Algorithm Design

Carefully designed algorithms may lead to dramatic performance improvements. This is illustrated through an example.

Problem : Given an array of real numbers, find the maximum contiguous non-negative sum in the array (instance of 1-dimensional pattern recognition)
History : Original problem occured in 2-dimensional arrays or matrices. Maximum non-negative sum sub-array of a matrix is a maximum likelihood estimator for a certain kind of pattern in a digitised picture.
Source : John Bentley, Programming Pearls, Addison Wesley, 2000

Input : One dimensional array of floats having positive and negative reals. For example, consider the following values :

a[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
13	-21	25	16	-40	37	44	95	-90	-10	88

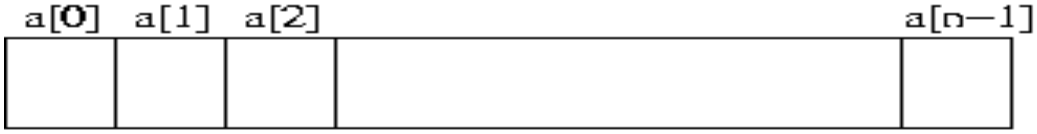
Output : Maximum contiguous non-negative sum in the array.
For the example above, the desired output is 177 which is the sum of array elements a[2] through to a[7].

Special Cases : If all array values are positive, the solution is trivial, it is the sum of the entire array.
If all array values are negative, an empty array is the maximum non-negative sub subarray, the sum is therefore 0.0.

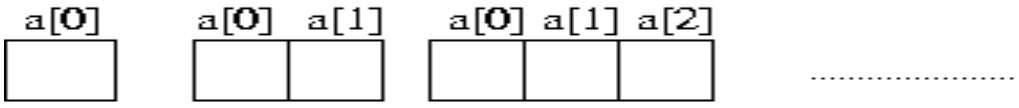
Algorithm 1

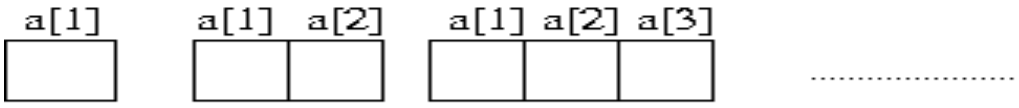
This is a naive but exhaustive and provably correct solution. The strategy is to examine all possible subarrays of the input array and
find the maximum among them.

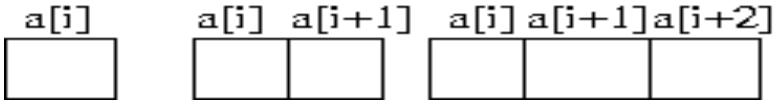
Let the original array, a[], be as shown below :



A few possible subarrays of a[] are shown below :







The program, given below uses this strategy. Since all possible sub-arrays are exhaustively considered, correctness is assured.

```
 float max ( float x, float y)  { if ( x <= y ) return y; else return x; }

float max_pos_sum( float b[], int size)
{ float maxsofar = 0.0;
  for (int i = 0; i < size; i++)
   for ( int j = i; j < size; j++)
     { float sum = 0.0;
       for ( int k = i; k <=j; k++) sum = sum + b[k];
       maxsofar = max(maxsofar, sum);
     };
 return maxsofar;
}
```

And a main() to test the implementation of max_pos_sum() follows.

```
int main()
{
  float a[SIZE];
  int num;
  cout << " give the number of elements ";
  cin >> num; cout << endl<< " give elements : " ;
  for ( int j = 0; j < 100; j++)
  {  for ( int i = 0; i < num; i++ )  cin >> a[i]; cout << endl;
     cout << " max +ve sum in array a[] = " << max_pos_sum(a, num) << endl;
  };
}
```

The time requirement of this algorithm turns out to be of the form : $c_1*n^3+c_2*n^2+c_3*n+c_4$

