2017 SCSC Workshop

Sogang univ.
System modeling & Optimization Lab
Sangdurk Han

Dynamic Programming

Dynamic Programming

- 큰 문제의 해답에 작은 문제의 해답이 포함되어 있고, 이를 재귀 호출 알고리즘으로 구현하면 지나친 중복이 발생하는 경우에 이 재귀적 중복을 해결하는 방법

- 일반적으로 최적화 문제에 적용

- 잦은 출제, 많이 풀어보자

피보나치 수열

- N번째 피보나치 수를 구해보자.

  $F(0) = 0,\ F(1) = 1$

  $F(N) = F(N-1) + F(N-2)$

  $0 \leq \mathrm{N} \leq 40$

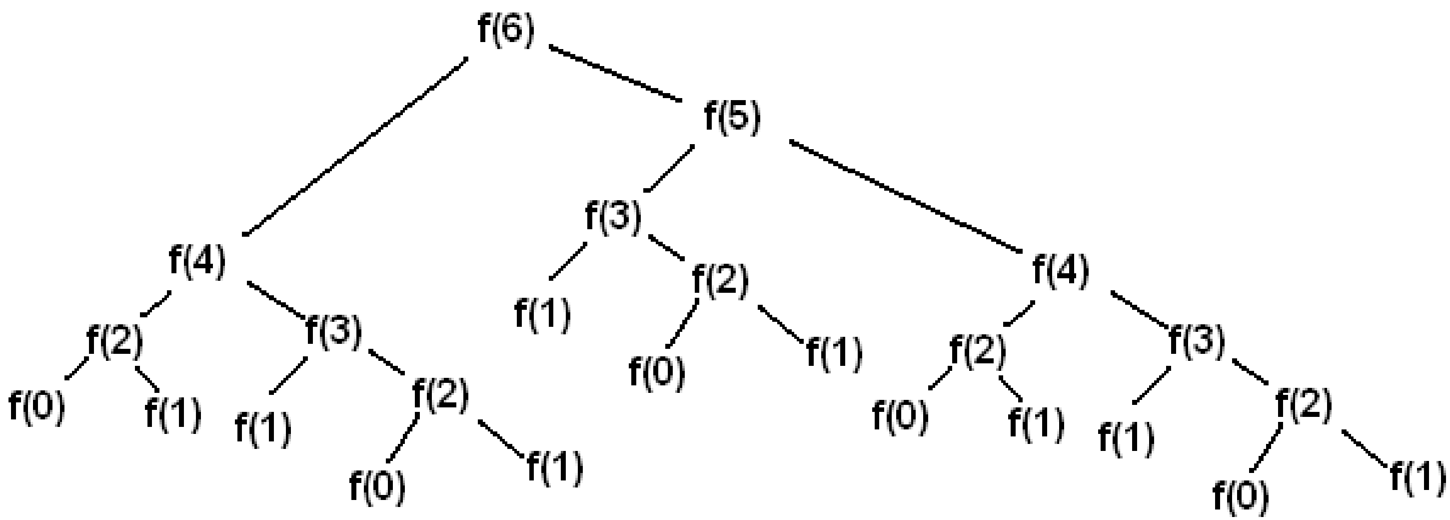
  Ex) 0, 1, 1, 2, 3, 5, 8, 13, 21, …

- N의 피보나치 수는 $N-1$의 피보나치 수와 $N-2$의 피보나치 수를 포함하고 있다.

피보나치 수열 (Recursive)

```c
#include <stdio.h>

int fibo(int N){
    if( N <= 1 ) return N;
    else fibo( N-1 ) + fibo( N-2 );
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", fibo(n));
}
```

피보나치 수열 (Recursive)



중복 된 연산이 많음
→ 적절한 방법으로 중복을 제거
(Dynamic Programming)

피보나치 수열 (Iterative)

```
#include <stdio.h>

int f[45];
int fibo(int N){
    f[0] = 0, f[1] = 1;
    for(int i=2; i<=N; i++)
        f[i] = f[i-1] + f[i-2];
    return f[N];
}

int main(){
    int n;
    scanf("%d", &n);
    printf("%d\n", fibo(n));
}
```

