Dynamic Programming (DP)

HKOI Camp 2019

Alex Poon

Agenda

- Basic idea of DP
- Problem sharing about
  - Segment Tree / Monotone Queue Optimization
  - Sketch DP
  - Divide & Conquer Optimization
  - CHT Optimization

Basics Idea of DP

What Dynamic Programming Solve

- In Competitive Programming, we also encounter problem related to Combinatorial Optimization (組合最佳化)
- i.e. You have many combination to choose, which is the best?

- DP is one of the most common way to solve it
- Or sometimes to solve the problem about "count the number of valid combination"

Problem I

Problem I – (Trivial DP task)

- Example:
- Given some events, each event has a starting time, ending time and happiness
- Find a set of event to attend to maximize your happiness such that no collision

```
Input:
1 2 2
3 4 2
2 3 5
5 7 4
8 9 5
```

```
Output:
14 (attending the 3rd 4th 5th event)
```

Problem I – (Trivial DP task)

- DP idea → Find optimal result for PART of events first, then use the result to help us compute the optimal result for ALL events
- Observation 1 → An event have earlier starting time must attend before another event have later starting time
- We can sort the event by the starting time first!
- 1 2 2
  2 3 5
  3 4 2
  5 7 4
  8 9 5

Problem I – (Trivial DP task)

- Assume we have an array res[] to store the following
  res[i] = the optimal happiness we can attend
  1. if we only consider the first i event
  2. we must attend the i^{th} event
  3. the i^{th} event must be the last event we attend

     (it is always optimal by observation 1)
- By this definition, after computing res[], the answer we need is the maximum in it

Problem I – (Trivial DP task)

- How to compute res[] ?
- 1. Think of base case → when i = 1, we only has 1 event, attend it!
  res[1] = happiness[1]
- 2. Think can we use res[1]..res[i] to compute res[i + 1]?

Problem I – (Trivial DP task)

- How to compute res[] ?
- 1. Think of base case → when i = 1, we only has 1 event, attend it!
  res[1] = happiness[1]
- 2. Think can we use res[1]..res[i] to compute res[i + 1]?
- Hint: Look once again what res[i] means

Problem I – (Trivial DP task)

- res[i] = the optimal happiness we can attend
  1. if we only consider the first i event
  2. we must attend the i^{th} event
  3. the i^{th} event must be the last event we attend

     (it is always optimal by observation 1)
- To compute res[i], assume the 2^{nd} last event we attend is event j
  How to compute the happiness with this assumption?

Problem I – (Trivial DP task)

- To compute res[i], assume the 2^{nd} last event we attend is event j
  How to compute the happiness with this assumption?
- res[i] = res[j] + happiness[i]
- We do not need to care what is the 3^{rd} , 4^{th} last event we attend as we know, consider all the event we attend before event j, the optimal happiness = res[j]
- We call this transition formula

Problem I – (Trivial DP task)

- So, what we need to do is only to iterate which is the 2nd last event we will attend
- ```
  for (int i = 1; i <= n; i++) {
    res[i] = happiness[i]; // assume event i is the 1st event we attend
    for (int j = 1; j < i; j++) // iterate the 2nd last event
      if (end[j] < start[i]) // no collision
        res[i] = max(res[i], res[j] + happiness[i]);
  }

  answer = max(res[1] .. res[n])
  ```
  ```

Problem I – (Trivial DP task)

- This is the general idea of DP
- The following will introduce some optimization of DP, classic way of DP by some example

Problem I – (Trivial DP task)

- This is the general idea of DP
- The following will introduce some optimization of DP, classic way of DP by some example

Common form of DP

Dp[i] = f[i] + g[j]
@MonotoneQueue, @SegmentTree

Dp[i] = f[i] + g[j]

- A common form of transition is dp[i] = f[i] + g[j]
- This means the current optimal value dp[i] (or res[i]) is only depends on some constant f[i] (it is constant as `i` is fixed when computing dp[i]) and a single varying value g[j]
- For example, the transition of the previous example is also in this form: res[i] = max(res[i], res[j] + happiness[i])

Dp[i] = f[i] + g[j]

- A common form of transition is dp[i] = f[i] + g[j]
- For this kind of transition we just care about the varying part g[j]
- i.e. we simply find the maximum g[j] among all valid j
- Usually it can be done by segment tree / monotone queue

Segment Tree Optimization

