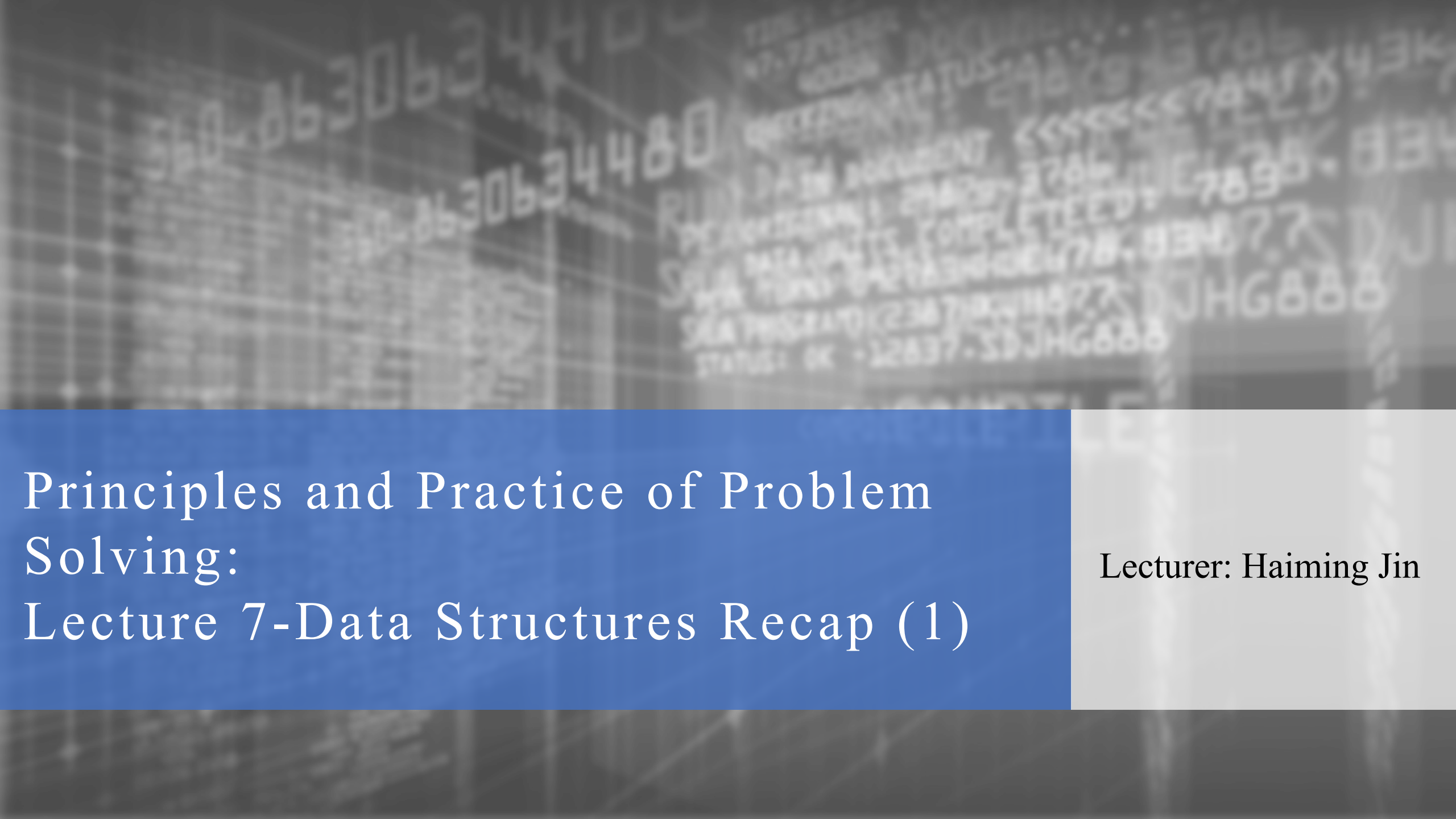


Principles and Practice of Problem Solving: Lecture 7-Data Structures Recap (1)

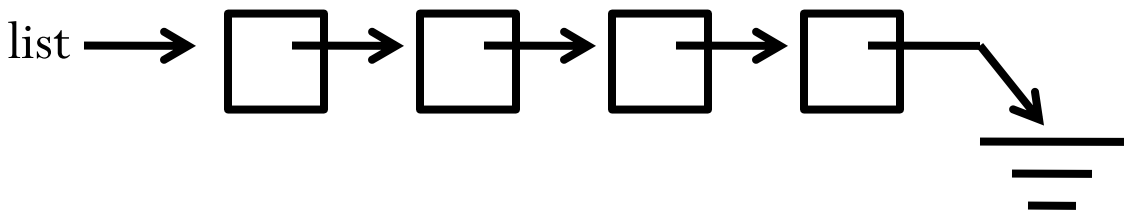
Lecturer: Haiming Jin

Outline

- Linked List
- Stack
- Queue
- Sort
- Linear-Time Selection
- Hashing
- Bloom Filter

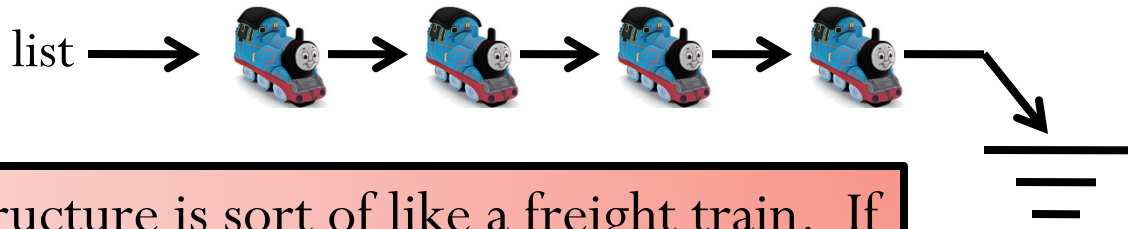
Linked List

- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another, like:



Linked List

- A linked structure is one with a series of zero or more data containers, connected by pointers from one to another, like:



A linked structure is sort of like a freight train. If you need to carry more freight, you get a new boxcar, connect it to the train, and fill it. When you don't need it any more, you can remove that boxcar from the train.

Stack

- A “pile” of objects where new object is put on **top** of the pile and the top object is removed first.
  - LIFO access: last in, first out.

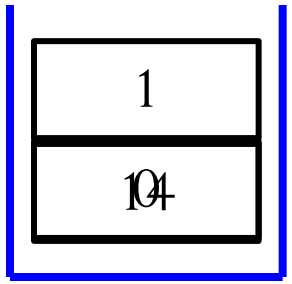
How to Realize Parentheses Matching?

```
( ( a + b ) * c + d - e ) / ( f + g )
0 1 2 3 4 5 6 7 8 9 10  12  14  16  18
```

- Scan expression from left to right.
- When a **left** parenthesis is encountered, push its position to the stack.
- When a **right** parenthesis is encountered, pop the top position from the stack, which is the position of the **matching left** parenthesis.
  - If the stack is empty, the **right** parenthesis is not matched.
- If string is scanned over but the stack is not empty, there are not-matched **left** parentheses.

Parentheses Matching

```
( ( a + b ) * c + d - e ) / ( f + g )
0 1 2 3 4 5 6 7 8 9 10  12  14  16  18
```

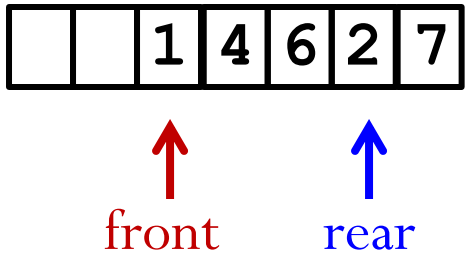


(1, 5) (0, 12) (14, 18)

Queue

- A “line” of items in which the **first** item inserted into the queue is the **first** one out.
  - Restricted form of a linear list: insert at **one end** and remove from **the other**.
  - FIFO access: first in, first out.

Queues Using Arrays

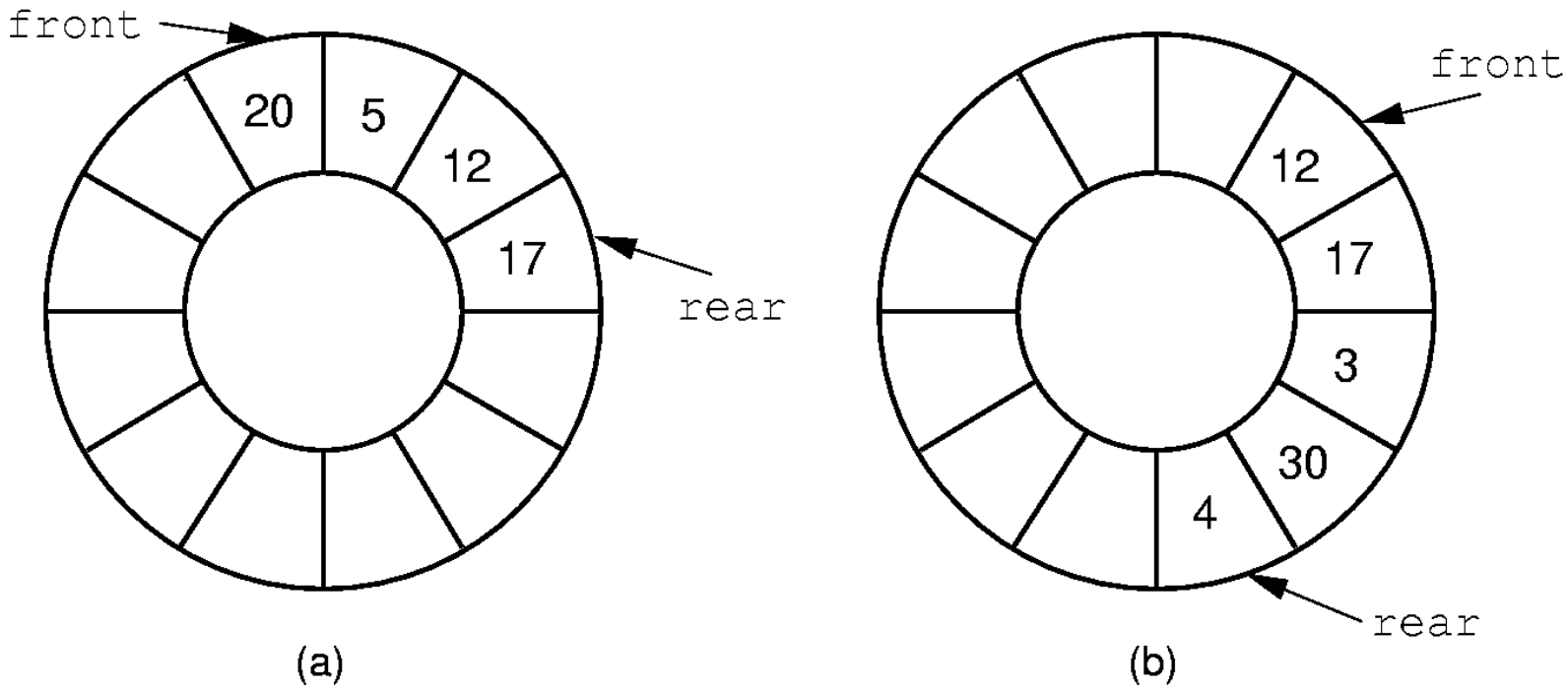


- We maintain two integers to indicate the front and the rear of the queue.

