Announcements

(1) Test 2

** Dec 01 (Thur) **

2:30pm – 4:30pm

Venue: KB223

Topics:

* Everything up to and including Sorting by comparison; emphasis on topics not in Test 1 *

(2) Programming assignment

- Any language as long as TAs know how to grade it.
- Deadline: 30 Nov (no postponement!!)

(3) Last written assignment

- Deadline: Next Friday (no postponement!!)

Outcome (2): Sorting algorithms

Lower bound on Sorting (by comparisons) & Sorting in linear time

- Bubble sort
- Selection sort
- Insertion sort
- Merge sort
- Quick sort
- Heap sort
- Counting sort
- Radix sort
- Bucket sort

Not discussed yet

Have you noticed that the sorting algorithms we learned so far has the following characteristics?

1 They make no assumptions on the values of the numbers and the sorting algorithms are based on the comparisons between elements.

2 The best algorithm we have learned runs in $O(n \log n)$ time in the worst case and examples exist that require $\Omega(n \log n)$ time.

Have you ever wondered if we can get a better algorithm that runs in $o(n \log n)$ based on comparisons between elements?

In fact, it can be shown that it is NOT possible if the algorithm is based on element comparison

Q: Does it mean that we cannot have a sorting algorithm which runs in o(n log n) time in the worst case?

No, if the algorithm does not base on comparisons between numbers, it may be possible to have faster algorithms.

$$\left.\begin{array}{l}\text{Counting sort}\\ \text{Radix sort}\\ \text{Bucket sort}\end{array}\right\}\ \text{not comparison based algorithm}$$

Counting sort

Assumptions: The given numbers are integers in the range 0 to k.

Idea: If I know that m numbers are less than x, then I know where I should place x in the sorted sequence (position m+1 ?)

Q) How we count the number of elements less than a particular element x?

Q) How to handle duplicate numbers? For example, if you have two numbers of value x, one should be placed at m+1, the other will be at m+2!

Use two arrays C[0..k] (working storage) and B[1..n] (output)

Step 1: Scan A, and store the number of occurrences of i in C[i]

$\boxed{O(n)}$

```
for (i = 1 to n) {
    C[A[i]] = C[A[i]] + 1      // assume C has been initialized to 0
}
```

$\boxed{O(n)}$

e.g. k = 6, A = [5, 4, 3, 5, 6, 1, 3, 2, 5, 6, 4, 2]

C = [0, 1, 2, 2, 2, 3, 2]

Step 2: Determine # of elements $\leq$ i for all i $\geq$ 1 based on C

$\boxed{O(k)}$

```
for (i = 1 to k) {
    C[i] = C[i] + C[i-1]
}
```

C = [0, 1, 3, 5, 7, 10, 12]

Note: C[i] counts also elements = i

Step 3: Place A[j] in the appropriate position using information in C

$\boxed{O(n)}$

```
for (i = n downto 1) {
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] – 1
}
```

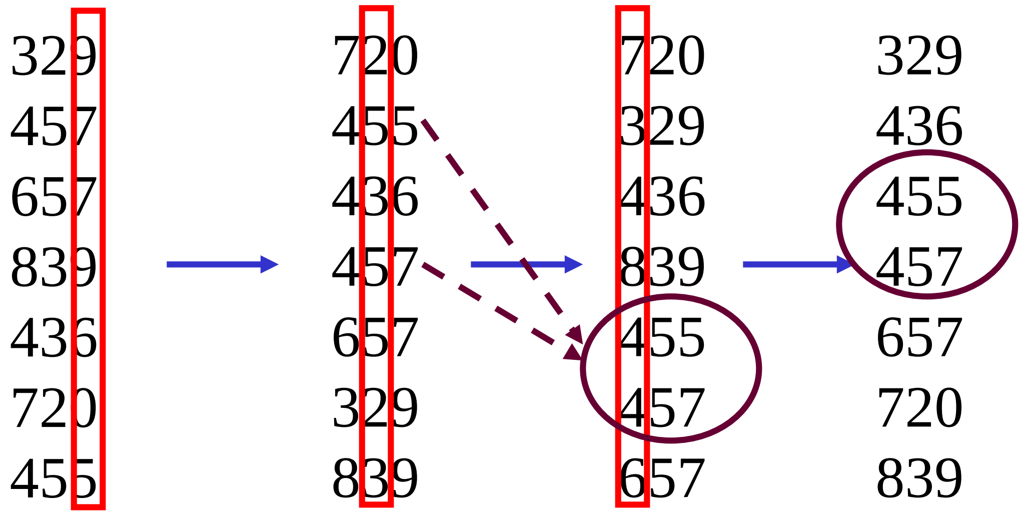
B = [1, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 6]

