COMP2013
Data Structures and Algorithms

Lecture 4
Sorting Algorithms

Dr. Jieming Shi
COMP2013

Outline

- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - **Master method**
- Quicksort
- Lower bound of comparison sorts
- Sort with additional information

Master Method (Just rules to remember)

- Given the recurrence: $T(n) = a\,T(n/b) + f(n)$
  - $f(n)$ must be ≥ 0
- Compare $f(n)$ and $n^{\log_b a}$
  - ε, v are some constants such that $\varepsilon > 0$, $v < 1$

$$T(n) = \begin{cases} O(n^{\log_b a} \log n) & \text{if } f(n) = \Theta(n^{\log_b a}) \\ O(n^{\log_b a}) & \text{if } n^{\log_b a} / f(n) = \Omega(n^{\varepsilon}) \\ O(f(n)) & \text{if } f(n) / n^{\log_b a} = \Omega(n^{\varepsilon}) \text{ and } a\,f(n/b) / f(n) \leq v \end{cases}$$

polynomially larger

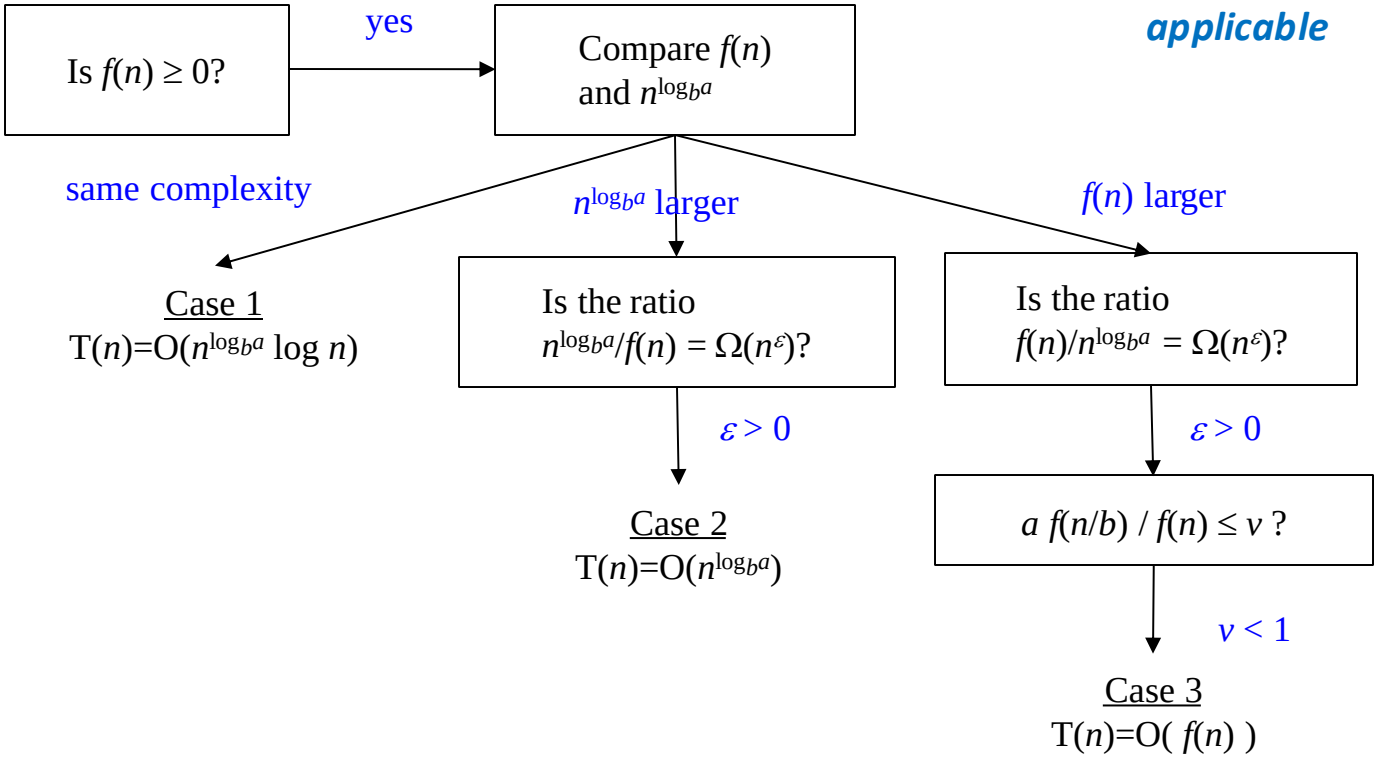
polynomially larger

regularity condition

Master Method

$T(n) = a\,T(n/b) + f(n)$

There exist cases where master method is not applicable



Master Method

- Given the recurrence $T(n) = 2\,T(n/2) + c\,n$
  - $a = 2,\ b = 2,\ n^{\log_b a} = n^{\log_2 2} = n$
  - $f(n) = c\,n$
  - Since $f(n) = \Theta(n^{\log_b a})$, we apply *Case 1* and get:
    - $T(n) = O(n^{\log_b a} \log n) = O(n \log n)$
- Given the recurrence $T(n) = 4\,T(n/2) + c\,n$
  - $a = 4,\ b = 2,\ n^{\log_b a} = n^{\log_2 4} = n^2$
  - $f(n) = c\,n$
  - Since $n^{\log_b a}/f(n) = \Omega(n^1)$, we apply *Case 2* and get:
    - $T(n) = O(n^2)$

Overview of these methods

- Recursion tree method
  - Draw a tree with the time used at each recursion level, then sum up the times at all levels
  - ☹ need to draw a tree and find the sum; some trees are hard to draw
- Substitution method
  - Make a good guess (usually it should be correct)
  - Prove the guess is correct
  - ☹ wrong guess is in vain. Know how to prove.
- Master method