- However, as items are added and removed, the queue "drifts" toward the end.
  - Eventually, there will be no space to the right of the queue, even though there is space in the array.

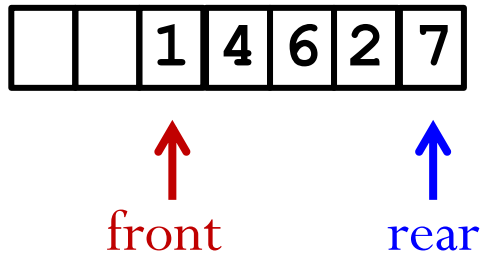
Queues Using Arrays

- To solve the problem of memory waste, we use a **circular array**.

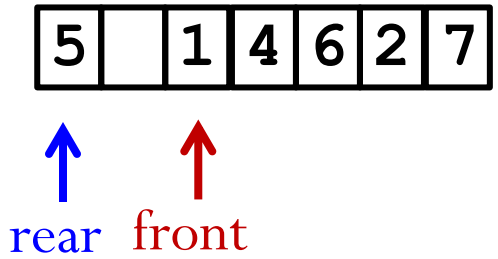


Circular Arrays

- We can implement a circular array using a plain linear array:
  - When front/rear equals the **last** index (i.e., MAXSIZE-1), increment of front/rear gives the **first** index (i.e., 0).



`enqueue(5)`

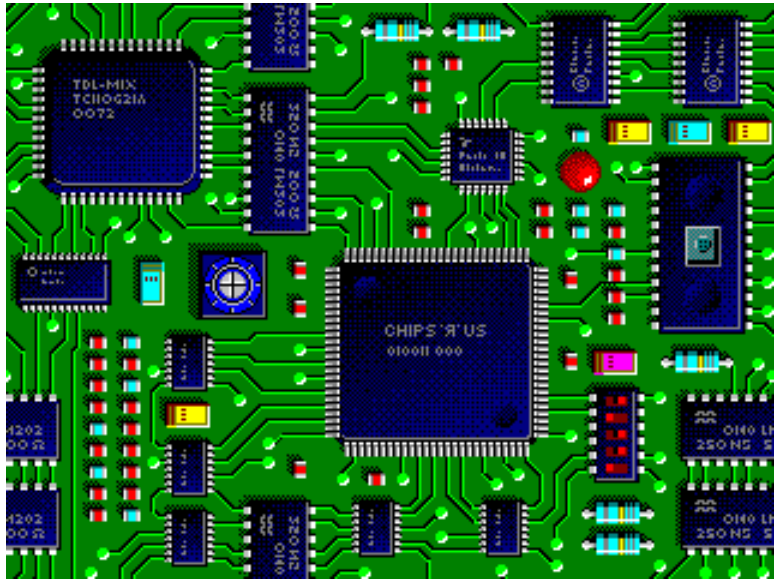


Application of Queues

- Request queue of a web server
  - Each user can send a request.
  - The arriving requests are stored in a **queue** and processed by the computer in a **first-come-first-serve** way.

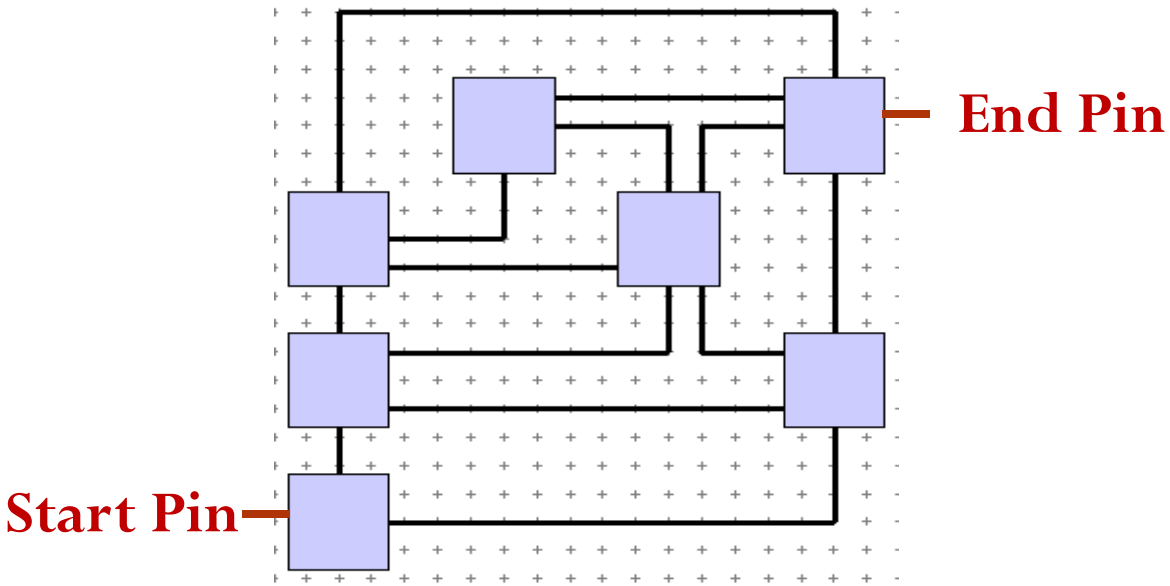
Application of Queue: Wire Routing

- Select paths to connect all pairs of pins that need to be connected together.
- An important problem in **electronic design automation**.



A Simplified Problem

- Condition: We have all blocks laid on the chip. We also have some of the wires routed.
- Problem: We want to connect the next pair of pins.
- Constraint: we can only draw wires horizontally or vertically.



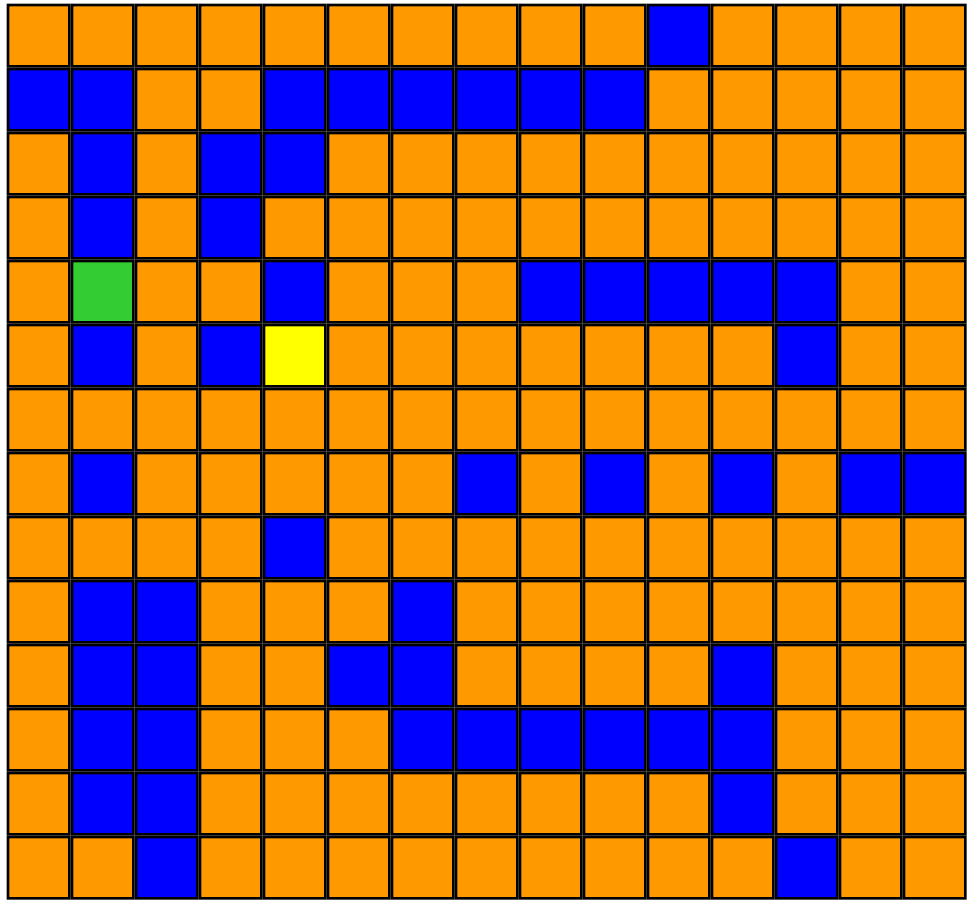
Modeling as a Grid

Start Pin

End Pin

- Blue squares are **blocked** squares.
- Orange squares are **available** to route a wire.

How to find a path from the start pin to the end pin?



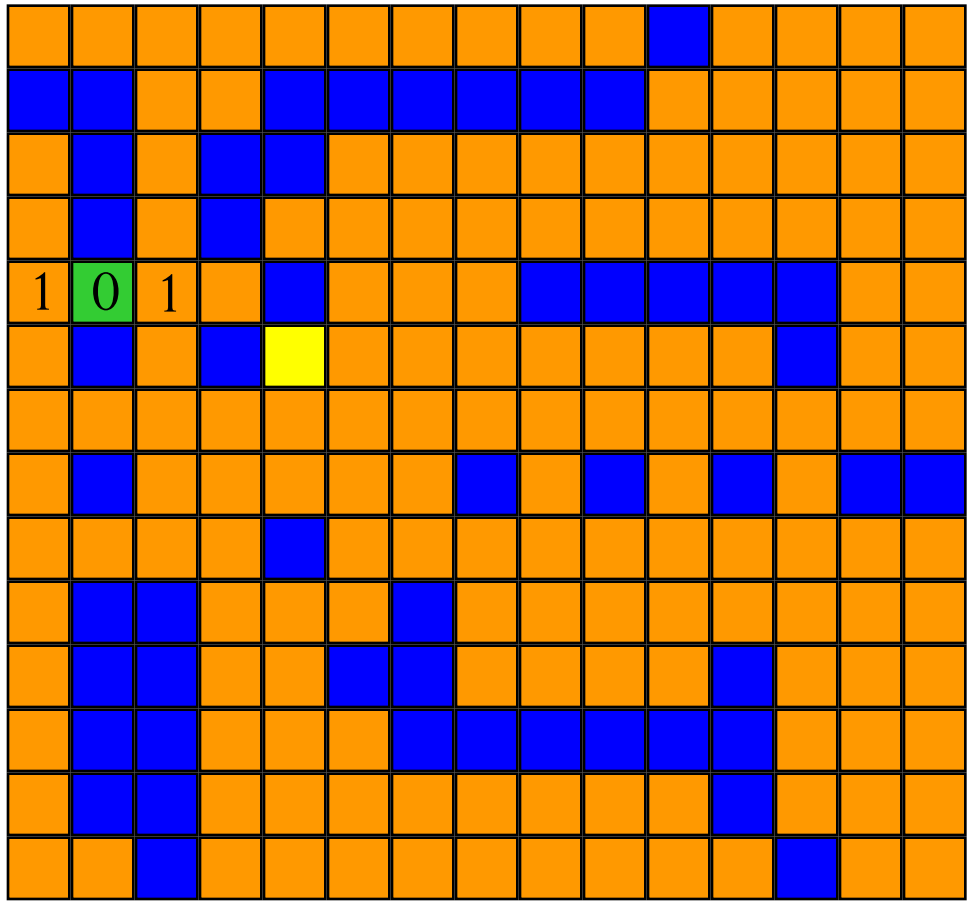
Wire Routing: Lee's Algorithm (1961)

- A **queue** of reachable squares from the start pin is used.
- The cell of the start pin is set with a distance value of 0.
- It is enqueued into an initial empty queue.
- **While** the queue is not empty.
  - A cell is **dequeued** from the queue and made the **examine cell**.
  - Is the examine cell the end pin? If yes, path found and return.
  - Otherwise, all **unreached unblocked** squares adjacent to the **examine cell** are marked with their distance (this is 1 more than the distance value of the **examine cell**) and **enqueued**.
- When queue becomes empty but not reach end pin yet, means no path found.

