Practice Sheet 2 Solutions

28th August, 2016

Dynamic Programming

1. **Question:** Given an array of n positive integers $a_1, \cdots, a_n$, give an algorithm to find the length of the longest subsequence $a_{i_1}, \cdots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that each term divides the next term, that is a_{i_j} divides $a_{i_{j+1}}, \forall 1 \leq j < k$.

   Answer:

   A subsequence $a_{i_1} \cdots a_{i_k}$ will be called a dividing subsequence if $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ and a_{i_j} divides $a_{i_{j+1}}, \forall 1 \leq j < k$. Hence it is required to find the length of the longest dividing subsequence.

   Let $dp(j)$ denote the length of the longest dividing subsequence ending at a_j. Clearly, $dp(1) = 1$.

   Suppose we want to calculate $dp(j+1)$.

   Consider the set $s(j+1) = \{x : x < j+1, a_x | a_{j+1}\}$. If $s(j+1) = \emptyset$, then any dividing subsequence ending at j+1 must consist of only a_{j+1} and so $dp(j+1) = 1$.

   Suppose $s(j+1)$ is not empty. Hence any longest dividing subsequence ending at $j+1$ has length atleast 2.

   We observe that if $x \in s(j+1)$, then any longest dividing subsequence ending at x can be extended to a dividing subsequence ending at j+1. Also any longest dividing subsequence ending at j+1, must have the last but one element to belong to the set $s(j+1)$.

   It follows that, any longest dividing subsequence ending at j+1 must be of the form $a_{i_1}, \cdots, a_{i_m}, a_x, a_{j+1}$ where $a_{i_1}, \cdots, a_{i_m}, a_x$ is a longest dividing subsequence of x and $a_x \in s(j+1)$

   Therefore, we get the recurrence,

$$dp(j+1) = 1 \text{ if, } s(j+1) = \emptyset$$
$$dp(j+1) = max\{1 + dp(x) : x \in s(j+1)\} \text{ if, } s(j+1) \neq \emptyset$$

   The required solution is $max\{dp(j) : 1 \leq j \leq n\}$.

   Pseudocode:

   We give an iterated version of the pseudocode. Here dp[] is an array which at the jth postition contains the length of the longest dividing subsequence ending at j.

Algorithm 1 Iterative - Dividingsubseq:

```
dp[0] = 1
ma = 1
for i = 2 to n do
    dp[i] = 1
    for j = 1 to (i-1) do
        if $a_j$ divides $a_i$ then
            dp[i] = max(dp[i],1+dp[j])
        end if
    end for
    ma = max(ma,dp[i])
end for
    return ma
```

Lines 1 and 2 of the algorithm initialize the conditions. Lines 3-9 compute dp(i) for all i, $2 \leq i \leq n$. Line 10 compares the current maximum with the value of dp[i] and updates it if necessary.

The first loop goes through n iterations and in each loop $O(n)$ steps are carried out. Hence the running time of the algorithm is $O(n^2)$.

2. **Question:** Given a $n \times m$ grid of integers where the rows are numbered from 1 to n and the columns are numbered from 1 to m,your task is to answer queries of the form (i,j,k,l) for different $1 \leq i,j \leq n$ and $1 \leq k,l \leq n$. The answer to such a query is the sum of the numbers in the sub-grid bounded by the i,j th rows and the k,l th columns. Preprocess the grid so as to efficiently answer each query in constant time.

   Answer:

   We will assume that the numbers are given in a 2-d array a[][].

   Let $dp(x,y) = \sum_{i=1}^{x} \sum_{j=1}^{y} a[i][j]$. dp(x,y) is the sum of all the numbers in the sub-grid bounded by the 1st,xth rows and 1st,yth columns. Clearly,

$$dp(1,1) = a[1][1]$$

$$\forall y > 1, dp(1,y) = \sum_{j=1}^{y} a[1][j] = a[1][y] + dp(1,y-1)$$

$$\forall x > 1, dp(x,1) = \sum_{i=1}^{x} a[i][1] = a[x][1] + dp(x-1,1)$$

Let $x > 1, y > 1$.

