



Approximation algorithms 2

Scheduling, Knapsack

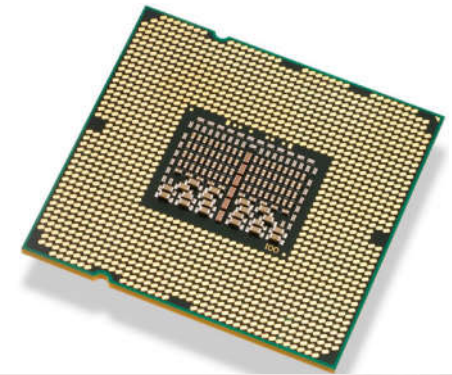
CS240

Spring 2020

Rui Fan

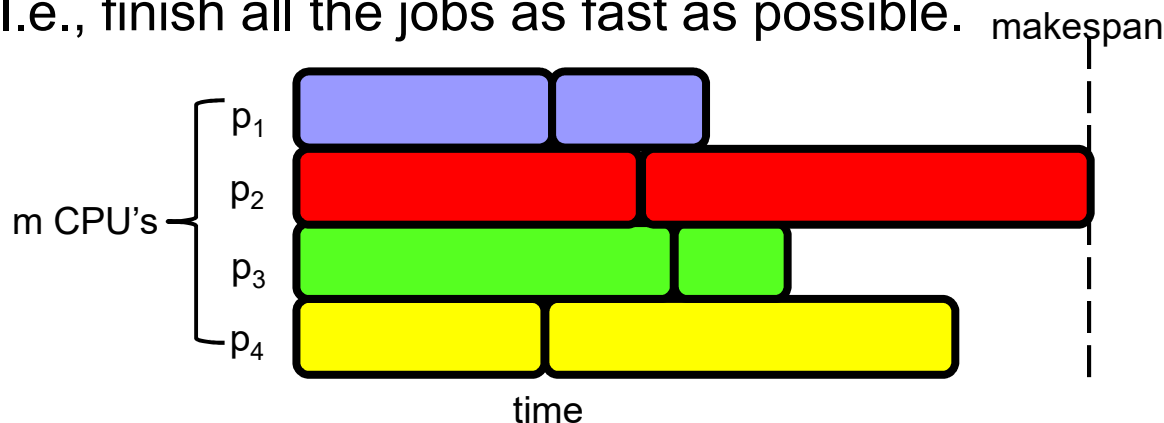
Parallel computing and scheduling

- 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 $J = S \cup \{s-2t\}$, and schedule them on 2 processors.

Minimizing makespan is NPC

- **Claim** If some subset of S sums to t , then min makespan is $s-t$.
- **Proof** Say $S' \subseteq S$ 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 $O(n \log n)$ time.
 - Compare this to $n! C(n+m-1, m-1)$ time to try all possible schedules and pick the best.

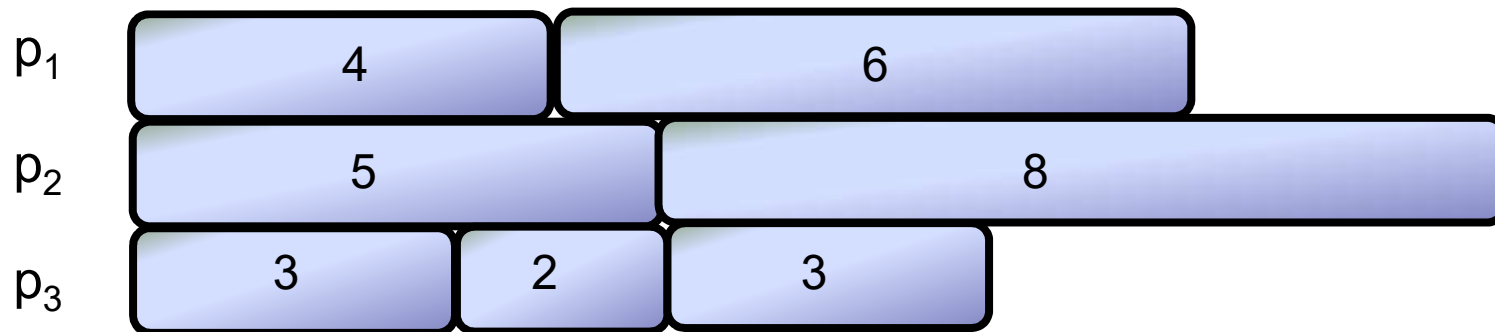


Graham's list scheduling

- 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.

