COMP3121/9101 Assignment 1
z5014567 Senlin Deng

1. You're given an array A of n integers, and must answer a series of n queries, each of the form: "How many elements a of the array A satisfy $L_k \leq a \leq R_k$?", where L_k and R_k ($1 \leq k \leq n$) are some integers such that $L_k \leq R_k$. Design an O(n log n) algorithm that answers all of these queries.

   Firstly, we use the **merge sort** to sort the array A. The complexity (worst case) of the merge sort is O(*nlogn*).

   Then, we use the **binary search** to find out the largest element which is not exceeding L_k (k denotes the index of each query), this will take O(*logn*) steps (worst case). And also, we can use the same way to find out the smallest element larger or equal to R_k which also takes O(*logn*) steps (worst case). And we can find the indices of those two elements within **constant time**. And we can find out the numbers of elements which satisfy the query by taking the difference of two indices and add one to the result. (e.g. *#elements = index2 – index1 + 1*). This result can be got within **constant time** as long as the indices are located. As a result, it takes O(2*logn*) plus a constant time taken to get the result. The complexity for this step will be O(*logn*) (*e.g. O(2logn) + O(1) = O(logn)*).

   It will take O(n*logn*) to answer all the queries, since we need to repeat the steps for *n* times with *n* queries. Since we only need to sort the array once before we answer all the queries, the total time complexity will be O(n*logn*) plus O(n*logn*) which is still O(n*logn*).

2. You are given an array S of n integers and another integer x.

   (a). Describe an O(n log n) algorithm (in the sense of the worst case performance) that determines whether or not there exist two elements in S whose sum is exactly x.

   Firstly, we use the **merge sort** to sort the array A. The complexity (worst case) of the merge sort is O(*nlogn*).

   Then, go through the array for each element $A_{[i]}$ (where i denotes the index of the element), we use the **binary search** to find whether there exists another element $A_{[j]}$ such that the sum of $A_{[i]}$ and $A_{[j]}$ is *x*. The time complexity to find out such pair is O(*logn*). There are *n* elements in total, so it will take O(*nlogn*) in the worst case to find out such pair.

   So that this algorithm will take O(*nlogn*) steps (e.g. O(*nlogn*) + O(*nlogn*) = O(*nlogn*)).

(b) Describe an algorithm that accomplishes the same task, but runs in O(n) expected (i.e., average) time.

When the array is **sorted**, this problem can be solved in linear time O(*n*).

Assuming the array in sorted in ascending order. In order to find whether or not such pair exists in the array. We set two cursors point to the first and last elements in the array. If the sum of those two elements is less than *x*, we move the front cursor forwards. Otherwise, we move the end cursor backwards until we find out such pair. This will only take linear time in average to solve the problem.

3. You are at a party attended by n people (not including yourself), and you suspect that there might be a celebrity present. A celebrity is someone known by everyone, but does not know anyone except themselves. You may assume everyone knows themselves. Your task is to work out if there is a celebrity present, and if so, which of the n people present is a celebrity. To do so, you can ask a person X if they know another person Y (where you choose X and Y when asking the question).

   (a). Show that your task can always be accomplished by asking no more than 3n – 3 such questions, even in the worst case.

   Firstly, pick two random persons (e.g. Alice and Bob). Then ask the question to Alice that if she know Bob? The answer can either be YES or NO. If the answer is YES, Alice is eliminated since a celebrity knows no one from a party. If the answer is NO, Bob is eliminated since a celebrity is known by everyone.

   As a result, once such a question is asked, one person will be eliminated from being a celebrity. It will take **n-1** steps to find the candidate, who is the last one to be eliminated. Since the candidate might or might not be a celerity. We need to ask the candidate if he knows everyone excluding him in the party. It takes **n-1** steps in the worst case. Also, we need to ask the rest of people if they know this candidate. It takes **n-1** steps as well (e.g. in the worst case, ask everyone that question).

   In total, the task can be solved by asking no more that 3(n-1) times.

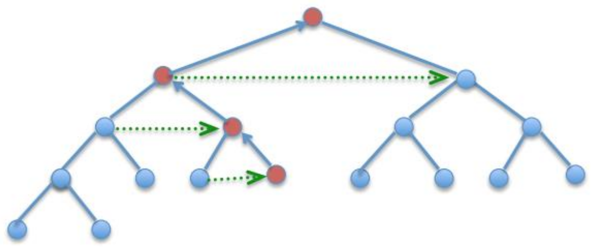
   (b). Show that your task can always be accomplished by asking no more than 3n−blog2 nc−2 such questions, even in the worst case.

   To solve this question, we form the people to a binary tree structure with *n* elements. It is easy to notice that the depth of the tree is $\lfloor \log_2 n \rfloor$.

