Practice 2

Exercise levels:
(L1) Reproduce: Reproduce basic facts or check basic understanding.
(L2) Apply: Follow step-by-step instructions.
(L3) Reason: Show insight using a combination of different concepts.
(L4) Create: Prove a non-trivial statement or create an algorithm or data structure of which the objective is formally stated.

▶ Lecture 4 Sorting in linear time

Sorting properties and lower bounds

Exercise 1

(a) *(L1)* Give the definition of a stable sorting algorithm.

Solution:
A sorting algorithm is stable if elements with the same key appear in the output array in the same order as they do in the input array.

(b) *(L2)* Sometimes it is desirable to sort data based on several keys. For example, you may want to sort all exercises by topic, and those with the same topic by complexity. Explain why a stable sorting algorithm can simply sort the data one key at a time.

Solution:
Assume we sort elements of an array A by two keys, key_1 and key_2. A stable sorting algorithm does not switch the order of elements with the same keys. Thus, if the elements are first sorted by key_2, and then by key_1, then for any two indices $i < j$ we will have that either $A[i].key_1 < A[j].key_1$, or $A[i].key_1 = A[j].key_1$ and $A[i].key_2 \leq A[j].key_2$. In both cases the order of the elements in the output is correct.

Exercise 2

(a) *(L4)* Prove, for all $n \geq 2$, that $n - 1$ comparisons are sometimes necessary to test whether an array with n distinct elements is sorted in decreasing order.

Solution:

Highlights:

- Note that the proof must work for *all* values of $n \geq 2$. Just giving one instance with a specific n is not good enough.
- You *cannot make any assumption* about the inner workings of the algorithm (excluding the assumption that comparisons are used) as it must be true for any algorithm. Particularly, for example, you cannot assume that element 1 and 2 are not compared, or that exactly one pair of adjacent elements in not compared (there might be more), or that every element is compared to some other element (that need not be the case), etc...
- It is a common approach to take one input and determine the answer a correct algorithm must give. In this case we create a decreasing array, thus assuming the algorithm is correct, it will correctly conclude that this is the case. Then we slightly alter the input to create a new input that should get a different answer (not decreasing), but that will follow the same path in the decision tree (and hence get the same answer).
- An alternative is to compute the minimum height of the decision tree (see (b)).

We will prove this by contradiction. Assume there exists an algorithm that uses at most $n - 2$ comparisons and always correctly checks if an array with n distinct elements is sorted in decreasing order. Consider the decision tree for this algorithm. We will construct two input arrays, one sorted and one not, on which the algorithm will trace exactly the same path from the root to some leaf, and so will produce the same answer. Consider an array of numbers sorted in decreasing order. Specifically, let array $A = \langle n, n-1, \dots, 1 \rangle$, that is, $A[i] = n + 1 - i$ for $1 \leq i \leq n$. As the algorithm is correct, by our assumption, it will correctly determine that array A is sorted in decreasing order. As A has n elements and the algorithm has made at most $n - 2$ comparisons, at least one neighboring pair of the elements has not been compared. Denote these elements as $A[i]$ and $A[i+1]$ for some index i, we know that $A[i] = A[i+1] + 1$.
Next, we make a new array A' that is a copy of A, but with exchanged elements $A[i]$ and $A[i+1]$. That is, $A'[j] = A[j]$ for all $1 \leq j < i$ and $i + 1 < j \leq n$, and $A'[i] = A[i+1]$ and $A'[i+1] = A[i]$. Array A' is not sorted.
Now we compare the paths that the algorithm traces in the decision tree when executed on the arrays A and A'. Any comparison that does not involve the elements with indices i and $i + 1$ will result in the same answer for both arrays, A and A'. Any comparison made by the algorithm with the elements at index i and some other index $j \neq i + 1$ will still result in the same answer for both arrays. Similarly, any comparison made by the algorithm with the elements at index $i + 1$ and some other index $j \neq i$ will result in the

same answer for both arrays. But then, the algorithm will follow exactly the same path in the decision tree from root to some leaf on both arrays, A and A'. So, the algorithm will incorrectly decide that A' is also sorted in decreasing order.
We have arrived at a contradiction, so our assumption must be incorrect and there cannot exist an algorithm that always correctly checks if an array with n distinct elements is sorted in decreasing order using at most $n-2$ comparisons.

(b) *(L3)* Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n.

Hint: What would the decision tree of such an algorithm look like?

Solution:

Highlights:

- There are $n!$ different inputs as each of the $n!$ different permutations needs a particular re-ordering to make the list sorted. Thus each permutation is a *distinct* input.
- The runtime is lowerbounded by the length of the path we need to traverse in the decision tree. The claim is thus that there are $\frac{n!}{2}$ different paths in the tree that have depth $O(n)$. Particularly, there should thus be at least $\frac{n!}{2}$ leaves with depth n.

Suppose there exists an algorithm $\mathcal{A}$ that achieves running time $O(n)$ for at least half of the $n!$ inputs of length n. Let T_n be the (binary) decision tree for this algorithm. For each n we have that T_n contains $n!$ different leaves (one for each permutation of an input). Moreover, by the assumption that the algorithm $\mathcal{A}$ achieves running time $O(n)$ it follows that there exists a positive constant c such that $n!/2$ leaves of T_n have depth at most $c \cdot n$. Intuitively, this is too many leaves already for a tree of only this height. We formalize this as follows.
Let T_n' be the tree obtained from T_n by removing all the nodes that have depth more than $c \cdot n$. Denote the number of leaves in T_n' as ℓ'. The tree T_n' has two types of leaves, *old leaves*, which were leaves in T_n, and *new leaves*, which were the roots of non-trivial subtrees in T_n. By our assumption on $\mathcal{A}$, the number of old leaves in T_n' is at least $n!/2$. Thus, the total number of leaves in T_n' is at least $n!/2$:

$$\ell' \geq n!/2\,.$$

On the other hand, by construction, the height h of T_n' is at most $c \cdot n$:

$$h \leq c \cdot n\,.$$

A simple upper bound on the number of leaves in a tree is the total number of nodes. The total number of nodes in a tree of height h is at most $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 < 2^{h+1}$. Thus, T_n' has less than $2^{c \cdot n+1}$ vertices, and hence less than $2^{c \cdot n+1}$ leaves:

$$\ell' < 2^{c \cdot n+1}\,.$$

So, we get that
$$n!/2 \leq \ell' < 2^{c \cdot n+1} .$$
Since the factorial grows faster than any exponential, the previous inequality will not hold for sufficiently large n. We have arrived to a contradiction, so our assumption must be incorrect and there cannot exist such a sorting algorithm that achieves $O(n)$ running time at least on half of the $n!$ inputs of length n.

Exercise 3 *(L3)* Professor F. Alsvanouds from the University of Harderwijk has made a discovery. He has figured out how to print the keys of a min-heap with n nodes in (increasing) sorted order in $O(n)$ time. Is this indeed possible? Explain how or prove why not.