$$\begin{aligned}
dp(x+1,y+1) &= \sum_{i=1}^{x+1}\sum_{j=1}^{y+1} a[i][j] \\
&= \sum_{i=1}^{x+1}\sum_{j=1}^{y} a[i][j] + \sum_{i=1}^{x+1} a[i][y+1] \\
&= \sum_{i=1}^{x+1}\sum_{j=1}^{y} a[i][j] - \sum_{i=1}^{x}\sum_{j=1}^{y} a[i][j] + \sum_{i=1}^{x}\sum_{j=1}^{y} a[i][j] + \sum_{i=1}^{x+1} a[i][y+1] \\
&= \sum_{i=1}^{x+1}\sum_{j=1}^{y} a[i][j] - \sum_{i=1}^{x}\sum_{j=1}^{y} a[i][j] + \sum_{i=1}^{x}\sum_{j=1}^{y+1} a[i][j] + a[x+1][y+1] \\
&= dp(x+1,y) + dp(x,y+1) - dp(x,y) + a[x+1][y+1]
\end{aligned}$$

Now suppose we are given a query of the form (i,j,k,l).

Let $r1 = min(i,j), c1 = min(k,l), r2 = max(i,j), c2 = max(k,l)$

The required answer is $\sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y]$. Let S = $\sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y]$.

$$\begin{aligned}
S &= \sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y] \\
&= \sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y] + \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] - \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] \\
&= \sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y] + \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] + \sum_{x=1}^{r2}\sum_{y=1}^{c1} a[x][y] - \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] - \sum_{x=1}^{r2}\sum_{y=1}^{c1} a[x][y] \\
&= \sum_{x=r1}^{r2}\sum_{y=c1}^{c2} a[x][y] + \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] + \sum_{x=1}^{r2}\sum_{y=1}^{c1} a[x][y] - \sum_{x=1}^{r1}\sum_{y=1}^{c1} a[x][y] + \sum_{x=1}^{r1}\sum_{y=1}^{c1} a[x][y] - \sum_{x=1}^{r1}\sum_{y=1}^{c2} a[x][y] \\
&\quad - \sum_{x=1}^{r2}\sum_{y=1}^{c1} a[x][y]
\end{aligned}$$

Notice that the first four terms combined is dp(r2,c2). The other three terms are dp(r1,c1), dp(r1,c2), dp(r2,c1) respectively. Hence the equation becomes,

$S = dp(r2,c2) - dp(r1,c2) - dp(r2,c1) + dp(r1,c1)$

Hence every query can be answered in constant time if the we preprocess the dp function for each i, j.

Pseudocode:

We provide an iterative version of the algorithm. Here dp[][] is a 2-d array, where dp[i][j] stores the value $\sum_{i=1}^{x}\sum_{j=1}^{y} a[i][j]$ and Q is the number of queries.

Algorithm 2 Prefix Sums:

```
dp[1][1] = a[1][1]
for i = 2 to n do
    dp[i][1] = a[i][1] + dp[i-1][1]
end for
for j = 2 to m do
    dp[1][j] = a[1][j] + dp[1][j-1]
end for
for i = 2 to n: do
    for j = 2 to m: do
        dp[i][j] = dp[i-1][j] + dp[i][j-1] - dp[i][j] + a[i][j]
    end for
end for
for g = 1 to Q: do
    (i,j,k,l) is the query
    r1 = min(i,j)
    r2 = max(i,j)
    c1 = min(k,l)
    c2 = max(k,l)
   Output (dp[r2][c2] - dp[r1][c2] - dp[r2][c1] + dp[r1][c1])
end for
```

Lines 1-7 initialize the necessary base cases for the recurrence. Lines 8-12 calculate dp[i][j] $\forall i, j \geq 2$. Lines 13-20 processes the given query and returns the value of the query in constant time.

Lines 2-4 and 5-7 each have one for loop running n-1 and m-1 iterations respectively and hence each of them take $O(n)$ and $O(m)$ time respectively. In Lines 8-12 there are n-1 iterations of the outer loop and each iteration has (m-1) iterations on the inner loop which take constant time to work out. Hence calculating dp[i][j] forall i,j takes $O(mn)$ time.

Lines 13-20 have one for loop having Q iterations with each iteration taking constant time and hence answering Q queries takes $O(Q)$ time.

Hence the overall complexity is $O(mn + Q)$.

3. **Question:** Given an array of n integers $a_1, \cdots a_n$. You start with a score of 0. In each step, you are allowed to choose an index i different from 1 and n and delete a_i from the array. The value of $a_{i-1} \times a_{i+1}$ gets added to your score. After deleting a_i, the new array is of length n-1 and is $a_1, \cdots, a_{i-1}, a_{i+1}, \cdots, a_n$. The process stops when you have only 2 elements remaining. Compute the maximum possible score you can get.

