Question 1 Solution)

This problem is very similar to the knapsack problem, which is known to be NP-hard, so no polynomial algorithm will be able to solve it.

An efficient way to solve this problem is a greedy approach, where objective function (maximize computing time) is maximized at each iteration.

```
Inputs: n data files, W bytes available, w_i: bytes of each data file, m_i: minutes to recompute

Output: S: subset of data files chosen


Sort n in decreasing order of m_i

cumul_capacity ← 0 // cumulated capacity of files chosen. Starts from 0

S ← ∅

for i in n
{
    file = n[i]                    // get first item with highest time
    if cumul_capacity + w_file ≤ W
       S ← S ∪{file}               // include file in the subset

       cumul_capacity ← cumul_capacity + w_file

}
```

As an example, we can have the following:

$n = \{f1, f2, f3, f4\}$

$w_{f1} = 1, w_{f2} = 1, w_{f3} = 1, w_{f4} = 2$

$m_{f1} = 20, m_{f2} = 10, m_{f3} = 40, m_{f4} = 30$

$W = 2$

The algorithm will start sorting n in decreasing order of m_i

$n = \{f3, f4, f1, f2\}$

In the first iteration it will include $f3$, $S = \{f3\}$

In the second iteration it will try to include $f4$, but capacity will exceed the limit, so we don't include it

In the third iteration it will include $f1$, $S = \{f3, f1\}$

In the second iteration it will try to include $f2$, but capacity will exceed the limit, so we don't include it

Finally, the subset chosen is $S = \{f3, f1\}$

Question 2 Solution)

A local search algorithm starts from a feasible solution and keeps iterating from solution space until a local optimum is found.

Here we'll start with a initial assignment (that can be obtained from a greedy solution):

$Alloc_i$: set of tasks allocated to machine i

A possible local search algorithm is the following:

First we'll define a procedure to evaluate the total processing time of a solution:

```
Evaluate (Alloc_i)
{
for i in M {
        proc_i = 0
        for j in Alloc_i{
                proc_i = proc_i + t_ij}}
return max(proc_i)
}
```

```
Inputs: Alloc_i: initial allocation (set of tasks allocated to machine i)
Output: FAlloc_i: final allocation (set of tasks allocated to machine i)

FAlloc_i ← Alloc_i
while no more improvement is found
{
     Get a job randomly, that were not yet tested
     Test the all the assignments of this job (Alloc2_i)
     If a better solution is found (Evaluate (Alloc2_i) < Evaluate (Alloc_i))
       FAlloc_i ← Alloc2_i
     Else
       stop
}
```

In worst case all jobs will be tested, so

- n: test all jobs
- $m - 1$: test the assignment of this job with all other machines
- $n * m$: complexity of evaluate procedure

$$n * (m - 1) * (n * m) = O(n^2 m^2)$$

An example would be:

$n = \{j1, j2, j3, j4, j5, j6, j7, j8\}$

$m = \{m1, m2\}$

A starting solution is:

$Alloc_{m1}$ = $\{j1, j2, j3, j4\}$

$Alloc_{m2}$ = $\{j5, j6, j7, j8\}$

In the first iteration we can choose job $j6$ and test allocate it to machine $m1$, total processing time decreased. So,

$Alloc_{m1}$ = $\{j1, j2, j3, j4, j6\}$

$Alloc_{m2}$ = $\{j5, j7, j8\}$

Now we take $j2$ and try to allocate to machine $m2$, total processing time did not get better. So we stop the execution with local minimum as:

$FAlloc_{m1}$ = $\{j1, j2, j3, j4, j6\}$

$FAlloc_{m2}$ = $\{j5, j7, j8\}$

Question 3 Solution)

Sets :

$P1$: First partition

$P2$: Second partition

Let's consider P1 < P2

Parameters:

c_{ij}: cost to match element i from $P1$ to element j in $P2$

r_{ij}: reward to match element i from $P1$ to element j in $P2$

Z: reward threshold

Variables:

x_{ij}: binary variable: 1 if element i from $P1$ is matched to element j in $P2$, 0 otherwise

Objective function

Minimize cost

$$Minimize \sum_{i\ in\ P1} \sum_{j\ in\ P2} c_{ij} x_{ij}$$

Constraints

Respect minimum threshold

$$\sum_{i\ in\ P1} \sum_{j\ in\ P2} r_{ij} x_{ij} \geq Z$$

All nodes from P1 can match to at most one node from P2 (in the statement it says matching CAN BE maximum not MUST BE maximum)

$$\sum_{j\ in\ P2} x_{ij} = 1 \quad \forall i\ in\ P1$$

All nodes from P2 can match to at most one node from P1

$$\sum_{i\ in\ P1} x_{ij} \leq 1 \quad \forall j\ in\ P2$$

Binary constraints

$$x_{ij} \in \{0.1\} \quad \forall i\ in\ P1, \forall j\ in\ P2$$

A greedy algorithm can be the following:

<u>*Inputs*</u>: same as model

<u>*Output*</u>: M – matchings between P1 and P2, so that M is a subset of the edges

First solve reward maximization problem

$Z_{tot} \leftarrow 0$

for i in $P1$

{

    Assign i to the element j in $P2$ (that is not yet assigned), with maximum r_{ij}

    $Z_{tot} \leftarrow Z_{tot} + r_{ij}$

}

If $Z_{tot} \geq Z$

    for i in $P1$

```
                test assign i to a free node in P2 that is free
                If solution has lower cost and total reward is still higher
than Z
                        Unassign i to its current match
                        Assign i to j
Else
        Problem is infeasible
```

Question 4 Solution)

This problem can be called the vertex cover problem, that is also known to be NP-Hard, so no polynomial algorithm will be able to solve it.

A heuristic way is described below

<u>*Inputs*</u>: $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ where V is the set of vertices and E is the set of edges

<u>*Output*</u>: D - subset of V such that every vertex not in D is adjacent to at least one member of D

Calculate the degree of each vertex deg_i, as the number of vertices i is adjacent with

$D \leftarrow \emptyset$

while $E \neq \emptyset$

{

$v1 \leftarrow \max\,(deg_i\ from\ V)$ *// get vertex v1 with highest degree*

$v2 \leftarrow \max(deg_i\ from\ V \text{ where } (v1, v2) \in E)$ *// get vertex v2 such that edge (v1,v2) exists in graph high highest degree*

$D \leftarrow D \cup v1 \cup v2$ *// add v1 and v2 to subset D*

$V \leftarrow V - \{v1\} - \{v2\}$ *// remove v1 and v2 from V*

}