# A Simple 2 O □□ x □ Algorithm for PARTITION and SUBSET SUM



**2 authors**, including:

Scott Donald Kerlin
University of North Dakota
**27** PUBLICATIONS **101** CITATIONS

# A Simple $2^{O(\sqrt{x})}$ Algorithm for PARTITION and SUBSET SUM

**Thomas E. O'Neil and Scott Kerlin**
Computer Science Department
University of North Dakota
Grand Forks, ND, USA 58202-9015

**Abstract -** *The PARTITION problem has been determined by Stearns and Hunt to have power index one-half based on an algorithm with complexity $2^{O(\sqrt{x})}$, where x is the total length in bits of the input set. The algorithm that achieves this result is a combination of backtracking and dynamic programming. Here we present a simpler algorithm that has a similar complexity of $2^{O(\sqrt{x})}$ and can be applied to solve either PARTITION or SUBSET SUM. The new approach appears to be general enough to apply to many other NP-complete problems that have pseudo-polynomial time complexity.*

**Keywords**: PARTITION, SUBSET SUM, complexity, NP-complete, dynamic programming.

## 1 Introduction

The PARTITION problem is defined as follows: given a set $S$ of positive integers, determine whether $S$ can be partitioned into two subsets that have the same sum. The SUBSET SUM problem is similar, but more general: given a set $S$ of positive integers and a target integer $t$, determine whether some subset of $S$ has sum $t$. We can easily recast PARTITION as a special case of SUBSET SUM: given a set of positive integers $S = \{y_1, y_2, \ldots y_n\}$, determine whether $S$ has a subset whose sum is $(y_1 + y_2 + \ldots + y_n)/2$. Both PARTITION and SUBSET SUM are NP-complete [1, 3].

There is an obvious exponential-time exhaustive search algorithm for SUBSET SUM: successively generate all subsets and compute their sums. If $S$ has $n$ elements, it has $2^n$ subsets, and the exhaustive search algorithm is $O(p(n)\cdot 2^n)$ for some polynomial function $p(n)$. As with many other NP-complete problems, the search can be accomplished using a depth-first strategy with various bounding conditions to reduce the size of the search space. While their worst-case time complexity remains $O(p(n)\cdot 2^n)$, such backtracking algorithms frequently represent the most practical algorithms when both time and space are taken into account.

There is another standard approach to the problem that is not so obviously exponential: for each positive sum $k$ less than or equal to the target sum $t$, determine whether $S$ has a subset with sum $k$. This is usually implemented using dynamic programming on a Boolean array $A$ with index range from 0 to $t$, with $A[0]$ initially *true* and the rest of the array *false*. The problem is solved in $n$ passes over the array, one pass for each $x_i$ in $S$. During the $i^{th}$ pass, $A_i[j+x_i]$ is set to *true* for each $j$ where $A_{i-1}[j]$ is *true*. A subset with the target sum $t$ is discovered if $A[t]$ becomes *true*. The number of steps for this algorithm is proportional to $n\cdot t$. Any non-trivial target sum $t$ must be less than the sum of elements in $S$, so the length of the array need not exceed $n\cdot m$, where $m$ is the maximum number in $S$. We can also discard numbers in the set that are greater than $t$, and we can assume that $m \geq n$. The range of non-trivial $t$ values is thus $n < t < m^2$. The complexity of the algorithm depends on $m$ and how it relates to $n$. If $m$ is $O(n)$, the algorithm has time complexity $O(n^3)$, which is polynomial-time. But the hard instances of the problem have $m = O(2^n)$ [2, 7]. In such cases, the number of steps in the algorithm is $O(n\cdot(2^n)^2)$. Many NP-complete problems involving sets of weighted objects have a similar analysis. These are the so-called weak NP-complete (or pseudo-polynomial time) problems (see [3]).

A set of integers can be classified as sparse or dense based on the ratio $n$:$m$, where $n$ is the number of integers and $m$ is the maximum integer. When the ratio is high, we have a dense set, and dynamic programming is very effective. When the ratio is low, we have a sparse set for which backtracking is probably more efficient. Stearns and Hunt [7] designed an algorithm for the PARTITION problem that combines backtracking with dynamic programming. The input set is ordered and partitioned into a denser subset and a sparser subset. Backtracking is employed on the sparse subset, while dynamic programming is used for the dense subset. The results are combined to achieve time complexity $2^{O(\sqrt{x})}$, where $x$ is the total length in bits of the input set. In this paper we define a simpler algorithm that achieves a similar time complexity, and it can be used for both SUBSET SUM and PARTITION.

