LEC15 Recursive Algorithms

4.4 https://github.com/GUMI-21/MIT6.006_note

How to Solve An Algorithms Problem(Review)

- Reduce to a problem you already know(Use data structure or algorithm)

Search Data Structures

- Array, Linked List, Dynamic Array, Sorted Array, Direct-Access Array, Hash Table, AVL Tree, Binary Heap

Sort Algorithms

- Insertion Sort, Selection Sort, Merge Sort, Counting Sort, Radix Sort, AVL Sort, Heap Sort

Graph Algorithms

- Breadth First Search, DAG Relaxation (DFS + Topo), Dijkstra, Bellman-Ford, Johnson

Design your own recursive algorithm

Constant-sized program to solve arbitrary input
Need looping or recursion, analyze by induction
Recursive function call: vertex in a graph, directed edge from A → B if B calls A
– Dependency graph of recursive calls must be acyclic (if can terminate)
Classify based on shape of graph

Today: Dynamic Programming 1 (of 4)

- SRTBOT recursive alg
- DP ~ recursion + memoization

```
def f(subprob):
    if subprob in memo0:
        return memo[subprob]
    basecase
    recurse via relation
    memo[s]↑
```

How to solve a problem recursively (SRT BOT)

1. *Subproblem* definition
2. *Relate* subproblem solutions recursively
3. *Topological* order on subproblems (⇒ subproblem DAG)
4. *Base* cases of relation
5. *Original* problem solution via subproblem(s)
6. *Time* analysis

Expample: merge_sort(A) n=|A|

-Subproblem: S(i,j) = sorted array on A[i:j]
-Relate: S(i,j) = merge(S(i,m),S(m,j))
-Topo order: increasing j-i
-Base case: S(i,j) = []
-Original Problem: S(0,n)
-Time: T(n) = 2T(n/2) + O(n) = O(nlogn) n = j-i

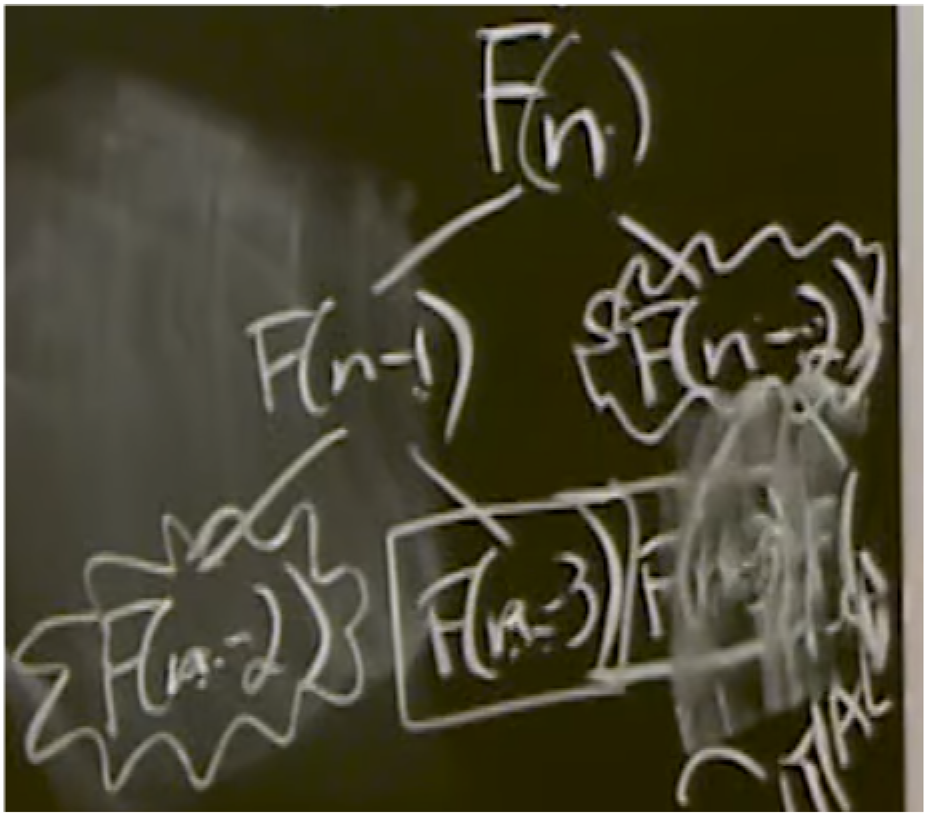
Fibonacci numbers: given n, compute $F_n = F_{n-1} + F_{n-2}, F_1 = F_2 = 1$

-Subproblem: F(i) = F_i 1<= i <=n
-Relate: F(i) = F(i-1) + F(i-2)
-Topo order: increasing i, for i = 1,2,....,n
-Base Case: F_1=F_2=1
-Origin: F(n)
-Time: T(n) = T(n-1) + T(n-2) + 1 additions > F_n ~ exponential, is bad

Dynamic programming

Big Idea: Memoization

remember & re-use solutions to subproblems



every F(i) need compute only once.

