SFU CMPT-307 2008-2 Lecture: Week 6

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on June 10, 2008, 5.30pm-8.20pm

Exercise 6.1. Consider the following modification of the above algorithm:

Permute-Without-Fixed-Point$(A[1 \ldots n])$

1: **for** $i \leftarrow 1$ **to** $n-1$ **do**
2: swap $A[i] \leftrightarrow A[$**Random**$(i+1, n)]$
3: **end for**
4: return A

What kind distributions of permutations does it produce?

Hypothesis: The procedure produces a uniform random permutation without "fixed points".

A **fixed point** of a permutation is an element which stays at the same position after permuting.

Example. Consider all permutations of the array $\langle 1, 2, 3\rangle$: $\langle 1, 2, 3\rangle$, $\langle 1, 3, 2\rangle$, $\langle 2, 1, 3\rangle$, $\langle 2, 3, 1\rangle$, $\langle 3, 1, 2\rangle$ and $\langle 3, 2, 1\rangle$.

Only two of them are without any fixed point: $\langle 2, 3, 1\rangle$ and $\langle 3, 1, 2\rangle$.

Exercise 6.2 (continued)**.** Study performance of the algorithm for $n = 3$ and $n = 4$.

Which outputs (permutations) does it produce?

What are the probabilities of outputs?

Is the above hypothesis correct?

Exercise 6.3 (continues)**.** Using the loop invariant technique show that the algorithm generates only permutations without fixed point.

Assignment Problem 6.1. (deadline: June 17, 5:30pm)
Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i \ldots n]$, we swapped it with a random element from anywhere in the array:

Permute-With-All$(A[1 \ldots n])$

1: **for** $i \leftarrow 1$ **to** n **do**
2: swap $A[i] \leftrightarrow A[$**Random**$(1, n)]$
3: **end for**
4: return A

Show that this code doesn't produce a uniform random permutation for all sizes of inputs. That is, find an integer n, for which the above algorithm doesn't produce a uniform random permutation. Justify your answer.

QuickSort — Average case

The **randomized QuickSort** which uses the first approach (randomly permutes the input before sorting) has expected running time equal to the average running time of (non-randomized) **QuickSort**.

We have seen some intuition that this time is $\Theta(n \log n)$. Let's try to calculate this average running time exactly.

We will count only the number of comparisons. Let $C(n)$ be the average number of comparison of **QuickSort** assuming that every order of n elements is equally likely to appear as the input (uniform distribution), i.e.,

$$C(n) = \frac{1}{n!} \sum_{\pi \in S_n} C_{QS}(\pi)$$

where $C_{QS}(\pi)$ is the number of comparisons performed by **QuickSort** on the input (ordered by) π.

Recall, S_n is the set of all permutations on n elements.

Calculating $C_{QS}(\pi)$, for every $\pi \in S_n$ would be extremely difficult. So, let's try to perform only one step of **QuickSort** to get the recurrent formula for $C(n)$.

Question: What happens in the first step of **QuickSort** on the input π?

The array is partitioned to two parts using $\pi(n)$ as the pivot. The first part will have $\pi(n) - 1$ elements and the second $n - \pi(n)$.

Let's divide the inputs to n sets depending on $\pi(n)$. For every $i \in \{1, \ldots, n\}$, let S_n^i be the set of all permutations $\pi \in S_n$ such that $\pi(n) = i$. Note that each S_n^i has the same number of permutations (namely, $(n-1)!$).

For every $\pi \in S_n^i$, run **Partititon** algorithm. The first partition will be a permutation from S_{i-1} and the second partition is a permutation of the set $i+1, \ldots, n$.

One can show that the set of the first partitions contains each permutation in S_{i-1} the same number of times and the set of the second partitions contains each permutation of $i+1, \ldots, n$ the same number of times (see the *extra problem* on the next slide).

