Midterm Exam

Cheat Sheet

We recall the following fundamental inequalities. To this end, let n be a positive natural number. We define 0^0 to be equal to 1. It holds that:

- $\sum_{i=1}^{n} \frac{1}{i} \leq \ln(n) + 1$ (partial Harmonic series).
- $\left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{\mathrm{e}}$.

Further, the Python files that you should edit contain the following additional functions (from the standard library). **Do not import any other functions.**

- The function `log(x, b)` returns the logarithm of `x` to the base `b`. The parameter `b` is optional. If not used, the function returns the natural logarithm of `x`.
- The function `log2(x)` returns the binary logarithm of `x`.
- The function `randint(a, b)` returns an integer that is at least `a` and at most `b` (that is, both values are inclusive!), drawn uniformly at random.
- The function `uniform(a, b)` returns a floating-point value that is at least `a` and less than `b`, drawn uniformly at random.

General Information

This exam uses four files, which you find on Moodle: `majority.py`, `rshs.py`, `test_majority.py`, and `test_rshs.py`. The first two correspond to the two different parts of this exam. They contain skeleton code that you are supposed to edit. In addition, some questions ask you to provide a text answer. For these answers, you find corresponding comments in those files under the respective label of the question, for example, `## Question 1` for Question 1.

The other two files are there for testing purposes, each referring to one of the previous two files. The test files work out of the box, and they do not have to be edited. In fact, any changes you make to these files are going to be ignored in the grading process. Thus, make sure to have all your answers in the other two files, that is, not in the test files.

When we ask for the run time of an algorithm, it is sufficient to use big-O notation. Sometimes, in the source code, you are further asked to provide a short justification (starting with `# Because:`). In such a case, a single meaningful sentence is enough.

Determining the Majority

The questions of this part refer to the file `majority.py`. You can test your solutions by running the file `test_majority.py`.

Let L be a list with $n \geq 1$ (not necessarily pairwise different) elements. We call an element a of L the *majority element of* L if and only if a appears strictly more than $n/2$ times in L. Note that L does not need to contain a majority element. We are interested in determining efficiently whether L contains a majority element and, if so, what it is. In the following, we exclusively consider lists that contain numbers.

In order to find the majority element, it is useful to know the frequency of a given value.

Question 1. Complete the function `getFrequency(v, L, start, stop)`, which is given a number `v`, a list `L`, and two indices, `start` and `stop`, which we assume to be between `0` and `len(L)`. The function should return the absolute frequency of `v` in `L` when only considering elements whose index is at least `start` and less than `stop` (that is, it should consider `L[start:stop]` but not create a copy of `L`). What is the worst-case run time of this function for a list of length n?

The naive approach for determining the majority element is to check for each element of the list whether it is the majority element.

Question 2. Complete the function `getMajorityNaively(L)`, which is given a list `L` and should return the majority element of `L` via the naive approach, if there is a majority element. If `L` does not have a majority element, then the function should return `False`. Do not make use of a dictionary. What is the worst-case run time of this function for a list of length n?

A better way to compute the majority element is to utilize a divide-and-conquer subroutine, explained in the following. Note that this subroutine only returns an element that *might* be the majority element. Thus, we check afterward whether this candidate actually *is* the majority element.

For a list of a single element, we return this element as the majority element. For a list L of length $n \geq 2$, we determine the majority element a of the first half of L and the majority element b of the second half of L, both times recursively. Then, for both a and b, we determine the frequency in L. Afterward, we compare the frequency of a and of b, and we return the element with the higher frequency. In case of a tie, we return a.

For the algorithm that computes the majority element, we need to check whether the candidate element we received via the divide-and-conquer subroutine is actually the majority element of the entire list. To this end, we count the frequency of the candidate element. If the element occurs more than half of the length of the entire list, we return it, otherwise we return `False`.

Question 3. Complete the function `getMajorityDaC(L)`, which is given a list `L` and should return the majority element of `L`, using the divide-and-conquer subroutine described above, if there is a majority element. If `L` does not have a majority element, then the function should return `False`. Implement the subroutine via the auxiliary function `helpGetMajorityDaC(start, stop)`, which should return a candidate for the majority element, via divide and conquer, from `L[start:stop]` (recall that `start` is inclusive and that `stop` is exclusive). What is the worst-case run time of `getMajorityDaC(L)` for a list of length n?

