{\text{max}}$ for an arbitrary $m \geq 2$ must be NP-hard. First let us define the decision versions of both problems.

PARTITION: Given a list of $n$ positive integers $s_1, s_2, ..., s_n$ and a value $b = \frac{\sum_{j=1}^{n} s_j}{2}$, does there exist a subset $J \subset I = \{1, \ldots, n\}$ such that

$$\sum_{j \in J} s_j = b = \sum_{j \in I \setminus J} s_j$$

Remark: Partition is NP-hard in the ordinary sense, i.e. the problem cannot be optimally solved by an algorithm with polynomial time complexity but with an algorithm of time complexity $O \left((n \cdot \max s_j)^k\right)$.

$P_2 || C_{\text{max}}$: Given $n$ jobs with processing times $p_j$ where $j \in \{1, 2, \ldots, n\}$ and a number $k$, the decision version of $P_2 || C_{\text{max}}$ is to check if you can schedule them on $m$ machines so as to complete by time $k$.

**Step 1:** $P_2 || C_{\text{max}}$ is in NP

If the schedule of the jobs for each of the 2 machines is given, it can be verified in polynomial time that
the completion times \( C_1, \ldots, C_n \) of all jobs \( J_1, \ldots, J_n \) are less than or equal to \( T \). Remark: Clearly, any \( C_k \) has a binary encoding which is bounded by a polynomial in the input length of the problem.) Thus, \( P^2 \| Cmax \) belongs to NP. Note: Every decision problem solvable in polynomial time belongs to NP. If we have such a problem \( P \) and an algorithm which calculates for each input \( x \) the answer \( h(x) \in \{\text{yes, no}\} \) in a polynomial number of steps, then this answer \( h(x) \) may be used as a certificate. This certificate can be verified by the algorithm. Thus \( P \) is also in NP which implies \( P \subseteq NP \).

**Step 2:** \( \text{PARTITION reduces to } P^2\|Cmax \): \( \text{PARTITION} \propto P^2\|Cmax \)

Note: For two decision problems \( P \) and \( Q \), we say that \( P \) reduces to \( Q \) (denoted \( P \propto Q \)) if there exists a polynomial-time computable function \( g \) that transforms inputs for \( P \) into inputs for \( Q \) such that \( x \) is a yes-input for \( P \) if and only if \( g(x) \) is a yes-input for \( Q \). A yes-answer for a decision problem can be verified in polynomial time (this is not the case for the no-answer).

The partitioning problem is reducible to \( P^2\|Cmax \):

1. The input of the scheduling problem can be computed in polynomial time given the input of the \( \text{PARTITION} \) problem:
   We must polynomial transform the input for \( \text{PARTITION} \) into an instance of \( P^2\|Cmax \) such that there is a solution for \( \text{PARTITION} \) if and only if there is a schedule with \( C_{max} \leq k \) for a suitable value \( k \). This is easy: We just set \( k = b \) and define the scheduling problem as follows: Consider the jobs \( J_j \) with \( p_j = s_j \) for \( j = 1, \ldots, n \). We choose \( k = b \) as the threshold for the corresponding decision problem.

2. a. If \( \text{PARTITION} \) has a solution, then the decision version of \( P^2\|Cmax \) has a solution:
   If \( \text{PARTITION} \) has a solution, then there exists an index set \( J \subseteq \{1, \ldots, n\} \) such that \( \sum_{i\in J} s_i = b \). In this case the schedule \( p_{i\in J} \) on \( P_1 \) and \( p_{i\notin J} \) on \( P_2 \) solves the decision version of problem \( P^2\|Cmax \).

2. b. If the decision version of \( P^2\|Cmax \) has a solution, then \( \text{PARTITION} \) has a solution:
   If the decision version of \( P^2\|Cmax \) has a solution, then the jobs with processing times \( p_1, \ldots, p_n \) are scheduled on \( P_1 \) (all \( p_{i\in J} \) and \( P_2 \) (all \( p_{i\notin J} \)) such that the makespan is less than or equal to \( k \). In this case the \( \text{PARTITION} \) \( \sum_{j\in J} s_j = b = k = \sum_{j\in J/ J} s_j \) solves the \( \text{PARTITION} \) problem.

Since \( P^2\|Cmax \) is in NP, and we can reduce an NP-complete problem to it in polynomial time, \( P^2\|Cmax \) must also be NP-complete.

3. **Time analysis of scheduling algorithms**

   (a) Write the Pseudocode of the List Scheduling algorithm to solve \( P \| Cmax \). Proof the correctness and perform a time analysis. Does LPT has the same time complexity?

