# Approximation Algorithms
# Greedy and Local Search

Advanced Algorithms

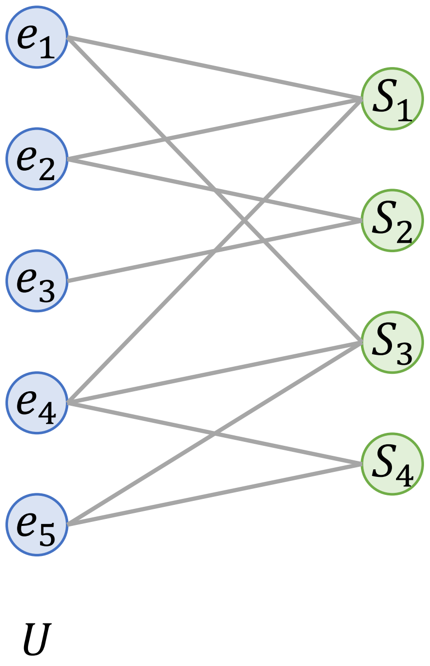
Nanjing University, Fall 2018

# Set Cover

**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

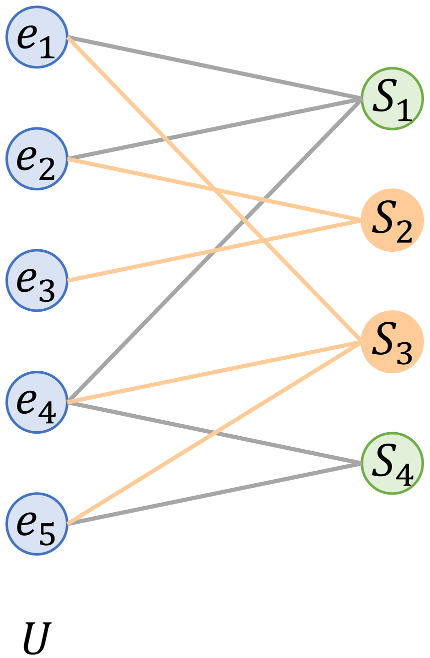
# Set Cover

**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.



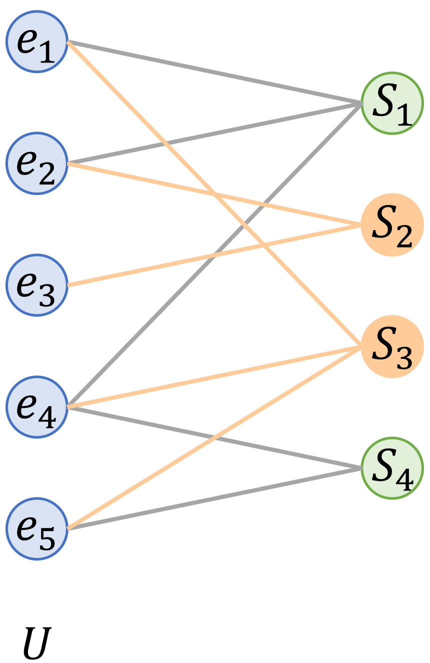
# Set Cover

**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.



# Set Cover

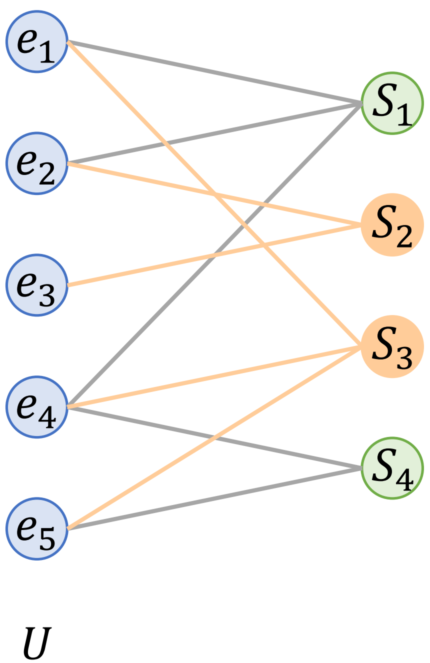
**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




- This problem is NP-hard!
- Decision version is one of Karp's 21 NP-complete problems.

# Set Cover

**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

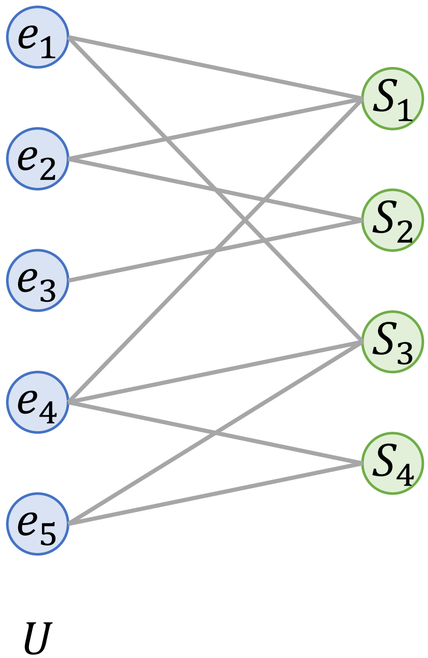



- This problem is NP-hard!
- Decision version is one of Karp's 21 NP-complete problems.

- Can we find good enough solutions efficiently?

# Set Cover

**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**<u>GreedyCover:</u>**

Set $C = \emptyset$.

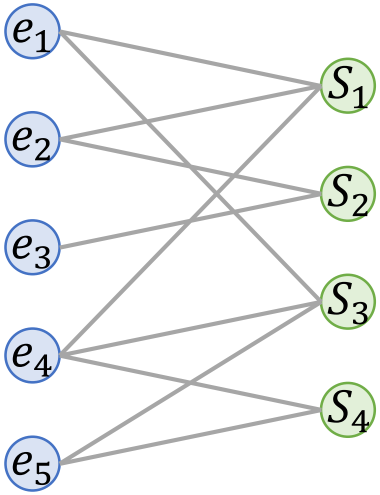
While $U \neq \emptyset$ do:

  Add $i$ with largest $|S_i \cap U|$ to $C$.

  $U = U - S_i$.

Return $C$.

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**GreedyCover:**

Set $C = \emptyset$.

While $U \neq \emptyset$ do:

  Add $i$ with largest $|S_i \cap U|$ to $C$.

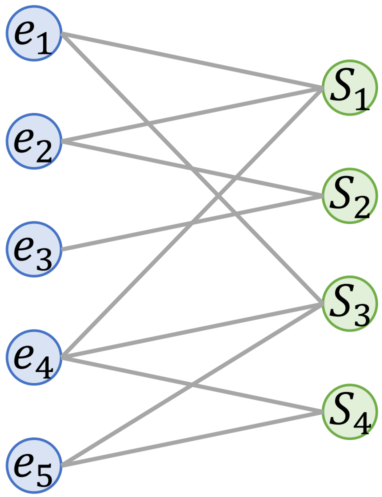
  $U = U - S_i$.

Return $C$.

$\mathrm{OPT}(I)$: value of minimum set cover of instance $I$

$\mathrm{SOL}(I)$: value of set cover returned by **GreedyCover** on instance $I$

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**GreedyCover:**

Set $C = \emptyset$.
While $U \neq \emptyset$ do:
    Add $i$ with largest $|S_i \cap U|$ to $C$.
    $U = U - S_i$.
Return $C$.

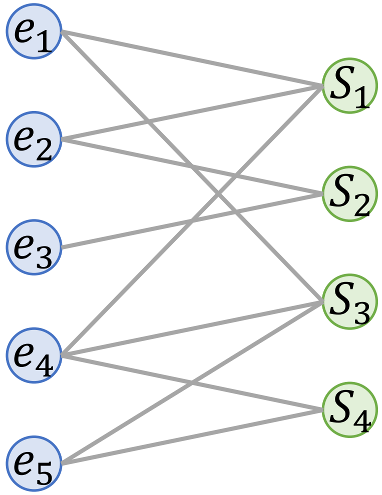
$\mathrm{OPT}(I)$: value of minimum set cover of instance $I$

$\mathrm{SOL}(I)$: value of set cover returned by **GreedyCover** on instance $I$

**GreedyCover** has *approximation ratio* $\alpha$ if

$$\forall \text{ instance } I, \quad \frac{\mathrm{SOL}(I)}{\mathrm{OPT}(I)} \leq \alpha$$

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**GreedyCover:**

Set $C = \emptyset$.
While $U \neq \emptyset$ do:
  Add $i$ with largest $|S_i \cap U|$ to $C$.
  $U = U - S_i$.
Return $C$.

$\mathrm{OPT}(I)$: value of minimum set cover of instance $I$

$\mathrm{SOL}(I)$: value of set cover returned by **GreedyCover** on instance $I$

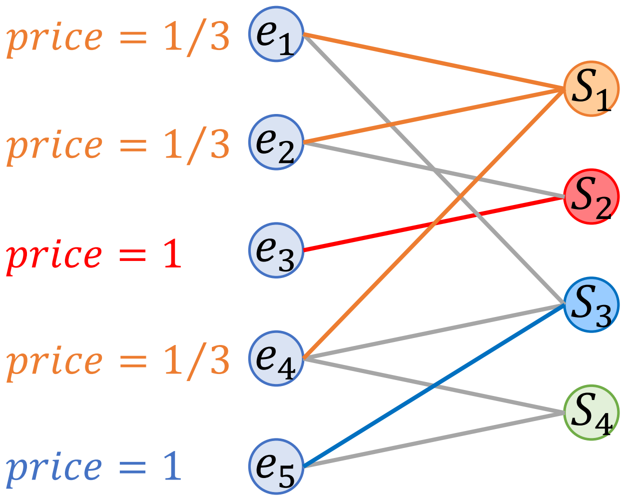
**GreedyCover** has *approximation ratio* $\alpha$ if

$$\forall \text{ instance } I, \quad \frac{\mathrm{SOL}(I)}{\mathrm{OPT}(I)} \leq \alpha$$

For minimization problems, we want $\mathrm{SOL}(I)/\mathrm{OPT}(I) \leq \alpha$ where $\alpha \geq 1$

For maximization problems, we want $\mathrm{SOL}(I)/\mathrm{OPT}(I) \geq \alpha$ where $\alpha \leq 1$

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.



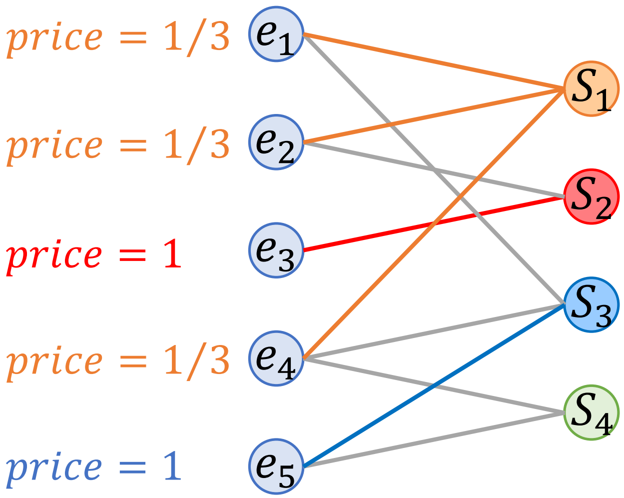

**GreedyCover:**

Set $C = \emptyset$.
While $U \neq \emptyset$ do:
  Add $i$ with largest $|S_i \cap U|$ to $C$.
  Set $price(e) = \frac{1}{|S_i \cap U|}$ for all $e \in S_i \cap U$.
  $U = U - S_i$.
Return $C$.

$$|C| = \sum_{e \in U} price(e)$$

- Initially, there must exist some subset that covers its elements with price at most $\mathrm{OPT}(I)/n$.
- Therefore, price of elements in the first subset covered by **GreedyCover** is at most $\mathrm{OPT}(I)/n$.
- After $k$ elements in $t$ subsets are covered by **GreedyCover**, there must exist some subset such that the price of its uncovered elements is at most $\mathrm{OPT}(I_t)/(n-k) \leq \mathrm{OPT}(I)/(n-k)$.
- In general, **GreedyCover** pays at most $\mathrm{OPT}(I)/(n-k+1)$ to cover the $k^{\text{th}}$ chosen element.

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**GreedyCover:**

Set $C = \emptyset$.

While $U \neq \emptyset$ do:

  Add $i$ with largest $|S_i \cap U|$ to $C$.

  Set $price(e) = \frac{1}{|S_i \cap U|}$ for all $e \in S_i \cap U$.

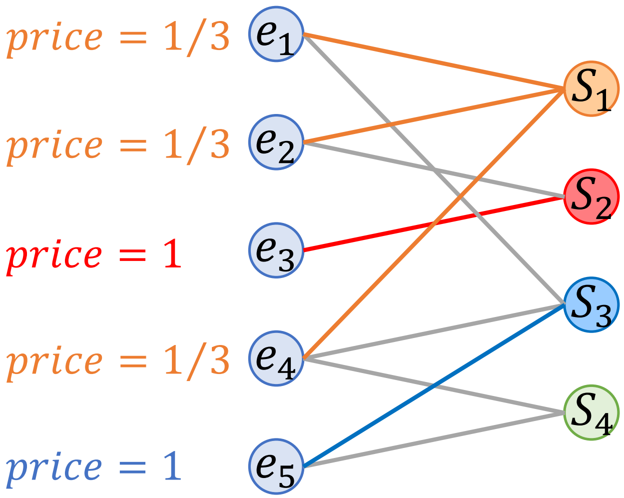
  $U = U - S_i$.

