Topic 6: Complexity of Algorithms

Read: Chpt.3, Rosen

Given a computational problem Π with a set of inputs $D(\Pi)$. To compute/solve Π is to develop an "efficient" program P such that it can be executed to generate a correct output for each and every input from $D(\Pi)$.

Q: What is a program?

Program = Algorithm + Data Structures

Q: What is an algorithm?

An algorithm A for Π is a finite sequence of step-by-step instructions, which are unambiguous and always terminates to generate a correct output for all possible inputs to Π. Hence,

Q: What is a data structure?

A data structure is an implementation of an Abstract Data Type (ADT) for the organization and manipulation of data.

In software development, two important Issues:

1. If A is an algorithm for solving a computational problem Π, how good/bad is this algorithm A?

(Should we use A for the implementation of our program?)

2. If we have several algorithms that can be used for solving Π, which algorithm should we use?

Fundamental Question:

How do we measure the "goodness" of an algorithm?

One Possible Approach:

Measure the amount of computer resource consumption when executing an algorithm/program.

Two Basic Cost (Complexity) Functions:

1. CPU Time requirement: Time complexity $T(n)$
2. Memory requirement: Space complexity $S(n)$

Remarks:

- Complexity functions are functions of n, where n is the size of input.
- Complexity functions $f(n)$ are (eventually) positive functions such that $\forall n \geq k > 0, f(n) > 0$, where k is a constant.
- Complexity functions $f(n)$ are non-decreasing functions such that $\forall n_1, n_2 > 0$, $n_1 < n_2$ implies that $f(n_1) \leq f(n_2)$.

Q: How do we compute the complexity function(s) of a given algorithm?

Basic Approach:

Compute the cost (amount of computing resources) in executing each statement (instruction) & then sum up their costs of all the statements in the algorithm.

Let R(n) be the amount of computing resource(s) required to execute a given algorithm A and cost(S_i) be the cost required in executing the statement S_i, $1 \le i \le m$.

Algorithm A:

S_1;

S_2;

•

•

•

S_m;

Hence,

$$R(n) = \sum_{i=1}^{m} \text{cost}(S_i).$$

Complications in Computing the Cost of a Statement:

1. *S_i is a **conditional statement**:*

   Examples: if-then-else, case, switch, etc.

   cost(S_i) = cost in evaluating the condition +
   cost in evaluating one of the branches

2. *S_i is a **repetition (loop)**:*

   Examples: do-loop, while-loop, doWhile-loop, etc.

   cost(S_i) = (# times the loop condition is evaluated * cost in evaluating the loop condition) +
   (# times the loop is evaluated * cost in evaluating the body of the loop)

   Warning: *It can be very tricky in determining how many times a loop will be executed! Be careful.*

3. *S_i is a **recursive call**:*

   Examples: *direct and indirect recursions.*

   Much more difficult to evaluate, may need to set up and then solve the corresponding recurrence equation for cost(S_i).

Remark: For illustrative purpose, we will concentrate on time complexity only since

- The methods in computing both complexity functions are similar.
- For any input of size n, $S(n) \leq T(n)$. Hence, $T(n)$ always provides a nice upper bound for $S(n)$.

Algorithmic Fundamentals:

Let D_n be the set of all possible inputs of P of size n,

C(I) be the amount of computing resource required to execute A with input I,

Pr(I) be the probability when I is the input to A,

R(n) be the complexity function of A when executed with any input of size n.

1. Best-Case Complexity:

$$R_b(n) = \min_{I \in D_n} C(I).$$

2. Worst-Case Complexity:

$$R_w(n) = \max_{I \in D_n} C(I).$$

3. Average-Case Complexity:

$$R_a(n) = \sum_{I \in D_n} \Pr(I) * C(I).$$

Remark: Observe that $R_b(n) \le R_a(n) \le R_w(n)$.

Basic Approaches in Complexity Analysis:

I. Complexity by Counting Elementary Operations:

Basic Approach:

(1) Identify the most important (elementary) operation(s) in the algorithm.

(2) Count only the number of elementary operations required in the algorithm.