Hence, the average running time in the recursive call on the first partition will be $C(i-1)$ and on the second partition $C(n-i)$. Therefore,

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(C(i-1) + C(n-i)) \qquad (1)$$

$$\approx 2n \ln n \approx 1.39 n \log n$$

Note: Next, we will show that any comparison-based sorting algorithm has the average number of comparison at least $\approx n \log n$, i.e., on average QuickSort is only 39% worse than the best possible comparison-based sorting algorithm.

Extra Assignment Problem. (3% added to the overall performance if solved completely)

Deadline: The last lecture. (Note: You get extra points only if your solution is completely correct. You can submit the solution several times. If it's not correct, I will point out the problem(s) in your solution and you can try again.)

Part 1. Fix $i \in \{1, \ldots, n\}$. Consider all permutations $\pi \in S_n^i$ and run **Partition** algorithm for each such permutation. Show that the set of the first partitions generated by the algorithm (permutations of S_{i-1}) have the same number of occurrences of each permutation. Show that the set of second partitions generated by the algorithm (permutations of elements $i+1, \ldots, n$) have the same number of occurrences of each permutation.

Part 2. Show that the solution to the recurrence (1) is very close to $2n \ln n$. It's not enough to show that it is $\Theta(2n \ln n)$, you need to show that it does not differ from $2n \ln n$ by more than linear term, i.e., that $C(n) = 2n \ln n + O(n)$.

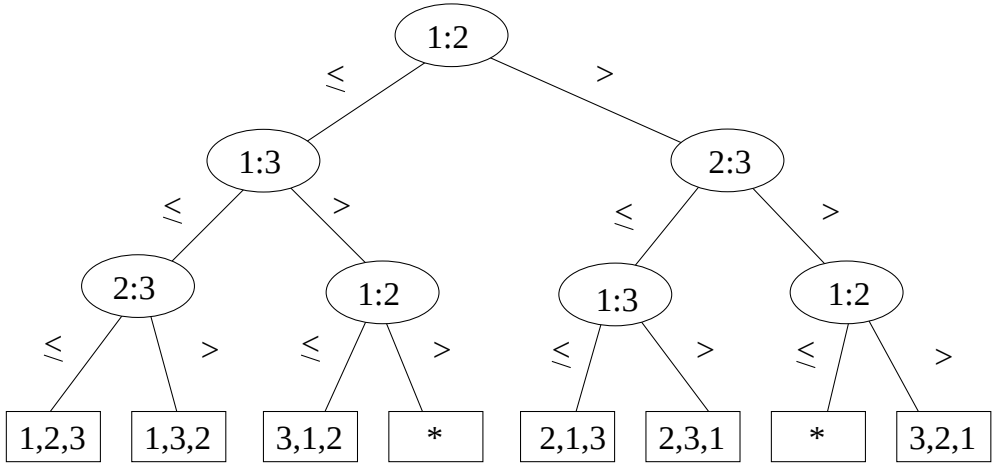
Average running time of comparison sorts — a lower bound

Consider a **decision tree** with ℓ leaves corresponding to a comparison sort on n elements.

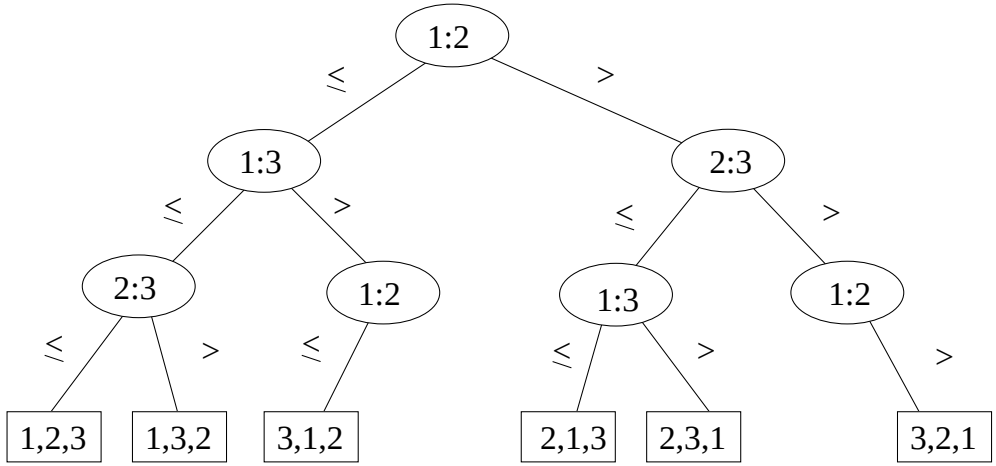
Remove all unreachable branches from the tree.

