COT 6405 Introduction to Theory of Algorithms

Topic 8. Quicksort

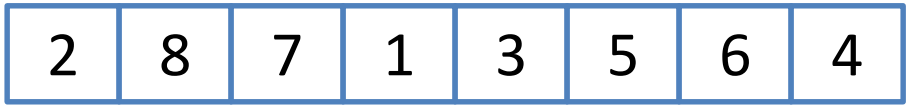
Quicksort

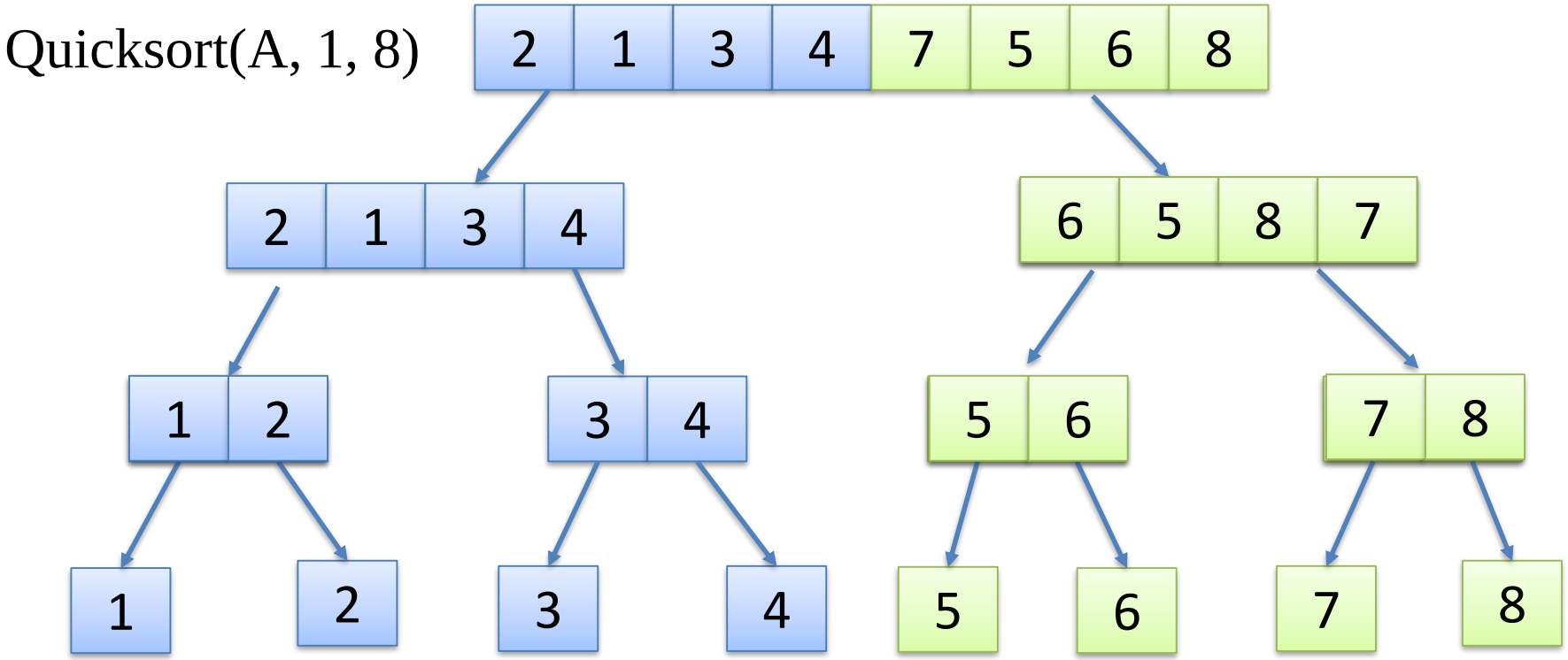
- Sorts “in place”
  - Only a constant number of elements stored outside the sorted array
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
- So why people use it instead of merge sort?
  - Merge sort does not sort “in place”

Quicksort: divide and conquer

- **Divide**: Array A[p..r] is partitioned into two non-empty subarrays A[p..q] and A[q+1..r]
  - All elements in A[p..q] are less than all elements in A[q+1..r]
- **Conquer:** The subarrays are recursively sorted by calls to quicksort
- **Combine:** No work is needed to combine the subarrays, because they are sorted in place.

An example of Quicksort





Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}  // what is the initial call?
```

Partition

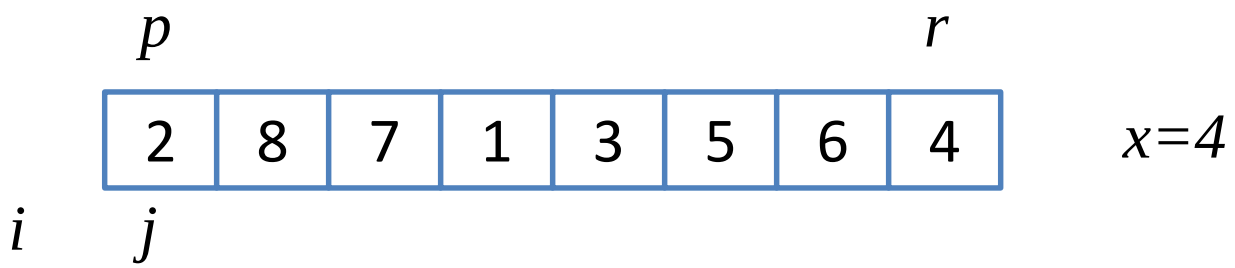
- Clearly, all the actions take place in the **`partition()`** function
  - Rearranges the subarray “in place”
  - End result:
    - Two subarrays
    - All values in 1st subarray < all values in 2nd
  - Returns the index of the “pivot” element separating the two subarrays
- How do we implement this function?

Partition array $A[p..r]$

```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        // move the pivot between the two subarraies
        exchange A[i + 1] ↔ A[r]
        // return the pivot
        return i + 1
```

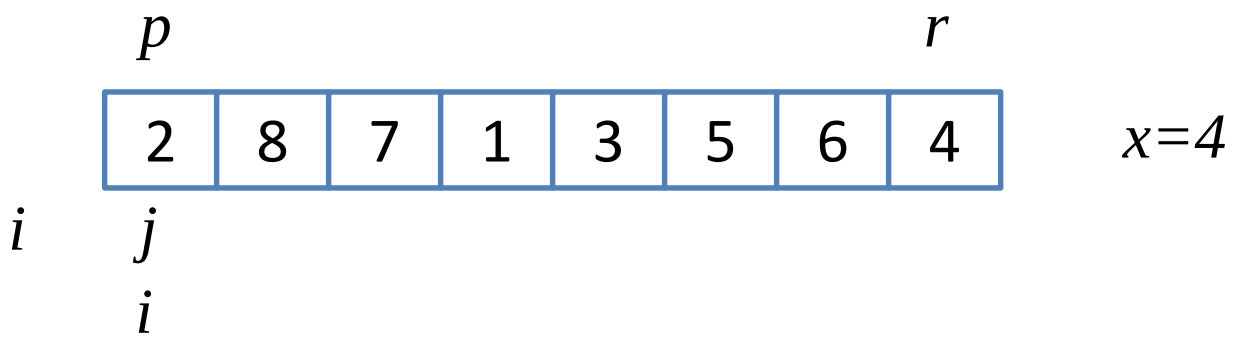
What is the running time of `partition()`?

An example of Partition



PARTITION(*A*, *p*, *r*)
$x \leftarrow A[r]$ // select the pivot
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
if $A[j] \leq x$
$i \leftarrow i + 1$
exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

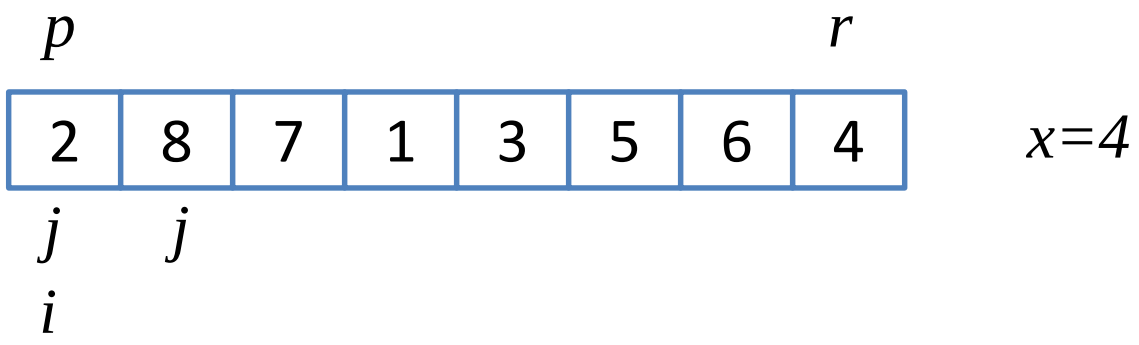
An example of Partition (cont'd)



PARTITION(*A*, *p*, *r*)

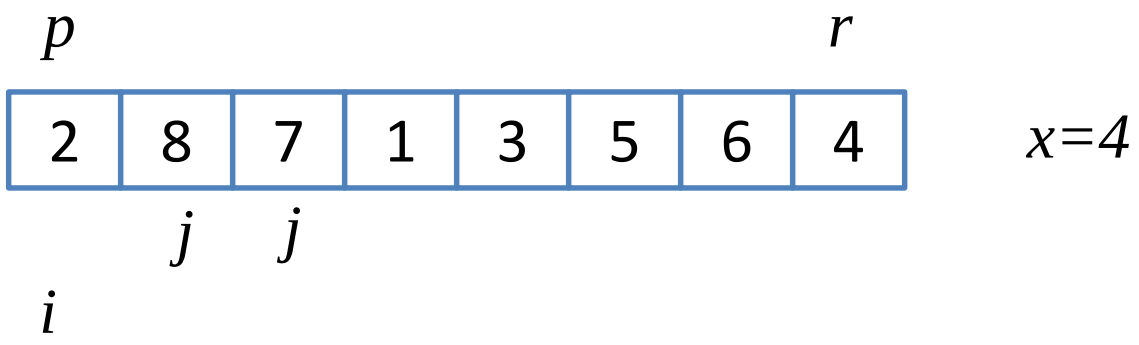
```
x ← A[r]          // select the pivot
i ← p − 1
for j ← p to r − 1
        if A[ j ] ≤ x
                i ← i + 1
                exchange A[i ] ↔ A[ j ]
exchange A[i + 1] ↔ A[r]
return i + 1
```

An example of Partition (cont'd)



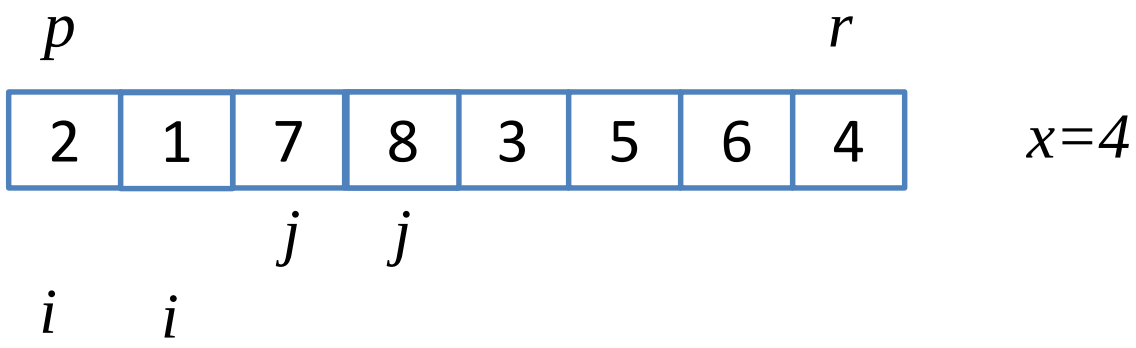

```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        exchange A[i + 1] ↔ A[r]
        return i + 1
```

An example of Partition (cont'd)




```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        exchange A[i + 1] ↔ A[r]
        return i + 1
```

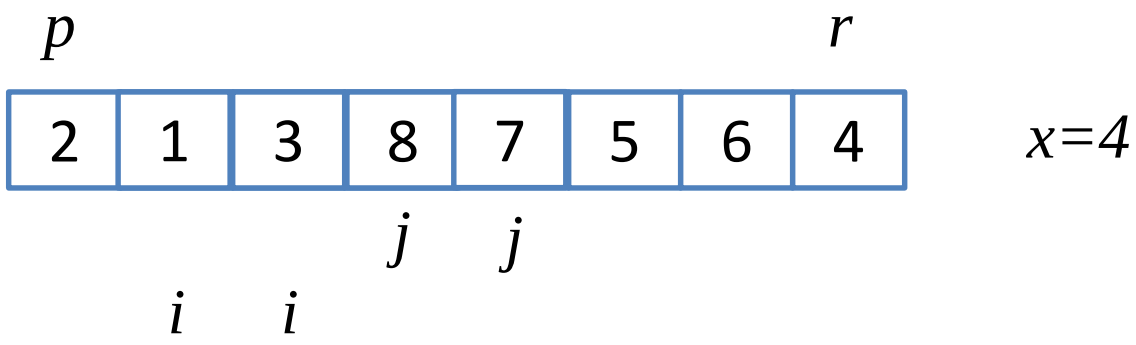
An example of Partition (cont'd)



PARTITION(*A*, *p*, *r*)

```
x ← A[r]          // select the pivot
i ← p − 1
for j ← p to r − 1
        if A[ j ] ≤ x
                i ← i + 1
                exchange A[i ] ↔ A[ j ]
exchange A[i + 1] ↔ A[r]
return i + 1
```

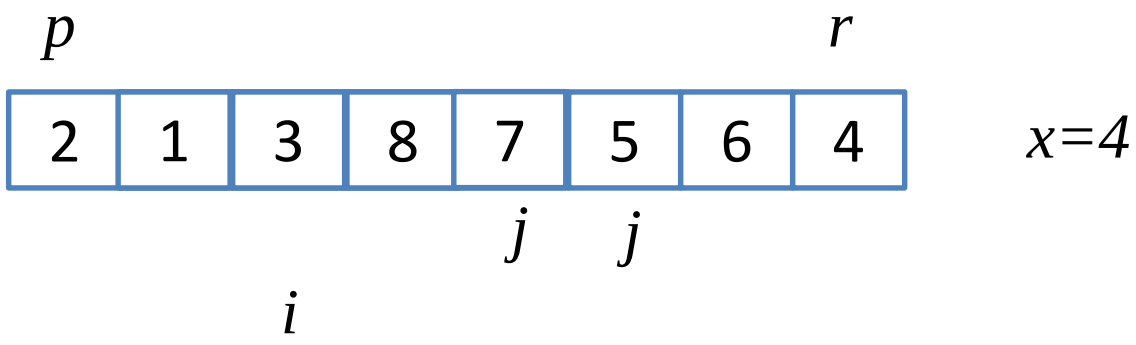
An example of Partition (cont'd)



PARTITION(*A*, *p*, *r*)

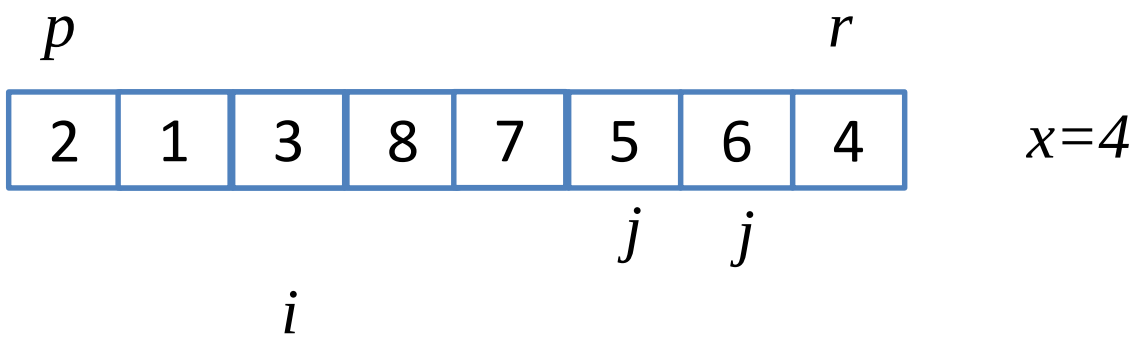
```
x ← A[r]          // select the pivot
i ← p − 1
for j ← p to r − 1
        if A[ j ] ≤ x
                i ← i + 1
                exchange A[i ] ↔ A[ j ]
exchange A[i + 1] ↔ A[r]
return i + 1
```

An example of Partition (cont'd)