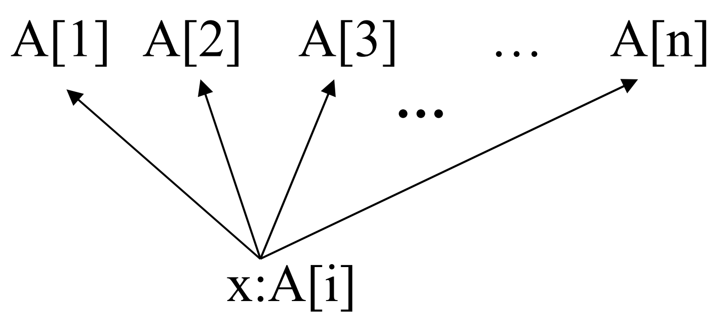
Example 1: Searching an Unordered Array.

Input: An array A[1..n] of distinct integers and an integer key x, $n \geq 1$.

Output: Return an integer i such that $A[i] = x$, $1 \leq i \leq n$, if exists; otherwise, return 0.

Sequential Search Approach:

Starting at A[1], sequentially compare x with A[1], A[2], …, A[n]. If there exists an i, $1 \leq i \leq n$, such that $x = A[i]$, then return i; else return 0.



Sequential_Search Algorithm:

```
Sequential_Search(A: array, x: integer);
        i = 1;                              (1)
        while i ≤ n and A[i] ≠ x do         (2)
            i = i + 1
        endwhile;
        if i ≤ n                            (3)
            then  return(i)
            else  return(0)
        endif;
end Sequential_Search;
```

Remark: There are three statements and five operations (assignment, comparison, logical-and, addition, and return) in this algorithm.

Complexity Analysis:

A Simplified Approach:

Count the number of comparisons between x and A[i].

Hence,

$$T_b(n) = 1,$$

$$T_w(n) = \sum_{i=1}^{n} 1 = n,$$

$$T_a(n) = (n+1)/2. \quad \text{(Can you compute it?)}$$

Example 2: Sorting an Unordered Array.

Input: An array A[1..n] of distinct integers, $n \geq 1$.

Output: Return a sorted array A[1..n] such that $\forall$ i, j, if i < j, then A[i] < A[j].

Selection Sort Algorithm:

```
for index = 1 to n−1 do
    select smallest element x from A[index..n];
    swap smallest element x found with A[index];
endfor;
```

Example: Sorting by insertion sort algorithm.

3	8	6	**2**	5
2	**8**	6	**3**	5
2	3	**6**	8	**5**
2	3	5	**8**	6
2	3	5	6	8

Complexity Analysis:

We will count the number of comparisons between elements in A[1..n]. Observe that since the algorithm consists of a single for-loop and the statements inside the loop will always be executed, $T_b(n) = T_a(n) = T_w(n)$.

Hence,

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

$$= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i$$

$$= n(n-1) - \frac{(n-1)n}{2}$$

$$= \frac{n(n-1)}{2}.$$

HW: Using previous approach, formalize and then compute $T_b(n)$ and $T_w(n)$ for Bubble Sort and Selection Sort.

II. Complexity by Normalizing the Cost of Basic Operations:

Basic Approach:

(1) Assume that the cost in evaluating any one of the basic operations, such as +, -, *, /, compare, return, exit, etc., in a computing environment is always a constant.

(2) Identify the dominating steps of the algorithm, which are statements requiring the most computing resources to execute in the algorithm. Statements

that are not part of a dominating step will be ignored.

(3) In a dominating step, all the basic operations in a "simple" statement will be grouped together and then assigned a constant cost. Again, only the dominating statement will be considered.