- now every output (= permutation of input elements) appears exactly once in the decision tree.
  – appears at least once, since for each permutation the sorting algorithm has to output the correct result
  – appears at most once, if the input is permuted in a certain way, then all decisions in the decision tree will be the same for each run of the algorithm, so we always end-up in the same leaf. Hence, all the other leaves with the same output are unreachable (and we removed them already).

Decision tree for **Selection-Sort** on 3-element inputs



decision tree with reachable nodes only



Note: every inner node has 1 or 2 children

Hence: number of leaves

$$\ell = \#\text{ of permutations of } n \text{ elements} = n!$$

what's the average running time?

- the running time depends on the number of comparisons, so we will only count the number of comparisons
- assume that each permutation appears equally likely
- for each permutation $\pi \in S_n$ of the input, the running time is the *depth* $\mathrm{depth}(\pi)$ of the leaf containing the permutation π in the decision tree = the number of edges (comparisons) from the root down to π

Hence, the **average number of comparisons** is

$$C_{\text{average}}(n) = \frac{\sum_{\pi \in S_n} \mathrm{depth}(\pi)}{n!}$$

For any tree T, let

$$D(T) = \sum_{x \text{ is a leaf of } T} \text{depth}(x)$$

Let $D(\ell)$ be the *minimum* of $D(T)$ over all **binary trees** T (not necessary complete) with ℓ leaves, i.e., $D(T) \geq D(\ell)$ for all T with ℓ leaves.

So we have:

$$\boxed{C_{\text{average}}(n) \geq \frac{D(n!)}{n!}}$$

We need a lower bound for $D(\ell)$.

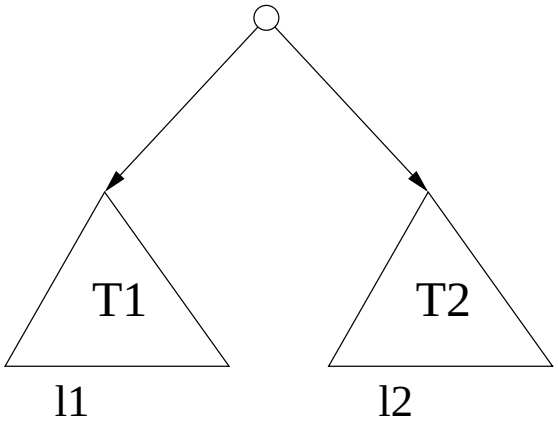
Guess: $D(\ell) \geq \ell \log \ell$

Proof: (by induction on ℓ)

Base case: $\ell = 1$. Obviously, the minimal value of $D(1)$ is $0 = 1 \cdot \log 1$.

Inductive step: Consider a tree with ℓ leaves with the minimal value of $D(\ell)$. The root of the tree has 1 or 2 children.

- 1 child — by removing a root we get again a tree with ℓ leaves, but with value of $D(\ell)$ smaller by ℓ, a contradiction with *minimality*
- 2 children:



if the left subtree has $\ell_1 > 0$ leaves and the right subtree has $\ell_2 > 0$ leaves, where $\ell_1 + \ell_2 = \ell$

then

$$\begin{aligned} D(\ell) = D(T) &= D(T_1) + D(T_2) + \ell \\ &\geq D(\ell_1) + D(\ell_2) + \ell \end{aligned}$$

we don't know what is the minimal division of leaves, but we have a recursive lower bound:

$$D(\ell) \geq \min_{1 \leq \ell_1 \leq \ell - 1} D(\ell_1) + D(\ell - \ell_1) + \ell$$

