HW 5

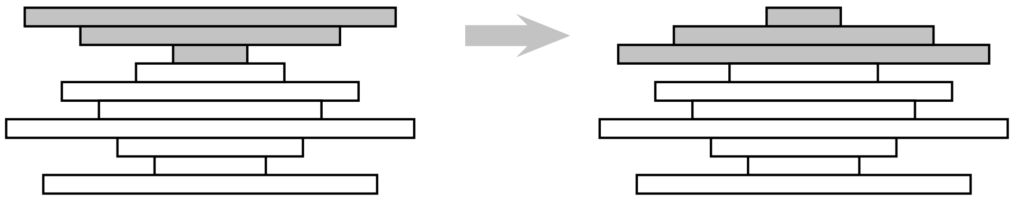
Due on **Monday, March 2, 2020 at 11pm**

CS/ECE 374 B: Algorithms & Models of Computation, Spring 2020 Version: **1.2**

Submission instructions as in previous homeworks.

13 (100 PTS.) **Pancake Algorithm.**

Suppose we have a stack of n pancakes of different sizes. We want to sort the pancakes so that the smaller pancakes are on top of the larger pancakes. The only operation we can perform is a *flip* - insert a spatula under the top k pancakes, for some k between 1 and n, and flip them all over.



13.A. (10 PTS.) Describe an algorithm to sort an arbitrary stack of n pancakes and give a bound on the number of flips that the algorithm makes. Assume that the pancake information is given to you in the form of an n element array A. $A[i]$ is a number between 1 and n and $A[i] = j$ means that the j'th smallest pancake is in position i from the bottom; in other words $A[1]$ is the size of the bottom most pancake (relative to the others) and $A[n]$ is the size of the top pancake. Assume you have the operation Flip(k) which will flip the top k pancakes. Note that you are only interested in minimizing the number of flips.

Solution:

Our algorithm will utilize an idea similar to that of selection sort. We start by finding the largest pancake. If it's not in the correct position, then we flip it to the top of the stack and flip it one more time to place it at the very bottom. The algorithm will then proceed by finding the next largest pancake, flipping it to the top (if it's not already in its correct position), and then flipping it to its correct position above the largest pancake. We do this process for each pancake.

```
PancakeSort(A):
    $n$ = length of $A$
    for $i = 1$ to $n$:
        if $A[i] \neq n - i - 1$:
            $m$ = FindIndexOfLargest($A, i$)
            Flip($m$)
            Flip($i$)
```

```
FindIndexOfLargest(A, j):
    $loc = 0$
    $value = -\infty$
    for $i = j$ to $n$:
        if $A[i] > value$:
            $loc = i$
            $value = A[i]$
    return $loc$
```

The correctness here follows by an easy induction (although a proof of correctness is not required for full credit on this problem).

The maximum number of flips required is $2n = O(n)$, as each pancake requires at most two flips to move to its correct position. Note that this bound can be reduced slightly to $2n-3$ flips by recognizing that the second-smallest pancake only requires at most a single flip to place it in the correct position, and the smallest pancake requires no flips.

In addition, note that the actual running time of the algorithm is asymptotically greater than $O(n)$ since each search for the largest pancake takes linear time. The implementation shown above is not very efficient - it runs in $O(n^2)$ time. One can do better by using data-structures, but it is not trivial.

13.B. Suppose one side of each pancake is burned. Describe an algorithm that sorts the pancakes with the additional condition that the burned side of each pancake is on the bottom. Again, give a bound on the number of flips. In addition to A, assume that you have an array B that gives information on which side of the pancakes are burned; $B[i] = 0$ means that the bottom side of the pancake at the i'th position is burned and $B[i] = 1$ means the top side is burned. For simplicity, assume that whenever Flip(k) is done on A, the array B is automatically updated to reflect the information on the current pancakes in A.

Solution:

We will run the same algorithm as in part (a), except with a slight modification. After we flip the largest unsorted pancake to the top, we check to see which side is burnt. If the burnt side is on top, then we do nothing, and the next step in our algorithm will flip it into its correct position with the burnt side on the bottom. However, if the largest unsorted pancake has a burnt side facing down when it is at the top of the stack, then we do a flip so that the burnt side is facing up.

```
BurntPancakeSort($A$):
  $n$ = length of $A$
  for $i = 1$ to $n$:
      if $A[i] \neq n - i - 1$ or $B[i] \neq 0$:
          $m$ = FindIndexOfLargest($A, i$)
          Flip($m$)
          if $B[n] = 0$:
              Flip($n$)
          Flip($i$)
```

The number of flips required is $3n = O(n)$. Each pancake requires at most three flips to move to its correct position: one to be flipped to the top, one flip to make sure the burnt side is in the right side, and one to put the pancake in the correct position. Note that once again, the actual running time of the algorithm is asymptotically greater than $O(n)$ since each search for the largest pancake takes linear time.

14 (100 PTS.) Fetching Bit by Bit.

