Topics in Theoretical Computer Science: Randomized Algorithms
EPFL, Fall semester, 2025

Homework 0 Solutions

Barras Simon <simon.barras@epfl.ch>
Doll Zachary <zachary.doll@epfl.ch>

---

1 Probability Problem 1

You want to sample uniformly at random from the set of all n-bit strings that are *balanced*, i.e., that contain exactly $n/2$-many 0's and $n/2$-many 1's (assume n is even). You do the following: Sample a uniform random string ω from the set of all n-bit strings, and output ω (and stop) if ω is balanced, else reject and sample again.

Show that the expected number of n-bit strings you sample is $O(\sqrt{n})$. (Hint: You may use Stirling's approximation.)

1.1 Solution

Proof. Let $\mathcal{S} = \{0,1\}^n$ be the set of bit strings of length n, where $n = 2k$ for $k \in \mathbb{Z}_+$. A bit string is said to be balanced if it contains as many 1s as 0s. More formally, a bit string of even length n is balanced if

$$\sum_{i=1}^{n} b_i = \frac{n}{2}$$

where b_i is the ith bit in the string. It follows that the number r of balanced strings in $\mathcal{S}$ is exactly

$$r = \binom{n}{n/2} = \frac{n!}{(n/2)!(n/2)!} = \frac{n!}{[(n/2)!]^2}$$

And thus, the probability p that a string picked uniformly at random from $\mathcal{S}$ is balanced is:

$$p = \frac{r}{|\mathcal{S}|} = \frac{\binom{n}{n/2}}{2^n}$$

Approximating r using Stirling's approximation, we get:

$$r = \frac{n!}{[(n/2)!]^2} \sim \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\pi n\left(\frac{n}{2e}\right)^n} = \frac{\sqrt{2\pi n}}{\pi n}2^n = a\frac{2^n}{\sqrt{n}}$$

where we defined the constant $a = \sqrt{\frac{2}{\pi}}$. Plugging this result back into our probability p, we finally obtain:

$$p = \frac{r}{2^n} \sim a\frac{2^n}{\sqrt{n}}\frac{1}{2^n} = \frac{a}{\sqrt{n}}$$

Let $X \sim Geom(p)$ be the number of independent samples from $\mathcal{S}$ (with replacement) until the first balanced string is observed. It's expectation is given by

$$\mathbb{E}[X] = \frac{1}{p} = \frac{\sqrt{n}}{a} = O(\sqrt{n})$$

The same asymptotic order $\Theta(\sqrt{n})$ holds if sampling is done without replacement, since the difference is negligible compared to $\sqrt{n}$.

□

2 Probability Problem 2

Let $x_1, x_2, \ldots, x_n$ be a random permutation of the numbers $[n] := \{1, 2, \ldots, n\}$. You scan the numbers from left to right. At any time, you maintain a number M in your hand, initially $M = -\infty$. When you see x_i, if $x_i > M$ then set $M \leftarrow x_i$ ("you change M"), else leave M unchanged. Show that the expected number of times M is changed is $H_n := 1 + {}^1/_2 + {}^1/_3 + \ldots + {}^1/_n$.

2.1 Solution

Proof. Let X_i be the indicator variable $\mathbb{I}\{x_i > M\}$, and let Z be the number of times M is changed:

$$\mathbb{E}[Z] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}[X_i]$$

It follows from the initialisation of M that:

$$M = -\infty < x_1,\ \forall x_1 \in \{1, ..., n\} \implies \mathbb{E}[X_1] = 1$$

As X_i is an indicator variable, the expectation of X_i is the probability that $x_i > M$:

$$\mathbb{E}[X_i] = P(x_i > M) = P(x_i > x_j),\ \ \forall j < i = \{2, ..., n\}$$

Because the numbers $x_1, ..., x_n$ are a random permutation of the set $\{1, ..., n\}$, the probability that the latest integer x_i is larger than all of its predecessors x_j is uniform over i. This results directly from the fact that in any permutation $\{1, ..., n\}$, each of the i f[**?**]irst numbers are picked with equal probability. Thus:

$$P\left(x_i = \max\{x_1, ..., x_i\}\right) = \frac{1}{i}$$

And finally:

$$\mathbb{E}[Z] = \sum_{i=1}^{n} \mathbb{E}[X_i] = \sum_i \frac{1}{i} \triangleq H_n = \Theta(\log n)$$

□

3 Probability Problem 3

Two judges are both giving opinions on a series of n questions. On each question, they answer True or False (T or F for short). On the first question they each independently answer T or F with probability 1/2. Thereafter, they steadily become more "stick-in-the-mud": for the i^{th} question, they independently repeat their own previous answer with probability $1 - \frac{1}{i+1}$, and give the opposite answer with probability $\frac{1}{i+1}$. What is the expected number of answers on which they agree?

3.1 Solution

Let A_i be the event that both judges agree on question i, and $c_i = \frac{i}{i+1}$ be the assumed probability that any of the judges gives an opposite answer for question i. In other words, c_i is the probability that any judge did not repeat their previous answer.

