COMP 9101

Assignment 1

Student ID : z5174741

Student name : GUANQUN ZHOU

Q1 For this question, I think the algorithm consist of 4 steps:

Step1:

We use Merge sort to sort the array and Merge sort is an $O(nlogn)$ algorithm. Merge sort is also very stable.

Step2:

For each query, we can use binary search to find the largest index of the element that is less than L_x. And the index is called mid in every binary search.
Note that in the process of this binary search, if the element that mid matches is equal to L_x. If so, then we have to continue to run the binary search to search in the group of smaller indexes until we find the largest index of the element that is less than L_x.

Step3:

For each query, we can use binary search to find the largest index of the element that is not greater than R_x. And the index is called mid in every binary search.
Note that in the process of this binary search, if the element that mid matches is equal to R_x. If so, then we have to continue to run the binary search to search in the group of larger indexes until we find the largest index of the element that is not greater than R_x.

Step4:

We can use these two index to calculate the number of elements a satisfy

$$L_x \leq a \leq R_x$$

The time complexity for each query is $O(logn)$ because binary search and there. are n queries. So the time complexity of the whole algorithm is $O(nlogn)$.

Q2:

(a) Firstly, we use Merge sort to sort the array and Merge sort is an $O(nlogn)$ algorithm. Merge sort is also very stable.
We assume the sorted array is called T.
For each index i from 1 to n, we will use binary search to check if there exist $(x - T[i])$ in the array where the range of index is from 1 to $(i - 1)$. In this way, we will consider every pair only once. And in case $x = 2 * T[i]$ but $T[i]$ only appears once in the array, the range is set to be T[1,2.. $i - 1$] in every binary search so we can get accurate results.
Thus, this is an $O(nlogn)$ algorithm.

(b) I think using hash table is a good option.
Firstly it is an $O(n)$ algorithm to establish hash table because we need to iterate the array. of n integers and every integer is used at least once.

Secondly for each index i from 1 to n, check if $(x - T[i])$ exists in the hash table. As. searching in hash table requires $O(1)$, for n integers, the time complexity is $O(n)$.
Thus, this is an $O(n)$ algorithm.

Q3:

(a) Assume there are n people present in the party and there is only one celebrity in the party. These people are numbered from 1 to n

Step1: assume person 1 is the celebrity, we can call this person A.

Step2: for the each person k from 2 to n, we just ask person k if he knows A.

Step3: if person k knows A, then k is not the celebrity and A can continue to be a potential celebrity.

Step4: if person k does not know A, then A is not the celebrity and we let this person k be A, which means this person is the potential celebrity . Then we keep asking the question.

In this way, we ask (n-1) questions and we find a potential celebrity A.

Step5: In order to check if A is the real celebrity. We need ask the other (n-1) people if they know A.

In this way, we ask another (n-1)questions to check if everyone knows A. If everyone knows A, then A can be the celebrity with high possibility. If there exists someone who does not know A, then we can say that there is no celebrity in this party.

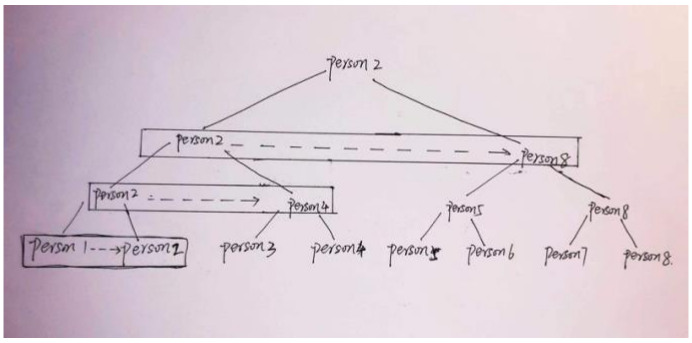
Step6: If we find that everyone knows A, then we need to check if A does not know anyone. So we need to ask A if he knows anyone of the other(n-1)people. We need to ask another (n-1) questions. If the answer is always no, which means A does not know anyone of the other people, then we can say A is the celebrity. If there exists someone is knows by A, then we can that there is no celebrity in this party.

According to 6 steps above, we need at most $3 * (n-1) = 3n - 3$ such questions to find the celebrity, even in the worst case.

(b) We can consider first 4 steps above as a binary tree. we have every single person as a leaf. of the binary tree. For each pair, we can ask the left child a question to find if the left child knows the right child. In this way, we can eliminate haft of the number of persons at each layer and finally we will obtain the potential celebrity A.

The height of tree is $\lfloor \log_2 n \rfloor$, so for step5 and step6 above, we can save $\lfloor \log_2 n \rfloor$ questions.

