1. (1) **Solution:** Definition an array **B** of size n*(n-1)/2, let 0<=**k**<n, k+1<=**m**<=n (m and k are integer). Go through all pairs (A[k], A[m]) and let **temp** = A[k]^2+A[m]^2. Check if an element in B is equals to temp by binary search. If searching successfully, return true; else, put temp into B in order, then k++, m++ and loop this step. When all pairs were visited and the loop ended, return false.

Pseudocode:

```
function solve(A, n)
   B = new int[n*(n-1)/2]
   k = 0
   m = k + 1
   for k in [0, n)
         for m in [k + 1, n]
               temp = A[k]^2+A[m]^2
               if binary_search(B, temp) == true
                     return true;
               else
                     insert_ordered(B, temp)
                     m++
               end if;
         end for;
            k++
   end for;
   return false;
end
```

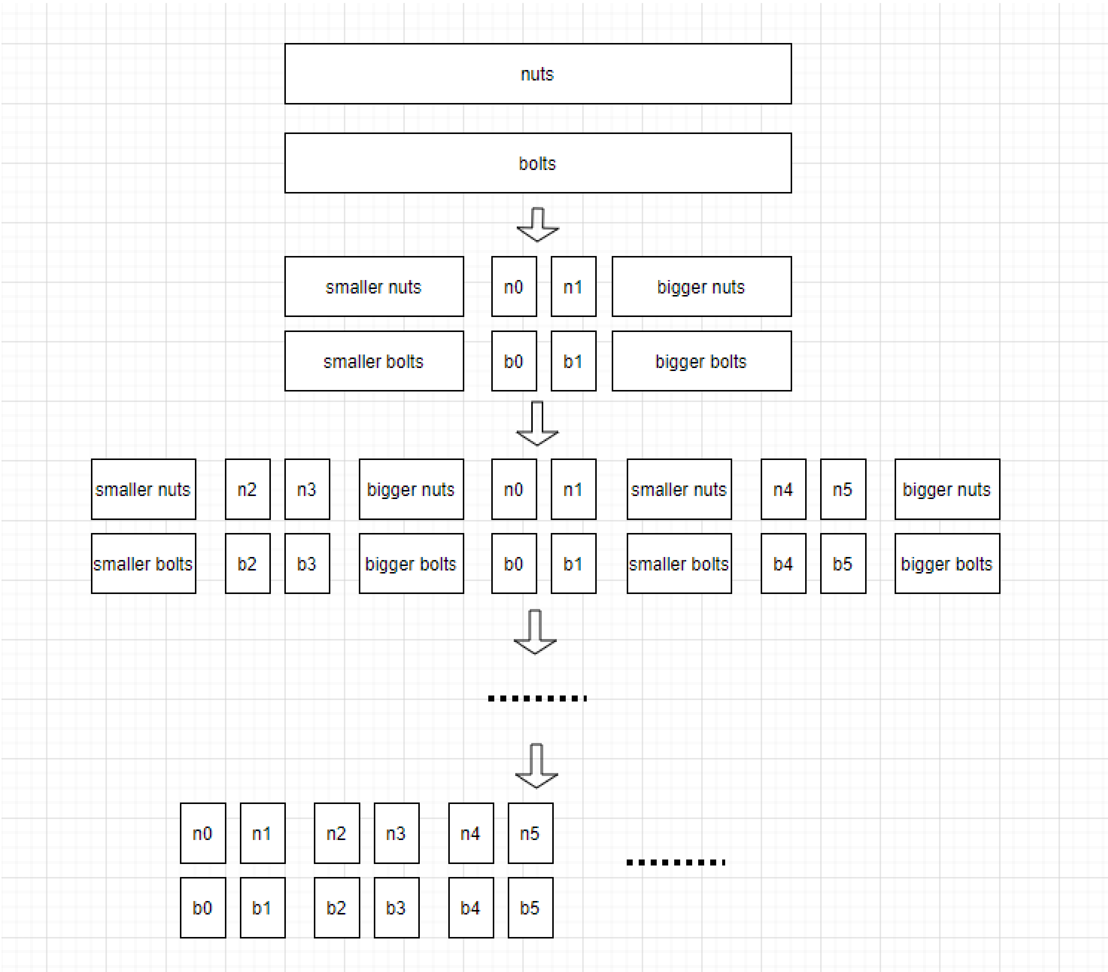
(2) **Solution: Sort** the array A. Define an array **B** of size **A[n-2]^2 + A[n-1]^2**. let 0<=**k**<n, k+1<=**m**<=n (m and k are integer). Go through all pairs (A[k], A[m]) and check if **B[A[k]^2+A[m]^2]**>0, if true, end the program; else, B[A[k]^2+A[m]^2]++ and go next loop. When all pairs were visited and the loop ended, return false.

