# CS240 Algorithm Design and Analysis

## Lecture 25

## Approximation Algorithms

Quan Li
Fall 2024
2024.12.24

# Approximation Algorithms

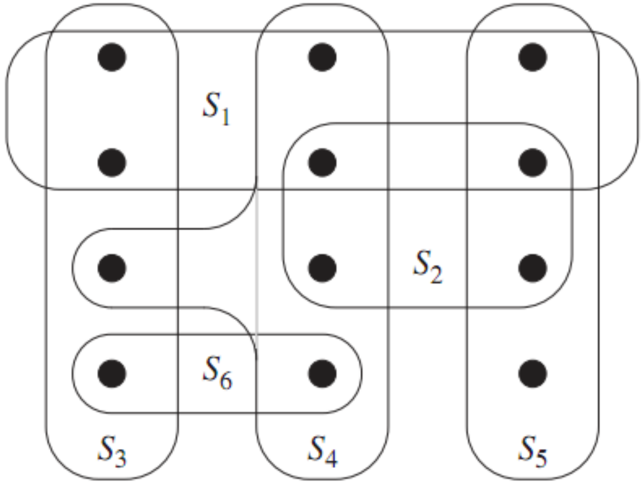
- Up to now, most of our algorithms have been exact, i.e., they find an optimal solution.
- But there are many problems for which we don't know how to find an optimal solution.
  - A key example is NP-complete problems. We don't know efficient algorithms for any NPC problem.
- Many such problems are important in practice. What do we do?
- If we can't get find the best answer, let's try for good enough.
- Approximation algorithms find an approximately optimal answer.

# Approximation Ratio

- Let X be a maximization problem. Let A be an algorithm for X.
- Let a>1 be a constant.
- A is an a-approximation algorithm for X if A always returns an answer that's at least 1/a times the optimal.
  - Ex If X is max-flow, A is a 2-approx algorithm if it always returns a flow that's at least ½ the optimal.
  - The closer a is to 1, the better the approximation.
- If X is a minimization problem, A is an a-approximation algorithm for X if it always returns an answer that's at most a times larger than the optimal.
  - Ex If X is min-cut, A is a 2-approx algorithm if it always returns a cut that's at most 2 times the size of the optimal.
  - Again, the closer a is to 1, the better the approximation.

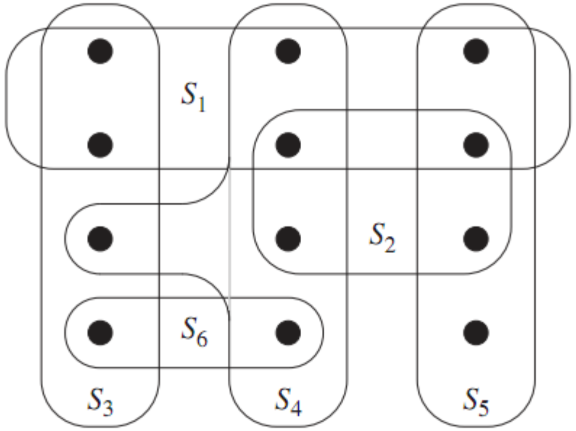
# Coverings

- Suppose there's a set of teachers, and each can teach a certain set of classes.
  - Let $S_i$ be the set of classes teacher i can teach.
- The entire set of classes is X.
- We want to pick the minimum set of teachers to teach all the classes.
  - Let T be set of teachers we pick.
  - We want $\cup_{i \in T} S_i = X$, and T to be the smallest possible.



# Set Covering

- **Input** A collection F of sets. Each set has a cost. The union of all the sets is X.
- **Output** A subset G of F, whose union is X.
- **Goal** Minimize the total cost of the sets in G.




*If all sets have same cost, $S_3$, $S_4$ and $S_5$ is a min cost set cover of X.*

- Minimum cost set cover is NP-complete.
- We'll see a ln(n)-approximation algorithm, where n=|X|.

