Back to our main discussion:

*The Big O-notation (Big Oh):* When we have only an *asymptotic upper bound*, we use the *Big Oh*, the *O*-notation.

Definition: For a given function *g*(*n*), we denote by *O*(*g*(*n*)), read *big Oh of g(n),* or *O of g(n),* or *order of g(n)*, the set of functions:

{*f*(*n*) | there exist positive constants *c* and *n*0 such that

*0* ≤ *f*(*n*) ≤ *cg*(*n*) for all *n* ≥ *n0*} = *O*(*g*(*n*)).

7The figure below shows the intuition behind *O*-notation.

cg(n)

f(n)

Figure 3.1 f(n) = g(n))

n

n0

What may be rather surprising is that any linear function *an* + *b* is in *O*(*n*2), which is easily verified by taking *c* = *a* + |*b*| and *n*0 = 1. Also we may find it strange that we should write, for example, *n* = *O*(*n*2). In this notes, however, when we write *f*(*n*) = *O*(*g*(*n*)) we are merely claiming that, some constant multiple of *g*(*n*) is an asymptotic upper bound on *f*(*n*), with no claim about how tight an upper bound is.

Using *O*-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure.

For example, the triply nested loop structure of the following algorithm:

*For (i = 1; i ≤ n; i++)*

*for (j = 1; j ≤ n; j++)*

*for (k = 1; k ≤ n; k++)*

*linear statements (no loops)*

*end for*

*end for*

*endfor*

immediately yields an *O*(*n*3) upper bound on the worst-case running time: the cost of the i-loop is bounded from above by *O*(*n*), the cost of the j-loop is bounded from above by *O*(*n*), therefore the cost of both loops together is bounded by *O*(*n*2), and so on.

As another example let’s consider the following loops:

*for (i = 1; i ≤ n; i++)*

*for (j1 = 1; j1 ≤ p; j1++)*

*for (k1 = 1; k1 ≤ p; k1++)*

*linear statements (no loops)*

*end for*

*end for*

*for (j2 = 1; j2 ≤ q; j2++)*

*for (k2 = 1; k2 ≤ q; k2++)*

*linear statements (no loops)*

*end for*

*end for*

*endfor*

“Clearly” this algorithm is O(n(p2 + q2)). The n-loop encloses the p2-loop and the q2-loop that are executed independently from each other.

That is, the first inner loop is by itself O(p2) and the second inner loop is by itself O(q2). The two inner loops executed one after the other are O(p2 + q2).

Since the n-loop encloses both of them the total upper bound is O(n(p2 + q2)) as stated above.he second inner loop is by itself O(q2). The two inner loops are O(p2 + q2

Since *O*-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well.

Thus, the *O*(*n*2) bound on worst-case running time of *insertion sort* also applies to its running time on every input.

Technically, it is an abuse to say that the running time of *insertion sort* is *O*(*n*2), since for a given *n*, the actual running time depends on the particular input of size *n*.

That is, strictly speaking, the running time is not really a function of *n*.

What we mean when we say "the running time is *O*(*n*2)" is that the worst-case running time (which is a function of *n*) is *O*(*n*2), or equivalently, no matter what particular input of size *n* is chosen for each *t* value of *n*, the running time on that set of inputs is *O*(*n*2).

Ω-notation (Omega): Just as *O*-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an *asymptotic lower bound*.

Definition: For a given function *g*(*n*), we denote by Ω(*g*(*n*)) the set of functions

{*f*(*n*) | there exist positive constants *c* and *n*0 such that

0 ≤ *cg*(n) ≤ *f*(*n*) for all *n* ≥ *n*0} = Ω(*g*(*n*)).

The intuition behind Ω-notation is shown in Figure 3.2 below. For all values *n* to the right of *n*0, the value of *f*(*n*) is on or above *g*(*n*).

cg(n)

f(n)

Figure 3.2 f(n) = g(n))

n

n0

Since Ω-notation describes a lower bound, when we use it to bound the best running time of the algorithm, by implication we also bound the running time of the algorithm on arbitrary inputs as well.

For example, the best-case running time of insertion sort is Ω(*n*), which implies that the running time of *insertion sort* is Ω(*n*).

The running time of *insertion sort* therefore falls between Ω(*n*) and *O*(*n*2), since it falls anywhere between a linear function of *n* and a quadratic function of *n*.

