1. [10 pt] Please find a tight asymptotic bound for the following recurrences and provide an explanation.

   (a) [5 pt] $T(n) = 2T(\frac{n}{4}) + \sqrt{n}$ where $T(n) = c$ for $n \leq 2$ and c is a constant positive integer.

   (b) [5 pt] $T(n) = 2T(\frac{n}{4} - 100) + \sqrt{n}$ where $T(n) = c$ for $n \leq 2$ and c is a constant positive integer.

(a) Using Master theorem

$a=2 \quad b=4 \quad f(n)=\sqrt{n}$

$\because n^{\log_b a} = \sqrt{n} \quad \& \quad f(n) = \Theta(\sqrt{n})$

$\therefore T(n) = \Theta(\sqrt{n}\log n)$

(b) Assume $T(n) = 2T(\frac{n}{4}-100) + \sqrt{n} = O(\sqrt{n}\log n)$

Basic step: for $n \leq 2$, $T(n) = c$

$T(n) = c = O(\sqrt{n}\log n) \quad (n \leq 2)$

Inductive hypothesis: Assume that $T(k) \leq c\sqrt{n}\log n$ for $k = 2, 3, \cdots n$

Inductive step: prove that $T(n+1) \leq c\sqrt{n+1}\log(n+1)$

$T(n+1) = 2T(\frac{n+1}{4} - 100) + \sqrt{n+1}$

$\leq 2c\sqrt{\frac{n+1}{4}-100}\log(\frac{n+1}{4}-100) + \sqrt{n+1}$

$\leq 2c\sqrt{\frac{n+1}{4}}\log(\frac{n+1}{4}) + \sqrt{n+1}$

$\leq c\sqrt{n+1}(\log n+1 - 1) + \sqrt{n+1}$

$\leq c\sqrt{n+1}\log n+1 + \sqrt{n+1} - 2c\sqrt{n+1} \leq c\sqrt{n+1}\log n+1$

$\therefore T(n) = O(\sqrt{n}\log n)$

2. [5 pt] Suppose n 6-sided fair dice are rolled independently. What is the expected sum of the outcome of these dice? Please provide the derivation.

Let X = the value of one dice is rolled

$E(X) = 1\cdot\frac{1}{6} + 2\cdot\frac{1}{6} + 3\cdot\frac{1}{6} + 4\cdot\frac{1}{6} + 5\cdot\frac{1}{6} + 6\cdot\frac{1}{6} = \frac{21}{6}$

$\therefore E(nX) = nE(X) = \frac{21n}{6}$

3. [10 pt] For each statement below, please identify if the following sorting algorithm is stable, in-place, or both stable and in-place, and provide a short explanation to support your answer. Sometimes, these sorting algorithms can have various modifications. In this assignment, the following algorithms refer to the pseudo-codes discussed in lectures.

	stable	in-place
(a) [2.5 pt] Insertion Sort	✓	✓
(b) [2.5 pt] Merge Sort	✓	×
(c) [2.5 pt] Rand Quick Sort	×	✓
(d) [2.5 pt] Counting Sort	✓	×

(a) Insertion Sort is both stable and in-place, it is stable because the condition of the while loop (A[i] > key) prevent us from changing the order of similar value, it will not insert if the condition can't be satisfied. It is in-place because, We are actually sorting it inside the array, We use key to insert into this array to re-arrange the order without using any additional array.

(b) Merge Sort is stable but not in-place. It is stable because the instruction (if L[i] ≤ R[j]) keep the merge sort stable. Which means that if We have the same value, then always the left's value will go into the array A first. It is not in-place because in the function MERGE(A, p, q, r), We create two new arrays L[1...n1+1] and R[1...n2+1].

(c) Rand Quick sort is not stable but in-place. It is not stable because if we have this array [5, 3a, 8, 3b, 7], and we pick 8 as pivot, one possible the final result will be [3b, 3a, 5, 7, 8]. It is in-place because the basic operation of the algorithm is just swapping two value, not creating new array.