# A Greedy Approximation Algorithm

- A natural greedy heuristic is to choose sets which cover points most cheaply.
  - For each set, let c be its cost, and m be the number of points it covers.
  - We want to use the set with the smallest c/m value, because this is the cheapest way to cover some new points.
- After we pick this set, remove all the points it covers. Then we consider the per unit cost of the remaining sets and again choose the cheapest.

# A Greedy Approximation Algorithm



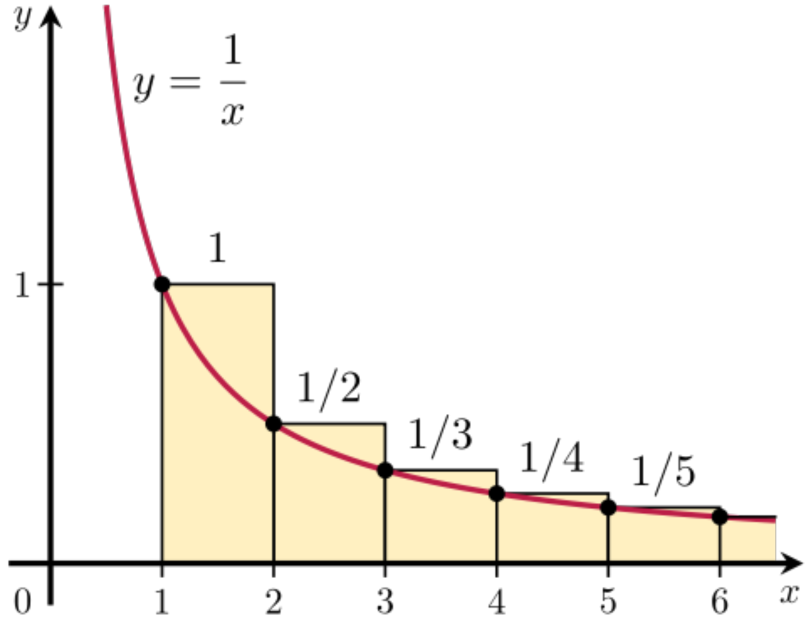
- F is the entire collection of sets. The union of these sets is X.
- Each set S in F has a cost cost(S).
- U is the set of elements of X we haven't covered yet.
- C is the set cover we eventually output.

- U = X
- C = ∅
- while $U \neq \emptyset$
  - choose $S \in F - C$ with min |cost(S)|/|S∩U|
  - C = C∪{S}
  - U = U − S
- output C

Per unit cost to cover new elements.

# Proof of Correctness

- We always output a set cover, because the while loop continues till X is covered.
- We'll prove the approximation ratio is at most 1+1/2+1/3+…+1/n ≈ ln(n).
  - If the min cost of a set cover is V, our set cover costs at most ln(n)*V.
- The basic plan is to bound the cost of the set cover the algorithm outputs using the "average cost" per element.



# Proof of Correctness

- Order the sets in C by when they're added to C, earliest set first.
  - Let the order be $S_1, S_2, \ldots, S_m$.
- Cost of the set cover is $L = \sum_i cost(S_i)$.
- Order the elements in X by when they're added, earliest element first.
  - Let the order be $e_1, e_2, \ldots, e_n$.
  - So, the first few e's are added by $S_1$, the next few added by $S_2$, etc.
  - Every element in X is in the list, because C covers X.

