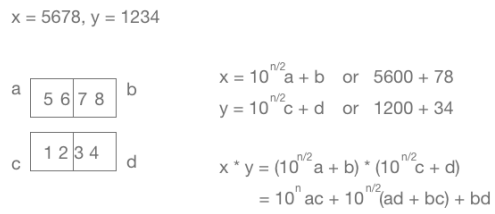
Part 1: Fast multiplication (20 pts)

1.1: Number multiplication (10 pts)

Let's dive into details of simply how numbers are multiplied. Let's assume that we deal with arbitrary non-negative integers stored in binary form (as a sequence of binary digits).

You should remember how to multiply two numbers from elementary school – you just sequentially multiply the first number by a single digit of the second one and shift the result to the left. In the end you sum all numbers calculated before. The exact process may differ but in fact you're always computing pairwise products of digits, multiplying them by corresponding power of base (10 in school, 2 in our case) and summing them up. This results in time complexity of $O(n^2)$. Let's call this grade-school multiplication algorithm

```
      23958233
×         5830
──────────────────
      00000000 (  =  23,958,233 ×     0)
     71874699  (  =  23,958,233 ×    30)
    191665864  (  =  23,958,233 ×   800)
+ 119791165    (  =  23,958,233 × 5,000)
──────────────────
  139676498390 ( = 139,676,498,390)
```

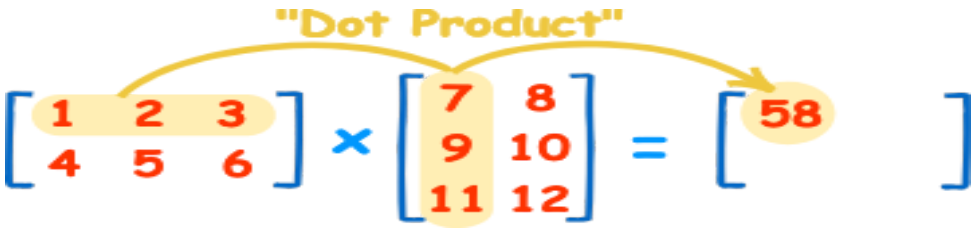


Standard algorithm isn't an optimal way to solve this problem. There are some algorithms that propose better time complexity. Karatsuba algorithm uses divide and conquer approach to multiply numbers faster. The high-level idea is to divide each number into 2 parts with equal lengths and split the problem into 3 not 4 subproblems with halved size. Karatsuba algorithm archives time complexity of $O(n^{log_2 3})$.

1. Implement grade-school multiplication algorithm **(1 pts)**
2. Implement Karatsuba multiplication algorithm **(2.5 pts)**
3. Derive theoretical time complexities
    a. Explain why is it $O(n^2)$ for grade-school algorithm **(0.5 pts)**
    b. Explain why is it $O(n^{log_2 3})$ for Karatsuba algorithm **(1.5 pts)**
4. Obtain actual execution times in experiments
    a. Plot execution time dependency on number length n **(2 pts)**
    b. Rescale y-axis to check if execution time measurements match theoretical complexity **(1.5 pts)**
5. Check if execution time depends on actual values of numbers **(1 pts)**

2.2: Matrix multiplication (10 pts)

Now since we're done with multiplication of simple numbers let's consider a bit more complicated case. Remind how one multiplies matrices: in product C = A * B the dot product of **i'th** row of matrix A and **j'th** column of matrix B composes the **ij'th** element of matrix C. If both A and B are square matrices with size $n \times n$ then calculating each dot product costs time $O(n)$. Finally, dot product is calculated n^2 times (for each row-column pair) so the resulting time complexity is $O(n^3)$. This is a trivial algorithm that mimics the definition of matrix multiplication.



As you have already suspected, this is not an optimal solution. In 1969 Volker Strassen proposed to reuse ideas of Karatsuba for matrix multiplication. Strassen's algorithm reuse the trick of reducing number of smaller subproblems to achieve subcubic complexity of $O(n^{\log_2 7})$

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{pmatrix}$$

Trick:

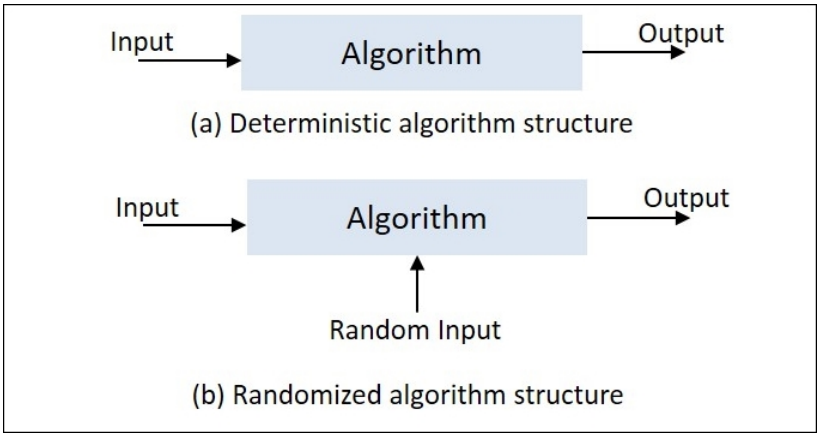
$P_1 = A \cdot (F-H)$
$P_2 = (A+B) \cdot H$
$P_3 = (C+D) \cdot E$
$P_4 = D \cdot (G-E)$
$P_5 = (A+D) \cdot (E+H)$
$P_6 = (B-D) \cdot (G+H)$
$P_7 = (A-C) \cdot (E+F)$

$AE+BG = P_5 + P_4 - P_2 + P_6$
$AF+BH = P_1 + P_2$
$CE+DG = P_3 + P_4$
$CF+DH = P_5 + P_1 - P_3 - P_7$

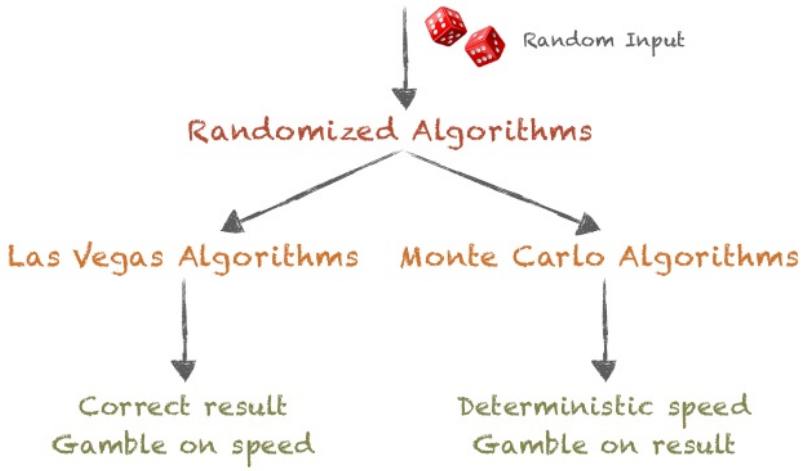
1. Implement trivial matrix multiplication algorithm **(1 pts)**
2. Implement matrix multiplication with Strassen's algorithm **(2.5 pts)**
3. Derive theoretical time complexities
    a. Explain why is it $O(n^3)$ for trivial algorithm **(0.5 pts)**
    b. Explain why is it $O(n^{\log_2 7})$ for Strassen's algorithm **(1.5 pts)**
4. Obtain actual execution times in experiments
    a. Plot execution time dependency on number length n **(2 pts)**
    b. Rescale y-axis to check if execution time measurements match theoretical complexity **(1.5 pts)**
5. Check if execution time depends on values inside matrices **(1 pts)**
6. **(Bonus)** Try different order of cycles in trivial algorithm and explain the difference **(3 pts)**

Part 2: Randomized algorithms (30 pts)

Adding some randomness inside an algorithm makes its execution and result non-deterministic. On the other hand this allows us to describe an algorithm's properties with probabilistic or statistical models. These types of algorithms are called randomized. Do not confuse them with complexity analysis by modeling input data probability distribution – randomized algorithms are non-deterministic for fixed input data.

