Greedy Algorithms and Dynamic Programming

Sep 17, 2020

Till now, we have seen problems that have some immediate polynomial time (eg, $O(n^3)$, $O(n^4)$) algorithms and we saw some tools like divide and conquer to make them more efficient.

For example $O(n^2) \longrightarrow O(n \log n)$

$O(n^3) \longrightarrow O(n)$

⋮

Now, we will see more examples where the obvious algorithm takes exponential time and the challange is to first design some polynomial time algorithm.

Towards, this, the two paradigms Greedy & DP are often helpful.

They will involve more clever and subtle ideas then what we have seen till now.

We will study Greedy & DP simultaneously.

pick up a problem and see whether Greedy works or not, whether DP works or not.

Greedy Algorithm

→ local optimality

→ near-sighted

→ don't care about long-run

Proof of correctness is important.

because correctness is not obvious in most cases.

we will see some basic format of how to argue correctness.

Dynamic Programming

→ Recursion with a better memory management.

or → a systematic approach to search through all possible solutions.

Problem 1 You are allowed to choose $\boxed{\text{five}}$ apples from a basket. You want to maximize the total weight of your apples.

Greedy approach woks here.

→ Choose heaviest, 2nd heaviest, ..., 5th heaviest

Problem 2 Total weight of the apples $\leq$ 2 kg.

Basket has apples of weight

→ 450 gms, 400 gms.

Greedy $4 \times 450 = 1800$ gms.

Alternate $5 \times 400 = 2000$ gms.

Problem 3

Subsequence Problem.

S_1 = a c b g a b a g b c a sequence

S_2 = c a b g c (subsequence)

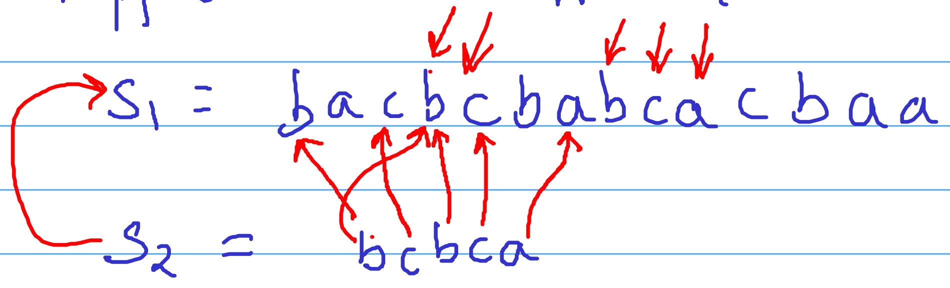
Que Given two sequences S_1 & S_2, whether S_2 is a subsequence of S_1

$|S_1| = n \quad |S_2| = p$

Approach 1 → search through all subsequences of length p and check if one of them is equal to S_2.

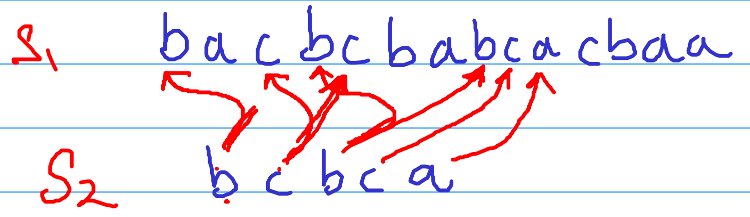
No. of such subsequences = $\binom{n}{p}$

Approach 2: match letter by letter



Match the current letter in S_2 with its first occurence you see in S_1 after the previous matching.

Argument for correctness



We want to argue that the greedy approach works. That is, if S_2 is a subsequence of S_1 then we should be able to see it by matching each subsequent letter of S_2 with its first occurence in S_1 after the previous matching.

