Problem 1:

```
let A = [l_1, l_2, ..., l_n]                                  O(n)
A.sort()   # I assume .sort() would sort A in ascending order.   O(n log n)
A.reverse()   # in decreasing order                           O(n)
count = 0                                                     O(1)
for i in A :                                                  O(n)
    if i >= h :   # we eliminate all the ropes with length > h, since   O(1)
        count ++  # they don't need pairs                     O(1)
highestIdx = count                                            O(1)
lowestIdx = n                                                 O(1)
while highestIdx < lowestIdx :   # pairing the rest ropes     O(n)
    if( A[highestIdx] + A[lowestIdx] >= h):                   O(1)
        count ++                                              O(1)
        highestIdx ++                                         O(1)
        lowestIdx --                                          O(1)
    else:                                                     O(1)
        lowestIdx --                                          O(1)
return Count                                                  O(1)
```

We see that the overall complexity is $O(n \log n)$.

Problem 2:

```
struct task    // setup
{ double s, f;  // this task's starting & finishing time
  bool operator<(const point& other) const
  {  // overload the "<" operator, s.t. taski < taskj : if taski.s < taskj.s, or
     if( this->s < other.s)       // if taski.s = taskj.s && taski.f < task.f,
     { return true; }             // all the other situations makes taski > taskj.
     else if (this->s == other.s)
     { if (this->f < other.f)
         { return true;}
         return false; }
     return false; } };
```

Pseudocodes:

```
Let A = [task1, task2, ..., taskn]   O(n)
A.sort()     O(n log n)
Let H = [ ]    O(1) # the purpose of H is to store the tasks we do not
We let H be a hash table   # check. For example, in the codes below, if S(j) contains
                           # task i, then |S(i)| ≥ |S(j)| for sure, so we skip checking task i.
curMin = ∞  O(1)  # keep track the min(S(j)).
curIdx = 0  O(1)  # keep track the index of min(S(j)).
# now consider the taskj, we want to compute the maximum number of tasks
# can be done before s(j), call it k, and the maximum number of tasks can be
# done after f(j), call it m. So S(j) = k+1+m.
for j in range(0, n):  O(n)
    if j in H:   O(1) since H is hash table
        continue
# for loop continued in next page
```

$k = 0$, $i = 0$, $lastTaskIdx = 0$ # if i=0, it is the first one, no last task.

repeatedly binary search for i s.t.: $O(\log n)$ # taski starts after the last task ends

$((A[i].f < A[j].s) \&\& ((i == 0) || (A[i].s > A[lastTaskIdx].f)))$:

taski ends before taskj starts

k++ $O(1)$

lastTaskIdx = i $O(1)$

$m = 0$, $i = n$, $lastTaskIdx = n$

repeatedly binary search for i s.t.: $O(\log n)$

$((A[i].s < A[j].f) \&\& ((i == n) || (A[i].f < A[lastTaskIdx].s)))$:

m++ $O(1)$

lastTaskIdx = i $O(1)$

H.insert(i) $O(1)$

if curMin > m+1+k: $O(1)$

curIdx = j $O(1)$

now the for loop ends.

return (A[curIdx])

#recall that we created the hash table "H" to avoid checking the later tasks
which the current tasks include. Thus, if 2 tasks are compatible, the later
one will not be checked, and if 2 tasks are not compatible, there would not
be a binary search between them, so binary search occurs for a constant time.

Overall, the algorithm is $O(n \log n)$

Problem 3:

When there are two tasks i and j, and assume they are accomplished consecutively, also assume the starting date is the same, say the k-th day. So there are $T-k$ days remained. If $T-k > d_i + d_j$:

① when i is completed first: i would make $(T-k-d_i) \cdot r_i$ revenue, and
j would make $(T-k-d_i-d_j) \cdot r_j$ revenue,
in total $(T-k-d_i-d_j)(r_i+r_j) + d_j \cdot r_i$ revenue.

② when j is completed first: i would make $(T-k-d_i-d_j) \cdot r_i$ revenue, and
j would make $(T-k-d_j) \cdot r_j$ revenue,
in total $(T-k-d_i-d_j)(r_i+r_j) + d_i \cdot r_j$ revenue.

Thus, i should be completed earlier than j if

$$(T-k-d_i-d_j)(r_i+r_j) + d_j \cdot r_i > (T-k-d_i-d_j)(r_i+r_j) + d_i \cdot r_j$$

$$d_j \cdot r_i > d_i \cdot r_j$$

$$\frac{r_i}{d_i} > \frac{r_j}{d_j}.$$

However, if $\frac{r_i}{d_i} > \frac{r_j}{r_j}$ and $d_i + d_j < T-k$: if $(d_i > T-k \;\&\; d_j > T-k)$, or $(d_i > T-k > d_j)$, we should complete d_i; if $(d_i < T-k < d_j)$, then we should complete d_j.
If $(d_i < T-k)$ and $(d_j < T-k)$, then we don't have time to complete either.

now our pseudocodes are:

```
A = [task1, task2, ..., taskn]          O(n)
A.sort() # by increasing order of ri/di    O(n log n)
taskOrder = []                 O(1)
for i in A:                    O(n)
    if di <= T:                O(1)
        taskOrder.append(di)   O(1)
        T = T - di             O(1)
return taskOrder # the task in the front means it should be done first.  O(1)
```

In total, this algorithm is $O(n \log n)$.

Problem 4:

We can regard each town as a vertex, and each road as an edge.

Since the towns are all connected, we have $m \geq n-1$.

Now suppose there are n towns, and Alice in town_i, Bob in town_j, Charlie in town_k.

Then we run Dijkstra's algorithms 3 times from town_i, town_j, town_k to get the shortest distance from this three towns to all other towns, with each $t(e)$ as weight.

Then, the town_x with minimum max(dist(town_i, town_x), dist(town_j, town_x) dist(town_k, town_x)) is the town we are looking for.

Notice that Dijkstra's algorithm is $O(m \log n + n \log n) = O(m \log n)$, and run it three times is still $O(m \log n)$. Finding the minimum time to reach the town is $O(n)$. So in total, $O(m \log n)$.

Problem 5.

We shall use heap queue:

```
let A=[planet1, planet2, ..., planetn]  # A contains all planets including s & t
let E=[(planetFrom1, planetTo1), (planetFrom2, planetTo2), ...(planetFromm, planetTom)
  # E contains all edges in form of 2-entry tuples.
let C=[C1, C2, ..., Cn]  # C contains the capacity of each planet, same planet has
                         # same index as in A.


minDays(s,t,C,E):  # s&t are the indices of planet s&t in A and C.
    set every plants' dist as ∞.
    dist[s]=0  # set the distance of planet s as 0

    heap=[]   # min heap
    for i in range(1, n+1):
        heap.append([(dist[i],i])
    heap.heapify()

    days = 0

    for plants in heap:
        distance, u = heap.pop()
        if u==t:  # reached t
            return C[u]+days
        for v in A:  # update other nodes.
            if dist[v]>dist[u]+C[u]:
                dist[v]=dist[u]+C[u]
                heap.push((dist[v],v))
        days += C[u]
```