Homework 2 Due Thursday Sept 23

- CLRS 6.5-8 (algorithm for merging lists)
- CLRS 7-5 (median of 3 partition)
- CLRS 7-6 (fuzzy sorting of intevals)

Chapter 8: Sorting in Linear Time

Lower Bound on Sorting

For all the sorting algorithms we have seen so far the worst-case running time is $\Omega(n \log n)$.

Heapsort	$n \log n$
Insertion Sort	n^2
Quicksort	n^2
Mergesort	$n \log n$
Randomized Quicksort	$n \log n$ expected

This raises the question of whether there is a sorting algorithm having worst-case running time of $o(n \log n)$.

A Formal Statement of the Question

Sorting can be viewed as the process of **determining the permutation that restores the order of input numbers.**

We will consider only **comparison sorts**, algorithms that sort numbers based only on comparisons between input elements. The sorting methods we have seen so far are all comparison sorts. We ask: **how many comparisons must a comparison sort execute to sort an array of n elements?**

Lower Bound on Comparison Sorts

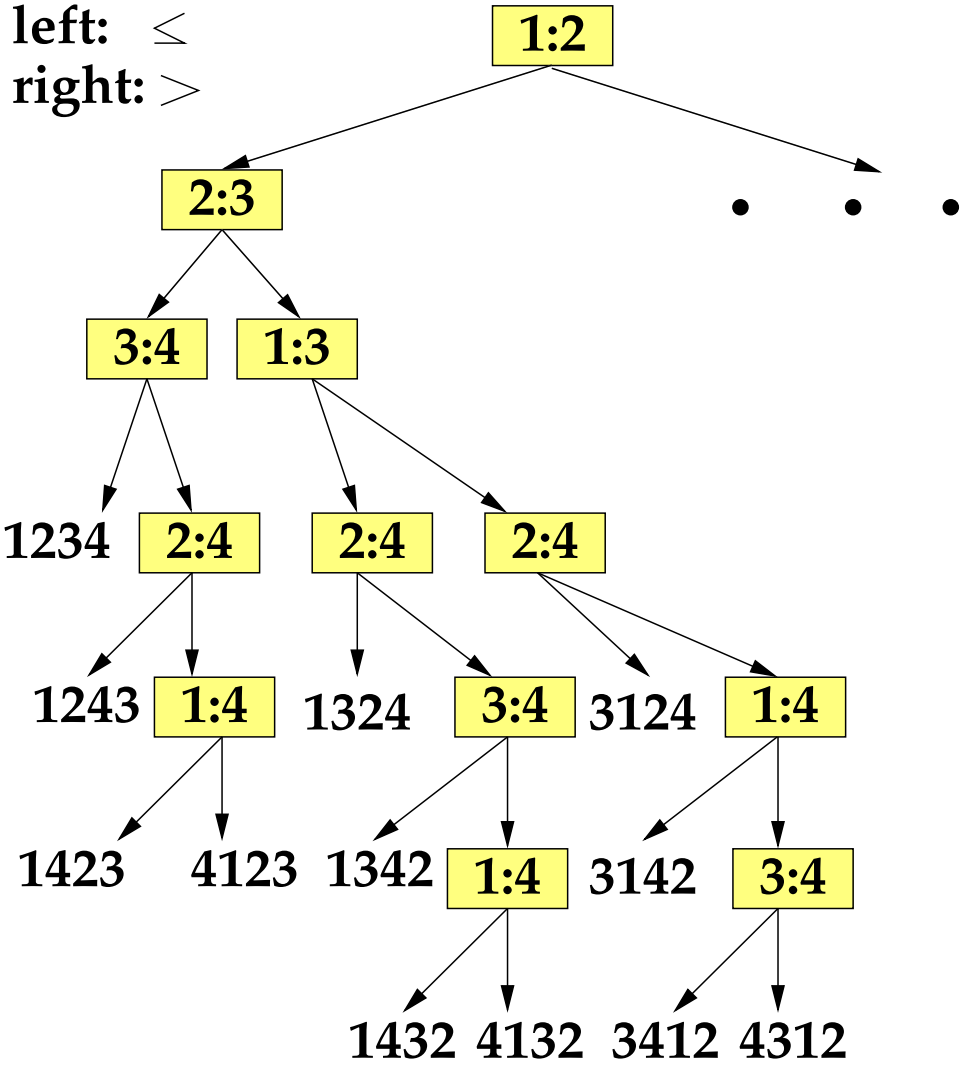
We think of two outcomes in a comparison of two numbers a and b: $a \leq b$ and $a > b$. (We may choose $a < b$ and $a \geq b$.)