(d) Counting sort is stable but not in-place. It is stable because of "for j = A.length downto 1 B[C[A[j]]] = A[j] C[A[j]] = C[A[j]] - 1" We are assigning the value backward, when we update an equivalent value to B, C will decrease one, therefore We have no change to change the original order. It is not in-place, because We using a new C array to help us to sort the array.

4.(a) [5 pt] The method for arranging the files along with an explanation that the run-time complexity of this method is indeed $\Theta(n + m)$ time. You can assume moving a single file takes constant time.

(b) [5 pt] The method for retrieving the files given a range of IDs in $O(1)$ time. Please also provide an explanation of the time complexity.

(a) Assumed that all file are Lined up randomly in a shelf A. The method is:

① use a notebook to write down a list, which index is from 0 to m, and it is used to store how many time the file ID appeared in the file shelf.

② and then update this list by List[i] = List[i] + List[i-1] i from 0 to m, now the list is containing the number of elements less than or equal to the index of it

③ now using a new shelf B to be the sorted shelf, now moving the file from A to B starting from the end of the A to the beginning by using the List. every time we move a file, we use the file ID to retrieve the list to find the right spot of the B that the file will be put in. After moving it, the value of the list will decrease one.

It is $\theta(n+m)$ because the time complexity of step① is $\theta(n)$, since the loop range is all the files, step② is $\theta(m)$. because we are calculating it from 0 to m, and step③ is $\theta(n)$ because we are moving all the files and the total number of it n. the total time complexity $T(n) = \theta(n) + \theta(m) + \theta(n) = c_1 n + c_2 m + c_3 n = (c_1 + c_3) n + c_2 m = \theta(n+m)$

(b)

After re-arranging all the files. Mr S uses tags to mark each file on the shelf, and records all files location in to the computer to build a hashmap.

If the range of the files is [a, b], then the retrieving time will be $O(1)$ because first Mr S use the key "a" and "b" to get the location of the files. then he can grab all the files between this two location. $T(a) = T(b) = O(1) \therefore T(n) = O(n)$

5.

(a) [5 pt] An asymptotic worst-case upper bound. Please make your bound as tight as possible.

(b) [10 pt] An asymptotic average-case upper bound. Please make your bound as tight as possible.
Hint: It might help to think in terms of recurrence tree and consider the number of elements that can be identified at different levels of the tree.

(a)

	cost	time
1: Let $s = 1$	c_1	1
2: Let $e = A.length$	c_2	1
3: **while** $s \le e$ **do**	c_3	$(\log n + 1)$
4: $m = \lfloor \frac{s+e}{2} \rfloor$	c_4	$\log n$
5: **if** $A[m] == v$ **then**	c_5	$\log n$
6: Return m	c_6	0
7: **else if** $A[m] < v$ **then**	c_7	$\log n$
8: Let $s = m + 1$	c_8	0
9: **else**	c_9	$\log n$
10: Let $e = m - 1$	c_{10}	$\log n$
11: Return unsuccessful.	c_{11}	0

In the worst case scenario, the v is the A[5]. So every time the problem will be halved until the size of A is 1. therefore we will run line 4~10 $\log n$ time.

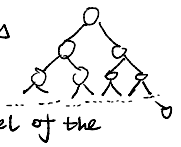
$$T(n) = \underbrace{c_1 + c_2 + c_6}_{+c_6} + c_3(\log n + 1) + (c_4 + c_5 + c_7 + c_9 + c_{10}) \log n$$
$$= \underbrace{[c_1 + c_2 + c_3]}_{+c_6} + [c_3 + c_4 + c_5 + c_7 + c_9 + c_{10}] \log n$$
$$= c' + c'' \log n$$

$\because \exists c,\ c' + c'' \log n \ge c \log n \ \therefore T(n) = O(\log n)$

