CS/ECE 374 Spring 2017 — Lanxiao Bai (lbai5)
Homework 4 Problem 1 — Renheng Ruan (rruan2)

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.

1. 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.
2. 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 $A$.

No proof of correctness necessary.

---

**Solution:** 1. Consider the sequence of pancakes, we can always use one FLIP to flip the largest pancake to the top from it's original position and flip the whole tower of pancakes to put it at bottom, where it should be. Then what left for us is to sort the pancakes on the lowest one, so we have $O(2n)$ flips.

```
SORT(A[s..n]):
  if n = s
      return
  else
      k_max ← s
      max = A[s]
      for i ← s to n
          if A[i] > max
              max ← A[i]
              k_max ← i
      FLIP(k_max)
      FLIP(s)
      SORT(A[s+1..n])
```

2. Consider one more requirement that the burned side should on the bottom, after each time the largest pancake was flipped to the top, we want to make sure that the burned side is on the top before flip the whole tower, so we have $O(3n)$ flips.

```
SortBurned(A[s..n], B[s..n]):
  if n = s
      if B[n] = 0
          Flip(A[n..n])
      return
  else
      k_max ← s
      max = A[s]
      for i ← s to n
          if A[i] > max
              max ← A[i]
              k_max ← i
      Flip(k_max)
      if B[n] = 0
          Flip(n)
      Flip(s)
      SortBurned(A[s+1..n], B[s+1..n])
```

■

CS/ECE 374 Spring 2017 — Lanxiao Bai (lbai5)
Homework 4 Problem 2 — Renheng Ruan (rruan2)

Suppose you are given $k$ sorted arrays $A_1, A_2, \ldots, A_k$ each of which has $n$ numbers. Assume that all numbers in the arrays are distinct. You would like to merge them into single sorted array $A$ of $kn$ elements. Recall that you can merge two sorted arrays of sizes $n_1$ and $n_2$ into a sorted array in $O(n_1 + n_2)$ time.

1. Use a divide and conquer strategy to merge the sorted arrays in $O(n \log k)$ time. To prove the correctnes of the algorithm you can assume a routine to merge two sorted arrays.
2. In MergeSort we split the array of size $N$ into two arrays each of size $N/2$, recursively sort them and merge the two sorted arrays. Suppose we instead split the array of size $N$ into $k$ arrays of size $N/k$ each and use the merging algorithm in the preceding step to combine them into a sorted array. Describe the algorithm formally and analyze its running time via a recurrence.

---

**Solution:** 1. Each time we merged two arrays of the same size, and repeat this routine for arrays that has larger scale.

```
MergeArrays(Arrays[1..k]) :
  if k = 1
      return Arrays
  else
      mergedArrays ← new Arrays[1..⌊k/2⌋]
      for i ← 1 to ⌊k/2⌋
          mergedArrays[i] = Merge(Arrays[i], Arrays[2i])
      if k is even
          return MergeArrays(mergedArrays)
      else
          return Merge(MergeArrays(mergedArrays), Arrays[k])
```

**Claim:** The algorithm for merging arrays is correct.

**Proof:** Base case: When $k = 1$, the array is already sorted and merged as the condition given.

Inductive hypothesis: Assume that for all $k \leq n$, the algorithm above correctly merged the arrays.

Inductive step: When $k = n + 1$, there are two conditions.

If $n$ is odd, then $k$ is even, so MergeArrays will be reduced to MergeArrays of half of its original size and $k/2$ Merge. By the correctness of Merge and inductive hypothesis, the algorithm is correct.

If $n$ is even, then $k$ is odd, so MergeArrays will be reduced to MergeArrays of half of its original size and $k/2 + 1$ Merge. By the correctness of Merge and inductive hypothesis, the algorithm is correct.

Hence, we conclude that the algorithm is correct. □

**Claim:** The time complexity of this algorithm is $O(n \log k)$.

**Proof:** From the algorithm we described above, we can see $T(k) = T(k/2) + cn$ and $T(1) = a$. So we expand the recursion tree,

$$\begin{aligned} T(k) &= T(k/2) + cn \\ &= T(k/4) + cn + cn \\ &\vdots \\ &= T(1) + cn + cn + \cdots + cn \end{aligned}$$

Let $k = 2^h$, then $h = \log k$. Thus, $T(k) = a + chn = O(n \log k)$. ■

2. By following the described algorithm above, we have

```
MergeSortK(A[1..n], k):
  if n = 1
      return A
  else
      interval = n/k
      for i ← 1 to k
          M_i ← MergeSortK(A[ik..ik + interval], k)
      return MergeArrays(M[1..k])
```

From the formalized algorithm, we see that $T(n) = kT(n/k) + ck \log n$ and $T(1) = a$. By expanding the recursion tree, we see that

$$\begin{aligned} T(n) &= kT(n/k) + Ck \log n \\ &= k(kT(n/k^2) + ck \log n) + ck \log n \\ &\vdots \\ &= ak^h + \sum_{i=0}^{h-1} k^i \cdot ck \log n \\ &= ak^h + \sum_{i=0}^{h-1} k^i \cdot ck \log n \\ &= ak^h + ck \log n \sum_{i=0}^{h-1} k^i \end{aligned}$$

Since $n/k^h = 1 \Rightarrow n = k^h \Rightarrow h = \log n$, we have

$$\begin{aligned} T(n) &= an + ck \log n \sum_{i=0}^{h-1} k^i \\ &= an + \frac{n-1}{k-1} ck \log n \\ &= O(n \log n) \end{aligned}$$

Hence we conclude that $T(n) = O(n \log n)$.