Dynamic Programming (I)

Theogry

Presquisture

Recursion

Divide and Conquer

Big-O notation

Recall

Fibonacci sequence:

$$f(n) = \begin{cases} f(n-1) + f(n-2) & \text{if } n > 2 \\ 1 & \text{otherwise} \end{cases}$$

How do we calculate f(n) using computer?

Fibonacci sequence

```
int f(int x)
    if (x <= 2) return 1;
    else return f(x-1) + f(x-2)
```

Time complexitiy?

Fibonacci sequence

```
int f(int x)
    if (x <= 2) return 1;
    else return f(x-1) + f(x-2)
```

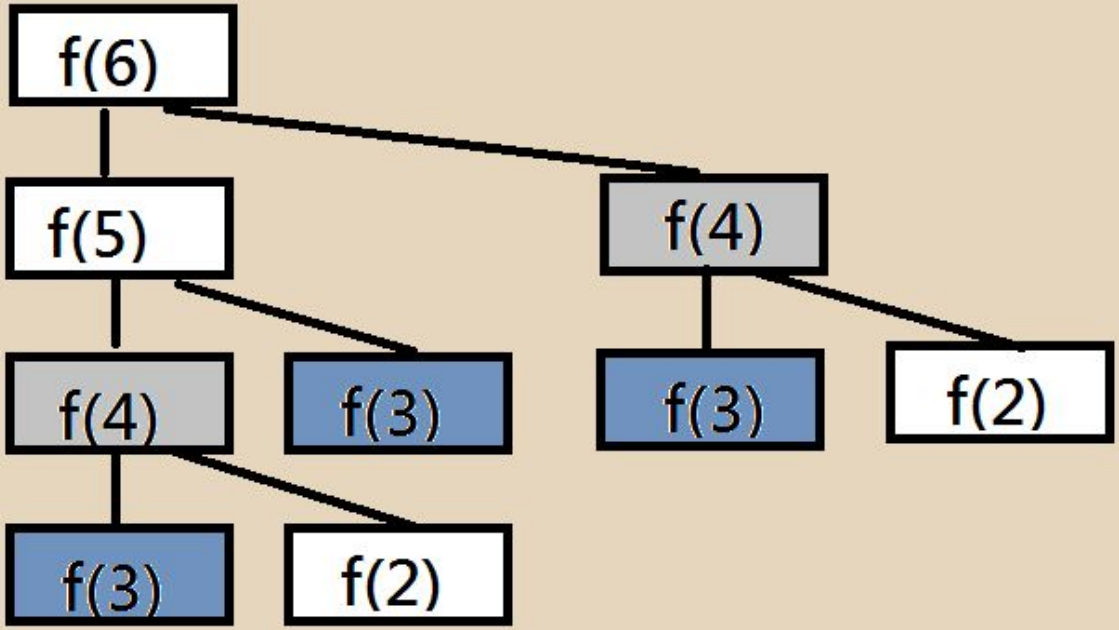
Time complexity?

O(x^2), where is the time “wasted”?

Fibonacci sequence

How do we calculate f(6)?

Fibonacci sequence



Much computer time is wasted!

Fibonacci sequence

Not optimal to do calculations from start

Instead, we can memorize the intermediate result!

Memoization

```
int f(int x)
    if (x <= 2) return 1;
    else if f(x) is calculated return f(x);
    else
        return f(x) = f(x-1) + f(x-2)
```

Time complexitiy?

Memoization

```
int f(int x)
    if (x <= 2) return 1;
    else if f(x) is calculated return f(x);
    else
        return f(x) = f(x-1) + f(x-2)
```

Time complexity?

O(x). Faster.

Memoization

Why memorization works?

- Big problem can be divided into subproblems with smaller size
- Subproblems are then repeatedly asked and calculated.
- Memorization cuts multiple calculation (from multtple query) into one

Dynamic programming

Dynamic programming is the technique of optimizing that make uses of memorization.

Consists of:

- State: Description of the (sub)problem
- Transition formula: How states relate

Grid Path Counting

Pure counting:

```
DFS(int x, int y){
     if (x == N && y == M) return 1; // base case
    else if (x, y) is not blocked
          return DFS(x+1, y) + DFS(x, y+1);
     else return 0; // no path can go through
}
```

Grid Path Counting

Time complexity?

How is subproblems repeatedly calculated?

How can we solve it using DP?

Grid Path Counting

Notice that we only need to calculate DFS(x,y) once for every x and y

Memorize intermediate result!

Grid Path Counting

Recall how DP works:

- State: Description of the (sub)problem
- Transition formula: How states relate

For this problem, what is the state?

what is the transition formula?

