Rathindra Nath Dutta

Design & Analysis of Algorithms

Contents

1 Dynamic Programming **1**
1.1 Dynamic Programming vs Divide-and-Conquer vs Greedy 4
1.2 *Memoization* 5
1.3 Weighted Independent Set in Path Graph 8
1.4 Rod Cutting 11
1.5 Subset Sum 14
1.5.1 A Generalization: Knapsack Problem 16
1.6 Matrix-chain Multiplication 18
1.7 A List of Problems 20
1.8 Another List 20
1.9 Summary 21

A Birth of 'Dynamic Programming' **23**

Bibliography **25**

CHAPTER 1

Dynamic Programming

> *Those who cannot remember the past are condemned to repeat it.*

Dynamic Programming , *Internet*[1]

Dynamic Programming , like the Divide-and-Conquer or Greedy method is just another algorithm design tool. The term 'Dynamic Programming ' was coined by Richard Bellman and the naming has little to do with what it does rather has a interesting bit of history; interested readers may refer to appendix A. One generally uses Dynamic Programming for solving some *optimization problem*[2] whose brute-force solution takes exponential time with respect to the input size and we simply cannot use Divide-and-Conquer , Greedy or any other easy approach.

One might wonder whether Dynamic Programming is kind last resort i.e. go to tool when other techniques like Divide-and-Conquer or Greedy fails. Rephrasing this, one can see that essentially the question is: can Dynamic Programming solve any problem? The answer is obviously no. There are yet unsolvable problems like the classical Halting problem. Furthermore there are also solvable problems where Dynamic Programming cannot be applied! We will study a few such cases . So now another question arises: given a problem, how can one know whether Dynamic Programming is a suitable tool or not!?

To apply Dynamic Programming one must first verify whether the problem exhibits the two essential properties namely: *optimal substructure* and *overlapping subproblem*. Note that for a given optimization problem, designing an efficient (hopefully polynomial time) algorithm is all about carefully looking into the solution space and cautiously examining very few(should be polynomial on input size) of the feasible solutions and reporting the optimal one. A problem must possess aforementioned two properties in order to get real benefit from applying Dynamic Programming .

[1] `https://www.hackerearth.com/practice/notes/dynamic-programming-i-1`
`https://www.quora.com/What-does-dynamic-programming-tell-you-about-life`

[2] A optimization problem asks one to optimize(minimize or maximize) some quantity expressed by a objective function with respect to some given set of constraints.

Optimal Substructure

The optimal solution of the problem must be composed of optimal solution(s) of some subproblem(s). Whenever we combine optimal solutions of more than one subproblem, they must be independent of each other. In essence, we can define the optimal solution recursively in terms of optimal solution(s) of the subproblem(s).

Overlapping Subproblem

A recursive algorithm tries to solve same sub problem over and over again. Total number of distinct subproblems solved is relatively small, typically polynomial on input size.

You might be wondering how can Dynamic Programming rely on on subproblems being both independent and overlapping - do they not contradict each other? The answer is no, they actually express two different notion. Two subproblems of the same problem are independent if they do not share resources. Therefore, they could easily be combined together to form a candidate solution. On the other hand, we say two subproblems are overlapping if they are really the same but occurs as subproblems of two different problems.

Let us first understand how these two actually helps in designing an efficient Dynamic Programming algorithm. Typically there are multiple independent subproblems which may correspond to various candidate solution of the original one. But not all of them leads to the optimal solution. If we somehow knew(say lucky guess) which subproblem(s) to pick, all it remains to build upon the optimal solution(s) of it(them). But there is no such thing as lucky guess in algorithm design! The first property actually helps us narrow down the candidate solution sets of a subproblem, as we only need to consider the optimal one. We can now practically apply brute force on this reduced search space and pick the subproblem(s) that actually leads to the optimal solution. Whereas the second property implies that is a recursive algorithm revisits same subproblem repeatedly, we can cache the previously computed result into some table. Thus we are actually computing only once, the first time; all subsequent requirements can be served by a constant time table lookup. This technique is called *memoization*. It might seem a little abstract, because it is, but this will become clear as we explore a few Dynamic Programming algorithms in the later sections.

Essentially a dynamic programming solutions for different problems varies on two things: how many distinct subproblems are actually needed to consider to get the optimal solution, and how many choices are there for determining which subproblem(s) to use in an optimal solution. Informally, the running time of a Dynamic Programming algorithm depends of these two factors.

Dynamic Programming algorithms can be formulated in two ways; either by a **top-down** recursive approach with memoization or using a **bottom-up** approach where we optimally solve smaller subproblems first then gradually increase the problem size until

we get optimal solution of the original problem. In general, if all subproblems must be solved at least once, a bottom-up strategy usually outperforms the corresponding top-down memoized algorithm by a constant factor. The top-down approach has overheads for the recursive calls and table maintenance. Alternatively, if some subproblems in the subproblem space need not to be solved at all, the memoized top-down version has the advantage to solve only the required ones.

