To understand material better:

- start by making up some examples and working through the problem with the examples.
- try writing your own proof.
- fix problems in notation.
- draw lots of pictures to aid understanding.
- teach it to someone else.

The Load Balancing problem.

We are given a set of m machines $M_1, M_2, \ldots, M_m$ and a set of n jobs numbered 1, 2, …, n.
The processing time for job j is t_j.

All machines are identical and can run any of the jobs. Each machine can only run one job at a time. Each job should be assigned to one machine.

The goal is to assign each job to a machine so that the maximum amount of time used by one of the machines is minimized.

The Load Balancing problem: Formal notation.

Machines: $M_1, M_2, \ldots, M_m$

The n jobs are numbered 1, 2, …, n.

The processing time for job j is t_j.

A_i = set that contains the job numbers of the jobs that are assigned to machine M_i.

Side note: For a legal assignment of jobs to machines, the sets A_i for i=1 to m give a partition of the set of job numbers {1, 2, …., n}.

T_i= time machine M_i takes to process its jobs,

$T_i = \sum_{j \in A_i} t_j$

The goal is to Minimize the makespan which is defined to be equal to $\text{Max}_{i=1 \text{ to } m} T_i$.

Load Balancing: List Scheduling

List-scheduling algorithm.

- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

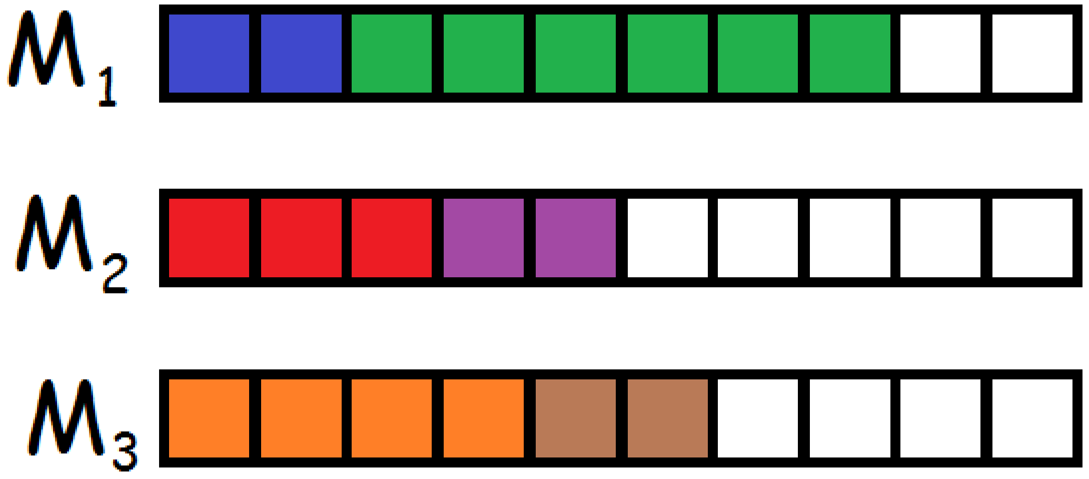
```
List-Scheduling(m, n, $t_1, t_2, \ldots, t_n$) {
    for i = 1 to m {
        $load_i \leftarrow 0$        ← load on machine i
        $A_i \leftarrow \phi$        ← jobs assigned to machine i
    }

    for j = 1 to n {
        Choose i so that $load_i$ is minimized.
        Add j to the set $A_i$.        ← assign job j to machine i
        $load_i \leftarrow load_i + t_j$        ← update load of machine i
    }
    return the sets $A_1, A_2, \ldots, A_m$
}
```

Implementation. O(n log m) using a priority queue.

One example:

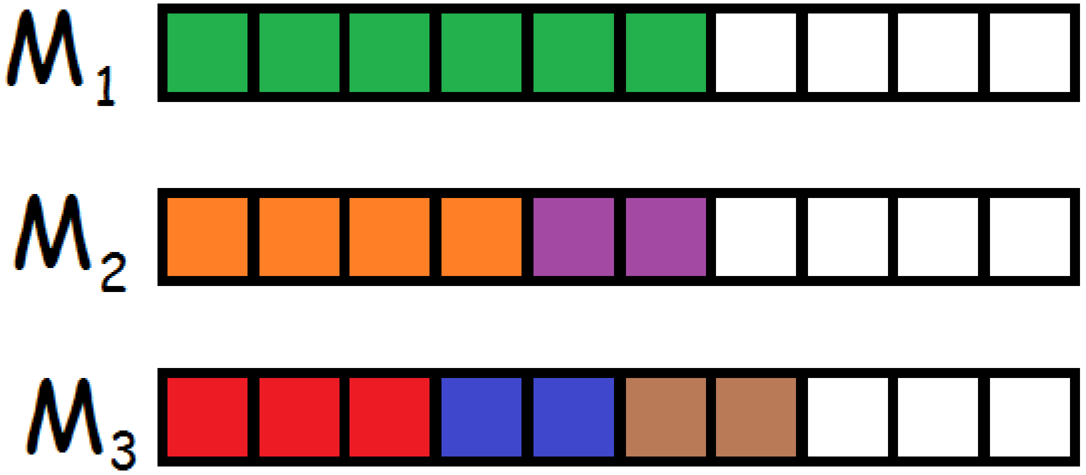
$m=3$, $n=5$, Job times: 2, 3, 4, 6, 2, 2

The makespan of this assignment is 8.



Reordering the job times in reverse sorted order:
m=3, n=5, Job times: 6, 4, 3, 2, 2, 2

The makespan of this assignment is 7.



Important observation:

If you have m integer values $a_1, a_2, \ldots, a_m$ then the minimum value is at most the floor of the average value and the maximum is at least the ceiling of the average value.

{50, 50, 50, 50, 50}

average: 50, min 50, max 50

{49, 49, 49, 49, 50, 50, 50, 50}

average: 49.5 min 49, max 50

Justification:

$n * \min \leq n * \text{average} \leq n * \max$ so

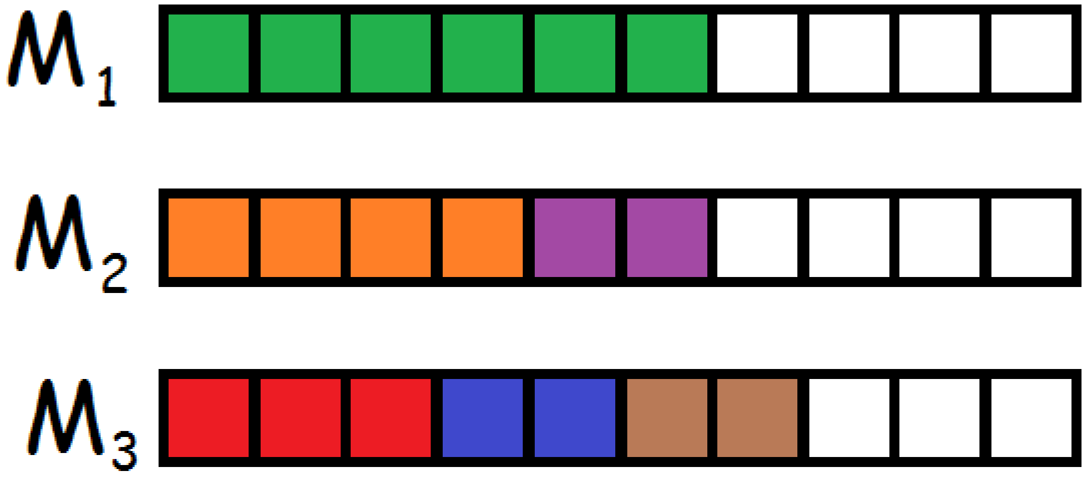
$\min \leq \text{average} \leq \max$.

The average time per machine will be

$(6 + 4 + 3 + 2 + 2 + 2)/3 = 19/3 = 6\frac{1}{3}$

The job times are integers so the makespan must be at least the ceiling of the average for any assignment.