Return $C$.

Enumerate $e_k$ in the order in which they are covered by **GreedyCover**:

$$price(e_k) \leq \frac{\mathrm{OPT}(I)}{n - k + 1}$$

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




**GreedyCover:**

Set $C = \emptyset$.

While $U \neq \emptyset$ do:

Add $i$ with largest $|S_i \cap U|$ to $C$.

Set $price(e) = \frac{1}{|S_i \cap U|}$ for all $e \in S_i \cap U$.

$U = U - S_i$.

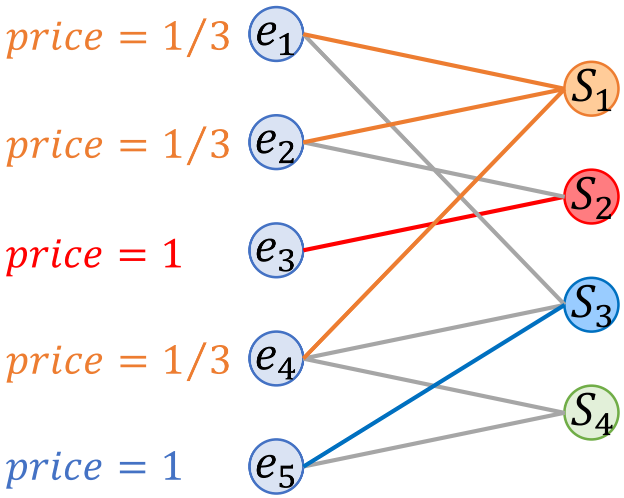
Return $C$.

Enumerate $e_k$ in the order in which they are covered by **GreedyCover**:

$$price(e_k) \leq \frac{\mathrm{OPT}(I)}{n-k+1}$$

$$|C| = \sum_{e \in U} price(e) \leq \sum_{k=1}^{n} \frac{\mathrm{OPT}(I)}{n-k+1} = H_n \cdot \mathrm{OPT}(I)$$

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

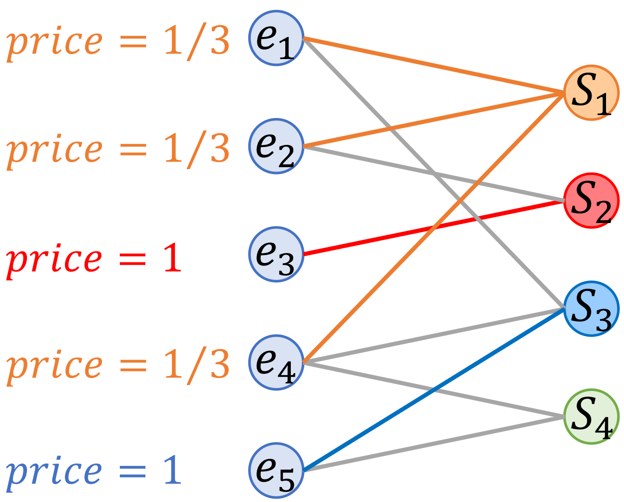



**GreedyCover:**

Set $C = \emptyset$.
While $U \neq \emptyset$ do:
  Add $i$ with largest $|S_i \cap U|$ to $C$.
  Set $price(e) = \frac{1}{|S_i \cap U|}$ for all $e \in S_i \cap U$.
  $U = U - S_i$.
Return $C$.

- **GreedyCover** has approximation ratio $H_n \approx \ln n + O(1)$.

**Set Cover Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.



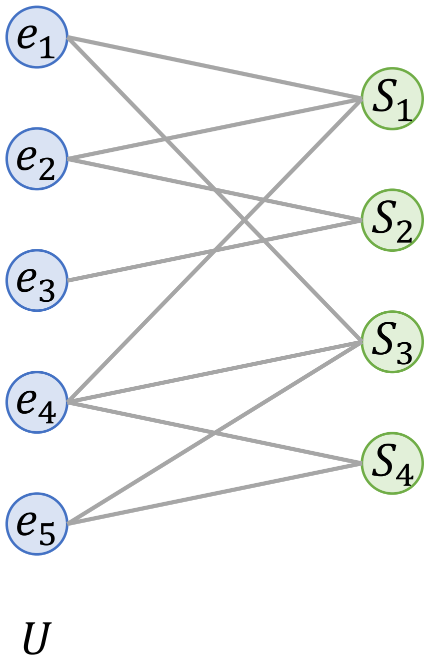

**GreedyCover:**

Set $C = \emptyset$.
While $U \neq \emptyset$ do:
    Add $i$ with largest $|S_i \cap U|$ to $C$.
    Set $price(e) = \frac{1}{|S_i \cap U|}$ for all $e \in S_i \cap U$.
    $U = U - S_i$.
Return $C$.

- **GreedyCover** has approximation ratio $H_n \approx \ln n + O(1)$.
- [Lund, Yannakakis 1994; Feige 1998] There is no poly-time $\left(1 - o(1)\right) \ln(n)$ approx. algorithm unless NP = quasi-poly-time.
- [Ras, Safra 1997] For some constant $c$, there is no poly-time $c \ln(n)$ approx. algorithm unless NP = P.
- [Dinur, Steuer 2014] There is no poly-time $\left(1 - o(1)\right) \ln(n)$ approx. algorithm unless NP = P.

# Set Cover

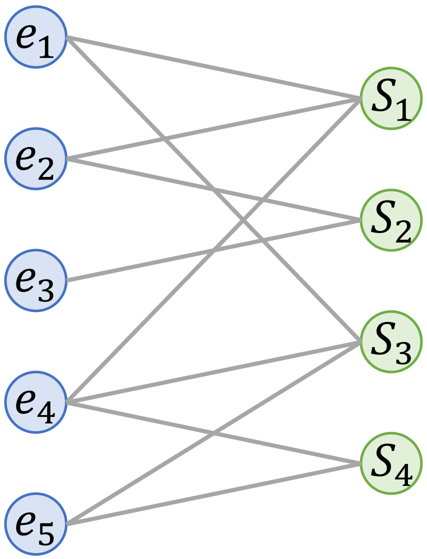
**Instance:** Given a collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$, find the smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.




- This problem is NP-hard.
- We have $O(\ln n)$ approx. alg.

- *Frequency* of an element:
  # of subsets the element is in.
- Use $f_I$ to denote the frequency of the most frequent element in instance $I$.

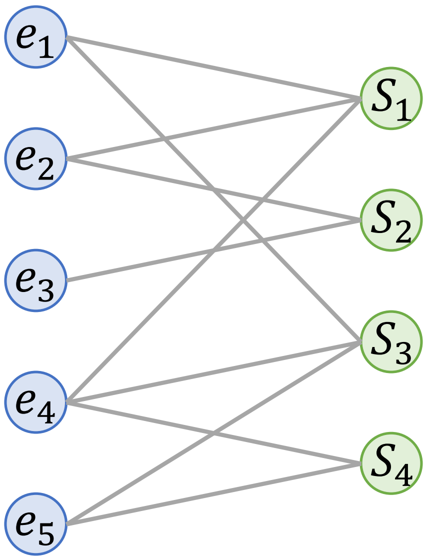
**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.



**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.

**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

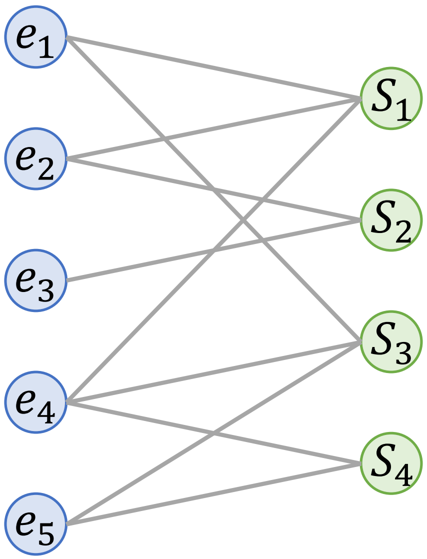
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.




Since every $e \in M$ must consume a subset to cover

$\forall C, \forall M$: $|M| \leq |C|$

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.

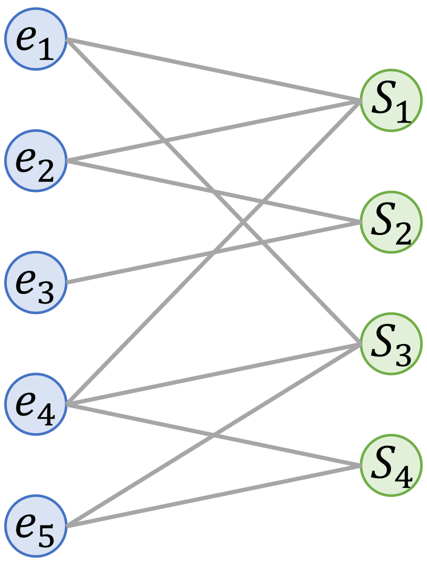



Since every $e \in M$ must consume a subset to cover

$$\forall C, \forall M: |M| \leq |C|$$

As a result, $\forall M: |M| \leq \mathrm{OPT}_{\mathrm{primal}} = \min |C|$

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.




Since every $e \in M$ must consume a subset to cover

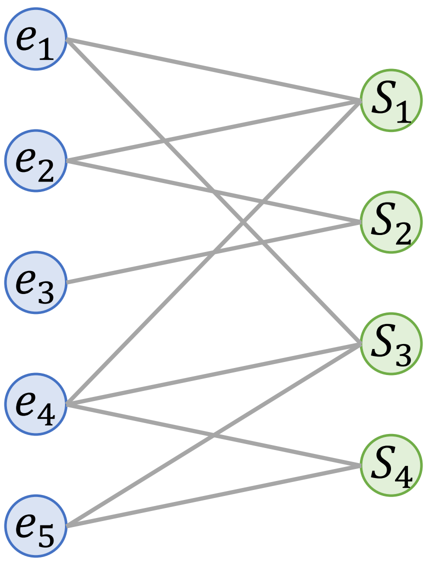
$$\forall C, \forall M: |M| \leq |C|$$

As a result, $\forall M: |M| \leq \mathrm{OPT}_{\mathrm{primal}} = \min |C|$

**GreedyMatchingCover:**

Find arbitrary maximal $M$ for the dual problem.
Return $C = \{i: S_i \cap M \neq \emptyset\}$.

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \le 1$ for all $i \in [m]$.




Since every $e \in M$ must consume a subset to cover
$$\forall C, \forall M: |M| \le |C|$$

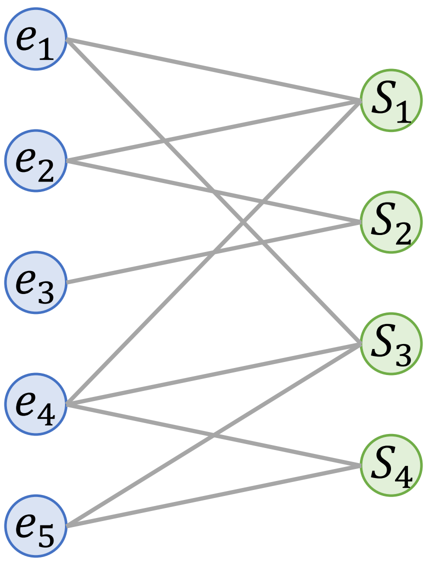
As a result, $\forall M: |M| \le \mathrm{OPT}_{\mathrm{primal}} = \min |C|$

**GreedyMatchingCover:**

Find arbitrary maximal $M$ for the dual problem.
Return $C = \{i : S_i \cap M \ne \emptyset\}$.

Since $M$ is maximal, returned $C$ must be a cover.

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.




Since every $e \in M$ must consume a subset to cover

$$\forall C, \forall M: |M| \leq |C|$$

As a result, $\forall M: |M| \leq \mathrm{OPT}_{\mathrm{primal}} = \min |C|$

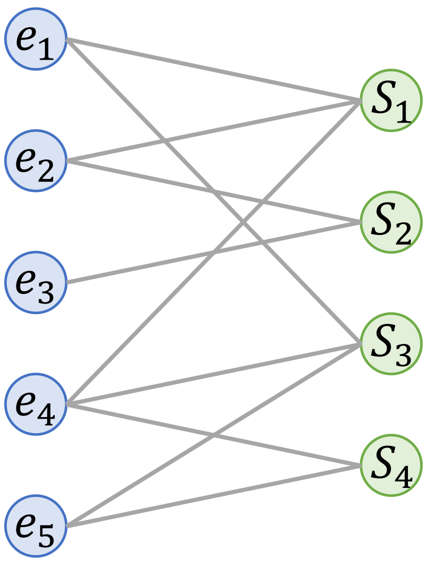
**GreedyMatchingCover:**

Find arbitrary maximal $M$ for the dual problem.
Return $C = \{i: S_i \cap M \neq \emptyset\}$.

