13 (100 PTS.) Not a sorting network.

You are given an array $\mathtt{A}[1 \ldots n]$, but you can not access it directly (or even read the values in it, or compare them directly). Instead, you are give some procedures that can access the array and do certain operations. Your task is to sort the array.

13.A. (30 PTS.) You are given a procedure **sortBlock**(i), which sorts (in increasing order and in place), all the elements in $A[i \ldots, i+u]$, where $u \geq 1$ is a prespecified fixed parameter (i.e., u is a known fixed number between 1 and n, but it is not under your control). For example, for the following array:

2.71828	18284	59045	23536	02874	7.1352	66249
1	2	3	4	5	6	7

With $u = 3$, the call **sortBlock**(2) would result in:

2.71828	02874	18284	23536	59045	7.1352	66249
1	2	3	4	5	6	7

Describe an algorithm, that uses $O\big((n/u)^2\big)$ calls to **sortBlock**, and sorts the array.
What is the running time of your algorithm if calling **sortBlock** takes $O(1)$ time?

Solution:

The idea is to simulate insertion sort. Let $\Delta = \lfloor (u+1)/2 \rfloor$.

```
for i = ⌊n/Δ⌋, ..., 0 do
    for j = 0, ..., j do
        s ← min(1 + jΔ, n − u)
        sortBlock(A[s, ..., s + u])
```

The bound on the number of calls to **sortBlock** is easy, since it is $O((n/\Delta)^2) = O(n^2/u^2)$.

As for correctness, it is easy to argue (by easy but maybe boring induction) that in the end of the tth outer iteration, the last $t\Delta$ elements of the array contains the $t\Delta$ largest elements of the array. Since these elements are in the right order, it follows that in the end of the execution of the algorithm it would be sorting all the elements correctly.

13.B. (50 PTS.) Congratulations! You just got a better sorting primitives **bMerge**, **copy**, and a work array $W[1, \ldots, n]$.

(i) **copy** can copy any block of length at most $u+1$ between the two arrays (or inside them).
(ii) **bMerge** is weirder. It takes two blocks L and U (both with at most $u+1$ elements), treat them as a single block, sort the unified block, and writes the smaller $|L|$ elements (in sorted order) to L, and the other (larger) $|U|$ elements in sorted order to U (the two blocks do not have to be of the same length).

For example, for

2.71828	18284	59045	23536	02874	7.1352	66249
1	2	3	4	5	6	7

With $u = 1$, the call **bMerge**$(A[2 \ldots 3], A[6 \ldots 7])$ would result in:

2.71828	7.1352	18284	23536	02874	59045	66249
1	2	3	4	5	6	7

bMerge also returns the number of elements in original block L that are still in L after the operation is completed. In this example, since 18284 was in L before the operation, and it is in L after the operation is completed, the returned value would be 1.

Note that the blocks given to **bMerge** can not overlap.

Assume that $A[1 \ldots n/2]$ and $A[n/2 + 1, \ldots n]$ are already sorted (n is even). Describe an algorithm that performs a minimal total number of calls to **sortBlock**, **copy** and **bMerge**, and sorts the array A. What is the running time of your algorithm if calling **sortBlock**, **copy** and **bMerge** takes $O(1)$ time? (Prove your bound.)

Solution:

The idea is of course to implement merge sort. To this end, we copy the prefix, of size u, of the two parts being merged into the working array, we sort them using a call to **bMerge**. We then consider the first u elements in the sorted block to be the smallest u elements in the parts of the arrays yet to be sorted. We add these u elements to the output, and step forward the two input arrays, according to how many elements were used in the output block. Here is the pseudo-code – we use B here to make the code clearer (but in reality it would just be a different part of A).