For which pair a comparison is done **depends only on the outcomes of the comparisons that have been made so far.** So, for each n, the action of a comparison sort on an n-element array can be viewed as a binary tree such that

- each node corresponds to a comparison and
- each leaf corresponds to the permutation that the algorithm outputs.

We call such a tree **binary (decision) tree**

An Example: a decision for sorting 4 things



Lower Bound Argument

In a binary tree each input is associated with **a downward path from the root to a leaf**.

On an input array of size n, there are $n!$ **possible outputs.**

If the tree has depth d, the number of leaves is at most 2^d. Since the tree must possess $n!$ distinct outputs, $2^d \geq n!$. This gives $d \geq \lceil \lg n! \rceil = \Omega(n \lg n)$.

Thus we have proven:

Theorem A No comparison sort has the worst-case running time of $o(n \lg n)$.

Linear Time Sorting Algorithms

Linear time sorting algorithms exist in certain situations.

1. Counting-Sort

This is an algorithm that is useful when there is a function $f(n) = O(n)$ such that

- for each n and for each input array A having size n, the keys are taken from $\{1, \ldots, f(n)\}$.

The idea behind Counting-Sort

After sorting has been completed the array should look like: a segment of 1's, a segment of 2's, a segment of 3's, etc., where each segment can be empty.

So, find out for each key i, $1 \leq i \leq f(n)$, the location $s_i \,..\, t_i$ of the segment of i. For each i, let d_i the number of occurrences of numbers $\leq i$. Then, for all i, $1 \leq i \leq f(n)$, $s_i = 1 + \sum_{j=1}^{i-1} d_j$ and $t_i = \sum_{j=1}^{i} d_j$.

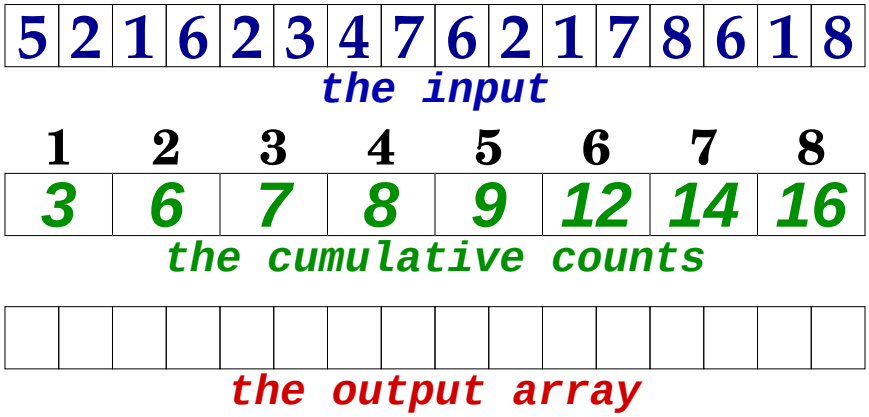
Suppose that these quantities have been computed. Then sorting can be done by scanning the input array backward and putting the j^{th} occurrence of the key i to position $t_i - j + 1$.

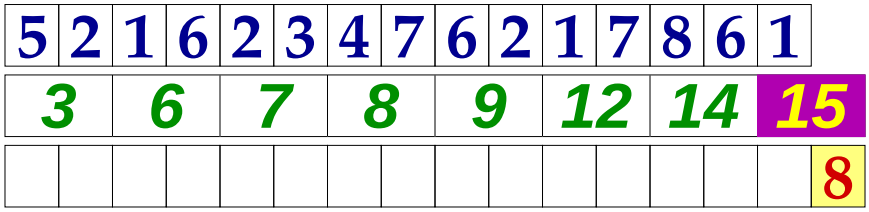
The Algorithm

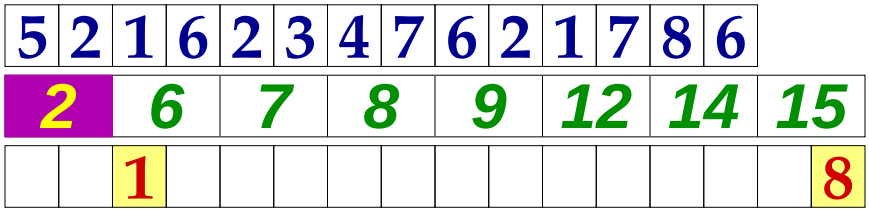
Counting-Sort(A, n, k)

