# 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 n0 such that

|f(n)| ≤ c|g(n)| for all n ≥ n0

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

0 ≤ f(n) ≤ cg(n) for all n ≥ n0

* 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).

f(n) = O(g(n))

n

0

cg(n)

f(n)

# Example: n2 + n = O(n3)

**Proof:**

* Here, we have f(n) = n2 + n, and g(n) = n3
* Notice that if n ≥ 1, n ≤ n3 is clear.
* Also, notice that if n ≥ 1, n2 ≤ n3 is clear.
* **Side Note:** In general, if a ≤ b, then na ≤ nb whenever n ≥ 1. This fact is used often in these types of proofs.
* Therefore,

n2 + n ≤ n3 + n3 = 2n3

* We have just shown that n2 + n ≤ 2n3 for all n ≥ 1
* Thus, we have shown that n2 + n = O(n3)

(by definition of Big-O, with n0 = 1, and c = 2.)

# Big-Ω notation

* **Definition:** f(n) = Ω(g(n)) iff there are two positive constants c and n0 such that

|f(n)| ≥ c|g(n)| for all n ≥ n0

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

0 ≤ cg(n) ≤ f(n) for all n ≥ n0

* 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).

n

0

cg(n)

f(n)

f(n) = O(g(n))

**Example:** n3 + 4n2 = Ω(n2)

**Proof:**

* Here, we have f(n) = n3 + 4n2, and g(n) = n2
* It is not too hard to see that if n ≥ 0,

n3 ≤ n3 + 4n2

* We have already seen that if n ≥ 1,

n2 ≤ n3

* Thus when n ≥ 1,

n2 ≤ n3 ≤ n3 + 4n2

* Therefore,

1n2 ≤ n3 + 4n2 for all n ≥ 1

* Thus, we have shown that n3 + 4n2 = Ω(n2)

(by definition of Big-Ω, with n0 = 1, and c = 1.)

# Big-Θ notation

* **Definition:** f(n) = Θ(g(n)) iff there are three positive constants c1, c2 and n0 such that

c1|g(n)| ≤ |f(n)| ≤ c2|g(n)| for all n ≥ n0

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

0 ≤ c1 g(n) ≤ f(n) ≤ c2 g(n) for all n ≥ n0

* 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).

c

2

g(n)

c

1

g(n)

f(n)

n

0

**Example:** n2 + 5n + 7 = Θ(n2)

**Proof:**

* When n ≥ 1, n2 + 5n + 7 ≤ n2 + 5n2 + 7n2 ≤ 13n2
* When n ≥ 0,

n2 ≤ n2 + 5n + 7

* Thus, when n ≥ 1

1n2 ≤ n2 + 5n + 7 ≤ 13n2

Thus, we have shown that n2 + 5n + 7 = Θ(n2) (by definition of Big-Θ, with n0 = 1, c1 = 1, and c2 = 13.)

# Arithmetic of Big-O, Ω, and Θ notations

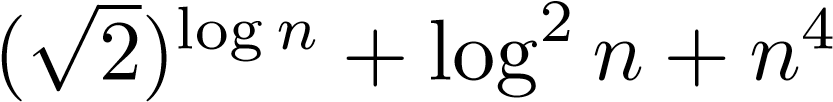
* Transitivity:
  + f(n) ∈ O(g(n)) and g(n) ∈ O(h(n)) ⇒ f(n) ∈ O(h(n))
  + f(n) ∈ Θ(g(n)) and g(n) ∈ Θ(h(n)) ⇒ f(n) ∈ Θ(h(n))
  + f(n) ∈ Ω(g(n)) and g(n) ∈ Ω(h(n)) ⇒ f(n) ∈ Ω(h(n))
* Scaling: if f(n) ∈ O(g(n)) then for any k > 0,f(n) ∈ O(kg(n))
* Sums: if f1(n) ∈ O(g1(n)) and f2(n) ∈ O(g2(n)) then

(f1 + f2)(n) ∈ O(max(g1(n),g2(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 5n2 + 3n + 20 = O(n2), we pick c = 5 + 3 + 20 = 28. Then if n ≥ n0 = 1,

5n2 + 3n + 20 ≤ 5n2 + 3n2 + 20n2 = 28n2, thus 5n2 + 3n + 20 = O(n2).

* This is not always so easy. How would you show that  is O(2n)? Or that n2 = O(n2 − 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 n0 that work. This gets easier with practice.