(similar to Assignment Problem 4.2)
by induction hypothesis we have

$$D(\ell) \geq \min_{1 \leq \ell_1 \leq \ell - 1} \ell_1 \cdot \log \ell_1 + (\ell - \ell_1) \cdot \log(\ell - \ell_1) + \ell$$

minimum for $\ell_1 = \frac{\ell}{2} = \ell - \ell_2$

$$\begin{aligned} D(\ell) &= \geq 2 \cdot \ell/2 \cdot \log(\ell/2) + \ell \\ &= \ell \cdot (\log \ell - 1) + \ell = \ell \log \ell \end{aligned}$$

Hence:

$$\boxed{D(\ell) \geq \ell \log \ell}$$

back to decision tree

now, $\ell = n!$, and so

$$C_{\text{average}}(n) \geq \frac{D(n!)}{n!} \geq \frac{n! \log(n!)}{n!} = \log(n!) = \Theta(n \log n)$$

Note: Bases on the analysis of $\log(n!)$, we know that that $n \log n \geq \log(n!) \geq cn \log n$, for any $c < 1$, hence $\log(n!) \approx n \log n$

An easy way, how to show that $\log(n!) = \Omega(n \log n)$:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdots n) = \log 1 + \log 2 + \cdots + \log n \\ &\geq \underbrace{\log \frac{n}{2} + \cdots + \log \frac{n}{2}}_{n/2} \\ &= \frac{1}{2} n \log \frac{n}{2} \in \Omega(n \log n)\end{aligned}$$

Hence, the **average running time** for any **comparison sort** is

$$\boxed{C_{\text{average}}(n) \geq \log(n!) = \Omega(n \log n)}$$

Sorting in linear time

We have seen a lower bound of order $\Omega(n \log n)$ for worst and average running time of comparison-based sorting algorithms.

Some algorithms achieve $O(n)$ average running time, given certain assumptions about the input.

Examples:

- **bucket sort**: assumes that the input elements are drawn from a uniform distribution;
- **counting sort**: assumes that the input elements are integers in range 0 to k, where $k = O(n)$;
- **radix sort**: assumes that the input elements have d digits, each digit is an integer in range 0 to k, where d is a constant and $k = O(n)$.

Bucket-Sort

Assumption: input numbers to be sorted are drawn from interval $[0, 1)$ with **uniform distribution**.

In this case, **expected** running time of bucket sort is $O(n)$. Algorithm maintains "buckets" (linked lists).

Basic idea:

- if you have n input elements, then we need n buckets
- divide $[0, 1)$ evenly into n consecutive sub-intervals $[0, 1/n)$, $[1/n, 2/n)$, $\ldots$, $[(n-1)/n, 1)$ (called *buckets*)
- given an element $A[i] \in [0, 1)$, throw it into the bucket with index $\lfloor n \cdot A[i] \rfloor$
- hope that input is distributed evenly among buckets
- sort buckets separately and concatenate results

Input $A = A[1], \ldots, A[n]$ with $A[i] \in [0, 1)$ drawn uniformly at random

Need auxiliary array $B[0], \ldots, B[n-1]$ of linked lists (buckets)

Bucket-Sort(A)

1: $n \leftarrow \text{length}(A)$
2: **for** $i \leftarrow 1$ **to** n **do**
3: insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
4: **end for**
5: **for** $i \leftarrow 0$ **to** $n-1$ **do**
6: sort list $B[i]$ with insertion sort
7: **end for**
8: concatenate lists $B[0], \ldots, B[n-1]$ together in order

Claim: expected running time is $O(n)$

Example

10 inputs elements: 0.32, 0.12, 0.78, 0.55, 0.91, 0.22, 0.41, 0.59, 0.72, 0.02,

buckets:
$[0, 1/10)$, $[1/10, 2/10)$, $\ldots [9/10, 1)$

After sorting buckets:

bucket	content
$[0, 1/10)$	0.02
$[1/10, 2/10)$	0.12
$[2/10, 3/10)$	0.22
$[3/10, 4/10)$	0.32
$[4/10, 5/10)$	0.41
$[5/10, 6/10)$	$0.55 \rightarrow 0.59$
$[6/10, 7/10)$	/
$[7/10, 8/10)$	$0.72 \rightarrow 0.78$
$[8/10, 9/10)$	/
$[9/10, 1)$	0.91

Analysis of Bucketsort

Correctness obvious

Why expected running time $O(n)$?

