Algorithm 2021 Fall

Writing Assignment 2

B06702064 會計五 林聖硯

“No collaborator”

Problem1

參考資料: https://walkccc.me/CLRS/Chap06/Problems/6-2/ (https://walkccc.me/CLRS/Chap06/Problems/6-2/)

a.

- parent of i-th child is $\lfloor \frac{i-2}{d} + 1 \rfloor$
- j-th child of i-th parent is $d(i-1)+j+1$
  (Reason: Observe that di is the **second last** child of i-th parent, $di+1$ is the last child of i-th parent. Thus, $(di+1)-(d-j) = d(i-1)+j+1$ is the j-th child of i-th parent)

b. Since each parent has d children, the height of a d-ary heap is $\Theta(\log_d n)$.

c. The most important operation in a heap is definitely the HEAPIFY function, I therefore implement it first.

```
d-ARY-PARENT(i)
    return floor((i - 2) / d + 1)
```

```
d-ARY-CHILD(i, j)
    return d(i - 1) + j + 1
```

Idea: same as max-heapify in binary heap, we need to find the largest child and swap it with the parent and recursively call the function until the heap restores.

```
d-ARY-MAX-HEAPIFY(A, i)
    largest_idx = i
    for k = 1 to d
        if d-ARY-CHILD(k, i) ≤ A.heap_size and A[d-ARY-CHILD(k, i)] > A[largest_idx]
            largest_idx = d-ARY-CHILD(k, i)
    if largest_idx != i
        swap(A[i], A[largest_idx])
        d-ARY-MAX-HEAPIFY(A, largest_idx)
```

```
d-ARY-HEAP-EXTRACT-MAX(A)
    if A.heap-size < 1
        print("Empty heap")
    max = A[1] #Root is the largest element in max hep
    A[1] = A[A.heap-size] #Delete it with the last element in heap
    A.heap_size = A.heap-size - 1
    d-ARY-MAX-HEAPIFY(A, 1) #Restore heap property
    return max
```

The time complexity of the extract-max algorithm is bounded by the height of d-ary heap (same as the binary heap), which is $O(d \log_d n)$.

d. The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

```
d-ARY-MAX-HEAP-INSERT(A, key)
    A.heap_size = A.heap_size + 1
    A[A.heap_size] = key
    i = A.heap_size
    while i > 1 and A[d-ARY-PARENT(i) < A[i]]
        swap(A[i], A[d-ARY-PARENT(i)])
        i = d-ARY-PARENT(i)
```

e. The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

```
d-ARY-INCREASE-KEY(A, i, key)
    if key < A[i]
        print ("new key is smaller than current key")
    A[i] = key
    #Only need to bubble up here, since key > A[i]
    while i > 1 and A[d-ARY-PARENT(i) < A[i]]
        swap(A[i], A[d-ARY-PARENT(i)])
        i = d-ARY-PARENT(i)
```

Problem2

參考資料: https://walkccc.me/CLRS/Chap07/Problems/7-4/ (https://walkccc.me/CLRS/Chap07/Problems/7-4/)

a. 在trail-recursive-quicksort的第五行內，使用了$p = q + 1$代替了原本的第二次recusively quick sort，但其實我們可以發現用這樣子取代的效果與原本的quick sort一模一樣。在trail-recursive-quicksort的第五行內，$p = q + 1$後會再call一次trail-recursive-quicksort，直到$p < r$。

b. 當今天array原本就是sort好的狀態，stack depth會達到最大，也就是$\Theta(n)$，因為在每次partition完後，q永遠回傳的是array裡面最後一個數值，導致任務被分成0與n-1長度的subproblem。也就是說，這個function要被recusively call n-1次才會解完，加上一開始call的function，總共有n個information會被push進去stack。

Problem3

參考資料: https://ita.skanev.com/08/problems/02.html (https://ita.skanev.com/08/problems/02.html)

b. Use partition in quick sort since there's only two values in an array.
c. Insertion sort
e.

```
MODIFIED-COUNTING-SORT(A, k)
    //let C[1 . . k] be a new array
    //which stores the cumulative number of elements less than or equal to i in C[i]
    for i = 1 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    for i = 2 to k
        C[i] = C[i] + C[i - 1]
    //my trick: use a counter `p` to record the current position
    p = 1
    for i = 1 to k
        for j = p to C[i] //These k inner loops run at most n times in total
            A[p] = i //Place it back to Array A
            p = p + 1
```

Space complexity: $O(k)$ (Since I design it as a in-place sorter)
Time complexity: $k + n + k + n = 2(n + k) \in O(n + k)$
This algorithm is not stable since the array C has no information about which element stores in the array first.

