Lecture 18: Dynamic Programming I: Memoization, Fibonacci, Crazy Eights

Lecture Overview

- Fibonacci Warmup
- Memoization and subproblems
- Crazy Eights Puzzle
- Guessing Viewpoint

Readings

CLRS 15

Introduction to Dynamic Programming

- Powerful algorithm design technique, like Divide&Conquer.
- Creeps up when you wouldn't expect, turning seemingly hard (exponential-time) problems into efficiently (polyonomial-time) solvable ones.
- Usually works when the obvious Divide&Conquer algorithm results in an exponential running time.

Fibonacci Numbers

$$0, 1, 1, 2, 3, 5, 8, 13, \ldots$$

Recognize this sequence?

It's the *Fibonacci sequence,* described by the recursive formula:

$$F_0 := 0;\ \ F_1 := 1;$$
$$F_n = F_{n-1} + F_{n-2},\ \text{for all } n \geq 2.$$

Clearly, $F_n \leq 2F_{n-1} \leq 2^n$.

[In fact, the following is true for all $n \geq 1$:

$$F_n := \frac{\phi^n - (1-\phi)^n}{\sqrt{5}},$$

where $\phi = \frac{\sqrt{5}+1}{2}$ is the golde ratio.]

So we don't need more than n bits to represent F_n, but how hard is it to compute it?

Trivial algorithm for computing F_n:

```
naive_fibo($n$):
    if $n = 0$: return 0
    else if $n = 1$: return 1
    else: return naive_fibo($n-1$) + naive_fibo($n-2$).
```

Figure 1 shows how the recursion unravels.

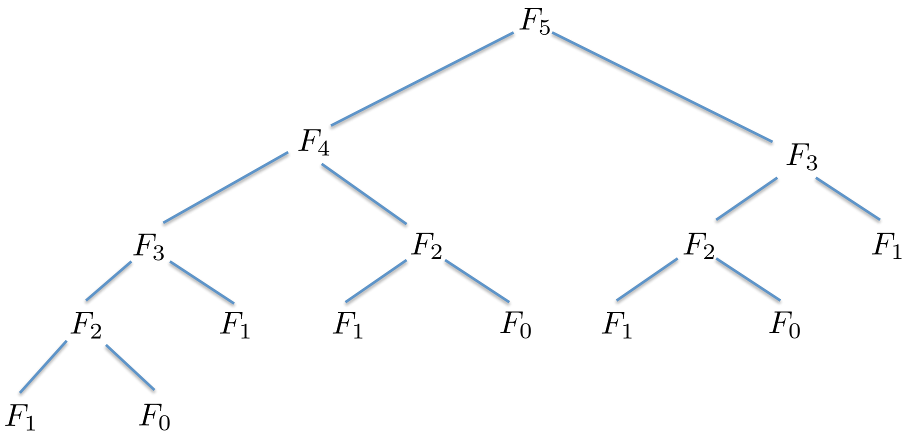


Figure 1: Unraveling the Recursion of the Naive Fibonacci Algorithm.

Runtime Analysis

Suppose we store all intermediate results in n-bit registers. (Optimizing the space needed for intermediate results is not going to change much.)

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + c \\ &\geq 2T(n-2) + c \\ &\geq \dots \\ &\geq 2^k T(n-2\cdot k) + c(2^{k-1} + 2^{k-2} + \ldots + 2 + 1) = \Omega(c2^{n/2}), \end{aligned}$$

where c is the time needed to add n-bit numbers. Hence $T(n) = \Omega(n2^{n/2})$.

EXPONENTIAL - BAD!

Problem with recursive algorithm:

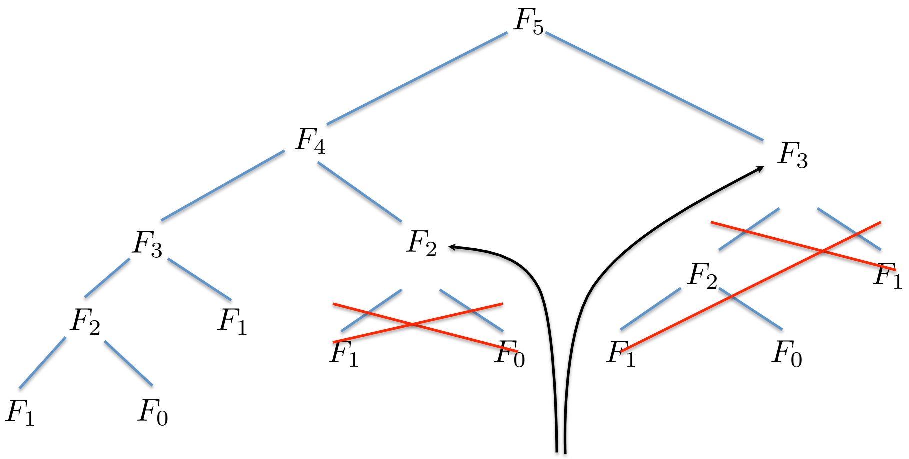
Computes $F(n-2)$ twice, $F(n-3)$ three times, etc., each time from scratch.

