3 Random Sampling

"So much of life, it seems to me, is determined by pure randomness."
Sidney Poitier

3.1 Disk model and known sequence length 3-1
3.2 Streaming model and known sequence length 3-5
3.3 Streaming model and unknown sequence length .. 3-6

This lecture attacks a simple-to-state problem which is the backbone of many randomized algorithms, and admits solutions which are algorithmically challenging to design and analyze.

Problem. *Given a sequence of items $S = (i_1, i_2, \ldots, i_n)$ and a positive integer $m \leq n$, the goal is to select a subset of m items from S uniformly at random.*

Uniformity here means that any item in S has to be sampled with probability $1/n$. Items can be numbers, strings or general objects either stored in a file located on disk or streaming through a channel. In the former scenario, the input size n is known and items occupy n/B pages, in the latter scenario, the input size may be even unknown yet the *uniformity* of the sampling process must be guaranteed. In this lecture we will address both scenarios aiming at efficiency in terms of I/Os, extra-space required for the computation (in addition to the input), and amount of randomness deployed (expressed as number of randomly generated integers). Hereafter, we will make use of a built-in procedure `Rand(a,b)` that randomly selects a number within the range $[a, b]$. The number, being either real or integer, will be clear from the context. The design of a good `Rand`-function is a challenging task, however we will not go into its details because we wish to concentrate in this lecture on the sampling process rather than on the generation of random numbers; though, the interested reader can refer to the wide literature about (pseudo-)random number generators.

Finally we notice that it is desirable to have the positions of the sampled items in *sorted* order because this speeds up their extraction from S both in the disk setting (less seek time) and in the stream-based setting (less passes over the data). Moreover it reduces the working space because it allows to extract the items efficiently via a scan, rather than using an auxiliary array of pointers to items. We do not want to detail further the sorting issue here, which gets complicated whenever $m > M$ and thus these positions cannot fit into internal memory. In this case we need a disk-based sorter, which is indeed an advanced topic of a subsequent lecture. If instead $m \leq M$ we could deploy the fact that positions are integers in a fixed range and thus use radix sort or any other faster routine available in the literature.

3.1 Disk model and known sequence length

We start by assuming that the input size n is known and that $S[1, n]$ is stored in a file on disk which cannot be modified because it may be the input of a more complicated problem that includes the current one as a sub-task. The first algorithm we propose is very simple, and allows us to arise some issues that will be attacked in the subsequent solutions.

Algorithm 3.1 Drawing from all un-sampled positions

```
1: Initialize the auxiliary array $S'[1, n] = S[1, n]$;
2: for $s = 0, 1, \ldots, m-1$ do
3:     $p = \texttt{Rand}(1, n-s)$;
4:     select the item (pointed by) $S'[p]$;
5:     swap $S'[p]$ with $S'[n-s]$.
6: end for
```

At each step s the algorithm maintains the following invariant: *the subarray $S'[n-s+1, n]$ contains the items that have been already sampled, the rest of the items of S are contained in $S'[1, n-s]$*. Initially (i.e. $s = 0$) this invariant holds because $S'[n-s+1, n] = S'[n+1, n]$ is empty. At a generic step s, the algorithm selects randomly one item from $S'[1, n-s]$, and replaces it with the last item of that sequence (namely, $S'[n-s]$). This preserves the invariant for $s+1$. At the end (i.e. $s = m$), the sampled items are contained in $S'[n-m+1, n]$. We point out that S' cannot be a *pure* copy of S but it must be implemented as an *array of pointers* to S's items. The reason is that these items may have variable length (e.g. strings) so their retrieval in constant time could not be obtained via arithmetic operations, as well as the replacement step might be impossible due to difference in length between the item at $S'[p]$ and the item at $S'[n-s]$. Pointers avoid these issues but occupy $\Theta(n \log n)$ bits of space, which might be a non negligible space when n gets large and might turn out even larger than S if the average length of S's objects is shorter than $\log n$.[1] Another drawback of this approach is given by its pattern of memory accesses, which acts over $O(n)$ cells in purely random way, taking $\Theta(m)$ I/Os. This may be slow when $m \approx n$, so in this case we would like to obtain $O(n/B)$ I/Os which is the cost of scanning the whole S.

