

**ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ**



ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

M.Sc. Program in Computer Science Department of Informatics

Design and Analysis of Algorithms

Problems with Sets and Partitions

Vangelis Markakis

markakis@gmail.com

Weighted set problems

SUBSET SUM

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$, and a positive integer W

Q: is there $A \subseteq S$ s.t. $\sum_{i \in A} w_i = W$?

PARTITION

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$

Q: is there $A \subseteq S$ s.t. $\sum_{i \in A} w_i = \sum_{i \in S-A} w_i (= \frac{1}{2} \sum_{i \in S} w_i)$?

0-1 KNAPSACK

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , and a value v_i , $i=1, \dots, n$, and a positive integer W

Q: find $A \subseteq S$ s.t. $\sum_{i \in A} w_i \leq W$ and $\sum_{i \in A} v_i$ is maximized

Weighted set problems

BIN PACKING

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$, and a positive integer W

Q: find a partition of S into A_1, \dots, A_m s.t. $\sum_{i \in A_j} w_i \leq W, j = 1, 2, \dots, m$ and m is minimized

i.e., minimize the number of bins to fit the objects

MAKESPAN ($P || C_{\max}$)

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$, and a positive integer m

Q: find a partition of S into A_1, \dots, A_m s.t. $\max_{1 \leq j \leq M} \left\{ \sum_{i \in A_j} w_i \right\}$ is minimized

Weighted set problems

- All these problems are NP-complete

E.g.:

- SUBSET-SUM \leq_p PARTITION
- PARTITION \leq_p BIN PACKING
- BIN PACKING \leq_p MAKESPAN
- PARTITION \leq_p MAKESPAN

A. SUBSET SUM and PARTITION

SUBSET SUM

We have already seen a pseudo-polynomial time algorithm in the first lectures.

We recall it here for the sake of completeness

- Let $S_i = \{w_1, w_2, \dots, w_i\}$
- **IDEA:** Compute the sums of all subsets of S_i using the sums of all subsets of S_{i-1} (exclude sums $> W$)
- Let L be a list of integers
- Notation: $L+b$ = a new list with all elements of L increased by b
e.g., if $L = [1, 2, 3, 5]$, $L+2 = [3, 4, 5, 7]$
- Recall the auxiliary method MERGE (L, L')
 - **Input:** 2 sorted lists of integers, L and L'
 - **Output:** a sorted list that is the merge of L and L' with no duplicates
 - Complexity $O(|L|+|L'|)$

SUBSET SUM

L_i : list of the sums of all subsets of S_i (keep only sums $\leq W$)

```
Algorithm SubsetSum (S, W) ;  
L0=[0] ;  
for i=1 to n do  
    Li=MERGE (Li-1, Li-1+wi) ;  
    Remove from Li every element > W ;  
Check if the largest element in L equals W ;
```

Example

$S=\{1,4,5\}$, $n=3$, $W=8$

$L_0=[0]$ $L_0+a_1=[1]$

$L_1=[0,1]$ $L_1+a_2=[4,5]$

$L_2=[0,1,4,5]$ $L_2+a_3=[5,6,9,10]$

$L_3=[0,1,4,5,6]$ **Answer: NO**

Complexity ?

SUBSET SUM

Complexity: $O(nW)$

At every step, the list we keep has at most W elements

- Not polynomial
- But pseudo-polynomial!

PARTITION

- Tightly related to SUBSET SUM
- $\text{SUBSET-SUM} \leq_p \text{PARTITION}$, hence NP-complete as well
- We could use the algorithm for SUBSET SUM, setting $W = \frac{1}{2} \sum w_i$
- PARTITION is also a special case of scheduling problems
- To solve PARTITION, think of 2 identical processors, with processing times equal to w_i
- Minimizing the makespan would tell us if there exists a solution to the PARTITION problem

B. Scheduling problems

Scheduling Problems

Any problem where

- We have a set of jobs/tasks
- We have a set of processors
- We want to assign the jobs to the processors so as to optimize some criterion

A plethora of such problems have been studied since the 60s

Variations:

- **Criterion to optimize:** makespan, throughput, sum of (weighted) completion times,...
- **Processors:** may be completely unrelated (each with a different speed), identical, or uniformly related
- **Jobs:** they may have arrival times or deadlines, precedence constraints (cannot execute job i before job j finishes), option for preemption (execute only one part of the job now and continue later with the rest)

