Data Structures and Algorithms

XMUT-COMP 103 - 2024 T1

Recursion and Algorithm Complexity

Mohammad Nekooei

School of Engineering and Computer Science

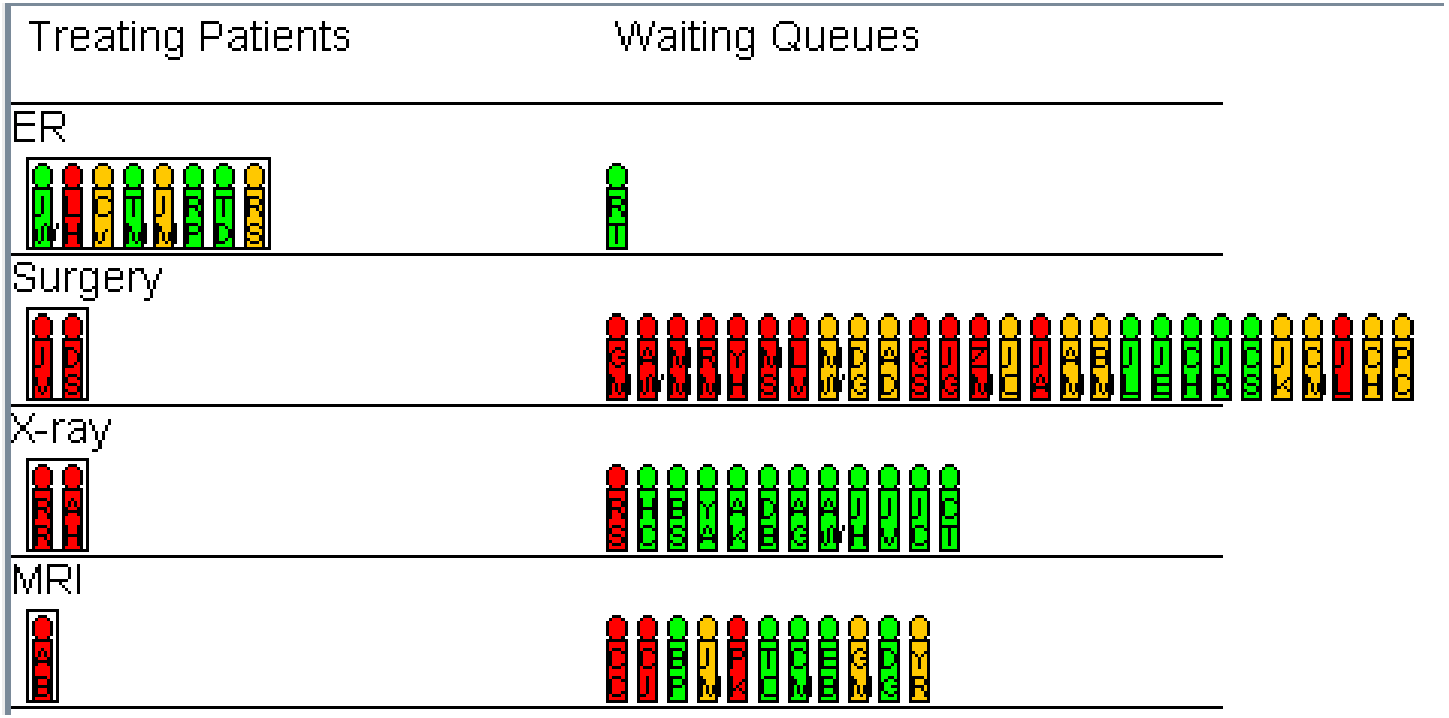
Victoria University of Wellington

Assignment 3

- Hospital simulation
  - Tick based simulation
  - Queues, priorityqueues, sets, lists of queues, maps,….
- MineSweeper
  - recursion!
- MedicalCenter

Aside: Priority Queues

- Why aren't the Patients in priority order when waiting in the queue?



- Note:
  - The front item in the priority queue is always the highest priority.
  - Higher priority items tend to be closer to the front.
  - But they aren't kept in exact order.
  - Priority Queues keep the items in a partially ordered tree structure

    ⇒ more efficient to add and remove items [O(log n) instead of O(n)]

    more details later in the course.

Analysing Costs (in general)

How can we determine the costs of a program?