```
for (int i = 1; i <= n; i++) {
  res[i] = happiness[i]; // assume event i is the 1st event we attend
  for (int j = 1; j < i; j++) // iterate the 2nd last event
    if (end[j] < start[i]) // no collision
      res[i] = max(res[i], res[j] + happiness[i]);
}
```

- What we care is the maximum res[j] among all j satisfying end[j] < start[i]
- Can use segment tree to compute

```
res[i] = happiness[i] + query_max_between(1, start[i] - 1)
update_value(end[i], res[i]); // push the value res[i] to segment tree end[i] pos
```

Segment Tree Optimization

- Time complexity improved from O(n^2) to O(n log n)

Problem II

@MonotoneQueue, @SegmentTree

Problem II - Number for Bowling++

- http://poj.org/problem?id=3350
- Given an array of integer consists of positive & negative number
- You can select K part of the array, each part contains M consecutive elements, overlap allowed
- Maximize the sum of the selected element, each element only counts once
- (**If the segment you choose contain the 1st or last element , you can choose < M consecutive element in that segment)

Problem II - Number for Bowling++

- Sample:
- K = 3, M = 3, array = {2, 8, -5, 3, 5, 8, 4, 8, -6}
- Segment 1: {2, 8, -5, 3, 5, 8, 4, 8, -6}
- Segment 2: {2, 8, -5, 3, 5, 8, 4, 8, -6}
- Segment 3: {2, 8, -5, 3, 5, 8, 4, 8, -6}
- Sum = 2 + 8 + 3 + 5 + 8 + 4 + 8 = 38

Problem II - Number for Bowling++

- Solution → Dynamic Programming
- Let's dp[i][j] = the optimal value can be attained where we have selected i segment, where the ith segment we selected is ended at the j element
- To calculate dp[i][j], we know that the ith segment we choose is ended at position j. What we care is the 1^{st} ~ $(i-1)^{th}$ segment we picked
- dp[i – 1][] store that information

Problem II - Number for Bowling++

- Assume the i – 1 segment we picked is ended at position k
- The optimal value we can attain is dp[i – 1][k]
- We also need to care to cases:
  - 1. The last segment and the current segment is overlapping
    - `dp[i][j] = dp[i – 1][k] + RangeSum(k+1, j)`
  - 2. Not overlapping
    - `dp[i][j] = dp[i – 1][k] + RangeSum(j–M+1, j)`

Problem II - Number for Bowling++

- 1. The last segment and the current segment is overlapping
  - `If (k >= j – M + 1)`
  - `dp[i][j] = dp[i – 1][k] + RangeSum(k+1, j)`
- 2. Not overlapping
  - `If (k < j – M + 1)`
  - `dp[i][j] = dp[i – 1][k] + RangeSum(j–M+1, j)`
- To calculation RangeSum, we can use partial sum →
- `sum[j] – sum[k] for case 1, sum[j] – sum[j – M] for case 2`

Problem II - Number for Bowling++

- Solution:
- ```
  for (int i = 1; i <= K; i++)
    for (int j = 1; j <= N; j++)
      for (int k = 1; k <= j; k++)
        if (k >= j – M + 1)
          dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] - sum[k]) (overlap)
        else
          dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] – sum[j-M])
  ```
- Time complexity: O(KN^2)
  ```

