Quiz 1

- Do not open this quiz booklet until you are directed to do so. Read all the instructions first.
- The quiz contains 6 problems, several with multiple parts. You have 2 hours for this midterm.
- This quiz booklet contains 17 pages, including this one and three sheets of scratch paper. **Do not remove any of these pages!**
- **Do not write anything on the back of the sheets, we will not scan and therefore not grade it!**
- Write your solutions in the space provided. If you run out of space, continue on a scratch page and make a notation.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to give an algorithm in this quiz, describe your algorithm in English or pseudocode, and provide a short argument for correctness and running time. You do not need to provide a diagram or example unless it helps make your explanation clearer.
- Do not spend too much time on any one problem.
- **Please write your name on every single page of this exam.**
- Good luck!

Problem	Title	Points	Parts	Grade	Initials
0	Name	1	1		
1	True/False	14	7		
2	Ordered Stack	15	2		
3	Find the Pairs	15	1		
4	Min-cut using MST	15	2		
5	Well Spaced Triples	20	2		
6	Bin Packing	20	3		
Total		100			

Name: ______________________________

Circle your recitation:

F10	F11	F12	F1	F2	F3	F11	F12	F1
Katerina	Themis	Shankha	Maryam	Nishanth	Prashant	Manolis	Shalom	Ethan
R01	R02	R03	R04	R05	R06	R07	R08	R09

Quiz 1-1: [14 points] T/F Questions

Mark each statement as either true or false. You have to **provide a short explanation for each**.

(a) Consider $f(x) = \frac{x^3}{3}$ and $x^{(0)} \neq 0$. The gradient descent algorithm with the following iteration

$$x^{(t+1)} \leftarrow x^{(t)} - \frac{1}{2} f'(x^{(t)})$$

always converges to $-\infty$.

Solution. False

It might converge to 0 depending on the value of $x^{(0)}$. If $x^{(0)} = 1$ for example, $x^{(t+1)} = x^{(t)} - \frac{1}{2}(x^{(t)})^2$ for all t. By induction it can be seen that $0 < x^{(t)} \leq 1 \forall t$ and hence the gradient descent algorithm doesn't converge to $-\infty$.

(b) We are given an array $A[1...n]$. We can delete its $n/2$ largest elements in time $O(n)$.

Solution. True

We know from class that we can find the median of A in $O(n)$ time. Once we have found the median, we can make a linear pass over the array and delete all elements larger than the median. This will end up deleting the $n/2$ largest elements in the array.

(c) The following definition is equivalent to that of a universal hash family:
Let $\mathcal{H}$ be a family of hash functions $h : \{0,1\}^u \to \{0,1\}^m$, then for any $h \neq h' \in \mathcal{H}$

$$\mathbb{P}_{k \in \{0,1\}^u}[h(k) = h'(k)] \leq \frac{1}{m}$$

Solution. False
We know that the set of all possible hash functions is universal. Hence consider this set as our universal family. Let $h(x) = 0 \forall x$ and $h'(x) = 0 \forall x \neq 0$ and $h'(0) = 1$. Clearly h and h' belong in our universal hash family but $\mathbb{P}_{k \in \{0,1\}^u}[h(k) = h'(k)] = 1 - \frac{1}{2^u} > \frac{1}{m}$ for an appropriately chosen m and u.

(d) We define n random variables $X_i = \sum_{j=1}^{n} c_{(i,j)}$, where for every (i,j), $c_{(i,j)}$ is determined by an independent unbiased coin flip, so $c_{(i,j)}$ is one with probability 1/2 and zero otherwise. Then, for every $\beta \in (0,1)$

$$\mathbb{P}\left[\sum_{i=1}^{n} X_i > (1+\beta)\frac{n^2}{2}\right] < e^{-\beta^2 n^2/6}$$