Examples:

1.
```
x = 2;                              (1)
y = 5;                              (2)
for i = 1 to n do                   (3)
    for j = 1 to n do
        for k = 1 to n do
            y = x * y / 2;
            x = x + y – 10;
        endfor;
    endfor;
endfor;
```

Observe that statement (3) is the dominating step of the given program segment and the cost in executing statements 3.1 and 3.2 is a constant.

$$\therefore T(n)$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} C$$

$$= Cn^3, C - \text{constant}.$$

2.
```
x = 5;                          (1)
y = 60;                         (2)
for i = 1 to n do               (3)
    for j = 1 to i do           (3.1)
        x = 2*x + 1;
    endfor;
    for k = 1 to n do           (3.2)
        y = x*y/2;
    endfor;
endfor;
```

Observe that statement (3) is the dominating step of the given program segment but the cost in executing statements 3.1 and 3.2 is not a constant.

$$\therefore T(n)$$

$$= \sum_{i=1}^{n} \left(\sum_{j=1}^{i} + \sum_{k=1}^{n}\right) C$$

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

$$= C\left[\frac{n(n+1)}{2} + n^2\right], C - \text{constant.}$$

3.

```
k = 1;                        (1)
x = 2;                        (2)
y = 3;                        (3)
while k <= n do               (4)
    for i = 1 to n do         (4.1)
        k = k + i;
    endfor;
    x = x + y;                (4.2)
endwhile;
```

Observe that the while-loop in statement (4) is the dominating step of the given program segment and it will only be executed once. Also, statement (4.1) is another dominating step inside statement (4). Hence, in computing the complexity function T(n), statement (4.2) will be ignored.

$$\therefore T(n)$$

$$= \sum_{i=1}^{n} C$$

$$= Cn, C - \text{constant.}$$

4.
```
k = 1;                          (1)
x = 6;                          (2)
y = 60;                         (3)
while k <= n^2 do               (4)
    x = (x*y + 2*x)/4;          (4.1)
    k = k*k;                    (4.2)
endwhile;
```

Observe that the while-loop in statement (4) is the dominating step of the given program segment. From statements (1) and (4.2), the loop-controlled index k is always equal to 1, resulting in an infinite loop. Hence, $T(n) = \infty$.

Warning: Recall that the complexity function is a function of the input size n. Hence, in computing a complexity function, you are not allowed to use a fixed value for n.

Observation: Observe that in all of the above examples, the complexity function T(n) is being characterized by a simple mathematical expression using only elementary functions. *If T(n) = g(n), where g(n) is an elementary function, then T(n) can be computed exactly by substituting n into g(n).* These are called the ***closed-form expression*** of T(n).

Q: What if such an expression cannot be found (either doesn't exist or much too difficult to compute) to represent T(n)?

Use approximation!

Since we often only interest in the behavior of T(n) when n is large, we may want to find a simple function g(n) such that $T(n) \leq cg(n)$, for all $n \geq n_0$, where $k > 0$ and $n_0 > 0$ are constants.

The study of the complexity function of algorithms when the input size n is sufficiently large is called the ***asymptotic analysis of algorithms***.

Q: What if the function(s) can have negative values?

Since we are only interest in the relative rate of growth of the functions, we only need to consider the ***absolute values*** of the function(s).

Basic Asymptotic Relations:

Dfn: $f(n) = O(g(n))$ iff there exist constants $k, n_0 > 0$ such that $n \geq n_0$ implies that $|f(n)| \leq k|g(n)|$.

Dfn. $f(n) = \Omega(g(n))$ iff there exist constants $k, n_0 > 0$ such that $n \geq n_0$ implies that $|f(n)| \geq k|g(n)|$.

Dfn. $f(n) = \Theta(g(n))$ iff there exist constants $k_1, k_2, n_0 > 0$ such that $n \geq n_0$ implies that $k_2|g(n)| \leq |f(n)| \leq k_1|g(n)|$.

Some Useful Results in Manipulating Absolute Values:

Given $x, y, z \in R$, we have

1. $|x| \geq 0$.
2. $|x| = 0$ iff $x = 0$.
3. $-|x| \leq x \leq |x|$.
4. $||x| - |y|| \leq |x \pm y| \leq |x| + |y|$.
5. $|x - y| \leq |x - z| + |z - y|$.
6. $|x*y| = |x|*|y|$.
7. $\forall\ y \geq 0,\ x - y \leq x$.
8. $\forall\ y \geq 0,\ x + y \geq x$.

Q: Can we avoid the use of absolute value?

