Unit - 1

Basics of Algorithm Analysis

- Analyzing algorithms involves thinking about how their resource requirements—the amount of time and space they use—will scale with increasing input size.
- To a rich understanding of computational tractability.
- Having done this, we develop the mathematical machinery needed to talk about the way in which different functions scale with increasing input size, making precise what it means for one function to grow faster than another.
- We then develop running-time bounds for some basic algorithms,
- To a survey of many different running times and certain characteristic types of algorithms that achieve these running times.
- In some cases, obtaining a good running-time bound relies on the use of more sophisticated data structures.

Computational Tractability

- To find efficient algorithms for computational problems.
- Design principles in the development of algorithms.
- The basic approaches to designing efficient algorithms.
- The general notion of computational efficiency - primarily on efficiency in running time.
- We want algorithms that run quickly.
- But it is important that algorithms be efficient in their use of other resources as well. In particular, the amount of *space* (or memory) used by an algorithm is an issue.

Some Initial Attempts at Defining Efficiency

- Proposed Definition of Efficiency (1): *An algorithm is efficient if, when implemented, it runs quickly on real input instances.*
- There is a significant area of research devoted to the careful implementation and profiling of different algorithms for discrete computational problems.
- This proposed definition above does not consider how well, or badly, an algorithm may *scale* as problem sizes grow to unexpected levels.
- A common situation is that two very different algorithms will perform comparably on inputs of size 100; multiply the input size tenfold, and one will still run quickly while the other consumes a huge amount of time.
- Concrete definition of efficiency that is platform-independent, instance-independent, and of predictive value with respect to increasing input sizes.

Some Initial Attempts at Defining Efficiency

- We need to take a more mathematical view of the situation.
- In considering the problem, we will seek to describe an algorithm at a high level, and then analyze its running time mathematically as a function of this input size *N*.

Worst-Case Running Times and Brute-Force Search

- We will look for a bound on the largest possible running time the algorithm could have over all inputs of a given size *N*, and see how this scales with *N*.
- In general the worst-case analysis of an algorithm has been found to do a reasonable job of capturing its efficiency in practice.
- Average-case analysis—the obvious appealing alternative, in which one studies the performance of an algorithm averaged over "random" instances.
- In general we will think about the worst-case analysis of an algorithm's running time.
- But what is a reasonable analytical benchmark that can tell us whether a running-time bound is impressive or weak?
- A first simple guide is by comparison with brute-force search over the search space of possible solutions.

Worst-Case Running Times and Brute-Force Search

- A compact representation, implicitly specifying a giant search space.
- For most of these problems, there will be an obvious brute-force solution: try all possibilities and see if any one of them works. Not only is this approach almost always too slow to be useful, it provides us with absolutely no insight into the structure of the problem we are studying.

Worst-Case Running Times and Brute-Force Search

- Proposed Definition of Efficiency (2): *An algorithm is efficient if it achieves qualitatively better worst-case performance, at an analytical level, than brute-force search.*
- Algorithms that improve substantially on brute-force search nearly always contain a valuable heuristic idea that makes them work; and they tell us something about the intrinsic structure, and computational tractability, of the underlying problem itself.
- What do we mean by “qualitatively better performance?” This suggests that we consider the actual running time of algorithms more carefully, and try to quantify what a reasonable running time would be.

Polynomial Time as a Definition of Efficiency

- How to quantify the notion of a “reasonable” running time.
- Search spaces for natural combinatorial problems tend to grow exponentially in the size *N* of the input; if the input size increases by one, the number of possibilities increases multiplicatively.
- We’d like a good algorithm for such a problem to have a better scaling property: when the input size increases by a constant factor—say, a factor of 2—the algorithm should only slow down by some constant factor *C*.

