1 Linear Time Selection

If we would like to get the kth largest number in an array, sorting the whole array and output the kth number at least takes $O(n \log n)$ time complexity (in most of the cases). Hence we would like to find some more efficient way to get the kth largest number in $O(n)$.

1.1 Randomized selection algorithm

The concept of randomized selection algorithm is quite similar to **quick sort**. It first randomly picks a pivot, and divides the array into two parts: on the left are elements less than the pivot, and on the right are those greater than the pivot. By knowing the current position of pivot t, and suppose you want to find the kth element, then:

- if $t > k$, then repeat the steps above in the left part and look for the kth element
- if $t < k$, then repeat the steps above in the right part and look for the $(k - t)$th element

the pseudo code of randomized selection goes as below:

```
Rselect(int A[], int n, int i) {
  // find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Choose pivot p from A uniformly at random;
  Partition A using pivot p;
  Let j be the index of p;
  if (j == i) return p;
  if (j > i) return Rselect(1st part of A, j-1, i);
  else return Rselect(2nd part of A, n-j, i-j);
}
```

Time complexity of some steps is known:

- Partition needs $O(n)$
- Choosing pivot randomly needs $O(1)$

These can help you better understand the quiz question on P11, lecture slide 5.
As what we have mentioned in RC week2, when encountering with random concept in one algorithm, we cannot simply judge the best / average / worst case according to the input itself. This is actually against the original definition of best / average / worst case (it should be related to input), hence we would like you to think about the the best / worst **scenario** instead of **case** in homework.
The **expected** runtime of randomized selection algorithm is $O(n)$.

1.2 Deterministic selection algorithm

Deterministic selection algorithm keeps the time complexity to be $O(n)$ in all cases. It improves the randomized selection algorithm by choosing median of medians as pivots.
We will use master theorem to calculate the time complexity. You can practice it in your homework0. The course slides have already given you most of the steps.

2 Hash

Hash is really important!!!
Hash table is quite widely used as a data structure in our daily life, because of its high efficiency on searching, inserting and deleting an element. Hash table works like an array, but it uses the hash key as index instead of directly using integer as index. Here I will show you how hash table works differently from those commonly used data structures.
Suppose we have a structure that stores the following information:

```
struct Student {
  string name;
  int score;
}
```

And I want to know the student's score with his / her name.

With link list:
If we use a link list to store the information of students in VE281, then we need to find Roihn's score in the following steps:

1. Check whether the *Student* that the head pointer is pointing at has the name "Roihn", if yes, return his score;
2. if not, check whether the next *Student* matches the name, if yes, return his score;
3. if not, check whether the next *Student* matches the name, if yes, return his score;
4. ...

You need to recursively traverse every node from the head to the end until you find the student that matches the given name. It takes $O(n)$ to find the student on average.

With array:
If we use an array to store the information of students in VE281, as we all know, normally we will **NOT** use a string as index of an array. Hence we may store the students' information in sequence. Then we need to repeat the steps like what we have done with link list, using a for loop to traverse from index 0 to index $n-1$, until we find the student we want. It takes $O(n)$ to find the student on average.

If given extra information, such as the student id (an integer), we may have some other choices.
Suppose all the student ids are in the form of 518370910XXX, then we can store every student's information in the array with index = *id* - 518370910000. Hence whenever we want to get the score of one student, we can use the last three digits as index to directly go to the slot we want in an array.
This concept of converting the key (student id) to another form is actually the concept of **hash table**. Here we use the hash function $f(x) = x - 518370910000$ to produce a hash key $f(x)$ for each student id x, and get to the correct place with the hash key. Here we only need $O(1)$ time complexity since we directly calculate out the location of the value we want. And this is how hash table works.

2.1 Hash Collision

For the scenario above, now suppose we do not know the student id. Then we can only search for the student's score by his/her name.
Let's take the ascii code of the first letter of one's name as the hash key (This is only one choice of hash function, you can think about numerous form of hash functions). Then, for hash function $f(x)$, we have:

$$f('Alice') = 65,\ f('Roihn') = 82$$

This is because ascii code of "A" is 65, while the ascii code of "R" is 82. Then Roihn's score will be stored into the slot with index = 82.
However, if there is a student Ryan who also takes VE281, his score should also be stored at index = 82. Then a collision happens. We will mention how to deal with such collision in detail in next lecture.

2.2 Hash Function

Page 16 of lecture slides has shown the Hash Function Design Criteria:

- Must compute a bucket for every key in the universe.
- Must compute the same bucket for the same key.
- Should be easy and quick to compute.
- Minimizes collision
  - Spread keys out **evenly** in hash table
  - Gold standard: completely random hashing
  - The probability that a randomly selected key has bucket i as its home bucket is $1/n$, $0 \leq i < n$.
  - Completely random hashing **minimizes the likelihood** of an collision when keys are selected at random.
  - However, completely random hashing is infeasible due to the need to remember the random bucket.

One popular hash function is **hashing by modulo**. Hashing by modulo uses the following hash function:

$$f(x) = x$$

where n is the size of home bucket (the size of the array).
There are mainly **two things** you need to pay attention to:

1. n **should be odd.**

   For example, if you know that all of your original keys satisfy the format of $4k,\ k \in Z$, such as 0, 4, 8, 12... Then if you use an even number as n, for example if $n = 4$, then you will find that all of your keys will be filled into the slot with index = 0.

If you use $n = 6$, then the keys will only be filled into slots with index = 0 or 2 or 4, which has **biased distribution** (original keys distribute uniformly while their hash keys only fall in three specific slots).

Hence odd n is required.

2. **a big prime number is highly recommended for n.**

   As we all know, all the non-prime numbers can be written as the multiplication of several small prime numbers. For example:

$$15 = 3 \times 5$$
$$3060 = 2^2 \times 3^2 \times 5 \times 11$$

   Similar biased distribution of home buckets happens in practice when the hash table size n is a multiple of small prime numbers. In other words, non-prime n leads to biased distribution, which is not we want to see.

   Also, The effect of each prime divisor p of n **decreases** as p gets larger. In other words, the bigger prime p leads to more unbiased distribution.

   Hence we need n to be a big prime number.