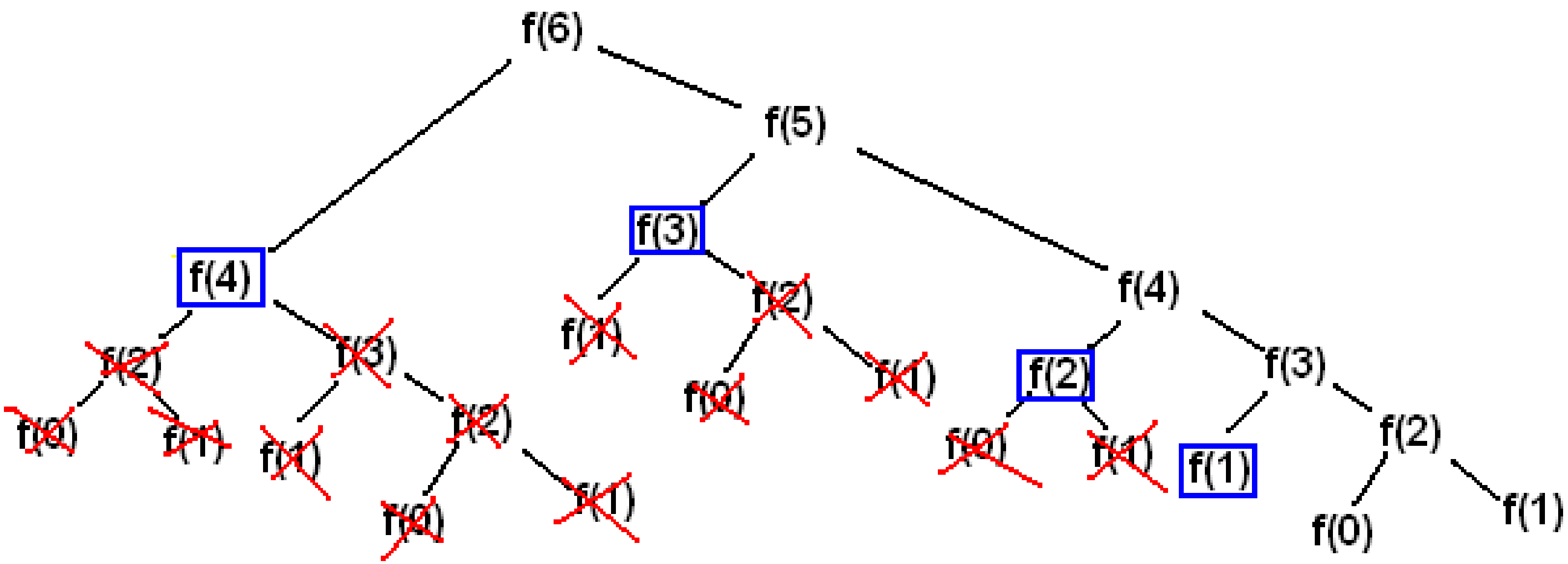
피보나치 수열 (Memoization)

```c
#include <stdio.h>

int f[N];

int fibo(int N){
    if( N <= 1 ) return N;
    else if( f[N] != 0 ) return f[N];
    else{
        f[N] = fibo( N-1 ) + fibo( N-2 );
        return f[N];
    }
}
```

재귀 호출을 사용하되 한 번 호출 된 것은 메모함으로써 중복 호출을 피함

메모하기 (Memoization)이라는 이름이 붙어있음

피보나치 수열 (Memoization)



중복 된 연산을 제거, O(N)

Dynamic Programming

- 1. 정의
- 2. 초기화
- 3. 점화식 (dp식)

피보나치 수열

- 1. 정의 d[i] = i번째 피보나치 수
- 2. 초기화 d[0] = 0, d[1] = 1
- 3. 점화식 (dp식) d[i] = d[i-1] + d[i-2]

Dynamic Programming

- 1. Iterative
- 2. Recursive
- 구현하기 편한 걸 사용하면 된다.

숫자 삼각형

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

- 맨 위의 숫자에서 한 번에 한 칸씩 아래로 내려가 맨 아래 줄까지 닿는 경로 중 숫자의 합이 최대인 값을 찾아보자.

- $1 \leq n \leq 1000$

Solution 1

- 모든 경우의 수를 확인
- $O(\ 2^n\ )$

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

Solution 2

```
        7
      3   8
    8   1   0
  2   7   4   4
4   5   2   6   5
```

1. 정의 d[i][j] = (0, 0)에서 (i, j)까지 최대 경로
2. 초기화 d[0][0] = a[0][0]
3. 점화식 d[i][j] = max(d[i-1][j-1], d[i-1][j]) + a[i][j]

Code

```
int a[N][N], d[N][N];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++)
        for (int j = 0; j <= i; j++)
            scanf("%d", &a[i][j]);

    d[0][0] = a[0][0];
```

Code

```
    for (int i = 1; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            if (j > 0)
                d[i][j] = max(d[i - 1][j - 1], d[i - 1][j]) + a[i][j];
            else
                d[i][j] = d[i - 1][j] + a[i][j];
        }
    }

    int ans = 0;
    for (int i = 0; i <= n; i++)
        ans = max(ans, d[n - 1][i]);
    printf("%d\n", ans);
}
```

