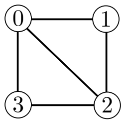
# Midterm Exam

## 1 Generating random walks and random spanning trees on a graph

Download the file `rw.py` and the file `test_rw.py`. The file `rw.py` contains a class `Graph` (with methods to complete). An object $g$ of type `Graph` has an attribute `g.n` that gives the number of vertices (the $n$ vertices have distinct labels in $[0..n-1]$), and an attribute `g.L` that is a list of lists, such that `g.L[i]` gives the list of neighbours of vertex $i$ (in whatever order), see Figure 1 for an example.

$g =$ 


$g.n = 4$
$g.adj = [[1, 2, 3], [0, 2], [0, 3, 1], [0, 2]]$

FIGURE 1 – Left : a graph $g$. Right : the encoding of $g$ as an instance of the class `Graph`. For instance the neighbours of vertex 1 are 0 and 2, so that $g.L[1] = [0, 2]$.

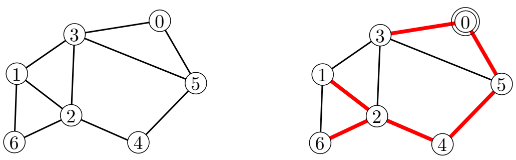
**Question 1** (1 pts)**.** Complete the method `random_neighbour(self,i)` that has to return one of the neighbours of vertex $i$ at random (i.e., if $i$ has $d$ neighbours, then each neighbour has probability $1/d$ of being returned). To test your method, uncomment the line containing `test_random_neighbour()` at the end of `test_rw.py`, and then run that file.

For $k \geq 0$, a random walk of length $k$ on a graph $g$ (starting from vertex 0) is a list $[v_0, v_1, \ldots, v_k]$ of vertices such that $v_0 = 0$, and for each $i \in [1..k]$, $v_i$ is a random neighbour of $v_{i-1}$.

**Question 2** (2 pts)**.** Complete the method `random_walk(self,k)` that has to return a random walk of length $k$ starting from 0 (the returned list should have length $k + 1$ and start with 0). To test your method, uncomment the line containing `test_random_walk()` at the end of `test_rw.py`, and then run that file.

When drawing a random walk $v_0, v_1, v_2, \ldots$, the *cover time* is the first time where we have visited (at least once) each of the $n$ vertices.

**Question 3** (3 pts)**.** Complete the method `random_walk_till_covered(self)` that draws a random walk starting at 0, stops at the cover time, and returns the walk seen so far (the returned list should start with 0, contain every entry in $[0..n-1]$ at least once, and the last entry of the list should occur just once in the list). To test your method, uncomment the line containing `test_cover_walk()` at the end of `test_rw.py`, and then run that file.



parent$= \{1\!:\!2, 2\!:\!4, 3\!:\!0, 4\!:\!5, 5\!:\!0, 6\!:\!2\}$

FIGURE 2 – Left : a graph $g$. Right : a spanning tree $T$ of $g$ (whose edges are bold red), and the encoding of $T$ as a dictionary `parent`. For instance the path from 2 to 0 in $T$ visits (after leaving 2) successively 4, 5 and 0, hence the parent of 2 is 4, so that `parent[2]=4`.

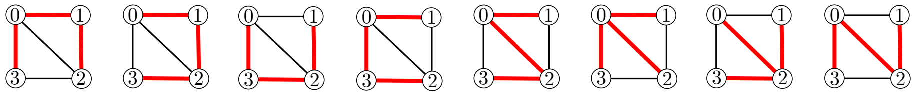



FIGURE 3 – The 8 spanning trees of the graph of Figure 1.

For $g$ a graph, a *spanning tree* $T$ of $g$ is a set of edges of $g$ that forms a tree and covers all vertices of $g$. Precisely, there is no cycle of edges from $T$, and for every vertex $i \in [1..n-1]$ there is a (necessarily unique) path of edges of $T$ from $i$ to 0. The next vertex after $i$ along this path is called the *parent* of $i$. We will store a spanning tree $T$ as a dictionary `parent` where the set of keys is $[1..n]$, and for each $i \in [1..n]$, `parent[i]` gives the label of the parent of $i$, see Figure 2 for an example.

