

## Lecture 15: Recursive Algorithms

### How to Solve an Algorithms Problem (Review)

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

Search Data Structures	Sort Algorithms	Graph Algorithms
Array	Insertion Sort	Breadth First Search
Linked List	Selection Sort	DAG Relaxation (DFS + Topo)
Dynamic Array	Merge Sort	Dijkstra
Sorted Array	Counting Sort	Bellman-Ford
Direct-Access Array	Radix Sort	Johnson
Hash Table	AVL Sort	
AVL Tree	Heap Sort	
Binary Heap		

- 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  $A \rightarrow B$  if  $B$  calls  $A$
- Dependency graph of recursive calls must be acyclic (if can terminate)
- Classify based on shape of graph

Class	Graph
Brute Force	Star
Decrease & Conquer	Chain
Divide & Conquer	Tree
Dynamic Programming	DAG
Greedy/Incremental	Subgraph

- Hard part is thinking inductively to construct recurrence on subproblems
- How to solve a problem recursively (**SRT BOT**)
  1. **Subproblem** definition
  2. **Relate** subproblem solutions recursively
  3. **Topological order** on subproblems ( $\Rightarrow$  subproblem DAG)
  4. **Base** cases of relation
  5. **Original** problem solution via subproblem(s)
  6. **Time** analysis

## Merge Sort in SRT BOT Framework

- Merge sorting an array  $A$  of  $n$  elements can be expressed in SRT BOT as follows:
    - Subproblems:  $S(i, j) = \text{sorted array on elements of } A[i : j] \text{ for } 0 \leq i \leq j \leq n$
    - Relation:  $S(i, j) = \text{merge}(S(i, m), S(m, j))$  where  $m = \lfloor (i + j)/2 \rfloor$
    - Topo. order: Increasing  $j - i$
    - Base cases:  $S(i, i + 1) = [A[i]]$
    - Original:  $S(0, n)$
    - Time:  $T(n) = 2T(n/2) + O(n) = O(n \lg n)$
  - In this case, subproblem DAG is a tree (divide & conquer)
- 

## Fibonacci Numbers

- Suppose we want to compute the  $n$ th Fibonacci number  $F_n$
- Subproblems:  $F(i) = \text{the } i\text{th Fibonacci number } F_i \text{ for } i \in \{0, 1, \dots, n\}$
- Relation:  $F(i) = F(i - 1) + F(i - 2)$  (definition of Fibonacci numbers)
- Topo. order: Increasing  $i$
- Base cases:  $F(0) = 0, F(1) = 1$
- Original prob.:  $F(n)$

```

1 def fib(n):
2     if n < 2: return n                      # base case
3     return fib(n - 1) + fib(n - 2)          # recurrence

```

- Divide and conquer implies a tree of **recursive calls** (draw tree)
- Time:  $T(n) = T(n - 1) + T(n - 2) + O(1) > 2T(n - 2)$ ,  $T(n) = \Omega(2^{n/2})$  exponential... :(
- Subproblem  $F(k)$  computed more than once! ( $F(n - k)$  times)
- Can we avoid this waste?

## Re-using Subproblem Solutions

- Draw subproblem dependencies as a DAG
- To solve, either:
  - **Top down:** record subproblem solutions in a memo and re-use (**recursion + memoization**)
  - **Bottom up:** solve subproblems in topological sort order (usually via loops)
- For Fibonacci,  $n + 1$  subproblems (vertices) and  $< 2n$  dependencies (edges)
- Time to compute is then  $O(n)$  additions

```

1 # recursive solution (top down)
2 def fib(n):
3     memo = {}
4     def F(i):
5         if i < 2: return i                      # base cases
6         if i not in memo:                      # check memo
7             memo[i] = F(i - 1) + F(i - 2)    # relation
8         return memo[i]
9     return F(n)                                # original

1 # iterative solution (bottom up)
2 def fib(n):
3     F = {}
4     F[0], F[1] = 0, 1                         # base cases
5     for i in range(2, n + 1):                  # topological order
6         F[i] = F[i - 1] + F[i - 2]            # relation
7     return F[n]                                # original