Polynomial Time as a Definition of Efficiency

- Suppose an algorithm has the following property: There are absolute constants $c > 0$ and $d > 0$ so that on every input instance of size N, its running time is bounded by cN^d primitive computational steps. (In other words, its running time is at most proportional to N^d.)
- if this running-time bound holds, for some c and d, then we say that the algorithm has a *polynomial running time.*
- If the input size increases from N to $2N$, the bound on the running time increases from cN^d to $c(2N)^d = c \cdot 2^d N^d$, which is a slow-down by a factor of $2d$. Since d is a constant, so is 2^d;

Polynomial Time as a Definition of Efficiency

- Proposed Definition of Efficiency (3): *An algorithm is efficient if it has a polynomial running time.*
- Wouldn't an algorithm with running time proportional to n^{100} —and hence polynomial—be hopelessly inefficient? Wouldn't we be relatively pleased with a nonpolynomial running time of $n^{1+.02(\log n)}$? The answers are, of course, "yes".
- Problems for which polynomial-time algorithms exist almost invariably turn out to have algorithms with running times proportional to very moderately growing polynomials like n, $n \log n$, n^2, or n^3.

Polynomial Time as a Definition of Efficiency

- An algorithm with exponential worst-case behavior generally runs well on the kinds of instances that arise in practice; and there are also cases where the best polynomial-time algorithm for a problem is completely impractical due to large constants or a high exponent on the polynomial bound.

Polynomial Time as a Definition of Efficiency

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Polynomial Time as a Definition of Efficiency

- It becomes possible to express the notion that *there is no efficient algorithm for a particular problem*.
- In a sense, being able to do this is a prerequisite for turning our study of algorithms into good science, for it allows us to ask about the existence or nonexistence of efficient algorithms as a well-defined question.

Polynomial Time as a Definition of Efficiency

- In particular, the first of our definitions, which was tied to the specific implementation of an algorithm, turned efficiency into a moving target: as processor speeds increase, more and more algorithms fall under this notion of efficiency.
- Our definition in terms of polynomial time is much more an absolute notion; it is closely connected with the idea that each problem has an intrinsic level of computational tractability: some admit efficient solutions, and others do not.

Asymptotic Order of Growth

- Our discussion of computational tractability has turned out to be intrinsically based on our ability to express the notion that an algorithm's worst-case running time on inputs of size *n* grows at a rate that is at most proportional to some function *f (n)*.
- The function *f (n)* then becomes a bound on the running time of the algorithm.
- When we provide a bound on the running time of an algorithm, we will generally be counting the number of such pseudo-code steps that are executed;

Asymptotic Order of Growth

- When we seek to say something about the running time of an algorithm on inputs of size n, one thing we could aim for would be a very concrete statement such as, "On any input of size n, the algorithm runs for at most $1.62n^2 + 3.5n + 8$ steps."
- First, getting such a precise bound may be an exhausting activity, and more detail than we wanted anyway.
- Second, because our ultimate goal is to identify broad classes of algorithms that have similar behavior, we'd actually like to classify running times at a coarser level of granularity so that similarities among different algorithms, and among different problems, show up more clearly.
- Finally, extremely detailed statements about the number of steps an algorithm executes are often—in a strong sense—meaningless.

Asymptotic Order of Growth

- The notion of a “step” may grow or shrink by a constant factor.
- For example, if it takes 25 low-level machine instructions to perform one operation in our high-level language, then our algorithm that took at most $1.62n^2 + 3.5n + 8$ steps can also be viewed as taking $40.5n^2 + 87.5n + 200$ steps when we analyze it at a level that is closer to the actual hardware.

Asymptotic Order of Growth

O, Ω, and Θ

- For all these reasons, we want to express the growth rate of running times and other functions in a way that is insensitive to constant factors and loworder terms.
- In other words, we'd like to be able to take a running time like the one we discussed above, $1.62n^2 + 3.5n + 8$, and say that it grows like n^2, up to constant factors.

