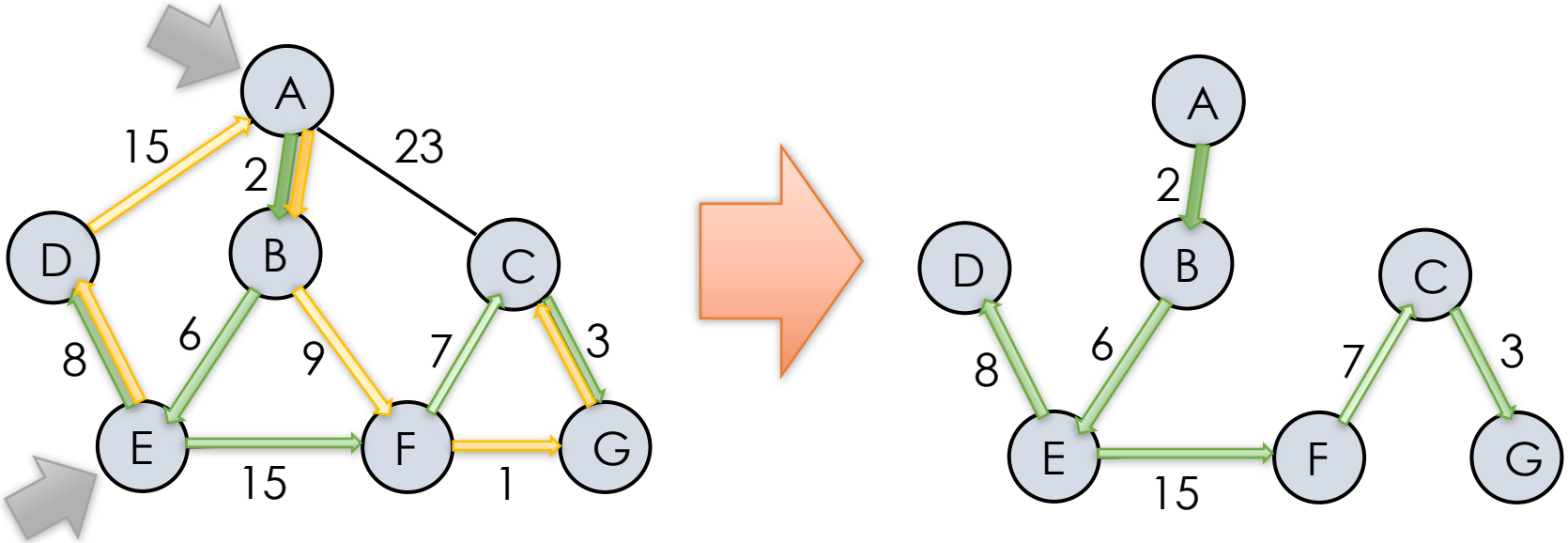
Minimum Spanning Trees, Topological Sort

Presented by Dr Anna Bosman

Spanning Trees

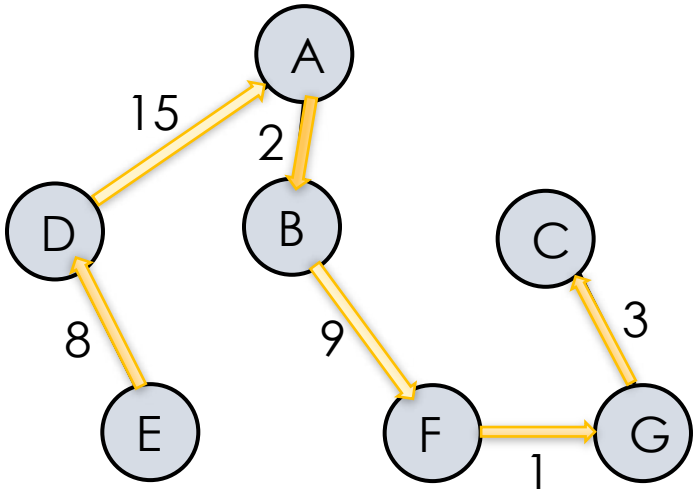
- What happens when DFS is performed on a graph?
  - Every vertex is visited exactly once
  - The edges used in DFS form a connected and acyclic structure
  - We call this structure a spanning tree



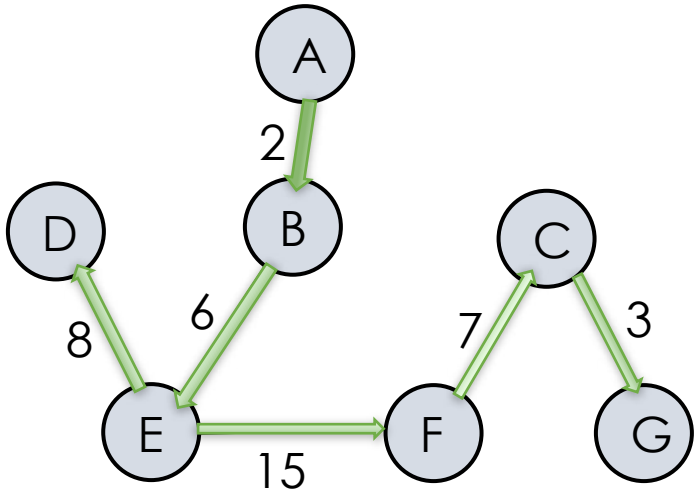
- The path will differ depending on the starting point and the order of vertices in the adjacency list
- Are all spanning trees equally useful?

Spanning Trees

- Are all spanning trees equally useful?
  - In a simple, unweighted graph – yes: $|E| = |V| - 1$
  - What about the weighted graph?



Total cost: 38

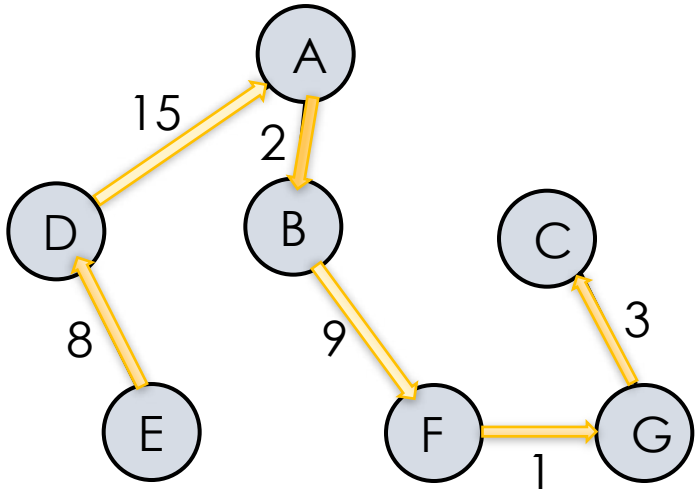


Total cost: 41

- Suppose you are installing fibre internet in the neighbourhood, and weights represents cable length
- You want to connect everyone in the neighbourhood and use as little cable as possible

Minimum Spanning Tree

- A minimum spanning tree is a spanning tree that minimizes the sum of edges
- Very useful in routing applications!
- How is it different from the shortest path problem?
  - Shortest path assumes a specific starting point
  - All-to-all shortest path algorithm finds multiple paths
- We are interested in a single cheapest way to connect all the dots (vertices)

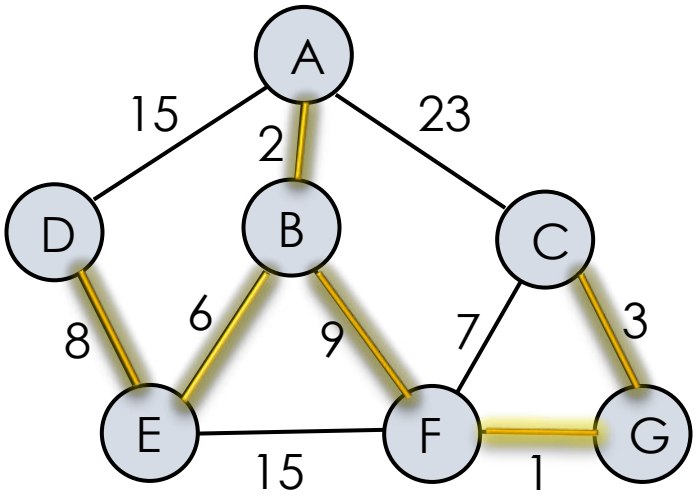


- Many algorithms have been proposed to minimize the spanning tree
- We will only look at two:
  - Kruskal algorithm
  - Dijkstra's algorithm (*not the same as his shortest path algorithm!*)

Kruskal algorithm: Minimum Spanning Tree

- Works on edges rather than vertices

1. Sort all the edges weight-wise in ascending order
2. While |spanning tree| < |V| - 1
   i. if edge(i) does not form a cycle, add it to the tree
   ii. i++



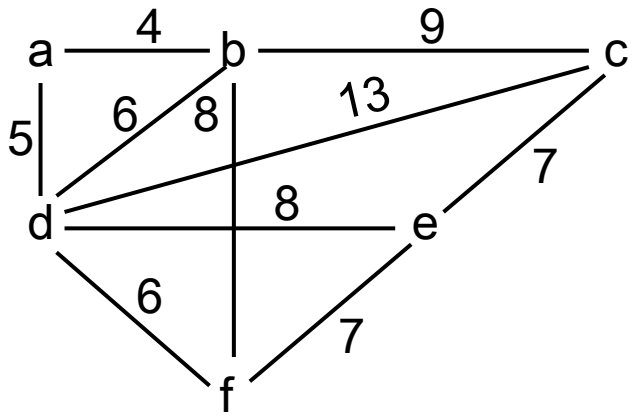
|V| = 7
|E| = 6

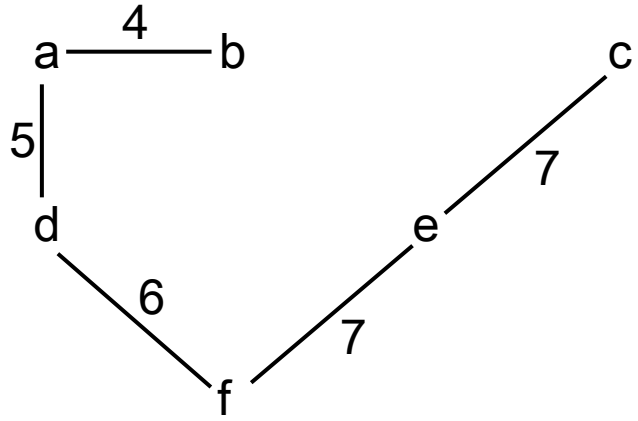
FG (1), AB (2), CG(3), EB (6), FC (7), DE (8), BF (9), AD (15), EF (15), AC (23)

Kruskal algorithm: Minimum Spanning Tree

```
KruskalAlgorithm(weighted connected undirected graph)
   tree = null;
   edges = sequence of all edges of graph sorted by weight;
   for (i = 1; i ≤ |E| and |tree| < |V| − 1; i++)
      if e_i from edges does not form a circuit with edges in tree
         add e_i to tree;
```

sorted edges: ab ad bd df ce ef de bf bc cd



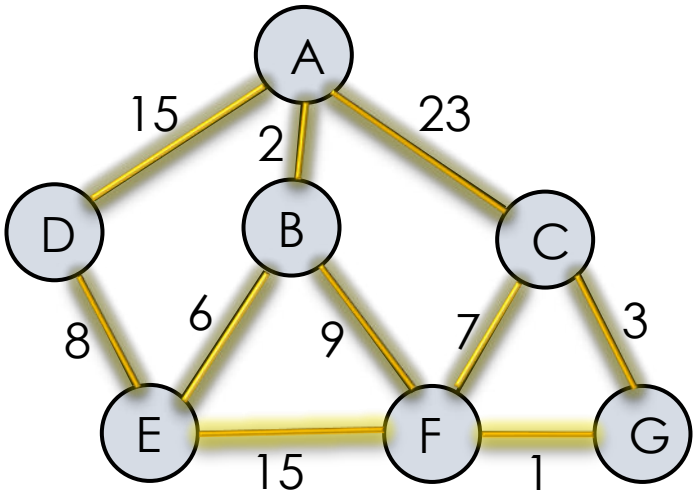


- How are we going to sort the edges?
  - Choice of sorting algorithm is important and will have an impact on performance
- How are we going to detect cycles?
  - We iterate through edges rather than vertices: use **union find**

Dijkstra algorithm: Minimum Spanning Tree

- "Lazy" version of Kruskal: does not sort the edges

1. For all |E|
   i. Add edge(i) to the spanning tree
   ii. If edge(i) formed a cycle, remove the edge with the largest weight from the cycle
   iii. i++



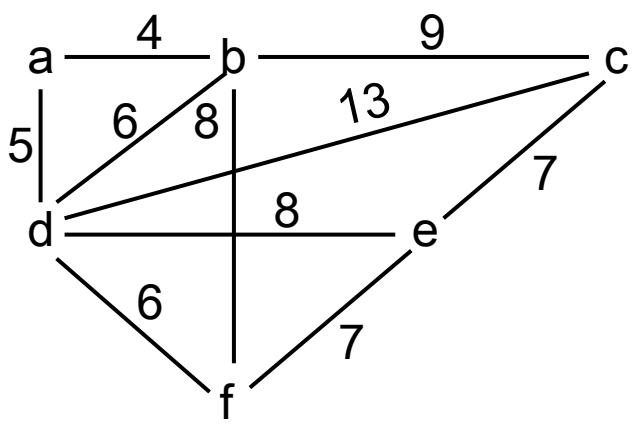
We arrived at the same spanning tree

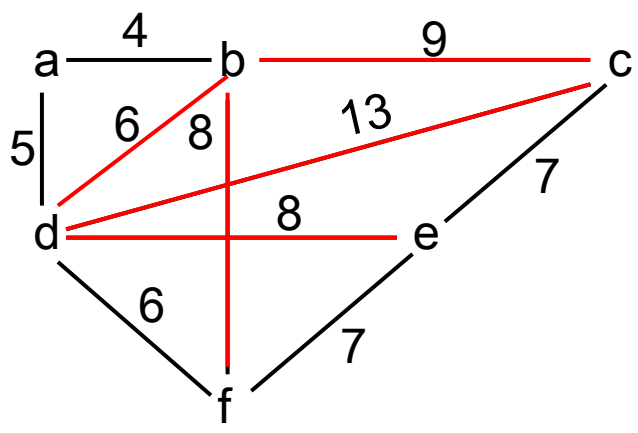
AB (2), AC (23), AD (15), BE (6), BF (9), CF (7), CG(3), DE (8), EF (15), FG (1)

Dijkstra algorithm: Minimum Spanning Tree

```
DijkstraMethod(weighted connected undirected graph)
   tree = null;
   edges = an unsorted sequence of all edges of graph;
   for i = 1 to |E|
      add e_i to tree;
      if there is a cycle in tree
         remove an edge with maximum weight from this only cycle ;
```

edges: ab ad bc bd bf cd ce de df ef

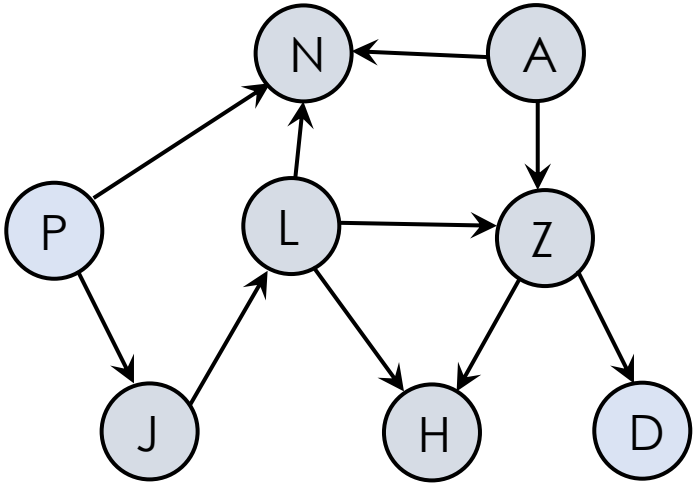




- Union find can be used for cycle detection

Directed Graphs: Topological Sort

- Suppose this graph describes dependencies between software packages:



To install H, L and Z have to be installed

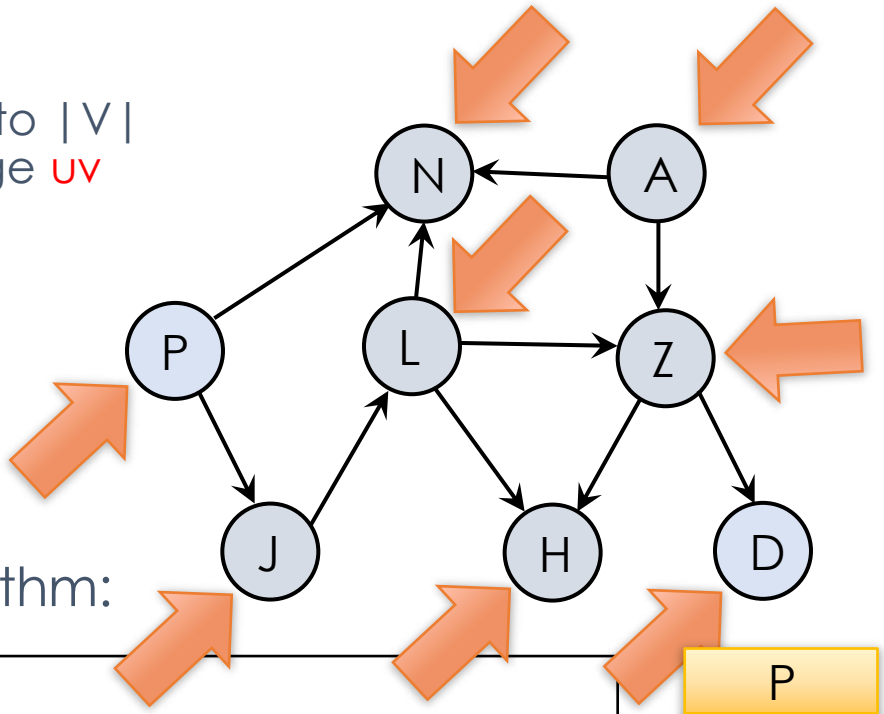
To install N, L, P and A have to be installed

Etc.

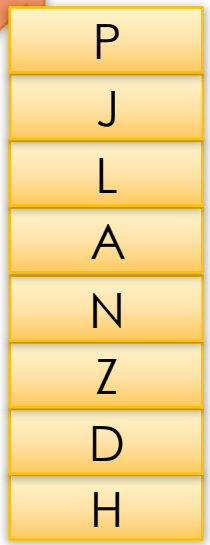
- You want to program an auto-install that will fetch the necessary packages and install them in correct order
- But what is the correct order?
  - P has no dependencies – we can start with P
  - Perform a DFS starting with P?
  - Problem: N also depends on A!

Directed Graphs: Topological Sort

- Topological sort:
  - Order (label) the vertices from 1 to |V| such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering
- Only works on acyclic directed graphs
  - There should be no circular dependencies
- Simplest topological sort algorithm:




```
topSort(digraph)
  for i = 1 to |V|
      find a vertex with no outgoing edges v;  // find a "sink"
      push v onto a stack;  // push onto a stack
      remove v and all incident edges from digraph;
```



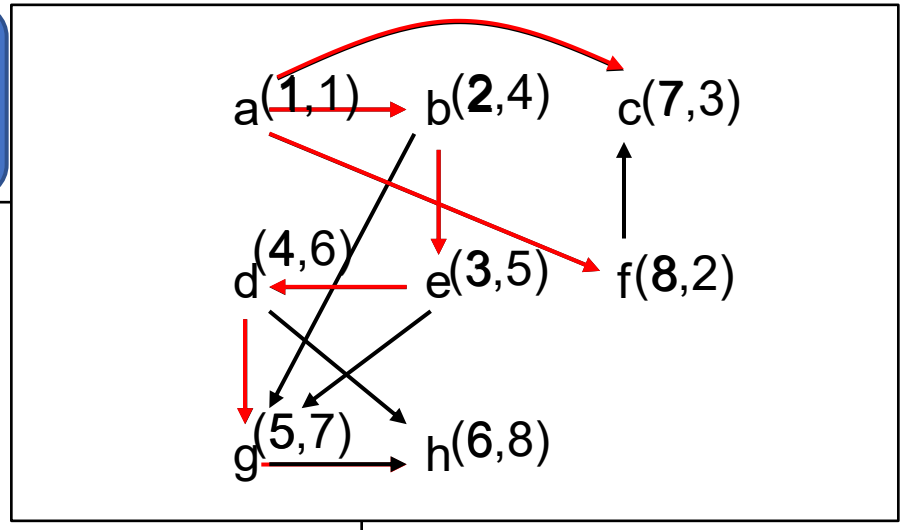
- Do we really need to destroy the graph in the process?
- No – we can use DFS + labelling instead

Topological Sort

Use two labels:
number (order of visiting) and
TS number (topological order)

```
topologicalSorting(digraph)
   for all vertices v
      num(v) = TSNum(v) = 0;
   i = 1; j = |V|;
   while there is a v such that num(v) is 0
      TS(v);
   output vertices according to their TSNum's;

TS(v)
   num(v) = i++;
   for all vertices u adjacent to v
      if num(u) is 0  // Visit unprocessed neighbours
         TS(u);       // recursively
      else if TSNum(u) is 0  // A visited vertex
         error;  // A cycle detected – halt algorithm.
   TSNum(v) = j--; // After processing all successors of v,
                   // assign to v a number smaller than
                   // assigned to any of its successors;
```



output:

a, f, c, b, e, d, g, h
1 2 3 4 5 6 7 8

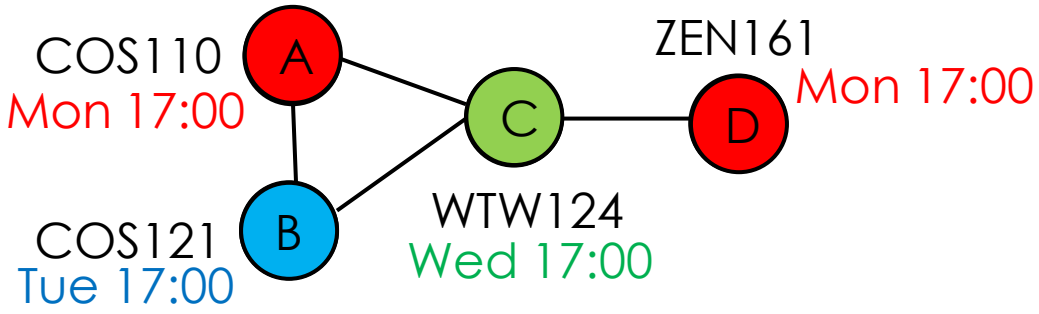
Lowest to highest TSnum indicates the dependency order

Graph Colouring

Presented by Dr Anna Bosman

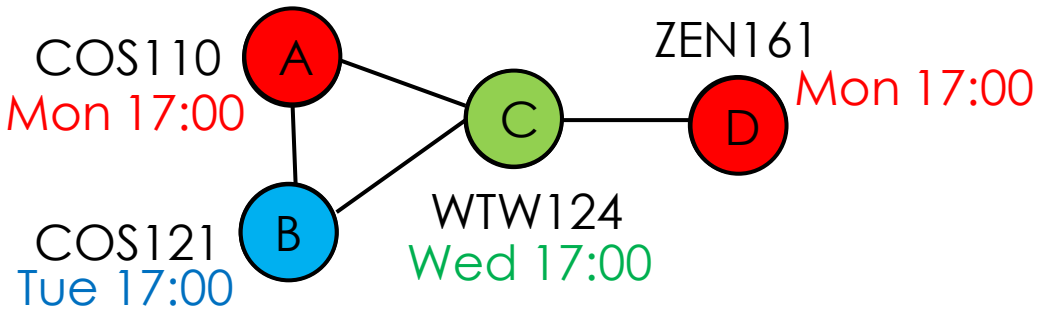
Graph colouring

- Suppose the following graph represents different modules that students take:



- Every edge(uv) means that there are students enrolled for both u and v
- The test week is coming up, and tests for all subjects need to be scheduled
- How many time slots should be reserved to avoid clashes?
  - Let the first time slot be represented by red: assign A to red
  - B and C are taken by the same students – can't use the same time
  - Assign B to another time slot – blue
  - C is taken by the same students – use another time slot, green
  - What about D? Some C students take it, but none of A or B do
  - Can not use green, but either red or blue time slot can be re-used

Graph colouring



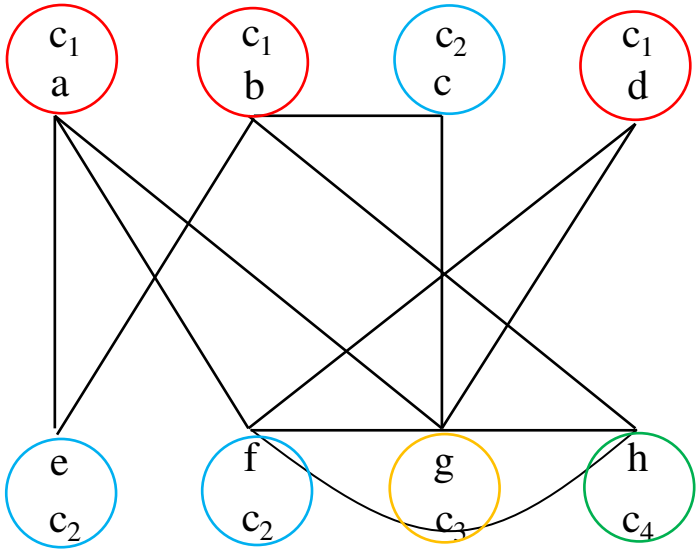
- Vertex colouring of a graph is an assignment of colours to the vertices of the graph so that no two adjacent vertices have the same colour
- In other words, the task is to find all sets of independent vertices, and assign a "colour", or label, to each independent set
- The minimum number of colours necessary to color the graph G is referred to as the chromatic number of G
- Chromatic number is not easy to determine – this problem is NP-complete *(that's why you always end up with a few clashes here and there)*
- We can use algorithms that approximate the chromatic number reasonably well

Graph colouring

```
sequentialColoringAlgorithm(graph = (V, E))
    put vertices in a certain order v1, v2, ..., v|V|;
    put colors in a certain order c1, c2, ..., ck;
        for i = 1 to |V|
        j = the smallest index of color that does not appear in any neighbor of vi;
        color(vi) = cj;
```

Does the order of colours matter?

Does the order of vertices matter?



Is 4 the chromatic number of this graph?

Can we do better?

Graph colouring

deg(v): degree of a vertex, i.e. number of neighbours

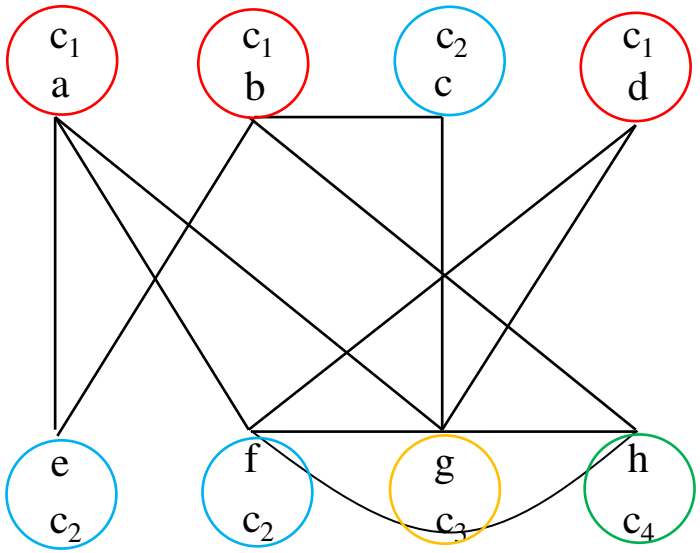
When colouring vertex v_i, at most $\min\{i-1, deg(v_i)\}$ of its neighbours already have colours.
Therefore, the color of v_i is at most $\min\{i, deg(v_i) + 1\}$

Theorem. For the sequential coloring algorithm, the number of colors needed to color the graph,

$$\chi(G) \le \max_i \min\{i, deg(v_i) + 1\}.$$

$\chi(G) \le \max \min(i, deg(\,) + 1)$
$= \max(\min(1, 4), \min(2, 4),$
$\min(3, 3), \min(4, 3),$
$\min(5, 3), \min(6, 5),$
$\min(7, 6), \min(8, 4))$
$= \max(1, 2, 3, 3, 3, 5, 6, 4)$
$= 6$

We want $\chi(G)$ to be as small as possible – how can we arrange the vertices to minimize $\chi(G)$?



Graph colouring

Theorem. For the sequential coloring algorithm, the number of colors needed to color the graph,

$$\chi(G) \leq \max_i \min\{i, deg(v_i) + 1\}.$$

What if we put vertices with largest $deg(v_i)$ first, and smallest $deg(v_i)$ last?

- In the beginning, i will be chosen over $deg(v_i)$ in $\min\{i, deg(v_i) + 1\}$
- In the end, $deg(v_i)$ will be chosen over i in $\min\{i, deg(v_i) + 1\}$

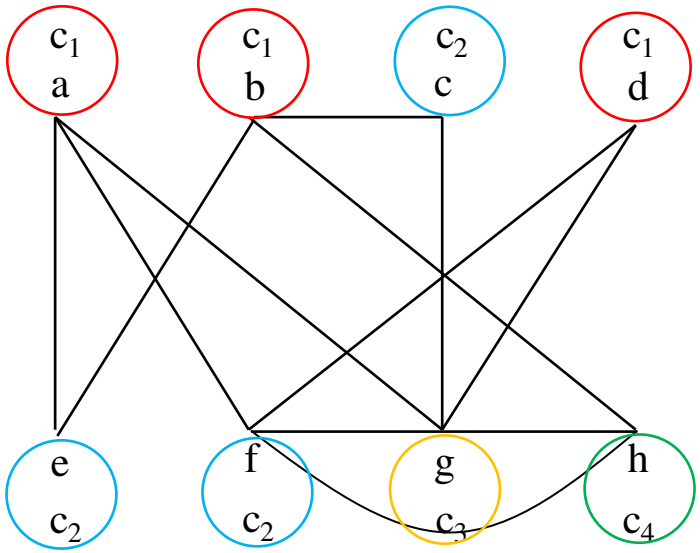
$\chi(G) \leq \max \min(i, deg(\,) + 1)$

$= \max(\min(1, 6), \min(2, 5),$
$\min(3, 4), \min(4, 4),$
$\min(5, 4), \min(6, 3),$
$\min(7, 3), \min(8, 3))$

$= \max(1, 2, 3, 4, 4, 3, 3, 3)$

$= 4$

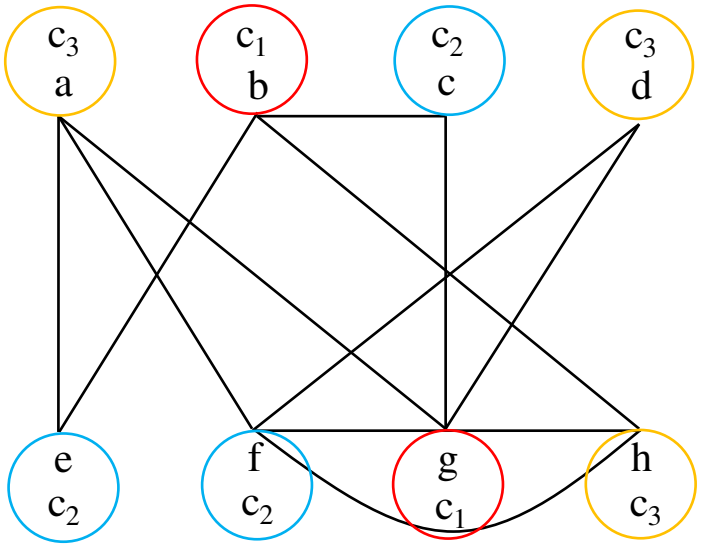
We can now re-write sequential colouring algorithm, ordering vertices according to their degrees



Graph colouring: Largest first

```
sequentialColoringAlgorithm(graph = (V, E))
    sort vertices in descending order by deg(v): v1, v2, ..., v|V|;
    put colors in a certain order c1, c2, ..., ck;
        for i = 1 to |V|
        j = the smallest index of color that does not appear in any neighbor of vi;
        color(vi) = cj;
```

Improvement: we used 3 colours instead of 4



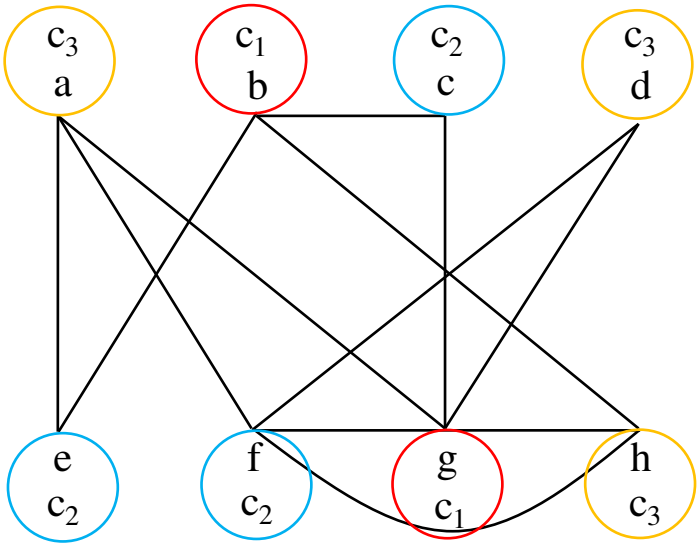
g	f	a	b	h	c	d	e
5	4	3	3	3	2	2	2

Can we do better?