## 2 Dynamic Dynamic Programming

The *SumSearch* algorithm uses a dynamic programming approach in which the array of subproblem results is replaced by a dynamically allocated list – dynamic programming with dynamic allocation. We keep a list of target sums, initially containing only the original target. For each $y_i$ in a set of positive integers $S = \{y_1, y_2, \ldots y_n\}$, we add new targets to the list by subtracting $y_i$ from the existing targets. The step count for processing each $y_i$ is bounded by the current length of the target list. With standard dynamic programming, the list would have a fixed maximum size equal to the target value. When a dynamically allocated list of targets is employed, the list grows and shrinks. Initially of size one, the list doubles in size with each of the early iterations, reaching a peak in iteration $i$ when $2^i$ first exceeds the maximum value on the current target list. From that point on, the list shrinks in size. The new targets added to the list are always smaller than previous values, and large targets are pruned from the list when the sum of the remaining numbers in the set is not sufficient to reach them. We also suppress duplication of targets on the list, guaranteeing that the length of the list is no greater than the largest target value. As a result, the length of the target list in iteration $i$ is bounded by the minimum of $2^i$ and the maximum current target value. A detailed analysis is developed in Section 3 to establish that the time complexity is $2^{O(\sqrt{x})}$ where $x$ is the total length in bits of the set $S$.

Pseudo-code for the *SumSearch* algorithm is shown in Figure 1. Lines 1 through 6 check for special cases that can be dealt with easily. If the target exceeds the sum of the entire set, then the algorithm returns *false*. If the target is equal to the sum of the entire set or is equal to some element of the set, the algorithm returns *true*. And if the target is greater than half the sum of the entire set, the algorithm adjusts the target. The rationale for this adjustment is that it will be less work to find a smaller sum, and $S$ has a subset with sum *target* if and only if $S$ has a subset with sum $\sum_{i=1}^{n} y_i - target$. For the PARTITION problem, the target is always half the sum of the entire set. Thus, for both SUBSET SUM and PARTITION,

$$target \le (\sum_{i=1}^{n} y_i)/2. \qquad (1)$$

A list of target values $T$ is initialized in line 7 to contain only the original target. There is one iteration of the outer *for* loop (lines 8-18) for each number in the set $S$. The set $S$ is assumed to be in increasing order, and the largest numbers are processed first, so $y_{n-i+1}$ is processed during the $i^{th}$ iteration. The target list $T$ is updated by the inner for loop (lines 11-17). The value $y$ is subtracted from each target on the list, placing the results on a second ordered target list. The old and new target lists are merged in the last step of the outer loop (line 18). The pseudo-code for the *merge* method is not shown, but it is a standard linear-time sequential list merge in which duplicate values are suppressed. Line 12 in the inner loop checks for successful termination. If some target has reached zero, then a sequence of numbers has been discovered whose sum is equal to the original target.

The inner loop also has two conditions that moderate the length of the target list. Line 16 excludes negative values from the list. A negative target represents a subset whose sum is larger than the original target. Also, in line 14, targets too large to be reached by the remaining subsets are removed from the list. If a target exceeds the sum of all the remaining unprocessed numbers, it is not reachable. The information needed for this test is stored in an array of partial sums $\Sigma y[1..n]$ where $\Sigma y[i]$ is the sum of the first $i$ numbers in $S$. This array is computed as a linear-time preprocessing step, before *SumSearch* is called.

```
/* Given a set of natural numbers S = {y1, y2, ..., yn} in
increasing order, determine whether some subset of
S has sum = target.  Let Σy[i] represent the sum of the
first i numbers in S.
*/

public boolean SumSearch (int target)
    {
 1)     if (target > Σy[n])
 2)        return false;
 3)     else if target = Σy[n] or S contains target
 4)        return true;
 5)     else if target > Σy[n] / 2
 6)        return findSum(Σy[n] – target);

 7)     T ← {target}   // T  is a list of target values

 8)     for i ← 1 to n
 9)        y  ← S[n – i +1];
10)        NewList ←  { };
11)        for each t in T
12)              if (t – y ) = 0
13)                    return true;
14)              else if t > Σy[n –  i +1]
15)                    remove t from T;
16)              else if t – y > 0
17)                    append (t –  y) to NewList;
           end for
18)        T← merge(T, NewList);
       end for
19)    return false;
    }
```

Figure 1. The *SumSearch* algorithm.

