Practice Problems for Quiz 1

- The following is a compilation of relevant problems from previous semesters.
- Do not waste time deriving facts that we have studied. Just cite results from class.
- When we ask you to *give an algorithm*, 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.
- **Good luck!**

Problem-1: True or False

Please circle **T** or **F** for the following. *No justification is needed (nor will be considered).*

(a) **T** **F**

Suppose that you modify the median algorithm so that instead of splitting into groups of 5 and using the median of the medians of these groups as a pivoting element, you split into groups of size 11. Then the running time is still $O(n)$.

Solution: True. There are $\geq n/22$ medians less than or equal to the medians of the medians. Thus, there are at least $6n/22$ elements less than the median of medians and also at least $6n/22$ elements greater than the median of medians. Thus our recurance will be:

$$T(n) = T(n/11) + T(16n/22) + O(n)$$

Note that $1/11 + 8/11 = 9/11 < 1$. Thus, $T(n) = O(n)$.

(b) **T** **F**

The running time of a divide-and-conquer algorithm on inputs of size n, $T(n)$, satisfies the recurrence $T(n) = 2^n \cdot T(n/2)^2$ with boundary condition $T(1) = 2$. This recurrence satisfies $T(n) = 2^{\Omega(n)}$.

Solution: True. Let $g(n) = \log_2(T(n))$.

$$T(n) = 2^n T(n/2)^2$$
$$g(n) = n + 2g(n/2)$$
$$T(1) = 2$$
$$g(1) = 1$$

This solves to $g(n) = n\log(n)$. Hence $T(n) = 2^{n\log(n)}$. Note that $n2^{\Omega(n)} = 2^{\Omega(n)+\log(n)} = 2^{\Omega(n)}$. So, if $T(n) = 2^{\Omega(n)}$ we satisfy this property. Hence

$$T(n) = 2^{n\log(n)} = 2^{\Omega(n)}$$

(c) **T** **F**

In a disjoint-set data structure where we do union by size and use path compression, when we merge disjoint sets A and B, all of the elements of the smaller of the two sets will end up with a direct pointer to the representative of the new joined set.

Solution: False. Find set will cause elements on which find set is run to point to the overall representative. A union operation only calls find set operations on one element in the smaller set. However, this does not update every single elements pointer.

(d) **T** **F**

Weighted Interval Scheduling* can be solved with a greedy algorithm that: (i) sorts the intervals with respect to weight; and then (ii) considers the intervals from highest to lowest weight, adding an interval to the schedule as long as it does not overlap with previously selected intervals.

**Weighted Interval Scheduling was discussed in Lecture 1: as input we are given n 3-tuples (t_i^s, t_i^e, w_i) where t_i^s is the leftmost point of interval i, t_i^e is the rightmost point of interval i, and w_i is the weight of the interval. The goal is to select a subset of non-overlapping intervals whose sum of weights is as large as possible.*

Solution: False. There are many possible counterexamples. Any case where a high weight interval is overlapped by multiple lower weight intervals that could instead all be selected is sufficient. For instance, $(0, 10, 7), (2, 3, 6), (4, 5, 6)$ is one such counterexample.

(e) **T** **F**

We consider an implementation of the ACCESS operation for self-organizing lists which, whenever it accesses an element x of the list, transposes x with its predecessor in the list (if it exists). This heuristic is 2-competitive.

Solution: False. Let elements x and y be the two elements at the end of the list. If we repeatedly access x and y (alternating between the two, starting with the later one), then each access costs $O(n)$. However, if we use the move-to-front (or something more optimal), we could have x and y at the beginning of the list, after which it only costs $O(1)$ per access.

(f) **T** **F**

Suppose algorithm $\mathcal{A}$ has two steps, and $\mathcal{A}$ succeeds if both the steps succeed. If the two steps succeed with probability p_1 and p_2 respectively, then $\mathcal{A}$ succeeds with probability $p_1 p_2$.

Solution: False. Unless the two steps are independent.

(g) **T** **F**

We are given an array $A[1 \ldots n]$. 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.