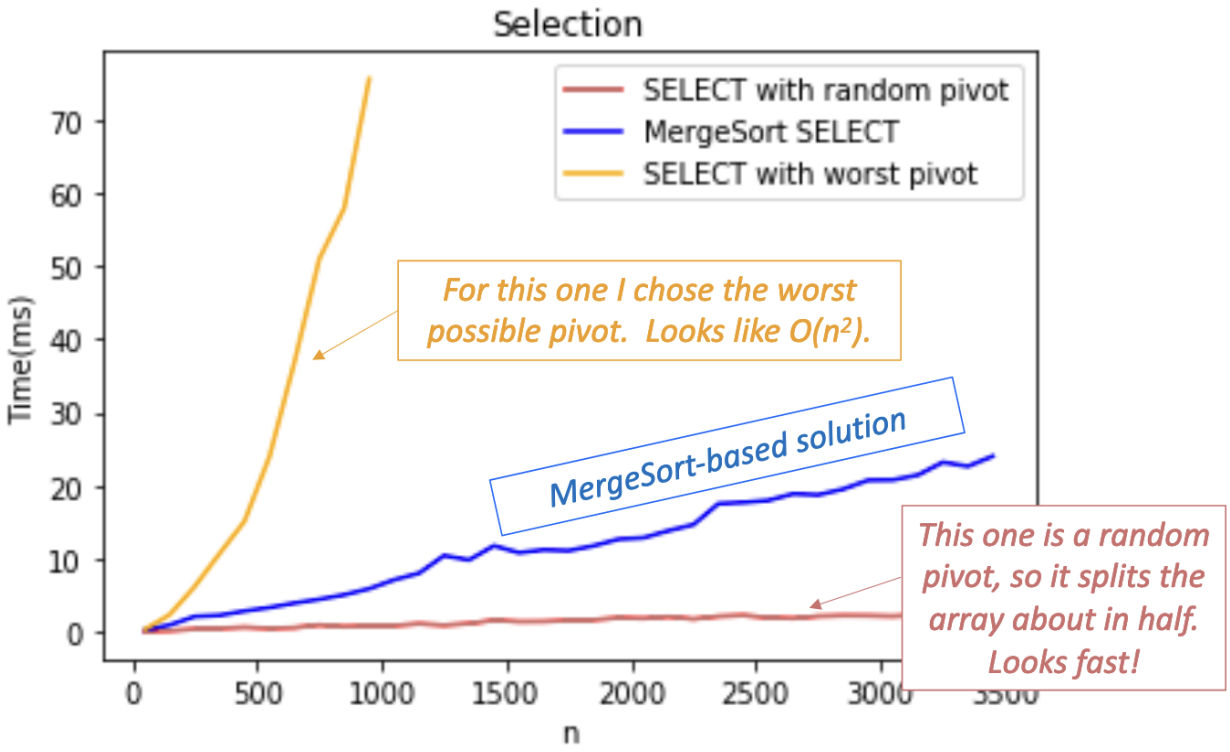


Typically, one decides between the resulting randomness in algorithm's output and execution time. **Las Vegas** algorithm always results in correct output but the execution time is non-deterministic and is described by a random variable. **The Monte Carlo** algorithm always has a deterministic sequence of performed steps and thus an execution time but may produce incorrect output.



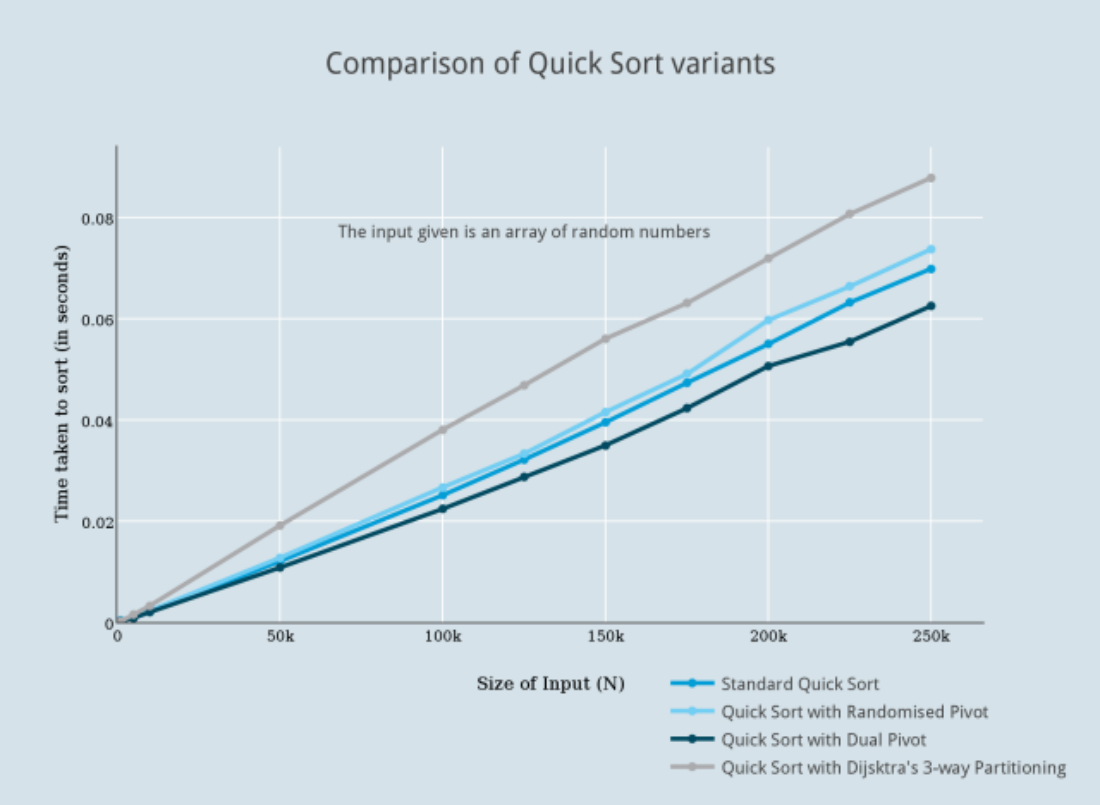
2.1 (Las Vegas): Median and K-smallest number (11 pts)