Problem4

參考資料: https://walkccc.me/CLRS/Chap08/Problems/8-4/ (https://walkccc.me/CLRS/Chap08/Problems/8-4/)

a. Use brute forth method. Select a red jug and compare it with a blue jug until you find the one that has the same volume. This algorithm will use at most $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$ comparisons.

b. We can make a decision tree which represents comparisons made between blue jugs and red jugs and we are interested in when one jug is greater than, less than, or equal in size to another jug, so the tree should have 3 children per node. Since there are $n!$ leaf nodes in the tree (which is the combination of n red jug and n blue jug), the decision tree must have height $\log_3(n!)$. From sterling's approximation, $n! \approx (\frac{n}{e})^n$, $h = \Omega(nlgn)$. Thus, an algorithm must make $\Omega(nlgn)$ comparisons to reach any leaf.

c.

參考資料: https://ita.skanev.com/08/problems/04.html (https://ita.skanev.com/08/problems/04.html)

Idea:

1. Choose, at random, a red jug for pivot, use $O(1)$
2. Pick a corresponding blue jug, use $O(n)$
3. Partition the blue jugs around the red pivot and put the blue pivot in place, use $O(n)$
4. Partition the red jugs around the blue pivot and but the red pivot in place, use $O(n)$

```
Partition-Red(R, p, r, x)
    i = p - 1
    for j = p to r
    if R[j] <= x
        i = i + 1
        exchange R[i] with R[j]
    return i

Partition-Blue(B, p, r, x)
    i = p - 1
    for j = p to r
        if B[j] <= x
            i = i + 1
            exchange B[i] with B[j]
    return i

Match-Jugs(R, B, p, r)
if p < r
    k = Random(p, r)
    q = Partition-Red(R, p, r, B[k])
    Partition-Blue(B, p, r, A[q])
    Match-Jugs(R, B, p, q - 1)
    Match-Jugs(R, B, q + 1, r)
```

The analysis of expected number of comparisons is the same as that of RANDOMIZED-QUICKSORT given on pages 181-184 of the book. Althoug, we are running the procedure twice so the expected number of comparisons is doubled, but this is absorbed by the big-O notation. In the worst case, we pick the largest jug each time, which gives a n-1:0 split.
$T(n) = T(n-1) + T(0) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)$ (By proof in the textbook)

Problem5

參考資料: https://walkccc.me/CLRS/Chap09/Problems/9-1/ (https://walkccc.me/CLRS/Chap09/Problems/9-1/)

