MTH407:Algorithms and Complexity

Solutions to End-Semester Examination on 27th April 2025

Question 1

Consider the following Python program:

```
def divfn(n):
    d1 = 1
    d2 = n
    ans = 0
    while d1 < d2:
        if d1*d2 == n:
            ans += d1+d2
        d1 += 1
        d2 = n//d1
    if d1*d2==n and d1==d2:
        ans += d1
    return ans
```

a. Calculate the function $M(n)$ that counts the number of multiplications in this algorithm as a function of the input n.
b. If the input n is specified using its digits, is the running time of this program a polynomial in the size of the input?

Solution to a (3 marks). We note that, in the `while` loop, $d_1 < d_2$ and $d_1 \cdot d_2 \leq n$. Thus, $d_1^2 < n$ in the loop. There is one multiplication (in the `if` check) in each iteration of the loop. Note that d_1 is a integer ≥ 1.

There is one further multiplication for checking the `if` clause. It follows that

$$M(n) = 1 + |\{d_1 \in \mathbb{N} : d_1^2 < n\}| = 1 + \lfloor \sqrt{n-1} \rfloor$$

Solution to b (2 Marks). The number of digits of n is roughly $k = \log_{10}(n)$. On the other hand we note that $M(n)$ is roughly $\sqrt{n}$ which is $\Theta(10^{k/2})$. Thus, the running time of the algorithm is *not* polynomial in the size of the input.

Question 2

Order the following functions in increasing order in terms of asymptotic behaviour as n goes to infinity.

$$\binom{n}{3} \; ; \; n! \; ; \; 2^{n^{1/2}} \; ; \; n + n\sin(n) \; ; \; \sum_{k=1}^{\infty} 1/k^{n+2} \; ; \; \sum_{k=1}^{n} 1/k$$

Solution (1 Mark for each correct position).

1. $\binom{n}{3} = n(n-1)(n-2)/6$ is $\Theta(n^3)$.
2. We have

$$n\log(n) \geq \log(n!) = \sum_{k=1}^{n} \log(k) \geq (n/2)\log(n/2)$$

   So $\log(n!)$ is $\Theta(n\log(n))$.
3. $\log(2^{n^{1/2}})$ is $\Theta(n^{1/2})$ which grows faster than $\log(P(n))$ for any polynomial function $P(n)$ of n.
4. $\sin(n)$ is bounded, so $n + n\sin(n)$ is $\Theta(n)$.
5. $\sum_{k=1}^{\infty} 1/k^{n+1}$ is a *decreasing* function of n which is bounded below by 1. So it is $\Theta(1)$.
6. $\sum_{k=1}^{n} 1/k$ is $\Theta(\log(n))$.

Thus, the order is

$$\sum_{k=1}^{\infty} 1/k^{n+1} \ ; \ \sum_{k=1}^{n} 1/n \ ; \ n + n\sin(n) \ ; \ \binom{n}{3} \ ; \ 2^{n^{1/2}} \ ; \ n!$$

Question 3

Given a "black-box" random generator $r()$ that outputs 0 or 1 with probability 1/2 each. Assume that all calls to $r()$ are independent events.

a. Give an algorithm that uses $r()$ to output 0, 1 or 2 with probability 1/3 each. (Hint: Consider three equally probable events made out of calls to $r()$.)

b. Calculate the expected number of calls to $r()$ of the algorithm in (a).

Solution to a (3 Marks). We make 2 calls to $r()$ which gives the events $(0,0),(0,1),(1,0),(1,1)$ with equal probabilities 1/4.

```
def r3():
    a, b = r(), r()
    while a==1 and b==1:
        a, b = r(), r()
    return a+2*b
```

Solution to b (2 Marks). The probability that we get $(1,1)$ is 1/4, thus the probability that we get it exactly k times is $(1/4)^k$. When this happens $r()$ is called $2(k+1)$ times. Thus, the expected number of calls to $r()$ is

$$E = \sum_{k=0}^{\infty} \frac{2(k+1)}{4^k}$$

Now

$$3E = 4E - E = 2 \cdot 4 + 2 \cdot \sum_{k=0}^{\infty} \frac{1}{4^k} = 2\left(4 + \frac{4}{3}\right)$$

So $E = 32/9$.

