# SFU CMPT-307 2008-2 Lecture: Week 7

**Ján Maňuch**

**E-mail: jmanuch@sfu.ca**

**Lecture on June 17, 2008, 5.30pm-8.20pm**

# Selection problem

**Sorting** yields a **complete information** on order of the input elements

But what if we don't really **need** all this information, but only want to know the value of the $k$-th smallest element?

Sorting clearly solves this problem, but it might take $\Theta(n \log n)$ time. Is there any faster way? Can we do it in linear time?

Simple for the smallest, the 2nd smallest, the $k$-th smallest element, where $k$ is a constant

But what about $n/2$-th smallest (also called median)? $\sqrt{n}$-th smallest?

We will assume for convenience that all elements are distinct numbers. But everything should work also without this assumption.

Exact formulation of the **selection problem**:
**Input:** Set $A$ of $n$ (distinct) numbers and a number $i \in \{1, \ldots, n\}$
**Output:** $x \in A$ that is larger than exactly $i - 1$ other elements of $A$

*Terminology:*

- element larger than exactly $i - 1$ other elements = the $i$-th smallest element = the $i$-th **order statistics**
  minimum = 1-st order statistics
  maximum = $n$-th order statistics
- $\lceil n/2 \rceil$-th smallest element is called the **median** (sometimes also the lower median)
  when $n$ is odd, the median is element exactly in the middle of sorted array
  when $n$ is even, there are 2 elements in the middle, the lower and upper medians

# Minimum and maximum

classic algorithm:

**Minimum**$(A)$

```
1: min ← A[1]
2: for i ← 2 to length(A) do
3:     if min > A[i] then
4:         min ← A[i]
5:     end if
6: end for
```

let $n$ be the number of elements of $A$

how many comparisons? obviously, $n - 1$

*Question:* is this the best?

yes:

– an algorithm that determined the minimum can be viewed as a competition among the elements consisting of matches = comparisons..

– winner of a match is the smaller element

– now every element, except the minimum, has to lose at least one match

– hence, at least $n - 1$ comparisons

# Simultaneous minimum and maximum

– in many applications one has to find the minimum and the maximum at the same time

we could use the above algorithm, to determine separately the minimum and separately the maximum — requires $2n - 2$ comparisons

*Question:* is this the best?

$\lfloor n/2 \rfloor + 2(\lceil n/2 \rceil - 1) = \lceil 3n/2 \rceil - 2$ comparisons are sufficient:
– maintain the minimum and the maximum elements seen so far
– in each step take a pair of unexplored elements
– compare them with each other
– the smaller one cannot be the maximum, hence it's enough to compare it with the current minimum
– similarly, it's enough to compare the bigger element of the pair with the current maximum

**Assignment Problem 7.1.** (deadline: June 24, 5:30pm)
Write the algorithm for finding the minimum and maximum elements of the array simultaneously. Verify that the number of comparisons it needs is $\lceil 3n/2 \rceil - 2$.

**Extra Assignment Problem 2.** (1.5% 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.)

Show that $\lceil 3n/2 \rceil - 2$ comparisons are necessary in the worst case to find both the maximum and minimum of $n$ numbers.

**Hint:** Consider how many numbers are potentially either the maximum or minimum, and investigate how a (different type of) comparison affects these counts.

**Exercise 7.1.** Show that the second smallest of $n$ elements can be found with $n + \lceil \log n \rceil - 2$ comparisons.

# The i-th element

We have seen that we can find the minimum and maximum elemets in linear time. Can we find the $i$-th smallest element in linear time?

*For linear time in average case, we could use the QuickSort idea:*

Basic idea:
– perform Quicksort, but solve recursively only one of 2 subproblems.