   (b) Write the Pseudocode of McNaughton’s algorithm to solve \( P \mid pmtn \mid Cmax \). Proof the correctness and perform a time analysis.

   (c) Prove that McNaughton’s algorithm needs never more than \( n - 1 \) preemptions to produce an optimal solution.

**Solution:**

(a) Pseudocode:

```
INPUT:
Jobs[1...N]; // each element is (jobID,duration)

OUTPUT:
Processor[1...M]; // each element points to an ordered list of jobIDs.
Cmax;
processorID_Cmax;
```
initialise Cmax = 0, processorID_Cmax=0;

function List_scheduling(Jobs[1...N],Processor[1...M])
{
1. create MIN_HEAP for M elements, all initialized to be (processorID,0);
   //two field of MIN_HEAP: processorID and amount_of_work_so_far, ordered by the latter
2. For I from 1 to N
3. assign (processorID, work) = the top of the MIN_HEAP;
4. remove the top of the MIN_HEAP;
5. add Jobs[I]’s jobID to Processor[processorID]’s list;
6. put (processorID, work+Jobs[I]’s duration) back to MIN_HEAP;
7. if (work+Jobs[I]’s duration > Cmax)
8. Cmax = work+Jobs[I]’s duration;
9. processorID_Cmax = processorID;
10. I++;
}

Correctness:
(1) This algorithm terminates, because after I iterates to N and removing the top element of MIN_HEAP
   until the last one, the function will return,(2) This algorithm returns correct result. This is because
   the for loop processes every job. Within the scope of for loop, it puts a job to the processorID on the
   top of the MIN_HEAP, which is guaranteed to have the smallest amount_of_work_so_far. This is
   exactly what List Scheduling does.

Time analysis:
Line 1 needs O(M); line 3,6 takes O(logM); other lines inside the for loop takes O(1), so they are
negligible. The total time complexity is O(M + NlogM).
Using LPT will not have the same complexity:

LPT:
LPT needs to sort the Jobs array, which usually takes O(nlogn) depending on the sorting algorithm.
As a result, the time complexity becomes O(M + N(logN + logM)).

(b) Pseudocode:

INPUT:
Jobs[1...N]; // each element is (jobID,duration)

OUTPUT:
Processor[1...M]; // each element points to an ordered list of (jobIDs, starting_time)
Cmax;

function McNaughton(Jobs[1...N],Processor[1...M])
{
   //compute Cmax
   int p_max = 0;
   int p_sum = 0;
   FOR I from 1 to N
      IF Jobs[I]’s duration > p_max
         p_max = Jobs[I]’s duration
         p_sum += Jobs[I]’s duration
\[ C_{\text{max}} = \max\{ \text{ceiling}\{p_{\text{sum}}/N\}, p_{\text{max}} \} \]

//process each job
initialize processor_pointer = 1, processor_size = 0;
For I from 1 to N
    IF (Jobs[I]'s duration < (C_{\text{max}} - processor_size))
        add (Jobs[I]'s jobID, processor_size) to Processor[processor_pointer];
        processor_size += Jobs[I]'s duration;
        I++;
    ELSE IF (Jobs[I]'s duration == (C_{\text{max}} - processor_size))
        add (Jobs[I]'s jobID, processor_size) to Processor[processor_pointer];
        processor_size = 0;
        processor_pointer++;
        I++;
    ELSE
        add (Jobs[I]'s jobID, C_{\text{max}} - processor_size) to Processor[processor_pointer];
        put (Jobs[I]'s jobID, Jobs[I]'s duration - (C_{\text{max}} - processor_size))
            to I of Jobs list;
        processor_pointer++;
        processor_size = 0;

}

Correctness:
(1) The algorithm terminates. This is because the algorithm will return after iterating all elements in Jobs list. (2) The algorithms returns correct answer. It first computes C_{\text{max}} according to the formula. Then for each job, it compares that job's duration with the vacant spaces in the processor. If there is enough space, then it just puts that job to the current processor and adds the duration to the current processor size. If there is no enough space, then it first feeds the processor as many as it can, then put the remaining as a new job back to the job list, so that it can be accessed again in the next iteration. In the next iteration, the remaining job will be put in the next processor. This is exactly what McNaughton algorithm does.

Time analysis:
Computing the C_{\text{max}} needs \( O(N) \). Since all three cases inside the for loop only takes constant time, we only need to find out how many iterations does the for loop have. Obviously it at least needs \( O(N) \) because there are \( N \) jobs. However, the last ELSE block may add iterations on top of that. The number of execution for the last ELSE block is \( O(M) \). So the time complexity is \( O(N + M) \).

(c):
Each job interrupted at most once. More interruptions would not yield to smaller makespans.