Illustration of Lee's Algorithm

start pin

end pin



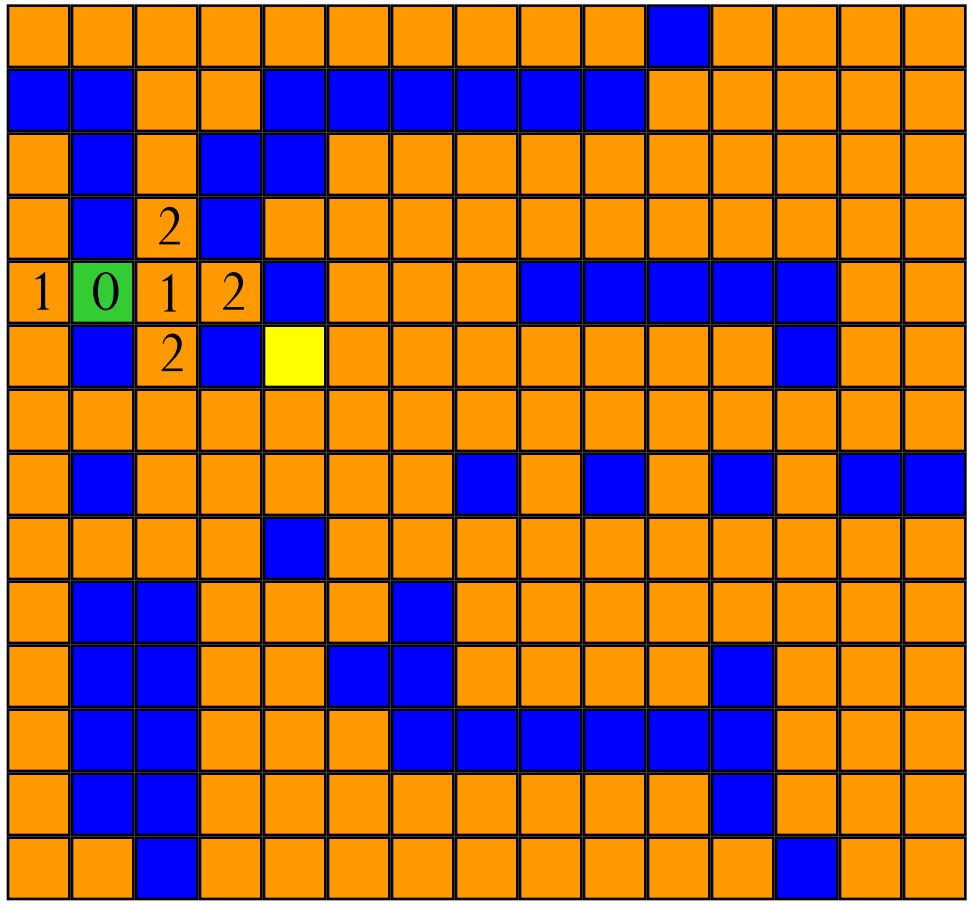
Expand "0"

queue: 0

Illustration of Lee's Algorithm

start pin

end pin



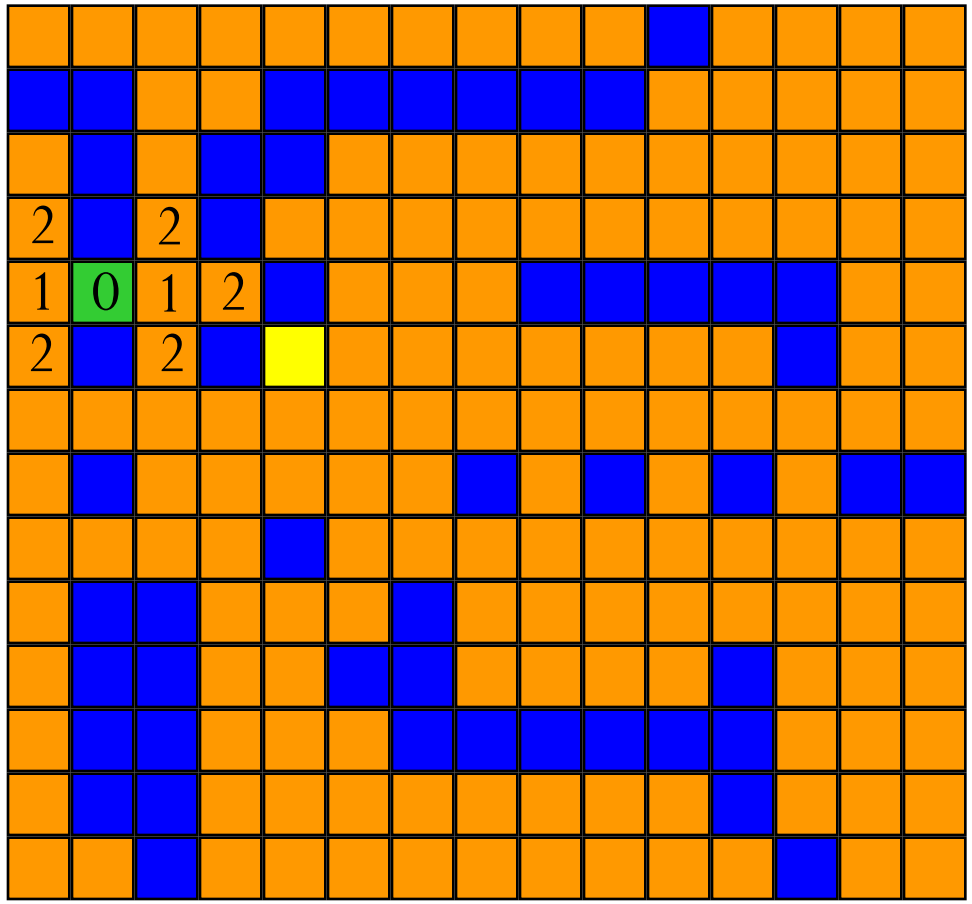
Expand right "1"

queue: 1, 1

Illustration of Lee's Algorithm

start pin

end pin



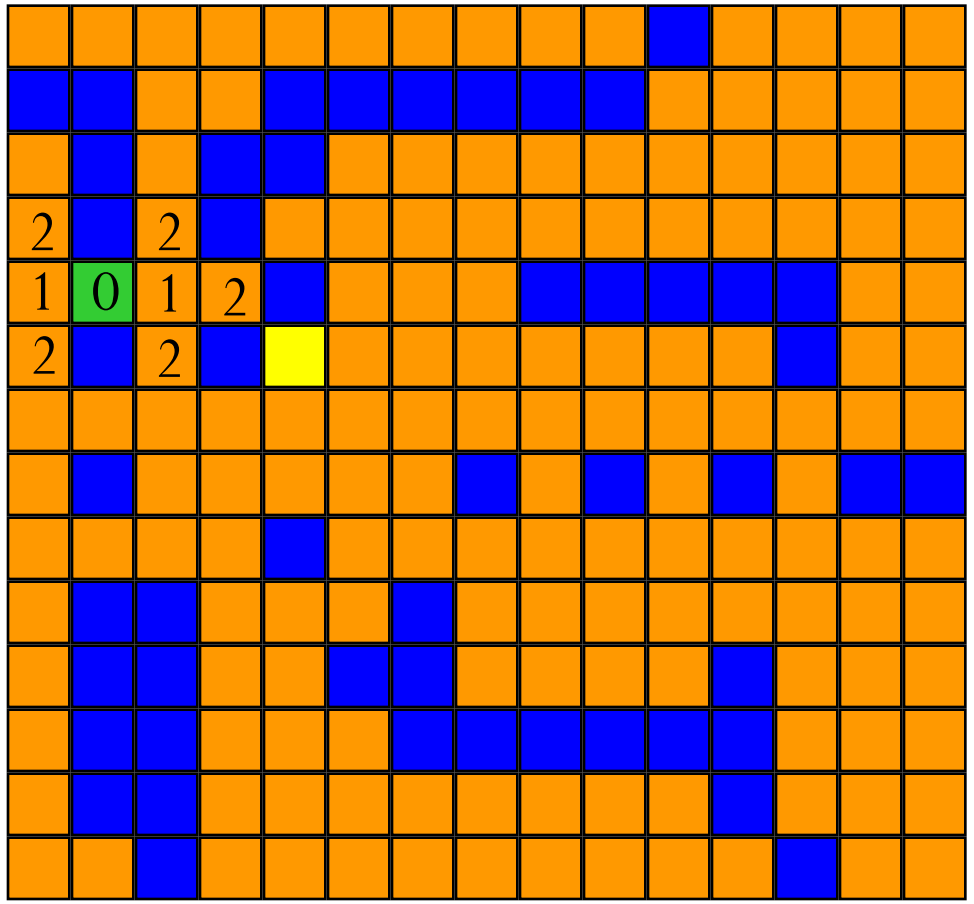
Expand left "1"

queue: 1,2,2,2

Illustration of Lee's Algorithm

start pin

end pin

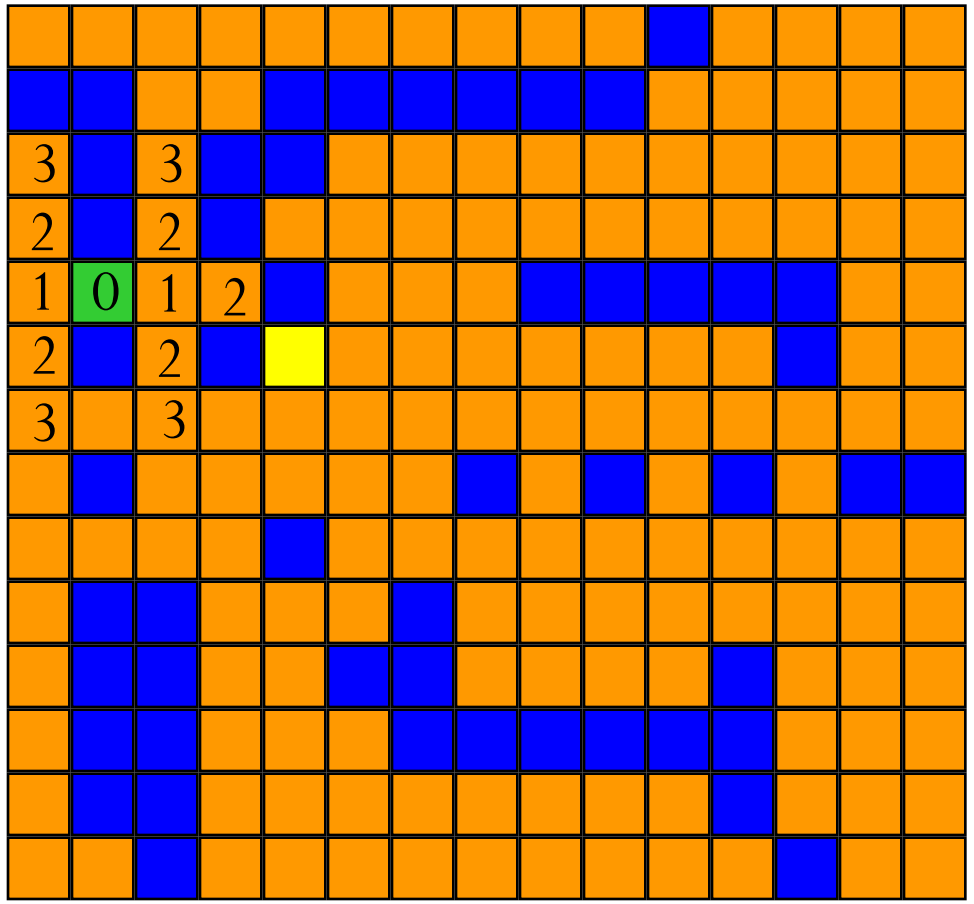


Expand and reach all squares 3 units from start.

