

Pseudo-Polynomial-Time Algorithms

A method for coping with **NP-complete** *number problems*

Subset Sum Problem (SS)

Instance: a sequence x_1, \dots, x_n of positive integers, and a positive integer k (the target sum)

Question: does there exist a subset S of $\{1, \dots, n\}$ such that
 $\sum_{r \in S} x_r = k$?

Example instance: $x_1=7, x_2=3, x_3=6, x_4=4, x_5=9, x_6=11$

If $k = 27$, the answer is yes

- e.g. if $S = \{2, 4, 5, 6\}$, $x_2 + x_4 + x_5 + x_6 = 3 + 4 + 9 + 11 = 27$

If $k = 32$, the answer is no

- SS is known to be **NP-complete** – Partition is a special case

A dynamic programming approach

- Parameterise: given an instance of **SS**, **SS(i, j)** is a decision problem which asks:
 - does there exist a subset **S** of $\{1, \dots, i\}$ such that $\sum_{r \in S} x_r = j$ ($0 \leq i \leq n$ and $0 \leq j \leq k$)?
 - let **B(i, j)** be the (Boolean) answer to the problem **SS(i, j)**
 - to solve **SS** for the original instance, we need to find **B(n, k)**

- Dynamic programming algorithm is based on the following *recurrence relation* for **SS**:

$$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \text{ for } i > 0$$

subject to

$$B(0, 0) = \text{true}, \text{ and } B(0, j) = \text{false} \quad \forall j > 0$$

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i	x_i	↓	↓	0	1	2	3	4	5	6	j	7	8	9	10	..	16	..	27
0																			
1	7																		
2	3																		
3	6																		
4	4																		
5	9																		
6	11																		

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i ↓	x_i ↓	0	1	2	3	4	5	6	j ↑	7	8	9	10	..	16	..	27
0		T	F	F	F	F	F	F		F	F	F	F	..	F	..	F
1	7																
2	3																
3	6																
4	4																
5	9																
6	11																

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \text{ for } i > 0$

Example: $x_1=7, x_2=3, x_3=6, x_4=4, x_5=9, x_6=11, k=27$

i	x_i	j														
↓	↓	0	1	2	3	4	5	6	7	8	9	10	..	16	..	27
0		T	F	F	F	F	F	F	F	F	F	F	..	F	..	F
1	7	T	F	F	F	F	F	F	T	F	F	F	..	F	..	F
2	3															
3	6															
4	4															
5	9															
6	11															

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \quad \text{for } i > 0$

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i	x_i	j														
↓	↓	0	1	2	3	4	5	6	7	8	9	10	..	16	..	27
0		T	F	F	F	F	F	F	F	F	F	..	F	..	F	
1	7	T	F	F	F	F	F	F	T	F	F	F	..	F	..	F
2	3	T	F	F	T	F	F	F	T	F	F	T	..	F	..	F
3	6															
4	4															
5	9															
6	11															

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \quad \text{for } i > 0$

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i ↓	x_i ↓	0	1	2	3	4	5	6	7	8	9	10	..	16	..	27
j																
0	T	F	F	F	F	F	F	F	F	F	F	..	F	..	F	
1	7	T	F	F	F	F	F	T	F	F	F	..	F	..	F	
2	3	T	F	F	T	F	F	F	T	F	F	T	..	F	..	F
3	6	T	F	F	T	F	F	T	T	F	T	T	..	T	..	F
4	4															
5	9															
6	11															

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \quad \text{for } i > 0$

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i ↓	x_i ↓	0	1	2	3	4	5	6	j ↑	7	8	9	10	..	16	..	27
0		T	F	F	F	F	F	F		F	F	F	..	F	..	F	
1	7	T	F	F	F	F	F	F	T	F	F	F	..	F	..	F	
2	3	T	F	F	T	F	F	F	T	F	F	T	..	F	..	F	
3	6	T	F	F	T	F	F	T	T	F	T	T	..	T	..	F	
4	4	T	F	F	T	T	F	F	T	T	F	T	T	..	T	..	F
5	9																
6	11																