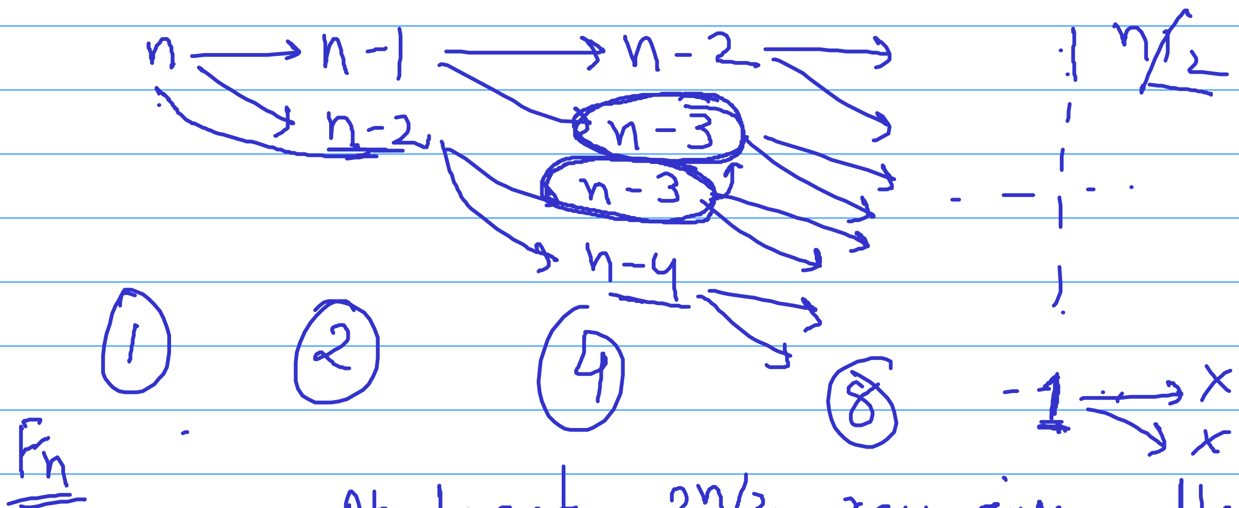
To show that this is indeed true, start with an arbitrary subsequence of S_1 that matches with S_2. Now, move the matching of first letter of S_2 to its first occurence in S_1. You still have the valid subsequence. Repeat the argument for every letter of S_2 one by one.

Dynamic Programming

Simple Example:

$$F_n = F_{n-1} + F_{n-2} \qquad F_0 = F_1 = 1$$

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



At least $2^{n/2}$ recursive calls

Bad Implementation.

Better Implementation

Array F of length n+1.

Memoization.

```
F[0] = 1
F[1] = 1
Bottom-up
for i=2 to n
        F[i] = F[i-1] + F[i-2].
end for.
```

$$F[i] = \underbrace{F[i-1]} + \underbrace{F[i-2]}$$

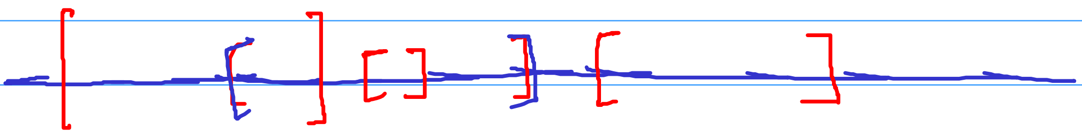
already stored in array.

In dynamic programming, you reduce your problem to one or many subproblems. And if the same subproblem instance is used multiple times then you solve it only once and afterwards, keep using its stored solution.
If the total number of distinct subproblem instances needed during the course of your algorithm is polynomially bounded then your algorithm is efficient.

for example, in the Fibonacci problem we had n distinct enstances of the subproblem
Fibonacci(n), Fibonacci(n-1), ..., Fibonacci(1)

Interval Scheduling. [Kleinberg Tardos Chapter 4]

Resource / server doing computation.



n requests.
(s_1, t_1) (s_2, t_2), ..., (s_n, t_n)

Maximize the number requests you can cater to.
Constraints:
→ If you accept a request then you have to allocate the resource for the whole duration of desired interval (No partial allocation)
→ The resource can be allocated to only one person at a time.