Illustration of Lee's Algorithm

start pin

end pin

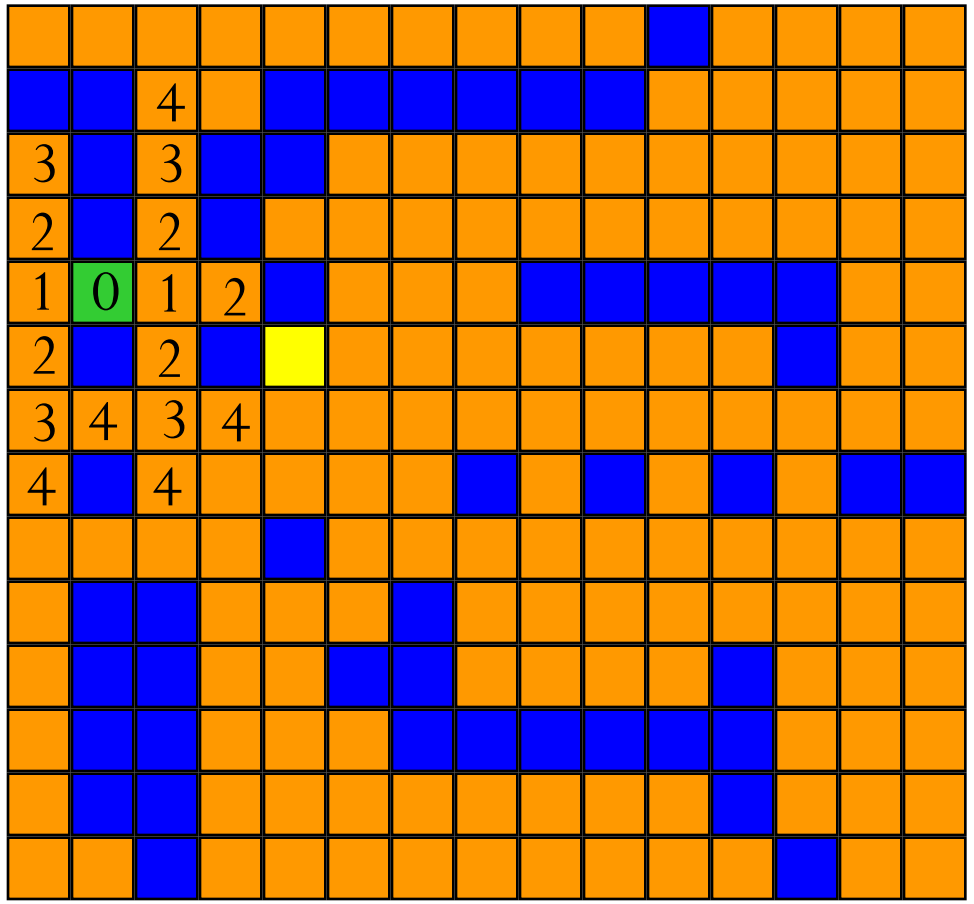


Expand and reach all squares 4 units from start.

Illustration of Lee's Algorithm

start pin

end pin

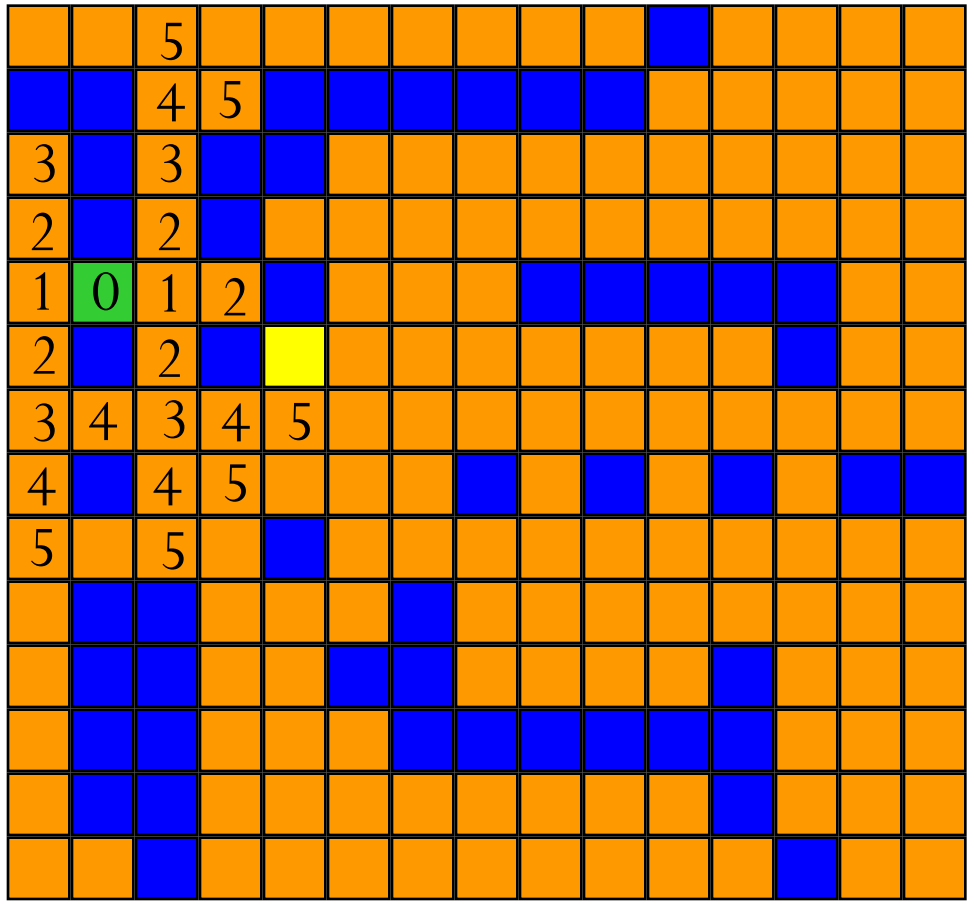


Expand and reach all squares 5 units from start.

Illustration of Lee's Algorithm

start pin

end pin

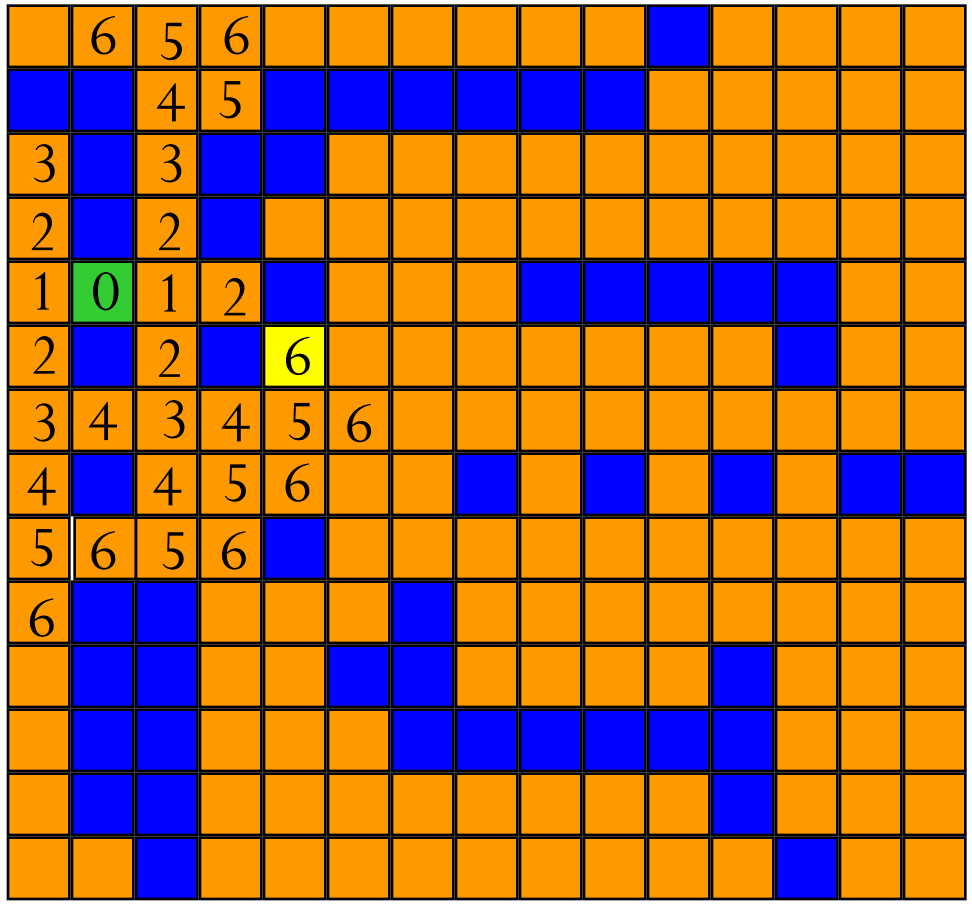


Expand and reach all squares 6 units from start.

Illustration of Lee's Algorithm

start pin

end pin

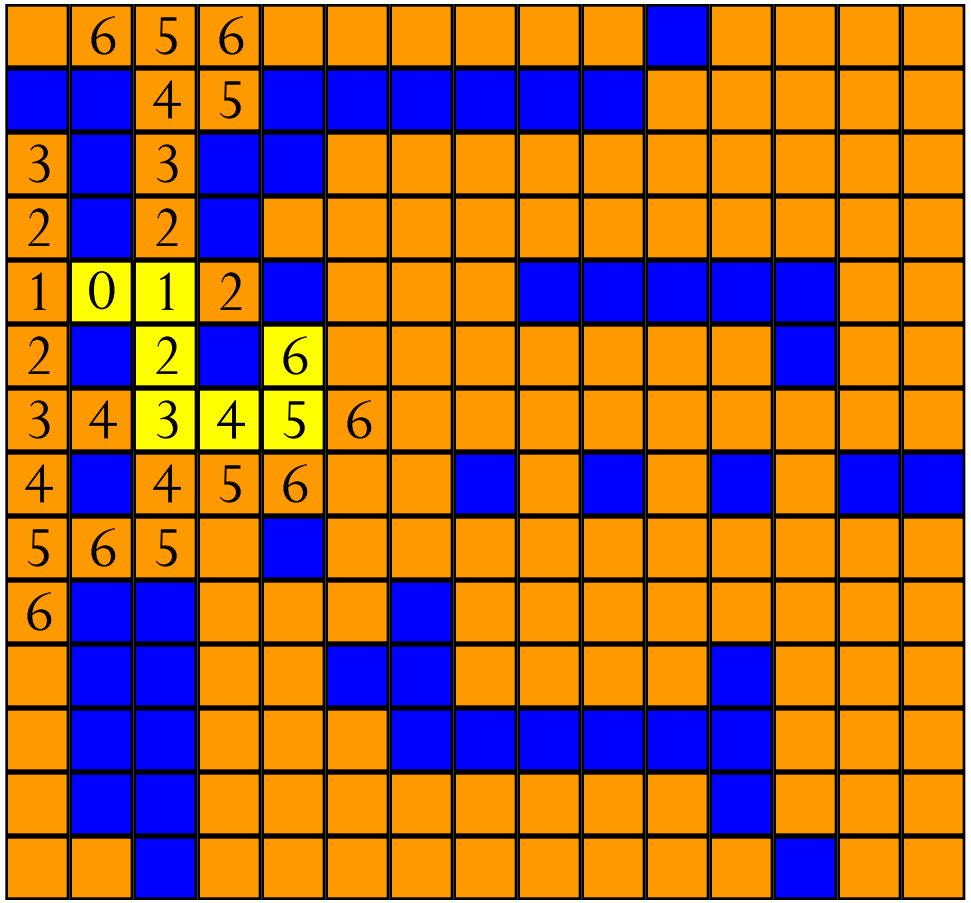


End pin reached. Trace back.