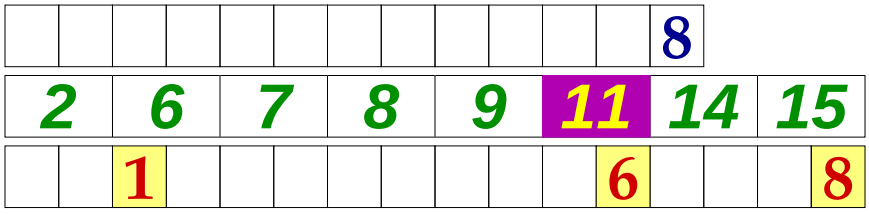
```
 1: ▷ k = f(n)
 2: for i ← 1 to k do d[i] ← 0
 3: for i ← 1 to n do
 4:     d[A[i]] ← d[A[i]] + 1
 5: ▷ Add 1 to the no. of occurrences of key i
 6: c[1] = d[1]
 7: for i ← 2 to k do
 8:     c[i] ← c[i − 1] + d[i]
 9: ▷ Compute t_i
10: for i ← n downto 1 do
11: {  B[c[A[i]]] ← A[i]
12:     c[A[i]] ← c[A[i]] − 1 }
13: ▷ Decrement the count by 1
14: for i ← 1 to n do
15:     A[i] ← B[i]
```

