A Polynomial-time Exact Algorithm for the Subset Sum Problem.

Andrea Bianchini, Electronic Engineer, http://www.es-andreabianchini.it

Abstract. Subset sum problem, (SSP), is an important problem in complexity theory, it belongs to complexity class NP-Hard, therefore to find a polynomial-time exact algorithm that solves subset sum problem proves that P=NP. In the present paper it will be shown a theorem that allows us to develop, as described in the paper, an algorithm of polynomial-time complexity. For a deepening on complexity theory and a proof about SSP complexity refer to : "Computers and Intractability: A guide to the theory of NP-completeness.", Michael R. Garey, David S. Johnson WH Freeman, 1979.

1.0 Definition of the problem.

Subset sum problem (SSP) can be defined as follow :
Given a set W of n positive integers and an integer c,

find

$$\max z=\sum x(i)w(i) \text{ ; } i=1,\dots,n \qquad 1.0$$

s.t. :

$$\sum x(i)w(i)\leq c;\ i=1,\dots,n \qquad 1.1$$

$$x(i)=0 \text{ or } 1;\ i=1,\dots,n \qquad 1.2$$

$$0<w(i)\leq c;\quad i=1,\dots,n \qquad 1.3$$

In the present paper it will be always assumed that W is sorted in ascending order, i.e., $w(i+1) \geq w(i)$, $i=1,\dots,n-1$.

1.1 Exploring solutions.

A trivial way to solve SSP is to enumerate all possible binary combinations of x and choose the optimal one, requiring, in the worst case 2^{n}, iterations.

The basic idea of the presented algorithm derive from the following question :

“does exist a way to explore all binary combination of x in a more efficient way ?”

the answer is : yes it do, and the complexity of this way is polynomial.

Let’s consider the following table that enumerates all binary combination of x for n=5 :

x	base	x	Base
00000	5	10000	5
00001	5	10001	5
00010	5	10010	5
00011	2	10011	2
00100	5	10100	5
00101	3	10101	3
00110	3	10110	3
00111	3	10111	3
01000	5	11000	5
01001	4	11001	5
01010	4	11010	5
01011	2	11011	2
01100	4	11100	5
01101	4	11101	5
01110	4	11110	5
01111	4	11111	5

Table 1.0

Definition 1.1.0 : *base* of a binary number.

The base of a binary number x is defined by following code :

```
int base(int x[], int n)
{
      int i;

      i=0;
      while(x[i]==0 && i<n)
            i++;
      // i is the position of first “1” bit

      i++;
      // “1” skipped

      while(x[i]==0 && i<n)
            i++;
      // all “0” skipped

      // i is the position of the second “1” bit
```

```
    while(x[i]==1 && i<n)
          i++;

    return i;
}
```

As you can see from table 1.0 and from code definition the base of a binary number x is the position of the at least second “1” bit whit successor “0” starting from less significant bit (rightmost bit).

We can obtain all binary numbers of base k starting from 0 adding “1” and shifting one by one until k then again adding “1” and shifting this last until k-1 and so on until all bit from 1 to k are “1”.

Definition 1.1.1 The base k of a binary number x is *pure* if x(i)=0 for all i>k.

Examples for n=5 :

x = 00011	base=2	pure.
x = 10011	base=2	not pure.
x = 00101	base=3	pure.
x = 10101	base=3	not pure.

Cardinality of set of all numbers in a given pure base pb can be easily computed as to be O(∑(pb-i),i=1,…,pb).

Definition 1.1.2 We denote with “x inc k” the increment of x by k positions in the same base of x, and similarly we denote with “x dec k” the decrement of x by k positions in the same base of x.

Examples for n=5 :

x	x inc 1	x dec 1
00001	00010	00000
00100	01000	00010
01100	01101	01010
01101	01110	01100

Proposition 1.1.0 Solutions z=x*w whit all x of the same base are monotone if W is monotone.

Proof. At each increment of x in the given base we add an item w[h] and eventually subtract an item w[k]≤w[h].

Proposition 1.1.1 Searching the maximum of z=x*w not exceeding c in all possible x of the same base can be performed in O(log(n)) time.

Proof. Binary search of a value in a sorted array of values.

1.2 Improving ideas.

Let be "xa" a feasible solution vector, (not necessarily optimal), of pure base "ba" and let be "a" the correspondent solution value, i.e., a=xa*w.

Theorem 1.2.0 if a≥a' for all possible a' with a and a' of any pure base b=2,…,n, let be ba the base of a, then do exist almost an optimal solution of value osa such that xosa<(xa inc 1) and xosa>2^{ba}-1, i.e., do exist an optimal solution vector xosa less than solution vector (xa inc 1) and greater than solution vector 2^{ba}-1. (the base of 2^{ba}-1 is ba'=ba-1, therefore the greater feasible solution of pure base ba' is, by definition, less than or equal to a)

Proof. If a=c proof is obvious. If a<c lets consider a capacity c=a+k, k>0. Let be xosa optimal solution vector obtainable under condition xosa<(xa inc 1) and xosa>2^{ba}-1, let be osa its solution value, i.e., osa=xosa*w, we can write osa=a+α, α≥0.
We can say that k≥α because osa=a+α≤c, but a=c-k, therefore, c-k+α≤c, therefore, k≥α.
Suppose that exists an optimal solution vector xos, xos>(xa inc 1) or xos≤2^{ba}-1, such that os>osa, we can write os=a'+α' and c=a'+k+z, z≥0, in fact a'≤a.
It can be proved that a+α≥a'+α', in fact a+α≤c=a'+k+z, therefore, a'+k+z≥a'+α', i.e., k+z≥α', that is always true.
It is important to notice that it is not excluded the presence of an optimal solution xos, xos>(xa inc 1) or xos≤2^{ba}-1, but simply, if such a solution exists then a solution of the same value do exists also for x<(xa inc 1) and x>2^{ba}-1.