Moreover, these bounds are asymptotically as tight as possible: for instance, the running time of *insertion sort* is not Ω(*n*2), since *insertion sort* runs in Ω(*n*) time when the input is already sorted. It is not contradictory, however, to say that the *worst-case* running time of insertion sort is Ω(*n*2),since there exists an input that causes the algorithm to take Ω(*n*2) time.

When we say that the *running time* (no modifier) of an algorithm is Ω(*g*(*n*)), we mean that no matter what particular input of size *n* is chosen for each value of *n*, the running time on that set of inputs is at least a constant times *g*(*n*), for sufficiently large *n*.

Θ - notation (theta): The Θ-notation asymptotically bounds a function from above and below.

Definition: For a given function *g(n*), we denote by Θ(g(n)) (theta of g(n)) the set of functions

{f(n) | there exist positive constants c1, c2, and n0 such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0} = Θ(g(n)).

A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be "sandwiched" between c1g(n) and c2g(n), for sufficiently large n.

Although Θ(g(n)) is a set, we write "f(n) = Θ(g(n))" to indicate that f(n) is a member of Θ(g(n)), or "f(n) ∈ Θ(g(n))." Figure 3.3 below gives an intuitive picture of functions f(n) and g(n), where f(n) = Θ(g(n)).

For all values of n to the right of n0, the value of f(n) lies at or above c1g(n) and at or below c2g(n).

In other words, for all n ≥ n0, the function f(n) is equal to g(n) to within a constant factor.

We say that g(n) is an *asymptotically tight bound* for f(n).

c2g(n)

c1g(n)

f(n)

Figure 3.3 f(n) = (n)g(n)

n

n0

The definition of Θ(g(n)) requires that every member of Θ(g(n)) be *asymptotically nonnegative*, that is, that f(n) be nonnegative whenever n is sufficiently large.

*Theorem 3.1: For any two functions f(n) and g(n), f(n) = Θ(g(n)) iff f(n) = O(g(n)) and f(n) = Ω(g(n)).*

Certainly, other choices for the constants exist, but the important thing is that some choice exists.

As an example, consider any *positive* quadratic function with *a* > 0 (if a is not > 0, the function is not positive!).

*f*(*n*) = *an*2 + *bn* + *c*, where *a*, *b*, and *c* are arbitrary constants. Throwing away the lower-order terms and ignoring the constant yields *f*(*n*) = Θ(*n*2). Here we take the constant .

Proof:

We have that 0 ≤ *c*1*n*2 ≤ *an*2 + *ban* + *c* ≤ *c*2*n*2 for all *n* ≥ *n*0. That is,



Coefficient *a* is always positive but *b* and *c* may not be. So we write:



We want the two terms (possibly negative) inside the parenthesis to be less than 1 (the 1 is the dominant term), that is:  and , therefore the value of n0 is given by . (Equivalently make *an2* ≥ *|b|n* 🡺 *n*≥ *|b|/a*  and *an2* ≥ *|c| 🡺 n* ≥ sqrt(*|c|/a). Done!*).

In general, for any polynomial





In the last expression is clear that the only term guaranteed to be positive is *ad* (for the function to be positive for large *n’*s). That is the reason why the remaining *ai*’s constants are put between vertical bars to make them positive. Making *n = 1* give us the largest term in parenthesis and thus the value of c2 when proving that *p*(*n*) = Θ(*nd*). The value of c1 is calculated from the constants in the expression of *p(n)* for values of *n* such that the expression begins to be positive as seen in the example given in the previous class.

Since any constant is a degree 0 polynomial, we can express any constant function as Θ(*n*0), or Θ(1).

This latter notation is a minor abuse, however, because it is not clear what variable is tending to infinity.

We shall often use the notation Θ(1) to mean either a constant or a constant function with respect to some variable.

Asymptotic notation in equations

We have already seen how asymptotic notation can be used within mathematical formulas.

For example, in introducing *O*-notation, we wrote "*n* = *O*(*n*2)."

We might also write 2*n*2 + 3*n* + 1 = 2*n*2 + Θ(*n)*.

How do we interpret such formulas?

When the asymptotic notation stands alone on the right-hand side of an equation, as in *n* = *O*(*n*2), we have already defined the equal sign to mean set membership: *n* ∈ *O*(*n*2).

In general, however, when asymptotic notation appears in a formula, we interpret it as standing for some anonymous function that we do not care to name.

For example, the formula 2*n*2 + 3*n* + 1 = 2*n*2 + Θ(*n*) means that 2*n*2 + 3*n* + 1 = 2*n*2 + *f*(*n*), where *f*(*n*) is some function in the set Θ(*n*). In this case, *f*(*n*) = 3*n* + 1, which indeed is in Θ(*n*).

