11. Complexity

11.1 Introduction

This chapter focuses on issues of program running time, including how to measure and analyze programs for their running time, as well as provide examples of techniques for improving program performance. Examples are taken from the areas of sorting and searching.

The pragmatic aspects of computing require one to be cognizant of the resource-usage aspects of a program. While such concerns should be secondary to those of the correctness of the program, they are nonetheless concerns that, like correctness, can make the difference between success and failure in computer problem solving. The general term used by computer scientists to refer to resource usage is "complexity". This term refers not to how complex the *program* is, i.e. how difficult it is to understand, but rather how much *resources* are consumed when the program is *executed.* Indeed, the least difficult to understand program might be fairly profligate in its use of resources.

The process of making a program more "efficient" unfortunately often has the effect of making it harder to understand. To develop a program *to a first approximation,* the following axiom might be applied.

Get it right first, *then* make it faster.

In particular, this axiom should be applied when considering small incremental improvements in code, which can shave off some fraction of execution time, but which make the program obscure and more difficult to debug.

The greater thrust of this chapter, however, is algorithmic improvements, that is make a program faster by choice or development of a better algorithm. Coding a new algorithm can be like starting afresh with respect to "getting it right" however. For this reason, it is best to have designed the overall program as a set of modules, with "plug replaceability" between a simple but slower module and a faster one.

11.2 Resources

By "resource", we typically are concerned with one or more of the following:

> **Execution time**: This is the time it takes a program to process a given input. Time is considered a resource for at least two reasons:

The time spent waiting for a solution (by a human, or by some other facet of automation) is time that could be put to other productive uses. In this sense, time is not the actual resource, but is instead reflective of resources that might go unused while waiting.

Thinking of the computer as providing a service, there is a limitation on the amount of service that can be provided in a given time interval. Thus programs that execute longer use up more of this service.

Memory space: This is the space used by a program to process a given input. Memory space used translates into cost of computation in the sense that memory costs money and the ability to use a certain amount of memory directly depends on the memory available.

Memory space could be further sub-divided along the lines of a *memory hierarchy*, some form of which is found in most computer systems:

Main memory: Semiconductor memory in which most of the program and data are stored when the program is running.

Cache memory: Very high-speed semiconductor memory that "caches" frequently-used program and data from main memory.

Paging memory: Slower memory, usually disk, which in simplistic terms serves as kind of "overflow" or "swapping" area for the main memory.

File memory: Disk or tape memory for file objects used by a program .

In these notes, our primary focus will be on execution time as the resource Some consideration will be given to memory requirements as well. As we shall see, it is often possible to "trade off" time resources for memory resources and vice-versa.

11.3 The Step-Counting Principle

Most often we will be interested in relative execution-time comparisons between two or more algorithms for solving a given problem. Rather than dealing with the actual times a computer would spend on an algorithm, we try to use a measure that is relatively insensitive to the particular computer being used. While the speeds of various primitive operations, such as addition, branching (change of control from one point in the program to another), etc. may vary widely, we make the assumption that for purposes of comparing algorithms on a given computer, we can just count the **number** of each kind of operation during execution, rather than be concerned with the actual times of those operations. This is not to say that every occurrence of a given kind of operation takes the same time; there will be a general dependency on the values of the arguments as well. However, for purposes of getting started, we make the assumption that the count is an adequate measure. We could then get an overall time estimate by multiplying the counts of various operations by the time taken by those operations and summing over all n different kinds of operations:

$$\text{Execution time} = \sum_{i=1}^{n} \text{count(Operation i)*time(Operation i)}$$

Straight-line Programs

Straight-line programs are programs with no branching: every operation in the program is executed. Thus the execution time is the same regardless of data. For example, consider the straight-line program:

```
a = b*c + d;
c = d/e + f;
f = a*c;
```

Here there are two multiply operations, one divide, and two additions. Thus the total time would be computed as

execution time =

```
  2*time(multiply)
+ 1*time(divide)
+ 2*time(add)
+ 3*time(assign)
```

where time(assign) refers to the time to assign a value to a variable explicitly.