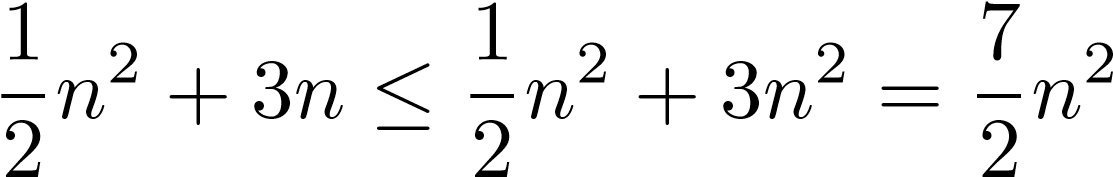
# Strategies for Ω and Θ

* Proving that a f(n) = Ω(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 n0 is needed.
  + Being able to do a little algebra helps.
  + We can sometimes simplify by ignoring terms of f ( n) with the positive coefficients. Why?
* The following theorem shows us that proving f(n) = Θ(g(n)) is nothing new:
  + **Theorem:** f(n) = Θ(g(n)) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)).
  + Thus, we just apply the previous two strategies.
* We will present a few more examples using a several different approaches.

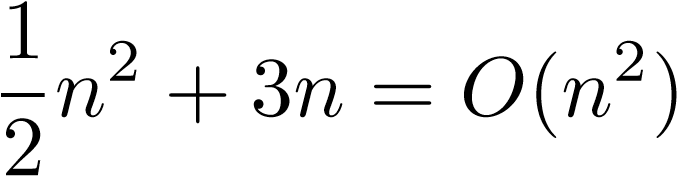
# Show that 12n2 + 3n = Θ(n2)

**Proof:**

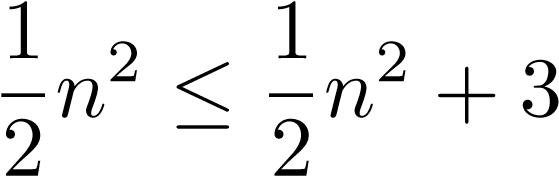
* Notice that if n ≥ 1,



* Thus,



* Also, when n ≥ 0,

n

* So

1

n2 + 3n = Ω(n2)

2

* Since 21n2 +3n = O(n2) and 21n2 +3n = Ω(n2),

1 2 + 3n = Θ(n2) n

2

## **Show that** (nlogn− 2n + 13) = Ω(nlogn)

**Proof:** We need to show that there exist positive constants c and n0 such that

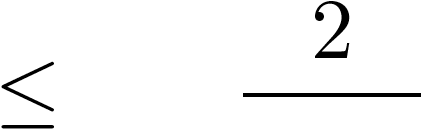
0 ≤ cnlogn ≤ nlogn − 2n + 13 for all n ≥ n0.

Since nlogn − 2n ≤ nlogn − 2n + 13,

we will instead show that

cnlogn ≤ nlogn − 2n,

which is equivalent to

c 1 − logn, when n > 1.

Ifsuffices. Thus ifn ≥ 8, then 2/c(log= 1n/)3≤and2/n3, and picking0 = 8, then for allc = 1/3 n ≥ n0, we have

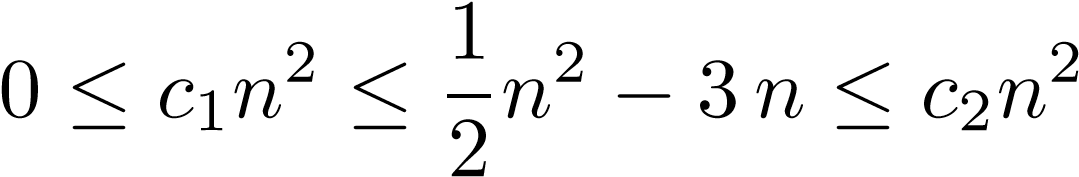
0 ≤ cnlogn ≤ nlogn − 2n ≤ nlogn − 2n + 13.

Thus (nlogn − 2n + 13) = Ω(nlogn).

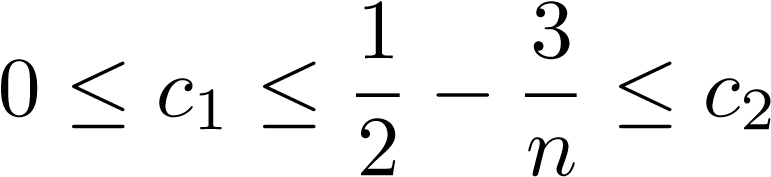
# Show that 12n2 − 3n = Θ(n2)

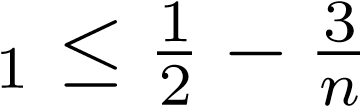
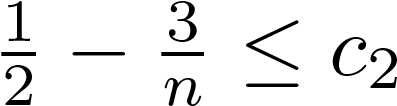
**Proof:**