Using asymptotic notation in this manner can help eliminate inessential detail and clutter in an equation.

For example, we can express the worst-case running time of merge sort as the recurrence

*T*(*n*) = 2*T*(*n*/2) + Θ(*n*) .

If we are interested only in the asymptotic behavior of *T*(*n*), there is no point in specifying all the lower-order terms exactly; they are all understood to be included in the anonymous function denoted by the term Θ(*n*).

The number of anonymous functions in an expression is understood to be equal to the number of times the asymptotic notation appears.

For example, in the expression



there is only a single anonymous function (a function of *i*).

This expression is thus *not* the same as

*O(1) + O(2) + . . . +O(n),*

which doesn't really have a clean interpretation.

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

2*n*2 + Θ(*n*) = Θ(*n*2).

We interpret such equations using the following rule:

*No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid.*

Thus, the meaning of our previous example is that for any function *f*(*n*) ∈ Θ(n), there is *some* function *g*(*n*) ∈ Θ(*n*2) such that 2*n*2 + *f*(*n*) = *g*(*n*) for all *n*.

In other words, the right-hand side of an equation provides coarser level of detail than the left-hand side.

A number of such relationships can be chained together, as in

2*n*2 + 3*n* + 1 = 2*n*2 + Θ(*n*)

= Θ(*n*2) .

We can interpret each equation separately by the rule above.

3.2 Calculating the Running Time of a Program

Before presenting these principles, it is important that we review how to *add* and *multiply* in "big oh" notation.

(Ex. 3 previous class). If T1(n) and T2(n) are the running times of two program fragments P1 and P2, and T1(n) is O(f(n)) and T2(n) is O(g(n)). Then T1(n) + T2(n), the running time of P1 followed by P2, is O(max(f(n), g(n)).

We proved this as follows: for constants c1, c2, n1, and n2,

T1 (n) ≤ c1f(n), for n ≥ n1

and

T2(n) ≤ c2g(n), if n ≥ n2.

Let n0 = max(n1, n2). Then

T1(n) + T2(n) ≤ c1f(n) + c2g(n), for n ≥ n0

From this we conclude that if n ≥ n0, then

T1(n) + T2(n) ≤ (c1 + c2)max(f(n), g(n)) = c\*max(f(n), g(n))

Where c = c1 + c2. Therefore

T1(n) + T2(n) = O(max(f(n), g(n)).

The rule for sums given above can be used to calculate the running time of a sequence of program steps, where each step may be an arbitrary program fragment with loops and branches.

Example

Suppose that we have three steps (subprograms) whose running times are, respectively,

O(n2), O(n3) and O(n log n).

Then the running time of the first two steps executed sequentially is:

O(max(n2, n3) = O(n3).

The running time of all three together is:

O(max(n3, n log n )) = O(n3).

In general, the running time of a fixed sequence of steps is, to within a constant factor, the running time of the step with the largest running time.

In rare circumstances there will be two or more steps whose running times are incommensurate (neither is larger than the other, nor are they equal).

For example, (returning to an important example) we could have steps of running times O(f(n)) and O(g(n)), where:



In such cases the sum rule must be applied directly; O(f(n) +O(g(n)) is O(max(f(n), g(n)), that is, n4 if n is even and n3 if n is odd.

Another useful observation about the sum rule is that if g(n) ≤ f(n) for all n above some constant n0, then O(f(n) + g(n)) is the same as O(f(n)).

For example, O(n2 + n) is the same as O(n2).

The *rule for products* is the following.

If T1(n) and T2(n) are O(f(n)) and O(g(n)), respectively, then T1(n)T2(n) is O(f(n)g(n)).

You should prove this fact using the same ideas as in the proof of the sum rule.

It follows from the product rule that *O(cf(n))* means the same thing as *O(f(n))* if *c* is any positive constant.

For example, O(n2/2) is the same as O(n2).

Exercises Measuring Time Complexity:

Exercise 1.

Find the time complexity of the following algorithm:

begin

for i = 1 to n do

for j = 1 to n do begin

C[i, j] = 0;

for k = 1 to n do

C[i, j] = C[i, j] + A[i, k]\*B[k, j]

end

end

Solution:

## In this case it should be clear that the “k-loop” is O(n) since it executes n times. (In fact the “for” line executes once more but this is irrelevant for an “order of magnitude” measurement). Now the “j-loop” also executes n times and both loops together execute O(n2). Lastly, including the “i-loop” which also runs n times, all three loops together execute order O(n3).

