We denote by $S(i)$ and $T(i)$ left and right partial sums of the array:

$$S(i) = \sum_{k=1}^{i} a_k, \qquad T(j) = \sum_{k=1}^{j} a_{n+1-k}.$$

We also denote by $i_1 < i_2 < \ldots$ all indexes corresponding to positive left sums: $S(i_\alpha) > 0$, and similarly $j_1 < j_2 < \ldots$, $T(j_\beta) > 0$.

The main idea of algorithm is directly check all three cases of positive parts.

Proposition 1. Case $(+; *; +)$ takes place iff $i_1 + j_1 < n$.
Proof. Left plus corresponds to partial sum $S(i_\alpha)$ for some α. Right plus corresponds to $T(j_\beta)$ for some beta. So

$$i_1 + j_1 \leq i_\alpha + j_\beta < n.$$

First part of algorithm:

1. Find i_1 and j_1 by calculating left and right partial sums,
2. If both exist and $i_1 + j_1 < n$, then return i_1 and $n - j_1$ as indexes of cuts.

Proposition 2. Case $(+; +; *)$ takes plase iff

$$S(i_\alpha) < S(i_{\alpha+1}) \qquad (*)$$

for some α.
Proof. If the inequality $(*)$ holds, then evidently we can make cuts on indexes $i_\alpha,\ i_{\alpha+1}$. And conversely, if cuts exist, they are made on some indexes $i_p < i_q$, so $S(i_p) < S(i_q)$, and $(*)$ holds for some α between p and q.

Second part of algorithm:

3. Remember current values of index and partial sum,
4. Continue calculation of left partial sums $S(i)$ and find next positive,
5. If it is greater than previous - return both indexes,
6. Repeat steps 3-5 till the end of array.

Optionally this part can be written in code:

$x = y = i_1; X = Y = S(i_1);$
while(1):
  do{ $y++;\ Y += a_y;$ } while($Y <= 0$ *and* $y < n$);
  if($y == n$) break;
  if($X < Y$) return $x, y;$
  $X = Y; x = y;$

If we achieved end of array, then the third case $(*, +, +)$ takes place. Repeat steps 3-6 for right partial sums $T(i)$.

Complexity is O(n): we have twice passed along the whole array, and at each step number of calculations was constant.