Since $M$ is maximal, returned $C$ must be a cover.

$$|C| \leq f_I \cdot |M| \leq f_I \cdot \mathrm{OPT}_{\mathrm{primal}}$$

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.




Since every $e \in M$ must consume a subset to cover

$$\forall C, \forall M: |M| \leq |C|$$

As a result, $\forall M: |M| \leq \mathrm{OPT}_{\mathrm{primal}} = \min |C|$

**GreedyMatchingCover:**

Find arbitrary maximal $M$ for the dual problem.
Return $C = \{i: S_i \cap M \neq \emptyset\}$.

Since $M$ is maximal, returned $C$ must be a cover.

$$|C| \leq f_I \cdot |M| \leq f_I \cdot \mathrm{OPT}_{\mathrm{primal}}$$

**GreedyMatchingCover** has approximation ratio $f_I$.

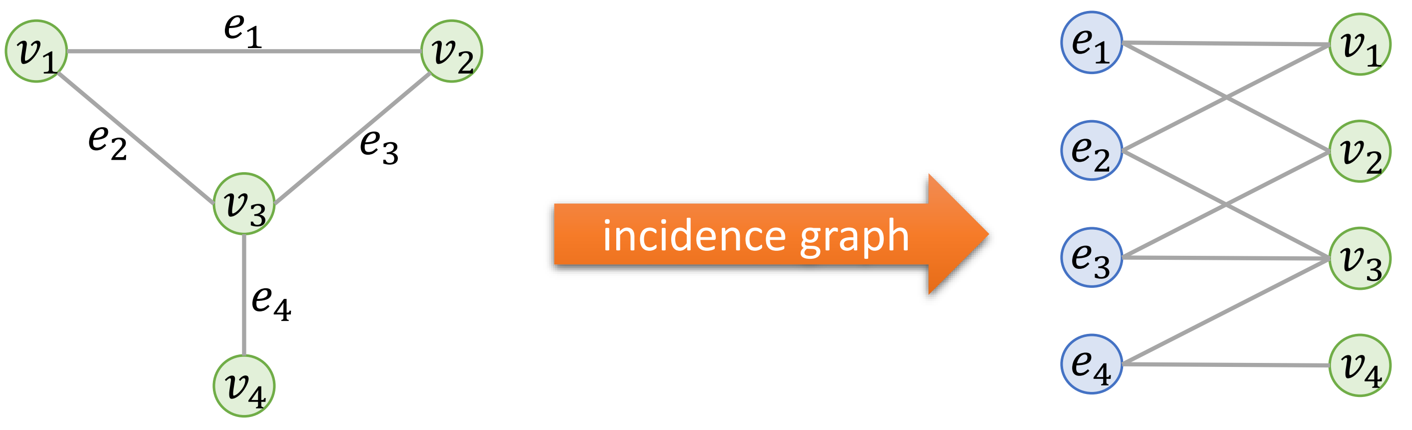
**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Set Cover:** Find smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

What if the frequency of each element is exactly 2?

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.

**Set Cover:** Find smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

What if the frequency of each element is exactly 2?



# Vertex Cover

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Set Cover:** Find smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

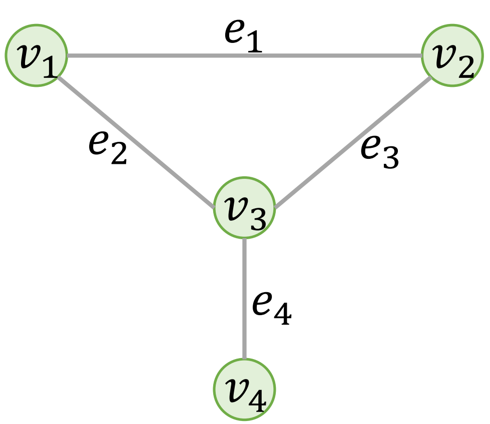
What if the frequency of each element is exactly 2?


**Instance:** An undirected simple graph $G = (V, E)$.
**Vertex Cover:** Find smallest $C \subseteq V$ s.t. $\forall e \in E: e \cap C \neq \emptyset$.

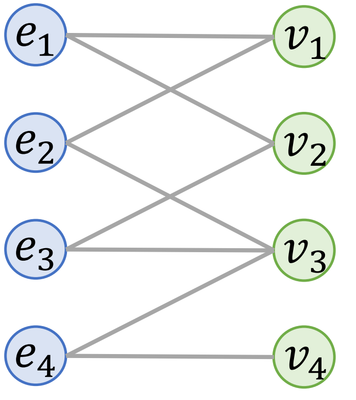
# Vertex Cover

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Set Cover:** Find smallest $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.

What if the frequency of each element is exactly 2?




incidence graph



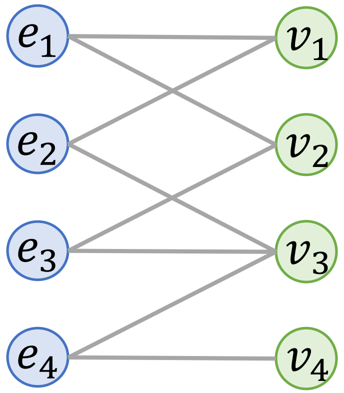

**Instance:** An undirected simple graph $G = (V, E)$.
**Vertex Cover:** Find smallest $C \subseteq V$ s.t. $\forall e \in E: e \cap C \neq \emptyset$.

- Vertex cover is also NP-hard.
- Decision version is one of Karp's 21 NP-complete problems.

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.

The frequency of each element is exactly 2

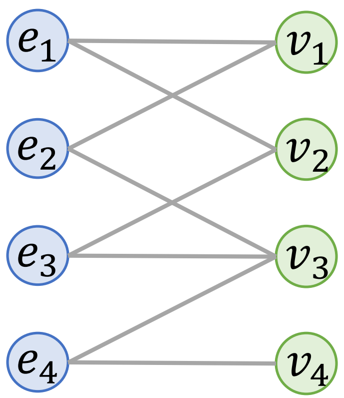
**Instance:** An undirected simple graph $G = (V, E)$.
**Primal:** Find $C \subseteq V$ s.t. $\forall e \in E$: $e \cap C \neq \emptyset$. (Vertex Cover)
**Dual:** Find $M \subseteq E$ s.t. $\forall v \in V$: $|v \cap M| \leq 1$. (Matching)



**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.

The frequency of each element is exactly 2

**Instance:** An undirected simple graph $G = (V, E)$.
**Primal:** Find $C \subseteq V$ s.t. $\forall e \in E$: $e \cap C \neq \emptyset$. (Vertex Cover)
**Dual:** Find $M \subseteq E$ s.t. $\forall v \in V$: $|v \cap M| \leq 1$. (Matching)




A 2-approximation algorithm for the vertex cover problem

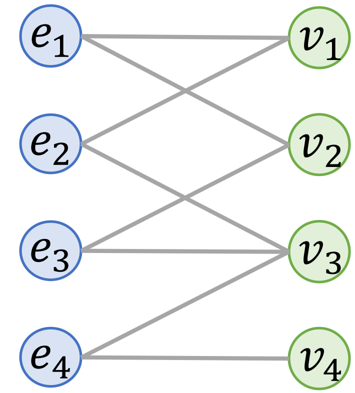
**GreedyMatchingCover:**

Find arbitrary maximal matching $M$ of the input graph.
Return $C = \{v : v \in V \text{ and } v \cap M \neq \emptyset\}$.

**Instance:** A collection of subsets $S_1, S_2, \cdots, S_m \subseteq U$.
**Primal:** Find $C \subseteq [m]$ such that $\bigcup_{i \in C} S_i = U$.
**Dual:** Find $M \subseteq U$ such that $|S_i \cap M| \leq 1$ for all $i \in [m]$.

The frequency of each element is exactly 2

**Instance:** An undirected simple graph $G = (V, E)$.
**Primal:** Find $C \subseteq V$ s.t. $\forall e \in E$: $e \cap C \neq \emptyset$. (Vertex Cover)
**Dual:** Find $M \subseteq E$ s.t. $\forall v \in V$: $|v \cap M| \leq 1$. (Matching)




A 2-approximation algorithm for the vertex cover problem

**GreedyMatchingCover:**

Find arbitrary maximal matching $M$ of the input graph.
Return $C = \{v : v \in V \text{ and } v \cap M \neq \emptyset\}$.

- There is no poly-time <1.36-approx. alg. unless P = NP.
- Assuming the unique game conjecture, there is no poly-time (2-ε)-approx. alg.

