Algorithm Design Techniques

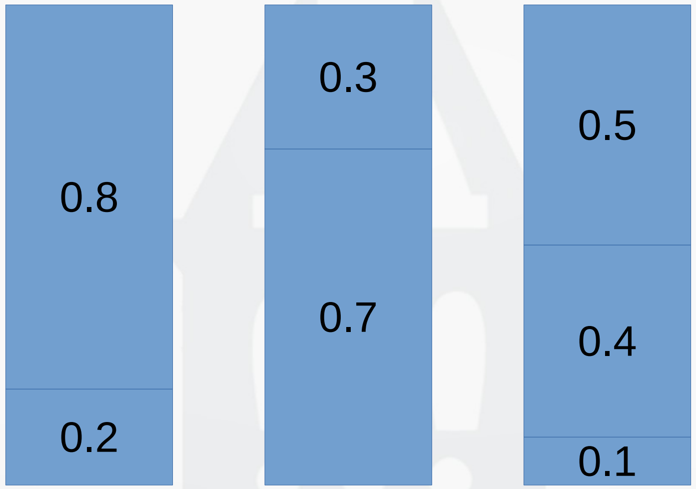
Greedy Algorithms
Dynamic Programming

Approximate Bin Packing

- Think about packing your belongings into boxes for moving
- n items of sizes $s_1, s_2, \ldots, s_n$
- $0 < s_i <= 1$
  - this means all items are greater than size 0 and no larger than size 1
- Goal: Pack into fewest number of bins of size 1
- NP-Complete problem: But we can use greedy algorithms to produce *good* solutions

Optimal Packing – Example

- Input sizes: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8



Bin Packing – Online vs Offline Algorithms

- Online
  - Process one item at a time
  - Cannot move an item once it is placed
  - Complexity? Polynomial

- Offline
  - Look at all items before placing the first item
  - Complexity? Exponential

Bin Packing – Online Algorithms

- Cannot guarantee optimal solution
  - Problem: Don’t know when the input will end
  - M small items ½ - epsilon; M large items ½ + epsilon
  - Can fit into M bins; 1 large and 1 small in each bin
  - If all small come first, place in M separate bins
    - If input is only M small items, have used at least twice as many bins as necessary
  - It has been shown there are inputs that force any online bin-packing algorithm to use at least 4/3 the optimal number of bins

Bin Packing – Online Algorithms

- Three approaches
  - Next Fit
  - First Fit
  - Best Fit

Bin Packing – Next Fit

- Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8
- Algorithm
  - If item fits in a bin with last item, place it there
  - Else, place in new bin

- Bin 1: 0.2, 0.5 (total 0.7)
- Bin 2: 0.4 (total 0.4)
- Bin 3: 0.7, 0.1 (total 0.8)
- Bin 4: 0.3 (total 0.3)
- Bin 5: 0.8 (total 0.8)

- Complexity? linear
- Let M be the optimal number of bins required to pack a list I of items. Then next fit never uses more than 2M bins
  - At most, half of the space is wasted

Bin Packing – First Fit

- Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8
- Algorithm
  - Scans all bins and places item in first bin large enough to hold it
  - If no bin is large enough, use a new bin

- Bin 1: 0.2, 0.5, 0.1 (total 0.8)
- Bin 2: 0.4, 0.3 (total 0.7)
- Bin 3: 0.7 (total 0.7)
- Bin 4: 0.8 (total 0.8)

- Complexity? polynomial
- Let M be the optimal number of bins required to pack a list l of items. Then first fit never uses more than ceil(17/10)M bins

Bin Packing – Best Fit

- Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8
- Algorithm
  - Scans all bins and places item in the bin with the tightest fit (will be fullest after item is placed)
  - If no bin is large enough, use a new bin

- Bin 1: 0.2, 0.5, 0.1 (total 0.8)
- Bin 2: 0.4 (total 0.4)
- Bin 3: 0.7, 0.3 (total 1.0)
- Bin 4: 0.8 (total 0.8)

- Complexity? polynomial
- Same performance (number of bins used) as first fit

Bin Packing – Offline

- Have a lot more choices, by doing some processing first, then packing; not trying for optimal solution, but fast, good solution.
- Here is an idea (or ideas)
  - Sort items (in decreasing order) for easier placement of large items
  - Then apply either first fit or best fit algorithm
  - Let M be the optimal number of bins required to pack a list of I items. Then first fit decreasing never uses more than ((11/9) + 4)M bins

The Knapsack Problem

a variant of bin packing

The Knapsack Problem

- The classic Knapsack Problem is:
  - A thief breaks into a store and wants to fill their knapsack of capacity K with items of as much value as possible
  - Decision version: Does there exist a collection of items that fits into the knapsack and whose total value is >= W?

The Knapsack Problem

- Input
  - Capacity K
  - n items with weights w_i and values v_i
- Output
  - A set of items S such that
    - the sum of weights of items S is at most K
    - the sum of the values of items in S is maximized

- Rather than optimizing based on size only, now optimizing on two parameters, weight and value

The Knapsack Problem – Fractional Version

- Fractional Knapsack Problem: items can be picked up partially
- The thief’s knapsack can hold 100 gms and has to choose from
  - 30 gms of gold dust at $1000/gm
  - 60 gms of silver dust at $500/gm
  - 30 gms of platinum dust at $1500/gm

The Knapsack Problem – Fractional Version

- Fractional Knapsack Problem: items can be picked up partially
- The thief's knapsack can hold 100 gms and has to choose from
  - 30 gms of gold dust at $1000/gm
  - 60 gms of silver dust at $500/gm
  - 30 gms of platinum dust at $1500/gm
- Solution (this is easy)
  - 30 gms of platinum
  - 60 gms of gold
  - 10 gms of silver

The Knapsack Problem – Fractional Version

- Greedy algorithm
  - Sort the items in increasing order of value/weight ratio (cost effectiveness)
  - Select from the sorted items until knapsack is full
    - If next item cannot fit, break it (fractional part) to exactly fill the knapsack

- Notice this is also an optimal solution!