Asymptotic Bounds

- Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.
- Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
- Tight bounds. $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

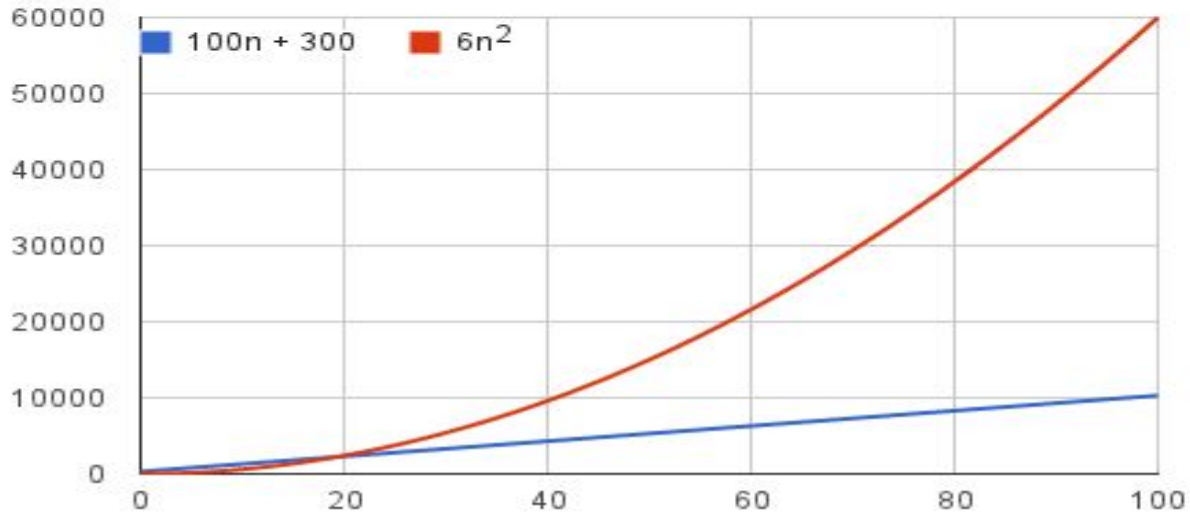
Asymptotic Bounds

example, is the conclusion we can draw from the fact that $T(n) = pn^2 + qn + r$ is both $O(n^2)$ and $\Omega(n^2)$.

There is a notation to express this: if a function $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$, we say that $T(n)$ is $\Theta(f(n))$. In this case, we say that $f(n)$ is an *asymptotically tight bound* for $T(n)$. So, for example, our analysis above shows that $T(n) = pn^2 + qn + r$ is $\Theta(n^2)$.