What if we use other criteria besides vertex degree to sort the vertices?

Graph colouring: Brelaz algorithm

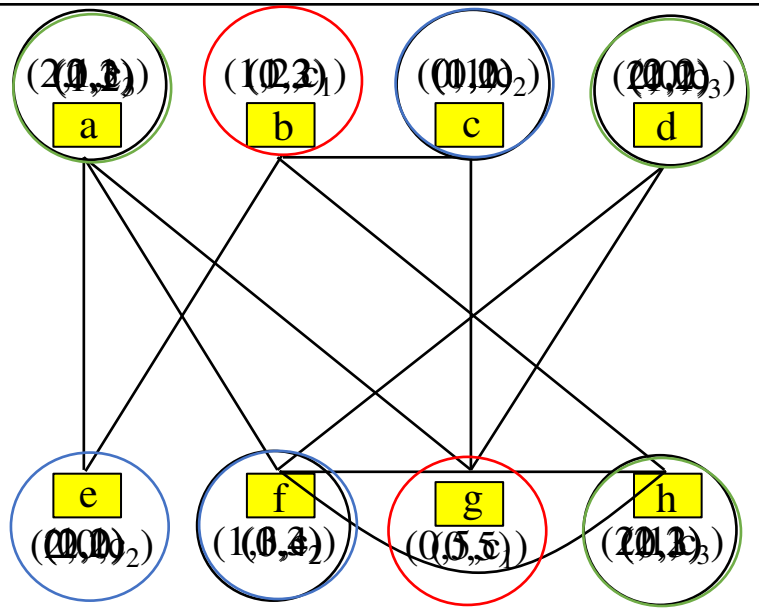
- Brelaz (1979) suggested using saturation degree and uncoloured degree to order vertices
  - Saturation degree of v: Number of different colours used to colour v's neighbours
  - Uncoloured degree of v: Number of uncoloured neighbours of v
  - Saturation degree is initialized to zero for all vertices
  - Uncoloured degree is initialized to deg(v)
  - Both are updated at every interation
- Especially useful when many vertices have the same degree



Graph colouring: Brelaz algorithm

```
BrelazColoringAlgorithm(graph)
   for  each vertex v
      saturationDeg(v) = 0;          // the number of different colors used to color neighbors of v;
      uncoloredDeg(v) = deg(v);      // the number of uncolored vertices adjacent to v;
   put colors in a certain order c1, c2, ..., ck;
   while  not all vertices are processed
      v = a vertex with maximum uncolored degree out of the vertices with highest saturation degree;
      j = the smallest index of color that does not appear in any neighbor of v;
      for  each uncolored vertex u adjacent to v
         if no vertex adjacent to u is assigned color cj
            saturationDeg(u)++;
         uncoloredDeg(u)--;
      color(v) = cj;
```

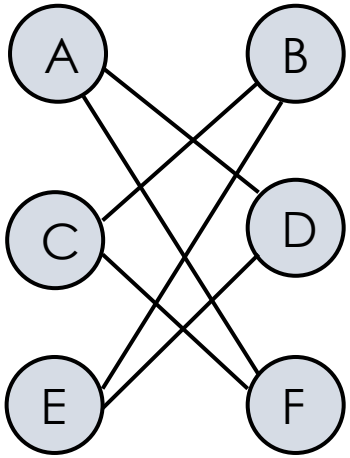
Same colouring was discovered with "largest first" ordering



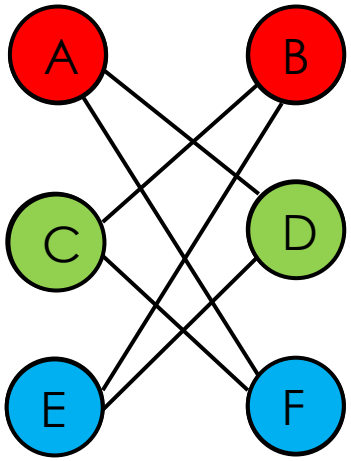
What if all vertices had the same degree?

Graph colouring

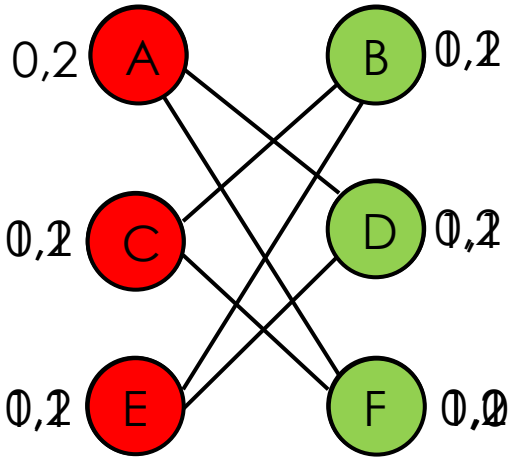
Consider this graph:



"Largest degree first"

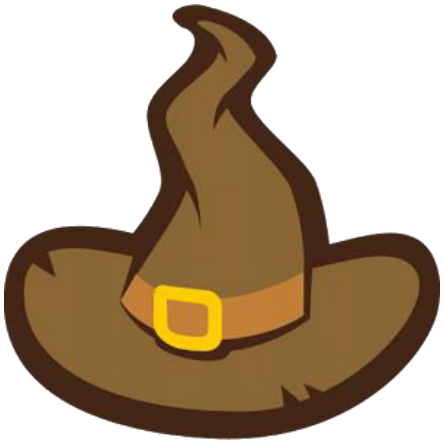


Brelaz method



COS 212
Chapter 9 (Sorting): Elementary Algorithms

Sorting is important



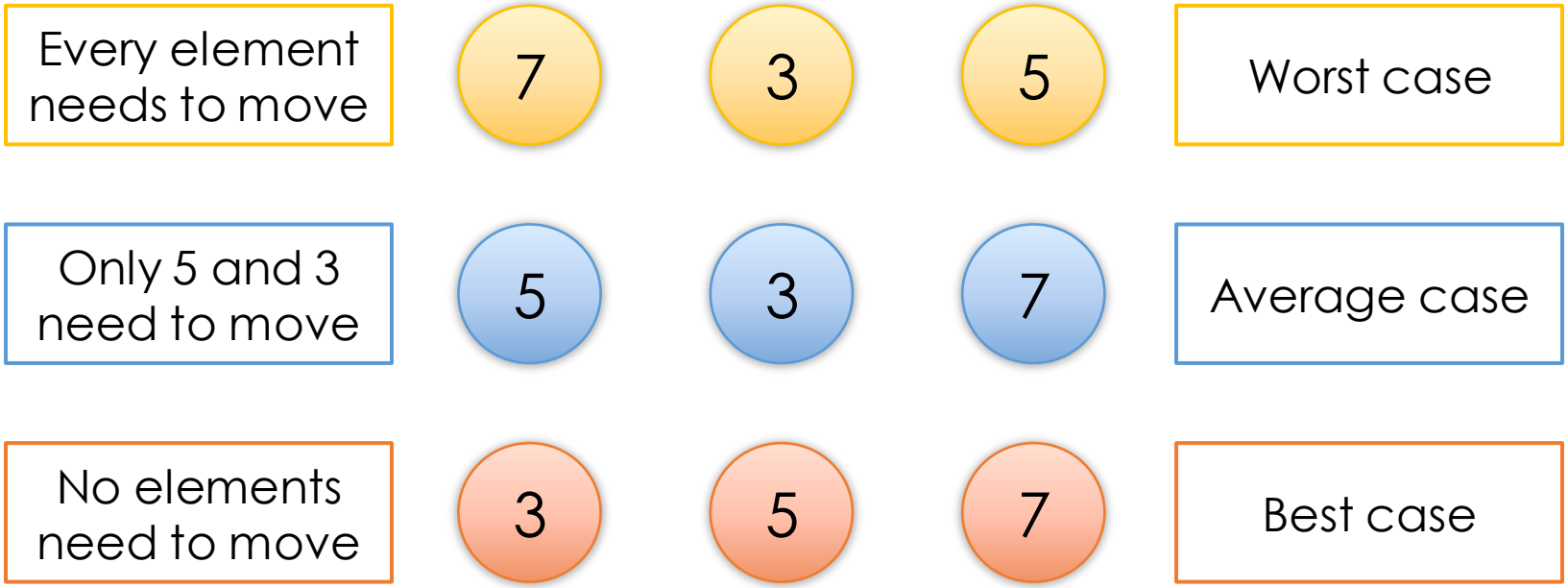
- Why do we sort data?
  - Because it makes searching so much easier!
  - Suppose we're looking for a student number…
  - 15367822 15990400 14367789 15562890 15782200 15378830
  - 14367789 15367822 15378830 15562890 15782200 15990400
  - For a human, sorted data is easier to comprehend and process
  - A computer, having no consciousness, does not care… But sorted data means we can find an item much more efficiently: consider binary search
  - Some algorithms rely on data being sorted
  - Some data structures rely on sorted data
  - Sorting is important, and we need good algorithms for it!

What makes a good sorting algorithm?

- Fast execution!
- I.e., if time is defined as $t = f(n)$, we want time to increase as slowly as possible as n, number of items, increases
- Big-O complexity can be used to estimate time efficiency of sorting
- What actions constitute sorting?
  - Comparisons
  - Movements

What makes a good sorting algorithm?

- The number of actions will depend on the order of elements
- Consider:

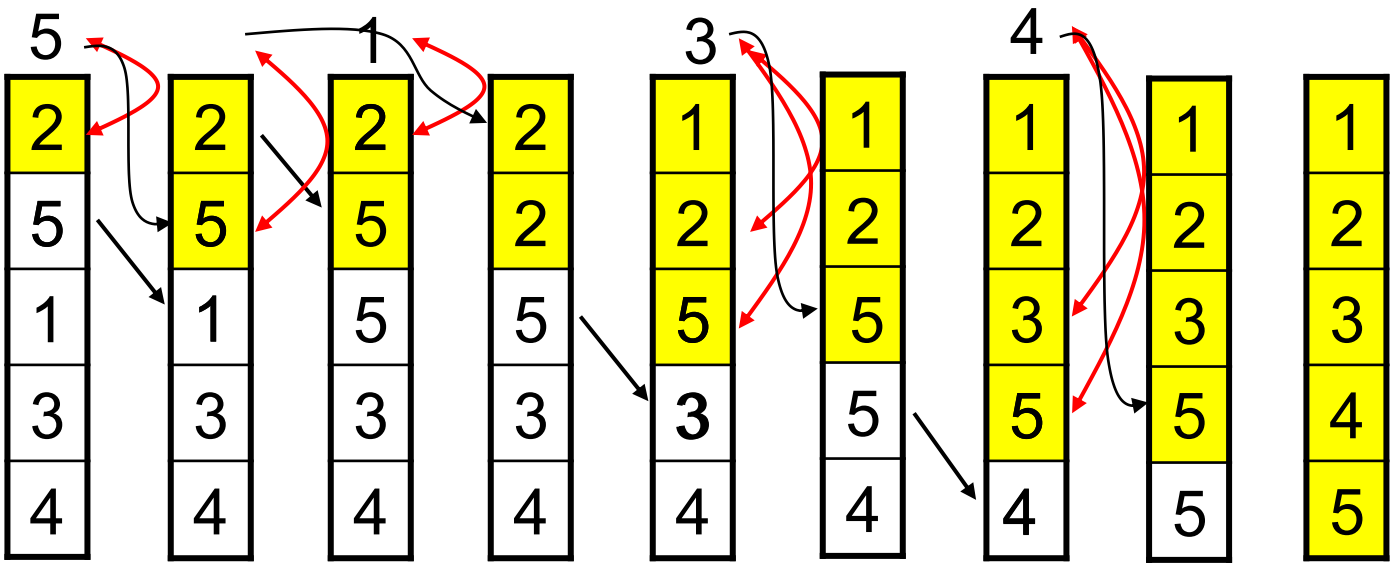


- We want an algorithm that performs the best in **all** cases

Insertion Sort

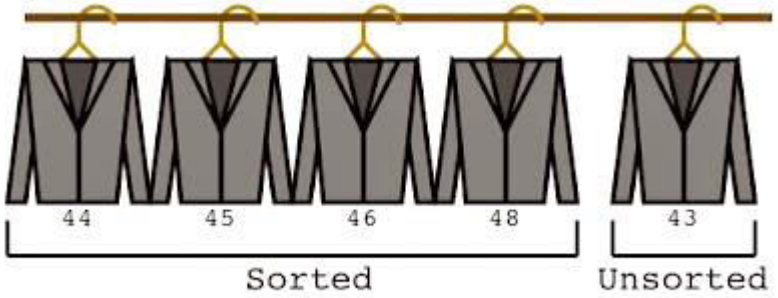
Main idea: take an element from the unsorted part of the array, insert it into the sorted part

```
insertionSort(data[ ])
  for i = 1 to data.length-1 // initialize i to 1 (2nd element)
    tmp = data[i]; // store i'th element in tmp
    j = i – 1;     // set j to the element preceding i
    while j >= 0 && data[j] > tmp // find correct position
       data[j + 1] = data[j]; // move j'th up by 1 position
       j--; // go to an earlier element, 1 pos. back
    data[j + 1] = tmp; // end of loop: pos. found!
```



Insertion Sort: Efficiency

n = data.length




```
insertionSort(data[ ])
  for i = 1 to n
    tmp = data[i];
    j = i – 1;
    while j >= 0 && data[j] > tmp
       data[j + 1] = data[j];
       j--;
    data[j + 1] = tmp;
```

How many times will the outer loop execute?

$(n - 1)$

How many times will the inner loop execute?

Depends on the case!

Best case (ordered)	Worst case (reversed)	Average case
0	1, 2, … $(n - 2)$, $(n - 1)$	½ of the worst case?

Insertion Sort: Efficiency

```
insertionSort(data[ ])
  for i = 1 to n-1
    tmp = data[i];
    j = i – 1;
    while j >= 0 && data[j] > tmp
       data[j + 1] = data[j]; j--;
    data[j + 1] = tmp;
```

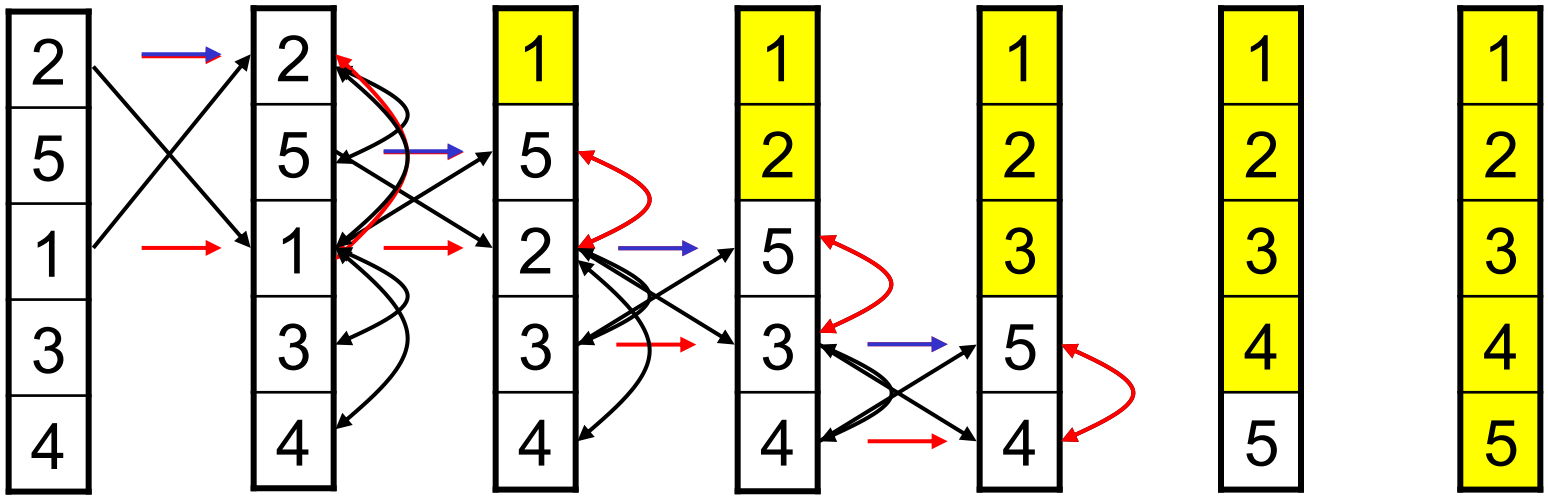
	best case	avg case	worst case
comparisons	O(n)	$O(n^2)$	$O(n^2)$
movements	O(n)	$O(n^2)$	$O(n^2)$

Is average case better than the worst case?

Selection Sort

Main idea: find the smallest element, put it in the correct position

```
selectionSort(data[ ])
        for i = 0 to data.length-1
          select the smallest element among
             data[i], …, data[data.length-1];
          swap it with data[i];
```



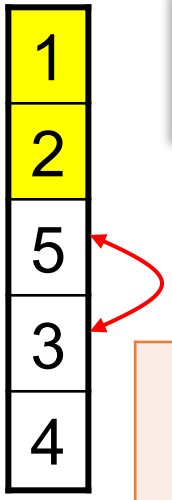
Selection Sort: Efficiency

```
selectionSort(data[ ])
        for i = 0 to n-1
          select the smallest element among
             data[i], …, data[n-1];
          swap it with data[i];
```

How many times will the outer loop execute? $(n - 1)$

How many times will the inner loop execute?

$(n - 1)$, $(n - 2)$, … 2, 1



Do any movements happen in the inner loop?

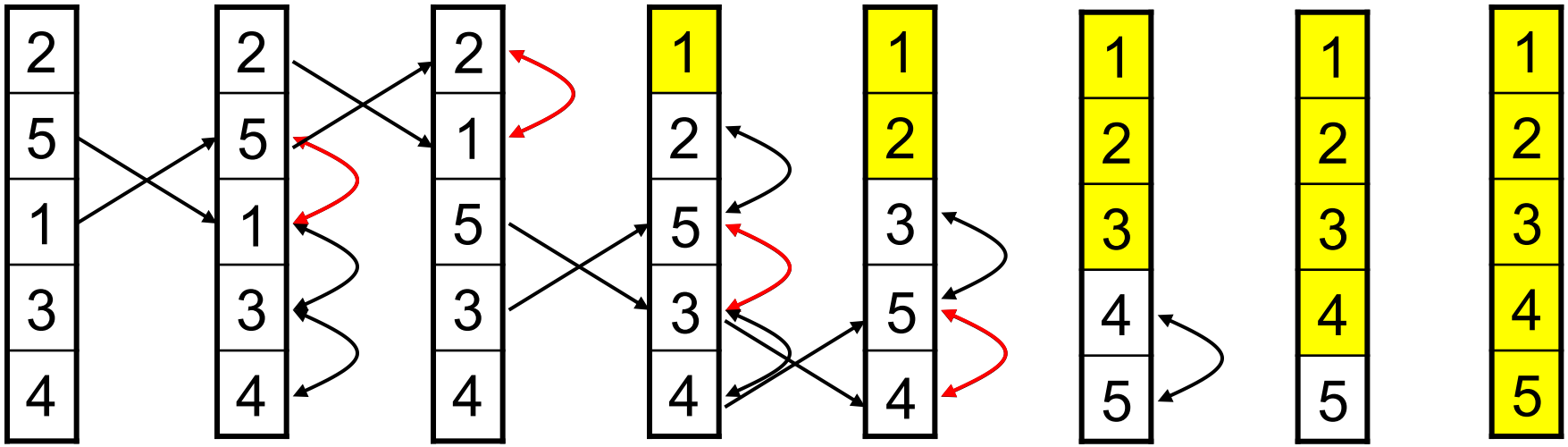
What about the outer loop?

	best case (ordered)	avg case (random)	worst case (largest first, the rest is ordered)
comparisons	$O(n^2)$	$O(n^2)$	$O(n^2)$
movements	0	$O(n)$	$O(n)$

Bubble Sort

Main idea: swap neighbour elements if they are out of order

```
bubbleSort(data[ ])
    for i = 0 to data.length-2
      for j = data.length-1 down to i+1
        if elements in positions j and j-1 are out of order,
          swap them;
```



Bubble Sort

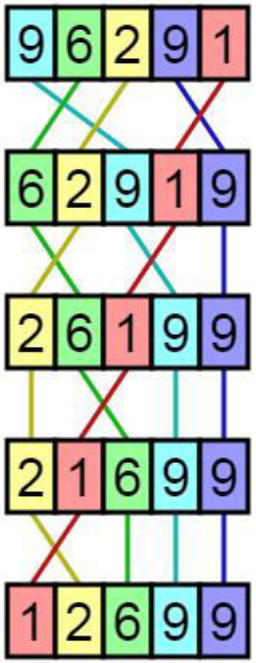
```
bubbleSort(data[ ])
    for i = 0 to n-2
      for j = n-1 down to i+1
        if elements in positions j and j-1
          are out of order, swap them;
```

$(n - 1)$

How many times will the outer loop execute?

How many times will the inner loop execute?

$(n - 1)$, $(n - 2)$, … 2, 1



Does the number of movements differ from the number of comparisons?

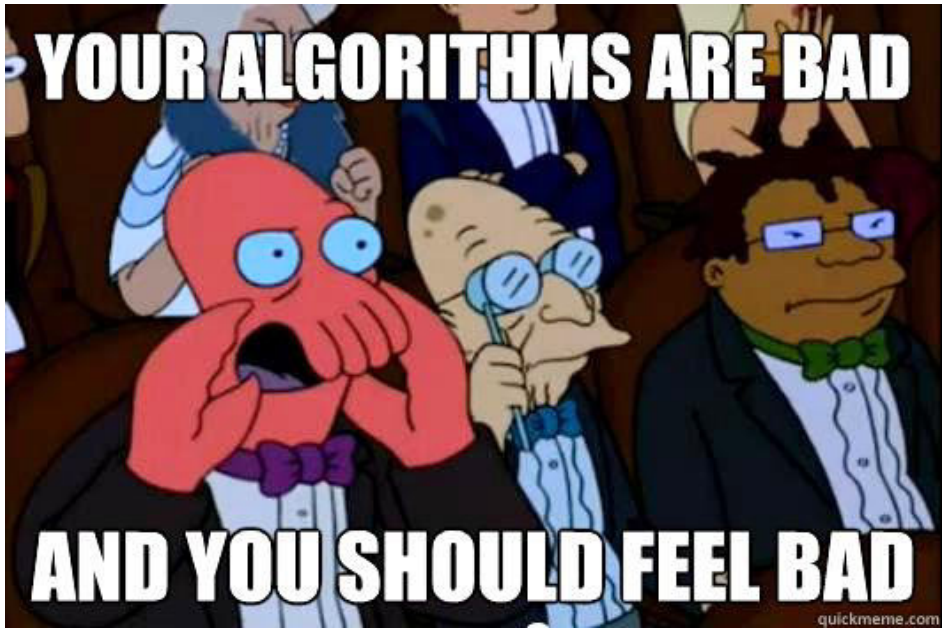
	best case (ordered)	avg case (random)	worst case (reversed)
comparisons	$O(n^2)$	$O(n^2)$	$O(n^2)$
movements	0	$O(n^2)$	$O(n^2)$

Who's the fastest of them all?

insertion	best	avg	worst
comp.	O(n)	$O(n^2)$	$O(n^2)$
move	O(n)	$O(n^2)$	$O(n^2)$

selection	best	avg	worst
comp.	$O(n^2)$	$O(n^2)$	$O(n^2)$
move	0	O(n)	O(n)

bubble	best	avg	worst
comp.	$O(n^2)$	$O(n^2)$	$O(n^2)$
move	0	$O(n^2)$	$O(n^2)$

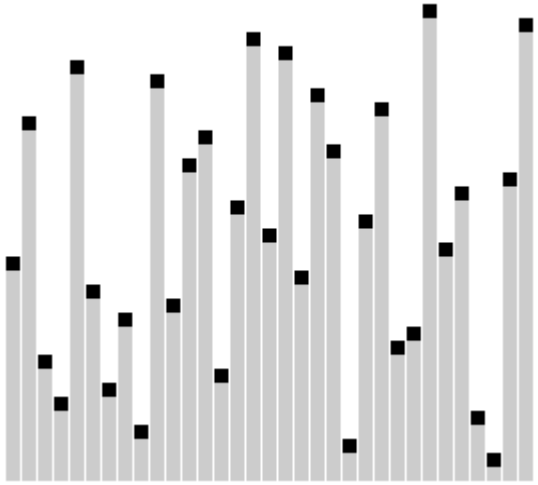


Why is bubble sort so slow?

- Because every element that is not in its right place is painstakingly swapped with other elements one-by-one

6 5 3 1 8 7 2 4

- Big elements move to their right places quite fast
- But small elements take forever to get to the start…
- Can we speed up the "turtles"?
- Idea: what if, instead of comparing adjacent elements, we compare every third element?
- **Comb sort**: start with sparse comparisons, decrease the gap over time



Comb Sort

```
combSort(data[ ])
    gap = n // this will be our shrinking gap
    swapped = false // stop when no swaps ha
    while gap > 1 or swapped
        swapped = false
        if gap > 1 then
            gap = floor(gap/1.3) // shrink!
        i = 0
        while i + gap < n
            if data[i] > data[i + gap]
                swap data[i] with data[i + gap]
            i += 1
```

Shrinking the gap step

Bubble sort step

How many times will the outer loop execute?

log n?

How many times will the inner loop execute?

$(n - n/1.3), (n - n/1.3^2), \ldots, n$

Comb Sort

33 98 74 13 55 20 77 45 64 83

gap = 10
gap/1.3 = 7

33 98 74 13 55 20 77 45 64 83
33 98 74 13 55 20 77 45 64 83
33 64 74 13 55 20 77 45 98 83
33 64 74 13 55 20 77 45 98 83

gap/1.3 = 5

33 64 74 13 55 20 77 45 98 83
20 64 74 13 55 33 77 45 98 83
20 64 74 13 55 33 77 45 98 83
20 64 74 13 55 33 77 45 98 83
20 64 45 13 55 33 77 74 98 83
20 64 45 13 55 33 77 74 98 83
20 64 45 13 55 33 77 74 98 83

Comb Sort

```
20 64 45 13 55 33 77 74 98 83
13 64 45 20 55 33 77 74 98 83
13 64 45 20 55 33 77 74 98 83
13 55 45 20 64 33 77 74 98 83
13 55 45 20 64 33 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
```

gap/1.3 = 3

gap/1.3 = 2

```
13 55 33 20 64 45 77 74 98 83
13 55 33 20 64 45 77 74 98 83
13 20 33 55 64 45 77 74 98 83
13 20 33 55 64 45 77 74 98 83
13 20 33 55 64 45 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
```

Comb Sort

gap/1.3 = 1

Efficiency?

Best: O(n lg n)

Worst: O(n^2)

```
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 64 55 77 74 98 83
13 20 33 45 55 64 77 74 98 83
13 20 33 45 55 64 77 74 98 83
13 20 33 45 55 64 77 74 98 83
13 20 33 45 55 64 74 77 98 83
13 20 33 45 55 64 74 77 83 98
13 20 33 45 55 64 74 77 83 98
```

COS 212
Sorting:
Shell Sort and Heap Sort

Intuitive Sorting Algorithms are Slow

- Why are the intuitive sorting algorithms so slow?
  - Always involves a nested loop, bringing complexity up to $O(n^2)$
  - Can we alleviate this quadratic bane?
- Let's look at an example

6	7	2	1	5	8	3	4

  - This array contains 8 elements
  - If sorting has $O(n^2)$ complexity, then $t(n) = n^2$, and $t(8) = 64$
- What if we split the array in two halves?

6	7	2	1

5	8	3	4

  - Using the same algorithm to sort the halves separately
    - For each half of the array, $t(4) = 16$
    - Therefore, for the entire array, $t(4) \times 2 = 32$
    - Much less time complex than sorting the whole array!

Efficient sorting

- However… sorting each half does not sort the entire array!
  - But it does bring the array closer to a fully sorted array
- Arrays can be subdivided in many ways – not just halved
- Let's consider comb sort again
  - Compare elements that are *m* places apart
  - Value of *m* is called a gap
  - The gap shrinks non-linearly for each pass through the array
  - In each pass we bubble sort only a **subset** of elements

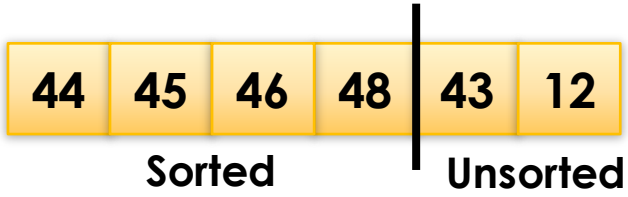
33	98	74	13	55	20	77	**45**	64	83
33	**98**	74	13	55	20	77	45	**64**	83
33	***64***	74	13	55	20	77	45	***98***	83
33	64	**74**	13	55	20	77	45	98	**83**

m = 7

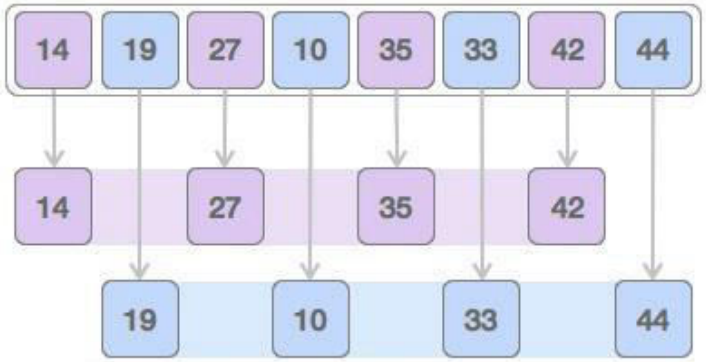
- This improves performance
  - Comb sort improves bubble sort to O(n lg n) in the best case
  - Can we improve insertion sort and selection sort in a similar way?