Scheduling Problems

We will focus on makespan

MAKESPAN (P || C_{\max})

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$, and a positive integer m

Q: find a partition of S into A_1, \dots, A_m s.t. $\max_{1 \leq j \leq M} \left\{ \sum_{i \in A_j} w_i \right\}$ is minimized

In other words (in scheduling terminology):

Given a set of n independent jobs J_1, J_2, \dots, J_n

And a set of m identical machines M_1, M_2, \dots, M_m

p_i : the execution time of job i on any machine (i.e., the weight w_i)

Given an assignment of jobs to machines

C_j : the time job j finishes its execution

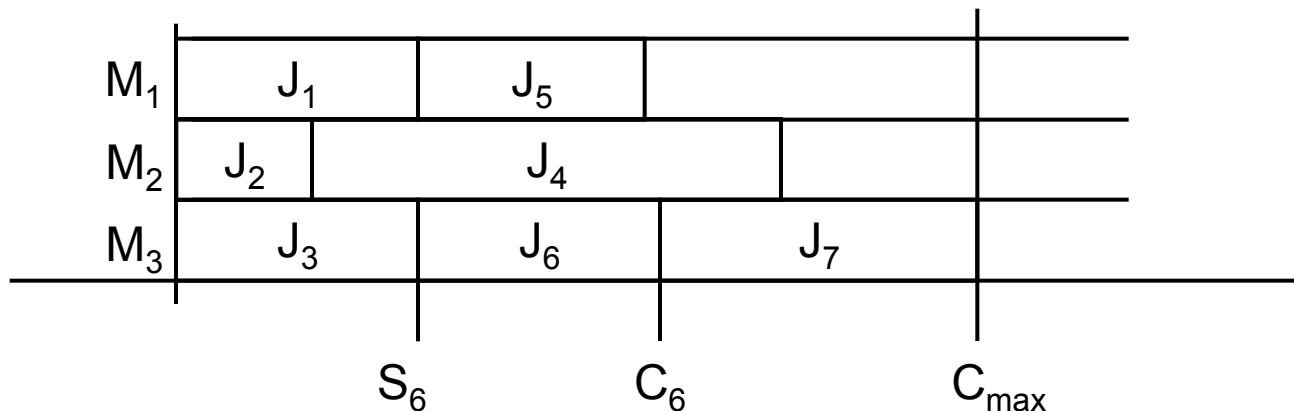
Problem: schedule the jobs on the machines

in order to minimize $C_{\max} = \max_j \{C_j\}$

Makespan (P | | C_{max})

List scheduling - A general scheduling methodology

- Construct a list of jobs (an ordering of the jobs according to some criterion)
- Whenever a machine becomes available, the next job in the list is scheduled on that machine



S_j: the time job j starts its execution

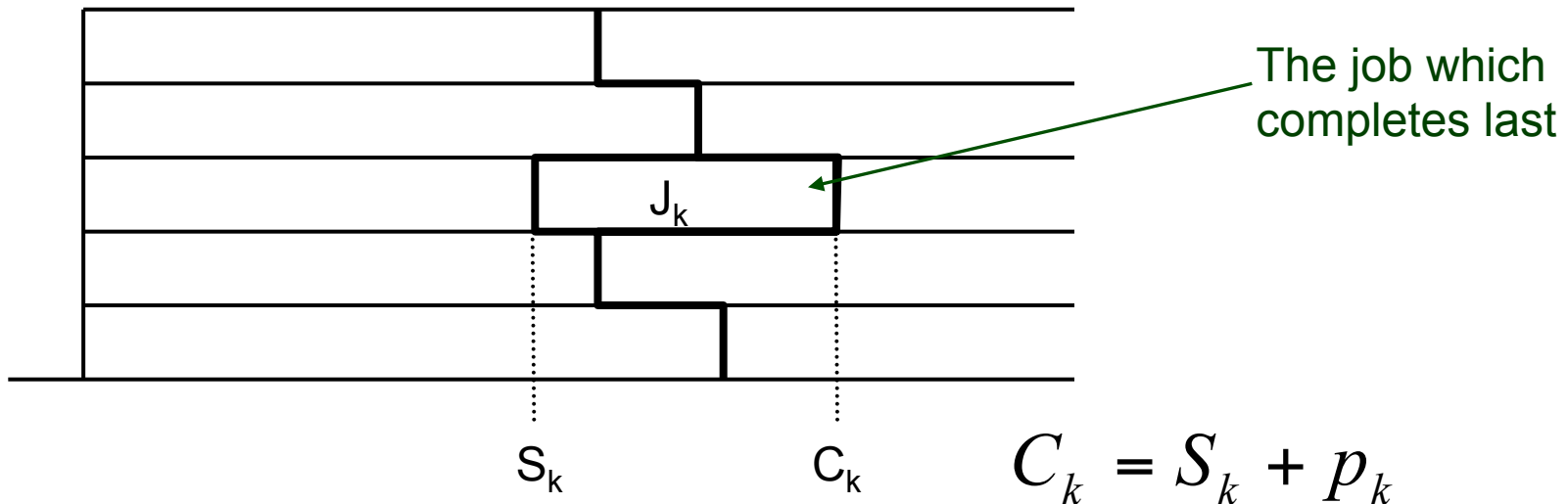
C_j: the time job j finishes its execution