```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        exchange A[i + 1] ↔ A[r]
        return i + 1
```

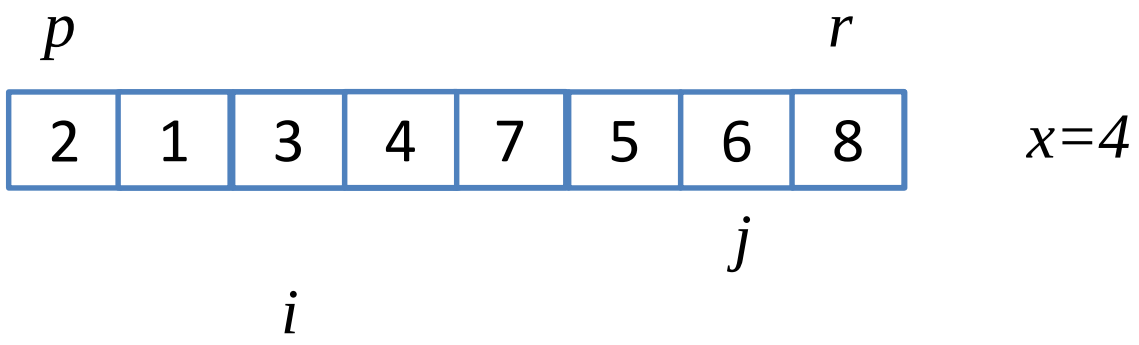
An example of Partition (cont'd)



PARTITION(*A*, *p*, *r*)

```
x ← A[r]          // select the pivot
i ← p − 1
for j ← p to r − 1
        if A[ j ] ≤ x
                i ← i + 1
                exchange A[i ] ↔ A[ j ]
exchange A[i + 1] ↔ A[r]
return i + 1
```

An example of Partition (cont'd)




```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        exchange A[i + 1] ↔ A[r]
        return i + 1
```

Partitioning

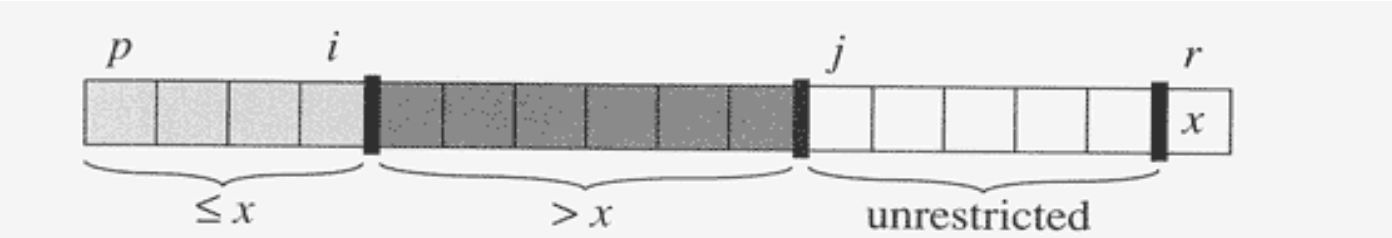
- PARTITION first selects the **pivot** (How?)
  - the last element $A[r]$ in the subarray $A[p \,..\, r]$
- The array is partitioned into four regions
  - some of which may be empty
- **Loop invariant:**

  1. All entries in $A[p \,..\, i]$ are ≤ pivot.

  2. All entries in $A[i + 1 \,..\, j - 1]$ are > pivot.

  3. $A[r]$ = pivot.

  4. It's not needed as part of the loop invariant, but the fourth region is $A[\, j \,..\, r-1]$, whose entries have not yet been examined.



Analyzing Quicksort

- Worst-case performance
  - The worst-case behavior for quicksort occurs when the partitioning routine produces with n-1 elements and one with 0 elements
- The recurrence is
  - $T(n) = T(n-1) + T(0) + \Theta(n)$

    $= T(n-1) + \Theta(n)$

Exercise

- For T(n) = T(n-1) + $\Theta(n)$, use substitution method to show that the $\mathrm{T(n)} = \Theta(n^2)$.

Exercise (cont'd)

- T(n) = T(n-1) + $\Theta(n)$
- Basis: n = 1, T(1) = Θ (1)
  Inductive step: suppose T(k) $\leq$ ck^2 for all k < n, then

  T(n) $\leq$ c$(n-1)^2$ + $c'n$

  $=$ cn^2- 2cn + c +$c'n$

  = cn^2 - (2c- c')n + c

  $\leq$ cn^2 - (2c- c')n + cn (n > 1)

  $\leq$ cn^2 when -(2c-c')n + cn $\leq$ 0 - > n_0= 1 and c' $\leq$ c

  Thus, T(n) = O(n^2)

Exercise (cont'd)

- T(n) = T(n-1) + $\Theta(n)$
- Basis: n = 1, T(1) = Θ (1)
  Inductive step: suppose T(k) $\geq$ ck^2 for all k < n, then

  T(n) $\geq$ c$(n-1)^2$ + $c'n$

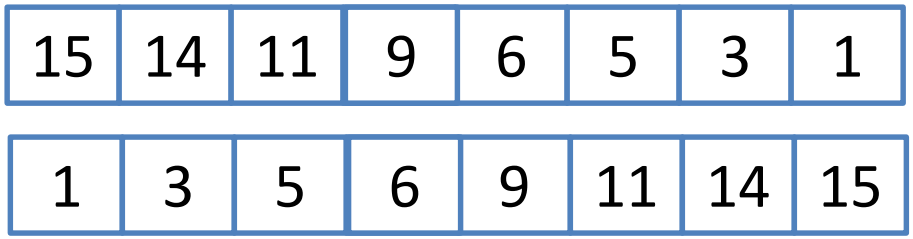
  $\geq$ cn^2- 2cn + c +$c'n$

  $\geq$ cn^2 – 2cn +$c'n$

  $\geq$ cn^2 if -2cn + $c'n \geq 0$ (c $\leq c'/2$)

A question

- Will any particular input elicit the worst case?
  - Yes, the array is already sorted in the reverse order
  - Or it is already sorted




```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        exchange A[i + 1] ↔ A[r]
        return i + 1
```

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
```

Best-case performance

- The best-case behavior occurs when Partition() produces two sub-problems of equal size, the total size of two sub-problems is n-1.
- The recurrence for the running time is
  - T(n) = 2T(n-1/2) + $\Theta(n)$
  - By case 2 of the master theorem, T(n) = Θ(nlgn)

Performance of quicksort

- The running time of quicksort depends on the partitioning of the subarrays:
  - If the subarrays are balanced, then quicksort can run as fast as mergesort.
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.

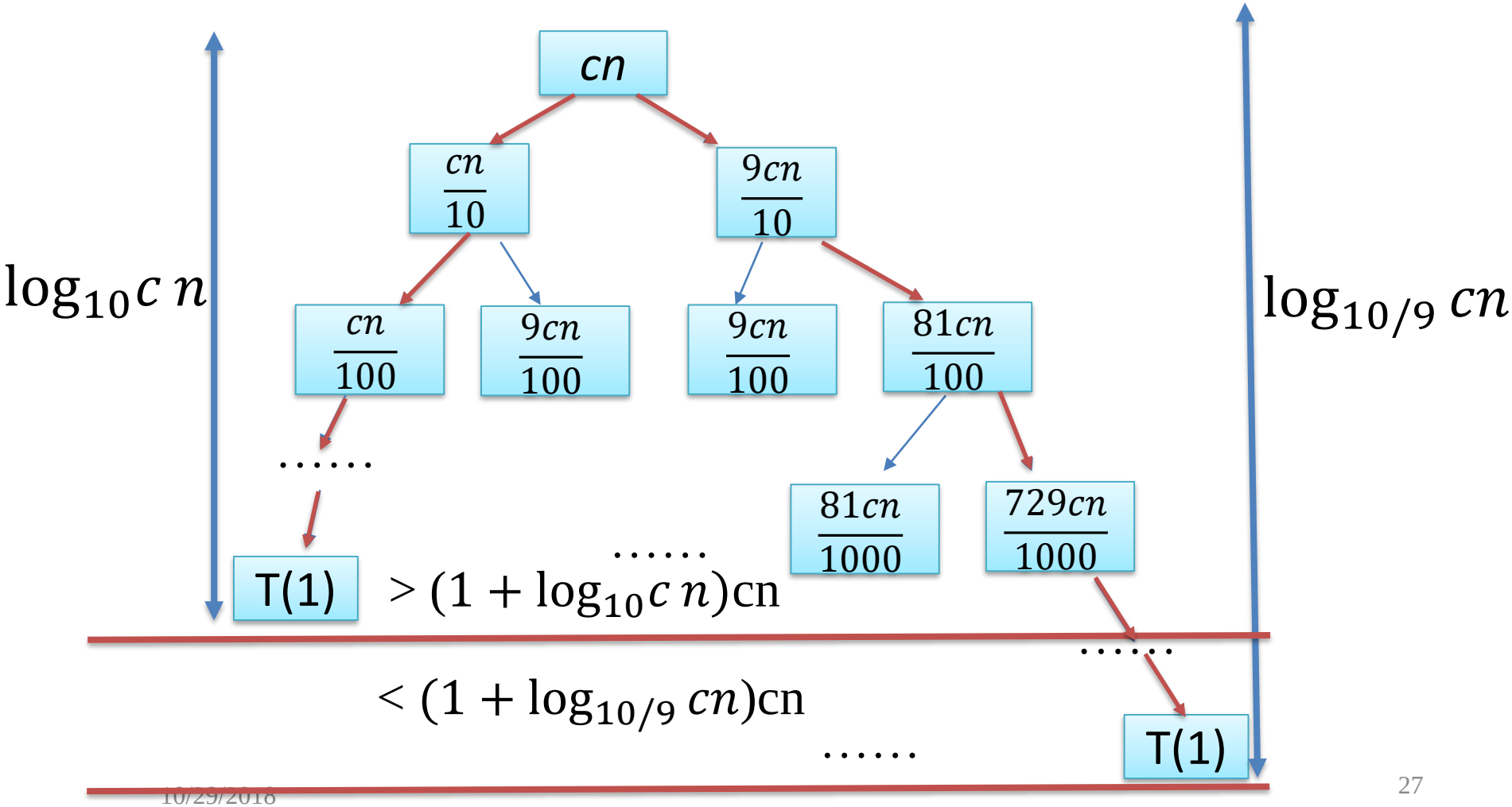
Analyzing Quicksort: Average Case

- Assuming random input → average-case running time is much closer to O(n lg n) than $O(n^2)$
- First, a more intuitive explanation/example:
  - Suppose that partition() always produces a 9-to-1 split. This looks quite unbalanced!
  - The recurrence is thus:

    $T(n) = T(9n/10) + T(n/10) + cn$
  - How deep will the recursion go? (draw it)

Average Case (cont'd)

- T(n) = T(9n/10) + T(n/10) + cn



Average Case (cont'd)

- For shortest path for the root to the leaf
  - The subproblem size for a node at depth *i* is $(\frac{1}{10})^i$cn
  - The subproblem size hits T(1), when $(\frac{1}{10})^i$cn= 1, or $i = \log_{10} cn$
  - Thus, the length of the shortest path is $\log_{10} cn$

Average Case (cont'd)

- For longest path for the root to the leaf
  - The subproblem size for a node at depth *i* is $(\frac{9}{10})^i$cn
  - The subproblem size hits T(1), when$(\frac{9}{10})^i$cn= 1, or $i = \log_{10/9} cn$
  - Thus, the length of the longest path is $\log_{10/9} cn$

Average Case (cont'd)

- Notice that every level of the tree has a cost of cn , until the recursion reaches a boundary condition at depth $\log_{10} cn = \Theta(lgn)$
- Then, the levels have cost at most cn
- The recursion terminates at depth $\log_{10/9} n = \Theta(lgn)$

Average Case (cont'd)

- The total cost of quicksort T(n)

  T(n) $> (1 + \log_{10} cn)$cn -> Ω (nlgn)

  T(n) $< (1 + \log_{10/9} cn)$cn -> O(nlgn)

  T(n) = $\Theta(\text{nlgn})$

Analyzing Quicksort: Average Case

- Intuitively, a real-life run of quicksort will produce a mix of "bad" and "good" splits
  - Randomly distributed among the recursion tree
  - Pretend for intuition that they alternate between best-case (n-1/2 : n-1/2) and worst-case (n-1 : 0)
  - What happens if we bad-split root node, then good-split the resulting size (n-1) node?
    - We end up with 3 subarrays, size 0, (n-1)/2-1, (n-1)/2
    - Combined cost of splits = n + n -1 = 2n -1 = Θ(n)
    - No worse than if we had good-split the root node!
      - Good-split: T(n) = 2T(n-1/2) + Θ(n)
      - Mix-split: T(n) = T(0) + T(n-1/2) + T((n-1)/2-1)+ Θ(n)

        <= 2T(n-1/2) + Θ(n) -> good split complexity

Partition cost in Elliptical Shading

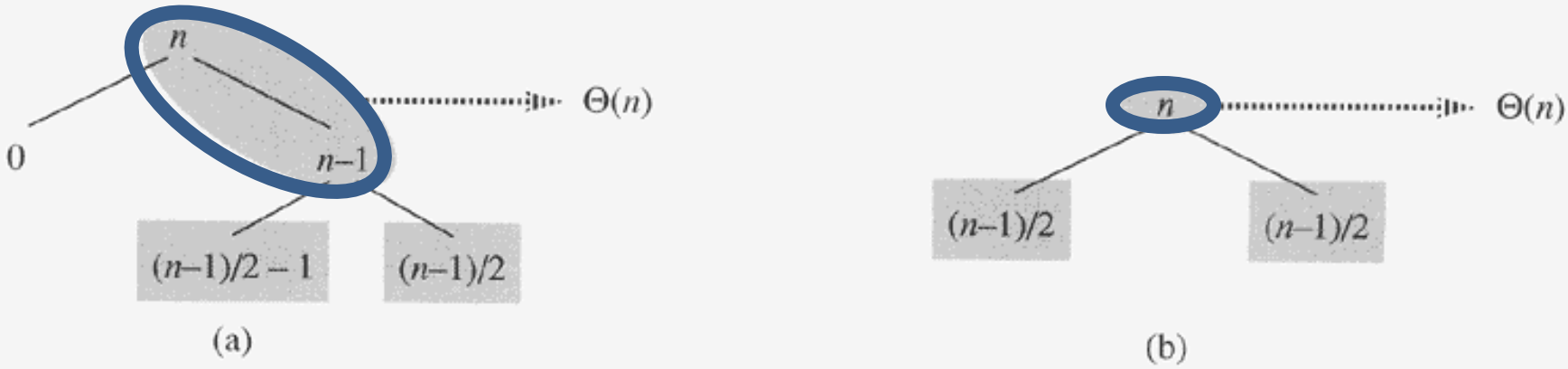


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

Analyzing Quicksort: Average case

- Intuitively, the O(n) cost of a bad split (or 2 or 3 bad splits) can be absorbed into the O(n) cost of each good split
- Thus running time of alternating bad and good splits is still O(n lg n), with slightly higher constants
- How can we be more rigorous?