Shell Sort

- How do insertion sort and selection sort work?
  - Keep an "invisible wall" between the sorted and unsorted part

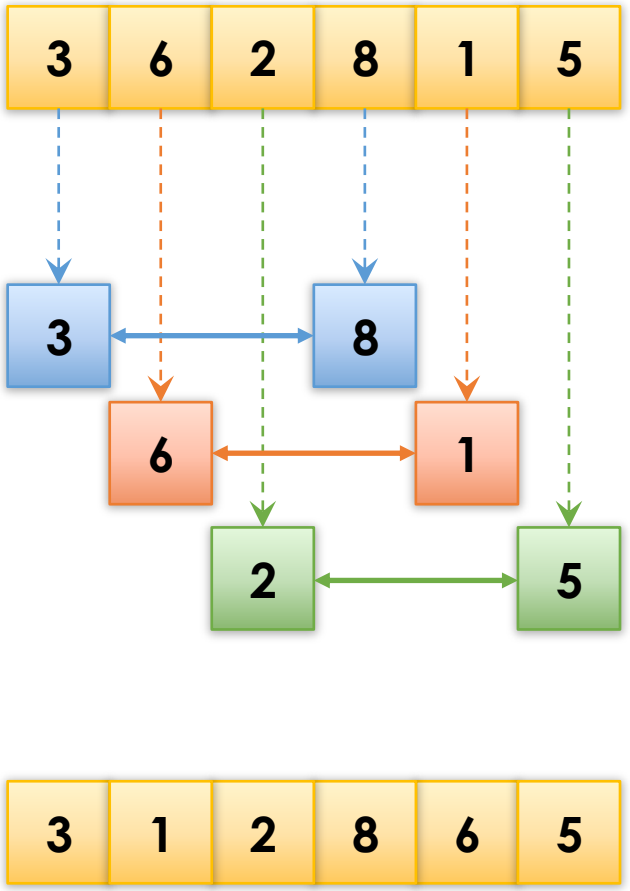


  - Insertion sort removes elements from unsorted, inserting them into the correct location in sorted
  - Selection sort swaps the smallest element in unsorted with the element at the head of unsorted, moving it into sorted
- The shell sort algorithm
  - Uses comb sort's idea: Use a variable gap to create sub-arrays
  - Sort sub-arrays using any algorithm, decrease gap, until sorted



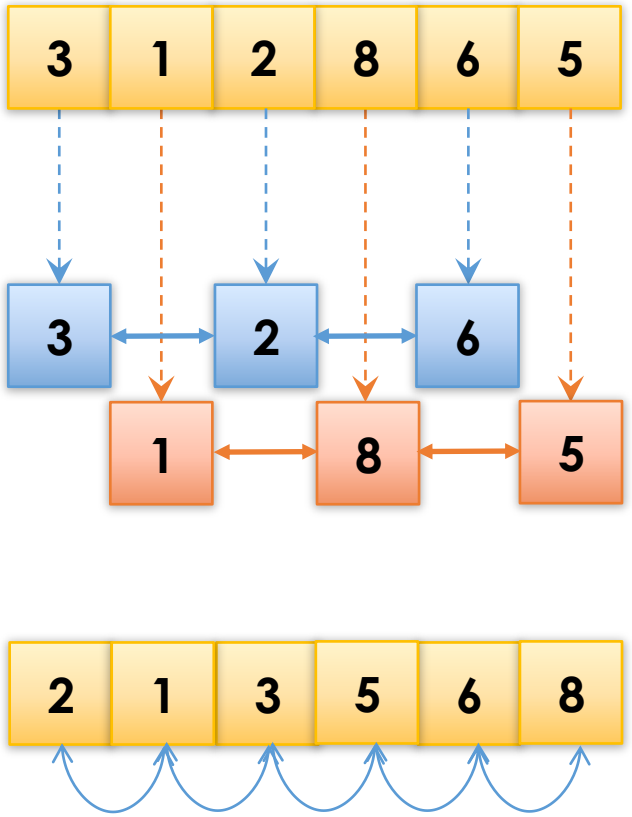
Shell Sort

- Let's look at an example
  - n = 6
  - Choose an initial gap = 3
  - Find sub-arrays
  - Apply a sorting algorithm to every sub-array
    - Use any sorting algorithm
    - Insertion sort is very commonly used, so we'll do the same for this example
  - Shrink the gap until gap = 1



Shell Sort

- Moving on to the second pass
  - We'll reduce the gap by 1 for this example
  - gap = 2
  - Find sub-arrays
  - Apply insertion sort to sub-arrays



- Now for the final pass
  - gap = 1
  - We're now working with the full array
  - Apply insertion sort

- We performed a single move on the last iteration!

Shell Sort: Efficiency

Iterate through all the gaps

```
for (gap = initialValue; gap >= 1; gap = shrinkGap(gap)){
  for (p = 0; p < gap; p++) {
    // insert your favourite algorithm here to
    // sort each sub-array starting at index p
  }
}
```

Generate all the sub-arrays

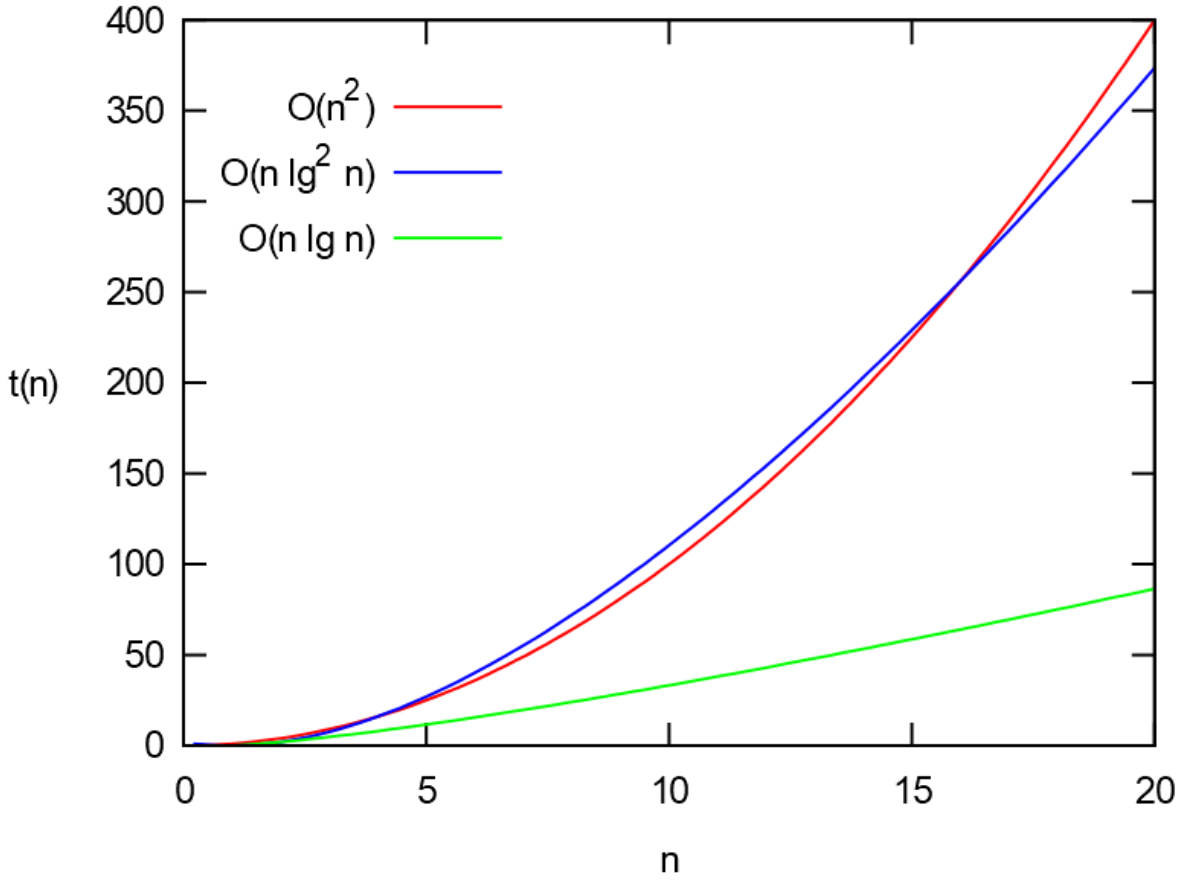
$O(n^2)$, but will improve significantly thanks to sorting of sub-arrays

- How do we choose the gaps?
- If the gaps decrease linearly
  - n-1, … , 3, 2, 1
  - How many times will the outer loop execute?
  - Even if the inner algorithm improves from $O(n^2)$ to $O(n)$, complexity of the outer loop will bring it back to $O(n^2)$
- Gaps must decrease non-linearly!
  - Divide by a factor
  - Hard-code a sequence of gaps

Best case: $O(n \lg n)$

Worst case: $O(n \lg^2 n)$

Shell Sort: Efficiency



Revisiting Selection Sort

```
selectionSort(data[ ])
        for i = 0 to n-2
          select the smallest element among
             data[i], ... , data[data.length-1];
          swap it with data[i];
```

Selection	Best	Average	Worst
Comparisons	$O(n^2)$	$O(n^2)$	$O(n^2)$
Movements	0	O(n)	O(n)

- Why is selection sort slow?
  - We have to search for the smallest element at every outer loop iteration
  - What if we had direct access to the smallest element?
  - Do you recall any data structure that allows exactly that?
- Yes! Heaps
  - Min-heap: The smallest element is always at the top
  - Max-heap: The largest element is always at the top
- Heaps are especially appropriate for sorting
  - They can be stored in arrays

Heap Sort

```
heapsort(data[])
  transform data[] into a max-heap;  // Floyd's algorithm
  for i = data.length-1 down to 2
    swap the root with the element in position i;
    restore heap property for the tree data[0], ... , data[i-1];
```

- Phase 1: Transform data into a max-heap
- Phase 2: The sorting procedure
  - The opposite of classic insertion sort (i.e. sorted part is on the right)
  - Start with entire array in unsorted part
  - Swap root (largest element in the heap) with last value in unsorted
  - This removes root from heap, adding it to the left of the sorted part
  - Thus the heap shrinks as the sorted part grows
  - Finally, restore the heap in the unsorted part, and repeat
- The result of this heap sort
  - Array in ascending order
  - What if you wanted descending order instead?

Heap Sort

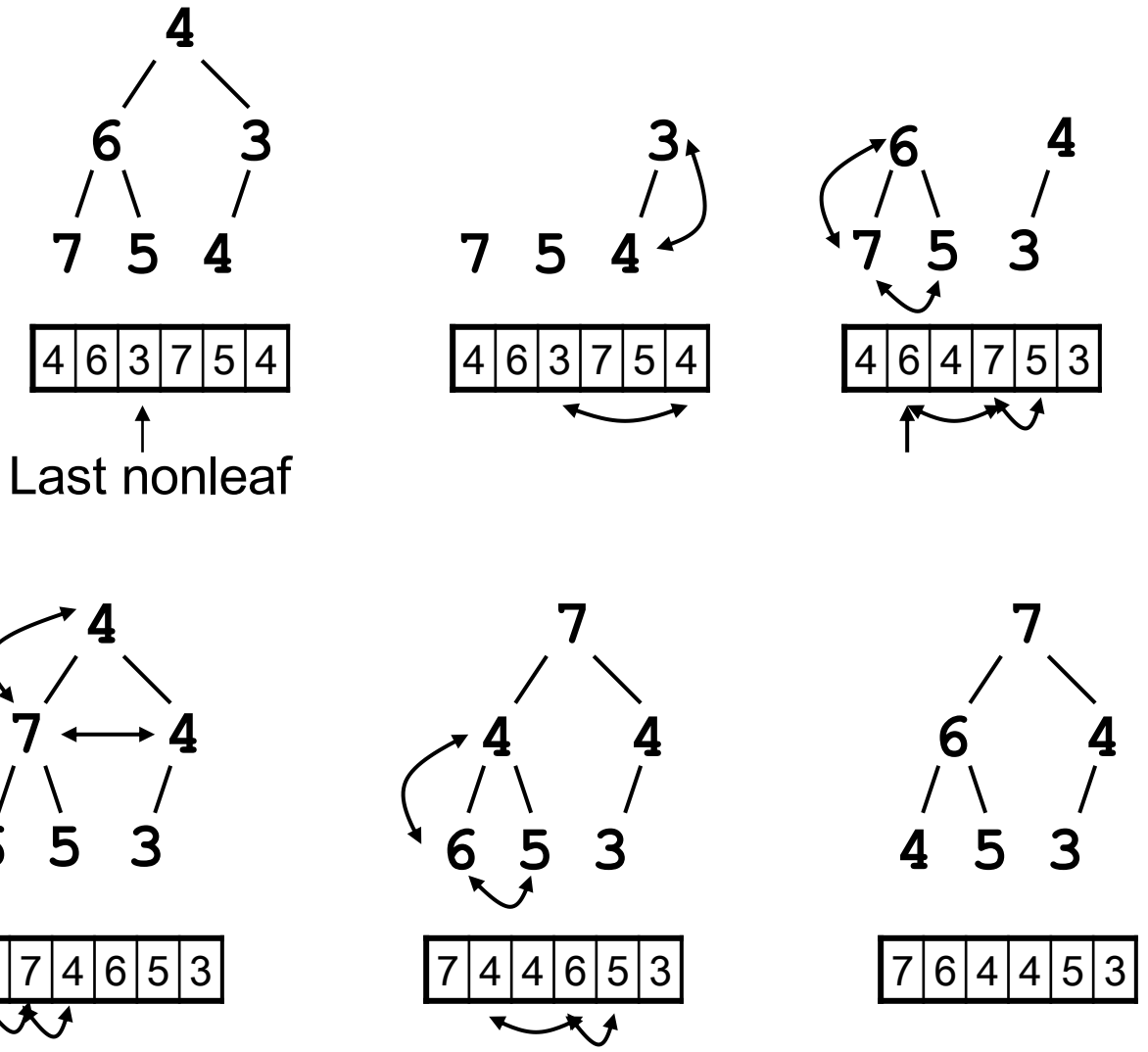
- Phase 1: Transform data into a max-heap

```
FloydAlgorithm(data[])
    i = index of the last nonleaf
    while i >= 0
        p = data[i];
        while p is not a leaf and p < any of its children
            swap p with the larger child;
    i = i--; // index of the previous non-leaf
```

- Floyd's algorithm
  - Start with the last non-leaf node
  - While the non-leaf node has a smaller value than any of its children
    - Swap the non-leaf node with the largest of these children
    - Stop if no larger valued children found, or non-leaf node becomes a leaf
  - Continue with the previous non-leaf node
  - Efficiency is O(n)

Heap Sort

- Phase 1: Transform data into a max-heap (Floyd's algorithm)



Heap Sort

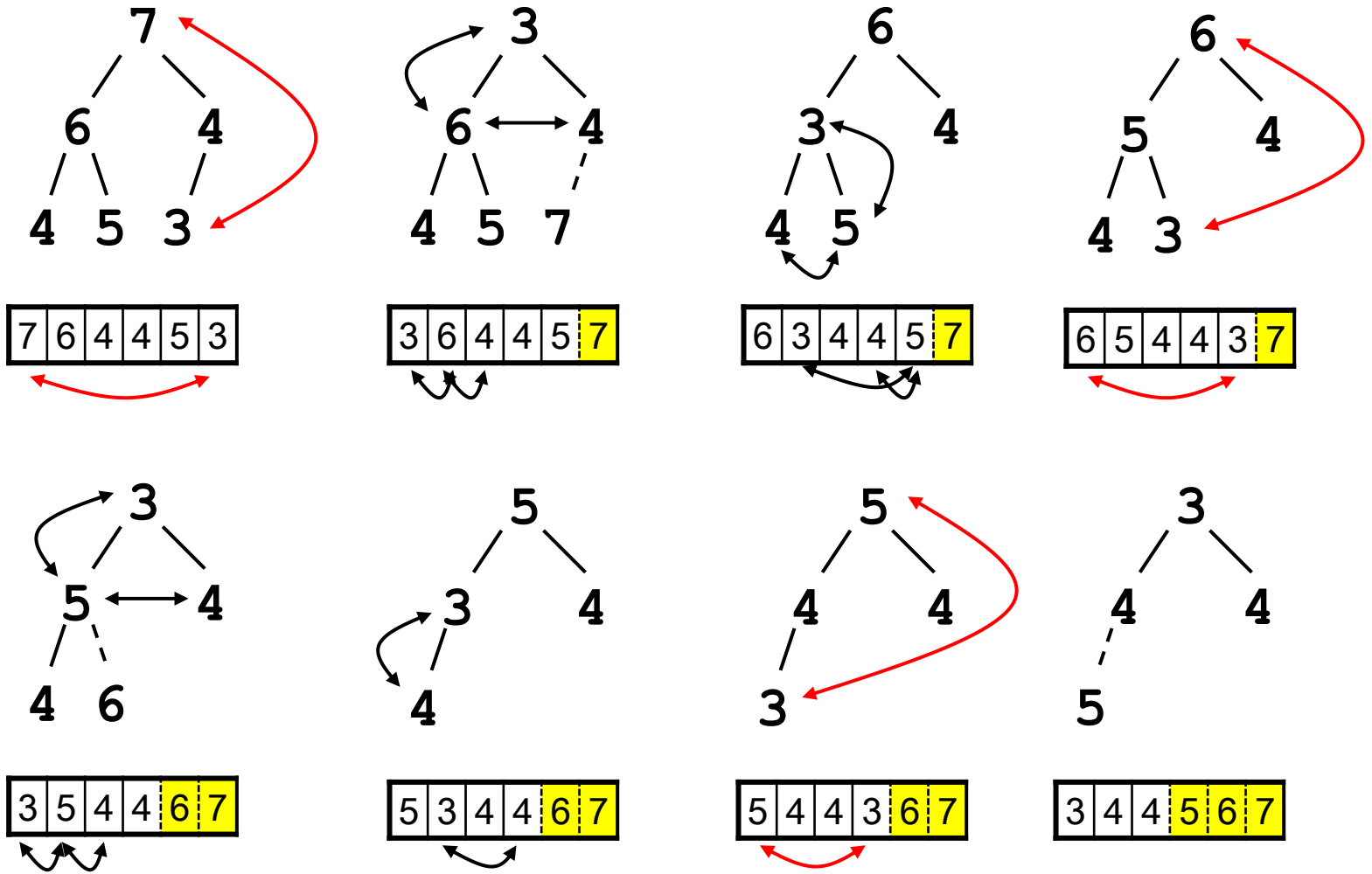
- Phase 2: The sorting procedure

```
for i = data.length-1 down to 2
  swap the root with the element in position i;
  p = the root;
  while p is not a leaf and p < any of its children
      swap p with the larger child;
```

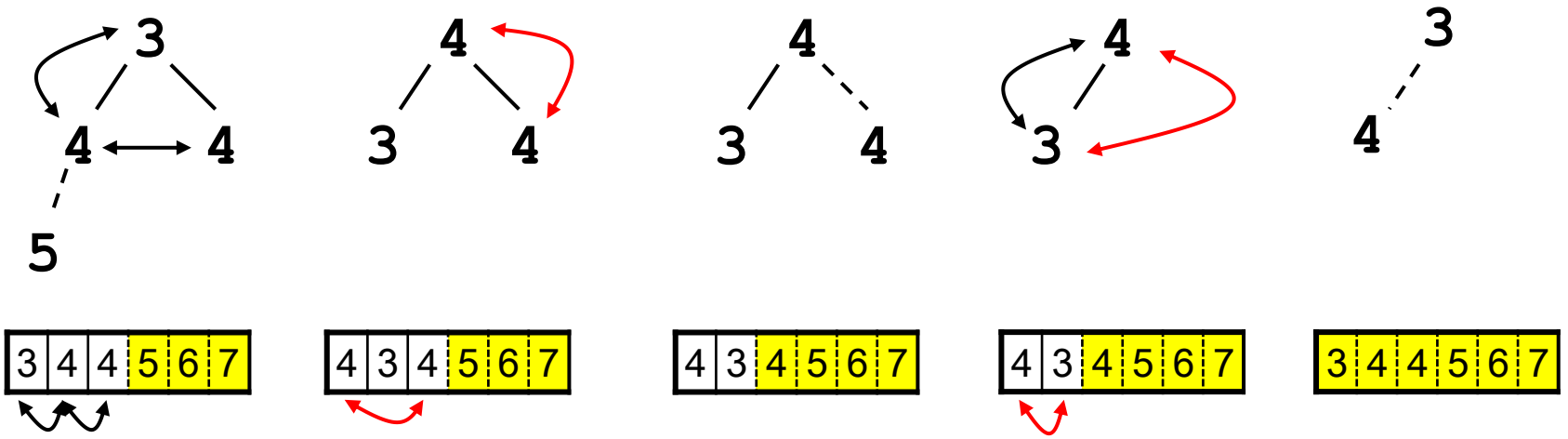
- Dequeuing algorithm
  - Swap largest element (always at data[0]) with last leaf (at data[i])
    - Both are efficient to work with, due to direct access
  - Restore the heap property
    - Set p to the root of the heap
    - While p is less than any of its children, swap p with its largest child
    - Efficiency is O(lg n)

Heap Sort

- Phase 2: The sorting procedure



Heap Sort



Heap Sort

- Efficiency of heap sort?

```
heapsort(data[])
  transform data[] into a heap;  // Floyd's heapifying algorithm
  for i = data.length-1 down to 2
    swap the root with the element in position i;
    restore the heap property for the tree data[0], …, data[i-1];
```

O(n)

O(n)

O(lg n)

Best case:
O(n lg n)

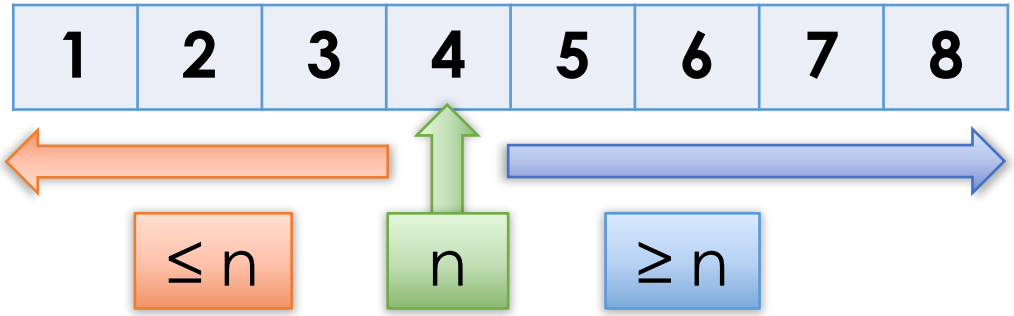
Worst case:
O(n lg n)

Average case:
O(n lg n)

COS 212
Sorting:
Quicksort and Mergesort

Properties of a Sorted Array

- What properties does a sorted array have?

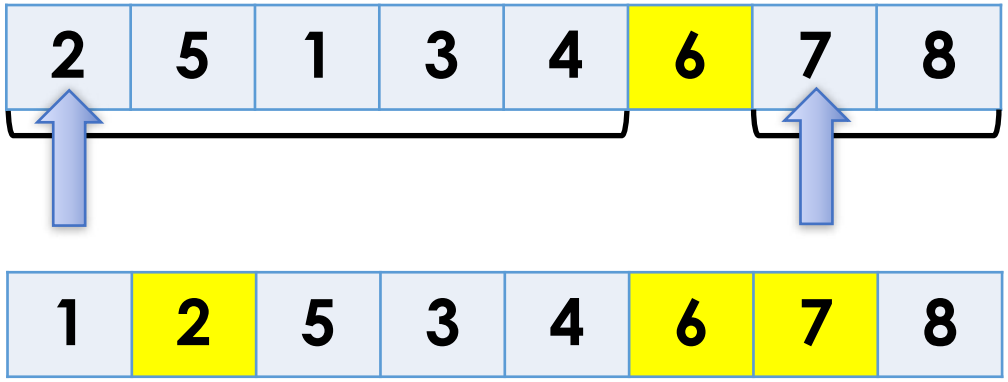


- For any value n
  - All elements to the left of n are less than or equal to n
  - All elements to the right of n are greater than or equal to n

- In an unsorted array, pick a pivot called n
  - Put all elements ≤ the pivot value to the left of n
  - Put all elements ≥ the pivot value to the right of n
  - Now 6 is in its sorted position, and does not need to move again

Quicksort

- Now, apply the same process to the left and right sides of 6



- Now 2 and 7 are both in their sorted positions
- There is no need to move either 2 or 7 again
- We repeat this process until every sub-array is of size 1
- When that point is reached, the array will be sorted
- This recursive sorting algorithm is known as quicksort

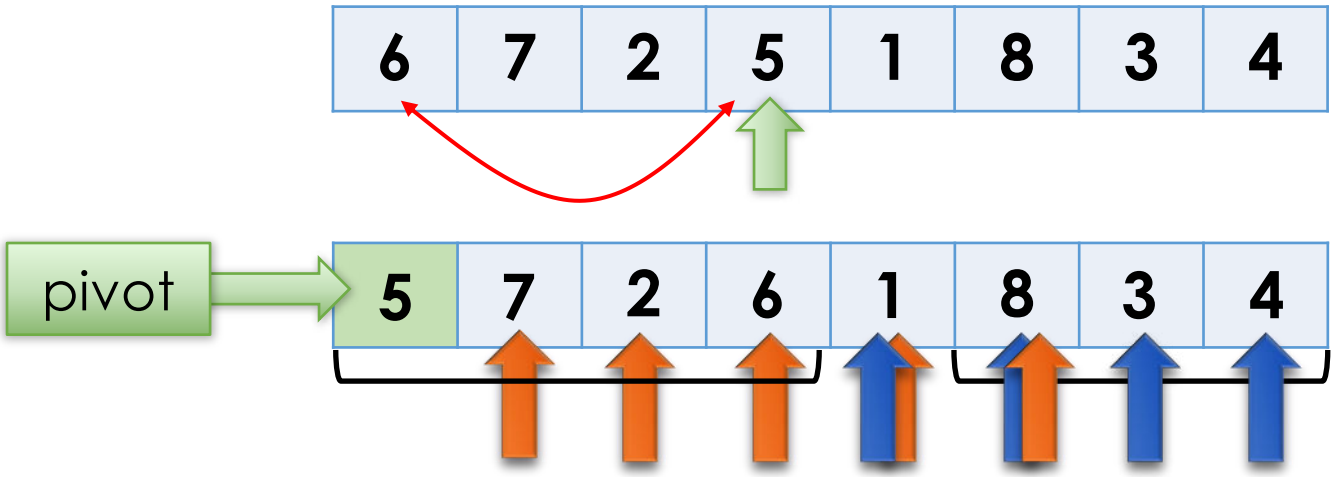
Quicksort

```
quicksort(data[])
     if data.length > 1
       choose pivot;
       while there is an element el left in data
         if (el ≤ pivot) add el to data1[];
         else if (el ≥ pivot) add el to data2[];
       quicksort(data1[]);                          // recursive call
       quicksort(data2[]);                          // recursive call
```

- Elegant, but a lot of details are unspecified
  - How are we going to choose the pivot?
  - Are all pivots equally good?
  - How are we going to reorder the data?
  - Do we need external data structures?
- Middle point is a safer pivot than first or last element
  - This results in more evenly sized sub-arrays
  - Uneven sub-arrays cause poor performance (more later)
- It is much more memory-efficient to use only one array

Quicksort: Partitioning algorithm

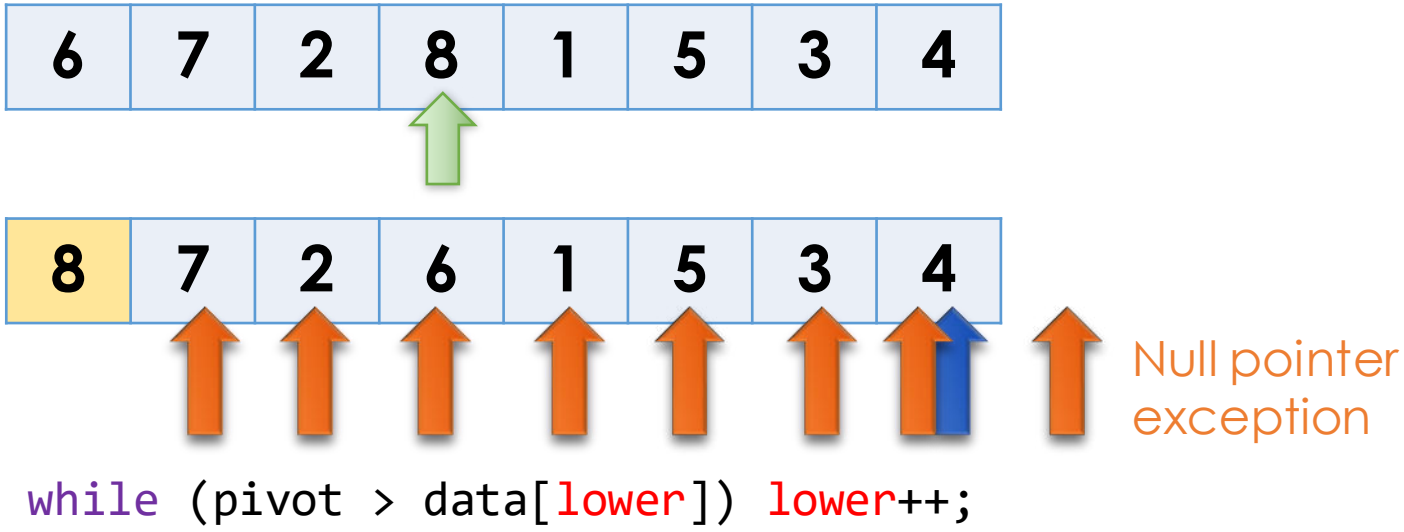
```
swap(data[first],data[(first + last)/2]); // swap pivot & 1st
lower = first + 1, upper = last;          // indices of 1st & last
T pivot = data[first];                    // store value of pivot
while (lower <= upper) {                  // partition the data
   while (data[lower] < pivot) lower++;   // skip values < pivot
   while (data[upper] > pivot) upper--;   // skip values > pivot
   if (lower < upper)                     // pair must be swapped
       swap(data[lower++],data[upper--]); // perform swap
}
swap(data[upper],data[first]);            // place pivot correctly
```



Exercise: Apply the swap process recursively to the halves on either side of the pivot

Quicksort: Partitioning algorithm

- Let's look at a different full array (i.e. not a sub-array)



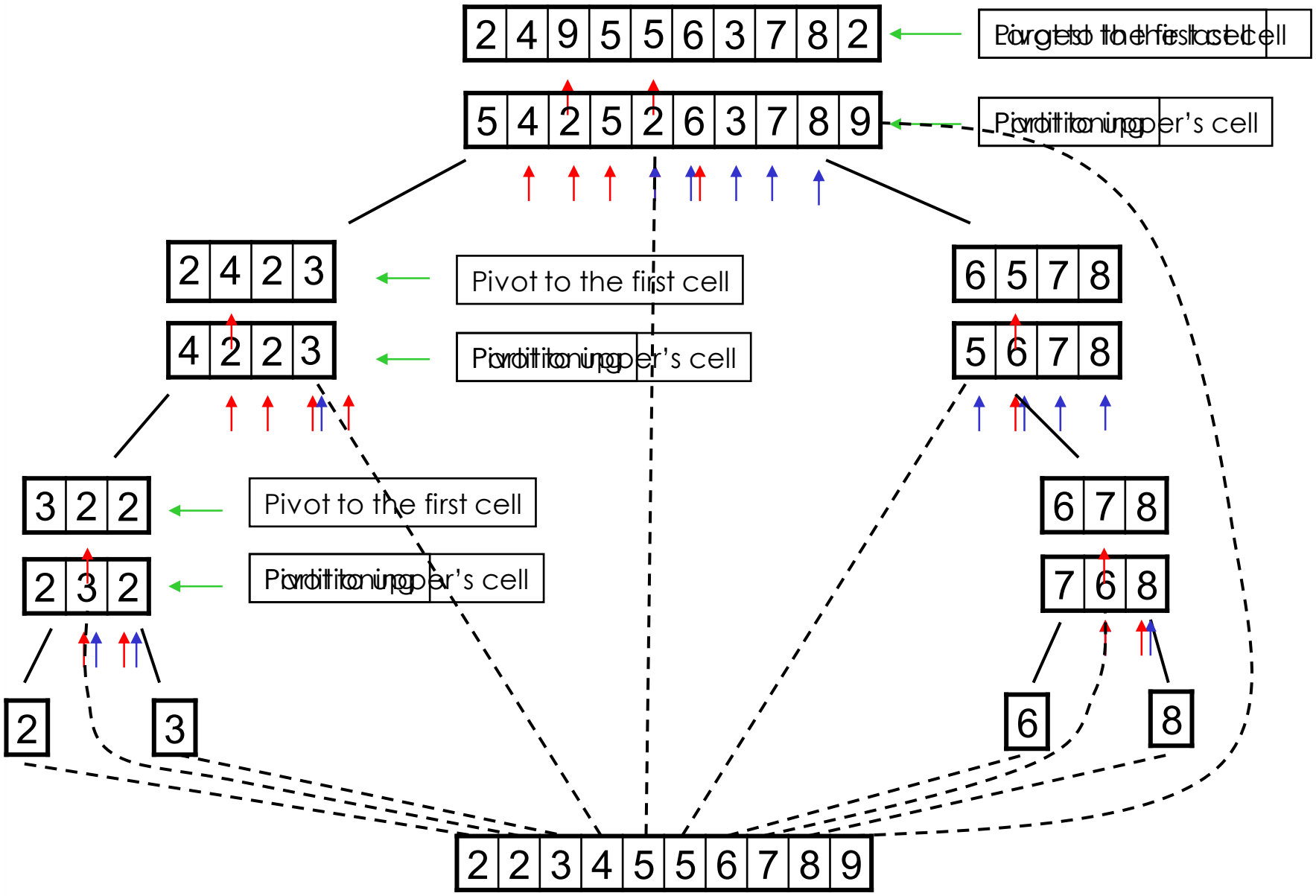

```
while (pivot > data[lower]) lower++;
```

- Simple trick to prevent null pointers
  - Pre-processing step before the first call to the recursive function
  - Find the largest element and swap with the last element

  - Ensures there's at least one element to stop the while loop
  - How does this impact efficiency?