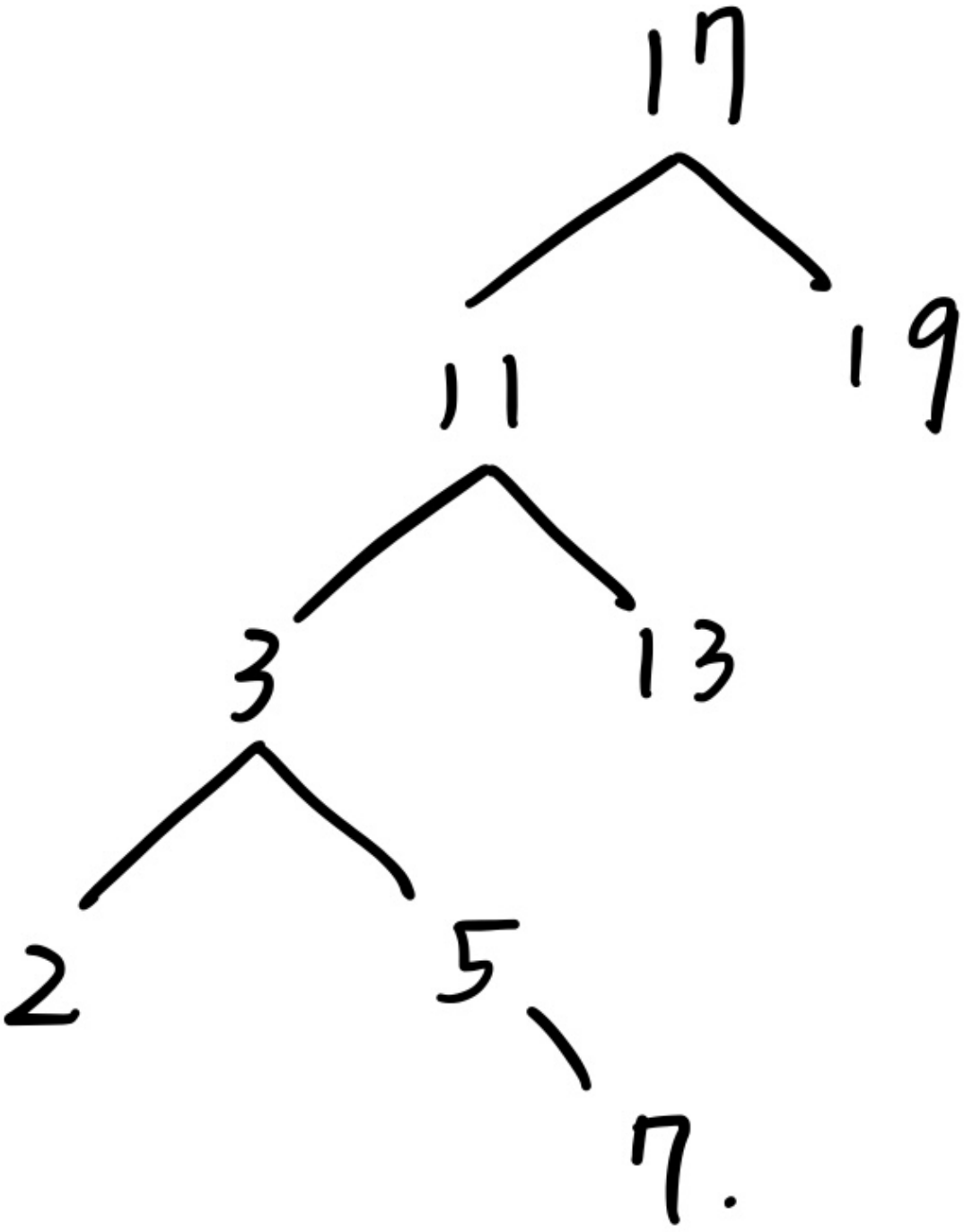
a. We can sort with any of the $O(n \lg n)$ algorithms (heap sort or quick sort) and then just take the first i elements linearly.
The total running time will be $O(n \lg n + i)$.

b. The running time of building a max-priority queue (using a heap) from the numbers is $O(n)$, and the running time of each call EXTRACT-MAX is $O(\lg n)$. Therefore, the total running time is $O(n + i \lg n)$.

c. We can use the SELECT algorithm from the chapter. The running time of finding and partitioning around the ith largest number is $O(n)$. Then,We can sort the i largest numbers with any of the $O(n \lg n)$ algorithms (heap sort or quick sort). Thus, the total time complexity is $O(n + i \lg i)$