   Answer:

   Let $a_i, \cdots, a_j$ denote the subarray from the ith to jth elements of the original array. Suppose we want to find the maximum score attainable restricted to this subarray.

   Observation 1: Suppose a_k is the last element to be removed. Then, the problem reduces to finding the maximum score in $a_i, \cdots, a_k$ and $a_k, \cdots, a_i$ and adding a cost of $a_i \times a_j$ to the combined maximum score.

   Let dp(i,j) denote the maximum score from the subarray $a_i, \cdots, a_j$ where a_i and a_j are not removed.

   Clearly, $dp(i, i) = 0$

Suppose $j > i$.

By our first observation, $dp(i,j) = max\{dp(i,k) + dp(k,j) + a_i \times a_j : i < k < j\}$

Clearly, the required answer for our original question is dp(1,n).

Pseudocode:

We give an iterarive version of the alogrithm where the iteration is done over the length of the subarrays. dp[][] is an array storing the maximum score for the subarray $a_i, \cdots, a_j$.

Algorithm 3 Comp-Opt:

```
for i = 1 to n do
    dp[i][i] = 0
end for
for i = 1 to n do
    for j = 1 to n-1 do
        if $i + j < n$ then
            dp[i][i+j] = 0
            for k = i+1 to i+j-1 do
                dp[i][i+j] = max(dp[i][k]+dp[k][i+j]+ $a[i] \times a[j]$, dp[i][i+j])
            end for
        end if
    end for
end for
    return dp[1][n]
```

Lines 1-3 initialize the base cases. Lines 4-13 implement the given recurrence by iterating over the length of the intervals. Clearly, the algorithm takes $O(n^3)$ time.

4. **Question:** Given a set S of n distinct positive integers $a_1, \cdots, a_n$ and a positive integer M, give an algorithm to compute the number of subsets T of S such that the sum of elements of T is exactly M.

   Answer:

   Let dp(x,i) denote the number of subsets of $\{1 \cdots i\}$ whose sum is exactly x.

   Since the numbers are distinct and positive, the number 0 if it occurs in the array occurs only once and hence we have,

$$dp(0,i) = 0, \text{ if } a_j \neq 0, \forall j, 1 \leq j \leq i$$

$$dp(0,i) = 1, \text{ if } \exists j, 1 \leq j \leq i \text{ such that } a_j = 0$$

   Also $dp(x,1) = 1$ if $x = a_1$ and 0 otherwise.

   Now let $x \geq 0, i > 1$. Consider the set of all subsets of $\{1 \cdots i\}$ whose sum is exactly x. Let this set be S. If $S = \emptyset$, then dp(x,i) = 0. Suppose $S \neq \emptyset$.

   Let s be a subset of $\{1, \cdots, i\}$ whose sum is exactly x.

   Clearly, $i \notin s \iff$ s is a subset of $\{1, \cdots, i-1\}$ whose sum is exactly x.
   Also, $i \in s \iff$ s is a subset of $\{1, \cdots, i-1\}$ whose sum is exactly x-a[i].

   Therefore, we get the recurrence, $dp(x,i) = dp(x, i-1) + dp(x - a[i], i-1)$.

Pseudocode:

dp[][] is a 2-d array where dp[i][j] stores the number of subsets of $\{1, \cdots, j\}$ whose sum is exactly i. We also assume that the numbers are given in an array a[].

Algorithm 4 Subset-Sum:

```
 1: f = 0
 2: for i = 1 to n: do
 3:     if a[i] = 0 then
 4:         f = 1
 5:     end if
 6:     dp[0][i] = f
 7: end for
 8: for j = 1 to M do
 9:     if j == a[1] then
10:         dp[j][1] = 1
11:     else
12:         dp[j][1] = 0
13:     end if
14: end for
15: for i = 2 to M do
16:     for j = 2 to n do
17:         dp[i][j] = dp[i][j-1]
18:         if $i \geq a[j]$ then
19:             dp[i][j] = dp[i][j] + dp[i-a[j]][j-1]
20:         end if
21:     end for
22: end for
        return dp[M][n]
```

Lines 1-10 initialize the base cases of the recurrence. Lines 11-18 fill up the other entries of the table as given by the recurrence.

It is clear that computing the base cases takes $O(n + M)$ time. In lines 11-18 there are M-1 iterations of the outer loop and each such iteration takes $O(n)$ time. Hence the overall complexity of computing the other entries takes $O(Mn)$ time.