Makespan (P | | C_{max}): A 2-approximation

Theorem [Graham, 1966]:

List scheduling using an arbitrary order of the jobs
yields a $(2 - 1/m)$ -approximation algorithm for P | | C_{max}

Proof:



$$C_k = S_k + p_k$$

$$C_{\max} = C_k$$

Makespan (P | | C_{max}): A 2-approximation

Proof (cont.):

- We need a lower bound for OPT
- We will actually use 2 lower bounds
- Let C* be the makespan of an optimal solution

$$C^* \geq p_k \quad (1) \quad (\text{in fact the makespan is at least as big as any } p_i)$$

$$C^* \geq \frac{1}{m} \sum_{i=1}^n p_i \quad (2) \quad (\text{best case is if all machines finish at the same time})$$

$$\sum_{i=1}^n p_i \geq m S_k + p_k \Rightarrow S_k \leq \frac{1}{m} \left(\sum_{i=1}^n p_i - p_k \right) \quad (3)$$

Makespan (P | | C_{max}): A 2-approximation

Proof (cont.):

Let C = cost of the algorithm's solution

$$C = C_k = S_k + p_k \stackrel{(3)}{\leq} \frac{1}{m} \left(\sum_{i=1}^n p_i - p_k \right) + p_k$$

$$= \frac{1}{m} \sum_{i=1}^n p_i + \left(1 - \frac{1}{m}\right) p_k$$

$$\stackrel{(1),(2)}{\leq} C^* + \left(1 - \frac{1}{m}\right) C^*$$

Hence: $C \leq (2 - 1/m) C^* = (2 - 1/m) \text{OPT}$

Tightness of $\rho=2$

Graham's ratio of $2-1/m$ is tight

Family of instances (parameterized by m , the number of processors):

- $m^2 - m + 1$ jobs in total
- $m^2 - m$ jobs of processing time 1
- 1 job of processing time m
- m machines

If the long job is last in the list, then $C = (m^2 - m)/m + m = m - 1 + m = 2m - 1$

The optimal schedule has length $C^* = m$ (why ?)

$$C/C^* = (2m - 1) / m = 2 - 1/m$$

Makespan (P | | C_{max}): A 3/2-approximation

List scheduling using LPT (Longest Processing Time first)

Drawback with previous approach:

- List created arbitrarily
- Long jobs at the end of the list may cause a large makespan

Theorem: List scheduling with the LPT rule yields a 3/2-approximation for P | | C_{max}

Proof:

- Let $p_1 \geq p_2 \geq \dots \geq p_m \geq p_{m+1} \geq \dots \geq p_k \geq \dots \geq p_n$ be the processing times
- Let J_k be the job that finishes last under LPT (say on machine say M_i)

□ If J_k is the only job on M_i , then the schedule is optimal ($OPT \geq p_k$)

□ Otherwise, M_i executes at least two jobs, that is $k \geq m+1$

Makespan (P | | C_{max}): A 3/2-approximation

As there are at least $m+1$ jobs (or more), then two of the first $m+1$ jobs are executed on the same machine **in any optimal schedule** (pigeonhole principle)