   To get the candidate of a celebrity, we use the same method from (a). But this time we do the search from bottom leaf left to right. And if the leaf is a potential candidate, we move the candidate to the parent root and record the pair. And It will take **n-1** steps.

   Now we have our candidate, we need to further check if the candidate is a celebrity. From the last step, we can at most recorded $\lfloor \log_2 n \rfloor$ pairs. The worst cast is that the candidate is in the bottom layer of the tree. The pair can be denoted as (candidate, sibling), where we can at most have as much as the depth of the tree.

So that, the algorithm can be further optimised with $\lfloor \log_2 n \rfloor$ steps.



4. Read the review material from the class website on asymptotic notation and basic properties of logarithms, pages 38-44 and then determine if f(n) = Ω(g(n)), f(n) = O(g(n)) or f(n) = Θ(g(n)) for the following pairs. Justify your answers.

$f(n)$	$g(n)$
$(\log_2 n)^2$	$\log_2(n^{\log_2 n}) + 2\log_2 n$
n^{100}	$2^{n/100}$
$\sqrt{n}$	$2^{\sqrt{\log_2 n}}$
$n^{1.001}$	$n \log_2 n$
$n^{(1+\sin(\pi n/2))/2}$	$\sqrt{n}$

(a).

$g(n) = log_2 n * log_2 n + 2log_2 n$

$(log_2 n)^2 < g(n) < 2 * (log_2 n)^2$

$Thus, f(n) = (log_2 n)^2 = \Theta(g(n))$

(b).

We take the logarithm for both side:

$\log(f(n)) = 100 * \log(n) = k_1 * logn$

$\log(g(n)) = \frac{\log 2}{100} * n = k_2 * n$

$so, \log(g(n)) = \Theta(\mathrm{n})\ and\ \log(f(n)) = \Theta(\log n)$

$so, \log(f(n)) < \log(g(n)) + k_3$

$so, f(n) = O(g(n))$

©.

We take the logarithm for both side:

$\log(f(n)) = \frac{1}{2} * log_2 n = k_1 * logn$

$\log\big(g(n)\big) = \sqrt{log_2 n} = k_2 * \sqrt{log_2 n}$
$so, \log\big(f(n)\big) > \log\big(g(n)\big) + k_3$
$so, f(n) = \Omega(g(n))$

(d).
Use the L'Hˆopital Rule:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\frac{d}{dn}*f(n)}{\frac{d}{dn}*g(n)} =$$

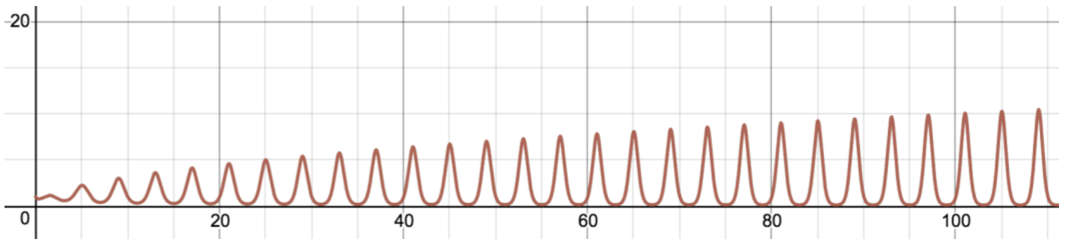
$$\lim_{n\to\infty} \frac{n^{0.001}}{1/\ln 2 + \log_2 n} = \lim_{n\to\infty} \frac{0.001 * n^{-0.999}}{1/(n * \ln 2)} = \lim_{n\to\infty} 0.001 * \ln 2 * n^{0.001} = \infty$$

$\text{Thus}, f(n) = \Omega(g(n))$

(e).
Use the L'Hˆopital Rule:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^{1/2} * n^{\frac{1}{2}*\sin(\pi n/2)}}{n^{1/2}} = \lim_{n\to\infty} n^{\frac{1}{2}*\sin(\pi n/2)}$$

We cannot determine neither the asymptotic upper bound nor lower bound for that natation, since for arbitrarily large *n* the result for the function could be arbitrarily large or close to zero. The following image shows how the function grows.



As a result, it can be found that *f(n)* could be *O(g(n)) either g(n)* could be *O(f(n))*. Thus, the relation of *f(n)* and *g(n)* is **not determinable**.

5. Determine the asymptotic growth rate of the solutions to the following recurrences. If possible, you can use the Master Theorem, if not, find another way of solving it.