An Example

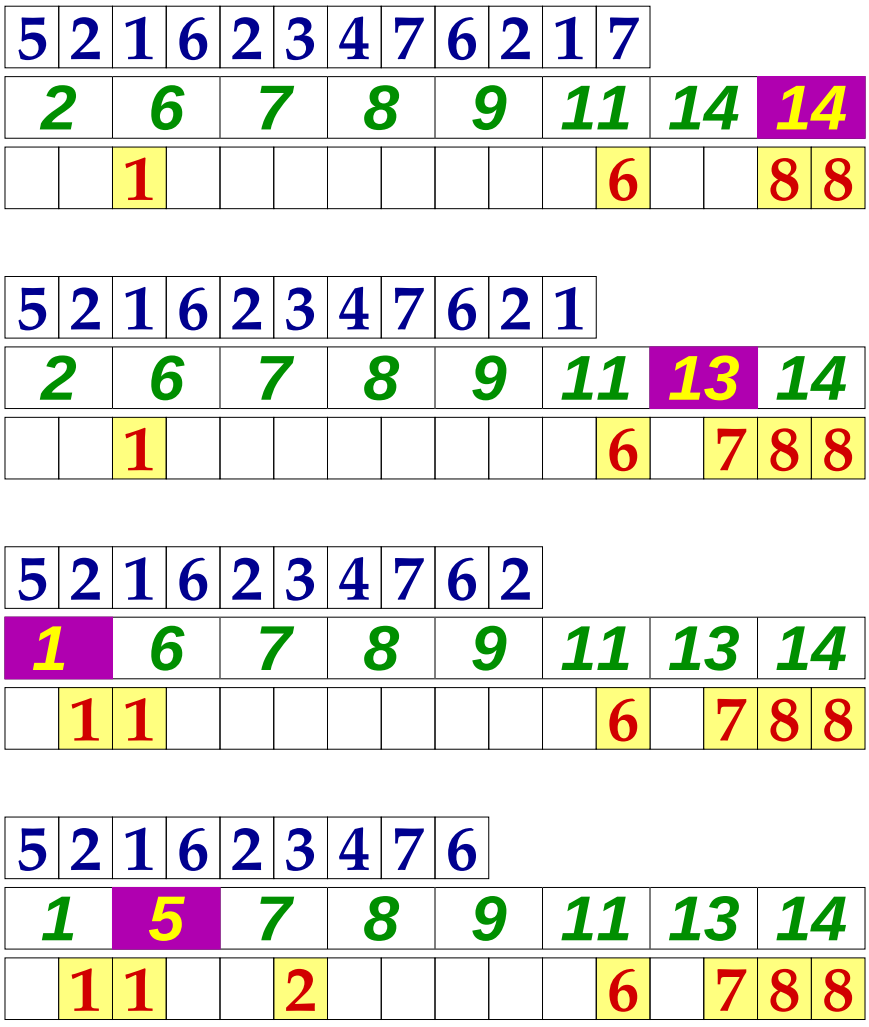

5 2 1 6 2 3 4 7 6 2 1 7 8 6 1 8
the input
1 2 3 4 5 6 7 8
3 6 7 8 9 12 14 16
the cumulative counts
the output array


5 2 1 6 2 3 4 7 6 2 1 7 8 6 1
3 6 7 8 9 12 14 15
8


5 2 1 6 2 3 4 7 6 2 1 7 8 6
2 6 7 8 9 12 14 15
1 8


8
2 6 7 8 9 11 14 15
1 6 8

An Example (cont'd)



Stable Sort

A **stable** sort is a sorting algorithm that **preserves the order of elements having the same key**.

Counting-Sort is stable.

2. Radix-Sort

Radix-Sort is a sorting algorithm that is useful when there is a constant d such that all the keys are d digit numbers.

To execute Radix-Sort, for $p = 1$ toward d sort the numbers with respect to the p^{th} digit from the right using any linear-time stable sort.

Radix-Sort is a stable sort.

An Example

input	digit 1	digit 2	digit 3	digit 4
1233	7650	4721	1145	0774
1145	4721	5522	6161	1145
7650	6161	6732	1233	1233
4721	6732	1233	7235	4265
6732	5522	7235	4265	4536
4265	1233	5336	5336	4721
7235	0774	4536	5522	5336
6161	1145	1145	4536	5522
0774	4265	7650	7650	6161
5336	7235	6161	6732	6732
4536	5336	4265	4721	7235
5522	4536	0774	0774	7650

What is the running time of Radix-Sort?

What is the running time of Radix-Sort?

Since a linear-time sorting algorithm is used d times and d is a constant, the running time of Radix-Sort is linear.

Proof of Correctness

We prove that for all p, $0 \leq p \leq d$, when the sorting with respect to the p^{th} has been completed,

- for each pair of strings (a, b), if $a < b$ and if a and b have the same prefix of length $d - p$, then a precedes b.

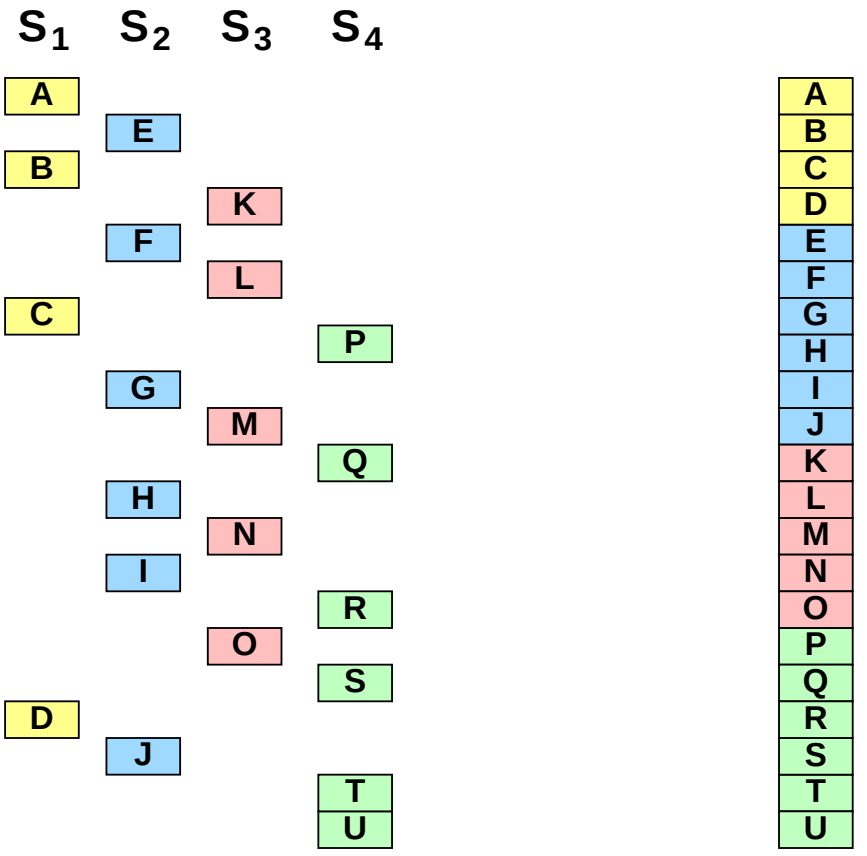
We prove this by induction on p. For the base case, let $p = 0$. The claim certain holds because two numbers having the same d-digit prefix are equal to each other.