Solution. True
We denote $\sum_{i=1}^{n} X_i$ by X. Then,

$$X = \sum_{i=1}^{n} X_i = \sum_{i,j=1}^{n} c_{(i,j)}$$

So, X is a sum of n^2 independent random variables which take value in $[0,1]$. Also, $\mathbb{E}[X] = \sum_{i,j=1}^{n} \mathbb{E}[c_{(i,j)}] = \frac{n^2}{2}$. So, from Chernoff bound we get:

$$\mathbb{P}\left[X > (1+\beta)\frac{n^2}{2}\right] = \mathbb{P}\left[\sum_{i=1}^{n} X_i > (1+\beta)\frac{n^2}{2}\right] < e^{-\beta^2 n^2/6}$$

(e) Let $G = (V,E)$ be a connected graph and let H be the subgraph of G induced by some set of vertices $V' \subset V$. That is, $H = (V', E')$ where E' consists of all edges both of whose endpoints are in V'. Then every MST of H is a subgraph of some MST of G.

Solution. False
Assume our graph is a triangle with vertices A, B and C and edge weights $w_{A,B} = 1$, $w_{B,C} = 2$, $w_{C,A} = 3$. Then, the unique MST of this graph consists of the edges $\{A,B\}$ and $\{B,C\}$. Now, if $V' = \{A,C\}$, the MST of H consists of the edge $\{A,C\}$, which is not a subgraph of the MST of G.

(f) Consider a k-bit vector $v = x_1 \dots x_k$ chosen uniformly at random from $\{0,1\}^k$. For any $S \subseteq \{1,\dots,k\}$ such that $S \neq \emptyset$, define $m_S(v) = \sum_{i \in S} x_i \pmod 2$. Then,

$$\mathrm{Var}\left[\sum_{S \subseteq \{1,\dots,k\}, S \neq \emptyset} m_S(v)\right] = \frac{2^k - 1}{4}$$

Solution. True
First, we will show that for S and S' such that $S \neq S'$, $m_S(v)$ and $m'_S(v)$ are pairwise independent. Without loss of generality we may assume that there exists an i such that $i \in S$, but $i \notin S'$.
Then,

$$\mathbb{P}[m_S(v) = \alpha \wedge m_{S'}(v) = \beta] = \mathbb{P}[x_i = \alpha + \sum_{j \in S \setminus i} x_j \pmod 2 \wedge m_{S'}(v) = \beta]$$

$$= \mathbb{P}[x_i = \alpha + \sum_{j \in S \setminus i} x_j \pmod 2]\mathbb{P}[m_{S'}(v) = \beta] = \mathbb{P}[m_S(v) = \alpha]\mathbb{P}[m_{S'}(v) = \beta]$$

Also, for every $S \subseteq \{1,\dots,k\}$

$$\mathrm{Var}\left[m_S(v)\right] = \frac{1}{4}$$

because $m_S(v)$ is a random variable that follows a Bernoulli distribution with success probability $1/2$.
Therefore,

$$\mathrm{Var}\left[\sum_{S \subseteq \{1,\dots,k\}, S \neq \emptyset} m_S(v)\right] = \sum_{S \subseteq \{1,\dots,k\}, S \neq \emptyset} \mathrm{Var}\left[m_S(v)\right] = \frac{2^k - 1}{4}$$

(g) If an algorithm runs in time $\Theta(n)$ with probability 0.9999 and in time $\Theta(n^2)$ with the remaining probability, then its expected run-time is $\Theta(n)$.

Solution. False
Expected runtime = $0.9999c_1 n + 0.0001c_2 n^2 = \Omega(n^2)$.

Quiz 1-2: [15 points] Ordered Stack

An ordered stack is a data structure that stores a sequence of items and supports the following operations.

- ORDEREDPUSH(x) removes all items smaller than x from the beginning of the sequence and then adds x to the beginning of the sequence.
- POP deletes and returns the first item in the sequence (or returns Null if the sequence is empty).

(a) [8 points] Show how to implement an ordered stack with a simple linked list. Prove that if we start with an empty data structure and perform a sequence of n arbitrary ORDEREDPUSH(x) and POP operations, then the amortized cost of each such operation is $O(1)$.