# Scheduling

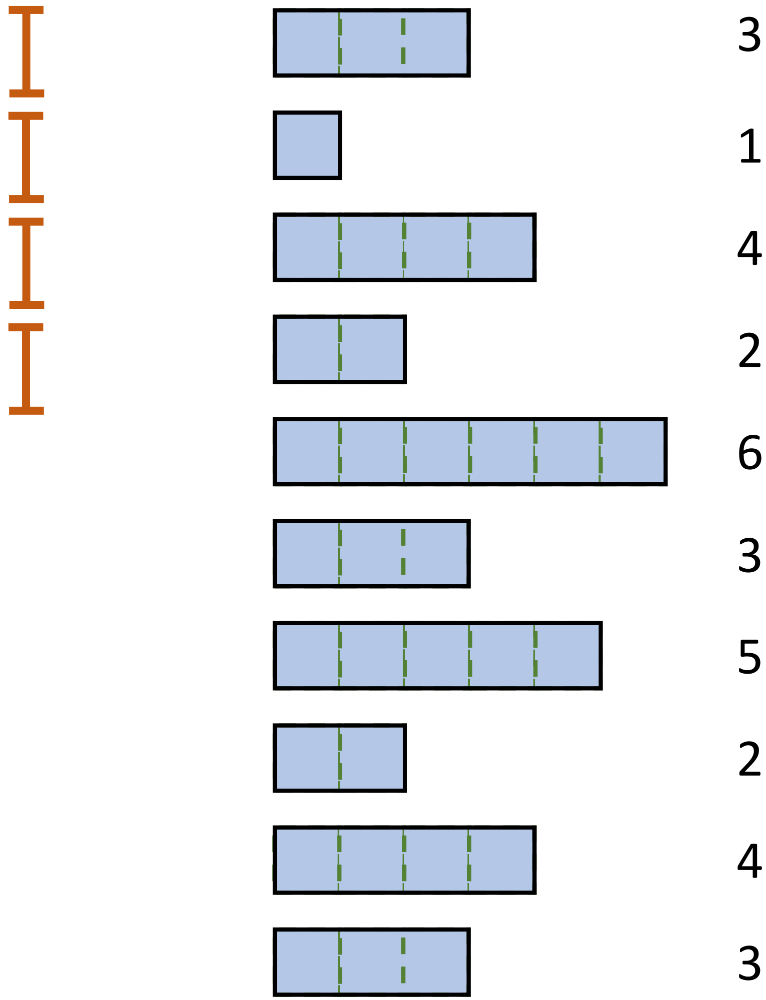
$m$ identical machines

# Scheduling

$m$ identical machines

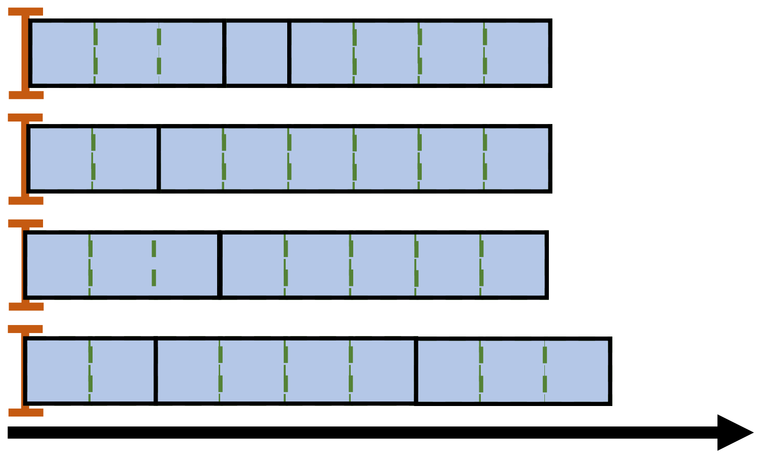
$n$ jobs

processing time $p_j$



# Scheduling

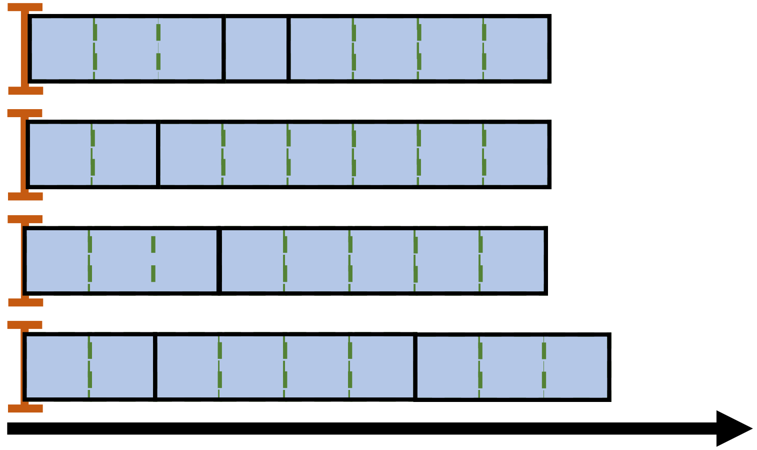
$m$ machines



$n$ jobs each with processing time $p_j$

# Scheduling

$m$ machines

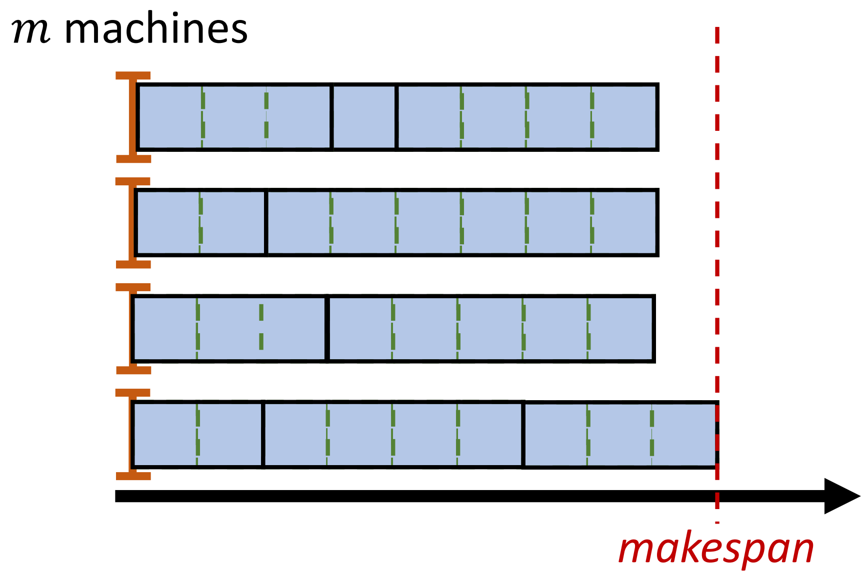


$n$ jobs each with processing time $p_j$

Completion time:
(of machine $i$)

$$C_i = \sum_{j\text{: jobs assigned to machine } i} p_j$$

# Scheduling




$n$ jobs each with processing time $p_j$

**Completion time:** (of machine $i$)

$$C_i = \sum_{j:\text{ jobs assigned to machine } i} p_j$$

**Makespan:**

$$C_{\max} = \max_i C_i$$

**Instance:** $n$ jobs $j = 1,2,\cdots,n$ each with processing time $p_j \in \mathbb{Z}^+$.

**Problem:** Find a schedule assigning $n$ jobs to $m$ identical machines so as the minimize the makespan.

**Instance:** $n$ jobs $j = 1,2, \cdots, n$ each with processing time $p_j \in \mathbb{Z}^+$.
**Problem:** Find a schedule assigning $n$ jobs to $m$ identical machines so as the minimize the makespan.

- "minimum makespan on identical machines"
- Scheduling problem has many variations:
  machines could be different, jobs could have release-dates/deadlines, etc…

**Instance:** $n$ jobs $j = 1,2,\cdots,n$ each with processing time $p_j \in \mathbb{Z}^+$.
**Problem:** Find a schedule assigning $n$ jobs to $m$ identical machines so as the minimize the makespan.

- "minimum makespan on identical machines"
- Scheduling problem has many variations:
  machines could be different, jobs could have release-dates/deadlines, etc…

If $m = 2$, the scheduling problem can be used to solve the partition problem!

**Instance:** $n$ positive integers $x_1, x_2, \cdots, x_n \in \mathbb{Z}^+$.
**Problem:** Determine whether there exists a partition of $\{1,2,\cdots,n\}$ into two sets $A$ and $B$ such that $\sum_{i\in A} x_i = \sum_{i\in B} x_i$.

**Instance:** $n$ jobs $j = 1,2,\cdots,n$ each with processing time $p_j \in \mathbb{Z}^+$.
**Problem:** Find a schedule assigning $n$ jobs to $m$ identical machines so as the minimize the makespan.

- "minimum makespan on identical machines"
- Scheduling problem has many variations:
  machines could be different, jobs could have release-dates/deadlines, etc…

If $m = 2$, the scheduling problem can be used to solve the partition problem!

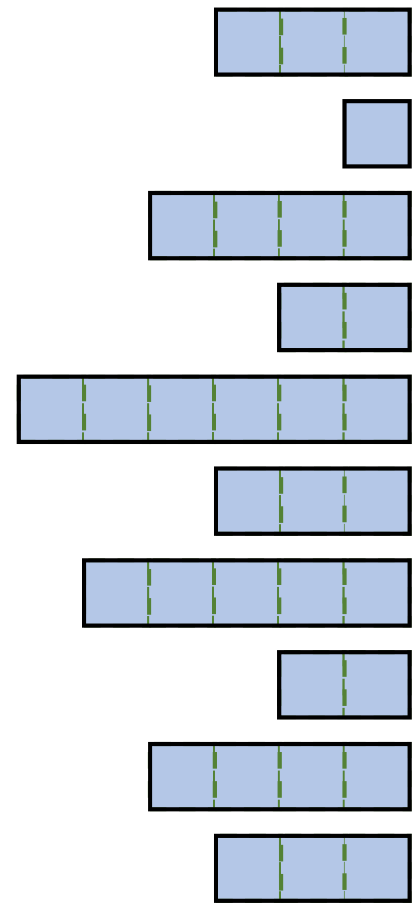
**Instance:** $n$ positive integers $x_1, x_2, \cdots, x_n \in \mathbb{Z}^+$.
**Problem:** Determine whether there exists a partition of $\{1,2,\cdots,n\}$ into two sets $A$ and $B$ such that $\sum_{i \in A} x_i = \sum_{i \in B} x_i$.

- The partition problem is one of Karp's 21 NP-complete problems.
- Thus the considered scheduling problem is NP-hard.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines

$n$ jobs each with processing time $p_j$



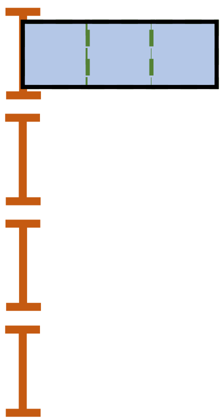
**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

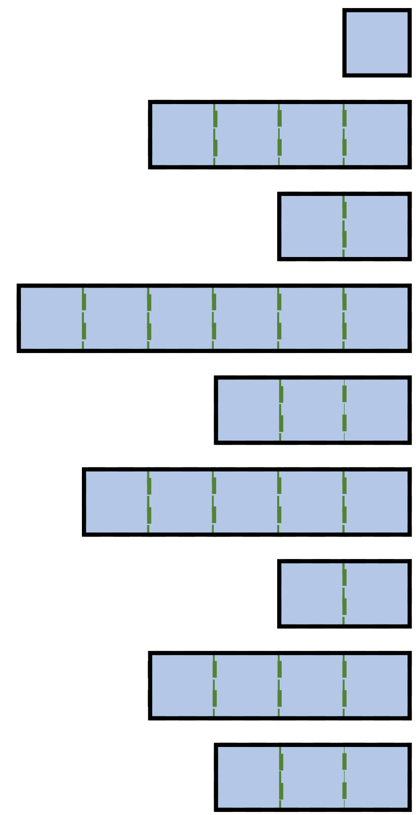
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines



$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

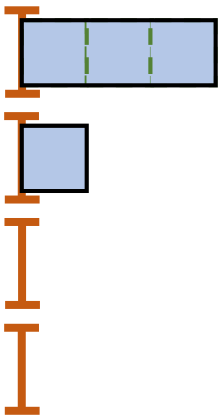
For each job $j = 1,2,\cdots,n$ do:

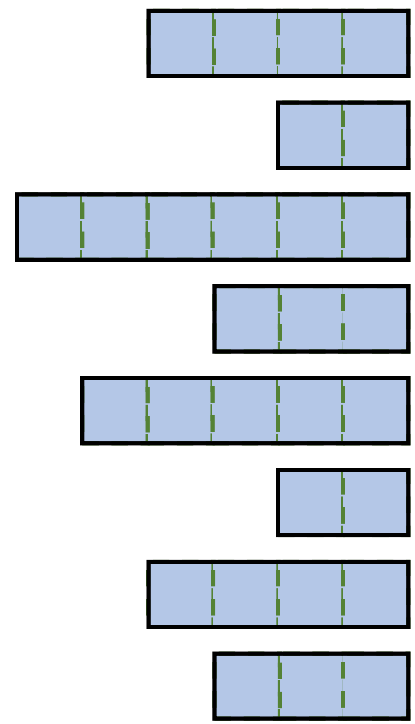
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines

$n$ jobs each with processing time $p_j$





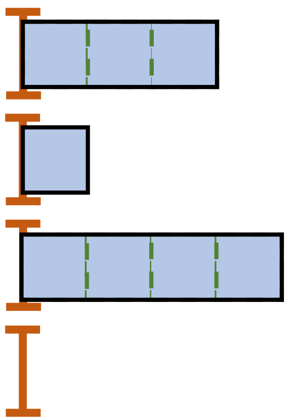
**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

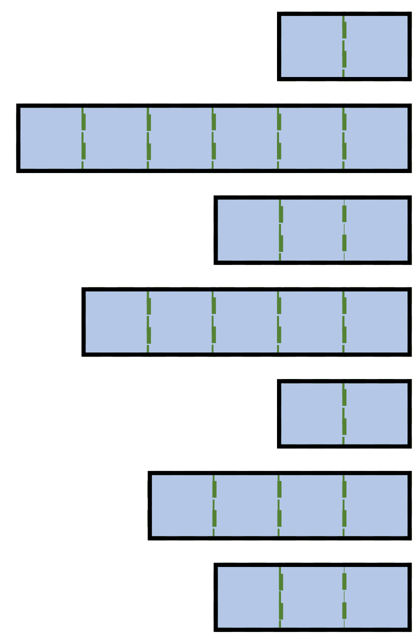
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines



$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

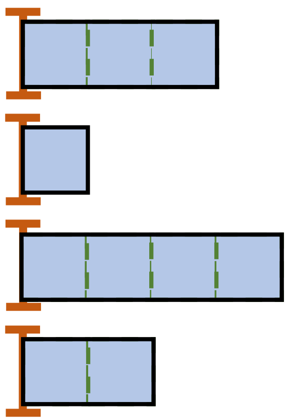
For each job $j = 1,2,\cdots,n$ do:

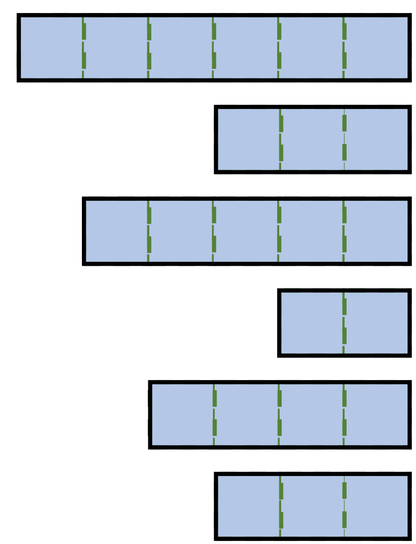
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines

$n$ jobs each with processing time $p_j$





**List** (Graham 1966):

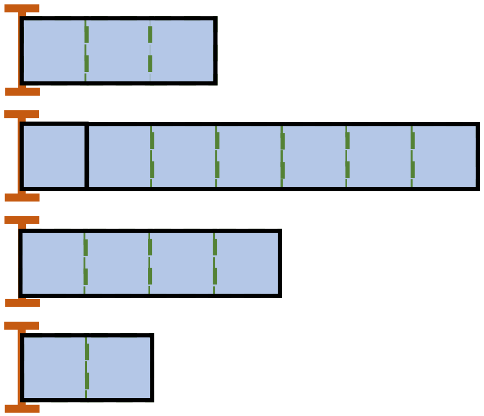
For each job $j = 1,2,\cdots,n$ do:

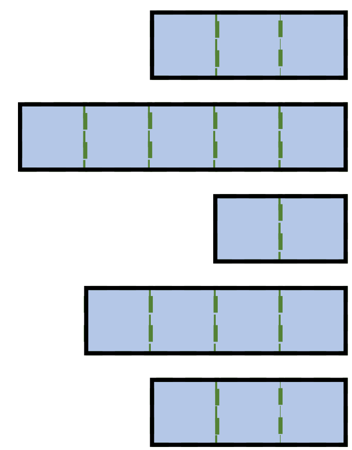
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines

$n$ jobs each with processing time $p_j$





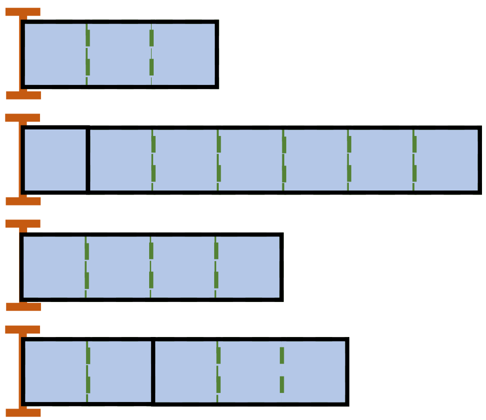
**List** (Graham 1966):

For each job $j = 1,2,\cdots,n$ do:

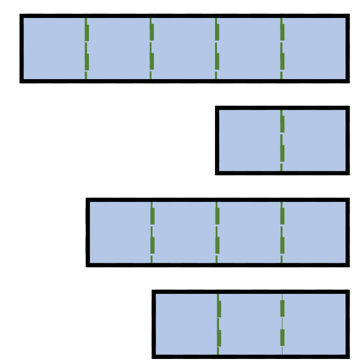
Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines



$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

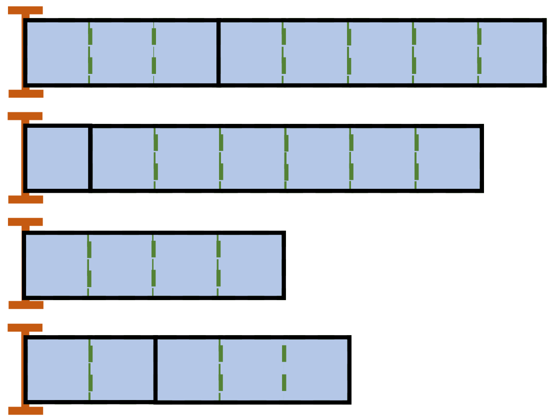
For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

# Graham's **List** Algorithm for Scheduling

$m$ identical machines

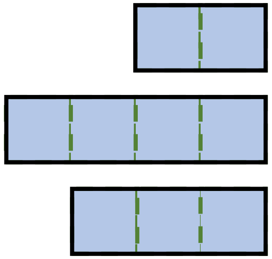
$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

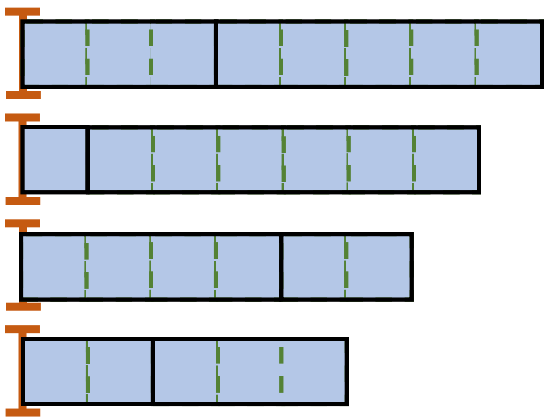
Assign job $j$ to a currently least loaded machine.



# Graham's **List** Algorithm for Scheduling

$m$ identical machines

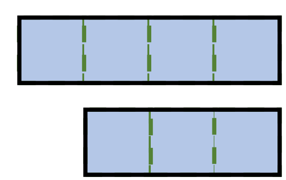
$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

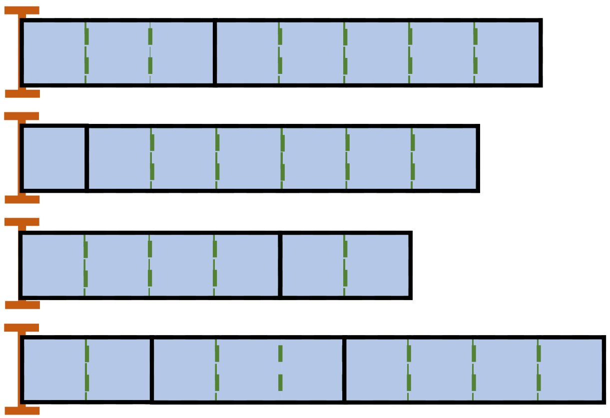
Assign job $j$ to a currently least loaded machine.



# Graham's **List** Algorithm for Scheduling

$m$ identical machines

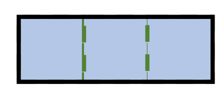
$n$ jobs each with processing time $p_j$



**List** (Graham 1966):

For each job $j = 1,2,\cdots,n$ do:

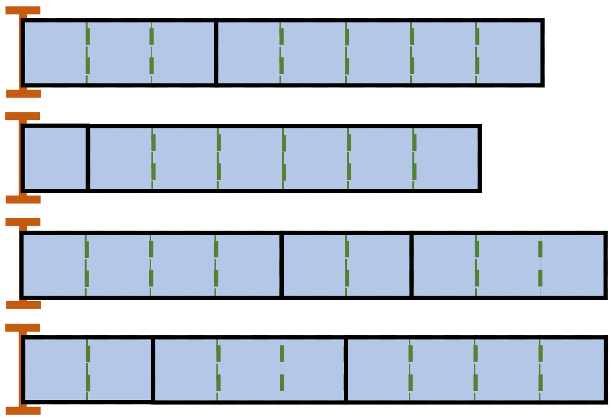
Assign job $j$ to a currently least loaded machine.



# Graham's **List** Algorithm for Scheduling

$m$ identical machines

$n$ jobs each with processing time $p_j$



**List** (Graham 1966)**:**

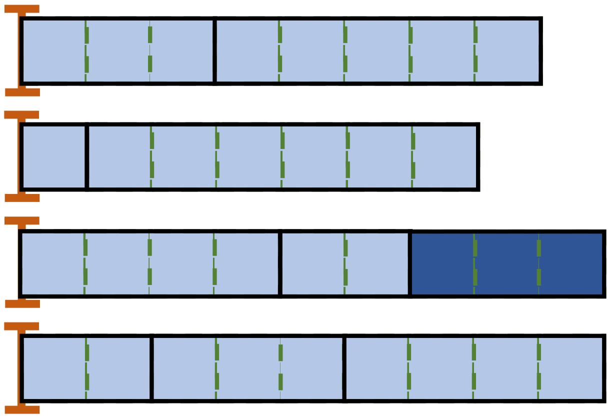
For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

**List** (Graham 1966):

For each job $j = 1,2,\cdots,n$ do:

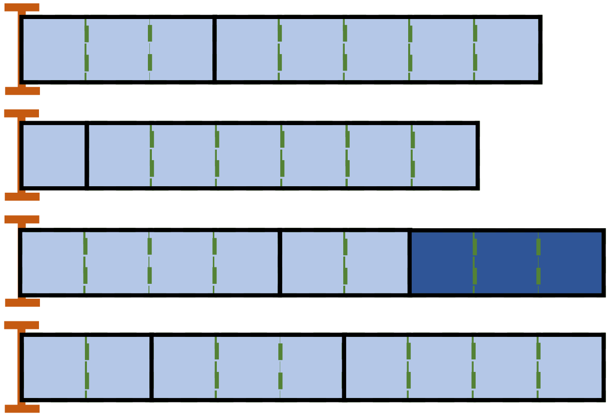
Assign job $j$ to a currently least loaded machine.



**List** (Graham 1966)**:**

For each job $j = 1, 2, \cdots, n$ do:

Assign job $j$ to a currently least loaded machine.

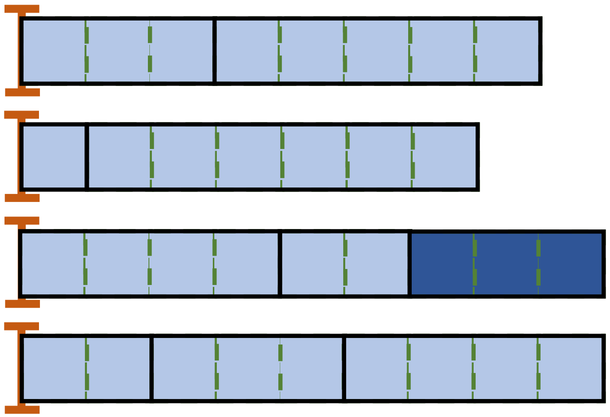


This algorithm finishes within poly-time.

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



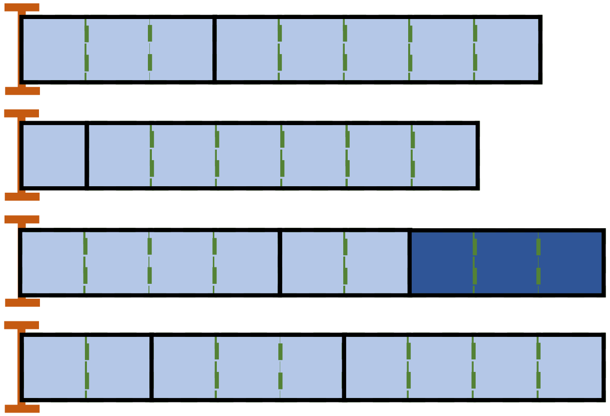
This algorithm finishes within poly-time.

What about the approximation ratio?

**List** (Graham 1966)**:**

For each job $j = 1, 2, \cdots, n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

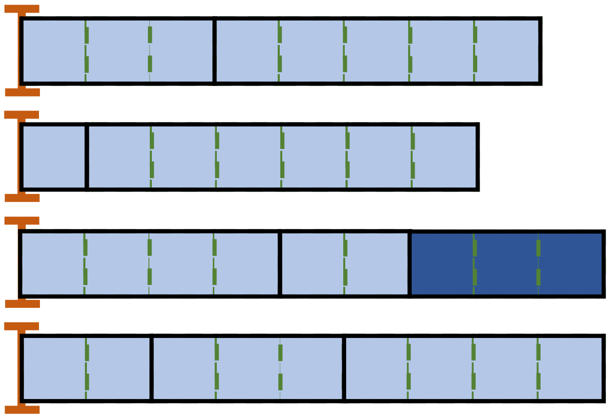
What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j$$

**<u>List</u>** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

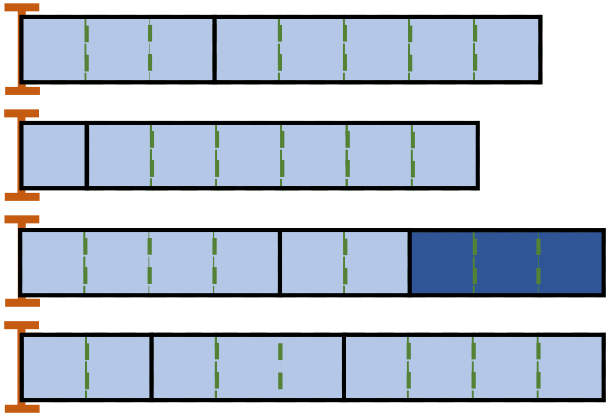
What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j \qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

What about the approximation ratio?

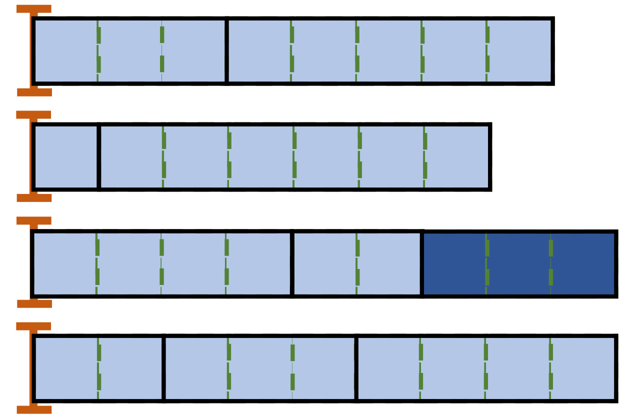
$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in the schedule, and last job on it is $l$.

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j \qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

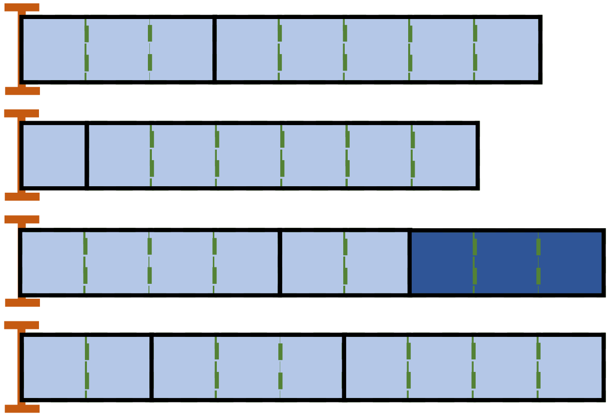
Assume machine $k$ finishes last in the schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in the schedule, and last job on it is $l$.