Analyzing Quicksort: Average case

- For simplicity, assume:
  - All inputs distinct (no repeats)
- Partition around a random element
    - all splits (0:n-1, 1:n-2, 2:n-3, … , n-1:0) are equally likely
    - In general, a split can be represented by (k : n-1-k)
- What is the probability of a particular split happening?
- Answer: 1/n

Analyzing Quicksort: Average case

- So partition generates splits
  (0:n-1, 1:n-2, 2:n-3, … , n-2:1, n-1:0)
  each with probability 1/n
- T(n) is the expected running time, T(n) = ?

$$T(n) = \frac{1}{n}\sum_{k=0}^{k=n-1} T(k) + T(n-1-k) + \Theta(n)$$

- What is each term under the summation for?
- What is the Θ(n) term for?

Average case

- T(n) = $\frac{1}{n}\sum_{k=0}^{k=n-1} T(k) + T(n-1-k) + \Theta(n)$
- We can rewrite the above equation as
- T(n) = $\frac{2}{n}\sum_{k=0}^{k=n-1} T(k) + \Theta(n)$

Why?

- T(n) =$\frac{1}{n}$(T(0)+T(n-1)+T(1)+T(n-2)+…+ T(n-1)+T(0))

Average case (cont'd)

- We can solve this recurrence using the dreaded substitution method
  - Guess the answer
    - T(n) = O(n lg n)
  - Assume that the inductive hypothesis holds
    - What's the inductive hypothesis?
    - T(*k*) $\leq$ *a k* lg *k* + b for some constants *a>0* and *b>0 and k<n*

Average case (cont'd)

- The recurrence to be solved
  - T(n) = $\frac{2}{n}\sum_{k=0}^{k=n-1} T(k) + \Theta(n)$
- What next?
  - Plug in the inductive hypothesis
  - $T(n) \leq \frac{2}{n}\sum_{k=0}^{k=n-1}(aklgk + b) + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) \leq \frac{2}{n}\sum_{k=0}^{k=n-1}(aklgk + b) + \Theta(n)$
- What next?
  - Expand out the k=0 case
  - For simplicity, when *n* = 0, we define
    - $anlgn = \lim_{n\to 0} anlgn = 0$
  - T(n) $\leq \frac{2}{n}\left[b + \sum_{k=1}^{k=n-1}(aklgk + b)\right] + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) \leq \frac{2}{n}\left[b + \sum_{k=1}^{k=n-1}(aklgk + b)\right] + \Theta(n)$

    $= \frac{2}{n}\left[\sum_{k=1}^{k=n-1}(aklgk + b)\right] + \frac{2b}{n} + \Theta(n)$
- 2b/n is just a constant, so fold it into Θ(n)
  - $T(n) \leq \frac{2}{n}\sum_{k=1}^{k=n-1}(aklgk + b) + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) \leq \frac{2}{n}\sum_{k=1}^{k=n-1}(aklgk + b) + \Theta(n)$
- What next?
  - Distribute the summation
  - $T(n) = \frac{2}{n}\sum_{k=1}^{k=n-1} aklgk + \frac{2}{n}\sum_{k=1}^{k=n-1} b + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) = \frac{2}{n}\sum_{k=1}^{k=n-1} aklgk + \frac{2}{n}\sum_{k=1}^{k=n-1} b + \Theta(n)$
- What next?
  - Evaluate the summation
  - $T(n) = \frac{2}{n}\sum_{k=1}^{k=n-1} aklgk + \frac{2}{n}\sum_{k=1}^{k=n-1} b + \Theta(n)$

$$= \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + \frac{2b}{n}(n-1) + \Theta(n)$$

Average case (cont'd)

- The recurrence to be solved
  - T(n) = $\frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + \frac{2b}{n}(n-1) + \Theta(n)$
- What next?
  - Since n-1<n, 2b(n-1)/n < 2b
  - T(n) $\leq \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + 2b + \Theta(n)$

Average case (cont'd)

$$T(n) \leq \frac{2a}{n} \sum_{k=1}^{k=n-1} klgk + 2b + \Theta(n)$$

It can be proved that

$$\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

This summation gets its own set of slides later

Average case (cont'd)

- The recurrence to be solved
  - T(n) $\leq \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + 2b + \Theta(n)$
- What next?
  - Substitute $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$
  - T(n) $\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) \leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$
- What next?
  - Distribute the (2a/n) term
  - $T(n) \leq anlgn - \frac{an}{4} + 2b + \Theta(n)$

Average case (cont'd)

- The recurrence to be solved
  - $T(n) \leq anlgn - \frac{an}{4} + 2b + \Theta(n)$
- What is our goal?
  - Our goal is to get $T(n) \leq anlg\ n + b$
  - We rewrite T(n) as

$$T(n) \leq anlgn + b + [\Theta(n) + b - \frac{an}{4}]$$

Average case (cont'd)

- T(n) $\leq$ *an*lg*n* + *b* + $[\Theta(n)+b-\frac{an}{4}]$
- What next?
  - $\Theta(n)+b-\frac{an}{4} \leq 0$
    - $\Theta(n)+b-\frac{an}{4} \leq \Theta(n) + bn - \frac{an}{4} \leq$ c'*n* +$bn - \frac{an}{4} \leq 0$, when $a \geq 4(c' + b)$.
    - Thus, pick *a* large enough that *an*/4 dominates $\Theta(n)+b$
  - Then, T(n) $\leq$ *an*lg*n* + *b*

Average case summary

$$T(n) = \frac{2}{n}\sum_{k=0}^{k=n-1} T(k) + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=0}^{k=n-1}(aklgk + b) + \Theta(n)$$

$$= \frac{2}{n}\left[b + \sum_{k=1}^{k=n-1}(aklgk + b)\right] + \Theta(n)$$

$$= \frac{2}{n}\left[\sum_{k=1}^{k=n-1}(aklgk + b)\right] + \frac{2b}{n} + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{k=n-1}(aklgk + b) + \Theta(n)$$

(when n -> ∞, $\frac{2b}{n}$ - > 0)

Average case summary(cont'd)

$$= \frac{2}{n}\sum_{k=1}^{k=n-1}(aklgk + b) + \Theta(n)$$

$$= \frac{2}{n}\sum_{k=1}^{k=n-1} aklgk + \frac{2}{n}\sum_{k=1}^{k=n-1} b + \Theta(n)$$

$$= \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + \frac{2b}{n}(n-1) + \Theta(n)$$

$$\leq \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + 2b + \Theta(n)$$

Average case summary (cont'd)

$$T(n) \leq \frac{2a}{n}\sum_{k=1}^{k=n-1} klgk + 2b + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + 2b + \Theta(n)$$

$$= an\lg n - \frac{an}{4} + 2b + \Theta(n)$$

$$= an\lg n + b + \Theta(n) + b - \frac{an}{4}$$

$$\leq an\lg n + b$$

Pick *a* large enough that *an*/4 dominates Θ(n)+b

Average case (cont'd)

- So T(n) $\leq$ an lg n + b for certain a and b
  - Thus the induction holds
  - Thus T(n) = O(n lg n)
  - Thus quicksort runs in O(n lg n) time on average
- Now let's prove the summation

$$\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}n^2\lg n - \frac{1}{8}n^2$$

Tightly Bounding

- Prove $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$
- Split the summation for a tighter bound

$$\sum_{k=1}^{k=n-1} klgk = \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil - 1} klgk + \sum_{k=\lfloor \frac{n}{2} \rfloor}^{k=n-1} klgk$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk = \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} klgk + \sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} klgk$
- The lgk in the second term is bounded by lg n

$$\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} klgk + \sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} klgn$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil -1} klgk + \sum_{k=\lceil \frac{n}{2} \rceil}^{k=n-1} klgn$
- Move the lgn outside the summation

$$\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil -1} klgk + \text{lgn} \sum_{k=\lceil \frac{n}{2} \rceil}^{k=n-1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} klgk$ + lgn$\sum_{k=\left\lceil\frac{n}{2}\right\rceil}^{k=n-1} k$
- What next?

  The lg k in the first term is bounded by lg n/2

  $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} klg\frac{n}{2}$ + lgn$\sum_{k=\left\lceil\frac{n}{2}\right\rceil}^{k=n-1} k$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} klg\frac{n}{2}$ + lgn$\sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} k$
- What next?

  lg n/2 = lg n - 1

  $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} k(lgn-1)$ + lgn$\sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} k$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} k(lgn-1)$ + lgn$\sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} k$
- What next?
  - Move (lg n - 1) outside the summation

$$\sum_{k=1}^{k=n-1} klgk \leq (lgn-1)\sum_{k=1}^{k=\lceil\frac{n}{2}\rceil-1} k + \text{lgn}\sum_{k=\lceil\frac{n}{2}\rceil}^{k=n-1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq (lgn - 1) \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil - 1} k$+ lgn$\sum_{k=\lceil \frac{n}{2} \rceil}^{k=n-1} k$
- What next?
  - Distribute the (lg n - 1)

$$\sum_{k=1}^{k=n-1} klgk \leq lgn \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil - 1} k - \sum_{k=1}^{k=\lceil \frac{n}{2} \rceil - 1} k + \text{lgn} \sum_{k=\lceil \frac{n}{2} \rceil}^{k=n-1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq lgn \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil -1} k - \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil -1} k + \lg n \sum_{k=\left\lceil \frac{n}{2} \right\rceil}^{k=n-1} k$
- What next?
  - The summations overlap in range; combine them

$$\sum_{k=1}^{k=n-1} klgk \leq lgn \sum_{k=1}^{k=n-1} k - \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil -1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq lgn \sum_{k=1}^{k=n-1} k - \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil - 1} k$
- What next?
  - The Gaussian series

$$\sum_{k=1}^{k=n-1} klgk \leq \lg\left(\frac{(n-1)n}{2}\right) - \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil - 1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq$lgn $(\frac{(n-1)n}{2}) - \sum_{k=1}^{k=\left\lceil \frac{n}{2} \right\rceil - 1} k$
- What next?
  - Rearrange first term, place upper bound on second

$$\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}[n(n-1)]lgn - \sum_{k=1}^{k=n/2-1} k$$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}[n(n-1)]lgn - \sum_{k=1}^{k=n/2-1} k$
- What next?
  - The Gaussian series
  - $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}[n(n-1)]lgn - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$

Tightly Bounding (cont'd)

- $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}[n(n-1)]lgn - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$
- What next?
  - Multiply it all out
  - $\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}(n^2 lgn - nlgn)$ - $\frac{1}{8}\ n^2\ +\ \frac{n}{4}$

$$= \frac{1}{2}n^2 lgn - \frac{1}{8}\ n^2 - \frac{1}{2}nlgn + \frac{n}{4}$$

$$\leq \frac{1}{2}n^2 lgn - \frac{1}{8}\ n^2 \text{ when n} \geq 2$$

Done !!!!

Tightly Bounding Summary

$$\sum_{k=1}^{k=n-1} klgk = \sum_{k=1}^{k=\lceil n/2\rceil-1} klgk + \sum_{k=\lceil n/2\rceil}^{k=n-1} klgk$$

$$\leq \sum_{k=1}^{k=\lceil n/2\rceil-1} klg(\frac{n}{2}) + \sum_{k=\lceil n/2\rceil}^{k=n-1} klgn$$

$$= \sum_{k=1}^{k=\lceil n/2\rceil-1} k(lgn-1) + \text{lgn} \sum_{k=\lceil n/2\rceil}^{k=n-1} k$$

$$= (lgn-1) \sum_{k=1}^{k=\lceil n/2\rceil-1} k + \text{lgn} \sum_{k=\lceil n/2\rceil}^{k=n-1} k$$

Tightly Bounding Summary (cont'd)

$$\sum_{k=1}^{k=n-1} klgk \leq (lgn-1)\sum_{k=1}^{k=\lceil n/2\rceil-1} k + \lg n \sum_{k=\lceil n/2\rceil}^{k=n-1} k$$

$$= lgn \sum_{k=1}^{k=\lceil n/2\rceil-1} k - \sum_{k=1}^{k=\lceil n/2\rceil-1} k + \lg n \sum_{k=\lceil n/2\rceil}^{k=n-1} k$$

$$= lgn \sum_{k=1}^{k=n-1} k - \sum_{k=1}^{k=\lceil n/2\rceil-1} k$$

$$= \lg n \left(\frac{(n-1)n}{2}\right) - \sum_{k=1}^{k=\lceil n/2\rceil-1} k$$

$$\leq \frac{1}{2}[n(n-1)]lgn - \sum_{k=1}^{k=n/2-1} k$$

Tightly Bounding Summary(cont'd)

$$\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}[n(n-1)]lgn - \sum_{k=1}^{k=n/2-1} k$$

$$= \frac{1}{2}[n(n-1)]lgn - \frac{1}{2}\left(\frac{n}{2}\right)\left(\frac{n}{2}-1\right)$$

$$= \frac{1}{2}(n^2lgn - nlgn) - \frac{1}{8}n^2 + \frac{n}{4}$$

$$= \frac{1}{2}n^2lgn - \frac{1}{8}n^2 - \frac{1}{2}nlgn + \frac{n}{4}$$

$$\leq \frac{1}{2}n^2lgn - \frac{1}{8}n^2 \text{ when } n \geq 2$$

Done !!!!

COT 6405 Introduction to Theory of Algorithms

Topic 9. Randomized Quicksort

Worst case quicksort

- What will happen if the array is already sorted?
  - The partitioning routine produces n-1 elements and one with 0 elements.
  - How about the running time?
  - T(n) = O(n^2)

Improving quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on an already-sorted input
- How to avoid this?
- Two solutions
  - Randomize the input array
  - Pick a random pivot element
- How will these solve the problem?
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Randomized version of quicksort

- We add randomization to quicksort.
  - We could randomly permute the input array: very costly
  - Instead, we use **random sampling** to pick one element at random as the pivot
    - Don't always use $A[r]$ as the pivot.

