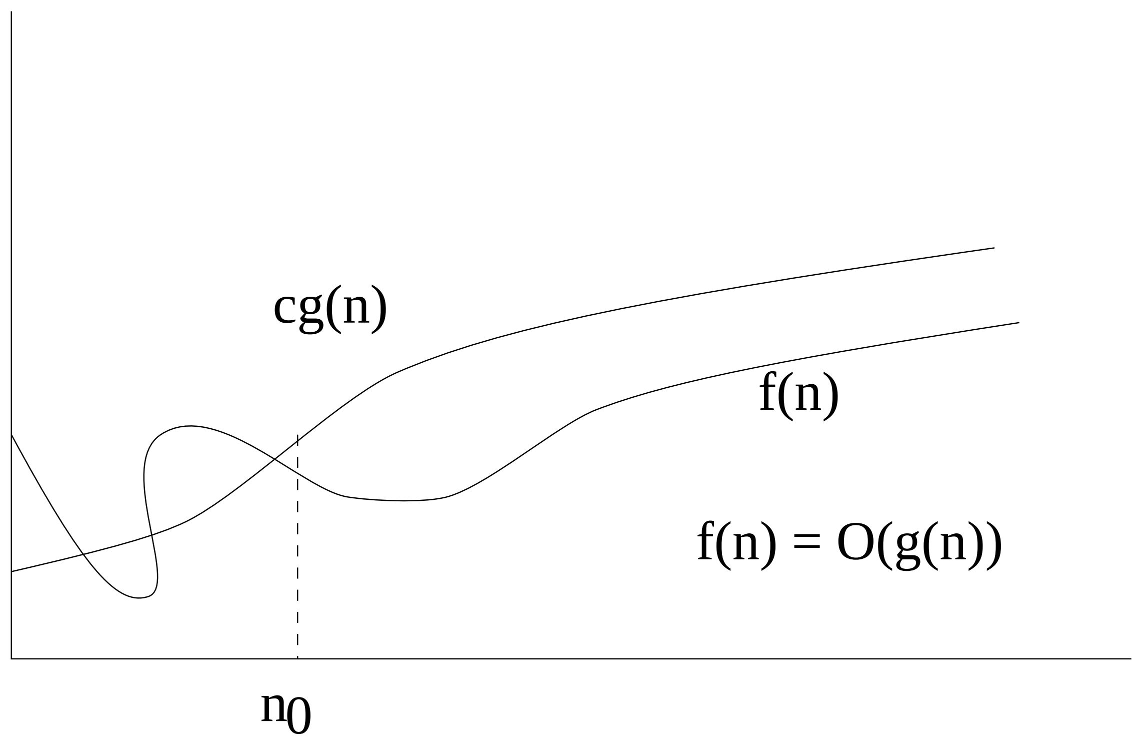
Growth of Functions and Aymptotic Notation

- When we study algorithms, we are interested in characterizing them according to their efficiency.
- We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the *asymptotic running time*.
- We need to develop a way to talk about rate of growth of functions so that we can compare algorithms.
- *Asymptotic notation* gives us a method for classifying functions according to their rate of growth.

Big-O Notation

- **Definition:** $f(n) = O(g(n))$ iff there are two positive constants c and n_0 such that
$$|f(n)| \leq c\,|g(n)| \text{ for all } n \geq n_0$$
- If $f(n)$ is nonnegative, we can simplify the last condition to
$$0 \leq f(n) \leq c\,g(n) \text{ for all } n \geq n_0$$
- We say that "$f(n)$ is big-O of $g(n)$."
- As n increases, $f(n)$ grows no faster than $g(n)$. In other words, $g(n)$ is an *asymptotic upper bound* on $f(n)$.



Example: $n^2 + n = O(n^3)$

Proof:

- Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$
- Notice that if $n \geq 1$, $n \leq n^3$ is clear.
- Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.
- **Side Note:** In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.
- Therefore,

