Lectures 11-12.
Counting Sort - Radix Sort - Medians

Introduction to Algorithms

Da Nang University of Science and Technology

Dang Thien Binh

dtbinh@dut.udn.vn

Sorting So Far (1/4)

- Insertion Sort
  - Easy to code
  - Fast on small inputs (less than ~ 50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting So Far (2/4)

- Merge Sort
  - Divide-and-Conquer
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - $O(nlgn)$ worst case
  - It does not sort in place

Sorting So Far (3/4)

- Heap Sort
  - It uses the very useful heap data structure
    - Complete binary tree
    - Heap property: parent key > children's keys
  - $O(nlgn)$ worst case
  - It sorts in place

Sorting So Far (4/4)

- Quick Sort
  - Divide-and-Conquer
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed !
  - $O(nlgn)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
    - Naïve implementation: worst case on sorted input
    - Address this with randomized quicksort

How Fast Can We Sort?

- We will provide a lower bound, then beat it
  - *How do you suppose we'll beat it?*
- First, an observation: all of the sorting algorithms so far are *comparison sorts*
  - The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements
  - Theorem: all comparison sorts are $\Omega(nlgn)$

Decision Trees

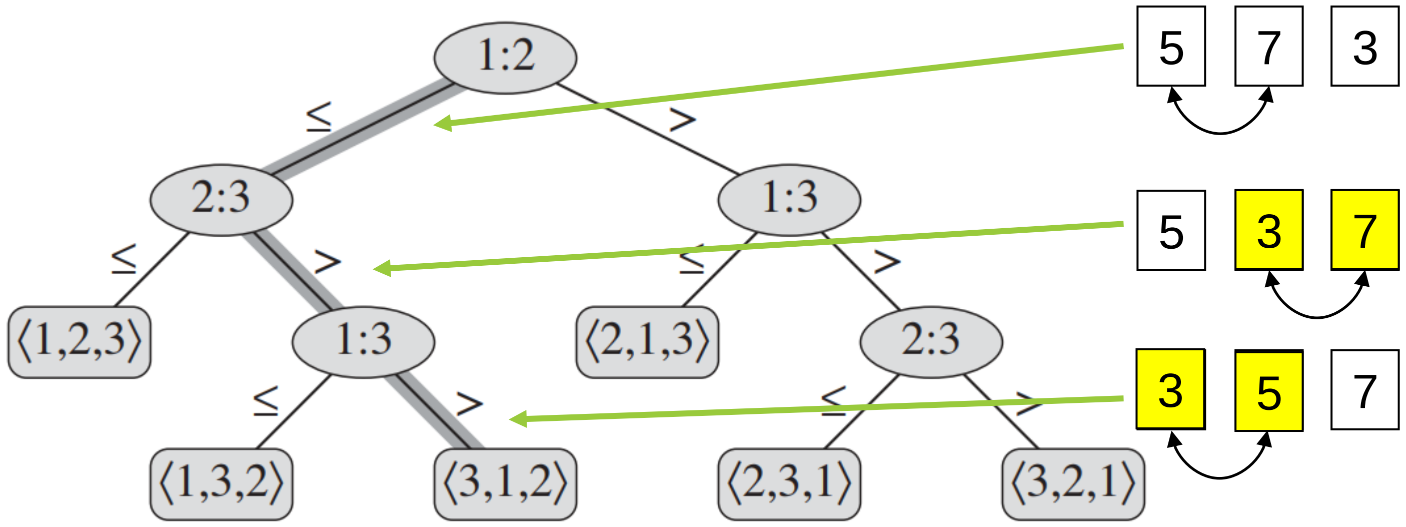
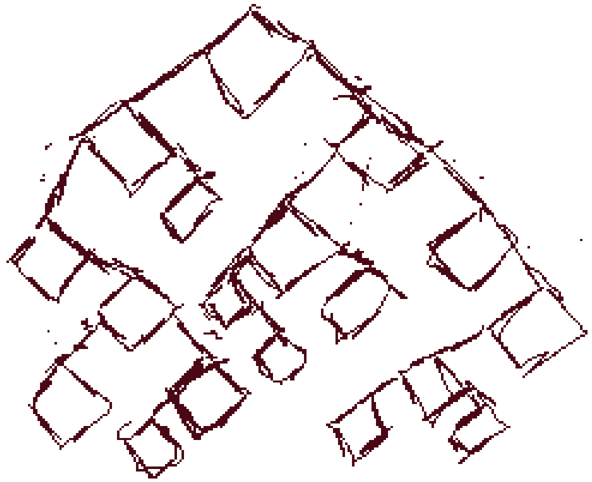


Figure 8.1 The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between a_i and a_j. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, and so the decision tree must have at least 6 leaves.

Decision Trees

- *Decision trees* provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort.
- *What do the leaves represent?*
- *How many leaves must there be?*



Practice Problems

- What is the smallest possible depth of a leaf in a decision tree for a comparison sort?
  - The absolute best case happens when we just check every element and see that the data is already sorted.
  - This will result in $n - 1$ comparisons and thus the leaf will have a depth of $n - 1$.

Decision Trees

- Decision trees can model comparison sorts
  For a given algorithm:
  - One tree for each n
  - Tree paths are all possible execution traces
  - *What's the longest path in a decision tree for insertion sort? For merge sort?*
- *What is the asymptotic height of any decision tree for sorting n elements?*
- Answer: $\Omega(nlgn)$ (Now let's prove it...)

A Lower Bound for Comparison Sorting

- Theorem: Any decision tree that sorts n elements has height $\Omega(nlgn)$
- *What's the minimum number of leaves?*
- *What's the maximum number of leaves of a binary tree of height h?*
- Clearly the minimum number of leaves is less than or equal to the maximum number of leaves

A Lower Bound for Comparison Sorting

- So we have…

$$n! \leq 2^h$$

- Taking logarithms:

$$\lg(n!) \leq h$$

- Stirling's approximation tells us:

$$n! > \left(\frac{n}{e}\right)^n$$

- Thus:

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

A Lower Bound for Comparison Sorting

- So we have

$$
\begin{aligned}
h &\geq \lg\left(\frac{n}{e}\right)^n \\
&= n\lg n - n\lg e \\
&= \Omega(n\lg n)
\end{aligned}
$$

- Thus the minimum height of a decision tree is $\Omega(n\lg n)$

A Lower Bound for Comparison Sorts

- Thus the time to comparison sort n elements is $\Omega(n \lg n)$
- Corollary: Heapsort and Merge sort are asymptotically optimal comparison sorts
- But the name of this lecture is "Sorting in linear time"!
  - *How can we do better than $\Omega(n \lg n)$?*

Sorting in Linear Time

- Counting Sort
  - No comparisons between elements !
  - ***But***…depends on assumption about the numbers being sorted
- Basic Idea
  - For each input element x,
    determine the number of elements less than or equal to x
  - Assume that each element is an integer in the range 0 to k, for each integer i ($0 \leq i \leq k$), count how many elements whose values are i
    - Then we know how many elements are less than or equal to i
- Algorithm Storage
  - A[1..n]: input elements
  - B[1..n]: sorted elements
  - C[0..k]: hold the number of elements less than or equal to i

Counting Sort

```
COUNTING-SORT($A, B, k$)
let $C[0..k]$ be a new array
for $i = 0$ to $k$                                $\Theta(k)$
    $C[i] = 0$
for $j = 1$ to $A.length$                         $\Theta(n)$
    $C[A[j]] = C[A[j]] + 1$
// $C[i]$ now contains the number of elements equal to $i$.
for $i = 1$ to $k$                                $\Theta(k)$
    $C[i] = C[i] + C[i-1]$
// $C[i]$ now contains the number of elements less than or equal to $i$.
for $j = A.length$ downto 1                       $\Theta(n)$
    $B[C[A[j]]] = A[j]$
    $C[A[j]] = C[A[j]] - 1$
```

$$\Theta(n+k)$$

Counting Sort Illustration

(Range from 0 to 5)