Solution. Let s be a pointer to the last node of the stack which is initially Null. Also, $s.prev$ and $s.next$ are the links to the previous and the next element in the linked list. The algorithms for ORDEREDPUSH(x) and POP() operations are shown below.

Algorithm 1 ORDEREDPUSH(x)

```
1: while $s \neq$ Null and $s.val < x$ do
2:     $s \leftarrow s.prev$
3: end while
4: $s.next \leftarrow$ Null
5: Allocate memory for a new node $a$
6: $a.val \leftarrow x$
7: $a.prev \leftarrow s$
8: $s.next \leftarrow a$
9: $s \leftarrow a$
```

Algorithm 2 POP()

```
1: if $s =$Null then
2:     return Null
3: else
4:     $x \leftarrow s.val$
5:     $s \leftarrow s.prev$
6:     return x
7: end if
```

Now, we analyze the running time. The actual cost in the POP is a constant a_1. The actual cost of the ORDEREDPUSH is a_2#popped elements$+ a_3$. Let a be the maximum of the a_1, a_2, and a_3. Let c_i denote the actual cost of operation i. Assume the size of the stack changed by t_i after this operation. In the POP, t_i is -1. In the ORDEREDPUSH t_i is $1 - \#$popped elements. It is not hard to see

$$c_i \leq 2\,a - a\,t_i$$

We define a potential function $\Phi(S)$ to be a times the number of elements in the stack for any given status of the stack S and some sufficiently large constant c. Let S_i be state of the stack after the i-th operation. Clearly, $\Phi(S_0) = 0$ and $\Phi(S_n) \geq 0$. For the i-th operation we define the amortized cost to be

$$c'_i = \Phi(S_i) - \Phi(S_{i-1}) + c_i = a\,t_i + c_i \leq 2\,a.$$

Thus c'_i's are $O(1)$. Since we have $\sum_{i=1}^{n} c_i = \Phi(S_0) - \Phi(S_n) + \sum_{i=1}^{n} c'_i \leq 2\,a\,n$, the amortized cost of each operation is $O(1)$.

We can have a similar argument in accounting method. Let say for each operation we put $\$2a$ in the bank account. Similarly, we can prove that the bank account will always have $2a$ times the number of elements in the stack. Thus, it never goes to below zero.

(b) [7 points] Suppose we are given an array $A[1 \ldots n]$ that stores the height of n buildings on a city street, indexed from west to east. The view of building i is defined as the number of buildings between building i and the first building west of i that is taller than i. For example if $A = [1, 5, 2, 3, 1, 2, 1, 8]$, then the view of building 4 (which has height 3) is 1.

You want to compute the view of the n buildings of A. Show how you can modify the ordered stack in order to do that in $O(n)$ time.

Solution. We modify the stack in three ways. First, we associate with each element the "view" of the element. Second, when doing an ORDEREDPUSH, sum $(element.view + 1)$ over the elements popped from the stack and store this value as the view of the item being added. Finally, when doing an ORDEREDPUSH, pop all elements with values less than or equal to the value of the element to be added (not just those elements with strictly smaller values).

With these modifications, we can simply ORDEREDPUSH the heights of the buildings in order from west to east onto an empty ordered stack.

Since the amortized cost of each operation is $O(1)$, the total running time is $O(n)$.

The proof of correctness is immediate after noticing that the view of each building b is given by $\sum_{s \in S} (view(s) + 1) - 1$, where S is the set of elements that get popped when b is ORDEREDPUSH into the stack. We will prove this by induction:

- For the most west building, this procedure gives view equal to zero, which is the correct view by definition.
- Now suppose that when we are about to ORDEREDPUSH the k^{th} building height $A[k]$ into the stack, all the items in the stack have their actual view stored in their