$$n^2 + n \leq n^3 + n^3 = 2n^3$$

- We have just shown that

$$n^2 + n \leq 2n^3 \text{ for all } n \geq 1$$

- Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Big-Ω notation

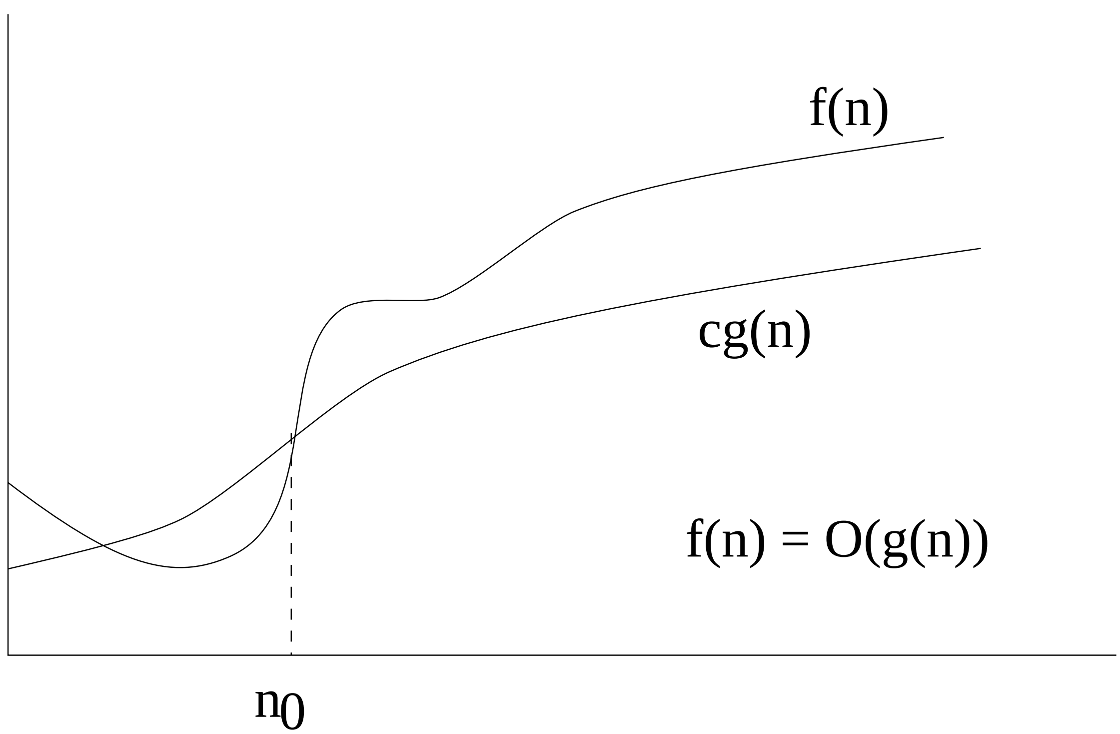
- **Definition:** $f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that

$$|f(n)| \geq c\,|g(n)| \text{ for all } n \geq n_0$$

- If $f(n)$ is nonnegative, we can simplify the last condition to

$$0 \leq c\,g(n) \leq f(n) \text{ for all } n \geq n_0$$

- We say that "$f(n)$ is omega of $g(n)$."
- As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an *asymptotic lower bound* on $f(n)$.



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof:

- Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$
- It is not too hard to see that if $n \geq 0$,
$$n^3 \leq n^3 + 4n^2$$
- We have already seen that if $n \geq 1$,
$$n^2 \leq n^3$$
- Thus when $n \geq 1$,
$$n^2 \leq n^3 \leq n^3 + 4n^2$$
- Therefore,
$$1n^2 \leq n^3 + 4n^2 \text{ for all } n \geq 1$$
- Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$.)

Big-Θ notation

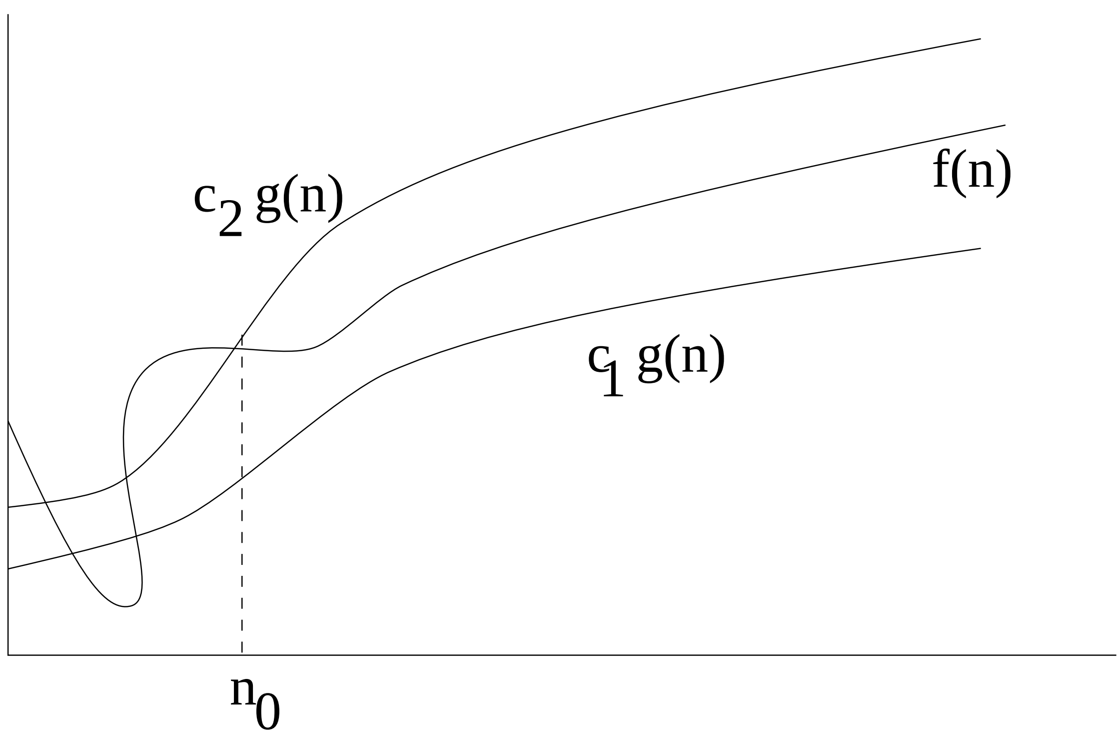
- **Definition:** $f(n) = \Theta(g(n))$ iff there are three positive constants c_1, c_2 and n_0 such that

  $$c_1|g(n)| \le |f(n)| \le c_2|g(n)| \text{ for all } n \ge n_0$$

- If $f(n)$ is nonnegative, we can simplify the last condition to

  $$0 \le c_1\, g(n) \le f(n) \le c_2\, g(n) \text{ for all } n \ge n_0$$

- We say that "$f(n)$ is theta of $g(n)$."
- As n increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an *asymptotically tight bound* on $f(n)$.