The code handles carefully the case where the number of remaining elements is smaller than u, thus the code looks a bit messier than desired.

```
mSort( A[1,...,N], B[1...,M] ):
  a ← 1
  b ← 1
  o ← 1
  while (a ≤ N and b ≤ M) do
    a' = min(i + u, N)
    Δ_A = a' − a + 1
    b' = min(i + u, M)
    Δ_B = b' − b + 1
    if Δ_A ≥ Δ_B then
      o' = o + Δ_A
      copy( A[a,...,a'] ⇒ W[o,...,o + Δ_A − 1])
      copy( B[b,...,b'] ⇒ W[o',...,o' + Δ_B − 1])
      λ ← bMerge(W[o,...,o + Δ_A − 1], W[o',...,o' + Δ_B − 1])
      λ' ← Δ_A − λ
      a ← a + λ
      b ← b + λ'
      o ← o + Δ_A
    else
      OtherCase
    if Δ_A < u and Δ_B < u then break loop
  Using ⌈(N + M)/u⌉ calls to copy, copy the sorted array back into A and B.
```

OtherCase:
$o' = o + \Delta_B$
copy($B[b, \ldots, b'] \Rightarrow W[o, \ldots, o + \Delta_B - 1]$)
copy($A[a, \ldots, a'] \Rightarrow W[o', \ldots, o' + \Delta_A - 1]$)
$\lambda \leftarrow$ **bMerge**$(W[o, \ldots, o + \Delta_B - 1], W[o', \ldots, o' + \Delta_A - 1])$
$\lambda' \leftarrow \Delta_B - \lambda$
$b \leftarrow b + \lambda$
$a \leftarrow a + \lambda'$
$o \leftarrow o + \Delta_B$

Clearly, in each outer iteration, the algorithm add at least u elements to the output array. In the last iteration, the two remaining parts are small enough, that they are now sorted correctly, an the algorithm can bail out.

As such, the running time of the algorithm

$$\textbf{mSort}\big(A[1 \ldots n/2],\ A[n/2 + 1, \ldots n]\big)$$

is $O(n/u + 1)$, and this also bounds the number of calls to the primitives provided.

13.C. (20 PTS.) Building on (B) and expanding on it, describe a sorting algorithm using these primitives that sort the given array A (that is initially not sorted). What is the running time of your algorithm if calling **sortBlock**, **copy** and **bMerge** takes $O(1)$ time? Naturally, the faster the better (Prove your bound).

Solution:

Just implement merge sort. If the subarray to be sorted is of size smaller than u, then sort directly using **sortBlock**, otherwise, break it into two subarrays, sort them recursively, and merge them using the procedure from (B). The running time is

$$T(n) = 2T(n/2) + O(n/u + 1),$$

where $T(n) = O(1)$ if $n \leq u$. As such, the solution to this recurrence is $O((n/u + 1)(1 + \log(n/u)))$ (which is essentially $O((n/u) \log(n/u))$), and this bounds the running time and number of calls to the primitives.

14 (100 PTS.) Not a sorting question.

Consider an array $A[0 \ldots n - 1]$ with n distinct elements. Each element is an ℓ bit string representing a natural number between 0 and $2^\ell - 1$ for some $\ell > 0$. The only way to access any element of A is to use the function **FetchBit**(i, j) that returns the jth bit of $A[i]$ in $O(1)$ time.

14.A. (20 PTS.) Suppose $n = 2^\ell - 1$, i.e. exactly one of the ℓ-bit strings does not appear in A. Describe an algorithm to find the missing bit string in A using $\Theta(n \log n)$ calls to **FetchBit** without converting any of the strings to natural numbers.

Solution:

The idea is to simply count the number of '1's and '0's in the jth bit of all strings. If all strings where there, we would have $2^{\ell-1}$ ones and $2^{\ell-1}$ zeros. Whichever is less, is the missing bit. We will store the bits of the missing string in $S[0, ..., \ell-1]$.

```
for j = 0, ..., l − 1 do
    c = 0
    for i = 0, ..., n − 1 do
        c = c + FetchBit(i, j)
    if c < 2^{ℓ−1}
        S[j] = 1
    else
        S[j] = 0
```

The algorithm calles **FetchBit** $\ell \times n$ times and $\ell = \log n$. Hence, $O(n \log n)$ calls.