Certainly depends on sizes of buckets (# of elements in linked lists)

Let n_i be a random variable denoting the size of the i-th bucket B_i.

Insertion sort runs in time $O(n^2)$, thus the overall running time is

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

What is the expected value of running time?

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

Now, we need to estimate $E[n_i^2]$.

Assignment Problem 6.2. (deadline: June 17, 5:30pm)
Roll 3 fair 6-sided dice. Consider random variable X equal to the sum of outcomes on all dice. (Hence, X has values from 3 to 18.) Calculate $E[X]^2$ and $E[X^2]$.

Claim: $E[n_i^2] = 2 - 1/n$ for all $0 \leq i \leq n-1$

Clearly **same expectations** for all buckets since the input is drawn from uniform distribution on $[0, 1)$: each value is **equally likely** to fall into **any** bucket

Define a random variable X_{ij} for $i = 0, \ldots, n-1$ and $j = 1, \ldots, n$ as follows:

$$X_{ij} = \begin{cases} 1 & A[j] \text{ falls into bucket } i \\ 0 & \text{otherwise} \end{cases}$$

Clearly,

$$n_i = \sum_{j=1}^{n} X_{ij}$$

because X_{ij} is equal to 1 for each element that falls into i-th bucket

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&\overset{(*)}{=} E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\neq j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\Big(E[X_{ij}^2] + \sum_{\substack{1\le k\le n\\ k\neq j}} E[X_{ij}X_{ik}]\Big)
\end{aligned}
$$

$(*)$ is because

$$\begin{aligned}
&\left(\sum_{j=1}^{n} X_{ij}\right)^2 \\
&= (X_{i1} + X_{i2} + \cdots + X_{i,n-1})^2 \\
&= X_{i1}X_{i1} + X_{i1}X_{i2} + \cdots + X_{i1}X_{i,n-1} + \\
&\quad X_{i2}X_{i1} + X_{i2}X_{i2} + \cdots + X_{i2}X_{i,n-1} + \cdots + \\
&\quad X_{i,n-1}X_{i1} + X_{i,n-1}X_{i2} + \cdots + X_{i,n-1}X_{i,n-1} \\
&= \sum_{j=1}^{n}\sum_{k=1}^{n} X_{i,j}X_{i,k}
\end{aligned}$$

By definition of expectation,

$$E[X_{ij}^2] = E[X_{ij}] = 0 \cdot \left(1 - \frac{1}{n}\right) + 1 \cdot \frac{1}{n} = \frac{1}{n}$$

(note that if X is a 0-1 variable then $E[X^2] = E[X] = P[X]$)

and when $k \neq j$, X_{ij} and X_{ik} are independent, and thus

$$E[X_{ij}X_{ik}] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

This gives

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^{n}\Big(\frac{1}{n} + \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2}\Big) \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\ &= 1 + \frac{n-1}{n} = 1 + \frac{n}{n} - \frac{1}{n} = 2 - \frac{1}{n} \end{aligned}$$

and therefore

$$\begin{aligned} E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}(E[n_i^2]) \\ &= \Theta(n) + \sum_{i=0}^{n-1} \mathcal{O}\left(2 - \frac{1}{n}\right) = \Theta(n) \end{aligned}$$

Note: Bucket sort may have linear running time even when input is not drawn from uniform distribution on $[0, 1)$:

- we're fine whenever sum of squares of bucket sizes is linear in # of elements (when using the insertion sort for sorting buckets);
- or if we use a more efficient sorting algorithm for sorting buckets, say, merge sort, then whenever
  $\sum_{i=0}^{n-1} n_i \log n_i = O(n)$

Counting sort

Assumption on input: input elements are integers in the range 0 to k.