**Problem:** Might be that we partition in a bad way and in each step of recursion follow a large sub-problem: $\Omega(n^2)$

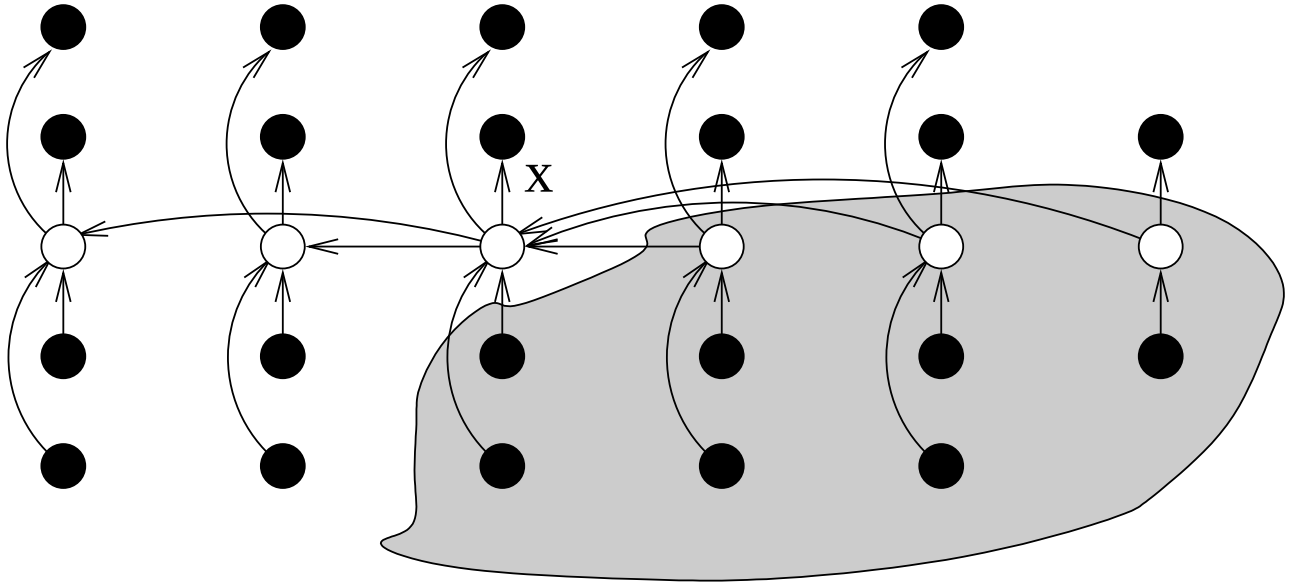
*Question.* Can we modify the algorithm so that it works in linear time also in the worst case?

- We need to make splits more balanced, which would result in $\Theta(n)$ running time. We will use more complicated algorithm to find a suitable pivot.

**Select**$(A, i)$

1. Divide $n$ elements into $\lfloor n/5 \rfloor$ groups of $5$ elements each, and at most one group containing the remaining: $n \bmod 5 < 5$ elements
2. Find the median of each of the $\lceil n/5 \rceil$ groups by sorting each one, and then picking median from sorted group elements
3. Call **Select** recursively on set of $K = \lceil n/5 \rceil$ medians found above with $i = \lceil K/2 \rceil$, giving median-of-medians $x$ (a pivot)
4. Partition input around $x$. Let $k$ be the number of elements on low side plus one, so that $x$ is the $k$-th smallest element and there are $n - k$ elements on high side of partition
5. If $i = k$, return $x$. Otherwise use **Select** recursively to find the $i$-th smallest element of low side if $i < k$, or the $(i - k)$-th smallest on high side if $i > k$

**Observations:**




- $n = 5 \cdot 5 + 3 = 28$ elements are circles
- groups are columns
- white circles are medians of groups
- $x$ is median of medians
- arrows from greater to smaller elements:

  three out of every full group to right of $x$ are greater than $x$, and three out of every group to left of $x$ are smaller than $x$ (simply because the corresponding medians are greater/smaller than $x$).
- elements on shaded background are guaranteed to be greater than $x$

*We want to lower-bound the number of elements greater (resp. smaller) than $x$.*

*Note:* approx. half of medians found in step 2 are greater than $x$

More precisely, since $x$ is $\lceil K/2 \rceil$-th element of the $K$ medians, there are $K - \lceil K/2 \rceil = \lfloor K/2 \rfloor$ medians larger than $x$. Each except the last one contributes 3 elements greater than $x$. The last one contributes at least one such element and we have 2 elements in the same group as $x$ larger than $x$.