Designing a Dynamic Programming algorithm more or less involves these four steps:

1. Characterize the structure of the optimal solution
2. Recursively define the value of the optimal solution
3. Compute the value of the optimal solution; typically in bottom-up fashion
4. Reconstruct (if required) the optimal solution from the choices made during step 3

In this chapter we will start by comparing Dynamic Programming with Divide-and-Conquer and Greedy followed by the implementation details of *memoization* and finish with a few case studies.

1.1 Dynamic Programming vs Divide-and-Conquer vs Greedy

Both Dynamic Programming and Divide-and-Conquer approach breaks the problem into subproblems and build the solution based on the solution of the subproblems. On the other hand both Dynamic Programming and Greedy rely on having optimal substructure property of the problem definition. Therefore it is intuitive compare these three strategies and know their differences.

In general, a problem having an efficient Divide-and-Conquer algorithm often lacks the overlapping subproblem property. In Divide-and-Conquer algorithms at each recursive call it generates new and new subproblems to solve; therefore memoization does not help whereas in Dynamic Programming same subproblems are revisited repeatedly and memoization does help significantly.

Although both the Greedy strategy Dynamic Programming exploits optimal substructure property one major difference is how they choose a subproblem. Instead of evaluating all candidate subproblems and choosing the one that leads to a globally optimal solution, a Greedy algorithm first makes a 'greedy' choice - that seems best at the moment, only then it tries to solve the chosen subproblem. Therefore Greedy algorithms are often faster but there is the risk of being stuck at local optima. One can think of Dynamic Programming as bottom-up approach where we choose suitable subproblems at each step and finally get the optimal solution, whereas Greedy works in top-down fashion where at each step it makes a greedy locally optimum choice and then only solves the resulting subproblem.

1.2 *Memoization*

Before delving further into Dynamic Programming paradigm let us first study how to implement memoization and how much benefit we can actually obtain. Consider the problem of calculating the n-th Fibonacci number recursively as per the definition[3].

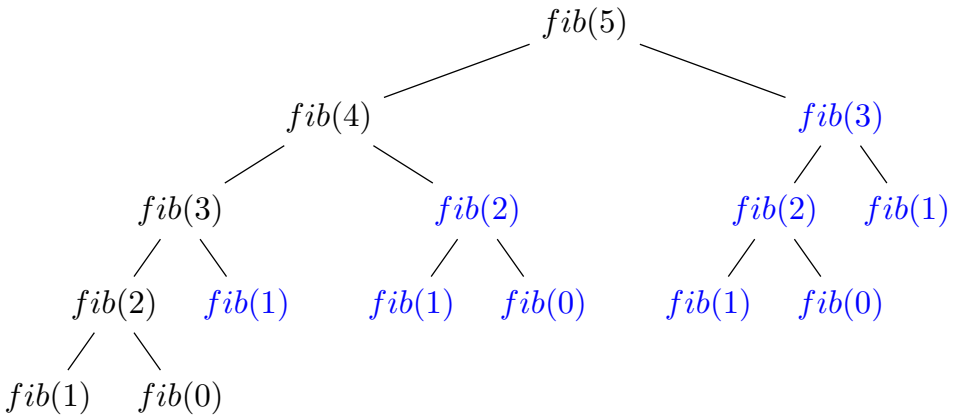

Figure 1.1: Recursion tree for $fib(n)$ when $n = 5$

Theorem 1.2.1. *There are* $\Theta(2^n)$ *recursive calls for a single call to* $fib(n)$.

Proof. Let $T(n)$ be the total number of recursive calls made for a single call to $fib(n)$. We will show that $2^{n/2} \leq T(n) \leq 2^n, \forall n \geq 2$ using induction on n.
Basis: We have $T(2) = 1 + 1 = 2$ therefore $2^{2/1} \leq T(2) \leq 2^2$. Similarly, $T(3) = T(2) + 1 = 3$ thus $2^{3/1} \leq T(3) \leq 2^3$.
Hypothesis: Let's assume that the statement holds for both $T(n-1)$ and $T(n-2)$.
Inductive step:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq 2^{n-1} + 2^{n-2} \\ &= 2^{n-2}[2+1] \\ &< 2^{n-2} \times 4 = 2^n \end{aligned}$$

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\geq 2^{(n-1)/2} + 2^{(n-2)/2} \\ &= 2^{(n-2)/2}[\sqrt{2}+1] \\ &> 2^{(n-2)/2} \times 2 = 2^{n/2} \end{aligned}$$

■

As depicted in figure 1.1 there are exactly n recursive calls we need to compute the respective value for the first time; all subsequent calls, shown in blue, can be replied with

