Advanced Computer Contest Preparation
Lecture 15

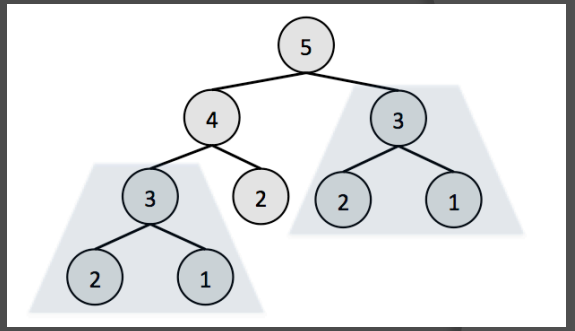
KNAPSACK PROBLEMS

Review of DP

- What kind of problems can DP be used on?
  - Optimization problems
  - Counting problems
- What properties must a problem have so that DP can be used?
  - Optimal substructure
  - Overlapping subproblems
- What techniques are applied to DP to achieve its runtime?
  - Memoization
  - DP state

DP Terms

- Subproblem
  - Another instance of the problem with smaller constraints
- Optimal substructure
  - A problem's solution depends on its subproblems' solutions
- Overlapping subproblems
  - The same subproblem might occur more than once
  - More than one problem depends on the same subproblem

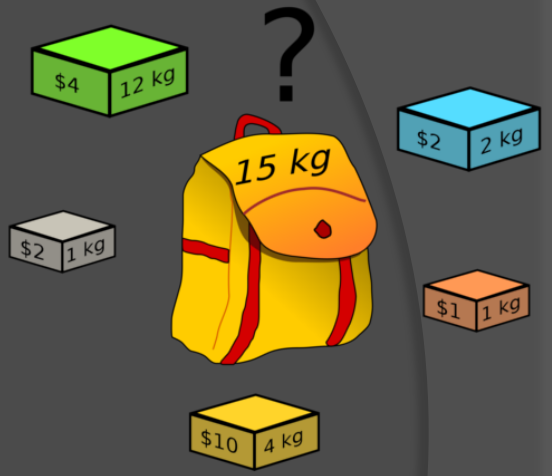


DP Terms

- Memoization
  - Storing answers to subproblems in a data structure so that they only need to be computed once
- DP state
  - A combination of important variables of each subproblem
  - Used to efficiently look up solutions

0-1 Knapsack Problem

- We have a knapsack with a maximum capacity of C
- There are N items, the i^{th} item has a value v_i and a weight of w_i
- **Each item can only be chosen once**
- We choose a combination of items such that the sum of their weights does not exceed C
- What is the maximum possible total value of a valid combination?



Greedy Approach

- The problem is an optimization problem, therefore, greedy might be possible
- What are possible greedy criterion?
- Most valued item first?
  - Counterexample:

$C=20$	Item 1	Item 2	Item 3
v_i	10	10	15
w_i	10	10	15

- Least valued item first?
  - Counterexample:

$C=20$	Item 1	Item 2	Item 3
v_i	1	19	20
w_i	20	19	20

- Highest value to weight ratio first?
  - Counterexample:

$C=20$	Item 1	Item 2	Item 3
v_i	21	21	40
w_i	10	10	11

Brute Force Approach

- Recursively generate all possible combinations, both valid and invalid
- For every combination:
  - Check if it exceeds limit
  - Find total value
- Runtime?
  - $O(2^N)$

DP Approach

- What is the overall problem?
  - What is the largest total value of a combination of the N items that does not exceed the capacity of the bag, C?
- What are the subproblems?
  - What is the largest total value of a combination of the first i items $for\ i = 0\ to\ N$ that does not exceed the capacity of the bag, j, $for\ j = 0\ to\ C$

DP Approach

- Is there optimal substructure?
- What does problem $p(i,j)$ depend on?
  - $p(i,j)$ depends on $p(i-1,j-w_i)$ and $p(i-1,j)$
  - We either choose the item, or we don't choose it
- What is the exact relationship between problem $p(i,j)$ and its subproblems?
  - $p(i,j) = max(p(i-1,j-w_i) + v_i, p(i-1,j))$
  - We want the maximum value, so we take the maximum of the two possibilities
- What are the base cases?
  - $p(i,j) = 0$ *if* $i = 0$ *or* $j = 0$

DP Approach

- Are there overlapping subproblems?
- Provide an example of where overlapping subproblems occur
  - $p(i,j)$ depends on $p(i-1,j-w_i)$ and $p(i-1,j)$
  - $p(i,j+w_i)$ depends on $p(i-1,j)$ and $p(i-1,j+w_i)$
  - $p(i-1,j)$ appears twice
  - Therefore, the problem exhibits overlapping subproblems

DP Approach

- DP state
  - There are two variables, the current item and the capacity
  - Therefore, we need a 2-D DP table, 1 dimension for each variable

Example

Capacity (C): 6

Item	v_i	w_i
1	1	2
2	3	3
3	5	1
4	2	5
5	6	3
6	10	5

Items allowed (i)