Let us attack these issues by proposing a series of algorithms that incrementally improve the I/Os and the space resources of the previous solution, up to the final result that will achieve $O(m)$ extra space, $O(m)$ average time and $O(\min\{m, n/B\})$ I/Os. We start by observing that the swap of the items in Step 5 of `Algorithm 3.1` guarantees that every step generates one distinct item, but forces to duplicate S and need $\Omega(m)$ I/Os whichever is the value of m. The following `Algorithm 3.2` improves the I/O- and space-complexities by avoiding the item-swapping via the use of an auxiliary data structure that keeps track of the selected positions in sorted order and needs only $O(m)$ space.

Algorithm 3.2 Dictionary of sampled positions

```
1: Initialize the dictionary $\mathcal{D} = \emptyset$;
2: while $(|\mathcal{D}| < m)$ do
3:     $p = \texttt{Rand}(1, n)$;
4:     if $p \notin \mathcal{D}$ insert it;
5: end while
```

[1]This may occur only if S contains duplicate items, otherwise a classic combinatorial argument applies.

`Algorithm 3.2` stops when $\mathcal{D}$ contains m (distinct) integers which constitute the positions of the items to be sampled. According to our observation made at the beginning of the lecture, $\mathcal{D}$ may be sorted before S is accessed on disk to reduce the seek time. In any case, the efficiency of the algorithm mainly depends on the implementation of the dictionary $\mathcal{D}$, which allows to detect the presence of duplicate items. The literature offers many data structures that efficiently support membership and insert operations, based either on hashing or on trees. Here we consider only an hash-based solution which consists of implementing $\mathcal{D}$ via a hash table of size $\Theta(m)$ with collisions managed via chaining and a universal hash function for table access [1]. This way each membership query and insertion operation over $\mathcal{D}$ takes $O(1)$ time on average (the load factor of this table is $O(1)$), and total space $O(m)$. Time complexity could be forced to be worst case by using more sophisticated data structures, such as dynamic perfect hashing, but the final time bounds would always be *in expectation* because of the underlying re-sampling process.

However this algorithm may generate *duplicate* positions, which must be discarded and *re-sampled*. Controlling the cost of the re-sampling process is the main drawback of this approach, but this induces a constant-factor slowdown on average, thus making this solution much appealing in practice. In fact, the probability of having extracted an item already present in $\mathcal{D}$ is $|\mathcal{D}|/n \leq m/n < 1/2$ because, without loss of generality, we can assume that $m < n/2$ otherwise we can solve the *complement* of the current problem and thus randomly select the positions of the items that are *not* sampled from S. So we need an average of $O(1)$ re-samplings in order to obtain a new item for $\mathcal{D}$, and thus advancing in our selection process. Overall we have proved the following:

FACT 3.1 `Algorithm 3.2` *based on hashing with chaining requires $O(m)$ average time and takes $O(m)$ additional space to select uniformly at random m positions in $[1, n]$. The average depends both on the use of hashing and the cost of re-sampling. An additional sorting-cost is needed if we wish to extract the sampled items of S in a streaming-like fashion. In this case the overall sampling process takes $O(\min\{m, n/B\})$ I/Os.*

If we substitute hashing with a (balanced) search tree and assume to work in the RAM model (hence we assume $m < M$), then we can avoid the sorting step by performing an *in-visit* of the search tree in $O(m)$ time. However, `Algorithm 3.2` would still require $O(m \log m)$ time because each insertion/membership operation would take $O(\log m)$ time. We could do better by deploying an *integer*-based dictionary data structure, such as a van Emde-Boas tree, and thus take $O(\log \log n)$ time for each dictionary operation. The two bounds would be incomparable, depending on the relative magnitudes of m and n. Many other trade-offs are possible by changing the underlying dictionary data structure, so we leave to the reader this exercise.

The next step is to avoid a dictionary data structure and use *sorting* as a basic block of our solution. This could be particularly useful in practice because comparison-based sorters, such as `qsort`, are built-in in many programming languages. The following analysis will have also another side-effect which consists of providing a more clean evaluation of the average time performance of `Algorithm 3.2`, rather than just saying re-sample each item at most $O(1)$ times on average.