[3] $fib(n) = \begin{cases} n & when\ n \leq 1 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$

the cached results. Caching return values in case of recursive calls is called memoization. The result is cached into a memo table indexed by the call argument(s). Assume that we have an array $f[0 \ldots n]$ stored somewhere globally. The following pseudo code should clarify how one can apply memoization. As evident from figure 1.1 this pseudocode runs

Algorithm 1 Memoized Fibonacci Computation

```
1: procedure INITTABLE(n)
2:     for i ← 0 to n do
3:         f[i] ← −1                          ▷ Some marker value
4:     end for
5: end procedure
```

```
1: procedure FIB(n)
2:     if f[n] ≠ −1 then return f[n]          ▷ return the memoized value
3:     end if
4:     if n ≤ 1 then                          ▷ base case
5:         f[n] ← n
6:         return f[n]
7:     end if
8:     f[n] ← FIB(n − 1) + FIB(n − 2)
9:     return f[n]
10: end procedure
```

in linear time as its only computes the value for the left branch from the root(the black nodes) and all right branch(the subtrees at rooted at some blue node) values are looked up in constant time from the table f. The same recursion tree essentially gets reduced to figure 1.2.

Now consider the problem of calculating binomial coefficient recursively as follows:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & when\ 0 < k < n \\ 1 & otherwise \end{cases} \tag{1.1}$$

Theorem 1.2.2. *Show that the above expression correctly computes* $\binom{n}{k}$

Proof. Left as an exercise. ■

Theorem 1.2.3. *Show that a simple recursive formulation takes* $\Omega(\binom{n}{k})$ *time.*

Proof. Left as an exercise. ■

We can use memoization here also; the memo table in this case generates the Pascal's triangle as depicted in figure 1.3. The table is filled line by line. In fact, it is not even

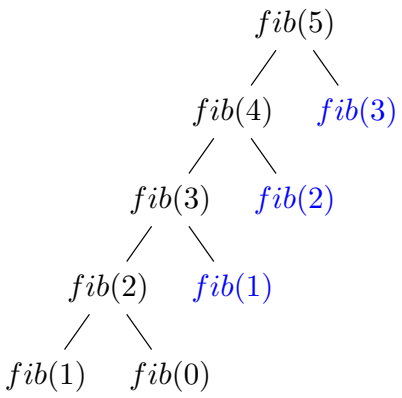

Figure 1.2: Memoized Recursion tree for $fib(n)$ when $n = 5$

required to store a matrix; it is sufficient to just keep a vector of length k for the current line and calculate the next line by updating the entries from left to right. Thus the algorithm takes $O(nk)$ time and $O(k)$ space for its execution. A practice exercise would be to ask you to formally write the algorithm (either in plain text or pseudocode format).

	0	1	2	...	k-1	k
0	1					
1	1	1				
2	1	2	1			
⋮						
n-1					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

Figure 1.3: Pascal's triangle

Again we have seen that memoization[4] helps. With this in mind we can now turn out attention to some interesting problems in the following sections which have very nice Dynamic Programming algorithms. Not only these examples are excellent candidates for studying design and analysis of algorithms but also they help you better understand the notion of Dynamic Programming .

[4]For practical purposes if not all of the possible subproblems are not required, we can memoize by hashing on the indices/function arguments, instead of creating a large table, to optimally use the storage space with negligible compromisation on the lookup time.

1.3 Weighted Independent Set in Path Graph

Let us begin by defining a few terms.

Path Graph A path graph P_n of n vertices is a degenerate tree where exactly two vertices have degree one(the end vertices) and all other $n-2$ vertices have degree exactly two. A path graph is generally drawn as a straight line with vertices lying on it.

Independent Vertex Set Given a graph $G = (V, E)$, two vertices $u, v \in V$ are said to be independent if the edge $(u, v) \notin E$. Note that if the graph is directed both $\langle u, v\rangle \notin E$ and $\langle v, u\rangle \notin E$ must hold. An independent vertex set S is a subset of vertices where every pair of vertices in S are independent. Using mathematical notations we can write, $S \subseteq V$ such that $\forall u, v \in S,\ \ (u, v) \notin E$.

Weighted Independent Set When the vertices have some weights associated with them, we define the weight of a independent set as sum of weights of the vertices in it.

Given a graph $G = (V, E)$, the maximum weighted independent set problem is to find an independent set S of G having the maximum weight. Therefore it a optimization problem where we try to maximize the set weight. Here we will restrict ourselves to only path graphs having positive vertex weights.