## Therefore the Time Complexity of this algorithm is O(n3).

Exercise 2.

Calculate in detail the time complexity of the following algorithm

begin

for i = 1 to n do

if odd(i) then begin

for j = i to n do

x = x + 1;

for j = 1 to i do

y = y + 1;

end

end

Solution:

We realize that the external loop, the i-loop”, executes n times (in fact the “for” line of code executes n+1 times). However the internal condition that n be odd makes the body of the first “for” loop execute only n/2 times (approximately). Therefore the two inner loops, the two “j-loops”, execute n/2 times. Since when evaluating the asymptotic order a constant is irrelevant (we know that O(n/2) and O(n) or for that matter any O(cn) are in fact all O(n)), then let’s just calculate as close as possible without thinking about the small details. If we call T(n) the total running time of this algorithm then:



where the “1s” stand for the constants inside the loop (O(1)). The first internal sum can be divided into two sums: a sum from 1 to n minus a sum from 1 to i-1 (this makes the total sum from i to n. The second internal sum is just i – 1 + 1 = i. We have then:



Exercise 3.

Prove that the following algorithm is O(n3)

begin

{some statements requiring O(1) time}

for i = 1 to n do

for j = i+2 to n do

for k = 1 to j do

{some statements requiring O(1) time}

{some statements requiring O(1) time}

end

Solution:

We proceed as in the last exercise:



YOU DO THE ALGEBRA! … the rest is just a trivial use of known formulas (as done in class).

Now let’s try some harder exercises!

Exercise 4.

i = n

while (i >= 1) {

for j = 1 to i

x = x + 1 //O(1) statement

i = i/2

}

Solution:

We can see that initially *i = n.* Let’s make a table to “see” how the while loop works.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| iteration  number | 1 | 2 | 3 | ....... | k |
| j = 1 to i | 1 to n | 1 to n/2 | 1 to n/22 | ....... | 1 to n/2k-1 |
| # of iterations  in the loop | n | n/2 | n/22 | ....... | n/2k-1 |

Therefore the total number of iterations is:

T(n) = n + n/2 + n/22 + n/23 + .... + n/2k-1 =

= n[1 + 1/2 + 1/22 + 1/23 + .... + 1/2k-1]

T(n) = n[1 - 1/2k]/[1 – 1/2] = 2n[1 - 1/2k]

Now, we assume that at the kth iteration n was reduced to the minimum and that at the next iteration i was reduced to “slightly less than 1” and we can approximate that the *while* loop was exited when:

n/2k = 1 => 2k = n => k = lg n

and thus:

T(n) = 2n[1 – 1/n] = 2n – 1 = O(2n) = O(n)

Notice the difference between this exercise and the next.

# Exercise 5

Assume now that the problem was:

i = n

while (i >= 1) {

for j = 1 to n

x = x + 1 //O(1) statement

i = i/2

}

Solution:

In this case the for loop inside the while loop always executes *n* times

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| iteration  number | 1 | 2 | 3 | ....... | k |
| value of i : | 1 to n | 1 to n/2 | 1 to n/22 | ....... | 1 to n/2k-1 |
| # of iterations  in the loop | n | n | n | ....... | n |

Thus in total we have that there were n\*k iterations. Therefore:

T(n) = n\*k

Now, using the same assumption as in the previous exercise at the kth kth iteration n was reduced to the minimum which in iterations is just ONE iteration, that is:

n/2k = 1 => k = lg n.

Therefore:

T(n) = n\*lg n = O(n lg n)

# Exercise 6

Assume now that the problem was:

i = n

while (i >= 1) {

for j = 1 to n

x = x + 1 //O(1) statement

i = i/3

}

Solution:

In this case proceed as before because this is just an equivalent exercise with different logs and with the loop stopping at n/3k = 1.

IS ALGORITHM TIME COMPLEXITY IMPORTANT?

(Taken from “The Design and Analysis of Computer Algorithms” & “Data Structures and Algorithms” both by Aho, Hopcroft & Ullman, 1974).

One might suspect that the tremendous increase in the speed of calcula­tions brought about by the advent of the present generation of digital com­puters would decrease the importance of efficient algorithms. However, just the opposite is true. As computers become faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be achieved with an increase in computer speed.

Suppose we have five algorithms A1 – A5, with the following time com­plexities.