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \quad \text{for } i > 0$

Example: $x_1=7, x_2=3, x_3=6, x_4=4, x_5=9, x_6=11, k=27$

i ↓	x_i ↓	0	1	2	3	4	5	6	j ↑	7	8	9	10	..	16	..	27
0		T	F	F	F	F	F	F		F	F	F	..	F	..	F	
1	7	T	F	F	F	F	F	F	T	F	F	F	..	F	..	F	
2	3	T	F	F	T	F	F	F	T	F	F	T	..	F	..	F	
3	6	T	F	F	T	F	F	T	T	F	T	T	..	T	..	F	
4	4	T	F	F	T	T	F	T	T	F	T	T	..	T	..	F	
5	9	T	F	F	T	T	F	T	T	F	T	T	..	T	..	F	
6	11	T	F	F	T	T	F	T	T	F	T	T	..	T	..	T	

$B(0, 0) = \text{true}$, and $B(0, j) = \text{false} \quad \forall j > 0$

$B(i, j) = B(i-1, j) \vee (x_i \leq j \wedge B(i-1, j - x_i)) \quad \text{for } i > 0$

- To determine the subset S , trace a path through the table from bottom right to top left

Example: $x_1=7$, $x_2=3$, $x_3=6$, $x_4=4$, $x_5=9$, $x_6=11$, $k=27$

i	x_i	0	1	2	3	4	5	6	7	8	9	10	..	16	..	27
j																
0		F	F	F	F	F	F	F	F	F	F	..	F	..	F	
1	7	T	F	F	F	F	F	F	F	F	F	..	F	..	F	
2	3	T	F	F	T	F	F	F	T	F	F	..	F	..	F	
3	6	T	F	F	T	F	F	T	T	F	T	T	F	
4	4	T	F	F	T	T	F	T	T	F	T	T	..	T	..	F
5	9	T	F	F	T	T	F	T	T	F	T	T	..	T	..	F
6	11	T	F	F	T	T	F	T	T	F	T	T	..	T	..	T

- At each cell reached, move vertically up to top-most ‘T’
- Then move diagonally according to how that ‘T’ entry arose
- The rows from which diagonal moves are made give the set **S**

Dynamic Programming Algorithm for Subset Sum

```
/** Input: array of ints x[1]...x[n] (N.B. x[0] unused)
 *          target positive integer k
 * Output: solution to SS(n,k) */

public boolean subsetSum(int k, int[] x) {

    int n = x.length - 1;
    boolean b[][] = new boolean[n+1][k+1];

    b[0][0] = true;
    for (int j = 1; j <= k; j++)
        b[0][j] = false;

    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= k; j++)
            b[i][j] = b[i-1][j] || (x[i]<=j && b[i-1][j-x[i]]);

    return b[n][k];
}
```

Complexity

- Size of DP table is $O(nk)$ ($n+1$ rows and $k+1$ columns)
- Each entry can be computed in $O(1)$ time
- So complexity of algorithm is $O(nk)$
- Why does this not prove that **SS** is in **P**?
 - and prove that **P = NP**, since **SS** is **NP-complete**?
- Answer – nk is not a polynomial function of the input size
 - $\log k$ bits suffice to represent k
 - $\leq n \log k$ bits suffice to represent the x 's
 - so size of the instance is at most $(n+1) \log k$
 - k could be polynomial in n , but need not be
 - for example, k could be 2^n , in which case input size is essentially n^2 , while complexity is no better than $O(n 2^n)$

Pseudo-Polynomial-Time Algorithms

- So what makes **SS** a hard problem is the fact that the numbers involved can be arbitrarily large
- If the numbers involved are bounded above in some way, **SS** is solvable in polynomial time
- Generally, let Π be a number problem and let **A** be an algorithm for Π
- We say that **A** is a *pseudo-polynomial-time algorithm* if, for any instance of x of Π , the time complexity of **A** is $O(q(|x|, \text{max}))$, where $|x|$ is the size of x , **max** is the largest number occurring in x and q is some polynomial function
- A decision problem is *strongly NP-complete* if it admits no pseudo-polynomial-time algorithm (unless $P = NP$)
 - e.g. TSP – remains NP-complete if all distances are 1 or 2

Knapsack Problem (KP) – Optimisation Version

Instance: n items, where item i has a weight w_i and a profit p_i , knapsack capacity c