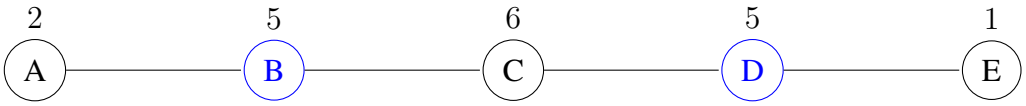

Figure 1.4: An example path graph P_5

The path graph shown in figure 1.4 has the following independent sets: $\varnothing$, {A}, {B}, {C}, {D}, {E}, {A,C},{A,D}, {A,E}, {B,D}, {B,E}, {C,E}, {A,C,E} out of these only {B,D} has the maximum weight value of 10. As you can imagine the possibilities of the independent subsets grows exponentially with n implying that we need an efficient algorithm rather than using a brute-force approach.

Theorem 1.3.1. *The number of possible independent sets of a path graph P_n is $\Theta(2^n)$.*

Proof. We know that there are 2^n possible subsets of a given set of n vertices out of these not all are independent set. For example {A,B,C} is not an independent set for figure 1.4. We will begin the proof by formulating a recurrence relation for the number of possible independent sets of given path graph.

Suppose we have indexed the vertices from 1 to n of a given path graph P_n, starting at one end and finishing at the other. Let $f(n)$ denotes the possible independent sets of

P_n. There are only two possible choices for the n-th vertex: either it is in a independent set or not. When we exclude the n-th vertex then we can build $f(n-1)$ independent sets with first $n-1$ vertices. Whereas if we include the n-th vertex then we must exclude $n-1$-st vertex, thus can build only $f(n-2)$ independent sets. Now these two counts are disjoint by construction. Therefore we have $f(n) = f(n-1) + f(n-2)$ with $f(0) = 1$ and $f(1) = 2$. Following the proof of theorem 1.2.1 we have $f(n) = \Theta(2^n)$ ■

Following the above proof we can see that our optimization problem has a nice recursive structure. The optimal solution of n vertices must be one of two following cases: optimal solution for first $n-1$ vertices when we exclude n-th vertex; or optimal solution for first $n-2$ vertices and we include weight n-th vertex. Let $OPT(n)$ be the optimal value, weight of the maximum weighted independent set of a given path graph P_n along with a weight vector W. Then we have

$$OPT_n = \begin{cases} 0 & when\ n = 0 \\ W_1 & when\ n = 1 \\ \max\{OPT_{n-1}, \quad W_n + OPT_{n-2}\} & otherwise \end{cases}$$

It is evident that the problem possess both optimal substructure and overlapping subproblem properties of Dynamic Programming . Assuming the input weight array $w[1 \ldots n]$ and the memo table $OPT[0 \ldots n]$ stored somewhere globally, the top-down algorithm with memoization looks as follows.

Algorithm 2 Maximum Weighted Independent Set in Path Graph Top-Down Approach

```
1: procedure INITTABLE(n)
2:     for i ← 0 to n do
3:         OPT[i] ← −1                          ▷ Some marker value
4:     end for
5:     OPT[0] ← 0                               ▷ base values
6:     OPT[1] ← w[1]
7: end procedure
```

```
1: procedure MAXINDSET(n)
2:     if OPT[n] ≠ −1 then return OPT[n]        ▷ return the memoized value
3:     end if
4:     OPT[n] ← max{MAXINDSET(n − 1), w[n] + MAXINDSET(n − 2)}
5:     return OPT[n]
6: end procedure
```

Similarly we can formulate a bottom-up algorithm as given below. Now its time for talking about correctness as well as time and space complexities for both of the approaches.

Algorithm 3 Maximum Weighted Independent Set in Path Graph Bottom-Up Approach

```
1: procedure MaxIndSet(n)
2:     OPT[0] ← 0                                          ▷ base values
3:     OPT[1] ← w[1]
4:     for i ← 2 to n do
5:         OPT[i] ← max{OPT[i − 1],  w[n] + OPT[i − 2]}
6:     end for
7:     return OPT[n]
8: end procedure
```

Both top-down and bottom-up strategies essentially relies on the above recurrence whose correctness follows from the following "cut-and-paste" argument. Note that optimization problem is a maximizing one. If we does not include optimal solution values of the subproblems to get the actual solution, we could then simply cut the non optimal subsolution and paste the optimal one in its place. This would give us a larger solution value of the original problem and contradicts the assumption of our solution being an optimal one. We can conclude the proof by stating that our case analysis was disjoint and exhaustive.

The space requirement is for both of theses algorithms is $\Theta(n)$. Both of them solves each of the subproblems exactly once and for each one of them we have just two choices: whether to include the last vertex or not, each choice involves constant number of operations. There are n many subproblems each requires only constant amount of work giving us running time of $\Theta(n)$.

We could also reconstruct the optimal independent set by back tracing our process of getting the optimal value. At each subproblem we had only two choices we just need to know which choice did us lead to the optimal result. Recall that a choice here is essentially deciding whether to include a vertex into the optimal independent set or not. One could also store the choice while making it to speedup the reconstruction. Writing a formal algorithm which reconstructs the optimal independent set in linear time is left as an exercise.