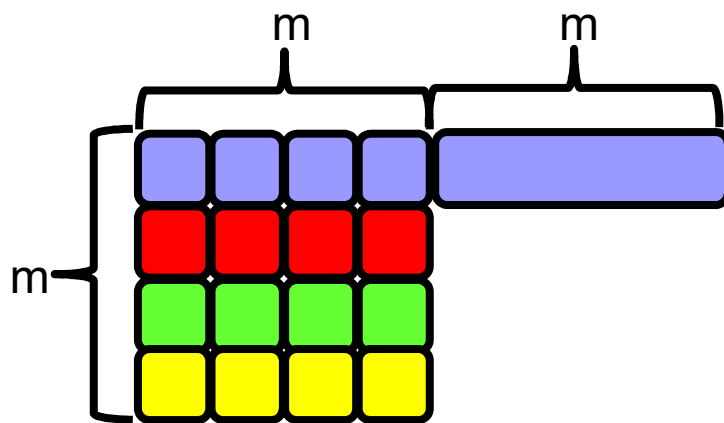
Example

- 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.

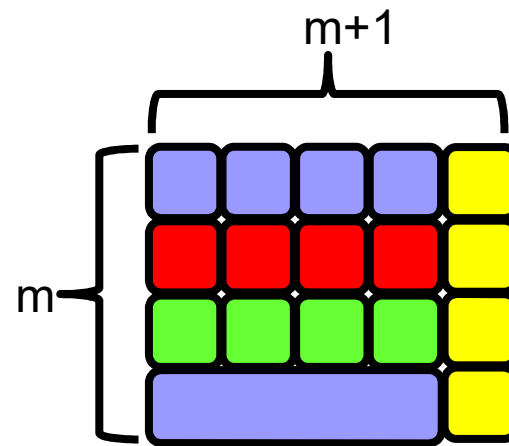


The worst case for LS

- 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 $1, 1, 1, \dots, 1, m$.
 - LS has makespan $2m$. Optimal makespan is $m+1$.
 - $\text{makespan}(\text{LS}) / \text{makespan}(\text{opt}) = 2m/(m+1) \approx 2$.
- This is worst possible case for list scheduling.