- **Time:**
  - Run the **program** and count the milliseconds/minutes/days.
  - Count number of steps/operations the **algorithm** will take.
- **Space:**
  - Measure the amount of memory the **program** occupies.
  - Count the number of elementary data items the **algorithm** stores.
- Applies to Programs or Algorithms? *Both.*
  - programs → "benchmarking"
  - algorithms → "analysis"

What is a good algorithm?

Obviously needs to do what is expected consistently. However most problems can be solved in many ways. What is most important?

- Clarity - easy to read/implement
- Efficiency - the cost of running it

Clarity is relatively simple to measure. Find somebody else to read you code.

But how do we measure efficiency of an algorithm?

Benchmarking: program cost

Measure:

- actual programs, on real machines, with specific input
- measure elapsed time
  - System.currentTimeMillis ()
    → time from the system clock in milliseconds
- measure real memory usage

Problems:

- what input? ⇒ use large data sets
  don't include user input
- other users/processes? ⇒ minimise
  average over many runs
- which computer? ⇒ specify details

- how to compare cross-platform? ⇒ measure cost at an abstract level

Analysis: Algorithm “complexity”

- Abstract away from the details of
  - the hardware, the operating system
  - the programming language, the compiler
  - the specific input

- Measure number of “steps” as a function of the data size
  - best case (easy, but not interesting)
  - worst case (usually easy)
  - average case (harder)

- The precise number of steps is not required
  - $3.47\ n^2 - 67n + 53$ steps
  - $3n \log(n) + 5n - 3$ steps

- Rather, we are interested in how the cost grows with data size on large data

Big-O Notation

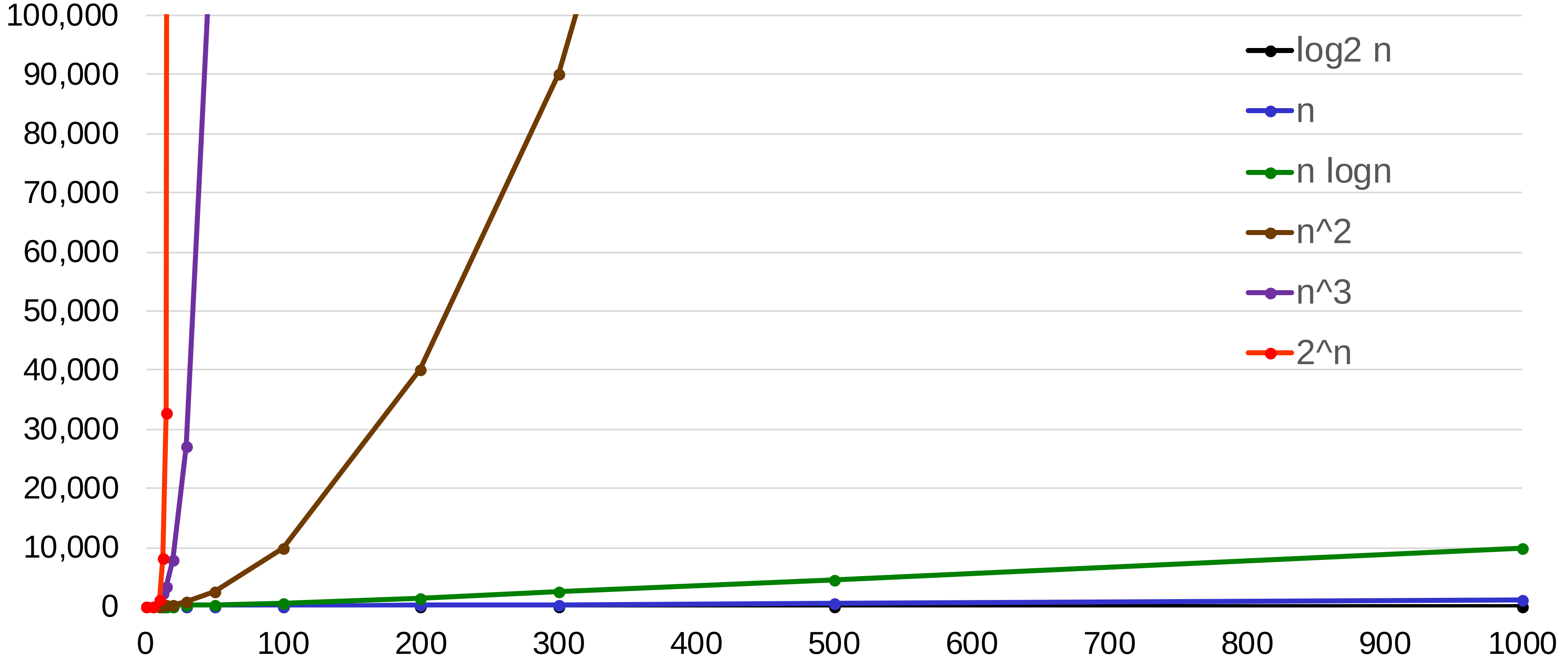
- "Asymptotic cost", or "big-O" cost describes how cost grows with large input size
- Only care about large input sets
  - Lower-order terms become insignificant for large n