시간복잡도

$$O(n * (n+1)/2) \Rightarrow O(n^2)$$

포도주 시식

- n잔의 포도주가 일렬로 나열되어 있다.
- 왼쪽에서부터 오른쪽으로 순서대로 포도주를 마시려고 한다. 단, 포도주를 마시지 않고 지나갈 수 있다.
- 연속으로 놓여 있는 포도주 3잔을 모두 마실 수는 없다.
- 각각의 잔의 포도주의 양이 주어졌을 때

  $a[i] = i$번째 포도주 잔의 양
- 가장 많이 시식할 수 있는 포도주의 양
- $1 \leq n \leq 10000$

Solution

- 1. 정의 d[i][3]

- d[i][0] = i번째 포도주를 마시지 않을 경우
- d[i][1] = i번째 포도주를 시식하고 연속으로
  1잔의 포도주를 시식한 상태
- d[i][2] = i번째 포도주를 시식하고 연속으로
  2잔의 포도주를 시식한 상태

Solution

- 2. 초기화

- d[1][0] = 0   // 첫 번째 포도주를 시식 X

- d[1][1] = a[1] // 첫 번째 포도주를 시식

- d[1][2] = 0   // 불가능한 경우

Solution

- 3. 점화식 dp식
- d[i][0] = max(d[i-1][0], d[i-1][1], d[i-1][2])
- d[i][1] = d[i-1][0] + a[i]
- d[i][2] = d[i-1][1] + a[i]

- 시간복잡도

  $O(cn) = O(n)$

Code

```
int a[N], d[N][3];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++)
        scanf("%d", a + i);

    d[1][1] = a[1];

    for (int i = 2; i <= n; i++) {
        d[i][0] = max(d[i - 1][0], max(d[i - 1][1], d[i - 1][2]));
        d[i][1] = d[i - 1][0] + a[i];
        d[i][2] = d[i - 1][1] + a[i];
    }

    int ans = max(d[n][0], max(d[n][1], d[n][2]));
    printf("%d\n", ans);
}
```

Code

```
int a[N], d[N][3];

int get(int i, int j) {
    if (i == 1) {
        if (j == 1) return a[1];
        else return 0;
    }
    else if (d[i][j] != -1)
        return d[i][j];
    else { // i >= 2
        if (j == 0)
            d[i][j] = max(get(i - 1, 0), max(get(i - 1, 1), get(i - 1, 2)));
        if (j == 1)
            d[i][j] = get(i - 1, 0) + a[i];
        if (j == 2)
            d[i][j] = get(i - 1, 1) + a[i];
        return d[i][j];
    }
}
```

Code (Memoization)

```
int a[N], d[N][3];

int get(int i, int j) {
    if (i == 1) {
        if (j == 1) return a[1];
        else return 0;
    }
    else if (d[i][j] != -1)
        return d[i][j];
    else { // i >= 2
        if (j == 0)
            d[i][j] = max(get(i - 1, 0), max(get(i - 1, 1), get(i - 1, 2)));
        if (j == 1)
            d[i][j] = get(i - 1, 0) + a[i];
        if (j == 2)
            d[i][j] = get(i - 1, 1) + a[i];
        return d[i][j];
    }
}
```

Code (Memoization)

```
int main() {
    int n;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", a + i);

    for (int i = 1; i <= n; i++)
        for (int j = 0; j < 3; j++)
            d[i][j] = -1; // 초기화

    int ans = max(get(n, 0), max(get(n, 1), get(n, 2)));
    printf("%d\n", ans);
}
```

RGB 거리

- RGB거리에 사는 사람들은 집을 빨강, 초록, 파랑중에 하나로 칠하려고 하고 이웃과 서로 다른 색의 집으로 칠하려고 한다.
- 집들은 일렬로 나열되어 있고, 집 i의 이웃은 i-1과 $i+1$이다.
- 각 집을 빨강, 초록, 파랑으로 칠하는 비용이 각각 주어 졌을 때, 모든 집을 칠할 때 드는 비용의 최솟값을 구해보자.
- $1 \le n \le 1000$

RGB 거리

Input

3

36 40 83

49 60 57

13 89 99

Output

36 + 57 + 13 = 96

Solution

- 1. 정의

d[i][0] = i번째 집을 빨강으로 칠했을 경우

d[i][1] = i번째 집을 초록으로 칠했을 경우

d[i][2] = i번째 집을 파랑으로 칠했을 경우

- 2. 초기화

d[1][0] = r[1]

d[1][1] = g[1]

d[1][2] = b[1]

Solution

- 3. 점화식 dp식

  d[i][0] = max(d[i][1], d[i][2]) + r[i]

  d[i][1] = max(d[i][0], d[i][2]) + g[i]

  d[i][2] = max(d[i][0], d[i][1]) + b[i]

- 시간복잡도
- $O(n)$

Longest Common Subsequence (LCS)

- 문자열 s, t
- $s[0], \ldots, s[n-1]$
- $t[0], \ldots, t[m-1]$

- 공통 부분 문자열 길이의 최대값

- $1 \leq n, m \leq 1{,}000$

Example

- S: abgstysdy
- T: bgtysyqqah

- 공통 부분 문자열 = b, bs,

- 최대 길이: ?