Question 4

Given a list of incomes of all people in a city, we wish to find the lowest 25% of the earners. (For simplicity assume that everyone has different incomes.)

a. If one sorts the complete list in order to do this, what is the worst case running time as a function of the number of people?

b. Is there an algorithm that is more efficient than (a) to do this? If so, give a brief description.

Solution to a (2 Mark). The best approach to sorting takes $\Theta(n \log(n))$ steps on a list of size n.

Solution to b (1+2 Marks). We have learned that we can select the k-th element of a list of size n in $O(n)$ steps using the median-of-medians approach to selection.

So we can run this algorithm for $k = n/4$. After finding this income, we can run through the list (with at most k comparisons) to find all the incomes less than or equal to this.

Some students have used a randomised algorithm based on random choice of pivot. The list then is separated into elements less than the pivot and more than the pivot in n steps. Suppose the shorter list has k elements. If $k > n/4$, we repeat this with the shorter list. Otherwise, we look for the smallest $n/4 - k$ elements of the larger list by the same method. The *expected* running time of this is $O(n)$.

Question 5

a. We are given a large database of student data including their registration number. What data structure will you use to manage this data so that we can efficiently find the data of a student associated with a certain registration number. (Assume that the student is uniquely determined by their registration number.)

b. What is the complexity (as a function of size n of the student body) of adding a student to the database?

c. A large class of m students is to be added to the database, is there a quicker way than inserting students one-by-one? Justify your answer.

Solution to a (1 Mark). We will use a balanced binary search tree (such as AVL or Red-Black) using the registration number as the comparison key. (We assume that there is a function `cmp` that can put a strict order on registration numbers.)

Solution to b (2 Marks). The time to insert a node in an AVL tree has worst case running time $O(\log(n))$ where n is the size of the database.

Solution to c (2 Marks). The worst case time to insert m nodes in an AVL tree is $O(m \log(n+m))$.

The worst case time to create an m node AVL tree is $O(m \log(m))$.

The worst case time to *merge* AVL trees of size m and n (with $m \leq n$) is $O(m \log(1 + (n/m))$.

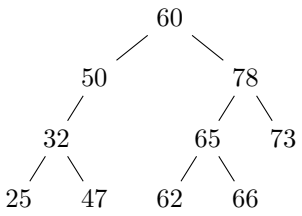
Thus, the times of the following two methods are roughly the same:

1. Insert one by one.
2. Create a new AVL tree of the new class and then merge it with the AVL tree of all earlier students.

In practical scenarios this one method may be better than the other based on additional facts about the registration numbers being merged.

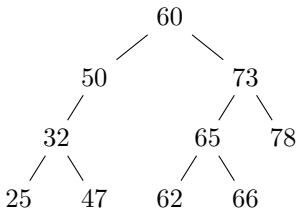
Question 6

Consider the following tree:



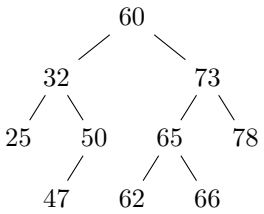
a. Convert this into a *binary search tree* (BST) with at most one swap of two nodes.
b. Convert the resulting BST into an AVL tree with at most one rotation.

Solution to a (2 Marks). We swap the nodes 73 and 78 to get the binary search tree.



Solution to b (3 Marks). We note that the tree below the node 50 is not AVL as its left sub-tree has depth 2 and right sub-tree has depth 0.

We perform a right-rotation at the edge joining 32 and 50 to get the AVL binary search tree. (Note that the right sub-tree of 50 is empty!)



Question 7

Consider a function $g(n)$ defined recursively as follows:

$$g(0) = g(1) = g(2) = 1 \ ; \ g(n) = g(n-1) + g(n-2)^2 + g(n-3)^3 \text{ for } n \geq 3$$

a. Which algorithmic strategy will you use to design an algorithm to compute $g(n)$ efficiently using this formula?
b. Write an efficient algorithm to calculate $g(n)$.

Solution to a (2 Marks). We will use Dynamic Programming as there are a lot of overlapping sub-problems.

Note that there does not appear to be an alternative to computing $g(m)$ for all $m < n$ in order to compute $g(n)$.