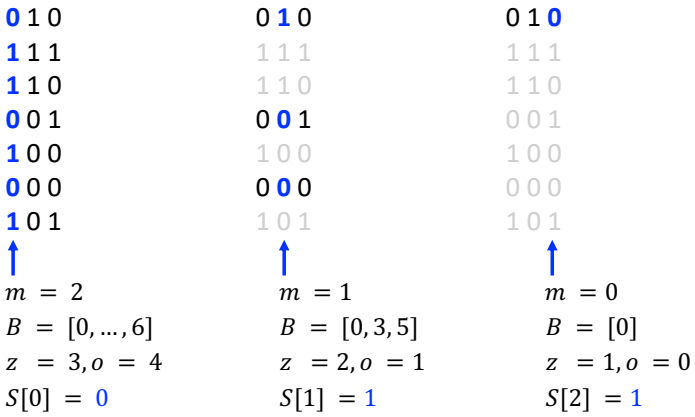
14.B. (40 PTS.) Suppose $n = 2^\ell - 1$. Describe an algorithm to find the missing bit string in A using only $O(n)$ calls to **FetchBit**.

Solution:

We will use the same the idea as the first part but we will keep track of the previous bit and rescursively search half of the array for the next missing bit.

```
findMissing(A[0, ..., n − 1], B[0, ..., 2^{m+1} − 2], S[0, ..., ℓ − 1], m)
  if m ≥ 0
      z = 0
      o = 0
      for i = 0, ..., 2^{m+1} − 2 do
          if FetchBit(B[i], m) = 1
              B_1[o] = B[i]
              o = o + 1
          else
              B_0[z] = B[i]
              z = z + 1
      if o < 2^m
          S[m] = 1
          findMissing(A, B_1, S, m − 1)
      else
          S[m] = 0
          findMissing(A, B_0, S, m − 1)
```

To find the missing bit string, we call the above algorithm with the following input **findMissing**$(A, [0, ..., n-1], S, \ell - 1)$. The below figure shows a run of the algorithm for $\ell = 3$ and A missing the string 011.



The number of calls to **FetchBit** is $2^{m+1}-1$ and it decreases by half with every recursive call starting from n in the first recursive call. Hence, the number of calls to **FetchBit** follows the following recurrence:

$$\begin{aligned} T(n) &= T(n/2)+n \\ &= T(n/4)+n/2+n \\ &= 1+2+4+\cdots+n/2+n \\ &= O(n) \end{aligned}$$

14.C. (40 PTS.) Suppose $n = 2^\ell - k$, i.e. exactly k of the $l-$bit strings do not appear in A. Describe an algorithm to find the k missing bit strings in A using only $O(n \log k)$ calls to **FetchBit**.

Solution:

We will again use the same idea. However, instead of recursing on one branch (either zeros or ones), we will recurse on both as long as there is a missing bit. We will use a $k \times \ell$ array S to store the missing strings and an array C to keep track of which missing strings each recursive call is looking searching for. To find the k missing strings, we call the below algorithm with the following input **findMissingk**$(A, [0, ..., n-1], S, [0, ..., k-1], \ell - 1)$. The figure below also shows a run of the algorithm for $\ell = 3$, $k = 3$, and A missing the strings 110, 011, 001.