Consider an array $A[0 \,..\, n-1]$ 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 $c < 2^{\ell-1}$
        $S[j] = 1$
    else
        $S[j] = 0$
```

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

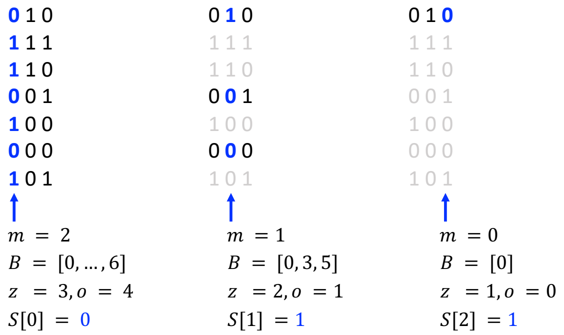
14.B. (40 PTS.) Suppose $n = 2^\ell - 1$ as before. Describe an algorithm to find the missing bit string in A using only $O(n)$ calls to **FetchBit**.

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$(A[0, ..., n-1], B[0, ..., 2^{m+1}-2], S[0, ..., \ell-1], m)$
    if $m \geq 0$
        $z = 0$
        $o = 0$
        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.

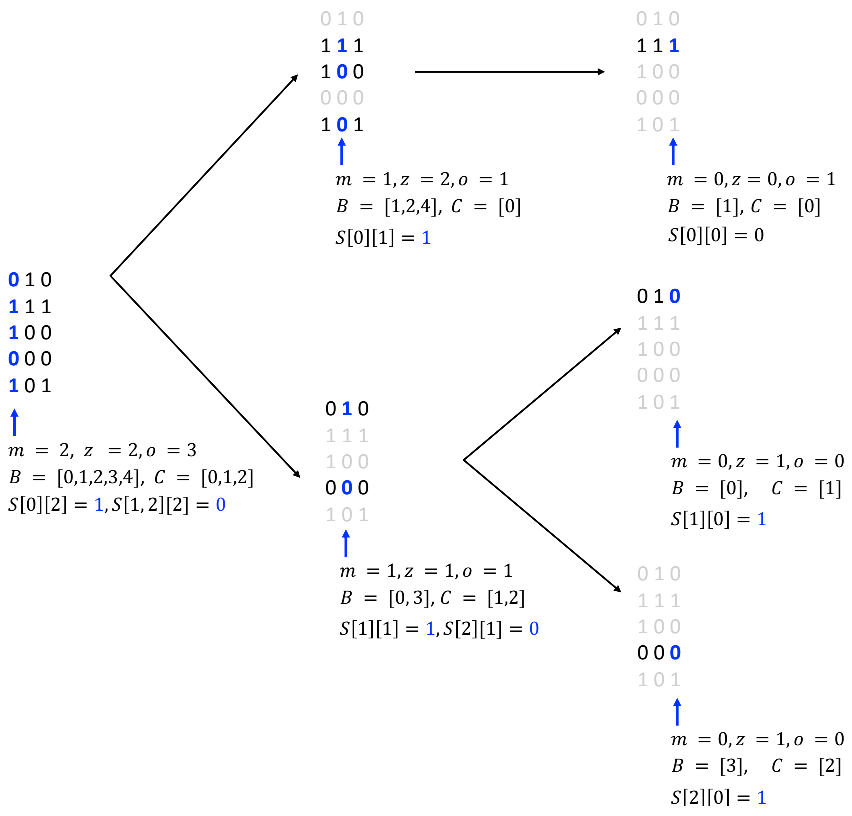
It might be tempting to assume that **findMissingk** recursively calls itself twice on half the input size leading to the following recurrence on the number of calls to **FetchBits**:

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

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 $m \geq 0$
    $z = 0$
    $o = 0$
    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:

$$T(n,k) = T(n/2,k_1) + T(n/2,k-k_1) + c_1 n$$

where k_1 is the number of strings in which the current missing bit is 1, $0 \le k_1 \le k \le n$, and $c_1 \ge 1$ is a constant. Observe that $T(n,0) = 0$, $T(n,1) = O(n)$ (from part B), and $T(1,1) = O(1)$.

We will prove by induction on k that $T(n,k) \le c_1 n \log k + c_2 n$ for $k \ge 1$ and some constants $c_2 \ge 2c_1$.

Base Case: $k = 1$

$$T(n,1) = T(n/2,1) + c_1 n \le 2c_1 n \le c_2 n \le c_1 n \log 1 + c_2 n.$$

Inductive Hypothesis: Assume for any $1 \le k' < k$, $T(n,k') \le c_1 n \log k' + c_2 n$

Proof:

$$T(n,k) = T(n/2,k_1) + T(n/2,k-k_1) + c_1 n$$

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

$$\begin{aligned} T(n,k) \le\ & c_1 \frac{n}{2} \log(k_1) + c_2 \frac{n}{2} + c_1 \frac{n}{2} \log(k-k_1) + c_2 \frac{n}{2} + c_1 n \\ \le\ & c_1 \frac{n}{2} \log(k_1(k-k_1)) + (c_2 + c_1) n \end{aligned}$$

Using basic calculus, we have $k_1 k - k_1^2 \le k^2/4$. Then,

$$\begin{aligned} T(n,k) \le\ & c_1 \frac{n}{2} \log(k^2/4) + (c_2 + c_1) n \\ \le\ & c_1 n \log k - c_1 \frac{n}{2} \log(4) + (c_2 + c_1) n \\ \le\ & c_1 n \log k + c_2 n \end{aligned}$$

If $k_1 = 0$ or $k_1 = k$, then $T(n,k) = T(n/2,k) + c_1 n$. 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,

$$\begin{aligned} T(n,k) =\ & c_1 n + c_1 n/2 + ... + c_1 n/2^j + T(n/2^{j+1}, k_1') + T(n/2^{j+1}, k - k_1') \\ \le\ & c_1 n + c_1 n/2 + ... + c_1 n/2^{j-1} + c_1 (n/2^j) \log k + c_2 n/2^j \\ \le\ & c_1 (n/2^j) \log k + 2c_1 n \\ \le\ & c_1 n \log k + c_2 n \end{aligned}$$