(b) $\because$ element v must exist in A, and $n = 2^K \ \therefore m_1 = \lfloor \frac{2^K + 1}{2} \rfloor = 2^{K-1}$, # elements in left $= 2^{K-1} - 1$

elements in right $= 2^K - 2^{K-1} = 2^{K-1}$

therefore right side has one more element, if we construct the BST it will look like this



and we define X = # while loop is executed, and we define that the # while loop is executed as the level of the tree plus one.

$$\therefore E(X) = 1 \cdot \frac{2^0}{n} + 2 \cdot \frac{2^1}{n} + 3 \cdot \frac{2^2}{n} + \cdots + \log n \frac{2^{\log n - 1}}{n} + (\log n + 1)\frac{1}{n} = \sum_{i=1}^{\log n} i \frac{2^i}{n} + (\log n + 1)\frac{1}{n} = \frac{1}{n} \sum_{i=0}^{\log n} i \cdot 2^i + \frac{\log n + 1}{n} = \frac{1}{n}(\log n - 1) 2^{\log n + 1} + 2 + \frac{\log n + 1}{n}$$
$$= \frac{1}{n}(\log n - 1) \cdot 2 \cdot n + 2 + \frac{\log n + 1}{n}$$
$$= 2 \log n + \frac{\log n}{n} + 1 = 2 \log n + \frac{\log n}{n} + \frac{1}{n}$$
$$= \log n \left(2 + \frac{1}{n}\right) + \frac{1}{n} = O(\log n)$$

(a) [10 pt] If small win means the player receives a profit P, where $\$0.3n < \$P < \$0.6n$, is there a randomised algorithm such that in the expected worst-case scenario, the player receives a small win? If there is, please provide the algorithm and explanation on the expected worst-case profit. If not, please explain why such an algorithm does not exist.

We know that the length of the string is at most n. The algorithm is that randomly choose a character to guess whether it is the prefix starting from index 1 of the string and end at the index n.

In the expected worst-case scenario, the length of the string that is typed by monkey is n. For each index the expected guessing time is: $\begin{cases} P(x) = \text{Guess is correct by asking } x \text{ questions.} \\ E(x) = 1\cdot P(1) + 2\cdot P(2) + 3\cdot P(3) + 4\cdot P(4) + 5\cdot P(5) = 2.5 \end{cases}$, so for a string length n we will ask $2.5n$ questions.

So the profit is $4n - 2.5n = 1.5n$

(b) [5 pt] If large win means the player receives a profit $\$P \geq \$0.6n$, is there a randomised algorithm such that in the expected worst-case scenario, the player receives a large win? If there is, please provide the algorithm and explanation on the expected worst-case profit . If not, please explain why such an algorithm does not exist.

(a) [5 pt] Formulate the problem of finding the right position of the helipad as one of finding medians of the X coordinates and the Y coordinates. Please also prove that such a formulation will minimise D.

Let $X[x_1, x_2, x_3 \cdots x_n]$, $Y[y_1, y_2, y_3 \cdots y_n]$, where $y_i \le y_{i+1}$, $x_i \le x_{i+1}$, so $x_h = X[\lfloor \frac{1+n}{2} \rfloor]$ $y_h = Y[\lfloor \frac{1+n}{2} \rfloor]$.

$\because D = \sum_{i=1}^{n} |x_i - x_h| + \sum_{i=1}^{n} |y_i - y_h|$, to prove when x_h, y_h is the median, the D is minimized

assume: $\sum_{i=1}^{n} |a_i - a_h|$. We show that if a_h is not the median, then we can never increase the total sum by moving a_h in the direction of the median.

so if $a_1, a_2, a_3 \cdots a_h \cdots a_m \cdots a_{n-2} a_{n-1} a_n$, let #numbers $> a_h$ be a, let #numbers $< a_h$ be b, let #numbers $= a_h$ be c and we assume that a_h is selected $< a_m$, so it is belonged to a, therefore $a \ge b+c$. now increase a_h by 1, so the $\sum_{i=1}^{n-1} |a_i - a_h|$ will increase by 1 while $\sum_{i=n+1}^{n} |a_i - a_h|$ will decrease by 1. As $a \ge b+c$, this cannot increase the total cost. Vice versa for another case.