```

- A subtlety is that Fibonacci numbers grow to  $\Theta(n)$  bits long, potentially  $\gg$  word size  $w$
- Each addition costs  $O(\lceil n/w \rceil)$  time
- So total cost is  $O(n\lceil n/w \rceil) = O(n + n^2/w)$  time

## Dynamic Programming

- Weird name coined by Richard Bellman
    - Wanted government funding, needed cool name to disguise doing mathematics!
    - Updating (dynamic) a plan or schedule (program)
  - Existence of recursive solution implies decomposable subproblems<sup>1</sup>
  - Recursive algorithm implies a graph of computation
  - Dynamic programming if subproblem dependencies **overlap** (DAG, in-degree > 1)
  - “Recurse but re-use” (Top down: record and lookup subproblem solutions)
  - “Careful brute force” (Bottom up: do each subproblem in order)
  - Often useful for **counting/optimization** problems: almost trivially correct recurrences
- 

## How to Solve a Problem Recursively (SRT BOT)

1. **Subproblem** definition    subproblem  $x \in X$ 
  - Describe the meaning of a subproblem **in words**, in terms of parameters
  - Often subsets of input: prefixes, suffixes, contiguous substrings of a sequence
  - Often record partial state: add subproblems by incrementing some auxiliary variables
2. **Relate** subproblem solutions recursively     $x(i) = f(x(j), \dots)$  for one or more  $j < i$
3. **Topological order** to argue relation is acyclic and subproblems form a DAG
4. **Base cases**
  - State solutions for all (reachable) independent subproblems where relation breaks down
5. **Original problem**
  - Show how to compute solution to original problem from solutions to subproblem(s)
  - Possibly use parent pointers to recover actual solution, not just objective function
6. **Time analysis**
  - $\sum_{x \in X} \text{work}(x)$ , or if  $\text{work}(x) = O(W)$  for all  $x \in X$ , then  $|X| \cdot O(W)$
  - $\text{work}(x)$  measures **nonrecursive** work in relation; treat recursions as taking  $O(1)$  time

<sup>1</sup>This property often called **optimal substructure**. It is a property of recursion, not just dynamic programming

## DAG Shortest Paths

- Recall the DAG SSSP problem: given a DAG  $G$  and vertex  $s$ , compute  $\delta(s, v)$  for all  $v \in V$
  - Subproblems:  $\delta(s, v)$  for all  $v \in V$
  - Relation:  $\delta(s, v) = \min\{\delta(s, u) + w(u, v) \mid u \in \text{Adj}^-(v)\} \cup \{\infty\}$
  - Topo. order: Topological order of  $G$
  - Base cases:  $\delta(s, s) = 0$
  - Original: All subproblems
  - Time:  $\sum_{v \in V} O(1 + |\text{Adj}^-(v)|) = O(|V| + |E|)$
  - DAG Relaxation computes the same min values as this dynamic program, just
    - step-by-step (if new value  $<$  min, update min via edge relaxation), and
    - from the perspective of  $u$  and  $\text{Adj}^+(u)$  instead of  $v$  and  $\text{Adj}^-(v)$
- 

## Bowling

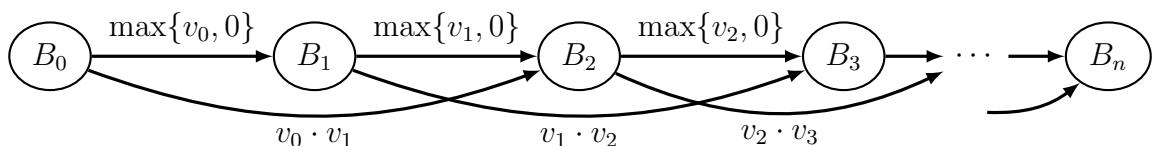
- Given  $n$  pins labeled  $0, 1, \dots, n - 1$
- Pin  $i$  has **value**  $v_i$
- Ball of size similar to pin can hit either
  - 1 pin  $i$ , in which case we get  $v_i$  points
  - 2 adjacent pins  $i$  and  $i + 1$ , in which case we get  $v_i \cdot v_{i+1}$  points
- Once a pin is hit, it can't be hit again (removed)
- Problem: Throw zero or more balls to maximize total points
- Example:  $[-1, \boxed{1}, \boxed{1}, \boxed{1}, \boxed{9, 9}, \boxed{3}, \boxed{-3, -5}, \boxed{2, 2}]$