- We care about how cost grows with input size
  - Don't care about constant factors
  - Multiplication factors (3, 102, 3 and 12 below) don't tell us how things "scale up"
  - Lower-order terms become insignificant for large n

$3.47\ n^2 + 102n + 10064$ steps → $O(n^2)$

$3n \log n + 12n$ steps → $O(n \log n)$

How the different costs grow



n: size of input

Big-O classes

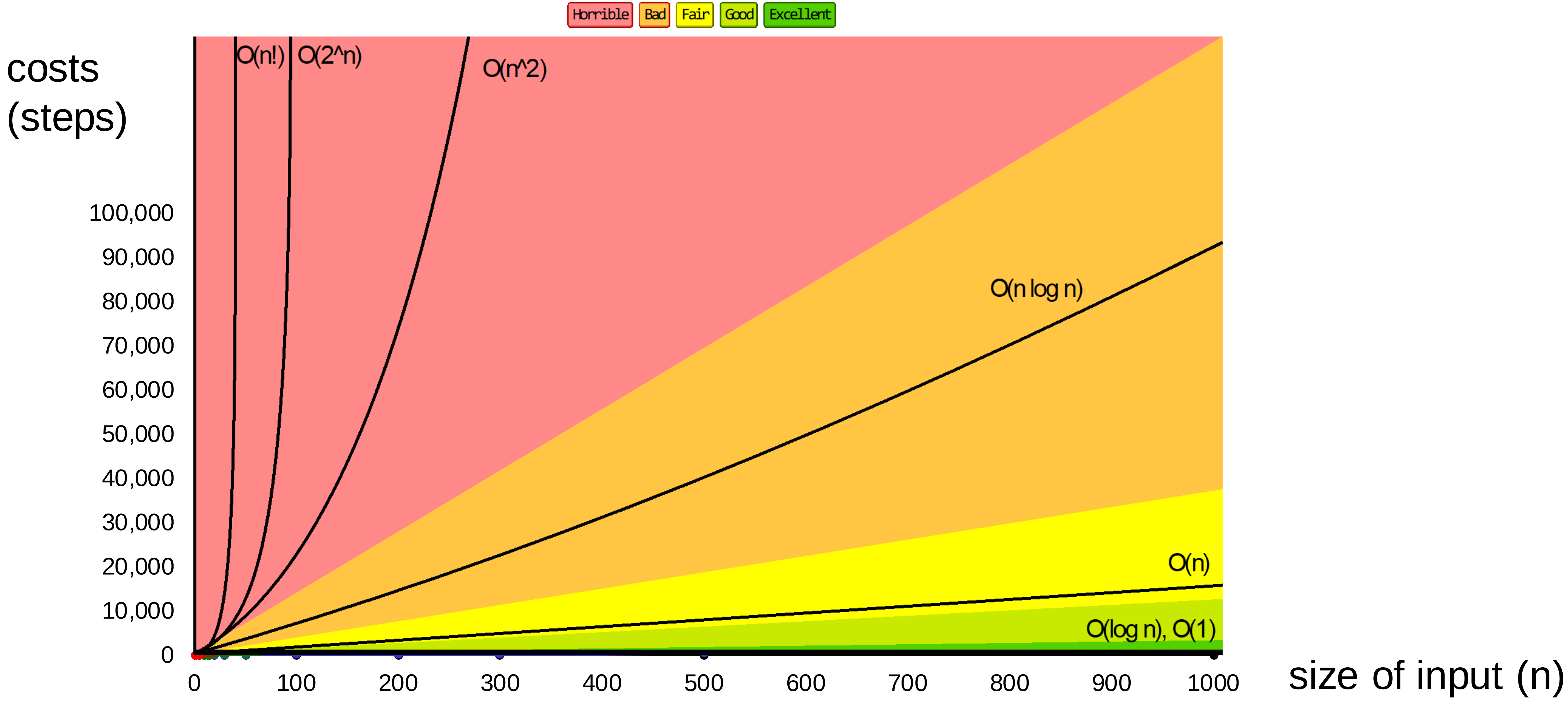
- Examples:
  - O(1) constant: cost is independent of n : Fixed cost!
    - Retrieve/insert in regular arrays, hashmap operations
  - O(log n) logarithmic: cost grows by 1, when n doubles : *almost constant*
    - Traversing a binary tree, some divide-conquer algorithms
  - O(n) linear: cost grows *linearly* with n :
    - Find a value in array, do something to all elements in an array, adding in the middle of ArrayList
  - O(n log n) log linear: cost grows a bit more than linear: *Slow growth!*
    - Good sorting algorithms (merge, quick, heap sort). Complex divide-conquer algorithms