In this way ,the total number of questions will be no more than $3n - 3 - \lfloor \log_2 n \rfloor$, even in the worst case.



Assume there are 8 people in the party and people are numbered from 1 to 8

We can see from the graph above, the total number of questions to find the potential celebrity B is (n-1) .For check phase we can save $\lfloor \log_2 n \rfloor$ questions which is the height of the tree.

Q4:

(a) $f(n) = (log_2 n)^2 \quad g(n) = log_2(n^{log_2 n}) + 2log_2 n$

First we can get $g(n) = log_2 n * log_2 n + 2log_2 n = (log_2 n)^2 + 2log_2 n$

Secondly we can find that when $c = 1$: $(log_2 n)^2 \le (log_2 n)^2 + 2log_2 n \; for\ all\ n \ge 1$

So we can say that $0 \le f(n) \le g(n) \quad for\ all\ n \ge 1$

We can find when $c = \frac{1}{2}$: $\quad \frac{1}{2} * ((log_2 n)^2 + 2log_2 n) \le (log_2 n)^2 \; for\ all\ n \ge 4$

So we can say that $0 \le \frac{1}{2} * \mathrm{g(n)} \le f(n) \; for\ all\ n \ge 4$

There exist positive constant $c_1 = \frac{1}{2} \quad c_2 = 1 \quad n_0 = 4$ such that

$$0 \le c_1 * g(n) \le f(n) \le c_2 * g(n) \quad for\ all\ n \ge 4$$

Thus, $f(n) = \theta(g(n))$

(b) $f(\mathrm{n}) = n^{100} \quad g(n) = 2^{n/100}$

Since both $f(\mathrm{n})$ and $g(n)$ are monotonically increasing function, so $\log(f(\mathrm{n}))$ and $\log(g(\mathrm{n}))$ can also represent the relationship between $f(\mathrm{n})$ and $g(n)$

$\log(f(\mathrm{n})) = 100 * \log(n) \quad \log(g(n)) = \frac{\log(2)}{100} * n$

From the above equations, since n grows faster than $log\, n$. so $\log(g(n))$ will grows faster than $\log(f(n))$.

Thus, $f(n) = O(g(n))$.

(c) $f(\mathrm{n}) = \sqrt{n} \quad g(n) = 2^{\sqrt{log_2 n}}$

Since both $f(\mathrm{n})$ and $g(n)$ are monotonically increasing function, so $\log(f(\mathrm{n}))$ and $\log(g(\mathrm{n}))$ can also represent the relationship between $f(\mathrm{n})$ and $g(n)$

$$\log_2(f(n)) = \frac{1}{2} * log_2 n \qquad \log_2(g(n)) = 2^{\sqrt{log_2 n}} = \sqrt{log_2 n}$$

According to L'Hˆopital Rule:

$$\lim_{n\to\infty}\frac{\log_2(g(n))}{\log_2(f(n))} = \lim_{n\to\infty}\frac{\frac{d(\log_2(g(n)))}{dn}}{\frac{d(\log_2(f(n)))}{dn}}\lim_{n\to\infty}\frac{1}{\sqrt{log_2 n}} = 0$$

So we can say $\log_2(f(n))$ will grow faster than $\log_2(g(n))$
Thus, $f(n) = \Omega(g(n))$.

(d) $f(\mathrm{n}) = n^{1.001}$ $g(n) = n * log_2 n$
According to L'Hˆopital Rule:

$$\lim_{n\to\infty}\frac{g(n)}{f(n)} = \lim_{n\to\infty}\frac{n * log_2 n}{n^{1.001}} = \lim_{n\to\infty}\frac{log_2 n}{n^{0.001}} = \lim_{n\to\infty}\frac{\frac{1}{nln(2)}}{0.001 * \frac{1}{n^{0.999}}} = \lim_{n\to\infty}\frac{1000}{\ln(2) * n^{0.001}} = 0$$

So $f(\mathrm{n})$ grow faster than $g(n)$
Thus, $f(n) = \Omega(g(n))$.

(e) $f(\mathrm{n}) = n^{(1+\sin(\frac{\pi n}{2}))/2}$ $g(n) = \sqrt{n}$

The range of $\sin\left(\frac{\pi n}{2}\right)$ is [-1,1] thus the range of $(1+\sin\left(\frac{\pi n}{2}\right))/2$ is [0,1], which means the range of $f(\mathrm{n})$ is [1,n]. So $f(\mathrm{n})$ is not monotonical function. $\mathrm{g(n)} = \sqrt{n} = n^{0.5}$ is a monotonical function. So there is no any positive constant c_1, n_0 that can realize

$$0 \le c_1 * g(n) \le f(n) \quad for\ all\ n \ge n_0$$

Moreover there is no any positive constant c_2, n_0 that can realize

$$0 \le f(n) \le c_2 * g(n) \quad for\ all\ n \ge n_0$$

In this case, I think $f(\mathrm{n})$ is not asymptotically comparable with $\mathrm{g(n)}$, all three relations(O, Ω, θ) don't satisfy.