Conclusion: this one is optimal.



makespan = 7

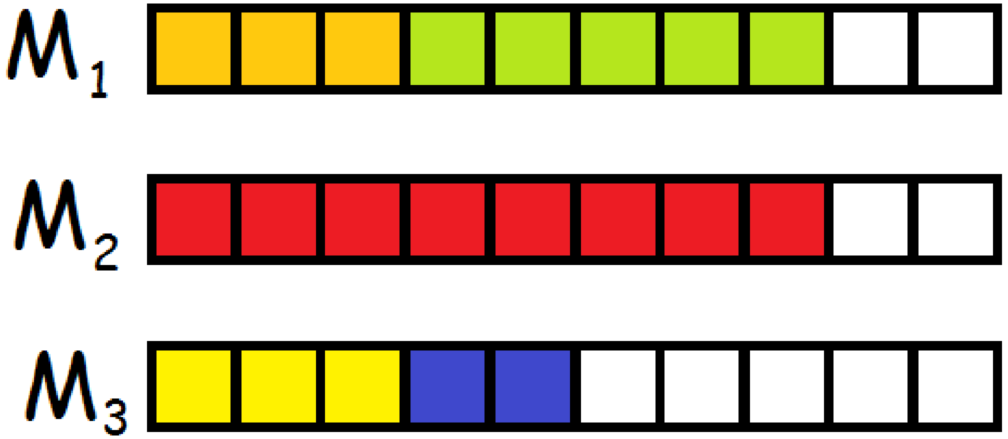
Goal: to show that for any arbitrary ordering of the job times, the resulting makespan is at most two times the optimal makespan.

Notation:
T= makespan of the greedy solution.
T^*= makespan of an optimal solution.

We want to prove that $T \leq 2\ T^*$.

Lemma 1: The makespan T^* of an optimal solution is at least $\text{Max}_{j=1 \text{ to } n}\, t_j$.

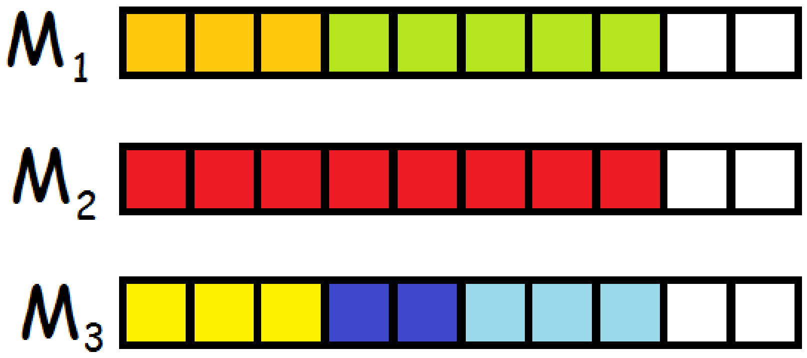
Proof: The machine i that is assigned a job j that has a maximum processing time t_j has T_i at least t_j and hence the makespan is at least t_j.



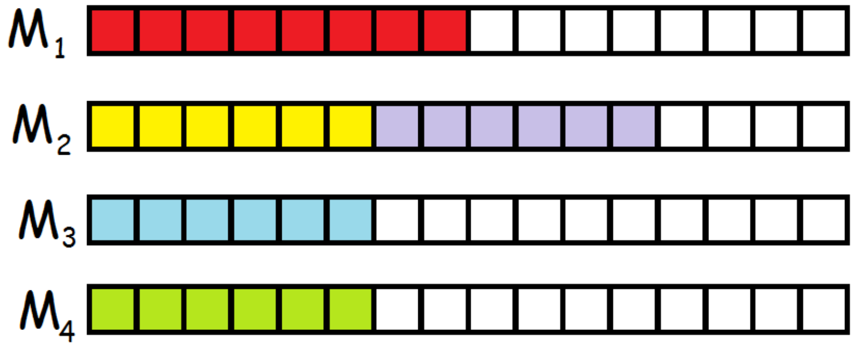
Lemma 2: The optimal makespan T* is at least as big as the average assignment time:

$$T^* \geq \frac{1}{m}\left(\sum_{j=1}^{n} t_j\right)$$

We divide by m since there are m processors.



T* = average value.



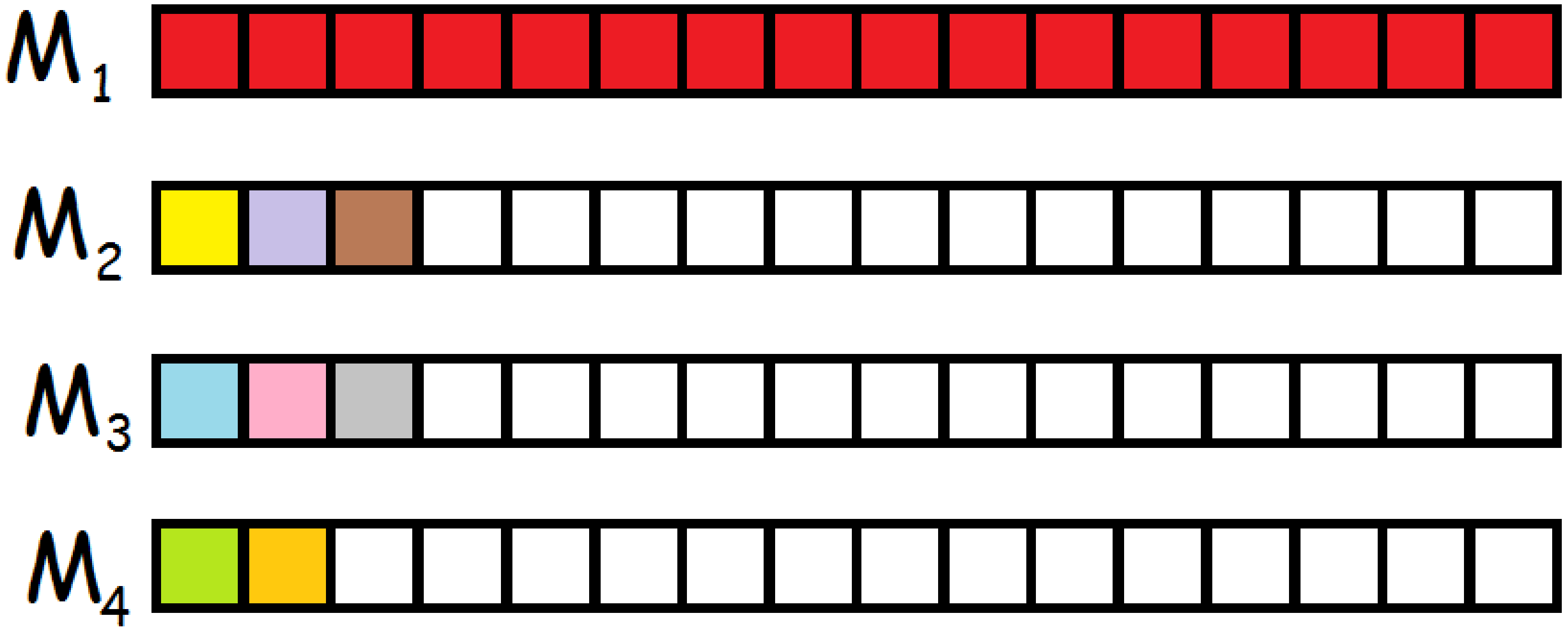
T* = 12 > average = 8.

A worse example:

Average=(16 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1)/4= 6

The makespan= 16.

But note: here our other bound on T^* is tight.