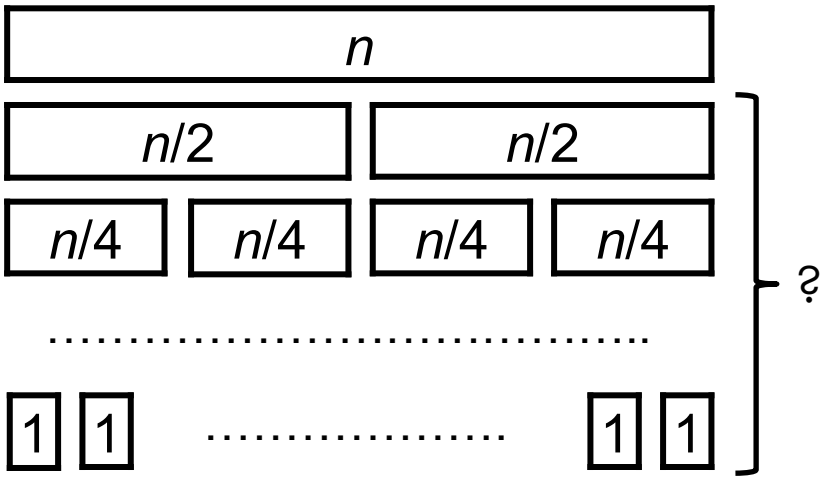
Quicksort: Example

```
while (pivot > data[lower]) lower++;
while (pivot < data[upper]) upper--;
```

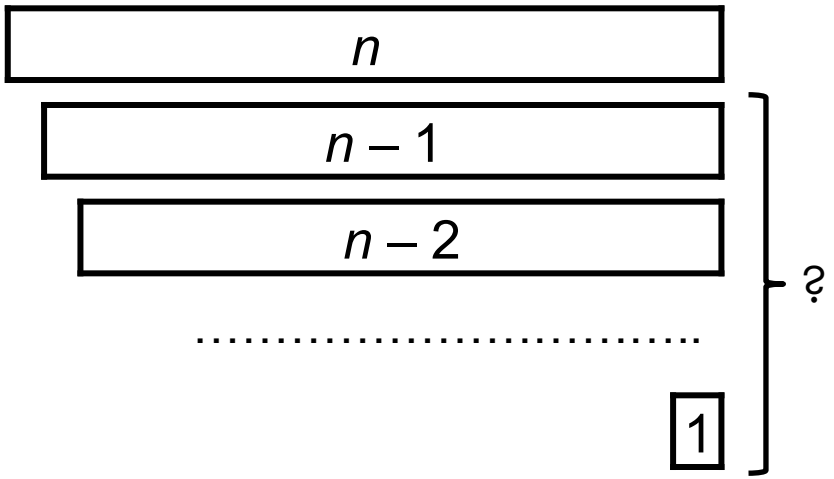


Quicksort: Efficiency

Best case: Selected pivots create two even-size arrays at every partitioning



Worst case: Selected pivots create one empty array at every partitioning



$n + \underbrace{n + \ldots + n}_{} = n \lg n$

How many times can we divide n by 2?

lg *n* times

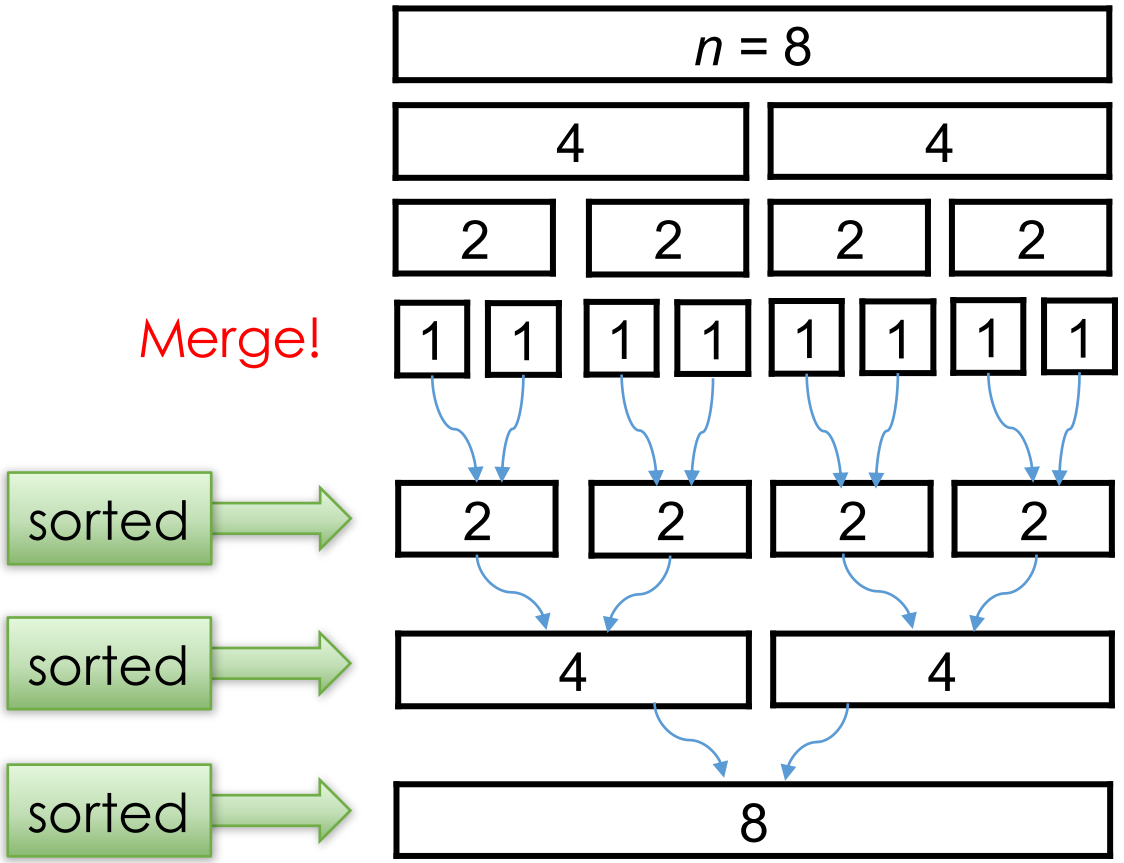
Therefore (lg *n*) + 1 total subsets

$$(n–1) + (n–2) + \ldots + 1 = \frac{(n–1)\,n}{2} = \frac{n^2 – n}{2}$$

Quicksort		
Best	Average	Worst
O(n lg n)	O(n lg n)	$O(n^2)$

Mergesort

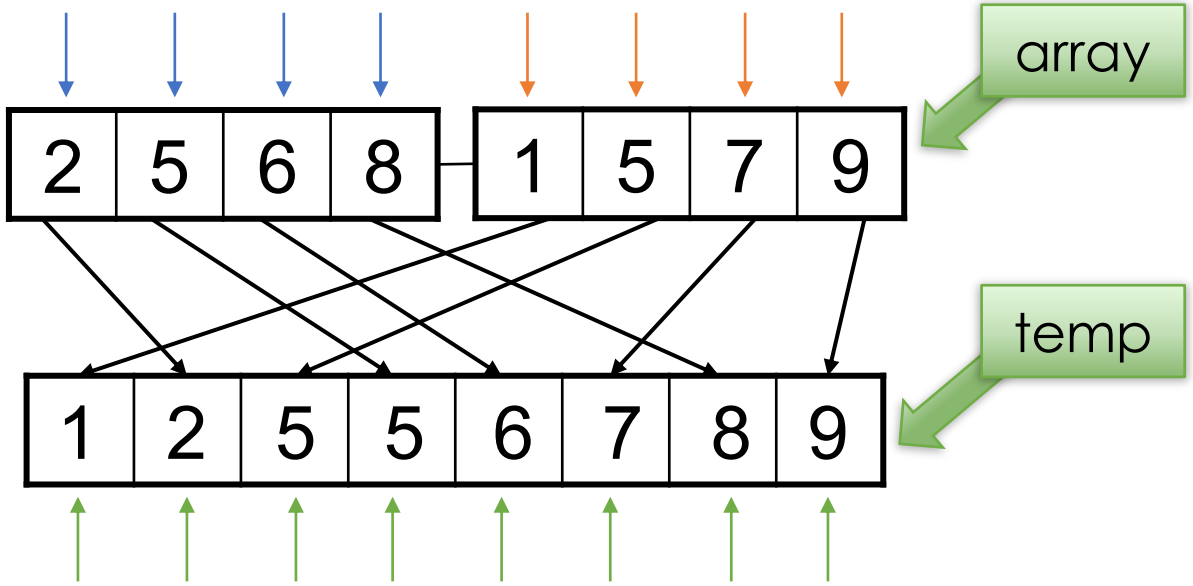
- Quicksort is efficient, but rarely performs optimally
  - This is because perfect pivots are hard to find
  - Quicksort partitions the array, then sorts each partition
  - Can we partition perfectly, then merge partitions while sorting?



If we always halve the arrays, we ensure logarithmic performance

How are we going to merge the sub-arrays so that they're sorted?

Mergesort



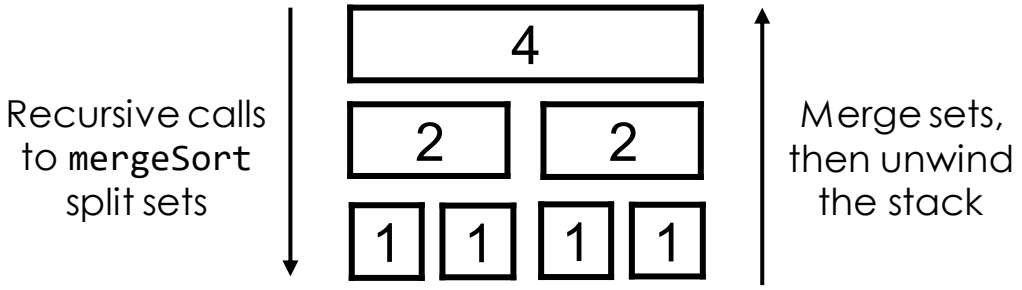
Can we do this without the temp array?

Look at 1, which is copied to the 1st array position

If this happens in the original array, it overwrites 2 and leaves two copies of 1 in the array

```
merge(array[], first, last)    // first and last are positions in array
   mid = (first + last) / 2;   // midpoint of array
   it1 = 0;                    // 1st element in temp
   it2 = first;                // 1st element in 1st subset of array
   it3 = mid + 1;              // 1st element in 2nd subset of array
   while both left and right sub-arrays contain elements
        if array[it2] < array[it3]
             temp[it1++] = array[it2++];
        else temp[it1++] = array[it3++];
   load remaining array[] elements into temp[];
   replace array[] with temp[];
```

Mergesort



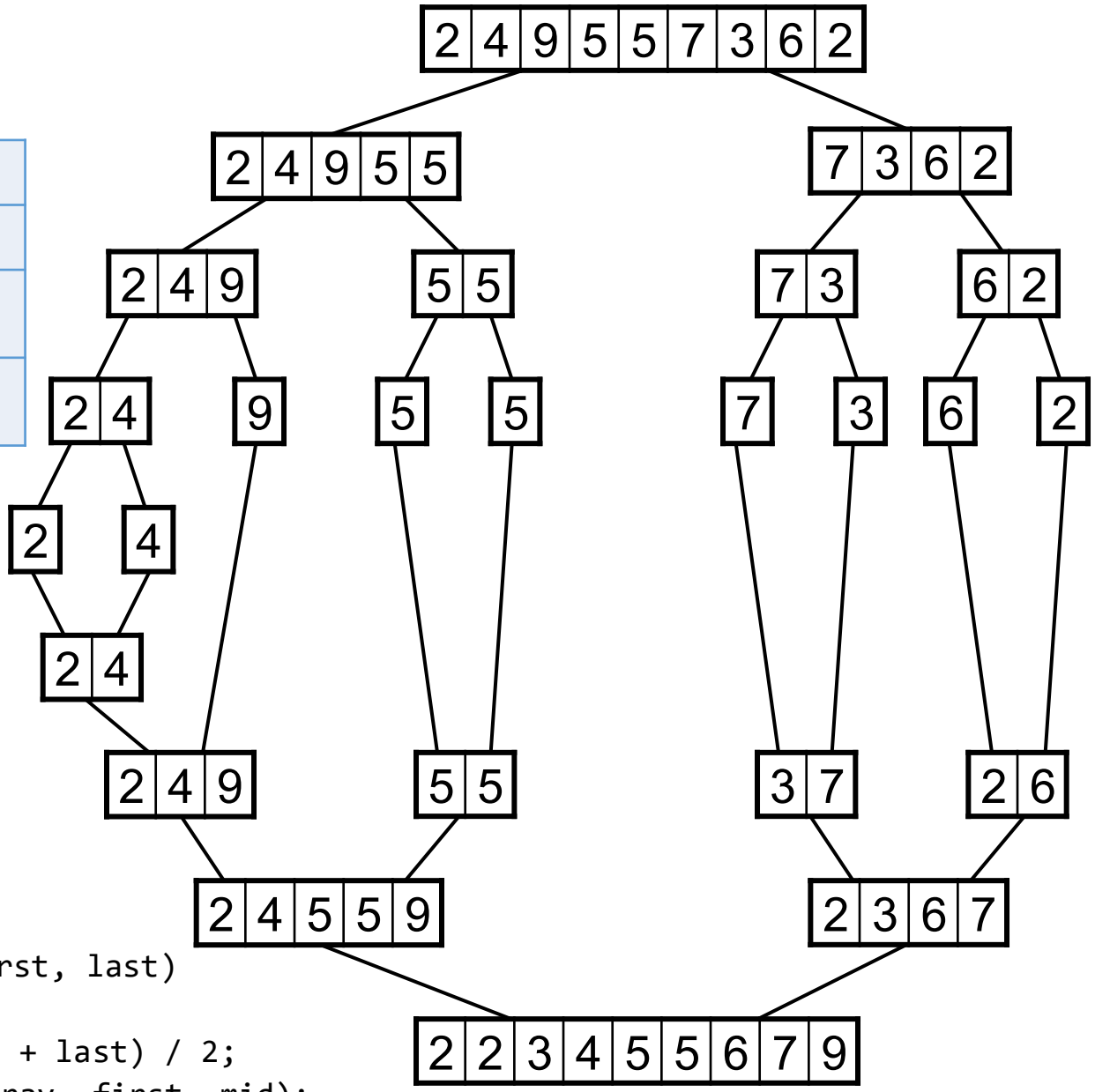

```
mergeSort(array[], first, last)
   if first < last                          // stops at singleton set
        mid = (first + last) / 2;           // find middle of array
        mergeSort(array, first, mid);       // sort left half
        mergeSort(array, mid + 1, last);    // sort right half
        merge(data, first, last);           // merge and sort halves
```

```
merge(array[], first, last)
   mid = (first + last) / 2;
   it1 = 0;
   it2 = first;
   it3 = mid + 1;
   while both left and right sub-arrays contain elements
        if array[it2] < array[it3]
             temp[it1++] = array[it2++];
        else temp[it1++] = array[it3++];
   load remaining array[] elements into temp[];
   replace array[] with temp[];
```

Mergesort

Mergesort	
Best	O(n lg n)
Average	O(n lg n)
Worst	O(n lg n)

Unfortunately, mergesort is memory-wasteful




```
mergeSort(array[], first, last)
    if first < last
        mid = (first + last) / 2;
        mergeSort(array, first, mid);
        mergeSort(array, mid + 1, last);
        merge(data, first, last);
```

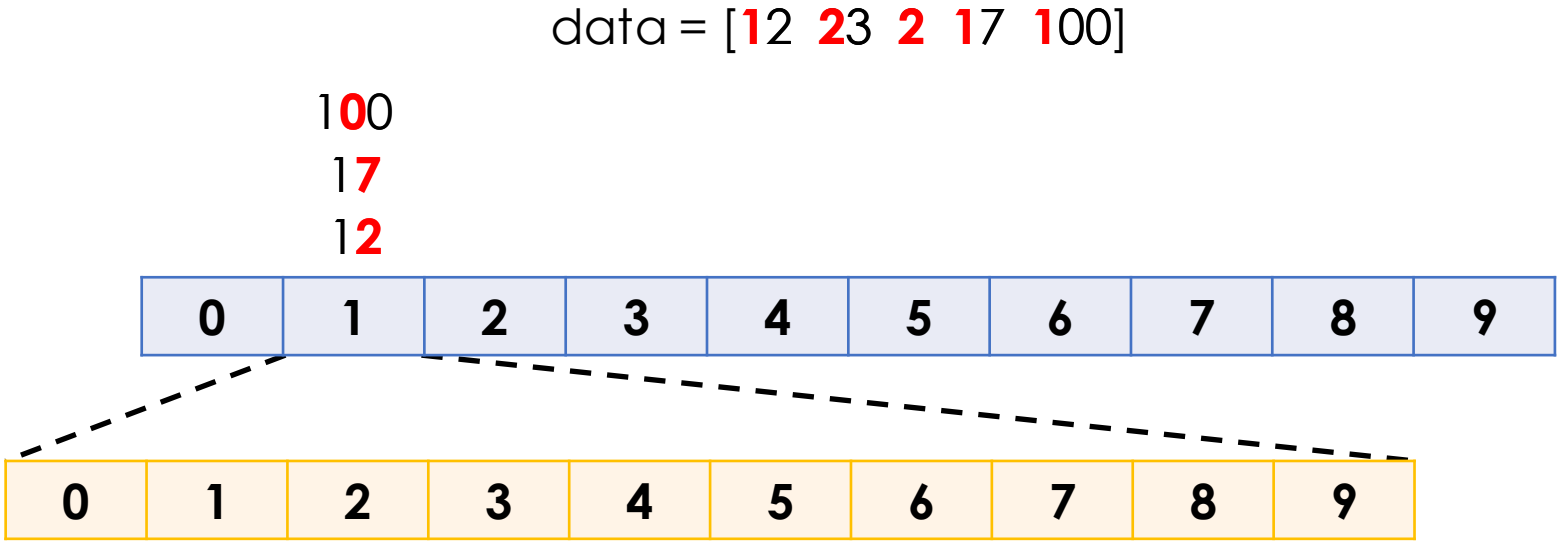
COS 212
Sorting:
Radix Sort & Counting Sort

Radix Sort

- How would you sort a pile of books?
  - You could sort the books by the name of the author
  - Create 26 separate piles, one for each letter of the alphabet
  - Books by authors whose names start with 'A' go in 1st pile, books by authors whose names start with 'B' go in 2nd pile, and so on…
- Now you have your books organised by first letter
  - But there are still a lot of books by different authors in each pile…
  - Sort each one of the 26 piles in the same way
    - Create 26 piles within one of the original piles
    - Books by authors whose names have a second letter of 'A' go in 1st pile, books by authors whose names have a second letter of 'B' go in 2nd pile, and so on…
- Do this until only one author's books are in each pile
- This is a legitimate sorting algorithm used in the real world

Radix Sort

- Can we do the same thing with numbers?
  - We can sort by each digit, creating 10 piles (for digits 0 to 9)
  - Let's see how this works if we start with the unit digit & move right



- But we immediately have a problem
  - The values starting with 1 are sorted into the incorrect order
  - Clearly sorting left to right doesn't work

Radix Sort

- How could we solve this problem?
  - Pad numbers with leading zeros to make them the same length
  - Sort right to left, starting with the rightmost digit
- To store the multiple values that map to a digit
  - We'll use one queue per digit
- Our previous example also wastes a lot of memory
  - A separate destination array for each digit
  - Process repeated as many times as maximum digits in a number
  - Leads to a proliferation of arrays
  - We'll move numbers back and forth between the array we're sorting and a single array of queues

```
radixSort(data[])
   n = number of digits in largest number
   add leading zeros to numbers until all have n digits
   for d = n down to 1
      distribute numbers in data[] among queues 0 to 9 according to digit d
      for i = 0 up to 9
         dequeue each number in queue i and add to next position in data[]
```

Radix Sort

- Let's apply radix sort to the array [25 5 20 315 11 77]

data = [02**5** 00**5** 02**0** 31**5** 01**1** 07**7**]

0	1	2	3	4	5	6	7	8	9

Radix Sort

- Let's apply radix sort to the array [25 5 20 315 11 77]

data = [020 011 025 005 315 077]

0	1	2	3	4	5	6	7	8	9

Radix Sort

- Let's apply radix sort to the array [25 5 20 315 11 77]

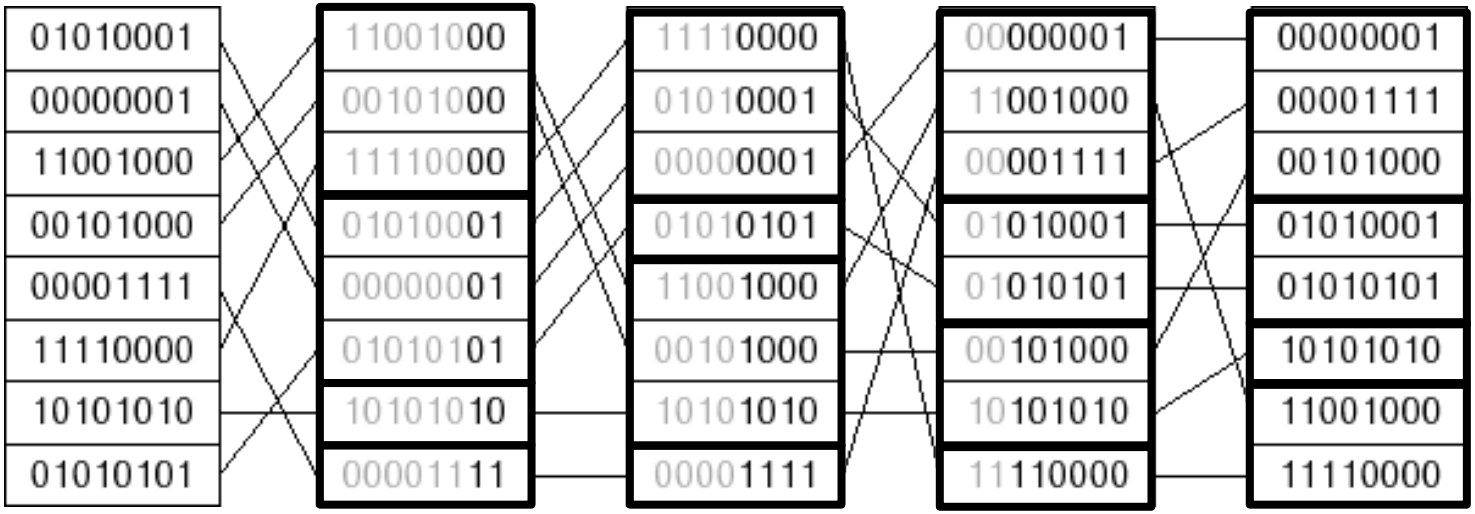
data = [005 011 315 020 025 077]

= [5 11 20 25 77 315]

0	1	2	3	4	5	6	7	8	9

Do you think it's possible to implement radix sort recursively?

Radix Sort



- Radix sort can be applied to binary representations
  - At each pass, sort according to b bits, starting from the right
  - How many queues?
    - Determined by the number of different bit strings of size b
    - For b = 2 there are 2^b bit strings, and therefore 4 queues: 00, 01, 10, 11
  - A larger value for b results in
    - More queues (wastes space) but fewer passes (saves time)
  - A smaller value for b results in
    - Fewer queues (saves space) but more passes (wastes time)

Radix Sort: Efficiency

```
private final int radix = 10;       // number of digits and piles
private final int digits = 10;      // max number of digits in a number

public void radixsort(int[] data) {
        int d, j, k, factor;
        Queue<Integer>[] queues = new Queue[radix]; // 1 queue per pile
        for (d = 0; d < radix; d++)
            queues[d] = new Queue<Integer>();
        // run through every digit, starting from the rightmost
        for (d = 1, factor = 1; d <= digits; factor *= radix, d++) {
            for (j = 0; j < data.length; j++)       // enqueue data[j]
                queues[(data[j] / factor) % radix].enqueue(data[j]);
            for (j = k = 0; j < radix; j++)         // rebuild data
                while (!queues[j].isEmpty())
                    data[k++] = queues[j].dequeue();
        }
}
```

O(1)

O(d)

O(n)

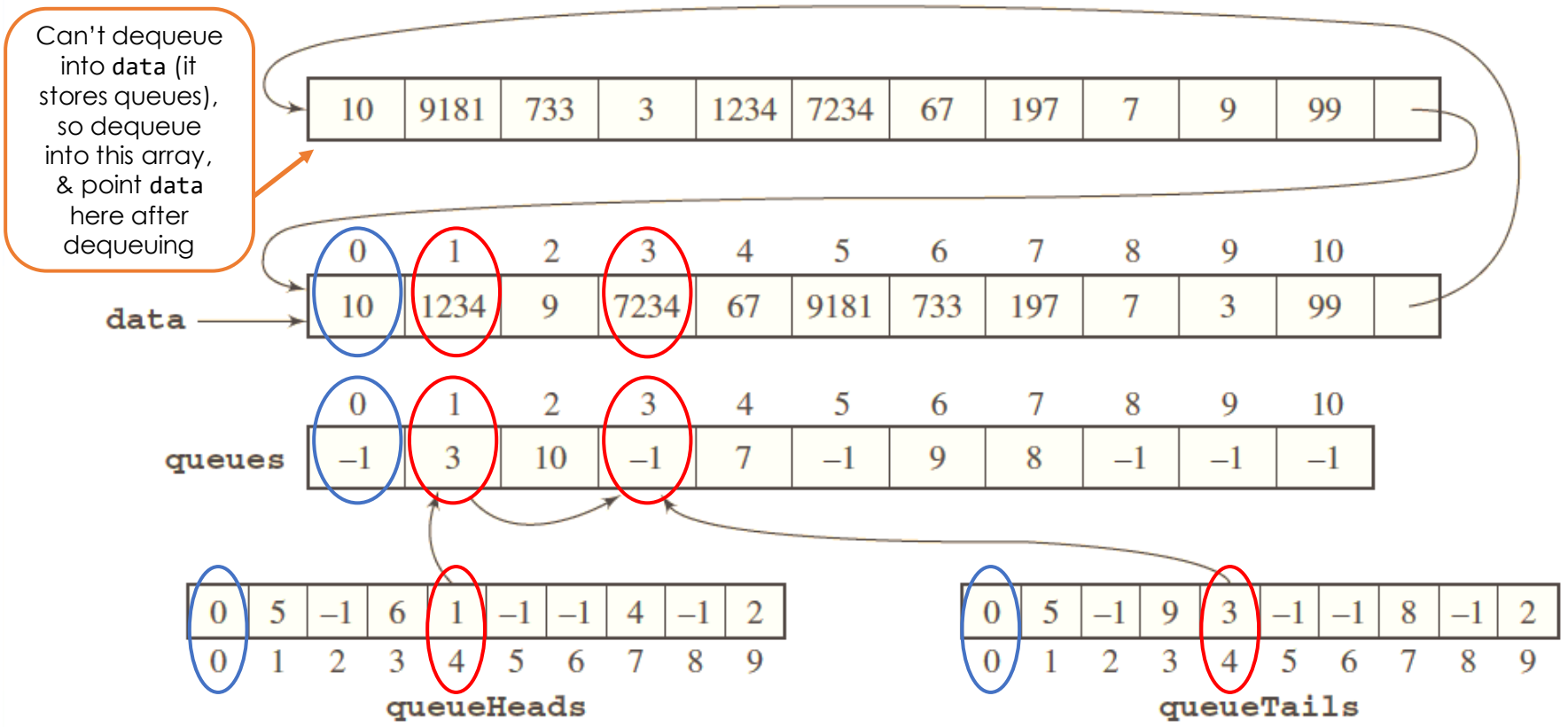
O(n)

Total complexity?

O(dn)

What about the complexity of using queues?

Radix Sort: Reducing Time Complexity



Instead of storing multiple queues, use three arrays

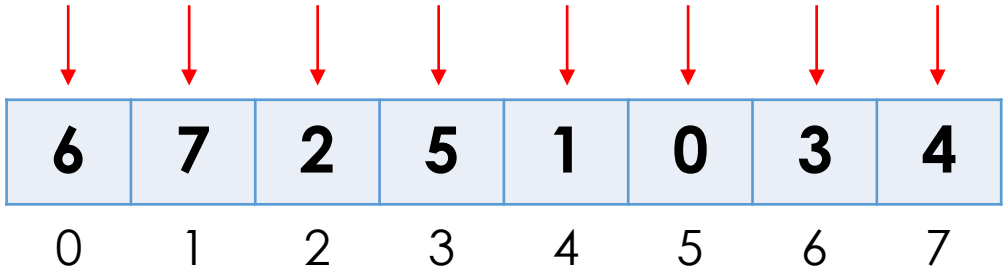
`queues[]` stores "linked lists" of data indices for each queue

`queueHeads[]` stores indices of the first element of each queue, where dequeuing starts in `data`

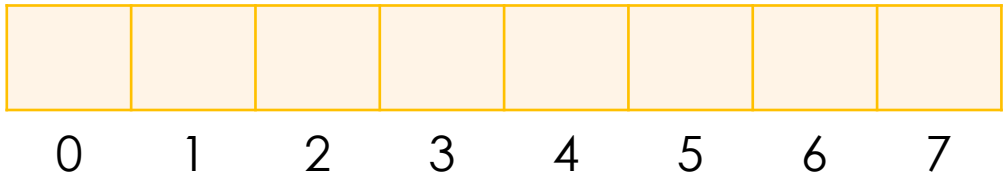
`queueTails[]` stores indices of the last element of each queue, where enqueuing takes place in `queues`

Counting Sort

- Array indices are always sorted (`0` up to `size-1`)



- What if we treat each element as an index?
- Create array `tmp` where last index is the largest data value



What if there are duplicate values?

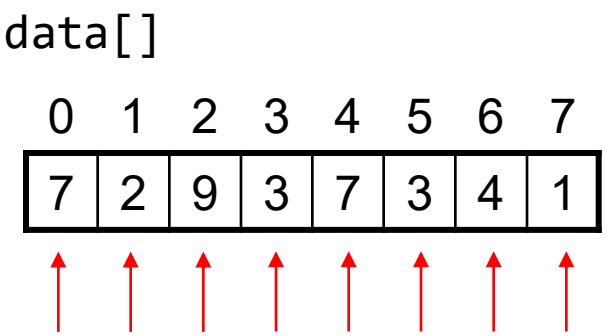
- Populate `tmp` array: `tmp[array[i]] = array[i]`

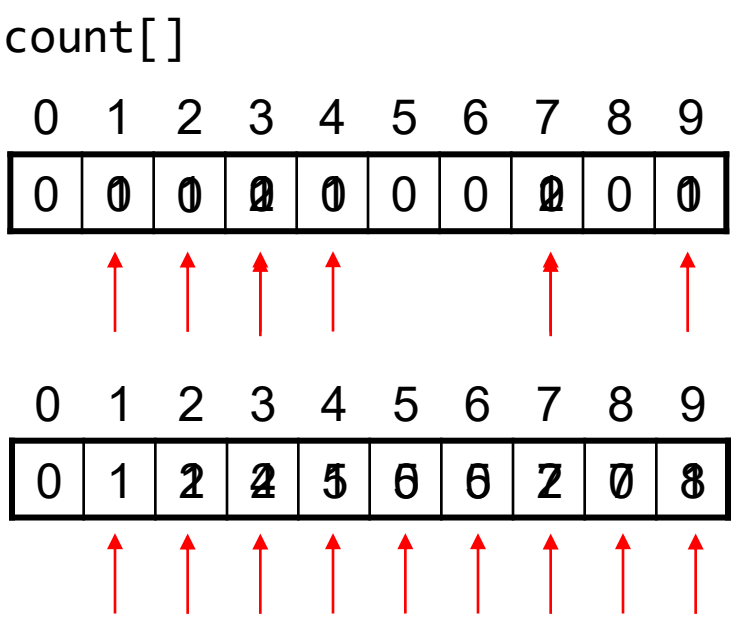
What if there are gaps between numbers?