# Proof of Correctness

- Let $n_i$ be the number of new elements $S_i$ covers.
  - So, $n_i$ is the number of elements in $S_i$, but not in $S_1,\ldots,S_{i-1}$.
- Divide the cost of $S_i$ evenly among the new elements it covers.
  - If e is newly covered by $S_i$, then cost(e)= cost($S_i$)/$n_i$.

$$\sum_{k} cost(e_k) = \sum_{i} n_i * \frac{cost(S_i)}{n_i} = \sum_{i} cost(S_i) = L$$

  - Every element is covered by some $S_i$, and $S_i$ covers $n_i$ new elements.
- **We'll prove cost($e_k$) $\leq$ OPT/(n-k+1), for any k.**
- Suppose this is true, then

$$L = \sum_k cost(e_k) \leq \sum_k OPT/(n-k+1) \approx \ln(n) * OPT$$

# The Per Element Cost



- Let's focus on some element $e_k$, and let $S_j$ be the set which covers $e_k$ for the first time.
- Let $C_1,\ldots,C_m$ be the sets in an optimal cover, each of which covers some elements of $U=\{e_k,e_{k+1},e_{k+2},\ldots,e_n\}$.
  - Let $n'_1,\ldots,n'_m$ be the number of elements of U which $C_1,\ldots,C_m$ cover.
- **Obs 1: $\sum_i n'_i \geq$ n-k+1.**
  - Because $C_1,\ldots,C_m$ cover U.
- **Obs 2: $\sum_i cost(C_i) \leq$ OPT.**
  - Because $C_1,\ldots,C_m$ are a subset of an optimal cover, which has cost OPT.

# The Per Element Cost



- **Obs 3** None of $C_1,\ldots,C_m$ are among $S_1,\ldots,S_{j-1}$.
  - If some $C_i$ is among $S_1,\ldots,S_{j-1}$, then since $C_i$ covers some e in U, e would be covered by $\{S_1,\ldots,S_{j-1}\}$. So, e would be among the first k-1 elements covered. Contradiction.
- **Obs 4** There exists some $C_i$ among $C_1,\ldots,C_m$ with $\frac{cost(C_i)}{n'_i} \le OPT/(n-k+1)$.
  - If every $C_i$ in $C_1,\ldots,C_m$ has $\frac{cost(C_i)}{n'_i} \ge OPT/(n-k+1)$, then

$$OPT \ge \sum_i cost(C_i) = \sum_i n'_i * \frac{cost(C_i)}{n'_i} > \sum_i n'_i * \frac{OPT}{n-k+1} \ge OPT/(n-k+1)\sum_i n'_i \ge \frac{OPT}{n-k+1} * (n-k+1) = OPT$$

Contradiction.

# Proof of Approximation Ratio



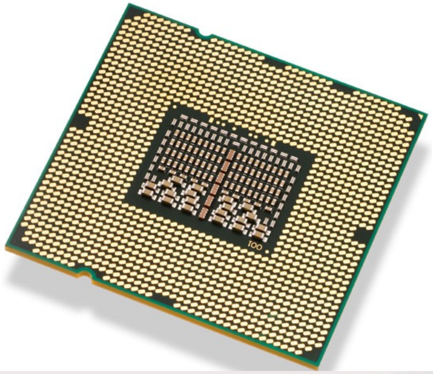
- **Lemma** $\boldsymbol{cost(S_j)/n_j \leq OPT/(n-k+1)}$
- **Proof** When choosing $S_j$, the only sets the algorithm is not allowed to choose are $S_1,\ldots,S_{j-1}$.
  - By obs 3, $C_1,\ldots,C_m$ aren't in $S_1,\ldots,S_{j-1}$.
  - By obs 4, there's some $C_i$ in $C_1,\ldots,C_m$, with $\frac{cost(C_j)}{n'_i} \leq OPT/(n-k+1)$.
  - $S_j$ was chosen so that $cost(S_j)/n_j$ is min among all sets not in $S_1,\ldots,S_{j-1}$.
  - So $\frac{cost(S_j)}{n_j} \leq \frac{cost(C_i)}{n'_i} \leq OPT/(n-k+1)$.
- Since $\frac{cost(S_j)}{n_j} = cost(e_k)$, we have $cost(e_k) \leq OPT/(n-k+1)$.
- The approximation ratio follows because

$$\boldsymbol{L = \sum_k cost(e_k) = \sum_k \frac{OPT}{n-k+1} \approx In(n) * OPT}$$

