Shooting Balloon (Finding Max Score)

27-April-2016 Advance Problem

Sundeep/Nishank

Shooting Balloon (Finding Max Score)

SWC-Advance-Test- 27-April-2016

There will be a N Balloons marked with value Bi (where B(i…N)).

User will be given Gun with N Bullets and user must shot N times.

When any balloon explodes then its adjacent balloons becomes next to each other.

User has to score highest points to get the prize and score starts at 0.

Below is the condition to calculate the score.

1. When Balloon Bi Explodes then score will be a product of Bi-1 & Bi+1 (score = Bi-I * Bi+1).
2. When Balloon Bi Explodes and there is only left Balloon present then score will be Bi-1.
3. When Balloon Bi Explodes and there is only right Balloon present then score will be Bi+1.
4. When Balloon Bi explodes and there is no left and right Balloon present then score will be Bi.

Write a program to score maximum points.

Conditions:

- Execution time limits 3 seconds.
- No of Balloons N, where 1 <= N <= 10
- Bi value of the Balloon 1 <= Bi <= 1000.
- No two Balloons explode at same time.

Input/ Output

Input:
Consists of TC (1 <= TC <= 50).
N – No of Balloons.
B0…..BN N Balloons with their values .
Output:
#TC SCORE

Sample Input:
5
4
1 2 3 4
5
3 10 1 2 5
7
12 48 28 21 67 75 85
8
245 108 162 400 274 358 366 166
10
866 919 840 944 761 895 701 912 848 799

Sample Output:
#1 20
#2 100
#3 16057
#4 561630
#5 6455522

Analysis

1)Aim is to find max score

2) Max score depend on points on neighbor, however there is no easy way to find which sequence which gives max score, so only way is to find the all possible sequence can get max out of it.

3) As order matters in sequence for input N we can have N! sequences, ie. nPn ways (1st balloon N ways, 2nd N-1 ways …last balloon 1 ways N*(N-1)(N-2)..2*1= N!

rock_climbing.txt

Complexity:

- To generate the all sequence O(N!)
- To Get the Score for 1 sequence, for each balloon in sequence we need to left and right neighbors worst case need complete traversal in array so complexity is O(N*N)
- Total complexity is O(N!) * O(N*N) (note: computation has done at end of each sequence)
- 50 TC , N<= 10 => 50 * is O(N! * N*N) => 50 * 100 * 10! => 5000 * 3628800 => 1.5 * 10^10 this cannot be executed in given 3 sec (10^9 instruction per second).
- So need to look for optimization

Pseudo code to generate all sequences.

```
INPUT[N]
CHOICE[N] <= -1 //initialize to -1
Permute(0)
Permute(Position)
{
//stop condition
If( all balloon shot )
{
Compute the score for this sequence in CHOICE[]
If score better than previous then store
}

For i:0~N-1
{
If (ith balloon not selected // CHOICE[i]==-1)
{
Select ith balloon  // CHOICE[Position]= i
Permute (Position+1)
Unselect ith balloon// CHOICE[Position]= -1
}
}
}
```

Optimization

- We can see in above algorithm 2 major operation are carried out 1) generate all sequence O(N!) and 2) computing score for each sequence O(N*N)
- We cannot optimize the algorithm generate all sequences however we can reduce the computing part further.
- Optimization computing part
  - If can optimize the finding the neighbor to O(1) we can reduce computation part to O(N) which leads our algo to execute in 1.5 * 10^9 which can be achieved in 3 sec.
  - Alternatively we can compute the score for each chosen balloon to shoot “on the go” here finding neighbor is extra when each time balloon is chosen which can be O(N) and also reduce 1.5 * 10^9
- If we combine 1 and 2 we can further reduce the time to 1.5 * 10^8

Algorithm to get neighbors

Naïve method by O(N):

Neighbor(chosen)

For Left: chosen-1~0 if Left th balloon not chosen break;

For Right: chosen+1~N-1 if right th balloon not chosen break;

if(Right==N)

Right=-1;

Return Left and right ;

Optimized way by O(1)