Solution:
As we cannot make any assumptions on the keys we can only employ comparison-based algorithms. We show using a proof by contradiction that the claim of Professor F. Alsvanouds is false.
Assume that there exists an algorithm $\mathcal{A}$ that can print the keys of a min-heap in sorted order in $O(n)$ time. Now consider the algorithm $\mathcal{B}$ that takes as input an unsorted array and returns the array in sorted order. Algorithm $\mathcal{B}$ works by first building a min-heap from the unsorted array using Build-Min-Heap. Then it uses algorithm $\mathcal{A}$ to print the keys of the heap in sorted order.
We know (from the book) that Build-Min-Heap takes $O(n)$ time. As $\mathcal{A}$ also takes $O(n)$ time, the total running time of algorithm $\mathcal{B}$ will be $O(n) + O(n) = O(n)$. But then algorithm $\mathcal{B}$ breaks the $\Omega(n \log n)$ lower bound on comparison-based sorting. We know this cannot be true as we have proven this lowerbound holds for all comparison-based sorting algorithms.
We have arrived at a contradiction, therefore our assumption must be false. There does not exist an algorithm that can print the keys of a min-heap in sorted order in $O(n)$ time.

Linear-time sorting

Exercise 4 *(L1)* Give the assumptions that must hold for CountingSort, RadixSort, and BucketSort respectively, to run in $O(n)$ time on an input of size n.

Solution:
CountingSort: the input elements are integers in the range from 0 to k, for some $k = O(n)$.
RadixSort: the input elements are integers with d digits, for some $d = O(1)$. Each digit is an integer in the range from 0 to k, for some $k = O(n)$.
BucketSort: the input elements are real numbers uniformly distributed on the interval $[0, 1)$.

Exercise 5 *(L3)* Explain how to make BucketSort run in worst-case $O(n \log n)$ time.

Solution:
Instead of sorting each bucket with InsertionSort in line 6 of the algorithm as given in the lectures, sort each bucket with a $O(n \log n)$ algorithm instead, for example, with MergeSort. Then, the total running time will be given by

$$O(n) + \sum_{i=0}^{n-1} O(n_i \log n_i) = O(n) + O\Big(\sum_{i=0}^{n-1} n_i \log n_i\Big),$$

where n_i is the number of elements in the bucket $B[i]$. We can bound the sum from above in the following way:

$$\sum_{i=0}^{n-1} n_i \log n_i \leq \sum_{i=0}^{n-1} n_i \log n = \log n \sum_{i=0}^{n-1} n_i = n \log n\,.$$

Thus in the worst case the total running time of BucketSort will be $O(n \log n)$.

Exercise 6 *(L2)* Let array A contain information about the students in Data Structures using the student IDs as the keys. A student ID is a k-digit integer, that is, an integer in the range $[10^{k-1}, 10^k - 1]$. Assuming that the student IDs are uniformly distributed in the given range, propose an average-case $O(n)$ running time algorithm to sort the array A. (*The running time of your algorithm needs to be independent of k.*)

Solution:
First, convert the range $[10^{k-1}, 10^k)$ into $[0, 1)$ by changing the keys in the following way. For a key *key*, create a new key $\frac{key - 10^{k-1}}{10^k - 10^{k-1}}$.
Create an auxiliary array B with the elements of A stored as a satellite data, with the new keys, sort the array B using BucketSort, and return the satellite data of B in the correct order (that is, return the original elements of A in the sorted order).
Proof of correctness: Subtracting the same value from all keys does not change their order. Similarly, dividing all keys by the same value does not change their order either. Thus the sorted order of the created new keys is the same as that of the original keys. Thus, assuming that BucketSort sorts array B correctly, we will obtain the correct sorted order of the elements of A.

Running time analysis (intuition): Modifying the keys takes $O(n)$ time as for each key we can compute the new key using $O(1)$ operations. Creating a new array of size n also takes $O(n)$ time. Finally, as the elements of array A are uniformly distributed over the range $[10^{k-1}, 10^k)$ the modified keys are uniformly distributed over the interval $[0, 1)$. By definition BucketSort takes on average $O(n)$ time if the elements are uniformly distributed over $[0, 1)$. In total, our algorithm has an average-case $O(n)$ running time.

Exercise 7 *(L4)* Prove using a loop invariant that RadixSort works. Where does your proof need the assumption that the intermediate sort is stable?

Solution:

Highlights:

- To prove correctness it needs to be *explicitly* shown how stability ensures the numbers are in sorted order. Particularly why numbers with the same most-important digit are in the correct relative order.

Given a set of integers with d digits, we can sort it using the RadixSort algorithm. Recall that we count the digits from right to left, that is, the 1st digit is the least significant one, and the dth digit is the leftmost digit. Let's review the pseudocode for RadixSort:

```
RadixSort(A, d)
    Input: Array A of integers with d digits each
    Output: Array A sorted in increasing order
1   for i = 1 to d
2       use a stable sort to sort array A on digit i
```

We will prove its correctness by loop invariant.
Loop invariant: A is sorted in increasing order on the $i-1$ least-significant digits of each element.
Initialization:
At the start of the loop we have $i = 1$ and the array A is unsorted. Indeed then by default A is sorted in increasing order on the last $1 - 1 = 0$ digits of each element.
Maintenance:
Assume that at the start of an arbitrary iteration the LI holds. Then A is sorted in increasing order on the $i-1$ least-significant digits of each element.
For any element x, denote the ith digit of x as x_i, and denote the value corresponding to the j least-significant digits of x as $x_{j..1}$.
When sorting the numbers on the ith digit, there are three cases that may occur for any two arbitrary elements e and f:

(a) $e_i > f_i$. As $e_i > f_i$, it must also be that $e_{i..1} > f_{i..1}$. Hence, any stable sorting algorithm will correctly place e after f in the sorted order.

(b) $e_i < f_i$. As $e_i < f_i$, it must also be that $e_{i..1} < f_{i..1}$. Hence, any stable sorting algorithm will correctly place e before f in the sorted order.

(c) $e_i = f_i$. As $f_i = e_i$, the most significant digit of $e_{i..1}$ and $f_{i..1}$ cannot be used to distinguish both values. Any stable sorting algorithm that uses the ith digit to sort the elements will preserve the relative position of e and f. Assume w.l.o.g. that $e_{i-1..1} \leq f_{i-1..1}$, thus by LI e precedes f in the array at the beginning of the iteration. As $e_i = f_i$ we have that $e_{i..1} \leq f_{i..1}$, and thus after the iteration e is still correctly placed before f in the array.

These three cases trivially cover all possibilities. Thus at the start of the next iteration A must

be correctly sorted by the i least-significant digits. As at the start of the next iteration we increase the value of i by one, the LI still holds.
Termination:
When the loop terminates $i > d$. As i is increased by 1 each iteration, we must have $i = d + 1$. By the loop invariant it then holds that A is correctly sorted on the last $d + 1 - 1 = d$ digits. But then A is sorted by increasing value.