Big-O classes

- Examples continued:
  - $O(n^2)$ quadratic: costs x 4 when n doubles: *limits problem size*
    - Do something to all elements in a 2d array. Nested loops
  - $O(n^c)$, $c>2$ polynomial: *limits problem size even more*
    - Do something to all elements in a 3d array. Many nested loops
  - $O(2^n)$ exponential: costs doubles when n increases by 1: *severely limits problem size*
    - Route finding, e.g. travelling salesman problem
  - Super-exponential: e.g.$O(n!)$ *don’t even think about it…*

How the different costs grow

- For growing n, the costs grow slower or faster depending on the cost function



Manageable problem sizes

- *How large can the data be?*
  - Assume one step takes one microsecond (i.e., 10^{-6} sec) on the computer
  - Then the following problem sizes can be handled by an algorithm in a given Big-O class within a given time unit

Time	**1 min**	**1 h**	**1 day**	**1 week**	**1 year**
$O(n)$	10^7	10^9	10^{11}	10^{12}	10^{13}
$O(n \log n)$	10^6	10^8	10^9	10^{10}	10^{12}
$O(n^2)$	10^4	10^5	10^5	10^6	10^7
$O(n^3)$	10^2	10^3	10^3	10^4	10^4
$O(2^n)$	25	31	36	39	44

How much is 1 year ? about half a million sec

What is a “step”?

- Any important actions that are at the centre of the algorithm
  - comparing data
  - moving data
  - anything you consider to be “expensive”
  - Doesn’t depend on size of data

```
public  E  remove (int index){
     if (index < 0 || index >= count) throw new ….Exception();
     E ans = data[index];
     for (int i=index+1; i< count; i++)
        data[i-1]=data[i];                  ← Key Step
     count--;
     data[count] = null;
     return ans;
   }
```

What’s a step: Pragmatics

- Count the most expensive actions?
  - Adding 2 numbers is cheap
  - Raising to a power is not so cheap
  - Comparing 2 strings *may* be expensive
  - Reading a line from a file *may* be very expensive
  - Waiting for input from a user or another program may take forever…

- Remember the Big (O) picture
- Sometimes we need to know about how the underlying operations are implemented in the computer to choose well (NWEN241/342).

Costs of Standard Collection classes

- ArrayList: O(1): clear, add, set, remove from end:
  O(n): add, remove, contains, Collections.reverse, .shuffle
  O(n log(n)) Collections.sort,

- ArrayDeque: O(1): clear, push, pop, offer, poll, add/remove First/Last:
  O(n): contains, remove(obj)
- PriorityQueue: O(log(n)): offer, poll

- HashSet: O(1): add, remove, contains
- TreeSet: O(log(n)): add, remove, contains

- HashMap: O(1): clear, containsKey, put, get
  But depends on the cost of hashCode

Example Algorithms

- Finding the Mode of a set of numbers
- Shuffle a List
- Find combinations of items to fill a pallett
- Find permutations of letters to make words.
  - (fix the dictionary!)

Finding the Mode of a list

- Mean = total/count
- Median = middle value, separating top 50% from bottom 50%
- Mode = most frequent number.

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

Algorithm:

- set *mode* to the first number and *modeCount* to 1
- **for** each value in the list:
    - step through the list to count how many times value occurs in the list
    - **if** *count* > *modeCount* **then** set *mode* and *modeCount* to *value* and *count*

What's the cost if there are *n* numbers?