Grid Path Counting

State: (x,y)--> number of paths from (x,y) to (N,M)

Transition formula:

(x,y) = 1 if (x == N && y == M)

0 if (x, y) is blocked

(x+1, y) + (x, y+1) else

Grid Path Counting

```
DFS(int x, int y){
     if (x == N && y == M) return 1;
    else if (x, y) is calculated return (x, y);
    else if (x, y) is not blocked
          return (x, y) = DFS(x+1, y) + DFS(x, y+1);
     else return (x, y) = 0;
}
```

Grid Path Counting

Time complexity?

We have O(NM) state
and each state requires O(1) transition

-->O(NM)

Grid Path Counting

We solved this problem in a top-down approach

- Asking big problem, big problem asks smaller problem...

But actually, we can solve this problem in a bottom-up approach!

- How?

Bottom-up approach

Define state (x, y): number of paths from (x,y) to (N,M)

transition: (x,y) --> (x-1, y) += (x,y)

(x, y-1) += (x,y)

Why it works?

Grid Path Counting

```
solve(){
    for every x, y a[x][y] = 0;
    a[N][M] = 1;
    for i from N to 1
        for j from M to 1
            a[i - 1][j] += a[i][j] if a[i][j] is not a block
            a[i + 1][j] += a[i][j] if a[i][j] is not a block
    output a[1][1];
}
```

Top-down vs Bottom-up

Top down: Recursion, reaches possible state only.

What is the problem of recursion?

Bottom-up: Iterative, reaches all state

In some definiton, not every state is reachable.

Dynamic programming

CAUTION!

DP is only applicable if the subproblems are dependent only by their size.

For example, If there are two options in D(3), and both options will call D(2), the option made cannot affect the result of D(2)

More Example

Easy Ones:

Matrix multiplication, Basic Knapsack,

Longest common subsequence

Difficult:

HKOJ 1222 Longest Increasing Subsequence, CF506A Mr. Kitayuta, The Treasure Hunter

Matrix Multiplication

Matrix is a rectangular array of numbers. Multiplication of two matrices, one with size n*p, and other one with size p*m, costs n*m*p calculation, and results in a n*m matrix.

What if we have three matrices to be multiplied?

Matrix Multiplication

Matrix is a rectangular array of numbers. Multiplication of two matrices, one with size n*p, and other one with size p*m, costs n*m*p calculation, and results in a n*m matrix.

What if we have three matrices to be multiplied?

Different order of multiplication results in a same matrix, but some orders may costs more calculation!