Hence:
$$C^* \geq 2p_{m+1} \geq 2p_k \Rightarrow p_k \leq \frac{C^*}{2}$$

Using this new bound:

$$\begin{aligned} C &\leq \frac{1}{m} \sum_{i=1}^n p_i + \left(1 - \frac{1}{m}\right) p_k \quad (\text{as before}) \\ &\leq C^* + \left(1 - \frac{1}{m}\right) \frac{C^*}{2} \\ &\leq C^* \left(1 + \frac{1}{2} - \frac{1}{2m}\right) \\ &= C^* \left(\frac{3}{2} - \frac{1}{2m}\right) \end{aligned}$$

Makespan ($P \parallel C_{\max}$): Can we do better?

Is the $(3/2 - 1/2m)$ -ratio for LPT scheduling tight ?

NO !

The lower bounds we have used in the analysis are too generous!

Theorem [Graham, 1969]:

List scheduling using LPT (Longest Processing Time first) yields a $(4/3 - 1/3m)$ -approximation algorithm for makespan

Makespan (P | | C_{\max}): Tightness for $\rho=4/3$

The $(4/3 - 1/3m)$ ratio for LPT scheduling is tight

Family of instances (parameterized by m , the number of processors):

- $2m + 1$ jobs in total
- 2 jobs for each of the weights $m+1, m+2, \dots, 2m-2, 2m-1$
 - $2m-2$ jobs
- 3 jobs of weight m
- m machines

Example:

$m = 5$

2 jobs of each of the weights 6, 7, 8, 9

(total 8)

3 jobs of weight 5

(3)

5 machines

total 11 (= $2m+1$)

Makespan (P | | C_{max}): Tightness for $\rho=4/3$

Example (cont.):

LPT

9	5	5	
9	5		
8	6		
8	6		
7	7		

C = 19

Optimal

9	6		
9	6		
8	7		
8	7		
5	5	5	

C* = 15

$$\frac{C}{C^*} = \frac{19}{15}$$

$$\frac{4}{3} - \frac{1}{3m} = \frac{4}{3} - \frac{1}{15} = \frac{19}{15}$$

Makespan (P | | C_{max})

Let us recap on List scheduling

arbitrary list: $\rho = 2 - 1/m$

LPT: $\rho = 4/3 - 1/3m$

Can we go beyond 4/3 ?

Another algorithm

- Find an optimal schedule for the k largest jobs (for some parameter k , to be determined later)
- Schedule the rest of the jobs using LPT rule

We will find the approximation of this algorithm as a function of k , $\rho = f(k)$

Makespan (P | | C_{\max}): $\rho=f(k)$

Let r be the completion time of the k longer jobs

C = makespan of the algorithm

C^* = optimal makespan

Case a) $C = r \Rightarrow C = C^*$ (this is the easy case)



Makespan (P | | C_{max}): ρ=f(k)

Case b) C > r

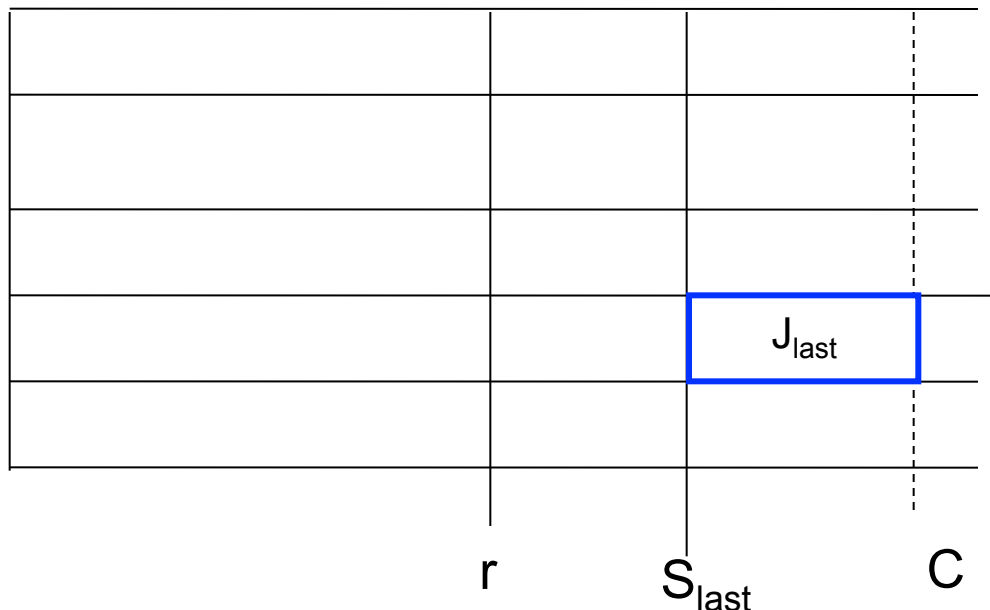
Let J_{last} be the job which finishes last