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof:

- When $n \geq 1$,

$$n^2 + 5n + 7 \leq n^2 + 5n^2 + 7n^2 \leq 13n^2$$

- When $n \geq 0$,

$$n^2 \leq n^2 + 5n + 7$$

- Thus, when $n \geq 1$

$$1n^2 \leq n^2 + 5n + 7 \leq 13n^2$$

Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, $c_1 = 1$, and $c_2 = 13$.)

Arithmetic of Big-O, Ω, and Θ notations

- Transitivity:
  - $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- Scaling: if $f(n) = O(g(n))$ then for any $k > 0$, $f(n) = O(kg(n))$
- Sums: if $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $(f_1 + f_2)(n) = O(\max(g_1(n), g_2(n)))$

Strategies for Big-O

- Sometimes the easiest way to prove that $f(n) = O(g(n))$ is to take c to be the sum of the positive coefficients of $f(n)$.
- We can usually ignore the negative coefficients. Why?
- **Example:** To prove $5n^2 + 3n + 20 = O(n^2)$, we pick $c = 5 + 3 + 20 = 28$. Then if $n \geq n_0 = 1$,

  $$5\,n^2 + 3\,n + 20 \leq 5\,n^2 + 3\,n^2 + 20\,n^2 = 28\,n^2,$$

  thus $5n^2 + 3n + 20 = O(n^2)$.
- This is not always so easy. How would you show that $(\sqrt{2})^{\log n} + \log^2 n + n^4$ is $O(2^n)$? Or that $n^2 = O(n^2 - 13n + 23)$? After we have talked about the relative rates of growth of several functions, this will be easier.
- In general, we simply (or, in some cases, with much effort) find values c and n_0 that work. This gets easier with practice.

Strategies for Ω and Θ

- Proving that a $f(n) = \Omega(g(n))$ often requires more thought.
  - Quite often, we have to pick $c < 1$.
  - A good strategy is to pick a value of c which you think will work, and determine which value of n_0 is needed.
  - Being able to do a little algebra helps.
  - We can sometimes simplify by ignoring terms if $f(n)$ with the positive coefficients. Why?
- The following theorem shows us that proving $f(n) = \Theta(g(n))$ is nothing new:
  - **Theorem:** $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.
  - Thus, we just apply the previous two strategies.
- We will present a few more examples using a several different approaches.

Show that $\frac{1}{2}n^2 + 3n = \Theta(n^2)$

Proof:

- Notice that if $n \geq 1$,

$$\frac{1}{2}n^2 + 3n \leq \frac{1}{2}n^2 + 3n^2 = \frac{7}{2}n^2$$

- Thus,

$$\frac{1}{2}n^2 + 3n = O(n^2)$$

- Also, when $n \geq 0$,

$$\frac{1}{2}n^2 \leq \frac{1}{2}n^2 + 3n$$

- So

$$\frac{1}{2}n^2 + 3n = \Omega(n^2)$$

- Since $\frac{1}{2}n^2 + 3n = O(n^2)$ and $\frac{1}{2}n^2 + 3n = \Omega(n^2)$,

$$\frac{1}{2}n^2 + 3n = \Theta(n^2)$$

Show that $(n\log n - 2\,n + 13) = \Omega(n\log n)$

Proof: We need to show that there exist positive constants c and n_0 such that

$$0 \le c\,n\log n \le n\log n - 2\,n + 13 \text{ for all } n \ge n_0.$$

Since $\quad n\log n - 2\,n \le n\log n - 2\,n + 13,$

we will instead show that

$$c\,n\log n \le n\log n - 2\,n,$$

which is equivalent to

$$c \le 1 - \frac{2}{\log n}, \text{ when } n > 1.$$

If $n \ge 8$, then $2/(\log n) \le 2/3$, and picking $c = 1/3$ suffices. Thus if $c = 1/3$ and $n_0 = 8$, then for all $n \ge n_0$, we have

$$0 \le c\,n\log n \le n\log n - 2\,n \le n\log n - 2\,n + 13.$$

Thus $(n\log n - 2\,n + 13) = \Omega(n\log n)$.

Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

Proof:

- We need to find positive constants c_1, c_2, and n_0 such that

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3\,n \le c_2 n^2 \text{ for all } n \ge n_0$$

- Dividing by n^2, we get

$$0 \le c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$$

- $c_1 \le \frac{1}{2} - \frac{3}{n}$ holds for $n \ge 10$ and $c_1 = 1/5$
- $\frac{1}{2} - \frac{3}{n} \le c_2$ holds for $n \ge 10$ and $c_2 = 1$.
- Thus, if $c_1 = 1/5$, $c_2 = 1$, and $n_0 = 10$, then for all $n \ge n_0$,

$$0 \le c_1 n^2 \le \frac{1}{2}n^2 - 3\,n \le c_2 n^2 \text{ for all } n \ge n_0.$$

Thus we have shown that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Asymptotic Bounds and Algorithms

- In all of the examples so far, we have assumed we knew the exact running time of the algorithm.
- In general, it may be very difficult to determine the exact running time.
- Thus, we will try to determine a bounds without computing the exact running time.
- **Example:** What is the complexity of the following algorithm?

```
for (i = 0; i < n; i ++)
  for (j = 0; j < n; j ++)
    a[i][j] = b[i][j] * x;
```

  Answer: $O(n^2)$

- We will see more examples later.