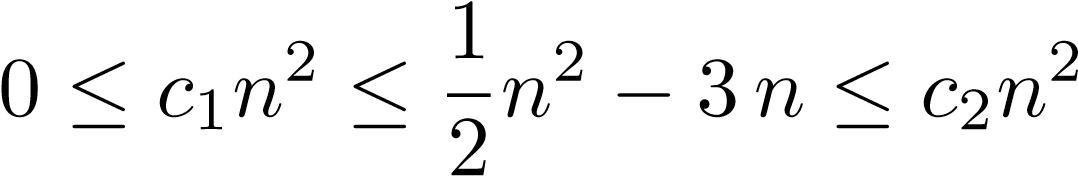
* We need to find positive constants c1, c2, and n0 such that

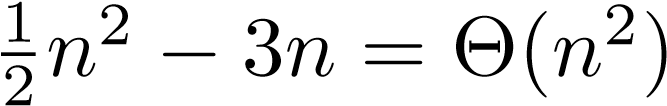
 for all n ≥ n0

* Dividing by n2, we get



* c holds for n ≥ 10 and c1 = 1/5
*  holds for n ≥ 10 and c2 = 1.
* Thus, if c1 = 1/5, c2 = 1, and n0 = 10, then for all n ≥ n0,

 for all n ≥ n0.

Thus we have shown that .

# 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(n2)

* We will see more examples later.

# Summary of the Notation

* f(n) ∈ O(g(n)) ⇒ f g
* f(n) ∈ Ω(g(n)) ⇒ f g
* f(n) ∈ Θ(g(n)) ⇒ f ≈ g
* It is important to remember that a Big-O bound is only an *upper bound*. So an algorithm that is O(n2) might not ever take that much time. It may actually run in O(n) time.
* Conversely, anan algorithm that isΩ bound is only aΩ(nlogn) might actually be*lower bound*. So

Θ(2n).

* Unlike the other bounds, aif an algorithm is Θ(n2), it runs in quadratic time.Θ-bound is precise. So,

# 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: Θ(k), for example Θ(1)
* Linear: Θ(n)
* Logarithmic: Θ(logk n)
* nlogn: Θ(nlogk n)
* Quadratic: Θ(n2)
* Polynomial: Θ(nk)
* Exponential: Θ(kn)

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

# Classification of algorithms - Θ(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 - Θ(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 - Θ(logn)

* A logarithmic function is the inverse of anx exponential function, i.e. b = n is equivalent to x = logb n)
* Always increases, but at a slower rate as n increases. (Recall that the derivative of logn is n1 , 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 - Θ(nlogn)

* Combination of O(n) and O(logn)
* 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(nlogn).

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

loga blogb n = logan.

**Classification of algorithms -** Θ(n2) • 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 Θ(nk) for constant k it is called a polynomial-time algorithm.

# Classification of algorithms - Θ(2n)

* 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 2n, incrementing a counter when a number has k nonzero bits. It is clear this is exponential in n.

# Comparison of growth rates

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| logn | n | nlogn | 2  n | 3  n | 2n |
| 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 |
| loglogm | logm |  |  |  | m |

# More growth rates

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | 100n | 2  n | 11n2 | 3  n | 2n |
| 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 | 2  n | 2  n − n | 2  n + 99 | 3  n | 3  n + 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 |



AsymptoticNotation



0

5000

10000

15000

20000

25000

30000

35000

40000

10

0

15

35

20

25

30

40

5

Polynomial Functions

x

x\*\*2

x\*\*3

x\*\*4



AsymptoticNotation



0

50

100

150

200

250

25

30

0

40

35

5

10

15

20

Slow Growing Functions

log(x)

x

x\*log(x)

x\*\*2



AsymptoticNotation



0

500

1000

1500

2000

2500

3000

3500

4000

4500

5000

6

8

10

0

2

4

Fast Growing Functions Part 1

x

x\*\*3

x\*\*4

\*\*x

2



AsymptoticNotation



0

50000

100000

150000

200000

250000

300000

350000

400000

450000

500000

0

10

15

20

5

Fast Growing Functions Part 2

x

x\*\*3

x\*\*4

2

\*\*x

Why Constants and Non-Leading Terms Don’t Matter

AsymptoticNotation



0

5e+07

1e+08

1.5e+08

2e+08

2.5e+08

3e+08

3.5e+08

4e+08

30

5

0

10

15

20

25

\*x

1000000

300000\*x\*\*2 + 300

\*x

\*\*x

2

AsymptoticNotation 31

|  |
| --- |
| **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 < logn < log2n < √n < n < nlogn n < nlogn < n(1.1) < n2 < n3 < n4 < 2n   * In other words, an algorithm that is Θ(nlog(n)) is more efficient than an algorithm that is Θ(n3). |