  - Obtain T(n) by using a theorem directly
  - ☹ not applicable in some cases

Some functions for your reference

- $\log_b c/d = \log_b c - \log_b d$
- $\log_b a = \log_x a / \log_x b$ (new base x: any positive number)
- $\log_b a^r = r \log_b a$
- $a^{\log_b n} = n^{\log_b a}$

$$\text{let} \quad F = a^{\log_b n}$$

$$\log_b F = \log_b a^{\log_b n}$$

$$\log_b F = \log_b n \log_b a$$

$$F = n^{\log_b a}$$

- $\sum_{i=1..n} i = n(n+1)/2$
- $\sum_{i=0..n-1} a\, r^i = a(1-r^n)/(1-r)$
  - when $|r| > 1$, the series becomes larger and larger
  - when $|r| < 1$, the series converges to $a/(1-r)$

Outline

- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method
- **Quicksort**
- Lower bound of comparison sorts
- Sort with additional information

Sorting problem

- **Input**: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - Problem size: n
- **Output**: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- Sort the records by age (*key* to sort), while the remainder of a record consists of satellite data

Record

Student ID	Name	Age	GPA	Year
101	bob	23	3.5	1
102	dave	22	3.3	2
105	adam	23	3.9	1
103	jack	21	3.6	3
106	tony	22	3.7	1

Sorting Algorithms

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Quicksort idea

- Divide
  - **Partition** (rearrange) array $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$
  - Element $A[q]$ is *pivot*
  - *Low* side: each elements in $A[p:q-1] \leq A[q]$
  - *High* side: each elements in $A[q+1:r] \geq A[q]$
  - Subarrays are possibly empty
  - Index q needs to be computed
- Conquer
  - Call quicksort recursively over the two subarrays $A[p:q-1]$ and $A[q+1:r]$
- Combine
  - Do nothing. Why?
  - When combining, two subproblems are already solved, i.e., $A[p:q-1]$ and $A[q+1:r]$ are already sorted. They also have the correct order w.r.t pivot $A[q]$. Thus, $A[p:r]$ is sorted.