Counting Sort

```
count occurrences of each number in data[];
store occurrences in count[] indexed with numbers in data[];
for i = 1 up to count.length-1
     count[i] = the number of elements <= i;
// transfer numbers from data[] to tmp[]
for i = n-1 down to 0
     tmp[count[data[i]] - 1] = data[i];
     decrement count[data[i]];
transfer numbers from tmp[] to data[];
```

Last index in `count[]` is equal to the largest number in `data[]`

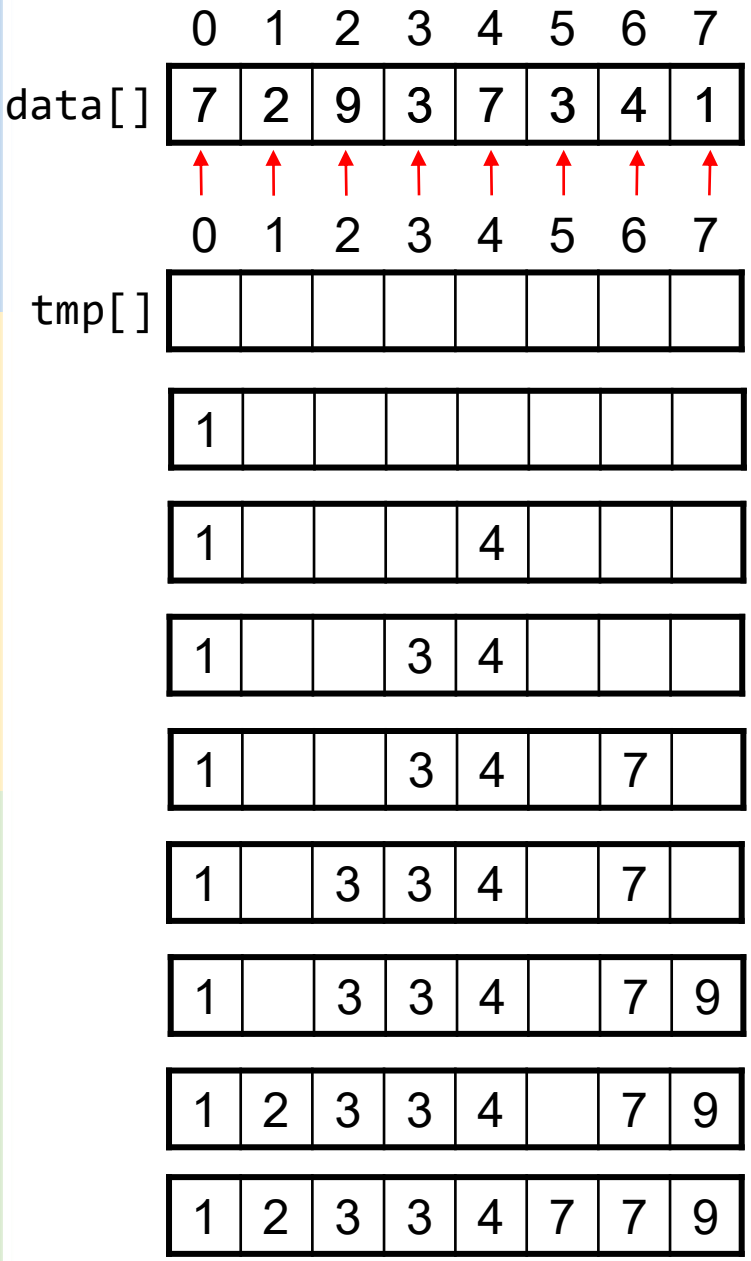


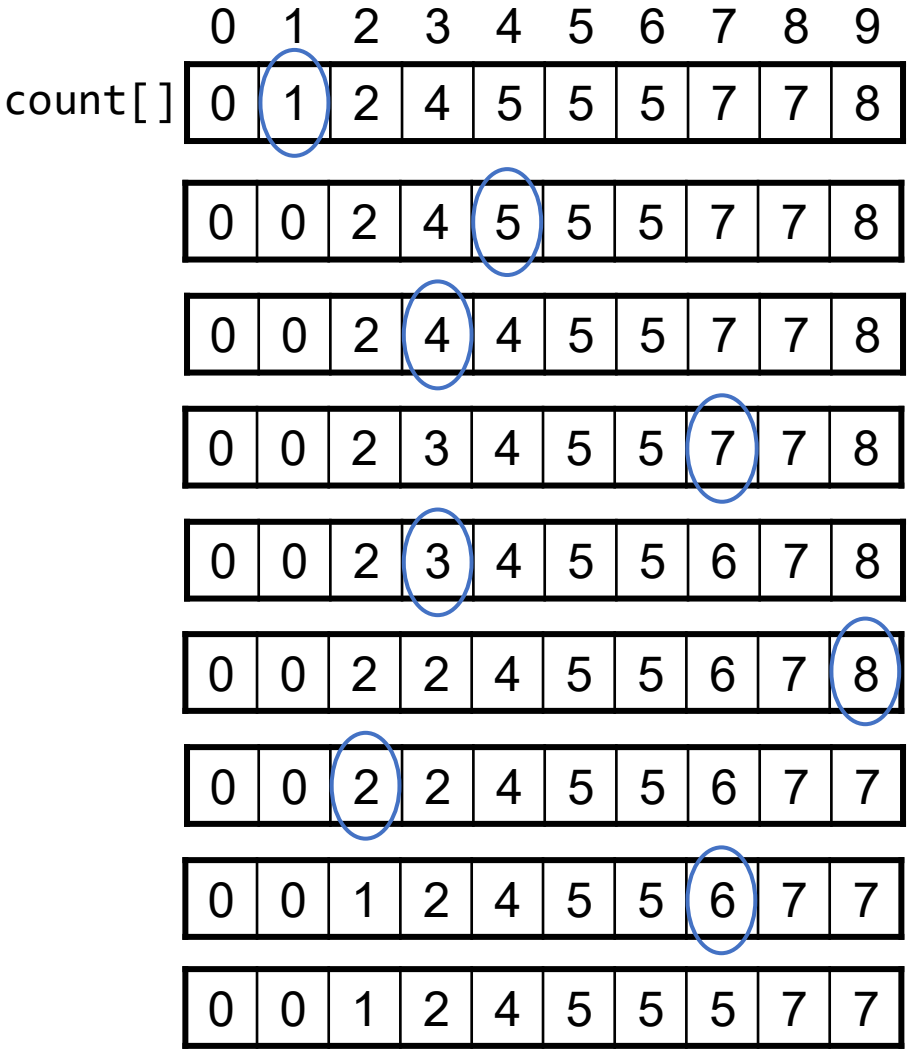


`count[i] = count[i – 1] + count[i]`

Counting Sort

```
for i = n-1 down to 0
  tmp[count[data[i]]-1] = data[i];
  decrement count[data[i]];
```





Counting Sort: Complexity

```
count occurrences of each number in data[];
store occurrences in count[] indexed with numbers in data[];
for i = 1 up to count.length-1
     count[i] = the number of elements <= i;
// transfer numbers from data[] to tmp[]
for i = n-1 down to 0
     tmp[count[data[i]] - 1] = data[i];
     decrement count[data[i]];
transfer numbers from tmp[] to data[];
```

O(n)

O(n)

O(n)

How many times will the loop execute?

O(max)

Total complexity?

O(n + max)

- Imagine `data` looks like this:

3	5	100000000

This number determines max

- Counting sort is very efficient, but only if max is not much larger than n
- Can counting sort work on non-integers?

Which Algorithm Should You Use?

	80,000 Integers		
	Ascending	Random	Descending
Insertion sort	.11	29 m 2.73	29 m 36.13
Selection sort	56 m 8.09	67 m 21.31	56 m 49.94
Bubble sort	52 m 6.90	87 m 9.62	83 m 6.68
Comb sort	.67	6.52	3.10
Shell sort	1.32	2.75	1.59
Heap sort	3.63	4.56	3.35
Merge sort	2.19	3.35	2.20
Quick sort	.93	2.04	.99
Radix sort	10.98	10.82	10.00
Bit Radix sort	28.40	31.04	29.00
Radix sort 2	1.16	1.26	1.15
Bit Radix sort 2	1.83	2.42	1.98
Counting sort	.22	.57	.20

It will always depend on the application
Just be sure to avoid the simple sorting algorithms!

COS 212
Hashing

The problem of searching

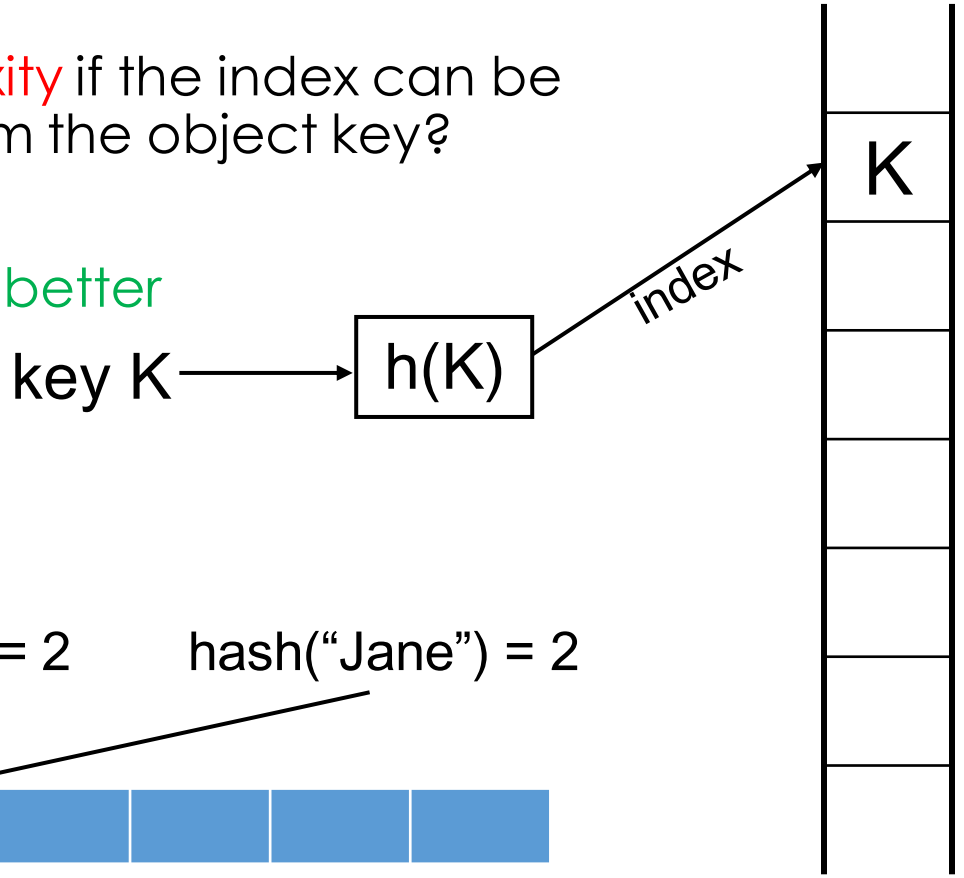
	Average case			Worst case		
	search	**insert**	**delete**	**search**	**insert**	**delete**
Unsorted array	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)
Sorted array	O(lg n)	O(n)	O(n)	O(lg n)	O(n)	O(n)
Unsorted linked list	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)
Sorted linked list	O(n)	O(1)	O(1)	O(n)	O(1)	O(1)
Skip list	O(lg n)	O(1)	O(1)	O(n)	O(1)	O(1)
BST	O(lg n)	O(lg n)	O(lg n)	O(n)	O(n)	O(n)
B-Tree	O(lg n)	O(lg n)	O(lg n)	O(lg n)	O(lg n)	O(lg n)

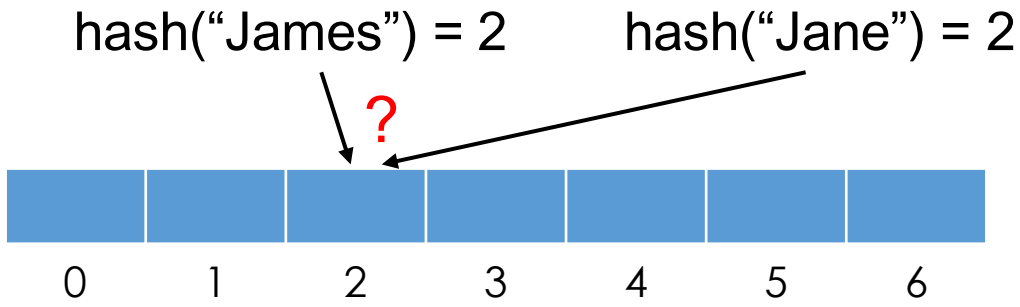
- What if we invented a function that, given the object key, can instantly calculate the index of the object?

Hash functions

- Basic idea: Save items in a key-indexed table (index is a function of the key)
- Hash function: Method for computing table index from key

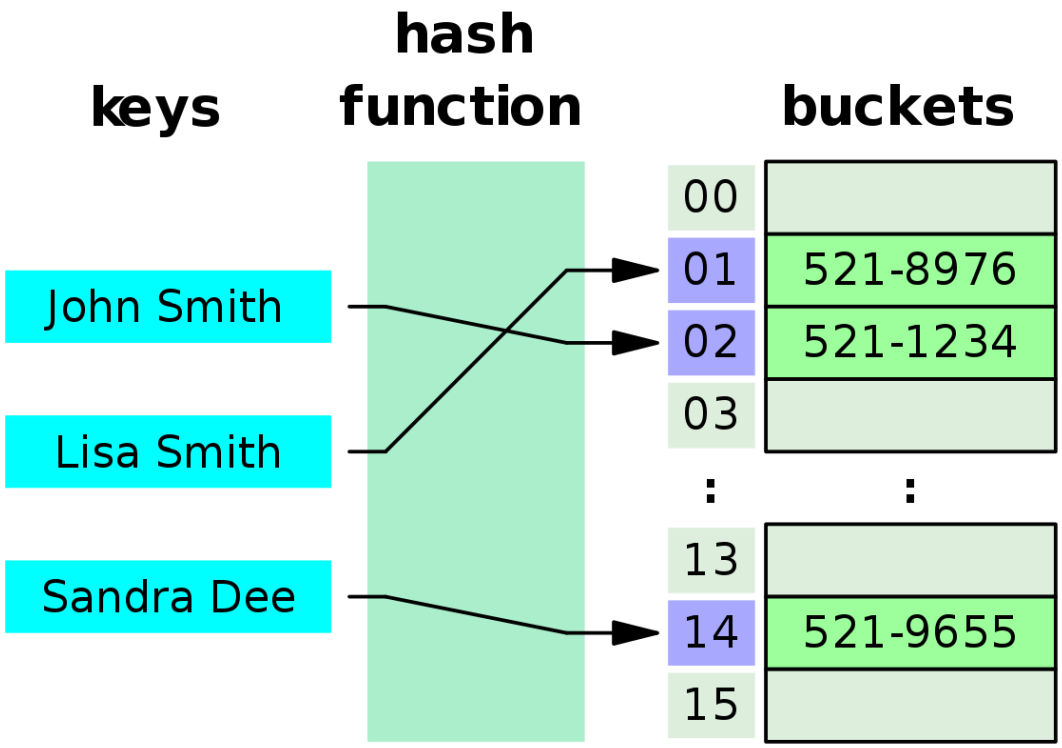
- What is the search complexity if the index can be instantaneously derived from the object key?
- O(1)
- The holy grail: you can't do better
- What we want from a hash function:
  - Easy to compute
  - No collisions





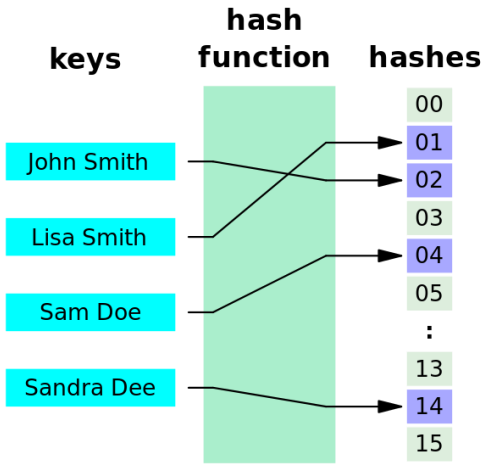
Hash table (hash map)

- Array-like data structure: associative array
- Maps keys to indexes associated with values
- A hash function is used to compute the index of each key

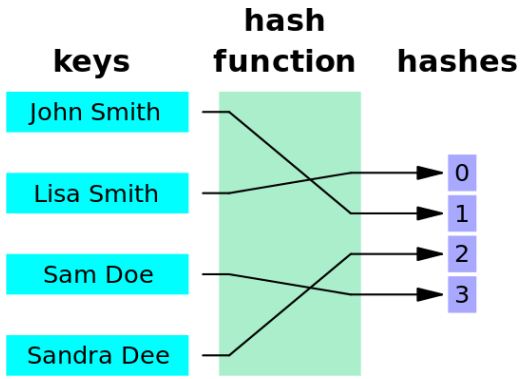


Hash functions

- What makes a good hash function?
  - It must be easy (efficient!) to calculate
  - Each index is equally likely to be generated for each key
    - A hash function where every key gets a different hash value is a perfect hash function



Perfect Hashing



Minimal Perfect Hashing

  - Example: how would you hash phone numbers?
    - Bad hashing: use first three digits (shared area codes)
    - Better hashing: use last three digits (unique extension)
    - Neither is perfect

Hash functions: Extraction

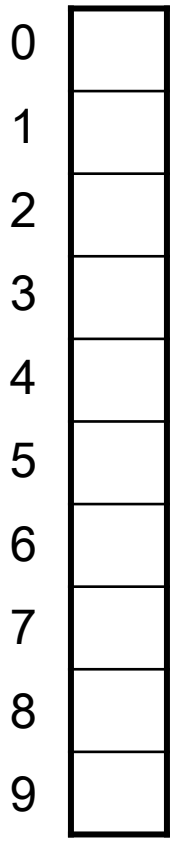
- Use a selection of digits as the key
- Eg, use the first three digits; or the last three; or first, last, and middle
- Choice of the appropriate subset depends on the application

- Part of the key that is least likely to repeat should be chosen
- This hashing function is extremely simple => efficient!
- Disadvantage: part of the key is used -> less variation

What assumption are we making by extracting 3 digits?

Hash functions: Division

- Suppose the size of the hash table is T
- The indices generated by the hash function must be in [0, T - 1]
- If key K is numeric (or interpreted as such):
  - hash(K) = K % T
  - Remainder (modulus) operator will return values in the desired range



33 16 27 25 18 1 72

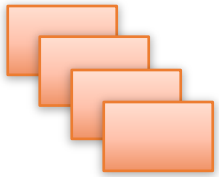
- h(K) = K % 10

- The size of the table is 10
- Now suppose the majority of your keys end in a 0
- Collisions everywhere!
- Division hashing works best for T = large prime number
- Alternatively:
  - hash(K) = (K % p) % T
  - p = prime number > T

Hash functions: Folding

- Designed to increase variability of hash codes
- Split the key into equal parts
- Combine the parts using a simple operation such as addition
- Apply division hashing to restrict the index to the desired range
- Shift folding:

15351871



15 + 35 + 18 + 71 = 139

139 % T

- Boundary folding:

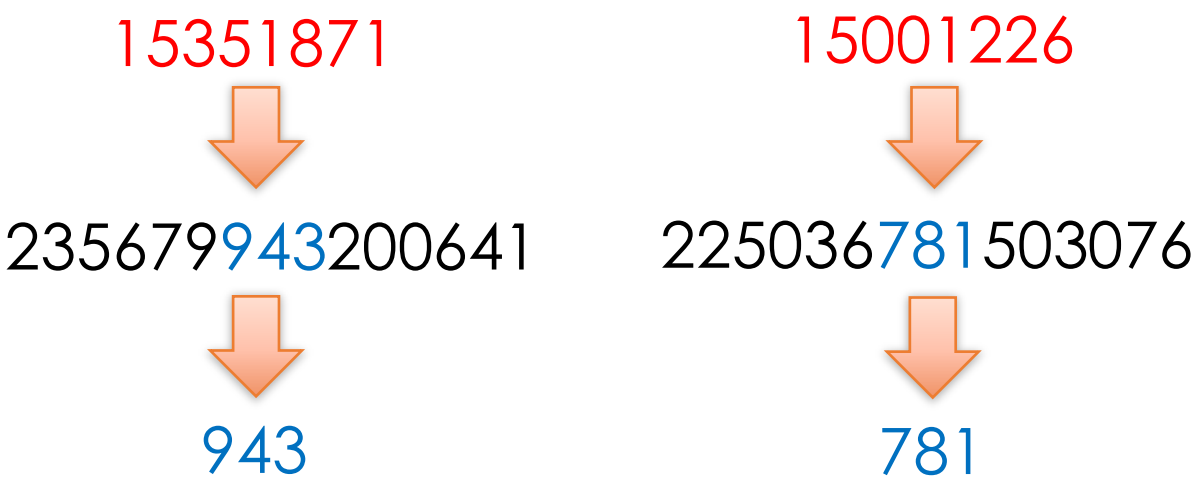
15351871

15 + 53 + 18 + 17 = 103

103 % T

Hash functions: Mid-Square Function

- Square the key
- Use the middle section of a desired length as the hash value



- Advantage: the entire key is used -> more variation
- Hash functions can be combined: apply folding first, followed by mid-squaring
- To simplify calculation: perform the same procedure on binary representations; logic operations alone (shifting & masking) can accomplish the task

Hash functions: Radix Transformation

- Convert the numeric key to another base
- Eg, convert decimal to octal
- Apply division hashing to restrict the index to the desired range

15351871

72440077

72440077 % T

15001226

71163212

71163212 % T

- Can increase variation
- Can be prone to collisions:
  - $(2560)_{10} = (5000)_8$
  - $(2048)_{10} = (4000)_8$

Hash functions: Universal Hashing

- Things often get out of hand with data, and a single hashing function may result in many collisions when data is dense
- Solution: use a set, or a family of hashing functions instead
- For every key, choose one of the functions from the family
- A family of hashing functions is universal if for two keys x != y, the probability of h(x) = h(y) for a randomly chosen h is 1/T (that's a pretty low probability)
- Universal hash function families:
  - H = { $h_{a,b}$ (K): $h_{a,b}$ (K) = ((aK+b) % p) % T }
  - 0 ≤ a, b < p, p > T and is prime
  - a & b are chosen randomly for each key

If **a** and **b** are random, how are we going to find the key after it is stored?

  - H = { h_a (K): h_a (K) = $\left(\left(\sum_{i=0}^{r-1} a_i K_i\right) \% p\right)$ % T } *(if key K is a sequence)*
  - K = $K_0 K_1 \ldots K_{r-1}$ ("cat" = c, a, t)
  - 0 ≤ a = $a_1 a_2 \ldots a_{r-1}$ (a = 4, 7, 2)
  - $\left(\sum_{i=0}^{r-1} a_i K_i\right)$ = 4 * c + 7 * a + 2 * t *(Use ASCII codes!)*

Hash functions: Hashing strings

- Strings are not numbers – how are they to be hashed?
- Use ASCII codes!
- Concatenate the ASCII codes:

ROSE

82, 79, 83, 69

82798369 % T

SORE

83, 79, 82, 69

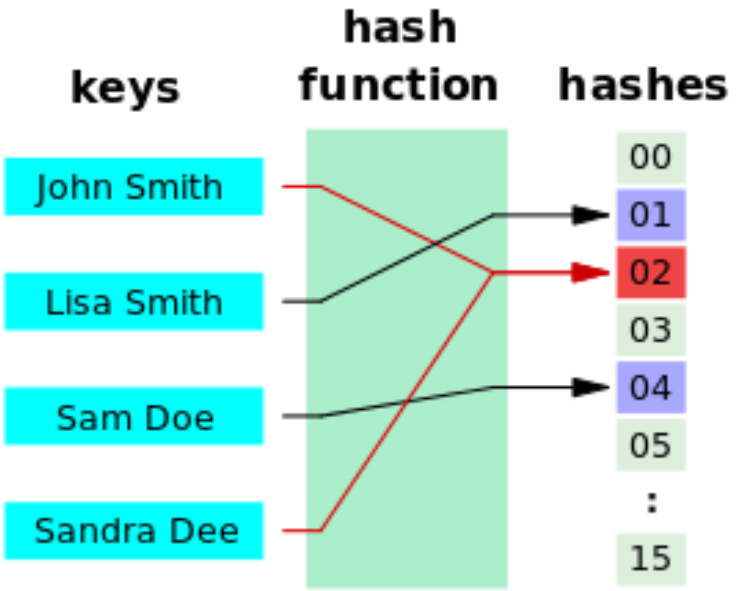
83798269 % T

- Adding the values would have resulted in a collision:
  - 82 + 79 + 83 + 69 = 83 + 79 + 82 + 69
- Even more efficient:
  - Treat strings as binary
  - Treat binary as integers

Hashing: Collision Resolution, Deletion

Collisions in Hashing

- No matter how good the hash function is, there is always a chance of collision
- We map a potentially unbounded space of inputs to a bounded table of entries
- The more entries there are, and the smaller the table => the higher is the chance of collision

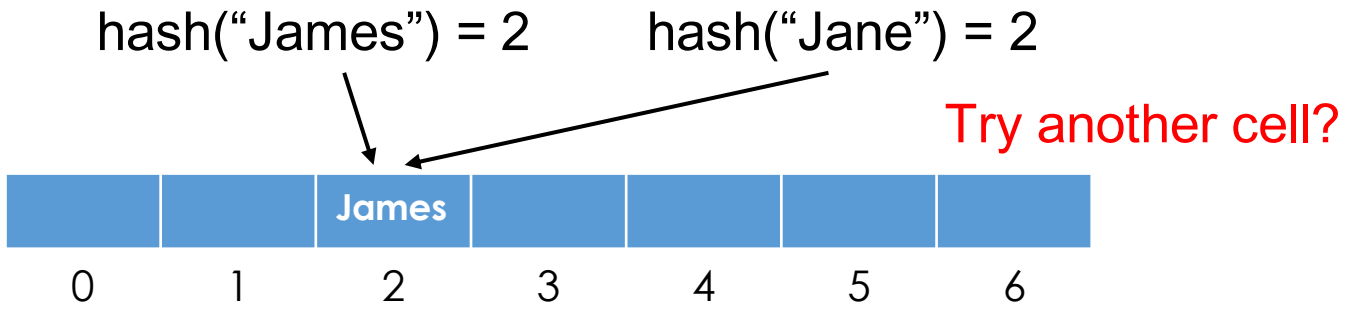


Should we try harder and invent a perfect hash function for all cases?

Can we perhaps devise an algorithm to resolve the collisions instead?

Collision Resolution

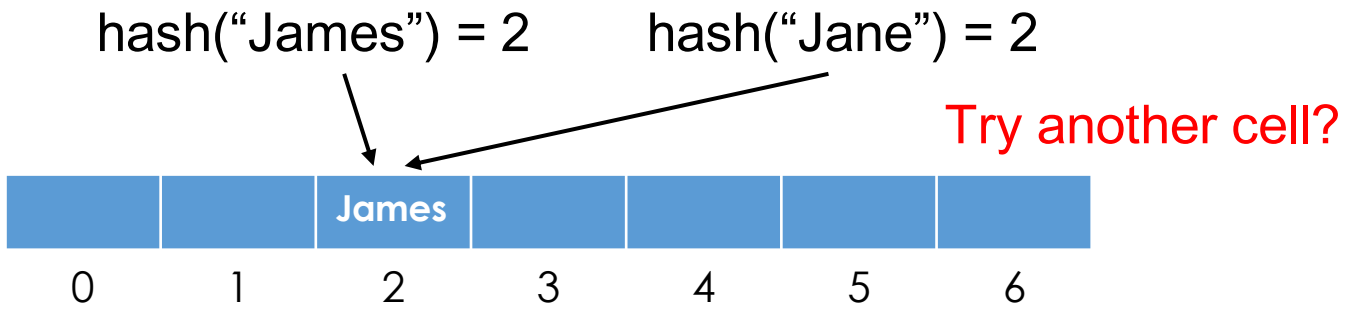
- Simplest approach: if the cell addressed by the hash code is occupied, try another cell



- But which cell must we try?
- The key should still be easily accessible – can't just use a random place
- We need a formula that will derive the next possible position of K

Collision Resolution

- We need a formula that will derive the next possible position of K



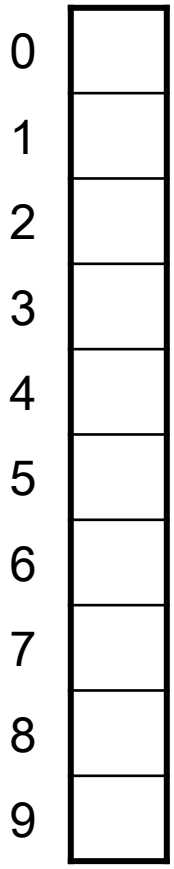
- Create a probing function p
- If a collision occurred, try the positions in the probing sequence:
  - norm(h(K) + p(1)), norm(h(K) + p(2)), …, norm(h(K) + p(i))
  - i is a "probe", i = {1, 2, 3, …, T - 1}
  - norm is a normalisation function, squashing the indexes to the allowed range (%)

Linear probing

- Use the identity function as the probe function: p(i) = i

33 16 27 23 26 79 88

(h(K) + 1)%T, (h(K) + 2) % T, …



- h(K) = K % 10

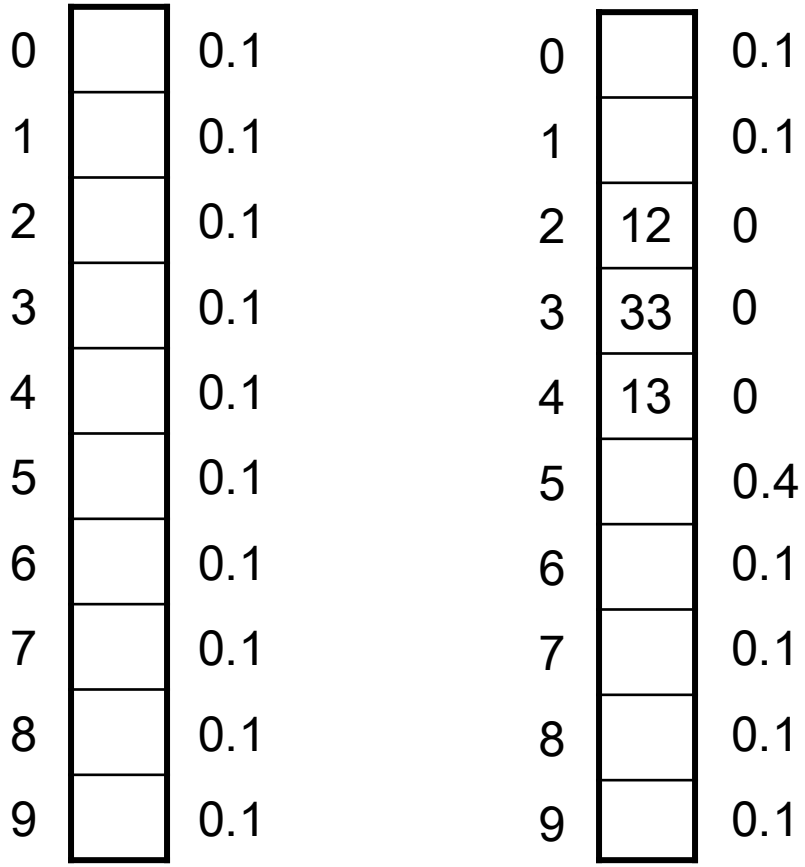
Search the table sequentially until an open position is found

If the end of the table is reached, start from the top of the table

The search stops when the original position is reached

Linear probing

- Problem with linear probing: colliding data will cause clusters
- Clusters will cause more collisions!



Quadratic probing

- Use a more complex probing sequence to avoid clusters:
  - p(i) = $(-1)^{i-1}(\lfloor {}^{i+1}/_{2} \rfloor)^2$ for i = 1,2,…,T – 1
  - What we are essentially trying to do:
    - Instead of going sequentially through i's: 1,2,3,…
    - Go though i's squares with alternating signs: 1, -1, 4, -4, 9, -9, 16, -16…
  - If a collision occurred, try the following:
    - (h(K) + i^2)%T
    - (h(K) - i^2)%T
    - for i = 1,2,…,(T – 1)/2

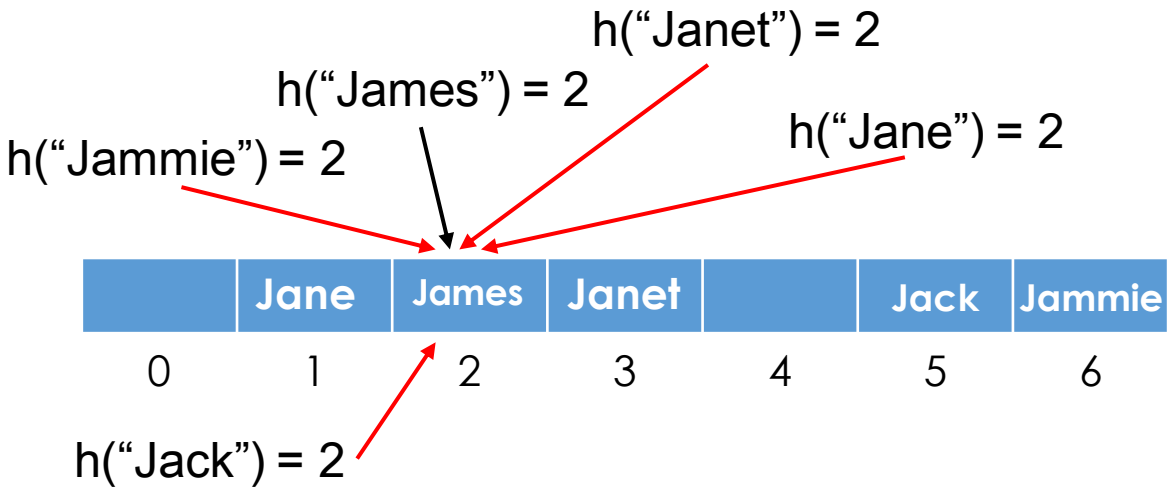
Does this approach avoid clusters?