$view$ field. Let $S = \{s_1, s_2, ..., s_j\}$ be the set of indices in A (in decreasing order) of the items that get popped when we push $A[k]$. Let b be the index in A of the first building that is strictly taller than $A[k]$. Now, for every $i \in \{1, ..., j\}$, $view(A[s_i]) = s_i - s_{i+1} - 1$ (where we define $s_{j+1} = b$) because $A[s_{i+1}]$ is the building that blocks the view of $A[s_i]$. Also, by the inductive hypothesis, the head of the stack currently has height $A[k-1]$. So, if $S \neq \emptyset$, $s_1 = k - 1$.
Then, the algorithm sets $view(k)$ as:

$$view(k) = \sum_{s \in S}(view(s) + 1) = \sum_{i \in \{1,...,j\}} (s_i - s_{i+1}) = s_1 - b = k - b - 1,$$

which is by definition the view of item k.
Other solutions are possible, such as pushing the building heights from east to west onto the ordered stack and setting the view of a building b to be the total number of pushes between when b was first pushed on and when it was popped off, as this counts precisely the number of buildings west of b that are not taller than b.

Quiz 1-3: [15 points] Find the Pairs

Consider a set U of $2n$ distinct balls numbered from $1 \ldots 2n$ distributed evenly across n containers (each container has exactly 2 balls). You are not allowed to look inside the containers. But, we will answer your queries of the following form:

"*What is the smallest number of containers that contain all the balls in* S?" where $S \subseteq U$ is any subset of the $2n$ balls.

For example if $n = 4$ and the balls were arranged in the following way: $\{1, 4\}, \{2, 8\}, \{3, 6\}, \{5, 7\}$ in 4 containers, an example query would be "*What is the smallest number of containers that contain the balls* $\{1, 4, 2, 6\}$?" to which we would reply 3.

Two balls are said to be paired if they lie in the same container. Your task is to figure out for each ball the other ball it is paired with. Give an algorithm for figuring out all the pairings using $O(n \log n)$ queries. Note that we only care about the total number of queries your algorithm uses and not the running time.

Solution. Assume we have the ball x and we want to find its pair y in $O(\log n)$ queries. Let S be a subset of $\{1, \ldots, n\}$ and $Q(S)$ be the answer of query for the set S. For a set S that does not contain x, we can determine if $y \in S$ by the following:

- If $Q(S \cup \{x\}) \neq Q(S)$, then y is not in S.
- If $Q(S \cup \{x\}) = Q(S)$, then y is in S.

Since the answer of the query is the smallest number of required bin, the above it true. Now, we can use the binary search to find the y. First, start with $S = \{1, \ldots, n\}$. Partition S into two subsets S_1 and S_2 of the size $\lceil \frac{|S|}{2} \rceil$ and $\lfloor \frac{|S|}{2} \rfloor$. By the above approach, test which one contains the y. Let the new S be that subset and continue until $|S|$ become one (i.e. $S = \{y\}$). It is not hard to see that

$$q(n) = q(\frac{n}{2}) + O(1).$$

Therefore, to find the each pair we use $O(\log n)$ queries. Thus, the total number of queries is $O(n \log n)$.

An alternate approach uses divide and conquer to split the problem into two sub-problems with $\frac{n}{2}$ pairs of balls each. Start with $S_1 = \{1, ...n\}, S_2 = \{n + 1, ..., 2n\}$. Now query for S_1. If the number of pairs is already $\frac{n}{2}$, recurse on S_1 and S_2 separately. Otherwise, we need to swap balls between S_1 and S_2 so that each pair is entirely contained in a single S_i.

So, for each ball i in S_2, query for $S_1 + \{i\}$. If the number of pairs does not change, then i should be matched with a ball in S_1, so add it to S_1. Continue adding balls that match until S_1 contains $\frac{n}{2}$ pairs. Now, iterate through each ball i in S_1 and query $S_1 - \{i\}$. If the value returned by the query is the same as for S_1, then remove i from S_1. Once we've done this for every ball in S_1, S_1 has exactly $\frac{n}{2}$ pairs, so S_2 does too, and we can recurse.

The recursion is

$$q(n) = O(n) + 2q(n/2)$$

which gives a final solution of $O(n \log n)$ queries.

Note: A similar approach would divide the balls into two equal sized sets and try to get each set to be perfectly paired by doing swaps. This approach is much harder to get to work using linear number of swaps and just mentioning that we can do the swaps in linear time did not receive a close to full score for this reason.