Quicksort

- Divide
  - **Partition** (rearrange) array $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$
  - Element $A[q]$ is *pivot*
  - *Low* side: each elements in $A[p:q-1] \leq A[q]$
  - *High* side: each elements in $A[q+1:r] \geq A[q]$
  - Subarrays are possibly empty
  - Index q needs to be computed
- Conquer
  - Call quicksort recursively over the two subarrays $A[p:q-1]$ and $A[q+1:r]$

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

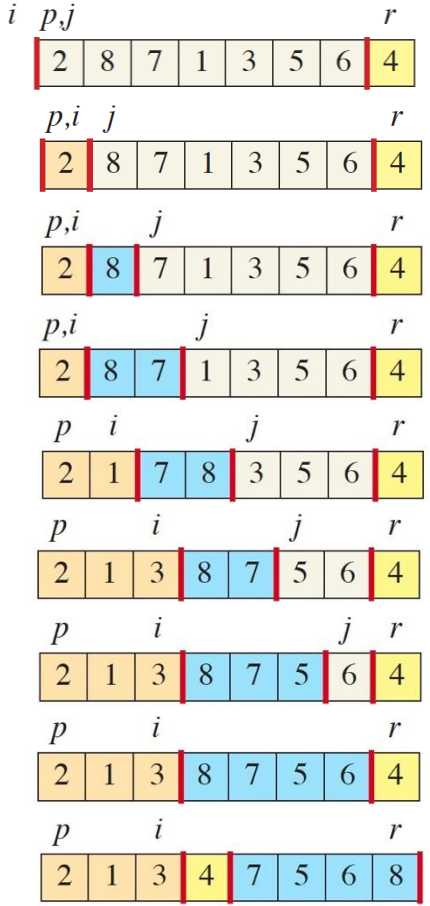
What is the base case?

How to partition?

Quicksort

- **Partition** (rearrange) array $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$
- Element $A[q]$ is *pivot*
- *Low* side: each elements in $A[p:q-1] \leq A[q]$
- *High* side: each elements in $A[q+1:r] \geq A[q]$

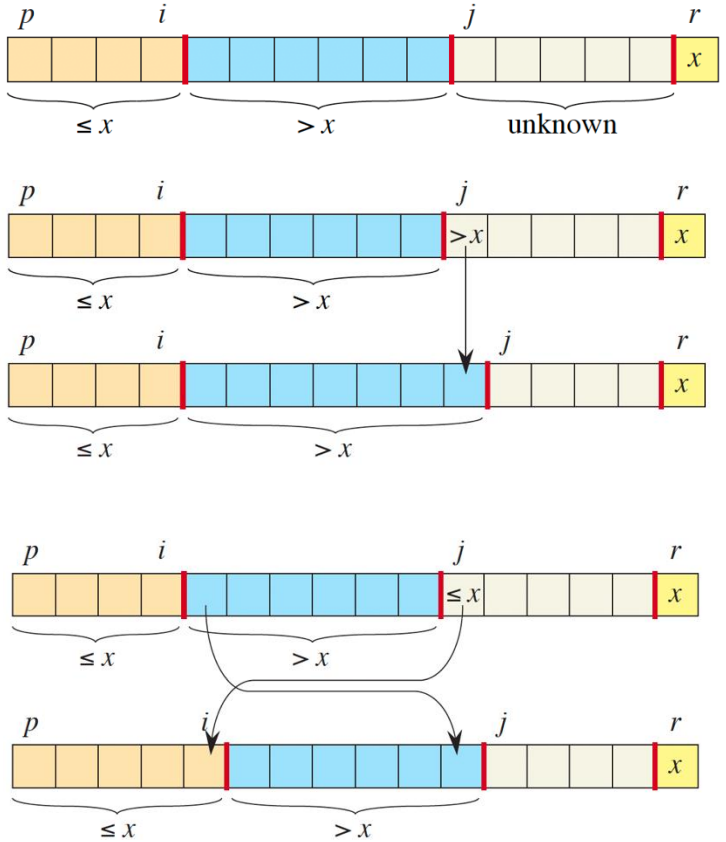
- Let $x = A[r]$ be the pivot value
- 4 regions in $A[p:r]$ during partition
  - low side, high side, unknown, pivot
- i is the highest index of low side
- j is the index of the current element to be compared with pivot
  - j starts from p to $r-1$
  - If $A[j] \leq x$, we move $A[j]$ to the low side
- Finally, we put pivot x between low side and high side at (i+1) position



