

Homework 4

Due April 14

Problem 4.1

We have discussed the following sorting algorithms:

- Merge Sort, see algorithms 4.1.1 and 4.1.2
 - Randomized Quick Sort see algorithms 4.1.3 4.1.4,
 - Heap Sort see algorithm 4.1.5 with the priority queue P being implemented as a heap, and
 - Insertion Sort see algorithm 4.1.6
- (a) Among the four, which are *stable*? A sorting algorithm is *stable* if elements with the same key values appear in the output sequence in the same order as they do in the input sequence.
- (b) For the unstable ones, give a small sequence example which sorts unstably.
- (c) Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

Algorithm 4.1.1 Merge-Sort($A[1, \dots, n]$)

```
if Length( $A$ ) > 1 then
   $mid \leftarrow \lceil n/2 \rceil$ 
  return Merge(Merge-Sort( $A[l, \dots, mid]$ ), Merge-Sort( $A[mid + 1, \dots, n]$ ))
else
  return  $A$ 
end if
```

Algorithm 4.1.2 Merge($X[1, \dots, k], Y[1, \dots, l]$)

```
if  $k == 0$  then
  return  $Y[1, \dots, l]$ 
else if  $l == 0$  then
  return  $X[1, \dots, k]$ 
else if  $X[1] \leq Y[1]$  then
  return  $X[1] \circ \text{Merge}(X[2, \dots, k], Y[1, \dots, l])$ 
else
  return  $Y[1] \circ \text{Merge}(X[1, \dots, k], Y[2, \dots, l])$ 
end if
```

Algorithm 4.1.3 Randomized-QuickSort($A, left, right$)

```
if  $left < right$  then
   $p \leftarrow$  Randomized-Partition( $A, left, right$ )
  Randomized-Quicksort( $A, left, p$ )
  Randomized-Quicksort( $A, p + 1, right$ )
end if
```

Algorithm 4.1.4 Randomized-Partition($A, left, right$)

```
Choose  $pivot$  as a random number between  $left$  and  $right$  inclusive
 $pivot\_value \leftarrow A[pivot]$ 
Exchange  $A[pivot]$  with  $A[right]$ 
 $index \leftarrow left$ 
for  $i$  from  $left$  to  $right - 1$  do
  if  $A[i] \leq pivot\_value$  then
    Exchange  $A[i]$  and  $A[index]$ 
     $index := index + 1$ 
  end if
end for
// Move pivot to its final place
 $A[index] \leftarrow pivot\_value$ 
return  $index$ 
```

Algorithm 4.1.5 PQ-Sort(A , heap-based priority queue P)

```
for  $i$  from 1 to  $length(A)$  do
   $e \leftarrow A[i]$ 
   $P.Insert-Item(A, e)$ 
end for
 $i \leftarrow 1$ 
while  $P$  is not empty do
   $e \leftarrow P.Remove-Min()$ 
   $A[i] \leftarrow e$ 
   $i \leftarrow i + 1$ 
end while
```

Algorithm 4.1.6 Insertion-Sort(A)

```
for  $j \leftarrow 2$  to  $length[A]$  do
   $key \leftarrow A[j]$ 
  //Insert  $A[j]$  into the sorted sequence  $A[1, \dots, j - 1]$ 
   $i \leftarrow j - 1$ 
  while  $i > 0$  and  $A[i] > key$  do
     $A[i + 1] \leftarrow A[i]$ 
     $i \leftarrow i - 1$ 
  end while
   $A[i + 1] \leftarrow key$ 
end for
```

Problem 4.2

In the CONTIGUOUS SUBSEQUENCE problem you are given an array A of integers x_1, \dots, x_n , and have to output maximum *spansum* that is the sum of the contiguous subsequence x_i, \dots, x_j that has the largest sum of all possible contiguous subsequences.

- (a) Suppose the list has an *alternating* sequence starting and ending with a negative number for example $-3, 2, -1, 5, -4, 2, -7$. Give an algorithm that finds the maximum spansum. What is the running time of your algorithm?
- (b) Now suppose you are given a list with a general sequence (containing positive, negative or zero). Give an algorithm to find the maximum spansum in this case. What is the running time of your algorithm?

Problem 4.3

- (a) Suppose we are given an n -element vector S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an $O(n \log n)$ -time algorithm to see who wins the election. Assume the candidate with the most votes wins and there is no tie. (Note: you may *not* assume that the IDs of the candidates are the integers $1, 2, 3, \dots$; instead think of them as something like the candidates' social security numbers. In particular, some candidate's ID could be as large as, say, n^n .)
- (b) Now suppose that we know the number $k < n$ of candidates running. Describe an $O(n + k)$ -time algorithm for determining who wins the election.

Problem 4.4

Suppose two binary trees, T_1 and T_2 , hold entries satisfying the heap-order property. Describe a method for combining T_1 and T_2 into a tree T whose internal nodes hold the union of the entries in T_1 and T_2 and also satisfy the heap-order property. Your algorithm should run in time $O(h_1 + h_2)$ where h_1 and h_2 are the respective heights of T_1 and T_2 . You may assume that all entries in the two trees are distinct. Note that the trees need only have the heap-order property; neither T_1, T_2 , nor the result, are necessarily left-full.