(2 + 1) % 7 = 3

(2 + 1) % 7 = 3
(2 - 1) % 7 = 1

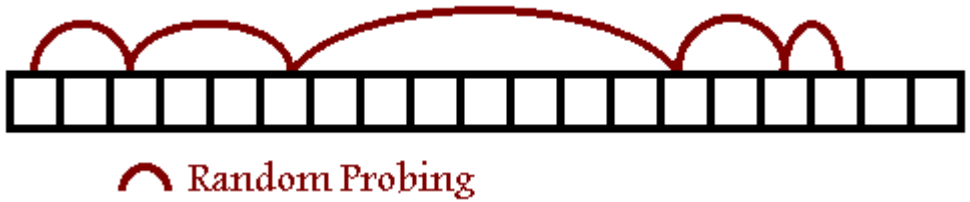
(2 + 1) % 7 = 3
(2 - 1) % 7 = 1
(2 + 4) % 7 = 6

(2 - 4) % 7 = ?
(-2) % 7 = 7 – 2 = 5



Randomized probing

- Quadratic probing will generate similar sequences for different keys, resulting in secondary clusters, or meta-clusters
- What if we generate a random sequence for each K?
  - No clustering!
  - But how are we going to find K in the table if it is placed randomly?
  - Solution: every K must use its own seed for the random number generator
    - Seed predetermines the sequence of numbers that the random number generator is going to produce
    - Use K itself as a seed
    - Or: use a numeric aspect of K, such as size, first byte, last byte, etc.



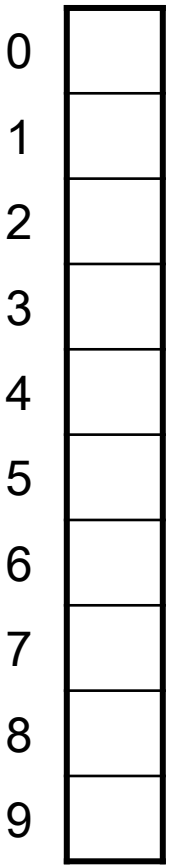
Double hashing

- Another probing technique that avoids clusters is double hashing:
  - Choose a primary hash function, h(K)
  - Choose a secondary hash function, $h_2(K)$
  - If h(K) results in a collision for K, use the following probing sequence:
  - (h(K) + $h_2(K)$)%T, (h(K) + $h_2(K)$ * 2)%T, …, (h(K) + $h_2(K)$ * i)%T

- How do we choose the secondary hash function?
  - To keep hashing efficient, $h_2(K)$ should be simple
  - It should be different from h(K)
  - It should never evaluate to 0 for any K
  - One simple hash: $h_2(K)$ = m – (K % m), where m < T and is a prime

Double hashing

33 16 27 23 46 79 38



- h(K) = K % 10
- h'(K) = 7 – (K % 7)

h'(23) = 7 – (23 % 7) = 5

(h(23) + h'(23))%10 = 3 + 5 = 8

h'(46) = 7 – (46 % 7) = 3

(h(46) + h'(46))%10 = 6 + 3 = 9

h'(79) = 7 – (79 % 7) = 5

(h(79) + h'(79))%10 = 14%10 = 4

h'(38) = 7 – (38 % 7) = 4

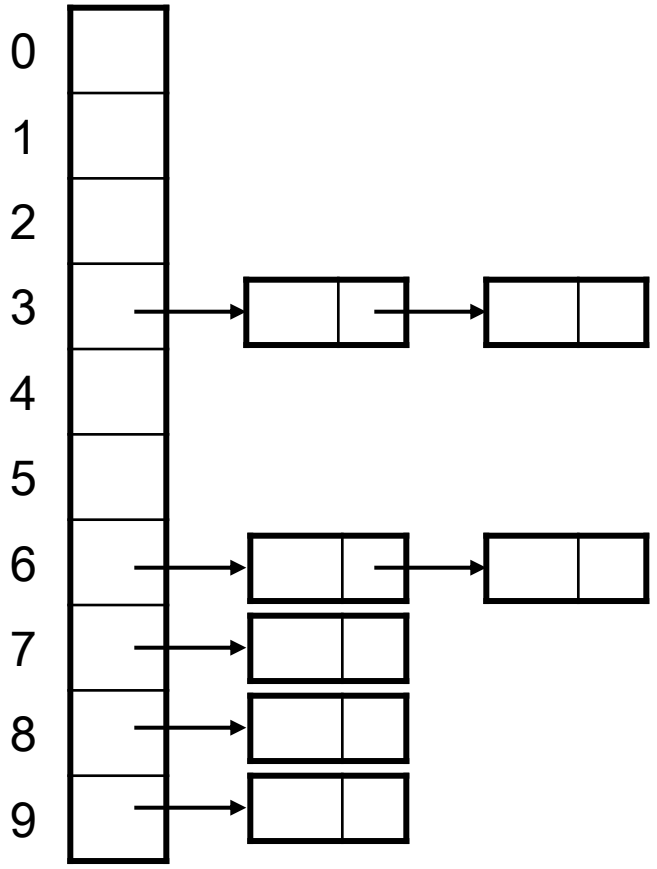
(h(38) + h'(38))%10 = 12%10 = 2

Chaining

- What's the problem with probing?
- If a lot of collisions occur, you loose the benefit of direct access
- Before the item is accessed, you have to find where it ended up as a result of avoiding collisions
- What if, instead of resolving collisions, we stored multiple keys per one hash code?
- Use a linked list
- Efficient if collisions are few
- Disastrous if collisions are many

h(K) = K % 10

33 16 27 23 26 79 88

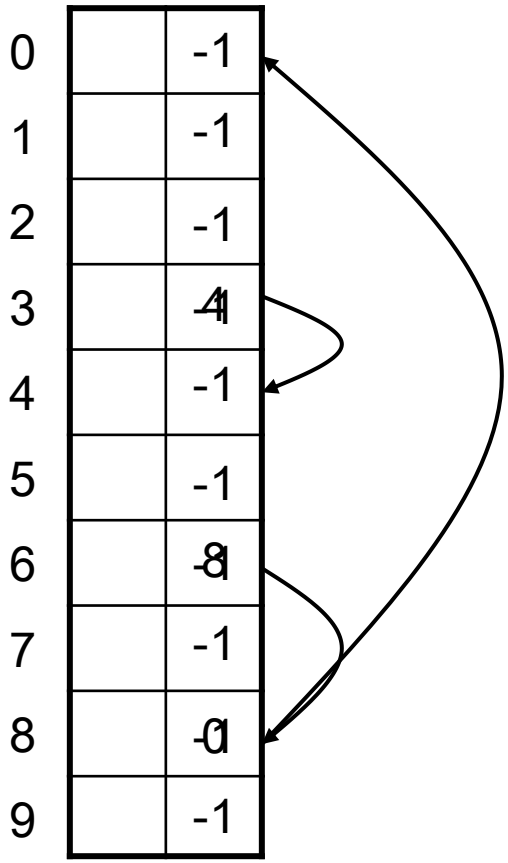


Coalesced hashing

- Trade-off solution: combine probing with chaining
- Use linear (or any other) probing to find the next available position
- Store the index of that position next to the position where the collision occurred
- Now we can go to the next key without re-calculating the hashing sequence
- If many collisions occur, this is still not very efficient
- Linear probing is prone to clusters, clusters will yield longer chains…

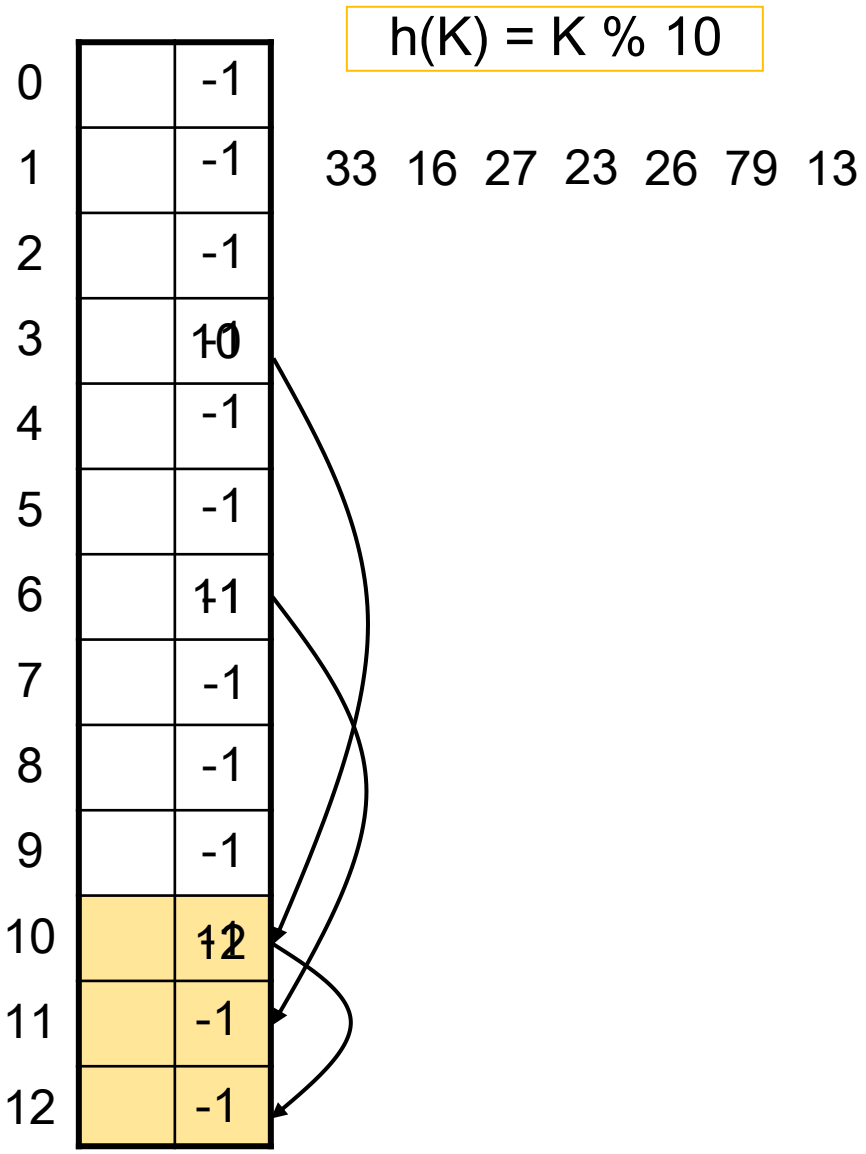
h(K) = K % 10

33 16 27 23 26 79 88



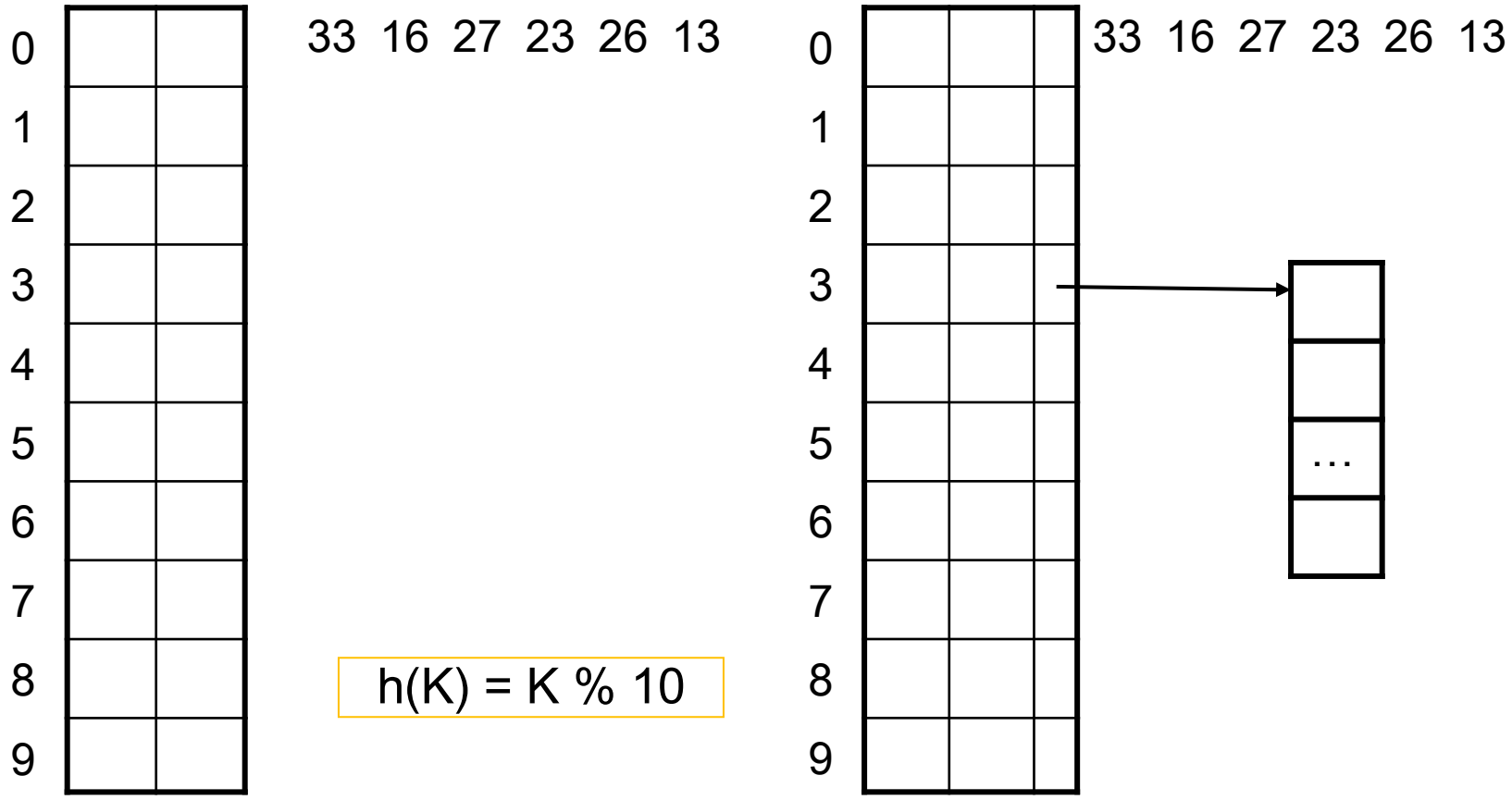
Coalesced hashing with a cellar

- Reserve part of the table for collision (overflow) data
- This subset is called "the cellar"
- All overflow goes into the cellar until the cellar is full
- If cellar is full, resort to probing again
- An appropriately-sized cellar can be efficient, and will not cause clustering!



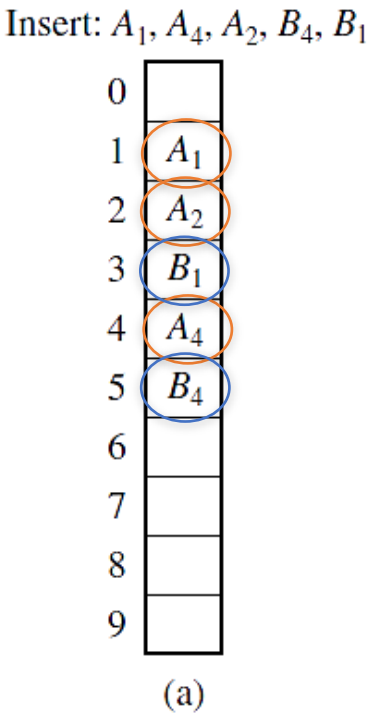
Bucket addressing

- Similar to chaining: store multiple keys in one row
- A row is a "bucket" that has enough space for multiple keys
- If a bucket overflows: use probing, or store a pointer to the overflow location



Deletion from a hash table

- If hashing is perfect: just calculate the hash and delete the corresponding item!
- If chaining is used: find the key to be deleted in the chain, remove, reconnect the chain
  - Similar to linked list deletion
- What if simple probing was used?



Look for B4

Not in the table!

Look for B1

Not in the table!

Two items are "lost"

Label deleted keys as deleted, but do not remove

Wasteful!

After a certain number of deletions, reorganise (rehash) the table

Rehashing, Cuckoo Hashing, Cichelli's Algorithm

Rehashing

- What if you run out of space in the hash table, or the table is so full that finding an empty slot is very hard?
- Rehashing: create a new larger hash table, and copy the elements over
  - Same or different hash function with the new T can be used
  - Double the size, or, even better, choose new T = large prime closest to old T * 2

h(K) = K % 7

Original Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

0	6
1	15
2	23
3	24
4	
5	
6	13

After Rehashing

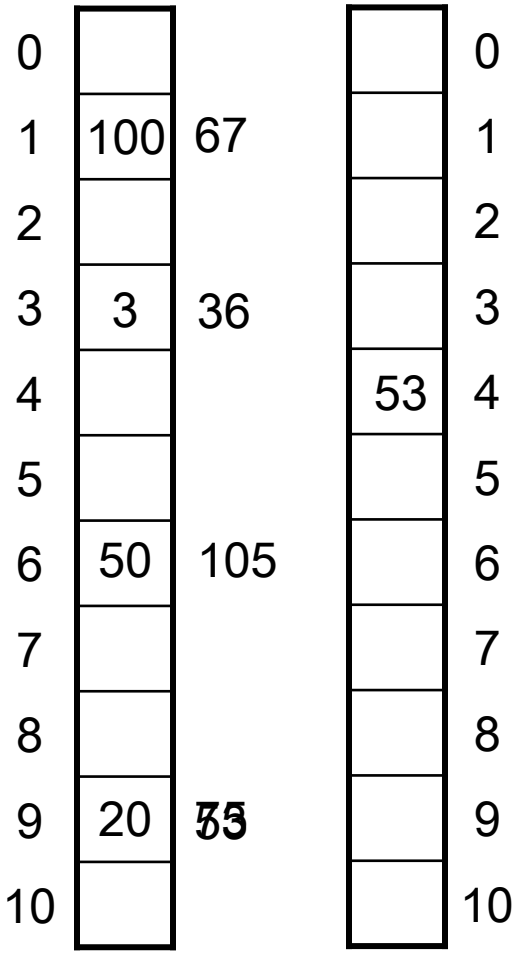
0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

h(K) = K % 17

Cuckoo hashing

- Collision resolution algorithm invented in 2001
- Guaranteed O(1) look-up time for all entries
- Two hash tables are used, each associated with its own hash function
- Each K can be either in one or in other table

k	h(k)	h'(k)
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3



h(K) = K % 11

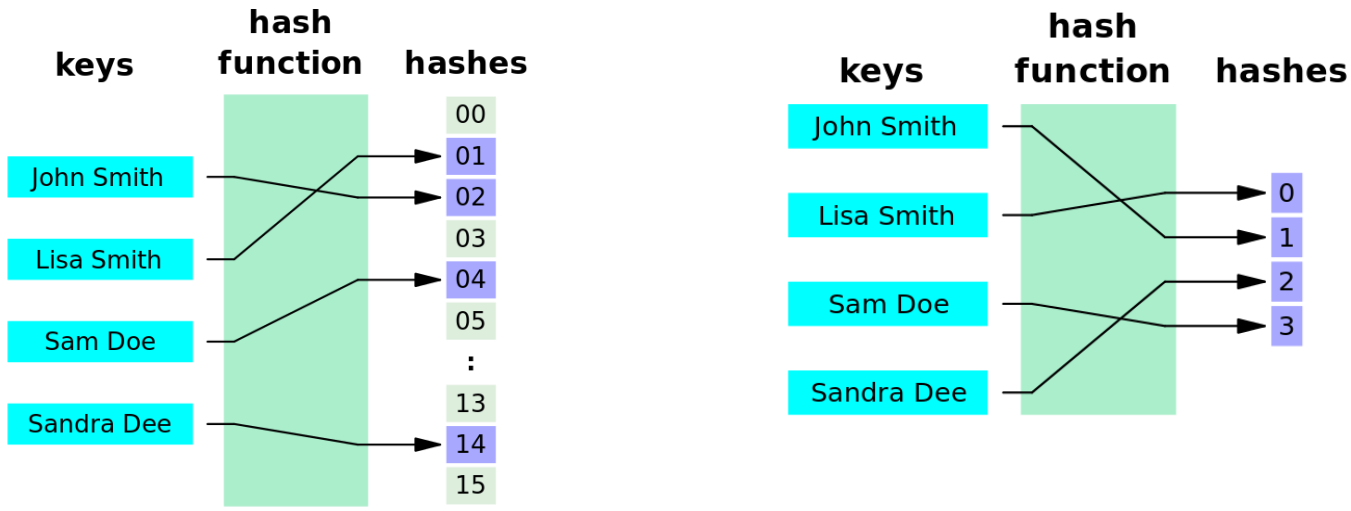
h'(K) = (K / 11) % 11

Problem: cuckoo hashing may produce an infinite loop

Add max iteration number to the algorithm; in case of deadlock, **rehash** both tables

Perfect hash functions

- A perfect hash function is a hash function that assigns a unique hash code to every key (no collisions)
- A minimal perfect hash function is a perfect hash function that uses every slot in the hash table and wastes no space



- Why are hash functions not perfect/minimal?
  - Because the input space is unlimited – can't prepare for everything!
  - What if we knew exactly what data and how much data needs to be stored?
  - Turns out you can design a perfect hash function if data (input space) is known beforehand!

Cichelli's Algorithm

- An algorithm that constructs a perfect minimal hash function for a set of strings
- Suppose we want to store different cities in a hash table:

  - We are re-inventing google maps: need to store a lot of information about each city, and be able to access it quick
  - Every city has a unique name: city names can be the keys
  - The number of cities is predetermined: reserve just enough space (and no extra!) to store them all
- Cichelli's hash function:

```
h(S) = [length(S) + firstLetter(S) + lastLetter(S)] % T
```

  - length(S) is the length of the string
  - firstLetter(S) and lastLetter(S) return the values assigned to the first and last letter of the string
  - Task: Assign values to all first/last characters to create a perfect hash

Cichelli's Algorithm

1. Count the frequencies of first/last characters

Pretoria	P = 1, A = 1
Johannesburg	J = 1, G = 1
Durban	D = 1, N = 1
Cape Town	C = 1, N = 2

P = 1
A = 1
J = 1
G = 1
D = 1
N = 2
C = 1

2. Sort the strings based on cumulative frequency (freq(first) + freq(last))

Durban	3
Cape Town	3
Pretoria	2
Johannesburg	2

Why sort:
Same letters may cause clashes, thus place the words that share letters first

Cichelli's Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
        - use the hash
        - assignHash(wordlist) // recursive step
      - else
        - Go to (II) and try other values for first(w), last(w)

Durban	3
Cape Town	3
Pretoria	2
Johannesburg	2

D = 0

N = 0

0	
1	
2	**Durban**
3	

```
h(Durban) = [6 + 0 + 0] % 4 = 2
```

Cichelli's Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
        - use the hash
        - assignHash(wordlist) // recursive step
      - else
        - Go to (II) and try other values for first(w), last(w)

Cape Town	3
Pretoria	2
Johannesburg	2

D = 0
N = 0
C = 0

0	
1	**Cape Town**
2	**Durban**
3	

```
h(Cape Town) = [9 + 0 + 0] % 4 = 1
```

Cichelli's Algorithm

3. assignHash(wordlist)
   I. Remove first word on the list
   II. Choose values for first(w), last(w)
   III. Compute hash(w) = [length(w) + first(w) + last(w)]%T
      - if(hash is not taken)
        - use the hash
        - assignHash(wordlist) // recursive step
      - else
        - Go to (II) and try other values for first(w), last(w)

Pretoria	2
Johannesburg	2

D = 0
N = 0
C = 0
P = 0
A = 0

0	**Pretoria**
1	**Cape Town**
2	**Durban**
3	

```
h(Pretoria) = [8 + 0 + 0] % 4 = 0
```

Cichelli's Algorithm

Johannesburg	2

D = 0
N = 0
C = 0
P = 0
A = 0
J = 0 | J = 1
G = 0 | G = 1 | G = 2 | ?

MAX = 2

0	Pretoria
1	Cape Town
2	Durban
3	Johannesburg

Final:

D = 0
N = 0
C = 0
P = 0
A = 0
J = 1
G = 2

```
h(Johannesburg) = [12 + 0 + 0] % 4 = 0
h(Johannesburg) = [12 + 0 + 1] % 4 = 1
h(Johannesburg) = [12 + 0 + 2] % 4 = 2
```

```
h(Johannesburg) = [12 + 1 + 0] % 4 = 1
h(Johannesburg) = [12 + 1 + 1] % 4 = 2
h(Johannesburg) = [12 + 1 + 2] % 4 = 3
```

Cichelli's Algorithm

If values have been assigned to letters, they should be re-used by all words

3. assignHash(wordlist)
   - Remove first word on the list
   - for(first(w) = 0; first(w) < MAX; first(w)++) // unless first(w) has a value
     - for(last(w) = 0; last(w) < MAX; last(w)++) // unless last(w) has a value
       - Compute hash(w) = [length(w) + first(w) + last(w)]%T
       - if(hash(w) not taken)
         - Put the word in position hash(w)
         - success = assignHash(wordlist) // recursive step
         - if(success) return success;
   - Put word back on the list
   - Return failure // backtracking

MAX is chosen to be a small value; # words / 2

Cichelli's Algorithm

Frequencies:

Anna	4 + 4 = 8
Maria	2 + 4 = 6
Jane	3 + 1 = 4
John	3 + 2 = 5
Julia	3 + 4 = 7
Megan	2 + 2 = 4

A = 4
M = 2
J = 3
E = 1
N = 2

0	
1	
2	
3	
4	
5	

8	Anna
7	Julia
6	Maria
5	John
4	Jane
4	Megan

Assign values to first/last characters!

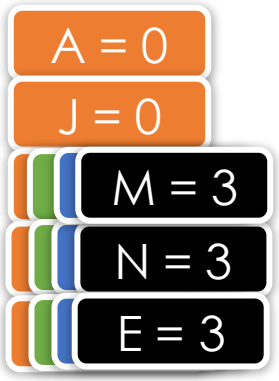
Cichelli's Algorithm

MAX = T / 2 = 6 / 2 = 3

Anna
Julia
Maria
John
Jane
Megan

Anna (4 + 0 + 0) % 6 = 4
Julia (5 + 0 + 0) % 6 = 5
Maria (5 + 0 + 0) % 6 = 5 X
Maria (5 + 1 + 0) % 6 = 0
John (4 + 0 + 0) % 6 = 4 X
John (4 + 0 + 1) % 6 = 5 X
John (4 + 0 + 2) % 6 = 0 X
John (4 + 0 + 3) % 6 = 1
Jane (4 + 0 + 0) % 6 = 4 X
Jane (4 + 0 + 1) % 6 = 5 X
Jane (4 + 0 + 2) % 6 = 0 X
Jane (4 + 0 + 3) % 6 = 1 X
Maria (5 + 2 + 0) % 6 = 1
// John will go to zero, but Jane won't find a place
Maria (5 + 3 + 0) % 6 = 2
// John will go to zero
// Jane will go to one

0	Maria
1	John
2	
3	
4	Anna
5	Julia



Nothing else to try – backtrack!

Cichelli's Algorithm

MAX = T / 2 = 6 / 2 = 3

Anna
Julia
Maria
John
Jane
Megan

Anna (4 + 0 + 0) % 6 = 4
Julia (5 + 0 + 0) % 6 = 5
Maria (5 + 3 + 0) % 6 = 2
John (4 + 0 + 2) % 6 = 0
Jane (4 + 0 + 3) % 6 = 1
Megan (5 + 3 + 2) % 6 = 4 X
John (4 + 0 + 3) % 6 = 1
Jane (4 + 0 + 2) % 6 = 0
Megan (5 + 3 + 3) % 6 = 5 X
Julia (5 + 1 + 0) % 6 = 0

0	John
1	Jane
2	Maria
3	
4	Anna
5	Julia

A = 0
J = 1
M = 3
N = 3
E = 3

Repeat the steps till the perfect hash is created

Complexity: exponential

What is the problem with this approach?

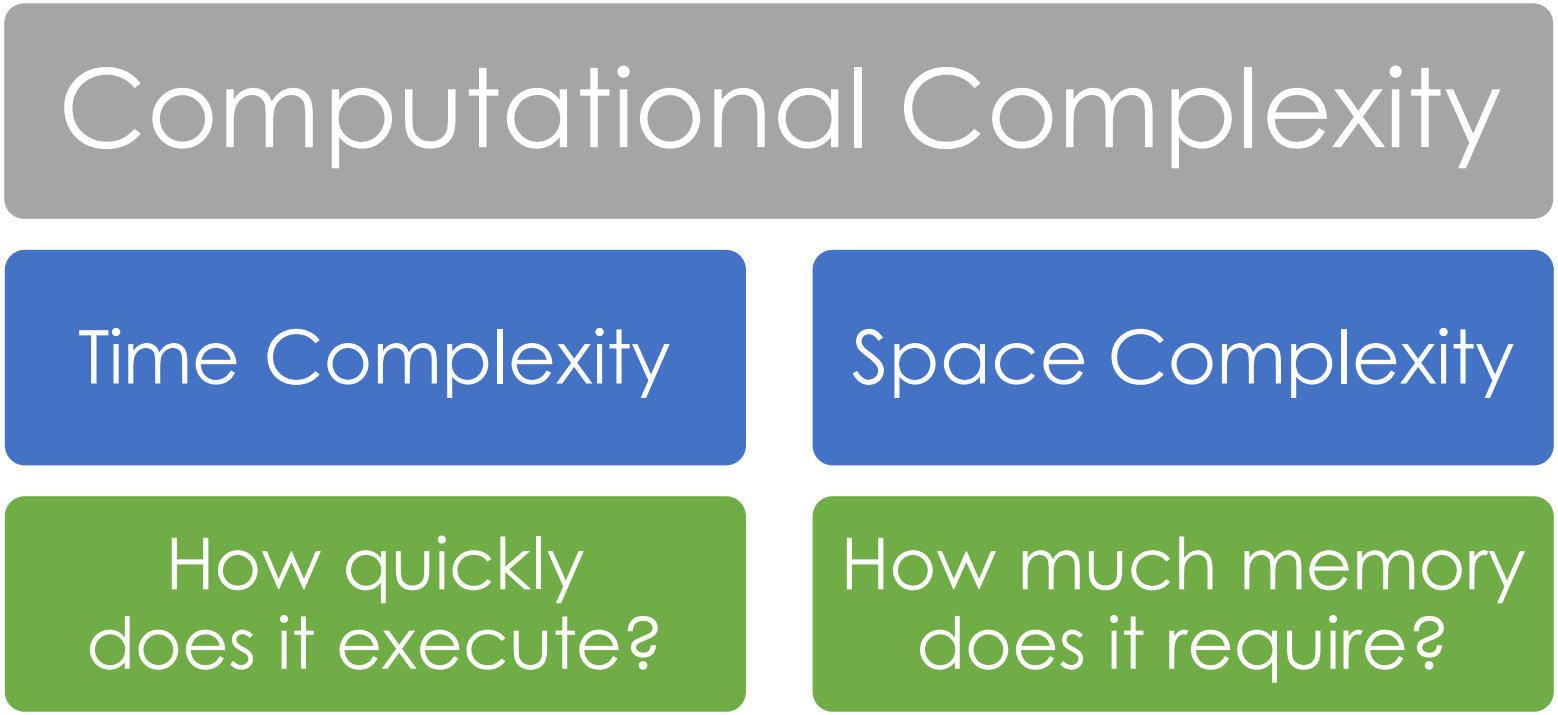
Infeasible for large sets

Backtracking can be very time-consuming

What if you had to store Jane and Jade?