-maintain dictionary(daa or hash table) mapping subproblems-solutions. *memo*

-recursive function returns stored solution, or if doesn't exist, compute & store it.

=>then Fib .time = n-2 additions(n-bit)

-then on a w-bit machine

w-bit addition O(1) time, $O(\lceil n/w \rceil * n)$ = $O(n^2/w)$

-Time <= $\sum_{subproblems}(relate\ nonrecursive\ work)$

assume all recursive call is free

DAG shortest paths:

given DAG G & vertex s

-subproblems: delta(s,v) for each v |V| subproblems

-Relate: $\delta(s,v) = min\{\delta(s,u) + w(u,v) | u \in Adj^{-}(v)\} \cup \{\infty\}$

-Topo order: topo order of G(DAG)

-Base: $\delta(s,s) = 0$

-Original: all subproblems

-Time: $\sum_{v \in V}(O(1 + Adj^{-}(v))) = O(|V| + |E|)$

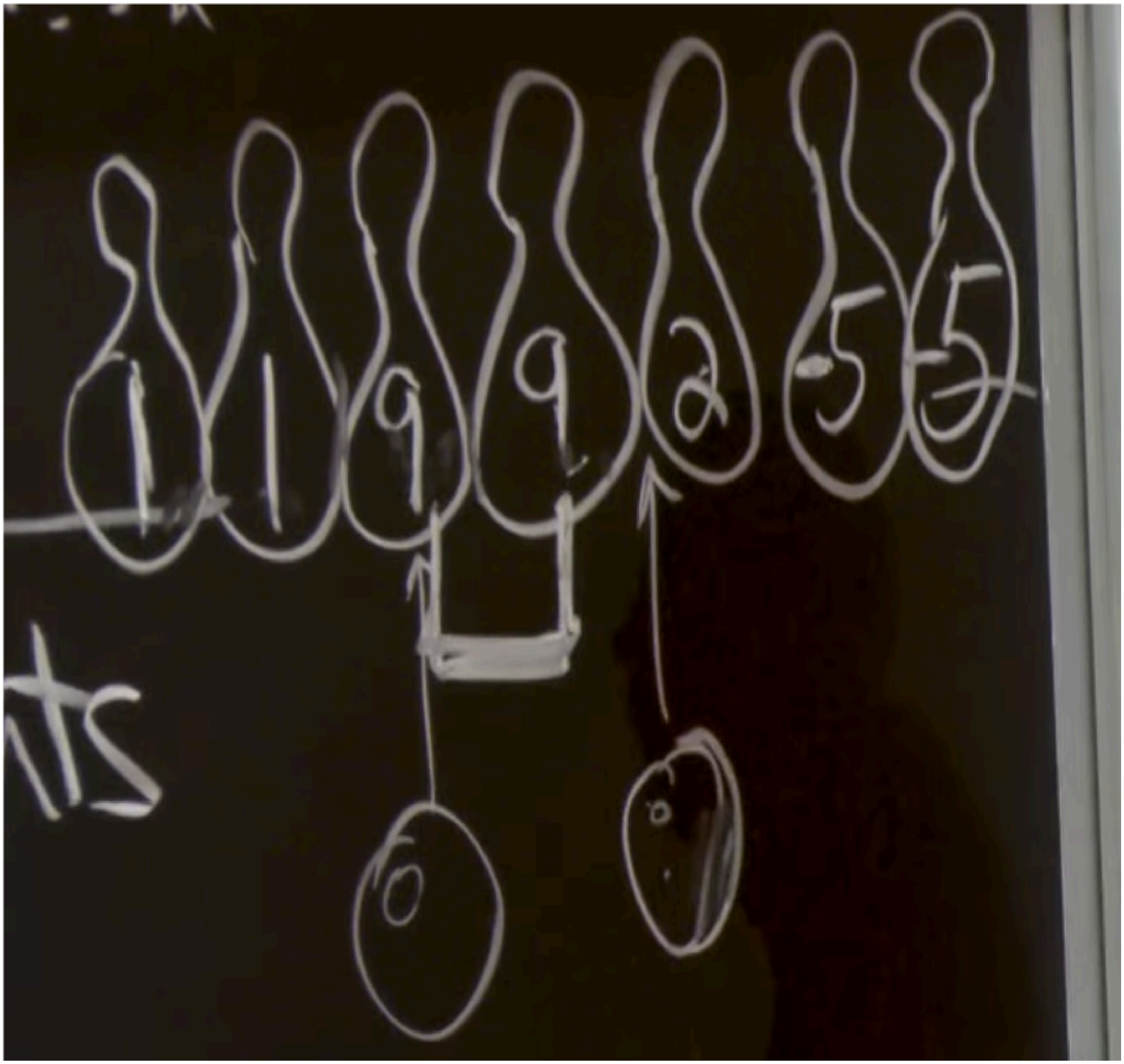
Bowling Game

given n pins O*i{0,....,n-1} in a line, you only can pull down 1/2/0 pins one time.*