Loop Programs

Very few programs of substance will be straight-line. Typically we have loops, the execution of which will depend on the data itself. In this case, the total time depends on the data. Consider

```
sum = 0;
for( i = 0; i < N; i++ )
  sum = sum +  i*i;
```

Here the number of times the loop body is executed will be N. Therefore, there will be N multiply operations. There will also be N additions in the loop body, as well as N additions of the form i++, and N comparisons i < N. We have as total execution time:

execution time =

```
   2*time(assign) +
   N*[time(multiply) + time(add) + time(compare) + time(increment) + time(assign)]
```

Recursive Programs

As we know, loop programs can be represented as recursive programs. However, recursive programs also have the possibility of "non-linear" recursion, making them sometimes more of a challenge to analyze. Consider the computation of the rex program `sum(1, N)` where `N >= 0`.

```
sum(M, N) => M >= N ? M;
sum(M, N) => K = (M+N)/2, sum(M, K) + sum(K+1, N);
```

`sum(M, N)` computes the sum of integers `M` through `N` by dividing the range into two, until the range is empty. The value of sum is obtained by summing the results of recursive calls to the function.

The following tree shows how the recursion decomposes for `sum(1, 6)`:

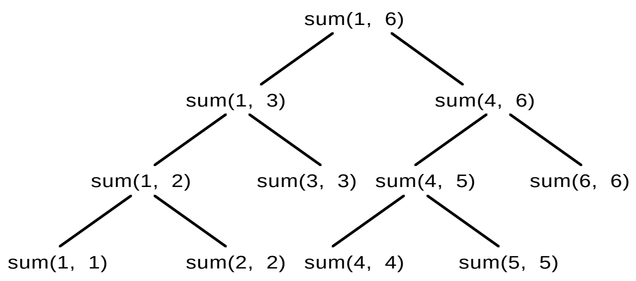


Figure 170: Tree of a recursive decomposition

The tree representation for the program's execution of `sum(1, N)` will have `N` leaves and therefore `N-1` interior nodes, i.e. `2*N-1` nodes altogether. For each node there will be a comparison to determine which rule to use, for a total of `2*N-1` comparisons. For each interior node, there will be 3 additions, and one division by 2. So overall we have

execution time =
(N-1)*[3*time(add) + time(divide)]
+ (2*N-1)*time(compare)

Here we are ignoring any overhead required to do function calls, and are assuming that the times to do the basic operations are constant, i.e. independent of the argument sizes. This is only an approximation to reality, especially if we have to deal with arbitrarily-large arguments. If time(add) = time(divide) = time(compare) = 1, then the total time is

4*(N - 1) + 2*N - 1

= 6*N - 5

The analysis above assumes that we already understand the algorithm well enough to see that a tree is involved, and that we know how to analyze a tree. An alternative approach that doesn't make such assumptions is to derive a recurrence formula for time patterned after the rules, but with the data size as an argument. In this case, the "size" is the range of numbers to be summed. For the basis case, there is only a comparison, so we have:

T(1) => time(compare);

For the induction rule, we make the simplifying assumption that the range is of even length, so we can divide it in half:

T(2*N) =>
    time(compare) + 3*(time add) + 1*time(divide) + 2*T(N);

Again assuming that all operations take the same time, we get

```
T(1) => 1;

T(2*N) => 5 + 2*T(N);
```

For example, to sum 8 numbers,

```
T(8)  ==> 5 + 2*T(4)
      ==> 5 + 2*(5 + 2*T(2))
      ==> 5 + 2*(5 + 2*(5 +2*T(1)))
      ==> 5 + 2*(5 + 2*(5 +2*1))
      ==> 43
```

which agrees with our earlier calculation of 6*N-5 when N = 8. We can also see that there is agreement for general N that is repeatedly divisible by 2. Such a number must be a power of 2, $N = 2^k$. Let $S(k) = T(2^k)$. Then we have the equivalent recurrence

```
S(0) ==> 1;

S(k+1) ==> 5 + 2*S(k);
```

We can "solve" this recurrence by successive substitutions:

```
S(k)  ==> 5 + 2*S(k-1)
      ==> 5 + 2*(5 + 2*S(k-2))
      ==> 5 + 2*(5 + 2*(5 + 2*S(k-3)))
      ==> ...
```

until the argument to S is reduced to 0. This will obviously occur after k substitutions, so

$$\begin{aligned}
S(k) &= 5 + 2*S(k-1) \\
&= 5 + 2*(5 + 2*S(k-2)) \\
&= 5 + 2*5 + 2^2*S(k-2) \\
&= 5 + 2*5 + 2^2*5 + 2^3*S(k-3) \\
&= 5 + 2*5 + 2^2*5 + 2^3*5 + 2^4*S(k-4) \\
&\quad \ldots. \\
&= 5*(1 + 2 + 2^2 + 2^3 + \ldots. + 2^{k-1}) + 2^k * S(0) \\
&= 5*(2^k - 1) + 2^k \\
&= 6*2^k - 5
\end{aligned}$$

11.4 Profiling

Many systems provide a software tool known as a "profiler". Such a tool counts executions of procedures and the places from which they are called. Using it, one can get an idea of how much time overall is being spent in various procedures, and thus possibilities for where to devote attention in improving run time.

