Q1.

a. In this case, each inserted item will become the right child of the rightmost leave. Inserting kth element costs $c(k-1)+d$ time, where c, d are constants. The cost of inserting n elements is $\sum_{i=1}^{n} c(i-1)+d$, say, $O(n^2)$. Thus the asymptotic performance of a TREE-INSERT operation for this case is $O(n)$.

b. In this case, it is trivial to prove that an element is added in level k only if all positions in level $k-1$ are occupied. Say, the height of the final tree in this case is same with a *complete binary tree*, which is $O(lgn)$. It is also the performance of a TREE-INSERT.

c. In this case, TREE-INSERT is actually to add an element to the end of a linked list. Thus the performance is $O(1)$.

d. The worst case of this case is all insertion happens on the right side or left side. Thus the performance is same with case **a**, say, $O(n)$.

It is easy to prove that the final tree after inserting n elements has the same expected height of a randomly built binary search tree (described in section 12.4 of textbook), say, $O(lgn)$. It is also the average performance of a TREE-INSERT**.**

Q2.

13.1-6.

Consider a red-black tree with black-height *k*. The smallest internal node number is $2^k - 1$, when every node is black. If only every other nodes is black the total internal node number is $2^{2k} - 1$ as the case of the largest number.

13.1-7.

The tree has the largest possible ratio of red internal nodes to black internal node if we interleave red and black nodes on each path and make the number of real red leaves as many as possible. In this case, each red node has two child black nodes, and root node should be black. Thus $n_{black} = 2n_{red} + 1$. Since $n_{black} + n_{red} = n$, then we have $\frac{n_{red}}{n_{black}} = \frac{n-1}{2n+1}$. On the other hand, the minimum ratio can be achieve with a tree that include black internal nodes only (it can happen with $n = 2^k - 1$,where *k* is the height). The ratio for this case is *0*.

13.2-1.

The pseudocode for RIGHT-ROTATE is symmetric to that of LEFT-ROTATE in the textbook. Exchange *left* and *right* everywhere in LEFT-ROTATE.

Q3.

a. Assume for the purpose of contradiction that there is no point of maximum overlap in an endpoint of a segment. The maximum overlap point *p* is in the interior of *m* segments. Actually, *p* is in the interior of the intersection of those *m* segments. Now look at one of the endpoints p' of the intersection of the *m* segments. Point p' has the same overlap as *p* because it is in the same intersection of *m* segments, and so p' is also a point of maximum overlap. Moreover, p' is in the endpoint of a segment (otherwise the intersection would not end there), which contradicts our assumption that there is no point of maximum overlap in an endpoint of a segment. Thus, there is always a point of maximum overlap which is an endpoint of one of the segments.

b. Keep a balanced binary tree of the endpoints. That is, to insert an interval, we insert its endpoints separately. With each left endpoint *e*, associate a value *p*[*e*] = +1 (increasing the overlap by 1). With each right endpoint *e* associate a value *p*[*e*] = −1 (decreasing the overlap by 1). When multiple endpoints have the same value, insert all the left endpoints with that value before inserting any of the right endpoints with that value.

Here is some intuition. Let $e_1,\ e_1, \dots,\ e_n$ be the sorted sequence of endpoints corresponding to our intervals. Let *s(i,j)* denote the sum $p[e_i] + p[e_{i+1}] + \cdots + p[e_j]$ for $1 \leq i \leq j \leq n$. We wish to find an *i* maximizing *s(*1*,i)*.

Each node *x* stores three new attributes. Suppose that the subtree rooted at *x* includes the endpoints $e_{l[x]}, \dots,\ e_{r[x]}$. We store *v*[*x*]=*s(l*[*x*],*r*[*x*]*)*, the sum of the values of all nodes in *x*'s subtree. We also store *m*[*x*], the maximum value obtained by the expression *s(l*[*x*],*i)* for any *i* in {*l*[*x*],*l*[*x*]+1*, . . . ,r*[*x*]}. Finally, we store *o*[*x*] as the value of *i* for which *m*[*x*] achieves its maximum. For the sentinel, we define *v*[*nil*[*T*]]=*m*[*nil*[*T*]]=0.