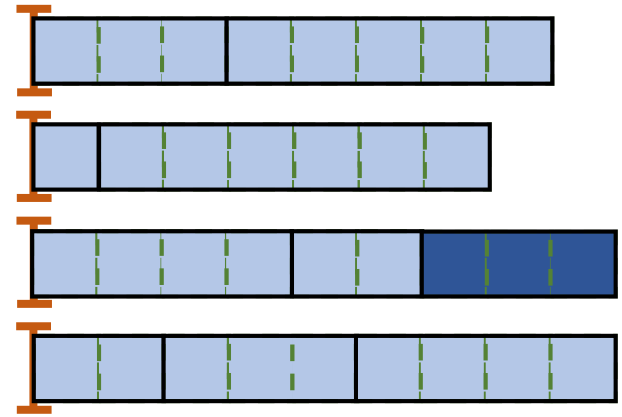
Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$p_l \leq \max_j p_j \leq \mathrm{OPT}$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in the schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$p_l \leq \max_j p_j \leq \mathrm{OPT}$

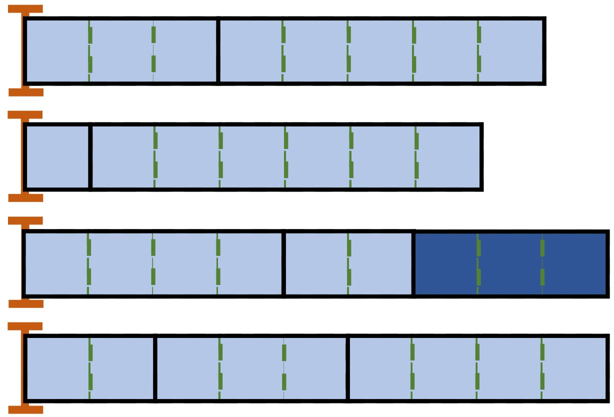
$C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$

since machine $k$ is least loaded when scheduling job $l$

**List** (Graham 1966):

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



This algorithm finishes within poly-time.

What about the approximation ratio?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in the schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l \leq 2 \cdot \mathrm{OPT}$

$p_l \leq \max_j p_j \leq \mathrm{OPT}$

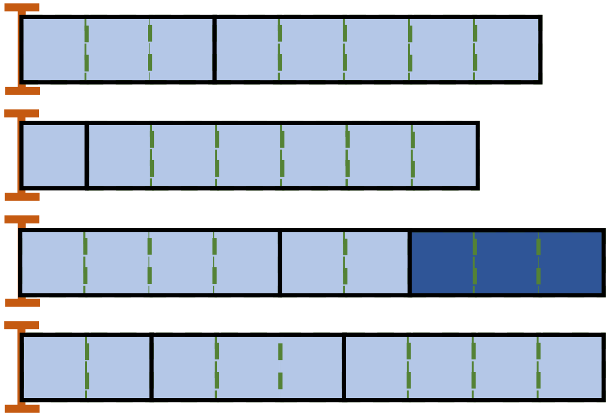
$C_k - p_l \leq \frac{1}{m}\sum_{j\neq l} p_j \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$

since machine $k$ is least loaded when scheduling job $l$

**List** (Graham 1966):

For each job $j = 1, 2, \cdots, n$ do:

Assign job $j$ to a currently least loaded machine.



Algorithm **List** finishes within poly-time.

Algorithm **List** has approximation ratio 2.

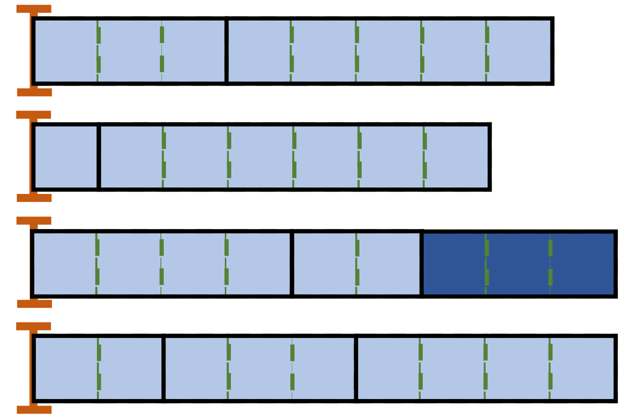
Makespan $C_{\max} = C_k = (C_k - p_l) + p_l \leq 2 \cdot \mathrm{OPT}$

$$p_l \leq \max_j p_j \leq \mathrm{OPT} \qquad C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

**List** (Graham 1966):

For each job $j = 1,2,\cdots,n$ do:

  Assign job $j$ to a currently least loaded machine.



Algorithm **List** finishes within poly-time.

Algorithm **List** has approximation ratio 2.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l \le 2 \cdot \mathrm{OPT}$

$$p_l \le \max_j p_j \le \mathrm{OPT}$$

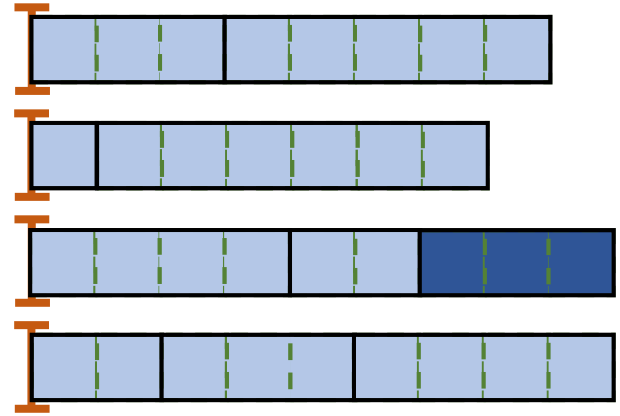
~~$C_k - p_l \le \frac{1}{m}\sum_{j\neq l} p_j \le \frac{1}{m}\sum_j p_j \le \mathrm{OPT}$~~

$$C_k - p_l \le \frac{1}{m}\sum_{j\neq l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.



Algorithm **List** finishes within poly-time.

~~Algorithm **List** has approximation ratio 2.~~

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$ ~~$\le 2 \cdot \mathrm{OPT}$~~ $\le \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

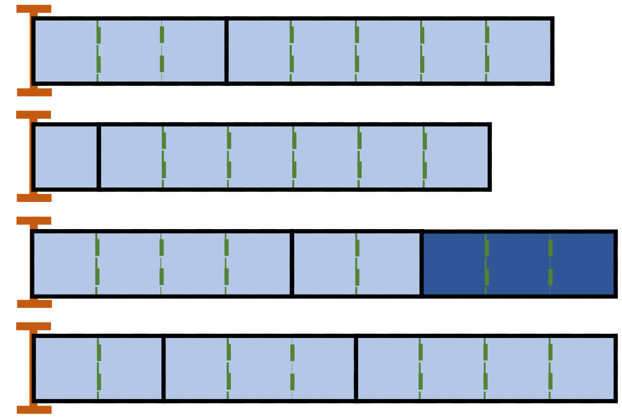
$p_l \le \max_j p_j \le \mathrm{OPT}$ ~~$C_k - p_l \le \frac{1}{m}\sum_{j \ne l} p_j \le \frac{1}{m}\sum_j p_j \le \mathrm{OPT}$~~

$$C_k - p_l \le \frac{1}{m}\sum_{j \ne l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

**<u>List</u>** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

  Assign job $j$ to a currently least loaded machine.



Algorithm **List** finishes within poly-time.

~~Algorithm **List** has approximation ratio 2.~~

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$ ~~$\le 2 \cdot \mathrm{OPT}$~~ $\le \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

$p_l \le \max_j p_j \le \mathrm{OPT}$

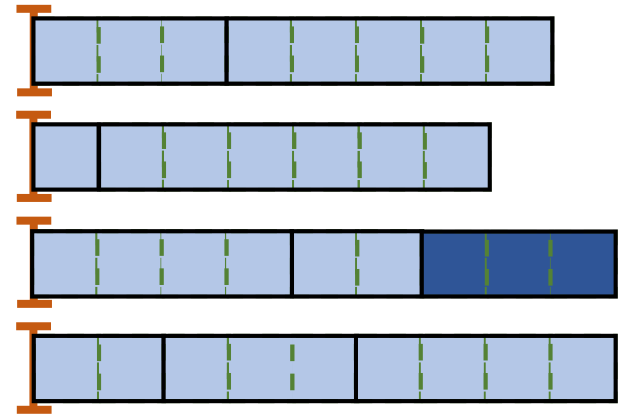
~~$C_k - p_l \le \frac{1}{m}\sum_{j \ne l} p_j \le \frac{1}{m}\sum_j p_j \le \mathrm{OPT}$~~

$$C_k - p_l \le \frac{1}{m}\sum_{j \ne l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

Algorithm **List** has approximation ratio $2 - 1/m$.

**<u>List</u>** <u>(Graham 1966):</u>

For each job $j = 1,2,\cdots,n$ do:
 Assign job $j$ to a currently least loaded machine.



Algorithm **List** finishes within poly-time.

~~Algorithm **List** has approximation ratio 2.~~

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$ ~~$\leq 2 \cdot \mathrm{OPT}$~~ $\leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

$p_l \leq \max_j p_j \leq \mathrm{OPT}$  ~~$C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$~~

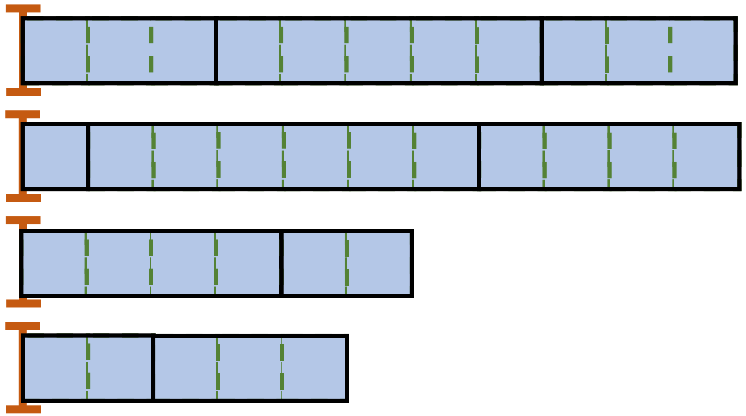
$$C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

Algorithm **List** has approximation ratio $2 - 1/m$.

This bound is tight in the worst case. [Almost tight example: $m^2$ unit jobs followed by a length $m$ job. **List** generates makespan of $2m$ while $\mathrm{OPT} = m + 1$.]

# **Local Search** for Scheduling

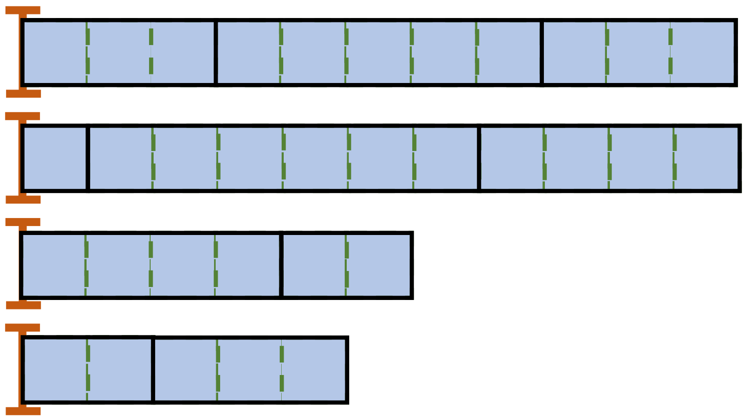
Start with an arbitrary solution:



Keep making improvements by *locally* adjusting the solution, until no further improvement can be made (**local optimum**)

# **Local Search** for Scheduling

Start with an arbitrary solution:



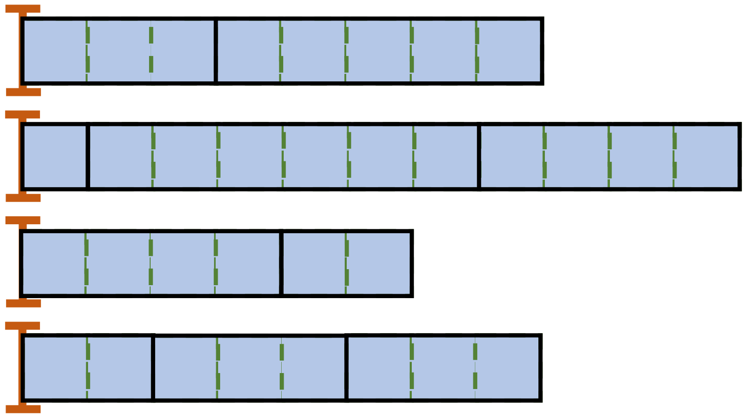
Keep making improvements by *locally* adjusting the solution, until no further improvement can be made (**local optimum**)

**LocalSearch:**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
    Let $l$ be a job that finished last.
    If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
        Transfer job $l$ to earliest such $i$.

# **Local Search** for Scheduling

Start with an arbitrary solution:



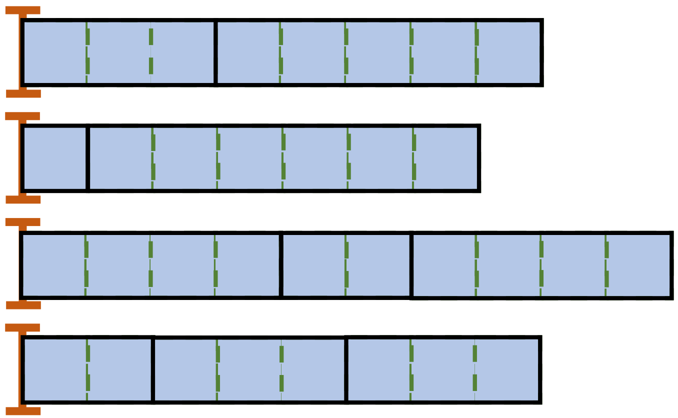
Keep making improvements by *locally* adjusting the solution, until no further improvement can be made (**local optimum**)

**<u>LocalSearch:</u>**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
Let $l$ be a job that finished last.
If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
Transfer job $l$ to earliest such $i$.