Because c_i is the same for both judges, the probability that exactly one judge switches his current answer from his previous one is

$$P[\text{exactly one judge changes answer}] = c_i(1 - c_i) + c_i(1 - c_i) = 2c_i(1 - c_i)$$

Similarly, we compute the probability that the judges both keep their answer or both switch their answer:

$$P[\text{both keep or both switch}] = c_i^2 + (1 - c_i)^2$$

Using the results we have just obtained, we compute the probability that both of their ith answers are the same

$$P(A_i) = P(\bar{A}_{i-1})2c_i(1 - c_i) + P(A_{i-1})[c_i^2 + (1 - c_i)^2]$$

Where $\bar{A}_j$ is the complement of the event A_j.

To reduce verbosity, let $p_i = P(A_i)$ and define

$$x_i \overset{\text{def}}{=} 2c_i(1 - c_i) \Longrightarrow c_i^2 + (1 - c_i)^2 = 1 - x_i$$

Plugging back into our initial expression, we obtain the recursive relation

$$p_i = (1 - p_{i-1})x_i \ + \ p_{i-1}(1 - x_i)$$

Substituting p_{i-1} by the given initial probability of a judge answering either true or false ($p = \frac{1}{2}$), we have:

$$p_i = (1 - p_{i-1})x_i \; + \; p_{i-1}(1 - x_i)\Big|_{p_{i-1}=\frac{1}{2}} = \frac{1}{2}$$

Finally, let Z_i be the indicator variable $Z_i = \mathbb{I}\{A_i\}$. Then the expected number of answers on which the two judges agree is

$$\mathbb{E}\left[\sum_{i=1}^{n} Z_i\right] = \sum_{i=1}^{n} \mathbb{E}[Z_i] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}$$

4 Algorithm Problem 1

Let $A[1 \ldots n]$ be an array of pairwise distinct numbers. A pair (i, j) of indices $1 \leq i < j \leq n$ is called an *inversion* of A if $A[i] > A[j]$. (For example, the array $\langle 8, 5, 2, 7, 9 \rangle$ has four inversions—$(1, 2), (1, 3), (1, 4)$, and $(2, 3)$.)

Give a deterministic algorithm running in time $\Theta(n \log n)$ for computing the number of inversions in A. You may assume that n is a power of two. (E.g., you could use divide-and-conquer.)

4.1 solution

The naive way to count the number of inversions needed to transform an unsorted array into a sorted one is to modify the bubble sort algorithm. This sorting algorithm is based on swapping elements, so we can add a counter that increments every time a swap occurs. The pseudo-code 3 has been modified to count the number of inversions.

The problem with this solution is that the time complexity of the bubble sort is $\Theta(n^2)$, whereas the desired complexity is $\Theta(n \log n)$. This complexity suggests that the algorithm we are looking for cannot be asymptotically faster than a standard sorting algorithm. The solution is therefore to modify the merge sort algorithm to count the number of inversions.

The intuition is as follows: Whenever an element from the right subarray is placed before an element from the left subarray during merging, it forms an inversion with each of the remaining elements in the left subarray. The total number of swaps required to sort the array is the sum of the swaps at the current step plus those from the two subarrays. The method Merge-and-count (algorithm 1) implements this idea by counting how many swaps are needed whenever an element from the right subarray is placed before elements from the left subarray. In Counting-merge-sort (algorithm 2), the total number of swaps is obtained by summing the swaps from the current step with those from the recursive calls.

Every inversion either (i) lies entirely in the left half, (ii) lies entirely in the right half, or (iii) spans across the two halves. Cases (i) and (ii) are counted recursively. Case (iii) is counted during the merge step by adding the number of remaining elements in the left subarray whenever an element from the right subarray is placed before them. Hence all inversions are counted exactly once.

Since the recursive structure of the algorithm call has not been modified ($T(n) = 2T(n/2) + O(n)$), the running time remains $\Theta(n \log n)$. This can be verified using the Master Theorem: at each step, two subproblems of size $n/2$ are created, which falls under the balanced case.

Algorithm 1 Merge and count number of inversions

```
procedure MERGE-AND-COUNT(Left, Right)
    $l, r \leftarrow 0$
    $Sorted \leftarrow []$
    $swaps \leftarrow 0$
    while $l \leq Left_{length}$ and $r \leq Right_{length}$ do
        if $Left[l] \leq Right[r]$ then
            append $Left[l]$ to $Sorted$
            $l \leftarrow l + 1$
        else
            append $Right[r]$ to $Sorted$
            $r \leftarrow r + 1$
            $swaps \leftarrow Left_{length} - l$          ▷ Swapped with the remaining elements in Left
        end if
    end while
    append $Left[l :]$ to $Sorted$
    append $Right[r :]$ to $Sorted$
    return $Sorted, swaps$
end procedure
```