Hence, we have at least

$$3(\lfloor K/2 \rfloor - 1) + 2 + 1 = 3 \left\lfloor \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rfloor \geq 3 \left( \frac{\lceil \frac{n}{5} \rceil - 1}{2} \right) \geq \frac{3n}{10} - \frac{3}{2}$$

Same is true for the number of elements smaller than $x$

Thus, in the worst case, **Select** is called recursively on at most

$$n - 1 - \left( \frac{3n}{10} - \frac{3}{2} \right) = \frac{7n}{10} + \frac{1}{2}$$

elements (in step 5)

Steps 1, 2, and 4 take $O(n)$ time each (step 2: $O(n)$ calls to insertion sort on sets of size $O(1)$)

Step 3: time $T(\lceil n/5 \rceil)$

Step 5: time at most $T(7n/10 + 1/2)$

Altogether:

$$T(n) \leq \begin{cases} \Theta(1) & n \leq 5 \\ T(\lceil n/5 \rceil) + T(7n/10 + 1/2) + O(n) & n > 5 \end{cases}$$

*Guess:* $T(n) = \mathcal{O}(n)$

Want to show that $T(n) \leq cn$ for some constant $c$

Assume $T(n) \leq cn$ for $c$ large enough and $n < 14$ (no problem, since all are constants)

Also, pick constant $a$ such that the $O(n)$ term is at most $an$ (the non-recursive component)

Let $n \geq 14$.

**Induction hypothesis:** for every $m < n$, we have $T(m) \leq cm$.

Substituting induction hypothesis we get:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 1/2) + an \\ &\leq c\frac{n+4}{5} + c(7n/10 + 1/2) + an \\ &= cn/5 + 4c/5 + 7cn/10 + c/2 + an \\ &= 9cn/10 + 13c/10 + an \\ &= cn + (-cn/10 + 13c/10 + an) \end{aligned}$$

Now $T(n) \leq cn$ if and only if

$$-cn/10 + 13c/10 + an \ \leq \ 0$$

equivalent to

$$c \geq 10a\frac{n}{n-13}$$

when $n \geq 14$.

Since, when $n \geq 14$, $\frac{n}{n-13} \geq 14$, it's enough to select $c \geq 140a$.

**Assignment Problem 7.2.** (deadline: June 24, 5:30pm)
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

(a) 7?

(b) 3?

In both cases find the worst-case partitioning, and build the recurrence for the worst-case running time (as we did on the lecture). Show in case (a), $T(n)$ can be upperbounded by $cn$, and in case (b), $T(n)$ can be lowerbounded by $dn \log n$, for suitable constants $c$ and $d$.

**Assignment Problem 7.3.** (deadline: June 24, 5:30pm)
Let $X[1 \ldots n]$ and $Y[1 \ldots n]$ be two arrays, each containing $n$ numbers already in sorted order. Assume for convenience, that all $2n$ numbers are distinct. Give an $\mathcal{O}(\log n)$ algorithm to find to find the $i$-th smallest element of all $2n$ elements in arrays $X$ and $Y$.

Hint 1: binary search
Hint 2: there are $2i$ possibilities where the $i$-the smallest element can occur in $X$ and $Y$.

# Dynamic sets

- mathematical sets are unchanging
- sets in a program (algorithm) can grown and shrink over time, hence they are called **dynamic**
- **dictionary** is a dynamic set that supports operations *insert, delete and test membership* of an element

**elements** of a dynamic set are

- objects; each object contains a **key** and **satellite data** as object fields
- the **key** identifies the object, so we assume that keys are all different we can think of the dynamic set as being a set of key values
- **satellite data** – other object fields, unused by set implementation, but carried around

# Operations on dynamic sets

*modifying operations*:

- **Insert**$(S, x)$ — adds the element pointed to by $x$ to the set $S$
- **Delete**$(S, x)$ — given a pointer $x$ to an element in the set $S$, removes $x$ from $S$

*queries*:

- **Search**$(S, k)$ — returns a pointer to an element in $S$ with the key equal to $k$, or NIL if no such element belongs to $S$

*other queries*:

- **Minimum**$(S)$ — returns a pointer to the element of $S$ with the smallest key
- **Maximum**$(S)$ — returns a pointer to the element of $S$ with the largest key

- **Successor**$(S, x)$ (resp. **Predecessor**$(S, x)$) — given a pointer $x$ to an element in the set $S$, returns a pointer to the element in $S$ with the next larger (resp. smaller) value of key (or NIL, if $x$ points to the maximum element)

**Important:** we want to maintain **dynamic set** (insertions and deletions) and we want to search for elements (all as fast as possible)

We can assume that the keys are from a **universe** (set) $U$, where $U = \{0, \ldots, u-1\}$ for some (typically large) $u$

*Simple approaches:*

- unsorted array
- sorted array

*Simple fast approach*:
**Direct addressing**: element with key $k$ is stored in slot $k$

Array of size $|U|$, operations are straightforward

But what if $K$, set of keys **actually stored**, is much much smaller than $U$?