(b) [10 pt] Design a randomized algorithm with an expected running time of $\Theta(n)$ to find the position (x_h, y_h) that minimises D. Your algorithm can only use at most a constant number of extra storage outside the array that contains the positions of the towns the helipad serves. Please also provide an explanation that the expected time complexity of the algorithm you propose is indeed $\Theta(n)$.
You can assume that the positions of the towns are independent and uniformly distributed, positions of the helipad and towns can overlap, and all X coordinates are distinct and all Y coordinates are also distinct.
Hint: You might want to think of modifying RandQuickSort.

This algorithm uses the modified RandQuickSort, one of the property of RandQuickSort. is that the pivot we choose every time is actually the ith smallest number in the array. So we use the pivot index to check whether it is middle index of the array If it is bigger than pivot index, then we will use the "Partition" function again but with different starting index and ending index. The same for the situation that pivot index is smaller than median index

code: int RandomizedFind(int[] a, int p, int r, int m){ Where Partition is same as lecture slides

```
if(p==r)
    return a[p];
pivot_index = Partition(a,p,r)
i_smallest = pivot_index - p + 1;
if(i_smallest == m)
    return a[pivot_index]
else if (m < i_smallest)
    return RandomizedFind(a, p, pivot_index -1, m)
else
    return RandomizedFind(a, pivot_index +1, r, m - i_smallest)
}
```

proof:

The pivot is selected Randomly. so for every k ($1 \le k \le n$), the probability of the $A[p \cdots q]$ will have k elements is $\frac{1}{n}$. Then We define for $k = 1, 2, 3 \cdots n$, $X_k = I\{A[p \cdots q]$ has exactly k elements$\}$, therefore we have $E[X_k] = \frac{1}{n}$. In order to get the upper bound of $T(n)$, We can assume that the median always be divided to the bigger number side by the pivot. so We will have $T(n) \le \sum_{k=1}^{n} X_k \cdot (T(\max(k-1, n-k)) + O(n)) = \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)$

And $E[T(n)] \le \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O(n) = \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n) = \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n)$. We also find that: $\max(k-1, n-k) = \begin{cases} k-1 & k > \lceil \frac{n}{2} \rceil \\ n-k & k \le \lceil \frac{n}{2} \rceil \end{cases}$.

Also noticed that:

$k=1 \to n-1$
$k=2 \to n-2$
$k=3 \to n-3$
$k=\frac{n-1}{2} \to \frac{n+1}{2}$
$k=\frac{n}{2} \to \frac{n}{2}$
$k=\frac{n+1}{2} \to n-1$
$k=n-2 \to n-3$
$k=n-1 \to n-2$
$k=n \to n-1$

if $n\%2=0$, then every terms from $T(\lceil \frac{n}{2} \rceil)$ to $T(n-1)$ will appear Twice, else all terms will appear twice except $T(\lceil \frac{n}{2} \rceil)$

So we have $E[T(n)] \le \frac{2}{n} \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} E[T(k)] + O(n)$.

We Guess $E[T(n)] = O(n)$

NEXT PAGE

proof: $E[T(n)] = O(n)$

Basic step $n=2$: $E[T(2)] \le \frac{2}{2}\sum_{k=1}^{1} E[T(1)] + O(n)$

$E[T(2)] \le O(n) + \theta(1)$

IH: $E[T(n)] \le cn$

Inductive step:

$$E[T(n+1)] \le \frac{2}{n+1}\sum_{k=\lfloor \frac{n+1}{2} \rfloor}^{n} c(k+1) + O(n+1)$$