Summary of the Notation

- $f(n) = O(g(n)) \Rightarrow f \preceq g$
- $f(n) = \Omega(g(n)) \Rightarrow f \succeq g$
- $f(n) = \Theta(g(n)) \Rightarrow f \approx g$
- It is important to remember that a Big-O bound is only an *upper bound*. So an algorithm that is $O(n^2)$ might not ever take that much time. It may actually run in $O(n)$ time.
- Conversely, an Ω bound is only a *lower bound*. So an algorithm that is $\Omega(n \log n)$ might actually be $\Theta(2^n)$.
- Unlike the other bounds, a Θ-bound is precise. So, if an algorithm is $\Theta(n^2)$, it runs in quadratic time.

Common Rates of Growth

In order for us to compare the efficiency of algorithms, we need to know some common growth rates, and how they compare to one another. This is the goal of the next several slides.

Let n be the size of input to an algorithm, and k some constant. The following are common rates of growth.

- Constant: $\Theta(k)$, for example $\Theta(1)$
- Linear: $\Theta(n)$
- Logarithmic: $\Theta(\log_k n)$
- $n \log n$: $\Theta(n \log_k n)$
- Quadratic: $\Theta(n^2)$
- Polynomial: $\Theta(n^k)$
- Exponential: $\Theta(k^n)$

We'll take a closer look at each of these classes.

Classification of algorithms - $\Theta(1)$

- Operations are performed k times, where k is some constant, independent of the size of the input n.
- This is the best one can hope for, and most often unattainable.
- **Examples:**

```
int Fifth_Element(int A[],int n) {
    return A[5];
}


int Partial_Sum(int A[],int n) {
    int sum=0;
    for(int i=0;i<42;i++)
       sum=sum+A[i];
    return sum;
}
```

Classification of algorithms - $\Theta(n)$

- Running time is linear
- As n increases, run time increases in proportion
- Algorithms that attain this look at each of the n inputs at most some constant k times.
- **Examples:**

```
void sum_first_n(int n) {
   int i,sum=0;
   for (i=1;i<=n;i++)
      sum = sum + i;
    }

void m_sum_first_n(int n) {
     int i,k,sum=0;
     for (i=1;i<=n;i++)
         for (k=1;k<7;k++)
                 sum = sum + i;
     }
```

Classification of algorithms - $\Theta(\log n)$

- A logarithmic function is the inverse of an exponential function, i.e. $b^x = n$ is equivalent to $x = \log_b n$)
- Always increases, but at a slower rate as n increases. (Recall that the derivative of $\log n$ is $\frac{1}{n}$, a decreasing function.)
- Typically found where the algorithm can systematically ignore fractions of the input.
- **Examples:**

```
int binarysearch(int a[], int n, int val)
{
int l=1, r=n, m;
   while (r>=1) {
     m = (l+r)/2;
     if   (a[m]==val) return m;
     if   (a[m]>val) r=m-1;
     else l=m+1; }
return -1;
}
```

Classification of algorithms - $\Theta(n \log n)$

- Combination of $O(n)$ and $O(\log n)$
- Found in algorithms where the input is recursively broken up into a constant number of subproblems of the same type which can be solved independently of one another, followed by recombining the sub-solutions.
- **Example:** Quicksort is $O(n \log n)$.

Perhaps now is a good time for a reminder that when speaking asymptotically, the base of logarithms is irrelevant. This is because of the identity

$$\log_a b \log_b n = log_a n.$$

Classification of algorithms - $\Theta(n^2)$

- We call this class quadratic.
- As n doubles, run-time quadruples.
- However, it is still polynomial, which we consider to be good.
- Typically found where algorithms deal with all pairs of data.
- **Example:**

```
int *compute_sums(int A[], int n) {
    int M[n][n];
    int i,j;
    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
            M[i][j]=A[i]+A[j];
    return M;
    }
```

- More generally, if an algorithm is $\Theta(n^k)$ for constant k it is called a polynomial-time algorithm.

Classification of algorithms - $\Theta(2^n)$

- We call this class exponential.
- This class is, essentially, as bad as it gets.
- Algorithms that use brute force are often in this class.
- Can be used only for small values of n in practice.
- **Example:** A simple way to determine all n bit numbers whose binary representation has k non-zero bits is to run through all the numbers from 1 to 2^n, incrementing a counter when a number has k nonzero bits. It is clear this is exponential in n.