## Bowling Algorithms

- Let's start with a more familiar divide-and-conquer algorithm:
  - Subproblems:  $B(i, j) = \max\{v_m \cdot v_{m+1} + B(i, m) + B(m+2, j), B(i, m+1) + B(m+1, j)\}$  for  $0 \leq i \leq j \leq n$
  - Relation:
    - \*  $m = \lfloor (i+j)/2 \rfloor$
    - \* Either hit  $m$  and  $m+1$  together, or don't
    - \*  $B(i, j) = \max\{v_m \cdot v_{m+1} + B(i, m) + B(m+2, j), B(i, m+1) + B(m+1, j)\}$
  - Topo. order: Increasing  $j - i$
  - Base cases:  $B(i, i) = 0, B(i, i+1) = \max\{v_i, 0\}$
  - Original:  $B(0, n)$
  - Time:  $T(n) = 4T(n/2) + O(1) = O(n^2)$
- This algorithm works but isn't very fast, and doesn't generalize well (e.g., to allow for a bigger ball that hits three balls at once)

---

- Dynamic programming algorithm: use suffixes
  - Subproblems:  $B(i) = \max\{v_i + B(i+1), v_i \cdot v_{i+1} + B(i+2)\}$  for  $0 \leq i \leq n$
  - Relation:
    - \* Locally brute-force what could happen with first pin (original pin  $i$ ): skip pin, hit one pin, hit two pins
    - \* Reduce to smaller suffix and recurse, either  $B(i+1)$  or  $B(i+2)$
    - \*  $B(i) = \max\{B(i+1), v_i + B(i+1), v_i \cdot v_{i+1} + B(i+2)\}$
  - Topo. order: Decreasing  $i$  (for  $i = n, n-1, \dots, 0$ )
  - Base cases:  $B(n) = B(n+1) = 0$
  - Original:  $B(0)$
  - Time: (assuming memoization)
    - \*  $\Theta(n)$  subproblems  $\cdot \Theta(1)$  work in each
    - \*  $\Theta(n)$  total time
- Fast and easy to generalize!
- Equivalent to maximum-weight path in Subproblem DAG:



## Bowling Code

- Converting a SRT BOT specification into code is automatic/straightforward
- Here's the result for the Bowling Dynamic Program above:

```

1 # recursive solution (top down)
2 def bowl(v):
3     memo = {}
4     def B(i):
5         if i >= len(v): return 0           # base cases
6         if i not in memo:               # check memo
7             memo[i] = max(B(i+1),        # relation: skip pin i
8                               v[i] + B(i+1),    # OR bowl pin i separately
9                               v[i] * v[i+1] + B(i+2)) # OR bowl pins i and i+1 together
10        return memo[i]
11    return B(0)                      # original

1 # iterative solution (bottom up)
2 def bowl(v):
3     B = {}
4     B[len(v)] = 0                   # base cases
5     B[len(v)+1] = 0
6     for i in reversed(range(len(v))): # topological order
7         B[i] = max(B[i+1],          # relation: skip pin i
8                       v[i] + B(i+1),    # OR bowl pin i separately
9                       v[i] * v[i+1] + B(i+2)) # OR bowl pins i and i+1 together
10    return B[0]                      # original