# **Local Search** for Scheduling

Start with an arbitrary solution:



Keep making improvements by *locally* adjusting the solution, until no further improvement can be made (**local optimum**)

**LocalSearch:**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
Let $l$ be a job that finished last.
If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
Transfer job $l$ to earliest such $i$.

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

    Let $l$ be a job that finished last.

    If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

        Transfer job $l$ to earliest such $i$.

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

Let $l$ be a job that finished last.

If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

**LocalSearch:**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
    Let $l$ be a job that finished last.
    If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
        Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

**LocalSearch:**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
    Let $l$ be a job that finished last.
    If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
        Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

  Let $l$ be a job that finished last.

  If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

    Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

**LocalSearch:**

Start with an arbitrary schedule.
Repeat until no job can be reassigned (i.e., local optimum reached):
  Let $l$ be a job that finished last.
  If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:
    Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

$$\mathrm{OPT} \geq \max_j p_j \qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

Let $l$ be a job that finished last.

If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

$$\mathrm{OPT} \geq \max_j p_j \qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$p_l \leq \max_j p_j \leq \mathrm{OPT}$$

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

  Let $l$ be a job that finished last.

  If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

    Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm?

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$p_l \leq \max_j p_j \leq \mathrm{OPT} \qquad C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

  Let $l$ be a job that finished last.

  If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

    Transfer job $l$ to earliest such $i$.

This algorithm finishes within poly-time. (No job is transferred twice!)

The approximation ratio of this algorithm? $(2 - 1/m)$

$$\mathrm{OPT} \geq \max_j p_j \qquad\qquad m \cdot \mathrm{OPT} \geq \sum_j p_j$$

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l \leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

$$p_l \leq \max_j p_j \leq \mathrm{OPT} \qquad\qquad C_k - p_l \leq \frac{1}{m}\sum_{j \neq l} p_j = \frac{1}{m}\sum_j p_j - \frac{p_l}{m}$$

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

Let $l$ be a job that finished last.

If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

Transfer job $l$ to earliest such $i$.

**List** (Graham 1966)**:**

For each job $j = 1,2, \cdots, n$ do:

Assign job $j$ to a currently least loaded machine.

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

Let $l$ be a job that finished last.

If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

Transfer job $l$ to earliest such $i$.

LocalSearch finds a schedule with makespan $C_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

Let $l$ be a job that finished last.

If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

Transfer job $l$ to earliest such $i$.

LocalSearch finds a schedule with makespan $C_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

The schedule returned by **List** must be a local optimum!

**LocalSearch:**

Start with an arbitrary schedule.

Repeat until no job can be reassigned (i.e., local optimum reached):

  Let $l$ be a job that finished last.

  If exists machine $i$ s.t. assigning job $l$ to $i$ allows $l$ finish earlier:

    Transfer job $l$ to earliest such $i$.

LocalSearch finds a schedule with makespan $C_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

  Assign job $j$ to a currently least loaded machine.

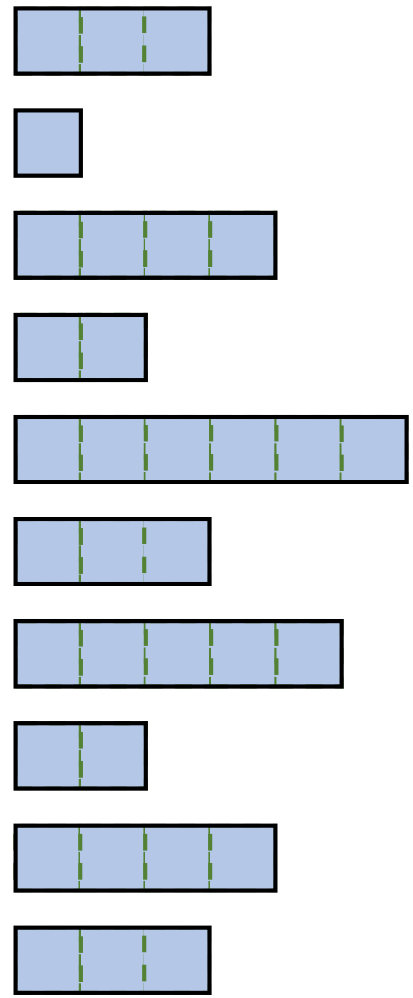
The schedule returned by **List** must be a local optimum!

**List** will find a schedule with makespan

$$C_{\max} \leq \left(2 - \frac{1}{m}\right) \cdot \mathrm{OPT}$$

$m$ identical machines

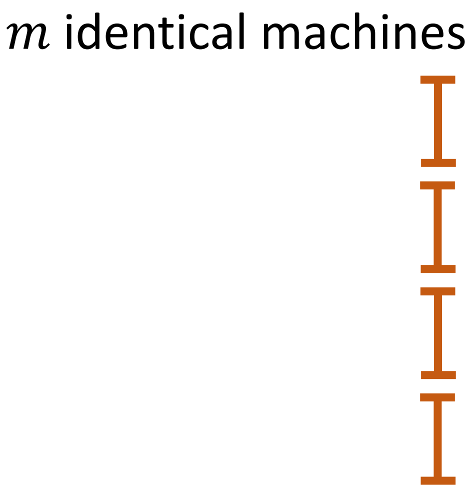
$n$ jobs

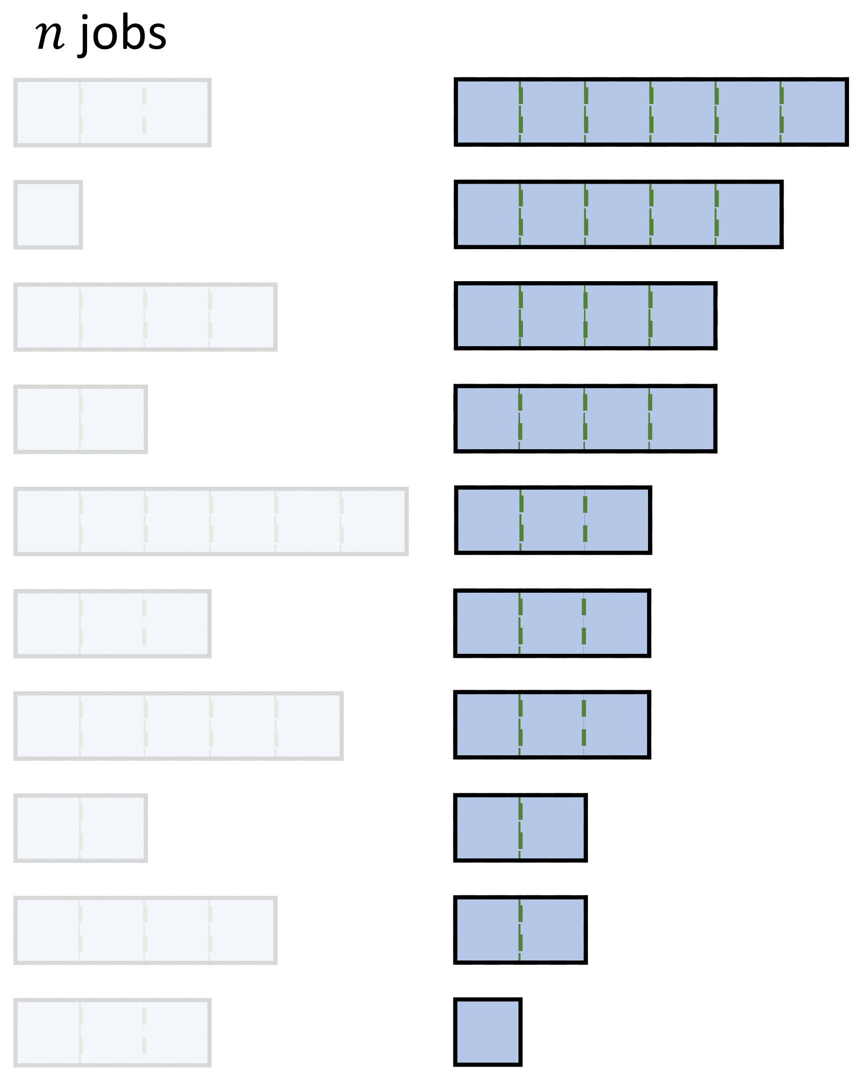


**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.




**List** (Graham 1966)**:**

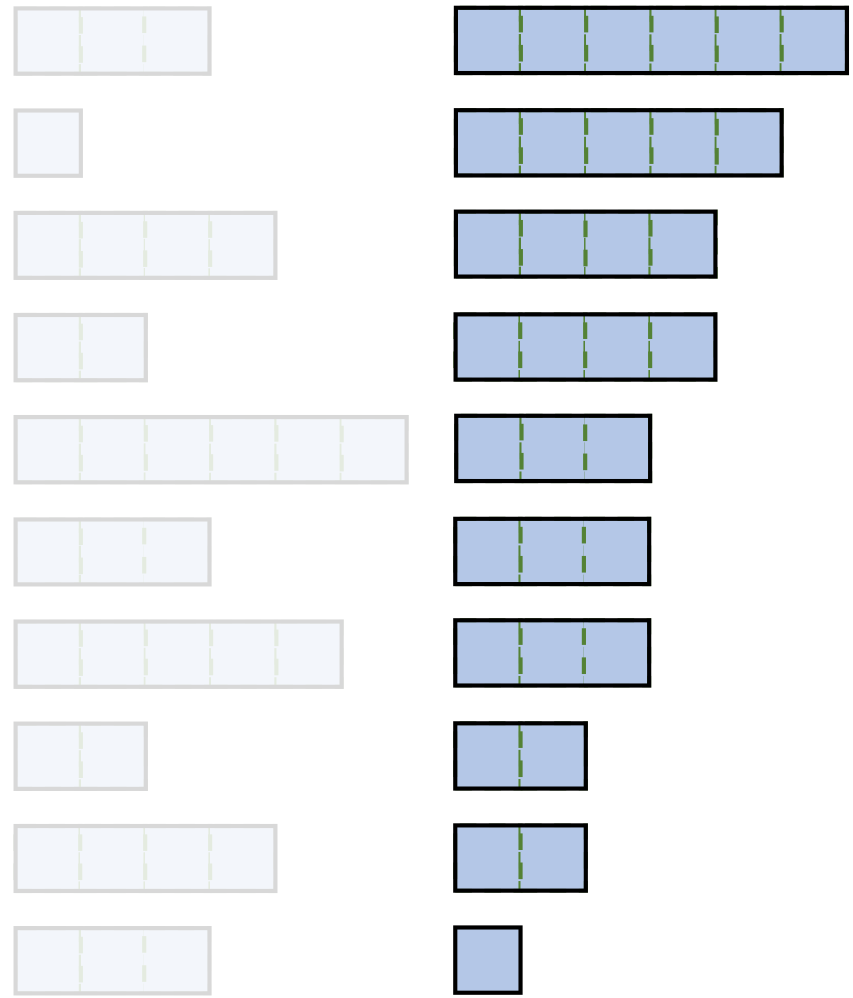
For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

# Longest Processing Time (LPT)

$m$ identical machines

$n$ jobs



**<u>List</u>** <u>(Graham 1966)</u>**<u>:</u>**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

**<u>LongestProcessingTime (LPT):</u>**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2,\cdots,n$ do:
    Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

For each job $j = 1,2,\cdots,n$ do:

  Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

W.l.o.g.:
- # of jobs > # of machines (i.e., $n > m$)
- makespan is achieved by some job bigger than $m$ (i.e., $l > m$)

Otherwise, LPT returns an optimal solution already!

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2, \cdots, n$ do:
    Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m} \sum_j p_j \leq \mathrm{OPT}$$

W.l.o.g.:
- # of jobs > # of machines (i.e., $n > m$)    $p_m + p_{m+1} \leq \mathrm{OPT}$
- makespan is achieved by some job bigger than $m$ (i.e., $l > m$)

Otherwise, LPT returns an optimal solution already!

**<u>LongestProcessingTime (LPT):</u>**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2,\cdots,n$ do:
Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

W.l.o.g.:
- # of jobs > # of machines (i.e., $n > m$) $p_m + p_{m+1} \leq \mathrm{OPT}$
- makespan is achieved by some job bigger than $m$ (i.e., $l > m$) $p_l \leq p_{m+1}$

Otherwise, LPT returns an optimal solution already!