Stability is required for the third case. When the two most significant digits are equal we need to know that the values remain sorted on the less significant digits.

Exercise 8

(a) *(L1)* Give a step-by-step description of the working of RadixSort using CountingSort as a subroutine.

Solution:
RadixSort sorts numbers digit by digit starting with the least-significant (rightmost) digit first.

```
RadixSort($A, d$)
    Input: Array $A$ of integers with $d$ digits each
    Output: Array $A$ sorted in increasing order
1   for $i = 1$ to $d$
2       sort the numbers in $A$ by their $i$-th digit using CountingSort
```

In the **for** loop in line 2 of the algorithm, we create an auxiliary array A' with keys being the ith digit of each number, and the number itself as a satellite data. We then sort A' using CountingSort. In the next iteration, we replace the keys in array A' with the $(i + 1)$-th digit of the respective numbers, and repeat.

(b) *(L2)* Illustrate the execution of RadixSort, using CountingSort as a subroutine, on the following input:

$$\langle 286, 125, 623, 916, 435, 522, 111, 736 \rangle .$$

For the first digit that is sorted, also show the steps of the execution of CountingSort.

Solution:

$$A = \langle 286, 125, 623, 916, 435, 522, 111, 736 \rangle$$

1. CountingSort on the first digit:

$$A' = \langle (6, 286), (5, 125), (3, 623), (6, 916), (5, 435), (2, 522), (1, 111), (6, 736) \rangle$$

The **for** loop in line 2 of CountingSort:

$$\begin{aligned}
& C = \langle 0,0,0,0,0,0,0,0,0,0\rangle \\
C[6] = C[6] + 1 \quad & C = \langle 0,0,0,0,0,0,1,0,0,0\rangle \\
C[5] = C[5] + 1 \quad & C = \langle 0,0,0,0,0,1,1,0,0,0\rangle \\
C[3] = C[3] + 1 \quad & C = \langle 0,0,0,1,0,1,1,0,0,0\rangle \\
C[6] = C[6] + 1 \quad & C = \langle 0,0,0,1,0,1,2,0,0,0\rangle \\
C[5] = C[5] + 1 \quad & C = \langle 0,0,0,1,0,2,2,0,0,0\rangle \\
C[2] = C[2] + 1 \quad & C = \langle 0,0,1,1,0,2,2,0,0,0\rangle \\
C[1] = C[1] + 1 \quad & C = \langle 0,1,1,1,0,2,2,0,0,0\rangle \\
C[6] = C[6] + 1 \quad & C = \langle 0,1,1,1,0,2,3,0,0,0\rangle
\end{aligned}$$

The **for** loop in line 4 of CountingSort:

$$\begin{aligned}
& C = \langle 0,1,1,1,0,2,3,0,0,0\rangle \\
C[1] = C[1] + C[0] \quad & C = \langle 0,1,1,1,0,2,3,0,0,0\rangle \\
C[2] = C[2] + C[1] \quad & C = \langle 0,1,2,1,0,2,3,0,0,0\rangle \\
C[3] = C[3] + C[2] \quad & C = \langle 0,1,2,3,0,2,3,0,0,0\rangle \\
C[4] = C[4] + C[3] \quad & C = \langle 0,1,2,3,3,2,3,0,0,0\rangle \\
C[5] = C[5] + C[4] \quad & C = \langle 0,1,2,3,3,5,3,0,0,0\rangle \\
C[6] = C[6] + C[5] \quad & C = \langle 0,1,2,3,3,5,8,0,0,0\rangle \\
C[7] = C[7] + C[6] \quad & C = \langle 0,1,2,3,3,5,8,8,0,0\rangle \\
C[8] = C[8] + C[7] \quad & C = \langle 0,1,2,3,3,5,8,8,8,0\rangle \\
C[9] = C[9] + C[8] \quad & C = \langle 0,1,2,3,3,5,8,8,8,8\rangle
\end{aligned}$$

The **for** loop in lines 6–7 of CountingSort:

$$B = \langle\ ,\ ,\ ,\ ,\ ,\ ,\ ,\ \rangle$$
$$C = \langle 0, 1, 2, 3, 3, 5, 8, 8, 8, 8\rangle$$

$B[C[6]] = (6, 736)$ $B = \langle\ ,\ ,\ ,\ ,\ ,\ ,\ , (6, 736)\rangle$
$C[6] = C[6] - 1$ $C = \langle 0, 1, 2, 3, 3, 5, 7, 8, 8, 8\rangle$

$B[C[1]] = (1, 111)$ $B = \langle(1, 111),\ ,\ ,\ ,\ ,\ ,\ , (6, 736)\rangle$
$C[1] = C[1] - 1$ $C = \langle 0, 0, 2, 3, 3, 5, 7, 8, 8, 8\rangle$

$B[C[2]] = (2, 522)$ $B = \langle(1, 111), (2, 522),\ ,\ ,\ ,\ ,\ , (6, 736)\rangle$
$C[2] = C[2] - 1$ $C = \langle 0, 0, 1, 3, 3, 5, 7, 8, 8, 8\rangle$

$B[C[5]] = (5, 435)$ $B = \langle(1, 111), (2, 522),\ ,\ , (5, 435),\ ,\ , (6, 736)\rangle$
$C[5] = C[5] - 1$ $C = \langle 0, 0, 1, 3, 3, 4, 7, 8, 8, 8\rangle$

$B[C[6]] = (6, 916)$ $B = \langle(1, 111), (2, 522),\ ,\ , (5, 435),\ , (6, 916), (6, 736)\rangle$
$C[6] = C[6] - 1$ $C = \langle 0, 0, 1, 3, 3, 4, 6, 8, 8, 8\rangle$

$B[C[3]] = (3, 623)$ $B = \langle(1, 111), (2, 522), (3, 623),\ , (5, 435),\ , (6, 916), (6, 736)\rangle$
$C[3] = C[3] - 1$ $C = \langle 0, 0, 1, 2, 3, 4, 6, 8, 8, 8\rangle$

$B[C[5]] = (5, 125)$ $B = \langle(1, 111), (2, 522), (3, 623), (5, 125), (5, 435),\ , (6, 916), (6, 736)\rangle$
$C[5] = C[5] - 1$ $C = \langle 0, 0, 1, 2, 3, 3, 6, 8, 8, 8\rangle$

$B[C[6]] = (6, 286)$ $B = \langle(1, 111), (2, 522), (3, 623), (5, 125), (5, 435), (6, 286), (6, 916), (6, 736)\rangle$
$C[6] = C[6] - 1$ $C = \langle 0, 0, 1, 2, 3, 3, 5, 8, 8, 8\rangle$

2. CountingSort on the second digit:

input: $A' = \langle(1, 111), (2, 522), (2, 623), (2, 125), (3, 435), (8, 286), (1, 916), (3, 736)\rangle$

output: $B = \langle(1, 111), (1, 916), (2, 522), (2, 623), (2, 125), (3, 435), (3, 736), (8, 286)\rangle$

3. CountingSort on the second digit:

input: $A' = \langle(1, 111), (5, 522), (6, 623), (1, 125), (4, 435), (2, 286), (9, 916), (7, 736)\rangle$

output: $B = \langle(1, 111), (1, 125), (2, 286), (4, 435), (5, 522), (6, 623), (7, 736), (9, 916)\rangle$

RadixSort output:

$$A = \langle 111, 125, 286, 435, 522, 623, 736, 916\rangle$$

(c) *(L2)* RadixSort is normally used to sort integers in a given range (as each value has at most d digits). What if we simply use CountingSort instead? Analyze the running time of CountingSort on an input of n integers with at most d digits each.