$$\text{makespan}(\text{LS}) = 2m$$



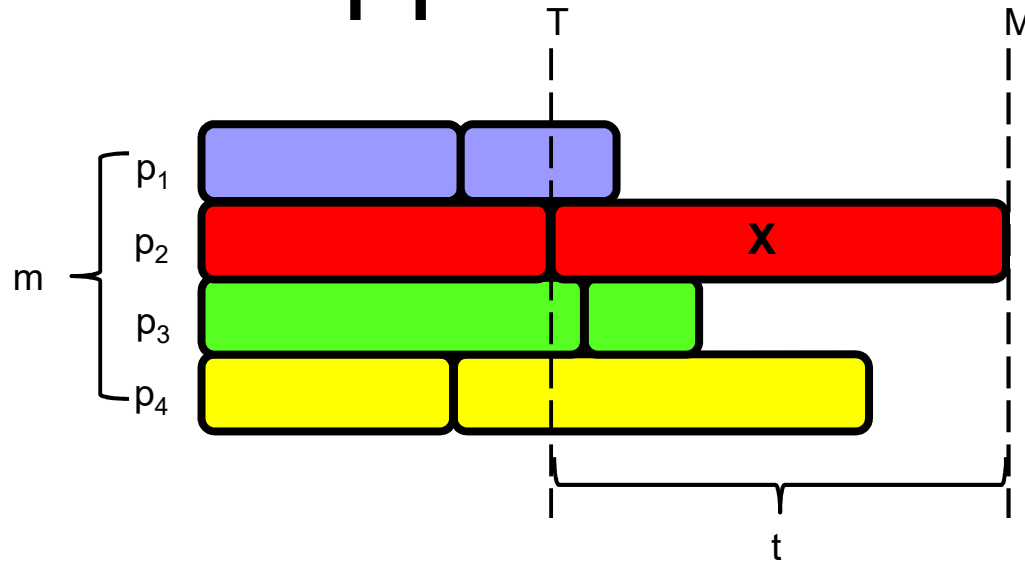
$$\text{makespan}(\text{opt}) = m+1$$



LS is a 2-approximation

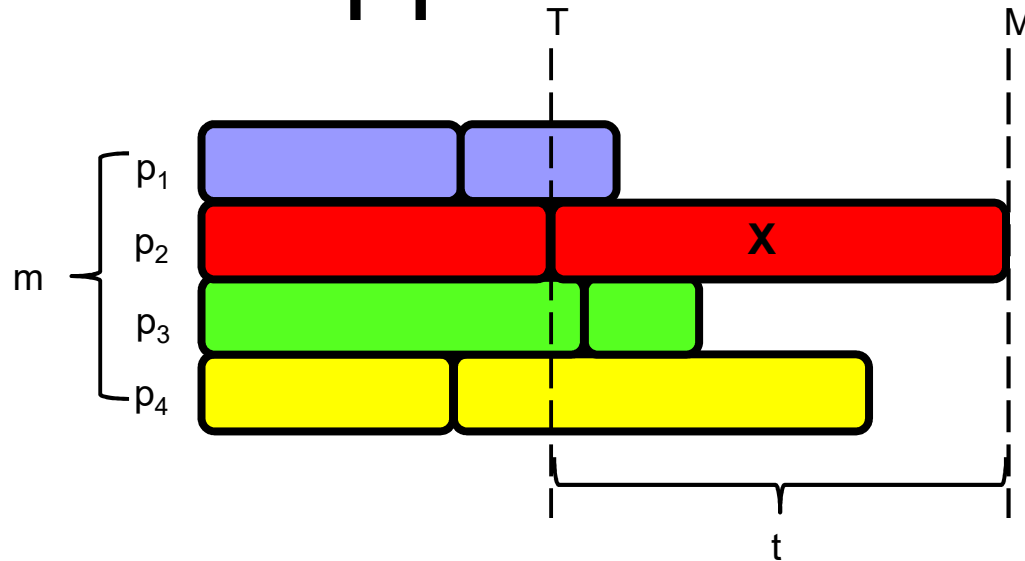
- Next, we prove LS always gives a schedule at most twice the optimal.
- Suppose LS gives makespan of M .
- Let the optimal schedule have makespan M^* .
- We prove that $M \leq 2M^*$.

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.

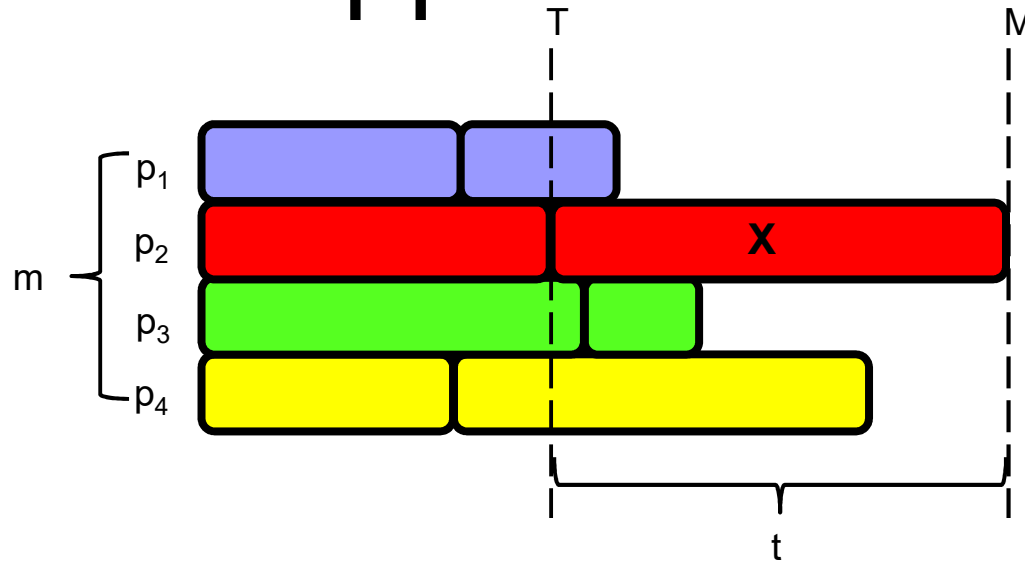
LS is a 2-approximation



■ 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 $\geq mT$.
- In the opt schedule, m processors complete at most m units of work per time unit.
- So length of opt schedule is $\geq (\text{total work})/m \geq mT/m = T$.