Algorithm 2 Merge-sort with counting modification

```
procedure COUNTING-MERGE-SORT(A)
    $swaps \leftarrow 0$
    if length of $A > 1$ then return A, 0
        $m \leftarrow \frac{\text{length of} A}{2}$
        $Left \leftarrow$ COUNTING-MERGE-SORT$(A[: m])$
        $Right \leftarrow$ COUNTING-MERGE-SORT$(A[m :])$
        $A, c \leftarrow$ MERGE-AND-COUNT$(Left_{arr}, Right_{arr})$
        $swaps \leftarrow Left_{swap} + Right_{swap} + c$
    end if
    return $A, swaps$
end procedure
```

The depth of the tree is $O(\log n)$, and each level requires $O(n)$ work, bringing the total complexity to $O(n \log n)$.

Algorithm 3 Bubble sort with modification to count the number of inversions

```
procedure COUNTING-BUBBLE-SORT(A)
    $swap \leftarrow 0$
    for $i \leftarrow 0$ to $A_{length}$ do
        for $j \leftarrow 1$ to $A_{length} - i$ do
            if $A[j-1] > A[j]$ then
                swap $A[j-1]$ and $A[j]$
                $swap \leftarrow swap + 1$
            end if
        end for
    end for
    return $swap$
end procedure
```

5 Algorithm Problem 2

You are given a (connected) directed acyclic graph $G = (V, E)$ where each edge $e \in E$ has length ℓ_e. Give a polynomial-time algorithm to find the longest directed path in the graph. Ideally your algorithm should run in linear time in the number of edges of G.

5.1 solution

Given a directed acyclic graph $G = (V, E)$ and a list of lengths L, the longest path is defined as a path from a source vertex $s \in V$ to a target $t \in V \backslash \{s\}$. Because the graph is acyclic, all edges and vertices on such path are unique. The key idea is to compute the longest path starting from each vertex in V, and take the maximum over all.

The longest path starting from a vertex v through an outgoing edge $e = (v, w)$ is given by the sum of length l_e and the longest path starting at w, denoted w^*. Thus, the longest path starting at v is expressed as:

$$\max_{e \in E_v}(l_e + w^*)$$

where E_v is the set of edges outgoing of v. The algorithm 4 implements this idea.

The algorithm uses a map S that stores the maximum path length for each vertex, thereby avoiding redundant computations. It is **DFS-based**, since a vertex's longest path is determined only after exploring all of its neighbors. The base case is 0, because if no outgoing edge or only negative paths are available, the longest path is to terminates at that vertex.

The longest path of the graph G is given by:

$$\max_{v \in V}(v^*)$$

Where v^* denotes the longest path starting from v. The method Maximum-path-length 5 computes the maximum by looping over all vertices. This ensures that every vertex is computed, even if the graph is not connected. For the dynamic programming implementation, the array S is initialized to NaN, and a variable m is used to store the current maximum.

Algorithm 4 DFS-based algorithm to get the maximum path length from vertex

```
procedure DFS-MAX-LENGTH(G, v, L, S)
    if $S[v]$ is $NaN$ then
        $S[v] \leftarrow 0$
        for $(v, w) \leftarrow e \in E_v$ do
            $S[v] \leftarrow \max(S[v], L_e + \text{DFS-MAX-LENGTH}(G, w, L, S))$
        end for
    end if
    return $S[v]$
end procedure
```

Algorithm 5 Maximum path length

```
procedure MAX-PATH-LENGTH(G, L)
    $S \leftarrow [NaN] \forall v \in V$
    $m \leftarrow -\infty$
    for $v \in V$ do
        $m \leftarrow \max(m, \text{DFS-MAX-LENGTH}(G, v, L, S))$
    end for
    return $m$
end procedure
```

Since every vertex is computed exactly once due to dynamic programming and the loop, the algorithm performs $O(|V|)$ at vertex-level operations. At each vertex, the work consists of iterating over outgoing edges ($O(E_v)$). Because all vertices are processed, every edge of the graph is considered exactly once, since:

$$\bigcup_{v \in V} E_v = E$$

Thus, the total complexity is linear to the size of the graph $O(|V| + |E|)$.

As the graph G is acyclic, this bound can also be expressed as being linear in the number of edges. To illustrate this, consider two extreme cases (excluding isolated vertices):

- **Sparse case**: Suppose the graph consists of disjoint pairs of vertices connected by a single edge. Then $|V| = 2|E|$. Substituting into the formula gives

$$O(|V| + |E|) = O(2|E| + |E|) = O(3|E|) = O(|E|)$$

- **Dense case**: Suppose every vertex has edges to all vertices that follow it in a topological ordering. Then:

$$|E| = |V|\frac{(|V| - 1)}{2} \Rightarrow |E| \geq |V|$$

  Substituting into the expression:

$$O(|V| + |E|) = O(|E| + |E|) = O(2|E|) = O(|E|)$$

Therefore, in both extremes, the algorithm runs in time $O(|E|)$, linear in the number of edges.