Algorithm 2

Algorithm 1 uses the innermost k loop to compute sums such as

sum = (a[i] + a[i+1] + ... a[j]) and

sum = (a[i] + a[i+1] + ... a[j+1])**.**

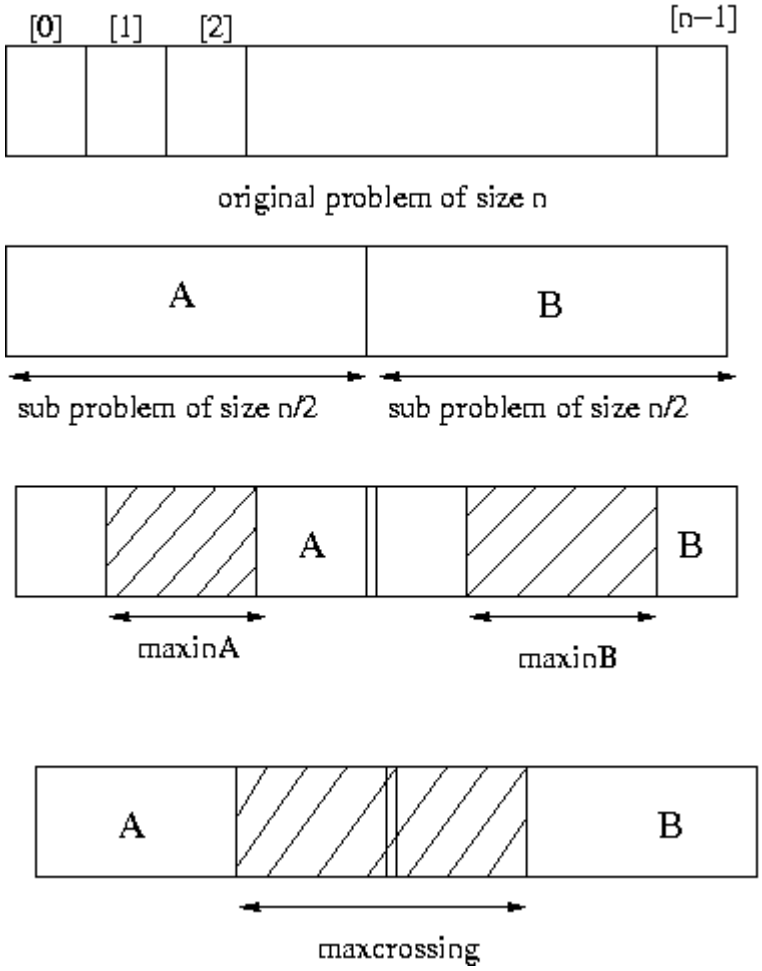
The second sum, however, need not be computed afresh from scratch but can be incrementally computed from the previous sum. Incorporating this observation, leads to an improved program..

The time requirement of this algorithm turns out to be of the form : $c_1*n^2+c_2*n+c_3$

Algorithm 3

This algorithm uses the "divide and conquer" method of designing solutions. The given problem of size n is subdivided into 2 problems, each of size n/2. The following observation is the key to the design.

There are three possibilities for the maximum non-negative sum to occur as shown in the following figure



The required max sum is therefore =
max (maxinA, maxinB, maxcrossing)

We can write another verion of max_pos_sum() that uses this strategy, loosely specified below.

```
float max_pos_sum( float b[], int size, int low, int high)
{ float maxsum = max2left = max2right =  maxcrossing = maxinA= maxinB=0.0;
  if ( low > high) return suitable value; // no recursion
  if ( low == high ) return suitable value;  // no recursion
  int mid = (low + high)/2;
  find max2left in the array b[mid] to b[low];
  find max2right in the array b[mid+1] to b[high];
  maxcrossing = max2left + max2right;
  maxinA = max_pos_sum(b, size, low, mid);
  maxinB = max_pos_sum(b, size, mid+1, high);
  return max(maxinA, maxinB, maxcrossing);
}
```

The time requirement of such algorithms are usually expressed in the form of a recurrence relation, given below, where T(n) reqpresents the time required by an algorithm to solve a problem with input size n.

$T(n) = 2T(n/2) + c * n; T(1) = 1$

As we shall learn later, the time requirement for this algorithm turns out to be of the form :
$c_1 * n\log_2 n + c_2 * n + c_3$

Can we do better ?

Starting from an exhaustive algorithm of time requirement proportional to n^3, we have been able to find better algorithms whose time requirements are proportional to n^2 and $n\log_2 n$.

An obvious question is : Is it possible to find an even faster algorithm ?

The answer is YES, there exists a linear time algorithm as function of input size n, for this problem, say $c_1 * n + c_2$

Finding this algorithm is left as an exercise. Note than any algorithm that solves this problem in linear time is an optimal algorithm since any correct solution has to examine each value of input at least once.