```

---

## How to Relate Subproblem Solutions

- The general approach we're following to define a relation on subproblem solutions:
  - Identify a question about a subproblem solution that, if you knew the answer to, would reduce to “smaller” subproblem(s)
    - \* In case of bowling, the question is “how do we bowl the first couple of pins?”
  - Then locally brute-force the question by trying all possible answers, and taking the best
    - \* In case of bowling, we take the max because the problem asks to maximize
  - Alternatively, we can think of correctly guessing the answer to the question, and directly recursing; but then we actually check all possible guesses, and return the “best”
- The key for efficiency is for the question to have a small (polynomial) number of possible answers, so brute forcing is not too expensive
- Often (but not always) the nonrecursive work to compute the relation is equal to the number of answers we're trying

## TODAY: Dynamic Programming I (of 4)

- recursive algorithm paradigm: SRTBOT
- DP  $\approx$  recursion + memoization
  - subproblem design
  - relations via guessing
  - bottom-up DP
- examples
  - Fibonacci
  - DAG shortest paths
  - bowling

Last part of 6.006: DESIGN of new algorithms  
(vs. reducing to/modifying given algorithms)

- solving arbitrary problem size  $n$  using  $O(1)$ -size algorithm → need to repeat steps. via loops or recursion
- we'll focus on recursive view:
  - reveals directed graph of computation/dependency
  - easy to prove correct (via induction)
  - can convert to loops if desired (bottom-up)

# Recursive algorithm design paradigm: SRTBOT

- Subproblem definition  $f(a,b)$
- Relate subproblem solutions recursively  $= \dots f(a',b')$
- Topological order on subproblems to guarantee acyclic relation  $\begin{cases} \text{for } a=1, \dots, m \\ \text{for } b=n, \dots, 0 \end{cases}$
- subproblem/call DAG  $a \rightarrow b \Rightarrow b \text{ needs a}$
- Base cases of relation  $f(0,b) = \dots$
- Original problem: solve via subproblem(s)  $f(m,0)$
- Time analysis  $T(m,n)$

Example: merge-sort(A)

$$n = |A|$$

- Subproblems:  $S(i,j) = \text{sorted array on } A[i:j]$   $\Theta(n^2)$  subproblems
- Relate:  $S(i,j) = \text{merge}(S(i,m), S(m,j))$   $\left\lfloor \frac{i+j}{2} \right\rfloor$  time
- Topo. order: increasing  $j-i$
- Base case:  $S(i, i+1) = [A[i]]$
- Orig. prob.:  $S(0, n)$
- Time:  $T(n) = 2T(n/2) + O(n) = O(n \lg n)$   $\downarrow j-i$  here call DAG = tree

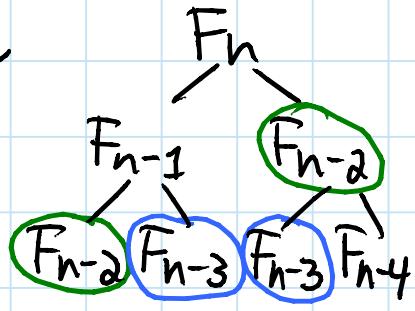
return  
here  
after  
next page

$$\leq \sum_{i=0}^{n-1} \sum_{j=i}^n \Theta(j-i) = \Theta(n^3) - \text{poor upper bound}$$

(most subproblems not computed)

Fibonacci numbers: given  $n$ , compute

$$F_n = F_{n-1} + F_{n-2}; F_1 = F_2 = 1$$



SRTBOT:

- Subproblems:  $F(k) = F_k$
- Relate:  $F(k) = F(k-1) + F(k-2)$
- Topo.: increasing  $k \sim$  for  $k=1, 2, \dots, n$
- Base:  $F(k) = 1$  for  $k=1, 2$
- Original:  $F(n)$
- Time:  $T(n) = T(n-1) + T(n-2) + 1$  additions  
 $= F_n \approx \varphi^n \rightarrow \frac{\sqrt{5}+1}{2} \approx 1.618$   
 $\geq 2T(n-2) \geq 2^{n/2}$  **EXONENTIAL** :-)