Problem II - Number for Bowling++

- ```
  if (k >= j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] - sum[k]) (overlap)
  else if (k < j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] – sum[j-M])
  ```
- Note that for overlapping case
  - `dp[i-1][k] + sum[j] – sum[k] = sum[j]+(dp[i-1][k]-sum[k]) = f[j]+g[k]`
- Note that for non-overlapping case
  - `dp[i-1][k] + sum[j] – sum[j-M] = (sum[j]-sum[j-M])+dp[i-1][k] = f[j]+g[k]`
- As f[j] is the constant part for j, what we wanna find is the maximum g[k] for a specific range or k
  ```

Problem II - Number for Bowling++

- We can use Segment tree Or Monotone queue to speed up
- ```
  if (k >= j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] - sum[k]) (overlap)
  else if (k < j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] – sum[j-M])
  ```
- `dp[i][j] = sum[j] + SegmentTreeQueryMax(j-M+1, k)`
- `dp[i][j] = max(dp[i][j], SegmentTreeTwoQueryMax(1, j – M)`
- `updateSegmentTreeValue(dp[i-1][k]-sum[k], k)`
- `updateSegmentTreeTwoValue(dp[i-1][k], k)`
  ```

Problem II - Number for Bowling++

- Note that the range (j – M + 1, j) and the range (1, j – M) is sliding window for j, Monotone queue also works
- (You may refer to dp III powerpoint for what is monotone queue)
- Time complexity: O(NK log N) (segment tree) or O(NK) (monotone queue)