**Waste of memory**

Time-efficient, but waste of memory

Let's see something more clever...

# Hash tables

**Direct addressing** (each key has its own slot) performs very well (constant worst-case time for all operations), but requires a lot of memory

What we want is to reduce size of table

**Hashing:** an element with key $k$ is stored in slot $h(k)$: we use a hash function $h$ to compute slot

When only operations **Insert**, **Search** and **Delete** as dictionary operations are needed, **hash tables** can be quite good:
we can still perform operations in $\mathcal{O}(1)$ time in average;
however the worst-case time is bigger

There are many variations of hash tables (or rather the hash functions implementing them), from not-so-fast but simple to extremely fast but complicated

**universe** of keys $U = \{0, \ldots, u-1\}$

We want to reduce size of table to $m$,
where $m << |U|$

We will use a **hash function**

$$h : U \to \{0, \ldots, m-1\}$$

We say element with key $k$ **hashes** into slot $h(k)$, and that $h(k)$ is **hash value** of $k$

*Problem:* two or more keys may hash to same slot (**collisions**)

We need some strategy to deal with this problem

*Ideal solution:* avoid collisions altogether

However: by assumption $|U| > m$, so there **must** be at least two keys with same hash value, thus complete avoidance **impossible**

*Another solution:* make it **random** so that the number of collisions is at least "minimized"

but still, the hash function has to be *deterministic*: given an input key $k$ it should always produce the same output $h(k)$

Even with a random hash function, there will be collisions anyway:
**How to resolve collisions?**

*Simplest solution:* **chaining**
(another solution later: **open addressing**)

# Collision resolution by chaining

The simplest of all collision-resolution protocols

- each **slot** is a **linked list** ("chain")
- slot $j$ contains a pointer to the head of the list of all elements that hash to $j$
- if there are no such elements, slot $j$ contains NIL
- when elements collide, just insert the new element into the list

Let $T$ be a hash table and $h$ a hash function.

Implementation of dictionary operations:

**Chained-Hash-Insert(T,x)**

insert $x$ at the head of list $T[h(\text{key}[x])]$

**Chained-Hash-Search(T,k)**

search for an element with key $k$ in list $T[h(k)]$

**Chained-Hash-Delete(T,x)**

delete $x$ from the list $T[h(\text{key}[x])]$

What about the worst-case running times?

- for each operations we have to compute $h(k)$, in addition:
- **Insert:** clearly $O(1)$ under assumption that element is not yet in table; otherwise first perform a search
- **Search:** proportional to the length of the list; we will analyze this later...
- **Delete:**

  **Assume**: argument to procedure is a pointer $x$ to an element, not its key $k$

  – if the list is a doubly-linked list, then we need a constant time $\mathcal{O}(1)$
  – if the argument were a key, then a search is necessary
  – similarly, if the lists are singly-linked, we need to find predecessor of $x$, the same running time as for search

*Hence:* it's important to determine how long it take to search for an element when a key is given

Consider a hash table $T$ with $m$ slots that stores $n$ elements

Define **load factor** $\alpha = n/m$ (average list size)

Analysis in terms of $\alpha$ (not necessarily greater than one!)

*Clear:* the worst-case performance is very bad: if all $n$ keys hash to the same slot, then all elements will be stored in one linked list $\Longrightarrow$ time $\Theta(n)$ — we might as well have used just one linked list for all elements

**Average performance** depends on how well hash function $h$ (that we still don't know) distributes keys, on average

**Assignment Problem 7.4.** (deadline: June 24, 5:30pm)
Show that if $|U| > nm$, there is a subset of $U$ of size $n$ consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$. Recall that $m$ is the size of the hash table, i.e., there are $m$ slots.

We will try to construct a good hash function later, but for now we shall assume that:

- Any given element is **equally likely** to hash into any of the $m$ slots, independently of where other elements hash to.