Hence the overall complexity is $O(Mn)$. This is pseudopolynomial in terms of M since it is linear in terms of M, but is exponential in terms of the number of bits required to represent M.

5. **Question:** Given an array of n distinct integers $a_1, \cdots, a_n$, give an algorithm to find the longest continuous mountain present in the array, that is, the longest sub-array $a_i, \cdots, a_j$ such that there exists an index k, $i \leq k \leq j$ such that the sequece $a_i, \cdots, a_k$ is strictly increasing and the sequence $a_k, \cdots, a_j$ is strictly decreasing. A sub-array is a contiguous portion of the array.

Answer:

In this solution, whenever we refer to mountains we always mean continuous mountains.

Call a mountain, a proper mountain if it is neither a strictly increasing sequence nor a strictly decreasing sequence.

Clearly, any proper mountain starting at a_i and ending at a_j consists of atleast three elements and contains a $k, i < k < j$ such that $a_i, \cdots, a_k$ is strictly increasing and $a_k, \cdots a_j$ is strictly decreasing.

Let $dp(i)$ denote the length of the longest mountain ending at a_i $\forall i, 1 \leq i \leq n$. Let $par(i) = i - dp(i) + 1$. (If M is the longest mountain ending at a_i, then par(i) is the starting point of M). It is clear that there cannot be more than one longest mountain ending at a_i.

Clearly, dp(1) = 1.

Suppose $j > 1$. Consider the longest mountain M ending at a_j. Clearly M always contains atleast two elements, since a_{j-1}, a_j is always a mountain.

So now the problem reduces to finding the optimal mountain among all the mountains ending at a_{j-1} which can be extended to a mountain ending at a_j.

We consider two cases:

Case i): $a_{j-1} < a_j$

If the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a strictly increasing sequence, then clearly, $a_{par(j-1)}, \cdots, a_j$ is the longest mountain ending at a_j.

Suppose the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a strictly decreasing sequence. Clearly, a_{j-1}, a_j is the only mountain ending at a_j and hence the longest mountain ending at a_j.

If the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a proper mountain, then $\exists k$ such that $par(j-1) < k < j-1$ and $a_{par(j-1)} \cdots a_k$ is a strictly increasing sequence and $a_k \cdots a_{j-1}$ is a strictly decreasing sequence. Therefore, a_{j-1}, a_j is the only mountain ending at a_j and hence the longest mountain ending at a_j.

Case ii): $a_{j-1} > a_j$

Suppose the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a strictly increasing sequence. Clearly, $a_{par(j-1)}, a_j$ is the the longest mountain ending at a_j.

If the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a strictly decreasing sequence, then clearly, $a_{par(j-1)}, \cdots, a_j$ is the longest mountain ending at a_j.

If the sequence $a_{par(j-1)}, \cdots, a_{j-1}$ is a proper mountain, then $\exists k$ such that $par(j-1) < k < j-1$ and $a_{par(j-1)} \cdots a_k$ is a strictly increasing sequence and $a_k \cdots a_{j-1}$ is a strictly decreasing sequence. Hence, $a_{par(j-1)}, \cdots, a_j$ is a proper mountain ending at a_j and no other mountain ending at a_j can have a length greater than the length of this mountain. Hence $a_{par(j-1)}, \cdots, a_j$ is the length of the longest mountain ending at a_j.

These observations immediately give rise to an algorithm.

Pseudocode:

In the following pseudocode, dp[] is an array, where dp[i] stores the length of the longest mountain ending at i. a[] is an array which stores the numbers and n is the size of the array. ma is the variable which stores the length of the longest mountain. mark[] is an array, where

i) mark[i] = -1 if the longest mountain ending at i is strictly decreasing
ii) mark[i] = 1 if the longest mountain ending at i is strictly increasing
iii) mark[i] = 0 if the longest mountain ending at i is a proper mountain