Quicksort

- An element falls into exactly one region
- some regions can be empty

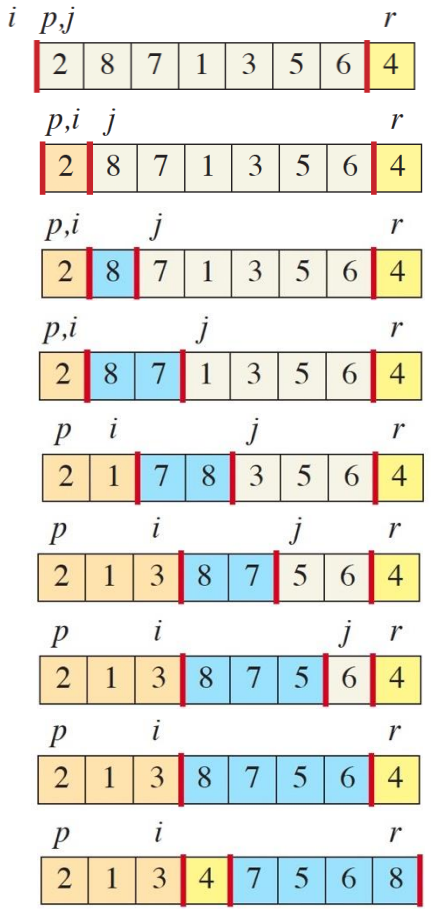
- low side, high side, unknown, pivot



Quicksort

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

What is the running time of this partition algorithm?