The Knapsack Problem – 0-1 Version

- An item can either be selected or left, it cannot be picked partially
- For example, gold bar, diamond ring, stereo, computer, cell phone

The Knapsack Problem – 0-1 Version

- Capacity of 100, list of items
  - X1: weight (41), value (410), value/weight = 10
  - X2: weight (70), value (630), value/weight = 9
  - X3: weight (60), value (480), value/weight = 8
  - X4: weight (40), value (280), value/weight = 7

The Knapsack Problem – 0-1 Version

- Capacity of 100, list of items
  - X1: weight (41), value (410), value/weight = 10
  - X2: weight (70), value (630), value/weight = 9
  - X3: weight (60), value (480), value/weight = 8
  - X4: weight (40), value (280), value/weight = 7
- Greedy by unit value: X1 + X4 = 690 value
  - After picking X1, only X4 is possible

The Knapsack Problem – 0-1 Version

- Capacity of 100, list of items
  - X1: weight (41), value (410), value/weight = 10
  - X2: weight (70), value (630), value/weight = 9
  - X3: weight (60), value (480), value/weight = 8
  - X4: weight (40), value (280), value/weight = 7
- Greedy by unit value: X1 + X4 = 690 value
  - After picking X1, only X4 is possible
- Greedy by largest size: X2 = 630 value

The Knapsack Problem – 0-1 Version

- Capacity of 100, list of items
  - X1: weight (41), value (410), value/weight = 10
  - X2: weight (70), value (630), value/weight = 9
  - X3: weight (60), value (480), value/weight = 8
  - X4: weight (40), value (280), value/weight = 7

- Greedy by unit value: X1 + X4 = 690 value
  - After picking X1, only X4 is possible
- Greedy by largest size: X2 = 630 value
- Greedy by smallest size: X4 + X1 = 690 value

The Knapsack Problem – 0-1 Version

- Capacity of 100, list of items
  - X1: weight (41), value (410), value/weight = 10
  - X2: weight (70), value (630), value/weight = 9
  - X3: weight (60), value (480), value/weight = 8
  - X4: weight (40), value (280), value/weight = 7

- Greedy by unit value: X1 + X4 = 690 value
  - After picking X1, only X4 is possible
- Greedy by largest size: X2 = 630 value
- Greedy by smallest size: X4 + X1 = 690 value

- Best choice (exhaustive search): X3 + X4 = 760 value

The Knapsack Problem – 0-1 Version

- The exhaustive solutions tries all possibilities
  - This is necessary, as the other (simpler) greedy methods don’t give optimal results
  - Recursively try all possibilities, but end up re-computing sub-problems repeatedly

- Guess what, compute all sub-problems in advance and use those: ***Dynamic Programming***!

The Knapsack Problem – 0-1 Version

- Let V(i, w) is the value of the set of items from the first i items that maximizes the value subject to the constraint that the sum of the values of the items in the set is <= w
- Value of the original problem corresponds to V(n, K)

- Recurrence Relation
  - $V(i, w) = \max(V(i - 1, w - w_i) + v_i, V(i - 1, w))$
    - First term corresponds to the case when x_i is included in the solution
    - Second term corresponds to the case when x_i is not included
  - V(0, w) = 0 (no items to choose from)
  - V(i, 0) = 0 (no weight allowed)

The Knapsack Problem – 0-1 Version

weight

which numbered item can be used

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

- Cell [i, j] : given that you can use items number i or less and up to j weight
  - what is the best value (for weight) you can get?

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

The Knapsack Problem – 0-1 Version

weight

which numbered item can be used

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

- If we don't put any items in, then no value; that should be obvious

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

The Knapsack Problem – 0-1 Version

- Let K = 5 (size of knapsack), and n = 1 (number of items)

weight

which numbered item can be used

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

- We can get a value of 0 or 10; select item 0 or 1
- Once item 1 is selected, have a remaining K = 0

The Knapsack Problem – 0-1 Version

- Let K = 10 (size of knapsack), and n = 2 (number of items)

weight

which numbered item can be used

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50

i	1	2
v_i	10	40
w_i	5	4

- If we select item 2, have a remaining K = 6; total value is 40
- Then select item 1; total value is 40 + 10 = 50

The Knapsack Problem – 0-1 Version

- Let K = 10 (size of knapsack), and n = 2 (number of items)

weight

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50

which numbered item can be used

i	1	2
v_i	10	40
w_i	5	4

- If we select item 1, have a remaining K = 5; total value is 10
- Then select item 2; total value is 10 + 40 = 50

The Knapsack Problem – 0-1 Version

- Let K = 10 (size of knapsack), and n = 2 (number of items)

weight

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70

which numbered item can be used

- If we select item 3, have a remaining K = 4; total value is 30
- Then select select item 2; total value is 30 + 40 = 70

i	1	2	3
v_i	10	40	30
w_i	5	4	6

The Knapsack Problem – 0-1 Version

- Let K = 10 (size of knapsack), and n = 2 (number of items)

weight

	0	1	2	3	4	5	6	7	8	9	10
i=0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

which numbered item can be used

- If we select item 4, K = 7; total value = 50
- Then select item 2, K = 3, total value = 50 + 40 = 90
- We still have K of 3, but no remaining items small enough to select

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

The Knapsack Problem – 0-1 Version

- Complexity (with respect to time)
  - Depends on the size of the knapsack, instead of only the number of elements
- O(nK)
  - n is number of items
  - K is size (capacity) of the knapsack