What if we have K matrices to be multiplied? K <= 100

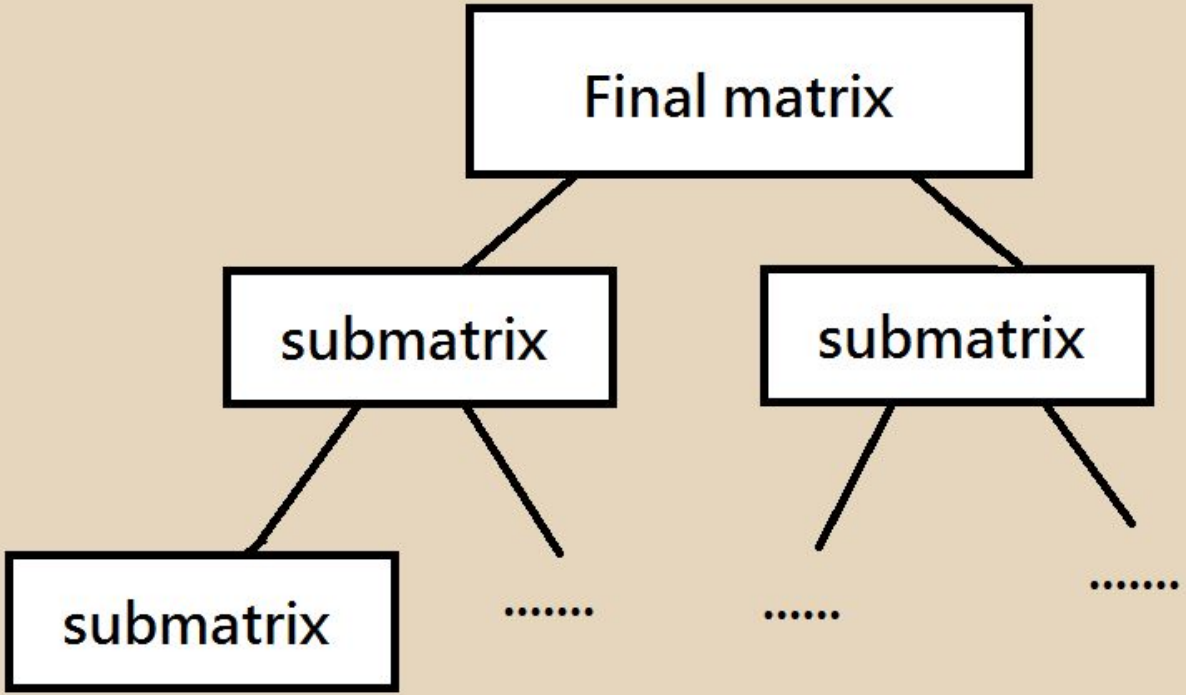
Matrix Multiplication

Problem: Given K (K <= 100) matrices, the ith matrix have dimension ai * bi, find out the least number of calculation needed.

(bi == a(i+1) for all i applicable)

How to resolve it into smaller subproblems?

Matrix Multiplication



Matrix Multiplication

Notice that the final step would be combining two matrices formed by the previous multiplication.

How to define such matrix?

(l, r): minimum cost to multiply matrices from l to r.

State is out, but what is the transition?

Matrix Multiplication

f(1, K) can be formed by:
multiply matrix 1 and matrix 2..K, or
multiply matrix 1..2 and matrix 3..K, or
multiply ....

Matrix Multiplication

Transition formula:

f(l, r) = 0 if (l == r)

min(p within l to r-1: f(l, p) + f(p+1, r) +

a[l] * b[r] * b[p]) else

Matrix Multiplication

```
int multiply(int l, int r){
    if (l == r) return 0;
    else if (l, r) is calculated return (l, r);
    else
        let (l, r) = inf
        for (k from l to r-1)
            (l, r) = min((l, r), multiply(l, k) +
                multiply(k+1, r) + a[l] * b[r] * b[k]
}
```

Matrix Multiplication

Time Complexity:

Number of state: O(K^2)
Transition per state: O(K)

total: O(K^3)

Matrix Multiplication

What if we want to know the actual order of multiplication?

Get another array for the optimal transition for state(l, r)!

Matrix Multiplication

What if we want to know the actual order of multiplication?

Get another array for the optimal transition for state(l, r)!

f(l, r) = 0 if (l == r)

min(p within l to r-1: f(l, p) + f(p+1, r) +

a[l] * b[r] * b[p]) else

d(l, r) = such optimal p for f(l, r)

Matrix Multiplication

```
Take_order(l, r){
    if (l == r) return;
    Take_order(l, d(l, r))
    Take_order(d(l, r) + 1, r)
    Output l, d(l, r), r;
}
```

Knapsack problem

In one day you bring V dollars to supermarket. In supermarket there are N items that you can buy. Each item costs ci dollars and gives you hi happiness. What is the maximal happiness you can gain? Remember you have V dollars only!

N, V <= 5000

Knapsack problem

Greedy?

Buy the items that hi/ci is highest, then second highest...

Try to create a case in which this algorithm fails

Brute force?

if all ci = 1, V = N / 2, in how many ways can we buy the product?

Knapsack problem

Remember how D&C works: for a big problem, try to decompose it to smaller problems.

Observation 1: the order of buying items does not matter.

Why?

Knapsack problem

Remember how D&C works: for a big problem, try to decompose it to smaller problems.

Observation 1: the order of buying items does not matter.

Why?

So we can decide, for each item, “buy” or “not buy”

Knapsack problem

But wait, number of possible solution is still O(2^N)!

Suppose we are considering item i, and we have j dollars left.

Buy item i: consider item i+1, have j-c[i] dollars left.

Do not buy item i: consider item i+1, have j dollars left.

Knapsack problem

But wait, number of possible solution is still O(2^N)!

Consider if you are in situation where you consider item i, and have j dollars left.

we want maximal happiness, therefore we want maximal happiness in this situation, too.

State can be (i, j): consider item i, have dollars left

What we record in (i, j) is the maximum happiness over (i, j)

Knapsack problem

If we have state, then what is the transition?

Remember: if we are in (i, j),

- we can choose: to buy, thus reaching (i+1, j - c[i])
  - of course j - c[i] >= 0
- OR not to buy, thus reaching (i+1, j)

We can build up a bottom-up approach!

Knapsack problem

```
solve(){
    for i from 1 to N+1, for j from 0 to V set a[i][j] = 0;
    for i from 1 to N
        for j from 0 to V
            a[i+1][j] = max(a[i+1][j], a[i][j]);
            if (j - c[i] >= 0)
                a[i+1][j-c[i]] = max(a[i+1][j-c[i]][j], a[i][j] +
                    h[i]);
    output maximum among a[N+1][0..V]
}
```

Knapsack problem

What if we prefer Top-down approach?