Dfn. A function f is a ***positive*** function iff $\forall x \in \mathbf{R}$, $f(x) > 0$. It is an ***eventually positive*** function iff There exists a constant n_0 such that $\forall x \geq n_0$, $f(x) > 0$.

Remark: If the functions are positive or eventually positive, we do not need to take the absolute value of the functions in proving/computing asymptotic behavior of the functions. Since complexity functions are all eventually positive functions, in asymptotic analysis of algorithms, we can simplify the above definitions without the use of absolute values.

Dfn. Given two eventually positive functions f(n) and g(n),

(1) $f(n) = O(g(n))$ iff there exist constants $k, n_0 > 0$ such that for all $n \geq n_0$, $f(n) \leq kg(n)$.

(2) $f(n) = \Omega(g(n))$ iff there exist constants $k, n_0 > 0$ such that for all $n \geq n_0$, $f(n) \geq kg(n)$.

(3) $f(n) = \Theta(g(n))$ iff there exist constants $k_1, k_2, n_0 > 0$ such that for all $n \geq n_0$, $k_2 g(n) \leq f(n) \leq k_1 g(n)$.

Theorem:

(1) $f(n) = O(g(x)n)$ iff $g(n) = \Omega(f(n))$.

(2) The following statements are equivalence:

(a) $f(n) = \Theta(g(n))$

(b) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

(c) $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Proving Asymptotic Behavior of Functions

Examples:

1. Prove that $2n^2 - 3n + 10 = O(n^2)$.

   Proof: If $2n^2 - 3n + 10 = O(n^2)$, then there exists constants $k > 0$, $n_0 > 0$ such that $2n^2 - 3n + 10 \le kn^2$, for all $n \ge n_0$.

   Observe that

$$\begin{aligned} 2n^2 - 3n + 10 &\le 2n^2 + 10, \ n \ge 1 \\ &\le 2n^2 + 10n^2, \ n \ge 1 \\ &\le 12n^2, \text{ for all } n \ge 1. \end{aligned}$$

   Hence, by choosing $k = 12$, $n_0 = 1$, we have proved that $2n^2 - 3n + 10 = O(n^2)$.

2. Prove that $2n^2 - 3n + 10 \ne O(n)$.

   Proof: If true, then there exists constants $k > 0$, $n_0 > 0$ such that $2n^2 - 3n + 10 \le kn$, for all $n \ge n_0$.

   $\therefore\ 2n - 3 + 10/n \le k$, for all $n \ge n_0$.

   As $n \to \infty$, $\infty \le k$, which is a contradiction.

   $\therefore\ 2n^2 - 3n + 10 \ne O(n)$.

3. Prove that $2n^2 - 3n + 10 = \Omega(n^2)$.
Proof: If $2n^2 - 3n + 10 = \Omega(n^2)$, then there exists constants $k > 0$, $n_0 > 0$ such that $2n^2 - 3n + 10 \geq kn^2$, for all $n \geq n_0$.
Observe that

$$\begin{aligned} 2n^2 - 3n + 10 &\geq 2n^2 - 3n, \ n \geq 1 \\ &\geq n^2 + (n^2 - 3n), \ n \geq 1 \\ &\geq n^2, \ n \geq 3. \end{aligned}$$

Observe that in order to have $n^2 - 3n \geq 0$, $n(n - 3) \geq 0$, or $n \geq 3$. Hence, by choosing $k = 1$, $n_0 = 3$, we have proved that $2n^2 - 3n + 10 = \Omega(n^2)$.

4. Prove that $n^3 - 168n^2 + 188n - 210 = \Omega(n^3)$.
Proof: Observe that

$$\begin{aligned} & n^3 - 168n^2 + 188n - 210 \\ & \geq n^3 - 168n^2 - 210, n \geq 1 \\ & = \frac{n^3}{3} + (\frac{n^3}{3} - 168n^2) + (\frac{n^3}{3} - 210), n \geq 1 \\ & \geq \frac{n^3}{3}, n \geq 504. \end{aligned}$$

Observe that in order to have both $n^3/3 - 168n \geq 0$ and $n^3/3 - 210 \geq 0$, we can choose $n \geq 504$. Hence, by choosing $k = 1/3$, $n_0 = 504$, we have proved that $n^3 - 168n^2 + 188n - 210 = \Omega(n^3)$.