Mode: the bad way

Analysis

```
public int mode(List<Integer>numbers){
    int mode = numbers.get(0);             1 x O(1)
    int modeCount = 1;                     1 x O(1)
    for (int value : numbers){
        int count = 0;                     n x O(1)
        for (int other : numbers){
            if (other == value) {          nxn x O(1)
                count++;                   nxn x O(1)
            }
        }
        if (count > modeCount) {           n x O(1)
            mode = value;                  1 … n x O(1)
            modeCount = count;             1 … n x O(1)
        }
    }
    return mode;                           1 x O(1)
}
```

n*n iterations

n iterations

$O(n^2)$

Finding the Mode of a list faster

- Much easier to see if the list is sorted in 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

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

- Algorithm
  - sort the list
  - set *mode* to first number and *modeCount* to 1
  - set *count* to 1
  - **step** through the list from index 1
    - **if** the number is the same as the previous number, **then** increment *count*
    - **else**
      - **if** *count* > *modeCount*, **then** set *mode* and *modeCount* to previous value and *count*
      - reset *count* to 1
  - **if** *count* > *modeCount*, **then** set *mode* and *modeCount* to previous value and *count*

What's the cost if there are *n* numbers?

Finding the Mode of a list faster

- Algorithm

```
                                                                           Analysis
                  • sort the list                                          1 x O(n log(n))
                  • set mode to first number and modeCount to 1            1 time x O(1)
                  • set count to 1                                         1 time x O(1)
                ┌ • step through the list from index 1
                │     • if number is same as previous number, then         n times x O(1)
                │         • increment count                                1 … n times x O(1)
n iterations ───┤     • else
                │         • if count > modeCount, then                     n … 1 times x O(1)
                │             • set mode and modeCount to previous number and count   n … 1 times O(1)
                │         • reset count to 1                               n … 1 times x O(1)
                └ • if count > modeCount, then                             1 time x O(1)
                      • set mode and modeCount to previous value and count
```

Total: O(n log(n))

Finding the Mode of a list even faster

- Count using a map to count without sorting:

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

5-2 7-1 16-1 18-2 19-1 21-4 22-1 23-2 25-1
27-2 28-1 31-1 39-1 42-1 43-3 45-1 49-1

What's the cost if there are *n* numbers?

- Algorithm
  - **for** each value in the list
    - **if** the value is in the map, **then** increment the associated *count*
    - **else** add the value to the map with an associated count of 1.
  - **for** each key in map,
    - **if** associated count > *modeCount*, **then** set *mode* and *modeCount* to key and count

Finding the Mode of a list even faster

- Algorithm

```
                                                                          Analysis
           for each value in the list
               if the value is in map, then                               n x O(1)      containskey(key)
n times            increment the associated count                         1…n x O(1)    get(..) & put(..)
               else
                   add value to map with associated count =1.             n…1 x O(1)    put(key, 1)
           for each key in map,                                           O(1)          get all keys
n times        if associated count > modeCount, then                      n x O(1)      get(key)
                   set mode and modeCount to key and count                1…n x O(1)
```

Total: O(n)

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
  - item = remove(index)
  - add(position, item)

n x O(1)

n x O(n)

n x O(n)

Total: O(n^2)

n times

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

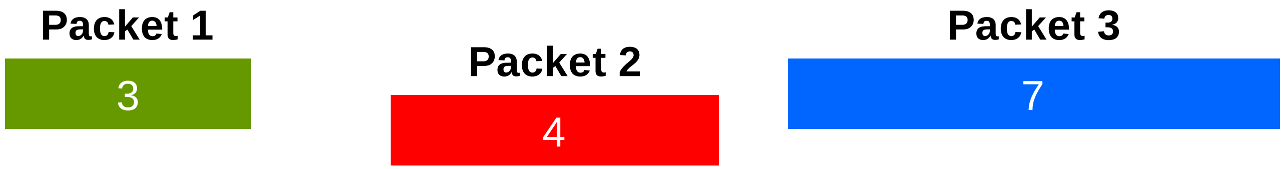
n x O(1)

n x O(1)

Total: O(n)

Combinations

- Given a set of n packets of weights w_1 , ..., w_n, and a shipping pallet/container/box that has size z
  - Example:



- Given the target z, what is the largest total weight <= z that can be achieved?
  - Example:
  - z <= 10 ?