COS 212
Data Compression: Basics & Huffman Coding

Computational Complexity



- Time complexity reduction
  - Through algorithm design and algorithm optimisation
- Space complexity reduction
  - Through data structure design and data compression

Data Compression

- Suppose you want to ship pillows, and have to pay per box

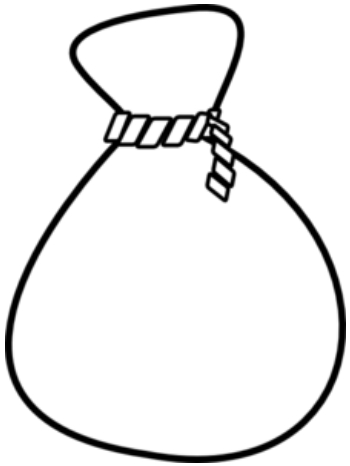
- The more pillows you stuff into a single box, the better
- If you squeeze all the air out, more pillows will fit into a box
- Once out of the box, the pillows can be fluffed up again
- Data compression
  - "Squeezes the air out of pillows"
  - Getting rid of irrelevant parts of data that can be restored easily

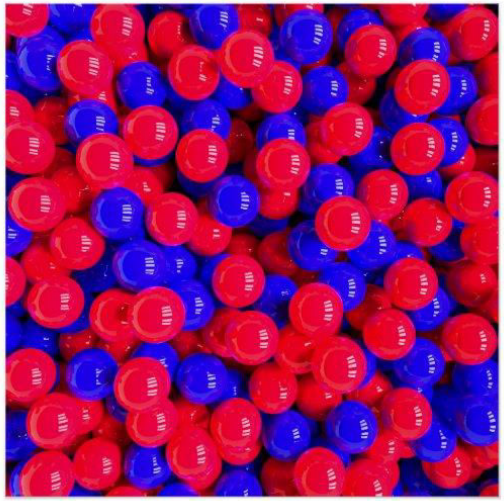
Information

- What is information?
  - The meaning associated with data
  - Yes, No, Yes, No, No, Yes
  - 1 0 1 0 0 1
  - Both sequences contain the same amount of information
  - Compression maximises the amount of information per bit
- How many bits do we need to store information?
  - Use as few as possible!
  - For {Yes, No}, use 1 bit, that can be either 1 (Yes) or 0 (No)
  - What if we had four symbols in a set?
    - {sunny, rainy, cloudy, foggy} = {0, 1, 2, 3}
    - If we use binary to represent N = 4, we can use lg N = 2 bits instead
  - What if we want to encode all 26 letters in the alphabet?
    - How many bits must be reserved *per character*?
    - If we use binary to represent N = 26, we can use lg N = 4.7 = 5 bits
    - But can we do better?

Entropy

- Entropy is a measure
  - Higher entropy denotes more unpredictability, surprise, or chaos
  - Also considered to be a measure of information
- Suppose you have a bag of red and blue marbles





- You take a random marble out of the bag
- If the bag holds an equal number of red and blue marbles
  - How surprised would you be by its colour?
  - In other words, can you predict what colour it will be?
- If the bag holds only blue marbles?
- If the bag holds mostly red marbles, with 1 or 2 blue ones?

Entropy: Minimal Code Length

- We want to compute minimal average bit code length
  - We can use Shannon's entropy
  - $H(S) = -\sum_i p_i \lg(p_i)$

    $= -p_1 \lg(p_1) - \ldots - p_N \lg(p_N)$
  - S is a set of N independent events (symbols)
  - P is a set of probabilities for the N events
  - p_i is the probability of event i
  - $H(S)$ is the entropy of set S

$-\lg(p_i)$ is the minimum number of bits needed to represent symbol i occurring with a probability of p_i

- Let's look at our first example
  - S = {Yes, No}
  - P = {0.5, 0.5}
  - $H(S) = -(0.5 \times \lg(0.5)) - (0.5 \times \lg(0.5))$
    - $\lg(0.5) = \lg(½) = \lg(2^{-1}) = -1$
    - $H(S) = -(0.5 \times (-1)) - (0.5 \times (-1)) = 0.5 + 0.5 = 1$
  - **The entropy is 1**
  - Amount of information = 1 bit (or, we can use 1 bit to encode S)

Data	Yes	No
Probability	0.5	0.5
Encoding	1	0

Entropy: Minimal Code Length

$$H(S) = -\sum_i p_i \lg(p_i)$$
$$= -p_1 \lg(p_1) - \ldots - p_N \lg(p_N)$$

- Let's look at our second example
  - S = {sunny, cloudy, rainy, foggy}
  - P = {0.5, 0.25, 0.125, 0.125}
  - $H(S) = -(0.5 \times \lg(0.5)) - (0.25 \times \lg(0.25))$
    $-(0.125 \times \lg(0.125)) - (0.125 \times \lg(0.125))$
    - $\lg(0.5) = \lg(1/2) = \lg(2^{-1}) = -1$
    - $\lg(0.25) = \lg(1/4) = \lg(2^{-2}) = -2$
    - $\lg(0.125) = \lg(1/8) = \lg(2^{-3}) = -3$
    - $H(S) = -(0.5 \times (-1)) - (0.25 \times (-2)) - (0.125 \times (-3)) - (0.125 \times (-3))$
      $= 0.5 + 0.5 + 0.375 + 0.375$
      $= 1.75$
  - Information content = 1.75 bits per symbol (on average)
  - But bits are indivisible!
  - We can round up to 2 bits
  - But this isn't optimal…
  - Can we get closer to 1.75 bits per symbol with another encoding?

Data	sunny	cloudy	rainy	foggy
Probability	0.5	0.25	0.125	0.125
Encoding	00	01	10	11

Entropy Compression

- Calculate the minimum average bit code length
- Need an encoding that gets as close to minimum as possible
- Not every encoding is good, and some will not even work!
- Four properties of an optimal encoding
  1. Each code must represent exactly one symbol

Yes	No
0	0

Yes	No
0	1

Entropy Compression

- Four properties of an optimal encoding
  1. Each code must represent exactly one symbol
  2. Prefix property: No code should be a prefix of another code

sunny	cloudy	rainy	foggy
0	1	01	10

The code for sunny is the prefix of the code for rainy

The code for cloudy is the prefix of the code for foggy

- Consider the bit sequence 0 1 1 0 0 1 0
- We can interpret it as sunny, cloudy, cloudy, sunny, …
- Or we can interpret it as rainy, foggy, rainy, …

sunny	cloudy	rainy	foggy
0	10	110	111

No code is the prefix of another code

- Again consider the bit sequence 0 1 1 0 0 1 0
- We can **only** interpret it as sunny, rainy, sunny, cloudy
- Therefore, the prefix property ensures unambiguous decoding

Entropy Compression

- Four properties of an optimal encoding
  1. Each code must represent exactly one symbol
  2. Prefix property: No code should be a prefix of another code
  3. For any two codes A and B, length(A) $\leq$ length(B) only if P(A) $\geq$ P(B)
     In other words, codes that occur more often should be shorter

sunny	cloudy	rainy	foggy
0.5	0.25	0.125	0.125
111	110	10	0

- Here the longest codes will be used 75% of the time, wasting space

sunny	cloudy	rainy	foggy
0.5	0.25	0.125	0.125
0	10	110	111

- Here the longest codes are used least, the shortest codes are used most

Entropy Compression

- Four properties of an optimal encoding
  1. Each code must represent exactly one symbol
  2. Prefix property: No code should be a prefix of another code
  3. For any two codes A and B, length(A) ≤ length(B) only if P(A) ≥ P(B)
  4. Every short code must be used as a stand-alone code or a prefix

sunny	cloudy	rainy	foggy
0.5	0.25	0.125	0.125
0	10	1110	1111

110 is not used as a stand-alone code or prefix

Unnecessary longer codes have been created

- Short codes (subject to property 2) are 0, 1, 10, 11, 110, 111

sunny	cloudy	rainy	foggy
0.5	0.25	0.125	0.125
0	10	110	111

All short codes used as a stand-alone code or prefix

No unnecessary longer codes have been created

- Short codes (subject to property 2) are 0, 1, 10, 11

Huffman Coding

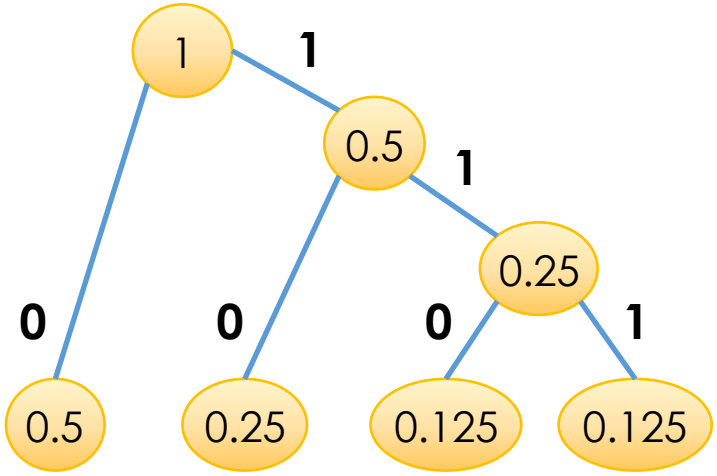
- Four properties of an optimal encoding
  1. Each code must represent exactly one symbol
  2. Prefix property: No code should be a prefix of another code
  3. For any two codes A and B, length(A) $\leq$ length(B) only if P(A) $\geq$ P(B)
  4. All short codes must be used as stand-alone codes or prefixes

- How are we going to construct an optimal encoding?
- We'll use the Huffman encoding algorithm
  1. For each symbol, create a tree with a single root node
  2. Repeat while more that one tree is left
     I. Take trees t_1 and t_2 with the lowest probabilities $p(t_1)$ and $p(t_2)$
     II. Make t_1 and t_2 the left and right children of a new tree t
     III. Set probability of the new tree $p(t) = p(t_1) + p(t_2)$
  3. Associate 0 with all left branches, and 1 with all right branches
  4. For each symbol s in the tree
     I. Traverse from the root of the tree to the leaf node for s
     II. Concatenate 0 and 1 values along traversal path into code for s

Huffman Coding

Can the while loop be implemented recursively?

1. For each symbol, create a tree with a single root node
2. Repeat while more that one tree is left
   I. Take trees t_1 and t_2 with the lowest probabilities $p(t_1)$ and $p(t_2)$
   II. Make t_1 and t_2 the left and right children of new tree t
   III. Set probability of new tree $p(t) = p(t_1) + p(t_2)$
3. Associate 0 with all left branches, and 1 with all right branches
4. For each symbol s in the tree
   I. Traverse from the root of the tree to the leaf node for s
   II. Concatenate 0 and 1 values along traversal path into code for s



sunny	cloudy	rainy	foggy
0.5	0.25	0.125	0.125
0	**10**	**110**	**111**

$H_{avg} = (0.5 \times 1) + (0.25 \times 2) + (0.125 \times 3) + (0.125 \times 3)$
$= 0.5 + 0.5 + 0.375 + 0.375$
$= 1.75$ (minimal average bit code length)

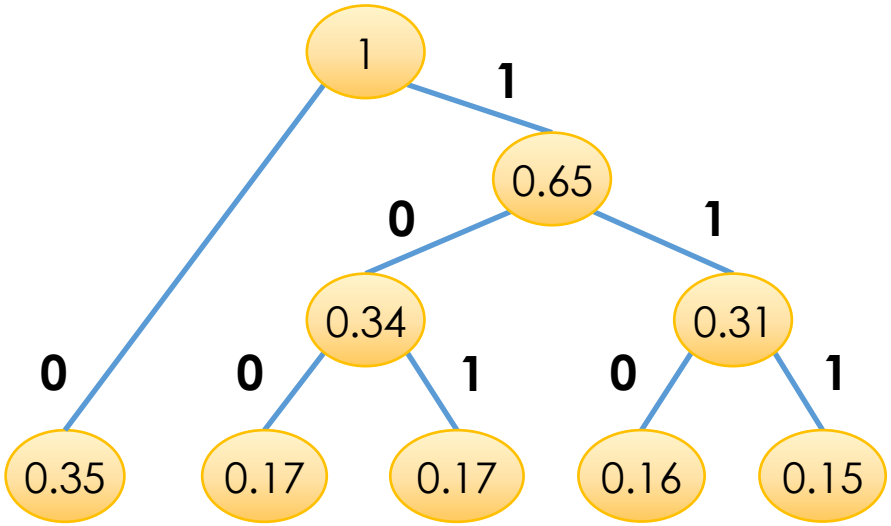
Recall: In each symbol's multiplication, the left factor is the probability & the right factor is the number of bits in the representation

$H_{Huff} = (0.5 \times 1) + (0.25 \times 2) + (0.125 \times 3) + (0.125 \times 3)$
$= 0.5 + 0.5 + 0.375 + 0.375$
$= 1.75$ (actual average bit code length)

So, according to entropy, we can't do better than this encoding!

Huffman Coding

- Let's build an encoding for another example



A	B	C	D	E
0.35	0.17	0.17	0.16	0.15
0	**100**	**101**	**110**	**111**

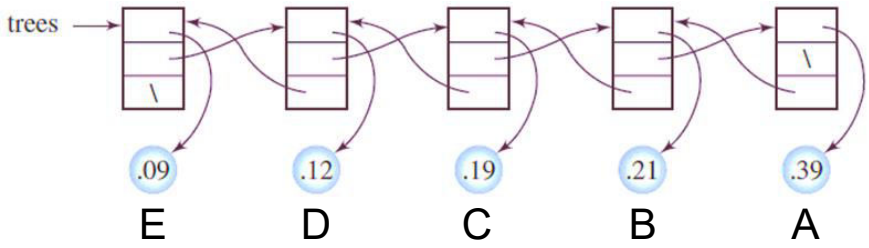
So, according to entropy, Huffman coding falls short of optimal compression performance

$H_{avg} = -(0.35 \times \lg(0.35)) - (0.17 \times \lg(0.17)) - (0.17 \times \lg(0.17)) - (0.16 \times \lg(0.16)) - (0.15 \times \lg(0.15))$
$= 0.5301 + 0.4346 + 0.4346 + 0.4230 + 0.4105$
$= 2.2328$ (minimal average bit code length)

$H_{Huff} = (0.35 \times 1) + (0.17 \times 3) + (0.17 \times 3) + (0.16 \times 3) + (0.15 \times 3)$
$= 0.35 + 0.51 + 0.51 + 0.48 + 0.45$
$= 2.3$ (actual average bit code length)

Huffman Coding: Implementation

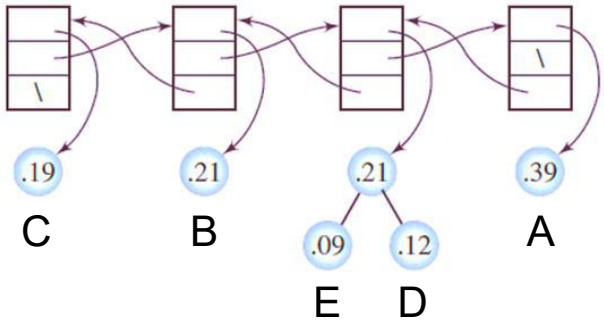
Use a doubly linked list containing trees, sorted using probabilities

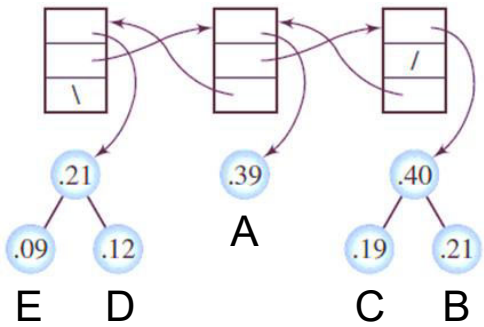


Each iteration, merge trees in 1st & 2nd nodes, and store in 2nd node

Move 2nd node based on merged probability

Delete 1st node

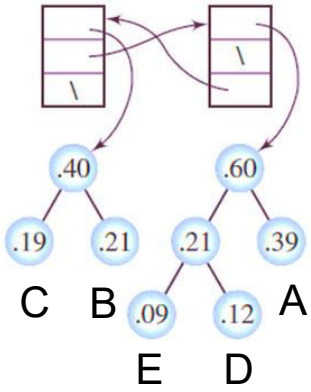


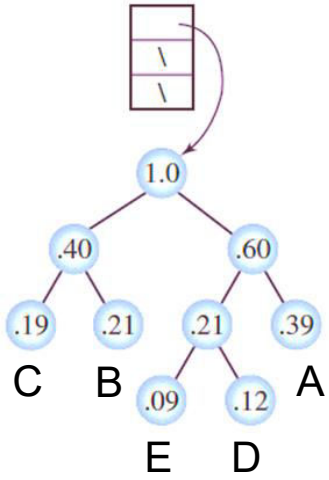


Implementing list as priority queue is a natural choice

Min-heap also works

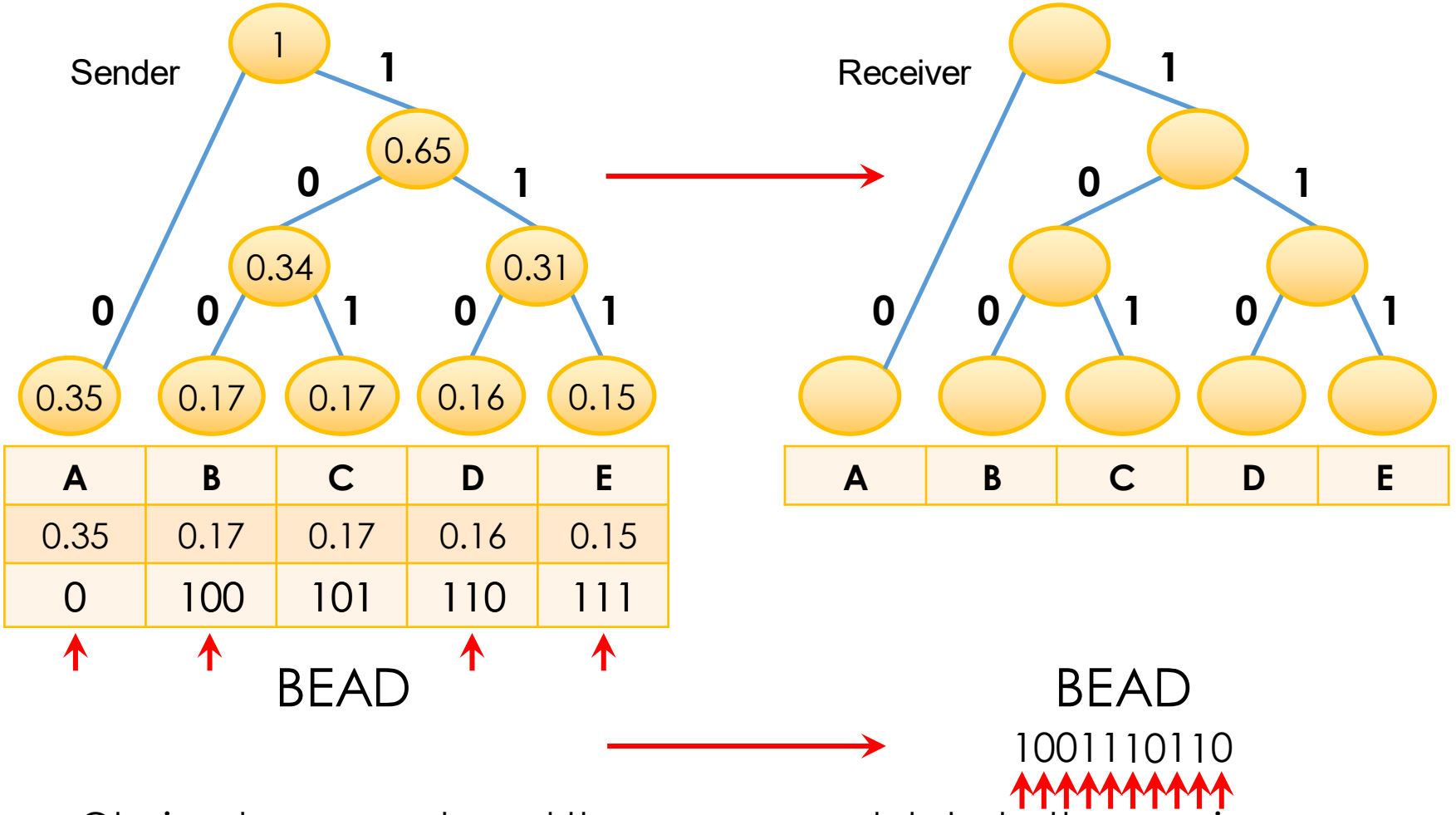
- Restore heap after removing 1st node
- Restore heap after updating 2nd node's probability





A	11
B	01
C	00
D	101
E	100

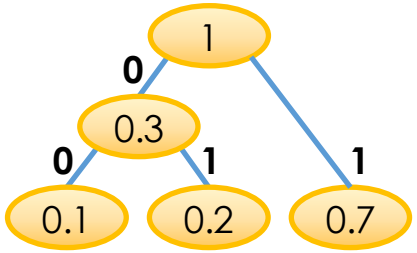
Huffman Coding: Encoding & Decoding

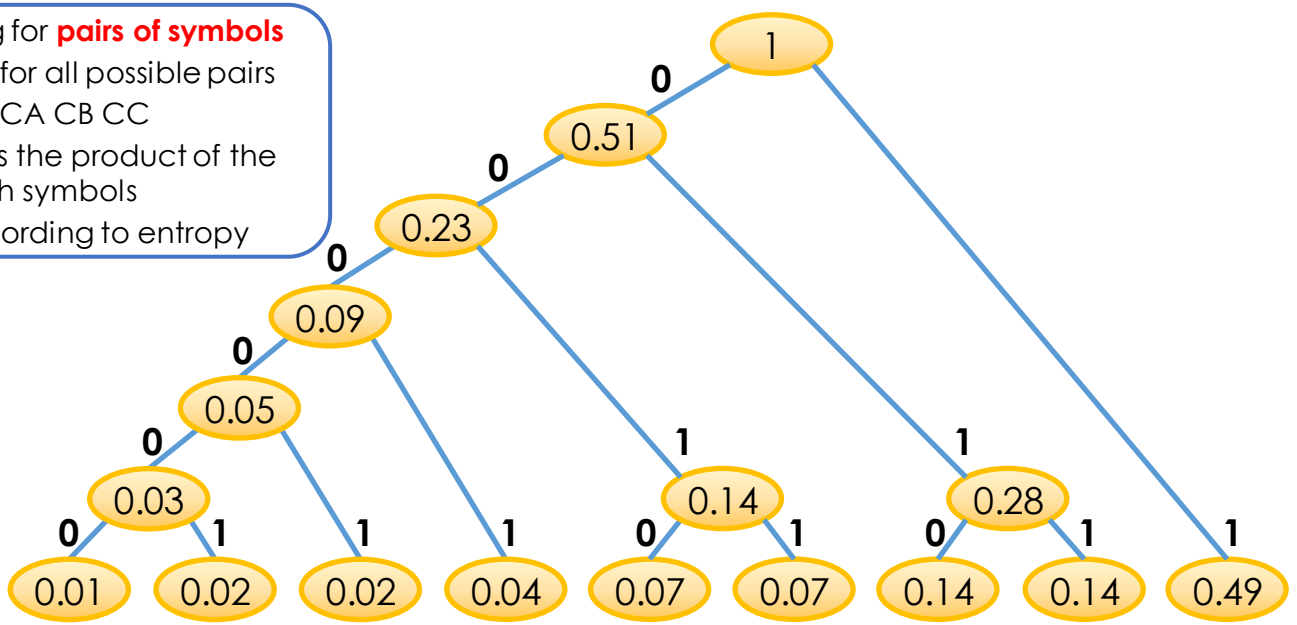


- Obviously, we must send the compressed data to the receiver
- Unfortunately, Huffman trees are not necessarily unique
  - Different trees are built when there are several valid merge options
  - Therefore, we must also send the Huffman code tree to the receiver

Huffman Coding: Improving Efficiency

Now let's create an encoding for **pairs of symbols**
We need to generate codes for all possible pairs
AA AB AC BA BB BC CA CB CC
Probabilities are computed as the product of the probabilities for both symbols
Encodings are ordered according to entropy





A	B	C
0.1	0.2	0.7
00	01	1

AA	AB	BA	BB	AC	CA	BC	CB	CC
0.01	0.02	0.02	0.04	0.07	0.07	0.14	0.14	0.49
000000	000001	00001	0001	0010	0011	010	011	1

Efficiency of 1st approach = 1.1568 ÷ 1.3 = 88.98%

Efficiency of 2nd approach = 2.3136 ÷ 2.33 = 99.30%

H_{avg} = – … × lg(0.04)) – 2 × (0.07 … 0.49))
= 0 … 5043
= 2 …

We've improved the efficiency of the compression at the expense of a larger tree

Why would a larger tree present a potential problem in a practical setting?

H_{Huff} = (0 … 2 × (0.07 × 4) + 2 × (0 …
= 2.33 (actual average bit code length)

COS 212

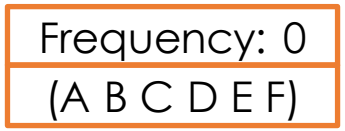
Data Compression: Adaptive Huffman Coding & Run-Length Encoding

Huffman Coding

- Basic idea behind Huffman coding
  - Construct a binary tree based on symbol probabilities
  - Determine the encoding for each symbol by tree traversal
- How do we know the probabilities?
  - Calculate average character frequencies in the language being encoded, and use these frequencies as probabilities
  - Will the same frequencies be optimal for the COS 212 textbook and "Harry Potter and the Philosopher's Stone"?
  - The simple solution
    - Calculate frequencies for the text being encoded, and send the corresponding Huffman codes together with the compressed file
    - But the text may be long and we now we need to run through it twice (once to compute frequencies, and once to compress it)
    - The table of codes can also be cumbersome to work with
  - Adaptive Huffman coding
    - Go through the text once, generate codes as you go
    - Initially inefficient encoding that improves as we "learn" frequencies
    - The receiver reconstructs the same Huffman tree dynamically
    - To decode, the sender and receiver must agree on alphabet order

Adaptive Huffman Coding

- Start with a Huffman tree with only one node
  - This node stores the entire alphabet and is called the alphabet node
  - The frequency for this node must be 0



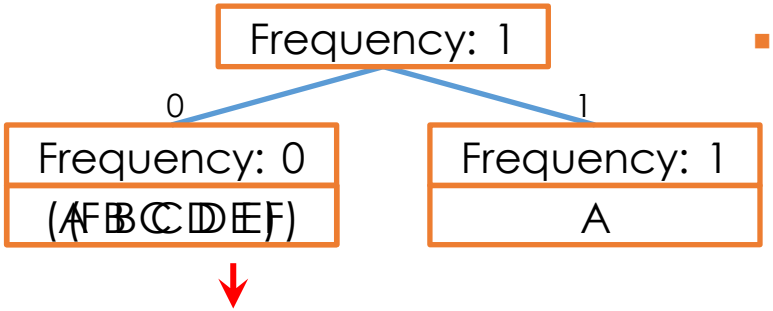
- We encode text letter-by-letter
  - There is no pre-processing to find letter frequencies
  - Move letters out of the alphabet node when they're first encoded
- Two cases for encoding a letter
  1. If the letter is still contained in the alphabet node
  2. If the letter is not contained in the alphabet node

Adaptive Huffman Coding

1. If the letter *i* is still contained in the alphabet node
   - Generate a code that identifies the position of *i* in the alphabet
     - Start with Huffman code of alphabet node (empty on 1st iteration)
     - Add a sequence of 1 bits where the number corresponds to the position of the letter in the alphabet
     - Indicate the end of the code with a single 0 bit

       Alphabet: (A B C D E F)

       Code for A : 10
       Code for D : 11110
     - Append the generated code to the encoded bit sequence
   - Split the letter *i* out of the alphabet node
     - In the alphabet node, move the last letter to overwrite the letter *i*
     - Create a new node for the letter *i*, with a frequency of 1



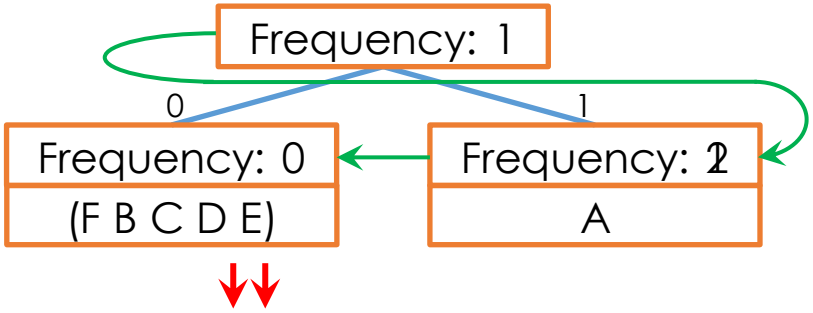
Input text: AAFCCCBDD

Encoding: **10**

   - Create a new parent node
     - Left child is the alphabet node
     - Right child is the new node
     - Cumulative frequency is 1
     - Increment counts in new node's ancestors

Adaptive Huffman Coding

2. If the letter *i* is **not** contained in the alphabet node
   - The letter is already in the Huffman tree
     - Build a Huffman code by traversing from the root to the letter's leaf
     - Append this code to the encoded bit sequence
     - Increment the frequency of the letter's leaf and every ancestor node to more accurately reflect the actual probabilities in the input text
   - Any frequency increment may break Huffman tree structure
     - We then need to repair the tree structure
     - We'll link the nodes using a linked list in breadth-first, right-to-left order
     - The sibling property must be maintained
       - If the frequencies in the list are non-increasing, the tree is a Huffman tree
       - If the sibling property is broken at any point, it must be restored



Input text: AAFCCCBDD

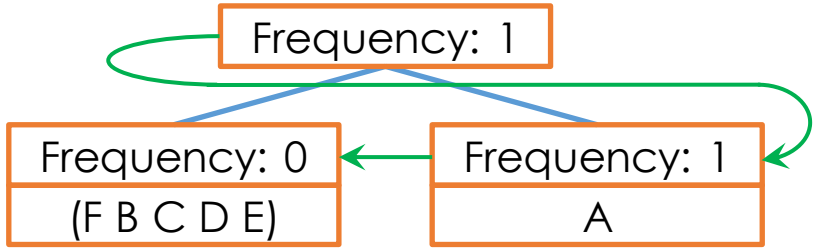
Encoding: 101

Frequencies in list: 1 → 2 → 0

Sibling property broken

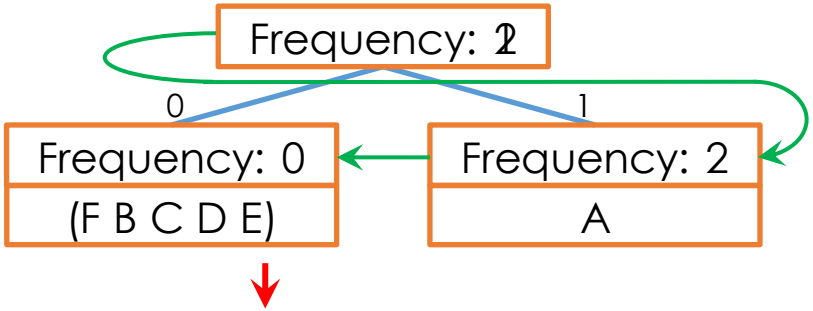
Adaptive Huffman Coding

2. If the letter *i* is **not** contained in the alphabet node
   - Restoring the sibling property
     - Sequences of linked list nodes with the same frequency are blocks
       - In the example, there were two blocks before the frequency increment



Frequencies in list: 1 → 1 → 0

     - Assume that the property is broken by a frequency update for node *i*
       - Swap node *i* with 1st node in its block, unless 1st node is the parent of *i*
       - Continue with frequency increments for all the ancestors of *i*



Note that several sibling property violations may be encountered, requiring a correction each time