Some Important Properties of big-O, big-Ω, & big-Θ:

1. **Reflexive property**:

   $C*f(n) = O(C*f(n)) = O(f(n))$,
   $C*f(n) = \Omega(C*f(n)) = \Omega(f(n))$,
   $C*f(n) = \Theta(C*f(n)) = \Theta(f(n))$, C-constant.

2. **Symmetric property**:

   $f(n) = \Theta(g(n))$ implies $g(n) = \Theta(f(n))$.

3. **Transitive property**:

   If $f(n) = O(g(n))$ and $g(n) = O(h(n))$,
   then $f(n) = O(h(n))$.
   If $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$,
   then $f(n) = \Omega(h(n))$.
   If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$,
   then $f(n) = \Theta(h(n))$.

4. **Sum Rule**:

   If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$,
   then $(f_1 + f_2)(n)$
   $= f_1(n) + f_2(n)$
   $= O(\max\{g_1(n), g_2(n)\})$.

   If $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, ..., $f_k(n) = O(g_m(n))$,
   then $f_1(n) + f_2(n) + ... + f_m(n)$
   $= O(\max\{g_1(n), g_2(n), \ldots, g_m(n)\})$, where
   m is a fixed constant integer.

5. **Product Rule**:
   Given two positive functions $g_1(n)$ and $g_2(n)$.
   If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then
   $$(f_1 * f_2)(n) = f_1(n) * f_2(n) = O(g_1(n))*O(g_2(n)) = O(g_1(n)*g_2(n)).$$

   Remark: Both the sum rule and product rule can be extended to k functions, where k is a fixed constant.

6. If $f(n) = a_m n^m + a_{m-1} n^{m-1} + \ldots + a_1 n^1 + a_0$, where a_i′s are constants with $a_m > 0$, then $f(n) = \Theta(n^m)$.

7. $H_n = \sum_{i=1}^{n} \frac{1}{i} = \ln n + \gamma + O(\frac{1}{n}), \gamma = 0.577\ldots,$
   where H_n is the nth harmonic number and γ is the Euler's constant.

Examples on Using Asymptotic Relations:

1. $f(n) = 3n^2 + n\lg n - 100n$
$$= O(3n^2 + n\lg n - 100n)$$
$$= O(\max\{3n^2, n\lg n, |-100n|\})$$
$$= O(3n^2)$$
$$= O(n^2).$$

2. $f(n) = (n + 1)\lg(4n^2 + 60)$

   Observe that
$$\lg(4n^2 + 60) \le \lg(4n^2 + 60n^2)$$
$$= \lg(64n^2)$$
$$= \lg(8n)^2$$
$$= 2*\lg 8n$$
$$\le 2*\lg n^2,\ n \ge 8$$
$$= 4\lg n,\ n \ge 8$$
$$= O(\lg n).$$

   Observe also that
$$(n+1) \le 2n,\ n \ge 1$$
$$= O(n).$$

   Hence,
$$f(n)$$
$$= (n + 1)\lg(4n^2 + 60)$$
$$= O((n + 1)\lg(4n^2 + 60))$$
$$= O(n\lg n).$$

Given an algorithm A,

Ideal Case:

Compute T(n) in closed-form.

First Approximation:

Compute a function f(n) such that $f(n) = \Theta(f(n))$.

Second Approximation:

Compute a function f(n) such that $f(n) = O(f(n))$.

Warning: One should never compare the performance of two algorithms using their big-O information.

Example: Consider two algorithms A_1 and A_2 with complexity $T_1(n) = O(n^3)$ and $T_2(n) = O(n^{1000})$. Even though $n^3 = O(n^{1000})$, you can never conclude that algorithm A_1 is more efficient than algorithm A_2 for sufficiently large n.