We can compute these attributes in a bottom-up fashion to satisfy the requirements of Theorem 14.1:

v[*x*] = *v*[*left*[*x*]] + *p*[*x*] + *v*[*right*[*x*]] *,*

$$m[x] = \max\begin{cases} m[left[x]] & (\text{max is in } x'\text{s left subtree}) \\ v[left[x]] + p[x] & (\text{max is at } x) \\ v[left[x]] + p[x] + m[right[x]] & (\text{max is in } x'\text{s right subtree}) \end{cases}$$

The computation of *v*[*x*] is straightforward. The computation of *m*[*x*] bears further explanation. Recall that it is the maximum value of the sum of the *p* values for the nodes in *x*'s subtree, starting at *l*[*x*], which is the leftmost endpoint in *x*.s subtree and ending at any node *i* in *x*.s subtree. The value of *i* that maximizes this sum is either a node in *x*'s left subtree, *x* itself, or a node in *x*'s right subtree. If *i* is a node in *x*'s left subtree, then *m*[*left*[*x*]] represents a sum starting at *l*[*x*], and hence *m*[*x*] = *m*[*left*[*x*]]. If *i* is *x* itself, then *m*[*x*] represents the sum of all *p* values in *x*'s left subtree plus *p*[*x*], so that *m*[*x*] = *v*[*left*[*x*]] + *p*[*x*]. Finally, if *i* is in *x*'s right subtree, then $m[x]$ represents the sum of all p values in x's left subtree, plus $p[x]$, plus the sum of some set of p values in x's right subtree. Moreover, the values taken from x's right subtree must start from the leftmost endpoint in the right subtree. To maximize this sum, we need to maximize the sum from the right subtree, and that value is precisely $m[right[x]]$. Hence, in this case, $m[x]=v[left[x]]+p[x] + m[right[x]]$.

Once we understand how to compute $m[x]$, it is straightforward to compute $o[x]$ from the information in x and its two children. Thus, we can implement the operations as follows:

- INTERVAL-INSERT: insert two nodes, one for each endpoint of the interval.
- INTERVAL-DELETE: delete the two nodes representing the interval endpoints.
- FIND-POM: return the interval whose endpoint is represented by $o[root[T]]$.

Because of how we have defined the new attributes, Theorem 14.1 says that each operation runs in $O(\lg n)$ time. In fact, FIND-POM takes only $O(1)$ time.

Q4.

15-2.

Denote a string as a sequence of characters *s[1,...,n]*. We find the longest palindrome subsequence by dynamic programming. Let's consider the longest palindrome subsequence of the sub-string *s[i,...,j]*. If *s[i]=s[j]*, then the palindrome includes *s[i]*, *s[j]* and the palindrome of sub-string *s[i+1,...,j-1]*. Otherwise, the longest palindrome of *s[i,...,j]* belongs to *s[i+1,...,j]* or *s[i,...,j-1]*. Note that each character is a palindrome, so the longest palindrome subsequence of *s[i,i]* is *s[i]*. Define *L(i,j)* as the length of the longest palindrome subsequence of sub-string *s[i,...,j]*, we have the following recursive formula.

$$L(i,j) = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ 2 + L(i+1, j-1) & \text{if } s[i] = s[j] \\ \max\{L(i+1,j), L(i,j-1)\} & \text{otherwise} \end{cases}$$

We can compute all values *L(i,j)* in bottom-up manner and then trace back to find the longest palindrome subsequence from *L(1,n)*. Each value *L(i,j)* is computed in $O(1)$ time, thus the total running time is $O(n^2)$.

15-6.

Let's consider the sub-tree rooted at *A*, if *A* is invited to the party then none of its children is invited. However, if *A* is not invited then each child may or may not be invited. Let *C(A)* be the maximum sum of the conviviality ratings of the sub-tree rooted at *A*. Let $C^i(A)$ and $C^{ni}(A)$ be the maximum value when *A* is invited and not invited, respectively. Then we have:

$$C(A) = \max\{C^i(A), C^{ni}(A)\}$$

$$C^i(A) = \sum_{x \text{ is child of } A} C^{ni}(x)$$

$$C^{ni}(A) = \sum_{x \text{ is child of } A} C(x)$$

If *A* is a leaf, $C^i(A) = c(A)$ and $C^{ni}(A) = 0$ where *c(A)* is the conviviality rating of *A*. Thus, we can compute the optimal value in bottom-up fashion. For each internal node, we spend time proportional to the number of its children, i.e., its degree. Since the total degree of nodes in a tree with n nodes is *O(n)*, the total running time is *O(n)*.

Q5.

16.1-3.

1). One of the counter example of least duration select is like the following table,
$\boldsymbol{i}$ denotes the activity index, the start and end time of activity $\boldsymbol{i}$ are described in rows $\boldsymbol{s_i}, \boldsymbol{t_i}$.