The two bounds:

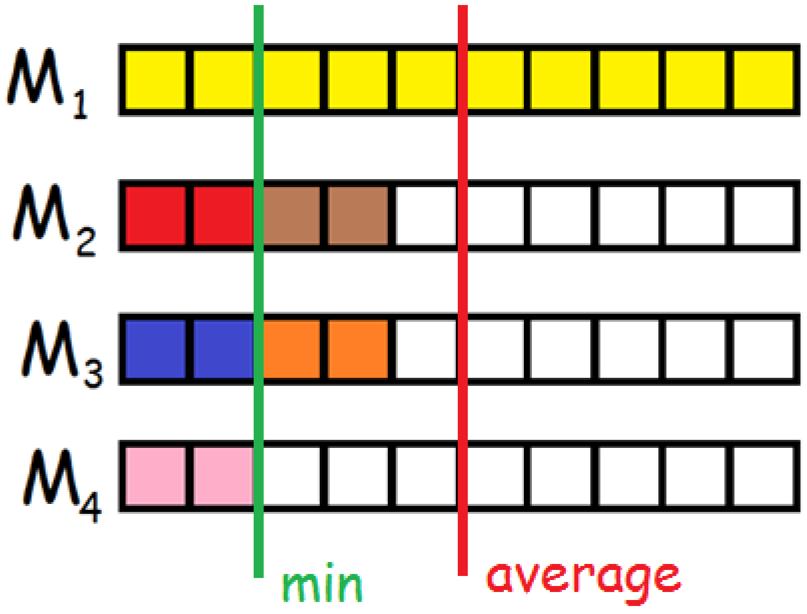
Lemma 1:
T^* is at least the maximum time of a job.

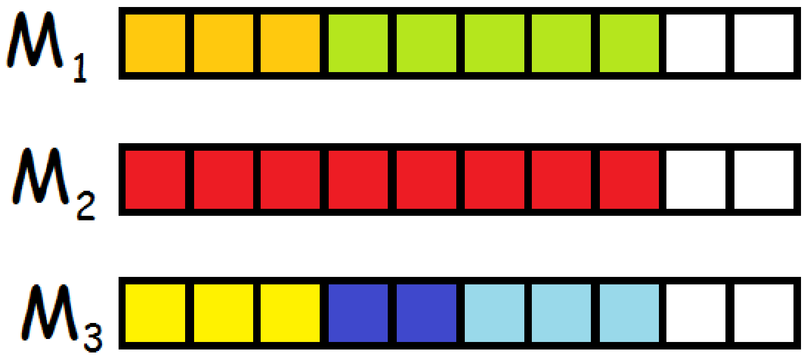
Lemma 2:
T^* is at least the average time per processor.

How can we use this to prove that the makespan T of any solution found by our greedy algorithm is at most $2\ T^*$?

Observations

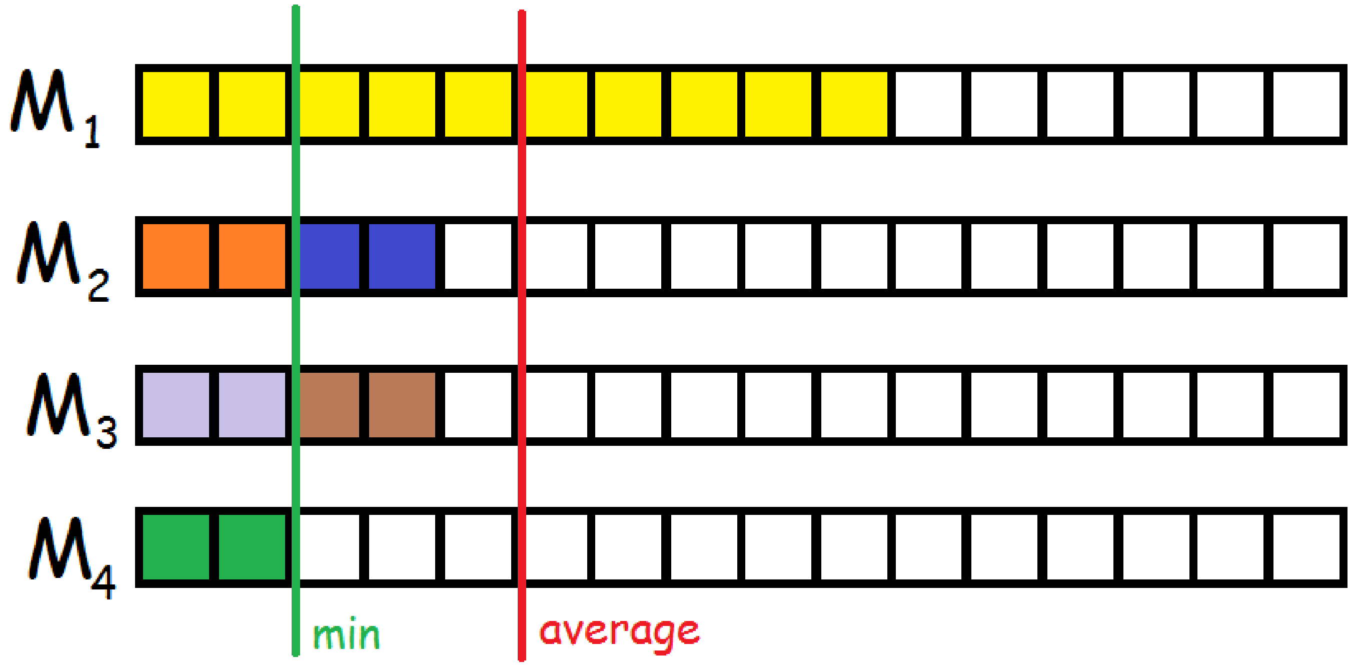
1. At every step of the greedy algorithm, the load of the processor i that is chosen to host job j has load that is at most the average load so far. [The average is always at least the min].
2. The average load so far is never more than the average load of the final solution.



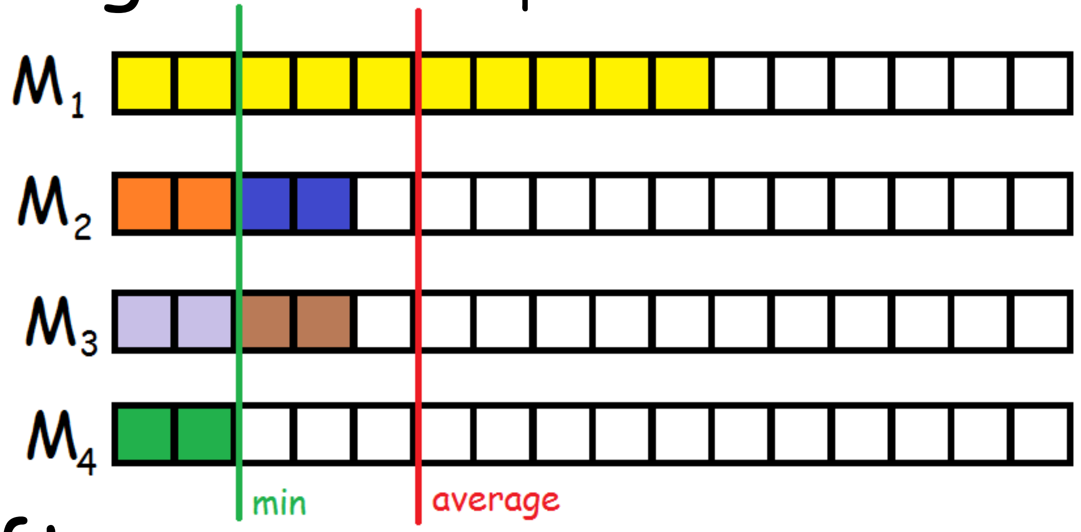


Equality can only occur when every machine has the same load.