Consider $T_1(n) = n^3 = O(n^3)$ and $T_2(n) = n = O(n^{1000})$. Clearly algorithm A_1 is **not** more efficient than algorithm A_2 even when n is large!

Remark: You can only compare the performance of two algorithms using their closed-form expression (or big-Θ information).

Some Useful Functions in Complexity Analysis:

f(n)	***Growth Rate***	***Algorithmic Performance***
n^n	Fastest	Worst performance
$n!$		
•		
•		
•		
3^n		
2^n		
•		
•		
•		
n^k, $k \geq 2$		
n^2		
nlgn		
n		
lgn		
c	Slowest	Best performance

Q: Why is it so important to design an efficient algorithm?

Importance on Efficient Algorithms:

When an algorithm A is used to compute a problem Π with input S, it requires 0.5ms (10^{-3}s) to execute A when |S| = 1,000. If the complexity of the algorithm A is given by the following closed-form expressions, compute the time required to execute the A when |S| = 1,000,000 = 10^6.

(a) T(n) = 210n.
(b) T(n) = $2\log_{10}$n.
(c) T(n) = n$\log_{10}$n.
(d) T(n) = n^2.
(e) T(n) = n^3.
(f) T(n) = 2^n.

Solution:

Observe that

$$\frac{T(n)}{C(n)} = \frac{T(n^*)}{C(n^*)},$$

$$C(n^*) = \frac{T(n^*)}{T(n)} * C(n) = \frac{T(n^*)}{T(n)} * 0.5ms.$$

(a) $T(n) = 2\log n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{2\log 10^6}{2\log 10^3} * 0.5ms = \frac{6}{3} * 0.5ms = 1.0ms.$$

(b) $T(n) = 210n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{210 * 10^6}{210 * 10^3} * 0.5ms = 10^3 * 0.5ms = 0.5s.$$

(c) $T(n) = n \log n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{10^6 * \log 10^6}{10^3 * \log 10^3} * 0.5ms = 10^3 * 2 * 0.5ms = 1s.$$

(d) $T(n) = n^2.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{(10^6)^2}{(10^3)^2} * 0.5ms = 10^6 * 0.5ms$$

$$= 10^3 * 0.5s \simeq 8.3 \min.$$

(e) $T(n) = n^3.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms = \frac{(10^6)^3}{(10^3)^3} * 0.5ms = 10^9 * 0.5ms$$

$$= 5 * 10^5 s \simeq 5.79 \text{ days}$$

(f) $T(n) = 2^n.$

$$C(n^*) = \frac{T(n^*)}{T(n)} * 0.5ms$$

$$= \frac{2^{10^6}}{2^{10^3}} * 0.5ms$$

$$= 2^{10^6 - 10^3} * 0.5ms$$

$$= 2^{999000} * 0.5ms.$$

Remark: 2^{64} =18,446,744,073,709,551,616.

Observe that

$$C(n^*) = 2^{64} * 0.5ms > 1.07 \times 10^{11} \text{days} = 293,150,684 \text{ years}.$$

When selecting an algorithm for solving a given problem, one must also consider the characteristics and the size of the input data set.

Example: Given two algorithms A_1 and A_2 with $T_1(n) = 2^{18}n^2$ and $T_2(n) = 2^n$. Which algorithm should we use in general? What if $n \le 20$?

Let's try to find the smallest input size n such that A_1 is faster than A_2. Hence, we need to find smallest integer $n > 0$ such that $2^{18}n^2 \le 2^n$.

$$\begin{aligned} 2^{18}n^2 &\le 2^n \\ \lg 2^{18}n^2 &\le \lg 2^n \\ \lg 2^{18} + \lg n^2 &\le n \\ 18 + 2\lg n &\le n \\ 0 &\le n - 2\lg n - 18 \end{aligned}$$

Take $n = 2^4$, we have $2^4 - 2\lg 2^4 - 18 = -10$
Take $n = 2^5$, we have $2^5 - 2\lg 2^5 - 18 = 4$.
Hence, $2^4 < n < 2^5$.

Q: How do you find the smallest n that will satisfy the above inequality?

Apply binary search to the region $(2^4, 2^5)$. (H.W.)