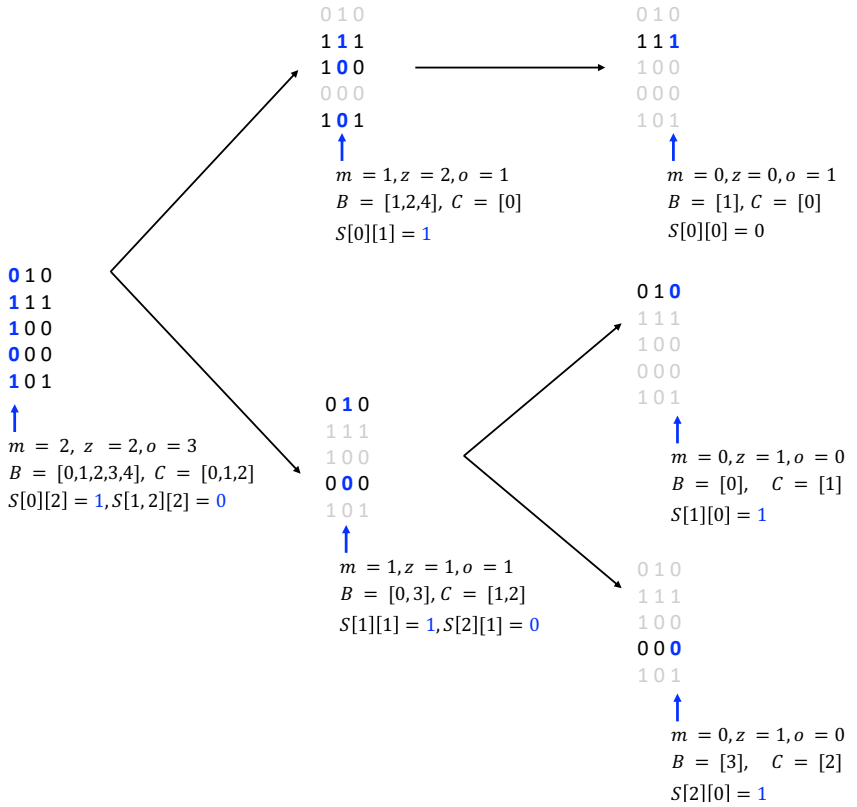
It might be tempting to assume that **findMissingk** recursively calls itself twice on half the input size leading to the following recurrence on the number of calls to **FetchBits**:

$$T(n) \leq 2T(n/2) + O(n)$$

which leads on an $O(n \log n)$ bound. However, this bound is not tight. Notice that the recursion tree for **findMissingk** will have at most k nodes at any level of the recursion. Hence, at level j, the total work is at most: $\min\{2^j, k\} \times n/(2^j)$. Thus,

$$\begin{aligned} T(n) &\leq \sum_{j=0}^{\ell-1} \min\{2^j, k\} \times n/(2^j) = \sum_{j=0}^{\log k} n + \sum_{j=\log k+1}^{\ell-1} kn/2^j = n \log k + (n-k) \\ &\leq O(n \log k) \end{aligned}$$

```
findMissingk(A[0, ..., n − 1], B[0, ..., 2^{m+1} − 1 − k], S, C[0, ..., k − 1], m)
  if m ≥ 0
    z = 0
    o = 0
    for i = 0, ..., 2^{m+1} − 1 − k do
      if FetchBit(B[i], m) = 1
        B_1[o] = B[i]
        o = o + 1
      else
        B_0[z] = B[i]
        z = z + 1
    k_1 = 2^m − o
    k_0 = 2^m − z
    if k_1 > 0
      for i = 0, ..., (k_1 − 1) do
        S[C[i]][m] = 1
        C_1[i] = C[i]
      findMissing(A, B_1, S, C_1, m − 1)
    if k_0 > 0
      for i = 0, ..., (k_0 − 1) do
        S[C[i + k_1]][m] = 0
        C_0[i] = C[i + k_1]
      findMissing(A, B_0, S, C_0, m − 1)
```



Alternative Proof:

We can write the two parameter recurrence for the number of calls to **FetchBits** as:

$$T(n,k) = \quad T(n/2,k_1) + T(n/2,k-k_1) + n$$

where k_1 is the number of strings in which the current missing bit is 1 and $0 \le k_1 \le k \le n$. Observe that $T(n,0) = 0$, $T(n,1) = O(n)$ (from part B), and $T(1,1) = O(1)$.