1.4 Rod Cutting

Given a rod of length n and a price table p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue value r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n is large enough, an optimal solution may require no cutting at all.

A naïve brute force solution would be to try all possible cuts and take the maximum among them. This algorithm will basically evaluate all possible (unique) partitions of the given integer n. Interested readers may read up more about Integer Partitioning. As you have already guessed the solution is space quite large making this approach is inefficient and possibly infeasible for large n.

One may argue about formulating a Greedy or Divide-and-Conquer algorithm which solves the problem in polynomial time. Thinking carefully one would realize making a greedy choice may not lead to the optimal cut whereas Divide-and-Conquer could not figure out which of the subproblem(s) will lead to the optimal solution so that it can recurs on it.

Therefore, we could try to fit the problem into Dynamic Programming paradigm. Let us first verify whether the problem exhibits the aforementioned two properties namely: optimal substructure and overlapping subproblem. For that we need to come up with a good recurrence for the problem itself. Suppose we cut the rod into two halves first(possibly also as $0 + n$) and the recursively solve the problems for the two smaller rods. But how come we could possibly know where to cut first! But once known the problem essentially reduces to optimally cut the tow smaller halves. We can thus calculate the optimal revenue value of rod cutting as maximum among all possible optimal cuts of smaller rods: $r_n = \max\{p_n, (r_1 + r_{n-1}), (r_2 + r_{n-2}), \ldots (r_i + r_{n-i}), \ldots, (r_{n-1} + r_1)\}$. This essentially exhibits that the problem does possess optimal substructure property. Moreover same r_i values are required over and over inside the recursive calls; implying the overlapping subproblem property.

Now we are in a good place to actually formulate a Dynamic Programming algorithm for the rod cutting problem. But before doing that let us optimize the recurrence a little bit. Suppose, instead of cutting at the middle we cut at one side; then could just optimally solve for the remaining rod. The above recurrence then turns into: $r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$ with revenue $r_0 = 0$. Writing the algorithm is now straight forward.

Lemma 1.4.1. *Procedure 4 takes exponential run time*

Proof. Left as an exercise. [hint: solution of $T(n) = \Theta(1) + \sum_{j=0}^{n-1} T(j)$ is $O(2^n)$] ■

Lemma 1.4.2. *This using memoization the time complexity reduces to* $\Theta(n^2)$

Proof. Writing the formal algorithm is left as an exercise. The memo table initialization requires $\Theta(n)$ operations. The actual procedure now recursively solves each subproblem

Algorithm 4 Rod Cutting Top-Down Approach

Require: The price array $p[1 \dots n]$ stored somewhere globally

```
1:  procedure CutRod(n)
2:      if n = 0 then return 0
3:      end if
4:      r ← −∞
5:      for i ← 1 to n do
6:          tmp ← p[i] + CutRod(n − i)
7:          if tmp > r then
8:              r ← tmp
9:          end if
10:     end for
11:     return r
12: end procedure
```

exactly once and all subsequent calls are replied in constant time. There are exactly $n+1$ subproblems of sizes $0, 1, \dots, n$. To solve each subproblem of size i the for loop iterates exactly i times. Therefore, total number of iterations over the entire execution is $1 + 2 + \cdots + n = \Theta(n^2)$. This gives a total cost of $\Theta(n^2)$. The space complexity is $\Theta(n)$. ■

Lemma 1.4.3. *Using a bottom-up approach the complexities remain same.*

Proof. We will begin by writing the formal algorithm (see algorithm 5). The procedure now iteratively solves each samller subproblems first then the larger ones. The outer loop iterates n times while at j-th pass of the outer loop the inner loop iterates j times. Thus the statements inside the inner loop gets executed a sum total of $1 + 2 + \cdots + n = \Theta(n^2)$ times. This gives a total running cost of $\Theta(n^2)$ same as before. The space complexity is same $\Theta(n)$ due to the OPT table. ■

Lemma 1.4.4. *Both the bottom-up and top-down procedure correctly solves the optimal rod cutting problem*

Proof. The proof of optimality for both of the algorithms essentially relies of the recurrence we formed for the r_n which again relies on optimal substructure property of the problem itself and uses "cut-and-paste" argument as before. Let us assume the optimal first cut is of length i as per the algorithm. Then we have $r_n = \max_{1 \le k \le n} (p_k + r_{n-k})$. Now suppose optimal value is some r_n^* which must have a first cut, say of length i $(1 \le i \le n)$.