Illustration of Lee's Algorithm

start pin

end pin



Sorting

- General types of comparison sort
  - Insertion-based: insertion sort
  - Selection-based: selection sort, heap sort
  - Exchange-based: bubble sort, quick sort
  - Merging-based: merge sort

- Non-comparison sort:
  counting sort, bucket sort, radix sort

Comparison sort: each item is compared against others to determine its order.

Non-comparison sort: each item is put into predefined "bins" independent of the other items presented.

- No comparison with other items needed.
- It is also known as **distribution-based sort**.

Characteristics of Sorting Algorithms

- Average-case time complexity
- Worst-case time complexity
- Space usage: **in place** or not?
  - **in place**: requires $O(1)$ additional memory
  - Don't forget the stack space used in recursive calls
  - **In place is better**
    - Why? The data can fit into cache, not main memory
  - Real example: quick sort versus merge sort. Both have average-case time complexity of $O(n \log n)$. Quick sort is faster, due to in place

Characteristics of Sorting Algorithms

- **Stability**: whether the algorithm maintains the relative order of records with equal keys

(4, b), (3, e), (3, b), (5, b) → (3, e), (3, b), (4, b), (5, b)

Sort on the first number

Stable!

- Usually there is a secondary key whose ordering you want to keep. Stable sort is thus useful for sorting over multiple keys
- Example: sort complex numbers a+bi
  - Ordering rule: first compare a; when there is a tie, compare b
  - One sorting method: first sort b, then sort a

3+5i, 2+6i, 3+4i, 5+2i → (Sort on b) 5+2i, 3+4i, 3+5i, 2+6i → (… sort on a) 2+6i, 3+4i, 3+5i, 5+2i

Stability is important!

Comparison Sorts Summary

	Worst Case Time	Average Case Time	In Place	Stable
Insertion	$O(N^2)$	$O(N^2)$	Yes	Yes
Selection	$O(N^2)$	$O(N^2)$	Yes	No
Bubble	$O(N^2)$	$O(N^2)$	Yes	Yes
Merge Sort	$O(N \log N)$	$O(N \log N)$	No	Yes
Quick Sort	$O(N^2)$	$O(N \log N)$	Weakly	No

- Theorem: A sorting algorithm that is based on pairwise comparisons must use $\Omega(N \log N)$ operations to sort in the worst case.

Non-Comparison Sort

- Counting Sort
- Bucket Sort
- Radix Sort

Counting Sort

- Sort an array A of **integers** in the range $[0, k]$, where k is known.

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]=C[i-1]+C[i]`
   - `C[i]` now contains number of items less than or equal to `i`.
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

- Time complexity: $O(N + k)$.

- The algorithm can be converted to sort integers in some other known range $[a, b]$.
  - Minus each number by a, converting the range to $[0, b - a]$.

Counting Sort

Example

1. Allocate an array **`C[k+1]`**.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

2. Scan array A. For **`i=1`** to **`N`**, increment **`C[A[i]]`**.

	0	1	2	3	4	5
C	2	0	2	3	0	1

3. For **`i=1`** to **`k`**, **`C[i]= C[i-1]+C[i]`**

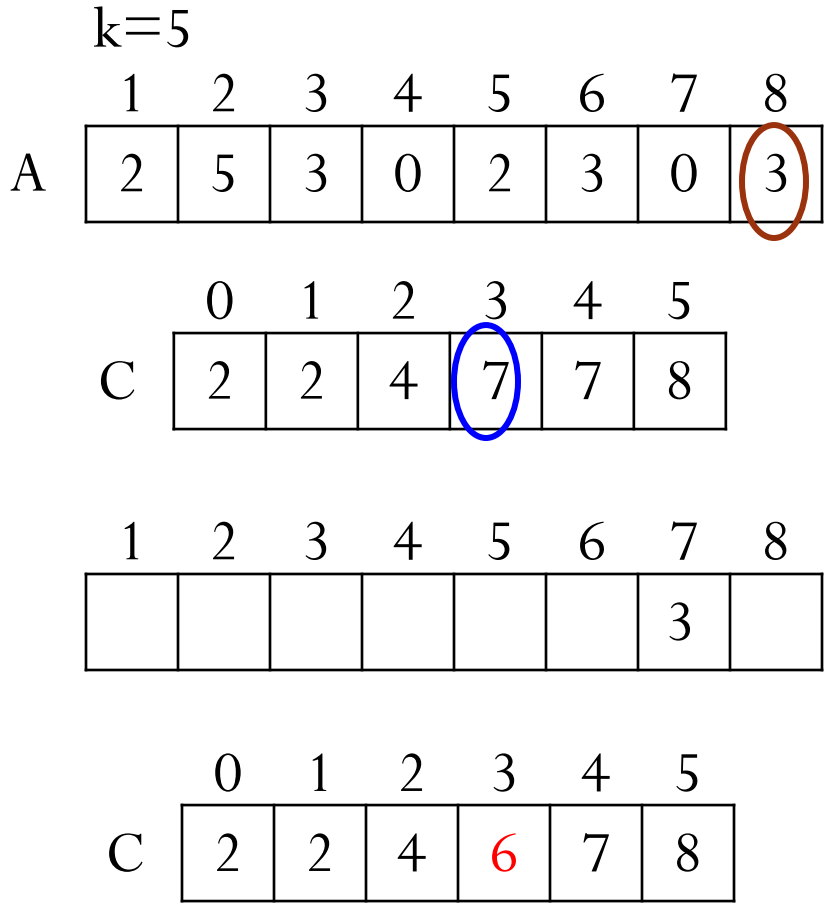
	0	1	2	3	4	5
C	2	2	4	7	7	8

4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

Counting Sort

Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

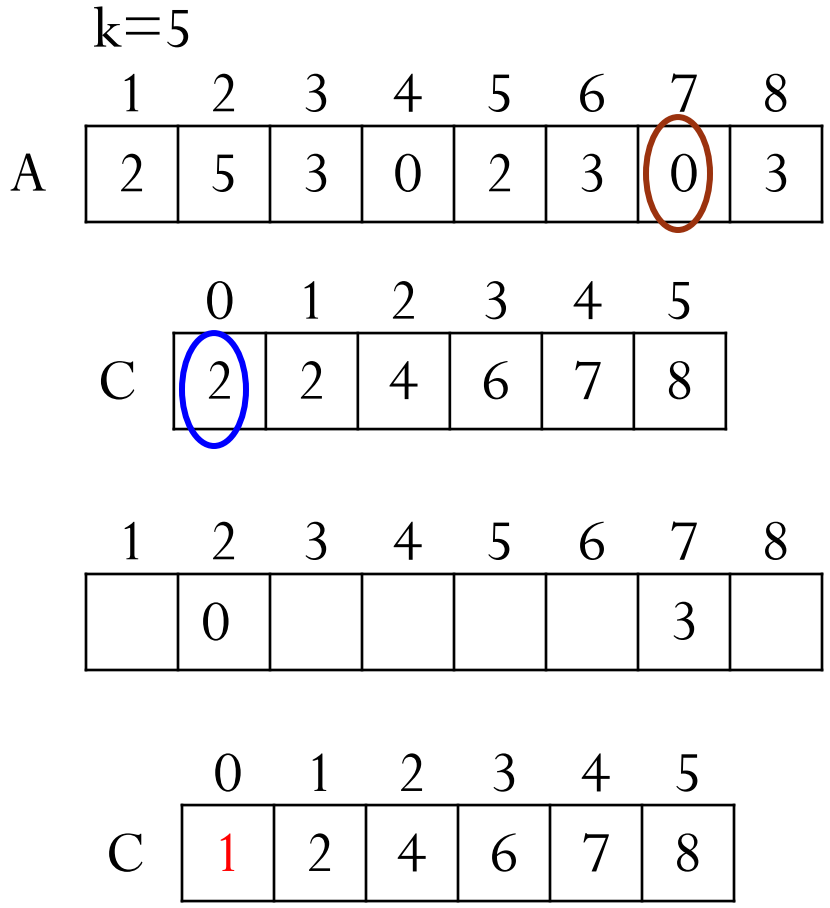


Why putting 3 at location 7 is correct?

Counting Sort

Example

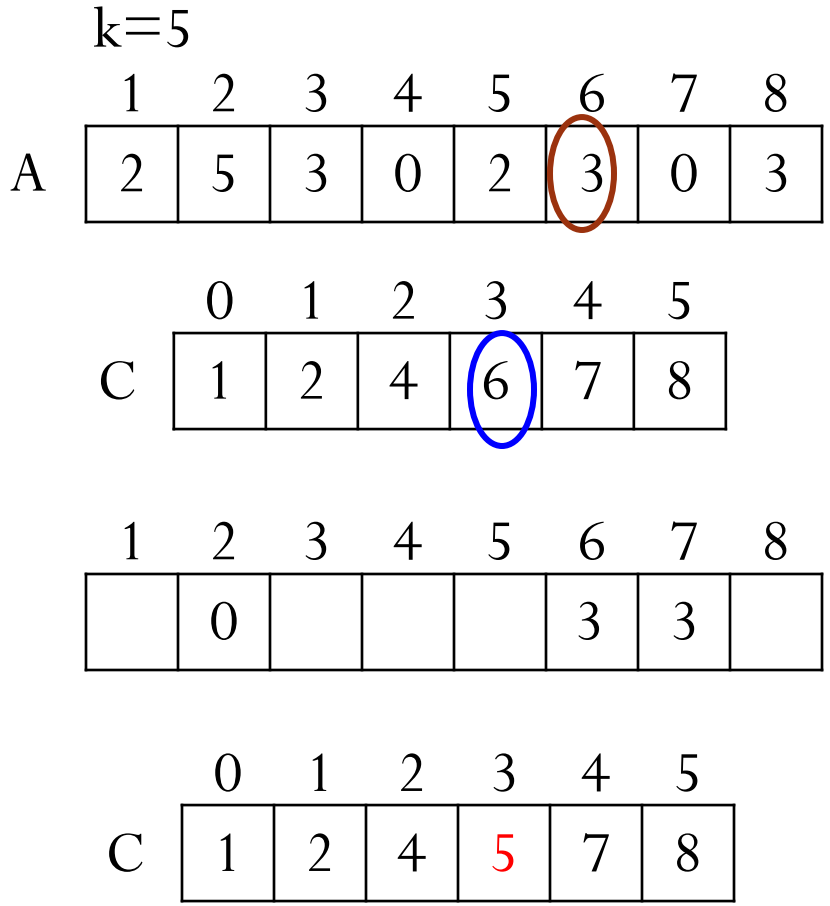
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.