Solution:
We know that CountingSort takes $\Theta(n + k)$ time if the input keys are integers in the range $0..k$. If the input keys are integers with at most d digits, then they fall in the range $0..10^d - 1$. So, if we use CountingSort to sort n integers with at most d digits each, the running time will be $O(n + 10^d - 1)$, which grows exponentially with d.

Exercise 9 *(L3)* Design an $O(n)$ running time algorithm to sort n words in lexicographic order. Each word has exactly l letters and $l = O(1)$. (*You may assume only basic characters occur.*)

Solution:

Highlights:

- It is interesting to consider how to do this if not every word has exactly l letters, but at most l letters.

We will use a modification of RadixSort. We will use a bijective function mapping letters to integers that preserves the lexicographic order, such as, for example, ASCII character map. Then, we can simply use CountingSort to sort the words in the input array on each letter from right to left.

```
LexSort($A, l$)
    Input: Array $A$ of words of length $l$ each
    Output: Array $A$ sorted in lexicographic order
1   for $i = l$ downto 1
2       sort the words in $A$ by their $i$-th letter with CountingSort
        using ASCII character codes as the keys
```

Proof of correctness: is similar to the proof of correctness of the RadixSort. See Exercise 7 of this practice set.
Running time: The running time of line 2 depends on the maximum value of the ASCII character codes, which is 122 (for letter 'z'). Thus, the total running time is $O(l(122+n)) = O(n)$ if $O(l) = 1$.

▶ Lecture 5 QuickSort and selection

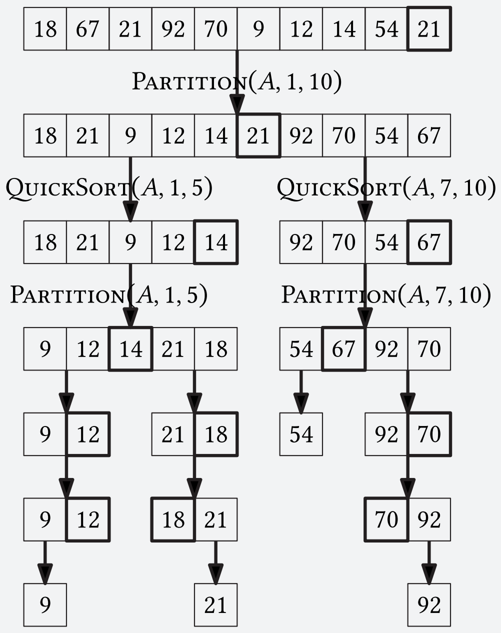
Exercise 10

(a) *(L2)* Illustrate the execution of QUICKSORT step by step on the following input:

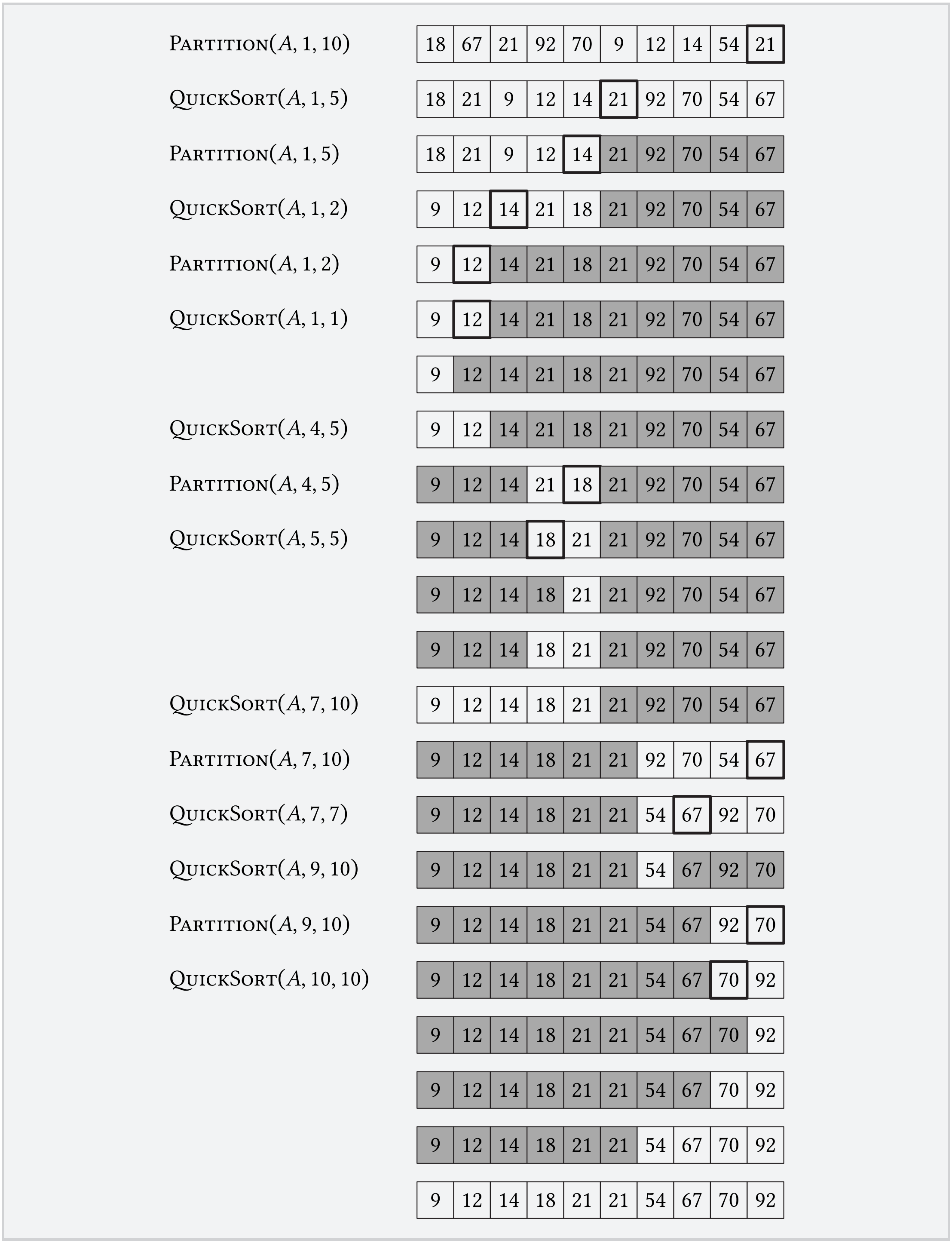
$$[18, 67, 21, 92, 70, 9, 12, 14, 54, 21]$$

One call to PARTITION counts as one step.