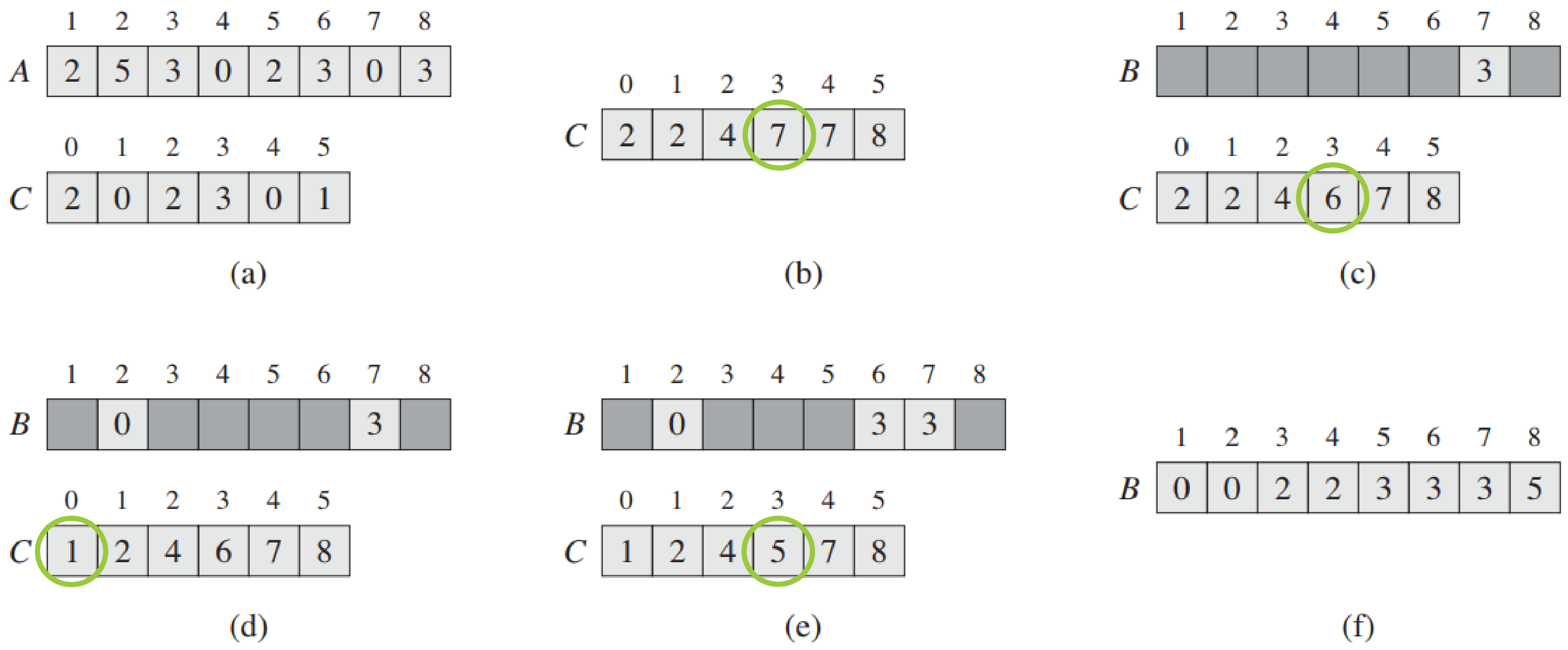


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort Illustration

A	2	5	3	0	2	3	0	3

B		0				3	3	
C	1	2	4	5	7	8		

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

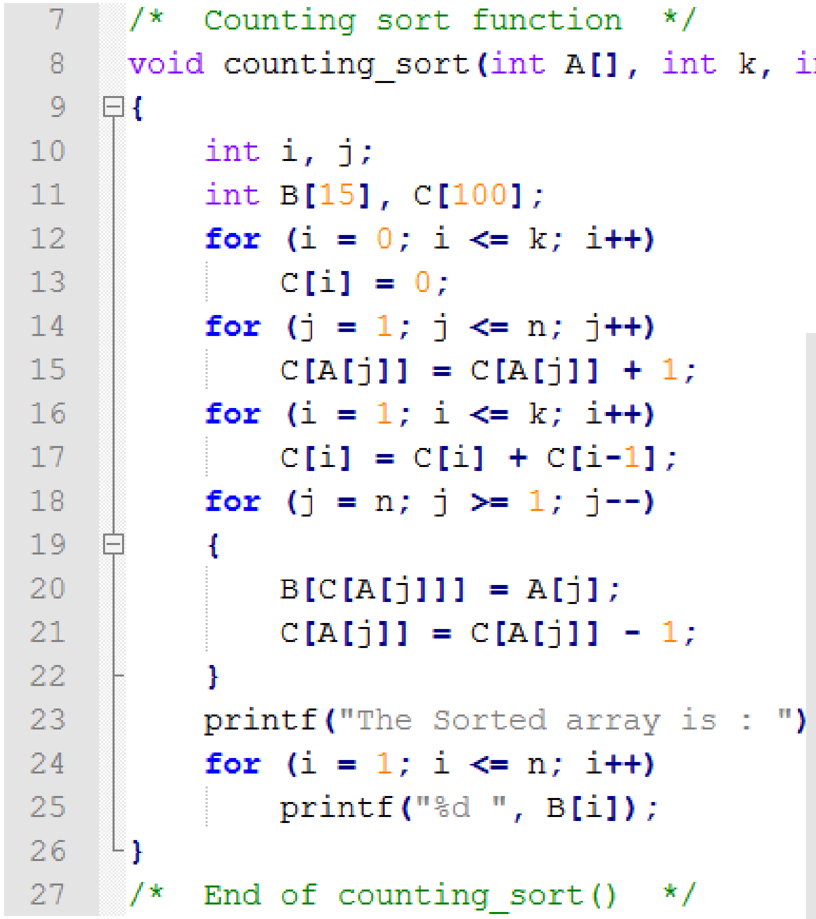
B	0	0		2		3	3	
C	0	2	3	5	7	8		

B	0	0		2	3	3	3	
C	0	2	3	4	7	8		

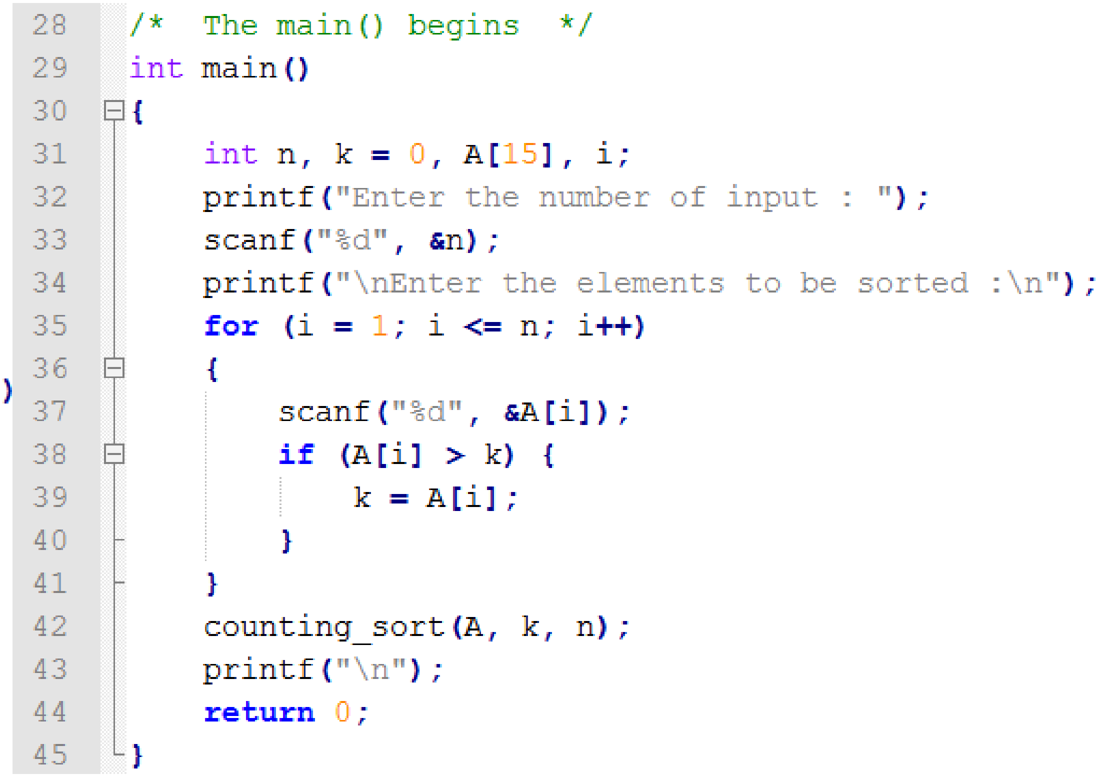
B	0	0		2	3	3	3	8
C	0	2	2	4	7	7		

B	0	0	2	2	3	3	3	8
C	0	2	3	4	7	7		

Counting Sort (1/4)



```c
/*  Counting sort function  */
void counting_sort(int A[], int k, int n)
{
    int i, j;
    int B[15], C[100];
    for (i = 0; i <= k; i++)
        C[i] = 0;
    for (j = 1; j <= n; j++)
        C[A[j]] = C[A[j]] + 1;
    for (i = 1; i <= k; i++)
        C[i] = C[i] + C[i-1];
    for (j = n; j >= 1; j--)
    {
        B[C[A[j]]] = A[j];
        C[A[j]] = C[A[j]] - 1;
    }
    printf("The Sorted array is : ")
    for (i = 1; i <= n; i++)
        printf("%d ", B[i]);
}
/*  End of counting_sort()  */
```



```c
/*  The main() begins  */
int main()
{
    int n, k = 0, A[15], i;
    printf("Enter the number of input : ");
    scanf("%d", &n);
    printf("\nEnter the elements to be sorted :\n");
    for (i = 1; i <= n; i++)
    {
        scanf("%d", &A[i]);
        if (A[i] > k) {