$$p_1 \geq p_2 \geq \dots \geq p_k \geq p_{k+1} \geq p_{k+2} \geq \dots \geq p_{last} \geq p_{last+1} \geq \dots \geq p_n, \quad n > m$$

(if $n < m$, the problem is trivial)

$$last \geq k+1 \Rightarrow p_{last} \leq p_{k+1}$$

All machines are busy till time $S_{last} \Rightarrow \sum_{i=1}^n p_i \geq m S_{last} + p_{last}$



$$S_{last} = C - p_{last}$$

Makespan (P | | C_{max}): ρ=f(k)

$$\sum_{i=1}^n p_i \geq m S_{last} + p_{last} = m(C - p_{last}) + p_{last}$$

$$C^* \geq \frac{1}{m} \sum_{i=1}^n p_i \geq \frac{1}{m} [m(C - p_{last}) + p_{last}] = C - p_{last} + \frac{1}{m} p_{last}$$

$$\Rightarrow C \leq C^* + \frac{m-1}{m} p_{last}$$

$$\Rightarrow C \leq C^* + \frac{m-1}{m} p_{k+1}$$

$$\Rightarrow \frac{C}{C^*} \leq 1 + \frac{(1-1/m)p_{k+1}}{C^*}$$

Makespan (P | | C_{max}): ρ=f(k)

$$\frac{C}{C^*} \leq 1 + \frac{(1 - 1/m)p_{k+1}}{C^*}$$

Consider the first k+1 jobs: $p_i \geq p_{k+1}$

At least one processor executes $\lfloor \frac{k}{m} \rfloor + 1$ of them

Hence,

$$C^* \geq (1 + \lfloor k/m \rfloor) p_{k+1}$$
$$\Rightarrow \frac{C}{C^*} \leq 1 + \frac{(1 - 1/m)p_{k+1}}{(1 + \lfloor k/m \rfloor) p_{k+1}} = 1 + \frac{1 - 1/m}{1 + \lfloor k/m \rfloor}$$

Makespan (P || C_{max}): $\rho=f(k)$

Recall the definition of PTAS and FPTAS:

- Polynomial Time Approximation Schemes (PTAS)
 - $C/C^* \leq 1 + \varepsilon$, for any $\varepsilon > 0$
 - Complexity: $O(\text{poly}(|I|))$
 - dependence on ε : allowed to be $O(\exp(1/\varepsilon))$, e.g. $O(n^{3/\varepsilon})$
- Fully Polynomial Time Approximation Schemes (FPTAS)
 - $C/C^* \leq 1 + \varepsilon$, for any $\varepsilon > 0$
 - Complexity: $O(\text{poly}(|I|))$
 - Dependence on ε : $O(\text{poly}(1/\varepsilon))$! e.g., $O((1/\varepsilon)^2 n^3)$

Makespan (P | | C_{max}): A PTAS

A PTAS from our new algorithm:

1. Given $\varepsilon > 0$, choose k such that $\frac{1 - \frac{1}{m}}{1 + \left\lfloor \frac{k}{m} \right\rfloor} \leq \varepsilon \quad \left(\Rightarrow k > \frac{m-1}{\varepsilon} - m \right)$

2. Schedule optimally the k longest jobs

3. Schedule the rest of the jobs using LPT

$$\frac{C}{C^*} \leq 1 + \varepsilon, \quad \forall \varepsilon > 0$$

Makespan (P | | C_{max}): A PTAS

Complexity

Step 1. $O(1)$

Step 2. $O(m^k)$ (why ?)