Comparison of growth rates

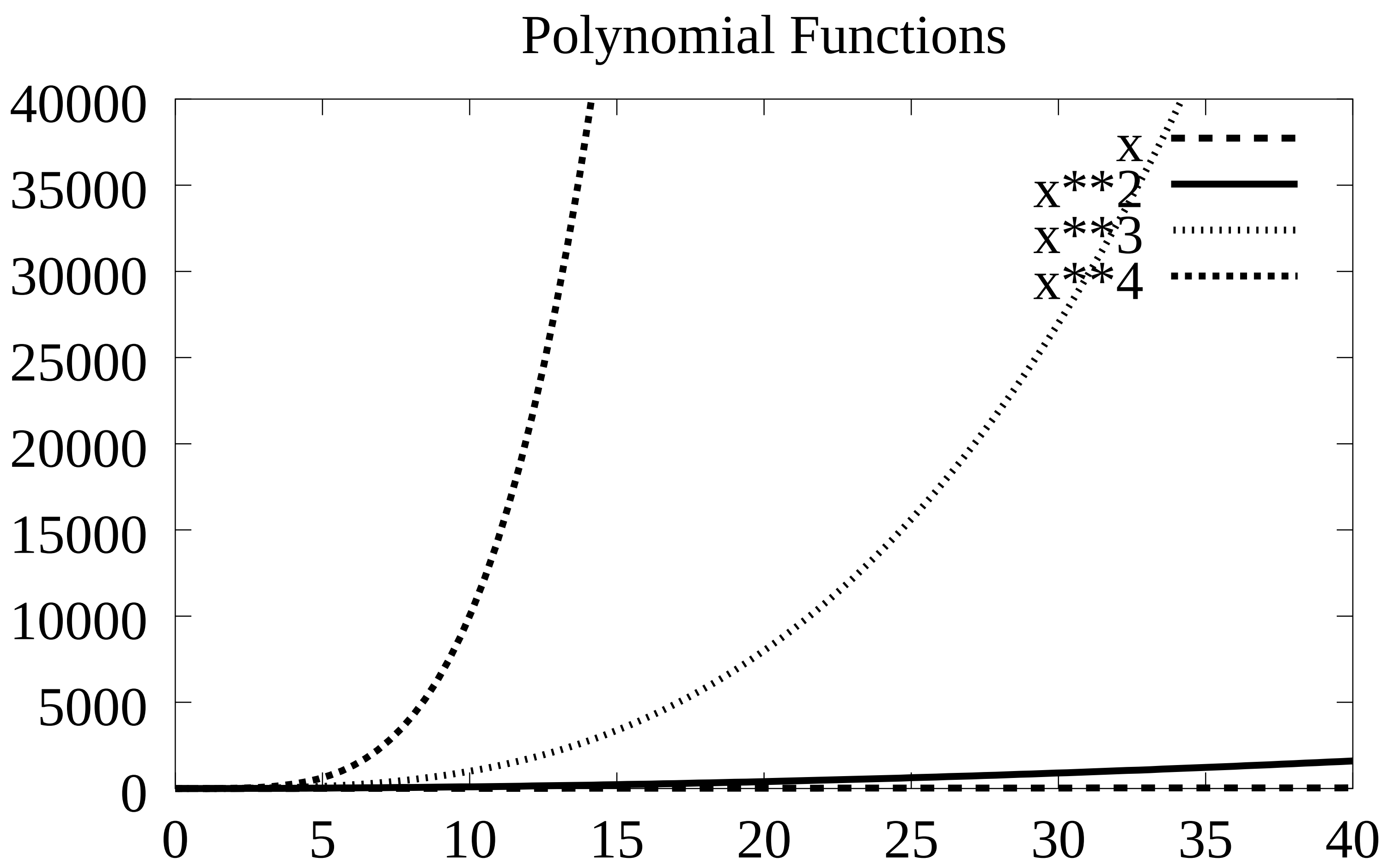
$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
0.6931	2	1.39	4	8	4
1.099	3	3.30	9	27	8
1.386	4	5.55	16	64	16
1.609	5	8.05	25	125	32
1.792	6	10.75	36	216	64
1.946	7	13.62	49	343	128
2.079	8	16.64	64	512	256
2.197	9	19.78	81	729	512
2.303	10	23.03	100	1000	1024
2.398	11	26.38	121	1331	2048
2.485	12	29.82	144	1728	4096
2.565	13	33.34	169	2197	8192
2.639	14	36.95	196	2744	16384
2.708	15	40.62	225	3375	32768
2.773	16	44.36	256	4096	65536
2.833	17	48.16	289	4913	131072
2.890	18	52.03	324	5832	262144
$\log \log m$	$\log m$				m