$\boldsymbol{i}$	**1**	**2**	**3**
$\boldsymbol{s_i}$	1	3	4
$\boldsymbol{t_i}$	4	5	7

The solution of the least duration select is $\{\boldsymbol{a_2}\}$ with duration 2. The optimal solution should be $\{\boldsymbol{a_1}, \boldsymbol{a_3}\}$.

2). One of the counter example of most compatible select is like the following table,

$\boldsymbol{i}$	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
$\boldsymbol{s_i}$	1	1	1	1	2	3	4	5	5	5	6
$\boldsymbol{t_i}$	2	3	3	3	4	5	6	7	7	7	7

The solution of this strategy is $\{\boldsymbol{a_i}, \boldsymbol{a_6}, \boldsymbol{a_j}\}$ where $\boldsymbol{i} \in \{1,2,3,4\}, \boldsymbol{j} \in \{8,9,10,11\}$. But the optimal solution should be $\{\boldsymbol{a_1}, \boldsymbol{a_5}, \boldsymbol{a_7}, \boldsymbol{a_{11}}\}$.

3). One of the counter example of most compatible select is like the following table,

$\boldsymbol{i}$	**1**	**2**	**3**
$\boldsymbol{s_i}$	1	2	3
$\boldsymbol{t_i}$	4	3	4

The solution of this strategy is $\{\boldsymbol{a_1}\}$, but the optimal solution is $\{\boldsymbol{a_2}, \boldsymbol{a_3}\}$.

16.2-5.

The idea is to start with the first point which is located left-most, cover by interval of unit length starting from that point and skip all the points covered by this interval. Repeat the same until all points have been covered. If we have a pothole at point x_i, then an interval with unit length, $u = [x_i, x_i + 1]$, covers that point x_j if $x_j \le x_i + 1$. Note that you should sort the point set first before you carry on the process if they are not in order.

The correctness of the algorithm can be shown by the following two facts:

Lemma 1: The greedy choice property denotes, if x is the leftmost point in the given list of points. Then there is an optimal solution containing the unit length interval $u = [x, x + 1]$. This can be shown by considering an optimal solution S. Suppose that interval $u = [x, x + 1]$ is not in S. But, there should be another interval covering x which will denote by $u' = [p, q]$. This will imply $p \le x$. If $p = x$, then $u' = u$. Therefore, $p < x$. We already know that there are no other intervals left of x, so we can move the interval u' such that its left point p will coincide with x. We will not lose any points that are covered by u'. This transition will make u and u' coincide. So, the above statement is correct.

Lemma 2: This problem has an optimal substructure property, such that if x is the leftmost point in the given list X. Let S^* be an optimal solution for X containing the unit interval $u = [x, x + 1]$. Let X' be the list of points in X that are not covered by u. Then, $S' = S^* - \{u\}$ is an optimal solution for X'. This can be shown by contradiction. Suppose S' is not an optimal solution to X'. Since S^* is a solution to X, all potholes which are not covered by u are covered by the intervals in S'. So, if S' is not an optimal solution to X', this means there exists another solution S'', which is smaller than S' and optimal ($|S''| < |S'|$). Then, we can say $S'' \cup \{u\}$ needs to be a solution for X. We already know that $|S''| < |S'|$, therefore we can say $|S'' \cup \{u\}| < |S^*|$, which is a contradiction to our assumption that S^* is an optimal solution. This proves the optimal substructure of the problem and the correctness of the algorithm.

The analysis is straight-forward, since we examine the points once, which will yield $O(n)$ time. If you assume that the given list of points are not sorted according to location, then we also need $O(nlgn)$ time to sort the points first. So, this will make the overall complexity $O(nlgn)$.

16-1.

a. The algorithm is:

- If $n = 0$, the optimal solution is to give no coins.
- If $n > 0$, determine the largest coin whose value is less than or equal to n. Let this coin have value c. Give one such coin, and then recursively solve the subproblem of making change for $n - c$ cents.

To prove the above algorithm always output an optimal solution, we need to show that the greedy-choice property holds, that is, that some optimal solution to making change for n cents includes one coin of value c, where c is the largest coin value such that $c \leq n$. Consider some optimal solution, we will prove that it includes c. Suppose that it does not include c, we consider 4 following cases:

- If $1 \leq n < 5$, then $c = 1$. A solution may consist only of pennies, and so it must contain the greedy choice.
- If $5 \leq n < 10$, then $c = 5$. By supposition, this optimal solution does not contain a nickel, and so it consists of only pennies. Replace five pennies by one nickel to give a solution with four fewer coins.
- If $10 \leq n < 25$, then $c = 10$. By supposition, this optimal solution does not contain a dime, and so it contains only nickels and pennies. Let S be an empty set, we add nickels to S one by one until there is no nickel left. Then we add pennies to S in the same manner. At some point, the total value of nickels and pennies in S is equal to 10 cents. Replace them by one 1 dime we get a better solution.