Improved Fibonacci Algorithm

Never recompute a subproblem $F(k)$, $k \leq n$, if it has been computed before. This technique of remembering previously computed values is called **memoization.**

Recursive Formulation of Algorithm:

```
memo = { }
fib($n$):
    if $n$ in memo: return memo[$n$]
    else if $n = 0$: return 0
        else if $n = 1$: return 1
        else: $f$ = fib($n-1$) + $\underbrace{\text{fib}(n-2)}_{\text{free of charge!}}$
        memo[$n$] = $f$
        return $f$
```



these values are already computed and stored in memo *when runtime processes these nodes of the recursion*

Figure 2: Unraveling the Recursion of the Clever Fibonacci Algorithm.

Runtime, assuming n-bit registers for each entry of memo data structure:

$$T(n) = T(n-1) + c = O(cn),$$

where c is the time needed to add n-bit numbers. So $T(n) = O(n^2)$.

[*Side Note*: There is also an $O(n \cdot \log n \cdot \log\log n)$- time algorithm for Fibonacci, via different techniques]

Dynamic Programming (DP)

* DP $\approx$ recursion + memoization (i.e. re-use)
* DP $\approx$ "controlled brute force"

DP results in an efficient algorithm, if the following conditions hold:

- the optimal solution can be produced by combining optimal solutions of subproblems;
- the optimal solution of each subproblem can be produced by combining optimal solutions of sub-subproblems, etc;
- the total number of subproblems arising recursively is polynomial.

Implementation Trick:

- Remember (<u>memoize</u>) previously solved "subproblems"; e.g., in Fibonacci, we memoized the solutions to the subproblems $F_0, F_1, \cdots, F_{n-1}$, while unraveling the recursion.
- if we encounter a subproblem that has already been solved, re-use solution.

Runtime $\approx \sharp$ of subproblems $\cdot$ time/subproblem

Crazy Eights Puzzle

Problem Formulation:

INPUT: a sequence of cards $c[\emptyset], c[1], \cdots, c[n-1]$, e.g., $7\heartsuit, 6\heartsuit, 7\diamondsuit, 3\diamondsuit, 8\clubsuit, J\spadesuit$;

QUESTION: the longest left-to-right "*trick subsequence*", i.e.

find $c[i_1], c[i_2], \cdots c[i_k]$ $(i_1 < i_2 < \cdots i_k)$
where $c[i_j]$ & $c[i_{j+1}]$ "<u>match</u>" for all $j = 1, \ldots, k$,
i.e. they have the same suit or rank or one has rank 8

In the above example, the longest trick is $7\heartsuit, 7\diamondsuit, 3\diamondsuit, 8\clubsuit, J\spadesuit$.

Algorithm for finding longest trick subsequence

- Let trick(i) = length of best trick starting at $c[i]$;
- **Can relate value of trick(i) with values of trick(j), for $j > i$, as follows:**

$$\text{trick}(i) := 1 + \max_{j>i \text{ s.t. } c[i] \text{ and } c[j] \text{ match}} \{\text{trick}(j)\}; \qquad (1)$$

- **Longest Trick**$= \max_{i:0 \leq i \leq n-1}\{\text{trick}(i)\}$;

- **Algorithm? Memoize!**
  Recursive Formulation of Algorithm:

```
memo={}
trick($i$):
        if $i$ in memo return memo[i]
        else if $i = n - 1$: return 1
        else:
            $f := 1 + \max_{j>i \text{ s.t. } c[i] \text{ and } c[j] \text{ match}}\{\text{trick}(j)\}$
            memo[$i$] = $f$
            return $f$
call trick(0)  /*this call will populate the memo array*/
return maximum value in memo
```

  Alternative "Bottom-Up" Formulation of Algorithm:

```
memo = {}
for $i = n - 1$ down to 0
        compute trick($i$) applying (1) to the values stored in memo[j], $j > i$
        store trick($i$) in memo[i]
return maximum value in memo
```

- **Runtime**

$$\begin{aligned}\text{time} &= \underbrace{\sharp\,\text{subproblems}}_{O(n)} \cdot \underbrace{\text{time/subproblem}}_{O(n)\text{ for going through max}}\\ &= O(n^2)\end{aligned}$$

- To find actual trick, trace through max's. Need some extra book-keeping, i.e. remembering for each i what j was selected by the max operator of Equation (1).

"Guessing" interpretation of DP

We can interpret recursion (1) as specifying the following:

> "To compute trick(i) all I need is to *guess* the next card in the best trick starting at i."

where *Guess* = try all possibilities.

For DP to work we need:
small $\sharp$ of subproblems + small $\sharp$ guesses per subproblem + small overhead to put solutions together

Then using memoization,

Runtime $\approx \sharp$ of subproblems $\times$ $\sharp$ guesses per subproblem $\times$ overhead.

In crazy eights puzzle: number of subproblems was n, the number of guesses per subproblem where $O(n)$, and the overhead was $O(1)$. Hence, the total running time was $O(n^2)$. [1]

In Fibonacci numbers: there were n subproblems, no guessing was required for each subproblem, ant the overhead was $O(n)$ (adding two n-bit numbers). So the overall runtime was $O(n^2)$.

---

[1] To be precise, we need $O(\log n)$ bits to store each value trick(i), since this is a number in $\{1, \ldots, n\}$. So the addition operation in (1) is an addition over $O(\log n)$-bit numbers, resulting in an overhead of $O(\log n)$. So, strictly speaking, the running time is $O(n^2 \log n)$. If n is small enough so that $\log n$ fits in a machine word, we get $O(n^2)$.