# Scheduling

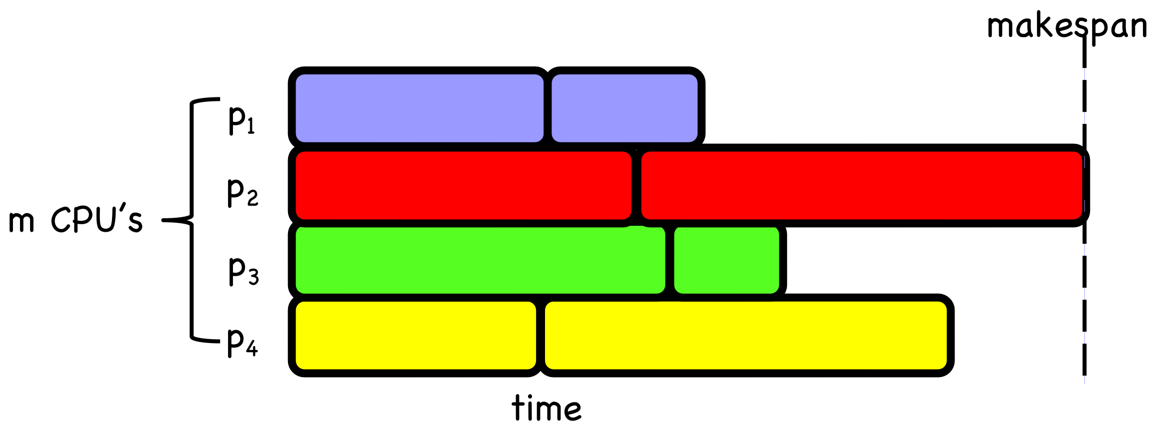
# Parallel Computing and Scheduling

- Computers today are parallel.
  - Multiple processors in a system.
  - Multiple tasks for the processors to run.
- Multiprocessor scheduling is the problem of deciding which tasks to run on which processors at what time.
- Many possible objectives.
  - Throughput, fairness, energy usage.
  - Latency, i.e. finishing all jobs as fast as possible.



# Makespan Scheduling

- n independent jobs.
  - Jobs have different sizes, i.e. time needed to perform job.
  - Jobs can be done in any order.
  - Any job can be done on any machine.
- m processors.
  - All have the same speed.
  - Each processors can do one job at a time.
- Assign the jobs to the processors.
- Makespan is when the last processor finishes all its jobs.
- Minimize the makespan.
  - i.e., finish all the jobs as fast as possible.



# Minimizing makespan is NPC



- The decision version of scheduling is obviously in NP.
- SUBSET-SUM: given a set of numbers S and target t, is there a subset of S summing to t?
  - Ex S={1,3,8,9}. t=9, yes. t=14, no.
  - This is NP-complete. We reduce SUBSET-SUM to scheduling.
- Let (S,t) be an instance of SUBSET-SUM.
  - Let s be sum of all elements in S.
- Make a set of jobs J = S∪{s-2t}, and schedule them on 2 processors.

> - **Recipe to establish NP-completeness of problem Y**
>   - Step 1. Show that Y is in NP
>   - Step 2. Choose an NP-complete problem X
>   - Step 3. Prove that $X \leq_p Y$