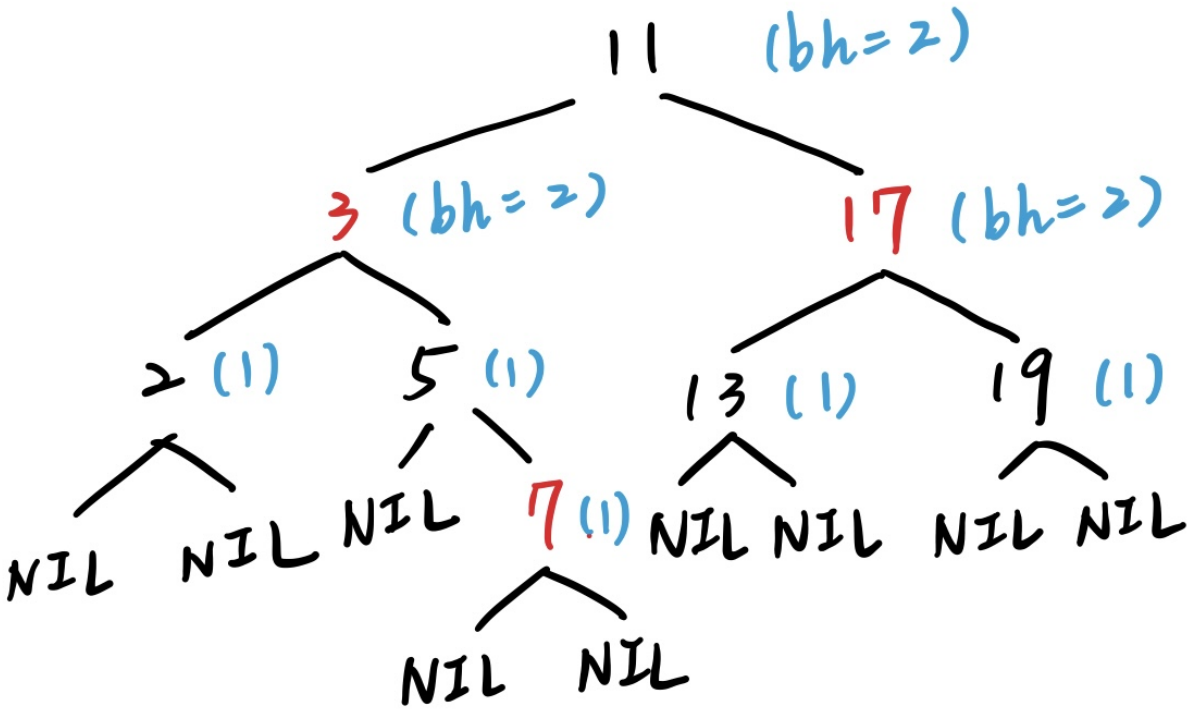
Problem6

a.



b. The tree is not balanced, and the fifth property is thus not possible to be satisfied. (Fifth property: every simple path from a node to a descendent leaf contains the same # of black nodes)

c. Do right rotation at key 11



Problem7

參考資料: http://guanzhou.pub/files/CLRS/CLRS-Part Answer.pdf (http://guanzhou.pub/files/CLRS/CLRS-Part%20Answer.pdf)

Let $D(x) = \sum_{i=1}^{x} d_i$, $D(0) = 0$, $D(n) = D$

Let $c(i, j)$ denote the minimum cost of the first ith month**s** that manufactureed j machines.(前i個月中，**總共**製造j台機器)

We can state the problem as the following recurrence formula:

$$c(i, j) = min_{j \geq k \geq D(i-1)}\{c(i-1, k) + max\{0, j-k-m\} \times c\} + h(j - D(i))$$

for $D \geq j \geq D(i)$

The DP table has size $O(nD)$ and we need to run through and take the minimum cost from D many options when computing a particular subproblem. Thus, the running time will be $O(nD^2)$

Problem8

a.

Subproblem: We have the maximum subarray ending at the previous position (say index $i-1$), and we need to take the maximum the following two values. **The maximum subarray ending at the previous position plus the value in Array[i]** and **the value of Array[i]** itself. (i.e.

$r_n = max(r_{n-1} + A[i], A[i]))$

Optimal substructure: If r_n is the optimal solution for an array of length n, then solution to its subproblem (r_{n-1}) is also an optimal solution of length $n-1$.

Prove optimal substructure by contradication.
If r_{n-1} is NOT an optimal solution, then we can replace it by other optimal solution (which contains $A[n-1]$) and thus get a value greater than r_n. This violates our assumption that r_n is an optimal cut. QED

b.

The recursion formula can be stated as:

$$r_n = max(r_{n-1} + A[i], A[i])$$

c.

```
MAXIMUM-SUB-ARRAY(A)
    let r[1 .. n] be a new array //r[i] record maximum subarray value in position i
    r[1] = A[1]
    max = r[1]
    for i = 2 to A.length
        r[i] = max(r[i-1] + A[i], A[i])
        if r[i] > max //record best solution
            max = r[i]
    return max
```

Time complexity is $\Theta(n)$.

Problem9

a.

- Case0: $kj \in C$, $k \notin [i, j]$, $M(i, j) = M(i, j-1)$
- Case1: $kj \in C$, $k \in (i, j)$, $M(i, j) = M(i, k-1) + 1 + M(k+1, j-1)$ (Compute M from i to k and from k to j, and then plus one back for arc jk)
- Case2: chord $ij \in C$, $M(i, j) = M(i+1, j-1) + 1$

Thus, the recurrence formula can be written as
(1) $M(i, j) = max\{M(i, j-1), M(i, k-1) + 1 + M(k+1, j-1)\}$, if $k \neq i$
(2) $M(i, j) = M(i+1, j-1) + 1$, if $k = i$

b.

```
MAXIMUM-PLANAR-SUBSET(C, N)
for i = 0 to 2N - 1
    M(i, i) = 0 //DP table
for l = 1 to 2N - 1
    for i = 0 to 2N - l - 1
        j = i + l
        if chord kj in C and k not in [i, j]
            M(i, j) = M(i, j - 1)
        else if k == i
            M(i, j) = M(i + 1, j - 1) + 1
        else
            max(M(i, j-1),  M(i, k-1) + 1 + M(k+1, j-1))
```

The time complexity of the algorithm is

$(2N+1)+(2N-1+2N-2+\cdots+1+0)=(2N+1)+\frac{2N(2N-1)}{2}\in O(N^2)$