Quiz 1-4: [15 points] Min-cut using MST

Consider the following algorithm that takes as input an *unweighted* graph and outputs a cut of this graph:

```
KCUT($G(V, E)$):
    Create weighted graph $G'(V, E, w)$, where $w$ is a uniformly random assignment of
        distinct weights from $\{1, \ldots, |E|\}$ to edges in $E$.
    Run KRUSKAL($G'$) to get a tree $T = (V, E_T)$.
    Let $e_f$ be the last edge that was added to $E_T$ in the above execution of KRUSKAL($G'$).
    $T' = (V, E_T \setminus \{e_f\})$ has exactly two connected components. Call them $C$ and $V \setminus C$.
    **return** the cut $(C, V \setminus C)$.
```

(a) [10 points] Prove that for any unweighted graph $G(V, E)$, the probability that KCUT(G) returns a minimum cut of G is at least $\Omega\left(\frac{1}{|V|^2}\right)$.

Solution. We are going show an equivalence between the $KCUT$ algorithm and $GUESSCUT$ algorithm which we saw in class. In particular, we are going to prove that one iteration of $KCUT$ exactly corresponds to one iteration of $GUESSCUT$ (i.e a sequence of $n-2$ edge contractions) in the sense that the cut the two algorithms finally output is drawn from the same distribution. The proof is by induction.

Base case:
Firstly, notice that the first edge KRUSKAL's algorithm picks here is a uniformly random edge of the graph since by symmetry each edge has probability 1/m of being the minimum weight edge. $GUESSCUT$ would also pick the first edge to contract uniformly at random.

Inductive step:
Now, assuming that $GUESSCUT$ and $KCUT$ have picked exactly the same edges $\{e_1, ..., e_{k-1}\}$ so far, we will show that they will pick e_k (i.e the next one) from the same distribution. $GUESSCUT$ picks a uniformly random edge from $G/\{e_1, ..., e_{k-1}\}$ (excluding self-loops), while KRUSKAL's algorithm in KCUT would pick the lightest of the remaining edges that does not create a cycle with other edges from $\{e_1, ..., e_{k-1}\}$. An edge e creates such a cycle if and only if its endpoints are connected by a path which only contains edges in $\{e_1, ..., e_{k-1}\}$. The latter happens if and only if e is a self-loop in $G/\{e_1, ..., e_{k-1}\}$ and therefore ignored by $GUESSCUT$. Since we just established that the two distributions have the same support (i.e they choose from the same set of edges), it remains to show that $KCUT$ will also pick uniformly from those edges as $GUESSCUT$ does. We can see that the latter is true by symmetry just like in the base case.

So, the joint distribution of the first n-2 edges to be picked by either algorithm is the same. Since these edges uniquely define two connected components (i.e a cut) in the graph, the output cut is drawn from the same distribution.

Thus, the probability that $KCUT$ returns a minimum cut is equal to the probability $GUESSCUT$ does, which is $\Omega(1/n^2)$.

(b) **[5 points]** We want to find a min-cut with high probablity, namely with probability at least $1 - \frac{1}{n}$. Design an algorithm that uses the KCUT procedure to achieve this, and analyze its running time. Recall that Kruskal's algorithm has running time $O(m \log m)$, where $m = |E|$.

Solution. As in Karger's algorithm, we will repeat the $KCUT$ algorithm $O(n^2 \log n)$ times and we will return the best cut we got. Since each repetition has probability of success at least $\frac{c}{n^2}$ for some constant c, our new algorithm has probability of failure at most $(1 - \frac{c}{n^2})^{c' \cdot n^2 \log n} < (\frac{1}{e})^{\frac{c'}{c} \log n} < \frac{1}{n}$ for an appropriate choice of constants c, c'. So, the algorithm succeeds with probability at least $1 - \frac{1}{n}$.