This assumption is called **simple uniform hashing**

For $j \in \{0, \ldots, m-1\}$ let $n_j = \text{length}(T[j])$

Clearly, $n_0 + n_1 + \cdots + n_{m-1} = n$

Also, average value of $n_j$ is $E[n_j] = \alpha = n/m$
(recall: "equally likely…")

Another assumption (not necessarily true): the value hash function $h(k)$ for a key $k$ can be computed in $O(1)$ time

Thus, the time required to search for some element with key $k$ depends linearly on length $n_{h(k)}$ of list $T[h(k)]$

Consider the expected number of elements in $T[h(k)]$ that are examined to see if their keys are equal to $k$
— this equal to the expected time for the search for an element with key $k$

We shall consider 2 cases:
**unsuccessful** (no element in the table has key $k$), and
**successful** searches.

# Unsuccessful searches

**Theorem.** Under assumption of simple uniform hashing, if using collision resolution by chaining then an unsuccessful search takes expected time $\Theta(1+\alpha)$.

Recall: $\alpha = n/m = \#\text{elements}/\#\text{slots}$.

**Proof.**

- any key $k$ not already in table (recall: unsuccessful) is equally likely hashed to any of the $m$ slots (*simple uniform hashing*)
- expected time to search unsuccessfully for $k$ is the expected time to search to end of $T[h(k)]$
- $T[h(k)]$ has expected length $E[n_{h(k)}] = \alpha$
- thus the expected number of examined elements is $\alpha$
- add 1 for computation of $h(k)$

Recall: $\alpha$ could be very small, thus $\Theta(1+\alpha)$ cannot be reduced to $\Theta(\alpha)$.

# Successful searches

*Difference:* **not** all lists equally likely to be searched

Why? Simple, probability that a list is searched is **proportional to the number of elements it contains** (we assume that the element being searched is equally likely any of the $n$ elements stored in the table).

| | | | | | |
|---|---|---|---|---|---|
| 111 | 22222 | 44 | 5555 | 66666666666 | 7 |

**Theorem.** Under assumption of simple uniform hashing, if using collision resolution by chaining then a successful search takes expected time $\Theta(1+\alpha)$ with $\alpha = n/m$.

**Proof.**

- the number of elements examined is 1 plus the number of elements in the $x$'s list before $x$
- these were inserted **after** $x$ itself (new elements are placed at front)

- to find the expected number of elements examined, take the average of 1 plus the expected number of elements added to the $x$'s list after $x$ was added, over the $n$ elements $x$ in table

For $1 \leq i \leq n$, let $x_i$ be the $i$-th element inserted into the table, and let $k_i = \text{key}(x_i)$

For keys $k_i, k_j$, define Bernoulli random variables $X_{ij} = 1$ if $h(k_i) = h(k_j)$

**question:** how many elements do we examine until we find $x_i$?

- all elements in the same list as $x_i$ appearing before, that is, all elements $x_j$ such that $j > i$ and $X_{ij} = 1$
- element $x_i$

**answer:** $m_i = 1 + \sum_{j=i+1}^{n} X_{ij}$

Under assumption of simple uniform hashing, $P(X_{ij} = 1) = 1/m$

Indeed: for any $z \in \{1, \ldots, m\}$, we have
$P(h(k_i) = z) = P(h(k_j) = z) = 1/m$, thus

$$\begin{aligned} P(X_{ij} = 1) &= \sum_{z=1}^{m} P(h(k_i) = z) \cdot P(h(k_j) = z) \\ &= \sum_{z=1}^{m} (1/m)^2 = 1/m \end{aligned}$$

Thus $E[X_{ij}] = P(X_{ij} = 1) = 1/m$.

we don't know which element we are looking for

each of them has same probability $1/n$, hence expected number of examined elements in a successful search is the average:

$$
\begin{aligned}
E\left[\frac{1}{n}\sum_{i=1}^{n} m_i\right] &= E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} X_{ij}\right)\right] \\
&\stackrel{(LOE)}{=} \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} E[X_{ij}]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= \left(\frac{1}{n}\sum_{i=1}^{n} 1\right) + \left(\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1
\end{aligned}
$$

$$
\begin{aligned}
&= 1 + \frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1 + \frac{1}{nm}\left(\sum_{i=1}^{n} n - \sum_{i=1}^{n} i\right) \\
&= 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) \\
&= 1 + \frac{n}{m} - \frac{n+1}{2m} = 1 + \alpha - \frac{n+1}{2n}\alpha \\
&= 1 + \alpha\left(1 - \frac{n+1}{2n}\right) = \Theta(1+\alpha)
\end{aligned}
$$

Adding $\Theta(1)$ for computation of $h(k)$, we end up with

$$\Theta(2+\alpha) = \Theta(1+\alpha)$$

*Consequence:* if $m$ (# slots) is at least proportional to $n$ (# elements), then $n = O(m)$ and $\alpha = n/m = O(1)$, thus searching takes **constant time** on average!