Solution:
The following figure illustrates the schematics of the PARTITION calls in line 2, and the recursive calls in lines 3–4. The pivots are framed in bold, and the first few calls to the PARTITION and the recursive calls to QUICKSORT are labelled.



The next figure illustrates the sequential changes in the array and the corresponding calls to the PARTITION and the recursive calls. At each step, the array is shown at the moment of the call of the corresponding procedures. The input array at each level of recursion is shown in white, while the rest of the array is shown grey.



Partition(A, 1, 10)	18	67	21	92	70	9	12	14	54	$\boxed{21}$
QuickSort(A, 1, 5)	18	21	9	12	14	$\boxed{21}$	92	70	54	67
Partition(A, 1, 5)	18	21	9	12	$\boxed{14}$	21	92	70	54	67
QuickSort(A, 1, 2)	9	12	$\boxed{14}$	21	18	21	92	70	54	67
Partition(A, 1, 2)	9	$\boxed{12}$	14	21	18	21	92	70	54	67
QuickSort(A, 1, 1)	9	$\boxed{12}$	14	21	18	21	92	70	54	67
	9	12	14	21	18	21	92	70	54	67
QuickSort(A, 4, 5)	9	12	14	21	18	21	92	70	54	67
Partition(A, 4, 5)	9	12	14	21	$\boxed{18}$	21	92	70	54	67
QuickSort(A, 5, 5)	9	12	14	$\boxed{18}$	21	21	92	70	54	67
	9	12	14	18	21	21	92	70	54	67
	9	12	14	18	21	21	92	70	54	67
QuickSort(A, 7, 10)	9	12	14	18	21	21	92	70	54	67
Partition(A, 7, 10)	9	12	14	18	21	21	92	70	54	$\boxed{67}$
QuickSort(A, 7, 7)	9	12	14	18	21	21	54	$\boxed{67}$	92	70
QuickSort(A, 9, 10)	9	12	14	18	21	21	54	67	92	70
Partition(A, 9, 10)	9	12	14	18	21	21	54	67	92	$\boxed{70}$
QuickSort(A, 10, 10)	9	12	14	18	21	21	54	67	$\boxed{70}$	92
	9	12	14	18	21	21	54	67	70	92
	9	12	14	18	21	21	54	67	70	92
	9	12	14	18	21	21	54	67	70	92
	9	12	14	18	21	21	54	67	70	92

(b) *(L3)* In general, how many times are two arbitrary keys x and y compared during the execution of Quicksort?

Solution:
Two keys are compared when one of them is a pivot. W.l.o.g. assume that x is a pivot when x and y are compared for the first time. Then, x is not part of an input to the lower level recursive calls, thus x and y cannot be compared for the second time.

On the other hand, it is possible for two keys not to be compared at all, when they are separated by another pivot. Thus, in general, any two keys can be compared either 0 or 1 times.

Exercise 11 *(L3)* In the algorithm Select, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7?

Solution:

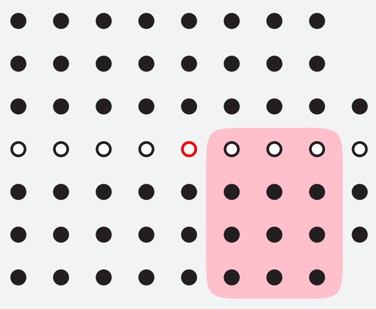
Highlights:

- To solve this simply follow the example from the lecture, but choose other values.
- Note that we are too lazy to work out all 126 base cases (fair enough I think). However as $n < 126$ the value of n is bound by a constant and thus the runtime for all these base cases is also bound by a constant. The biggest constant runtime over all 126 base cases is still constant and hence we can bound it by some slightly larger constant c_1.
- Note that the choice of $n \geq 126$ is explained later in the proof. Choosing $n = 140$ as in the lecture slides works equally well.

We consider the following variation of the algorithm Select:

1. Divide the n elements into the groups of 7 $\Rightarrow \lceil n/7 \rceil$ groups
2. Sort each of $\lceil n/7 \rceil$ groups and find their medians
3. Find the median x of the $\lceil n/7 \rceil$ medians recursively
4. Partition the array around $x \Rightarrow x$ is kth element in the array
5. If $i = k$ return x. If $i < k$, recursively find the ith smallest element on the low side. If $i > k$, recursively find the $(i - k)$th smallest element on the high side.

For groups of 7, the algorithm still works in linear time. To see that we repeat the computation of the number of elements larger and smaller than the median of medians (similarly to the lectures).
In the following image we schematically depict the $\lceil n/7 \rceil$ groups of 7 elements, sorted from top to bottom. The red point represents the median of medians x.



We lowerbound the number of elements in the array that are definitely larger than the median of medians x. Firstly, half of the $\lceil n/7 \rceil$ groups have a median that is larger than x. The elements in these groups that are larger than their medians are by transitivity also larger than x (these elements are highlighted in red in the figure). The last group may be incomplete, and the group containing x itself has fewer elements larger than x is well. We are left with $\lceil \frac{1}{2}\lceil n/7 \rceil \rceil - 2$ groups with each at least 4 elements that are larger than x. Then, the number of elements that are greater than x is at least

$$4(\lceil \tfrac{1}{2}\lceil n/7 \rceil \rceil - 2) \geq 2n/7 - 8\,.$$

Similarly, the number of elements that are smaller than x is at least

$$4(\lceil \tfrac{1}{2}\lceil n/7 \rceil \rceil - 2) \geq 2n/7 - 8\,.$$

That implies, that the largest size of the subproblem the algorithm recurses into is at most $n - (2n/7 - 8) = 5n/7 + 8$.
The recurrence becomes

$$T(n) = \begin{cases} O(1) & \text{if } n < 126 \\ T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n) & \text{if } n \geq 126 \end{cases}$$

which solves to $T(n) = O(n)$. We will prove this fact by substitution:
Let a be a positive constant such that the $O(n)$ term in the recursion is $\leq a \cdot n$ for all $n > 0$. We will show that there exists such positive c that $T(n) \leq c \cdot n$ for all sufficiently large n.
Base case ($0 < n < 126$): For these n, we have that $T(n) = O(1)$. Let c_1 be a positive constant large enough that $T(n) \leq c_1 n$ for all $0 < n < 126$. Then the base case will hold for any choice of $c \geq c_1$.
Inductive step:
IH: Assume that for some fixed $n \geq 126$, $T(k) \leq cn$ for all $0 < k < n$. Then, we will show that $T(n) \leq cn$:

$$\begin{aligned} T(n) &\leq c\lceil n/7 \rceil + c(5n/7 + 8) + an && \{\text{IH on } T(\lceil n/7 \rceil) \text{ and } T(5n/7 + 8)\} \\ &\leq cn/7 + c + 5cn/7 + 8c + an \\ &= 6cn/7 + 9c + an \\ &= cn + (-cn/7 + 9c + an)\,. \end{aligned}$$

The inequality holds when $(-cn/7 + 9c + an) \leq 0$. That is, when

$$c \geq \frac{7an}{n - 63}$$

For all $n \geq 2 \cdot 63 = 126$, the fraction $\frac{n}{n-63} \leq 2$ (thus the choice of 126 for the recurrence formula). Thus, if we choose $c \geq 7a \cdot 2 = 14a$, the inequality $(-cn/7 + 9c + an) \leq 0$ will hold for all $n \geq 126$.
In conclusion, if we choose $c = \max\{c_1, 14a\}$, the inequalities in the base case and in the inductive step will hold, and thus, $T(n) \leq cn$ for all $n > 0$. That is, $T(n) = O(n)$.
For groups of 7, the algorithm Select still works in linear time.