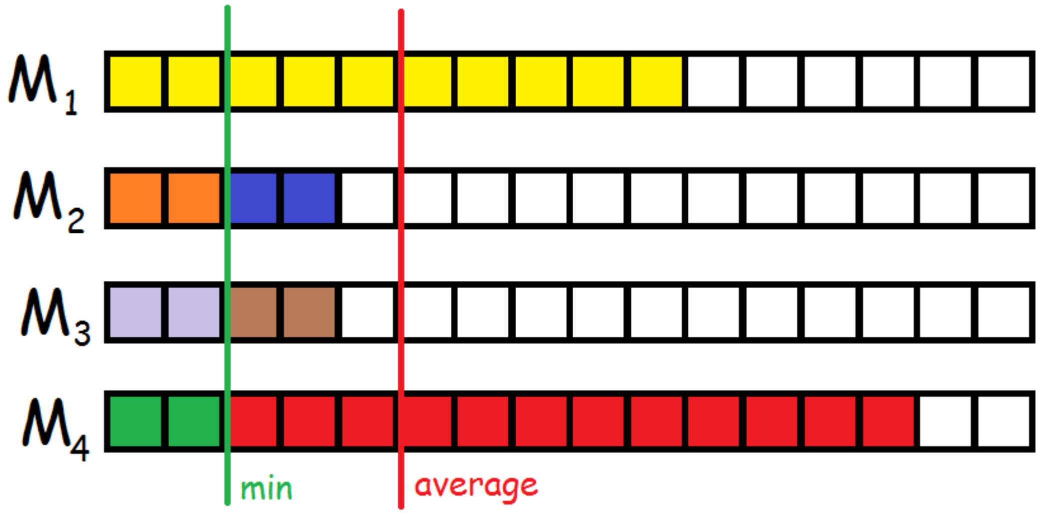
Let M_x be a processor that has maximum processing time in the greedy solution. Equivalently, the makespan T of the greedy solution is equal to T_x. We consider the load on x before its last job j was added to its assignment:



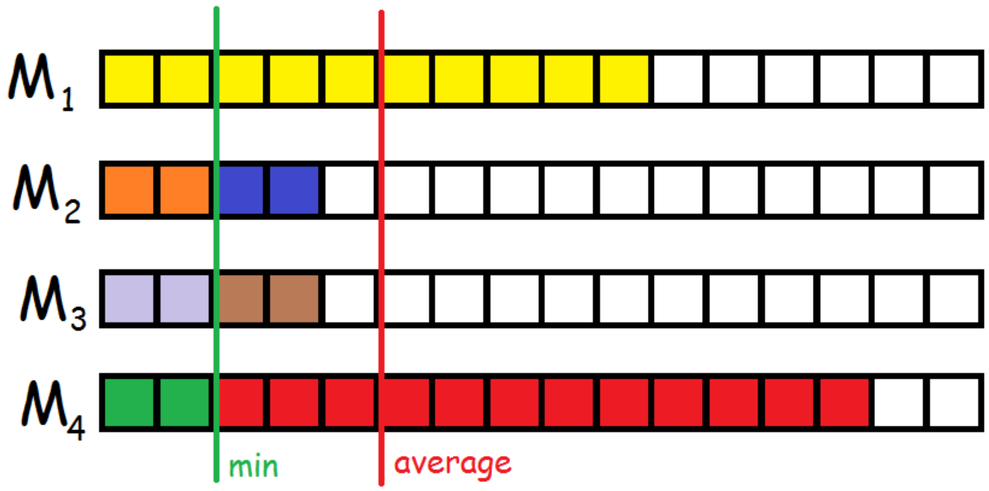
Before the last job given to M_4 is assigned to M_4:



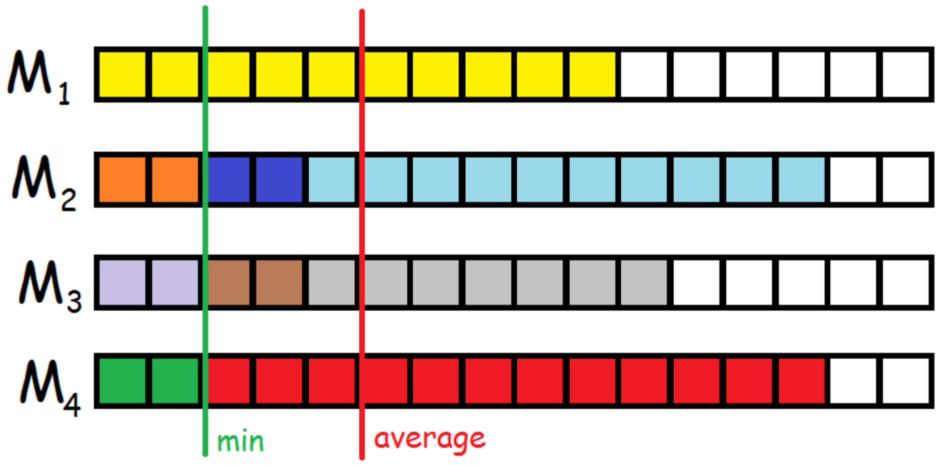
After:



Some other jobs can get assigned to other processors: the critical point is that for machine x, its load is equal to the makespan T when we are done.

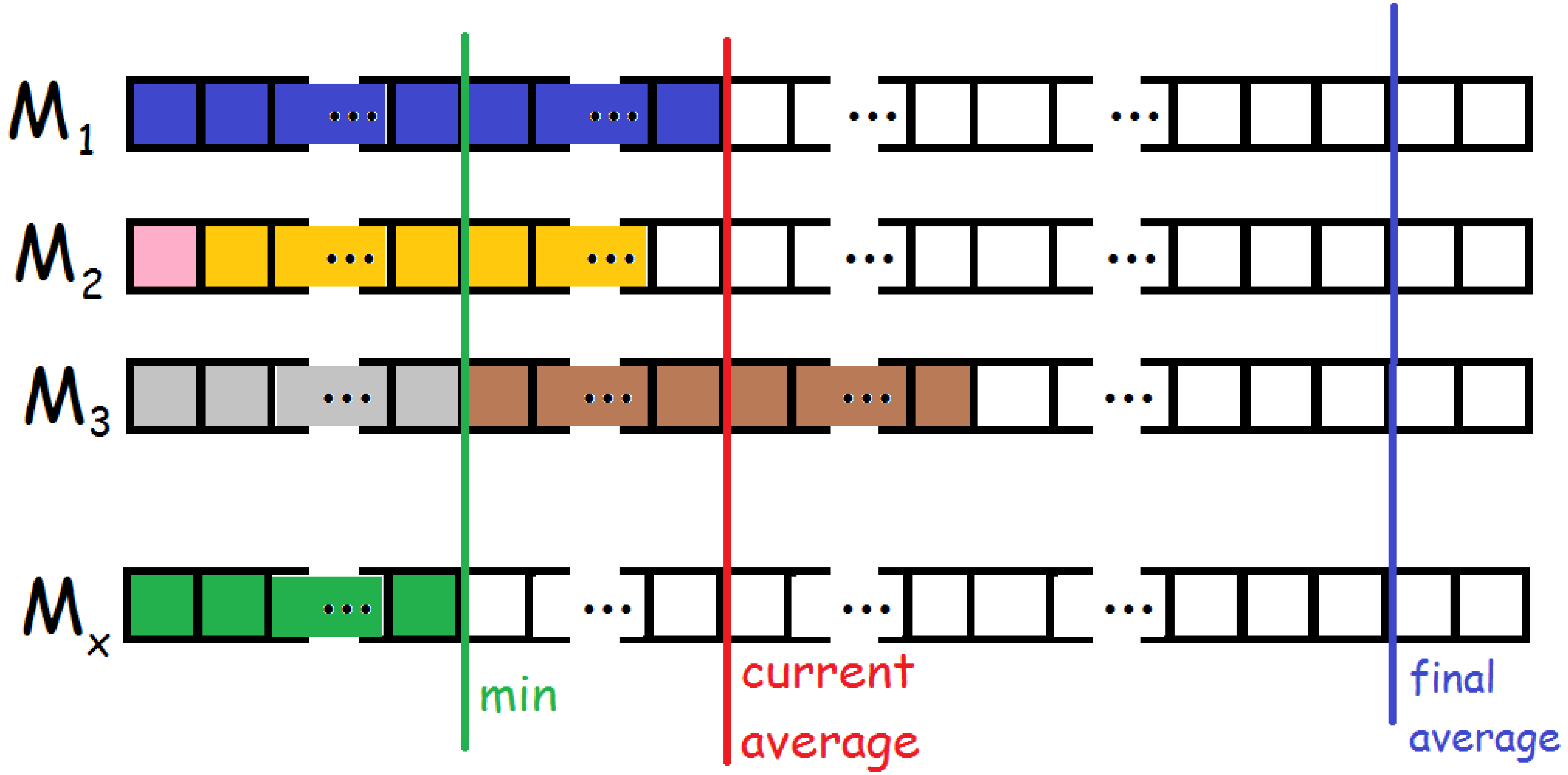


Just after the last job was assigned to machine M_4.