            k = A[i];
        }
    }
    counting_sort(A, k, n);
    printf("\n");
    return 0;
}
```

Counting Sort (2/4)

- Video Content
  - An illustration of Counting Sort.

Counting Sort (3/4)

Algorithm

Counting Sort

Counting Sort (4/4)

- Total time: $O(n + k)$
    - Usually, $k = O(n)$
    - Thus, counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$!
    - No contradiction
        - This is not a comparison sort
        - In fact, there are *no* comparisons at all !
    - Notice that this algorithm is ***stable***

Radix Sort

- Intuitively, you might sort on the **most significant digit**, then the second msd, etc.
- Problem
  - Lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea
  - Sort the *least* significant digit first

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

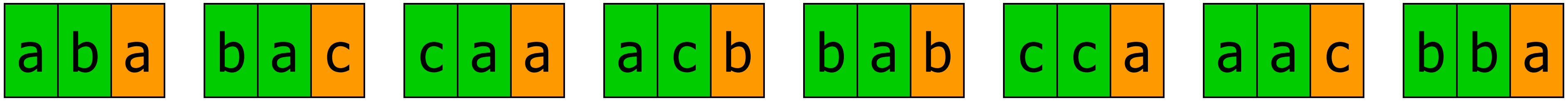
Radix Sort Example (1/10)

Input data:

a	b	a	b	a	c	c	a	a	a	c	b	b	a	b	c	c	a	a	a	c	b	b	a

Radix Sort Example (2/10)

Pass 1: Looking at rightmost position.

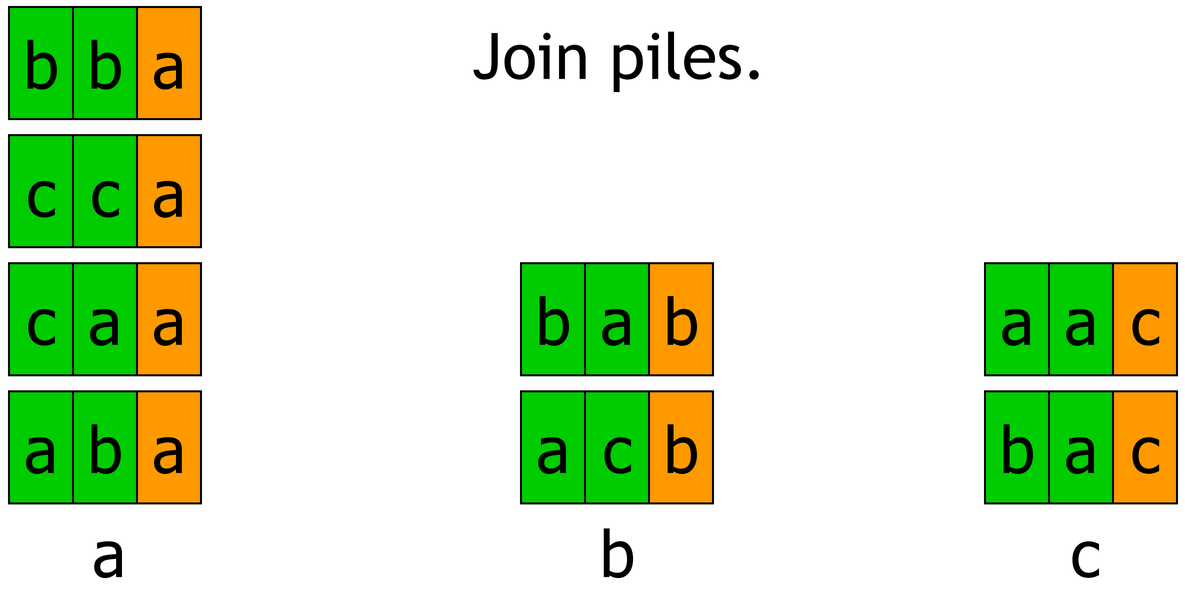


Place into appropriate pile.

Radix Sort Example (3/10)

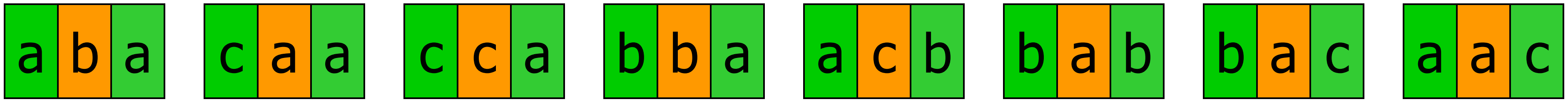
Pass 1: Looking at rightmost position.

Join piles.



Radix Sort Example (4/10)

Pass 2: Looking at next position.