Capacity (j)

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	3	3	4	4
3	0	5	5	6	8	8	9
4	0	5	5	6	8	8	9
5	0	5	5	6	11	11	12
6	0	5	5	6	11	11	15

Pseudocode

```
int val[],weight[];
int dp[][];
int N,C;
//get input
for (int i = 1; i <= N; i++){
        for (int j = 0; j <= C; j++){
                if (j < weight[i]) dp[i][j] = dp[i-1][j];
                else dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i]] + val[i]);
        }
}
print dp[N][C];
```

Analysis

- How many states are there?
  - $O(NC)$
- How many subproblems does each state depend on?
  - $O(1)$
- What is the time complexity needed to compute the solution to a problem?
  - $O(1)$
- Therefore, the final time complexity is $O(NC)$

Finding a Valid Combination

- General case: after filling in the DP array, can we work backwards to print out a combination that gives us an optimal answer?
- In the case of 0-1 knapsack, can we print out a valid combination of items that has the maximal total value?

Method 1

- To each problem, attach the subproblem that was used to generate the solution
- After finishing the DP array, start from the final DP state and use attached subproblems to “backtrack” to the base cases
- Can store the best attached subproblem in a separate array
- Requires additional memory, but is normally quite fast
- Runtime is proportional to the number of elements in the combination

Pseudocode – Method 1

```
int dp[][];
state prv[][];
//fill in DP array
state cur = final DP state;
vector<state> comb;
comb.push_back(cur);
while(!isBaseCase(cur)){
      cur = prv[cur];
      comb.push_back(cur);
}
```

Method 2

- After filling the DP array, start from the final DP state
- At each problem, check its subproblems to see which one was used to generate the answer to the current problem
  - The satisfying subproblem will be the next problem
  - If there are multiple satisfying subproblems, choose any
- Repeat until base case is reached
- Insignificant additional memory required
- Runtime is proportional to the number of elements in the combination multiplied by the time complexity to solve one problem

0-1 Knapsack – Method 2

- The relationship between a problem and its subproblems is:

$$p(i,j) = max(p(i\text{-}1,j\text{-}w_i) + c_i, p(i\text{-}1,j))$$

- Therefore, either $p(i,j) = p(i\text{-}1,j)$

or

$$p(i,j) = p(i\text{-}1,j\text{-}w_i) + v_i$$

- Pick the subproblem that satisfies the condition
- If both subproblems are valid, choose any one of them

0-1 Knapsack – Method 2

Capacity (C) : 6

Item	v_i	w_i
1	1	2
2	3	3
3	5	1
4	2	5
5	6	3
6	10	5

Capacity (j)

Items allowed (i)	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	0	0	1	3	3	4	4
3	0	5	5	6	8	8	9
4	0	5	5	6	8	8	9
5	0	5	5	6	11	11	12
6	0	5	5	6	11	11	15

Thus, items 3 and 6 were used.

Pseudocode – 0-1, Method 2

```
int N,C;
int val[], weight[];
int dp[][];
//fill DP array
int item = N, cap = C;
vector<int> comb;
while(item != 0 && cap != 0){
	if (dp[item][cap] != dp[item-1][cap]){
		comb.push_back(item);
		cap -= weight[item];
	}
	item--;
}
```

Memory Optimization

- Current memory complexity is $O(NC)$
- Note that any problem $p(i,j)$ only depends on problems in the form $p(i-1,k)$, k is a non-negative integer
- Therefore, all other problems $p(x,k)$, $x < i-1$, can be discarded
- We can either:
  - Use 2 1-D arrays and swap them after processing an item
    - One array represents row i, the other represents row $i-1$
  - Use a single 1-D array, but iterate *for* $j = C$ *to* 0 *(or* w_i*)* instead
    - We do not want the answers to $p(i,j)$ affecting answers to $p(i,j+k)$, k is a positive integer
    - If we have just updated $p(i,j+k)$, the DP array will look like:

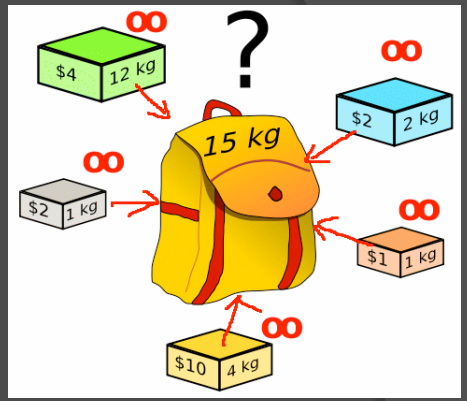
      $$..., p(i-1,j-1), p(i-1,j), p(i-1,j+1), ..., p(i-1,j+k-1), p(i,j+k), p(i,j+k+1), ...$$
- Note that we can no longer generate a combination
- New memory complexity is $O(C)$