Thus, $T(n,k) \le c_1 n \log k + c_2 n = O(n \log k)$

15 (100 PTS.) **Solving Recurrence Relations.**

Solve the following recurrence relations. For parts (a) and (b), give an exact solution. For parts (c) and (d), give an asymptotic one. In both cases, justify your solution.

15.A. (10 PTS.) $A(n) = A(n-1) + 2n + 1; A(0) = 0$

Solution:

We will unroll the recurrence:

$$\begin{aligned}
A(n) = \;& A(n-1) + 2n + 1 \\
= \;& A(n-2) + 2(n-1) + 1 + 2n + 1 &&= A(n-2) + 4n + 1 - 1 \\
= \;& A(n-3) + 2(n-2) + 1 + 4n &&= A(n-3) + 6n + 1 - 4 \\
= \;& A(n-4) + 2(n-3) + 1 + 6n - 3 &&= A(n-4) + 8n + 1 - 9 \\
& \dots \\
= \;& A(n-k) + 2kn + 1 - (k-1)^2 \\
= \;& A(0) + 2n^2 + 1 - (n-1)^2 \\
= \;& n^2 + 2n
\end{aligned}$$

15.B. (10 PTS.) $B(n) = B(n-1) + n(n-1) - 1; B(0) = 0$

Solution:

We will unroll the recurrence:

$$\begin{aligned}
B(n) = \;& B(n-1) + n^2 - n - 1 \\
= \;& B(n-2) + (n-1)^2 - (n-1) - 1 + n^2 - n - 1 \\
= \;& B(n-3) + (n-2)^2 - (n-2) - 1 + (n-1)^2 - (n-1) - 1 + n^2 - n - 1 \\
& \dots \\
= \;& B(n-k) + \sum_{i=0}^{k-1} \big((n-i)^2 - (n-i) - 1 \big) \\
= \;& B(0) + \sum_{i=0}^{n-1} \big((n-i)^2 - (n-i) - 1 \big) = \sum_{m=1}^{n} \big(m^2 - m - 1 \big) \\
= \;& n(n+1)(2n+1)/6 - n(n+1)/2 - n = n(n^2-4)/3
\end{aligned}$$

15.C. (10 PTS.) $C(n) = C(n/2) + C(n/3) + C(n/6) + n$

Solution:

We model this recurrence as a recursion tree.

$$n$$

$$\frac{n}{2} \qquad \frac{n}{3} \qquad \frac{n}{6}$$

$$\frac{n}{4} \quad \frac{n}{6} \quad \frac{n}{12} \quad \frac{n}{6} \quad \frac{n}{9} \quad \frac{n}{18} \quad \frac{n}{12} \quad \frac{n}{18} \quad \frac{n}{36}$$

At each level, the nodes sum up to n. The maximum depth of the tree is $\log n$. Therefore, we have $\log n$ levels with n work per level, so the recurrence is $O(n \log n)$.

15.D. (10 PTS.) $D(n) = D(n/2) + D(n/3) + D(n/6) + n^2$

Solution:

We model this recurrence as a recursion tree.

$$n^2$$

$$\frac{n^2}{4} \qquad \frac{n^2}{9} \qquad \frac{n^2}{36}$$

$$\frac{n^2}{16} \quad \frac{n^2}{36} \quad \frac{n^2}{144} \quad \frac{n^2}{36} \quad \frac{n^2}{81} \quad \frac{n^2}{324} \quad \frac{n^2}{144} \quad \frac{n^2}{324} \quad \frac{n}{1296}$$

At the root, the work sums up to n^2. At level 2, the work sums up to $\frac{7}{18}n^2$. At level 3, the work sums up to $\frac{49}{324}n^2$. At level i, the work sums up to $(\frac{7}{18})^i n^2$. Hence, the work at each level is a decreasing geometric series. The overall recurrence is dominated by the root node. Therefore, the recurrence is $O(n^2)$.