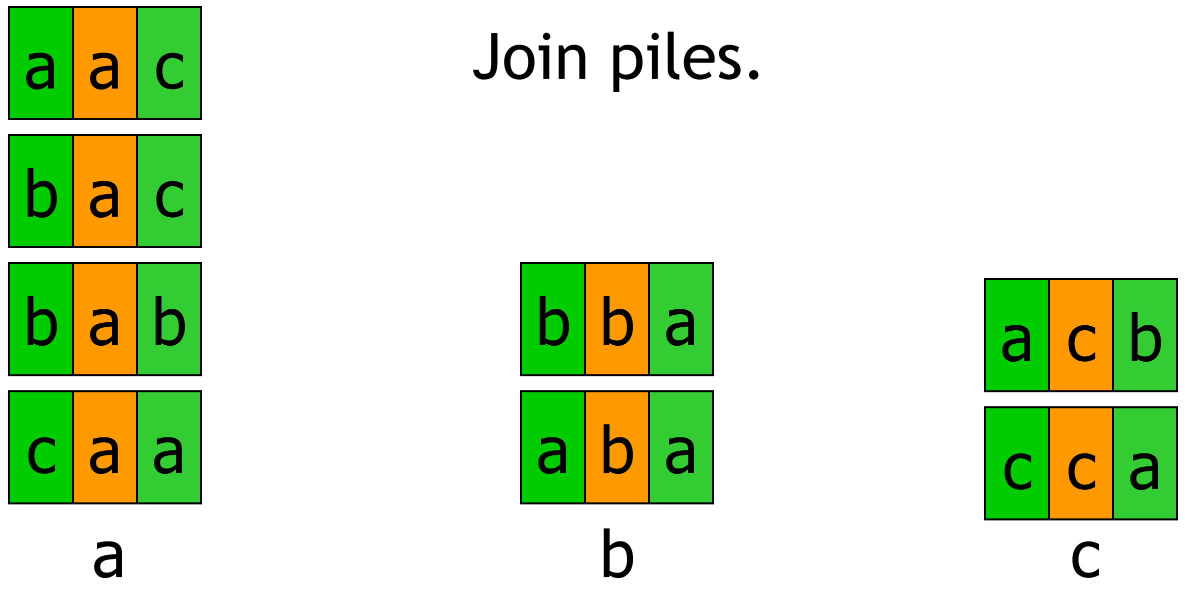


Place into appropriate pile.

Radix Sort Example (5/10)

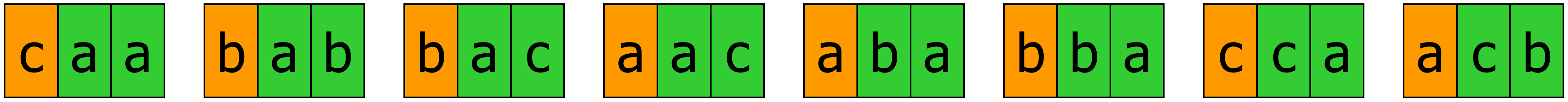
Pass 2: Looking at next position.

Join piles.



Radix Sort Example (6/10)

Pass 3: Looking at last position.



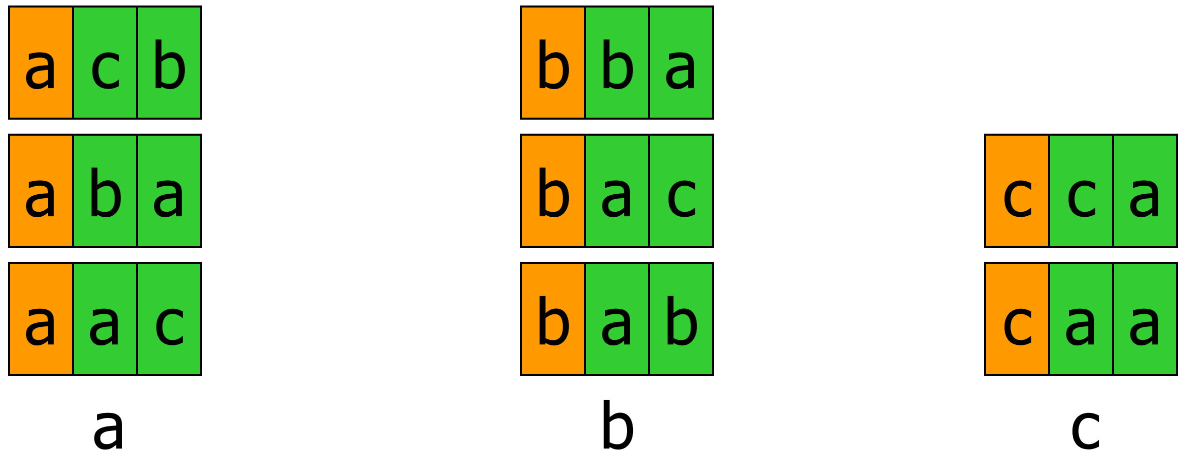
Place into appropriate pile.

a b c

Radix Sort Example (7/10)

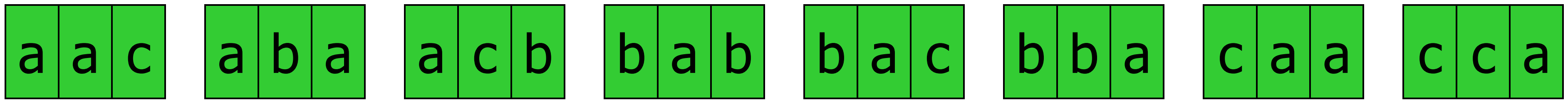
Pass 3: Looking at last position.

Join piles.



Radix Sort Example (8/10)

Result is sorted.

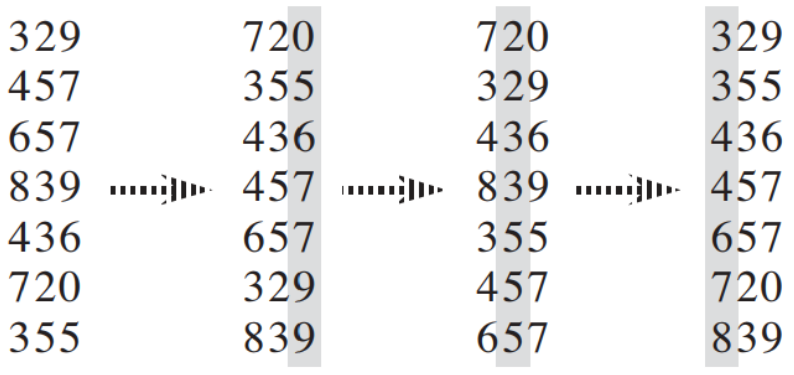


Radix Sort Example (9/10)

13 13	28	56
30 30	36	36
28 28	56	30
56 56	13	28
36 36	30	13

				36			36			
	30	13	28	13 30		56	56		28	
Radix Bins	0	1	2	3	4	5	6	7	8	9

Radix Sort Example (10/10)



329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

d digits in total

```
RADIX-SORT($A, d$)
1  for $i = 1$ to $d$
2      use a stable sort to sort array $A$ on digit $i$
```