**BIG IDEA:**

Memoization: remember & re-use previously computed subproblem solutions

- maintain dictionary mapping subprobs.  $\rightarrow$  solutions
- recursive function returns stored solution,  
or if doesn't exist, computes & stores it
- $\Rightarrow$  only pay once for every subproblem
- $\Rightarrow$  time  $\leq \sum_{\text{subprob.}} \text{nonrecursive work in relation}$   
 $\leq \# \text{ subproblems} \cdot \text{nonrec. work in relation}$

$\Rightarrow F(n)$  costs  $\leq n \cdot 1$  additions  
 $\quad \quad \quad \# \text{ subprobs.} \uparrow \quad \quad \quad \uparrow \text{nonrec. work in relation}$

- but: numbers are  $O(n)$  bits long, potentially  $\gg w$
- $\Rightarrow O(n \lceil n/w \rceil) = O(n+n^2/w)$  time

# Dynamic Programming: (DP)

powerful algorithmic design technique  
especially for optimization problems (min/max)  
e.g. shortest paths

- conceptually simple:

DP  $\approx$  recursion + memoization

- base concepts are few but powerful

- difficult to master

- applying concepts is difficult

(algorithm design is hard)

- in these 4 lectures, will see many examples

- can seemingly brute-force (some) problems  
in polynomial time

→ as in Bellman-Ford! [L12]

## History: Richard E. Bellman (1920-1984)

"Bellman... explained that he invented the name 'dynamic programming' to hide the fact that he was doing mathematical research at RAND under a Secretary of Defense who 'had a pathological fear and hatred of the term, research.' He settled on the term 'dynamic programming' because it would be difficult to give a 'pejorative meaning' and because 'It was something not even a Congressman could object to.'"

[John Rust 2006]

## Generic DP algorithm (top-down):

memo = {}

def f(subprob):

if subprob not in memo:

    memo[subprob] = base or recursive relation

    return memo[subprob]

return original

DAG shortest paths: given DAG  $G$  & vertex  $s$ ,  
compute  $S(s, v)$  for all  $v \in V$

- Subproblems:  $S(s, v)$  for all  $v \in V$

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

- Topo. order = topo. order of  $G$

- Base:  $S(s, s) = \emptyset$

- Original: all subprobs.

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

- DAG relaxation computes the same min's.  
just step by step (if  $<$  min, update min)

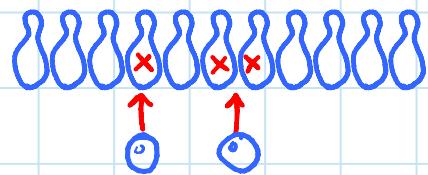
& from  $u$  &  $\text{Adj}^+(u)$  perspective

- recursion + memoization gives topo. order  
~ this actually does DFS in reversed graph  
(memo table marks visited nodes/subprobs.)  
& executes relation in finishing order  
= reversed finishing order in original graph

→ from 6.006 F18

Bowling: given  $n$  pins  $0, 1, \dots, n-1$

- pin  $i$  has value  $v_i$
- ball of similar size can hit
  - 1 pin  $i \rightarrow v_i$  points
  - OR 2 adjacent pins  $i$  &  $i+1 \rightarrow v_i \cdot v_{i+1}$  points
- goal: maximize points (any # rolls)
- intuitively: want to pair up high-value neighbouring pins, which may leave single pins...



inspired by Kayles  
[Henry Dudeney 1908]

-1 1 1 1 9 9 3 -3 -5 2 2  
 ↑↑↑↑ ↗↑↑ ↗

Divide & conquer algorithm:

- Subproblems:  $B(i, j) = \text{max. score starting with just pins } i, i+1, \dots, j-1$
- Relate:  $m = \lfloor (i+j)/2 \rfloor$   
 either bowl  $m$  &  $m+1$  or don't  

$$B(i, j) = \max \{ v_m \cdot v_{m+1} + B(i, m) + B(m+2, j), B(i, m+1) + B(m+1, j) \}$$