The cost of `Algorithm 3.3` depends on the number of times the sorting step is repeated and thus do exist duplicates in the sampled items. We argue that a small number of re-samplings is needed. So let us compute the probability that `Algorithm 3.3` executes just one iteration: this means that the m sampled items are all distinct. This analysis is well known in the literature and goes under the name of *birthday problem*: how many people do we need in a room in order to have a probability larger than $1/2$ that at least two of them have the same birthday. In our context we have that people = items and birthday = position in S. By mimicking the analysis done for the birthday problem, we

Algorithm 3.3 Sorting

```
1: $\mathcal{D} = \emptyset$;
2: while ($|\mathcal{D}| < m$) do
3:     $\mathcal{X}$ = randomly draw $m$ positions from $[1, n]$;
4:     Sort $\mathcal{X}$ and eliminate the duplicates;
5:     Set $\mathcal{D}$ as the resulting $\mathcal{X}$;
6: end while
```

can estimate the probability that a duplicate among m randomly-drawn items does not occur as:

$$\frac{m! \binom{n}{m}}{n^m} = \frac{n(n-1)(n-2)\cdots(n-m+1)}{n^m} = 1 \times (1 - \frac{1}{n}) \times (1 - \frac{2}{n}) \times \cdots (1 - \frac{m-1}{n})$$

Given that $e^x \geq 1 + x$, we can upper bound the above formula as:

$$e^0 \times e^{-1/n} \times e^{-2/n} \times \cdots e^{-(m-1)/n} = e^{-(1+2+\cdots+(m-1))/n} = e^{-m(m-1)/2n}$$

So the probability that a duplicate *does not* occur is at most $e^{-m(m-1)/2n}$ and, in the case of the birthday paradox in which $n = 365$, this is slightly smaller than one-half already for $m = 23$. In general we have that $m = \sqrt{n}$ elements suffices to make the probability of a duplicate at least $1 - \frac{1}{\sqrt{e}} \approx 0.4$, thus making our algorithm need re-sampling. On the positive side, we notice that if $m \ll \sqrt{n}$ then e^x can be well approximated with $1 + x$, so $e^{-m(m-1)/2n}$ is not only an upper-bound but also a reasonable estimate of the collision probability and it could be used to estimate the number of re-samplings needed to complete `Algorithm 3.3`.

FACT 3.2 `Algorithm 3.3` *requires a constant number of sorting steps on average, and $O(m)$ additional space, to select uniformly at random m items from the sequence $S[1, n]$. This is $O(m \log m)$ average time and* $\min\{m, n/B\}$ *worst-case I/Os if $m \leq M$ is small enough to keep the sampled positions in internal memory. Otherwise an external-memory sorter is needed. The average depends on the re-sampling, integers are returned in sorted order for streaming-like access to the sequence S.*

Sorting could be speeded up by deploying the specialty of our problem, namely, that items to be sorted are m random integers in a fixed range $[1, n]$. Hence we could use either radix-sort or, even better for its simplicity, bucket sort. In the latter case, we can use an array of m slots each identifying a range of n/m positions in $[1, n]$. If item i_j is randomly selected, then it is inserted in slot $\lceil jm/n \rceil$. Since the m items are randomly sampled, each slot will contain $O(1)$ items on average, so that we can sort them in constant time per bucket by using insertion sort or the built-in `qsort`.

FACT 3.3 `Algorithm 3.3` *based on bucket-sort requires $O(m)$ average time and $O(m)$ additional space, whenever $m \leq M$. The average depends on the re-sampling. Integers are returned in sorted order for streaming-like access to the sequence S.*

We conclude this section by noticing that all the proposed algorithms, except `Algorithm 3.1`, generate the set of sampled positions using $O(m)$ space. If $m \leq M$ the random generation can occur within main memory without incurring in any I/Os. Sometimes this is useful because the randomized algorithm that invokes the random-sampling subroutine does not need the corresponding items, but rather their positions.