Total Weight

3 + 7 = 10

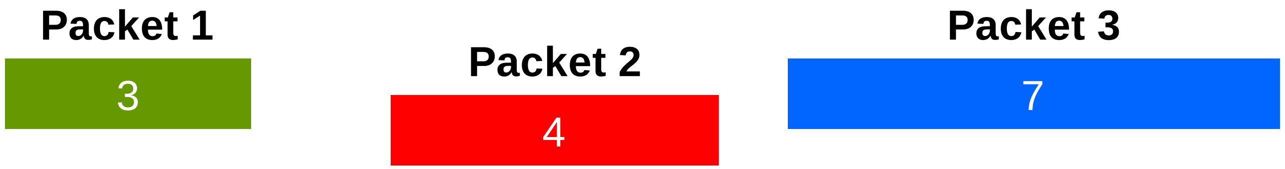
  - z <= 6 ?

Total Weight

4

Combinations – Largest total weight

- Given a set of n packets of weights w_1 , ..., w_n
  - Example:



Packet 1
3
Packet 2
4
Packet 3
7

- What is the largest total weight of any combination?
  - Example:
  - The best combination:

3
4
7

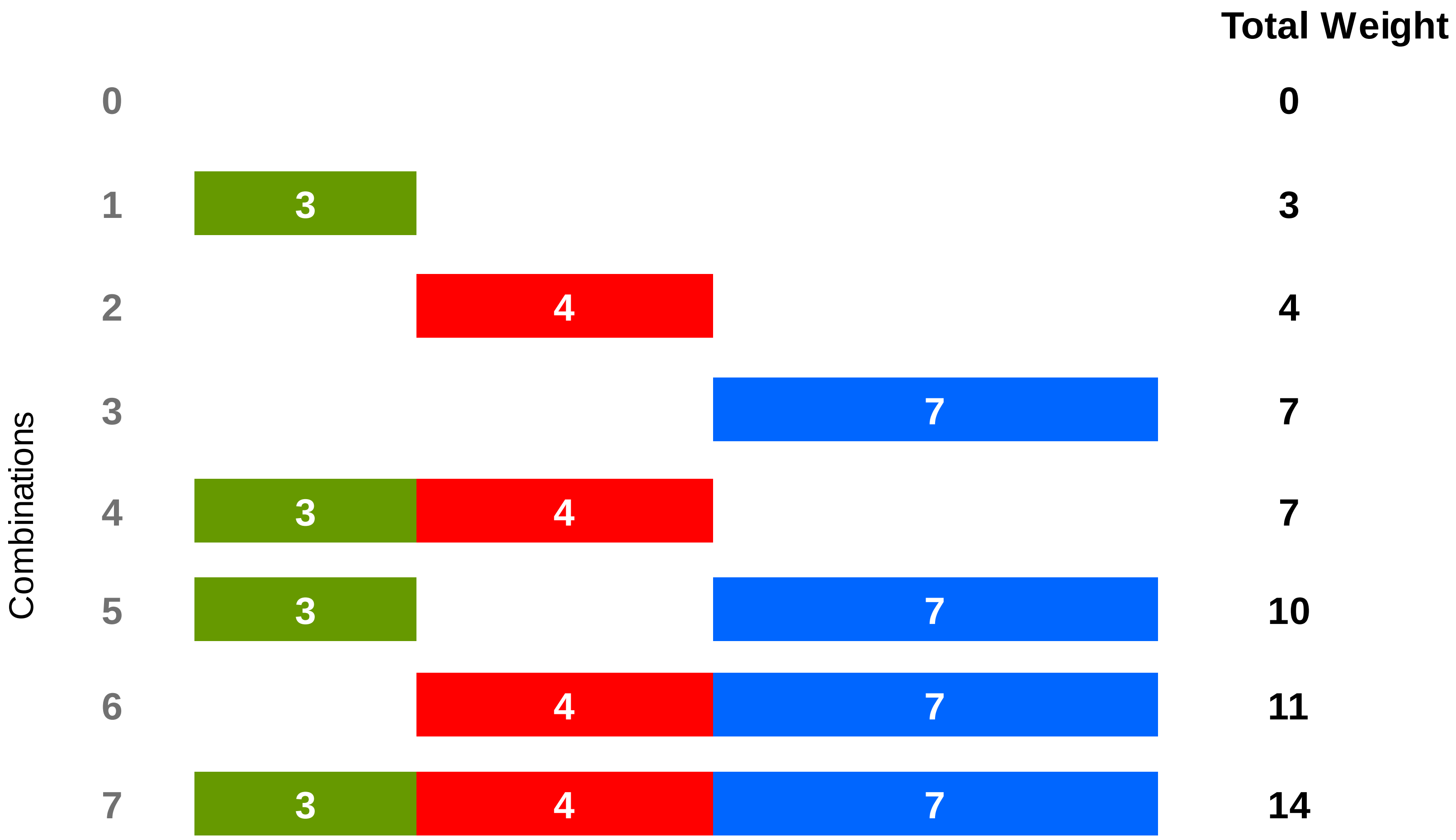
Total Weight

3 + 4 + 7 = 14

  - If all weights are positive, then selecting all packets gives the largest total weight

Combinations – List all

- Can we list all combinations with their respective total weight?



Combinations				Total Weight
0				0
1	3			3
2		4		4
3			7	7
4	3	4		7
5	3		7	10
6		4	7	11
7	3	4	7	14

- How many combinations are of n packets are there?
  - 2^n

Combinations – Selecting Packets

- How can we ensure that we did not forget any combination?
  - We just decide for each packet whether it should be selected for the combination or not
  - Yes = “packet selected”, No = “packet not selected”

Combinations	3	4	7	Total Weight
0	No	No	No	0
1	Yes	No	No	3
2	No	Yes	No	4
3	No	No	Yes	7
4	Yes	Yes	No	7
5	Yes	No	Yes	10
6	No	Yes	Yes	11
7	Yes	Yes	Yes	14

How to represent combinations?

- Anything that can be improved?
  - For an algorithm we better use 1 and 0 rather than Yes and No

		3	4	7	Total Weight
Combinations	0	0	0	0	0
	1	1	0	0	3
	2	0	1	0	4
	3	0	0	1	7
	4	1	1	0	7
	5	1	0	1	10
	6	0	1	1	11
	7	1	1	1	14

- We use a binary representation for combinations:
  - Example: 011 stand for packets 2 and 3

How to represent combinations?

- Does this idea also work for more than 3 packets?
  - Yes, here an example for n = 14:
  - 10001110011010 stands for the packets 1, 5, 6, 7, 10, 11,13

- Step through all numbers from 0 to 111 to try all combinations

```
for combn from 0 to 111