Problem 4.5

- (a) The number of steps that a sorting algorithm must execute depends not only on the length of its input, but also on the ordering of the input. Provide a list of five integers that maximizes the number of steps executed by insertion sort. That is, no other list of five integers would cause insertion sort to execute more steps. Similarly, provide a list of five integers that maximizes the number of steps executed by selection sort.

- (b) Banks often record transactions on an account in order of the time the transaction occurred, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number and merchants typically cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost sorted input. Which sorting procedure Quicksort or Insertion sort would tend to work faster in this case?

Problem 4.6

Consider any family of Hash functions H which maps objects from a finite set U to a finite set B such that $|U| > |B|$. Prove that for all H , the collision property of H cannot be better than the following:

For all pairs of distinct elements k and l in U ,

$$\Pr[h(k) = h(l)] \geq \frac{1}{|B|} - \frac{1}{|U|}$$

where the probability is taken over the drawing of hash function h at random from the family H .

In other words, for all H , the probability of collision between some pair of distinct items k and l must be at least $1/|B| - 1/|U|$.

Hint: $1/|B| - 1/|U| = (|U|/|B| - 1)/|U|$

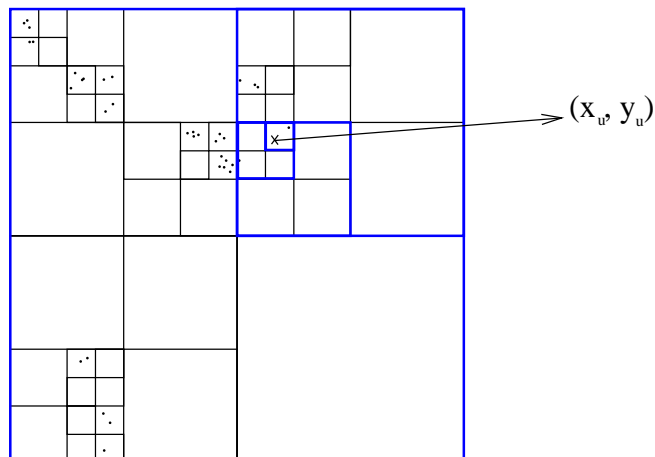
Problem 4.7

Suppose you take an internship with Google Maps. Google has a database of *places*, each assigned an (x, y) position on a map of the US (assume (x, y) pairs have $0 \leq x \leq M$ and $0 \leq y \leq M$ for some M , and (x, y) coordinates are like graph paper: x increases to the East, y increases to the North). You are told to design an algorithm, which, when a user clicks on position (x_u, y_u) , returns the place in the database nearest (x_u, y_u) if it is within some minimum distance M_{min} , and otherwise returns NONE.

To do so, you will represent the set of places in a particular kind of tree we will call a GoogleTree. Each node in this tree will represent a square region of the map (represented by the coordinates of its upper right and lower left corners). Each internal node v in this tree will have four children, representing a partition of the region represented by v into equal-size quadrants. A node in a GoogleTree will have no children when either (a) there are no places in the database that occupy the node's region, or (b) when the region represented by the node is of some minimum size, say $M_{min} \times M_{min}$. In this latter case, a linked list of all places in the region is stored at the node.

You may assume that the size of the side of the whole map $M = 2^k M_{min}$ for some positive integer k , i.e. the map can be divided up by repeated partitioning

into even-sized chunks. The structure will look something like the following picture:



To find the place closest to the user's click (x_u, y_u) , we start at the root node. We first query whether our node has children, and if so, which child contains (x_u, y_u) . Repeating this process until a leaf ℓ is reached gives us a list of places in the same minimum-size region as (x_u, y_u) ; we can iterate through these places to determine which of them is closest to (x_u, y_u) . Now, we are not necessarily done; there may be nearer places across one of the borders of ℓ . So for each of the directions up, down, left, and right, if the border is nearer to (x_u, y_u) than the nearest place, we search for the region corresponding to moving M_{min} in that direction. So for instance, if we are looking to the right, we would query for the region containing $(x_u + M_{min}, y_u)$. We then examine each of the places (if any) in those regions to see if it is within M_{min} of (x_u, y_u) and is the nearest such place. If after all of this we have found a place, we return it; otherwise we return NULL.

To insert a new place at position (x, y) into the GoogleTree, we start at the root node, and search down for the smallest node containing (x, y) . If this region is empty and is larger than $M_{min} \times M_{min}$, we partition the region into quadrants and create 4 children, one assigned to each quadrant. We then try to insert (x, y) into the child whose quadrant contains (x, y) . On the other hand, if the node has a region is of size $M_{min} \times M_{min}$ then we add (x, y) to the list of places in that region.

- Suppose we want to remove a place at position (x, y) such as a restaurant that has closed. Describe how to do this, while maintaining the property that an empty square region should be as large as possible.
- For what areas of the actual US would you expect queries to take the longest in this data structure. For what areas would queries be faster?
- For the US, $M \approx 3500$ miles. Google would like $M_{min} \leq 50$ feet. What is k , the number of nodes you need to traverse to get from the root of the tree

to a leaf representing a minimal region?