COMP3121/9101 19T1 — Assignment 1 (UNSW)

Wenfei Guo , z5135080

1. First of all, using merge sort to sort the array A in ascending order. Because average time complexity of merge sort is O(nlogn).

   Then, using Binary Search in the sorted array A to find the first element not exceed Rk and the first element larger or equal to Ln.

   The difference between these two indices is the answer. Time complexity of each Binary Search algorithm is O(logn). Therefore, algorithm runs in O(nlogn) in total.

   In addition. If Binary Search meets Ln, it is important to check if the preceding element is smaller than Ln. If the preceding element is equal to Ln, you have to go through the next smaller elements until find the first element equal to Ln.

   Due to the same reason, If Binary Search meets Rk, it is important to check if the next element is larger than Rk. If the next element is equal to Rk, you have to go through the next larger elements until find the first element greater than Rk.

2. (a).
   First of all, using merge sort to sort the array S in ascending order. Because worst time complexity of merge sort is O(nlogn).

   For each i from 1 to n, using Binary Search in the sorted array S to know if x-S[i] exists in the sorted array S[1…i-1].

   This algorithm is usable because each pair of elements is considered once and S[i] is not included. Note that in special case x = 2*S[i], S[i]'s value only appeared once in S.

   Therefore, it is an O(nlogn) 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.

   (b).
   First of all, using Spaghetti (Poll) sort to sort the array S in ascending order. Because worst time complexity of Spaghetti (Poll) sort is O(n).

   Then, using a hash map (hash table) to check if elements exist in the array S. Because each insertion and lookup take O(1) expected time. For each i from 1 to n, at index I, we firstly assume S[1…i-1] is in hash map. After that, we check if x−S[i] is in the hash map in O(1), then insert S[i] into the hash map, also in O(1).

3.

(a)

There is at most one person to be celebrity because there is celebrity is known by everyone but does not know anyone except herself/himself.

First, we choose a particular guest to be the only potential celebrity. Set the particular guest g to be one, and for other people, each person i from two to n. We ask the particular guest g whether he/she knows i or not. If g does not know any i, then g still can be the celebrity in the party and still remain as the potential celebrity. If g knows any i, then we treat i as new potential celebrity g, the old g cannot be celebrity and will be treat as i. Therefore, we can find out that the only final g is possibly a celebrity with (n-1) questions.

There is a chance that the newly set g is also not a celebrity. j represents guests that are not considered to be potential celebrity. Thus, we need to ask (n-1) times “Does j know g?” questions. If any answer is no, there is no celebrity in the party. If all answers are yes, there is still a chance for g to be the celebrity.

After that, we need to ask (n-1) times “Does g know j?” questions. If any answer is yes, there is no celebrity in the party. If all answers are no, the only celebrity in the party is found.

Therefore, the algorithm needs at most (n-1)+ (n-1)+ (n-1) = 3n-3 questions.

(b).

We treat the n people attended the party as leaves of a balanced full tree. The reasons we choose balanced full tree are every node in balanced full tree have two or zero children and the depth of tree can be squeeze as small as possible.

To achieve the goal, we add a = [log2 n] and make the perfect binary tree with 2^a <= n leaves.

If 2^a < n add two children to each of the leftmost (n−2^a) leaves of the perfect binary tree. After that we can get 2(n – 2^a) + (2^a − (n – 2^a)) = 2*n−2^(a+1) +2^a−n+2^a = n leaves. We can get each leave’s depth is equal or larger than [log2 n] and each leave owns a pair.

Based on the answer in (a), if we ask each leave pair whether the left node knows right node or not, we can get the answer and put the possible celebrity one level closer to root.

It takes (n-1) times of question to find a possible celebrity. And for the verification we can save one question on each level which turns out to be [log2 n] questions saved. The reason is we can use the answer that we get through the path that the possible celebrity went through the tree again.

Based on above, we can say (3*n-3-[log2 n]) questions which is less than the required bound (3*n-2-[log2 n]) suffice.

4.

a. f(n) = (log2 n)^2 g(n) = log2(n^log2 n) + 2*log2 n

Answer: f(n) = Θ (g(n))

f(n) = (log2 n)^2, the derivative of it is: $\frac{\partial \mathrm{f}}{\partial \mathrm{n}} = \frac{2*logn}{(log2)^2 n}$

g(n) = log2(n^log2 n) + 2*log2 n: the derivative of it is: $\frac{\partial \mathrm{g}}{\partial \mathrm{n}} = \frac{2*\log n+2}{log2\ n}$

use L'Hˆopital Rule:

$$\lim_{\mathrm{n}\to\infty}\left(\frac{f(n)'}{\mathrm{g(n)}'}\right)$$

= 1

Based on L'Hˆopital Rule, we can find that f(n) and g(n) have same growth rate. Thus, we can determin that **f(n) = Θ (g(n))** .

b. f(n) = n^100 g(n) = 2^(n/100)