Problem II - Number for Bowling++

- We can use segment tree Or Monotone queue to speed up
- ```
  if (k >= j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] - sum[k]) (overlap)
  else if (k < j – M + 1)
    dp[i][j] = max(dp[i][j], dp[i-1][k] + sum[j] – sum[j-M])
  ```
- `dp[i][j] = sum[j] + SegmentTreeQueryMax(j-M+1, k)`
- `dp[i][j] = max(dp[i][j], SegmentTreeTwoQueryMax(1, j – M)`
- `updateSegmentTreeValue(dp[i-1][k]-sum[k], k)`
- `updateSegmentTreeTwoValue(dp[i-1][k], k)`
  ```

Problem III

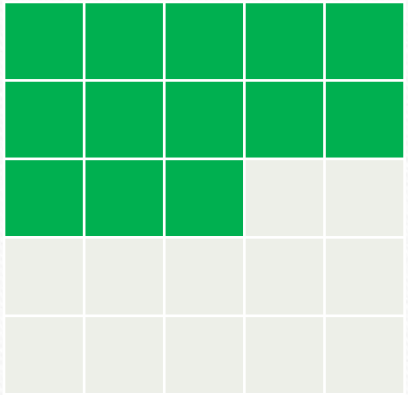
@Sketch DP, @state dp

Problem III – Campus Design

- https://icpcarchive.ecs.baylor.edu/external/66/6634.pdf
- Given an N * M "01" grid. Use 1x2, 2x1 and 1x1 block to cover all cells marked as "1" where number of 1x1 block used >= C && <= D
- Given N, M, C, D and the 01 grid, find the number of valid covering
- N <= 100, M <= 10, C <= D <= 20

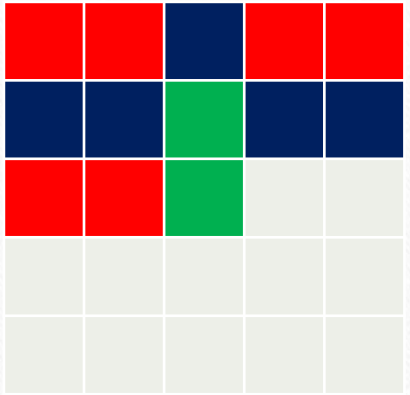
Problem III – Campus Design

- dp[i][j][c] = number of way to cover the board from 1, 1 to i, j, used c 1x1 board
- Each time we try to put a 1x2 block or 2x1 block where it cover (i, j) and (i, j – 1) or (i, j) and (i – 1, j)
  i.e. we do not allow putting at (i, j) and (i, j + 1) as dp[i][j][c] is defined as cover from 1, 1 to i, j only



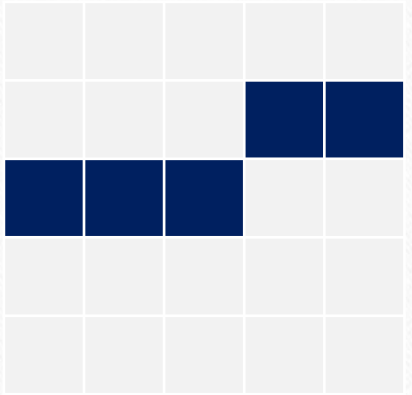
Problem III – Campus Design

- dp[i][j][c] = number of way to cover the board from 1, 1 to i, j, used c 1x1 board
- However, we may not needed to cover the whole grid in some intermediate step
- E.g. In (i, j) = (3, 3), even (2, 3) is not covered before, we are still able to cover it, we may need to add a dimension in our dp array



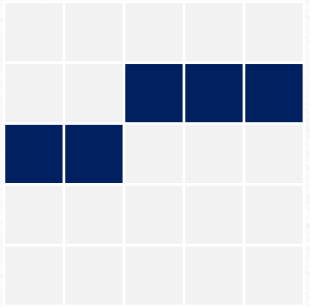
Problem III – Campus Design

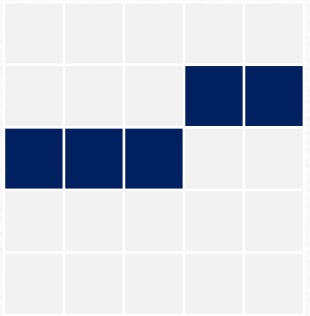
- dp[i][j][c][s] = number of way to cover the board from 1, 1 to i, j, used c 1x1 board, s is the state of … this 5 cells
- If we store the state of these cells, we can easily transit from the last cell (i, j – 1) to the our current cell



Problem III – Campus Design

- If we add a 1 x 2 block: we need the last cell is empty
- `if ((j-1)th bit of s of dp[i][j-1][c][s] is empty)`
  `dp[i][j][c][s | 1<<(j – 1) | 1<<j] += dp[i][j-1][c][s]`
- If we add a 2 x 1 block: we need the last row cell is empty
- `if (jth bit of s of dp[i][j-1][c][s] is empty)`
  `dp[i][j][c][s | 1<<(j)] += dp[i][j-1][c][s]`

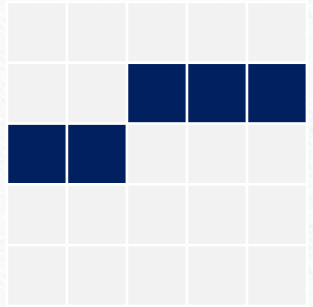


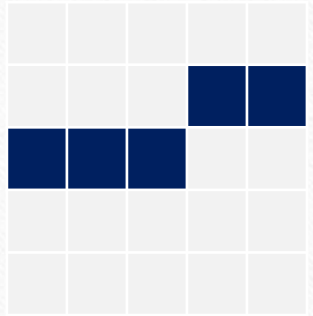


Problem III – Campus Design

- Instead of simply store the state of last row, storing the state of cell in this way sometimes make things way easier

- Coding time!!!





Problem IV

@D&C Optimization

Problem IV – Grouping

- Link: Add later
- You are given an array of N integers, create another array of K integers such that
- For each of the N integers, find the number in the K integers which have smallest absolute different with it
- We want the sum of the absolute different is minimum, find this minimum possible value
- Sample N = 5, K = 3, {1, 5, 7, 10, 14}
  We may choose {1, 7, 14}, the different is 0 + 2 + 0 + 3 + 0 = 5 which is minimum
- N, K <= 3000

Problem IV – Grouping

- Observation: We can transform the problem to the following:
  - Divide the integers into K groups
  - Then we find the best integer that minimize the sum of different to the element in that group
- E.g. {1, 5, 7, 10, 14}
- Group 1 → {1}
- Group 2 → {5, 7, 10}
- Group 3 → {14}

Problem IV – Grouping

- Observation 2:
- We can sort the element first, then consecutive element will belongs to the same group
  - Intuitively, closer element belong to the same group is more optimal
- For each group, the element that can minimize the sum of different is the median of all element in that group
- E.g. {1, 2, 7, 8, 9}, we will choose 7 to minimize the sum of different
- For even number e.g. {1, 6, 7, 8}, both 6 also 7 also minimize

Problem IV – Grouping

- With these observation, we can think of the DP solution
- Let dp[i][j] = the minimum value we can attain by dividing the first j elements to i groups
- Think of base case first: dp[1][j] = median of 1..i
- To calculate dp[i][j], we can iterate the group the element j belongs to: Let k be the 1st element that belongs to the same group with element j
- `dp[i][j] = min(dp[i][j], dp[i - 1][k - 1] + SumOfDiff(k, j))`

Problem IV – Grouping

- Code:
- ```
  for (int i = 1; i <= K; i++)
    for (int j = 1; j <= N; j++)
      for (int k = 1; k < j; k++)
        dp[i][j] = min(dp[i][j], dp[i-1][k-1] + SumOfDiff(k, j))
  ```
- Time complexity: O(KN^2) too slow for K, N <= 3000
  ```