Pseudocode:

```
function solve(A, n)
   sort_asc(A)
   B = new int[A[n-2]^2 + A[n-1]^2]
   k = 0
   m = k + 1
   for k in [0, n)
         for m in [k + 1, n]
               if B[A[k]^2+A[m]^2]>0
                     return true;
               else
                     B[A[k]^2+A[m]^2]++
                     m++
```

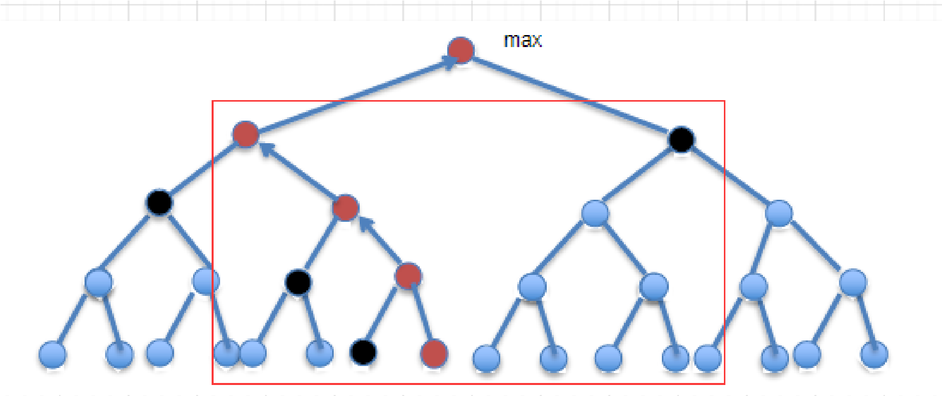
```
                end if;
        end for;
            k++
    end for;
    return false;
end
```

2. **Solution:** Define two arrays nuts and bolts. Take one in the nuts array, we can divide the bolts array into 3 parts: smaller than that, bigger than that, and fitting with that. Now we get two pairs of nut and bolt, smaller nuts and smaller bolts, and bigger nuts and bigger bolts. Repeat the progress for the smaller arrays and the bigger arrays separately. Finally, we'll make each bolt with a nut of a fitting size.



3. **Solution:** We use the idea of the championship algorithm: **only those who have played against the champion are likely to be runners-up**. Let n = 10. Use the divide and conquer method to divide the array into 2 groups, and use the method of comparing and eliminating each other to find the maximum value in each group (2^n-1 comparisons in total). Compare the last two maximum values, and the larger one is the maximum value. The second largest value is in the smaller one and all the elements(n-1 in total) compared

with the maximum value. We only need to compare the smaller one and the n-1 elements that have been compared with the maximum value (here we need n-1 comparisons). So the number of comparisons is 2^n + n -1 = 1032.

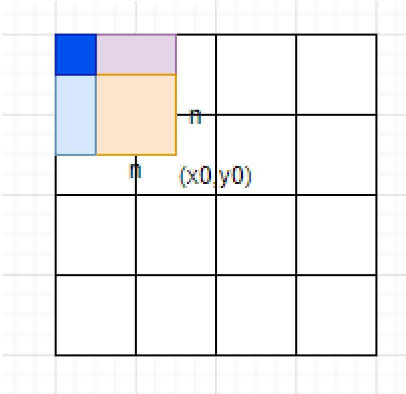


The secondary values are in the red nodes other than the maximum and the black nodes at the top.

4. Solution: Define a 2D array A, let A[i, j] represent the total of apples from (0, 0) to (i, j), 0<=i<=4n, 0<=j<=4n. Use the apple map and calculate all elements of A. This step takes 16n^2 times. Then calculate all squares of area n^2. if the quare's right-bottom position is (x0, y0), then its left-top position is (x0-n, y0-n), and total of apples in square

$total_s$ = A[x0, y0] – A[x0-n, y0] – A[x0, y0-n] + A[n, n].

There are (3n+1)^2 squares. We need to find the maxium $total_s$ While calculate totals. The maxium $total_s$ is our aim. The time is 16n^2 + (3n+1)^2 = O(n^2).



5. For the convenience, I'm using log(n) instead of $\log_2(n)$.

(a) f(n) = (log(n))^2 = log(n) * log(n);

g(n) = log([n ^ log(n)]^2) = 2 * log([n ^ log(n)]) = 2 * log(n) * log(n).

∵ f(n) = log(n) * log(n) = 1/2 * g(n), and ∀ n>=1, f(n)<=1/2 * g(n).

∴ f(n) = O(g(n)).

(b) We want to show thar f(n) = n^10 = O(2^(n^(1/10))) = O(g(n)), which means that we have to show that n^10 < c + 2^(n^(1/10)) for some positive c and all sufficiently large n. But, since the log function is monotonically increasing, this will hold just in case

log f(n) <= log C + log g(n),

just in case

log n <= log C + n^(1/10).

We now see that if we take c = 1 then it is enough to show that

log n <= n^(1/10)

We use the L'Hopital's rule,

n->+∞, (log n)/ (n^(1/10)) = (1/(ln 2 * n))/ (1/10 * n^(-9/10)) = 10/ln2 * (1/n^(1/10)).

When n is large, 10/ln2 * (1/n^(1/10)) < 1, so log n <= n^(1/10).

So, f(n) = O(g(n)).

(c) Assume ∃ c1 can meet ∀ n0, f(n0) = n0^2 or 1 <= c1 * g(n0) = c1*n0.

f(n0) = n0^2 or 1 and f(n0) <= c1 * n0, then ∀n>c1, and n%2==0, f(n)=n^2, c1*g(n)=c1*n; then f(n)> c1*g(n), NOT f(n) <= c1*g(n). This is not consistent with the assumption.

so f(n) ≠ O(g(n)).

!

Assume ∃ c2 can meet ∀ n1, g(n1) = n1 <= c1 * f(n1) = c2 * (n1^2 or 1).

g(n1)=n1 and g(n1)<c2*(n1^2 or 1), then ∀n>c1, and n%2!=0, f(n)=1, g(n) > c2 * f(n), NOT g(n) <= c2 * f(n). This is not consistent with the assumption.

so g(n) ≠ O(f(n)).