Randomized version of quicksort

RANDOMIZED-PARTITION*(A, p, r)*

i ←RANDOM*(p, r)*

exchange *A*[*r*] ↔ *A*[*i*]

return PARTITION*(A, p, r)*

Randomization of quicksort stops any specific type of array from causing the worst case behavior

- E.g., an already-sorted array causes worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

Randomized version of quicksort

RANDOMIZED-QUICKSORT*(A, p, r)*

if $p < r$

then $q \leftarrow$RANDOMIZED-PARTITION*(A, p, r)*

RANDOMIZED-QUICKSORT*(A, p, q* − 1*)*

RANDOMIZED-QUICKSORT*(A, q* + 1*, r)*

Analysis of quicksort

- We will analyze
  - the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT
  - the expected (average-case) running time of QUICKSORT and RANDOMIZED-QUICKSORT

Worst-case analysis

- We saw a worst-case split (0:n-1) at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which,
  - Intuitively, is the worst-case running time
- We now prove this assertion

Worst-case analysis (cont'd)

- Let T (n) be the worst-case time for the procedure QUICKSORT on an input of size n, we have the recurrence
- $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$
  - q ranges between 0 and n-1, because the procedure PARTITION produces two subproblems with total size n-1
- We guess that T(n) $\le cn^2$ for some constant c

Worst-case analysis (cont'd)

- Substitution this guess into the recurrence, we obtain

$$T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + \mathrm{c}(n-q-1)^2) + \Theta(n)$$

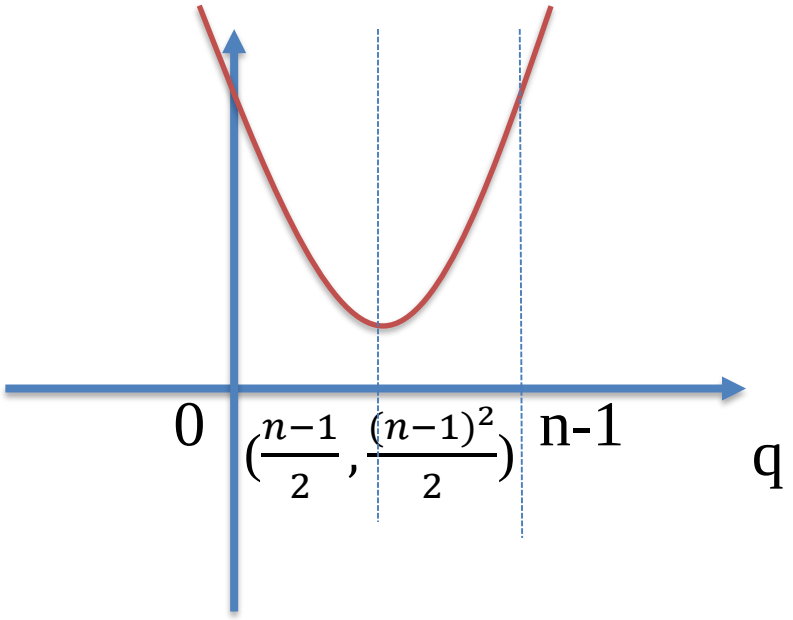
$$= c \max_{0 \leq q \leq n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

Exercise

- What values of q can enable the expression $q^2 + (n - q - 1)^2$ to achieve the maximum value?

Worst-case analysis (cont'd)

- $q^2 + (n - q - 1)^2 = 2q^2 - 2(n-1)q + (n-1)^2$
- What's the shape of this function?
  - A cup-shaped parabola



Worst-case analysis (cont'd)

- $T(n) \leq \max_{0 \leq q \leq n-1} (cq^2 + (n - q - 1)^2) + \Theta(n)$

The expression $q^2 + (n - q - 1)^2$ achieves the maximum value when q is either 0 or n-1.

Worst-case analysis (cont'd)

- This observation gives us the bound
  - $\max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) = (n-1)^2$

    $= n^2 - 2n + 1$
- Continuing with our bounding of T(n), we obtain

$$T(n) \le c \max_{0 \le q \le n-1} (q^2 + (n - q - 1)^2) + \Theta(n)$$

$$= \text{c } n^2 \text{ - c(2n-1) + } \Theta(n)$$

$$\le \text{c } n^2$$

Since we can pick *c* large enough so that *c*(2*n*-1) dominates $\Theta(n)$, T(*n*) = O (n^2)

Exercise

- Let T (n) be the worst-case time for the procedure QUICKSORT on an input of size n. Prove T(n) = $\Omega\ (n^2)$

Worst-case analysis (cont'd)

- $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$
- We guess that T(n) $\geq dn^2$ for some constant d
  Substitution this guess into the recurrence, we obtain

$$T(n) \geq \max_{0 \le q \le n-1} (dq^2 + d(n-q-1)^2) + \Theta(n)$$

$$= d \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

$$= \text{d } n^2 \text{ - d(2n-1) + } \Theta(n)$$

$$\geq \text{d } n^2$$

Since we can pick a small *d* so that $\Theta(n)$dominates *d*(2*n*-1) , T(*n*) = $\Omega\ (n^2)$

Average case analysis

- The dominant cost of the algorithm is partitioning.
- What is the maximum number of calls to the function PARTITION?
  - Hint: PARTITION removes the pivot element from future consideration each time.
  - Thus, PARTITION is called at most n times.

Partition array $A[p..r]$

```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p − 1
        for j ← p to r − 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        // move the pivot between the two subarraies
        exchange A[i + 1] ↔ A[r]
        // return the pivot
        return i + 1
```

Average case analysis (cont'd)

Lemma 7.1: Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

The amount of work of each call to PARTITION is a constant plus the number of comparisons performed in its for loop

Find the number of comparisons

- For ease of analysis:
  - Rename the elements of A as $z_1, z_2, \ldots, z_n$, with z_i being the i-th smallest element.
- Each pair of elements is compared at most once. Why?
- Because elements are compared only to the pivot element, and then the pivot element is never in any later call to PARTITION.

Cont'd

- Our analysis uses indicator random variables
- Let $X_{i,j} = I\{z_i \text{ is compared to } z_j\}$.

$$= \begin{cases} 1 \text{ if } z_i \text{ is compared to } z_j \\ 0 \text{ if } z_i \text{ is not compared to } z_j \end{cases}$$

- Considering whether z_i is compared to z_j at any time during the entire quicksort algorithm, not just during one call of PARTITION.

Cont'd

- Since each pair is compared at most once, the total number of comparisons performed by the algorithm is

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \ .$$

Take expectations of both sides, use Lemma 5.1 and linearity of expectation:

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathrm{E}[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\} \ . \end{aligned}$$

Exercise

- Prove E[X_{ij}] = Pr (z_i is compared to z_j)

Cont'd

- $\mathrm{E}[X_{ij}] = 1 \cdot Pr(X_{ij} = 1) + 0 \cdot Pr(X_{ij} = 0)$

  $= Pr(X_{ij} = 1)$

  = *Pr* (z_i is compared to z_j)

Cont'd

- Now we need to find the probability that two elements are compared.
- Think about when two elements are *not* compared.
  - numbers in separate partitions will not be compared.
  - {8, 1, 6, 4, 0, 3, 9, 5} and the pivot is 5, so that none of the set {1, 4, 0, 3} will be compared to any of the set {8, 6, 9}

Cont'd

- Once a pivot x is chosen, such that z_i< x < z_j, then z_i and z_j will never be compared at any later time
- If either $z_i$or z_j is chosen as a pivot before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij} , except itself.
- The probability that z_i is compared to z_j is the probability that either z_i or z_j is the first element chosen to be the pivot

Cont'd

- Assume pivots are chosen randomly and independently.
- z_i and z_j must be in the same set after partition, otherwise they will never be compared
- Thus, the probability that any particular one of them is the first one chosen is 1/ n_{ij}, where n_{ij} is the size of this set

Cont'd

- Therefore
- *Pr* (z_i is compared to z_j) = *Pr* (z_i or z_j is the first pivot chosen from the set) = *Pr* (z_i is the first pivot chosen from the set) + *Pr* (z_j is the first pivot chosen from the set)

  = $1/ n_{ij} + 1/ n_{ij} = 2/ n_{ij}$

Cont'd

- E(X) = $\sum_{i=1}^{i=n-1} \sum_{j=i+1}^{n} Pr$ (z_i is compared to z_j)

$$= \sum_{i=1}^{i=n-1} \sum_{j=i+1}^{n} \frac{2}{n_{ij}}$$

When i = 1 and j = 2, n_{ij} reaches the smallest value of 2, and when i = 1 and j = n, n_{ij} reaches the largest value of n. Thus, n_{ij} ranges between 2 and n, and by changing variable (let $k = n_{ij}$), we have

$$E(X) = \sum_{i=1}^{i=n-1} \sum_{j=i+1}^{n} \frac{2}{n_{ij}} = \sum_{i=1}^{i=n-1} \sum_{k=2}^{n} \frac{2}{k}$$

Cont'd

- E(X)= $\sum_{i=1}^{i=n-1} \sum_{k=2}^{n} \frac{2}{k} = \sum_{i=1}^{i=n-1} O(lgn)$

  = O(nlgn)

Harmonic Series:

$\sum_{k=1}^{n} \frac{2}{k} = 2 \sum_{k=1}^{n} \frac{1}{k} < 2\ln n+1 = O(\lg n)$

COT 6405 Introduction to Theory of Algorithms

Topic 10. Linear Time Sorting

How fast can we sort?

- The sorting algorithms we learned so far
  - Insertion Sort, Merge Sort, Heap Sort, and Quicksort
- How fast are they?
  - Insertion sort O(n^2)
  - Merge Sort O(nlgn)
  - Heap Sort O(nlgn)
  - Quicksort O(nlgn)

Common property

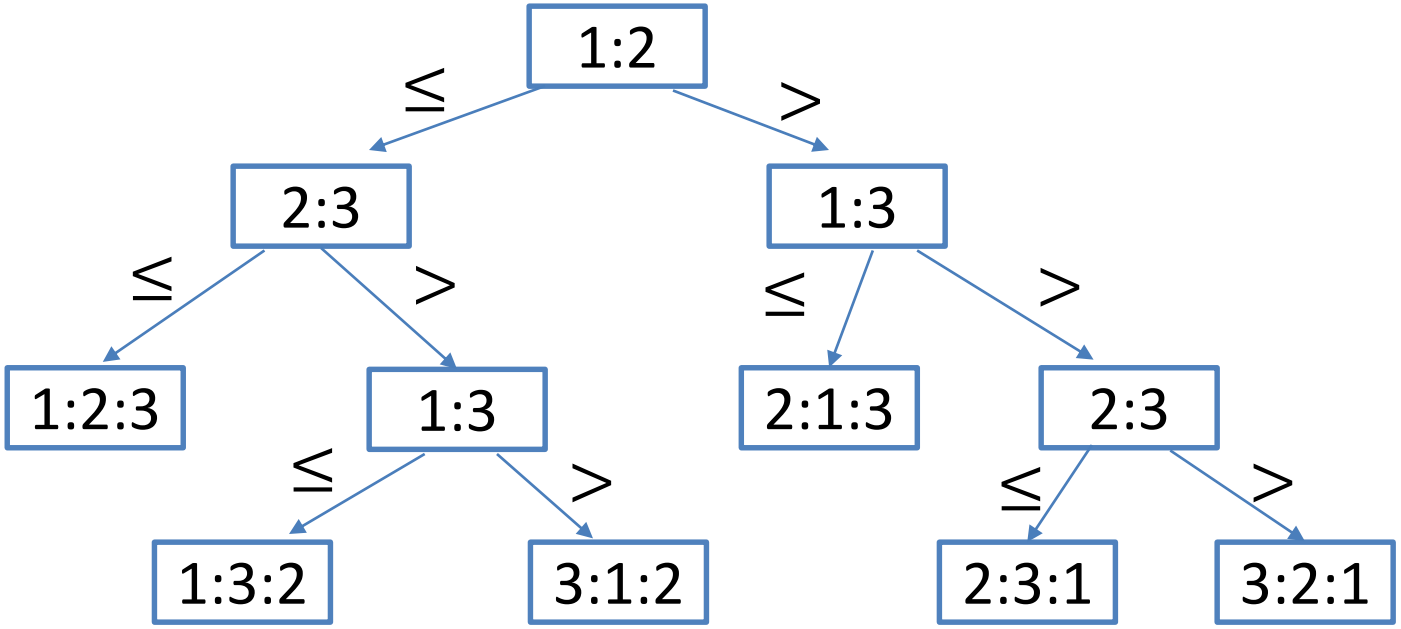
- Use only comparisons between elements to gain order information about an input sequence
- Comparison sort
  - Given two elements a_i and a_j, we perform one of the following tests to determine their relative order
  - $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$

Decision trees

- We can view comparison sorts abstractly in terms of decision trees
  - A decision tree is a binary tree that represents the comparisons between elements
  - Each node on the tree is a comparison of *i*:*j*, i.e., a_i v.s. a_j

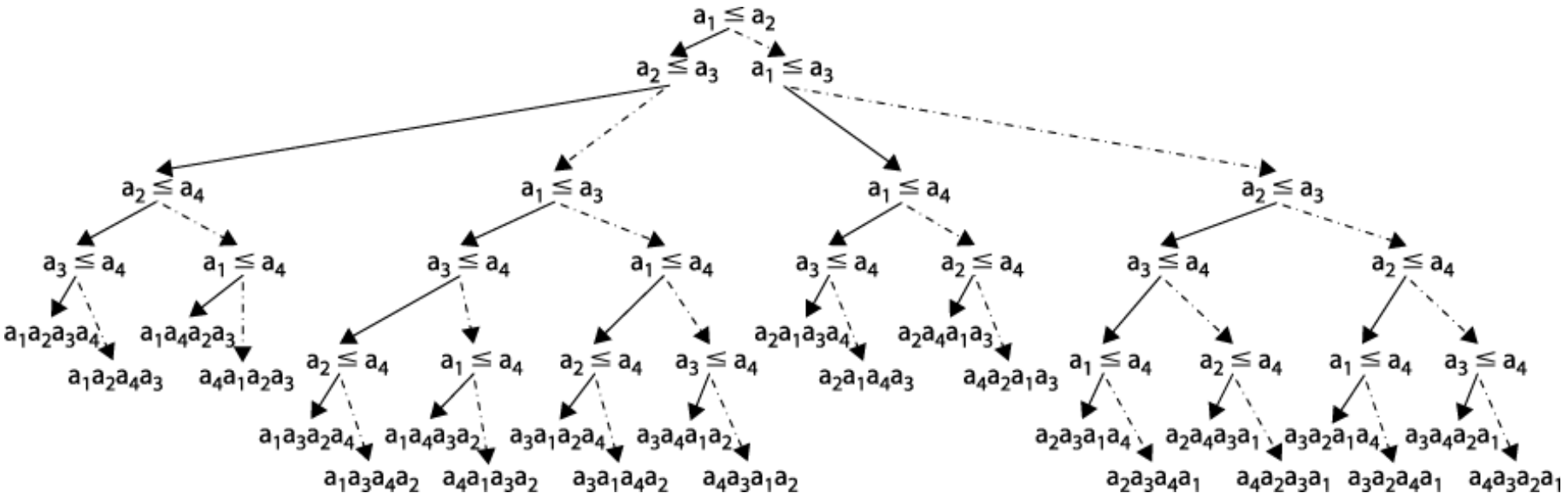
Constructing the decision tree

- Given an input sequence $\{a_1, a_2, a_3\}$



Decision tree for an input set of four elements

Given an input sequence $\{a_1, a_2, a_3, a_4\}$



Decision trees (cont'd)

- What do the leaves represent?
  - The leaf node in the tree indicates the sorted ordering
- How many leaves must be there for an input of size *n*
  - Each of the n! permutations on n elements must appear as one of the leaves of the decision tree

Lemma

- Any binary tree of height h has $\leq 2^h$ leaves
- In other words:
  - i = number of leaves
  - h = height
  - Then, $i \leq 2^h$
- How to prove this?

Theorem 8.1

- Any comparison sort algorithm requires $\Omega(nlgn)$ comparisons.
- How to prove?
  - By proving that the height of the decision tree is $\Omega(nlgn)$
  - What's the # of leaves of a decision tree? l = ?
  - A decision tree is a binary tree. What's the maximum # of leaves of a general binary tree?

    l_{max} = ?