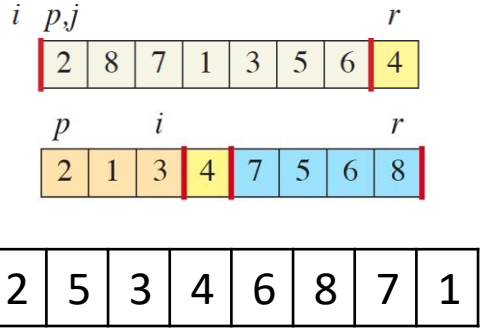


Quicksort running time

- Divide (Line 3)
  - Main computation
  - What sizes will the two subproblems have?
    - Sizes: m and $n-1-m$; total is *n-1*
- Quicksort running time recurrence:
  - $T(n) = \max\{T(m) + T(n-1-m) : 0 \le m \le n-1\} + \Theta(n)$

```
QUICKSORT(A, p, r)
1  if p < r
2       // Partition the subarray around the pivot, which ends up in A[q].
3       q = PARTITION(A, p, r)
4       QUICKSORT(A, p, q − 1)  // recursively sort the low side
5       QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Quicksort running time



- Divide (Line 3)
  - Sizes of two subproblems:
    - Sizes: m and $n-1-m$; total is *n-1*
    - *Balanced partition?*
    - *Unbalanced partition?*
- Quicksort running time recurrence:
  - $T(n) = \max\{T(m) + T(n-1-m) : 0 \le m \le n-1\} + \Theta(n)$

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Quicksort running time

- Divide (Line 3)
  - A worst-case partition (low side and high side)
    - Sizes: 0 and $n-1$;

2	5	3	4	6	8	7	1

- If the worst-case partition happens in *every* recursive call:
  - $T(n) = \max\{T(m) + T(n-1-m): 0 \le m \le n-1\} + \Theta(n)$
  - $T(n) = T(0) + T(n-1) + \Theta(n)$
    $= \Theta(n^2)$

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

What kind of array is the worst case of quick sort?

Quicksort: Worst-Case Input

- Is the following array the worst-case input (at n=8)?
  - Worst-case is sorted arrays.

13	26	29	40	47	59	70	85

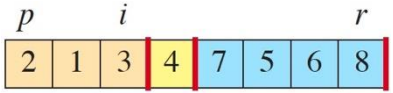
Quicksort worst-case running time

- Guess $O(n^2)$ for $T(n) = \max\{T(m) + T(n-1-m): 0 \le m \le n-1\} + \Theta(n)$
  - (Previous slides do not contain a formal proof)
- Prove by substitution
  - Assume $T(n') \le cn^2$ holds for some constant $c > 0$ for $n' < n$
  - $T(n) = \max\{T(m) + T(n-1-m): 0 \le m \le n-1\} + \Theta(n)$
    $\le \max\{cm^2 + c(n-1-m)^2: 0 \le m \le n-1\} + \Theta(\mathrm{n})$
    $= c\ \max\{(\mathrm{n}-1)^2 - 2m(n-1) + 2m^2: 0 \le m \le n-1\} + \Theta(\mathrm{n})$
    $= c\ \max\{(\mathrm{n}-1)^2 + 2m(m-(n-1)): 0 \le m \le n-1\} + \Theta(\mathrm{n})$
    $\le c(n-1)^2 + \Theta(n) \le cn^2$ (when c is large enough)
  - Base case: T(1) trivially holds

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Quicksort running time

- Divide (Line 3)
  - A best-case partition (low side and high side)
    - Sizes: $n/2$ and $n/2$;



- If the best-case partition happens in *every* recursive call:
  - $T(n) = 2T(n/2) + \Theta(n)$
    $= \Theta(n \log n)$

Quicksort running time

- $T(n) = \max\{T(m) + T(n-1-m): 0 \le m \le n-1\} + \Theta(n)$
- Balanced partitioning
  - What if subproblems are always with size ratio 9-to-1?
  - $T(n) = T(9n/10) + T(n/10) + \Theta(n)$
    - How to solve this?
    - $O(n \log n)$
  - What about $T(n) = T(99n/100) + T(n/100) + \Theta(n)$?
    - The number of levels in the recursion tree is still $\Theta(\lg\ n)$
    - The cost per level is still $O(n)$
    - $O(n \log n)$

```
QUICKSORT(A, p, r)
1  if p < r
2       // Partition the subarray around the pivot, which ends up in A[q].
3       q = PARTITION(A, p, r)
4       QUICKSORT(A, p, q − 1)  // recursively sort the low side
5       QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

Randomized Quicksort

- How to avoid bad cases?
  - Introduce randomness
- For quicksort, randomization yields a fast and practical algorithm.
  - Randomly choose a pivot
- Expected running time
  - *O(n log n)*

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Sorting Algorithms

- *In place* means that at most a constant number of elements of the input array are ever sorted outside the array
  - Quicksort; Insertion sort;
- Not in place:
  - Merge sort

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Sorting Algorithms

- Insertion sort is fast for small input sizes, since its inner loops are tight (small hidden constant factors)
- Quicksort is a practical choice since it is efficient on average. The hidden constant factors are small in $\Theta(n \lg n)$ notation, though its worst-case running time is $\Theta(n^2)$

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Outline

- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method
- Quicksort
- **Lower bound of comparison sorts**
- Sort with additional information

Comparison sorts

- Insertion sort, merge sort, quicksort, heapsort, bubble sort, and selection sort are all *comparison sorts*
  - Determine the sorted order of an input array **only** by comparing elements

3	8	4	2	6

Lower bound of comparison sorts

- For *any* comparison sort algorithm, it has worst-case lower-bound running time $\Omega(n \log n)$