(h) **T** **F**

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:

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

(i) **T** **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,

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

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 $S \subseteq \{1, \ldots, k\}$

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

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

Therefore,

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

(j) **T** **F**

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 = $0.9999c_1 n + 0.0001c_2 n^2 = \Omega(n^2)$.

Problem-2: Democratic Feudal System

A kingdom has a single monarch. The monarch has three loyal dukes who reign over three parts of his kingdom. Each of these dukes in turn has three loyal advisors, who each have three loyal subjects, and so on, down to the peasants. This feudal society can be modeled by a tree with branching factor 3 and height h. Voting day has now arrived, and each peasant votes either for or against some public policy. This is modeled by assigning a value of 0 or 1 to each of the leaves of the tree. Any intermediate officer votes according to the majority of his immediate constituents. That is, the value assigned to any non-leaf node of the tree is the majority of the values of its immediate children, whose own value is the majority of the values of their own immediate children, and so on so forth down to the leaves of the tree. If the value thus assigned to the root of the tree (the king's node) is 1, the policy passes; otherwise, the policy fails. Since collecting votes is expensive, the king would like to determine an efficient algorithm to evaluate the final vote.

(a) Show that in the worst case, a deterministic algorithm will have to query the vote of all of the $n = 3^h$ leaves in order to determine the value assigned to the root. That is, argue that it is impossible for a deterministic algorithm to correctly determine the value at the root of the tree without knowing every single leaf's value.

Solution: An adversary can set leaf values to either 0 or 1 as the algorithm queries them in order to force all 3^h leaves to be evaluated. Essentially, for each internal node x, the first two children of x that algorithm queries evaluate to different values, so that third child must be evaluated as well.

(b) Design a randomized algorithm for computing the majority of three bits b_1, b_2, b_3, such that, no matter what the values of b_1, b_2, b_3 are, with probability at least $\frac{1}{3}$, the algorithm does not inspect one of the values.

Solution: The algorithm inspects the bits according to a random order. If the first two bits have the same value, that value is the majority, and the algorithm does not inspect the third bit. Otherwise, the algorithm inspects all bits. The reason that the third bit is not inspected with probability at least $\frac{1}{3}$ is that at least two out of the three bits have the same value, and there is probability at least $\frac{1}{3}$ that two identical bits come first.

(c) Design a randomized algorithm that evaluates the final vote correctly with probability 1, but only inspects the value of n^α leaves in expectation, for some constant $\alpha < 1$ (independent of n).

Solution: Let $T(t)$ be the number of leaves queried for a node at height t. Then either

$$E[T(t)] = \frac{1}{3} \cdot 2E[T(t-1)] + \frac{2}{3} \cdot 3E[T(t-1)] = \frac{8}{3}T(t-1)$$

if exactly two children have the same value, or

$$E[T(t)] = 2E[T(t-1)]$$

if all three have the same value. In either case, we can bound this as

$$E[T(t)] \leq \frac{8}{3}E[T(t-1)]$$

and since $T(0) = 1$, this solves to $E[T(t)] = (\frac{8}{3})^t$. Thus

$$E[T(h)] \leq (\frac{8}{3})^{\log_3 n} = n^{\log_3 8/3} \leq n^{0.8928}$$

(d) Show that there exists some constant $c \geq 1$ (independent of n) such that, with probability at least 0.99, your algorithm from (c) inspects at most cn^α leaves, where α is the constant you derived in (c).

Solution: By Markov inequality.

Problem-3: Fast Fourier Transform (FFT)

Ben Bitdiddle is trying to multiply two polynomials using the FFT. In his trivial example, Ben sets $a = (0, 1)$ and $b = (0, 1)$, both representing $0 + x$, and calculates:

$$A = \mathcal{F}(a) = B = \mathcal{F}(b) = (1, -1),$$

$$C = A * B = (1, 1),$$
$$c = \mathcal{F}^{-1}(C) = (1, 0).$$

So c represents $1 + 0 \cdot x$, which is clearly wrong. Point out Ben's mistake in one sentence; no calculation needed. (Ben swears he has calculated FFT $\mathcal{F}$ and inverse FFT $\mathcal{F}^{-1}$ correctly.)