When all jobs have been assigned to processors.

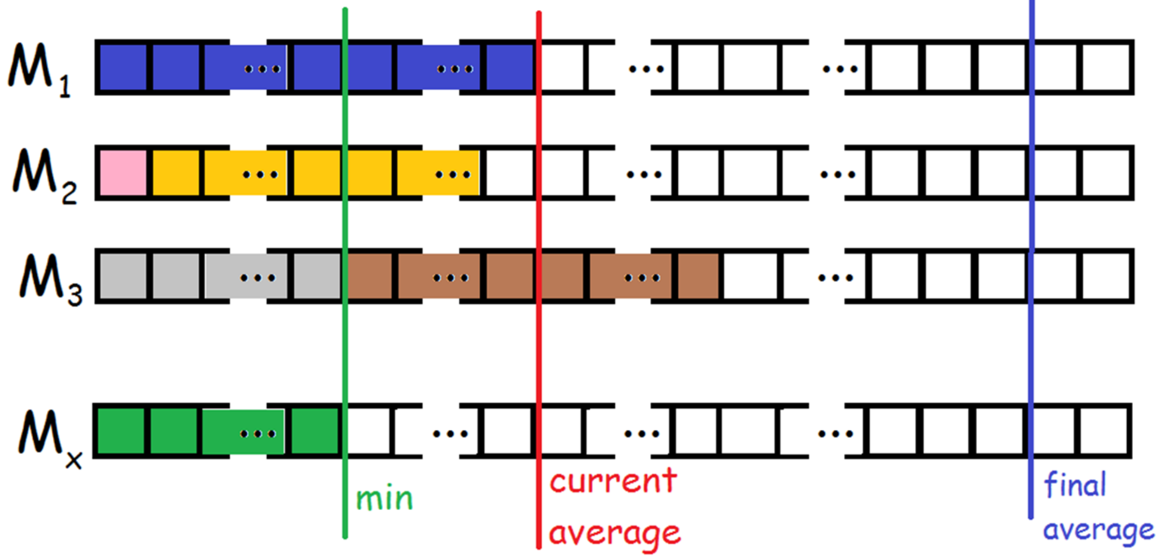
The generic picture:
Before the last job for M_x is assigned:



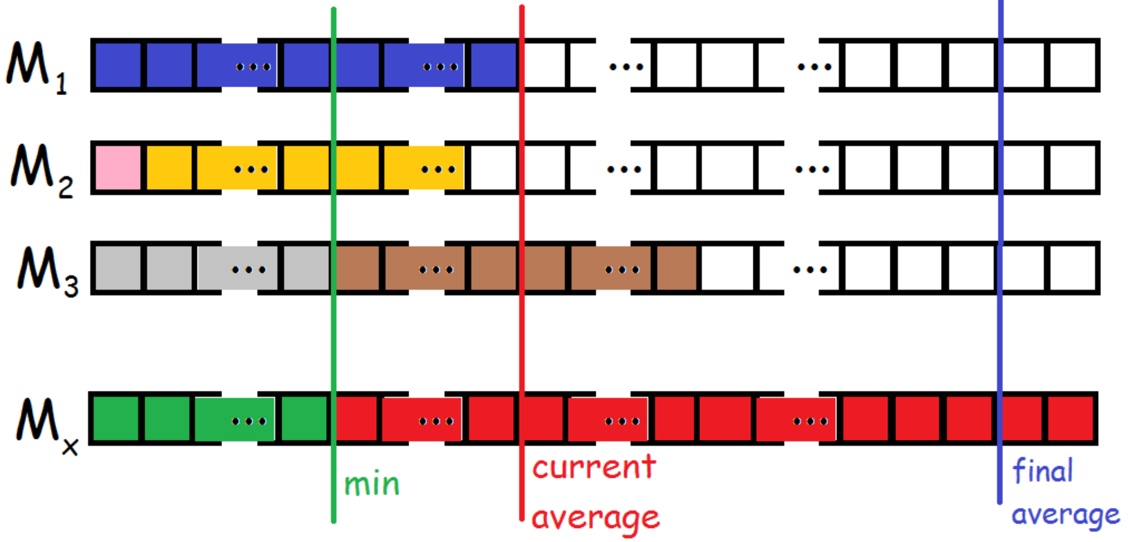
IMPORTANT POINT:

min ≤ current average ≤ final average

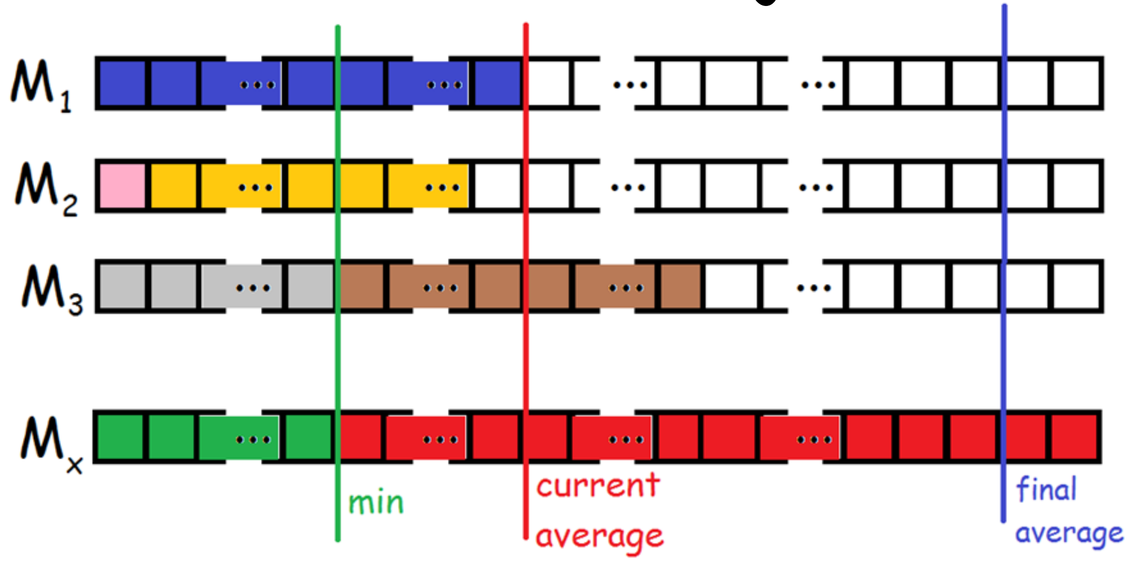
Before the last job for M_x is assigned:



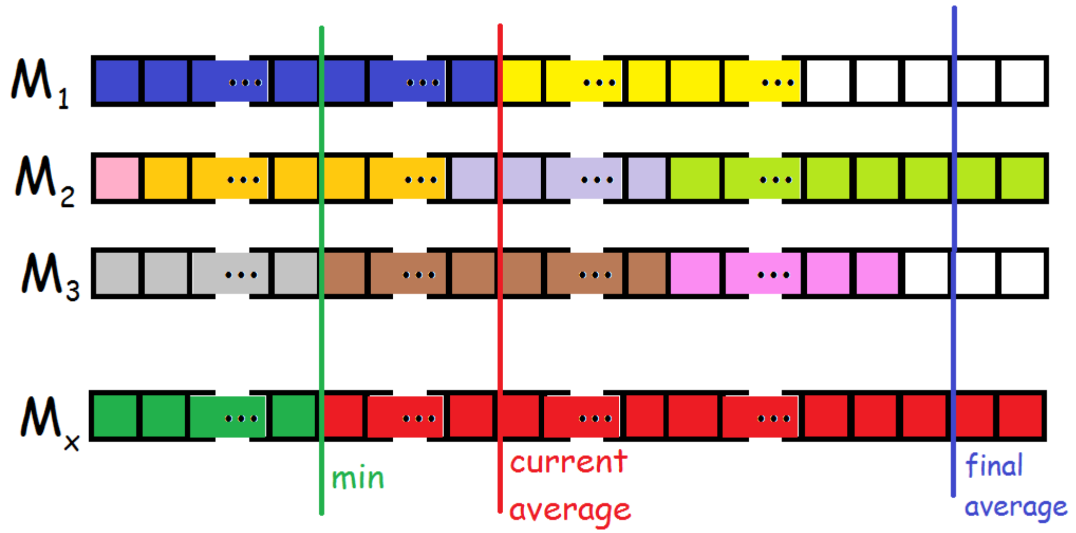
Just after the last job for M_x is assigned:

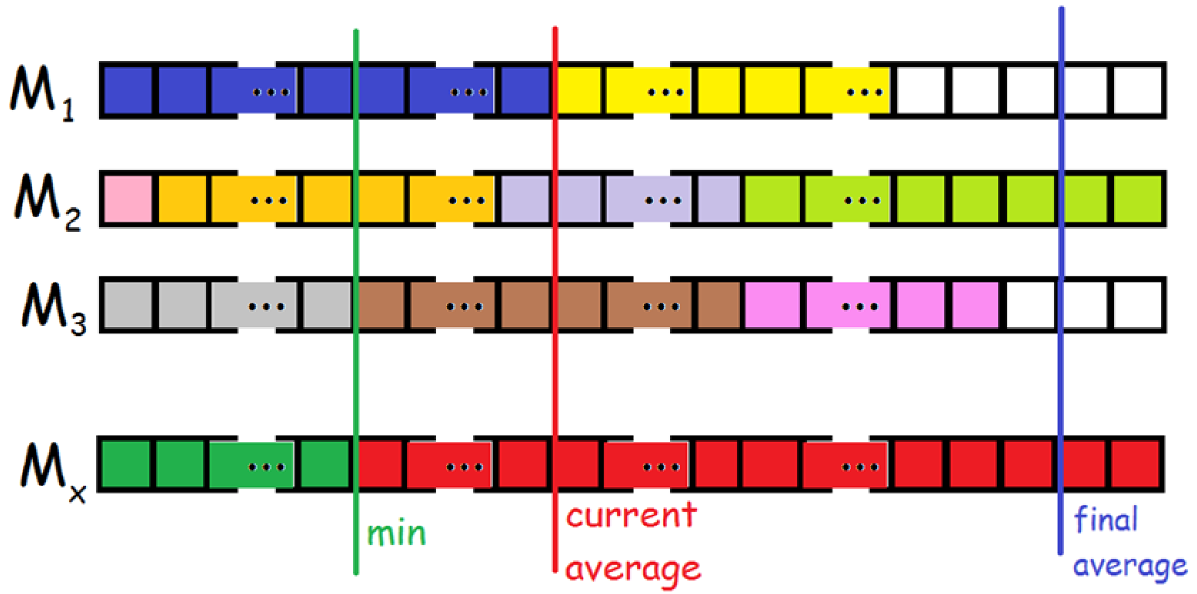


Just after the last job for M_x is assigned:



When we are done:





min ≤ current average ≤ final average

The load of Mx when we are done is equal to
min + time for the red job
which is at most
final average + time for the longest job
which by our lemmas is at most
$T^* + T^* = 2\,T^*$.

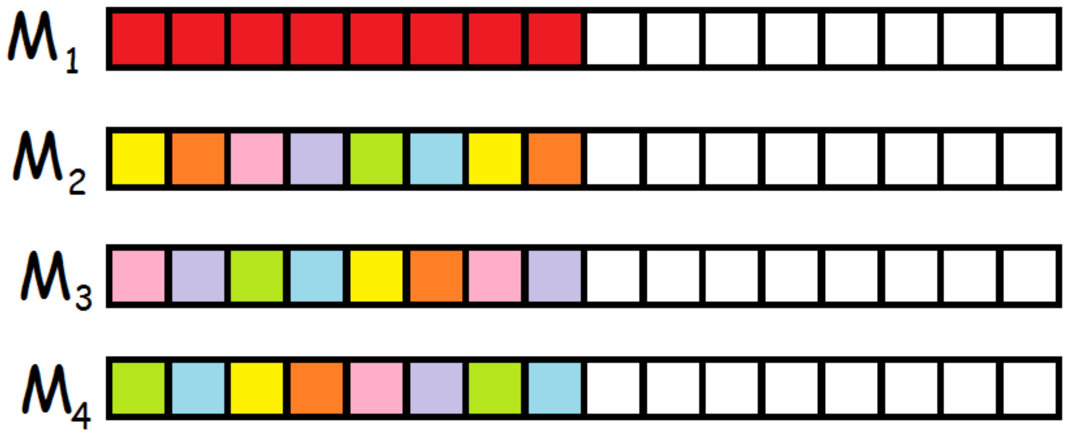
We have a bound on the greedy algorithm performance:

T = greedy makespan
T^* = optimal makespan
$T \leq 2T^*$

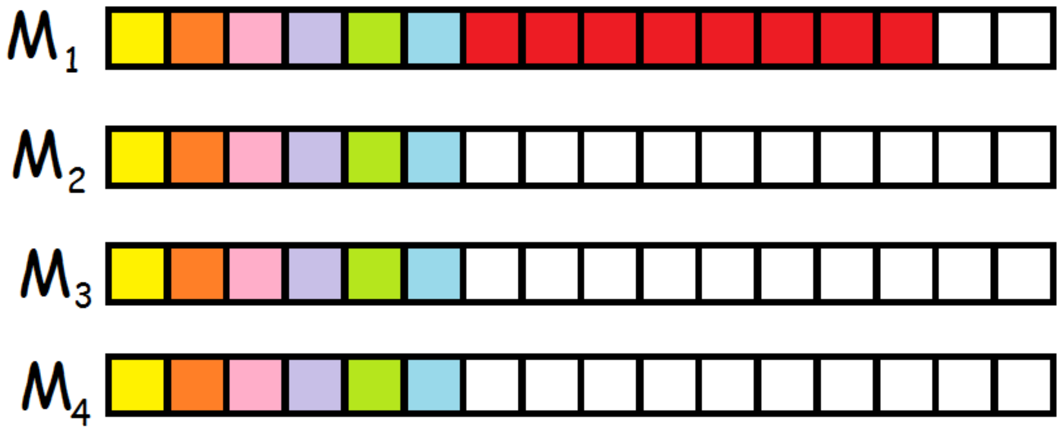
But are there examples that have greedy solutions that are twice as bad as the optimal solution (or close to it)?

Or is our analysis too sloppy to be tight?

How badly can things go wrong?