Proof

- l = n! and $l_{max} = 2^h$
- Clearly, the # of leaves of a decision tree is less than or equal to the maximum # of leaves in a general binary tree
- So we have: $n! \leq 2^h$
- Taking logarithms: $\lg(n!) \leq h$

Proof (cont'd)

- Stirling's approximation tells us:

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

- Thus, $h \geq \lg(n!)$

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

Sorting in linear time

- Counting sort
  - No direct comparisons between elements!
  - Depends on assumption about the numbers being sorted
    - We assume numbers are in the range [0.. k]
  - The algorithm is NOT "in place"
    - Input: A[1..*n*], where A[j] $\in$ {0, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted
    - Auxiliary counter storage: Array C[0..*k*]
    - notice: A[], B[], and C[] → not sorting in place

Counting sort

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3          C[i]= 0;
4     for j= 1 to A.length
5          C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7          C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

A counting sort example

Numbers are in the range [0.. 5]

A

2	5	3	0	2	3	0	3

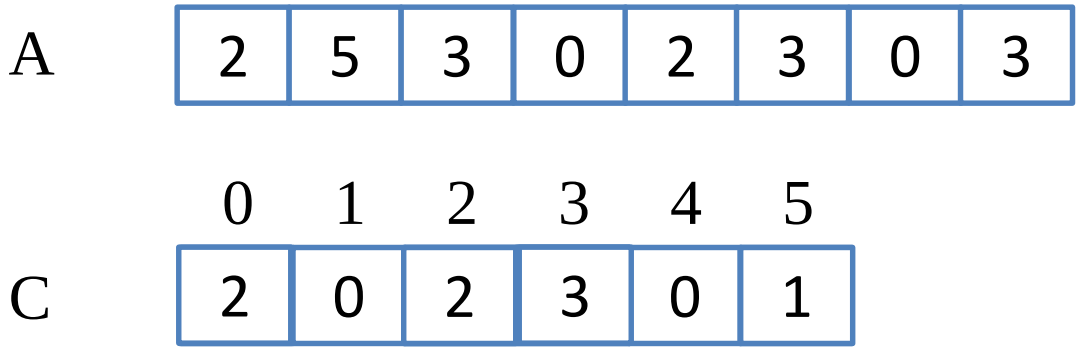
k = 5

C

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

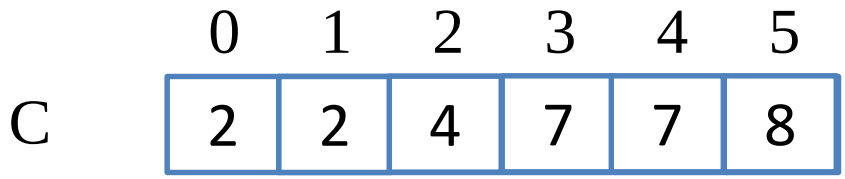
```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3           C[i]= 0;
4     for j= 1 to A.length
5           C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7           C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9           B[C[A[j]]] = A[j];
10          C[A[j]] -= 1;
```

Filling the C array



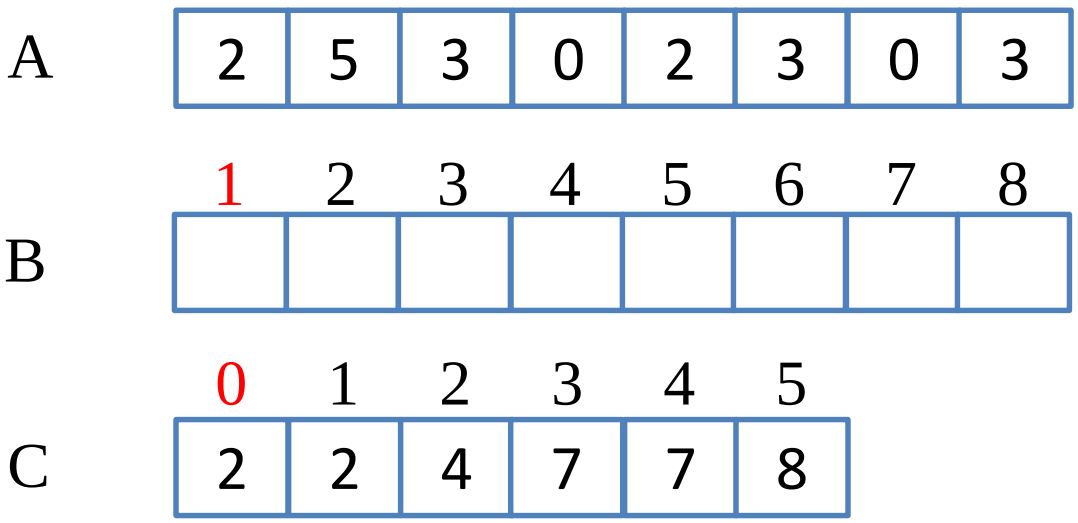

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3          C[i]= 0;
4     for j= 1 to A.length // counting each number
5          C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7          C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Filling the C array (Cont'd)



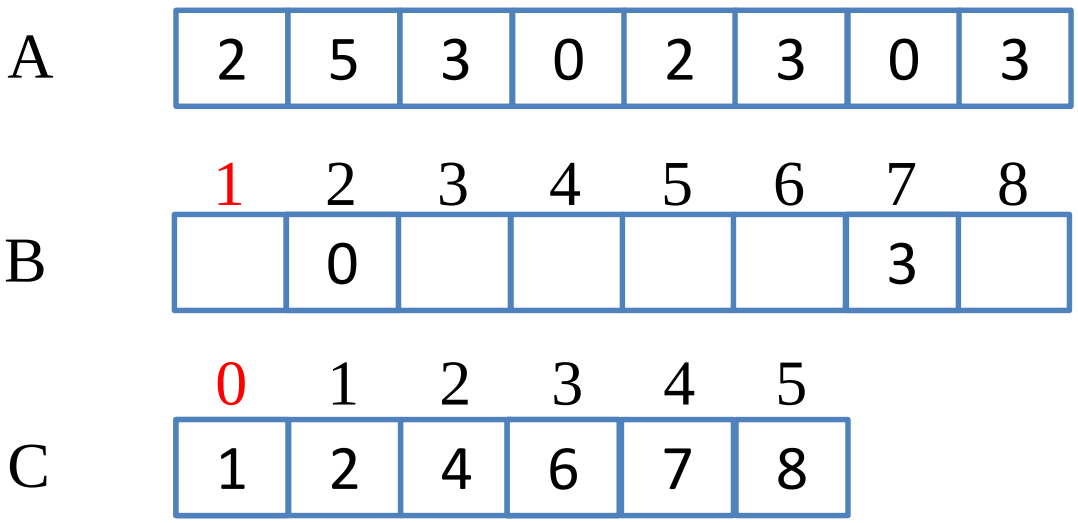

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3          C[i]= 0;
4     for j= 1 to A.length
5          C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7          C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Sorting the numbers



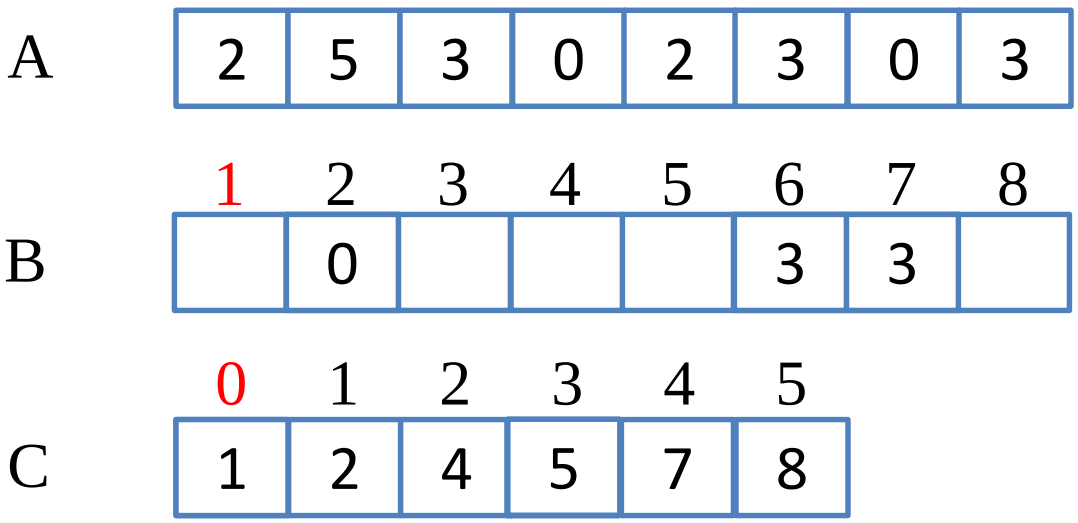

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3           C[i]= 0;
4     for j= 1 to A.length
5           C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7           C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9           B[C[A[j]]] = A[j];
10          C[A[j]] -= 1;
```

Sorting the numbers




```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3          C[i]= 0;
4     for j= 1 to A.length
5          C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7          C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Sorting the numbers



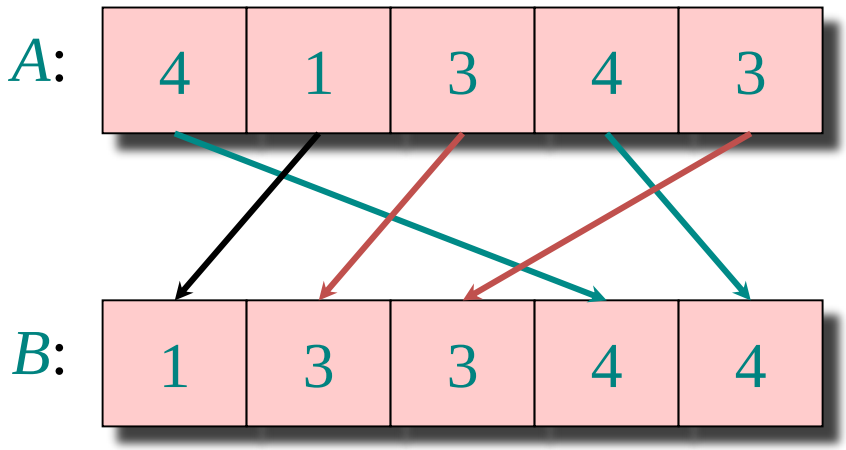

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3           C[i]= 0;
4     for j= 1 to A.length
5           C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7           C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9           B[C[A[j]]] = A[j];
10          C[A[j]] -= 1;
```

Counting sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$ → $k < c\ n$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$! Contradiction?
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is stable
    - The elements with the same value is in the same order as the original
    - index $i < j$, $a_i = a_j$ → new index $i' < j'$

Stable sorting

Counting sort is a stable sort: it preserves the input order among equal elements.



Counting Sort

- Why don't we always use counting sort?
- Because it depends on range *k* of elements
- Could we use counting sort to sort 32 bit integers? Why or why not?
- Answer: no, *k* too large (2^{32} = 4,294,967,296)
  - We need huge arrays, e.g., C[4,294,967,296]?
  - k >> n → O(n+k) = O(k)

Radix Sort

- Intuitively, we may sort on the most significant digit (MSD), then the second msd, etc.
- Recursive MSD radix sort:
  - Take the k-th most significant digit (MSD)
  - Sort based on that digit, grouping same digit elements into one bucket
  - In each bucket, start with the next digit and sort recursively
  - Finally, concatenate the buckets in order

An example of a forward recursive MSD radix sort

- Original sequence: 170, 045, 075, 090, 002, 024, 802, 066
- 1st pass- Sorting by most significant digit (100's):
  - Zero bucket: 045, 075, 090, 002, 024, 066
  - One bucket: 170
  - Eight bucket: 802

An example (cont'd)

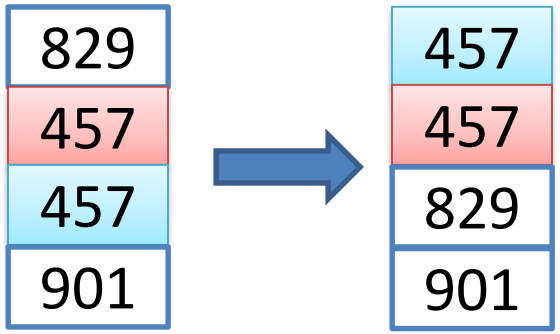
- 2nd pass- Sorting by next most significant digit (10's), only needed by numbers in zero bucket:
  - 045, 075, 090, 002, 024, 066
  - Zero bucket: 002
  - Twenties bucket: 024
  - Forties bucket: 045
  - Sixties bucket: 066
  - Seventies bucket: 075
  - Nineties bucket: 090

An example (cont'd)

- 3rd pass- Sorting by least significant digit (1's): no need because there are no tens buckets with more than one number.
- 4th pass- The sorted zero hundreds buckets are concatenated and joined in sequence to give 002, 024, 045, 066, 075, 090, 170, 802
  - Zero bucket: 002
  - Twenties bucket: 024
  - Forties bucket: 045
  - Sixties bucket: 066
  - Seventies bucket: 075
  - Nineties bucket: 090
  - Zero bucket: 045, 075, 090, 002, 024, 066
  - One bucket: 170
  - Eight bucket: 802

Most Significant Digit (MSD) Radix Sort

- Problem:
  - lots of intermediate piles of cards to keep track of
    - *d* digits –> T(n) = O(*dn*)
    - Needs to use many buckets to store intermediate results, each is a linked list of size up to *n*
  - MSD sort does not necessarily preserve the original order of duplicate keys
    - Depending on how we sort the bucket

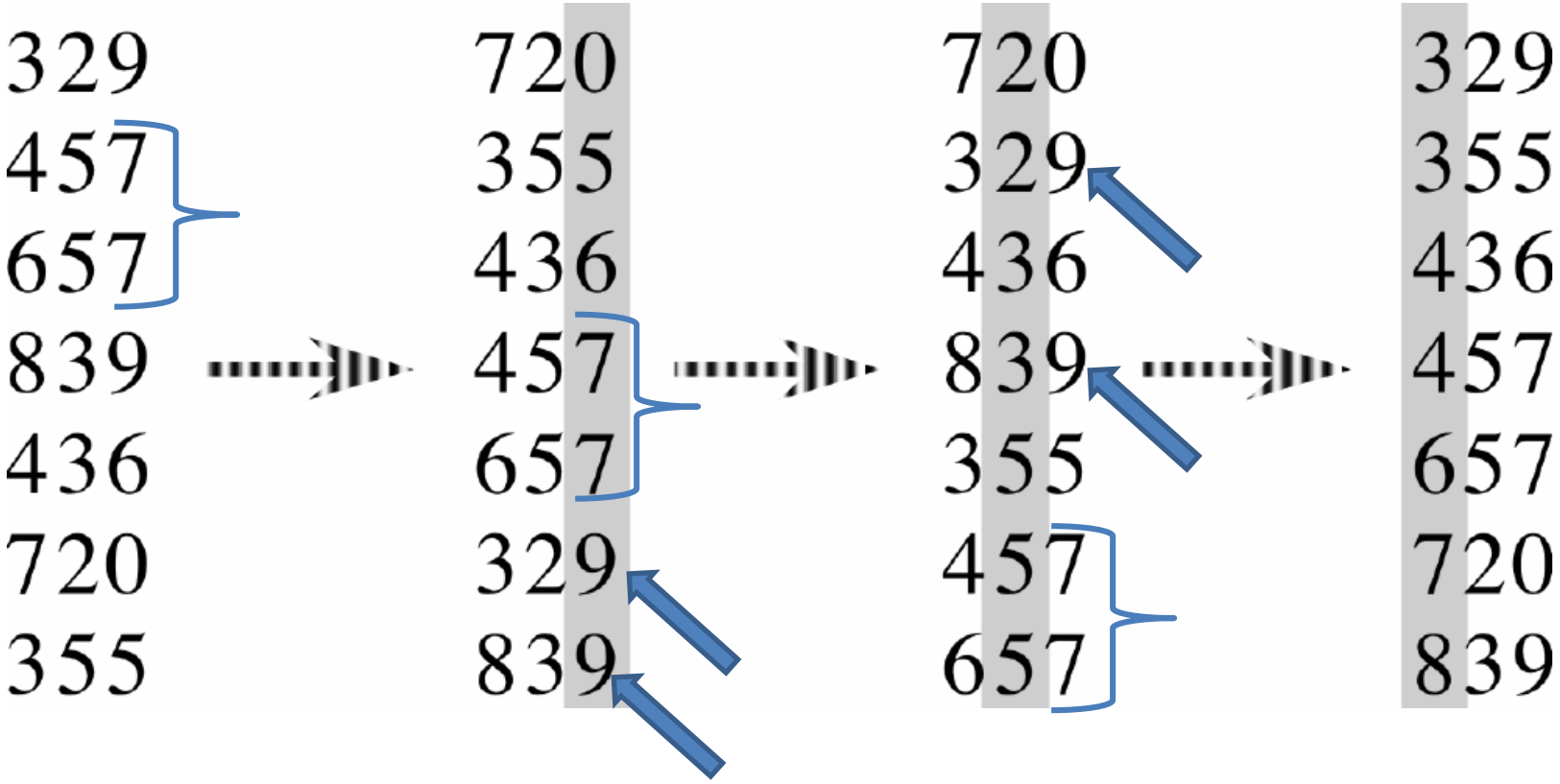


Least significant digit (LSD) Radix Sort

- Key idea: sort the least significant digit first
- Assume we have d-digit numbers in A