We will prove that $T(n,k) = O(n \log k)$ for $k \ge 1$ this by induction on k.

Base Case: $k = 1$

$T(n,1) = O(n) = O(n \log c)$ for some constant $c > 1$.

Inductive Hypothesis: Assume for any $1 \le k' < k$, $T(n,k') = O(n \log k')$

Proof:

$$T(n,k) = T(n/2,k_1) + T(n/2,k-k_1) + n$$

If $1 \le k_1 < k$, then we can apply by inductive hypothesis:

$$\begin{aligned} T(n,k) \le & \quad O(n\log(k_1)) + O(n\log(k-k_1)) + n \\ \le & \quad O(n\log(k_1(k-k_1))) \\ \le & \quad O(n\log(k^2)) \\ \le & \quad O(n\log k) \end{aligned}$$

If $k_1 = 0$ or $k_1 = k$, then $T(n,k) = T(n/2,k) + O(n)$. However, since the missing strings are distinct, there must exist a different bit that will trigger two recursive calls at some level of the recursion. Let $0 \le j \le \ell - \log k - 1$ be the first level that triggers two recursive calls. Then,

$$\begin{aligned} T(n,k) = & \quad n + n/2 + ... + n/2^j + T(n/2^{j+1}, k_1') + T(n/2^{j+1}, k-k_1') \\ \le & \quad O(n) + O(n/2^{j+1}\log k) \\ \le & \quad O(n\log k) \end{aligned}$$

15 (100 PTS.) Don't want to walk too much.

You are given a set of n distinct points on a line with x-coordinates $x_1, x_2, .. x_n$. The points are not sorted and their values are stored in an array X where $X[i] = x_i$. Each point is associated with a positive weight w_i such that $\sum w_i = 1$. The weights are also stored in an array W where $W[i] = w_i$. Our goal is to find x_j that minimizes the weighted distance given by: $\sum_i w_i |x_j - x_i|$.

15.A. (10 PTS.) (Easy.) Show that if all the weights are equal, x_j is the median of X.

Solution:

Since all the weights are equal minimizing the weigted distance is the same as minimizing: $p(x) = \sum_i |x - x_i|$. Let x_M be the median. Assume the point that minimizes $p(x)$ is $x_j \neq x_M$. Then, $p(x_M) - p(x_j) > 0$. Without loss of generality, assume $x_j > x_M$. Let $\Delta = x_j - x_M$ then with perseverance we can write:

$$\begin{aligned}
p(x_M) - p(x_j) = & \sum_{x_i \leq x_M} (x_M - x_i) + \sum_{x_i > x_M} (x_i - x_M) - \sum_{x_i \leq x_j} (x_j - x_i) - \sum_{x_i > x_j} (x_i - x_j) \\
= & -\sum_{x_i \leq x_M} \Delta + \sum_{x_i > x_j} \Delta + \sum_{x_M < x_i \leq x_j} (x_i - x_M) - \sum_{x_M < x_i \leq x_j} (x_j - x_i) \\
\leq & -\sum_{x_i \leq x_M} \Delta + \sum_{x_i > x_M} \Delta - \sum_{x_M < x_i \leq x_j} (x_j - x_i) \\
\leq & \; 0 - \sum_{x_M < x_i \leq x_j} (x_j - x_i) \\
< & \; 0
\end{aligned}$$

which is a contradiction! You can get a similar result by assuming $x_j < x_M$.

15.B. (20 PTS.) In general, show that x_j is the point that satisfies the following property:

$$\sum_{x_i < x_j} w_i < \frac{1}{2} \qquad \text{and} \qquad \sum_{x_i > x_j} w_i \leq \frac{1}{2}.$$

Solution:

We will first sort the points x_i and rearrange the weights accordingly. (This step is not necessary. However, it simplifies the notation in our proof.)