The running time of the new algorithm is: $O(m \log m \cdot n^2 \log n)$

Quiz 1-5: Well Spaced Triples

Suppose we are given a bit string $B[1 \dots n]$. A triple of distinct indices $1 \le i < j < k \le n$ is called a *well-spaced triple* in B if $B[i] = B[j] = B[k] = 1$ and $k - j = j - i$.

(a) **[5 points]** Describe an algorithm to determine whether B has a well-spaced triple in $O(n^2)$ time.

Solution.

The algorithm is the following:

```
WELL-SPACED-TRIPLE($B[1 \dots n]$):
    For each $i \in \{1, ..., n\}$
       For each $k \in \{1, ..., n\}$
           If $i + k$ is even AND $B[i] = 1$ AND $B[k] = 1$ AND $B[\frac{i+k}{2}] = 1$
              return yes

    return no
```

The running time of the above algorithm is $O(n^2)$, since we need to check all the pairs of indices (i, k).

By the definition of well-spaced triples, B has one if and only if there exists a pair (i, k) such that $i + k$ is even AND $B[i] = 1$ AND $B[k] = 1$ AND $B[\frac{i+k}{2}] = 1$. So, the algorithm returns **yes** if and only if B has a well-spaced triple.

(b) **[15 points]** Describe an algorithm to determine the number of well-spaced triples in B in $O(n \log n)$ time.

(**Hint:** Use FFT)

Solution. The algorithm is as follows: Let $B[i]$ represent the i^{th} entry of the bit string B.

```
FAST-WELL-SPACED-TRIPLE($B[1 \dots n]$):
    Construct the polynomial $P(x) = \sum_i B[i]x^i$.
    Use FFT to compute $Q(x) = P(x) \cdot P(x)$.
    Let us denote the coefficients of $Q(x)$ by $(q_0, q_1, \dots, q_{2n})$.
    For each $j = 0, 1, ..., n$
        If $B[j] = 1$, $result = result + (q_{2j} - 1)$
    return result/2
```

The running time of the above algorithm is $O(n \log n)$. We perform FFT to multiply two polynomials of degree n, which needs $O(n \log n)$ time and then we perform $O(n)$ comparisons and operations.

Since $P(x)$ has degree n, we know that $Q(x)$ has degree less than or equal to $2n$. The $2j$-th coefficient of $Q(x)$ can be written as $q_{2j} = \sum_{i+k=2j} B[i]B[k]$. So, q_{2j} is not equal to zero if and only if there exists an (i, k) such that $B[i] = B[k] = 1$. However, this condition does not guarantee that B has a well-spaced triple. That is because we have not enforced the condition $i < k$. In fact, if $B[j] = 1$, then $B[j]B[j] = 1$, which means that $q_{2j} \geq 1$. However, if q_{2j} is strictly greater than one, then we know that there is a pair (i, k) such that i, j, k are distinct and $B[i] = B[k] = 1$. Also, if (i, k) satisfies $B[i] = B[k] = 1$, then the same holds for (k, i) because of symmetry. So, the value $(q_{2j} - 1)/2$ gives us exactly how many distinct pairs (i, k) with $i < k$ exist such that $B[i] = B[k] = 1$. We only want to consider such pairs if $B[j] = 1$ which is handled by the check in the final if condition in the code.

Quiz 1-6: Bin Packing Let X_i, $1 \le i \le n$ be independent and identically distributed random variables following the distribution

$$\mathbb{P}\left(X_i = \frac{1}{2^k}\right) = \frac{1}{2^k} \quad \textit{for} \quad 1 \le k \le \infty$$

Each X_i represents the size of the item i. Our task is to pack the n items in the least possible number of bins of size 1. Let Z be the random variable that represents the total number of bins that we have to use if we pack the n items optimally. Consider the following algorithm for performing the packing:

```
BIN-PACKING($x_1, ..., x_n$):
    Sort $(x_1, ..., x_n)$ by size in decreasing order. Let $(z_1, ..., z_n)$ be the sorted array of items
    Create a bin $b_1$
    For each $i \in \{1, ..., n\}$
        If $z_i$ fits in some of the bins created so far
            Choose one of them arbitrarily and add $z_i$ to it
        Else create a new bin

    return the number of created bins.
```

(a) **[8 points]** Prove that the above packing algorithm is optimal for this problem.