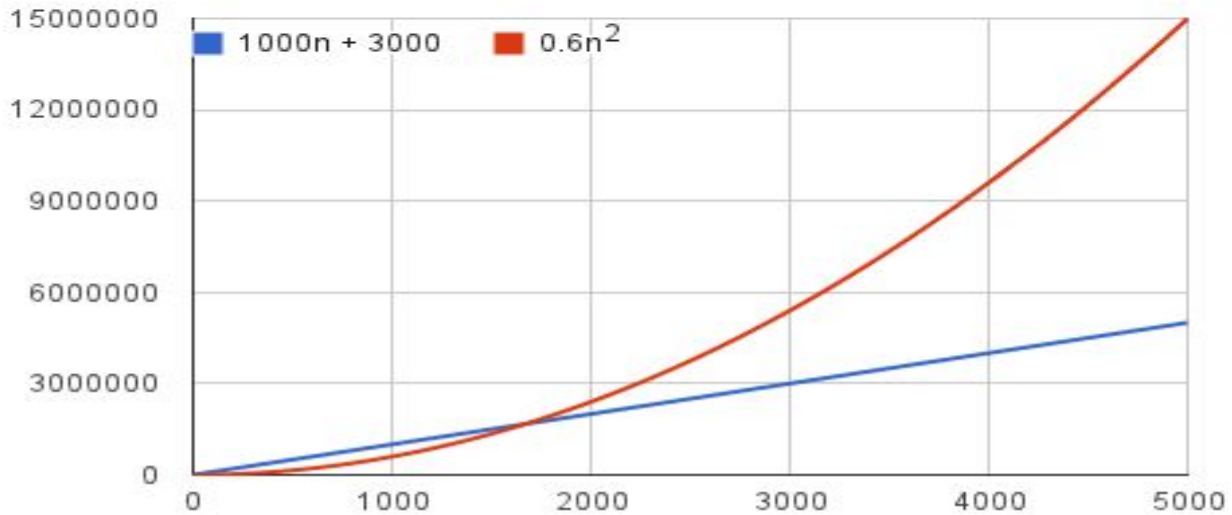
Asymptotic Order of Growth

The second idea is that we must focus on how fast a function grows with the input size. We call this the **rate of growth** of the running time. To keep things manageable, we need to simplify the function to distill the most important part and cast aside the less important parts. For example, suppose that an algorithm, running on an input of size n, takes $6n^2 + 100n + 300$ machine instructions. The $6n^2$ term becomes larger than the remaining terms, $100n + 300$, once n becomes large enough, 20 in this case. Here's a chart showing values of $6n^2$ and $100n + 300$ for values of n from 0 to 100:



Asymptotic Order of Growth

We would say that the running time of this algorithm grows as n^2, dropping the coefficient 6 and the remaining terms $100n + 300$. It doesn't really matter what coefficients we use; as long as the running time is $an^2 + bn + c$, for some numbers $a > 0$, b, and c, there will always be a value of n for which an^2 is greater than $bn + c$, and this difference increases as n increases. For example, here's a chart showing values of $0.6n^2$ and $1000n + 3000$ so that we've reduced the coefficient of n^2 by a factor of 10 and increased the other two constants by a factor of 10:



Asymptotic Order of Growth

Here's a list of functions in asymptotic notation that we often encounter when analyzing algorithms, ordered by slowest to fastest growing:

1. $\Theta(1)$
2. $\Theta(\log_2 n)$
3. $\Theta(n)$
4. $\Theta(n \log_2 n)$
5. $\Theta(n^2)$
6. $\Theta(n^2 \log_2 n)$
7. $\Theta(n^3)$
8. $\Theta(2^n)$
9. $\Theta(n!)$

Comparing Function Growth

Rank these functions according to their growth, from slowest growing (at the top) to fastest growing (at the bottom).

- 1
- n^2
- $(3/2)^n$
- n
- 2^n
- n^3

Comparing Function Growth

Which kind of growth best characterizes each of these functions?

	Constant	Linear	Polynomial	Exponential
$2n^3$	◯	◯	◯	◯
1000	◯	◯	◯	◯
$(3/2)^n$	◯	◯	◯	◯
2^n	◯	◯	◯	◯
$3n$	◯	◯	◯	◯
1	◯	◯	◯	◯
$(3/2)n$	◯	◯	◯	◯
$3n^2$	◯	◯	◯	◯

Comparing Function Growth

	Constant	Linear	Polynomial	Exponential
$2n^3$	○	○	●	○
1000	●	○	○	○
$(3/2)^n$	○	○	○	●
2^n	○	○	○	●
$3n$	○	●	○	○
1	●	○	○	○
$(3/2)n$	○	●	○	○
$3n^2$	○	○	●	○

Comparing Function Growth

- Rank these functions according to their growth, from slowest growing to fastest growing.

8^{2n}, $n \log_2 n$, $n \log_6 n$, 64, $4n$, $\log_8 n$, $\log_2 n$, $8n^2$, $6n^3$

Comparing Function Growth

- Rank these functions according to their growth, from slowest growing to fastest growing.

8^{2n}, n $\log_2$n, n $\log_6$n, 64, 4n, $\log_8$ n, $\log_2$ n, $8n^2$, $6n^3$

To conclude, the correct order of the functions would be:

- 64
- $\log_8 n$
- $\log_2 n$
- $4n$
- $n \log_6 n$
- $n \log_2 n$
- $8n^2, 6n^3$
- 8^{2n}

Asymptotic Bounds

(2.1) *Let f and g be two functions that*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is equal to some number $c > 0$. Then $f(n) = \Theta(g(n))$.

Proof. We will use the fact that the limit exists and is positive to show that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, as required by the definition of $\Theta(\cdot)$.