Pseudocode – Mem. Optimization

```
int val[],weight[];
int dp[];
int N,C;
//get input
for (int i = 1; i <= N; i++){
        for (int j = C; j >= weight[i]; j--){
                dp[j] = max(dp[j],dp[j-weight[i]] + val[i]);
        }
}
print dp[C];
```

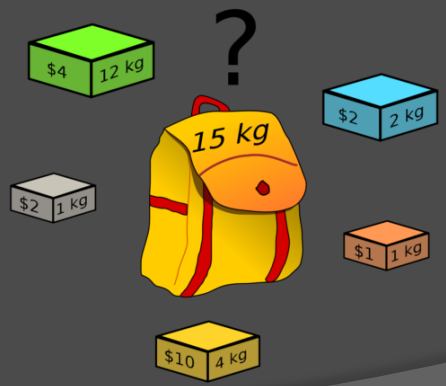
Unbounded Knapsack Problem

- We have a knapsack with a maximum capacity of C
- There are N items, the i^{th} item has a value v_i and a weight of w_i
- **Each item can be chosen an unlimited number of times**
- We choose a combination of items such that the sum of their weights does not exceed C
- What is the maximum possible total value of a valid combination?



0-1 Approach

- We can generate finite copies of an item
  - C/w_i copies of the i^{th} item
- Can be slow if C is large or w_i is small
- Worst case runtime is $O(NC^2)$



Removing Useless Items

- If $w_i > C$, remove item i
- If $w_i <= w_j$ and $v_i >= v_j$, remove item j (j is always worse than i)
- Does not improve asymptotic time complexity

Binary Idea

- Each item i has several copies
- Each copy k from $k = 0$ represents 2^k of that item
  - Value of copy k is $v_i \times 2^k$
  - Weight of copy k is $w_i \times 2^k$
- Perform 0-1 knapsack algorithm on the copies
- This works because any integer (number of copies) can be represented as the sum of powers of 2.
- Runtime is $O(NC \log C)$

Back to DP Formulation

- The first two methods (0-1 approach, removing items) do not asymptotically improve the runtime
- The third approach is faster, but can we do better?
- Go through the DP formulation process again
- Note that unbounded knapsack, like 0-1 knapsack, has optimal substructure and overlapping subproblems
- Problems, subproblems, and DP states are the same as 0-1 knapsack

DP Approach

- What does problem $p(i,j)$ depend on?
  - $p(i,j)$ depends on $p(i-1,j-w_i)$, $p(i,j-w_i)$, and $p(i-1,j)$
  - We either choose the first copy of the item, another copy of the item, or we don't choose it at all

DP Approach

- What is the exact relationship between problem $p(i,j)$ and its subproblems?
  - $p(i,j) = max(p(i,j-w_i) + v_i, p(i-1,j))$
  - $p(i-1,j-w_i)$ is not necessary as it is an immediate subproblem to $p(i-1,j)$
    - If $p(i-1,j-w_i)$ is the most optimal, its value would be in $p(i-1,j)$
  - We want the maximum value, so we take the maximum of the two possibilities
- What are the base cases?
  - $p(i,j) = 0$ *if* $i = 0$ *or* $j = 0$

Example

Capacity (C): 6

Item	v_i	w_i
1	1	2
2	3	3
3	5	4
4	2	3
5	6	4
6	7	5

Items allowed (i)

Capacity (j)

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	2	2	3
2	0	0	1	3	3	4	6
3	0	0	1	3	5	5	6
4	0	0	1	3	5	5	6
5	0	0	1	3	6	6	7
6	0	0	1	3	6	7	7

Pseudocode

```
int val[],weight[];
int dp[][];
int N,C;
//get input
for (int i = 1; i <= N; i++){
        for (int j = 0; j <= C; j++){
                if (j < weight[i]) dp[i][j] = dp[i-1][j];
                else dp[i][j] = max(dp[i-1][j], dp[i][j-weight[i]] + val[i]);
        }
}
print dp[N][C];
```

Memory Optimization

- Like 0-1 knapsack, we can reduce the memory complexity from $O(NC)$ to $O(C)$ by compressing the DP array into 1 dimension
- Again, we cannot generate a combination if we do this
- Because $p(i,j) = max(p(i,j-w_i) + v_i, p(i-1,j))$, we can iterate $for\ j = 0\ to\ C$
- We want $p(i,j)$ to affect $p(i,j+k)$, k is a positive integer
  - If we have just updated $p(i,j)$, the array will look like:

    $..., p(i,j-1), p(i,j), p(i-1,j+1), ..., p(i-1,j+k-1), p(i-1,j+k), p(i-1,j+k+1), ...$

Pseudocode – Mem. Optimization

```
int val[],weight[];
int dp[];
int N,C;
//get input
for (int i = 1; i <= N; i++){
        for (int j = weight[i]; j <= C; j++){
                dp[j] = max(dp[j],dp[j-weight[i]] + val[i]);
        }
}
print dp[C];
```

THANK YOU!