Problem IV – Grouping

- Let's say can we use the technique we taught to optimize the solution:
- Segment tree or Monotone queue only works for DP formula in form of
  - `dp[i] = f[i] + g[j]`
- In this problem, our dp formula is
  - `dp[i][j] = dp[i-1][k-1] + SumOfDiff(k, j)`
  - Even if we remove the constant i
  - `dp[j] = dp[k – 1] + SumOfDiff(k, j)` → `dp[i] = g[j] + h[j][i]`
- The h[j][i] exists, so we cannot use this technique

Problem IV – Grouping

- The optimization we may use here is D&C
- This optimization can be used when the following holds:
  - `For dp formula dp[i] = max/min(dp[i], f[i] + g[j] + h[i][j])`
  - Let opt[i] be the best j for dp[i]
  - `i.e. dp[i] = f[i] + g[opt[i]] + h[i][opt[i]]`
  - If opt[i] is monotonic increase / decrease, then we can use D&C optimization
  - `i.e. opt[i] <= opt[i + 1]  holds for all i`

Problem IV – Grouping

- `dp[i][j] = dp[i-1][k-1] + SumOfDiff(k, j)`
- The way we optimize is, we calculate dp[i][mid] and find the opt[mid] first
- Then for calculating dp[i][j] for j < mid, we just need to iterate k from 1 to opt[mid]
- For calculating dp[i][j] for j > mid, we just need to iterate k from opt[mid] to j
- We optimize the range to search recursively to achieve a better time complexity

- Here we can calculate dp[i][mid] before dp[i][j (j < mid)] because the info we needed to compute dp[i][j] is only dp[i-1][], so rearrange the order of computing still works.

Problem IV – Grouping

- Code:

```
void FindOptimal(int i, int l, int r, int lb, int rb) {
  int mid = (l + r) / 2;
  for (int k = lb; j <= rb; i++)
    if (dp[i-1][k-1] + SumOfDiff(k, mid) < dp[i][mid])
      dp[i][mid] = dp[i-1][k-1] + SumOfDiff(k, mid),
      opt[mid] = k;
  FindOptimal(i, l, mid - 1, lb, opt[mid]);
  FindOptimal(i, mid + 1, r, opt[mid], rb);
}
```

Problem V

@CHT DP

Problem V – Commando

- Link: https://judge.hkoi.org/task/AP101
- You are given an array of N integers, we want to divide the sequence into some consecutive groups
- For each group, we compute the some of them, let them be groupSum(i)
- Then, given a, b, c, we calculate f(x) = ax^2+bx+c for each groupSum()
- We want to maximize the sum of f(groupSum())
- Constraints: -5 < a < -1, -1e7 <= b, c <= 1e7, 1 <= a[i] <= 100, N <= 1,000,000