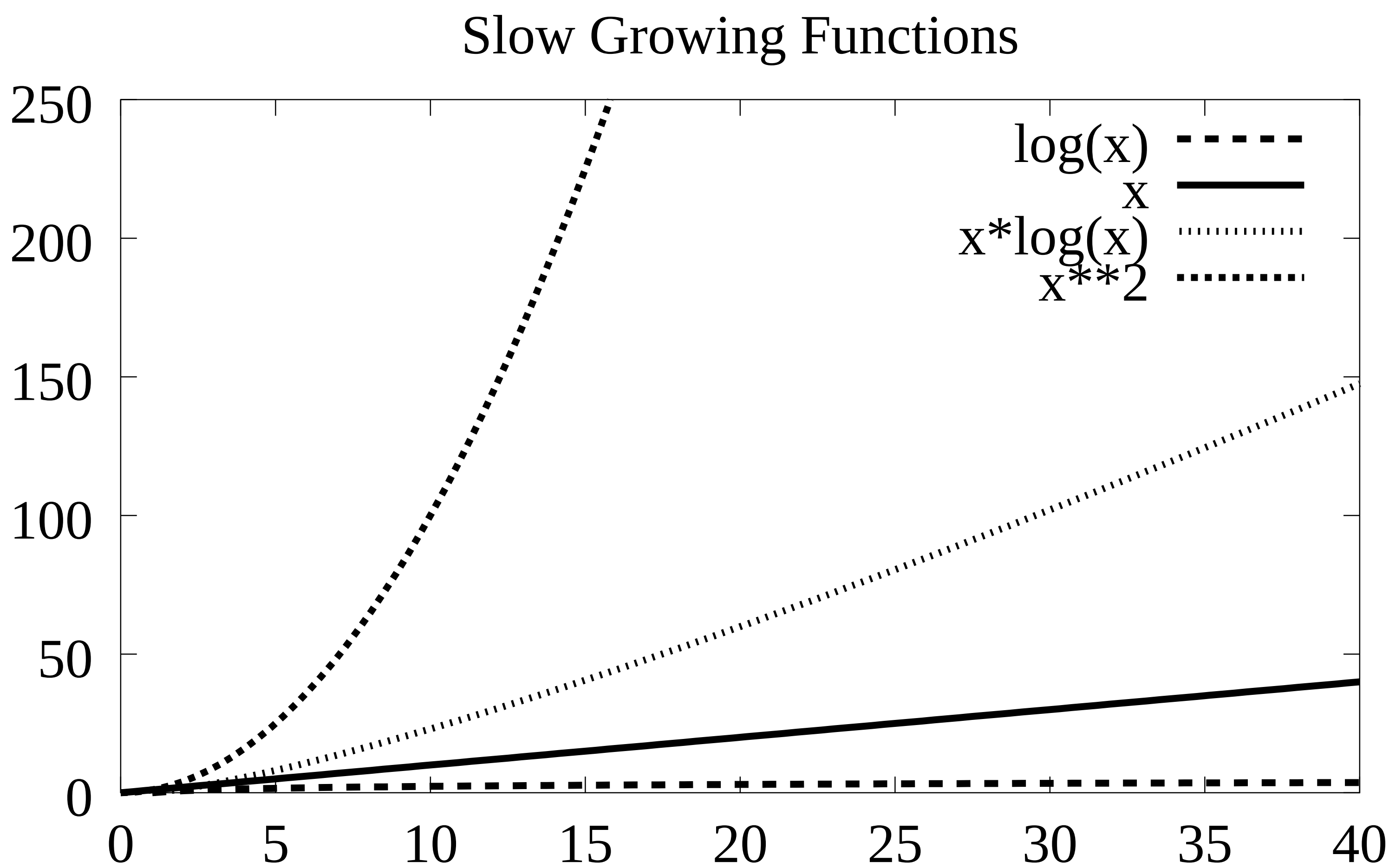
More growth rates

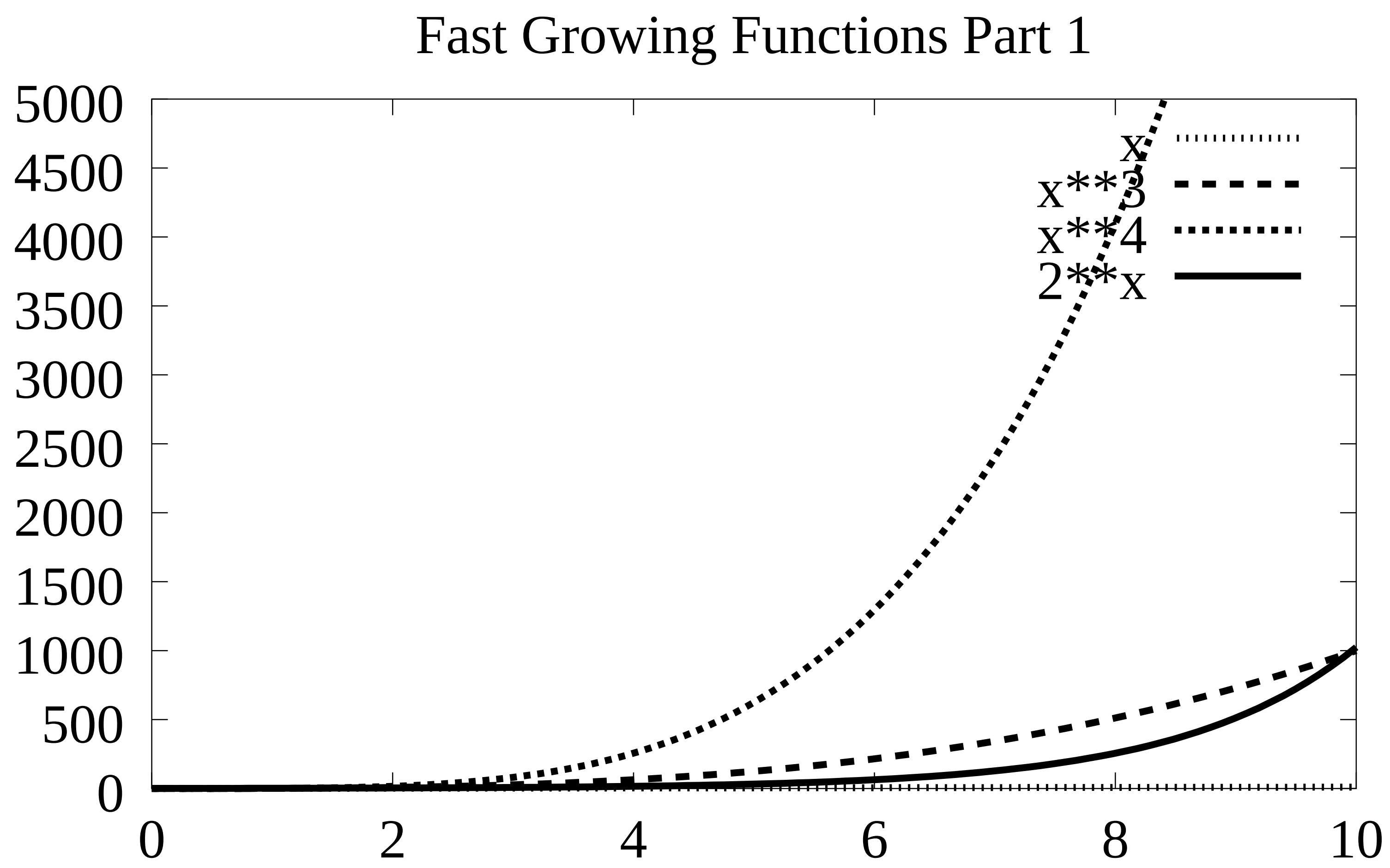
n	$100n$	n^2	$11n^2$	n^3	2^n
1	100	1	11	1	2
2	200	4	44	8	4
3	300	9	99	27	8
4	400	16	176	64	16
5	500	25	275	125	32
6	600	36	396	216	64
7	700	49	539	343	128
8	800	64	704	512	256
9	900	81	891	729	512
10	1000	100	1100	1000	1024
11	1100	121	1331	1331	2048
12	1200	144	1584	1728	4096
13	1300	169	1859	2197	8192
14	1400	196	2156	2744	16384
15	1500	225	2475	3375	32768
16	1600	256	2816	4096	65536
17	1700	289	3179	4913	131072
18	1800	324	3564	5832	262144
19	1900	361	3971	6859	524288

