Problem 1.

You are given points $p_0, p_1, \ldots, p_n$, and $p_i$ has coordinates $x[i], y[i]$. Assume that $x[0] = y[0] = 0$.

a. Assume that all points other than point 0 are in quadrant I ($x[i] > 0$ and $y[i] > 0$) or in quadrant III ($x[i] < 0$ and $y[i] < 0$). Describe how to test if $p_0$ is extreme in this set using $O(n)$ steps.

Answer. We find $M_I$, the maximum slope of a line connecting $p_0$ with a point in the first quadrant, and also $m_I$, the minimum such slope. We do the same for quadrant III. If the intervals $[m_I, M_I]$ and $[m_{III}, M_{III}]$ are not disjoint, $p_0$ is not extreme: there is some slope $s$ in both intervals, which means that there are points of the convex hull in both quadrants such the $p_0$ is on a line that connects them. Conversely, if those intervals are disjoint, $p_0$ is extreme. Suppose that every slope from quadrant III is smaller then every slope to quadrant I, then every line connecting points in quadrant III and I will cross Y axis above $p_0$ (the other case is symmetric.

As an unnecessary part of the answer, we can see the detailed pseudocode:

```plaintext
mi = Mi = mIII = MIII = -1;
for (i = 1; i <= n; i++) {
    assert(x[i] != 0);
    slope = y[i]/x[i];
    if (x[i] > 0) {
        assert(y[i] > 0);
        if (mi < 0 || mi > slope)
            mi = slope;
        if (mi < 0 || Mi < slope)
            Mi = slope;
    }
    else {
        // x[i] < 0
        assert(y[i] < 0);
        if (mIII < 0 || mIII > slope)
            mIII = slope;
        if (mIII < 0 || MIII < slope)
            MIII = slope;
    }
    if (mi > MIII || Mi < mIII)
        conclude "extreme";
else
    assert(x[i] != 0);
    conclude "not extreme";
```

b. Modify your test so it still uses $O(n)$ steps, but it does a bit more: if $p_0$ is extreme, it finds neighbors of $p_0$ on the polyline of the convex hull of our point set (see Figure 3.5 on page 110, where $p_1$ has neighbors: $p_3$ and $p_6$.)

**Answer.** If $p_0$ is extreme because $m_I > M_{III}$ then the neighbors of $p_0$ on the convex hull are: a point in quadrant I with slope $m_I$, and a point in quadrant III with slope $M_{III}$. If there is more than one point with such a slope we choose one which is the furthest from $p_0$, so it has the largest $|x[i]|$.

One can modify the search for the largest and smallest slope to record the candidates for neighbors of $p_0$ any time the “best slope” is either improved or we are looking at a point with the best slope, but further away than the one we recorded before.

**Problem 2.**

You have array of numbers $A[0], \ldots, A[n-1]$, and target number $t$. Check if for some $i < n/2$ and $j \geq n/2$ we have $A[i] + A[j] = t$.

a. Describe a reasonable brute force algorithm. What is the running time?

```java
for (i = 0; i < n/2+n&1; i++)
    for (j = n/2+n&1; j < n; j++)
            report i and j;
            break;
        }
```

The running time is $\Theta(n^2)$.

b. Using sorting and binary search, describe how to find the answer in time $O(n \log n)$.

**Answer.** We start with sorting the left and right halves of array $A$, in sorting we place smallest element at the beginning, the largest element at the end. Then we proceed as follows:

```java
i = n/2+n&1-1;
j = i+1;
while (i >= 0 && j < n)
        i--;
        j++;
    else
        break;
// either A[i]+A[j] == t or it is not possible
```

We start with the largest number on the left and smallest on the right. If the sum is too large, the largest on the left cannot be matched with a number from the right to add to the target: even the smallest of such sums is too large; thus we will never have to look at this number again. A symmetric reasoning applies when the sum is too small. Thus we either can decrement $i$ (to never look at $A[i]$ again) or we can increment $j$, or we find the target.
Problem 6 on page 119.

I do not know how one can reduce the number of sets to consider in brute force approach. Instead, explain how problem 2 can be used to reduce the time needed to $O(2^{n/2})$ or $O(n^{2^{n/2}})$.

**Answer.** We can inspect every subset and see if it adds to half of the total. Perhaps we can try to be a bit smart. First problem, how to inspect every set in an orderly manner, and avoid unnecessary additions. We compute $\text{sum}$ of the array entries, quit if $\text{sum}$ is odd, and otherwise we invoke recursive call $\text{SubsetSum}(A, n, \text{sum}/2)$:

```c
int SubsetSum(int *A, int m, int t)
{   if (target == 0)
         return 1;
    else if (target < 0 || m == 0)
         return 0;
    else
         return SubsetSum(A, m-1, t) || SubsetSum(A, m-1, t-A[m-1]);
}
```

We are making two calls, one assumes that the last entry in the array is not used, and the other assumes it is used (so we decrease the target). There is some avoidance of unnecessary calls: if our assumptions “we use this entry” add to more than the original target, we make a call with a negative target and this stops the process. “Lazy or” also helps.

The alternative approach is to fill two halves of an array with subset sums of $A$: sums of subsets of the left half of $A$ would go to the left half of $B$ and the same with the right halves. Now we apply the algorithm of Problem 2.
Problem 7 on page 119.

Additionally, explain why the clique problem has an algorithm that runs in time \(O(2^{n/2})\).

**Answer.** We can inspect every subset and see if it is a clique. Again, we can do it recursively and try to be a bit smart. We need a representation of the graph, say, nodes are numbers from 0 to \(n-1\) and \(A[i][j]\) tells if \(\{i,j\}\) is an undirected edge. We also need an array that stores our assumptions: what is in the set we are constructing and what is not, we will use \(Clique[n]\). We apply the recursive call \(FindClique(A,n,k)\).

```c
int FindClique(int *A, int m, int k)
{
    int j;
    if (k == 0)
        return 1;
    if (k > m)
        return 0;
    B[m-1] = 0;
    if (FindClique(A,m-1,k))
        return 1;
    // check if we can consider A[m-1] as a clique member
    for (j = m; m < n; m++)
        if (B[j] && !A[m-1][j])
            break;
    if (j < m)
        return 0;
    B[m-1] = 1;
    return FindClique(A,m-1,k-1);
}
```

It is easy to see that this recurrence runs in time \(O(n2^n)\) because each call makes linear amount of work of its own, and makes one or two calls with size smaller by 1.

If we replace edges with non-edges, i.e. we replace each \(A[i][j]\) with \(!A[i][j]\), then cliques become independent sets and we can apply an algorithm for finding a maximum size independent set. We learned in class that there exists and algorithm that runs in time \(O(2^{0.41n})\), and \(2^{0.41n} \in O(2^{0.5n})\).