**Insertion** and **Deletion** also take constant time (even in the worst-case) if doubly-linked lists are used, thus

**all operations take constant time on average!**

(*However*: we need assumption of single uniform hashing)

**Assignment Problem 7.5.** (deadline: June 24, 5:30pm)
Suppose we use a hash function $h$ to hash $n$ distinct keys into an array $T$ of length $m$. Assuming simple uniform hashing, what is the expected number of collisions, that is what is the expected number of elements of the set

$$\{\{k,l\} : \quad k \neq l \text{ and } h(k) = h(l)\}\,?$$

Hint: use random variables $X_{ij}$ define on the lecture.

# Hash functions

So far, haven't seen a single hash function

**What makes a good hash function?**

Satisfies (more or less) the assumption of single uniform hashing:

Each key is equally likely to hash to any of the $m$ slots, independently of where other keys hash to

However, typically **impossible**, certainly depending on how keys are chosen
— usually, we don't know the probability distribution according to which the keys are drawn, and the keys may not be drawn independently.

Sometimes we **know** the key distribution.

**Example:** keys are real random numbers in $k \in [0, 1)$, independently and uniformly chosen, then $h(k) = \lfloor k \cdot m \rfloor$ satisfies the assumption of uniform hashing

**Design:**

- heuristic (division and multiplication methods)
- randomization (universal hashing)

**Heuristic:**

*Example:* compiler;
we want to store identifiers in our hash table;
it's very likely that similar strings occur in the same program
("minelement", "minposition")
we should minimize the chance to hash them to the same slot

in general, we should design a hash function so that it's independent on any patterns in the data

another usually good property: map keys which are similar to slots far apart
(this will be very useful in open addressing method discussed later)

**Usual assumption:** universe of keys is $\{0, 1, 2, \ldots\}$, i.e., somehow interpret real keys as natural numbers ("usually" easy enough...)

Two very simple hash functions:

**1. Division method**: $h(k) = k \bmod m$

*Example:* hash table has size 25, key $k = 234$, then
$h(k) = 234 \bmod 25 = 9$

Quite fast, but drawbacks

Want to avoid certain values of $m$, e.g. powers of 2

Why? If $m = 2^p$, then $h(k) = k \bmod m = k \bmod 2^p$, the $p$ lowest-order bits of $k$

*Example:* $m = 2^5 = 32$, $k = 168$, $h(k) = 168 \bmod 32 = 8 = (1000)_2$, and $k = 168 = (10101\underline{000})_2$

Better to make hash function depend on **all** bits of key

Good idea (usually) for $m$: prime not too close to power of two

**Assignment Problem 7.6.** (deadline: June 24, 5:30pm)
Assume that the keys are strings with each character having $p$ bits.
Assume that we choose $m = 2^p - 1$ in the division method. Show that if string $x$ can be derived from string $y$ by permuting its characters, then $x$ and $y$ hash to the same value.

**2. Multiplication method**:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

explanation:

- $A$ is constant with $0 < A < 1$
- Thus $kA$ is real with $0 \leq kA < k$
- $kA \bmod 1$ is fractional part of $kA$, i.e., $kA - \lfloor kA \rfloor$
  *Example:* $A = 0.23$, $k = 234$, then $kA = 53.82$ and $kA \bmod 1 = 0.82$
- $kA \bmod 1 \in [0, 1)$
- Therefore $m(kA \bmod 1) \in [0, m)$, and $\lfloor m(kA \bmod 1) \rfloor \in [0, 1, \ldots, m-1]$

Advantage: value of $m$ **not** critical

Typically power of two (no good with division method!), since then the implementation is easy,

**Exercise 7.2.** Consider a hash table of size $m = 1000$ and a corresponding hash function

$$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$

for $A = (\sqrt{5} - 1)/2$. Compute the location to which the keys $61, 62, 63, 64$ and $65$ are mapped.