3.2 Streaming model and known sequence length

We next turn to the case in which S is flowing through a channel and the input size n is known and big (e.g. Internet traffic or query logs). We will turn to the more general case in which n is unknown at the end of the lecture, in the next section. This stream-based model imposes that no preprocessing is possible (as instead done above where items' positions were re-sampled and/or sorted), every item of S is considered once and the algorithm must immediately and irrevocably take a decision whether or not that item must be included or not in the set of sampled items. Possibly future items may kick out that one from the sampled set, but no item can be re-considered again in the future. Even in this case the algorithms are simple in their design but their probabilistic analysis is a little bit more involved than before. The algorithms of the previous section offer an *average* time complexity because they are faced with the re-sampling problem: possibly some samples have to be eliminated because duplicated. In order to avoid re-sampling, we need to ensure that each item is not considered more than once. So the algorithms that follow implement this idea in the simplest possible way, namely, they scan the input sequence S and consider each item once for all. This approach brings with itself two main difficulties which are related with the guarantee of both conditions: uniform sample from the range $[1, n]$ and sample of size m.

We start by designing an algorithm that draws just one item from S (hence $m = 1$), and then we generalize it to the case of a subset of $m > 1$ items. This algorithm proceeds by selecting the item $S[j]$ with probability $\mathcal{P}(j)$ which is properly defined in order to guarantee both two properties above.[2] In particular we set $\mathcal{P}(1) = 1/n, \mathcal{P}(2) = 1/(n-1), \mathcal{P}(3) = 1/(n-2)$ etc. etc., so the algorithm selects the item j with probability $\mathcal{P}(j) = \frac{1}{n-j+1}$, and if this occurs it stops. Eventually item $S[n]$ is selected because its drawing probability is $\mathcal{P}(n) = 1$. So the proposed algorithm guarantees the condition on the sample size $m = 1$, but more subtle is to prove that the probability of sampling $S[j]$ is $1/n$, independently of j, given that we defined $\mathcal{P}(j) = 1/(n - j + 1)$. The reason derives from a simple probabilistic argument because $n - j + 1$ is the number of remaining elements in the sequence and all of them have to be drawn uniformly at random. By induction, the first $j - 1$ items of the sequence have uniform probability $1/n$ to be sampled; so it is $1 - \frac{j-1}{n}$ the probability of not selecting anyone of them. As a result,

$$\mathcal{P}(\text{Sample } i_j) = \mathcal{P}(\text{Not sampling } i_1 \cdots i_{j-1}) \times \mathcal{P}(\text{Picking } i_j) = (1 - \frac{j-1}{n}) \times \frac{1}{n-j+1} = 1/n$$

Algorithm 3.4 Scanning and selecting

```
1: $s = 0$;
2: for ($j = 1$; $j \leq n$; $j$++) do
3:     $p = \mathrm{Rand}(0, 1)$;
4:     if ($p \leq \frac{m-s}{n-j+1}$) then
5:         select $S[j]$;
6:         $s$++;
7:     end if
8: end for
```

[2]In order to draw an item with probability p, it suffices to draw a random real $r \in [0, 1]$ and then compare it against p. If $r \leq p$ then the item is selected, otherwise it is not.

`Algorithm 3.4` works for an arbitrarily large sample $m \geq 1$. The difference with the previous algorithm lies in the probability of sampling $S[j]$ which is now set to $\mathcal{P}(j) = \frac{m-s}{n-j+1}$ where s is the number of items already selected before $S[j]$. Notice that if we already got all samples, it is $s = m$ and thus $\mathcal{P}(j) = 0$, which correctly means that `Algorithm 3.4` does not generate more than m samples. On the other hand, it is easy to convince ourselves that `Algorithm 3.4` cannot generate less than m items, say y, given that the last $m - y$ items of S would have probability 1 to be selected and thus they would be surely included in the final sample (according to Step 4). As far as the uniformity of the sample is concerned, we show that $\mathcal{P}(j)$ equals the probability that $S[j]$ is included in a random sample of size m given that s samples lie within $S[1, j-1]$. We can rephrase this as the probability that $S[j]$ is included in a random sample of size $m - s$ taken from $S[j, n]$, and thus from $n - j + 1$ items. This probability is obtained by counting how many such combinations include $S[j]$, i.e. $\binom{n-j}{m-s-1}$, and dividing by the number of all combinations that include or not $S[j]$, i.e. $\binom{n-j+1}{m-s}$. Substituting to $\binom{b}{a} = \frac{b!}{a!\,(b-a)!}$ we get the formula for $\mathcal{P}(j)$.