Solution: The resulting polynomial is of degree 2, so Ben need to pad a and b with zeroes (or Ben needs at least 3 samples to do FFT).

Here is the correct calculation (not required in the solution). Let $a = b = (0, 1, 0, 0)$; then,

$$A = \mathcal{F}(a) = B = \mathcal{F}(b) = (1, -i, -1, i)$$

$$C = A * B = (1, -1, 1, -1)$$
$$c = \mathcal{F}^{-1}(C) = (0, 0, 1, 0)$$

which represents x^2.

It is also okay to set $a = b = (0, 1, 0)$.

Problem-4: Amortized Analysis

Design a data structure to maintain a set S of n distinct integers that supports the following two operations:

(a) INSERT(x, S): insert integer x into S.

(b) REMOVE-BOTTOM-HALF(S): remove the smallest $\lceil \frac{n}{2} \rceil$ integers from S.

Describe your algorithm and give the worse-case time complexity of the two operations. Then carry out an amortized analysis to make INSERT(x, S) run in amortized $O(1)$ time, and REMOVE-BOTTOM-HALF(S) run in amortized 0 time.

Solution: Use a singly linked list to store those integers. To implement INSERT(x, S), we append the new integer to the end of the linked list. This takes $\Theta(1)$ time. To implement REMOVE-BOTTOM-HALF(S), we use the median finding algorithm taught in class to find the median number, and then go through the list again to delete all the numbers smaller or equal than the median. This takes $\Theta(n)$ time.

Suppose the runtime of REMOVE-BOTTOM-HALF(S) is bounded by cn for some constant c. For amortized analysis, use $\Phi = 2cn$ as our potential function. Therefore, the amortized cost of an insertion is $1 + \Delta\Phi = 1 + 2c = \Theta(1)$. The amortized cost of REMOVE-BOTTOM-HALF(S) is $cn + \Delta\Phi = cn + (-2c \times \frac{n}{2}) = 0$.

Problem-5: Verifying Polynomial Multiplication

This problem will explore how to check the product of two polynomials. Specifically, we are given three polynomials:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0,$$

$$q(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_0,$$

$$r(x) = c_{2n} x^{2n} + c_{2n-1} x^{2n-1} + \cdots + c_0.$$

We want to check whether $p(x) \cdot q(x) = r(x)$ (for all values x). Via FFT, we could simply compute $p(x) \cdot q(x)$ and check in $O(n \log n)$ time. Instead, we aim to achieve $O(n)$ time via randomization.

(a) Describe an $O(n)$-time randomized algorithm for testing whether $p(x) \cdot q(x) = r(x)$ that satisfies the following properties:

i. If the two sides are equal, the algorithm outputs YES.

ii. If the two sides are unequal, the algorithm outputs NO with probability at least $\frac{1}{2}$.

Solution: Pick a value $a \in [1, 4n]$, and check whether $p(a)q(a) = r(a)$. The algorithm outputs YES if the two sides are equal, and NO otherwise. It takes $O(n)$ time to evaluate the three polynomials of degree $O(n)$. Thus the overall running time of the algorithm is $O(n)$.

(b) Prove that your algorithm satisfies Property **Problem-5:**(a)i.

Solution: If $p(x) \cdot q(x) = r(x)$, then both sides will evaluate to the same thing for any input.

(c) Prove that your algorithm satisfies Property **Problem-5:**(a)ii.

Hint: Recall the Fundamental Theorem of Algebra: A degree-d polynomial has (at most) d roots.

Solution: $s(x) = r(x) - p(x) \cdot q(x)$ is a degree-$2n$ polynomial, and thus has at most $2n$ roots. Then

$$\Pr\{s(a) = 0\} \leq \frac{2n}{4n} = \frac{1}{2}$$

since a was picked from a set of size $4n$.

(d) Design a randomized algorithm to check whether $p(x) \cdot q(x) = r(x)$ that is correct with probability at least $1 - \epsilon$. Analyze your algorithm in terms of n and $1/\epsilon$.