The divide-and-conquer approach explained above also works well in higher dimensions. For example, given a square matrix M—which is a list that contains 2^k lists all of the length 2^k—, the approach works analogously by splitting M into four equally large square matrices and then recurring on each of these four matrices. At the very end, we need to check again whether the element we receive actually is the majority element of M.

Question 4. Complete the function `getMajorityInMatrixDaC(M)`, which is given a square matrix `M` and should return the majority element of `M` via the method sketched above, if there is a majority element. If `M` does not have a majority element, then the function should return `False`. As in the previous question, implement the subroutine via the auxiliary function `helpGetMajorityDaC(rowStart, rowStop, colStart, colStop)`, which returns a candidate for the majority element in the square sub-matrix `M[rowStart:rowStop][colStart:colStop]`. Note that we provide a function `getFrequencyFromMatrix(v, M, rowStart, rowStop, colStart, colStop)`, which returns the frequency of `v` in the specified square sub-matrix of `M` (provided that Question 1 is answered correctly). What is the worst-case run time of `getMajorityInMatrixDaC(M)` for a matrix containing $n = 2^k \cdot 2^k$ elements?

We return to the one-dimensional case. Boyer and Moore came up with a variant that is better than the divide-and-conquer approach above and only requires a single run over the list in order to determine a candidate for the majority element. Afterward, this element needs to be checked whether it actually is the majority element.

The Boyer–Moore algorithm is sketched in Figure 1. It works by maintaining a candidate for the majority element, namely `curr`, as well as a value, namely `freq`, of how much more often `curr` occurred

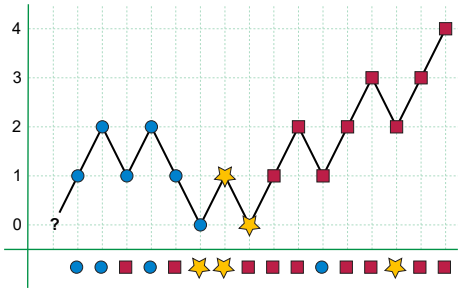

Figure 1: A visual representation of the Boyer–Moore algorithm (from Wikipedia). The x-axis denotes the index of the list, which itself is depicted below the x-axis. The y-axis shows the values of `freq`. The curve shows the values of `freq` (the y-value) and of `cur` (the symbol) at each iteration.

than other elements. We initialize `freq` with `0`, and we initialize `curr` with `None`. Then, we do the following for each element of the list: If `freq` is `0`, we set `curr` to the current element of the list, and we increase `freq` by `1`. Otherwise, if the element is different from `curr`, we decrease `freq` by `1`. Otherwise, we increase `freq` by `1`. After we iterate over the list, `curr` is our candidate for the majority element. We check its frequency over the entire list and either return `curr` (if it is the majority element) or return `False`.

Question 5. Complete the function `getMajorityBoyerMoore(L)`, which is given a list `L` and should return the majority element of `L` via the Boyer–Moore algorithm, if there is a majority element. If `L` does not have a majority element, then the function should return `False`. What is the worst-case run time of this function for a list of length n?

Last, we go for a randomized approach. Given a list, we choose one index uniformly at random and check whether the element at that index is the majority element. If it is, we return the element. If not, we return `False`.

Question 6. Assuming that a list of $n \geq 1$ elements contains a majority element, what is *at least* the probability that the randomized approach above finds it? In the same setting, what is *at most* the probability that $m \geq 1$ independent tries of this randomized approach all incorrectly return `False`?

Complete the function `getMajorityRandomized(L, p)`, which is given a list `L` and a probability bound `p` (a value in $[0, 1)$) and which should return with probability at least `p` the majority element of `L`, if there is a majority element. Otherwise, it should return `False`. Your function should iterate the randomized approach above as little times as possible while still satisfying the requirement on `p`.

Randomized Search Heuristics

The questions of this part refer to the file `rshs.py`. You can test your solutions by running the file `test_rshs.py`.

Randomized search heuristics (RSHs) are optimization algorithms applied to scenarios we know little about. In other words, the objective function f to optimize acts as a *black box*. Here, we consider pseudo-Boolean *maximization*, where f maps bit strings of the same length $n \geq 1$ to real numbers[1]. Note that higher values of f are better for us. We also say that a bit string x is better than another bit string y if and only if $f(x)$ is strictly greater than $f(y)$—and analogously for other relations.