Counting Sort

Example

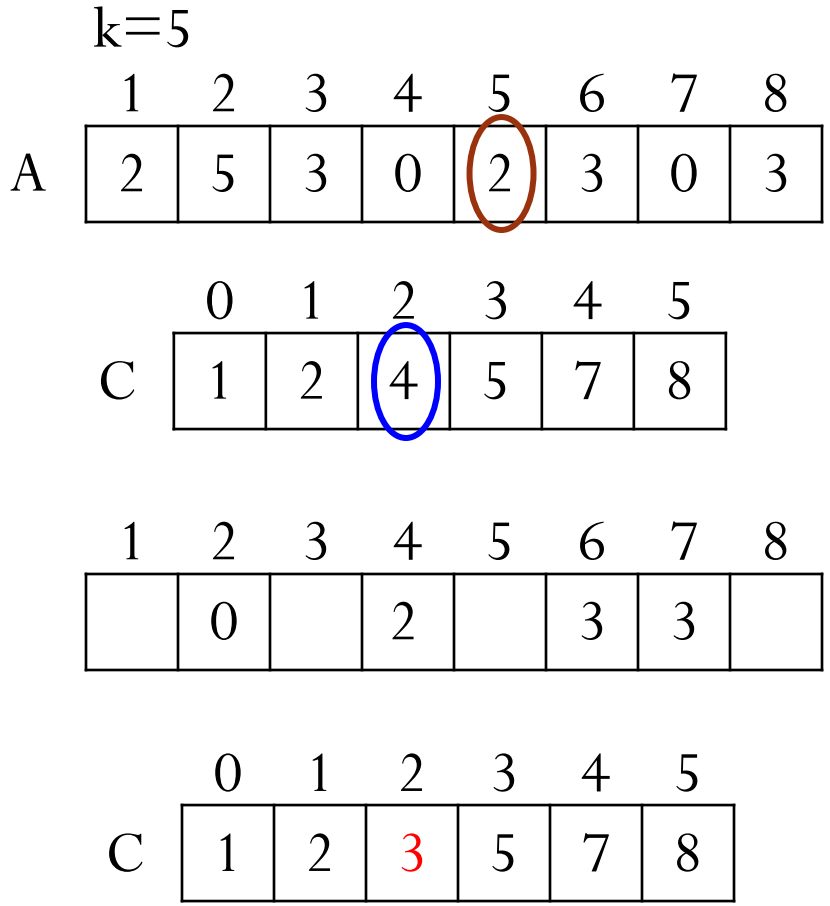
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.



Counting Sort

Example

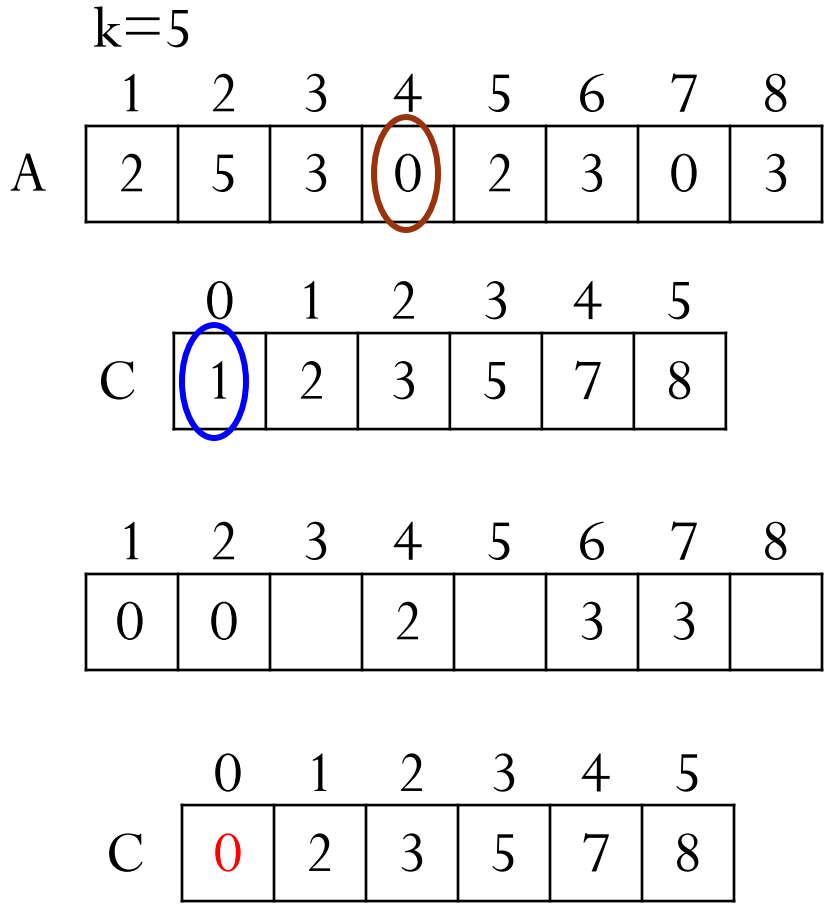
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.



Counting Sort

Example

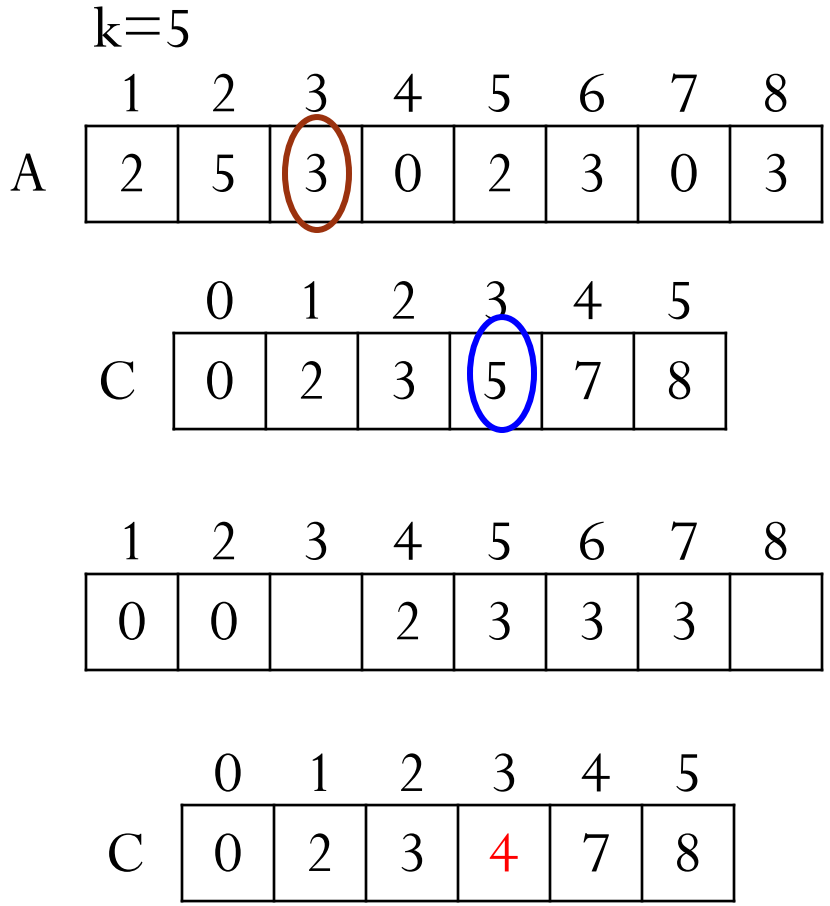
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.



Counting Sort

Example

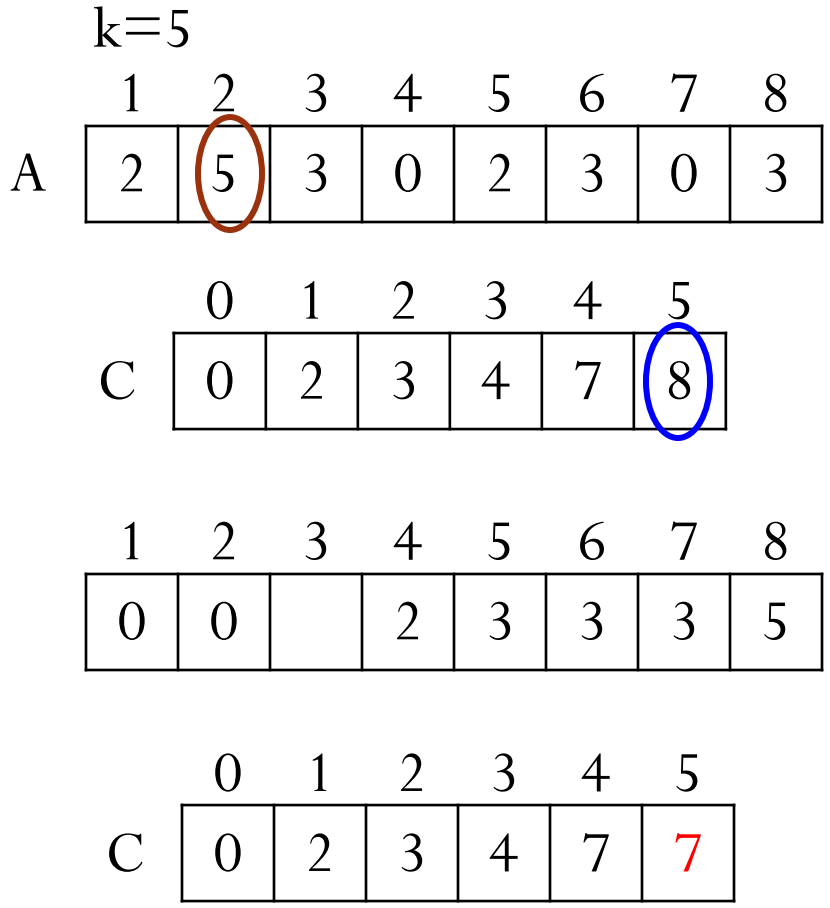
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.



Counting Sort

Example

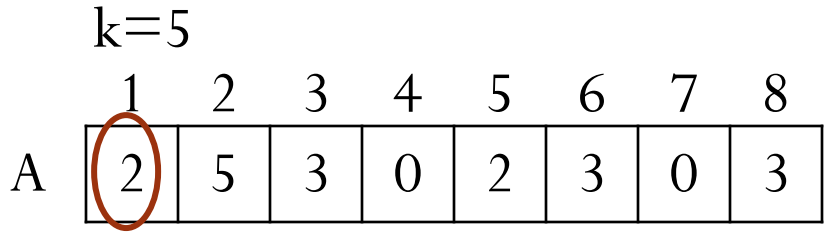
1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.