Since

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0,$$

it follows from the definition of a limit that there is some n_0 beyond which the ratio is always between $\frac{1}{2}c$ and $2c$. Thus, $f(n) \leq 2cg(n)$ for all $n \geq n_0$, which implies that $f(n) = O(g(n))$; and $f(n) \geq \frac{1}{2}cg(n)$ for all $n \geq n_0$, which implies that $f(n) = \Omega(g(n))$. ■

Properties of Asymptotic Growth Rate

Transitivity A first property is *transitivity*: if a function f is asymptotically upper-bounded by a function g, and if g in turn is asymptotically upper-bounded by a function h, then f is asymptotically upper-bounded by h. A similar property holds for lower bounds. We write this more precisely as follows.

(2.2)

(a) If $f = O(g)$ and $g = O(h)$, then $f = O(h)$.

(b) If $f = \Omega(g)$ and $g = \Omega(h)$, then $f = \Omega(h)$.

Properties of Asymptotic Growth Rate

Proof. We'll prove part (a) of this claim; the proof of part (b) is very similar.

For (a), we're given that for some constants c and n_0, we have $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n_0', we have $g(n) \leq c'h(n)$ for all $n \geq n_0'$. So consider any number n that is at least as large as both n_0 and n_0'. We have $f(n) \leq cg(n) \leq cc'h(n)$, and so $f(n) \leq cc'h(n)$ for all $n \geq \max(n_0, n_0')$. This latter inequality is exactly what is required for showing that $f = O(h)$. ■

Combining parts (a) and (b) of (2.2), we can obtain a similar result for asymptotically tight bounds. Suppose we know that $f = \Theta(g)$ and that $g = \Theta(h)$. Then since $f = O(g)$ and $g = O(h)$, we know from part (a) that $f = O(h)$; since $f = \Omega(g)$ and $g = \Omega(h)$, we know from part (b) that $f = \Omega(h)$. It follows that $f = \Theta(h)$. Thus we have shown

(2.3) *If $f = \Theta(g)$ and $g = \Theta(h)$, then $f = \Theta(h)$.*

Properties of Asymptotic Growth Rate

Sums of Functions It is also useful to have results that quantify the effect of adding two functions. First, if we have an asymptotic upper bound that applies to each of two functions f and g, then it applies to their sum.

(2.4) *Suppose that f and g are two functions such that for some other function h, we have $f = O(h)$ and $g = O(h)$. Then $f + g = O(h)$.*

Proof. We're given that for some constants c and n_0, we have $f(n) \leq ch(n)$ for all $n \geq n_0$. Also, for some (potentially different) constants c' and n'_0, we have $g(n) \leq c'h(n)$ for all $n \geq n'_0$. So consider any number n that is at least as large as both n_0 and n'_0. We have $f(n) + g(n) \leq ch(n) + c'h(n)$. Thus $f(n) + g(n) \leq (c + c')h(n)$ for all $n \geq \max(n_0, n'_0)$, which is exactly what is required for showing that $f + g = O(h)$. ■

Properties of Asymptotic Growth Rate

(2.5) *Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k$ and h be functions such that $f_i = O(h)$ for all i. Then $f_1 + f_2 + \cdots + f_k = O(h)$.*

(2.6) *Suppose that f and g are two functions (taking nonnegative values) such that $g = O(f)$. Then $f + g = \Theta(f)$. In other words, f is an asymptotically tight bound for the combined function $f + g$.*

Proof. Clearly $f + g = \Omega(f)$, since for all $n \geq 0$, we have $f(n) + g(n) \geq f(n)$. So to complete the proof, we need to show that $f + g = O(f)$.

But this is a direct consequence of (2.4): we're given the fact that $g = O(f)$, and also $f = O(f)$ holds for any function, so by (2.4) we have $f + g = O(f)$. ■

Asymptotic Bounds for Some Common Functions

Some common functions - polynomials, logarithms, and exponentials

Polynomials Recall that a polynomial is a function that can be written in the form $f(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$ for some integer constant $d > 0$, where the final coefficient a_d is nonzero. This value d is called the *degree* of the polynomial. For example, the functions of the form $pn^2 + qn + r$ (with $p \neq 0$) that we considered earlier are polynomials of degree 2.