Proposition 1.2.1 Finding a≥a' for all possible a' with a and a' of any pure base b=2,…,n, can be performed in O(n*log(n)) time.

Proof. It will be shown the algorithm maxABase of O(n*log(n)) complexity.

```
int maxABase(int[] w, int n, int c)
{
      int k,k1,lsb1,lsb2,lsbmax,lsbmin,i,amax,basemax,base,nitems;

      i=n-1;
      k=0;
      while(k<c && i>=0)
      {
            if (k+w[i]<=c)
            {
                  k+=w[i];
                  lsb1=i;
            }
            else
                  break;
            i--;
      }
      lsbmin=0;
      lsbmax=lsb1-1;
      lsb2=binarySearch(w,c-k,lsbmin,lsbmax);
      if (lsb2>-1)
```

```
            k+=w[lsb2];

      amax=k;
      basemax=n;
      base=n;

      while(base>1)
      {
            if (lsb2>-1)
                  k-=w[lsb2];
            i=base-1;
            k-=w[i];
            i=lsb1-1;
            while(k<c && i>=0)
            {
                  if (k+w[i]<=c)
                  {
                        k+=w[i];
                        lsb1=i;
                  }
                  else
                        break;
                  i--;
            }
            lsbmin=0;
            lsbmax=lsb1-1;
            lsb2=binarySearch(w,c-k,lsbmin,lsbmax);
            if (lsb2>-1)
                  k+=w[lsb2];

            if (k>amax)
            {
                  amax=k;
                  basemax=base-1;
            }
            base--;
      }

      return basemax;
}
```

binarySearch() is a function that searches for item $w(lsb2)=\max w(i) \leq c-k$, $i=lsbmin,\dots,lsbmax$, that can be executed in $O(\log(n))$ time.

Proposition 1.2.2 given a≥a' for all possible a' with a and a' of any pure base b=2,…,n, then finding optimal solution xosa, xosa<(xa inc 1) and xosa>2^{ba}-1, can be performed in O(n^3*log(n)) time.

Proof. Lets consider SSP', i.e., finding ∑x(i)w(i)≤c-a with items w(i),i<lsb(a), then SSP'', i.e., finding ∑x(i)w(i)≤c-(a-$2^{lsb(a)}$) with items w(i),i<lsb(a), then SSP''', i.e., finding ∑x(i)w(i)≤c-(a-$2^{lsb(a)}$-$2^{lsb'(a)}$) with items w(i),i<lsb'(a), then SSP'''', i.e., finding ∑x(i)w(i)≤c-(a-$2^{lsb(a)}$-$2^{lsb'(a)}$-$2^{lsb''(a)}$) with items w(i),i<lsb''(a), until finding ∑x(i)w(i)≤c-($2^{msb(a)}$ +$2^{msb'(a)}$) with items w(i),i<msb'(a)-1.
In reality it is not necessary to consider all significant bits in "a" but all of it while condition α'<a' is true. In fact if α'>a' it is also true that a+α>a'+α' because under such conditions we have a+α>2*a' that is also true. It can be done better considering instead of α'<a', θ*α'<a' with optimal θ calculated as : θ=(∏$\log_{10}$(n/10^i))/2+2*c/(n*(MAX+MIN)), i=0,..,$\log_{10}$(n)-1, MAX=w(n-1), MIN=w(0).
Because of number of significant bits in "a" are O(n) we need O(n)*O(n*log(n))*O(n)= O(n^3*log(n)) time to find the solution.

where :

lsb(a) = less significant bit of a.
lsb'(a) = second less significant bit of a.
lsb''(a) = third less significant bit of a.
msb(a) = most significant bit of a.
msb'(a) = second most significant bit of a.

Worst case time-complexity of algorithm.

We can summarize the steps needed to implement the whole algorithm for a worst-case total time-complexity evaluation of algorithm.

STEP	COMPLEXITY
1 – Sort of w in ascending order.	O(n*log(n))
2 – Search of max pure base "a".	O(n*log(n))
3 – Search of optimal solution x<(xa inc 1), x>(2 ^{ba})-1.	O(n^3*log(n))

Worst-case time complexity of algorithm : **O(n^3*log(n))**.
Expected time complexity of algorithm : **O(n*log(n))**.
Space consumption of algorithm : **O(n)**.

References.

[1] Silvano Martello, PaoloToth, 1990. Knapsack Problems Algorithms And Computer implementations.
[2] Hans Kellerer, Ulrich Pferschy, David Pisinger, 2004. Knapsack Problems.
[3] Michael R. Garey, David S. Johnson WH Freeman, 1979. Computers and Intractability: A guide to the theory of NP-completeness.