# Minimizing makespan is NPC

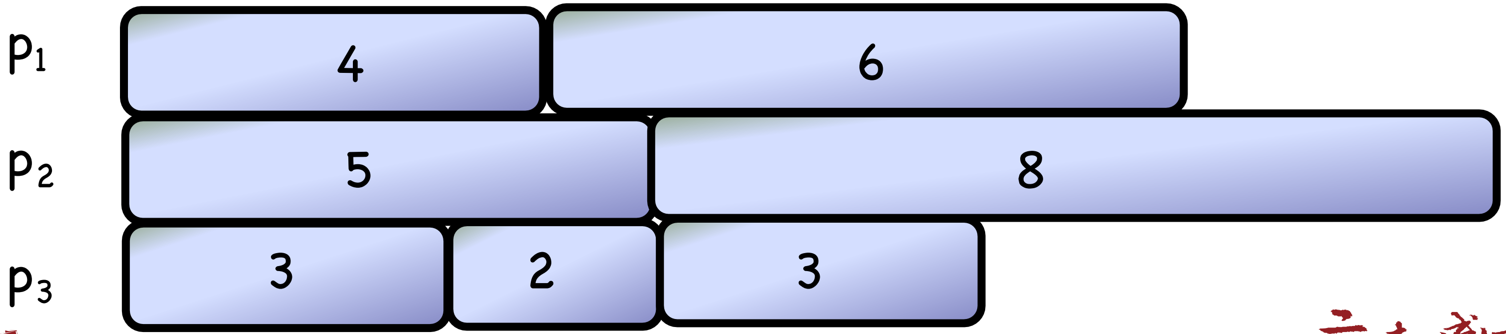
- **Claim** If some subset of S sums to t, then min makespan is s-t.
- **Proof** Say S′⊆S sums to t. Schedule the jobs in S′ and job s-2t on processor 1. So proc 1 finishes at time t+s-2t=s-t. Proc 2 does the jobs in S-S′, so it finishes at time s-t as well.
- **Claim** If the min makespan is s-t, there exists a subset of S that sums to t.
- **Proof** Suppose WLOG proc 1 does the s-2t job. Since makespan is s-t, the other jobs proc 1 does must have total size s-t-(s-2t)=t.
- So (S,t) is yes instance of SUBSET-SUM iff makespan = s-t.
  - So SUBSET-SUM $\leq p$ scheduling, and scheduling is NP-complete.

# Graham's List Scheduling

- Since scheduling is NPC, it's unlikely we can find the min makespan in polytime.
- List scheduling is a simple greedy algorithm.
  - Finds a schedule with makespan at most twice the minimum.
  - A 2-approximation.
- If there are n tasks and m processors, list scheduling only takes $O(n \log n + m)$ time.
  - Compare this to $n!\, C(n+m-1, m-1)$ time to try all possible schedules and pick the best.

# Graham's List Scheduling

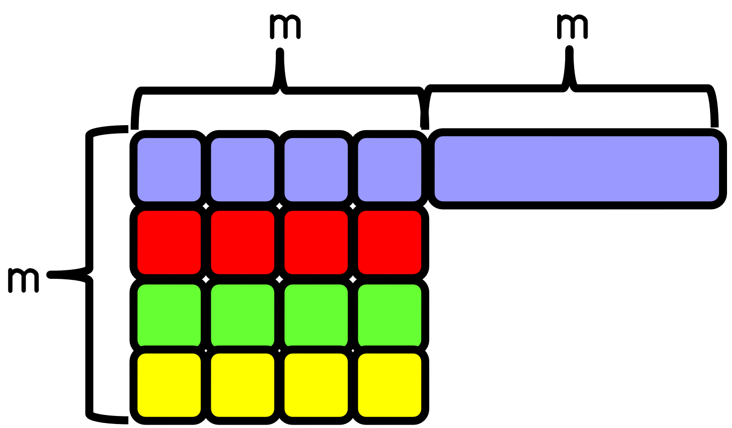
- List the jobs in any order.
- As long as there are unfinished jobs.
  - If any processor doesn't have a job now, give it the next job in the list.
- Example
- 3 processors. The jobs have length 2, 3, 3, 4, 5, 6, 8.
- List them in any order. Say 4, 5, 3, 2, 6, 8, 3.
- Initially, no proc has a job. Give first 3 jobs to the 3 procs.
- At time 3, proc 3 is done. Give it next job in list, 2.
- At time 4, proc 2 is done. Give it next job in list, 6.
- At time 5, both 1, 3 are done. Give them next jobs in list, 8,3.
- Everybody finishes by time 13.
  - The makespan of this schedule is 13.