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 $4/3$.

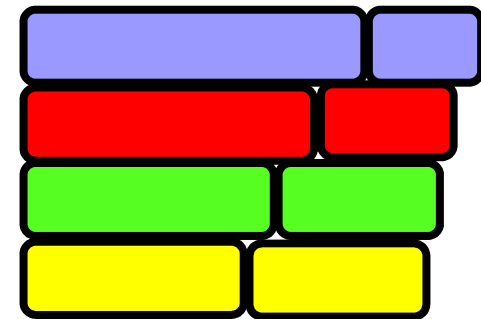


LPT is a $4/3$ -approximation

- **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.

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 $1, \dots, m$, then on $m, \dots, 1$.
 - LPT also schedules tasks this way, so it's optimal.



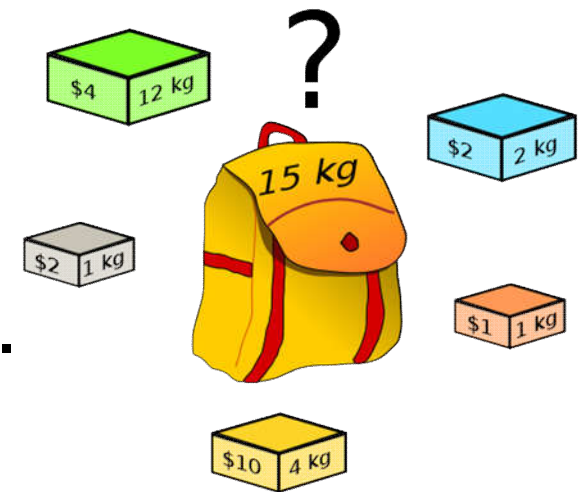


LS vs LPT

- LPT gives better approx 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.

The knapsack problem

- We have a set of items, each having a weight and a value.
- We have a knapsack that can carry up to W amount of weight.
- We want to put items in the knapsack to maximize the total value, but not exceed the weight limit.
- **Ex** Items 3 and 4 are the highest value items with weight ≤ 11 .
- Assume all items have weight $\leq W$, i.e. any single item fits in knapsack.



$W = 11$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

A dynamic program for knapsack

- Let $\text{OPT}(i,v)$ = minimum weight of a subset of items $1,\dots,i$ that has value $\geq v$.
- If optimal solution uses item i .
 - Then we pay w_i weight for item i , and need to achieve value $\geq v-v_i$ using items $1,\dots,i-1$ using min weight.
 - So $\text{OPT}(i,v)=w_i+\text{OPT}(i-1,v-v_i)$.
- If optimal solution doesn't use item i .
 - Then we need to achieve value $\geq v$ using items $1,\dots,i-1$.
 - So $\text{OPT}(i,v)=\text{OPT}(i-1,v)$.
- Choose the case that gives smaller weight.
- $$\text{OPT}(i,v) = \begin{array}{ll} 0 & \text{if } v=0 \\ \infty & \text{if } i=0, v>0 \\ \min(\text{OPT}(i-1,v), w_i+\text{OPT}(i-1,v-v_i)) & \text{otherwise} \end{array}$$



Running time of dynamic program

- Say there are n items, and the largest value of any item is v^* .
- The max value we can pack into the knapsack is nv^* , where v^* is the largest v value.
- Solve all subproblems of the form $\text{OPT}(i,v)$, where $i \leq n$ and $v \leq nv^*$.
 - This is a total of $O(n^2v^*)$ subproblems.
- The solution to Knapsack is the max value V that can be packed with weight $\leq W$.
- Having solved all the subproblems, we can find V by finding the subproblem with the largest value that has optimum weight $\leq W$.
 - $V = \max_{v \leq nv^*} \text{OPT}(n,v) \leq W$.
- So solving Knapsack takes total time $O(n^2v^*)$.



Running time of dynamic program

- The DP gives an optimal solution to Knapsack and takes $O(n^2v^*)$ time. Have we found a polytime algorithm for an NP-complete problem?
- No. The problem size is $O(n \log(v^*))$, because it takes $\log(v^*)$ bits to express each item's value. But $O(n^2v^*)$ is not polynomial in $n \log(v^*)$.
- To make this DP fast, we have to make the largest value small.