Step 3. $O(n \log n)$ (why ?)

$$\text{Total: } O(n \log n + m^k) \sim O(n \log n + m^{\frac{m-1}{\varepsilon} - m})$$

→ $O(\text{poly}(n))$

→ $O(\text{poly}(m))$ for **FIXED** m (when $m=O(1)$)

→ $O(\exp(1/\varepsilon))$

A PTAS for constant m

-
- There is also an FPTAS for fixed m
 - And there is a PTAS for $P | | C_{\max}$ for any m [Hochbaum, Shmoys '87]
 - There is no FPTAS, for $P | | C_{\max}$, unless $P = NP$

C. Knapsack problems




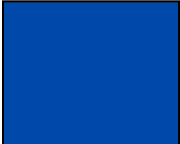
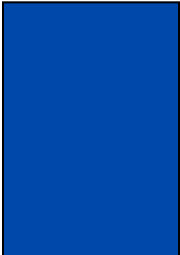
Knapsack problems

- We are given a knapsack with maximum capacity W , and a set $S = \{1, 2, \dots, n\}$ of n items
- Each item i has a weight w_i and a value v_i
 - assume all w_i , v_i and W are integers

Problem: How to pack the knapsack to achieve maximum total value of packed items?

Knapsack problems

Max weight: $W = 20$

Items	Weight w_i	Value v_i
	2	3
	3	4
	4	5
	5	8
	9	10

$W = 20$

Knapsack problems

Three (basic) versions of the problem:

1. Fractional knapsack

Items are divisible: **any fraction of an item can go into the knapsack**
poly-time solvable by a greedy algorithm

2. 0-1 knapsack

Items are indivisible: **either take an item or not**
NP-complete, $O(nW)$, by a dynamic programming algorithm,
PTAS based on DP

3. Integer knapsack

Multiple copies of indivisible items: **take any number of copies of an item**

Exercise !

Knapsack problems

Fractional knapsack

$$\max \sum_{i \in S} v_i x_i, \text{ s.t. } \sum_{i \in S} w_i x_i \leq W, \text{ and } x_i \in [0,1]$$

0-1 knapsack

$$\max \sum_{i \in S} v_i x_i, \text{ s.t. } \sum_{i \in S} w_i x_i \leq W, \text{ and } x_i \in \{0,1\}$$

Integer knapsack

$$\max \sum_{i \in S} v_i x_i, \text{ s.t. } \sum_{i \in S} w_i x_i \leq W, \text{ and } x_i \in \mathbb{N}$$

Fractional Knapsack

Greedy algorithm:

- Start with an empty knapsack
- Consider the item with the maximum value per unit (v_i/w_i) among the remaining items,
- Take as much quantity of this item as the capacity of the knapsack allows

Note: at the end of the algorithm, the knapsack is loaded by the whole weights of all chosen items, except possibly the last included item

Complexity: $O(n \log n)$

Fractional Knapsack

Example:

Item	Weight	Value
1	10	60
2	20	100
3	30	120

Suppose $W=50$

The greedy algorithm will select:

- All of item 1
- All of item 2
- $2/3$ of item 3

Fractional Knapsack: Correctness

Optimal sub-structure property:

- Consider an optimal solution with weight W
- Suppose we remove a weight of w of an item j from the optimal load
- Then, the remaining load must be optimal for a knapsack of capacity $W-w$

Proof of correctness:

- Greedy “stays ahead” of the optimal solution at each step
- E.g., if the optimal solution does not take all of the item with the highest v_i/w_i , then we could replace some part of the optimal solution with the choice of greedy and get a better solution
- Do it as an exercise

0-1 Knapsack

Brute-force approach

- there are 2^n possible combinations of n items
- Go through all combinations and find the one with the most total value and with total weight less or equal to W
- Running time: $O(2^n)$

Can we do better?