Total time complexity = O(n+k),
if k = O(n),
the overall time complexity = O(n)

Radix sort

Idea: Sort a set of integers by one digit in each pass. If the integers have d digits, d passes are sufficient to sort them.

Example



Note: It is essential that the relative order of two elements with the same digit being considered is not changed

At each pass, we only look at one digit starting from the least significant digit (the rightmost digit)

For each pass, any available sorting algorithm can be used as long as it maintains the relative order of two elements with the same digit being considered (we call these algorithms “stable”).

A sorting algorithm is stable if numbers with the same value appear in the output array in the same order as they do in the input array.

Q: Is Counting sort stable?

e.g. k = 6, A = [5, 4, 3, 5, 6, 1, 3, 2, 5, 6, 4, 2]

C = [0, 1, 2, 2, 2, 3, 2]

C = [0, 1, 3, 5, 7, 10, 12]

B = [1, 2, 2, 3, 3, 4, 4, 5, 5, 5, 6, 6]

Yes, Counting sort is a stable sort

```
for (i = 1 to n) {
    C[A[i]] = C[A[i]] + 1
}
for (i = 1 to k) {
    C[i] = C[i] + C[i-1]
}
for (i = n downto 1) {
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] – 1
}
```

Q: If we change the last loop in Counting sort to

```
for (i = 1 to n) {
    B[C[A[i]]] = A[i]
    C[A[i]] = C[A[i]] – 1
}
```

Is it still correct? If yes, is it still stable?

```
RadixSort(A, d){     // each integer has d digits
    for i = 1 to d {     // let digit 1 be the least significant (rightmost) digit
        use Counting sort to sort A on digit i; }
}
```

// can be substituted by any stable sort //

<u>Time complexity:</u>
Assume that each digit can have k different values.
Counting sort takes $O(n+k)$ time
So, the overall time complexity is $O(d(n+k))$

If d is a constant and $k = O(n)$, radix sort runs in linear time

Bucket sort

Won't have many numbers in one bucket

Assumption: numbers are uniformly distributed over the interval [0, 1)

Idea:

1) Divide [0, 1) into m equal-sized subintervals (called buckets).
2) Distribute the n numbers into these buckets according to their values.
3) Sort numbers in each bucket separately, say, by insertion sort, then output numbers in each bucket one by one.

Example

Bucket B[0] etc.

0.78	[0.0 – 0.1):		[0.0 – 0.1):
0.17	[0.1 – 0.2): 0.17 => 0.12		[0.1 – 0.2): 0.12 => 0.17
0.39	[0.2 – 0.3): 0.26 => 0.21 => 0.23		[0.2 – 0.3): 0.21 => 0.23 => 0.26
0.26	[0.3 – 0.4): 0.39		[0.3 – 0.4): 0.39
0.72	→ [0.4 – 0.5):		[0.4 – 0.5):
0.94	[0.5 – 0.6):	$\xrightarrow[\text{individually}]{\text{sort bucket}}$	[0.5 – 0.6):
0.21	[0.6 – 0.7): 0.68		[0.6 – 0.7): 0.68
0.12	[0.7 – 0.8): 0.78 => 0.72		[0.7 – 0.8): 0.72 => 0.78
0.23	[0.8 – 0.9):		[0.8 – 0.9):
0.68	[0.9 – 1.0): 0.94		[0.9 – 1.0): 0.94

```
BucketSort(A) {
    for i = 1 to n {
        insert A[i] into Bucket B[$\lfloor mA[i] \rfloor$]
    }
    for i = 0 to m-1 {      // for each bucket
        sort elements in Bucket B[i] using insertion sort
    }
    Concentenate B[0], B[1], ..., B[m-1] in order
}
```