More growth rates

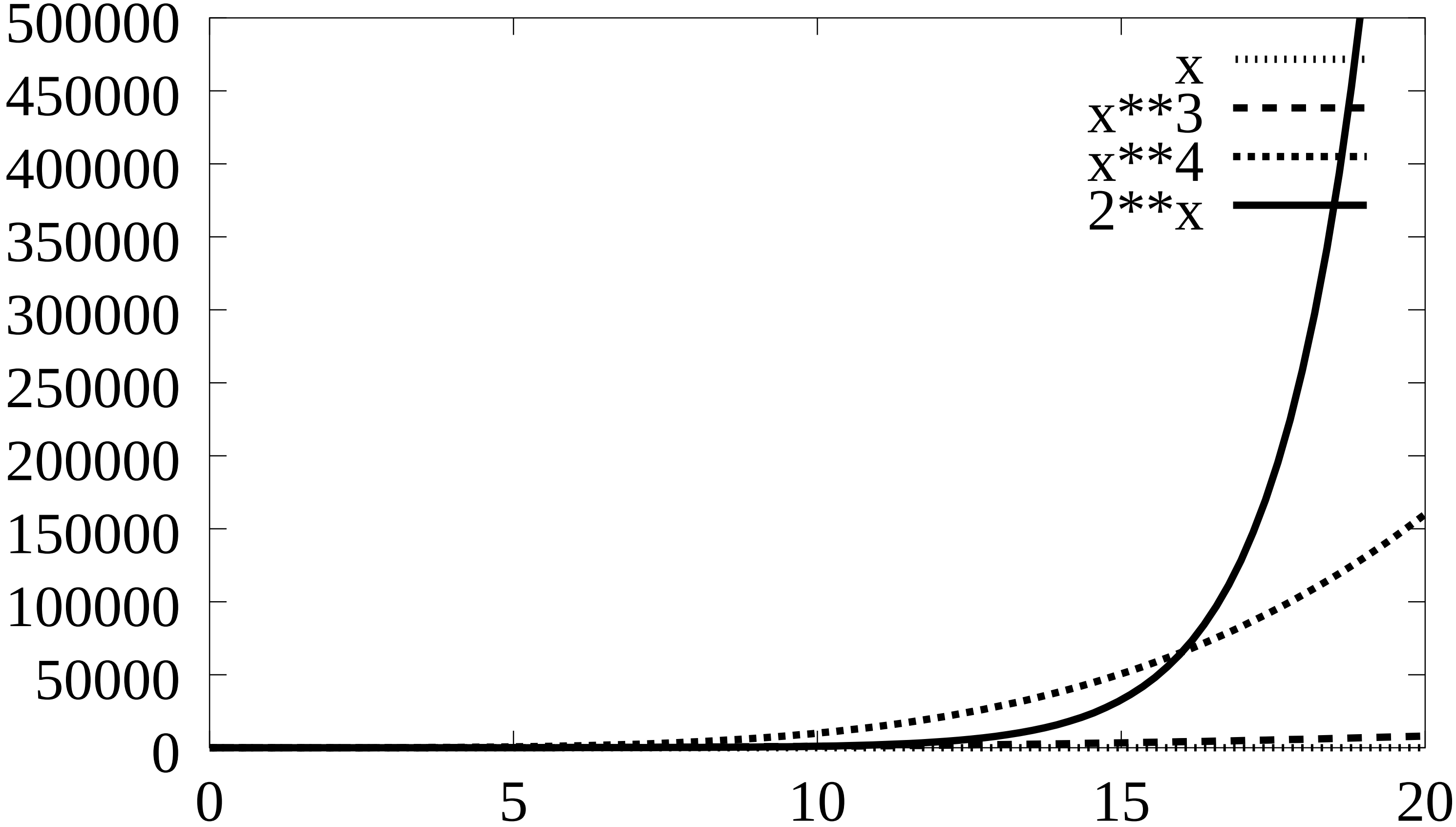
n	n^2	$n^2 - n$	$n^2 + 99$	n^3	$n^3 + 234$
2	4	2	103	8	242
6	36	30	135	216	450
10	100	90	199	1000	1234
14	196	182	295	2744	2978
18	324	306	423	5832	6066
22	484	462	583	10648	10882
26	676	650	775	17576	17810
30	900	870	999	27000	27234
34	1156	1122	1255	39304	39538
38	1444	1406	1543	54872	55106
42	1764	1722	1863	74088	74322
46	2116	2070	2215	97336	97570
50	2500	2450	2599	125000	125234
54	2916	2862	3015	157464	157698
58	3364	3306	3463	195112	195346
62	3844	3782	3943	238328	238562
66	4356	4290	4455	287496	287730
70	4900	4830	4999	343000	343234
74	5476	5402	5575	405224	405458