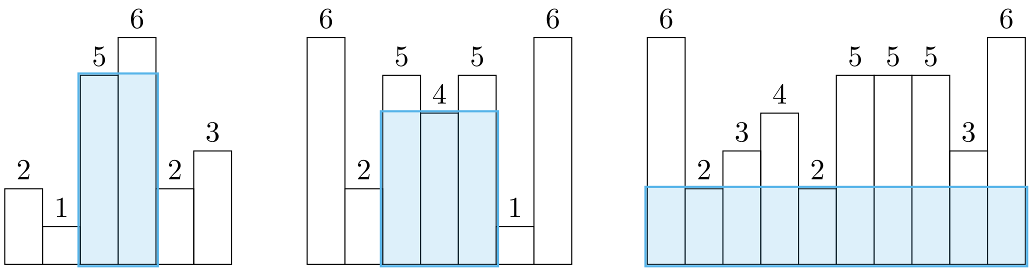
We let $\mathcal{ST}(g)$ be the set of spanning trees of $g$ (see Figure 3 that shows the set $\mathcal{ST}(g)$ of 8 spanning trees of the graph $g$ of Figure 1). We would like to have a method that returns a random element in $\mathcal{ST}(g)$ (for instance for the graph of Figure 1, each of its 8 spanning trees shown in Figure 3 should have probability 1/8 of being chosen). As it turns out, one can use a random walk till cover time to do that. Precisely, if $W$ is a random walk of $g$ (starting at 0) till cover time, for each $i \in [1..n-1]$ we let $p(i)$ be the label of the vertex preceding the *first occurence* of $i$ in $W$. Then it can be shown (admitted here) that the tree formed by the $n-1$ edges $\{i, p(i)\}$ is a random spanning tree of $g$ (each element of $\mathcal{ST}(g)$ has the same probability of being chosen). It is easy to see that $p(i)$ is the parent of $i$ in this tree (indeed the path from $i$ to 0 visits $p(i)$, then $p(p(i))$, etc.).

**Question 4** (2 pts)**.** Complete the method `random_spanning_tree(self)` that has to return a dictionary `parent` associated to a random spanning tree of `self` (your code should not call the method `random_walk_till_covered`, but it should have a quite similar structure). To test your method, uncomment the line containing `test_random_tree()` at the end of `test_rw.py`, and then run that file.

## 2 Largest Rectangle in Histogram

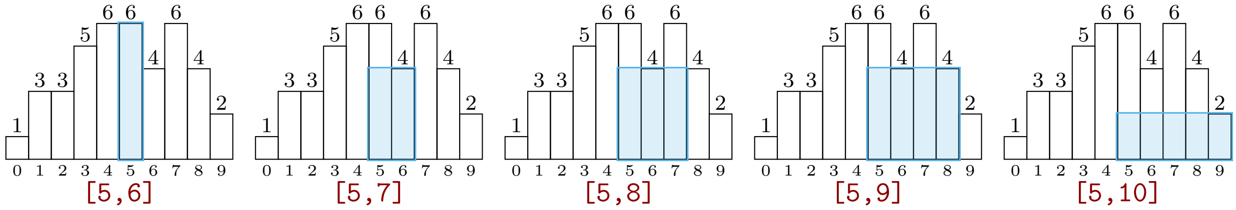
**Guidelines.** Download `rect_hist.py` and `rect_hist_test.py`. Write your solution in `rect_hist.py`. To test your code, run the file `rect_hist_test.py`. You don't have to justify the complexity of your programs.

We are given a histogram, a sequence of positive values, and we want to find the area of the largest rectangle in the histogram. In this exercise, $n$ will always refer to the length of the histogram. The largest such rectangle is illustrated below on several examples :




`[2,1,5,6,2,3]` max area : 10 `[6,2,5,4,5,1,6]` max area : 12 `[6,2,3,4,2,5,5,5,3,6]` max area : 20

At first, we consider the following simpler problem : given a position $i$, find the area of the largest rectangle starting at position $i$. The example below illustrates all possible such rectangles on a histogram with $i = 5$.




**Question 5** (2 pts)**.** Write an algorithm `rect_from_left(hist, i)` that given an array `hist` representing the histogram and a position `i`, returns the area of the largest rectangle in the histogram that starts at position `i`. Make sure your algorithm has complexity $O(n)$. Test your implementation using `rect_hist_test.py`.

**Question 6** (1 pts)**.** Write a "brute force" algorithm `rect_hist_brute(hist)` that given an array `hist` representing the histogram returns the area of the largest rectangle in the histogram, using `rect_from_left`. Make sure your algorithm has complexity $O(n^2)$. Test your implementation using `rect_hist_test.py`.