Solution: We run part a m times, and output YES if and only if all answers output YES. In other words, we amplify the probability of success via repetition.

Our test works with probability $\geq 1 - \left(\frac{1}{2}\right)^m$. Thus we need

$$\left(\frac{1}{2}\right)^m \leq \epsilon$$

$$\Rightarrow m \geq \log \frac{1}{\epsilon}$$

Problem-6: Median of two sorted arrays

Finding the median of a sorted array is easy: return the middle element. But what if you are given two sorted arrays A and B, of size m and n respectively, and you want to find the median of all the numbers in A and B? You may assume that A and B are disjoint.

(a) Give a naïve algorithm running in $\Theta(m+n)$ time.

Solution: Merge the two sorted arrays (which takes $O(m+n)$ time) and find the median using linear-time selection.

(b) If $m = n$, give an algorithm that runs in $\Theta(\lg n)$ time.

Solution: Pick the median m_1 for A and median m_2 for B. If $m_1 = m_2$, return m_1. If $m_1 > m_2$, remove the second half of A and the first half of B. Then we get two subarrays with size $n/2$. Repeat until both arrays are smaller than a constant. $m_1 < m_2$ is symmetric.

(c) Give an algorithm that runs in $O(\lg(\min\{m,n\}))$ time, for any m and n.

Solution: Without loss of generality, assume $|A| = m > n = |B|$. We can safely remove elements $A[0 : \frac{m-n}{2}]$ and $A[\frac{m+n}{2} : m-1]$ because none of these elements can be the median of $A+B$. After this process, we get two arrays of size approximately n. Then we can run part (b). The complexity is $\Theta(\lg(\min\{m,n\}))$.

Problem-7: Forgetful Forrest

Prof. Forrest Gump is very forgetful, so he uses automatic calendar reminders for his appointments. For each reminder he receives for an event, he has a 50% chance of actually remembering the event (decided by an independent coin flip).

(a) Suppose we send Forrest k reminders for each of n events. What is the expected number of appointments Forrest will remember? Give your answer in terms of k and n.

Solution: These are all independent events. So linearity of expectation applies. Each given event has been remembered with probability $1-2^{-k}$. So in expectation $n(1-2^{-k})$ appointments are remembered.

(b) Suppose we send Forrest k reminders for a *single* event. How should we set k with respect to n so that Forrest will remember the event with high probability, i.e., $1-1/n^{\alpha}$?

Solution: This problem is equivalent to how many times we must flip a coin to get a head with high probability. The probability of k tails in a row is $1/2^k$. Thus exactly $\alpha \lg n$ coin flips suffice.

(c) Suppose we send Forrest k reminders for each of n events. How should we set k with respect to n so that Forrest will remember *all* the events with high probability, i.e., $1 - 1/n^{\alpha}$?

Solution: We must send at least $k = \Omega(\lg n)$ reminders, because we needed this many reminders to remember one event with high probability.

If we send $k = (\alpha + 1) \lg n$ reminders, then each event is remembered with probability $1-1/n^{\alpha+1}$. By a union bound, we know that all events are remembered with probability $1 - 1/n^{\alpha}$. So, the number of reminders needed is $k = O(\lg n)$.

Problem-8: k-Sum Subset

Suppose we have an array A with n elements, each of which is an integer in the range of $[0, \ldots, 50n]$. Give an $O(n \log n)$ algorithm that when given an integer $0 \leq t \leq 5 \cdot 50n$, determines whether there exist at most 5 (not necessarily distinct) elements of A which sum to t. That is, output "YES" if there exist $i_1, \ldots, i_k$ with $k \leq 5$ such that $A[i_1]+\cdots+A[i_k] = t$ and output "NO" otherwise.

