Dynamic Programming

Aims to solve the problem of multiple computations. The idea here is to store the results of computations which would be used again by the algorithm.

A very famous example here is the efficient algo. for Fibonacci numbers. The step where the results are stored is called as "Memo-ization".

Problem:- Weighted interval Scheduling – find the selection of non-conflicting jobs so that the total weight is maximized.

Algorithm:- Order the jobs in an increasing order of finish time.

$$f(1) \le f(2) \le \dots \le f(n)$$

We also define a function p, which returns the latest ending job ending before the argument which does not conflict with it

We can see that:-

If $f(n)$ is in OPT, then we can recurse on jobs till $p(n)$

$f(n)$ is not in OPT, the recurse on 0 to $(n-1)$

This requires us to know if n is in OPT. We solve the problem for both cases and choose the one which has largest total weight. We can use dynamic programming to make this easier.

The steps that we have defined earlier are for the last element of the sequence. We recurse on it as such:-

wtSch(i) $\rightarrow$ The problem for the sub sequence $f(0), \dots, f(i)$

st(i) $\longrightarrow$ The stored value for the sub-problem

```
wtSch(i):
    if (i == 0)
        return 0
    if (st(i) is not Empty)
        return st(i)
    else           ith elem chosen          ith elem discarded
        st(i) <- max { w(i) + wtSch(p(i)) , wtSch(i-1) }
        return st(i)
```

Correctness Proof

Lemma   At every $i \in [n]$, the algorithm computes the optimal solution for the sub-problem $\{0, 1, \dots i\}$ where the intervals are arranged in an increasing order.

This can be proven by induction.

For $i=0$, the choice is trivial.

Assume that all choices upto $n-1$ are optimal (including $n-1$)

At n, the algorithm chooses it based on the value of $w(n) + wtSch(P(n))$ and $wtSch(n-1)$.

By inductive hypothesis:- $w(n) + OPT(P(n))$, $OPT(n-1)$

can see that this is optimal!

$\therefore$ The algorithm takes the right intervals every time.

QED

Time Analysis Finding values of $p(i)$ for all jobs → $O(n\log n)$ binary search

Sorting all schedules in ascending order → $O(n\log n)$

Writing and reading from st is done n times → $O(n)$

$\Downarrow$

$O(n\log n)$

- There are a few families of problems which employ dynamic programming. We state a few without the solution.

1) Parenthesization

A string of matrices is provided. Find the most efficient way to multiply them to save time and resources

2) Longest Increasing Subsequence

3) Longest Common Subsequence

4) Segmentation

Given a string, segment it such that the chunks make sense in English.

5) Edit Number

Given strings x, y; find minimum number of edits needed to convert one to another.

* ## Steps in Dynamic Programming

1) Formulate a method to break problem into smaller sub-problems.
2) Define a Recursive procedure for the sub-problems.
3) Decide on a Memoization strategy.
4) Check that the sub-problem dependencies are acyclic.
5) Analyse the time complexity.

Problem **Parenthization**

Input — A string of 'n' matrices A_i, with dimensions $c_i \times r_i$.

Goal — Minimum way to multiply them.

Define $para(i,j)$ to be the minimum cost needed for computing $A_i \times A_{i+1} \times \ldots \times A_j$. Notice that $para(i,j)$ can be defined recursively as:-

$$para(i,j) = \min_k \{ para(i,k) + para(k+1,j) + para(i,k,j) \}$$

Memo-isation

Now notice that $para(i,j)$ will be called upon multiple times. We can therefore, store $para(i,j)$ in a matrix to speed up the computation time

The subproblem is proper as acyclic in nature. The time taken would be :-

- Filling $O(n^2)$ table $\longrightarrow O(n^2)$
- Each $para(i,j)$ has to check $(j-i)$ values $\rightarrow O(n)$

$$\Downarrow$$

$$O(n^3)$$

* Shortest path for a Weighted directed Graph

Input — A graph $G(V,E)$ and a weight function $\omega: E \to \mathbb{Z}$.

Also, designated start (s) and terminating (t) nodes.

The graph is acyclic to avoid cycles with -ve weights.

Notice that Dijksta's algorithm cannot be used for this question as we have assumed positive weights in its correctness proof.

- Define $Opt(u,v)$ to be the minimum weight between the nodes u, v. We will work backwards from t instead of starting from s. The reason will become apparent.

$$opt(v,t) = \min_{u \in V,\ (v,u) \in E} \left\{ \omega(v,u) + opt(u,t) \right\}$$

This works backwards, and the acyclic nature of the graph ensures that recursion is acyclic as well.

The time complexity of the algo would be $O(|V|(V+1))$