FACT 3.4 `Algorithm 3.4` *takes $O(n/B)$ I/Os, $O(n)$ time, n random samples, and $O(m)$ additional space to sample uniformly m items from the sequence $S[1, n]$ in a streaming-like way.*

We conclude this section by pointing out a sophisticated solution proposed by Jeff Vitter [2] that reduces the amount of randomly-generated numbers from n to m, and thus speeds up the solution to $O(m)$ time and I/Os. This solution could be also fit into the framework of the previous section (random access to input data), and in that case its specialty would be the avoidance of re-sampling. Its key idea is not to generate random *indicators*, which specify whether or not an item $S[j]$ has to be selected, but rather generate random *jumps* that count the number of items to skip over before selecting the next item of S. Vitter introduces a random variable $G(v, V)$ where v is the number of items remaining to be selected, and V is the total number of items left to be examined in S. According to our previous notation, we have that $v = m - s$ and $V = n - j + 1$. The item $S[G(v, V) + 1]$ is the next one selected to form the uniform sample. It goes without saying that this approach avoids the generation of duplicate samples, but yet it incurs in an average bound because of the cost of generating the jumps according to the following distribution:

$$\mathcal{P}(G = g) = \binom{V - g - 1}{v - 1} \Big/ \binom{V}{v}$$

In fact the key problem here is that we cannot tabulate (and store) the values of all binomial coefficients in advance, because this would need space exponential in $V = \Theta(n)$. Surprisingly Vitter solved this problem in $O(1)$ average time, by adapting in an elegant way the von Neumann's rejection-acceptance method to the discrete case induced by G's jumps. We refer the reader to [2] for further details.

3.3 Streaming model and unknown sequence length

It goes without saying that the knowledge of n was crucial to compute $\mathcal{P}(j)$ in `Algorithm 3.4`. If n is unknown we need to proceed differently, and indeed the rest of this lecture is dedicated to detail two possible approaches.

The first one is pretty much simple and deploys a min-heap $\mathcal{H}$ of size m plus a real-number random generator, say `Rand(0,1)`. The key idea underlying this algorithm is to associate a random key to each item of S and then use the heap $\mathcal{H}$ to select the items corresponding to the top-m keys. The pseudo-code below implements this idea, we notice that $\mathcal{H}$ is a min-heap so it takes $O(1)$ time

Algorithm 3.5 Heap and random keys

```
1: Initialize the heap $\mathcal{H}$ with $m$ dummy pairs $\langle -\infty, \emptyset \rangle$;
2: for each item $S[j]$ do
3:     $r_j$ = Rand(0, 1);
4:     $m$ = the minimum key in $\mathcal{H}$;
5:     if ($r_j > m$) then
6:         extract the minimum key;
7:         insert $\langle r_j, S[j] \rangle$ in $\mathcal{H}$;
8:     end if
9: end for
10: return $\mathcal{H}$
```

to detect the minimum key among the current top-m ones. This is the key compared with r_j in order to establish whether or not $S[j]$ must enter the top-m set.

Since the heap has size m, the final sample will consists of m items. Each item takes $O(\log m)$ time to be inserted in the heap. So we have proved the following:

FACT 3.5 `Algorithm 3.5` *takes $O(n/B)$ I/Os, $O(n \log m)$ time, generates n random numbers, and uses $O(m)$ additional space to sample uniformly at random m items from the sequence $S[1, n]$ in a streaming-like way and without the knowledge of n.*

We conclude the lecture by introducing the elegant *reservoir sampling* algorithm, designed by Knuth in 1997, which improves `Algorithm 3.5` both in time and space complexity. The idea is similar to the one adopted for `Algorithm 3.4` and consists of properly defining the probability with which an item is selected. The key issue here is that we cannot take an irrevocable decision on $S[j]$ because we do not know how long the sequence S is, so we need some freedom to *change* what we have decided so far as the scanning of S goes on.

