**Researches about theory (R)**

8\_R.

Do a research about the following topics:  
  
- The law of large numbers LLN, the various definitions of convergence  
  
- The convergence of the Binomial to the normal and Poisson distributions  
  
- The central limit theorem [in anticipation of a topic we will study later]

“In probability theory, the law of large numbers (LLN) is a theorem that describes the result of performing the same experiment a large number of times. According to the law, the average of the results obtained from a large number of trials should be close to the expected value and will tend to become closer to the expected value as more trials are performed.” (Wikipedia)

**Applications / Practice (A)     [work on this at least 30' a day, all days]**

8\_A. Exercise (also partially described in video 04)  
  
Generate and represent m "sample paths" of n point each (m, n are program parameters), where each point represents a pair of:

time index t, and **relative frequency** of success f(t),  
  
where f(t) is the sum of t Bernoulli random variables with distribution B(x, p) = p^x(1-p)^(1-x) observed at the various times up to t: j=1, ..., t..

At time n (last time) and one other chosen inner time 1<j<n (where j is a user parameter) represent with a histogram the distribution of f(t).

See also what happens if you replace the relative frequency f(t) with the **absolute frequency** n(t) or by **normalized relative frequency**: f(t) / sqrt(p(1-p)/n).  
  
Comment briefly on the result.

(The general scheme of this exercise, will also be "reused" in next homeworks where we will consider other more interesting stochastic processes.)

It’s clear that if we set N to be low (in my application you can insert at least 20 as input for N and there in no maximum), for example n = 20 or n = 30 we have lower probability that the results (Bernoulli trials) will fall around the expected probability P (the one we set at the starting point before running the chart plotting). Instead, if try with high values of N (like n = 500 or even n = 2000 or 3000) we have way higher probability of having the results really close to the expected final value. Moreover, we can see (still with high trials) that this LLN is reflected into the histograms that we produce at the middle and at the end of the chart. Closer we are to the expected value and higher will be the bar, if we move from the expected value, we will see lower bars (similar to the Gauss chart).

**Researches about applications (RA)**

6\_RA. Do a web research about the various methods proposed to compute the running median (one pass, online algorithms).  
Store (cite all sources and attributions) the algorithm(s) that you think is(are) a good candidate, explaining briefly how it works and possibly try a quick demo.

|  |
| --- |
| from heapq import heappush,heappop, heapify,\_heapify\_max |
|  |  |
|  | arr = [11, 8, 9, 6, 20, 5, 7, 14, 12, 3] |
|  |  |
|  | # `low` is a max-heap & `high` is a min-heap |
|  | low = [] |
|  | high = [] |
|  |  |
|  | # Init median to zero |
|  | median = 0 |
|  |  |
|  | for i in arr: |
|  | # array value is less than median, add to median |
|  | # add the value to max-heap & heapify |
|  | # else add to min-heap & heapify |
|  | if i < median: |
|  | # as we don't have inbuilt function called \_heappush\_max |
|  | # so use heappush [O(log N) per push] and call \_heapify\_max [O(N)] |
|  | heappush(low, i) |
|  | \_heapify\_max(low) |
|  | else: |
|  | heappush(high, i) |
|  |  |
|  | # The size of height difference between heaps |
|  | # is more than 1, then move the element & heapify both |
|  | if len(low) > len(high)+1: |
|  | heappush(high, heappop(low)) |
|  | \_heapify\_max(low) |
|  | elif len(high) > len(low) + 1: |
|  | heappush(low, heappop(high)) |
|  | \_heapify\_max(low) |
|  |  |
|  | # Calculate Median: |
|  | # If lengths equal then median is average of both heaps |
|  | # Else median is larger heap element |
|  | if len(low) == len(high): |
|  | median = float(low[0] + high[0])/2.0 |
|  | else: |
|  | median = float(low[0]) if len(low) > len(high) else float(high[0]) |
|  |  |
|  | print("Median: " , median) |
|  |  |
|  |  |
|  | # Time Complexity = O(log N) + O(1) ≈ O(log N). |
|  | # Space complexity = O(N) linear space to hold input in heap containers. |

It’s interesting also the one posted here <https://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/>

**References**

<https://en.wikipedia.org/wiki/Law_of_large_numbers>

<https://medium.com/mind-boggling-algorithms/streaming-algorithms-running-median-of-an-array-using-two-heaps-cd1b61b3c034>

<https://gist.githubusercontent.com/anilpai/48d058d6a1e982d57ab55700c2e440c0/raw/7a79ae3f2952d84552d5c9646f09e351dfaaa8db/running_median.py>

<https://www.geeksforgeeks.org/median-of-stream-of-integers-running-integers/>