## 3 Time Analysis of *SumSearch*

Let $S = \{y_1, y_2, ..., y_n\}$, and assume the numbers are stored in increasing order ($y_i < y_{i+1}$). The total number of steps is closely related to the size of the target list $T$. With each iteration of the outer for loop, the $T$ list is traversed and possibly extended (requiring 2 passes – one in the inner *for* loop and the other in the sequential merge step in line 18). The total amount of work is closely estimated (within a factor of 2) by $\sum_{i=1}^{n} |T(i)|$ where $|T(i)|$ is the length of list $T$ at the beginning of outer loop iteration $i$.

The list of targets will contain no duplicates, so we can describe its length as at most $T_{max}(i)$, the largest number in the list at the beginning of iteration $i$. The list could be smaller than this, since all the numbers between zero and the maximum may not be present.

$$|T(i)| \le T_{max}(i) \tag{2}$$

We also know that the length of the list can, at most, double with each loop iteration, so regardless of the maximum value in the list, its length cannot exceed $2^i$. This gives us

$$|T(i)| \le min(2^i, T_{max}(i)). \tag{3}$$

The length of the target list will grow rapidly and later possibly shrink as $i$ approaches $n$. Our goal is to find an upper bound for $T_{max}(i)$. Initially $T_{max}(1) = target$, which is no more than half the sum of $S$. Only smaller numbers are added to the list, and eventually the larger numbers are removed when the condition in line 14 becomes true, enforcing

$$T_{max}(i) \le \sum_{j=1}^{n-i+1} y_j. \tag{4}$$

Bounding $T_{max}(i)$ thus reduces to finding an upper bound for $y_k$, where $k = n - i + 1$. Given $y_k$, we can bound the sum as

$$\sum_{j=1}^{k} y_j \le k \cdot y_k. \tag{5}$$

Assume that the total number of bits required to represent the set $S$ is $x$.

$$x = bitlength(S) = \sum_{i=1}^{n} bitlength(y_i). \tag{6}$$

The highest possible value for $y_k$ is obtained by reserving as few bits as possible for the smaller numbers in the set and as many bits as possible for $y_k$ and the larger numbers. This is accomplished by setting $y_{k-1} = k-1$ and distributing the remaining bits equally among the higher $n-k+1$ numbers. The remaining bits are distributed equally because no smaller number can have more bits than a larger number, and we want the smallest number ($y_k$) to have as many bits as possible. Then

$$\begin{aligned} m_k &= bitlength(y_k) \\ &\le (x - bitlength(\{1, \ldots, k-1\}))/(n-k+1). \end{aligned} \tag{7}$$

At this point we are ready to assemble the pieces and complete the analysis. We have

$$\sum_{i=1}^{n} |T(i)| \le \sum_{i=1}^{n} min(2^i, T_{max}(i)). \tag{8}$$

We consider the total bit length $x$ of the input set $S$, and examine the step counts for two cases where the number of integers in the set $S$ is less than or greater than the square root of its total bit length. For each case we bound either $2^i$ or $T_{max}(i)$, whichever is more expedient (keeping in mind that $min(a, b) \le a$ and $min(a, b) \le b$).

Case 1. $n \le \sqrt{x}$. Here we have

$$\sum_{i=1}^{n} |T(i)| \le \sum_{i=1}^{n} min(2^i, T_{max}(i)) \le n \cdot 2^n \le n \cdot 2^{\sqrt{(x)}}. \tag{9}$$

Case 2. $n > \sqrt{x}$. In this case we split the summation at $i = \sqrt{x}$.

$$\begin{aligned} \sum_{i=1}^{n} |T(i)| &\le \sum_{i=1}^{n} min(2^i, T_{max}(i)) \\ &\le \sum_{i=1}^{\sqrt{x}-1} min(2^i, T_{max}(i)) + \sum_{i=\sqrt{x}}^{n} min(2^i, T_{max}(i)) \\ &\le (\sqrt{x}-1) \cdot 2^{\sqrt{x}-1} + \sum_{i=\sqrt{x}}^{n} min(2^i, T_{max}(i)) \end{aligned} \tag{10}$$