Input text: AAFCCCBDD

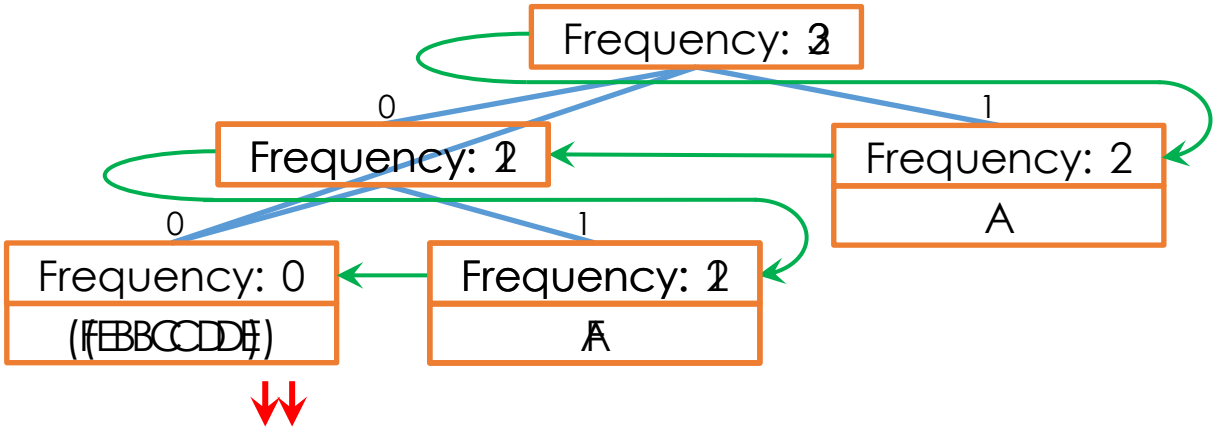
Encoding: 101

Frequencies in list: 1 2 → 2 → 0

Sibling property broken

Adaptive Huffman Coding

- The letter F is still contained in the alphabet node
  - Generate a code for the letter F
    - Huffman code of the alphabet node, sequence of 1 bits to indicate position in alphabet, and a 0 bit to terminate the code
  - Split the letter F out of the alphabet node
    - In the alphabet node, move the last letter to overwrite the letter F
    - Create a new node for the letter F, with a frequency of 1
  - Create a new parent node for the alphabet node
    - Cumulative frequency is 1
    - Increment frequencies in new node's ancestors



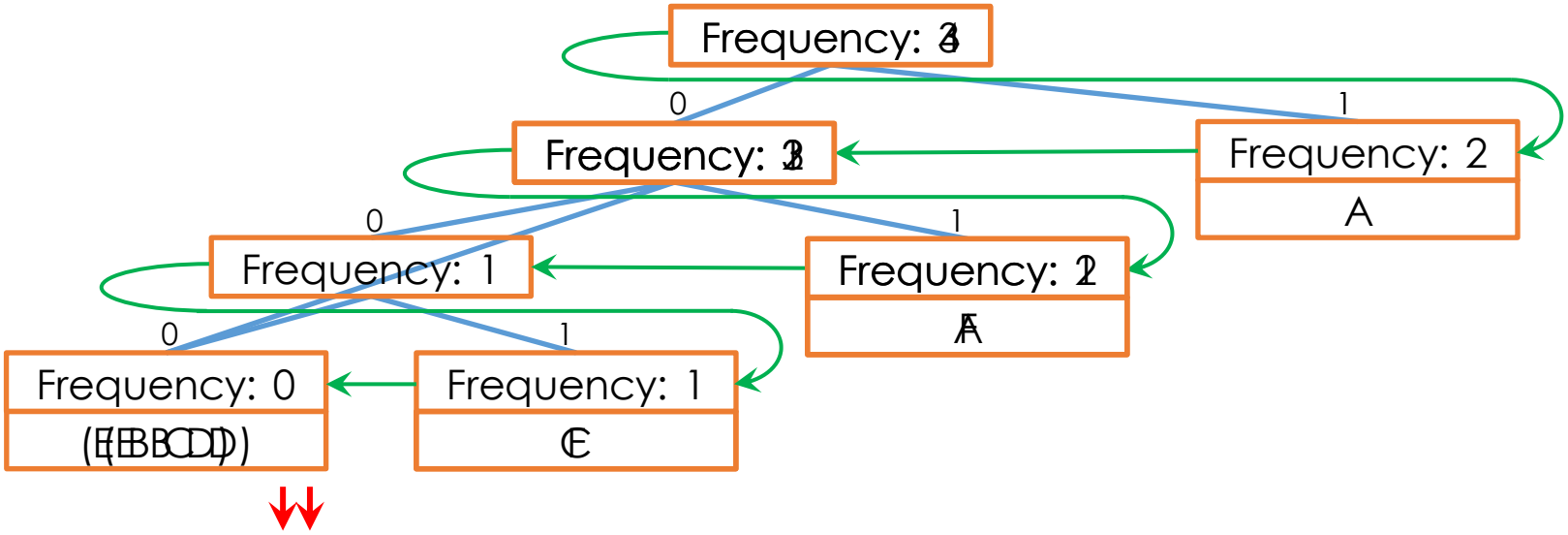
Input text: AAFCCCBDD

Frequencies in list: 3→2→0→1→0

Encoding: 101**010**

Adaptive Huffman Coding

- The letter C is still contained in the alphabet node
  - Generate a code for the letter C
    - Huffman code of the alphabet node, sequence of 1 bits to indicate position in alphabet, and a 0 bit to terminate the code
  - Split the letter C out of the alphabet node
    - In the alphabet node, move the last letter to overwrite the letter C
    - Create a new node for the letter C, with a frequency of 1
  - Create a new parent node for the alphabet node
    - Cumulative frequency is 1
    - Increment frequencies in new node's ancestors



Input text: AAFCCCBDD

Encoding: 101010**001110**

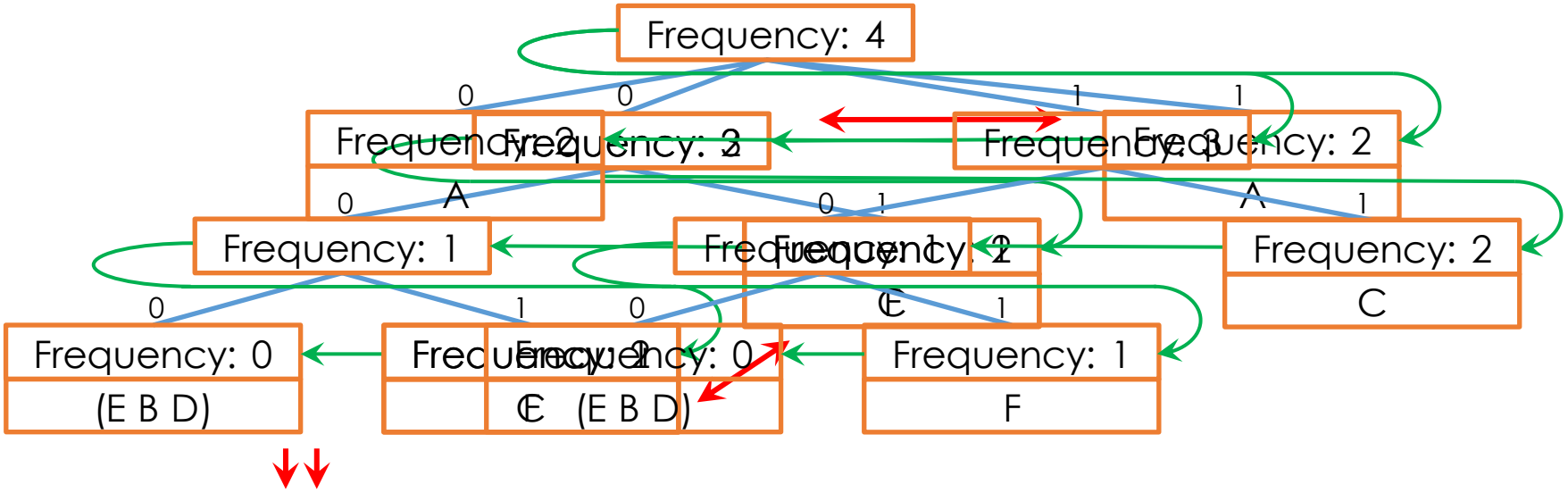
Frequencies in list: 4 → 2 → 1 → 1 → 0 → 1 → 0

Adaptive Huffman Coding

- The letter C is **not** contained in the alphabet node
  - Generate Huffman code for the letter C and increment the frequency for node C
  - While not at the root, check if the frequency update breaks the sibling property
    - If it has, restore the sibling property by swapping the node with the 1st node in its block
    - Perform no swap if the 1st node in the block is the parent of the node
    - Update the frequency of the parent node, and repeat

As an exercise, work through the remaining four inputs according to the procedure we used for this example

Decoding follows a very similar procedure to build the tree from the encoded bits – Try to work out how the algorithm must change



Input text: AAFCCCBDD

Encoding: 101010001110**001**

Frequencies in list: 4→3→2→2→1→1→0

Sibling property broken

Run-Length Encoding

- Relies on the presence of "runs" in the data to be encoded
  - Runs are sequences of exactly the same character

AAAABBCDDDDEE

- Instead of sending or storing AAAA, store 4A
- But, when would you ever see such text in the real world?
  - It's very unlikely that you would
  - Run-length is inefficient for text!
- But, think about images…

Run-Length Encoding

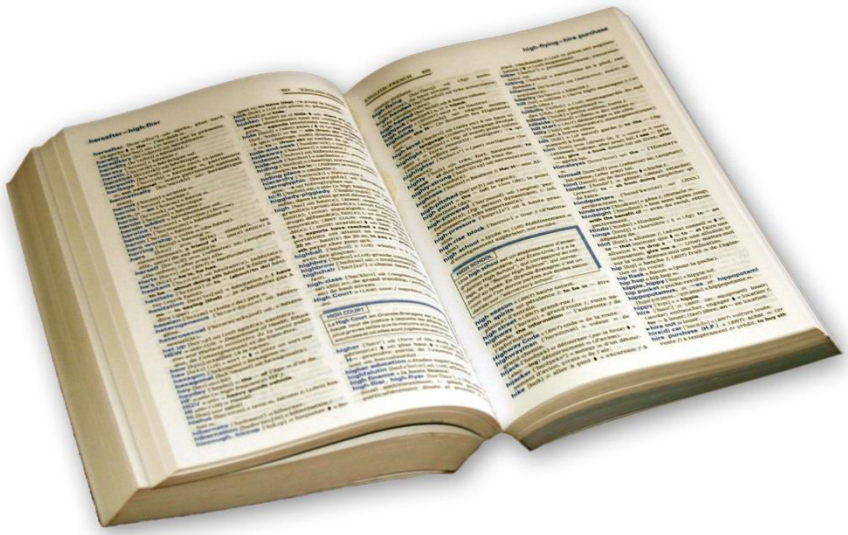
- We iterate through the letters in the input text
  - Encode each run with just two characters

  - 4A2B1C4D2E
  - The 1st, 2nd, 4th & 5th parts are either compressed or remain the same length
  - The one exception is C, where we've actually increased the space used
- Solution to this problem
  - Compress only the runs that are long enough
  - How will we know what is compressed and what isn't?
  - Use a special character (an escape character) to show compressed runs
  - For example, if % is the escape character, the encoding is %4A%2BC%4D%2E
  - But BB and EE is actually shorter than %2B and %2E
  - Solve this by compressing only runs that are 3 or more symbols long
  - For example, %4ABBC%4DEE
- Consider AAABBB versus ABABAB
  - Huffman encoding would compress both, but could run-length encoding?
- In binary there are lots of runs of 0 and 1 bits
  - How would you apply run-length encoding to binary data?

COS 212
String Matching

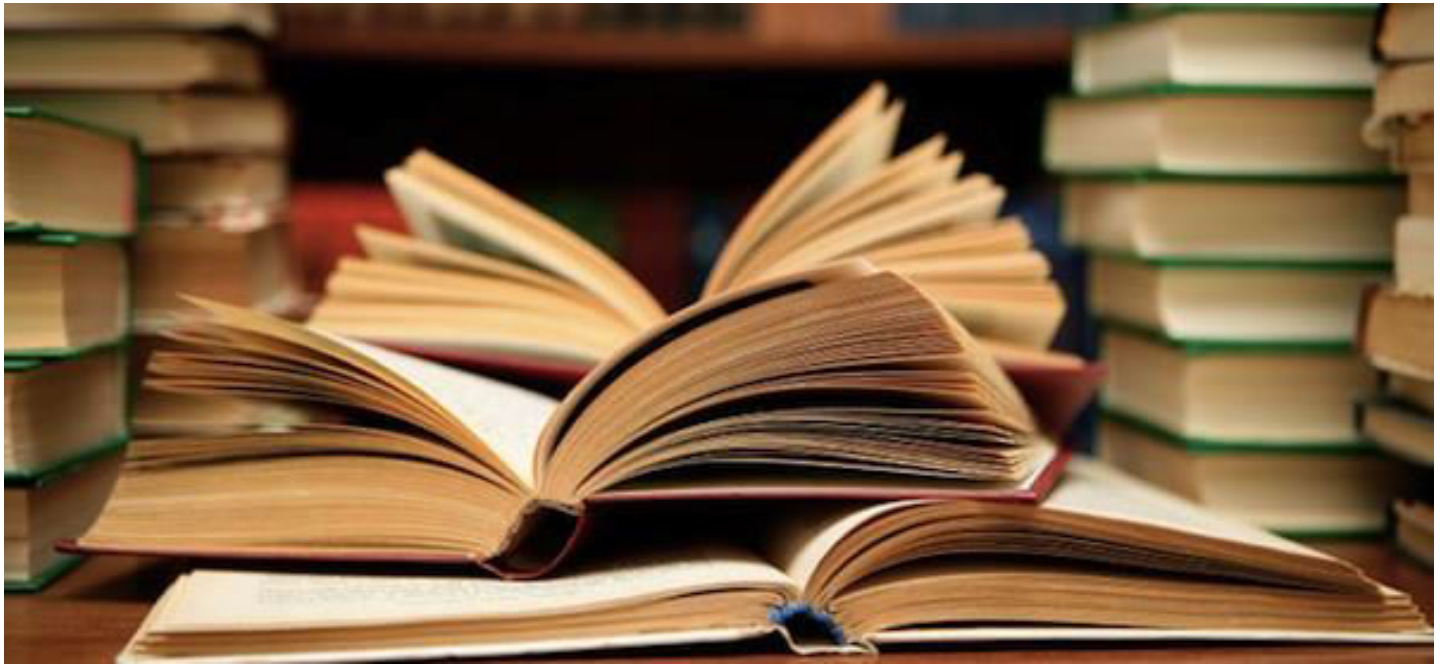
The Problem of Searching

- Searching for a key in a data structure is a difficult task
  - Searching unsorted structures with sequential search: O(n)
  - Searching sorted structures with binary search: O(lg n)
  - Searching hash tables allows direct access: O(1)
- What if you are looking for a word in a piece of text?
- Can text be considered a data structure?



String Matching

- Language
  - The main communication medium of humans
  - It is too complex for simplistic data structures
- String matching
  - Looking for a particular subset of text in a large text file
  - What is the fastest way of automating the process?

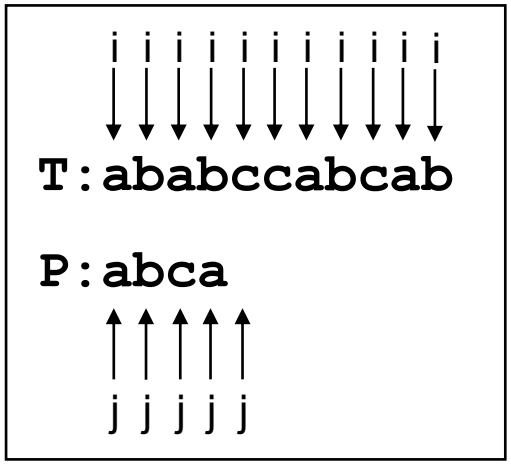


Differences from Textbook

- Take careful note!
  - The algorithm pseudocode in these slides differs from the pseudocode given in the textbook, due to errors
  - When answering questions related to these algorithms, you must use the algorithms outlined in these slides
  - This is especially important for the last approach we discuss (the Knuth-Morris-Pratt algorithm and its `findNext` method)
  - The errors in the Knuth-Morris-Pratt algorithm also mean that the textbook's discussion on the algorithm's time complexity are inaccurate

Brute Force String Matching

- Exact string matching
  - Find pattern P in text T
- Naïve approach
  - Brute force string matching
  - Run through T letter-by-letter
  - At every letter of T, compare P to T letter-by-letter
  - Reset P each time a mismatch is found




```
bruteForceStringMatching(pattern P, text T)
   i = 0;                              // index for T
   while i ≤ |T| - |P|                 // while P fits into T
      j = 0;                           // index for P
      while j < |P| and T_i == P_j
         i++;                          // try to match all characters in P
         j++;
      if j == |P|                      // have we matched the entire P?
         return match at i - |P|;      // success if the end of P is reached
      i = i - j + 1;                   // mismatch, so move right 1 in T
   return no match;                    // fail if too few characters left in T
```

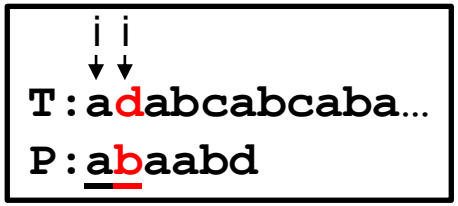
Complexity?

O(|T| |P|)

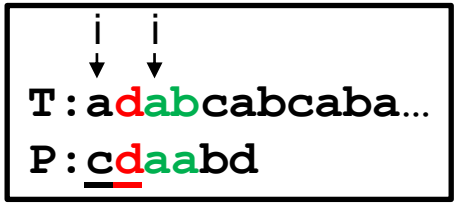
Can we do better?

Improving Brute Force

- Hancart proposed an improvement to basic brute force
  - Start comparisons from the second character of P and T
  - Compare the first character in P and T last
  - Allows optimisation of how i is updated after a mismatch is found
  - Optimisation is based on whether 1st two characters in P match
- If P_0 != P_1 && T_{i+1} != P_1
  - Increment index i by one and start search again

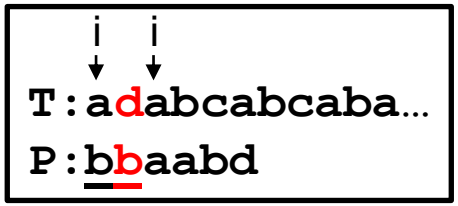


- If P_0 != P_1 && T_{i+1} == P_1
  - Look ahead in T, matching P character by character
  - If we match against all remaining characters in P and T_i also matches P_0
    - Report that a match has been found
  - If a mismatch is found during the look ahead (e.g. between b and a)
    - Because T_{i+1} == P_1 and P_0 != P_1, we already know that T_{i+1} != P_0
    - Increment index i by two and start search again

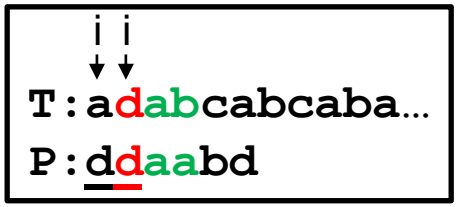


Improving Brute Force

- Hancart proposed an improvement to basic brute force
- If P_0 == P_1 && T_{i+1} != P_1
  - Because T_{i+1} != P_1 and P_0 == P_1, we already know that T_{i+1} != P_0
  - Increment index **i** by two and start search again



- If P_0 == P_1 && T_{i+1} == P_1
  - Look ahead in T, matching P character by character
  - If we match against all remaining characters in P and T_i also matches P_0
    - Report that a match has been found
  - If a mismatch is found during the look ahead (e.g. between b and a)
    - Increment index **i** by one and start search again



Difference from brute force: Sometimes, we can take a step of 2 in the text T instead of 1

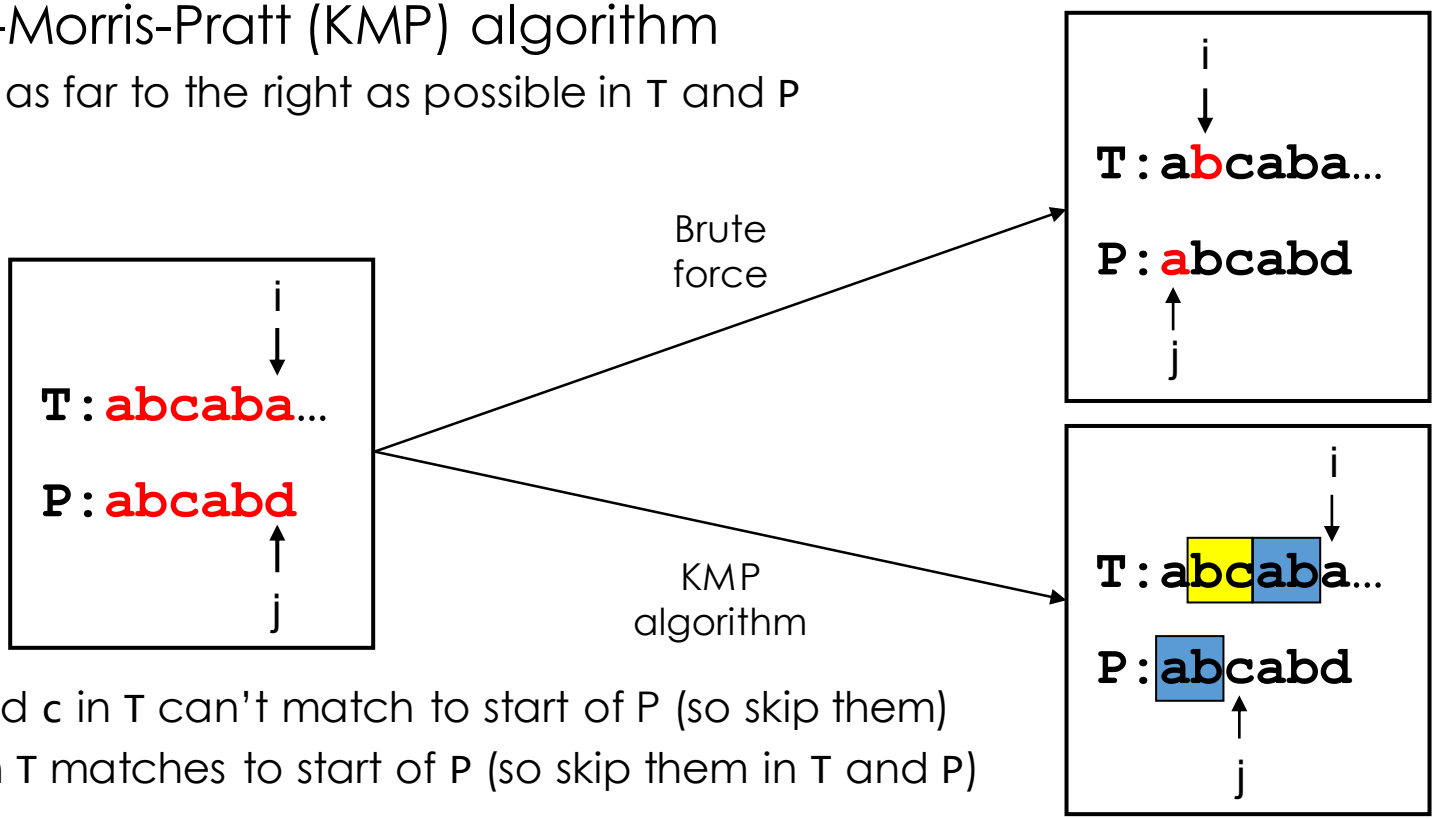
Complexity?

O(|T| |P|) (worst case)

Better on average

Knuth-Morris-Pratt Algorithm

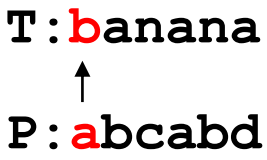
- Consider the following situation
  - The first part of P (i.e. abcab) has been matched in T
  - The last character in P (i.e. a) doesn't match the last character in P (i.e. d)
- Brute force
  - Shifts one position to the right from the start of the partial match in T
- Knuth-Morris-Pratt (KMP) algorithm
  - Shift as far to the right as possible in T and P



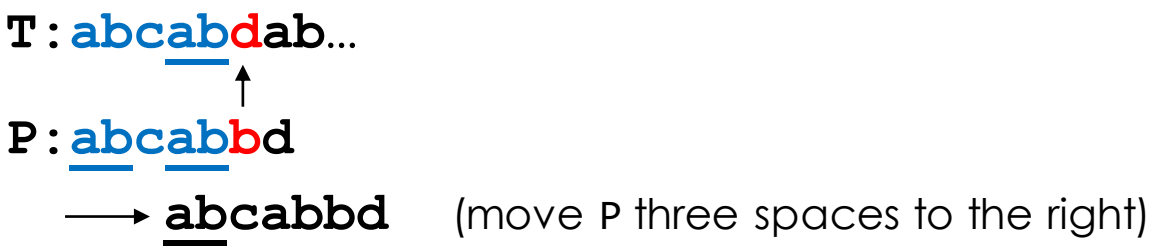
  - b and c in T can't match to start of P (so skip them)
  - ab in T matches to start of P (so skip them in T and P)

Knuth-Morris-Pratt Algorithm

- How do we know how far to skip ahead?



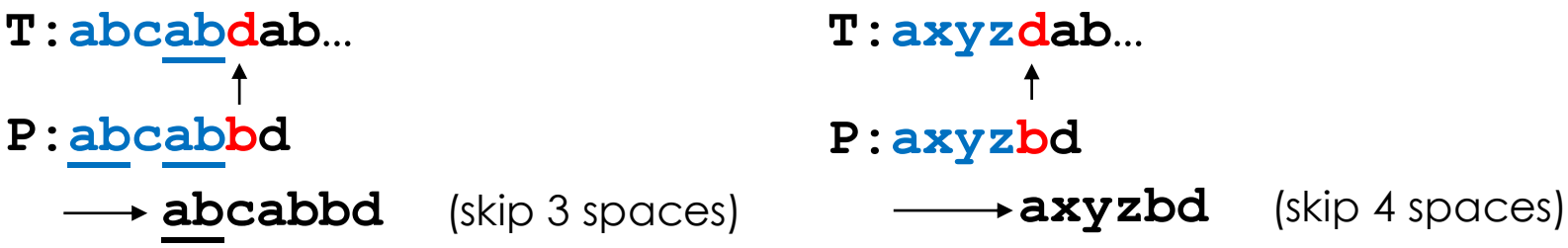
  - Here we haven't found a partial match yet
  - We can only skip ahead one space in T



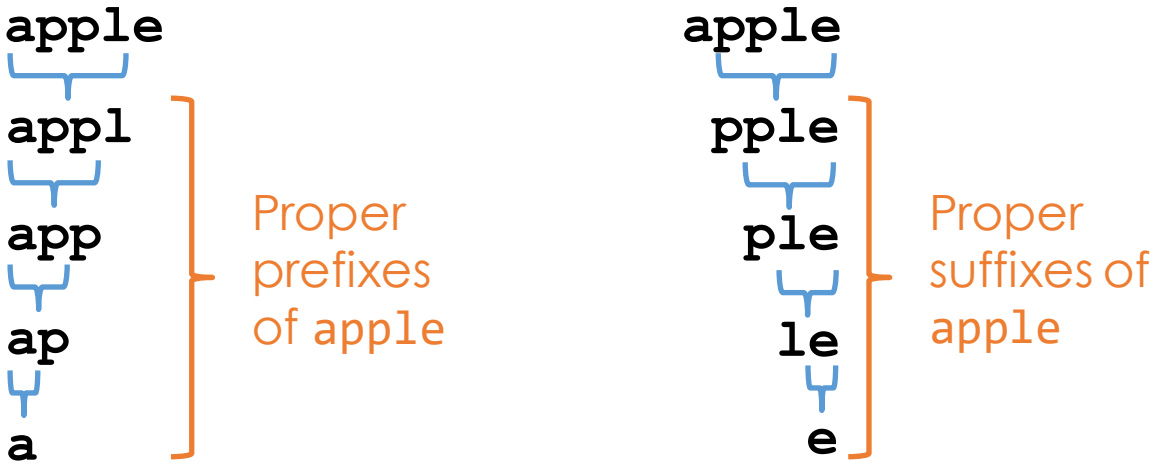
  - Here we have a partial match ending with characters (ab) at start of P
  - Skip ahead, match from after characters (ab) at the end of partial match

  - Here we have a partial match not ending with characters at start of P
  - We can skip over the entire partial match, begin again from start of P

Knuth-Morris-Pratt Algorithm



- To skip the appropriate number of characters
  - We need to know if the start of P matches the end of P
- Proper prefix
  - All the characters in a string, with one or more cut off the end
- Proper suffix
  - All the characters in a string, with one or more cut off the start



Knuth-Morris-Pratt Algorithm

- The Knuth-Morris-Pratt algorithm requires pre-processing of `P`
  - We construct a table with a column for each character in `P`
  - For each column `j` in the table
    - Consider only characters in `P` that are before `j` (substring `P[0]` to `P[j-1]`)
    - Store the length of the longest proper prefix matching a proper suffix

$$\text{next}[j] = \begin{cases} -1 & \text{if } j = 0 \\ \max\{ k: 0 < k < j \text{ and } P[0 \ldots k-1] = P[j-k \ldots j-1] \} & \text{if such a } k \text{ exists} \\ 0 & \text{otherwise} \end{cases}$$

P:	**a**	**a**	**b**	**a**	**a**	**c**	**a**	**a**	**b**	**d**
j:	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
next:	**-1**	**0**	**1**	**0**	**1**	**2**	**0**	**1**	**2**	**3**

j = 1: `0 < k < 1`

j = 2: aa — **prefixes in a:** a — **suffixes in a:** a

j = 3: aab — **prefixes in aa:** a, aa — **suffixes in ab:** b, ab

j = 4: aaba — **prefixes in aab:** a, aa, aab — **suffixes in aba:** a, ba, aba

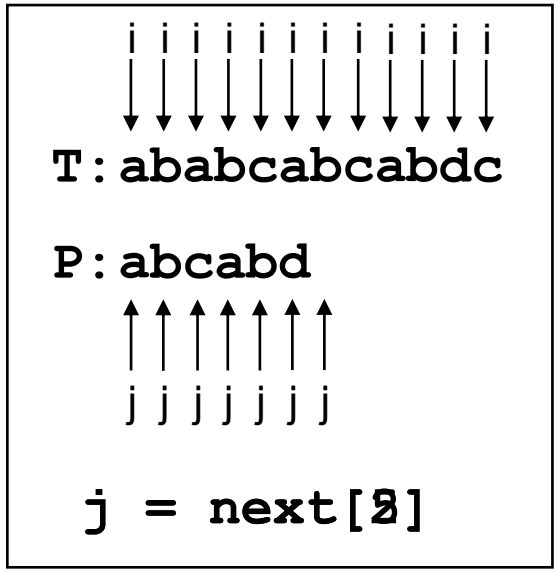
j = 5: aabaa — **prefixes in aaba:** a, aa, aab, aaba — **suffixes in abaa:** a, aa, baa, abaa

Knuth-Morris-Pratt Algorithm

- Knuth-Morris-Pratt algorithm
  - Adapted from the brute force algorithm
- Main difference
  - Partial match table `next[]` is used to move `j` in `P` when mismatch occurs
  - `i` never backtracks (meaning `i` only ever increments by 1)

P:	**a**	**b**	**c**	**a**	**b**	**d**
j:	**0**	**1**	**2**	**3**	**4**	**5**
next[j]:	**-1**	**0**	**0**	**0**	**1**	**2**

```
KnuthMorrisPratt(P, T)
   findNext(P,next);              // populate next[]
   i = j = 0;
   while i - j ≤ |T| - |P|        // P can still fit in T
      while j == -1 or (j < |P| and T_i == P_j)
         i++;
         j++;                     // match in T from i
      if j == |P|                 // successful match
         return match at i - |P|;
      j = next[j];                // skip matched part of P
                                  // i does not backtrack
   return no match;
```



Complexity of `KnuthMorrisPratt(P, T)`?

O(|T|)

Knuth-Morris-Pratt Algorithm

- Populating the `next[]` array
  - Match prefixes and suffixes within `P` itself
- We can adapt Knuth-Morris-Pratt
  - Suffix uses index `i`, prefix uses index `j`
  - After a mismatch, rewind prefix index (`j`) to start matching with a new suffix
  - Store length of matches in `next[]` array

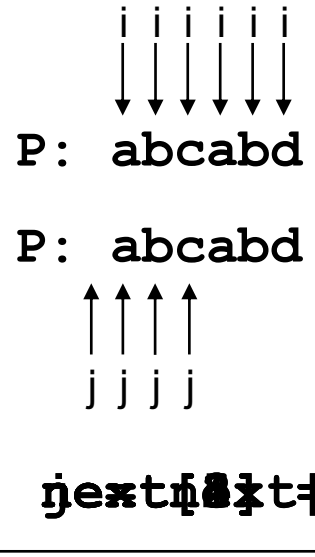
P:	a	b	c	a	b	d
i:	0	1	2	3	4	5
next[i]:	-1	0	0	0	1	2

```
findNext(P, next)
   next[0] = -1;              // next[0] is always -1
   i = 0;                     // suffix counter
   j = -1;                    // prefix counter

   while i < |P| - 1          // stop when next[] populated

      while j == -1 or (i < |P| - 1 and P_i == P_j)
         i++;
         j++;                 // match prefix to suffix
         next[i] = j;         // j is 0 or a match length

      j = next[j];            // rewind j
```



Complexity of `findNext(P, next)`?

O(|P|)

Overall complexity?

O(|T|+|P|)