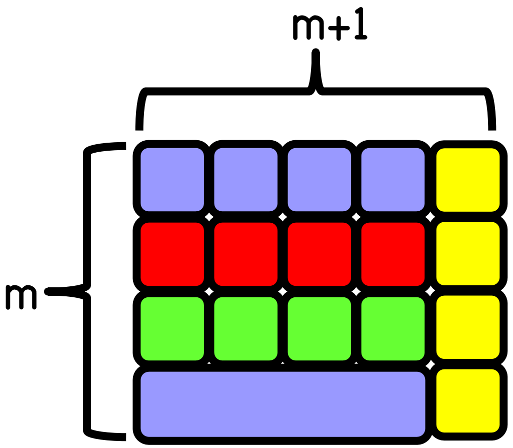
# The Worst Case for LS



- How badly can list scheduling do compared to optimal?
- Say there are $m^2$ jobs with length 1, and one job with length m.
  - Suppose they're listed in the order 1,1,1,…,1,m.
  - LS has makespan 2m. Optimal makespan is m+1.
  - makespan(LS) / makespan(opt) = $2m/(m+1) \approx 2$.
- This is worst possible case for list scheduling.




makespan(LS) = 2m

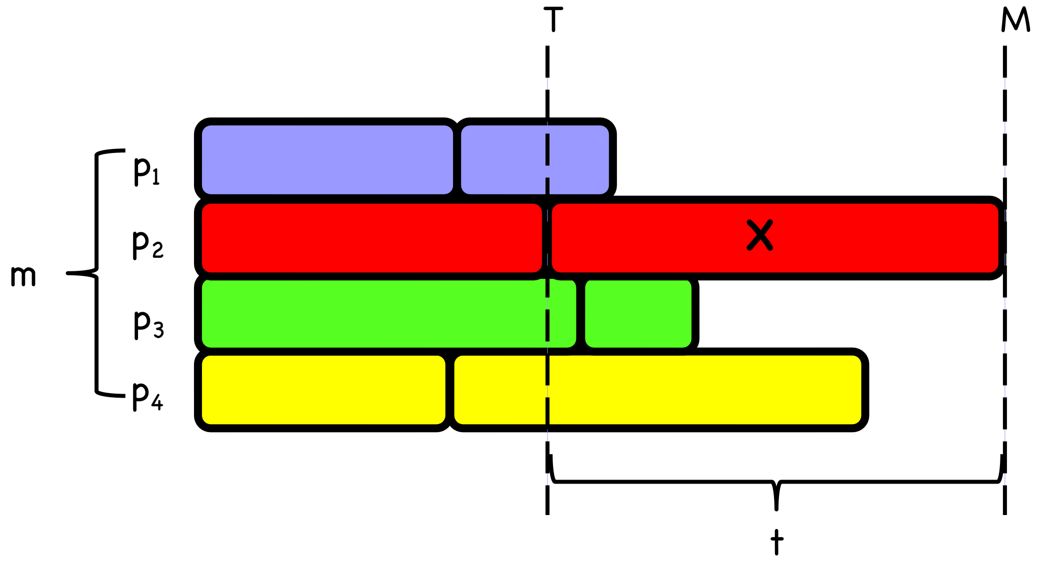



makespan(opt) = m+1

# LS is a 2-approximation

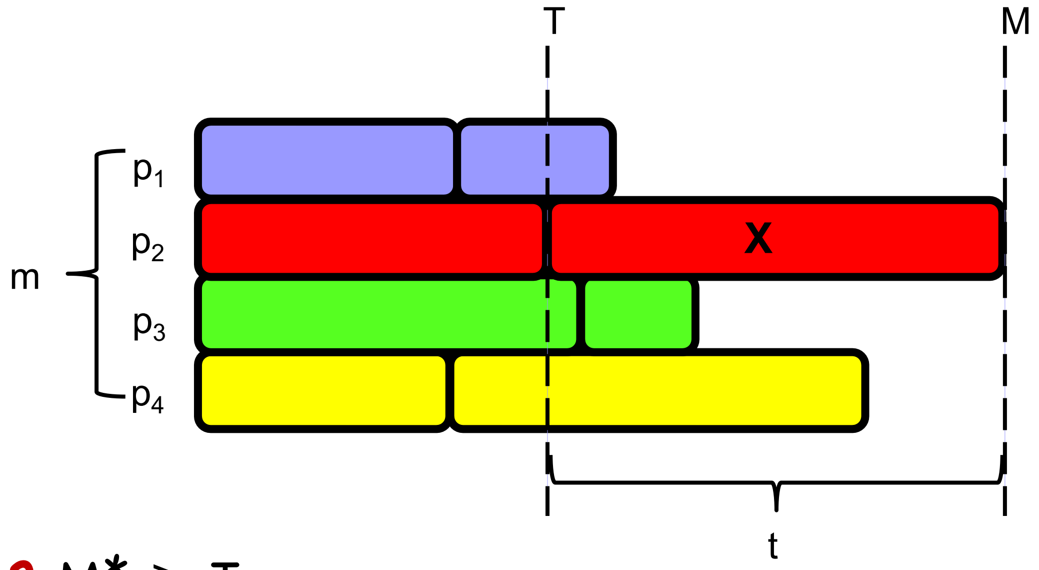