```
RadixSort(A, d)
        for i= 1 to d
            StableSort(A) on digit i
```

Example: LSD Radix Sorting



Radix Sort

- Can we prove it works?
- Sketch of an inductive argument (induction on the number of passes)
  - Assume lower-order digits {j: j < i } are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits are irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Questions?

- Can we use any sorting algorithms instead of stable sorting in LSD Radix sorting?

Why stable sorting

- 6**5**7 6**5**8 469 595
- If the sorting algorithm is not stable
- First pass: 595 6**5**7 6**5**8 469
- Second pass: 6**5**8 6**5**7 469 595
- Third pass: 469 595 6**5**8 6**5**7

Radix Sort

- What sort will we use to sort on digits?
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from 0..k
  - Time: O(n + k)
- Each pass over n numbers with d digits takes time O(n+k), so the total time O(dn+dk)
  - When d is constant and k= O(n), takes O(n) time

How to break words into digits?

- We have *n* word
- Each word is of *b* bits
- We break each word into *r*-bit digits, d = $\lceil b/r \rceil$
- Using counting sort, $k = 2^r - 1$
- E.g., 32-bit word, we break into 8-bit digits
  - $d = \lceil 32/8 \rceil = 4$, $k = 2^8 - 1 = 255$
- $T(n) = \Theta(d*(n+k)) = \Theta(b/r * (n + 2^r))$

How to choose r?

How to choose r? Balance b/r and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta(bn/\lg n)$.

Still in O(n)

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

Radix Sort Example

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - $r = 16$, $d = 4$, $n = 10^6$, and $b = 64$
  - $b/r * (n + 2^r) = 4{,}262{,}144 \approx 4n$
  - We can sort in just four passes with radix sort!
- Compares well with typical O(n lg n) comparison sort
  - Requires approximately lg n = 20 operations per number being sorted
  - So why would we ever use anything but radix sort?
  - Doesn't sort in place (why?)
  - Depends on implementation, e.g., quicksort uses cache better

Summary: Radix Sort

- Assumption: input has *d* digits ranging from 0 to *k*
  - Basic idea:
    - Sort elements by digit starting with least significant
    - Use a stable sort (like counting sort) for each stage
  - 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
  - Fast, stable, and Simple to code
    - Doesn't sort in place
    - Depends on implementation, e.g., quicksort uses cache better
    - Cannot easily sort floating point numbers

Bucket Sort

- Assumes the input is generated by a random process that distributes elements uniformly over [0, 1).
- ***Idea:***
  - Divide [0, 1) into n equal-sized *buckets*.
  - Distribute the n input values into the buckets.
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one.

Bucket Sort (cont'd)

- Input:
  - $A[1 . . n]$, where $0 \le A[i] < 1$ for all i .
- Auxiliary array:
  - $B[0 . . n - 1]$ of linked lists, each list initially empty.

Bucket sort Implementation

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$

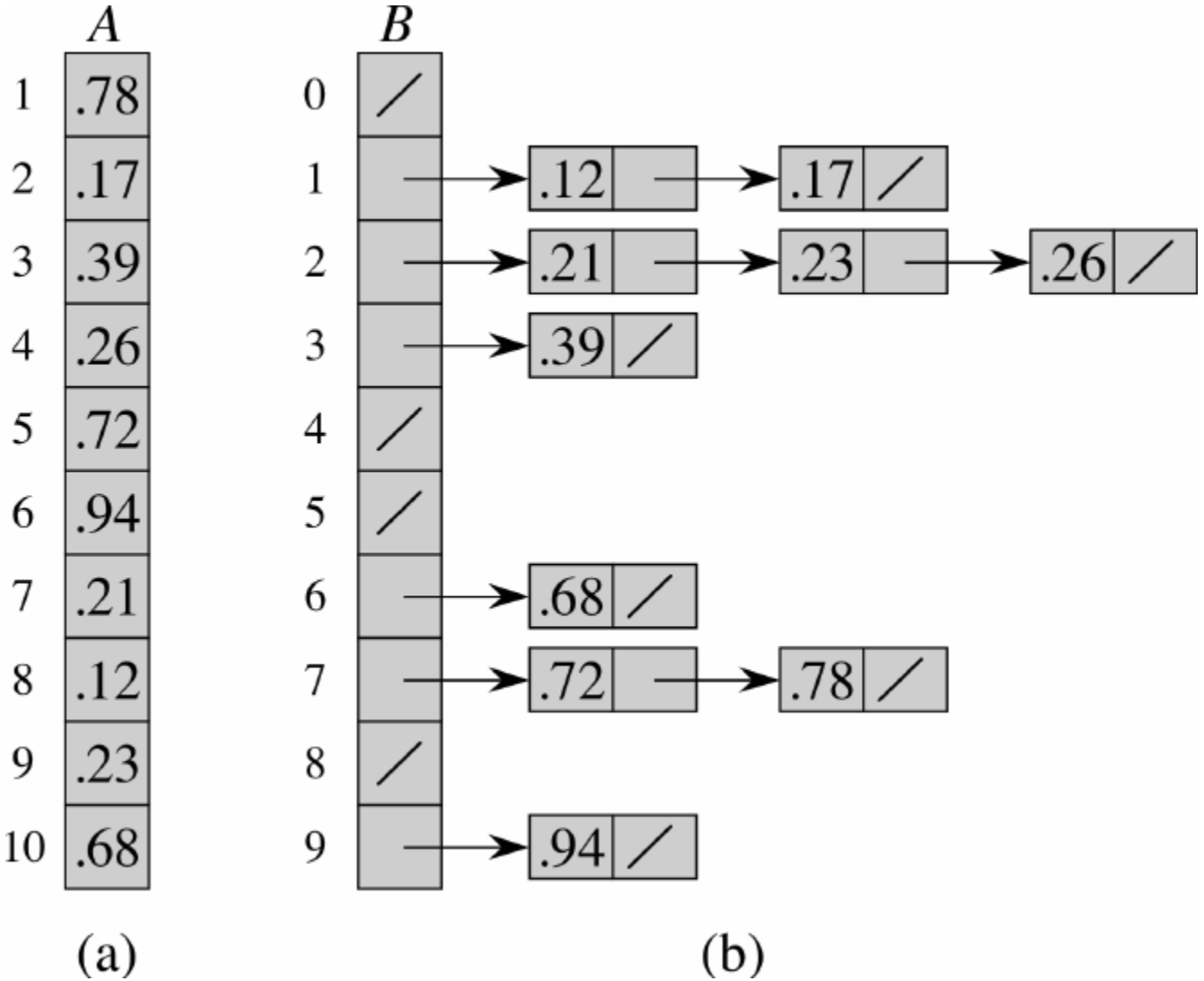
  do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order

return the concatenated lists

Easily compute the bucket index $\lfloor n \cdot A[i] \rfloor$

Bucket sort with 10 buckets



Correctness

- Consider $A[i]$ and $A[j]$
  - Assume without loss of generality that $A[i] \leq A[j]$
  - Then, bucket index $n \cdot A[i] \leq n \cdot A[j]$
- A[i] is placed into the same bucket as $A[j]$ or into a bucket with a lower index
  - If same bucket, insertion sort fixes up
  - If earlier bucket, concatenation of lists fixes up

Informal Analysis

- All lines of algorithm except insertion sorting take $\Theta(n)$ altogether
- Since the inputs are uniformly and independently distributed over [0,1), we do not expect many numbers to fall into each bucket
- Intuitively, if each bucket gets a constant number of elements, it takes $O(1)$ time to sort each bucket $\Rightarrow O(n)$ sort time for all buckets.

Formal Analysis

- Define a random variable:

  n_i = the number of elements placed in bucket $B[i]$
- Because insertion sort runs in quadratic time, bucket sort time is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Formal Analysis (Cont'd)

Take expectations of both sides:

$$\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] && \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) && (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}$$

n_i = the number of elements placed in bucket $B[i]$

n_i = the number of elements placed in bucket $B[i]$

Claim

$\mathrm{E}[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

$X_{i,j}$ = I{A[j] falls in bucket i}.

= { 1 if A[j] falls in bucket i

0 if A[j] doesn't fall in bucket i

The Claim

Then

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}\left[X_{ij}X_{ik}\right] \quad \text{(linearity of expectation)}
\end{aligned}$$

$$(x_1+x_2+x_3)(x_1+x_2+x_3)$$
$$= x_1^2 + x_1x_2 + x_1x_3$$
$$+ x_2^2 + x_1x_2 + x_2x_3$$
$$+ x_3^2 + x_1x_3 + x_2x_3$$

$$\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}$$

Analysis

$\mathrm{E}[X_{ij} X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$\begin{aligned} \Rightarrow \mathrm{E}[X_{ij} X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \end{aligned}$$

Therefore:

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{n^2}$$

Analysis (Cont'd)

$$
\begin{aligned}
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

■ (claim)

Therefore:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

Analysis conclusion

- This is a probabilistic analysis
  - We used probability to analyze an algorithm whose running time depends on the distribution of inputs.
- With bucket sort, if the input isn't drawn from a uniform distribution on [0, 1), all bets are off
  - Performance-wise, but the algorithm is still correct

Bucket Sort Summary

- Assumption: input is *n* real #'s from [0, 1)
  - We can map other number into the range of [0, 1)
- Basic idea:
  - Create *n* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*n*
  - Add each input element to appropriate bucket and sort buckets with insertion sort
- Uniform input distribution → O(1) bucket size
  - Therefore the expected total time is O(n)

Linear Sorting Common Mistakes

- Using counting sort, when memory is limited
  - The size of k → the size of C[0..k]
- Using bucket sort, when the input are not uniform distributed

Linear-time Sorting Summary

- We have learned three linear-time sorting algorithms
- Their assumptions on input
  - Counting sort → [0..k]
  - Radix sort → d digits
  - Bucket sort → uniform distribution [0, 1)

COT 6405 Introduction to Theory of Algorithms

Topic 11. Order Statistics

Order statistic

- 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, we have 2 medians: lower median n/2 and upper median n/2+1
    - By our convention, “median” normally refers to the lower median

How to calculate

- How can we calculate order statistics?
- What is the running time?
  - Simple method: Sort first, e.g., Heapsort O(n lg n)
  - then return the i-th element

Find the minimum

- How many comparisons are needed to find the minimum element in a set? Or the maximum?

```
MINIMUM(A)
  min=A[1]
  for i=2 to A.length
        if min > A[i]
              min = A[i]
  return min
```

Find both the minimum & the maximum

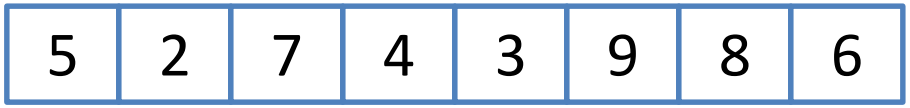
- We can find the minimum with n-1 comparisons
- We can find the maximum with n-1 comparisons
- So we can find both the minimum and the maximum with 2(n-1) comparisons

Can we reduce the cost?

- Can we find the minimum and maximum with less than twice the cost, 2(n-1) ?
- Yes: walk through elements by pairs
  - Compare each element in pair to the other
  - Compare the larger one to maximum, the smaller one to minimum
- Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding order statistics: The Selection Problem

- A more interesting problem is the selection problem
  - finding the *i*-th smallest element of a set
- A naïve way is to sort the set
  - Running time takes O(nlgn)
- We will study a practical randomized algorithm with O(n) expected running time
- We will then study an algorithm with O(n) worst-case running time

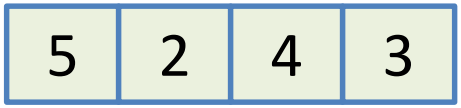


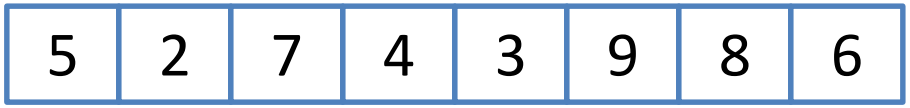
Find the 3rd smallest element

We first partition this array. Assume pivot = 6, after partition

4 elements are smaller than the pivot and 3 are larger than the Pivot. So the pivot is the 5th smallest element of the original array

The index of the pivot is 5, and 3 is less than 5. Hence, the 3rd smallest element of the original array is indeed the 3rd smallest element of the first subarray after partition





Find the 7th smallest element

We first partition this array. Assume pivot = 6, after partition

4 elements are smaller than the pivot and 3 are larger than the Pivot. So the pivot is the 5th smallest element of the original array

The index of the pivot is 5, and 7 is larger 5. Hence, the 7th smallest element of the original array is indeed the 2ed smallest element of the second subarray after partition

Randomized Selection

- Key idea: use partition() from Quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a randomized partition

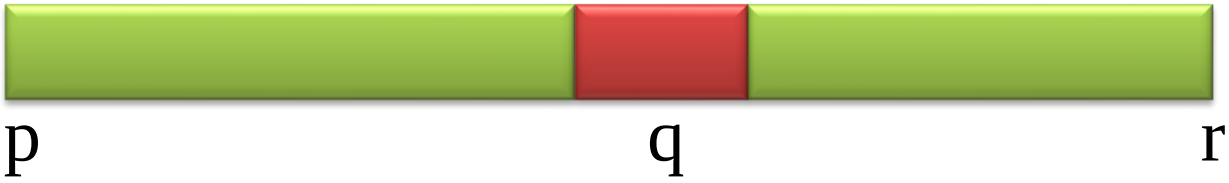
q = RANDOMIZED-PARTITION*(A, p, r)*

RANDOMIZED-PARTITION*(A, p, r)*

i ←RANDOM*(p, r)*

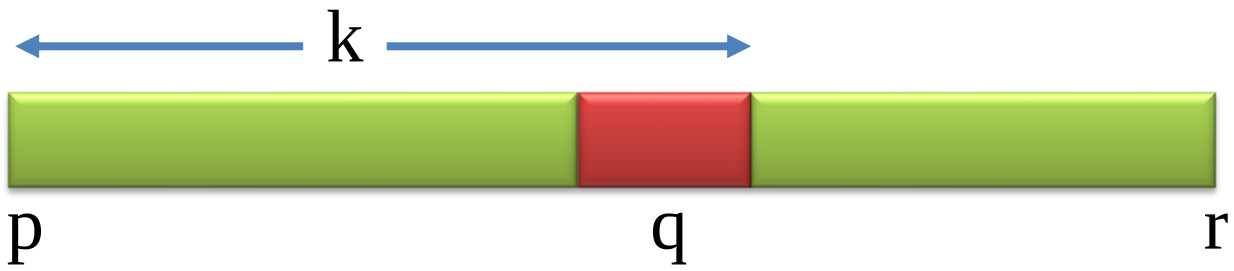
exchange *A*[*r*] ↔ *A*[*i*]

return PARTITION*(A, p, r)*



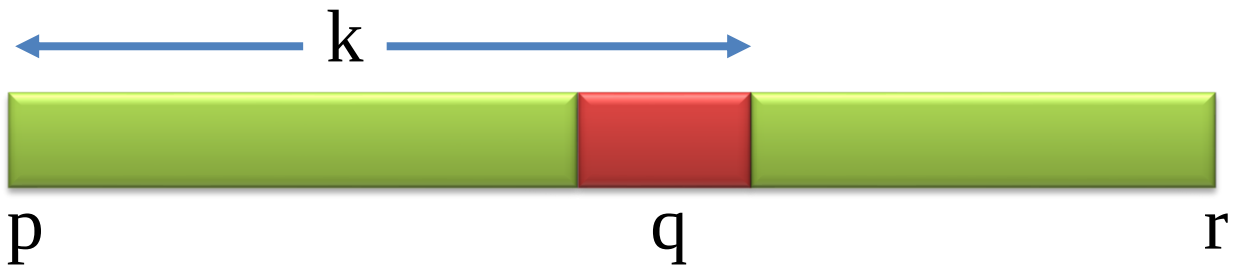
Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, ?);
    else
        return RandomizedSelect(A,q+1,r, ?? );
```