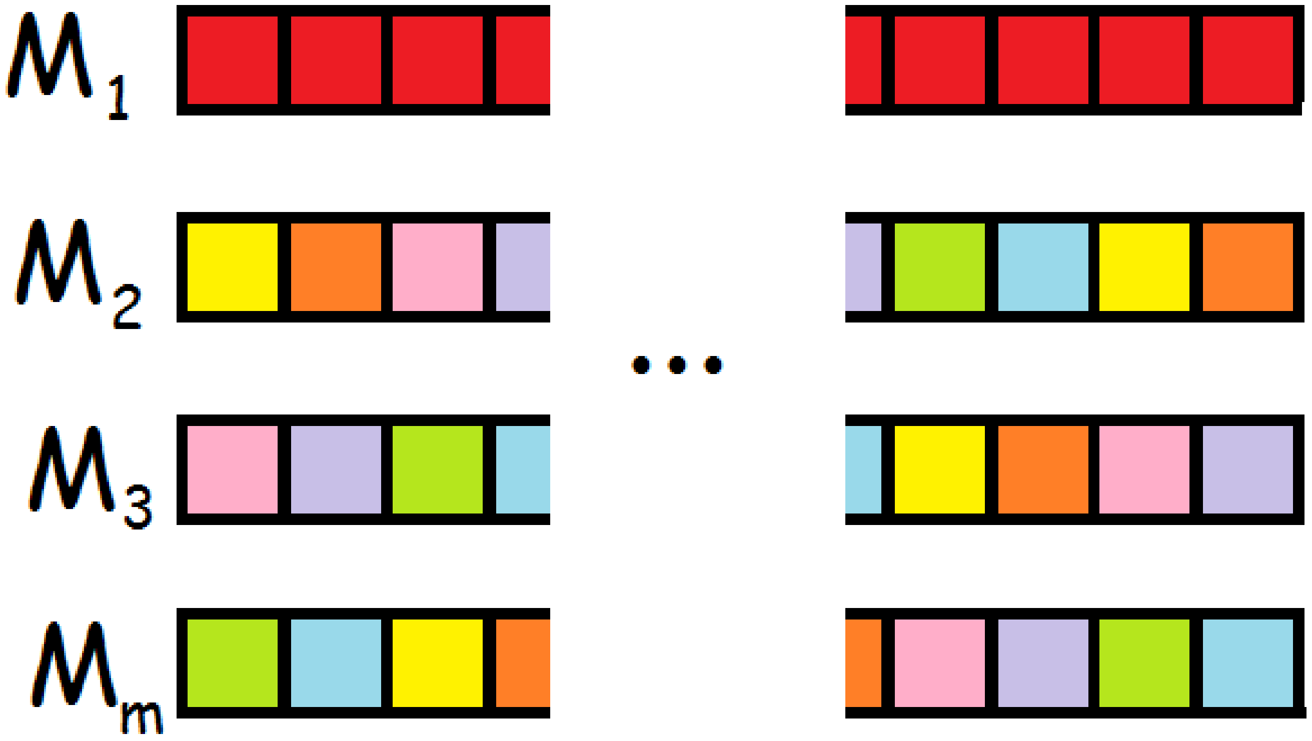


Optimal makespan is 8.



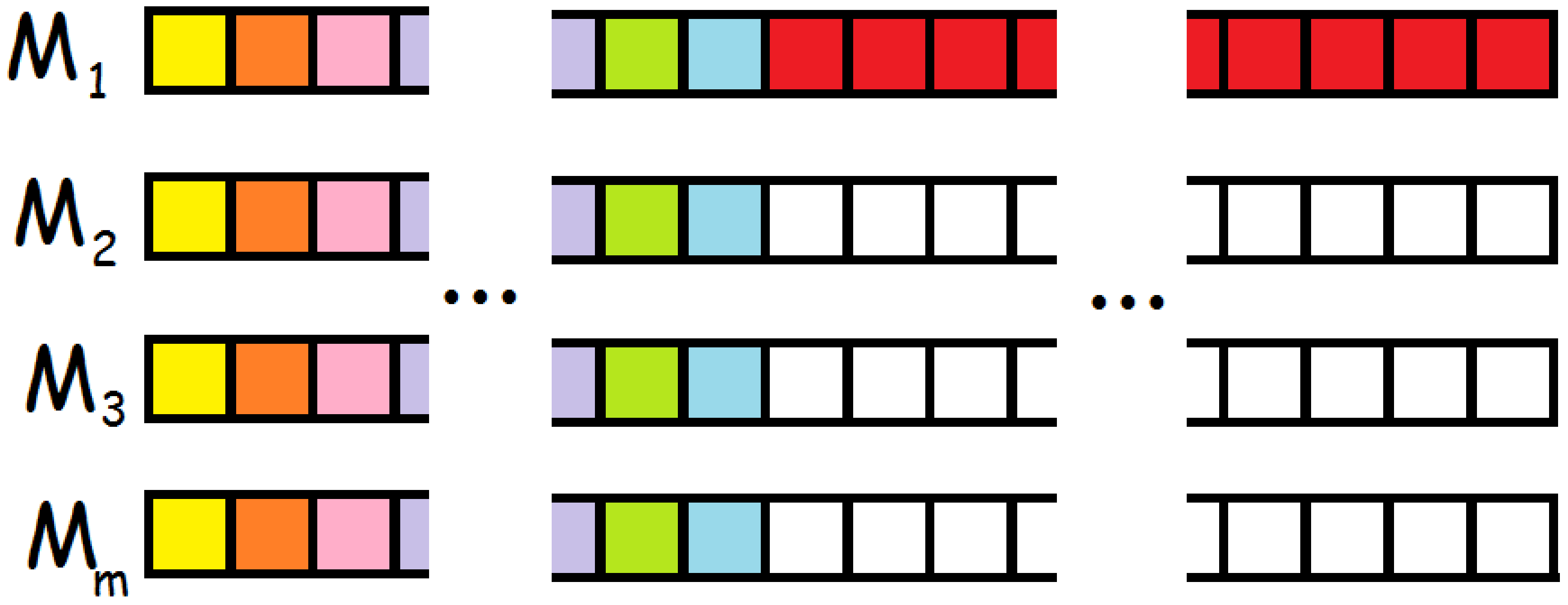
Greedy makespan is 14.

Optimal solution: makespan = m*k.



To ensure the number of problems of size 1 is divisible by m (allowing greedy to distribute evenly): (m-1) * (m* k) problems of size 1 and one problem of size (m*k).

Greedy solution.



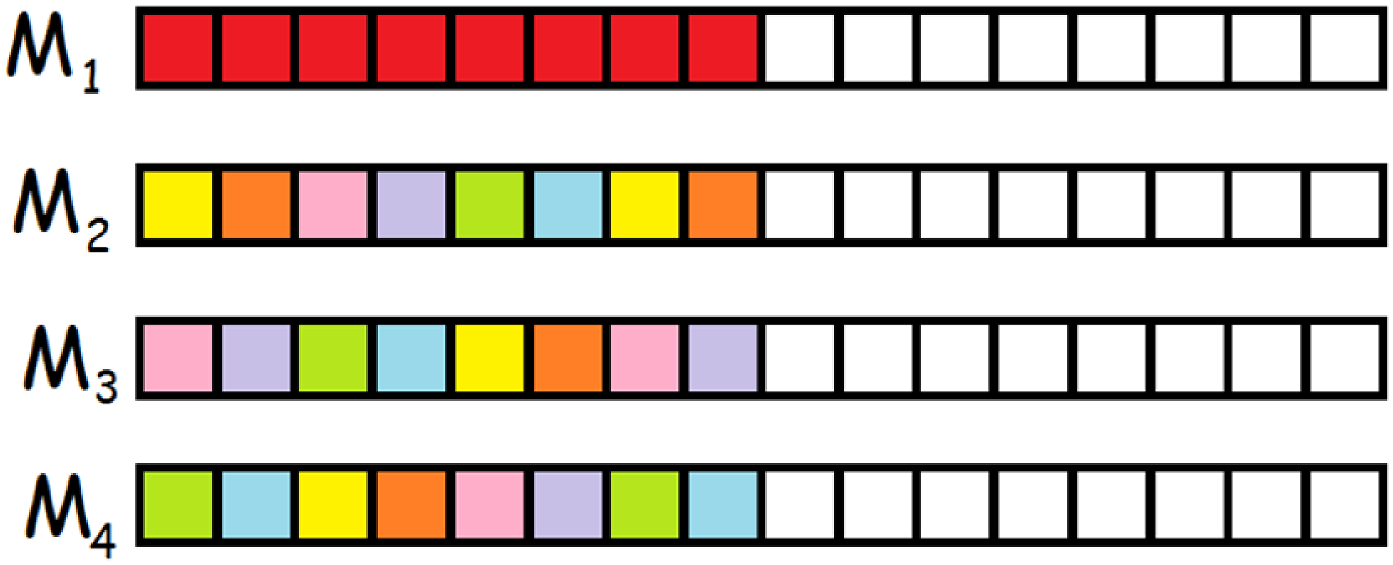
Number of problems:

Size 1: (m-1) * m*k,   Size m*k: 1

Makespan is k * (m-1) + m*k= 2mk - k

Ratio T/T^*= (2mk -k)/(mk) = 2 – 1/m ≈ 2

The worst case examples considered the jobs in increasing order of times.
Can we do better if we consider the jobs in the reverse order (decreasing order)?
The greedy algorithm gives an optimal solution for the worst case scenario we just examined:



Lemma 3: If we have at most m jobs, the greedy algorithm gives an optimal solution.