Induction Step

Let $1 \leq p \leq d$. Suppose that the claim holds for all smaller values of p. Let a and b be two strings such that $a < b$ and such that a and b have an identical length-$(d-p)$ prefix. Suppose that a and b have an identical p-th digit. Then a and b have an identical length-$(d-p+1)$ prefix. So, by our induction hypothesis, by the end of the previous round a has been moved before b. Since the sorting algorithm for digit-wise sorting is stable, a will be placed before b in this round. Thus, the claim holds.

Suppose that the p-th digit of a is different from that of b. Then the p-th digit of a is smaller than that of b. So, the digit-wise sorting algorithm will certainly move a before b. Thus, the claim holds.

Induction Step



3. Bucket-Sort

This is a sorting algorithm that is effective when the numbers are taken from the interval $U = [0, 1)$.

To sort n input numbers, Bucket-Sort

1. partitions U into n non-overlapping intervals, called **buckets**,
2. puts each input number into its bucket,
3. **sorts each bucket using a simple algorithm,** e.g. Insertion-Sort, and then
4. **concatenates** the sorted lists.

What is the worst-case running time of Bucket-Sort?

What is the worst-case running time of Bucket-Sort?

$\Theta(n^2)$.

The worst cases are when all the numbers are in the same bucket.

But I expect on average the numbers are evenly distributed.

The Expected Running Time of Bucket-Sort

Assume that the keys are subject to the uniform distribution.

For each i, $1 \leq i \leq n$, let a_i be the number of elements in the i-th bucket. Since Insertion-Sort has a **quadratic running time**, the expected running time is:

$$O(n) + \sum_{i=1}^{n} O(\mathsf{E}[a_i^2]).$$

This is equal to $O(\sum_{i=1}^{n} \mathsf{E}[a_i^2])$. Since the keys are chosen under the uniform distribution, for all i and j, $1 \leq i < j \leq n$, the distribution of a_i is equal to that of a_j. So, the expected running time is equal to $O(n\mathsf{E}[a_1^2])$. We will prove that $\mathsf{E}[a_1^2] = 2 - 1/n$, which implies that Bucket-Sort has $O(n)$ expected running time.

For each i, $1 \leq i \leq n$, let X_i be the random variable whose value is 1 if the i^{th} element falls in the first bucket and is 0 otherwise. Then, for all i, $1 \leq i \leq n$, the probability that $X_i = 1$ is $1/n$.

It holds that $a_1 = \sum_{i=1}^{n} X_i$ so

$$a_1^2 = \sum_{i=1}^{n} X_i^2 + \sum_{1 \leq i,j \leq n, i \neq j} X_i X_j.$$

Thus,

$$\mathsf{E}[a_1^2] = \sum_{i=1}^{n} \mathsf{E}[X_i^2] + \sum_{1 \leq i,j \leq n, i \neq j} \mathsf{E}[X_i X_j].$$

For all i, $1 \leq i \leq n$,

$$\mathsf{E}[X_i^2] = 1^2(1/n) + 0^2(1 - 1/n) = 1/n,$$

and, for all i and j, $1 \leq i < j \leq n$,

$$\mathsf{E}[X_i X_j] = 1(1/n)^2 + 0(1 - (1/n)^2) = 1/n^2.$$

So,

$$\mathsf{E}[a_1^2] = n(1/n) + n(n-1)(1/n^2) = 2 - 1/n.$$