By convention, we take mark[1] = 1

Algorithm 5 Longest-Mountain:

```
dp[1] = 1
mark[1] = 1
if n is equal to 1 then
    return 1
end if
if $a[2] > a[1]$ then
    dp[2] = 2
    mark[2] = 1
else
    dp[2] = 2
    mark[2] = -1
end if
ma = 2
for i = 3 to n do
    if $a[i] > a[i-1]$ then
        if mark[i-1] equal to 1 then
            dp[i] = dp[i-1]+1
            mark[i] = 1
        else if mark[i-1] equal to -1 then
            dp[i] = 2
            mark[i] = 1
        else
            dp[i] = 2
            mark[i] = 1
        end if
```

```
26:     else
27:         if mark[i-1] equal to 1 then
28:             dp[i] = dp[i-1]+1
29:             mark[i] = 0
30:         else if mark[i-1] equal to -1 then
31:             dp[i] = dp[i-1]+1
32:             mark[i] = -1
33:         else
34:             dp[i] = dp[i-1]+1
35:             mark[i] = 0
36:         end if
37:     end if
38:     ma = max(ma,dp[i])
39: end for
         return ma
```

Lines 1-12 are used to initialize the base cases for i = 1 and 2. The loop in line 14 runs for n-2 iterations and at each iteration, it exhausts all possible cases discuused above and stores the required values.

Line 38 updates the current value of ma with the value of the longest mountain seen so far.

The running time of this program is clearly $O(n)$.

Greedy

6. **Question:** Given n real numbers on the real line $a_1, \cdots, a_n$ you need to cover all of them using closed intervals of length 1 (i.e) you need to place some intervals on the real line such that every a_i is contained within at least one of the closed intervals. Give an algorithm to find the minimum number of such intervals that you need.

Answer:

We can assume that the given numbers are distinct. Whenever we refer to an optimal solution in this problem, we assume that the intervals in the optimal solution are sorted according to their starting point.

Clearly, any optimal solution cannot contain an interval which contains none of the given numbers.

Lemma 1: There exists an optimal solution O = $\{I_1, \cdots, I_k\}$, (where $I_1, \cdots, I_k$ are the intervals in the optimal solution) such that $I_a \cap I_{a+1} = \emptyset \forall a, 1 \leq a \leq k$. (We will call such a property as having no intersections).

Proof: Let O' = $\{I_1, \cdots, I_k\}$ be any optimal solution and suppose that a is the first position, where $I_a \cap I_{a+1} \neq \emptyset$. Let b_a, e_a denote the starting and ending points of I_a and b_{a+1}, e_{a+1} denote the starting and ending points of I_{a+1}.

Clearly, $b_{a+1} \leq e_a$. Consider the new interval I'_{a+1} whose starting point is the first a_l such that $a_l > e_a$ and whose ending point is $a_l + 1$.

Consider the set $O = \{I_1, \cdots, I_a, I'_{a+1}, I_{a+2}, \cdots, I_k\}$. Clearly, all elements in the array which were covered by O' are covered by O as well and the number of intersections between the intervals has reduced by one. Also, the number of the intervals in O is the same as that of O'.

Hence O is also an optimal solution. Applying this process iteratively, we get an optimal solution which contains no intersections.

Lemma 2: There is an optimal solution O = $I_1, \cdots, I_k$ having no intersections such that $b_j = a_l, \forall j, 1 \leq j \leq k$ for some a_l, where b_j is the starting point of I_j and where a_l can depend on I_j

Proof: Let O = $I_1, \cdots, I_k$ be an optimal solution with no intersections. We will inductively transform O into the needed optimal solution.

Base Case: Suppose $b_1 = a_1$, then we are done. Suppose $b_1 \neq a_1$. Clearly, $b_1 < a_1$. Updating $b_1 = a_1$and $e_1 = a_1 + 1$ and resolving the intersections as was done in Lemma 1, we see that the new solution O' is still optimal.

Induction Hypothesis: Suppose $b_j = a_l, j \geq 1$ for some l such that $I_1, \cdots, I_j$ has no intersections.

Induction Step: Suppose $b_{j+1} = a_m$ for some m. Then the induction step is true.
Suppose $b_{j+1} \neq a_m$ for all m. Let $a_{m-1} < b_{j+1} < a_m$. Set $b_{j+1} = a_m$ and $e_{j+1} = a_m + 1$ and resolve the intersections (if any) that occur from I_{j+1} to I_n. Again, we see that in the new solution, the optimality is preserved.

Therefore, by the Principle of Mathematical Induction, Lemma 2 is true.

Motivated by these two lemmas, we give an algorithm and prove its correctness.

We sort the numbers in ascending order. We process through the list in ascending order and whenever we find an a_l which has not been covered by any interval, we place an interval starting at a_l and remove all the points from the list which lie in this interval.

Pseudocode:

Here count is the variable which on temination, gives the minimum number of intervals.

