A polynomial-time exact algorithm for the Subset Sum problem

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

1.0 Definition of the problem.

Subset sum problem (SSP) can be defined as follow:
given a set $W$ of $n$ positive integers and a integer $c$, (capacity of the knapsack),

\[
\begin{align*}
\text{find} & \\
\max z &= \sum x(i)w(i) & 1.0 \\
\text{s.t.} & \\
\sum x(i)w(i) & \leq c & 1.1 \\
x(i) &= 0 \text{ or } 1; \ i = 1, \ldots, n & 1.2 \\
0 & < w(i) \leq c; \ i = 1, \ldots, n & 1.3
\end{align*}
\]

In the present paper it will be always assumed that $W$ is sorted in ascending order, i.e., $w(i+1) \geq w(i), \ i = 0, \ldots, n-2$.

Subset sum problem is a well known problem in operations research and it can be proved that it belongs to complexity class $NP$-Hard, therefore finding an algorithm that solves SSP in polynomial-time prove that $P=NP$. 
1.1 Exploring solutions.

A trivial way to solve SSP is to enumerate all possible binary combination for $x$ and chose 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 $n=5$:

<table>
<thead>
<tr>
<th>$x$</th>
<th>base</th>
<th>$x$</th>
<th>base</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>5</td>
<td>10000</td>
<td>5</td>
</tr>
<tr>
<td>00001</td>
<td>5</td>
<td>10001</td>
<td>5</td>
</tr>
<tr>
<td>00010</td>
<td>5</td>
<td>10010</td>
<td>5</td>
</tr>
<tr>
<td>00011</td>
<td>2</td>
<td>10011</td>
<td>2</td>
</tr>
<tr>
<td>00100</td>
<td>5</td>
<td>10100</td>
<td>5</td>
</tr>
<tr>
<td>00101</td>
<td>3</td>
<td>10101</td>
<td>3</td>
</tr>
<tr>
<td>00110</td>
<td>3</td>
<td>10110</td>
<td>3</td>
</tr>
<tr>
<td>00111</td>
<td>3</td>
<td>10111</td>
<td>3</td>
</tr>
<tr>
<td>01000</td>
<td>5</td>
<td>11000</td>
<td>5</td>
</tr>
<tr>
<td>01001</td>
<td>4</td>
<td>11001</td>
<td>5</td>
</tr>
<tr>
<td>01010</td>
<td>4</td>
<td>11010</td>
<td>5</td>
</tr>
<tr>
<td>01011</td>
<td>2</td>
<td>11011</td>
<td>2</td>
</tr>
<tr>
<td>01100</td>
<td>4</td>
<td>11100</td>
<td>5</td>
</tr>
<tr>
<td>01101</td>
<td>4</td>
<td>11101</td>
<td>5</td>
</tr>
<tr>
<td>01110</td>
<td>4</td>
<td>11110</td>
<td>5</td>
</tr>
<tr>
<td>01111</td>
<td>4</td>
<td>11111</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1.0
Definition 1.1.0 : base of a binary number

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

```c
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 $p_b$ can be easily computed as to be $O(\sum(p_b-i), i=1,\ldots,p_b)$.

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

Examples for $n=5$:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$x$ inc 1</th>
<th>$x$ dec 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001</td>
<td>00010</td>
<td>00000</td>
</tr>
<tr>
<td>00100</td>
<td>01000</td>
<td>00010</td>
</tr>
<tr>
<td>01100</td>
<td>01101</td>
<td>01010</td>
</tr>
<tr>
<td>01101</td>
<td>01110</td>
<td>01100</td>
</tr>
</tbody>
</table>

**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 $w[k]\leq w[h]$.

**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 general feasible solution vector of pure base “ba” and let be “a” the corresponding sum, i.e., \( a = xa \cdot w \).

**Proposition 1.2.0** if \( a \geq a' \) for all possible a’ with a and a’ of any pure base \( b = 2, \ldots, n \), let be ba the base of a, then there exist at least an optimal solution of value osa such that \( xosa \leq xa \) and \( xosa > (2^ba) - 1 \), i.e., there exists an optimal solution vector xosa less than or equal to solution vector xa and greater than \((2^ba) - 1 \). (base of \((2^ba) - 1 \) is ba’=ba-1, therefore grater feasible solution of pure base ba’ is, for definition, less than or equal to a)

**Proof.** If \( a = c \) proof is obvious. Let’s consider a capacity \( c = a + k , k > 0 \). Let be xosa the optimal solution vector obtainable under condition \( xosa \leq xa \) and \( xosa > (2^ba) - 1 \), let be osa it’s solution value, i.e., \( osa = xosa \cdot w \), we can write osa = a+\( \alpha \), \( \alpha \geq 0 \).

We can say that \( k \geq \alpha \) because osa = a+\( \alpha \leq c \), but a = c-k therefore c-k+\( \alpha \leq c \) therefore \( k \geq \alpha \).

Suppose that a solution vector xos > xa or xos \( \leq (2^ba) - 1 \) exists such that os > osa, then we can write os = a'+\( \alpha' \), osa = a+\( \alpha \), but a = c-k therefore a'+\( \alpha' \) \geq c-k+\( \alpha \) \geq k, therefore, \( a'+\alpha' > c \) that, for definition, is impossible.

It’s important to note that it’s not excluded the presence of an optimal solution xos > xa or xos \( \leq (2^ba) - 1 \), but simply if such solution exists then the same solution value do exists for x \( \leq xa \) and x > \((2^ba) - 1 \).
Proposition 1.2.1 Finding \( a \geq a' \) for all possible \( a' \) with a and \( a' \) of any pure base \( b=2,\ldots,n \), can be performed in \( O(n^2 \log(n)) \) time.

Proof. It will be shown the \( O(n^2 \log(n)) \) algorithm \( \text{maxA}
\text{Base} \).