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A,q+1,r, i-k);
```



Analyzing Randomized-Select()

- Worst case: partition always 0:n-1
  - $T(n) = T(n-1) + O(n) = O(n^2)$
  - No better than sorting!
- "Best" case: suppose a 9:1 partition
  - $T(n) \leq T(9n/10) + O(n) = O(n)$ (why?)
  - Master Theorem, case 3
  - Better than sorting!

Average case analysis

- We can upper-bound the time needed for the recursive call by the time needed for the recursive call on the largest possible input
- In other words, to obtain an upper bound, we assume that the i-th element is always on the side of the partition with the greater number of elements

Average case analysis (cont'd)

- We have a total of *n* partition outcome, and k can range between 1 and *n*. Therefore, the expected average case time T(*n*) is

$$T(n) \leq \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

Average case analysis (cont'd)

- If n is even, T($\lceil n/2 \rceil$) up to T(n-1) appears exactly twice.
  - E.g., n = 4, T(n) = 1/4(T(max(0, 3)) + T(max(1, 2)) + T(max(2, 1)) + T(max(3, 0)) = 2/4 (T(3) + T(2))
- If n is odd, all these terms appear twice and T($\lfloor n/2 \rfloor$) appears once
  - E.g., n = 5, T(n) = 1/5(T(max(0, 4)) + T(max(1, 3)) + T(max(2, 2)) + T(max(3, 1)) + T(max(4, 0))
    = 2/4 (T(4) + T(3)) + T(2)

Average case analysis (cont'd)

$$T(n) \leq \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

$$= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

$$\leq \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

n is even

Average case analysis (cont'd)

$$T(n) \leq \frac{1}{n}\sum_{k=1}^{n} T(\max(k-1, n-k)) + O(n)$$

n is odd

$$= \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + \frac{1}{n}T(\lfloor n/2 \rfloor) + O(n)$$

$$\leq \frac{2}{n}\sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + \frac{2}{n}T(\lfloor n/2 \rfloor) + O(n)$$

$$= \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n-1} T(k) + O(n)$$

Average case analysis (cont'd)

- Use substitution method: Assume T(k) $\leq ck$, for sufficiently large c
- $T(n) \leq \frac{2}{n}\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} ck + an$

$$= \frac{2c}{n}(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k) + an$$

$$= \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{(\lfloor \frac{n}{2} \rfloor - 1)\lfloor \frac{n}{2} \rfloor}{2}\right) + an$$

Average case analysis (cont'd)

- $T(n) \leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\left\lfloor\frac{n}{2}\right\rfloor - 1\right)\left\lfloor\frac{n}{2}\right\rfloor}{2}\right) + an$

$$\leq \frac{2c}{n}\left(\frac{(n-1)n}{2} - \frac{\left(\frac{n}{2}-2\right)\left(\frac{n}{2}-1\right)}{2}\right) + an$$

$$= \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{\frac{n^2}{4}-\frac{3n}{2}+2}{2}\right) + an$$

$$= \frac{c}{n}\left(\frac{3n^2}{4} + \frac{n}{2} - 2\right) + an$$

Average case analysis (cont'd)

- $T(n) \leq \frac{c}{n}\left(\frac{3n^2}{4} - \frac{n}{2} - 2\right) + an$

$= c\left(\frac{3n}{4} + \frac{1}{2} - \frac{2}{n}\right) + an$

$\leq \frac{3cn}{4} + \frac{c}{2} + an$

$\leq \frac{3cn}{4} + \frac{n}{2} + an$ when $n \geq c$

$= cn - \left(\frac{cn}{4} - \frac{n}{2} - an\right)$

$\leq cn$ when

$\frac{cn}{4} - \frac{n}{2} - an \geq 0 -> c \geq 4a + 2$

Worst-Case Linear-Time Selection

- Randomized selection algorithm works well in practice
- We now examine a selection algorithm whose running time is O(n) in the worst case.

Worst-Case Linear-Time Selection

- The worst-case happens when a 0:n-1 split is generated. Thus, to achieve O(n) running time, we *guarantee* a good split upon partitioning the array.
- Basic idea:
  - Generate a good partitioning element

Selection algorithm

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 $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

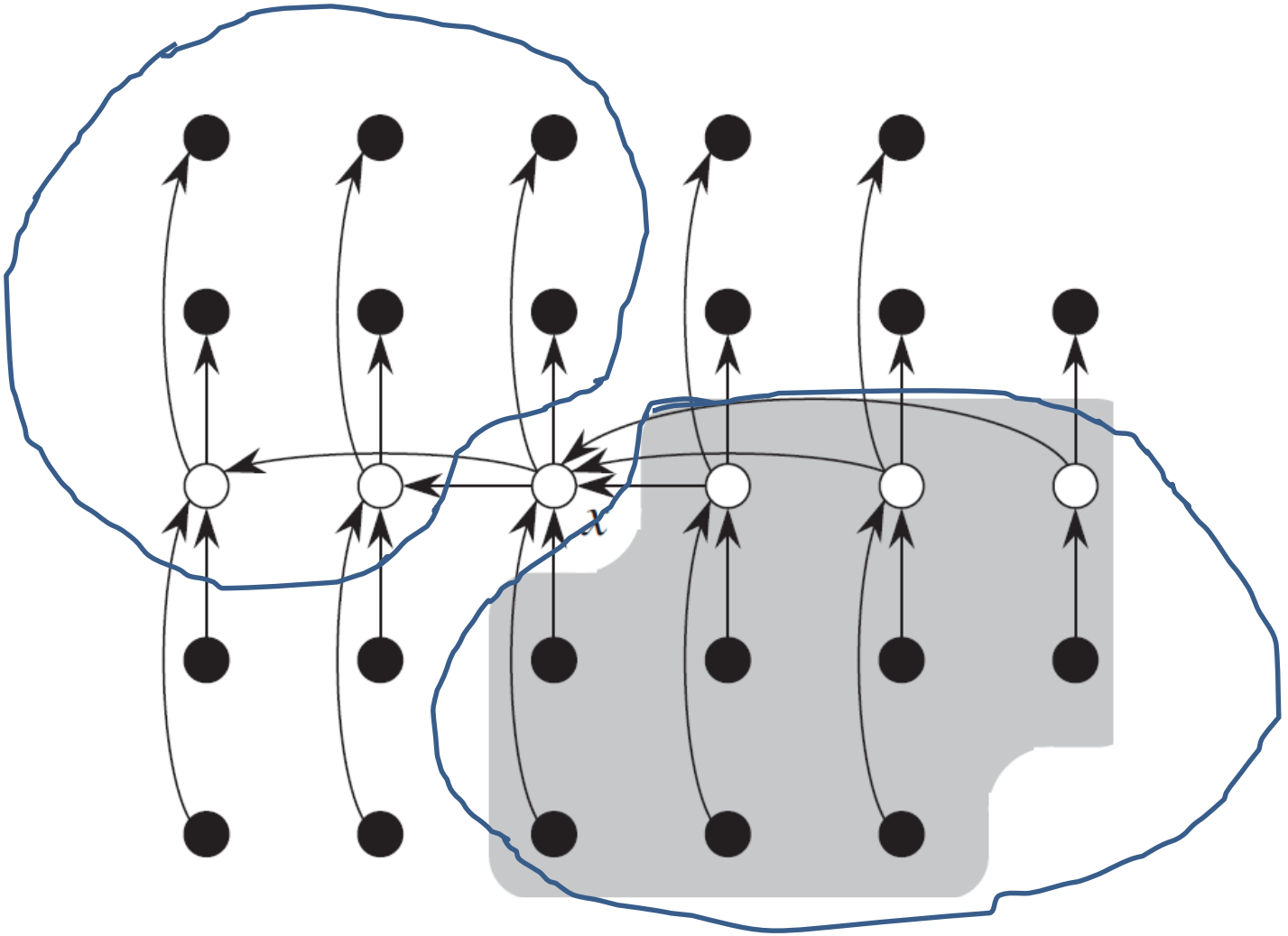
   if (i < k) **then**

   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

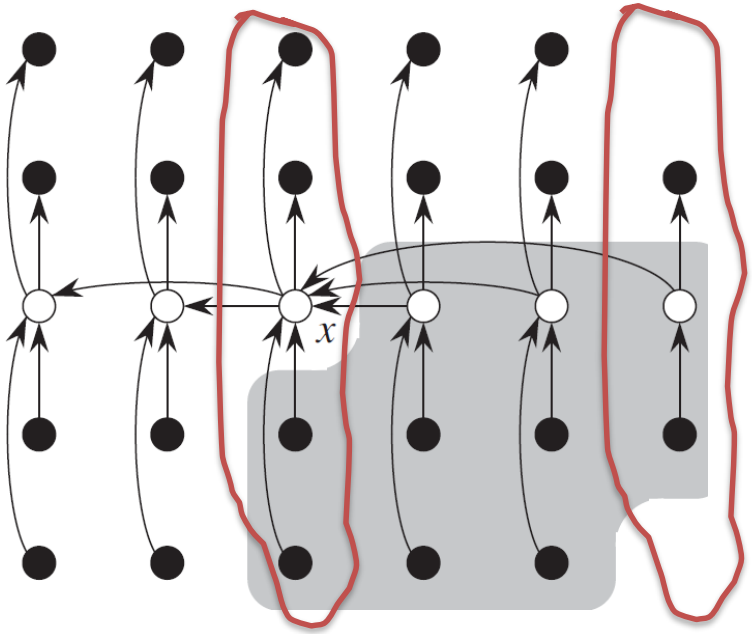
   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Example



Running time analysis

- At least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x,
  - except for the one group that has fewer than 5 elements, and the one group containing x itself



Running time analysis (Cont'd)

- The number of elements greater than x is at least

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rceil -2\right) \geq \frac{3n}{10} - 6$$

- At most $\frac{7n}{10} + 6$ elements are less than x. This means that, in the worst case, step 5 calls SELECT recursively on at most $\frac{7n}{10} + 6$ elements.

Running time analysis (cont'd)

- Step 1 takes O(n) time
- Step 2 consists of O(n) calls of insertion sort on sets of size O(1)
- Step 3 takes time T($\lceil n/5 \rceil$)
- Step 4 takes O(n) time
- Step 5 takes time at most T(7n/10 + 6)

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 ⌈n/5⌉ 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 the low side of the partition
   else
   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis (cont'd)

- We can therefore obtain the recurrence
- $T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$
- Assume $T(k) \leq ck$ for $k < n$, use the substitution method
- $T(n) \leq c\lceil n/5 \rceil + c(7n/10 + 6) + an$

  $\leq cn/5 + c + 7cn/10 + 6c + an$

  $= 9cn/10 + 7c + an$

  $= cn + (-cn/10 + 7c + an)$

Running time analysis (cont'd)

- $T(n) \leq cn + (-cn/10 + 7c + an)$

  $\leq cn + (-cn/10 + 7n + an)$ when $n \geq c$
- Which is at most cn if
  - $-cn/10 + 7n + an \leq 0$ when $c \geq 70 + 10a$

Worst-case Quicksort

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - T(n) = 2T(n/2) + O(n) = O(n lg n)

Worst-case Quicksort (Cont'd)

```
Quicksort(A, p, r)
{  if (p < r)
    {
        q = Median-Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}
Median-Partition(A, p, r)
{       median = ⌈(p-r+1)/2⌉;
        x = Select(A, p, r, median);
        i = rank(x);
       exchange A[r] and A[i]
        return Partition(A, p, r)
}
```

Summary

- Selection() does not require assumptions on the input
  - Do not need to sort the whole array, then pick i-th element
  - Counting/Radix/Bucket sort assume certain inputs

COT 6405 Introduction to Theory of Algorithms

Midterm II review

Overview

- Exam time: Nov 5th 3:30pm to 4:45pm
- Exam location: ENB 118 (regular session) and ENB 313 (online session)
- Coverage:
  - Lectures 8, 9, 10, 11, and midterm II review

Quicksort

- Sorts “in place”
  - Only a constant number of elements stored outside the sorted array
- Sorts O(n lg n) in the average case
- Sorts $O(n^2)$ in the worst case
- So why people use it instead of merge sort?
  - Merge sort does not sort “in place”

Quicksort Code

```
Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q-1);
        Quicksort(A, q+1, r);
    }
}  // what is the initial call?
```

Partition

- Clearly, all the actions take place in the **`partition()`** function
  - Rearranges the subarray “in place”
  - End result:
    - Two subarrays
    - All values in 1st subarray < all values in 2nd
  - Returns the index of the “pivot” element separating the two subarrays
- How do we implement this function?

Partition array $A[p..r]$

```
PARTITION(A, p, r)
        x ← A[r]          // select the pivot
        i ← p – 1
        for j ← p to r – 1
                if A[ j ] ≤ x
                        i ← i + 1
                        exchange A[i ] ↔ A[ j ]
        // move the pivot between the two subarraies
        exchange A[i + 1] ↔ A[r]
        // return the pivot
        return i + 1
```

What is the running time of `partition()`?

Performance of quicksort

- The running time of quicksort depends on the partitioning of the subarrays:
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.
  - If the subarrays are balanced, then quicksort can run as fast as mergesort. The following inequality is used for the average case analysis of quicksort

$$\sum_{k=1}^{k=n-1} klgk \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$$

Improving quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on an already-sorted input
- How to avoid this?
- Two solutions
  - Randomize the input array
  - Pick a random pivot element
- How will these solve the problem?
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time

Randomized version of quicksort

- We add randomization to quicksort.
  - We could randomly permute the input array: very costly
  - Instead, we use **random sampling** to pick one element at random as the pivot
    - Don't always use $A[r]$ as the pivot.

Analysis of quicksort

- We analyzed
  - the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT
  - the expected (average-case) running time of QUICKSORT and RANDOMIZED-QUICKSORT

Worst-case analysis

- We saw a worst-case split (0:n-1) at every level of recursion in quicksort produces a $\Theta(n^2)$ running time, which,
  - Intuitively, is the worst-case running time
- We have prove this assertion

Average case analysis

- The dominant cost of the algorithm is partitioning.
- What is the maximum number of calls to the function PARTITION?
  - PARTITION is called at most n times.

Average case analysis (cont'd)

Lemma 7.1: Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

The amount of work of each call to PARTITION is a constant plus the number of comparisons performed in its for loop
The expectation of X is the average case running time, and
E(x) = O(nlgn)

Decision trees

- We can view comparison sorts abstractly in terms of decision trees
  - A decision tree is a full binary tree that represents the comparisons between elements
  - Each node on the tree is a comparison of *i*:*j*, i.e., a_i v.s. a_j

Theorem 8.1

- Any comparison sort algorithm requires $\Omega(nlgn)$ comparisons in the worst case
- How to prove?
  - By proving that the height of the decision tree is $\Omega(nlgn)$
  - What's the # of leaves of a decision tree? l = ?
  - What's the maximum # of leaves of a general binary tree? l_{max} = ?

Proof

- $l_{min} = n!$ and $l_{max} = 2^h$
- Clearly, the minimum # of leaves l_{min} is less than or equal to the maximum # of leaves, l_{max}
- So we have: $n! \leq 2^h$
- Taking logarithms: $\lg(n!) \leq h$

Proof (cont'd)

- Stirling's approximation tells us:

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

- Thus, $h \geq \lg(n!)$

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

Sorting in linear time

- Counting sort
  - No direct comparisons between elements!
  - Depends on assumption about the numbers being sorted
    - We assume numbers are in the range [0.. k]
  - The algorithm is NOT "in place"
    - Input: A[1..*n*], where A[j] $\in$ {0, 2, 3, …, *k*}
    - Output: B[1..*n*], sorted
    - Auxiliary counter storage: Array C[0..*k*]
    - notice: A[], B[], and C[] → not sorting in place

Counting sort