Exercise 12

(a) *(L1)* Give a step-by-step description of QUICKSORT using linear-time median finding as a subroutine to find a good pivot.

Solution:
If we use the median as a pivot in QUICKSORT, we get a balanced partition of the array into two smaller subarray that we can sort recursively. The algorithm SELECT run on the array A with a parameter $\lfloor \frac{n}{2} \rfloor$ will return a median element m from this array. Then, we need to partition the array A into two subarrays, one containing the elements with keys $\leq m.key$, and the other, containing the elements with keys $> m.key$.
SELECT returns the element m itself, and not its index in the array A. So, to partition the array around the pivot m, we first need to find the index of m in the array A. We can do so by linearly scanning the array for $i = 1$ to n and comparing $A[i].key$ to $m.key$. Once an index i_m of m is found (note, that there may be multiple elements with the same key), we swap the elements $A[i_m]$ and $A[n]$, and call PARTITION.

```
QUICKSORTM($A, p, r$)
1 if $p < r$
2     $m$ = SELECT($A[p : r]$, $\lfloor \frac{r+1-p}{2} \rfloor$) // Find median in subarray $A[p : r]$
3     $i = p$
4     while $A[i].key \neq m.key$
5         $i = i + 1$
6     swap elements $A[i] \leftrightarrow A[r]$
7     $q$ = PARTITION($A, p, r$)
8     QUICKSORTM($p, q - 1$)
9     QUICKSORTM($q + 1, r$)
```

(b) *(L2)* Assuming that all the elements of the input array are distinct, analyze the running time of your algorithm. What changes if the elements need not be distinct?

Solution:
If all the elements are distinct, we can be sure that the sizes of the subarrays in the recursive calls in lines 8 and 9 are the same (or differ by at most 1). Then, we obtain recurrence $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$.
If the elements are not distinct, then we cannot claim that the sizes of the subproblems are the same.

(c) *(L4)* Modify the algorithm to also handle duplicate elements efficiently.

Solution:
To make our algorithm QUICKSORTM handle the case of duplicate elements, what we need is to remove all the elements with the same keys as $m.key$ from the recursive calls. Then we can no longer simply use PARTITION as a subroutine.
We can create three auxiliary arrays B, C, and D, with B containing all the elements of A with keys that are strictly less than $m.key$, C containing all the elements of A with

keys that are larger than $m.key$, and D containing all the elements from A with keys that are equal to $m.key$ (and the element m as well).
Our algorithm becomes:
Find the median m, create three new arrays B, C, and D. For each element in A put it either in B, C, or D, depending on the $A[i].key$. Recursively sort the two arrays B and C, and return the concatenation B, D, and C.
The sizes of the arrays B and C are not greater than $n/2$, thus, we have the recurrence $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$.
Note, that this solution can be modified to obtain an in-place algorithm. Instead of creating auxiliary arrays we can create pointers that will mark the boundaries between the set of elements strictly smaller than the pivot, the set of elements equal to the pivot, and the set of elements strictly larger than the pivot.

Exercise 13 *(L4)* Let $A[1:n]$ and $B[1:n]$ be two sorted arrays, each containing n numbers. Describe an algorithm that finds the median of all $2n$ elements in arrays A and B in $O(\log n)$ time. Don't forget to analyze the running time and prove the correctness of your algorithm.

Solution:
Algorithm. We define a recursive algorithm. When $n = 1$ we simply return the smaller of the two elements. Otherwise, we make a case distinction based on whether n is odd or even. In either case let $k = \lceil \frac{n}{2} \rceil$ and let $m_A = A[k]$, the median of A and $m_B = B[k]$, the median of B.

- n is odd. If $m_A \leq m_B$ then recurse on $A[k:n]$ and $B[1:k]$, otherwise recurse on $A[1:k]$ and $B[k:n]$.
- n is even. If $m_A \leq m_B$ then recurse on $A[k+1:n]$ and $B[1:k]$, otherwise recurse on $A[1:k]$ and $B[k+1:n]$.

Running time. Each operation except for the recurrence can be done in $O(1)$ time. There is only a single recursive call on two arrays of size k. Since $k = \lceil \frac{n}{2} \rceil$ we can recurse $O(\log n)$ times until we have reduced our arrays to a size of 1, so the resulting running time is $O(\log n)$.
Correctness. We first introduce the main idea of this algorithm before proving correctness. Let C be an array and $m(C)$ the median of C. If we remove equally many elements that are smaller than $m(C)$ as we remove elements that are larger than $m(C)$, then $m(C)$ is still the median of the remaining number. For example, consider the numbers $1, 2, \ldots, 9$ with median 5. Now consider the numbers $1, 4, 5, 8, 9$ where we remove two numbers smaller than 5 and two larger than 5, the median is still 5. This is our goal in the algorithm, to remove equally many elements that are smaller and larger than the median.
For the correctness proof we first assume that all elements are distinct and will show how to adjust the proof for inputs with equal elements at the end.
We make our proof by induction on n.
The **base case** $n = 1$ is simple as the lower median of two elements is the smaller of the two, which is what the algorithm returns.
For the **step case** assume (IH) that our algorithm returns the median of two (sorted) subarrays of length k for any $1 \leq k < n$. We then prove that the algorithm returns the median of two (sorted) arrays of size n.
First consider the case where n is odd and $m_A < m_B$. (Note that since we assume distinct elements we can ignore equality of m_A and m_B.) In this case we know that there are at least $2k - 1 = n$ elements that are $> m_A$, namely $k - 1$ in A and k in B. It follows that any element in $A[1:k-1]$ has at least $n + 1$ elements that are larger and therefore these elements are all smaller than the median of both arrays combined. Similarly there are at least $2k - 1 = n$ elements $< m_B$, so elements in $B[k+1:n]$ are all larger than the median. Conveniently $A[1:k-1]$ and $B[k+1:n]$ both contain exactly $k - 1$ numbers so removing both ensures that the median of the remaining numbers is still the median of the entire set. The arrays on which we recurse have size $k < n$, so by our IH we know that the algorithm returns the correct median of $A[k:n]$ and $B[1:k]$, which is then also the correct median of A and B.
The case where n is odd and $m_A > m_B$ is completely symmetric.
Next consider the case where n is even and $m_A < m_B$. In this case there are at least $2k+1 = n+1$ elements that are $> m_A$ and at least $2k-1 = n-1$ elements $< m_B$. Following the same reasoning as above, for every element in $A[1:k]$ there are at least $n + 1$ elements that are larger, so

