ALGORITHMS AND COMPLEXITY

1. a) Give a tight bound for each of the following recurrence relations, or explain why it's not possible to do so.

Carefully justify your answers.

i) $T(n) = T(n/3) + T(n/3) + T(n/3) + 1$ [3]

Answer:
$d = 0, a = 3, b = 3$. So $\log_b a = \log_3 3 = 1 > 0 = d$ and the complexity is $O(n^{\log_b a}) = O(n)$.

ii) $T(n) = 9T(n/3) + 3n^2$ [3]

Answer:
$d = 2, a = 9, b = 3$. So $\log_b a = \log_3 9 = 2 = d$ and the complexity is $O(n^d \log n) = O(n^2 \log n)$.

iii) $T(n) = T((n/2)^2)/2 + 1/n$ [4]

Answer:
Firstly, this recurrence does not fit the conditions of the Master Theorem. Secondly, it cannot give a well-defined bound as $(n/2)^2 > n$. So, for example $T(8) = T(16)/2 + 1/8$, $T(16) = T(64)/2 + 1/16$, etc.

b) Give a bound on the complexity of the following operations, and justify your answers in terms of a specific algorithm. *The algorithm code does not need to be given, unless it is needed to support your argument. You do not have to give the best possible algorithm.*

i) Multiplication of two $n \times n$ matrices. [4]

Answer:
If $M = A \cdot B$ then $M_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$. Computing each item M_{ij} takes $\Theta(n)$ operations and there are n^2 such operations so the total number of operations is $\Theta(n^3)$. This gives an upper bound of $O(n^3)$ on matrix multiplication.

ii) Sorting an array of length n. [6]

Answer:
One option is merge sort, which is a form of divide and conquer. Divide the array into two parts of length $n/2$. Conquer these by recursion. Combine by iterating simultaneously over the subarrays, taking the smaller of the two parts at each stage. If total complexity is $T(n)$ then each subarray will have $T(n/2)$ steps and the combination step is $O(n)$ so $T(n) = 2T(n/2) + O(n)$. Applying the master theorem gives $a = b = 2, d = 1$ so $\log_b a = \log_2 2 = 1 = d$ and therefore $T(n) = O(n \log n)$.

Master Theorem. If $T(n)$ satisfies

$$T(n) = a\,T(\,n/b\,) + O(n^d)$$

for some $a > 0$, $b > 1$ and $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

2. A univariate polynomial p in variable x consists of a sum of n_p terms:

$$p = \sum_{i=1}^{n_p} c_p[i] x^{d_p[i]}$$

where $c_p[1], \ldots, c_p[n]$ are coefficients and $d_p[1], \ldots, d_p[n]$ are integer degrees with $d_p[i] \geq 0$.

You may assume that any arithmetical operation is $O(1)$ and that appending to a vector is $O(1)$.

a) Multiplication of two polynomials p and q is defined as:

$$p \times q = \sum_{i=1}^{n_p} \sum_{j=1}^{n_q} c_p[i] \cdot c_q[j] \cdot x^{d_p[i]+d_q[i]}$$

i) What is the time complexity of polynomial multiplication based on the formula above? [3]

Answer:
The two sums become two nested loops with lengths n_p and n_q. Each iteration does a fixed number of arithmetic operations and appends an item so is $O(1)$. Therefore, the total time complexity is $O(n_p n_q)$.

ii) Using this algorithm, what is the best achievable time complexity for calculating p^k for an arbitrary polynomial p and k a power of 2 ($k = 2, 4, 8, 16, \ldots$)? [7]

Answer:
We can form p^2 with complexity $O(n_p^2)$ and the result has length n_p^2. Then we form $p^4 = (p^2)^2$, $p^8 = (p^4)^2$, etc.

Let $N_i = n_p^{2^i}$, then $q_i = p^{2^i}$ has length N_i and the squaring operation $q_i = q_{i-1}^2$ has time complexity $O(N_i)$. So the total time complexity is

$$\sum_{i=1}^{\log_2 k} O(N_i) = O\left(\sum_{i=1}^{\log_2 k} N_i\right)$$

$$= O(\overbrace{n_p^2 + n_p^4 + \cdots + n_p^{k/2}}^{\log_2 k - 1 \text{ terms each smaller than } n_p^{k/2}} + n_p^k)$$

$$= O((\log_2 k - 1) n_p^{k/2} + n_p^k)$$

$$= O(n_p^k)$$

b) A non-zero polynomial is *canonical* if $d_p[i] = i - 1$ and $c_p[n_p] \neq 0$. This ensures that all powers of x must be present, they are ordered from smallest to largest power, there are no repeated terms, and the degree of the polynomial is $n_p - 1$.

If a polynomial is not known to be canonical, we call it *irregular*.

i) Give pseudocode for converting an irregular polynomial p to a canonical polynomial, and state the time complexity. You may rely on common library operations on vectors. [6]

Answer:
Python code:

```
def canonical(coeffs, degrees):
    n = max(degrees) # O(n_p)
    canonical_coeffs = zeros(n+1, dtype=int) # O(1)
    for i in range(len(coeffs)):              # n_p times
        canonical_coeffs[degrees[i]] += coeffs[i] # O(1)
    return canonical_coeffs
```

Total cost is $O(n_p)$.

ii) If we have two canonical polynomials p and q and want to produce a canonical result $p \times q$, what is the time complexity? You do not need to give pseudo-code, but justify your result. [4]

Answer:
We still need to compute $c_p[i]c_q[j]$ for all $1 \leq i \leq n_p$, $1 \leq j \leq n_q$ so the algorithm must be $\Omega(n_p n_q)$, and from the earlier part the result is $O(n_p n_q)$ therefore it must be $\Theta(n_p n_q)$.

iii) What is the time complexity of calculating the canonical result p^k where p is canonical and k is a power of 2? [6]

Answer:
Calculating p^2 requires $O(n_p^2)$ operations but the result has length $2n_p$ instead of n_p^2. So p^4 requires $(2n_p)^2$ operations, p^8 requires $(4n_p)^2$ operations and so on. Therefore the total complexity is

$$\begin{aligned}\sum_{i=1}^{\log_2 k} O(2^{2i} n_p^2) &= O\left(n_p^2 \sum_{i=1}^{\log_2 k} 2^{2i}\right) \\ &= O(n_p^2 2^{2\log_2 k}) \\ &= O(n_p^2 k^2)\end{aligned}$$

iv) Is it better to calculate p^k using irregular or canonical polynomials? [4]

Answer:
The complexity for irregular polynomials is $O(n_p^k)$ and for canonical it is $O(k^2 n_p^2)$. If $n_p > 1$, the irregular method is exponential in k whereas the canonical method is polynomial in k, therefore the canonical method is better.