A specific example, using the Java interpreter with -prof option will put profile results from the run in a file `java.prof`.

Let's suppose that we have the following break down of the time devoted to various pieces of code A, B, C:

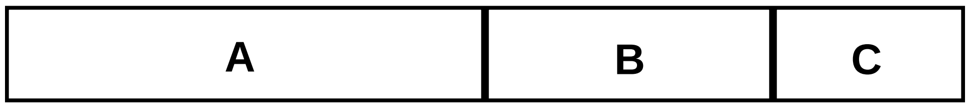


Figure 171: Execution time profile

The suggestion is that A takes about 50% of the time, B 30%, and C 20%. The question is where to concentrate code improvements to reduce the execution time? Intuitively we should concentrate on A, because there we stand to achieve the biggest reduction. But how much improvement can we get from A alone? In the very best case, we could eliminate A altogether. This would result in a 50% reduction in execution time. On the other hand, if we eliminated C altogether, we would still have 80% of the time we did before, or only a 20% reduction.

Such considerations are quantified in a rule known as Amdahl’s law. In general, if chunk *A* takes fraction *f* of the overall time, then the speedup achieved by reducing *A* to 0 is at most 1/(1-*f*). So, execution of *A* would have to occupy about 90% of the execution to enable a 10-fold reduction in execution time if *A* were eliminated completely. Amdahl's law was originally derived for application to parallel computing and we'll see more about it in the chapter *Limitations of Computing*.

11.5 Suppressing Multiplicative Constants

Quite often we make the further simplifying assumption that the operation times are the same for all operations. While this may seem like a drastic oversimplification, it is useful for comparative purposes. If every operation requires a constant time, then each time can be expressed as some factor times one of the operations. Thus, in assuming all operations have the same time, the resulting time estimate will be off by a factor that is at most the maximum of these factors. For reasons to be explained, it is defensible to make the assumption that all times are the same, so long as it is clear that this assumption is being made. With this assumption, we would have the following for the above examples:

straight-line example:	execution time = 8 steps
loop example:	execution time = 5*N + 2 steps
recursive example:	execution time = 6*N - 5 steps

11.6 Counting Dominant Operations

In many cases, we can get an idea of the execution time by simply focusing on the number of dominant operations. For example, in the loop program, we could focus on the number of multiplies or the number of times the loop body is executed. In both cases, we would end up with an execution time of N steps. In the recursive program, we could count the number of times the recursive rule is used, which would give us N-1 steps.

11.7 Growth-Rate

Although we may, on occasion, engage in estimating time for a *specific* input to a program, in general we will be interested in a much broader measure to give an idea of the quality of the program or its algorithm. Such a measure is found in the form of *growth-rate comparisons*, as we now discuss.

Most programs are designed to work with not just a single input, but rather with a wide, and usually infinite, set of input possibilities. We often can associate a measure of the input, usually in the form of a parameter that implies the *size* of the input. Some examples are:

Program application	**Possible measure(s) of input**
word processing	number of characters in the document, or number of editing commands
solving linear equations	number of equations, and/or number of unknowns
sorting an array	number of elements in the array
displaying a scene graphically	number of polygons in the scene

With each type of program, we try to focus on one key measure in which to express the program's performance. We try to express the programs resource usage, e.g. execution time, as a function of this measure. For example, in the loop program above, we could use the value N as the measure. We derived that

execution time = 5*N + 2 steps

With slight modification, we can convert that program into one that sums the squares of an array of N elements:

```
sum = 0;
for( i = 0; i < N; i++ )
      sum = sum + a[i]*a[i];
```

Now the input measure is equated to the *size* of the array.