- If $25 \leq n$, then $c = 25$. By supposition, this optimal solution does not contain a quarter, and so it contains only dimes, nickels, and pennies. If it contains three dimes, we can replace these three dimes by a quarter and a nickel, giving a solution with one fewer coin. If it contains at most two dimes, then some subset of the dimes, nickels, and pennies adds up to 25 cents, and so we can replace these coins by one quarter to give a solution with fewer coins.

Thus there exists an optimal solution that include the greedy choice i.e. the above algorithm is correct. The algorithm can be simplified as following:

- Give $q = \lceil n/25 \rceil$ quarters. That leaves $n_q = n\ mod\ 25$ cents to make change.
- Then give $d = \lceil n_q/10 \rceil$ dimes. That leaves $n_d = n_q\ mod\ 10$ 0 cents to make change.
- Then give $k = \lceil n_d/5 \rceil$ nickels. That leaves $n_k = n_d\ mod\ 5$ cents to make change.
- Finally, give $p = n_k$ pennies.

The running time is O(1).

b. We firstly prove the below claim.
Claim: For $i = 0,1, \dots, k$, let a_i be the number of coins of denomination c^i used in an optimal solution to the problem of making change for n cents. Then for $i = 0,1, \dots, k-1$; we have $a_i < c$.
Proof: If $a_i > c$, we can replace c coins of c^i by a coin c^{i+1} to get a solution with fewer coins.

The greedy algorithm: When the coin denominations are $c^0, c^1, \dots, c^k$, the greedy algorithm to make change for n cents works by finding the denomination c^j such that $j = \max\{0 \leq i \leq k: c^i \leq n\}$, giving one coin of denomination c^j, and recursing on the subproblem of making change for $n - c^i$ cents.
To show that the greedy solution is optimal, we show that any non-greedy solution is not optimal. Consider a optimal solution, which uses no coins of denomination c^j or higher. Let the non-greedy solution use a_i coins of denomination c^i, for $i = 0,1, \dots, j-1$; thus we have $\sum_{i=0}^{j-1} a_i c^i = n$. Since $n > c^j$, we have that $\sum_{i=0}^{j-1} a_i c^i > c^j n$. By the above claim, $a_i < c - 1$ for $i = 0,1, \dots, j-1$. Thus, $\sum_{i=0}^{j-1} a_i c^i \leq \sum_{i=0}^{j-1} (c-1) c^i = c^j - 1 < c^j$. It's contradicted, i.e. the greedy solution is optimal.

c. Let the denominations are $\{4,3,1\}$. With $n = 6$, the optimal solution is two 3 cent coins but the greedy solution is 3 coins including 1 four cent coin, 2 one cent coin.

d. We use dynamic programming to design the algorithm here. Define $c[j]$ to be the minimum number of coins we need to make change for j cents. Let the coin denominations be $d_1, d_2, \dots, d_k$. Since one of the coins is a penny, there is a way to make change for any amount $j \geq 1$. Because of the optimal substructure, we have:

$$c[j] = \begin{cases} 0, & \text{if } j \leq 0 \\ 1 + \min_{1 \leq i \leq k, d_i < j}\{c[j - d_i]\}, & \text{if } j > 0 \end{cases}$$

Now we use the array c of size n to store the values of $c[j], j \leq n$. Another array, *denom*, of size n is used to store the chosen coin to achieve the optimal value $c[j]$.

```
COMPUTE-CHANGE(n, d, k)
for $j = 1$ to $n$ do
  $c[j] \leftarrow \infty$
  for $i = 1$ to $k$ do
    if $j > d_i$ and $1 + c[j - d_i] < c[j]$ then
      $c[j] \leftarrow 1 + c[j - d_i]$
      $denom[j] \leftarrow d_i$
    end if
  end for
end for
return $c$ and *denom*
```

Computing each $c[j]$ takes $O(k)$ time. The running time is $O(nk)$. After having *c* and *denom*, we can output the set of coins by tracking back from $denom[n]$ as the following algorithm.

```
GIVE-CHANGE( j, denom)
if $j > 0$ then
  give one coin of denomination $denom[j]$
  GIVE-CHANGE($j - denom[j]$, *denom*)
end if
```

This algorithm takes call itself at most n time. Thus the running time is $O(n)$.