Let x_j be the point that minimizes the weighted distance. $p(x) = \sum_i w_i |x - x_i|$. Then, $p(x_j) - p(x_{j+1}) < 0$ and $x_{j+1} > x_j$.

$$\begin{aligned}
p(x_j) - p(x_{j+1}) = & \sum_i w_i |x_j - x_i| - \sum_i w_i |x_{j+1} - x_i| \\
= & \sum_{i<j} w_i (x_j - x_i) + \sum_{i \geq j} w_i (x_i - x_j) - \sum_{i < j+1} w_i (x_{j+1} - x_i) - \sum_{i \geq j+1} w_i (x_i - x_{j+1}) \\
= & \left(\sum_{i \geq j} w_i - \sum_{i<j} w_i \right) (x_{j+1} - x_j) < 0
\end{aligned}$$

$$\text{Thus,} \quad \sum_{i \geq j} w_i - \sum_{i<j} w_i < 0 \quad \text{and} \quad \sum_i w_i = 1 \Rightarrow \sum_{i>j} w_i \leq \frac{1}{2}$$

Similarly, $p(x_j) - p(x_{j-1}) > 0$ and $x_{j-1} < x_j$ which will lead to $\sum_{i<j} w_i < \frac{1}{2}$.

Alternative Solution:
$p(x)$ is continuous. We show that $\frac{dp(x)}{dx} > 0$ for $x > x_j$ and $\frac{dp(x)}{dx} < 0$ for $x < x_j$ to conclude that x_j is a global minimum among the x_i points.

15.C. (20 PTS.) Given, X and W, describe in few lines an algorithm to find x_j. What is the running time of your algorithm.

Solution:

The idea is to sort X and rearrange the elements of W accordingly. Then, sum the weights until we exceed $1/2$. The point at which we exceed $1/2$ is x_j. The runtime is $O(n \log n) + O(n) = O(n \log n)$.

```
Sort(X, W)
s ← 0
i ← 1
while s < 1/2 do
    s = s + W[i]
    i = i + 1
x_j ← X[i − 1]
```

15.D. (50 PTS.) Given, X and W, describe an algorithm to find x_j in $O(n)$ time. Prove the bound on the running time of your algorithm.

Solution:

The idea is to use binary search. We find the median using the linear time selection algorithm from class. We then compute the weights of all elements smaller than the median. If it is greater than $1/2$, we recurse on elements smaller than the median. If it is smaller than $1/2$, we recurse on elements larger than the median while keeping track of the sum of weights of points smaller than the median. We run the below algorithm with input WeightedMedian$(X, W, 0)$.

```
WeightedMedian(X[1, .., n], W[1, ..., n], t)
  if n = 1
      return X[1]
  else
      s ← 0, l ← 1, g ← 1
      y ← Select(X, ⌊(n + 1)/2⌋)
      for i = 1, ..., n do
          if X[i] < y
              s = s + W[i]
              X_<[l] = X[i], W_<[l] = W[i]
              l = l + 1
          else
              X_>[g] = X[i], W_>[g] = W[g]
              g = g + 1
      if t + s > 1/2
          WeightedMedian(X_<, W_<, t)
      else
          WeightedMedian(X_>, W_>, s)
```

The algorithm requires $O(n)$ to find the median and another $O(n)$ to divide all elements of X into $X_<$ and $X_>$. Then the algorithm recursively calls itself on $X_<$ or $X_>$ both of which are of size $n/2$. Hence, for some constant $c > 1$, the number of operations is:

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(n/4) + cn/2 + cn \\ &= T(n/2^k) + cn \sum_{i=0}^{k-1} 1/2^i \\ &= T(1) + cn \sum_{i=0}^{\log n - 1} 1/2^i \\ &= O(n) \end{aligned}$$