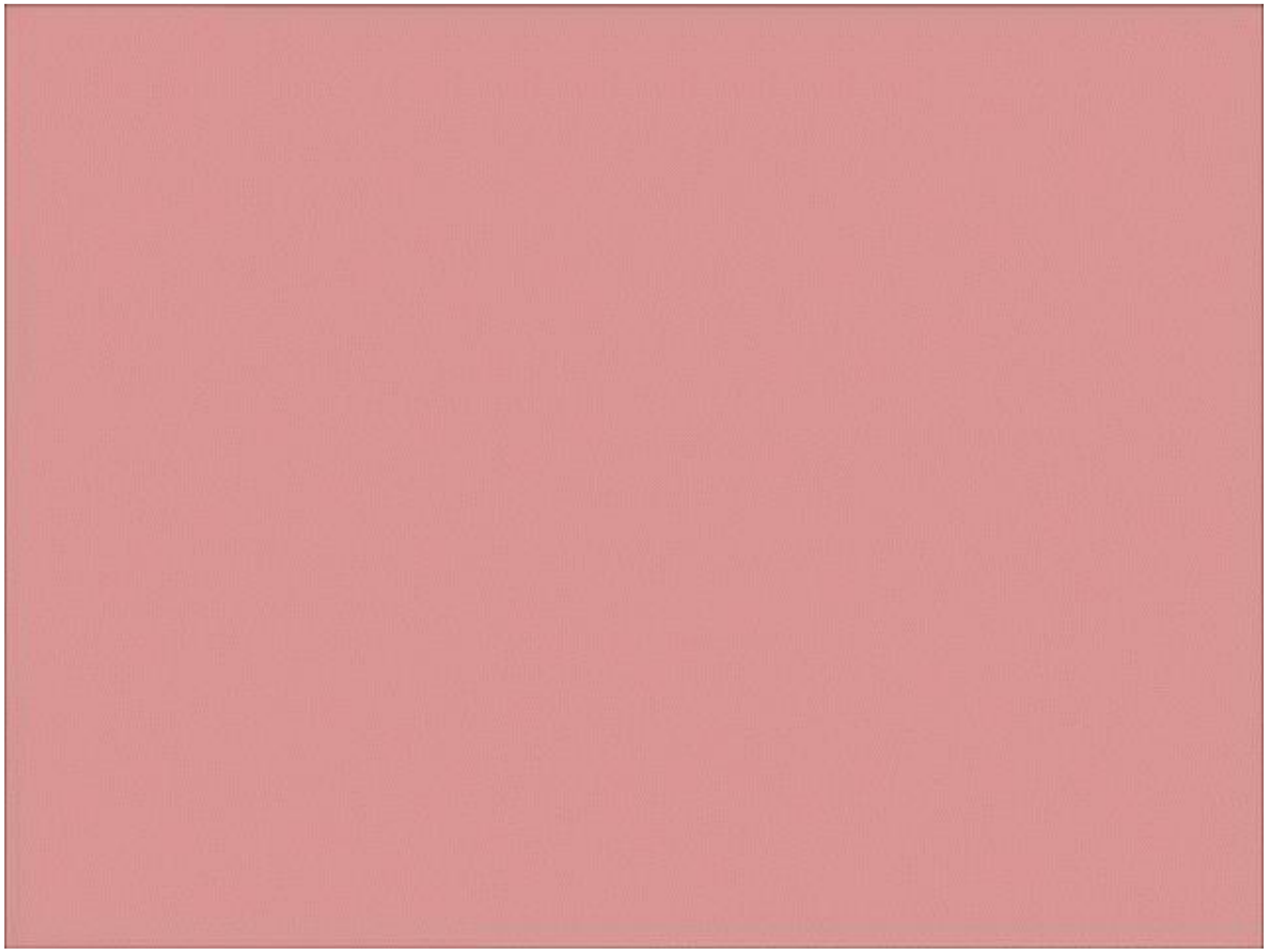
Radix Sort

- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over *n* numbers with *d* digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When *d* is constant and $k = O(n)$, takes $O(n)$ time
- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., $O(n)$)
  - Simple to code

Radix Sort

- Video Content
  - An illustration of Radix Sort.

Radix Sort



Practice Problems

- Show how to sort *n* integers in the range 0 to $n^3 - 1$ in $O(n)$ time.
  - Treat the numbers as 3-digit numbers in base *n*.
  - A number $0 \le x \le n^3 - 1$ can be represented in the base *n* as $\overline{abc}$: where $x = a \times n^2 + b \times n^1 + c \times n^0 (0 \le a, b, c \le n-1)$
  - Example: number $n^3 - 1$ can be represented as $\overline{(n-1)(n-1)(n-1)}$ because $(n-1) \times n^2 + (n-1) \times n^1 + (n-1) \times n^0 = n^3 - 1$

→ Sort these 3-digit numbers with radix sort → $O(n + n) = O(n)$.

Medians and Order Statistics

- The i^{th} order statistic of a set of n elements is the i^{th} smallest element
- The *median* is the halfway point
- Define the selection problem as:
  - Given a set A of n elements, find an element $x \in A$ that is larger than x-1 elements
- Obviously, this can be solved in $O(n \lg n)$. Why?
- Is it really necessary to sort *all* the numbers?

Order Statistics

- The i^{th} *order statistic* in a set of *n* elements is the i^{th} smallest element
- The *minimum* is thus the 1st order statistic
- The *maximum* is the n^{th} order statistic
- The *median* is the $n/2$ order statistic
  - If *n* is even, there are 2 medians
    - The lower median $\lfloor (n+1)/2 \rfloor$
    - The upper median $\lceil (n+1)/2 \rceil$
- *How can we calculate order statistics?*
- *What is the running time?*

Minimum and Maximum

```
MINIMUM($A, n$)
$min \leftarrow A[1]$
for $i \leftarrow 2$ to $n$
    do if $min > A[i]$
        then $min \leftarrow A[i]$
return $min$
```

- The maximum can be found in exactly the same way by replacing the > with < in the above algorithm.

Simultaneous Min and Max

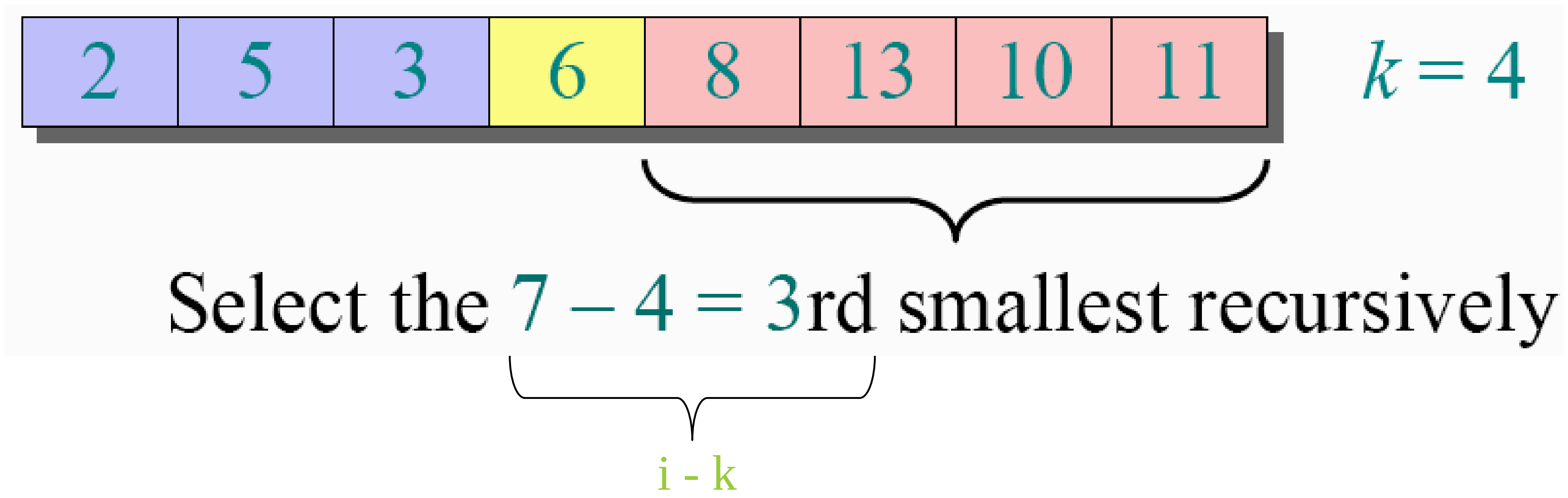
- How many comparisons are needed to find the minimum element in a set? The maximum?
- Can we find the minimum and maximum with less than twice the cost?
- Yes:
  - Walk through elements by pairs
    - Compare each element in pair to the other
    - Compare the largest to maximum, smallest to minimum
      - Pair (a_i, a_{i+1}), assume $a_i < a_{i+1}$, then
      - Compare (min, a_i), (a_{i+1}, max)
      - 3 comparisons for each pair.
  - Total cost: 3 comparisons per 2 elements
    - $O(3n/2)$

The Selection Problem

- A more interesting problem is *selection*
  - Finding the ith smallest element of a set
- We will show
  - A practical randomized algorithm with $O(n)$ expected running time
  - A cool algorithm of theoretical interest only with $O(n)$ worst-case running time