    work out total weight of combination
    if weight <= target  and  weight > best so far
        remember weight and combn
```

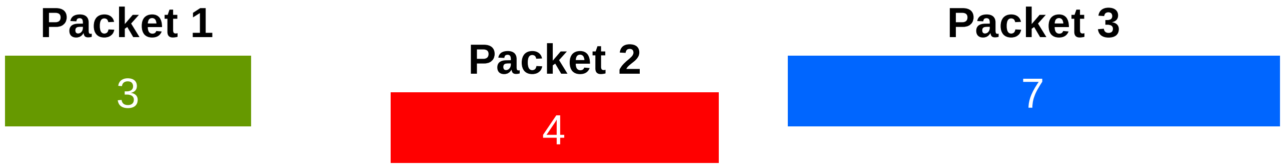
Cost of Algorithm with loop

- 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)

$\left.\begin{matrix} O(n) \\ O(1) \\ O(1) \end{matrix}\right\} 2^n \text{ times}$

Combinations – Can we do better?

- Given a set of n packets of weights w_1 , ..., $w_{n,}$ and a target z
  - Example:



Target z = 12

- Idea: Consider two options
- First option: if packet 1 has weight <= target z, then select it and we still have n-1 packets to choose from, but target must be reduced by the weight of packet 1
- Second option: do not select packet 1, then we still have n-1 packets to choose from, and target is still the same

Combinations – Can we use recursion?

- Idea: divide the problem (of size n) into two smaller subproblems (of size n-1)
  - So we can use recursion

- First option: if packet 1 has weight <= target z, then select it and we still have n-1 packets to choose from, but target must be reduced by the weight of packet 1

First subproblem of size n-1

- Second option: do not select packet 1, then we still have n-1 packets to choose from, and target is still the same

Second subproblem of size n-1

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

Combinations – Using Recursion

- 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…:

  // first option with first subproblem of size n-1

  - if including packet i would still be <= 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

  // second option with second subproblem of size n-1

  - find combinations using additional packets from index i+1… < RECURSIVE CALL

Cost of Algorithm with recursion

- 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)