Q5:

(a) $T(n) = 2 * T\left(\frac{n}{2}\right) + n(2 + \sin n)$

According to the Master Theorem, $a = 2, b = 2\ and\ f(n) = n(2 + \sin n)$.
Let $g(n) = n^{log_b a} = n$. The range of $f(n)$ is [n,3n], so for any positive constant $0 \le c_1 \le 1$,
$0 \le c_1 * g(n) \le f(n)$ and for any positive constant $3 \le c_2$, $0 \le f(n) \le c_2 * g(n)$. So we can say that $f(n) = \theta(g(n))$.
According to the second case in the Master Theorem,

$$T(n) = \theta(nlogn)$$

(b) $T(n) = 2 * T\left(\frac{n}{2}\right) + \sqrt{n} + logn$

According to the Master Theorem, $a = 2, b = 2\ and\ f(n) = \sqrt{n} + logn$.

Let $g(n) = n^{log_b a} = n$. In this case , as $\sqrt{n}$ grows faster than $logn$, for $f(n)$, $\sqrt{n} = n^{\frac{1}{2}}$ dominates. So there exist a constant $\varepsilon > 0$, $f(n) = O(n^{log_b a - \varepsilon}) = O(n^{1-\varepsilon})$.

According to the first case in the Master Theorem,

$$T(n) = \theta(n)$$

(c) $T(n) = 8 * T\left(\frac{n}{2}\right) + n^{logn}$

According to the Master Theorem, $a = 8, b = 2$ *and* $f(n) = n^{logn}$.
Let $g(n) = n^{log_b a} = n^3$. Since $logn$ is monotonically increasing function , $logn$ will eventually exceed 3. So there exist a constant $\varepsilon > 0$, $f(n) = \Omega(n^{log_b a+\varepsilon}) = \Omega(n^{3+\varepsilon})$.

For the second requirement, $af\left(\frac{n}{b}\right) = 8 * f\left(\frac{n}{2}\right) = 8 * \left(\frac{n}{2}\right)^{\log\left(\frac{n}{2}\right)}$ $\quad cf(n) = c * n^{logn}$.

Since $af\left(\frac{n}{b}\right)$ *and* $cf(n)$ are monotonically increasing function, $\log\left(af\left(\frac{n}{b}\right)\right)$ and $\log(cf(n))$ can also represent the relationship between $af\left(\frac{n}{b}\right)$ *and* $cf(n)$.

We can get $\log\left(af\left(\frac{n}{b}\right)\right) = \log(8) + log^2\left(\frac{n}{2}\right)$ *and* $\log(cf(n)) = \log(c) + log^2(n)$.

$$\log\left(af\left(\frac{n}{b}\right)\right) = \log(8) + (logn - \log(2))^2$$
$$= log^2(n) - 2 * \log(2) * logn + log^2(2) + \log(8)$$

For the requirement:

$$af\left(\frac{n}{b}\right) \le cf(n)$$
$$\Longrightarrow \log\left(af\left(\frac{n}{b}\right)\right) \le \log(cf(n))$$
$$\Longrightarrow log^2(n) - 2 * \log(2) * logn + log^2(2) + \log(8) \le \log(c) + log^2(n)$$
$$\Longrightarrow 2 * \log(2) * logn \ge log^2(2) + \log(8) - \log(c)$$
$$\Longrightarrow logn \ge \frac{log^2(2) + \log(8) - \log(c)}{2 * \log(2)}$$

Since c is a constant, $\log(c)$ is a constant. As $logn$ is monotonically increasing function, with sufficient large n, $logn$ will eventually be greater or equal to $\frac{log^2(2)+\log(8)-\log(c)}{2*\log(2)}$.

According to third case in the Master Theorem,

$$T(n) = \theta(n^{logn})$$

(d) In this question, the Master Theorem cannot apply here, so we need to try unwinding the recurrence and sum up the linear overheads.

$$T(n) = T(n-1) + n$$
$$T(n-1) = T(n-2) + n - 1$$
$$T(n-2) = T(n-3) + n - 2$$
$$\cdots$$
$$T(2) = T(1) + 2$$
$$T(1) = T(0) + 1$$

As $T(0) = 0$, we can get that $T(n) = \frac{(1+n)*n}{2}$.

Thus, $T(n) = O(n^2)$