Problem V – Commando

- Sample: N = 4, a, b, c = {-1, 10, -20}, array = {2, 2, 3, 4}
- Answer = 9
- Divide to {2, 2}, {3}, {4} → {4, 3, 4}
- -1 * 4 * 4 + 10 * 4 - 20 = 4
- -1 * 3 * 3 + 10 * 3 – 20 = 1
- → 4 + 1 + 4 → 9

Problem V – Commando

- Solution: dp
- `dp[i]=dp[j]+a*(sum[i]-sum[j])*(sum[i]-sum[j])+b*(sum[i]-sum[j])+c`

- Naïve looping over the optimal j need O(n^2)
- To optimize this, we need to simplify the dp formula first

Problem V – Commando

- `dp[i]=dp[j]+a*(sum[i]-sum[j])*(sum[i]-sum[j])+b*(sum[i]-sum[j])+c`
- `dp[i] = dp[j] + a(sum[i]^2 + sum[j]^2 – 2sum[i]sum[j]) + b(sum[i]-sum[j]) + c`

- `dp[i] = (a * sum[i]^2 + b * sum[i] + c) + (a * sum[j]^2 – b * sum[j]) - 2a * sum[i]sum[j]`

- Note that the first part is constant part when fixing i, the second part is related to parameter j only
- `dp[i] = f[i] + g[j] + 2a * sum[i] * sum[j]`

Problem V – Commando

- `dp[i] = f[i] + g[j] + 2a * sum[i] * sum[j]`
- `We say this is form of:`
- `dp[i] = f[i] + g[j] + h[i] * l[j]`

- For dp formula look like this, we can use CHT (convex hull trick to optimize it) from O(n^2) to O(n log n log n) (also by CDQ D&C))
- If l[j] is monotonic, we can use CHT only O(n log n)
- If both l[j] and h[i] monotonic, a stronger version of CHT can be applied to achieve O(n)

Problem V – Commando

- `dp[i] = f[i] + g[j] + 2a * sum[i] * sum[j]`

- For dp formula look like this, we can use CHT (convex hull trick to optimize it)

Problem V – Commando

- Consider use j and k (k < j < i) to update i
- dp[i] = f[i] + g[j] + 2a * sum[i] * sum[j] vs
- dp[k] = f[i] * g[k] + 2a * sum[i] * sum[k]
- j is better than k if:
- f[i] + g[j] + 2a * sum[i] * sum[j] >= f[i] * g[k] + 2a * sum[i] * sum[k]
- g[j] – g[k] >= 2a * sum[i] * (sum[k] – sum[j]) → always < 0
- (g[j] – g[k]) / (sum[j] – sum[k]) >= 2a * sum[i]
- Normally, moving sum[k] – sum[j] to another side need reverse the >=
  We change to sum[j] – sum[k] instead

Problem V – Commando

- if (g[j] – g[k]) / (sum[j] – sum[k]) >= 2a * sum[i]
- Let m(j, k) = (g[j] – g[k]) / (sum[j] – sum[k])
- j is better than k for i otherwise, k is better than j
- Consider we have 3 value (l, k, j) to update dp[i] (l < k < j)
- if m(k, l) >= m(j, k) then k is no longer useful
  - Case 1: m(k, l) >= m(j, k) >= 2a * sum[i] → j is better than k in this case
  - Case 2: m(k, l) >= m(j, k), m(j, k) <= 2a * sum[i]
    → l is better than k in this case for this I
    → for other i that m(j, k) >= 2a * sum[i], as m(k, l) >= m(j, k)
    → we know m(k, l) also >= 2a * sum[i] → l is better than k

Problem V – Commando

- if m(k, l) >= m(j, k) then k is no longer useful
- Therefore, we can keep a set of useful point only, to update dp[i]
- When adding a point j to consideration, we pop all the point in our useful point set k where m(k, l) >= m(j, k)
- We can keep a monotonic slope point set → then we can use ternary search to optimize our dp

Problem V – Commando

- Code: Now code