(a). T(n) = 2T(n/2) + n(2 + sin n)

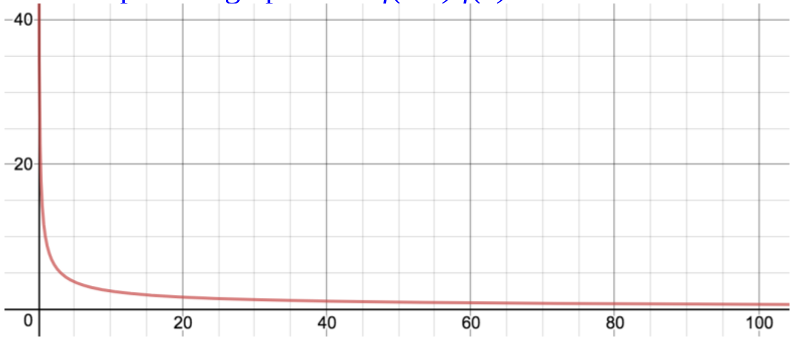
Then $n^{\log_2 2} = n$
Since 2+*sin n* is in range [1,3], which can be regarded as constant,
So that, $f(n) = n * (2 + \sin n) = k * n = \Theta(\mathrm{n})$ (where k denotes some constant)
Thus, condition of case 2 is satisfied; and so,
$T(n) = \Theta\big(n^{\log_2 2} * \log_2 n\big) = \Theta(\mathrm{n} * \log_2 n)$

(b). T(n) = 2T(n/2) + √ n + log n

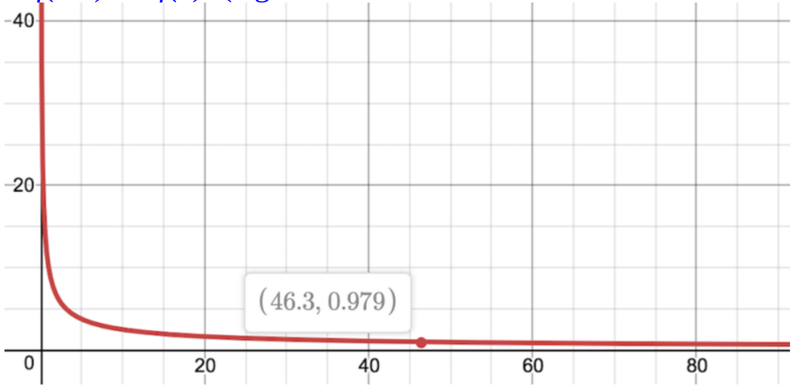
Then $n^{\log_2 2} = n$
From L'Hˆopital Rule:
$\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} k * n^{-1/2} = 0$
So that $\log n = O(\sqrt{n})$
Thus $f(n) = \sqrt{n} + \log n = \Theta(\sqrt{n}) = O(n^{1-\varepsilon})\ for\ any\ \varepsilon < \frac{1}{2}$
Condition of case 1 satisfied,
And so, $T(n) = \Theta(\text{n})$

©. T(n) = 8T(n/2) + $\text{n}^{\log \text{n}}$

Then $n^{\log_2 8} = n^3$
Since $\log n = \Omega(3)\ , so\ n^{\log n} = \Omega(n^3)$
So that $f(n) = n^{\log n} = \Omega(n^{3+\varepsilon})\ for\ any\ \varepsilon > 0$
Now we plot the graph for *a*f(n/b)/f(n)*.



It can be concluded that for sufficiently large n, we can find a *c* which satisfies *a*f(n/b)<c*f(n)*. (e.g. n> 45 we can find some c which is lower that 1.)



So that condition of case 3 satisfied,
And so, $T(n) = \Theta(\text{f(n)}) = \Theta(n^{\log n})$

(d). T(n) = T(n – 1) + n

Since

T(n) = T(n -1) + n

We have that:

T(n – 1) = T(n – 2) + n-1
T(n – 2) = T(n – 3) + n-2
T(n – 3) = T(n – 4) + n-3
……
T(2) = T(1) + 2

Now we sum the all the equations:

$$RHS = \sum_{i=2}^{n} T(i)\,, while\ LHS = \sum_{i=1}^{n-1} T(i) + \sum_{j=2}^{n} j$$

So that we have,

$$\sum_{i=2}^{n} T(i) = \sum_{i=1}^{n-1} T(i) + \sum_{j=2}^{n} j$$

$$\sum_{i=2}^{n} T(i) - \sum_{i=1}^{n-1} T(i) = \sum_{j=2}^{n} j$$

$$T(n) = T(1) + \sum_{j=2}^{n} j = T(1) + \frac{(n+2)*(n-1)}{2} \quad (sum\ of\ arithmetic\ sequence)$$

$\mathrm{T(n)} = \mathrm{T(1)} + k_1 * n^2 + k_2 * n + k_3$ $(where\ k_i\ denotes\ constant)$
Since T(1) takes constant time,
Thus T(n) = Θ (n^2)