Recall here that $T_{max}(i)$ decreases as $i$ increases.

$$\begin{aligned} &\le (\sqrt{x}-1) \cdot 2^{\sqrt{x}-1} + (n-\sqrt{x}+1) \cdot T_{max}(\sqrt{x}) \\ &\le (\sqrt{x}-1) \cdot 2^{\sqrt{x}-1} + (n-\sqrt{x}+1) \cdot \sum_{j=1}^{n-\sqrt{x}+1} y_j \\ &\le (\sqrt{x}-1) \cdot 2^{\sqrt{x}-1} + (n-\sqrt{x}+1)^2 \cdot y_{n-\sqrt{x}+1} \end{aligned} \tag{11}$$

At this point, we can compute the bound for $m_k$, where $k = n-\sqrt{x}+1$ and replace $y_{n-\sqrt{x}+1}$ with $2^{m_k}$. Applying formula (7) above,

$$m_{n-\sqrt{x}+1} \le (x - bitlength(\{1, .., n-\sqrt{x}\}))/\sqrt{x} < x/\sqrt{x} = \sqrt{x}.$$

So we have $m_{n-\sqrt{x}+1} < \sqrt{x}$ and we continue from formula (11) above by replacing $y_{n-\sqrt{x}+1}$ with $2^{\sqrt{x}}$ :

$$\begin{aligned} &< (\sqrt{x}-1) \cdot 2^{\sqrt{x}-1} + (n-\sqrt{x}+1)^2 \cdot 2^{\sqrt{x}} \\ &< (\sqrt{x}) \cdot 2^{\sqrt{x}} + n^2 \cdot 2^{\sqrt{x}} \\ &< 2\, n^2 \cdot 2^{\sqrt{x}}. \end{aligned} \tag{12}$$

This establishes that the time complexity of *SumSearch* is $O(p(n)2^{\sqrt{x}})$ for a polynomial function $p(n)$. And since $n \le x$, we have $p(n) \le p(x)$. So $O(p(n)2^{\sqrt{x}})$ can be simplified to $2^{O(\sqrt{x})}$.

# 4 Conclusion

The *SumSearch* algorithm demonstrates that dynamic programming with dynamic allocation can be used to prove that both SUBSET SUM and PARTITION have time complexity $2^{O(\sqrt{x})}$ where $x$ is the total bit length of $n$ input numbers. This result is similar to the complexity for PARTITION in established in [7], and it is achieved by a simpler algorithm. The approach appears to be simple and general enough to be applied to other NP-complete problems that have pseudo-polynomial time complexity, such as KNAPSACK and BIN PACKING with a fixed number of bins. This possibility, combined with the proposition that CLIQUE is more appropriately classified as $O(2^n)$ than $2^{O(\sqrt{x})}$ (see [4]), leads to the conjecture that the class of "easier hard problems" defined by Stearns and Hunt [6,7] actually corresponds to the class of pseudo-polynomial-time NP-complete problems.

# 5 References


[1] M. Garey and D. Johnson. C*omputers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman Press, San Francisco, CA, 1979.

[2] B. Hayes. "The Easiest Hard Problem". *American Scientist* 90(2), p. 116, March-April 2002.

[3] R. Karp. "Reducibility Among Combinatorial Problems". In *Complexity and Computer Computations*, ed. R. E. Miller and J. W. Thatcher, pp. 85-103, Plenum Press, New York, 1972.

[4] T. E. O'Neil. "The Importance of Symmetric Representation," *Proceedings of the 2009 International Conference on Foundations of Computer Science (FCS '09)*. CSREA Press, pp. 115-119, Las Vegas, Nevada, July 16, 2009.

[5] T. E. O'Neil. "Dynamic Dynamic Programming for the 0/1 KNAPSACK Problem," submitted for publication, March 2010.

[6] R. Stearns. "It's Time to Reconsider Time". *Communications of the ACM*, Vol. 37, No. 11, pp. 95-99, 1994.

[7] R. Stearns and H. Hunt. "Power Indices and Easier Hard Problems". *Mathematical Systems Theory* 23, pp. 209-225, 1990.