1. a. $a = 2, b = 2, \log_b a = 1, f(n) = \log n$. If $\epsilon = 1/2$, then $\log n$ is $O(n^{\log_b a - \epsilon})$.
   Case 1: $T(n) = \Theta(n)$.

   b. $a = 8, b = 2, \log_b a = 3, f(n) = n^2$. If $\epsilon = 1/2$, then $n^2$ is $O(n^{\log_b a - \epsilon})$.
   Case 1: $T(n) = \Theta(n^3)$.

   c. $a = 16, b = 2, \log_b a = 4, f(n) = n^4 \log^4 n$. We have $n^4 \log^4 n$ is $\Theta(n^4 \log^4 n)$.
   Case 2: $T(n) = \Theta(n^4 \log^5 n)$.

   d. $a = 7, b = 3, \log_b a \approx 1.77, f(n) = n$. If $\epsilon = \log_b a - 1$, then $n$ is $O(n^{\log_b a - \epsilon})$.
   Case 1: $T(n) = \Theta(n^{\log_3 7})$.

   e. $a = 9, b = 3, \log_b a = 2, f(n) = n^3 \log n$. If $\epsilon = 1/2$, then $n^3 \log n$ is $\Omega(n^{\log_b a + \epsilon})$.
   Case 3: $T(n) = \Theta(n^3 \log n)$.

2. Spend $O(n \log n)$ to sort the points and place them in a list allowing $O(1)$ time removal.
   Place a guard at $x_0 + 1$, and remove all pantings that are now guarded. Recurse on the list of unguarded pantings, using the same strategy, until the list is empty.
3. In the base case a single interval \([a, b]\) at height \(h\) is returned as the upper envelope \(H : (a, h), (b, 0)\). From there we consider the problem of merging two upper envelopes. The merge algorithm is given in Python below.

```python
def merge(H1, H2):
    H = []
    height1, height2 = 0, 0
    active = 1
    while H1 and H2:
        if H1[0][0] < H2[0][0]:
            current = H1.pop(0)
            height1 = current[1]
            if height1 > height2:
                H.append(current)
                active = 1
            else:
                if active == 1:
                    H.append((current[0], height2))
                else:
                    current = H2.pop(0)
                    height2 = current[1]
                    if height2 > height1:
                        H.append(current)
                        active = 2
                    else:
                        if active == 2:
                            H.append((current[0], height1))

    if H1:
        H += H1
    if H2:
        H += H2
```

The merge function above runs in linear time. So assuming we divide the problem roughly in half we will get the same runtime as merge sort, i.e., \(O(n \log n)\).

4. In this problem we are tasked with determining if a substring of \(A\) sums to \(N/2\). Consider the subproblems \(P_{i,j}\) of determining if \(A_i = \{a_1, a_2, \ldots, a_i\}\) has a subsequence summing to \(j\). The original problem is then to determine if \(P_{n,N/2}\) is true. The subproblems are related via the following recursion \(P_{i,j} = P_{i-1,j} \text{ or } P_{i-1,j-a_i}\), i.e., at each step in the recursion we either use \(a_i\) in the subsequence or we do not. We have the base cases \(P_{i,0} = \text{true}\) for all \(i\), and \(P_{1,j}\) is \(\text{true}\) when \(j = a_1\) and \(\text{false}\) otherwise. The table \(P\) has size \(n \times N/2\) and filling out each cell takes \(O(1)\) work so the runtime is \(O(nN)\).
5. For this problem consider the subproblems \( P_i \) of determining the length of the last word in a valid break up of the first \( i \) characters into words. I want the length of the last words to make it easier to insert spaces after I have found the valid break up. The recursion is then \( P_i = k \) where \( P_{i-k} \) is nonzero (meaning it has a valid break up into words) or \( i = k \) and the last \( k \) letter of \( P_i \) form a word, assuming such a \( k \) exists. If no such \( k \) exists, then \( P_i = 0 \). The base case is handle by the \( i = k \) case. The array \( P \) has size \( n \) where \( n \) is the number of characters in the array, and filling out each cell takes at most \( O(n) \) time. Thus we have a \( O(n^2) \) algorithm.

For fun I programmed this problem in Python. To use it you will need a word list for the spell checker. I used the list at http://www.sil.org/linguistics/wordlists/english/.

```
WORDS = set()

def build_words():
    global WORDS
    with open('wordsEn.txt') as f:
        for w in f:
            WORDS.add(w.strip())

def spy_split(s):
    n = len(s)
    P = [0] * (n + 1)
    for i in xrange(len(s)):
        if i == 0 or P[i]:
            for j in xrange(i, n + 1):
                if s[i:j] in WORDS:
                    P[j] = j - i
    ss = []
    while s:
        w = P[len(s)]
        ss = [s[-w:]] + ss
        s = s[:-w]
    print ss
```