Algorithm 5 Rod Cutting Bottom-Up Approach

Require: The price array $p[1 \dots n]$ stored somewhere globally

```
1:  procedure CutRod(n)
2:      let OPT[0...n] be a new array
3:      for j ← 1 to n do
4:          r ← −∞
5:          for i ← 1 to j do
6:              tmp ← p[i] + CutRod(j − i)
7:              if tmp > r then
8:                  r ← tmp
9:              end if
10:         end for
11:         OPT[n] ← r
12:     end for
13:     return OPT[n]
14: end procedure
```

Therefore we can write $r_n^* = p_i + r_{n-i}^*$. Now we have

$$\begin{aligned} r_n^* &> r_n \\ p_i + r_{n-i}^* &> \max_{1 \le k \le n} (p_k + r_{n-k}) \\ p_i + r_{n-i}^* &> p_i + r_{n-i} \\ r_{n-i}^* &> r_{n-i} \end{aligned}$$

This question the optimality of r_{n-i}; which is a contradiction. ■

Lemma 1.4.5. *We can also obtain the cut sizes that leads to the optimal solution.*

Proof. This will require augmenting our algorithm to keep track of which length was the optimal(winner) for each sub problem. Thus another n sized table would suffice. After computing the final solution value, we can then reconstruct the cut sizes from this auxiliary table in backward fashion. Writing formal algorithms is left as an exercise. ■

1.5 Subset Sum

Given n items indexed from 1 though n each having a nonnegative weight w_i, for $i = 1, 2, \ldots, n$ and a maximum weight bound W. We are to select a subset of items so that total weight of the subset is maximized but not exceeding W. Note that total weight of a subset means the sum of weights of the items in that subset. We will restrict ourselves to only integral values for the weights and the weight bound. We can state this as a *linear programming problem* whose solution is a vector $x[1 \ldots n]$ with $x_i = 1$ denoting i-th element is in the optimal subset and 0 otherwise.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} x_i w_i & \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i w_i & \leq W \\ & x_i & \in \{0, 1\} \quad \forall i = 1, 2, \ldots, n \end{aligned}$$

A straight forwards brute force approach is to examine all of the 2^n possible subsets to get the optimal one which makes it a very inefficient algorithm. One can argue about formulating some greedy strategy; then again it is possible to create instances where the Greedy strategy fails[5]. For example if a Greedy algorithms picks the weights in decreasing order then it fails to find the optimal solution for the instance $\{\frac{W}{2} + 1, \frac{W}{2}, \frac{W}{2}\}$. Again if it selects the smallest weight first then also it fails for the instance $\{1, \frac{W}{2}, \frac{W}{2}\}$.

Just like previous two problems, let us try to formulate a recursive definition of the optimal value (the maximum possible weight). For the n-th item we really have only two choices either to include it in the optimal set or not. If the optimal solution does not include n-th item then it must also be optimal solution of first $n - 1$ items. Whereas if the optimal solution does include n-th item then we reserve the weight for n-th item and then look at optimal solution of first $n - 1$ items but with the remaining weight capacity. Note that the subproblem varies in two degrees: one is number of items it is solving for another is current remaining weight capacity. Furthermore, the n-th item can only be added if its weight does not exceed the current (remaining) weight capacity. Let us formalize it as follows:

$$OPT_{n,W} = \begin{cases} 0 & when\ n = 0 \\ OPT_{n-1,W} & when\ w_n > W \\ \max\{OPT_{n-1,W}, \quad w_n + OPT_{n-1,W-w_n}\} & otherwise \end{cases}$$

Theorem 1.5.1. *Show that the above recurrence correctly computes the optimal value.*

Proof. Left as an exercise. [hint: use the "cut-and-paste" argument with induction] ■

Note that although the problem does exhibits optimal substructure property it may lack the overlapping subproblem property if all the weight values including the bound are

[5] So far no known Greedy strategy exist which solves the problem optimally.

not integral. Following this recurrence, one can easily formulate an algorithm. Assume that the input weight array $w[1 \ldots n]$ and the memo matrix $OPT[0 \ldots n, 0 \ldots W]$ stored somewhere suitably.

Algorithm 6 Subset Sum Top-Down Approach

```
1:  procedure INITTABLE(n, W)
2:      for i ← 0 to n do
3:          for j ← 0 to W do
4:              OPT[i, j] ← −1                          ▷ some marker value
5:          end for
6:      end for
7:      for j ← 0 to W do
8:          OPT[0, j] ← 0                               ▷ base values
9:      end for
10: end procedure
```

```
1:  procedure SUBSETSUM(n, W)
2:      if OPT[n, W] ≠ −1 then return OPT[n, W]      ▷ return the memoized value
3:      end if
4:      tmp ← OPT[0, j] ← 0
5:      if w[n] > W then
6:          OPT[n, W] ← SUBSETSUM(n − 1, W)
7:      else
8:          OPT[n, W] ← max { SUBSETSUM(n − 1, W),
                              w[n] + SUBSETSUM(n − 1, W − w[n]) }
9:      end if
10:     return OPT[n, W]
11: end procedure
```

Both top-down and the bottom-up approach essentially does the same amount of work: solving the subproblems optimally exactly once and generate the optimal solution using them. As it is evident there are $(n+1) \times (W+1)$ subproblems and solving for i-th one is essentially deciding whether to include i-th item or not; we only require constant amount of work in either case. Thus total running time is $O(nW)$ and space requirement is also $O(nW)$. We call this type of algorithms *pseudo-polynomial* and are considered reasonably fast. It should be noted that if W is large enough top-down approach may run faster as it might not need to solve all the subproblems.