- Next, we prove LS always gives a schedule at most twice the optimal.
- Suppose LS gives makespan of M.
- Let the optimal schedule have makespan M*.
- We prove that $M \leq 2M^*$.

# LS is a 2-approximation



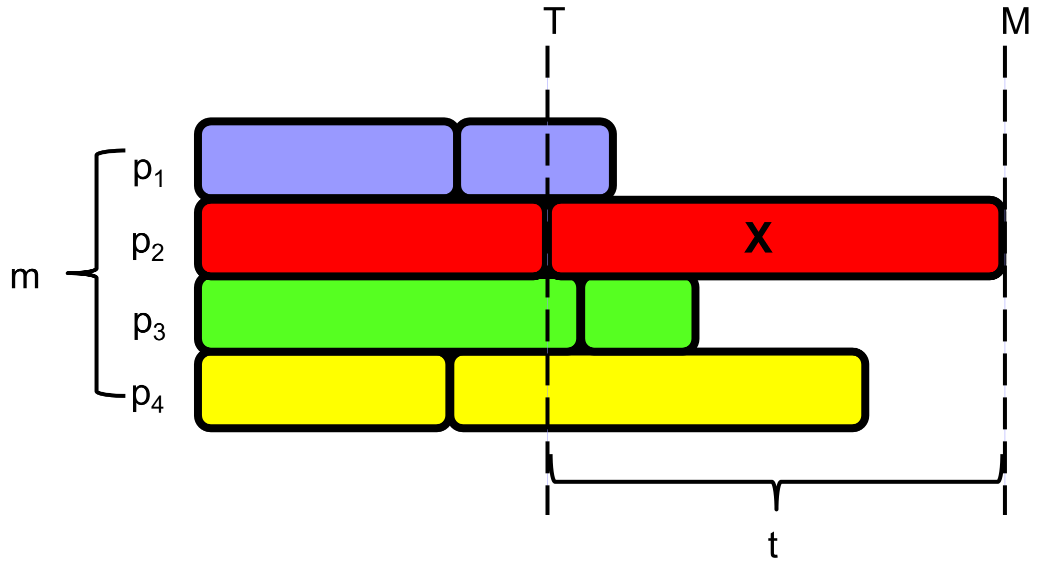

- The picture above is the schedule produced by list scheduling.
- Consider task X that finishes last.
  - Say X starts at time T, and has length t.
- **Claim 1** $M^* \geq t$.
  - In any schedule, X has to run on some process.
  - Since X takes t time, every schedule, including the opt, takes $\geq t$ time.