Solution. We will first prove the following claim:

Claim: At any step of the algorithm there is at most one non-full created bin.

proof: We prove the claim by induction on the item being placed on the bins:

- After we place the first item, there is only one created bin. So, the base case holds.
- Assume that we have at most one non-full created bin after we place item k.
- Then, for item $k+1$ there are two cases. The first case is that after the placement of item k, all the bins are full. Then, we create a new bin for item $k+1$ and this bin is the only non-full created bin. The second case is that the bin where item k is placed, b, is not full and all the other bins are full. We need to show that item $k+1$ fits in this bin. We know that the sizes of items are in decreasing order and that all the sizes are of the form 2^{-i}. So, all the item that are in bin b have sizes that are multiples of the size of item k. Assume that the size of item k is 2^{-s}. Then, the free space of bin b is of the form $1 - \lambda 2^{-s} > 0$ for some $\lambda \in \mathbb{N}$. Since λ is integer, it must be the case that $\lambda \in \{1, \ldots, 2^s - 1\}$. Thus, $1 - \lambda 2^{-s} \ge 2^{-s}$. Since the size of item $k+1$ is smaller than 2^{-s}, it fits in bin b.

□

From the claim above, we know that when the algorithm finishes, there will be at most one non-full created bin. If this algorithm was not optimal, there would be a solution with fewer bins. But, this cannot happen, since all the bins but one are already full.

(b) **[7 points]** Prove
$$\mathbb{E}[Z] \le \frac{n}{3} + 1$$

Solution.
By the claim of part (a), we know that
$$Z = \lceil \sum_{i=1}^{n} x_i \rceil$$
where x_i is the size of item i.
Therefore,
$$\mathbb{E}[Z] = \lceil \sum_{i=1}^{n} \mathbb{E}[x_i] \rceil \le n\mathbb{E}[x_1] + 1$$
since all the item are chosen independently from the same distribution.
Also,
$$\mathbb{E}[x_1] = \sum_{s=1}^{\infty} \frac{1}{2^s}\frac{1}{2^s} = \sum_{s=1}^{\infty} \frac{1}{4^s} = \frac{1}{3}$$
Therefore,
$$\mathbb{E}[Z] \le \frac{n}{3} + 1$$

(c) **[5 points]** Prove that
$$\mathbb{P}\left(Z \ge \frac{n}{3} + \sqrt{n}\right) \le \frac{1}{e}$$

Solution. Let us define X as $\sum_{i=1}^{n} x_i + 1$. Then, X is the sum of n independent random variables with value in $[0, 1]$ and $\mathbb{E}[X] = \frac{n}{3} + 1$. So, by Chernoff bound:
$$\mathbb{P}\left(X \ge (1+\beta)\left(\frac{n}{3} + 1\right)\right) \le e^{-\beta^2(\frac{n}{3}+1)/2}$$
For $\beta = \frac{\sqrt{n}-1}{1+\frac{n}{3}}$ and appropriately large n ($n > 64$), we get:
$$\mathbb{P}\left(X \ge \frac{n}{3} + \sqrt{n}\right) \le e^{-\frac{(\sqrt{n}-1)^2}{2(\frac{n}{3}+1)}} \le \frac{1}{e}$$
since for $n > 64$, $(\sqrt{n} - 1)^2 \ge \frac{3n}{4}$ and $2\left(\frac{n}{3} + 1\right) \le \frac{3n}{4}$.
Now, we remark that since $Z \ge \alpha \Rightarrow X \ge \alpha$, we have:
$$\mathbb{P}(X \ge \alpha) \ge \mathbb{P}(Z \ge \alpha)$$
for every $\alpha \in \mathbb{R}$.
So, we get that
$$\mathbb{P}\left(Z \ge \frac{n}{3} + \sqrt{n}\right) \le \frac{1}{e}.$$

Scratch page

*Name*____

Scratch page

Scratch page