Lecture 23
Knapsack Solved All Ways

EECS 281: Data Structures & Algorithms

http://xkcd.com/287

Knapsack Problem

- A thief robbing a safe finds it filled with *N* items
  - Items have various weights or sizes
  - Items have various values
- The thief has a knapsack of capacity *M*
- Problem: Find maximum value the thief can pack into his/her knapsack that does not exceed capacity *M*

Example: Knapsack

- Assume a knapsack with capacity *M* = 11
- There are *N* different items, where
  - Items have various sizes
  - Items have various values

Size	1	2	5	6	7
Value	1	6	18	22	28

- Return *maxVal* (max value the thief can carry)

Variations on a Theme

- Each item is unique
  - Known as the 0-1 Knapsack Problem
  - Must take an item (1) or leave it behind (0)
- Finite amount of each item (explicit list)
- Infinite amount of each item
- Fractional amount of each item
- Using weight (w_i) instead of size

Solve Knapsack Problem

Using All (Most?) Algorithmic Approaches

- Brute Force
- Greedy
- Dynamic Programming
- Backtracking
- Branch and Bound

Knapsack: Brute-Force

- Generate possible solution space
  - Given an initial set of *N* items
  - Consider all possible subsets
- Filter feasible solution set
  - Discard subsets with *setSize* > *M*
- Determine optimal solution
  - Find *maxVal* in feasible solution set

Brute-Force Pseudo-Code

```
bool array possSet[1..N] (0:absent,1:present)
int maxVal = 0
for int i = 1 to 2^N
   possSet[] = genNextPower(N)
   int setSize = findSize(possSet[])
   int setValue = findValue(possSet[])
   if setSize <= M and setValue > maxVal
      bestSet[] = possSet[]
      maxVal = setValue
return maxVal
```

Brute-Force Efficiency

- Generate possible solution space
  - Given an initial set of N items
  - Consider all possible subsets
  - $O(2^N)$
- Filter feasible solution set
  - Discard subsets with *setSize* > *M*
  - $O(N)$
- Determine optimal solution
  - Find *maxVal* in feasible solution set
  - $O(N)$

$O(N2^N)$

Greedy Approach

Approaches

- Steal ***highest value*** items first
  - Might not work if large items have high value
- Steal ***lowest size (weight)*** items first
  - Might not work if small items have low value

- What to do? What to do?

Greedy

Approaches

- Sort items by ***ratio*** of value to size
- Choose item with highest ***ratio*** first

Will this approach always provide optimal solution?

Example: Greedy Knapsack

- Assume a knapsack with capacity *M* = 11
- There are *N* different items, where
  - Items have various sizes
  - Items have various values

Size	1	2	5	6	7
Value	1	6	18	22	28
Ratio	1	3	3.6	3.67	4

Greedy Pseudo-Code

Input: integer capacity M, integer array size[1..N], integer array val[1..N]

Output: integer maxVal which is maximum value a knapsack of size M can carry

```
maxVal = 0, currentSize = 0
ratio[] = buildRatio(value[], size[])
sortedRatio[] = sortRatio(ratio[])
// Sort size[] and value[] arrays by ratio
```

Greedy Pseudo-Code

```
for int i = 1 to N //sorted by ratio
   if size[i] + currSize <= M
      currSize = currSize + size[i]
      maxVal = maxVal + value[i]

return maxVal
```

Greedy Efficiency

- Sort items by ratio of value to size
  - $O(N \log N)$
- Choose item with highest ratio first
  - $O(N)$

$$O(N \log N) + O(N) \Rightarrow O(N \log N)$$

Fractional Knapsack: Greedy

- Now suppose that we can take portions of an item
- What happens if we apply a Greedy strategy?
- Is it optimal?

Dynamic Programming

- Will consider three approaches
  - Simple Recursive (*non-DP*)
  - Top-Down
    - Recursive DP
  - Bottom-Up
    - Iterative DP

- Notes:
  - Pseudo-code style changes slightly
  - Assume infinite amount of each item

Recursive Approach

- For each item
  - Place item in the knapsack
  - Find the optimal packing for a ‘smaller’ knapsack
  - Remember the best packing
- Algorithm is direct recursive solution and takes exponential time

Recursive Pseudocode

```
Algorithm knapsack(int capacity)
  max_val = 0
  for each item in N
    space_rem = capacity - item.size
    if (space_rem >= 0)
      new_val = knapsack(space_rem) + item.val
      if (new_val > max_val)
        max_val = new_val
  return max_val
```

Recursive Implementation

```
int knapsack(int cap, Item items[], int n) {
    int space, max, t;
    for (int i = 0, max = 0; i < n; i++)
        if ((space = cap - items[i].size) >= 0 ) {
            t = knapsack(space, items, n) + items[i].val;
            if (t > max)
                max = t;
        } // if
    return max;
} // knapsack()
```

Dynamic Programming: Top-down

```
int knapTD(int capacity) {
  if (maxKnown[capacity] is known)
    return maxKnown[capacity];
  max_val = 0;
  for (auto item : N) {
    space_rem = capacity - item.size;
    if (space_rem >= 0) {
      new_val = knapTD(space_rem) + item.val;
      if (new_val > max_val)
        max_val = new_val;
    } } // if and for
  maxKnown[capacity] = max_val;
  return max_val;
} // knapTD()
```

- First check if requested value has been calculated
- Otherwise calculate and save additional values
- Run time is $O(MN)$

Dynamic Programming: Bottom-up

Run time is $O(MN)$

```
int knapBU(int cap) {
    int cap_rem;
    int V[cap + 1];
    V[0] = 0;
    for(int i = 1; i <= cap; i++) {
        V[i] = V[i - 1];
        for (int j = 0; j < N; j++) {
            cap_rem = i - item[j].size;
            if ( (cap_rem >= 0)
                && ((item[j].value + V[cap_rem]) > V[i]))
                V[i] = item[j].value + V[cap_rem];
        } // for j
    } // fori
    return V[cap];
} // knapBU()
```

Summary: Knapsack Problem

- Solved using many different approaches
  - Brute-Force
  - Greedy
  - Dynamic Programming
  - Backtracking
  - Branch and Bound