Algorithm 6 Intervals

```
count = 0
Sort the given numbers in ascending order.
i = 1
while $i \leq n$ do
    count = count + 1
    x = a[i] + 1
    while $a[i] \leq x$ do
        i++
    end while
end while
    return count
```

We will prove that the above algorithm returns the minimum number of intervals necessary to cover all the numbers.

Let O = $I_1, \cdots, I_k$ be an optimal solution with no intersections such that each one of it's intervals start at some a_l. Let I = $J_1, \cdots, J_q$ be the solution generated by our algorithm.

Claim: Suppose $a_i \in I_w$, then $a_i \in J_r$ for some r where $r \leq w$.
Proof: Throughout this proof we use b_j to denote the starting point of I_j.

We proceed by induction on the number of elements in the array a.

Base Case: Clearly, $a_1 \in I_1$ and $a_1 \in J_1$. Hence the claim is true for a_1.

Induction Hypothesis: Suppose the claim is true $\forall a_j, 1 \leq j \leq i$.

Induction Step: Suppose $a_{i+1} \in I_w$. There are two cases:

Case i: $b_w = a_{i+1}$

Hence, $a_i \in I_{w-1}$. By I.H, $a_i \in J_r$ for some $r \leq w-1$. Therefore, a_{i+1} can either belong to J_r or it can belong to J_{r+1}. In either case, $a_i \in J_r$ for some $r \leq w$ and so the induction step is complete.

Case ii: $b_w < a_{i+1}$

By assumption, we can take b_w to be some a_j. By I.H, $a_j \in J_r$ for some $r \leq w$.

Suppose $r \leq w-1$. Clearly, $a_i \in J_e$ for some $e \leq w$ and hence the induction step is complete.

Suppose $r = w$ and a_j is the starting point of J_r. Clearly, $a_i \leq a_j + 1$ and hence $a_i \in J_r$.

Suppose $r = w$ and a_j is not the starting point of J_w. Therefore, $a_{j-1} \in J_w$ and $a_{j-1} \in I_{w-1}$ (because a_j is the starting point of I_w). But this is not possible due to I.H.

Hence for all possible cases of a_j the induction step holds.

Hence by the Principle of Induction, the claims is true for all elements of the array.

The claim immediately implies that I is an optimal solution.

We can use mergesort to sort the numbers in $O(nlogn)$ time. The while loop in the pseudocode, clearly checks an element only once and so the number of iterations in the while loop is $O(n)$ with each iteration taking constant time.

Hence the overall complexity is $O(nlogn)$.

7. Question: You work in a pizza shop which offers a wide range of toppings. The toppings are stored in separate jars which are kept in a cupboard (assume that the jars have unlimited capacity). But the cupboard is far from the counter, and so you keep some jars in a nearby desk. Unfortunately the desk is small and you can keep at most S jars on it at any given time. So when a customer asks for a topping which is not on your desk, and if your desk is full, you will have to return some jar back to the cupboard and replace it with the necessary jar. Miraculously you know the exact sequence of toppings that will be demanded by your customers, which is i$a_1, ..., a_n$. Each a_i corresponds to a topping. Describe an algorithm to find the least number of trips that you will have to make to the cupboard. You can start off with any S jars on the desk. Assume $1 \leq a_i \leq n$.

Answer: We first present an algorithm and prove it's optimality.

Algorithm:

While we need to bring a_i to the desk and a_i is not in the desk:
Put back the item which is needed farthest in the future and include a_i in the desk.

We will now prove its optimality. Let I_F be the sequence of moves made by our algorithm.

Definition: Let I be any sequence of moves and define I' to be the reduction of I as follows:

Suppose in step i, we bring in an item a_j which was not requested at step i. In the new sequence I', we do not bring in a_j. We bring in a_j only when it is requested in the step k.

Hence, the trip made at the move by I' at step k, can be accounted for by the trip made by I at step i.

Therefore, the number of trips made by I' is atmost that of I.

Lemma : Let I be a reduced sequence that makes the same trips as I_F through the first j steps. Then there is a reduced sequence I" which makes the same trips as I_F through the first j+1 steps and incurs no more trips than I does.

Proof: Consider the j+1 th request. Since I and I_F have agreed up till this point, if a_{j+1} lies in the desk of I, then no trip is necessary. Hence we can set $I" = I$. Suppose a_{j+1} is not in the desk of I and further suppose that I and I_F both remove the same element from the desk. Then also, we can set $I" = I$ and we would be done.