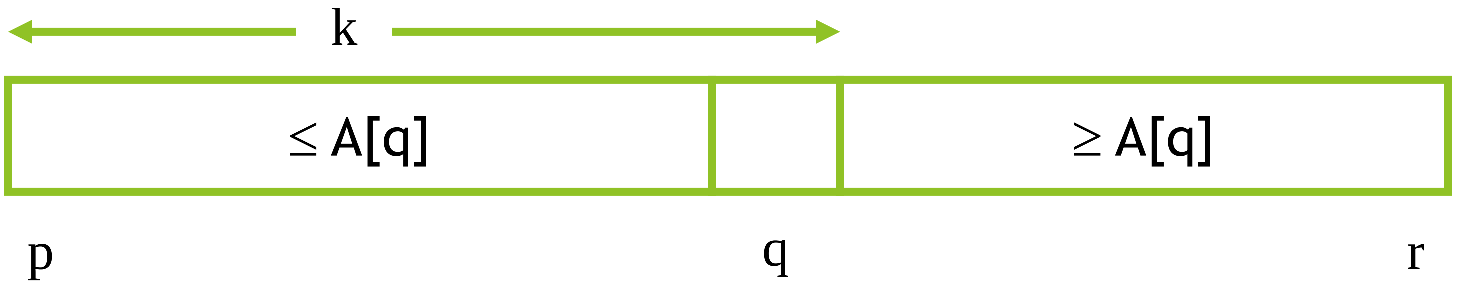
Randomized Selection (1/8)

- Key idea: use partition() from quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: $O(n)$
- Select the i=7th smallest



Randomized Selection (2/8)

```
RANDOMIZED-SELECT$(A, p, r, i)$
if $p = r$
   then return $A[p]$
$q \leftarrow$ RANDOMIZED-PARTITION$(A, p, r)$
$k \leftarrow q - p + 1$
if $i = k$            $\triangleright$ pivot value is the answer
   then return $A[q]$
elseif $i < k$
   then return RANDOMIZED-SELECT$(A, p, q - 1, i)$
else return RANDOMIZED-SELECT$(A, q + 1, r, i - k)$
```



Randomized Selection (3/8)

- Analyzing **`Randomized-Select()`**
  - Worst case: partition always 0:n-1

$$\begin{aligned} T(n) &= T(n-1) + O(n) \\ &= O(n^2) \qquad \text{(arithmetic series)} \end{aligned}$$

    - No better than sorting !
  - Best case: suppose a 9:1 partition

$$\begin{aligned} T(n) &= T(9n/10) + O(n) \\ &= O(n) \qquad \text{(Master Theorem, case 3)} \end{aligned}$$

    - Better than sorting!
    - *What if this had been a 99:1 split?*

Randomized Selection (4/8)

- Analysis of expected time
  - The analysis follows that of randomized quicksort, but it's a little different.
  - Let $T(n)$ be the random variable for the running time of **`Randomized-Select`** on an input of size *n*, assuming random numbers are independent.
  - For $k = 0, 1, \ldots, n-1$, define the ***indicator random variable***

$$X_k = \begin{cases} 1 & \text{if } \textsc{Partition} \text{ generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

Randomized Selection (5/8)

- To obtain an upper bound, assume that the ith element always falls in the larger side of the partition:

$$T(n) = \begin{cases} T(\max\{0, n-1\}) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(\max\{1, n-2\}) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(\max\{n-1, 0\}) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right).$$

Randomized Selection (6/8)

- Take expectations of both sides
  - $$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(\max\{k, n-k-1\}) + \Theta(n)\right)\right]$$
  - $$T(n) \le \frac{1}{n}\sum_{k=0}^{n-1} T(\max(k, n-k-1)) + \Theta(n)$$
    $$\le \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$
- Let's show that $T(n) = O(n)$ by substitution

Randomized Selection (7/8)

- Assume $T(k) \leq ck$ for sufficiently large c:

$$T(n) \leq \frac{2}{n}\sum_{k=n/2}^{n-1} T(k) + \Theta(n)$$ *The recurrence we started with*

$$\leq \frac{2}{n}\sum_{k=n/2}^{n-1} ck + \Theta(n)$$ *Substitute $T(n) \leq cn$ for $T(k)$*

$$= \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k\right) + \Theta(n)$$ *"Split" the recurrence*

$$= \frac{2c}{n}\left(\frac{1}{2}(n-1)n - \frac{1}{2}\left(\frac{n}{2}-1\right)\frac{n}{2}\right) + \Theta(n)$$ *Expand arithmetic series*

$$= c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n)$$ *Multiply it out*

Randomized Selection (8/8)

- Assume $T(k) \leq ck$ for sufficiently large c:

$$\begin{aligned}
T(n) &\leq c(n-1) - \frac{c}{2}\left(\frac{n}{2}-1\right) + \Theta(n) && \textit{The recurrence so far} \\
&= cn - c - \frac{cn}{4} + \frac{c}{2} + \Theta(n) && \textit{Multiply it out} \\
&= cn - \frac{cn}{4} - \frac{c}{2} + \Theta(n) && \textit{Subtract c/2} \\
&= cn - \left(\frac{cn}{4} + \frac{c}{2} - \Theta(n)\right) && \textit{Rearrange the arithmetic} \\
&\leq cn \quad \text{(if c is big enough)} && \textit{What we set out to prove}
\end{aligned}$$

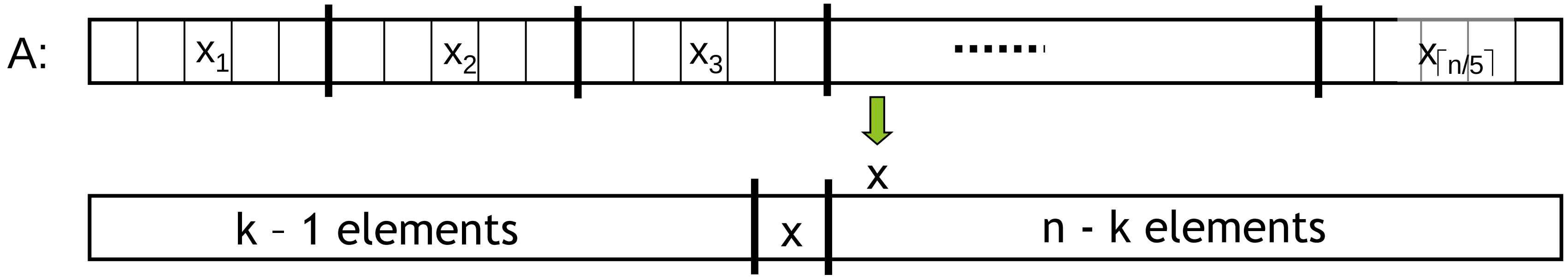
Selection in Worst-Case Linear Time

- Randomized algorithm works well in practice
- What follows is a worst-case linear time algorithm, really of theoretical interest only
- Basic idea:
  - Generate a good partitioning element
  - Call this element *x*