- Yes, with an algorithm based on dynamic programming

DP for 0-1 Knapsack

Subproblem:

$I(k, w)$ = sub-problem consisting of the first k items $S_k = \{1, 2, \dots, k\}$ and capacity w

$V[k, w]$ = optimal solution of the subproblem $I(k, w)$

Item k can either be in the optimal solution of $V[k, w]$ or not

- First case: $w_k > w$
 - item k cannot be in the optimal solution of $V[k, w]$
 - $V[k, w] = V[k-1, w]$
- Second case: $w_k \leq w$
 - item k can be in the optimal solution
 - the best solution for $V[k, w]$ is one of the next two:
 - $V[k-1, w]$ or
 - the best solution for $I(k-1, w-w_k)$ plus the value of item k : $V[k-1, w-w_k] + v_k$
 - $V[k, w] = \max \{ V[k-1, w], V[k-1, w-w_k] + v_k \}$

DP for 0-1 Knapsack

Recursive Formula

$$V[k, w] = \begin{cases} 0 & \text{if } k = 0 \text{ or } w = 0 \\ V[k - 1, w] & \text{if } k, w \geq 1 \text{ and } w_k > w \\ \max \{V[k - 1, w], V[k - 1, w - w_k] + v_k\} & \text{if } k, w \geq 1 \text{ and } w_k \leq w \end{cases}$$

What we want is $\text{OPT} = V[n, W]$

DP for 0-1 Knapsack

0-1 Knapsack Value-only($\{w_i\}$, $\{v_i\}$, W)

```
for w := 0 to W do V[0,w] := 0;
```

```
for k = 1 to n do
```

```
  for w := 0 to W do
```

```
    if  $w_k \leq w$  // item i can be in the solution
```

```
      then  $V[k,w] := \max \{ v_k + V[k-1,w-w_k], V[k-1,w] \}$ 
```

```
      else  $V[k,w] := V[k-1,w]$  //  $w_k > w$ 
```

Complexity: $O(nW)$

- Pseudopolynomial time algorithm
- Polynomial when W is small

Note: All known pseudopolynomial time algorithms for NP-hard problems are based on dynamic programming

DP for 0-1 Knapsack

Can we find the actual set of items we should include in the optimal solution?

```
0-1 Knapsack( $\{w_i\}$ ,  $\{v_i\}$ , W)
{
  Run 0-1 Knapsack Value-only algorithm
   $k := n$ ;  $w := W$ ;  $S := \{\}$ 
  While  $k \neq 0$  and  $w \neq 0$  do
    { if  $V[k, w] \neq V[k-1, w]$  then {  $S := S \cup \{k\}$ ;  $w := w - w_k$  }
       $k := k - 1$  }
  return S
}
```

complexity of this algorithm ?

Another DP for 0-1 Knapsack

Previous Algorithm:

OPT = V[n, W] and can be found in $O(nW)$ time

Another Algorithm:

Claim: OPT can be found in $O(n \text{ OPT})$, that is $O(n^2 v_{\max})$ time
where $v_{\max} = \max_i v_i$

Hint:

Subproblem: $C[k, v]$ = the minimum capacity yielding a value v using only the items 1, 2, ..., k

OPT = maximum v for which $C[n, v] \leq W$

Think of a DP algorithm based on $C[k, v]$

FPTAS for 0-1 KNAPSACK

$$\text{OPT} = \text{Max } \sum_i v_i x_i$$

$$\text{such that: } \sum_i x_i w_i \leq W$$

$$\text{and: } x_i \in \{0, 1\}, 1 \leq i \leq n$$

- We will utilize the $O(n^2 v_{\max})$ dynamic programming algorithm for 0-1 KNAPSACK
- Recall that $v_{\max} = \max_i \{v_i\}$ and $v_{\max} \leq \text{OPT} \leq n v_{\max}$
- Recall also that we have assumed all quantities are integers (the w_i 's, the v_i 's, and W)

FPTAS for 0-1 KNAPSACK

Main ingredients for designing an FPTAS

- Inspired by the dynamic programming algorithm
- The DP algorithm implies that if the values are small (polynomial in n), then we can solve the problem efficiently
- **Idea:** round down the values by ignoring some of their least significant bits
- Solve the “rounded” or “scaled” instance (which can be seen as a “perturbation” of the original instance)
- Prove that the solution found is a good approximation to the original instance
- The scaling should be dependent on ϵ
- The scaled values should be bounded by a polynomial in n and $1/\epsilon$

FPTAS for 0-1 KNAPSACK

SCALED INSTANCE:

Scale all item values by a parameter k , i.e., for item j , $v_j(k) = \lfloor v_j / k \rfloor$