- Topo: increasing  $j-i$
- Base:  $j-i=0 \Rightarrow B(i, j)=0$  → in case  $v_i < 0$   
 $j-i=1 \Rightarrow B(i, j)=\max \{ v_i, 0 \}$

- Original:  $B(\emptyset, n)$
- Time: without memoization (call tree):
 
$$\begin{aligned} T(n) &= 4 T(n/2) + O(1) \\ &= O(n^2) \end{aligned}$$

- would get worse if we had e.g. different ball sizes

Subproblem design is the hard part

- trick: when given sequences, often use

- prefixes  $S[:i]$  for all  $i$

$\Theta(n)$  {prefer}

- suffixes  $S[i:]$  for all  $i$

$\Theta(n)$

OR - substrings  $S[i:j]$  for all  $i < j$

$\Theta(n^2)$

DP algorithm:

- Subproblems:  $B(i) = \max$  score starting with just pins  $i, i+1, \dots, n-1$   $\{\Theta(n)\}$

- Relate:  $B(i) = \max \{ v_i + B(i+1),$  - hit just  $i$   
if  $i+1 < n \rightarrow v_i \cdot v_{i+1} + B(i+2),$  - hit 2:  $i \& i+1$   
in case  $v_i < 0 \rightarrow B(i+1) \}$  - skip  $i$

- idea: brute force what could happen with first pin  $\rightarrow$  reduce to smaller suffix

- Topo.: decreasing  $i \sim$  for  $i = n, n-1, \dots, \emptyset$

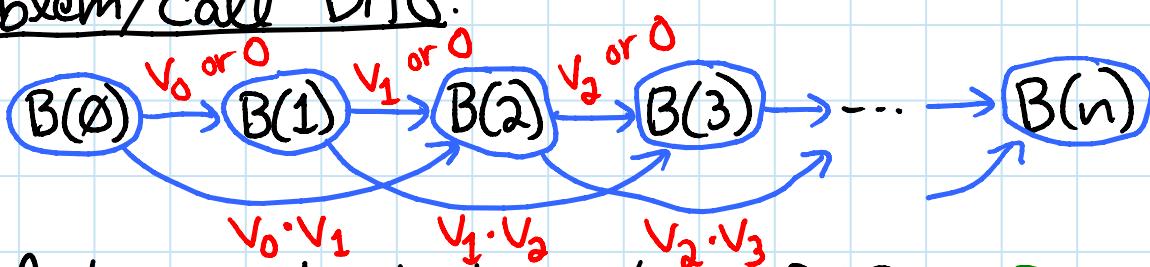
- Base:  $B(n) = \emptyset$

- Original:  $B(\emptyset)$

- Time:  $\Theta(n)$  subprobs.  $\cdot \Theta(1)$  nonrec. work  $= \Theta(n)$

- easy to generalize to different ball sizes

Subproblem/call DAG:



- solution = shortest path in DAG } not for Fibonacci
- Common if relation reduces to 1 subprob.

Relation trick: identify part of the subproblem solution that, if you knew, reduces the subproblem to smaller subproblem(s)

- locally brute force to find best choice
- guessing perspective:
  - allow "guessing" part of subprob. solution
  - assume relation algorithm makes correct guess
  - in analysis, multiply relation run. time by # possible guesses (brute force)
- key for efficiency: design subproblems so few possible choices/guesses to brute force

Bottom-up DP: for loop over topological order instead of recursion + memoization

- e.g.  $B(n) = \emptyset$
- for  $i = n-1, n-2, \dots, 0$ :
- $$B(i) = \max \{ v_i + B(i+1),$$

$$v_i \cdot v_{i+1} + B(i+2) \text{ if } i+1 < n,$$

$$B(i+1) \}$$

return  $B(\emptyset)$

original

- exactly same computations & order  
(unrolled recursion)
- simpler & practically faster when topological order is easy (for loops)
- time analysis also simpler