If we are in (i, j), we may:

arrive from (i-1, j) OR arrive from (i-1, j+c[i-1])

(i, j) = 0 if (i == 1)

max((i-1, j), (i-1, j+c[i-1])+h[i-1]) else

Knapsack problem

```
Knapsack(int i, int j){
    if (i == 1) return 0
    else if (j > V) return 0 // invalid case
    else if (i, j) is calculated return (i, j)
    else return (i, j) = max(Knapsack(i-1, j),
     Knapsack(i-1, j+c[i-1]) + h[i-1]);
}
```

Knapsack problem

Time complexity:

Number of state = O(NV)

Transition per state = O(1)

Total = O(NV)

Knapsack problem

What if:

we can buy infinite number of item i?

OR we can buy pi number of item i?

OR we must buy some item first so that we gain access to other items?

To be covered in later sessions

Longest common subsequence

A subsequence of a string S is any string that is formed by removing some(possibly none) characters of S.

For example, ab, abc, and ac are all subsequences of string “abc”

Given two strings S and T, find out their longest common subsequence

common: equal

Longest common subsequence

|S|, |T| <= 2000

e.g. “abcde” and “acebd” have a longest common subsequence of length 3 (“ace” or “acd” or “abd”)

Longest common subsequence

if we know the LCS length for |S| from 1 to i-1, |T| from 1 to j-1, what can we do?

if S[i] == T[j], it is always better to extend it,

    therefore (i+1, j+1) = max(itself, (i, j) + 1);

else, we can either form longer LCS with:

    Matching S[i] to T[k], where k > j, OR

    Matching T[j] to S[k], where k > i

Longest common subsequence

Hence (i+1, j) = max(itself, (i, j));

(i, j+1) = max(itself, (i, j));

Base case?

(1,1) = 0

Answer?

(|S|+ 1, |T|+ 1)

Longest common subsequence

Time complexity:

Number of state: O(|S||T|)

Transition per state: O(1)

--> O(|S||T|)

Longest increasing subsequence

Given an array a[], find out the longest increasing subsequence

subsequence: same as that of string

increasing: for the subsequence b[], b[i] < b[i+1] for all applicable i

e.g. the LIS of {3, 4, 2, 9, 1, 9, 6, 7} is {3, 4, 6, 7}

Longest increasing subsequence

Easy DP:

f(i) is the LIS of (1..i) that ends in i

f(i) = 0 if i == 1

max(f(k) + 1, k from 1 to i -1, a[i] > a[k]) else

Time complexity?

Longest increasing subsequence

Time complexity?

Number of state = O(N)

Transition per state = O(N)

total = O(N^2), which will TLE for N = 50000

How can we do better?

Longest increasing subsequence

Let S[j] be the smallest last element of a LIS of length j

Initially S[j] = {} and the LIS is 0 (empty)

We scan the array once and update S[] per element.

we want each S[j] to be as small as possible

Why?

Longest increasing subsequence

For each element, consider two case

--> It can possibly form a new LIS

--> Insert it into S. Now LIS is one unit longer.

--> It cannot form a new LIS

--> try replacing some element in S with this (smaller) number

Longest increasing subsequence

Case (1): Let the LIS currently is x, we know that S[x] is the smallest element of LIS of length x, therefore if a[i] > S[x], we can insert a[i] into S[x+1] and a longer LIS formed

Case (2): Find out the smallest element that is larger than a[i] in S, and replace it to a[i].

Why we cannot replace all element larger than a[i] in S?

Longest increasing subsequence

Time complexity:

Number of scan: O(N)

Finding smallest element >= a[i] : O(N)

Still O(N^2), how to improve?

Longest increasing subsequence

Time complexity:

Number of scan: O(N)

Finding smallest element using binary search >= a[i] :

O(lg N)

O(N lg N), can finish job in 1 second even if N = 200000

Mr. Kitayuta, The Treasure Hunter

http://codeforces.com/contest/506/problem/A

What is the subproblem?

What is the transition formula?

What is the time complexity?

Further Reading

Topcoder tutorial about DP

http://community.topcoder.com/tc?module=Static&d1=features&d2=040104

Codeforces further optimization of DP

(some may be covered in DP(II) and DP(III))

http://codeforces.com/blog/entry/8219

If we have time

HKOJ 1058 The Triangle II

HKOJ 1053 Longest Common substrings

HKOJ 1221 Falling Gift Game

CF 498 B Name that Tune

CF 480 D Parcels

FB HackerCup R1 C Winning at Sports

Reference

Wikipedia

Past OI DP(I) by Kelly Choi, 09

Topcoder editorial

Stackoverflow questions about DP

Codeforces