A basic fact about polynomials is that their asymptotic rate of growth is determined by their "high-order term"—the one that determines the degree.

Asymptotic Bounds for Some Common Functions

(2.7) *Let f be a polynomial of degree d, in which the coefficient a_d is positive. Then $f = O(n^d)$.*

Proof. We write $f = a_0 + a_1 n + a_2 n^2 + \cdots + a_d n^d$, where $a_d > 0$. The upper bound is a direct application of (2.5). First, notice that coefficients a_j for $j < d$ may be negative, but in any case we have $a_j n^j \leq |a_j| n^d$ for all $n \geq 1$. Thus each term in the polynomial is $O(n^d)$. Since f is a sum of a constant number of functions, each of which is $O(n^d)$, it follows from (2.5) that f is $O(n^d)$. ■

One can also show that under the conditions of (2.7), we have $f = \Omega(n^d)$, and hence it follows that in fact $f = \Theta(n^d)$.

Asymptotic Bounds for Some Common Functions

So algorithms with running-time bounds like $O(n^2)$ and $O(n^3)$ are polynomial-time algorithms. But it's important to realize that an algorithm can be polynomial time even if its running time is not written as n raised to some integer power. To begin with, a number of algorithms have running times of the form $O(n^x)$ for some number x that is not an integer. For example, in Chapter 5 we will see an algorithm whose running time is $O(n^{1.59})$; we will also see exponents less than 1, as in bounds like $O(\sqrt{n}) = O(n^{1/2})$.

Asymptotic Bounds for Some Common Functions

Logarithms Recall that $\log_b n$ is the number x such that $b^x = n$. One way to get an approximate sense of how fast $\log_b n$ grows is to note that, if we round it down to the nearest integer, it is one less than the number of digits in the base-b representation of the number n. (Thus, for example, $1 + \log_2 n$, rounded down, is the number of bits needed to represent n.)

So logarithms are very slowly growing functions. In particular, for every base b, the function $\log_b n$ is asymptotically bounded by every function of the form n^x, even for (noninteger) values of x arbitrary close to 0.

Asymptotic Bounds for Some Common Functions

(2.8) *For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$.*

One can directly translate between logarithms of different bases using the following fundamental identity:

$$\log_a n = \frac{\log_b n}{\log_b a}.$$

This equation explains why you'll often notice people writing bounds like $O(\log n)$ without indicating the base of the logarithm. This is not sloppy usage: the identity above says that $\log_a n = \frac{1}{\log_b a} \cdot \log_b n$, so the point is that $\log_a n = \Theta(\log_b n)$, and the base of the logarithm is not important when writing bounds using asymptotic notation.

Asymptotic Bounds for Some Common Functions

Exponentials Exponential functions are functions of the form $f(n) = r^n$ for some constant base r. Here we will be concerned with the case in which $r > 1$, which results in a very fast-growing function.

In particular, where polynomials raise n to a fixed exponent, exponentials raise a fixed number to n as a power; this leads to much faster rates of growth. One way to summarize the relationship between polynomials and exponentials is as follows.

Asymptotic Bounds for Some Common Functions

(2.9) *For every $r > 1$ and every $d > 0$, we have $n^d = O(r^n)$.*

In particular, every exponential grows faster than every polynomial. And as we saw in Table 2.1, when you plug in actual values of n, the differences in growth rates are really quite impressive.

Just as people write $O(\log n)$ without specifying the base, you'll also see people write "The running time of this algorithm is exponential," without specifying which exponential function they have in mind. Unlike the liberal use of $\log n$, which is justified by ignoring constant factors, this generic use of the term "exponential" is somewhat sloppy. In particular, for different bases $r > s > 1$, it is never the case that $r^n = \Theta(s^n)$. Indeed, this would require that for some constant $c > 0$, we would have $r^n \leq cs^n$ for all sufficiently large n. But rearranging this inequality would give $(r/s)^n \leq c$ for all sufficiently large n. Since $r > s$, the expression $(r/s)^n$ is tending to infinity with n, and so it cannot possibly remain bounded by a fixed constant c.