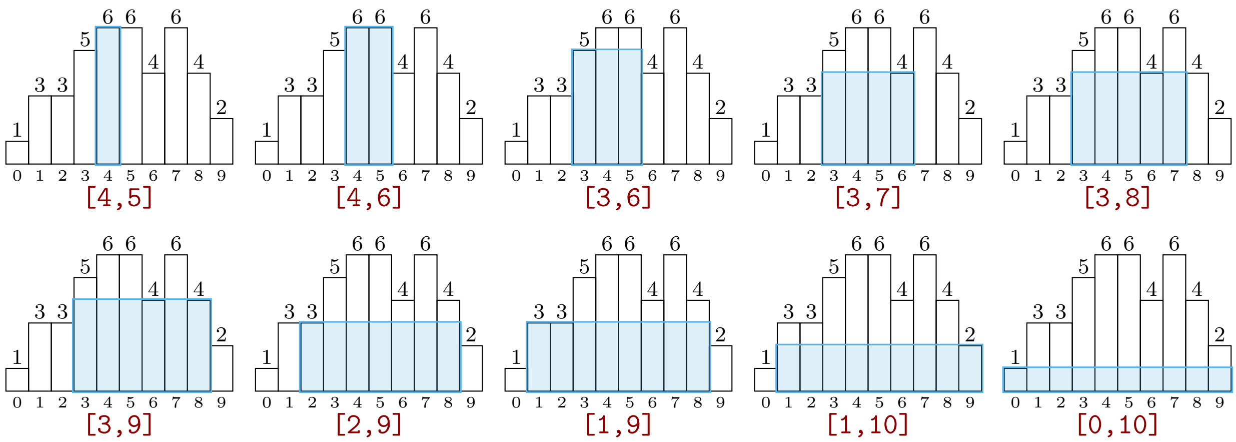
Now we would like to design a divide and conquer algorithm to solve this problem more efficiently. Consider a histogram $H$ and a sub-histogram $H[i:j]$ with $i < j$, and let $m = \lfloor \frac{i+j}{2} \rfloor$. Then there are three possibilities for a rectangle spanning $H[l:r]$ of largest area :

— it is entirely within $H[i:m]$,

— it is entirely within $H[m+1:j]$,

— it is within $H[i:j]$ and contains $m$, that is $l \leqslant m < r$.

We now focus on the latter case : starting from a rectangle containing only $A[m]$, we repeatedly expand this rectangle to the left or the right until it fills all of $A[i:j]$. We then take the maximum area of all rectangles is this sequence. This process is illustrated on an example below where `H=[1,3,3,5,6,6,4,6,4,2]` and `m=4`. We have represented the different steps of the expansion process and the rectangle spanning `H[l:r]`.




Formally, given a sub-histogram $H[i:j]$ and a rectangle spanning $H[l:r]$ (with $[l:r] \subset [i:j]$), an *expansion step* consists in expanding the rectangle to the left or the right (we only expand in one direction) :

— if $l = i$ then expand to the right ($r' = r + 1$),

— if $r = j$ then expand to the left ($l' = l - 1$),

— if $H[l-1] > H[r]$ then expand to the left,

— otherwise expand to the right.

**Question 7** (3 pts)**.** Write a function `expand_rect(hist, i, j, l, r, h)` that implements one expansion step and returns a tuple `l',r',h'`. In the input, `h` is assumed to be the minimum height in `hist[l:r]`. In the output, `l'` and `r'` are such that `[l',r']` is the updated range, and `h'` has to be the minimum height in `hist[l':r']` Your algorithm should have complexity $O(1)$. Test your implementation using `rect_hist_test.py`.

**Question 8** (3 pts)**.** Write a function `best_from_middle(hist, i, j, m)` that implements the expansion process by repeatedly calling `expand_rect` starting from `l=m` and `r=m+1`, and until `l==i` and `r==j`. It must return the area of the largest rectangle in the sequence. Your algorithm should have complexity $O(j - i)$. Test your implementation using `rect_hist_test.py`.

**Question 9** (3 pts)**.** Write a recursive function `rect_hist_dac_aux(hist, i, j)` that implements the divide-and-conquer approach outlined at the beginning and uses `best_from_middle`. It must return the area of the largest rectangle in the sequence, or `-math.inf` when `i==j`. Your algorithm should have complexity $O(n \log(n))$. Test your implementation using `rect_hist_test.py`.