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2,\cdots,n$ do:
    Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT} \qquad\qquad p_l \leq p_{m+1} \leq \frac{1}{2}(p_m + p_{m+1}) \leq \frac{\mathrm{OPT}}{2}$$

W.l.o.g.:
- # of jobs > # of machines (i.e., $n > m$) $\quad p_m + p_{m+1} \leq \mathrm{OPT}$
- makespan is achieved by some job bigger than $m$ (i.e., $l > m$) $\quad p_l \leq p_{m+1}$

Otherwise, LPT returns an optimal solution already!

**LongestProcessingTime (LPT):**

Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2,\cdots,n$ do:
 Assign job $j$ to a currently least loaded machine.

This algorithm finishes within poly-time.

The approximation ratio of this algorithm?

Assume machine $k$ finishes last in final schedule, and last job on it is $l$.

Makespan $C_{\max} = C_k = (C_k - p_l) + p_l \leq \frac{3}{2} \cdot \mathrm{OPT}$

$$C_k - p_l \leq \frac{1}{m}\sum_j p_j \leq \mathrm{OPT}$$

$$p_l \leq p_{m+1} \leq \frac{1}{2}(p_m + p_{m+1}) \leq \frac{\mathrm{OPT}}{2}$$

W.l.o.g.:
- # of jobs > # of machines (i.e., $n > m$) $p_m + p_{m+1} \leq \mathrm{OPT}$
- makespan is achieved by some job bigger than $m$ (i.e., $l > m$) $p_l \leq p_{m+1}$

Otherwise, LPT returns an optimal solution already!

**LongestProcessingTime (LPT):**

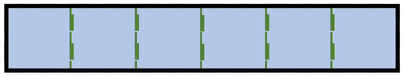
Sort jobs so that $p_1 \geq p_2 \geq \cdots \geq p_n$.
For each job $j = 1,2,\cdots,n$ do:
Assign job $j$ to a currently least loaded machine.

- We have shown **LPT** has approximation ratio (at most) $3/2$**.**
- By a more careful analysis, it can be shown **LPT** is actually a $4/3$ approximation algorithm.
- The problem of "minimum makespan on identical machines" has a **PTAS** (**P**olynomial **T**ime **A**pproximation **S**cheme).
  $\forall \epsilon > 0$, $\exists$poly-time $(1+\epsilon)$-approx. alg. for the problem

# Online Scheduling

$m$ identical machines

Jobs arrive (revealed) one-by-one

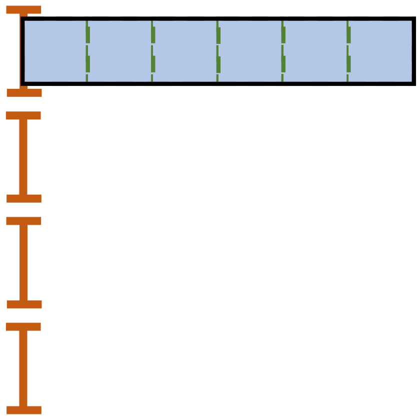


Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

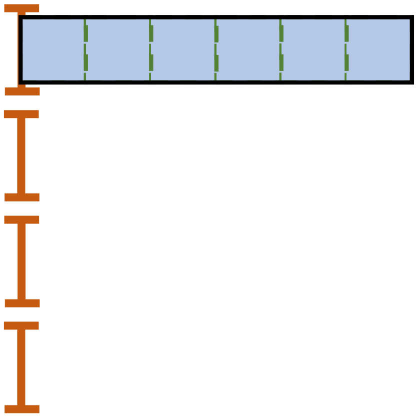
Jobs arrive (revealed) one-by-one



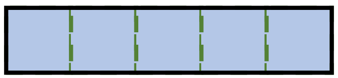
Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines



Jobs arrive (revealed) one-by-one

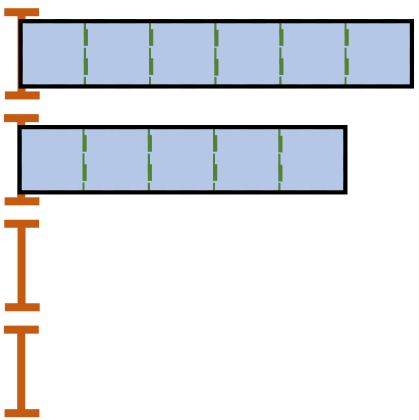


Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

Jobs arrive (revealed) one-by-one

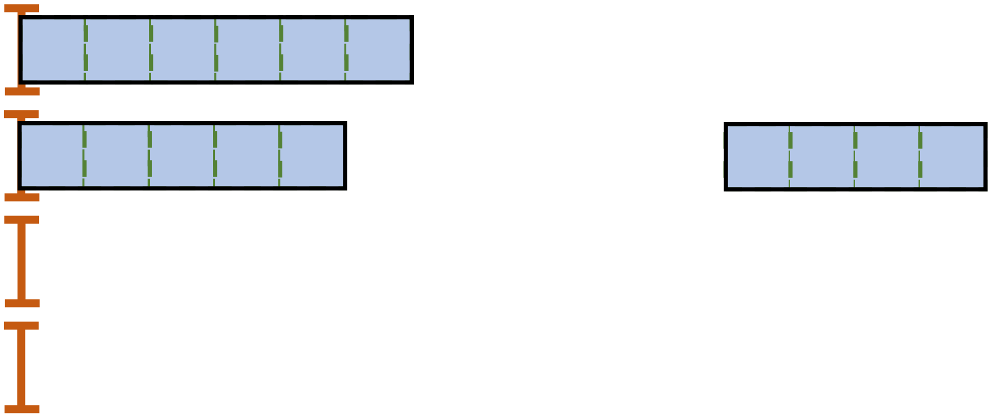


Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

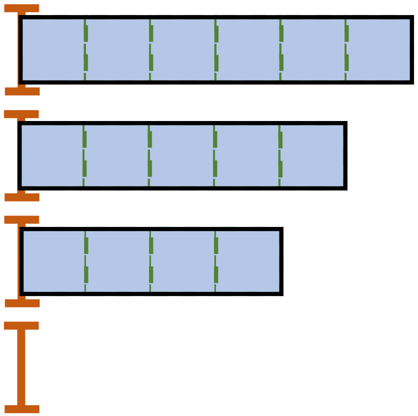
Jobs arrive (revealed) one-by-one



Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines



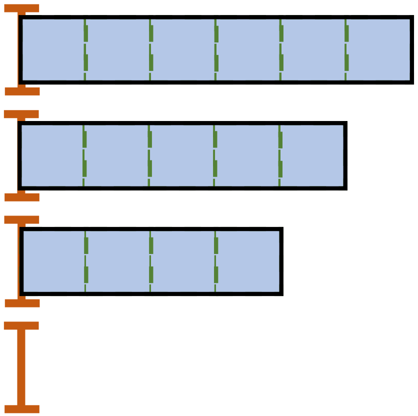
Jobs arrive (revealed) one-by-one

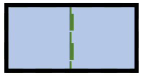
Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

Jobs arrive (revealed) one-by-one



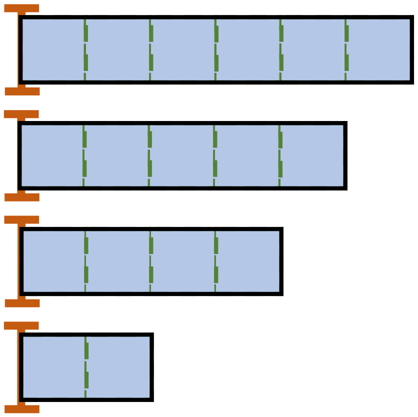


Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

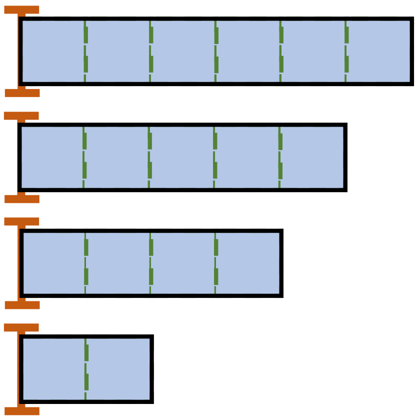
Jobs arrive (revealed) one-by-one



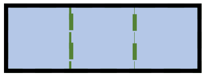
Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines



Jobs arrive (revealed) one-by-one

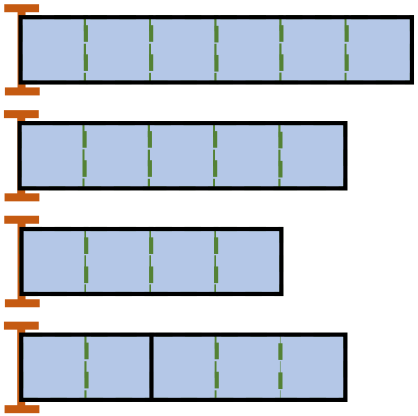


Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

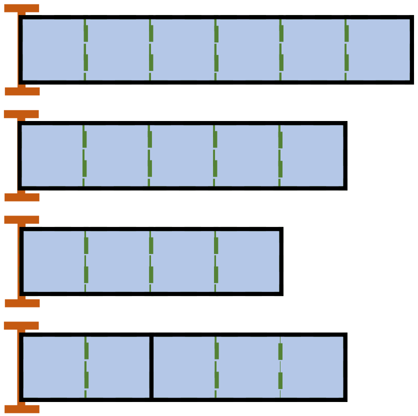
Jobs arrive (revealed) one-by-one



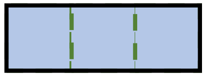
Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines



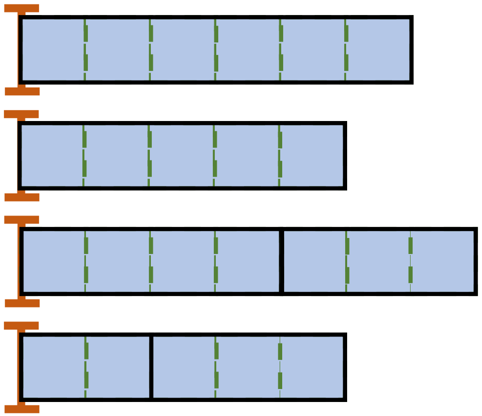
Jobs arrive (revealed) one-by-one



Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines



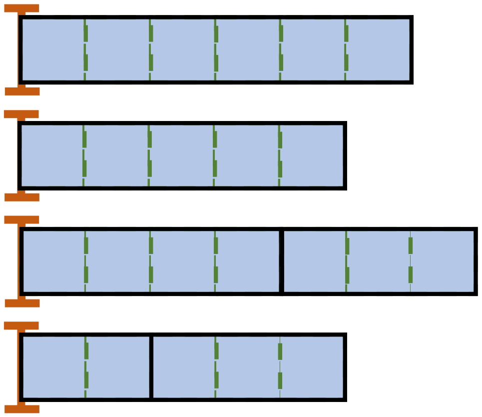
Jobs arrive (revealed) one-by-one

Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

# Online Scheduling

$m$ identical machines

Jobs arrive (revealed) one-by-one



Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

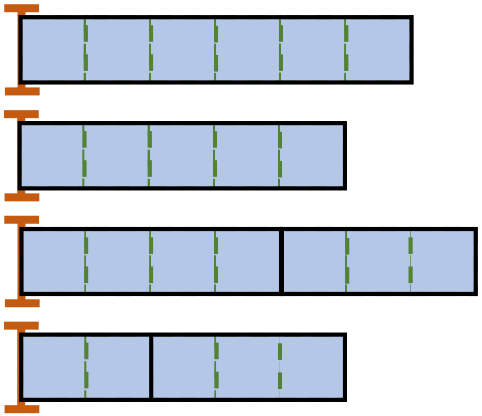
**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

# Online Scheduling

$m$ identical machines



Jobs arrive (revealed) one-by-one

Schedule decision must be made *once* a job arrives, *without* seeing jobs in the future.

**List** (Graham 1966)**:**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

**LPT** is not an online alg. for scheduling.

# Competitive Analysis

The competitive ratio of an **online algorithm** $\mathcal{A}$ is $\alpha$ if:

For every possible input sequence $I$ of the considered problem:

$$\frac{\text{solution value returned by online alg. } \mathcal{A} \text{ on } I}{\text{solution value returned by optimal offline alg. on } I} \leq \alpha$$

# Competitive Analysis

**List** (Graham 1966)**:**

For each job $j = 1,2, \cdots, n$ do:

Assign job $j$ to a currently least loaded machine.

The competitive ratio of an **online algorithm** $\mathcal{A}$ is $\alpha$ if:

For every possible input sequence $I$ of the considered problem:

$$\frac{\text{solution value returned by online alg. } \mathcal{A} \text{ on } I}{\text{solution value returned by optimal offline alg. on } I} \leq \alpha$$

# Competitive Analysis

**<u>List</u>** <u>(Graham 1966)</u>**<u>:</u>**

For each job $j = 1,2,\cdots,n$ do:

Assign job $j$ to a currently least loaded machine.

The competitive ratio of an **online algorithm** $\mathcal{A}$ is $\alpha$ if:

For every possible input sequence $I$ of the considered problem:

$$\frac{\text{solution value returned by online alg. } \mathcal{A} \text{ on } I}{\text{solution value returned by optimal offline alg. on } I} \leq \alpha$$

**List** is a 2-competitive online algorithm