Algorithm 3.6 Reservoir sampling

```
1: Initialize array $R[1, m] = S[1, m]$;
2: for each next item $S[j]$ do
3:     $h$ = Rand(1, $j$);
4:     if $h \leq m$ then
5:         set $R[h] = S[j]$;
6:     end if
7: end for
8: return array $R$;
```

The pseudo-code of `Algorithm 3.6` uses a "reservoir" array $R[1, m]$ to keep the candidate samples. Initially R is set to contain the first m items of the input sequence. At any subsequent step j, the algorithm makes a choice whether $S[j]$ has to be included or not in the current sample. This choice occurs with probability $\mathcal{P}(j) = m/j$, in which case some previously selected item has to be kicked out from R. This item is chosen at random, hence with probability $1/m$. This double-choice is implemented in `Algorithm 3.6` by choosing an integer h in the range $[1, j]$, and making the substitution only if $h \leq m$. This event has probability m/j: exactly what we wished to set for $\mathcal{P}(j)$.

For the correctness, it is clear that `Algorithm 3.6` selects m items, it is less clear that these items are drawn uniformly at random from S, which actually means with probability m/n. Let's see why by assuming inductively that this property holds for a sequence of length $n-1$. The base case in which $n = m$ is obvious, every item has to be selected with probability $m/n = 1$, and indeed this is what Step 1 does by selecting all $S[1, m]$ items. To prove the inductive step (from $n-1$ to n items), we notice that the uniform-sampling property holds for $S[n]$ since by definition that item is inserted in R with probability $\mathcal{P}(n) = m/n$ (Step 4). Computing the probability of being sampled for the other items in $S[1, n-1]$ is more difficult to see. An item belongs to the reservoir R at the n-th step of `Algorithm 3.6` iff it was in the reservoir at the $(n-1)$-th step and it is not kicked out at the n-th step. This latter may occur either if $S[n]$ is not picked (and thus R is untouched) or if $S[n]$ is picked and $S[j]$ is not kicked out from R (being these two events independent of each other). In formulas,

$$\begin{aligned}\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) &= \mathcal{P}(S[j] \in R \text{ after } n-1 \text{ items}) \times [\mathcal{P}(S[n] \text{ is not picked}) \\ &\quad + \mathcal{P}(S[n] \text{ is picked}) \times \mathcal{P}(S[j] \text{ is not removed from } R)]\end{aligned}$$

Now, each of these items has probability $m/(n-1)$ of being in the reservoir R, by the inductive hypothesis, before that $S[n]$ is processed. Item $S[j]$ remains in the reservoir if either $S[n]$ is not picked (which occurs with probability $1 - \frac{m}{n}$) or if it is not kicked out by the picked $S[n]$ (which occurs with probability $\frac{m-1}{m}$). Summing up these terms we get

$$\mathcal{P}(\text{item } S[j] \in R \text{ after } n \text{ items}) = \frac{m}{n-1} \times [(1 - \frac{m}{n}) + (\frac{m}{n} \times \frac{m-1}{m})] = \frac{m}{n-1} \times \frac{n-1}{n} = \frac{m}{n}$$

To understand this formula assume that we have a reservoir of 1000 items, so the first 1000 items of S are inserted in R by Step 1. Then the item 1001 is inserted in the reservoir with probability 1000/1001, the item 1002 with probability 1000/1002, and so on. Each time an item is inserted in the reservoir, a random element is kicked out from it, hence with probability 1/1000. After n steps the reservoir R contains 1000 items, each sampled from S with probability $1000/n$.

FACT 3.6 `Algorithm 3.6` *takes $O(n/B)$ I/Os, $O(n)$ time, n random numbers, and exactly m additional space, to sample uniformly at random m items from the sequence $S[1, n]$ in a streaming-like way and without the knowledge of n. Hence it is time, space and I/O-optimal in this model of computation.*

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. Chapter 11: "Hashing", The MIT press, third edition, 2009.

[2] Jeffrey Scott Vitter. Faster methods for random sampling. *ACM Computing Surveys*, 27(7):703–718, 1984.