Polynomial Functions
40000
35000
30000
25000
20000
15000
10000
5000
0
0
5
10
15
20
25
30
35
40
x
x**2
x**3
x**4

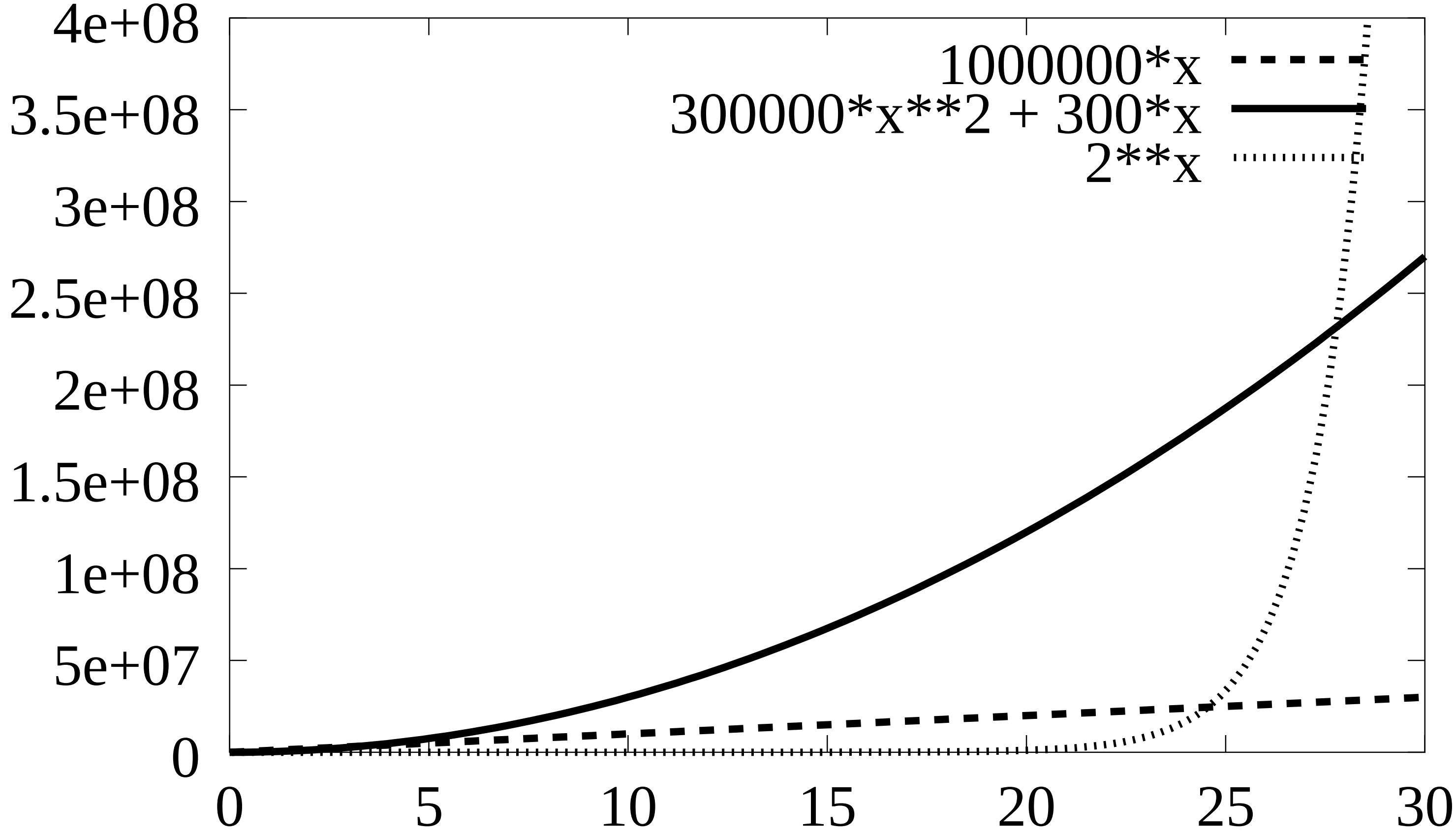

Slow Growing Functions
250
200
150
100
50
0
0
5
10
15
20
25
30
35
40
log(x)
x
x*log(x)
x**2


Fast Growing Functions Part 1
x
x**3
x**4
2**x
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
2
4
6
8
10

Fast Growing Functions Part 2



Why Constants and Non-Leading Terms Don’t Matter



Classification Summary

We have seen that when we analyze functions asymptotically:

- Only the leading term is important.
- Constants don't make a significant difference.
- The following inequalities hold asymptotically:

$$c < \log n < log^2 n < \sqrt{n} < n < n \log n$$

$$n < n \log n < n^{(1.1)} < n^2 < n^3 < n^4 < 2^n$$

- In other words, an algorithm that is $\Theta(n \log(n))$ is more efficient than an algorithm that is $\Theta(n^3)$.