# LS is a 2-approximation




- **Claim 2** $M^* \geq T$.
  - Up to time T, no processor is ever idle.
    - Up to T, there's always some unfinished job.
    - As soon as a processor finishes one job, it's assigned another one.
  - So at time T, each processor completed T units of work.
  - So total amount of work in all the jobs is $\geq mT$. Up to T: mT
  - In the opt schedule, m processors complete at most m units of work per time unit.
  - So length of opt schedule is $\geq$ (total work)/m $\geq mT/m = T$.

# LS is a 2-approximation




- From Claims 1 and 2, we have $M^* \geq t$ and $M^* \geq T$.
- So $M^* \geq \max(T,t)$.
- $M = T + t$, because X is last job to finish.
- So $M/M^* \leq (T+t)/\max(T,t) \leq 2$.

# LPT Scheduling

- Worst case for LS occurred when longest job was scheduled last.
  - Large jobs are "dangerous" at end.
- Let's try to schedule longest jobs first.
- Longest processing time (LPT) schedule is just like list scheduling, except it first sorts tasks by nonincreasing order of size.
- **Ex** For three processors and tasks with sizes 2, 3, 3, 4, 5, 6, 8, LPT first sorts the jobs as 8,6,5,4,3,3,2. Then it assigns $p_1$ tasks 8,3, $p_2$ tasks 6,3, $p_3$ tasks 5,4,2, for a makespan of 11.
- LPT has an approximation ratio of 4/3.

# LPT is a 4/3-approximation

上海科技大学 ShanghaiTech University

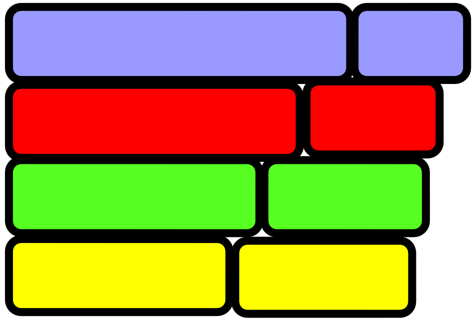
- **Thm** Suppose the optimal makespan is $M^*$, and LPT produces a schedule with makespan M. Then $M \leq 4/3\ M^*$.
- Let X be the last job to finish. Assume it starts at time T and has size t.
- Assume WLOG that X is the last job to start.
  - If not, then say Y starts after T.
  - Y finishes before T+t. So we can remove Y without increasing the makespan.
- **Cor 1** X is the smallest job.
  - X is the last job to start, so due to LPT scheduling it's the smallest.

立志成才报国裕民

# LPT is a 4/3-approximation

- **Claim 1** LPT's makespan = $T+t \leq M^*+t$.
  - As in LS, no processor is idle up to time T, so $M^* \geq T$.
- **Case 1** $t \leq M^*/3$.
  - Then LPT's makespan $\leq M^* + t \leq M^* + M^*/3 = 4/3\ M^*$.

- **Case 2** $t > M^*/3$.
  - Since X is the smallest task, all tasks have size $> M^*/3$.
  - So the optimal schedule has at most 2 tasks per processor. So $n \leq 2m$.
  - If $1 \leq n \leq m$, then LPT and optimal schedule both put one task per processor.
  - If $m < n \leq 2m$, then optimal schedule is to put tasks in nonincreasing order on processors 1,…,m, then on m,…,1.
    - LPT also schedules tasks this way, so it's optimal.



# LS VS. LPT

- LPT gives better approximation ratio, has same running time. Why bother with LS?
- LS is online.
  - Imagine the jobs are coming one by one.
    - LS just puts them on any idle computer.
- LPT is offline
  - It needs to know all the jobs that will ever arrive, in order to sort them.
- In a realistic parallel computation, you get jobs on the fly.
  - Online is more realistic.
  - LS is usually more useful.