these are all smaller than the median. Similarly for every element in $B[k+1:n]$ there are at least n elements that are smaller so x is larger than the median. The subarrays $A[1:k]$ and $B[k+1:n]$ both contain exactly k numbers so removing both ensure that the median of the remaining numbers is still the median of the entire set. The arrays on which we recurse have size $k < n$, so by our IH we know that the algorithm returns the correct median of $A[k+1:n]$ and $B[1:k]$, which is then also the correct median of A and B.
The case where n is even and $m_A > m_B$ is again completely symmetric.
Since we have proven the algorithm returns a median for arrays of size 1 and for size $n \geq 2$ given that the algorithm returns the median for arrays of size $1 \leq k < n$, by the principle of strong induction, the algorithm returns the median of two arrays of any length $n \geq 1$.
Equal elements. Now to deal with an array that contains equal elements. This requires two observations. The first is that when removing an equal amount of elements larger and smaller than the median we may also remove elements equal to the median and count them as either larger or smaller, so long we as we do not remove all of them. To see this consider that we can impose an arbitrary order on elements of equal values to break ties and choose such an order to decide which elements are larger and which are smaller. Careful analysis of our correctness proof shows that we never remove all equal elements unless we have established they are strictly larger or smaller than the median.

▶ Lecture 6 Hash tables

Exercise 14 *(L4)* (This is exercise 11.1-1 from the book.) A dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

Solution:
Option 1: We can go through all entries in the table and keep track of the maximum element we have found so far so we can report it at the end. The running time is $O(|U|)$ in the worst case since you have to go through all entries in the table.
Option 2: We can search for all values in the universe from highest to lowest and as soon as we find a value in the table we know it is the maximum. This however may still take $O(|U|)$ time in the worst case as in the worst case we may need to go through all elements in the universe.

Exercise 15

(a) *(L3)* You have a universe of 30 numbers $\{0, 1, \dots, 29\}$ and a hash-table of size 10. Which hash-function is better, $h_1(k) = k \mod 10$, or $h_2(k) = \lfloor k/10 \rfloor$?

Solution:
The first hash-function selects the last digit of a number as a hash value, and the second – the first. The first hash function is better because its image covers the whole hash-table, whereas the image of the second hash function, $\{0, 1, 2\}$, is only a subset of the hash-table. Moreover, the first hash-function distributes the hash-values evenly over the hash table, thus if the keys drawn from the universe are random, there will be few collisions.

(b) *(L3)* How many different hash functions exist that map from a universe of size n to the integers 1 to m? Assuming that $n = a \cdot m$, how many different uniform hash functions exist?

Solution:
The number of functions that map from a universe of size n to the integers $1..m$ is m^n, as each of the n elements can map to one of m integers.
The number of functions that map from a universe of size $a \cdot m$ to the integers $1..m$ uniformly can be counted in the following way: pick a subset of a elements from the universe and map them to 1, pick a subset of a elements remaining in the universe and map them to 2, and so on. Thus, the total number is

$$\binom{n}{a}\binom{n-a}{a}\binom{n-2a}{a}\cdots\binom{a}{a} = \frac{n!}{a!(n-a)!}\,\frac{(n-a)!}{a!(n-2a)!}\cdots\frac{a!}{a!} = \frac{n!}{(a!)^m}$$

Exercise 16 *(L3)* Consider the family of hash functions defined by

$$h_{ab}(k) = ((ak + b) \mod p) \mod m,$$

where m is the table size, p is a large prime and $a \geq 1$ and $b \geq 0$ are integers that are at most $p - 1$. As explained in the book, this family of hashing functions is universal. The values p and m are chosen in the design of the algorithm and are fixed. The values a and b are chosen at random when the hash table is initialized. We could instead pick these values either earlier or later, but this may result in changes to the behaviour or analysis of the algorithm.

(a) How does the behavior and analysis of the hashing function change if we pick a and b when designing the algorithm?

> **Solution:**
> The behavior of the hash function does not change much. The only difference is that a and b are always the same between different initializations of the hash table. However, the main purpose of choosing these at runtime is to ensure the analysis is not dependent on the distribution of the input. When we fix a and b then this is no longer a universal hashing scheme.

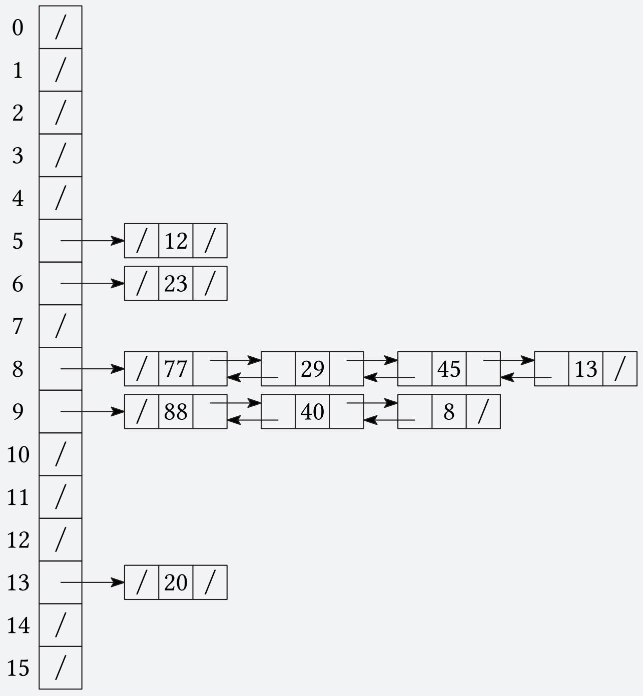
(b) How does the behavior and analysis of the hashing function change if we pick a and b later? That is, if we do not fix them when the table is initialized, but pick new random a and b each time the hash function is used?

> **Solution:**
> This would be a terrible idea as such a hash function is not usable in a hash table. To be able to search for an element in the hash table one would need to use the same a and b when searching as when the element was inserted. If these are chosen at random each time we can no longer search correctly.

Exercise 17

(a) *(L2)* Draw the hash table of length $m = 16$ resulting from hashing the keys 8, 12, 40, 13, 88, 45, 29, 20, 23, and 77, using the hash function $h(k) = (3k + 1) \mod 16$ and assuming collisions are handled by chaining. Is $m = 16$ a good choice for the size of a table? Why or why not?