-pin i has value V_i

-hit 1 pin i get V_i points

-hit 2 pins get $V_iV{i+1}$ points
-goal: max.source



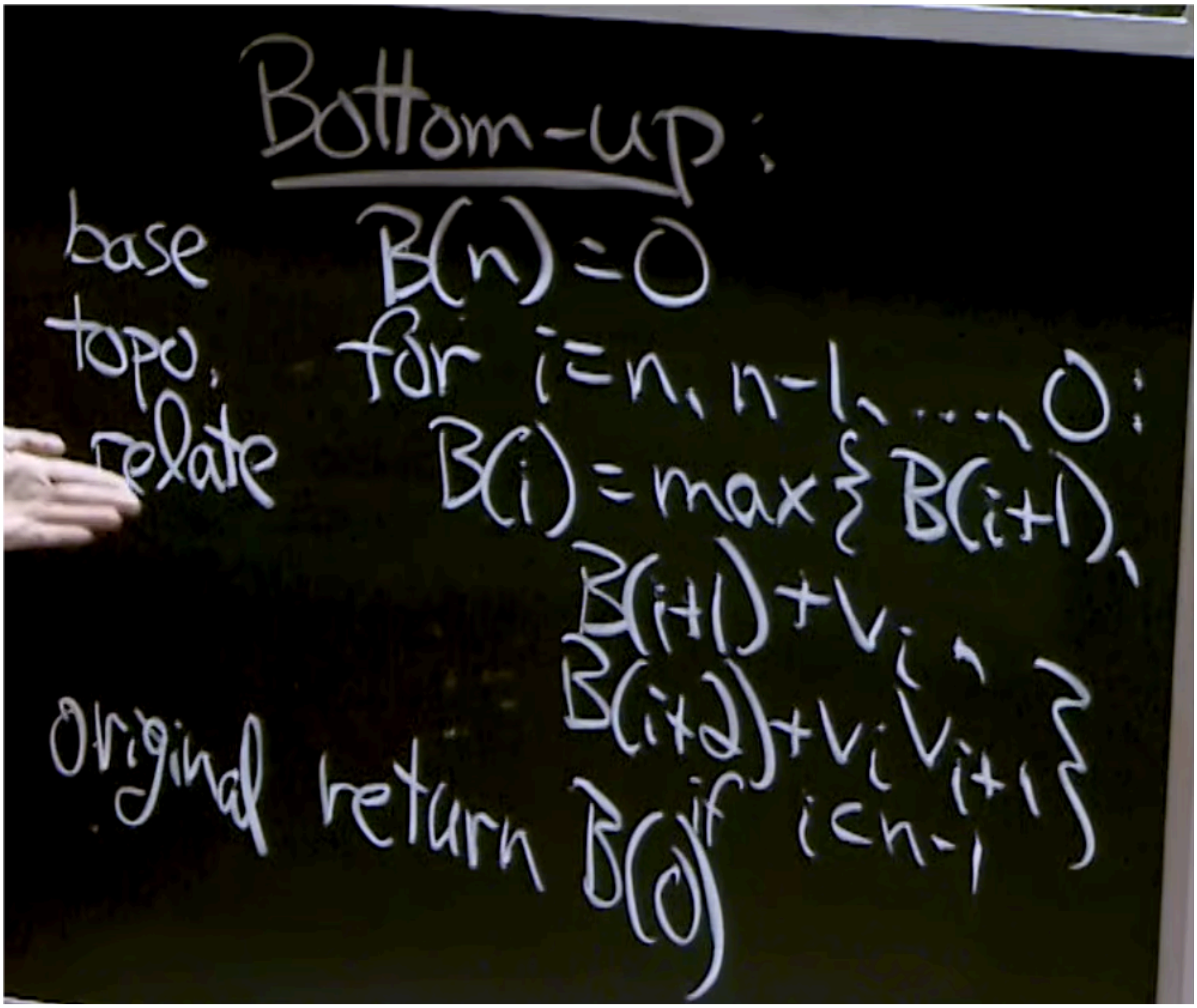
The input is a sequence of numbers.
Sub-problems design:
a trick: If input is a sequence x, good subproblems are:
-prefixes x[:i] $\theta(n)$
-suffixed x[i:] $\theta(n)$
-substrings x[i:j] $\theta(n^2)$

Bowling DP

-subproblems: B(i) = max-score possible tarting with pins i,i+1,....,n-1
-original problem: B(0)
-Relate: B(i) = max{B(i+1), B(i+1)+v_i,, B(i+2) + $v_i \cdot v_{i+1}$ $\theta(1)$

-Topo: decreasing i , for i=n,n-1.....,0
-Base: B(n) = 0
Time: $\theta(n) \cdot \theta(1) = \theta(n)$

- *Buttom-up DP*
  B(n) = 0, for i=n,n-1,....,0:
  B(i) = max{B(i+1), B(i+1) +v_i, B(i+2)+v_iv_i+1}
  if i<n-1
  return B(0)



DP~ local brute force