1. Design and implement the deterministic algorithm for finding:
    a. Maximum and minimum element of the array **(0.5 pts)**
    b. Median of the array (via quicksort paradigm) **(1 pts)**
    c. The k-smallest element of the array (via quicksort paradigm) **(1 pts)**
2. Design and implement the randomized algorithm by taking uniformly random pivot element instead of fixed one for finding:
    a. The median of the array **(0.5 pts)**
    b. The k-smallest element of the array **(0.5 pts)**
3. Derive the time complexity:
    a. For 1b,1c consider best – O(n) and worst – O(n^2) cases **(1 pts)**
    b. For 2a,2b derive the expectation using approach from lecture **(1.5 pts)**
    c. For 2a,2b derive the expectation by induction **(2 pts)**
4. Obtain actual execution times in experiments for the following inputs:
    a. For sorted array
    b. For inversely sorted array
    c. For randomly sorted array
5. Plot the dependency of execution time on problem size n **(3 pts)**
6. **(Optional)** Plot and analyze the distributions of execution times **(up to 3 pts)**



2.2 (Las Vegas): Quicksort (11 pts)

1. Implement quicksort by selecting pivot as:
    a. Fixed position in the array (e.g. first element or middle) **(0.5 pts)**
    b. Deterministic median of the array (see Part 1:1b) **(0.5 pts)**
    c. Randomized median of the array (see Part 1:2a) **(0.5 pts)**
    d. Uniformly random element of the array **(0.5 pts)**
2. Derive the time complexity:
    a. For 1a consider best – O(n log n) and worst – O(n^2) cases **(0.5 pts)**
    b. For 1b combine Part 1:3a and Master theorem **(0.5 pts)**
    c. For 1c combine Part 1:3b/3c and Master theorem **(0.5 pts)**
    d. For 1d derive the expectation using approach from lecture **(2 pts)**
    e. For 1d derive the expectation by induction **(2.5 pts)**
3. Obtain actual execution times in experiments for the following inputs:
    a. For sorted array
    b. For inversely sorted array
    c. For randomly sorted array
4. Plot the dependency of execution time on problem size n **(3 pts)**
5. **(Optional)** Plot and analyze the distributions of execution times **(up to 3 pts)**



2.3 (Monte Carlo): HyperLogLog (8 pts)

1. Get familiar with the [cardinality estimation problem and HLL algorithm solution]
2. Implement following algorithms for cardinality estimation:
    a. Trivial counting using hashmap **(0.5 pts)**
    b. Flajolet-Martin algorithm **(0.75 pts)**
    c. LogLog algorithm **(0.75 pts)**
    d. SuperLogLog algorithm **(0.75 pts)**
    e. HyperLogLog algorithm **(0.75 pts)**
3. Generate input data: **(1.5 pts)**
    a. Sample random numbers from Poisson distribution (lamba = 1000)
    b. Choose sample sizes N = 1k, 10k, 100k, 1M, 10M
4. Test your algorithms on generated data and analyze following features:
    a. Execution time **(1 pts)**
    b. Output error **(1 pts)**
    c. Space complexity (for hashmap) **(1 pts)**