Time Complexity (only a rough idea):

$$T(n) = O(n) + \sum_{j=0}^{m-1} O(n_j{}^2)$$

where n_j is the number of elements in B[j]

If m = O(n), then n_j is probably a constant.
=> T(n) = O(n)

For a formal proof, please take a look at the MIT book

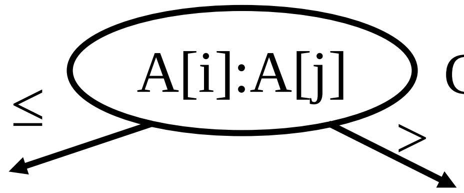
Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst case

We call this a lower bound result and will prove it in this lecture

One technique to prove lower bound result is to use Decision Tree

We can use a decision tree, which is a full binary tree, to represent the comparisons between elements that are performed by a particular sorting algorithm on an input of a given size. How?

Internal node



Compare elements A[i] and A[j] (original position)

Represent the case that $A[i] \leq A[j]$

Represent the case that $A[i] > A[j]$

Root represents the first comparison to be done by the algorithm

Leaf node

$< A[\pi(1)], A[\pi(2)],\ldots A[\pi(n)]>$

Represent a permutation such that the elements are in increasing order, e.g. <A[2], A[1], A[3]>

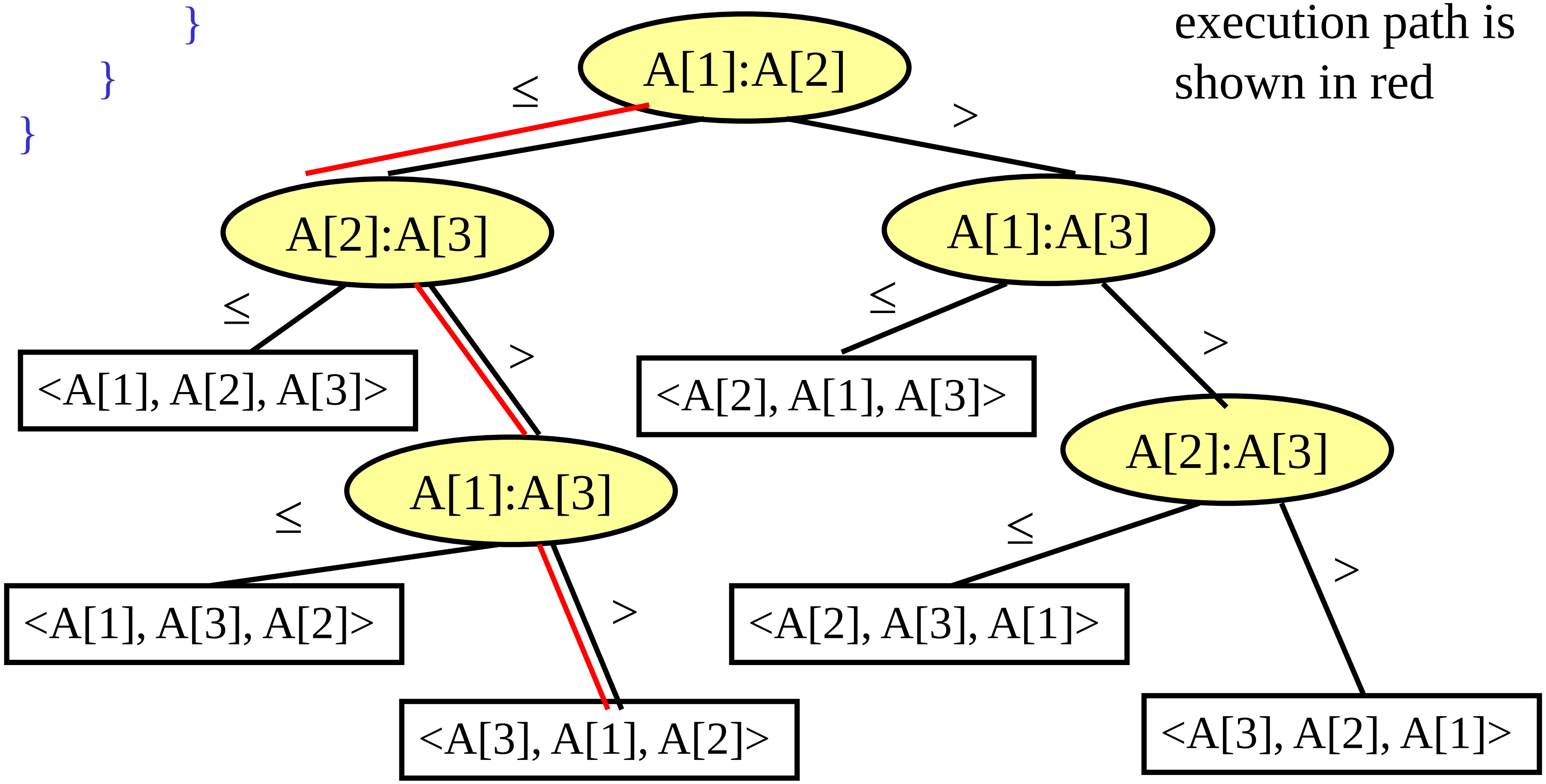
An execution of the algorithm corresponds to a path from root to a leaf