Suppose a_{j+1} is not in the desk of I and I takes way b while I_F takes away c where $c \neq b$. Then construct I" as follows. Let the first j moves be the same as I. Let the $j + 1$th move be that of taking away c.

From the j+2 th move let I" be the same as that of I until one of the following happens:

Case i):

$d \neq b, c$ is requested and I takes away c to make room for d. Since the desk of I" and I are the same except for b and c, it must be that d is not in the desk of I. Hence we can take away b and bring in d in I" and from this point onwards, we can again set I" = I

Case ii):

b is requested and I takes away c'. If c' is equal to c, then we can simply access b from the desk of I" and we can set I" = I from this point onwards.

Suppose $c' \neq c$. Then , we can make I" take away c' and bring in c and set I" = I from this point onwards. But I" is no longer a reduced sequence, therefore we make I" to be the reduction of I".

Hence in both the cases, we have a new reduced sequence I" which agrees with I_F till the first j+1 steps and incurs no more trips than I does.

One of these two cases, will appear before c is requested, because I_F took away c which would be the farthest in the future to be requested and hence b would always be requested before c is and hence case ii) comes into hold.

Using this Lemma, we prove optimality as follows. Let $\bar{I}$ be a reduced sequence. Using the Lemma we can construct a sequence I_1 which would agree through with the first move of I_F and incur no more trips than $\bar{I}$.

By induction, we can construct for all j, I_j in the same manner. In every step optimality is preserved.

Hence I_F incurs no more trips than $\bar{I}$ and hence is optimal.

8. **Question:** You are given 2 integers, l and r. You need to find two numbers L and R, which satisfy $l \leq L \leq R \leq r$, and which maximize the bit-wise XOR of L and R.

   Answer: We first give an algorithm and prove it correctness.

   Let $l_1, \cdots, l_m$ and $r_1, \cdots, r_n$ be the binary representations of l and r respectively. If $n \neq m$, pad zeroes in the front of l_1 to make both of them to of length n.

Algorithm 7 XOR

```
 1: if l == r then return 0
 2: end if
 3: i = 1
 4: if $l_i$ == 0 and $r_i$ == 1 then
 5:     Let L be such that the binary representation of L is $0, 1, \cdots, 1$ where the number of 1's is n-1.
 6:     Let R be such that the binary representation of R is $1, 0, \cdots, 0$ where the number of 0's is n-1.
        return L and R.
 7: else
 8:     i++
 9: end if
10:
11: while $i \leq n$ do
12:     if $l_i$ == 0 and $r_i$ == 1 then
13:         Let L be such that the binary representation of L is $l_1, \cdots, l_{i-1}, 0, 1, \cdots, 1$ where the number of 1's after $l_{i-1}$ is n-i.
14:         Let R be such that the binary representation of R is $r_1, \cdots, r_{i-1}, 1, 0, \cdots, 0$ where the number of 0's after $r_{i-1}$ is n-i.
            return L and R
15:     end if
16:     i++
17: end while
```

Proof of optimality:

Suppose the binary representation of l and r are the same. Then $l = r$ and hence the maximum xor is 0.

Suppose 'i' is the first point where $l_i \neq r_i$. Then, $l_i = 0$ and $r_i = 1$ because $r \geq l$. Clearly, any number lying in between l and r must have the same first $i - 1$ digits as that of l and r in its binary representation.

Therefore, to maximise the bitwise XOR between any two numbers between l and r, we can only modify the digits from i to n. Therefore, we consider L and R as constructed in the algorithm. Clearly, $l \leq L \leq r$ and $l \leq R \leq r$. Let M be the bitwise XOR of L and R. Clearly, the first i-1 digits of M are the same as the XOR of l_j and r_j for $1 \leq j \leq i - 1$. The next n-i+1 digits of M are 1 by construction.

Since we cannot modify the first i-1 digits of L and R, it follows that L and R are the required numbers.

The time complexity of the algorithms is $O(logl + logr)$.

Divide and Conquer

9. Question: Given n integers, $a_1, ..., a_n$ you need to find the sub-array with the maximum sum. A sub-array is a contiguous portion of the array, and its sum is just the sum of all the array values which it covers.

Answer:

We split the array into two parts, $a_1, \cdots, a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2 \rfloor + 1}, \cdots, a_n$. Any subarray with maximum sum can either lie entirely in the first or second partitions or lies in between the two.