PTAS

- Let $\varepsilon > 0$ be any number. We'll give a $(1+\varepsilon)$ -approximation for knapsack.
- By setting ε sufficiently small, we can get as good an approximation as we want!
 - This type of algorithm is called a polynomial time approximation scheme, or PTAS.
- Contrast this with earlier algs we studied, which had worse approx ratios, e.g. 2 or $\log n$.
- But the running time will be $O(n^3/\varepsilon)$. Hence we can't set $\varepsilon=0$ get the optimal solution.
- We're trading accuracy for time. The more accurate (smaller ε), the more time the algorithm takes.

Main idea: rounding

- Since we only need an approximate solution, we can change the values of the items a little (round the values) and not affect the solution much.
- We scale and round the original values to make them small.
- The previous DP took $O(n^2v^*)$ time. So if the rounded values are small, this DP is fast.

W = 11						W = 11		
Item	Value	Weight				Item	Value	Weight
1	134,221	1				1	2	1
2	656,342	2				2	7	2
3	1,810,013	5				3	19	5
4	22,217,800	6				4	23	6
5	28,343,199	7				5	29	7



Rounding

- Let $\varepsilon > 0$ be the precision we want.
- Set $\theta = \varepsilon v^*/2n$ to be a scaling factor.
 - v^* is the largest value of any item.
- Scale all values down by θ then round up.
 - $v' = \lceil v/\theta \rceil$.
- Make a problem where each value v_i is replaced by v'_i .
 - Call this the scaled rounded problem.
- Let v^\wedge be max value in the scaled rounded problem. Then $v^\wedge = \lceil v^*/\theta \rceil = \lceil v^*/(\varepsilon v^*/2n) \rceil = \lceil 2n/\varepsilon \rceil$.
- Running time of DP on scaled rounded problem is $O(n^2 v^\wedge) = O(n^3/\varepsilon)$.



Solving the original problem

- Make another new problem in which each value v_i is replaced by $u_i = \lceil v_i/\theta \rceil * \theta$.
 - Call this the rounded problem.
 - We have $u_i \geq v_i$, and $u_i \leq v_i + \theta$.
- Note u values are equal to v' values multiplied by θ .
 - Thus, the optimal solution for the rounded problem and the scaled rounded problem are the same.
- We now have 3 problems, the original problem, the scaled rounded problem, and the rounded problem.
- Let S be the optimal solution to the scaled rounded problem, which we can find in time $O(n^3/\varepsilon)$. S is also optimal for the rounded problem.
- We'll show S is a $1+\varepsilon$ approximation for the original problem.

Correctness

- **Thm** Let S^* be the optimal solution to the original problem. Then $(1+\varepsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$.
Hence S is a $(1+\varepsilon)$ -approximate solution.

- **Proof**

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} u_i$$

$$u_i \geq v_i$$

$$\leq \sum_{i \in S} u_i$$

S is opt soln for rounded problem

$$\leq \sum_{i \in S} (v_i + \theta)$$

$$u_i \leq v_i + \theta$$

$$\leq \sum_{i \in S} v_i + n \theta$$

$$|S| \leq n$$

Correctness

- Suppose item j has the largest value, so

$$v^* = v_j. \text{ Then } n\theta = \frac{\varepsilon}{2} v_j \leq \frac{\varepsilon}{2} u_j \leq \frac{\varepsilon}{2} \sum_{i \in S} u_i$$

□ Last inequality because item j itself is feasible solution, so opt solution S is no smaller.

- So $\sum_{i \in S} v_i \geq \sum_{i \in S} u_i - n\theta \geq \left(\frac{2}{\varepsilon} - 1\right) n\theta$, where first inequality comes from last page.

- Assuming $\varepsilon \leq 1$, then $n\theta \leq \varepsilon \sum_{i \in S} v_i$

- Finally, we have

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S} v_i + n\theta \leq \sum_{i \in S} v_i + \varepsilon \sum_{i \in S} v_i = (1 + \varepsilon) \sum_{i \in S} v_i$$



Summary

- We gave a DP for Knapsack.
- We scale and round to reduce number of different item values.
- Running the DP on the scaled rounded problem and using the solution for the original problem leads to an arbitrarily good approximation for Knapsack, a PTAS.
- There are PTAS's for a number of other problems.
 - Multiprocessor scheduling.
 - Bin packing.
 - Euclidean TSP.
- However, there are also many problems for which PTAS's do not exist, unless $P=NP$.