FPTAS-Knapsack(k)

```
{ Produce the scaled instance;  
  Solve the scaled problem by the last DP algorithm;  
  Let  $S(k) \subseteq \{1,2,\dots,n\}$  be the optimal solution to the scaled problem;  
  Return  $S(k)$  for the original problem; }
```

The parameter k will be determined by the analysis

In particular:

Theorem: The algorithm above with $k = \varepsilon v_{\max}/n$ is an FPTAS for 0-1 KNAPSACK

FPTAS for 0-1 KNAPSACK

Proof:

- Let $S^* \subseteq \{1,2,\dots,n\}$ be the optimal solution of original problem, with value OPT
- Let $S(k) \subseteq \{1,2,\dots,n\}$ be the output of the algorithm, with value $OPT(k)$ for the original problem

(*) for the scaled instance, $S(k)$ is of greater value than any other solution (hence better than S^* too)

$$OPT(k) = \sum_{j \in S(k)} v_j \stackrel{\text{by (2)}}{\geq} \sum_{j \in S(k)} k v_j(k)$$

$$= k \sum_{j \in S(k)} v_j(k) \stackrel{\text{by (*)}}{\geq} k \sum_{j \in S^*} v_j(k) =$$

$$\stackrel{\text{by (1)}}{\geq} k \sum_{j \in S^*} \left(\frac{v_j}{k} - 1 \right) = \sum_{j \in S^*} v_j - k \sum_{j \in S^*} 1 = OPT - k|S^*|$$

$$\geq OPT - kn, \quad \text{since } |S^*| \leq n$$

$$\frac{v_j}{k} - 1 \stackrel{(1)}{<} v_j(k) = \left\lfloor \frac{v_j}{k} \right\rfloor \stackrel{(2)}{\leq} \frac{v_j}{k}$$

FPTAS for 0-1 KNAPSACK

Proof (cont.):

Thus:

$$OPT(k) \geq OPT - n \cdot k,$$

Choose $k = \frac{\varepsilon \cdot v_{\max}}{n} \leq \frac{\varepsilon \cdot OPT}{n}$, since $v_{\max} \leq OPT$

Hence, $OPT(k) \geq OPT - \varepsilon \cdot OPT = (1 - \varepsilon)OPT$

Complexity

$$O\left(n^2 \left\lceil \frac{v_{\max}}{k} \right\rceil\right), \text{ that is } O\left(n^3 \frac{1}{\varepsilon}\right), \text{ since } \left\lceil \frac{v_{\max}}{k} \right\rceil = \frac{n}{\varepsilon}$$

$O(\text{poly}(n, 1/\varepsilon))$, hence FPTAS !

D. BIN PACKING

Bin Packing

Recall the problem:

BIN PACKING

I: A set of objects $S = \{1, \dots, n\}$, each with a positive integer weight w_i , $i = 1, \dots, n$, and a positive integer W (bin capacity)

Q: find a partition of S into A_1, \dots, A_m (m bins) s.t. $\sum_{i \in A_j} w_i \leq W$, $j = 1, 2, \dots, m$ and m is minimized

i.e., minimize the number of bins to fit the objects

Negative news:

- There is no approximation algorithm achieving a factor better than $3/2$, unless $P = NP$
- The proof is by showing that it is NP-hard to distinguish between instances with $OPT = 2$ and instances with $OPT = 3$

Bin Packing

On the positive side:

Greedy algorithms can achieve constant approximations

First-Fit algorithm:

- Start with one empty bin
- Process the items in an arbitrary order
- Try to place the next item in one of the existing bins (if it fits)
- If not, then create a new bin and put it there

Theorem: First-Fit achieves a 2-approximation

Relatively simple (do it as an exercise)

Bin Packing

Improving on First-Fit:

First-Fit Decreasing algorithm (FFD):

- First sort the items in decreasing order
- Run First-Fit but by processing the items in this order

Theorem: FFD uses at most $11/9 \text{ OPT} + 1$ bins

Bin Packing

And further improvements:

- Bin Packing does not admit a PTAS (since we have 3/2-hardness result)
- But it does achieve an *Asymptotic* PTAS

Theorem [Fernandez de la Vega, Lueker, '81]:

For any $\epsilon > 0$, there exists an algorithm using at most $(1 + \epsilon)\text{OPT} + 1$ bins

- Asymptotic refers to the fact that when OPT grows the approximation ratio approaches 1