Asymptotic Bounds for Some Common Functions

- Logarithms, polynomials, and exponentials serve as useful landmarks in the range of possible functions that you encounter when analyzing running times.
- Logarithms grow more slowly than polynomials, and polynomials grow more slowly than exponentials.

A Survey of Common Running Times

Linear Time

- An algorithm that runs in *O(n)*, or linear, time has a very natural property: its running time is at most a constant factor times the size of the input.
- One basic way to get an algorithm with this running time is to process the input in a single pass, spending a constant amount of time on each item of input encountered.

A Survey of Common Running Times

Linear Time

- *Ex: Computing the Maximum*
- Ex: Merging Two Sorted List

A Survey of Common Running Times

Linear Time

- Ex: Merging Two Sorted List

```
To merge sorted lists A = a_1, ..., a_n and B = b_1, ..., b_n:
  Maintain a Current pointer into each list, initialized to
    point to the front elements
  While both lists are nonempty:
    Let a_i and b_j be the elements pointed to by the Current pointer
    Append the smaller of these two to the output list
    Advance the Current pointer in the list from which the
      smaller element was selected
  EndWhile
  Once one list is empty, append the remainder of the other list
      to the output
```

A Survey of Common Running Times

Linear Time

Ex: Merging Two Sorted List

- The better way to argue is to bound the number of iterations of the While loop by an "accounting" scheme. Suppose we *charge* the cost of each iteration to the element that is selected and added to the output list.
- An element can be charged only once, since at the moment it is first charged, it is added to the output and never seen again by the algorithm. But there are only $2n$ elements total, and the cost of each iteration is accounted for by a charge to some element, so there can be at most $2n$ iterations.
- Each iteration involves a constant amount of work, so the total running time is $O(n)$.

A Survey of Common Running Times

O(n log n) time

- It is the running time of any algorithm that splits its input into two equal-sized pieces, solves each piece recursively, and then combines the two solutions in linear time.
- Sorting is perhaps the most well-known example of a problem that can be solved this way.
- Specifically, the *Mergesort* algorithm divides the set of input numbers into two equal-sized pieces, sorts each half recursively, and then merges the two sorted halves into a single sorted output list.

A Survey of Common Running Times

Quadratic time

- Suppose you are given *n* points in the plane, each specified by *(x, y)* coordinates, and you'd like to find the pair of points that are closest together.
- The natural brute-force algorithm for this problem would enumerate all pairs of points, compute the distance between each pair, and then choose the pair for which this distance is smallest.
- What is the running time of this algorithm?

A Survey of Common Running Times

Quadratic time

What is the running time of this algorithm? The number of pairs of points is $\binom{n}{2} = \frac{n(n-1)}{2}$, and since this quantity is bounded by $\frac{1}{2}n^2$, it is $O(n^2)$. More crudely, the number of pairs is $O(n^2)$ because we multiply the number of ways of choosing the first member of the pair (at most n) by the number of ways of choosing the second member of the pair (also at most n). The distance between points (x_i, y_i) and (x_j, y_j) can be computed by the formula $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ in constant time, so the overall running time is $O(n^2)$. This example illustrates a very common way in which a running time of $O(n^2)$ arises: performing a search over all pairs of input items and spending constant time per pair.

A Survey of Common Running Times

Quadratic time

- Quadratic time also arises naturally from a pair of *nested loops*: An algorithm consists of a loop with *O(n)* iterations, and each iteration of the loop launches an internal loop that takes *O(n)* time.
- Multiplying these two factors of *n* together gives the running time $O(n^2)$.

A Survey of Common Running Times

Quadratic time

- The brute-force algorithm for finding the closest pair of points can be written in an equivalent way with two nested loops:

A Survey of Common Running Times

Quadratic time

- The brute-force algorithm for finding the closest pair of points can be written in an equivalent way with two nested loops:

```
For each input point (x_i, y_i)
  For each other input point (x_j, y_j)
    Compute distance d = sqrt((x_i - x_j)^2 + (y_i - y_j)^2)
    If d is less than the current minimum, update minimum to d
  Endfor
Endfor
```

Note how the "inner" loop, over (x_j, y_j), has $O(n)$ iterations, each taking constant time; and the "outer" loop, over (x_i, y_i), has $O(n)$ iterations, each invoking the inner loop once.