One can view the subset sum problem as a *job scheduling problem* on a system where the items are the jobs and the weights are the resource requirement of each jobs and of course the system has some fixed amount of resources available only giving us a maximum weight bound.

Algorithm 7 Subset Sum Bottom-Up Approach

```
1: procedure SUBSETSUM(n, W)
2:     for j ← 0 to W do
3:         OPT[0, j] ← 0                                   ▷ base values
4:     end for
5:     for i ← 0 to n do
6:         for j ← 0 to W do
7:             if w[i] > W then
8:                 OPT[i, j] ← OPT[i − 1, j]
9:             else
10:                OPT[i, j] ← max { OPT[i − 1, j],
                                   { w[i] + OPT[i − 1, j − w[i]] }
11:            end if
12:        end for
13:    end for
14:    return OPT[n, W]
15: end procedure
```

The above procedures only gives us the total weight for the optimal subset. Suppose we wish to also obtain the optimal set itself. We then could augment our algorithm to keep track of our decision for each of the subproblems and later reconstruct the solution set. For this we will use an auxiliary boolean matrix $sol[1 \ldots n, 0 \ldots W]$.

Writing a non-recursive (iterative) reconstruction procedure is left as an exercise. In either case we require $O(nW)$ to store the decision we made at each subproblem and the actual reconstruction is done in linear time(a single pass from n down to 1).

1.5.1 A Generalization: Knapsack Problem

The problem is often associated with a story which goes as follows: a thief robbing a jewelry store wishes to take as much valuable things as he can into his knapsack such that total weight of the loot does not exceed the weight capacity of the knapsack.

Therefore we have n items indexed from 1 though n each having a nonnegative weight w_i and also a value v_i, for $i = 1, 2, \ldots, n$ and a maximum weight bound W. We are to select a subset of items to put into the knapsack so that total weight of the subset is maximized but not exceeding W, capacity of the knapsack. We will again restrict ourselves to only integral values for the weights and the weight bound.

The subset sum problem can be seen just as a special case of knapsack problem where v_i is same as w_i, $\forall i = 1, 2, \ldots, n$.

Algorithm 8 Subset Sum Reconstruction

```
1:  procedure SUBSETSUM(n, W)
2:      for j ← 0 to W do  OPT[0, j] ← 0                          ▷ base values
3:      end for
4:      for i ← 0 to n do
5:          for j ← 0 to W do
6:              tmp ← w[i] + OPT[i − 1, j − w[i]]
7:              if w[i] > W or OPT[i − 1, j] > tmp then            ▷ i-th item isn't selected
8:                  OPT[i, j] ← OPT[i − 1, j]
9:                  sol[i, j] ← 0
10:             else                                              ▷ i-th item is selected
11:                 OPT[i, j] ← tmp
12:                 sol[i, j] ← 1
13:             end if
14:         end for
15:     end for
16:     return OPT[n, W]
17: end procedure
1:  procedure RECONSTRUCT(n, W)                                   ▷ Recursive
2:      if n = 0 then return
3:      else if sol[n, W] = 0 then                                ▷ n-th item isn't selected
4:          RECONSTRUCT(n − 1, W)
5:      else                                                      ▷ n-th item is selected
6:          RECONSTRUCT(n − 1, W − w[n])
7:          print n
8:      end if
9:  end procedure
```

We can state this as a *linear programming problem* whose solution is a vector $x[1 \dots n]$ with $x_i = 1$ denoting i-th element is in the optimal subset and 0 otherwise.

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} x_i v_i \\ \text{subject to} \quad & \sum_{i=1}^{n} x_i w_i \leq W \\ & x_i \in \{0,1\} \quad \forall i = 1, 2, \dots, n \end{aligned}$$

Devising formal algorithms relies on the following recurrence, and thus left as exercises

$$OPT_{n,W} = \begin{cases} 0 & when\ n = 0 \\ OPT_{n-1,W} & when\ w_n > W \\ \max\{OPT_{n-1,W},\quad v_n + OPT_{n-1,W-w_n}\} & otherwise \end{cases}$$

1.6 Matrix-chain Multiplication

Given a sequence(chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, we wish to compute the product $A_1 \times A_2 \times \cdots \times A_n$. Recall that in general matrix multiplication is commutative but not associative. The order of the multiplications of the matrices can be denoted by fully parenthesizing the given chain. Therefore we have many possible parenthesization to choose from to obtain the same final resultant matrix. Assume that we are using standard matrix multiplication algorithm (see algorithm 9) which takes two matrices A and B along their dimensions as input and output the resultant matrix C.