Simple RSHs maintain a single solution x^* (that is, a bit string) and try to improve it by making some random changes to x^*, resulting in another solution y. This operation is often referred to as a *mutation*. If y is at least as good as x^*, we replace x^*, otherwise we keep x^*. We iterate this procedure of mutating x^* and potentially updating it until a certain, user-defined, termination criterion is met. Then, we return x^*. We initialize x^* with a solution that is chosen uniformly at random from the domain $\{0, 1\}^n$. Note that we model bit strings as lists of length n that only contain values from $\{0, 1\}$.

[1] On the computer, we use floating-point values.

Question 7. Complete the function `createBitStringUniformly(n)`, which is given a positive integer `n` and should return a bit string of length `n` that is chosen uniformly at random from $\{0,1\}^n$.

An easy way of mutating an individual is to copy it, choose one position of the copy uniformly at random, and then invert the bit at that position. This mutation is also known as *single-bit flip*.

Question 8. Complete the function `singleBitFlip(x)`, which is given a bit string `x` and should return a *new* bit string that implements the single-bit flip.

A similar operation chooses for *each* position of a (copied) bit string of length n independently whether to invert the bit at the respective position. Each inversion occurs with a probability of $1/n$ (independently of other potential inversions). This mutation is known as *standard bit mutation*.

Question 9. Complete the function `standardBitMutation(x)`, which is given a bit string `x` and should return a *new* bit string that implements the standard bit mutation. What is the *expected number* of bits that are different between `x` and the result of the mutation?

The simple RSH (maintaining only a single solution) that implements the single-bit flip is known as *random local search* (RLS). The simple RSH that implements the standard-bit mutation is known as the $(1+1)$ *evolutionary algorithm* ($(1+1)$ EA).

Question 10. Complete the function `rsh(n, f, terminated, mutation)`, which is given the length `n` of the bit strings it optimizes, the objective function `f` (which takes a single bit string as argument and returns a number), the termination criterion `terminated` (which takes a tuple `(x, it)` as input, where `x` is the best bit string so far and `it` the current number of iterations, and it returns `True` when the algorithm should stop and `False` otherwise), as well as the mutation operator `mutation` (which is either the single-bit flip or the standard bit mutation). The function `rsh(n, f, terminated, mutation)` should implement the framework of a simple RSH, as described above. Your function should return a tuple `(x, it)`, where `x` is the best solution upon termination and `it` is the number of total iterations.

When conducting run time analysis of RSHs, one is typically interested in the *expected* time until the algorithm finds an optimal solution for the *first* time. We call this value the expected run time of the algorithm. To this end, one considers *specific* benchmark functions (which *we* know but the algorithm does not). Two very popular benchmark functions are OneMax and LeadingOnes, which map a bit string $x \in \{0,1\}^n$ to an integer as follows:

1. $\textsc{OneMax}(x) = \sum_{i=1}^{n} x_i$ (returns the number of 1s of x).
2. $\textsc{LeadingOnes}(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j$ (returns the longest prefix of x that consists exclusively of 1s).

Note that both functions only return integers in the interval $[0,n]$. Further, for both functions, the unique global optimum is the all-1s bit string.

The expected run time of simple RSHs on these functions can be easily bounded from above by assuming that the RSH currently has a solution with an objective-function value of $i \in [0..n-1]$. In order to improve this solution, it suffices to create a solution with a better objective-function value. We call this improving event A_i. Hence, in expectation, it takes $1/\Pr[A_i]$ iterations until the algorithm improves a solution with an objective-function value of i. Since the run time T of a simple RSH is the sum of the waiting times spent improving its current solution and since the simple RSH needs to improve a solution at most n times before it creates an optimal solution, it follows that

$$\mathrm{E}[T] \leq \sum_{i=0}^{n-1} \frac{1}{\Pr[A_i]} \,. \tag{1}$$

Question 11. Let n be the length of the bit strings to optimize. Provide your answers in the comments at the bottom of `rshs.py`.

- Consider the $(1+1)$ EA maximizing OneMax. For each $i \in [0..n-1]$, provide a good lower bound for $\Pr[A_i]$. Calculate an asymptotic upper bound in terms of n for the expected run time, using inequality (1).
- Consider the RLS maximizing LeadingOnes. For each $i \in [0..n-1]$, determine the exact probability of $\Pr[A_i]$. Calculate an asymptotic upper bound in terms of n for the expected run time, using inequality (1).