```
1  CountingSort(A, B, k)
2     for i= 0 to k    // counter initialization
3          C[i]= 0;
4     for j= 1 to A.length
5          C[A[j]] += 1;
6     for i= 1 to k    // aggregate counters
7          C[i] = C[i] + C[i-1];
8     for j= A.length downto 1 //move results
9          B[C[A[j]]] = A[j];
10         C[A[j]] -= 1;
```

Counting sort

- Total time: $O(n + k)$
  - Usually, $k = O(n)$ → $k < c\ n$
  - Thus counting sort runs in $O(n)$ time
- But sorting is $\Omega(n \lg n)$! Contradiction?
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is stable
    - The elements with the same value is in the same order as the original
    - index $i < j$, $a_i = a_j$ → new index $i' < j'$

Counting Sort

- Why don't we always use counting sort?
- Because it depends on range *k* of elements
- Could we use counting sort to sort 32 bit integers? Why or why not?
- Answer: no, *k* too large (2^{32} = 4,294,967,296)
  - We need huge arrays, e.g., C[4,294,967,296]?
  - k >> n → O(n+k) = O(k)

Least significant digit (LSD) Radix Sort

- Key idea: sort the least significant digit first
- Assume we have d-digit numbers in A

```
RadixSort(A, d)
        for i= 1 to d
            StableSort(A) on digit i
```

Radix Sort

- What sort will we use to sort on digits?
- 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 the total time O(dn+dk)
  - When d is constant and k= O(n), takes O(n) time

How to break words into digits?

- We have n word
- Each word is of b bits
- We break each word into r-bit digits, d = $\lceil b/r \rceil$
- Using counting sort, k = 2^r -1
- E.g., 32-bit word, we break into 8-bit digits
  - d = $\lceil 32/8 \rceil$ = 4, k = 2^8 -1 = 255
- T(n) = $\Theta(\ d*(n+k)\ ) = \Theta(b/r * (n + 2^r))$

How to choose r?

How to choose r? Balance b/r and $n + 2^r$. Choosing $r \approx \lg n$ gives us $\Theta\left(\frac{b}{\lg n}(n+n)\right) = \Theta(bn/\lg n)$.

Still in O(n)

- If we choose $r < \lg n$, then $b/r > b/\lg n$, and $n + 2^r$ term doesn't improve.
- If we choose $r > \lg n$, then $n + 2^r$ term gets big. Example: $r = 2\lg n \Rightarrow 2^r = 2^{2\lg n} = (2^{\lg n})^2 = n^2$.

Bucket Sort

- Assumes the input is generated by a random process that distributes elements uniformly over [0, 1).
- ***Idea:***
  - Divide [0, 1) into n equal-sized *buckets*.
  - Distribute the n input values into the buckets.
  - Sort each bucket.
  - Then go through buckets in order, listing elements in each one.

Bucket Sort (cont'd)

- Input:
  - $A[1 . . n]$, where $0 \le A[i] < 1$ for all i .
- Auxiliary array:
  - $B[0 . . n - 1]$ of linked lists, each list initially empty.

Bucket sort Implementation

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n

  do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$

for $i \leftarrow 0$ **to** $n - 1$

  do sort list $B[i]$ with insertion sort

concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order

return the concatenated lists

Easily compute the bucket index $\lfloor n \cdot A[i] \rfloor$

Formal Analysis

- Define a random variable:

  n_i = the number of elements placed in bucket $B[i]$
- Because insertion sort runs in quadratic time, bucket sort time is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) .$$

Formal Analysis (Cont'd)

Take expectations of both sides:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] && \text{(linearity of expectation)} \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(\mathrm{E}[n_i^2]) && (\mathrm{E}[aX] = a\mathrm{E}[X])
\end{aligned}
$$

n_i = the number of elements placed in bucket $B[i]$

n_i = the number of elements placed in bucket $B[i]$

Claim

$\mathrm{E}[n_i^2] = 2 - (1/n)$ for $i = 0, \ldots, n-1$.

Proof of claim

Define indicator random variables:

- $X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$
- $\Pr\{A[j] \text{ falls in bucket } i\} = 1/n$
- $n_i = \sum_{j=1}^{n} X_{ij}$

$X_{i,j}$ = I{A[j] falls in bucket i}.

= { 1 if A[j] falls in bucket i

0 if A[j] doesn't fall in bucket i

The Claim

Then

$$\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} \mathrm{E}\left[X_{ij}^2\right] + 2\sum_{j=1}^{n-1}\sum_{k=j+1}^{n} \mathrm{E}\left[X_{ij}X_{ik}\right] \quad \text{(linearity of expectation)}
\end{aligned}$$

$$(x_1+x_2+x_3)(x_1+x_2+x_3)$$
$$= x_1^2 + x_1x_2 + x_1x_3$$
$$+ x_2^2 + x_1x_2 + x_2x_3$$
$$+ x_3^2 + x_1x_3 + x_2x_3$$

$$\begin{aligned}
\mathrm{E}\left[X_{ij}^2\right] &= 0^2 \cdot \Pr\{A[j] \text{ doesn't fall in bucket } i\} + 1^2 \cdot \Pr\{A[j] \text{ falls in bucket } i\} \\
&= 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} \\
&= \frac{1}{n}
\end{aligned}$$

Analysis

$\mathrm{E}[X_{ij} X_{ik}]$ for $j \neq k$: Since $j \neq k$, X_{ij} and X_{ik} are independent random variables

$$\begin{aligned} \Rightarrow \mathrm{E}[X_{ij} X_{ik}] &= \mathrm{E}[X_{ij}]\,\mathrm{E}[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} \end{aligned}$$

Therefore:

$$\mathrm{E}[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + 2 \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \frac{1}{n^2}$$

Analysis (Cont'd)

$$
\begin{aligned}
&= n \cdot \frac{1}{n} + 2\binom{n}{2}\frac{1}{n^2} \\
&= 1 + 2 \cdot \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 1 + 1 - \frac{1}{n} \\
&= 2 - \frac{1}{n}
\end{aligned}
$$

■ (claim)

Therefore:

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(2 - 1/n) \\
&= \Theta(n) + O(n) \\
&= \Theta(n)
\end{aligned}
$$

Order statistic

- 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, we have 2 medians: lower median $n/2$ and upper median n/2+1
    - By our convention, “median” normally refers to the lower median

Can we reduce the cost?

- Can we find the minimum and maximum with less than twice the cost, 2(n-1) ?
- Yes: Walk through elements by pairs
  - Compare each element in pair to the other
  - Compare the larger one to maximum, the smaller one to minimum
- Total cost: 3 comparisons per 2 elements = O(3n/2)

Finding order statistics: The Selection Problem

- A more interesting problem is the selection problem
  - finding the *i*-th smallest element of a set
- A naïve way is to sort the set
  - Running time takes O(nlgn)
- We will study a practical randomized algorithm with O(n) expected running time
- We will then study an algorithm with O(n) worst-case running time

Randomized Selection

- Key idea: use partition() from Quicksort
  - But, only need to examine one subarray
  - This savings shows up in running time: O(n)
- We will again use a randomized partition

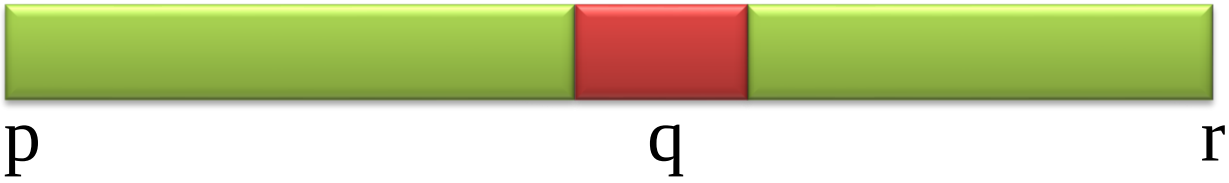
q = RANDOMIZED-PARTITION*(A, p, r)*

RANDOMIZED-PARTITION*(A, p, r)*

i ←RANDOM*(p, r)*

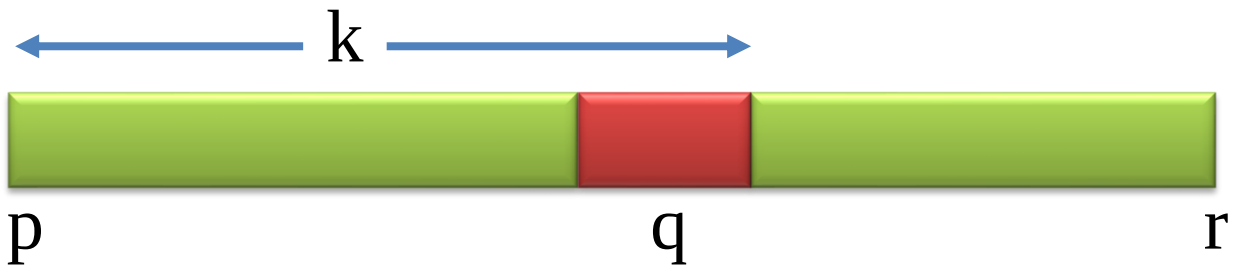
exchange *A*[*r*] ↔ *A*[*i*]

return PARTITION*(A, p, r)*



Randomized Selection

```
RandomizedSelect(A, p, r, i)
    if (p == r) then return A[p];
    q = RandomizedPartition(A, p, r)
    k = q - p + 1;
    if (i == k) then return A[q];
    if (i < k) then
        return RandomizedSelect(A, p, q-1, i);
    else
        return RandomizedSelect(A,q+1,r, i-k);
```



Analyzing Randomized-Select()

- Worst case: partition always 0:n-1
  - $T(n) = T(n-1) + O(n) = O(n^2)$
  - No better than sorting!
- "Best" case: suppose a 9:1 partition
  - $T(n) = T(9n/10) + O(n) = O(n)$ (why?)
  - Master Theorem, case 3
  - Better than sorting!

Worst-Case Linear-Time Selection

- Randomized selection algorithm works well in practice
- We now examine a selection algorithm whose running time is O(n) in the worst case.

Worst-Case Linear-Time Selection

- The worst-case happens when a 0:n-1 split is generated. Thus, to achieve O(n) running time, we *guarantee* a good split upon partitioning the array.
- Basic idea:
  - Generate a good partitioning element

Selection algorithm

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 $\lceil n/5 \rceil$ medians
4. Partition the *n* elements around *x*. Let *k* = rank(*x*)
5. **if** (i == k) **then** return x

   if (i < k) **then**

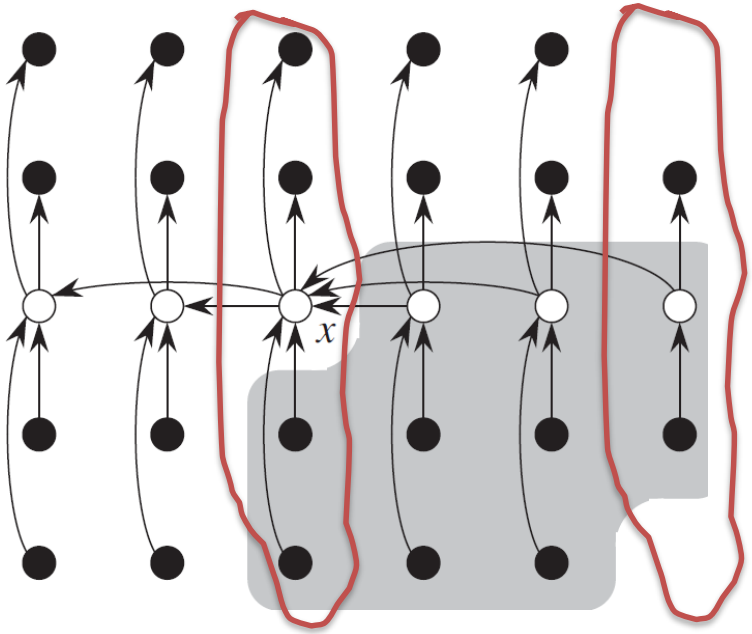
   use Select() recursively to find *i*-th smallest element in the low side of the partition

   else

   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis

- At least half of the $\lceil n/5 \rceil$ groups contribute at least 3 elements that are greater than x,
  - except for the one group that has fewer than 5 elements, and the one group containing x itself



Running time analysis (Cont'd)

- The number of elements greater than x is at least

$$3(\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\text{-}2) \geq \frac{3n}{10} - 6$$

- Similarly, at least $\frac{3n}{10} - 6$ elements are less than x. Thus, in the worst case, step 5 calls SELECT recursively on at most $\frac{7n}{10} + 6$ elements.

Running time analysis (cont'd)

- Step 1 takes O(n) time
- Step 2 consists of O(n) calls of insertion sort on sets of size O(1)
- Step 3 takes time T($\lceil n/5 \rceil$)
- Step 4 takes O(n) time
- Step 5 takes time at most T(7n/10 + 6)

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 ⌈n/5⌉ 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 the low side of the partition
   else
   (i > k) use Select() recursively to find (*i*-*k*)-th smallest element in the high side of the partition

Running time analysis (cont'd)

- We can therefore obtain the recurrence
- T(n) $\leq$ T($\lceil n/5 \rceil$) + T(7n/10 + 6) + O(n)
- Assume T(k) $\leq$ ck for k < n, use the substitution method
- T(n) $\leq$ c$\lceil n/5 \rceil$ + c(7n/10 + 6) + an

  $\leq$ cn/5 + c + 7cn/10 + 6c + an

  = 9cn/10 + 7c +an

  = cn + (-cn/10 + 7c + an)

Running time analysis (cont'd)

- T(n) $\leq$ cn + (-cn/10 + 7c + an)
- Which is at most cn if
  - -cn/10 + 7c + an ≤ 0
  - c $\geq 10a(n/(n-70))$ when n > 70

Linear-Time Median Selection

- Given a “black box” O(n) median algorithm, what can we do?
  - *i*-th order statistic:
    - Find median *x*
    - Partition input around *x*
    - if ($i \leq (n+1)/2$) recursively find *i*-th element of first half
    - else find (*i* - (n+1)/2)-th element in second half
    - T(n) = T(n/2) + O(n) = O(n) (why?)

Worst-case quicksort

- Worst-case O(n lg n) quicksort
  - Find median *x* and partition around it
  - Recursively quicksort two halves
  - T(n) = 2T(n/2) + O(n) = O(n lg n)
  - Input assumption?