Algorithm 9 Standard Matrix Multiplication

```
1:  procedure MATRIXMULTIPLY(A, B, p, q, r)
2:      take a new matrix C[1...p, 1...r]
3:      for i ← 1 to p do
4:          for j ← 1 to r do
5:              C[i, j] ← 0
6:              for k ← 1 to q do
7:                  C[i, j] ← C[i, j] + A[i, k] · B[k, j]
8:              end for
9:          end for
10:     end for
11:     return C
12: end procedure
```

Recall that we can multiply two matrices only if they are *multiplication compatible*, that is, number of columns of first matrix must be equal to number of rows in the second one. Here our two input matrices have are of dimensions $p \times q$ and $q \times r$, therefore resultant matrix will be of dimension $p \times r$. Following the algorithm, we need to compute all pr cells in C, for which need have to perform a dot product of two q sized vectors (pair wise multiplication followed by summations). Therefore we need a total of $\Theta(pqr)$ operations, to be more precise there exactly pqr number of scalar multiplication involved.

Let us consider a case with chain size of 3 and the matrices A_1, A_2, A_3 are of dimensions $20 \times 12, 12 \times 19, 19 \times 92$ respectively. There are only two possible parenthesizations: $((A_1 \times A_2) \times A_3)$ and $(A_1 \times (A_2 \times A_3))$. Multiplying A_1withA_2 requires exactly $20 \times 12 \times 19$ scalar multiplications and results into a matrix of dimension 20×19. Therefore $((A_1 \times A_2) \times A_3)$ performs a total of $(20 \times 12 \times 19) + (20 \times 19 \times 92) = 4560 + 34960 = 39520$ scalar multiplications. Similarly, $(A_1 \times (A_2 \times A_3))$ requires a total of $(12 \times 19 \times 92) + (20 \times 12 \times 92) = 20976 + 22080 = 43056$ scalar multiplications. As it is evident different parenthesization incurs different cost(number of operations). This brings us to the problem finding the optimal possible parenthesization which minimizes the cost.

Formally, the ***matrix-chain multiplication problem*** is as follows: given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices along with their dimensions we want to fully parenthesize the product $A_1 \times A_2 \times \cdots \times A_n$ in a way that minimizes the number of total scalar multiplications.

You might have guessed the number of possible parenthesizations grows rapidly as n increases. Combinatorics allows us to nicely calculate the number of possible parenthesizations. Let us denote that number by P_n. Given a chain of length n there are exactly $n-1$ we can split it into two sub chains. This gives us the following recurrence:

$$P_n = \begin{cases} 1 & when\ n = 1 \\ \sum_{k=1}^{n-1} P_k + P_{n-k} & otherwise \end{cases}$$

It can be shown that $P_n = C_{n-1}$ where C_n is n-th ***Catalan number***[6] defined as follows.

$$C_n = \frac{1}{n+a} = \Omega(4^n/n^{3/2})$$

To show brute force fails
show it has optimal substructure overlapping subproblem little birdie table?
formulate the recurrence need to reconstruct
top-down
bottom-up
complexity
reconstruction
complexity

[6]Catalan number, named after the Belgian mathematician Eugène Charles Catalan, often comes up in context various counting problems. It is often used to denote the total number of possible Binary Search Trees(a type of ordered binary tree) with n distinct keys. Our parenthesization problem is essentially corresponds to building an expression tree where the operands matrices appear in a fixed (left-to-right) order as leave nodes of the tree.

1.7 A List of Problems

1.8 Another List

1.9 Summary

- We may apply Dynamic Programming to some optimization problem generally when Divide-and-Conquer and Greedy fails.
- We can only use Dynamic Programming whenever the problem exhibits both *optimal substructure* and *overlapping subproblem* properties.

APPENDIX A

Birth of 'Dynamic Programming'

> *An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, 'programming.' I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying—I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. This, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities.*
>
> Richard Bellman, *Eye of the Hurricane: An autobiography*, 1984

The Bellman-Ford shortest path algorithm presented earlier is named after Richard Bellman and Lester Ford. In fact that algorithm can be viewed as a dynamic program.

Although the quote is an interesting bit of history it does not tell us much about dynamic programming. But perhaps the quote will make you feel better about the fact that the term has little intuitive meaning! In short, the term programming essentially means planning (like in linear programming) while the word dynamic express the non static nature of the algorithm design.

Bibliography