Solution to b (3 Marks). The following program implements dynamic programming.

```
def g(n):
    ans={}
    ans[0]=1
    ans[1]=1
    ans[2]=1
    for r in range(3,n+1):
        ans[r] = ans[r-1] + ans[r-2]**2 + ans[r-3]**3
    return ans[n]
```

Note that when n is large, the *lookup* of value `ans[n]` could also take time! However, since we are using Python dictionaries, the lookup time is $O(\log_2(n)$. (This aspect has not been taken into account while grading the answer papers.)

Question 8

Consider the function recursively defined as follows:

$$f(0) = f(1) = 1 \ ; \ f(n) = 2f(n-1) + 3f(n-2) \text{ for } n \geq 2$$

a. Give an efficient algorithm to compute the value of $f(n)$ for large n.
b. Estimate the running time $T(n)$ of the algorithm as a function of the input n.

Solution to a (3 Marks). Define the matrix

$$F(n) = \begin{pmatrix} f(n+2) & f(n+1) \\ f(n+1) & f(n) \end{pmatrix}$$

We note that for $n = 0$, we have

$$F(0) = \begin{pmatrix} 5 & 1 \\ 1 & 1 \end{pmatrix}$$

and for $n \geq 1$, we have

$$\begin{pmatrix} f(n+2) & f(n+1) \\ f(n+1) & f(n) \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} f(n+1) & f(n) \\ f(n) & f(n-1) \end{pmatrix}$$

Thus, by induction n, we see that:

$$\begin{pmatrix} f(n+2) & f(n+1) \\ f(n+1) & f(n) \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 1 & 0 \end{pmatrix}^n \cdot \begin{pmatrix} 5 & 1 \\ 1 & 1 \end{pmatrix}$$

We can thus calculate $f(n+2)$ efficiently using the method for calculating n-th powers in $\log_2(n)$ steps.

```
def matmul(a,b):
    c = (a[0]*b[0]+a[1]*b[2], a[0]*b[1]+a[1]*b[3], \
        a[2]*b[0]+a[3]*b[2], a[2]*b[1]+a[3]*b[3])
    return c

def fmat(n):
    return matpow(n,(2,3,1,0),(5,1,1,1))

def matpow(n,a,b):
    if n%2 == 1:
        b = matmul(a,b)
```

```
    if n <= 1:
        return b
    else:
        a2 = matmul(a,a)
        return matpow(n//2, a2, b)

def f1(n):
    if n <= 1:
        return 1
    elif n == 2:
        return 5
    else:
        return fmat(n-2)[0]
```

The alternative (which is not as efficient) is to use:

```
def f2(n):
    ans=[-1]*(n+1)
    for r in range(2,n+1):
        ans[r] = ans[r-1]*2 + ans[r-2]*3
    return ans[n]
```

The least efficient is the recursive program:

```
def f3(n):
    if n <= 1:
        return 1
    else:
        return f3(n-1)*2 + f3(n-2)*3
```

Some students have observed the formula $f(n) = (3^n + (-1)^n)/2$. Even in this case, the most efficient approach is to use the powering method (like `f1`) to calculate 3^n. Note that this uses a mathematical accident which is not applicable if 2 and 3 above are replaced by other constants!

Solution to b (2 Marks). We note that the values of $f(n)$ grow *faster* than Fibonacci numbers. Hence, they grow exponentially in n. Thus multiplication of such numbers is the most expensive operation. So we count multiplications to estimate the running time.

We note that `f3` makes 2 multiplications for each call and computes $f(m)$ for *all* $m \leq n$ multiple times (in fact, the values $f(m)$ for $m < n/2$ are calculated an exponential number of times as a function of n!). Thus, this is the least efficient algorithm.

Next, we note that `f2` makes 2 multiplications for each call and computes $f(m)$ exactly *once* for all $m \leq n$. Thus, it has roughly $2n$ multiplications. (Actually, it is $2(n-1)$ multiplications.)

Finally, we note that `f1` only involves multiplications when `matmul` is called. Each such call has 8 multiplications. Moreover, `matmul` is called 2 times (in the worst case) during the calculation of `matpow`. Thus, each call to `matpow` results in 16 multiplications. To calculate `fmat`$(n-2)$ we see that `matpow` is called roughly $\log_2(n)$ times. Thus, the number of multiplications as roughly $16\log_2(n)$.

A more careful estimate would take into account the fact that multiplication of smaller numbers takes less time than multiplications of bigger numbers.