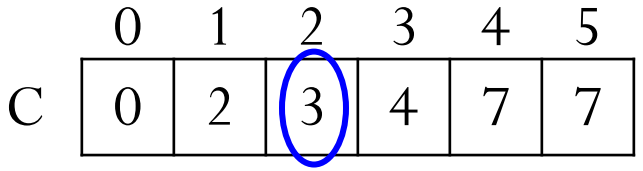


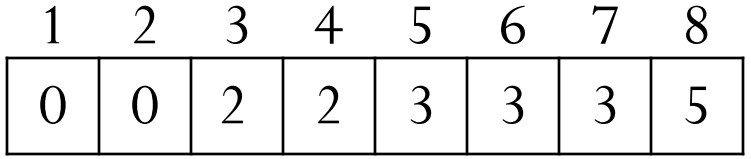
Counting Sort

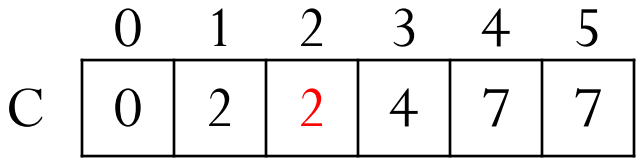
Example

1. Allocate an array `C[k+1]`.
2. Scan array A. For `i=1` to `N`, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**.









Done!

Is counting sort stable? Yes!

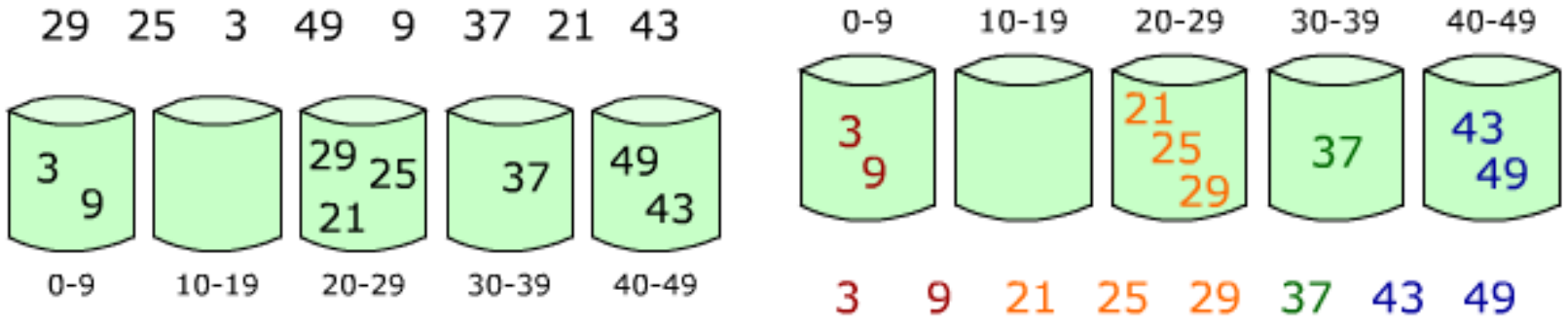
Bucket Sort

- Instead of simple integer, each key can be a complicated record, such as a real value.
- Then instead of incrementing the count of each bucket, **distribute** the records **by their keys** into appropriate buckets.
- Algorithm:

1. Set up an array of initially empty "buckets".
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket by a comparison sort.
4. Gather: Visit the buckets in order and put all elements back into the original array.

Bucket Sort

- Example



- Time complexity
  - Suppose we are sorting cN items and we divide the entire range into N buckets.
  - Assume that the items are uniformly distributed in the entire range.
  - The average case time complexity is $O(N)$.

Radix Sort

- **Radix sort** sorts integers by looking at one digit at a time.
- Procedure: Given an array of integers, from the least significant bit (LSB) to the most significant bit (MSB), repeatedly do **stable** bucket sort according to the current bit.

- For sorting base-b numbers, bucket sort needs b buckets.
  - For example, for sorting decimal numbers, bucket sort needs 10 buckets.

Radix Sort

Example

- Sort 815, 906, 127, 913, 098, 632, 278.
- Bucket sort 81<u>5</u>, 90<u>6</u>, 12<u>7</u>, 91<u>3</u>, 09<u>8</u>, 63<u>2</u>, 27<u>8</u> according to the least significant bit:

0	1	2	3	4	5	6	7	8	9
		63<u>2</u>	91<u>3</u>		81<u>5</u>	90<u>6</u>	12<u>7</u>	09<u>8</u> 27<u>8</u>	

- Bucket sort 6<u>3</u>2, 9<u>1</u>3, 8<u>1</u>5, 9<u>0</u>6, 1<u>2</u>7, 0<u>9</u>8, 2<u>7</u>8 according to the second bit.

Radix Sort

Example

- Bucket sort 632, 913, 815, 906, 127, 098, 278 according to the second bit.

0	1	2	3	4	5	6	7	8	9
906	913 815	127	632				278		098

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

Radix Sort

Example

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

0	1	2	3	4	5	6	7	8	9
098	127	278				632		815	906 913

- The final sorted order is: 098, 127, 278, 632, 815, 906, 913.

Radix Sort: Correctness

- Claim: after bucket sorting the i-th LSB, the numbers are sorted according to their last i digits

- Proof by mathematical induction

Radix Sort

Time Complexity

- Let k be the maximum number of digits in the keys and N be the number of keys.
- We need to repeat bucket sort k times.
  - Time complexity for the bucket sort is $O(N)$.

- The total time complexity is $O(kN)$.

Radix Sort

- Radix sort can be applied to sort keys that are built on **positional notation**.
  - **Positional notation**: all positions uses the same set of symbols, but different positions have different weight.
  - Decimal representation and binary representation are examples of positional notation.
  - Strings can also be viewed as a type of positional notation. Thus, radix sort can be used to sort strings.
- We can also apply radix sort to sort records that contain multiple keys.
  - For example, sort records (year, month, day).

Linear Time Selection

- Randomized selection algorithm
- Deterministic selection algorithm

The Selection Problem

- Input: array A with n distinct numbers and a number i
  - "**Distinct**" for simplicity
- Output: i-th smallest element in the array
  - Assume index starts from 1
- Example: $A = (6, 3, 5, 4, 2), i = 3$
  - Should return 4
- Special cases
  - $i = 1$: the smallest item.
  - $i = n$: the largest item.
  - $i = n/2$: the median

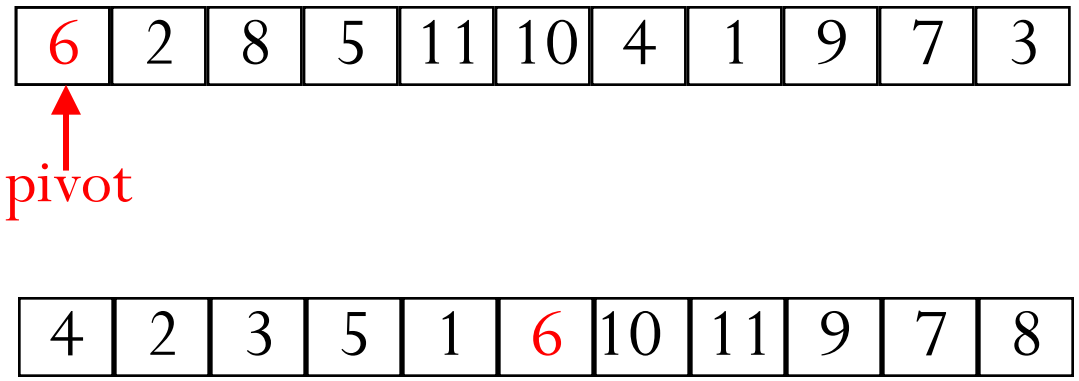
Solution: Reduction to Sorting

- Step 1: Do merge sort
- Step 2: output the i-th element of the sorted array
- Time complexity is $O(n \log n)$

- Can we do better?
  - This essentially asks whether selection is **fundamentally easier** than sorting
  - Answer: Yes!
  - We will show an $O(n)$ time randomized algorithm by modifying quick sort
  - Also will show an $O(n)$ time deterministic algorithm (However, not as practical as the randomized algorithm)

Recall: Partitioning in Quick Sort

- Pick a pivot
- Put all elements $<$ pivot to the left of pivot
- Put all elements $\geq$ pivot to the right of pivot
- Move pivot to its correct place in the array

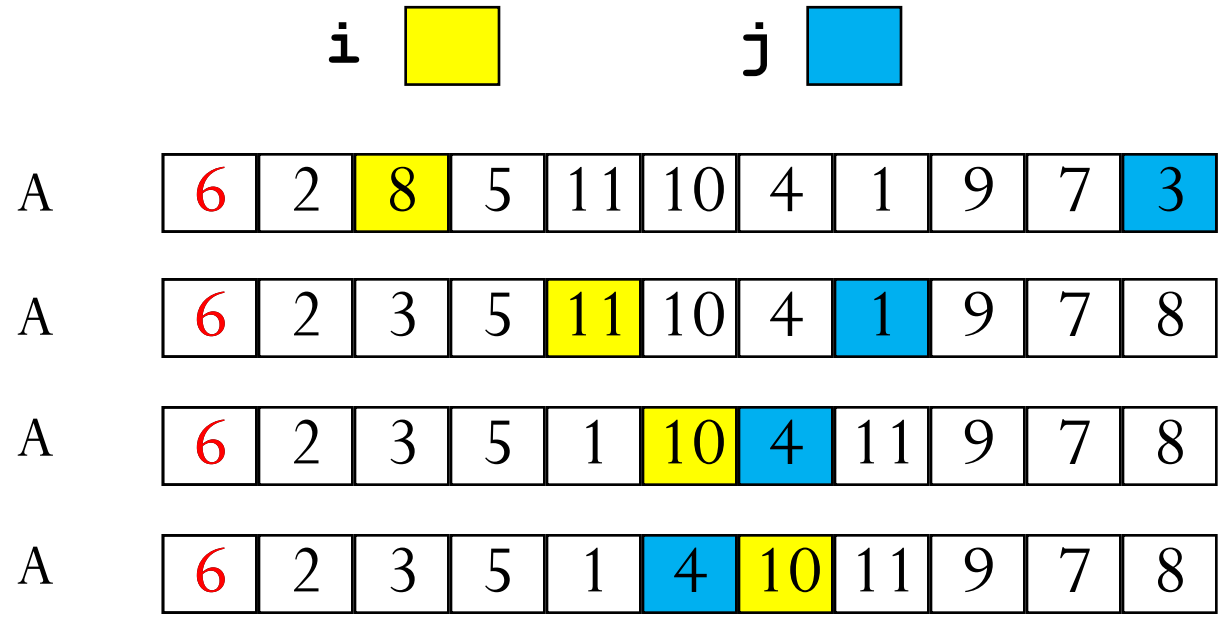


Recall: In-Place Partitioning the Array

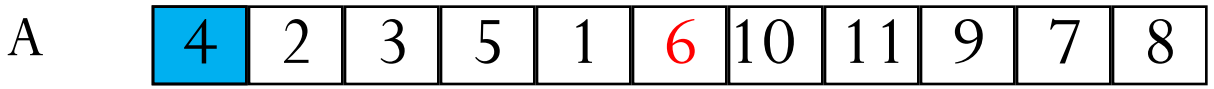
1. Once pivot is chosen, swap pivot to the beginning of the array.
2. Start counters **`i=1`** and **`j=N-1`**.
3. Increment **`i`** until we find element **`A[i]>=pivot`**.
   - **`A[i]`** is the leftmost item ≥ pivot.
4. Decrement **`j`** until we find element **`A[j]<pivot`**.
   - **`A[j]`** is the rightmost item < pivot.
5. If **`i<j`**, swap **`A[i]`** with **`A[j]`**. Go back to step 3.
6. Otherwise, swap the first element (pivot) with **`A[j]`**.

Recall: In-Place Partitioning the Array

Example



- Now, **`j < i`**, swap the first element (pivot) with **`A[j]`**.