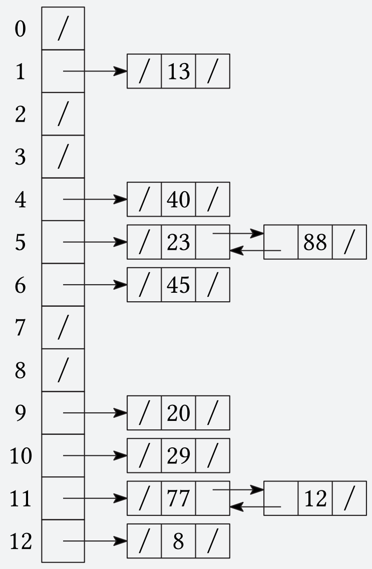
Solution:
The following image illustrates the keys inserted into a hash-table of size 16:



For the given hash function, $m = 16$ is a bad choice for the size of a hash table because it is a power of 2. When we are using the division method to create a hash function, if the size of the table $m = 2^p$, then the hash values will only depend on the lower p bits. This may lead to more collisions.

(b) *(L2)* Repeat the previous exercise for a table of size $m = 13$ using the hash function $h(k) = (3k+1) \mod 13$. Why is $m = 13$ a better choice for the size of a hash table than $m = 16$?

Solution:
The following image illustrates the keys inserted into a hash-table of size 13:



$m = 13$ is a better choice for a size of a hash table when using the division method to create a hash function because it is prime and it is far enough from a power of two. A hash function created by using the division method will not only depend on the lower bits of the keys.

(c) *(L2)* What is the result of the previous exercise, assuming collisions are handled by linear probing?

Solution:

77	13			40	88	45	23		20	29	12	8
0	1	2	3	4	5	6	7	8	9	10	11	12

For keys $\{8, 12, 40, 13, 88, 45, 29, 20\}$ there were 0 unsuccessful probes. For keys 23 and 77 there were 2 unsuccessful probes. The total number of unsuccessful probes is 4.

(d) *(L2)* What is the result of (b) assuming collisions are handled by double hashing with a primary hash function $h'(k) = h(k)$ and a secondary hash function $h''(k) = k \mod 15$?

Solution:

23	13	77		40	88	45			20	29	12	8
0	1	2	3	4	5	6	7	8	9	10	11	12

For keys $\{8, 12, 40, 13, 88, 45, 29, 20\}$ there were 0 unsuccessful probes. For key 23 there was 1 unsuccessful probe, and for key 77 there were 2 unsuccessful probes. The total number of unsuccessful probes is 3.

For sub-problems (c) and (d), note for every key how often you had to probe unsuccessfully before you could insert the key, and for each hashing scheme the total number of unsuccessful probes.

Exercise 18 *(L3)* Describe algorithms for Hash-Chain-Insert, Hash-Chain-Delete, and Hash-Chain-Search for a hash table with chaining, where the lists are stored in a sorted order. Analyze the running time of the algorithms.

Solution:

(a) Hash-Chain-Insert: instead of inserting x at the head of $T[h(x.key)]$ we need to find the proper position of x in the list. So, we compare the elements of $T[h(x.key)]$ one by one with x until we reach the end of the list or until we find an element y such that $y.key > x.key$. Then, we insert x in from of y, or at the end of the list if no such y was found.

Running time analysis: The worst-case running time of Hash-Chain-Insert is $\Theta(n)$ (if all the elements hash into the same slot, and x is the maximum element, then the whole list needs to be traversed until we can insert x). On average, if the load factor of the hash table is α, the running time of Hash-Chain-Insert will be $\Theta(1+\alpha)$.

(b) Hash-Chain-Search: search for an element with key k in the list $T[h(k)]$. The algorithm is the same as for the non-sorted lists. The running time is $\Theta(1+\alpha)$.

(c) Hash-Chain-Delete: delete element x from the list $T[h(x.key)]$. The algorithm is the same as for the non-sorted lists: when deleting x from a sorted list, the sorted order of the remaining elements is preserved. The running time is $\Theta(1)$ if the lists are doubly linked.

Exercise 19 *(L4)* Consider two sets of integers, $S = \{s_1, s_2, ..., s_m\}$ and $T = \{t_1, t_2, ..., t_n\}$, $m \leq n$. Describe an $O(n)$ expected time algorithm that uses a hash table of size m to test whether S is a subset of T. You may assume the existence of a suitable hash function that satisfies the assumption of simple uniform hashing.

Solution:
Let H be the hash table of size m and $h()$ be a corresponding uniform hash function. The algorithm is the following: First, insert all the elements of T into H. Next, for each element s_i in S search for s_i in H. If for some s_i the search was unsuccessful, then $s_i \notin T$, and thus $S \not\subset T$. Otherwise, report that $S \subset T$.

```
CheckSubset($S, T$)
    Input: Two sets of integers $S$ and $T$, $|S| = m$, $|T| = n$, $m \leq n$
    Output: TRUE if $S \subset T$, FALSE otherwise
1   create hash table $H$ of size $m$ with chaining; uniform hash function $h()$
2   for $i = 1$ to $n$
3       Hash-Chain-Insert($H, T[i]$)
4   for $i = 1$ to $m$
5       if Hash-Chain-Search($H, S[i]$) == NIL
6           return FALSE
7   return TRUE
```

Proof of correctness: After lines 1-3 all the values of T will be stored in hash-table H.

If $S \subset T$, then each element of S must also be an element of T and hence the element must be inserted in H in line 1-3. Thus if $S \subset T$, the guard of the if in line 5 will never be satisfied. Once the for-loop of line 4-6 is finished the algorithm correctly reports **true**.
If $S \not\subset T$, then there will be an element $s_i \in S$ such that $s_i \notin T$. Thus when in line 5 searching for s_i in H will return NIL. But then the guard of line 6 is satisfied and the algorithm correctly reports **false**.
Running time analysis: Line 1 takes $O(n)$ time. Line 3 takes $O(1)$ time as inserting an element can be done in $O(1)$ time. Thus the for-loop in lines 2-3 takes $\sum_{i=1}^{n} O(1) = O(n)$ time. Line 7 takes $O(1)$ time.
The load factor of the hash table is $\alpha = n/m$, thus the expected number of elements in each cell is α. Line 5 takes $O(n_i)$ time in expectation, where n_i is the number of elements in the cell into which the element $S[i]$ hashes. The expected value of the total running time of the for-loop in lines 4-6 is $E[\sum_{i=1}^{m} O(n_i)] = \sum_{i=1}^{m} O(E[n_i]) = \sum_{i=1}^{m} O(\alpha) = O(\alpha) \cdot m = O(\alpha \cdot m) = O(n)$.

Exercise 20 *(L3)* (This is Exercise 11.3-1 from the book.) You wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might you take advantage of the hash values when searching the list for an element with a given key?

Solution:
Comparing two long strings to see if they are the same takes time proportional to the length of the string. Doing this repeatedly is not very efficient if the strong are long. However, we know from how hash functions work that if two strings have different hash values, then they cannot be the same string. We can use this to our advantage by first comparing the hash values, if they are not the same we can move on to the next string. If they are the same we still have to compare the strings as it is possible that two different strings have the same hash value.