When $k = \mathcal{O}(n)$, then running time: $\Theta(n)$.

Basic idea: for each input element x, find the number of elements smaller than x. Let this number be ℓ. Then we can place x directly at position $\ell + 1$.

Complications: several elements might have the same value.
— we have to avoid putting them at the same position.

algorithm uses 3 arrays:

- $A[1 \ldots n]$ — input array
- $B[1 \ldots n]$ — output array
- $C[0 \ldots k]$ — temporary array, $C[p]$ = number of elements less or equal than p

Counting-Sort(A)

```
1:  n ← length(A)
2:  for i ← 0 to k do
3:      C[i] ← 0
4:  end for
5:  for j ← 1 to n do
6:      C[A[j]] ← C[A[j]] + 1
7:  end for
8:  /* C[i] contains # of elements equal to i */
9:  for i ← 1 to k do
10:     C[i] ← C[i] + C[i − 1]
11: end for
12: /* C[i] contains # of elements ≤ i */
13: for j ← n downto 1 do
14:     B[C[A[j]]] ← A[j]
15:     C[A[j]] ← C[A[j]] − 1
16: end for
```

explanation:

- loop 2–4: initialize array C
- loop 5–7: if the value of an input element is i, increment $C[i]$ by 1; hence, after the loop, $C[i]$ contains the number of elements equal to i
- loop 9–11: after loop $C[i]$ contains the number of elements that are less than or equal to i

  the loop invariant:
  - before the i-th iteration, for all $k < i$, $C[k]$ contains number of elements smaller than or equal to k
- loop 13–16: before the first iteration, $C[i]$ contains the last position where an element with value i should go; after we place $A[j]$ into B, we decrement $C[i]$ so that the next element with value i would be place at position smaller by 1

Counting-Sort(A)

```
1:  n ← length(A)
2:  for i ← 0 to k do
3:      C[i] ← 0
4:  end for
5:  for j ← 1 to n do
6:      C[A[j]] ← C[A[j]] + 1
7:  end for
8:  /* C[i] contains # of
    elements equal to i */
9:  for i ← 1 to k do
10:     C[i] ← C[i] + C[i − 1]
11: end for
12: /* C[i] contains # of
    elements ≤ i */
13: for j ← n downto 1 do
14:     B[C[A[j]]] ← A[j]
15:     C[A[j]] ← C[A[j]] − 1
16: end for
```

running time:

- loop 2–4: $\Theta(k)$
- loop 5–7: $\Theta(n)$
- loop 9–11: $\Theta(k)$
- loop 13–16: $\Theta(n)$

total: $\Theta(n+k) = \Theta(n)$ if $k = \mathcal{O}(n)$

Counting-Sort(A)

1: $n \leftarrow \text{length}(A)$
2: **for** $i \leftarrow 0$ **to** k **do**
3: $\quad C[i] \leftarrow 0$
4: **end for**
5: **for** $j \leftarrow 1$ **to** n **do**
6: $\quad C[A[j]] \leftarrow C[A[j]] + 1$
7: **end for**
8: `/*` $C[i]$ `contains # of elements equal to` i `*/`
9: **for** $i \leftarrow 1$ **to** k **do**
10: $\quad C[i] \leftarrow C[i] + C[i-1]$
11: **end for**
12: `/*` $C[i]$ `contains # of elements` $\leq i$ `*/`
13: **for** $j \leftarrow n$ **downto** 1 **do**
14: $\quad B[C[A[j]]] \leftarrow A[j]$
15: $\quad C[A[j]] \leftarrow C[A[j]] - 1$
16: **end for**

Example.

input A:

4	2	4	8	8	4	4

after loop 5–7:

C:

0	1	2	3	4	5	6	7	8
0	0	1	0	4	0	0	0	2

after loop 9–11:

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	5	5	5	5	7

loop 13–16, after 1st iteration:

B:

1	2	3	4	5	6	7
–	–	–	–	4	–	–

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	4	5	5	5	7

Counting-Sort(A)

1: $n \leftarrow \text{length}(A)$
2: **for** $i \leftarrow 0$ **to** k **do**
3: $\quad C[i] \leftarrow 0$
4: **end for**
5: **for** $j \leftarrow 1$ **to** n **do**
6: $\quad C[A[j]] \leftarrow C[A[j]] + 1$
7: **end for**
8: `/*` $C[i]$ `contains # of elements equal to` i `*/`
9: **for** $i \leftarrow 1$ **to** k **do**
10: $\quad C[i] \leftarrow C[i] + C[i-1]$
11: **end for**
12: `/*` $C[i]$ `contains # of elements` $\leq i$ `*/`
13: **for** $j \leftarrow n$ **downto** 1 **do**
14: $\quad B[C[A[j]]] \leftarrow A[j]$
15: $\quad C[A[j]] \leftarrow C[A[j]] - 1$
16: **end for**

input A:

4	2	4	8	8	4	4

loop 13–16, after 2nd iteration:

B:

1	2	3	4	5	6	7
–	–	–	4	4	–	–

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	3	5	5	5	7

loop 13–16, after 3rd iteration:

B:

1	2	3	4	5	6	7
–	–	–	4	4	–	8

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	3	5	5	5	6

Counting-Sort(A)

1: $n \leftarrow \text{length}(A)$
2: **for** $i \leftarrow 0$ **to** k **do**
3: $\quad C[i] \leftarrow 0$
4: **end for**
5: **for** $j \leftarrow 1$ **to** n **do**
6: $\quad C[A[j]] \leftarrow C[A[j]] + 1$
7: **end for**
8: `/* ` $C[i]$ `contains # of elements equal to` i `*/`
9: **for** $i \leftarrow 1$ **to** k **do**
10: $\quad C[i] \leftarrow C[i] + C[i-1]$
11: **end for**
12: `/* ` $C[i]$ `contains # of elements` $\leq i$ `*/`
13: **for** $j \leftarrow n$ **downto** 1 **do**
14: $\quad B[C[A[j]]] \leftarrow A[j]$
15: $\quad C[A[j]] \leftarrow C[A[j]] - 1$
16: **end for**

input A:

4	2	4	8	8	4	4

loop 13–16, after 4th iteration:

B:

1	2	3	4	5	6	7
–	–	–	4	4	8	8

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	3	5	5	5	5

loop 13–16, after 5th iteration:

B:

1	2	3	4	5	6	7
–	–	4	4	4	8	8

C:

0	1	2	3	4	5	6	7	8
0	0	1	1	2	5	5	5	5

loop 13–16, after 6th iteration:

B:

1	2	3	4	5	6	7
2	–	4	4	4	8	8

C:

0	1	2	3	4	5	6	7	8
0	0	0	1	2	5	5	5	5

loop 13–16, after 7th iteration:

B:

1	2	3	4	5	6	7
2	4	4	4	4	8	8

C:

0	1	2	3	4	5	6	7	8
0	0	0	1	1	5	5	5	5

Note: no a single comparison during the run of the algorithm — clearly, this is not a comparison sort

Stability

- the counting sort is *stable*: the numbers with the same value appear in the output in the same order as in the input
- of course, this property is important only if some satellite data is carried around with the elements we are sorting — that is each element contains a lot of records, but we are sorting elements only according to one of the records (the **key**)

Exercise 6.4. Is Bucket-Sort stable?

Assignment Problem 6.3. (deadline: June 17, 5:30pm)
Suppose that the **for** loop 13–16 is rewritten so that the loops start with $j = 1$ and end with $j = n$, i.e., the line 13 is changed to

for $j \leftarrow 1$ **to** n **do**

Show that the algorithm still works properly. Is the modified algorithm stable?

Assignment Problem 6.4. (deadline: June 17, 5:30pm)
Which of the following sorting algorithms are stable: selection sort, merge sort, heapsort, and quicksort? If the algorithm is not stable, give an example of an input showing it. If the algorithm is stable, prove it.