Recall: In-Place Partitioning the Array

Time Complexity

1. Once pivot is chosen, swap pivot to the beginning of the array.
2. Start counters **`i=1`** and **`j=N-1`**.
3. Increment **`i`** until we find element **`A[i]>=pivot`**.
4. Decrement **`j`** until we find element **`A[j]<pivot`**.
5. If **`i<j`**, swap **`A[i]`** with **`A[j]`**. Go back to step 3.
6. Otherwise, swap the first element (pivot) with **`A[j]`**.

- Scan the entire array no more than twice.
- Time complexity is $O(N)$, where N is the size of the array.

Basic Idea

- Suppose we are looking for 6^{th} smallest item in an array of length 12. We do partition.
  - Suppose the pivot is at position 4. Then we only need to focus on the sub-array right of the pivot and look for the 2^{nd} item in the array
  - Suppose the pivot is at position 8. Then we only need to focus on the sub-array left of the pivot and look for the 6^{th} item in the array
  - In both cases, recurse!

Randomized Selection

```
Rselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Choose pivot p from A uniformly at random;
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Rselect(1st part of A, j-1, i);
  else return Rselect(2nd part of A, n-j, i-j);
}
```

Average Runtime of Rselect

- Theorem: for every input array of length n, the average runtime of Rselect is $O(n)$
  - Holds for every input data (no assumption on data)
  - "Average" is over random pivot choices made by the algorithm

Average Runtime Analysis

- Note: Rselect uses $\leq cn$ operations outside of recursive call (from partitioning)
- Observation: the length of the array the algorithm works on decreases
- Definition: We say Rselect is in phase j if current array size is between $(\frac{3}{4})^{j+1}n$ and $(\frac{3}{4})^{j}n$
- X_j denote the number of recursive calls in phase j
- $runtime \leq \sum_j X_j \cdot c \cdot (\frac{3}{4})^j \, n$

We need to further get $E[X_j]$

$$E[runtime] \leq E\left[\sum_j X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right] = cn \sum_j \left(\frac{3}{4}\right)^j E[X_j]$$

Average Runtime Analysis

- **Claim**: If Rselect chooses a pivot so that the **left sub-array**'s size is am, where $a \in [\frac{1}{4}, \frac{3}{4}]$ and m is the old length, then the current phase ends
  - Because new sub-array length is at most 75% of the old length
  - **"Good pivot"**
- What is the probability of $a \in [\frac{1}{4}, \frac{3}{4}]$ (i.e., good pivot)?
  - Answer: 0.5
- Claim: $E[X_j] \leq$ Expected number of times you need to get a good pivot
  - Same as the expected number of times you flip a fair coin to get a "head". (Heads: good pivot; tails: bad pivot)

Coin Flipping Analysis

- Let N be the number of coin flips until you get heads
  - N is a geometric random variable: $P(N = k) = \frac{1}{2^k}, k = 1,2, \ldots$

#flips when 1st is head

#flips when 1st is tail

- $E[N] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (1 + E[N])$

$$\Rightarrow E[N] = 2$$

Prob. 1st flip is head

Prob. 1st flip is tail

Therefore, $E[X_j] \leq E[N] = 2$

Average Runtime Analysis

$$E[runtime] \leq E\left[\sum_j X_j \cdot c \cdot \left(\frac{3}{4}\right)^j n\right]$$

$$= cn\sum_j \left(\frac{3}{4}\right)^j E[X_j] \quad \leq 2cn\sum_j \left(\frac{3}{4}\right)^j \leq 2cn\frac{1}{1-\frac{3}{4}}$$

$$= 8cn = O(n)$$

Outline

- Randomized selection algorithm

- Deterministic selection algorithm

A Good Pivot

- Best pivot: the median
  - But, this is a circular problem
- Goal: find pivot guaranteed to be good enough
- Idea: use “median of medians”

A Deterministic ChoosePivot

`ChoosePivot(A, n)`

- A subroutine called by the deterministic selection algorithm
- Steps:

1. Break A into n/5 groups of size 5 each
2. Sort each group (e.g., use insertion sort)
3. Copy n/5 medians into new array C
4. Recursively compute median of C
   - By calling the deterministic selection algorithm!
5. Return the median of C as pivot

Deterministic Selection Algorithm

```
Dselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Break A into groups of 5, sort each group;
  C = n/5 medians;
  p = Dselect(C, n/5, n/10);
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Dselect(1st part of A, j-1, i);
  else return Dselect(2nd part of A, n-j, i-j);
}
```

ChoosePivot

Same as Rselect

The function has two recursive calls

Runtime of Dselect

- Theorem: For every input array of length n, Dselect runs in $O(n)$ time

- Warning: not as good as Rselect in practice
  - Worse constants
  - Not-in-place

Runtime of Dselect

Assume the runtime is $T(n)$

```
Dselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
1 if(n == 1) return A[1];
2 Break A into groups of 5, sort each group; Θ(n)
3 C = n/5 medians;  Θ(n)
4 p = Dselect(C, n/5, n/10); T(n/5)
5 Partition A using pivot p; Θ(n)
6 Let j be the index of p;
7 if(j == i) return p;
8 if(j > i) return Dselect(1st part of A, j-1, i);
9 else return Dselect(2nd part of A, n-j, i-j);
}
```

$T(?)$

Recurrence

- There exists a positive constant c such that
  - $T(1) \leq c$
  - $T(n) \leq cn + T\left(\frac{n}{5}\right) + T(?)$

- The next question is what is the size of the array of the second recursive call

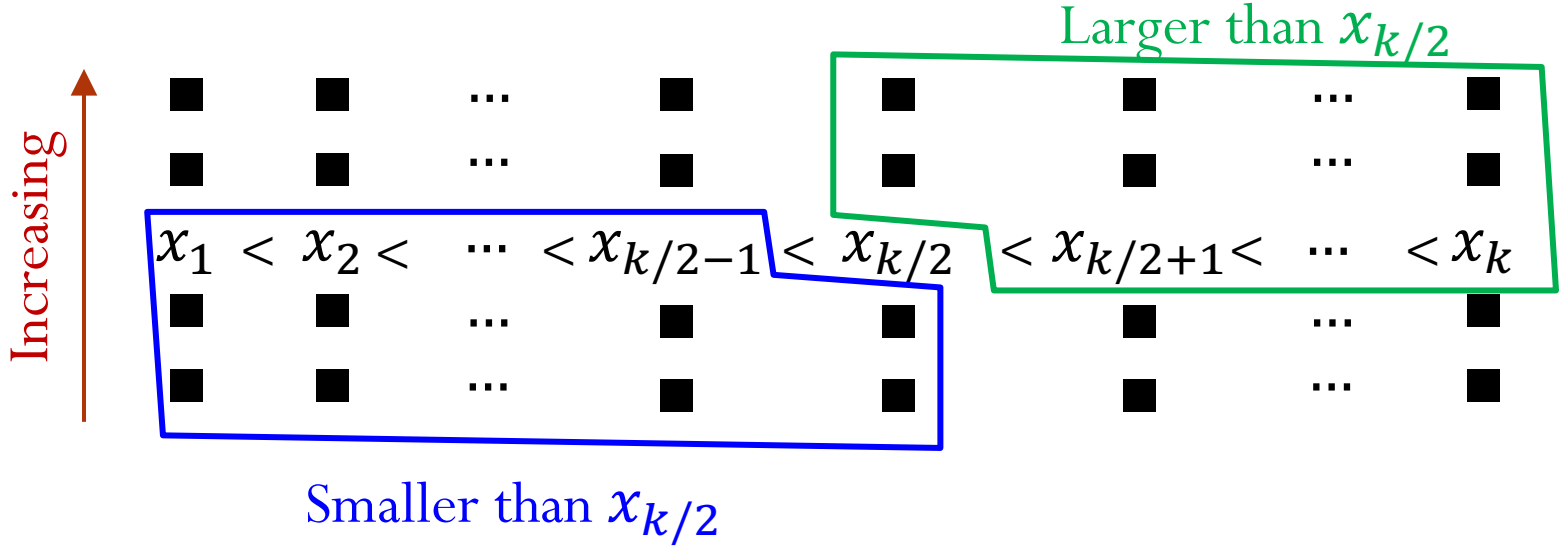
Lemma on Size

- Lemma: 2nd recursive call guaranteed to be on an array of size $\leq 0.7n$ (roughly)

- (Rough) proof:
  - Let $k = n/5$: number of groups
  - Let x_i be the i-th smallest of the k medians
  - Thus, the pivot is $x_{k/2}$
  - Goal
    - >=30% of input array smaller than $x_{k/2}$
    - >=30% of input array larger than $x_{k/2}$

Proof of Lemma

- Imagine we layout elements of A in a 2-D grid



- At least ~ (3/5)*(1/2) = 30% elements smaller than $x_{k/2}$
- At least ~ 30% elements larger than $x_{k/2}$
- Result: Number of elements $< x_{k/2}$ is in between 30% and 70%. The same for number of elements $> x_{k/2}$

Example

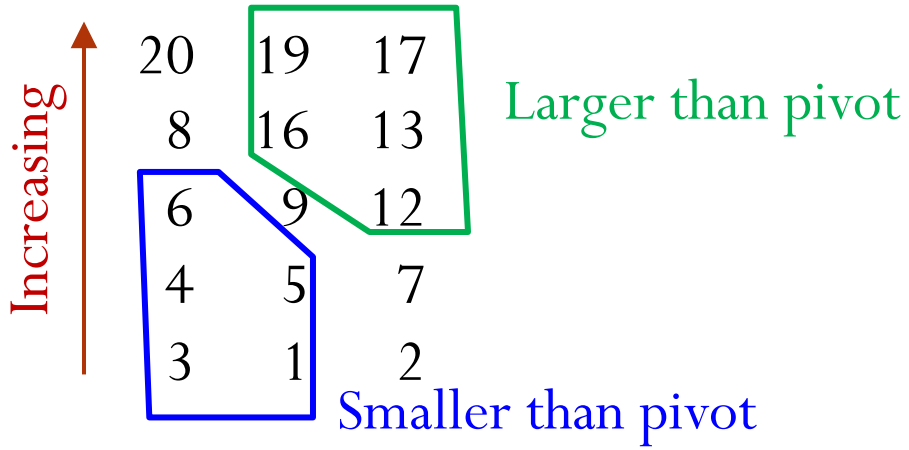
- Input:

7,2,17,12,13,8,20,4,6,3,19,1,9,5,16

- After sorting each group of 5 elements

2,7,12,13,17,3,4,6,8,20,1,5,9,16,19

pivot



Recurrence

- There exists a positive constant c such that
  - $T(1) \leq c$
  - $T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$
- How can we solve this?
  - Strategy: Hope and check
- Hope: there is a constant a (independent of n) such that $T(n) \leq an$ for all $n > 1$
  - Then $T(n) = O(n)$
- We choose $a = 10c$

Proof $T(n) = O(n)$

- Claim: suppose there exists a positive constant c such that
  1. $T(1) \leq c$
  2. $T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$

  Then $T(n) \leq 10cn$
- Proof by induction
  - Base case: $T(1) \leq 10c$
  - Inductive step: inductive hypothesis $T(k) \leq 10ck, \forall k < n.$ Then

$$T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) \leq cn + 2cn + 7cn = 10cn$$

Dselect runs in linear time

Reference

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 13 Pointers and Linked Lists

- **Data Structures and Algorithm Analysis Edition 3.2 (C++ Version)**, by *Clifford A. Shaffer* (2012)
  - Chapter 7 Internal Sorting