

The Dynamic Programming Technique

This note defines and gives examples of the Dynamic Programming Technique.

We introduce the technique by giving an example. We wish to solve the following:

SIMPLE-KNAP-SACK-WEIGHT(SKSW):

INPUT: $W[1], \dots, W[n] \in \mathbb{N}$ and a capacity $c \in \mathbb{N}$

PROBLEM: Find the weight of a heaviest subcollection of weights with total weight $\leq c$. Each weight can be used at most once. That is, $\max\{\sum_{i \in S} W[i] : S \subseteq \{1, \dots, n\} \text{ and } \sum_{i \in S} W[i] \leq c\}$

One greedy approach would be the following:

```
Sort the weights so that  $W[1] \geq W[2] \geq \dots \geq W[n]$ ;  
max=0;  
for(int i=1; i<=n; i++){  
    if(max+W[i]<=c)max=max+W[i];  
}
```

You can try this algorithm on the instance with weights 15, 11, 9, 7, 7, 5 and capacity 15. Try it on the above instance but with capacity 12, 27, 16.

Claim: The greedy algorithm above does very poorly

Proof: We construct a general instance where it does very bad.

Assume that c is even and consider instances of the form $W=c/2, c/2, c/2+1$ with capacity c . The greedy algorithm above returns just the weight $c/2+1$ which is not optimal. An optimal solution is actually c comprised of weights $c/2, c/2$.

The Dynamic Programming Approach

Briefly, it is the following: When trying to solve a problem P we

1. Define an array A (which would be useful in solving our problem P). This definition is high level, a few words in english.
2. Determine how you would use A to solve your problem
3. Write down a recurrence relation which defines A
4. Write an *iterative* algorithm to fill in A and solve problem P
5. Analyse the algorithm

We follow this approach for the SKSW problem above.

1.

$$A[j, m] = \begin{cases} true & \text{if } \exists S \subseteq \{1, \dots, m\} \text{ such that } \sum_{i \in S} W[i] = j \\ false & \text{otherwise} \end{cases}$$

2. $\max\{j : 1 \leq j \leq c \text{ and } A[j, n]\}$ solves SKSW

3.

$$A[j, m] = \begin{cases} true & \text{if } j = 0, m = 0 \\ false & \text{if } j > 0, m = 0 \\ false & \text{if } j < 0 \\ A[j - W[m], m - 1] \vee A[j, m - 1] & \text{otherwise} \end{cases}$$

```

4. boolean [] fillInArray(int [] W, int c){
    // W consists of elements W[1],...,W[n]
    boolean A[c,n]; // We assume that this is initialized to false
    A[0,0]=true;
    for(int m=1;m<=n;m++){
        for(int j=1;j<=c;j++){
            if(j-W[m]>=0)A[j,m]=A[j-W[m],m-1];
            A[j,m]=A[j,m] or A[j,m-1];
        }
    }
    return(A);
}
int SKSW(int [] W, int c){
    // W consists of elements W[1],...,W[n]
    boolean A[c,n]=fillInArray(W,c);
    max=0;
    for(int i=0;i<=c;i++){
        if(A[i,n]==true)max=i;
    }
    return(max);
}

```

We have to be careful to determine that the algorithm above actually works. If you think about it for a minute, you will see that fillInArray only uses previously defined values of A to determine a new value of A. In fact, we could have really used only a 1 dimensional array for A instead of the 2-dimensional one we have used.

5. The running time of SKSW consists of the running time of fillInArray+running time of the for loop determining the solution. This is $O(n \cdot c) + O(c)$ so the total running time is $O(n \cdot c)$.