Answer: f(n) = O(g(n))

f(n) = n^100, the derivative of it is: $\frac{\partial \mathrm{f}}{\partial \mathrm{n}} = 100 * x^{99}$

g(n) = 2^(n/100): the derivative of it is: $\frac{\partial \mathrm{g}}{\partial \mathrm{n}} = \frac{1}{100}\ln(2)\, e^{\frac{\ln(2)x}{100}}$

use L'Hˆopital Rule:

$$\lim_{\mathrm{n}\to\infty}\left(\frac{f(n)'}{\mathrm{g(n)}'}\right)$$

= 0

Based on L'Hˆopital Rule, Based on L'Hˆopital Rule,

we can find that growth rate of g(n) is faster than the growth rate of f(n). Thus, we can determin that **f(n) = O (g(n))** .

c. f(n) = √n g(n) = 2^√log2 n

Answer: f(n) = O(g(n))

f(n) = √n, the derivative of it is: $\frac{\partial \mathrm{f}}{\partial \mathrm{n}} = \frac{1}{2\sqrt{n}}$

g(n) = 2^√log2 n: the derivative of it is: $\frac{\partial \mathrm{g}}{\partial \mathrm{n}} = \frac{\sqrt{log2} * e^{\sqrt{log2} * \sqrt{logn}}}{2n\sqrt{logn}}$

use L'Hˆopital Rule:

$$\lim_{\mathrm{n}\to\infty}\left(\frac{f(n)'}{\mathrm{g(n)}'}\right)$$

= 0

Based on L'Hˆopital Rule, Based on L'Hˆopital Rule,

we can find that growth rate of g(n) is faster than the growth rate of f(n). Thus, we can determin that **f(n) = O (g(n))** .

d. n^1.001 g(n) = n*log2 n

Answer: f(n) = Ω (g(n))

f(n) = n^1.001, the derivative of it is: $\frac{\partial \mathrm{f}}{\partial \mathrm{n}} = 1.001 * n^{0.001}$

g(n) = n*log2 n: the derivative of it is: $\frac{\partial \mathrm{g}}{\partial \mathrm{n}} = log2(x) + \frac{1}{\ln(2)}$

use L'Hˆopital Rule:

$$\lim_{\mathrm{n}\to\infty}\left(\frac{f(n)'}{\mathrm{g(n)}'}\right)$$

= ∞

Based on L'Hˆopital Rule,

we can find that growth rate of f(n) is faster than the growth rate of g(n). Thus, we can determin that **f(n) = Ω (g(n))** .

e. n^((1+sin(πn/2))/2) g(n) = √n

It does not have solution

use L'Hˆopital Rule:

$$\frac{f(n)}{g(n)} = \frac{n^{\frac{1+\sin\left(\frac{\pi n}{2}\right)}{2}}}{\sqrt{n}} = \frac{n^{\frac{\sin\left(\frac{\pi n}{2}\right)}{2}}}{1}$$

As it well known, sin has no limit

Therefore, It does not have solution.

5.

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

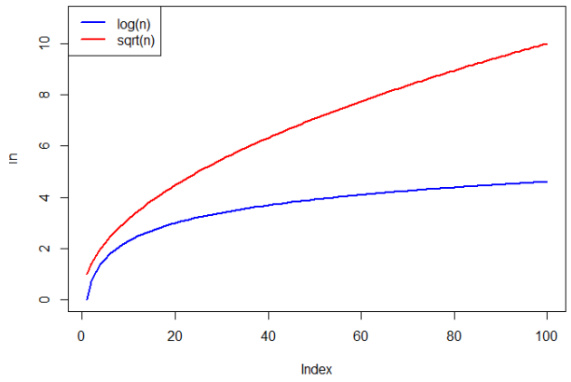
Answer: Θ (n logn)

- use the Master Theorem
- a = 2, b = 2
- then n^(log b a) = n^(log2 2) = n;
- thus f(n) = n (2 + sin n) = Θ(n) = Θ(n^(log2 2)).
- Thus, condition of case 2 is satisfied; and so,
- T(n) = Θ((n ^ (log2 2))*log2 n) = Θ (n logn)

(b). T(n) = 2T(n/2) +√n + log n

Answer: Θ (n)

- use the Master Theorem
- a = 2, b = 2
- then n^(log b a) = n^(log2 2) = n;
- based on the graph below, √n grows faster.
- thus f(n) = √n + log n = O(n^(log 2(2-ε))) for ε<2.
- Thus, condition of case 1 is satisfied; and so,
- T(n) = Θ((n ^ (log2 2))) = Θ (n)



(c). T(n) = 8T(n/2) + n^ log n

Answer: Θ (n^logn)

- use the Master Theorem
- a = 8, b = 2
- then n^(log b a) = n^(log2 8) = n^3;
- f(n) = n^log n = Ω (n^(3+ε))
- n^(log(n))
  =(k^(logk(n)))^(log(n))
  = k^(k*(log(n)^2)).
  Since (log(n))^2 < n for n large enough, then this means that n^(log(n)) will grow slower than k^n
- Therefore, af(n/b) = 8f(n/2) <= cf(n) for some 0<c<1
- Thus, condition of case 3 is satisfied;
- T(n) = **Θ (f(n))**
  = **Θ (n^logn)**

(d). T(n) = T(n−1) + n

Answer: Θ (n^2)

Use algebraic substitution

T(n) = T(n−1) + n

= T(n−2) + (n-1) + n

…

$$= \Theta(1) + \sum_{i=1}^{n} i$$

= Θ(1) + (1+n)n/2

= Θ(1) + ½*n^2 + ½ *n