Solution

- 1. 정의
- d[i][j] = s문자열은 i번째 까지 선택했고
  t문자열은 j번째 까지 선택

- 2. 초기화
  d[i][j] = 0
  $1 \le i \le n,\ 1 \le j \le m$

Solution

- 3. 점화식 dp식
- (1) if(s[i] == t[j])

  d[i][j] = d[i-1][j-1] + 1

ex) abcd

aacdd

- (2) if(s[i] != t[j])

  d[i][j] = max(d[i-1][j], d[i][j-1])

ex) abcd

aacdd

Code

```
char s[N], t[M];
int d[N][M], n, m;

for(int i=0; i<n; i++){
    for(int j=0; j<m; j++){
        if( s[i] == t[j] )
            d[i+1][j+1] = d[i][j] + 1;
        else
            d[i+1][j+1] = d[i][j+1], d[i+1][j] );
    }
}

printf("%d\n", d[n][m]);
```

Longest Increasing Subsequence. (LIS)

- 길이 n 수열 $a_0, a_1, \ldots, a_{n-1}$
- 증가 부분 수열 중 최장의 것의 길이

- $1 \leq n \leq 10^3$
- $1 \leq a_i \leq 10^6$

Example

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

- 최대 길이는 5

Solution

- 1. 정의
- d[i] = i번째까지 선택했을 때 최대부분 증가수열 길이

- 2. 초기화
- d[i] = 1
- $1 \leq i \leq n$

Solution

- 3. 점화식 dp식
- $1 \leq j < i$이고, $a_j < a_i$인 모든 j에 대해서

  d[i] = max(d[j] + 1)
- ex) 1, 3, 7, 4, 5

- 시간복잡도
- $O(n)$

Code

```
int main(){
    int n;
    scanf("%d", &n);
    for(int i=1; i<=n; i++)
        scanf("%d", a+i);

    for(int i=1; i<=n; i++) d[i] = 1;

    for(int i=1; i<=n; i++)
        for(int j=1; j<i; j++)
            if( a[j] < a[i] )
                d[i] = max( d[i], d[j] + 1);
    int ans = 0;
    for(int i=1; i<=n; i++)
        ans = max( ans, d[i]);
    printf("%d\n", ans);
}
```

Better Solution

- $O(nlogn)$
- Binary search를 이용

```
fill( d, d+n, INF );
for(int i=0; i<n; i++)
    (*lower_bound(d, d+n, a[i])) = a[i];
printf("%d\n", n-(int)(lower_bound(d, d+n, INF) - d));
```

Lower Bound

- 정렬된 배열로부터 이진탐색에 의해 특정 값이 들어 갈 수 있는 가장 작은 위치의 주소를 반환함.

```cpp
// lower_bound/upper_bound example
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main () {
  int myints[] = {10,20,30,30,20,10,10,20};
  vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
  vector<int>::iterator low,up;

  sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

  low=lower_bound (v.begin(), v.end(), 20); //          ^
  up= upper_bound (v.begin(), v.end(), 20); //                   ^

  cout << "lower_bound at position " << int(low- v.begin()) << endl;
  cout << "upper_bound at position " << int(up - v.begin()) << endl;

  return 0;
}
```

Output:

```
lower_bound at position 3
upper_bound at position 6
```

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	INF	INF	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	INF	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	INF	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	40	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	INF	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	70	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	INF	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	60	INF	INF

Table

- **10 20 40 30 70 50 60**
- **10 20 40 30 70 50 60**

i	0	1	2	3	4	5	6
D[i]	10	20	30	50	60	INF	INF

- (int)((lower_bound(d, d+n, INF)-d) => 5

Better Solution

```
int main(){
    int n;
    scanf("%d", &n);
    for(int i=0; i<n; i++)
        scanf("%d", a+i);
    fill( d, d+n, INF );
    for(int i=0; i<n; i++)
        (*lower_bound(d, d+n, a[i])) = a[i];
    printf("%d\n", n-(int)(lower_bound(d, d+n, INF) - d));
}
```

Q&A