Assuming all elements are distinct

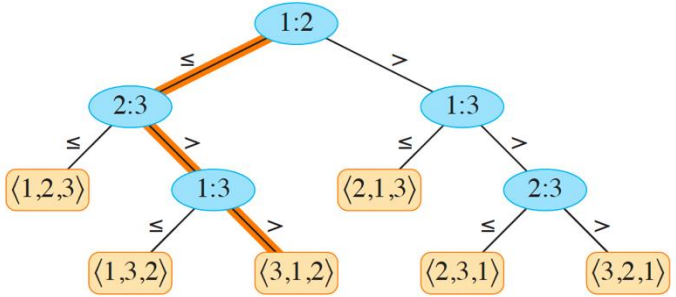
Lower bound of comparison sorts

- Merge sort is asymptotically optimal comparison sort
- Quicksort achieves this on average

Algorithm	Worst-case running time	Average/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

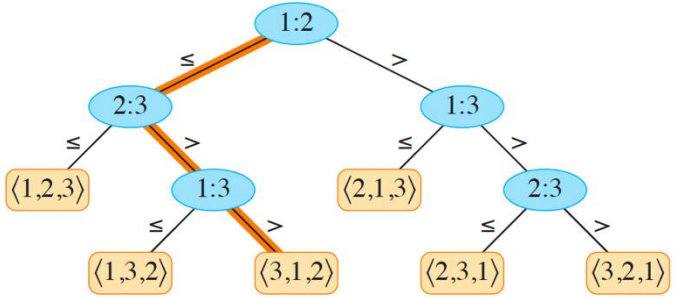
Lower bound of comparison sorts

- To sort three elements a_1, a_2, a_3, how many possible sorting orders? (Hint: permutation)
  - 3!
- A binary decision tree
  - A node has a parent, except root
  - At most two children
  - The height of a node is the max length of paths from the node down to any leaf
  - The height *h* of the tree is the height of the root
- This is not a complete binary tree.
  - (A complete binary tree having all the levels of the tree filled completely except the lowest level nodes which are filled from as left as possible.)



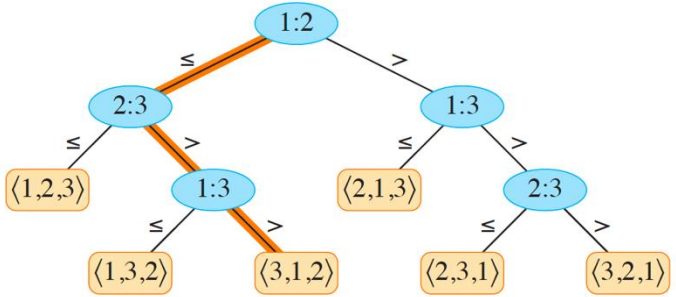
Lower bound of comparison sorts

- To sort three elements a_1, a_2, a_3, how many possible sorting orders? (Hint: permutation)
  - 3!
- A path from root to a leaf is a solution.
  - The length of the path is the running time (i.e., the number of comparisons).
- The height *h* of the tree is the worst-case lower bound running time



Lower bound of comparison sorts

- To sort n elements $a_1, \ldots, a_n$, how many possible sorting orders?
  - $n!$
  - The height h of the tree is the worst-case lower bound running time
  - A binary decision tree that is not complete
  - How many leaves does a complete binary tree with height h have at most?
    - 2^h
  - $n! \leq 2^h$
  - $h \geq \lg n!$
  - Since $\lg n! = \Theta(n \lg n)$, $h = \Omega(n \lg n)$



Outline

- Solving Recurrence
  - Recursive-tree method
  - Substitution method
  - Master method
- Quicksort
- Lower bound of comparison sorts
- **Sort with additional information**