Example

```
InsertionSort(A) {
    for (i = 2 to n) {          // n-1 passes
        j = i-1                 // Insert A[i] into A[1], A[2],..., A[i-1]
        while ((j ≥ 1) and (A[j] > A[j+1])) {
            swap(A[j], A[j+1])
            j--;
        }
    }
}
```

Let n = 3

If A=[6, 8, 5], the execution path is shown in red



All comparison sort algorithms can be represented using this kind of decision tree

What's the implication?

The longest path represents the worst case for a particular algorithm and the length of this path (ie., the height of the tree) is the worst case number of comparisons needed

So, we want to give a lower bound on the height of the tree for *any* possible algorithm

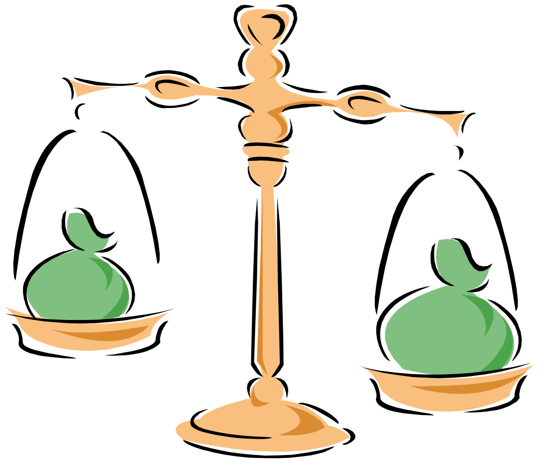
For any algorithm, it must be able to handle all kinds of input sequence, so, # of leaves in the corresponding decision tree
$\geq$ # of possible permutations of n numbers = n!

Let h be the height of the tree. Then, $2^h \geq n!$
$\Rightarrow h \geq \log(n!)$
$\Rightarrow h = \Omega(n \log n)$

In other words, any sorting algorithm that bases only on comparisons between elements takes $\Omega(n \log n)$ time in the worst case.

Exercise

Assuming that we have 9 coins of which one of them is counterfeit and is lighter than others, you are given a balance which can only tell which side of the balance is heavier or both sides are of equal weight.



Q1) In how many weightings, one can identify the counterfeit coin?

Q2) Can you prove that the answer you give in Q1 is the best possible?