Solution: Construct a polynomial $p(x)$ where the exponents of the polynomial correspond to elements of array A as

$$p(x) = x^{A[0]} + x^{A[1]} + ... + x^{A[n-1]}$$

When we raise $p(x)$ to some power, we get a resulting polynomial that has a sum of terms, each of which is some variation of the form $x^{A[i]} * x^{A[j]+\cdots} = x^{A[i]+A[j]+\cdots}$, where the number of terms in the sum is equal to the power we raised the polynomial to. For example, for $p^2(x)$, each term in the polynomial appears as $x^{A[i]+A[j]}$, and for $p^3(x)$, each term appears as $x^{A[i]+A[j]+A[k]}$. Therefore, we just need to check whether x^t appears in any power z of $P(x)$ for $0 \leq z \leq 5$, to determine whether there exist at most 5 elements of A that sum to t.

Algorithm:
while z $\leq$ 5:

1. Construct $p(x) = x^{A[0]} + x^{A[1]} + ... + x^{A[n-1]}$.
2. Compute $p^z(x)$ via FFT by repeated multiplications of $p(x)$. (You can do this step by multiplying the polynomial from the previous iteration with $p(x)$ again.
3. Scan through the resulting polynomial and check whether x^s is a term.

Analysis:
Constructing the polynomial $p(x)$ takes $O(n)$ time, as each element of A is in the range $[0, \ldots, 50n]$, so $p(x)$ is a degree of at most $50n$. Each polynomial multiplication using FFT takes time $O(n \log n)$, and since we're only doing at most 5 multiplications, the overall cost of this step is still $O(n \log n)$. Finally, scanning through $p^z(x)$ takes $O(n)$, because the degree for each scan is bounded by a constant ($250n$ max for $p^5(x)$). Therefore, combining all of this together, the overall run time of this algorithm is $O(n \log n)$.

Problem-9: 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) 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)

```
while s ≠ Null and s.val < x do
    s = s.prev
end while
s.next = Null
Allocate memory for a new node a
a.val = x
a.prev = s
s.next = a
s = a
```