Sort with additional information

- Sort not only by comparison but also additional information

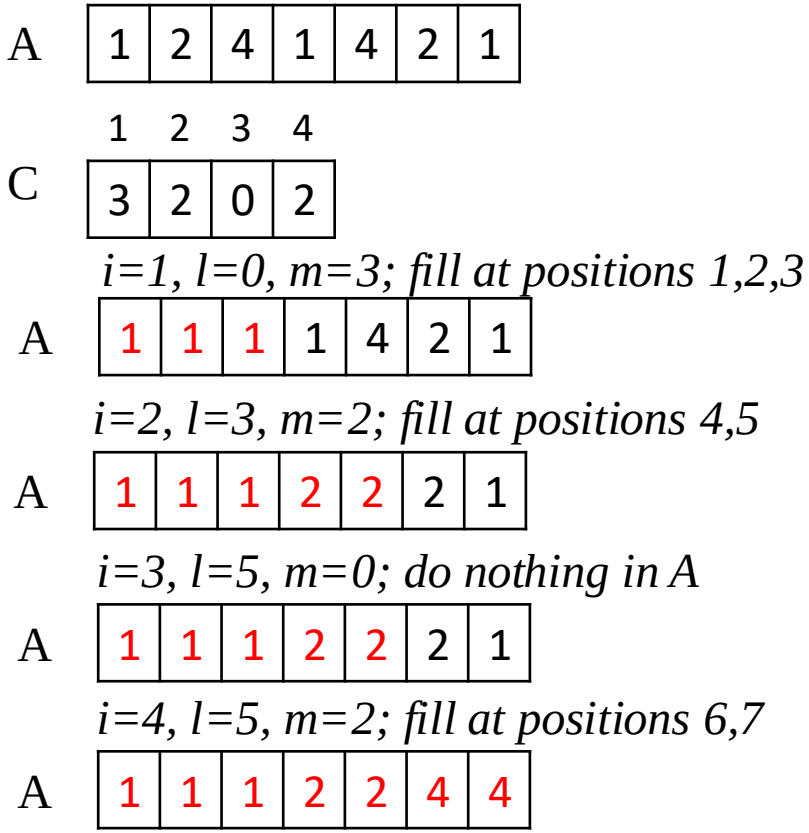
- Sort n elements *that are integers in range 1 to k*
  - Counting sort runs in $\Theta(n + k)$ time, which is $\Theta(n)$, if $k = O(n)$

Sort with additional information

- Sort *n* elements *that are integers in range 1 to k*
  - How to sort this array of integers in [1,4]:
    - {1, 2, 4, 1, 4, 2, 1}?
  - Count the frequency of each integer, and then just place the integers in order and follow their frequencies
    - 1: 3
    - 2: 2
    - 3: 0
    - 4: 2
    - The sorted array is {1, 1, 1, 2, 2, 4, 4}

How to get the frequencies? What is the running time for this?

Sort *n* elements *that are integers in range 1 to k*




```
Counting-Sort(A, n, k)
1.  C[1:k]      //frequency of each integer
2.  For i = 1 to k
3.      C[i]=0
4.  For j = 1 to n
5.      C[A[j]]++
6.  l = 0
7.  For i = 1 to k
8.      m = C[i]
9.      for q=1 to m
10.         A[l+q]=i
11.     l = l+m
```

Sort *n* elements *that are integers in range 1 to k*

- Running time:
  - Line 1-3: O(k)
  - Line 4-5: O(n)
  - Line 6: O(1)
  - Line 7: O(1)* k times
  - Line 8: O(1)* k times
  - Line 9: O(1); how many times?
    - $\sum_{i=1}^{k} C[i] = n$
  - Line 10: O(1); $\sum_{i=1}^{k} C[i] = n$ times
  - Line 11: O(1) * k times
- Total is O(n+k)

```
Counting-Sort(A, n, k)
1.  C[1:k]      //frequency of each integer
2.  For i = 1 to k
3.      C[i]=0
4.  For j = 1 to n
5.      C[A[j]]++
6.  l = 0
7.  For i = 1 to k
8.      m = C[i]
9.      for q=1 to m
10.         A[l+q]=i
11.     l = l+m
```

- Thank you!