Radix sort

Assumption on input: the elements have d digits, and each digit has at most k possible values

Examples.

$A = \{329, 457, 657, 839, 436, 720, 355\}$ — all elements have 3 digits, $k = 10$

$A = \{abab, aaba, bbab, abbb, baab, bbbb\}$ — all elements have 4 digits, $k = 2$

$A =$ {"June 26, 93", "April 13, 04", "October 1, 71"} — all elements have 3 digits, $k = 100$

The running time is $\Theta(d(n + k))$, where

n — size of input

d — number of digits

k — number of possible values, one digit can take.

Hence, if d is a constant and $k = \mathcal{O}(n)$, then *linear time*.

Basic idea:

- sort inputs on the **least significant** digit, then on second least significant digit, etc.
- for sorting on digits use some **stable** sorting algorithm, for example **Counting-Sort** or

— might seem counterintuitive, why we start with the least significant digit?

— when we want to compare two elements, we would first compare the most significant digits of the elements, if we get tie, we would continue with the second significant digits, etc.

— however, when sorting we go backwards, using the fact that we use a **stable** sorting algorithm on digits, to obtain the correct order of elements in the end

Example.

3.29		7.2	0		7.	2	0		3	.29
4.57		3.5	5		3.	2	9		3	.55
6.57		4.3	6		4.	3	6		4	.36
8.39	$\Longrightarrow$	4.5	7	$\Longrightarrow$	8.	3	9	$\Longrightarrow$	4	.57
4.36		6.5	7		3.	5	5		6	.57
7.20		3.2	9		4.	5	7		7	.20
3.55		8.3	9		6.	5	7		8	.39

- consider an input array A containing elements with d digits
- let digit 1 be the least significant, and digit d the most significant

Radix-Sort(A, d)

1: **for** $i \leftarrow 1$ **to** d **do**
2: use Counting-sort to sort array A on digit i
3: **end for**

Running time: $\Theta(d(n+k))$

Correctness. Loop invariant:

- after the i-th iteration the input is sorted on the digits $i, \ldots, 1$

Initialization: trivially true after the first iteration

Maintenance: after $(i-1)$-th iteration the input is sorted on digits $i-1, \ldots, 1$; during the i-th iteration we use a stable sort on digit i; consider any two elements $x = x_d \ldots x_1$ and $y = y_d \ldots y_1$ of the input, then:

- if $x_i < y_i$, then element x appears before y in current ordering
- if $x_i = y_i$, then, since the used sort is stable, they appear in the same order as when sorted on digits $i-1, \ldots, 1$
- $\Longrightarrow$ elements x and y are sorted correctly on digits $i, \ldots, 1$

Termination: after the last iteration, the input is sorted on digits $d, \ldots, 1$, i.e., sorted on value of elements

Usage of Radix-sort

1. used for card-sorting (punched cards) on early computers
2. nowadays, useful when sorting data with multiple keys
   (for example: year, month, day)
3. breaking the key into digits:
   - input: n b-bit numbers (keys)
   - for any integer $r \leq b$, we can sort the input using **Radix-sort** in time $\Theta\left(\frac{b}{r} \cdot (n + 2^r)\right)$
     - each key can be viewed as having $d = \lceil b/r \rceil$ digits, each digit consists of r bits, i.e., $k = 2^r$
     - hence, using **Radix-sort**, we can sort input in time $\Theta(d(n + k)) = \Theta(b/r \cdot (n + 2^r))$
   - how to choose r optimally?
     - if $b < \log n$, then choosing $r = b$ ($d = 1$) gives time $\Theta(n)$
     - if $b \geq \log n$, then $r = \lfloor \log n \rfloor$ gives the best time within a constant factor: $\Theta(bn/\log n)$

Assignment Problem 6.5. (deadline: June 17, 5:30pm)
Show how to sort n integers in the range 0 to $n^2 - 1$ in $\mathcal{O}(n)$ time.