Algorithm 2 POP()

```
if s =Null then
    return Null
else
    x = s.val
    s = s.prev
    return x
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

$$c_i \leq 2a - at_i$$

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

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

Thus c'_i's are $O(1)$. Since we have

$$\sum_{i=1}^{n} c_i = \Phi(S_0) - \Phi(S_n) + \sum_{i=1}^{n} c'_i \leq 2an$$

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) 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 $S = \{s_1, s_2, \ldots, s_j\}$ 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 $i \in \{1, \ldots, j\}$, $view(A[s_i]) = s_i - s_{i+1} - 1$ (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:

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

  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.

Problem-10: Find the Pairs

Consider a set U of $2n$ distinct balls numbered from $1 \dots 2n$ 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 $\{1, \dots, n\}$ 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 $S = \{1, \dots, n\}$. Partition S into two subsets S_1 and S_2 of the size $\left\lceil \frac{|S|}{2} \right\rceil$ and $\left\lfloor \frac{|S|}{2} \right\rfloor$. 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

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

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 $S_1 = \{1, \dots, n\}, S_2 = \{n + 1, \dots, 2n\}$. 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.

Problem-11: Bin Packing

Let $X_i, 1 \leq i \leq n$ be independent and identically distributed random variables following the distribution

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

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) 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, $1 - \lambda 2^{-s} \geq 2^{-s}$. 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) Prove that

$$\mathbb{E}[Z] \leq \frac{n}{3} + 1$$

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,

$$\mathbb{E}[Z] = \lceil \sum_{i=1}^{n} \mathbb{E}[x_i] \rceil \leq n\mathbb{E}[x_1] + 1$$

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,

$$\mathbb{E}[Z] \leq \frac{n}{3} + 1$$

(c) Prove that

$$\mathbb{P}\left(Z \geq \frac{n}{3} + \sqrt{n}\right) \leq \frac{1}{e}$$

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:

$$\mathbb{P}\left(X \geq (1+\beta)\left(\frac{n}{3} + 1\right)\right) \leq e^{-\beta^2(\frac{n}{3}+1)/2}$$

For $\beta = \frac{\sqrt{n}-1}{1+\frac{n}{3}}$ and appropriately large n ($n > 64$), we get:

$$\mathbb{P}\left(X \geq \frac{n}{3} + \sqrt{n}\right) \leq e^{-\frac{(\sqrt{n}-1)^2}{2(\frac{n}{3}+1)}} \leq \frac{1}{e}$$

since for $n > 64$, $(\sqrt{n}-1)^2 \geq \frac{3n}{4}$ and $2\left(\frac{n}{3}+1\right) \leq \frac{3n}{4}$. Now, we remark that since $Z \geq \alpha \Rightarrow X \geq \alpha$, we have:

$$\mathbb{P}(X \geq \alpha) \geq \mathbb{P}(Z \geq \alpha)$$

for every $\alpha \in \mathbb{R}$. So, we get that

$$\mathbb{P}\left(Z \geq \frac{n}{3} + \sqrt{n}\right) \leq \frac{1}{e}.$$

Problem-12: Sampling without Erasures

Ben Bitdiddle wants to implement the Reservoir Sampling algorithm from the class. Recall that this algorithm samples a uniformly random element from the stream $x_1, \ldots, x_n$ while not knowing n ahead of time and storing only a single element of the stream at any given moment. To achieve this, one starts with $x = x_1$ and then, in each round i, one replaces the stored element x with the element x_i with probability $p_i = \frac{1}{i}$, independently at random.

Ben realized, however, that the memory on his computer is non-erasable and as a result, instead of replacing the stored element, he has to append that element to the list of already stored elements, for as many elements as he picks from the stream.

(a) Show that with probability at least $\frac{3}{4}$, Ben will end up storing $O(\log n)$ elements.
Note: Recall that $\sum_{j=1}^{k} 1/j = \Theta(\ln k)$.

Solution: Denote by X_i the event that item i is sampled (that is, $X_i = 1$ if i is sampled, 0 otherwise). Then, the number of stored elements is $X = \sum_i X_i$. By linearity of expectation

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} \Pr[X_i = 1] \\ &= \sum_{i=1}^{n} \frac{1}{i} \\ &\leq c \log n \end{aligned}$$

for some $c > 0$. Further, X is a non-negative random variable so we can apply Markov's inequality. Picking $a = 4c \log n$

$$\begin{aligned} \Pr[X > a] &\leq E[X]/a \\ &\leq 1/4. \end{aligned}$$

So, with probability greater than $3/4$, Ben will store less than $4c \log n = O(\log n)$ items.

(b) Assume now a different scenario. Ben knows the length of the stream n ahead of time, but he wants to avoid having to store any unnecessary elements. Specifically, he wants to have an algorithm that takes a pass through the stream and for each element x_i for

$i \in \{1, \ldots, n\}$ selects this element with some probability $\hat{p}_i$ as the uniformly randomly chosen element of the stream. From that point on, he ignores the rest of the stream.

What should the probabilities $\hat{p}_i$ be to ensure that this algorithm is correct? Justify your answer.

Solution: To get intuition, let's compute $\hat{p}_i$ for $i = 1, 2, 3$ "by hand". Clearly, $\hat{p}_1 = 1/n$ as otherwise item 1 wouldn't be picked with probability $1/n$.

Item 2 is picked if item 1 is not picked and item 2 is picked in its turn. This happens with probability $(1 - \hat{p}_1)\hat{p}_2 = (1 - 1/n)\hat{p}_2$ and we want this to be equal to $1/n$. This yields that $\hat{p}_2 = 1/(n-1)$. Item 3 is picked if items 1 and 2 are not picked and item 3 is picked in turn. This happens with probability p_3 where

$$\begin{aligned} p_3 &= (1 - \hat{p}_1)(1 - \hat{p}_2)\hat{p}_3 \\ &= \left(1 - \frac{1}{n}\right)\left(1 - \frac{1}{n-1}\right)\hat{p}_3 \\ &= \left(\frac{n-1}{n}\right)\left(\frac{n-2}{n-1}\right)\hat{p}_3 \end{aligned}$$

If we want p_3 to be $1/n$, we need $\hat{p}_3 = 1/(n-2)$.

Generalizing, we get $\hat{p}_i = \frac{1}{n-i+1}$ which can be proven to be correct by induction on i.