Shuffle a list

Given a list, put items into a random order

23,22,49,25,43,23,5,31,43,27,21,45,43,16,5,21,18,27,39,18,21,7,42,28,21,19

- For each position, grab a random item and put it in that position
  - add(position, remove(random))

  vs
  - swap [set(position, set(index, get(position))] or Collections.swap(…)

- Use the built-in shuffle!
  - Collections.shuffle(list)

Shuffle a list

n times

- For each position from n-1 to 0,
  - choose a random index ≤ position — **n x O(1)**
  - item = remove(index) — **n x O(n)**
  - add(position, item) — **n x O(n)**

Total: $O(n^2)$

n times

- For each position from n-1 to 0,
  - choose a random index <= position — **n x O(1)**
  - swap(index, position) — **n x O(1)**

Total: O(n)

Combinations

Given

- a set of packages of different weights, and
- A shipping pallet/container/box that has a maximum weight

Find

- combination of packages that has the maximum weight but not over the target

Solution:

- build all possible combinations by adding packages until over the target weight.
- remember the best combination

Combinations

- packet 0 yes no
- packet 1 yes no
- packet 2 yes no
- packet 3 yes no
- packet 4 yes no
- packet 5 yes no
- packet 6 yes no
- packet 7 yes no
- packet 8 yes no
- packet 9 yes no
- packet 10 yes no
- packet 11 yes no

Algorithm1

- Start with an empty combination
- initialise bestCombination and bestTotal to 0;
- Find combinations using additional packets from index 0.

- To find combinations using additional packets from index i…:
  - if including packet i would still be within target
    - add it to the current combination
    - if it beats the current best, then remember total and combination.
    - find combinations using additional packets from index i+1… < RECURSIVE CALL
    - remove it from the current combination
  - find combinations using additional packets from index i+1… < RECURSIVE CALL

Algorithm2

- use a binary integer to represent a combination:

  10001110011010 => packets 1, 5, 6, 7, 10, 11,13

- Step through all numbers from 1 to 1111111111…111 to try all combinations

  - **for** combn from 1 to 11111111…1
    - work out total weight of combination
    - **if** weight <= target and weight > best so far
      - remember weight and combn

Cost of Algorithm2

- if n packets, then max combination represented by 2^n

  - **for** combn from 1 to max — with n packets, max = 2^n
    - work out total weight of combination — O(n)
    - **if** weight <= target and weight > best so far — O(1)
      - remember weight and combn — O(1)

  } 2^n times

Cost of Algorithm1

- Start with an empty combination
- initialise bestCombination and bestTotal to 0;
- Find combinations (target, packets, combn, 0, 0).

- To find combinations (target, packets, combSoFar, index, totalSoFar)
  - packet = packets[index]
  - newTotal= totalSoFar+ packet.weight
  - if newTotal <= target
    - add packet to the combSoFar
    - **if** newTotal > bestTotal then bestTotal=newTotal, bestCombination = combSoFar
    - find combinations(target, packets combSoFar, index+1, newTotal)
    - remove packet from combSoFar
  - find combinations(target, packets, combSoFar, index+1, totalSoFar)

Cost of Algorithm 1

- Cost(n) = cost of finding with n remaining packets to try

- Cost(1) = O(1)
- Cost(n) = O(1) + Cost(n-1) + Cost(n-1)
  = 2 Cost(n-1) + O(1)
  = 2(2Cost(n-2) + O(1)) + O(1)

The cost approximately doubles when n increase by 1 => O(2^n)