Output: Maximum k such that there exists a subset S of $\{1, \dots, n\}$ for which $\sum_{r \in S} w_r \leq c$ and $\sum_{r \in S} p_r = k$

That is, choose a subset of the items of maximum total profit such that the knapsack capacity is not exceeded

Example: Knapsack capacity 65. 5 items:

Weights: $w_1=23, w_2=15, w_3=15, w_4=33, w_5=32$

Profits: $p_1=33, p_2=23, p_3=11, p_4=35, p_5=11$

Choosing all the items gives total weight 118

Choosing items 1,4 gives total weight $56 \leq c$ and total profit 68

Choosing items 2,3,4 gives total weight $63 \leq c$ and total profit 69 (optimal)

Trying a Greedy Algorithm for KP

- Calculate the profit/weight ratios of each of the **n** items
- Sort the items in order of these ratios (highest first)

```
/** Input: array of weights w[1]...w[n], array of profits
 * p[1]...p[n] (N.B. w[0] and p[0] unused)
 * p[i]/w[i] ≥ p[i+1]/w[i+1], 1 ≤ i ≤ n-1; capacity c
 * Output: a profit obtained by choosing a collection of
 * the items without exceeding the knapsack capacity */

public int greedyKP(int [] w, int [] p, int c)
{ int weight = 0; int profit = 0;
  int i = 1;      int n = w.length - 1;
  while ( weight < c && i ≤ n)
  { if ( w[i] + weight ≤ c )
    { weight += w[i]; profit += p[i]; }
    i++;
  }
  return profit;
}
```

The Greedy Algorithm does not work

- Example:

item	1	2	3
weight	10	20	30
profit	60	100	120
profit/weight	6	5	4

- Knapsack capacity **50**
- Greedy algorithm chooses items 1, 2
 - total weight **30** and total profit **160**
- Optimal solution comprises items 2, 3
 - total weight **50** and total profit **220**
- Decision version of **KP** is NP-complete – reduction from **SS**

A dynamic programming approach

- Parameterise: given an instance of KP, $KP(i, j)$ is a problem which asks for the maximum total profit over all subsets S of $\{1, \dots, i\}$ such that $\sum_{r \in S} w_r \leq j$ ($0 \leq i \leq n$ and $0 \leq j \leq c$)
 - let $S(i, j)$ be the solution to $KP(i, j)$
 - to solve KP for the original instance, find $S(n, c)$
- Dynamic programming algorithm is based on the following *recurrence relation* for KP:

$$S(i, j) = \begin{cases} S(i-1, j) & \text{if } w_i > j \\ \max (S(i-1, j), S(i-1, j - w_i) + p_i) & \text{otherwise} \end{cases}$$

for $i > 0$, subject to $S(0, j) = 0$ ($0 \leq j \leq c$)

- This implies an $O(nc)$ algorithm

Dynamic Programming Algorithm for KP

```
/** Input: array of weights w[1]...w[n], array of profits
 * p[1]...p[n] (N.B. w[0] and p[0] unused); capacity c
 * Output: solution to KP(n,c) */

public int dynamicKP(int [] w, int [] p, int c)
{ int n = w.length - 1;
int [][] s = new int[n+1][c+1];
for (int j=0; j<=c; j++)
    s[0][j] = 0;

for (int i=1; i<=n; i++)
    for (int j=0; j<=c; j++)
        if ( w[i] > j )
            s[i][j] = s[i-1][j];
        else
            s[i][j] = Math.max(s[i-1][j], s[i-1][j-w[i]]+p[i]);

return s[n][c];
}
```

An actual optimal choice of items

- Can be found by a traceback through the table
 - analogous to the case of the SS algorithm

Complexity

- Size of dynamic programming table is $O(nc)$
 - $n+1$ rows and $c+1$ columns
- Each entry can be computed in $O(1)$ time
- The overall complexity of the algorithm is $O(nc)$
- So this is a pseudo-polynomial time algorithm for KP

MY HOBBY: EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

CHOTCHKIES RESTAURANT	
~~ APPETIZERS ~~	
MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80
~~ SANDWICHES ~~	
BARBECUE	6.55