Selection in Worst-Case Linear Time

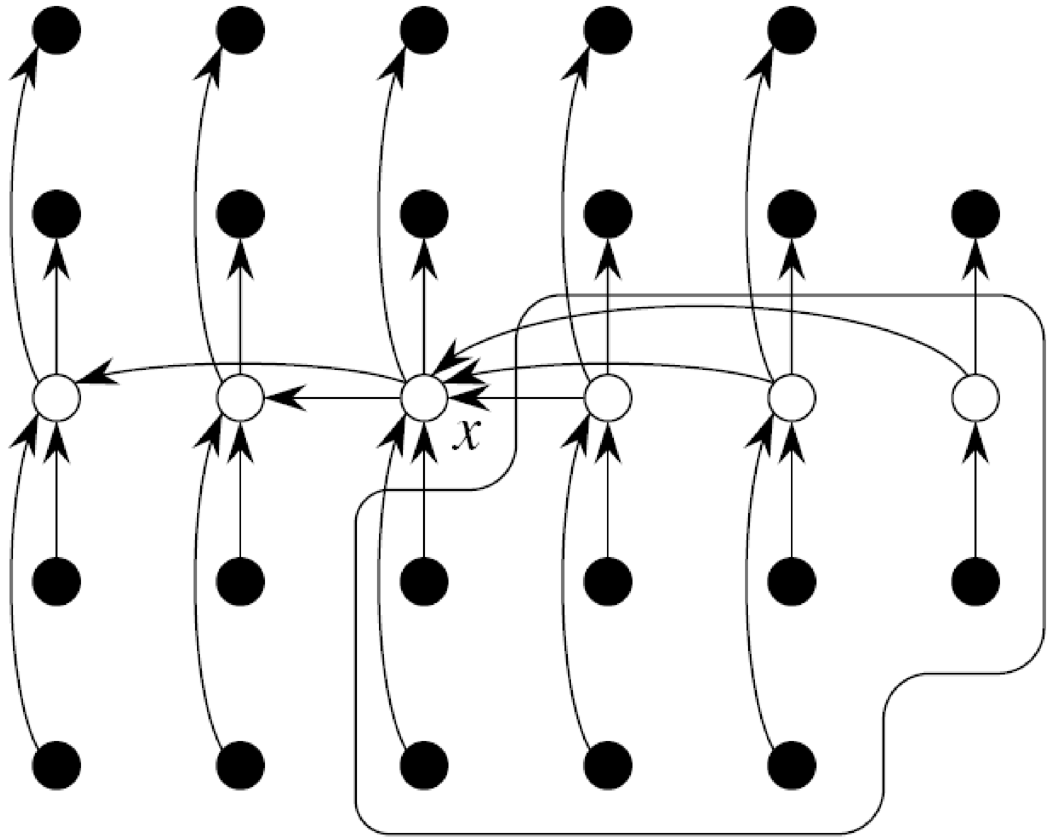
- The algorithm in words:
  1. Divide *n* elements into groups of 5
  2. Find median of each group (*How? How long?*)
  3. Use Select() recursively to find median *x* of the $\lfloor n/5 \rfloor$ medians
  4. Partition the *n* elements around *x*. Let $k = rank(x)$
  5. **if** (i == k) **then** return x

     if (i < k) **then** use Select() recursively to find *i*th smallest element in first partition
     else (i > k) use Select() recursively to find (*i*-*k*)th smallest element in last partition