A Survey of Common Running Times

Cubic time

- We are given sets $S_1, S_2, \ldots, S_n$, each of which is a subset of $\{1, 2, \ldots, n\}$
- we would like to know whether some pair of the sets is disjoint—in other words, has no elements in common.

A Survey of Common Running Times

Cubic time

- Let's suppose that each set S_i is represented in such a way that the elements of S_i can be listed in constant time per element, and we can also check in constant time whether a given number p belongs to S_i.

For pair of sets S_i and S_j

  Determine whether S_i and S_j have an element in common

Endfor

A Survey of Common Running Times

Cubic time

```
For each set S_i
  For each other set S_j
    For each element p of S_i
      Determine whether p also belongs to S_j
    Endfor
    If no element of S_i belongs to S_j then

          Report that S_i and S_j are disjoint
        Endif
      Endfor
    Endfor
```

A Survey of Common Running Times

Cubic time

- Each of the sets has maximum size *O(n)*, so the innermost loop takes time *O(n)*.
- Looping over the sets S_j involves *O(n)* iterations around this innermost loop; and looping over the sets S_i involves *O(n)* iterations around this.
- Multiplying these three factors of *n* together, we get the running time of $O(n^3)$.

A Survey of Common Running Times

$O(n^k)$ time

- In the same way that we obtained a running time of $O(n^2)$ by performing bruteforce search over all pairs formed from a set of n items, we obtain a running time of $O(n^k)$ for any constant k when we search over all subsets of size k.

A Survey of Common Running Times

$O(n^k)$ time

```
For each subset S of k nodes
  Check whether S constitutes an independent set
  If S is an independent set then
    Stop and declare success
  Endif
Endfor
If no k-node independent set was found then
  Declare failure
Endif
```

$$\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}.$$

A Survey of Common Running Times

Beyond Polynomial time

```
For each subset S of nodes
  Check whether S constitutes an independent set
  If S is a larger independent set than the largest seen so far then
    Record the size of S as the current maximum
  Endif
Endfor
```

A Survey of Common Running Times

Sublinear time

Given a sorted array A of n numbers, we'd like to determine whether a given number p belongs to the array. We could do this by reading the entire array, but we'd like to do it much more efficiently, taking advantage of the fact that the array is sorted, by carefully *probing* particular entries. In particular, we probe the middle entry of A and get its value—say it is q—and we compare q to p. If $q = p$, we're done. If $q > p$, then in order for p to belong to the array A, it must lie in the lower half of A; so we ignore the upper half of A from now on and recursively apply this search in the lower half. Finally, if $q < p$, then we apply the analogous reasoning and recursively search in the upper half of A.

A Survey of Common Running Times

Sublinear time

The point is that in each step, there's a region of A where p might possibly be; and we're shrinking the size of this region by a factor of two with every probe. So how large is the "active" region of A after k probes? It starts at size n, so after k probes it has size at most $(\frac{1}{2})^k n$.

Given this, how long will it take for the size of the active region to be reduced to a constant? We need k to be large enough so that $(\frac{1}{2})^k = O(1/n)$, and to do this we can choose $k = \log_2 n$. Thus, when $k = \log_2 n$, the size of the active region has been reduced to a constant, at which point the recursion bottoms out and we can search the remainder of the array directly in constant time.

A Survey of Common Running Times

Sublinear time

- So the running time of binary search is *O(*log *n)*, because of this successive shrinking of the search region.
- In general, *O(*log *n)* arises as a time bound whenever we're dealing with an algorithm that does a constant amount of work in order to throw away a constant *fraction* of the input.
- The crucial fact is that *O(*log *n)* such iterations suffice to shrink the input down to constant size, at which point the problem can generally be solved directly.