```c
int \text{maxA}
\text{Base}(\text{int[]} \ w, \text{int} n, \text{int} c)
{
    \text{int } k,k1,\text{lsb}1,\text{lsb}2,\text{lsbmax},\text{lsbmin},i,\text{amax},\text{basemax},\text{base};
    \text{i}=n-1;
    k=0;
    \text{while}(k\text{<c }\&\& i\geq0)
    {
        \text{if } (k+w[i]\leq c)
        {
            k+=w[i];
            \text{lsb}1=i;
        }\text{else}
        {
            \text{break};
        }
        i--;
    }
    \text{lsbmin}=0;
    \text{lsbmax}=\text{lsb}1-1;
    \text{lsb}2=\text{binarySearch}(w,c-k,\text{lsbmin},\text{lsbmax});
    \text{if } \text{lsb}2>1
    {
        k+=w[\text{lsb}2];
    }
    \text{amax}=0;
    \text{basemax}=n;
    \text{base}=n;
    \text{while}(\text{base}>1)
    {
        \text{if } \text{lsb}2>1
        {
            k-=w[\text{lsb}2];
        }
        i=\text{base}-1;
        k-=w[i];
        \text{base}--;
        \text{i}=\text{lsb}1-1;
        \text{while}(k\text{<c }\&\& i\geq0)
        {
            \text{if } (k+w[i]\leq c)
            {
                k+=w[i];
                \text{lsb1}=i;
            }\text{else}
            {
                \text{break};
            }
            i--;
        }
    }
```
let lsbmin=0;
let lsbmax=lsb1-1;
let lsb2=binarySearch(w,c-k,lsbmin,lsbmax);
if (lsb2>l-1)
    k+=w[lsb2];
if (k>amax)
    { 
        amax=k;
        basemax=base;
    }
    base--;
return basemax;

binarySearch is a function that searches for an item \( w(\text{lsb}2)=\max w(i) \leq c-k \), \( i=\text{lsbmin}, \ldots,\text{lsbmax} \), which can be performed in \( O(\log(n)) \) time.

Proposition 1.2.2 given \( a \geq a' \) for all possible \( a' \) with \( a \) and \( a' \) of any pure base \( p_b=2, \ldots, n \), then finding optimal solution \( x_{osa} \leq x_a \) and \( x_{osa} > (2^{ba} - 1) \), can be performed in \( O(n^2 \times \log(n)) \) time.

Proof. We consider SSP', i.e., finding \( \sum_{i<\text{lsb}} x(i)w(i) \leq c-a \) with items \( w(i), i<\text{lsb} \) (less significant bit), then SSP'', i.e., finding \( \sum_{i<\text{lsb}} x(i)w(i) \leq c-(a \text{ dec } 1) \) with items \( w(i), i<\text{lsb} \), until finding \( \sum_{i<\text{lsb}} x(i)w(i) \leq c-(a \text{ dec } z) \) with items \( w(i), i<\text{lsb} \), where \( x_a \text{ dec } z \) is the first available binary number of base \( a \).
Cardinality of set of all numbers in a given pure base \( p_b \) can be easily computed as to be \( O(\sum(p_b-i), i=1, \ldots, p_b) \).
Worst-case time complexity of algorithm.

We can now summarize steps of algorithm for a global worst-case time complexity evaluation:

<table>
<thead>
<tr>
<th>STEP</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 – Sort of weights array w in ascending order.</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>2 – Search of max pure base a.</td>
<td>$O(n \log(n))$</td>
</tr>
<tr>
<td>3 – Search of optimal solution vector $x \leq x_a$, $x &gt; (2^b a) - 1$.</td>
<td>$O(n^2 \log(n))$</td>
</tr>
</tbody>
</table>

worst case time complexity of algorithm: $O(n^2 \log(n))$.
expected time complexity of algorithm: $O(n \log(n))$. 
References.