Algorithm Time Complexity

A1 n

A2 n lg n

A3 n2

A4 n3

A5 2n

The time complexity here is the number of time units required to process an input of size *n*. Assuming that one unit of time equals one millisecond, algo­rithm A1 can process in one second an input of size 1000, whereas algorithm A5, can process in one second an input of size at most 9. The details of some computations are as follows: A1 makes n = 1000msec at once; A3 makes n2 = 1000 🡺 n = 31.82 🡺 n = 31msec; A5 makes 2n = 1000 🡺 n = lg 1000 = 9.97 🡺 9msec. If the time is a minute then A1 makes n = 60,000msec, A3 makes n2 = 60,000 🡺 n = 244.95 🡺 n = 244msec; A5 makes 2n = 60,000 🡺 n = lg 60,000 = 15.87 🡺 15msec. Figure 1.1 summarizes the sizes of problems that can be solved in one second, one minute, and one hour by each of these five algorithms.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time  Complexity | Maximum Problem Size | | |
| 1 sec | 1 min | 1 hour |
| A1 | n | 1000 | 60,000 | 3.6\*106 |
| A2 | n lg n | 140 | 4,893 | 2.0\*105 |
| A3 | n2 | 31 | 244 | 1897 |
| A4 | n3 | 10 | 39 | 153 |
| A5 | 2n | 9 | 15 | 21 |

Fig. 1-1. Limits on problem size as determined by growth rate.

Suppose that the next generation of computers is ten times faster than the current generation. Figure 1.2 shows the increase in the size of the problem we can solve due to this increase in speed. Note that with algorithm A1, a tenfold increase in speed increases ten times because the algorithm is linear. With algorithm A2, the problem is quasi-linear for large s2. With algorithm A3 the increase is proportional to the square root of ten (which is 3.16) while with algorithm A4 the increase is proportional to the cubic root of ten (which is 2.15). Lastly with algorithm A5, a tenfold increase in speed only increases by three the size of problem that can be solved (lg 10 = 3.3), whereas with for example algorithm A3 the size more than triples.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Time  Complexity | Maximum Problem  Size before speed up | Maximum Problem  Size After speed up |
|
| A1 | n | s1 | 10 s1 |
| A2 | n lg n | s2 | ~ 10 s2 |
| A3 | n2 | s3 | 3.16 s3 |
| A4 | n3 | s4 | 2.15 s4 |
| A5 | 2n | s5 | s5 + 3.3 |

Fig. 1-2. Effect of tenfold speed up

Instead of an increase in speed, consider the effect of using a more efficient algorithm. Refer again to Fig. 1.1. Using one minute as a basis for com­parison, by replacing algorithm A4 with A3 we can solve a problem six times larger; by replacing A4 with A2 we can solve a problem 125 times larger.

These results are far more impressive than the twofold improvement obtained by a tenfold increase in speed. If an hour is used as the basis of comparison, the differences are even more significant. We conclude that the asymptotic com­plexity of an algorithm is an important measure of the goodness of an algorithm, one that promises to become even more important with future increases in computing speed.

Despite our concentration on order-of-magnitude performance, we should realize that an algorithm with a rapid growth rate might have a smaller con­stant of proportionality than one with a lower growth rate. In that case, the rapidly growing algorithm might be superior for small problems, possibly even for all problems of a size that would interest us.

For example, suppose the time complexities of algorithms A1, A2, A3, A4, and A5 were really l000n, l00n lg n, 10n2, n3, and 2n. Then:

A5 would be best for problems of size 2 ≤ n ≤ 9,

A3 would be best for 10 ≤ n ≤ 58,

A2 would be best for 59 ≤ n ≤ 1024, and

A1 best for problems of size greater than 1024.

To see this it’s enough to plot all of these functions of *n* versus *n* and see which one is better and in what range.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| n | A1: f(n) = l000n | A2: f(n) = l00n lg n | A3: f(n) = 10n2 | A4: f(n) = n3 | A5: f(n) = 2n |
| 2 | 2000 | 200 | 40 | 8 | 4 |
| 8= 23 | 8000 | 2400 | 640 | 512 | 256 |
| 9 | 9000 | 2853 | 810 | 729 | 512 |
| 10 | 10000 | 3322 | 1000 | 1000 | 1024 |
| 58 | 58000 | 33977 | 33640 | 195192 | 2.8823\*1017 |
| 59 | 59000 | 34708 | 34810 | 205379 | 5.7646\*1017 |
| 1024=210 | 1024000 | 1024000 | 1.04857\*108 | 1.07374\*109 | inf |
| 1025 | 1025000 | 10251443 | 1.05062\*108 | 1.07689\*109 | inf |