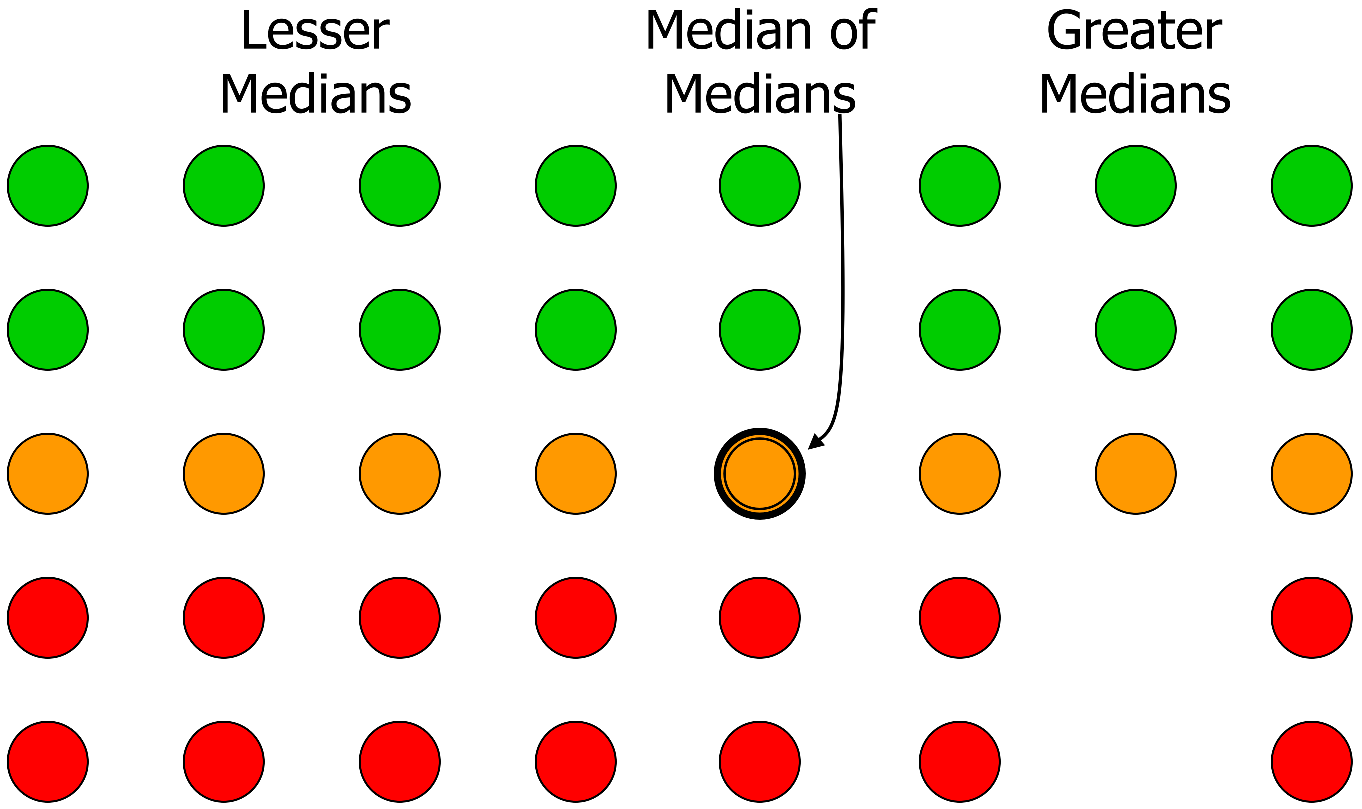
Order Statistics Analysis

🟢 🟢	Lesser Elements
🟠	Median
🔴 🔴	Greater Elements



One group of 5 elements

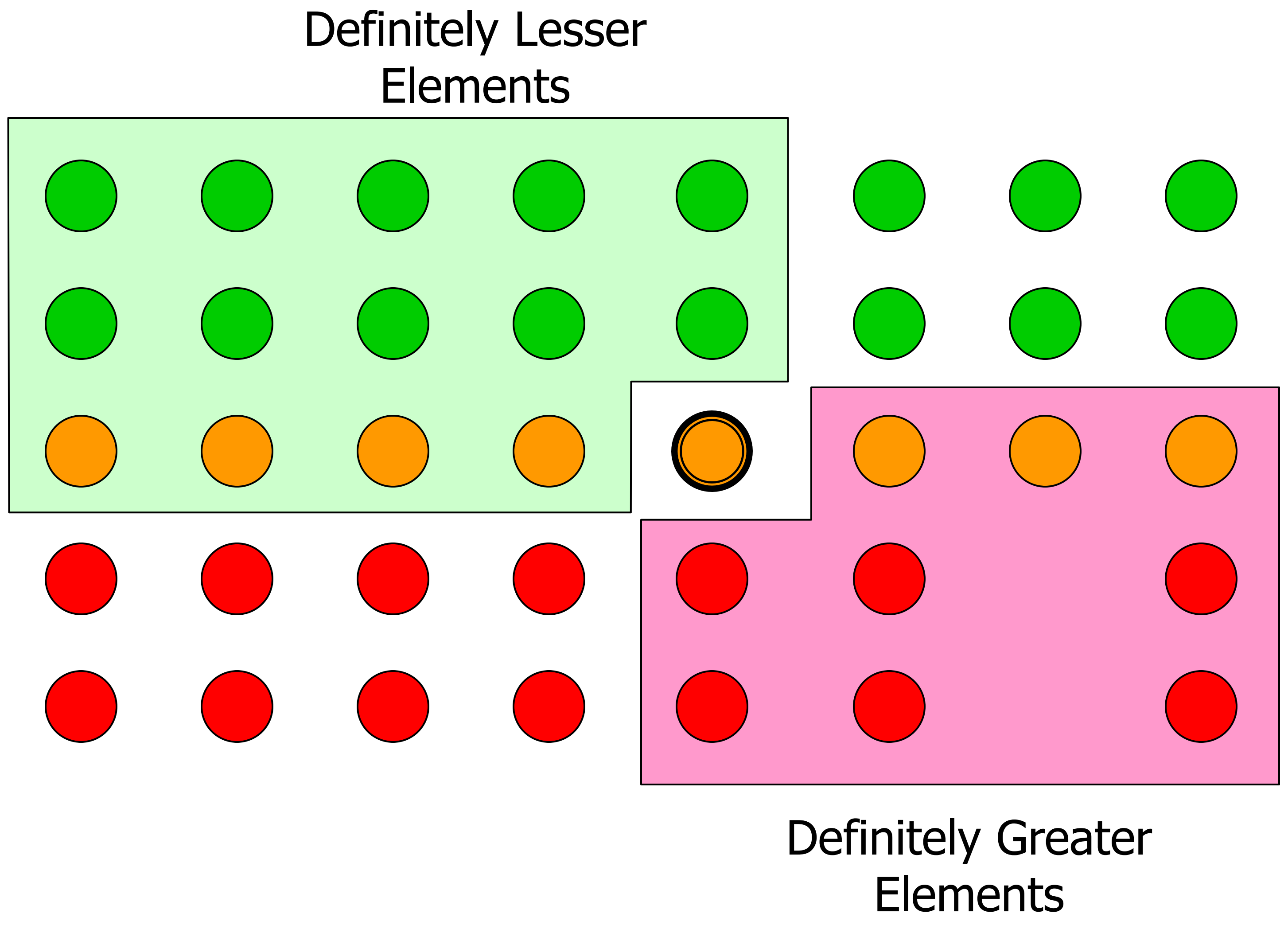
Order Statistics Analysis



All groups of 5 elements

(And at most one smaller group)

Order Statistics Analysis



Example

- Find the 11th smallest element in array:
  - A = {12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27, 34, 2 ,19 ,12 ,5 ,18 ,20 ,33, 16, 33, 21, 30, 3, 47}
- Divide the array into groups of 5 elements

12	4	43	2	20	30
34	17	82	19	33	3
0	32	25	12	16	47
3	3	27	5	33	
22	28	34	18	21	

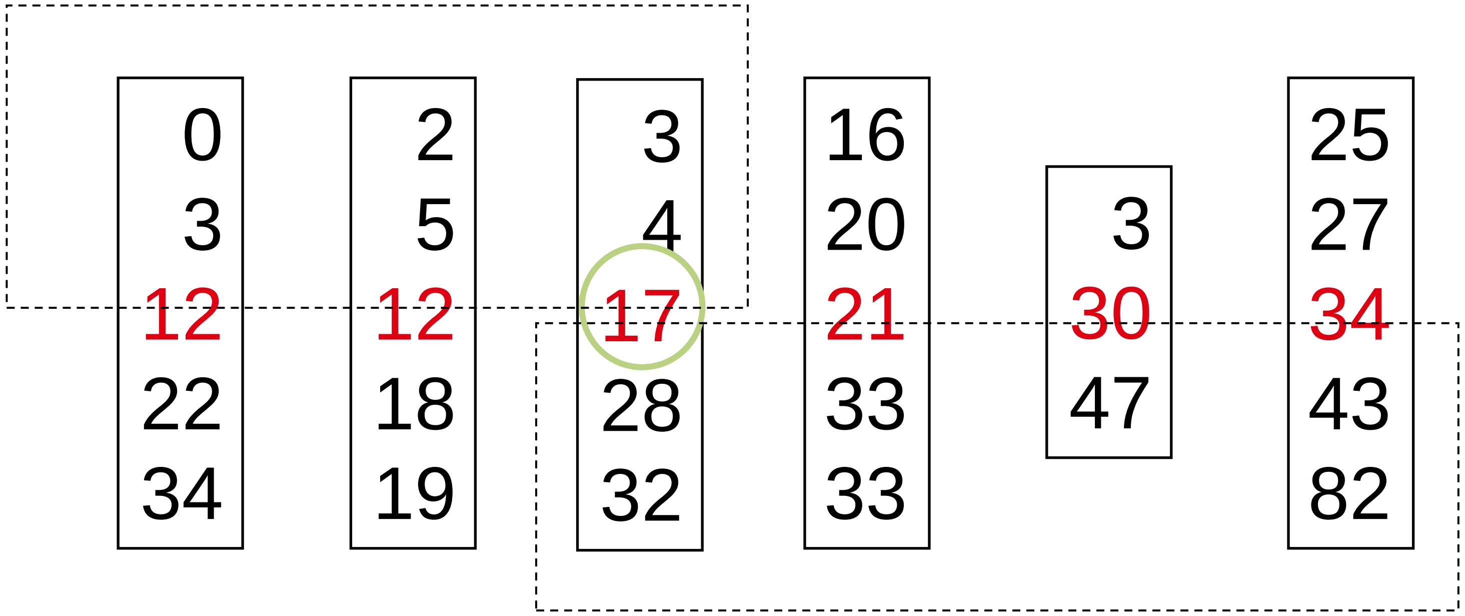
Example

- Sort the groups and find their medians

0	3	25	2	16	3
3	4	27	5	20	30
12	17	34	12	21	47
22	28	43	18	33	
34	32	82	19	33	

Example

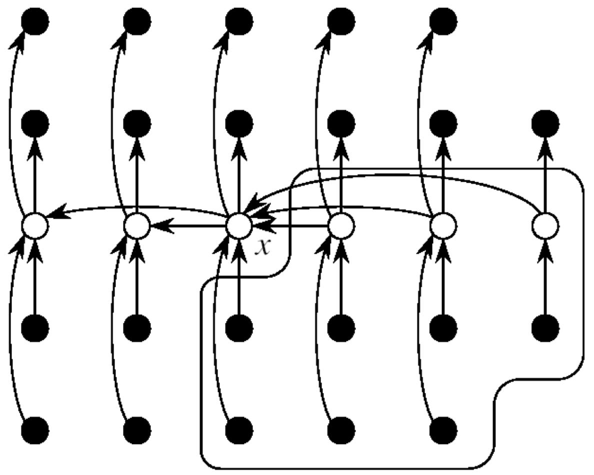
- Find the median of the medians



Example

- Partition the array around the median of medians (17)
  - First partition:
    - {12, 0, 3, 4, 3, 2, 12, 5, 16, 3}
  - Pivot:
    - 17 (position of the pivot is q = 11)
  - Second partition:
    - {34, 22, 32, 28, 43, 82, 25, 27, 34, 19, 18, 20, 33, 33, 21, 30, 47}
- To find the 6th smallest element we would have to recur our search in the first partition

Analysis of Running Time (1/3)



- At least half of the medians found in step 2 are ≥ x
- All but two of these groups contribute 3 elements > x

$\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2$ groups with 3 elements > x

- At least $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil\right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements greater than x
- SELECT is called on at most $n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$ elements

Analysis of Running Time (2/3)

- Step 1: making groups of 5 elements takes
  - $O(n)$
- Step 2: sorting $n/5$ groups in $O(1)$ time each takes
  - $O(n)$
- Step 3: calling SELECT on $\lceil n/5 \rceil$ medians takes time
  - $T(\lceil n/5 \rceil)$
- Step 4: partitioning the n-element array around x takes
  - $O(n)$
- Step 5: recursing on one partition takes
  - $\leq T(7n/10 + 6)$
- $T(n) = T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
  - Show that $T(n) = O(n)$

Analysis of Running Time (3/3)

- Let's show that $T(n) = O(n)$ by substitution

$$\begin{aligned}
T(n) &\le c\lceil n/5\rceil + c\,(7n/10 + 7) + an \\
&\le cn/5 + 7cn/10 + 7c + an \\
&= 9cn/10 + 7c + an \\
&= cn + (-cn/10 + 7c + an) \\
&\le cn \quad if\!: -cn/10 + 7c + an \le 0
\end{aligned}$$

Thanks to contributors

Mr. Pham Van Nguyen (2022)

Mr. Bui Phuoc Nguyen (2022)

Dr. Dang Thien Binh (2017 – 2022)

Prof. Hyunseung Choo (2017 – 2022)