We evaluate the maximum subarray sum in both the partitions and let M be the maximum of those two.

Now, any subarray lying in between the two partitions must contain $a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2 \rfloor + 1}$.

Therefore, among all subarrays ending at $a_{\lfloor n/2 \rfloor}$, we find the one with maximum sum. Also among all subarrays starting at $a_{\lfloor n/2 \rfloor + 1}$, we find the one with maximum sum and append these two.

If the resulting sum of this subarray is greater than M, we report this sum. Else we report M.

Implementation:

Algorithm 8 MaxSum(A,n)

```
if n == 1 then return A[1]
end if
if n == 2 then return max(A[1],A[2],A[1]+A[2])
end if
$g = \lfloor n/2 \rfloor$

Let $A_1$ be $a_1, \cdots, a_g$
Let $A_2$ be $a_{g+1}, \cdots, a_n$.
$M_1 = MaxSum(A_1, g)$
$M_2 = MaxSum(A_2, n - g)$

$M = max(M_1, M_2)$

i = g-1
ma1 = A[g]
sum = A[g]
while $i \geq 1$ do
    sum = sum + A[i]
    ma1 = max(sum,ma1)
end while

i = g+2
ma2 = A[g+1]
sum = A[g+1]
```

```
25: while $i \leq n$ do
26:     sum = sum + A[i]
27:     ma2 = max(sum,ma2)
28: end while
29:
30: ma = ma1 + ma2
        return max(ma,M)
```

Lines 1 till 4 constitute the base case. Lines 5-12 form the recursive part of the algorithm. Lines 14-30 compute the maximum sum of the subarray containing $a_{\lfloor n/2 \rfloor}$ and $a_{\lfloor n/2 \rfloor + 1}$.

For the above algorithm, we get the recurrence, $T(n) = 2T(n/2) + O(n)$ whose solution is clearly $O(nlogn)$

There is an $O(n)$ dynamic progrraming solution for this problem.

Let dp(i) denote the maximum sum of all subarrays ending at i.

Clearly, dp(1) = a_1.

Suppose we want to compute $dp(i), i > 1$. There are two cases: Either the maximum sum subarray ending at i starts at i or does not start at i. In the former case, the sum is just a[i]. In the latter case, i-1 certainly lies in the subarray and so the maximum sum would be $dp(i-1) + a[i]$.

Therefore, we get the recursion, $dp(i) = max(a[i], dp(i-1) + a[i])$

Computing dp(i) from 1 to n, and taking the maximum among all those, we see that the whole solution takes $O(n)$ time.

10. Question: You are given n integers, $a_1, ..., a_n$, in an array. A majority element of this array is any element occurring in more than $\lceil n/2 \rceil$ positions. Assume that elements cannot be ordered or sorted, but can be compared for equality. Design a divide and conquer algorithm to find a majority element in this (or determine that no majority element exists)

Answer: We split the array into two parts : $a_1, \cdots, a_{\lfloor n/2 \rfloor}$ and $a_{\lceil n/2 \rceil}, \cdots, a_n$. Suppose an element occurs more than $\lceil n/2 \rceil$ positions in the original array. Then that element must occur for more tha $\lceil n/4 \rceil$ positions in one of the arrays.

Therefore, we find all elements which occur atleast for $\lceil n/4 \rceil$ in the left partition and also in the right partition.

Now we compare the elements which we have got from the left partition to all the elements in the right partition to check if any element occurs more than $\lceil n/2 \rceil$ positions and vice versa.

The following observations can be summed up into the pseudocode as follows:

Algorithm 9 MoreThan(A,n)

```
if n == 1 then return A[1]
end if
if n == 2 then
    if A[1] == A[2] then return A[1]
    else
      return None
    end if
end if

$g = \lfloor n/2 \rfloor$
$A_1 = a_1, \cdots, a_g$
$A_2 = a_{g+1}, \cdots, a_n$

$X = MoreThan(A_1, g)$

if $X \neq None$ then
    Test X with all other cards in $A_2$ and return X if the total number of occurrences are greater than $\lceil n/2 \rceil$
else
    $X = MoreThan(A_2, n - g)$
    if $X \neq None$ then
        Test X with all other cards in $A_1$ and return X if the total number of occurrences are greater than $\lceil n/2 \rceil$
    end if
end if
      return None
```

We get the recurrence $T(n) = 2T(n/2) + cn$ for the above algorithms whose solution is $O(nlogn)$.