Finding Optimal Structures

If we were initially given the **search** problem of *finding the structure with optimal value*, then we modify the above dynamic programming approach as follows:

1. Consider the value related problem, *find the value of the optimal structure*
2. Define an array A (which would be useful in solving the value problem). This definition is high level, a few words in english.
3. Determine how you would use A to solve the value problem
4. Write down a recurrence relation which defines A
5. Write an *iterative* algorithm to fill in A
6. Define an array D[] which keeps track of critical decisions made in filling out A
7. Modify the iterative algorithm which fills in A so that it also fills in D
8. Use D and A to obtain an optimal structure.
9. Analyse the algorithm

Consider now, the search problem

SIMPLE-KNAP-SACK-SEARCH(SKSS):

INPUT: $W[1], \dots, W[n] \in \mathbb{N}$ and a capacity $c \in \mathbb{N}$

PROBLEM: Find a heaviest subcollection of weights with total weight $\leq c$. Each weight can be used at most once. That is, find $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} W[i]$ is maximal and $\sum_{i \in S} W[i] \leq c$

1. Consider SKSW as defined above
2. Done above
3. Done above
4. Done above
5. Done above
6. $D[j] = m$ if before considering $W[m]$ we could not make total weight j , but including $W[m]$ allows us to create total weight j . So $D[j] = m$ if $A[j - W[m], m - 1] \wedge \neg A[j, m - 1]$.

```
7. (boolean [],int []) fillInArrays(int [] W, int c){
    // W consists of elements W[1],...,W[n]
    boolean A[c,n]; // We assume that this is initialized to false
    int D[c]; // We assume that this is initialized 0
    A[0,0]=true;
    for(int m=1;m<=n;m++){
        for(int j=1;j<=c;j++){
            if(j-W[m]>=0){
                if(A[j-W[m],m-1]==true and A[j,m-1]==false){
                    A[j,m]=true;
                    D[j]=m;
                }
            }
            A[j,m]=A[j,m] or A[j,m-1];
        }
    }
    return(A,D);
}
```

```
8. set SKSS(int [] W, int c){
    // W consists of elements W[1],...,W[n]
    boolean A[c,n];
    int D[c];
    (A,D)=fillInArrays(W,c);
    max=0;
    for(int i=0;i<=c;i++){
        if(A[i,n]==true)max=i;
    }
    // Now we know what max is, we can look in D[] to
    // determine a set of weights which gives us weight max
    int weight=max;
```

```

S={};
while(weight!=0){
    add D[weight] to S;
    weight=weight-W[D[weight]];
}
return(S);
}

```

9. Assuming that set operations can be done simply, this algorithm again has running time $O(n \cdot c)$.

The Dynamic Programming Approach: Some points

1. In general, if we were given SKSS as our initial problem, to return an optimal structure, we would consider first the value related problem. That is, find the value of an optimal structure (the weight of the heaviest packed knap sack). We solve this using the dynamic programming technique outlined above. Once we have a solution to the optimal value problem, we then modify the algorithm by adding an array which records some of the decisions made in arriving at an optimal solution.
2. Dynamic programming is in general more powerful than greedy algorithms. It works well for problems that have

- **Optimal Substructure** Optimal solutions, contain within them; optimal solutions to subproblems. This is what gives rise to the recurrence relation defining the array $A[]$.
- **Overlapping Subproblems** Solutions to subproblems are used again and again in determining a solution. Consider what would have happened in the above example if we wrote a recursive subroutine for A and all the $W[i]=1$. Consider how many times particular values of $A[j,m]$ would be computed.

Say we wrote function $\text{funA}(j,m)$ which computes the value of $A[j,m]$. Such a function would (because of overlapping subproblems) continually compute the same values over and over again. Using $W[i]=1$ for all i , Computing $\text{funA}(j,m)$ would call $\text{funA}(j-1,m-1)$ and $\text{funA}(j,m-1)$, so 2 recursive calls. Each of these would require 2 calls to $\text{funA}(*,m-2)$ (a total of 4 calls to $\text{funA}(*,m-2)$). That is, $\text{funA}(j-1,m-1)$ calls $\text{funA}(j-2,m-2)$ and $\text{funA}(j-1,m-2)$. $\text{funA}(j,m-1)$ calls $\text{funA}(j-1,m-2)$ and $\text{funA}(j,m-2)$. Each call to $\text{fun}(*,m-2)$ would require 2 calls to $\text{funA}(*,m-3)$ (a total of 8 calls to $\text{funA}(*,m-3)$). This gives rise to something like $O(2^m)$ running time for computing $\text{funA}(j,m)$.