1. Keep 2 array left[] and right[] which contain neighbors of each balloon.
2. Initially neighbor are known, for ith balloon left is i-1 and right is i+1 except that 1st balloon will have no left and last have no right.
3. When balloon is chosen we can obtain its right and left by O(1)
4. When a balloon is shot update neighbor left[i+1]=left[i] right[i-1]=right[i]

Note:

Instead of calling the new function to get left and right calculating left and right inside the recursive faction will reduce many hidden instructions as to call new function compiler add many instruction which can be reduced

Alternative Way

Way to compute the score on the go

- Pass current score variable to recursive function
- When a balloon is chosen to shoot get the left and right neighbors
- Compute the score gained by shooting chosen balloon
- Add this to given score and pass to next level

```
Permute(Position, score)
{
//stop condition
If( all balloon shot )
{
If score better than previous then store
}

For i:0~N-1
{
If (ith balloon not selected // CHOICE[i]==-1)
{
Select ith balloon  // CHOICE[Position]= i
Gain = Compute the gain by shooting ith balloon
Permute (Position+1, score+ Gain)
Unselect ith balloon// CHOICE[Position]= -1
}
}
}
```

Errors/Bugs

- Error in algorithm to generate permutation
- Not optimizing the Computing the score algorithm.
- Stop condition in recursive problem
- Selecting greedy methods

Alternative optimized approach(Divide and Conquer) and Dynamic programming

The problem at first doesn't seem like a divide and conquer problem.

- Reason: If we select a balloon(for bursting) then our array would be divided into two sub arrays. But these two sub arrays won't be independent sub problems.
  - Example
    - Consider 5 balloons B1,.., B5. Bursting B3 divides the array into two sub-array {B1, B2} and {B4, B5}. But these two sub array are not independent of each other ie. score for bursting B4 is dependent on bursting order of {B1, B2}.

B1 B2 X B4 B5

Key Insight

- To divide the problem into two halves we have to ensure that any action(bursting of balloon) in one half doesn't affect score of the other half.
- If we fix a balloon and ensure that we won't burst it until we burst all the balloons to the left of it and all the balloon to the right of it then we can successfully divide the problem into two sub-problems.
- Example
  - Consider the previous case of five balloons. Now instead of bursting B3 we fix that we will burst B3 after all the balloons this makes {B1, B2} and {B4, B5} independent of each other ie score for bursting B4 is now independent of {B1, B2}.

- Another way to visualize the divide and conquer approach is that we think of the problem in reverse. The parallel problem would be given a set of n deflated balloons each with a score, choose the order in which you will inflate the balloon. The score for inflating the balloon is equal to product of score attached to the balloons located left and right to the mentioned balloon.

Pseudo Code

- Note:
  - We store the the input score values in the array inp_arr[N+2].
  - The values corresponding to the ith baloon is store at inp_arr[i].
  - inp_arr[0] = inp_arr[N+1] = 1;

```
getMaxScore(inp_arr, left_limit, right_limit, N){
   initialize max_score = 0;   //Max Score Value to Be Returned
   for(i: left+1 to right-1){
      initialize curr_score = 0;
      curr_score = getMaxScore(inp_arr, left, i, N) + getMaxScore(inp_arr, i, right,
      if(left == 0 && right == N){
         curr_score += inp_arr[i];
      }
      else{
         curr_score += inp_arr[left]*inp_arr[right];
      }
      //Update max_score value
      if(curr_score> max_score){
         max_score = curr_score;
      }
   }
   return max_score;
}
```

The above problem can be easily optimized to include memoization using 2 Dimensional DP Matrix.

Execution time for different approach

For input given in this document.

Generating sequences and computing at end by list for finding neighbor

Execution time: 0.934000 seconds.

On the way compute

- external call for finding neighbor : Execution time: 1.223000 seconds.
- Inline for finding neighbor: Execution time: 0.657000 seconds.
- List for finding neighbor: Execution time: 0.616000 seconds.

Divide and conquer:

- Execution time: 0.004000 seconds.
- with DP: Execution time: 0.001000 seconds.