$$\le \frac{2c}{n+1}\left(\sum_{k=1}^{n} k+1 - \sum_{k=1}^{\lfloor \frac{n+1}{2} \rfloor - 1} k+1\right) + a(n+1) \quad \text{for some } a$$

$$\le \frac{2c}{n+1}\left(\frac{(2+n+1)n}{2} - \frac{(2+\frac{n+1}{2}-1)(\frac{n+1}{2}-2)}{2}\right) + a(n+1) = \frac{2c}{n+1}\left(\frac{(n+3)n}{2} - \frac{(n+3)\frac{1}{2}\cdot(n-3)\frac{1}{2}}{2}\right) + a(n+1) = \frac{2c}{n+1}\left(\frac{(n+\frac{5}{2})(n+1) - \frac{1}{4}(n+1)^2}{2}\right) + a(n+1)$$

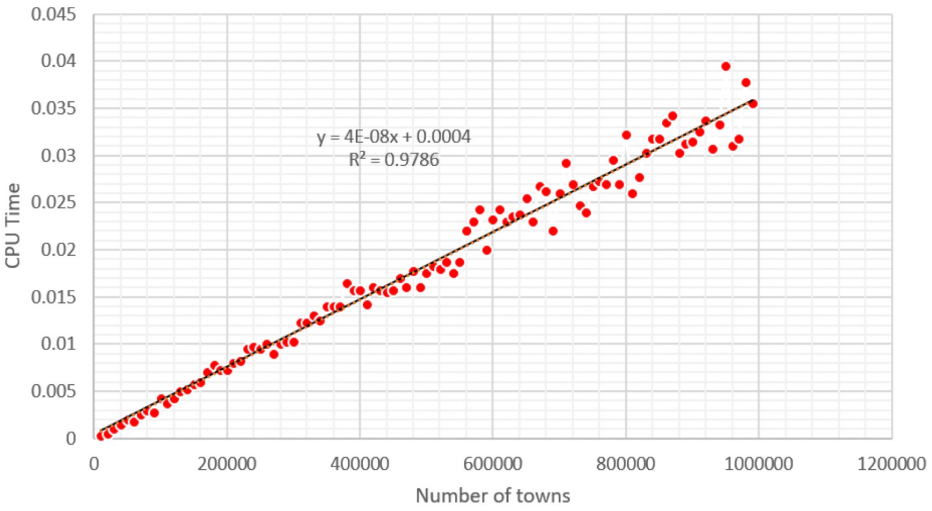
$$= c\left(n+\frac{5}{2}-\frac{1}{4}(n+1)\right) + a(n+1) = c\left(\frac{3}{4}n+\frac{9}{4}\right) + a(n+1) \le c\left(\frac{9}{4}n+\frac{9}{4}\right) + a(n+1) = \left(\frac{9}{4}c+a\right)(n+1) \le c'(n+1)$$

So prove it

In the best case scenario, the pivot that the "Randomized partition" function chooses is exactly the median, However, it will still walk through the whole array to partition it, so it is $T(n) \ge Cn$, therefore $T(n) = \Omega(n)$

therefore we can say that $T(n) = \theta(n)$

(d) [10 pt] Experimental analysis of the time complexity of your algorithm and comparison against the theoretical analysis (7b). You will need to have sufficient data to fit a function well and will need to generate your own test cases for this purpose.



The range of the data is from 10000 towns to 990000 towns. and for each test case i ($i \in [10000, 990000]$) will run 20 time to get the average value. The number of towns is close to 2^{20}, so it will show the performance of this algorithm properly, and taking the mean value of each test cases also guarantee that contingency is eliminated.

And then using the average value from 10000 to 990000 We calculate the cpu time, and have a function: $y = 4E\text{-}08x + 0.0004$ with $R^2 \approx 0.98$

the R^2 is very close to one, And the time complexity given in 7(b) is $\theta(n) = Cn$.

$\therefore 4E\text{-}08n + 0.0004 = \theta(n)$

$\therefore$ 7(b) analysis is correct.