Proof: each job is placed on its own processor and hence the value of T^* is the length of a longest job, and we cannot do any better than that.

Lemma 4: If we have more than m jobs then the makespan of an optimal solution is at least 2 times the time of job m+1: $T^* \geq 2\, t_{m+1} \Longrightarrow t_{m+1} \leq T^*/2$.

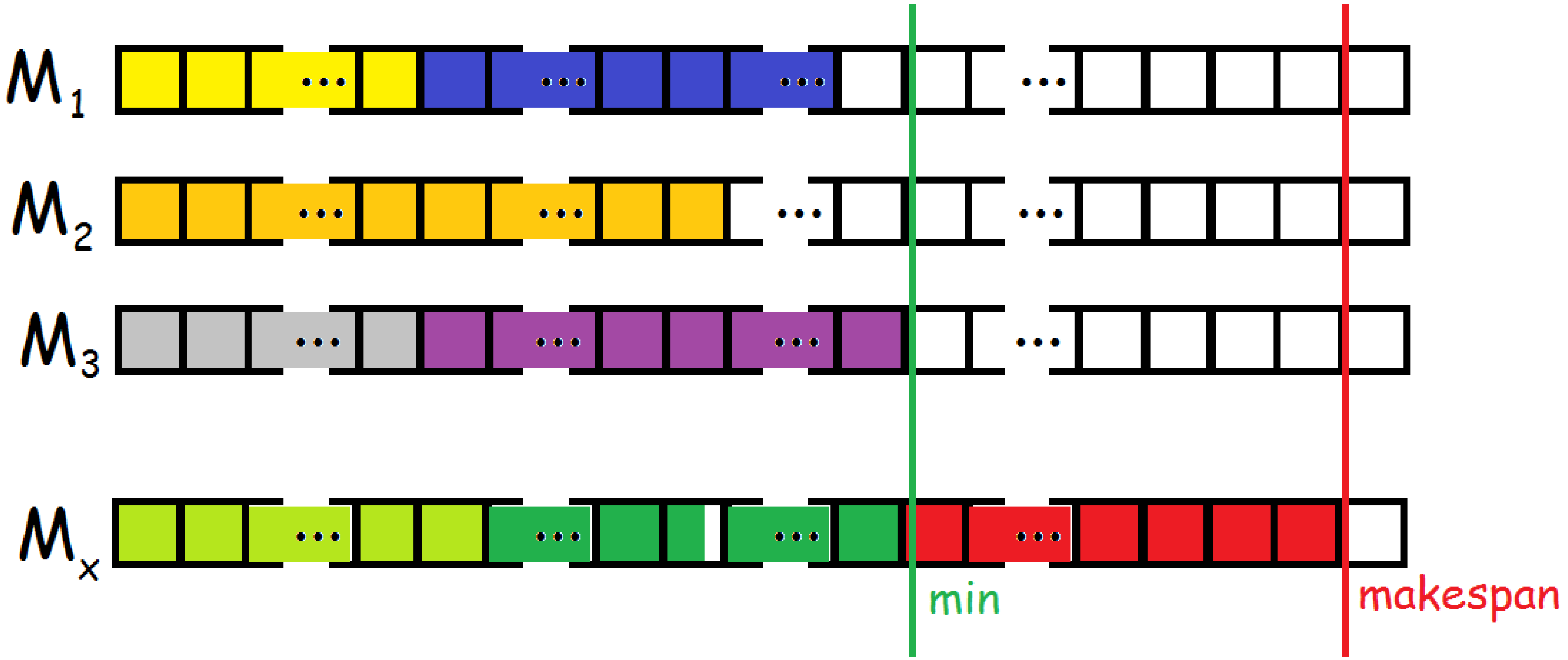
Proof [pigeonhole principle]

Consider only the first m+1 jobs. By the pigeonhole principle, in any assignment of jobs to machines, some machine must be assigned at least two of these jobs.

Since the job times are

$t_1 \geq t_2 \geq t_3 \geq \ldots \geq t_m \geq t_{m+1}$

the times for these two jobs are both at least t_{m+1}.



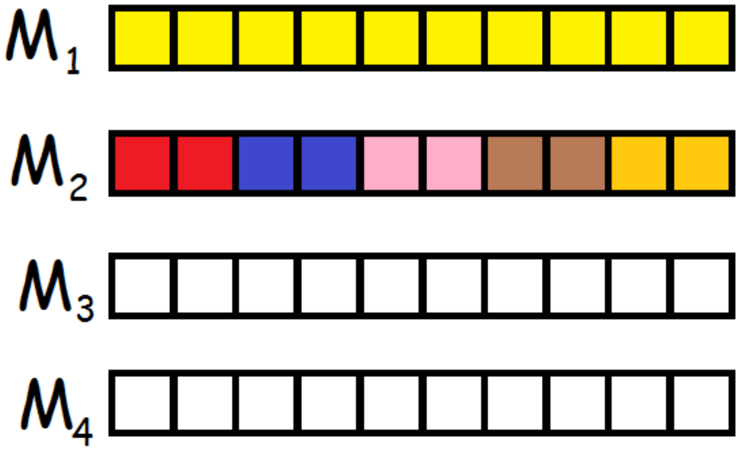
With more than m jobs:

$$T \le \text{average load} + \text{time of red job}$$

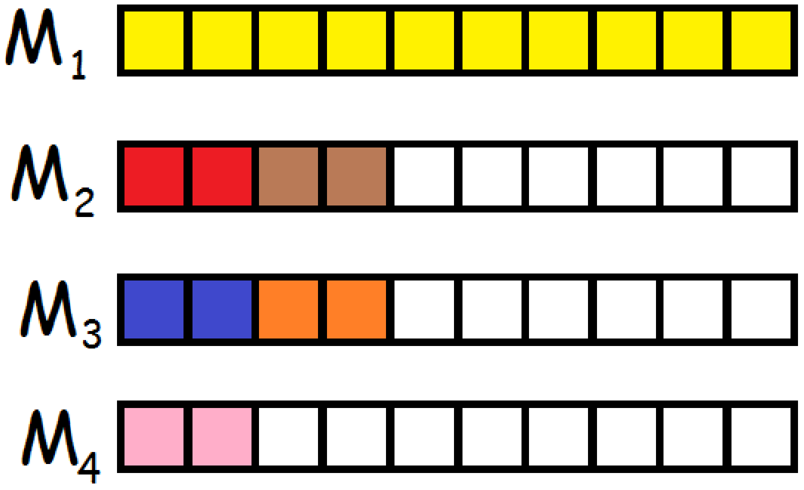
$$\le T^* + t_{m+1}$$

$$\le T^* + T^*/2$$

The problem statement says that we are trying to make the loads on the machines as "balanced as possible". But the formal statement of the problem is that we should minimize the maximum processing time of a machine. These are not the same! Input times: 10, 2, 2, 2, 2, 2.



An optimal solution that is not well-balanced.



A solution that is more balanced.

If you ever have to deal with customers to find out what they really want:
make sure that the expression of the problem is really precise.

Here the "informal statement" of the problem does not match the formal statement of the optimization problem.
Formal statement: Minimize makespan.
Informal statement: Balance machine loads.

What do they really want?

Some things I do not like about the notation:
Each job j …
The jobs are numbered from 1 to n and the time for the job numbered j is t_j.

If t_j is a very large job… t_j is a time not a job.

A(i)= set of jobs assigned to processor i.
This to me looks liked the notation for a function and not a set. A set is not a function.
A_i = set of jobs assigned to machine i.

Slides:

J(i)= set of jobs assigned to machine i.

It's nicer to not use both J and j.

T= $\max_i T_i$ or $T^* \geq \sum_j t_j$

Especially when you get to algorithm analysis, it is nice to see limits on sums and expressions like this:

$$T = \underset{i=1 to\ m}{\text{Max}}\ Ti$$

$$T^* \geq \sum_{i=1}^{n} t_j$$

If you can say things concisely in words it is often easier to understand than mathematical formulas.

It helps to illustrate the points you are making in the proof with examples.

It's easier to see which time slots are assigned to which jobs using the colors than if I tried to use numbers instead.