Now consider sorting an array, using the following minimum-selection sort algorithm expressed in Java. (Here calling the constructor on an array of doubles sorts the array in place; the object created can then be discarded).

```
class minsort
{
private double array[];       // The array being sorted
int N;                        // The length of the prefix to be sorted

  //  Calling minsort constructor on array of doubles sorts the array.
  //  Parameter N is the number of elements to be sorted (which might
  //  be fewer than are in the array itself).

  minsort(double array[], int N)
    {
    this.array = array;
    this.N = N;

    for( int i = 0; i < N; i++ )
      {
      swap(i, findMin(i));
      }
    }


   //  findMin(M) finds the index of the minimum among
   //  array[M], array[M+1], ...., array[N-1].

  int findMin(int sortFrom)
    {
    // by default, the element at minSoFar is the minimum
    int minSoFar = sortFrom;

    for( int j = sortFrom+1; j < N; j++ )
      {
      if( array[j] < array[minSoFar] )
        {
        minSoFar = j;   // a smaller value is found
        }
      }
    return minSoFar;
    }


   //  swap(i, j) interchanges the values in array[i] and array[j]

  void swap(int i, int j)
    {
    double temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    }
```

If we count comparisons, as in `array[j] < array[minSoFar]`, as the dominant operation, then we could derive

execution time = n*(n-1)/2 steps

To see this, let us extract the loop structure essence of the program:

```
   for( i = 0; i < n; i++ )
     {
     for( j = i + 1; j < n ; j++ )
       *** one step ***
     }
```

Here one step represents the comparison operation that we are counting. Now examine the number of times the inner loop body executes as a function of the outer loop index:

i = 0	j = 1, 2, ..., n - 1	n - 1 steps
i = 1	j = 2, 3, ..., n - 1	n - 2 steps
...		
i = n-1	j = n, ..., n - 1	0 steps

In total, we have 0 + 1 + 2 + ... + (n-1) steps, which sums to n*(n-1)/2. This summation can be shown by induction. This is a special case of the sum of an *arithmetic series*.

In terms of the topic of this section, we would say that the sorting program's growth-rate is represented by the function

$n \to n*(n-1)/2$

that is, the function that, with argument *n*, yields the value of n*(n –1)/2. It is important to keep in mind that the growth rate is a *function*, even though it is often written just as an *expression*

n*(n –1)/2

with the argument *n* being implicit for simplicity.

Not all programs run the same amount of time for a given input measure. For those that do not, it is common to use the *maximum* over all inputs having a given value of the measure as the growth rate function. For example, suppose we had a program that inputs strings of 0's and 1's, with the following observed execution times:

Input	Time
λ	0
0	1
1	1
00	1
01	4
10	4
11	2
000	1
001	9
010	9
011	9
100	8
101	6
110	4
111	9
...	

If we use the *length* of the input as the measure, then a growth-rate of $n \rightarrow n^2$ is suggested, even though not all inputs of length n require n^2 time. Thus, we are often content with focusing on the **worst-case** among inputs of a given value of the input measure, rather than considering all inputs, in order to get an idea of the complexity. Another way of looking at it is that the derived function forms an **envelope** around the actual executions times, or is an **upper bound** on the execution time of the algorithm. The figure below demonstrates this for the example at hand.

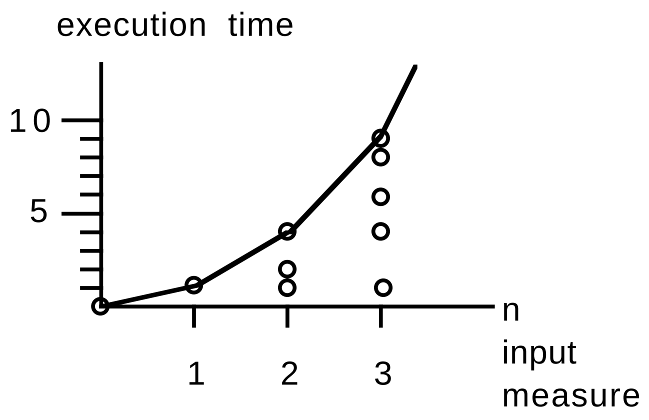


Figure 172: Execution times for a string-processing program, plotted vs. input string length. The quadratic curve is an upper-bound on the times.

11.8 Upper Bounds

In general, a function can be an **upper bound** on execution time without having all of its points correspond to actual execution times. For example, if the above hypothetical algorithm used at most 12 steps to process any input of length 4, then the upper bound of $n \rightarrow n^2$ would still be consistent. Likewise, the function $n \rightarrow n^3$ would also be an upper bound, although qualitatively a poorer one with respect to the given data.

Informally, when an upper bound fits the data points closely, we say it is a **tight** upper bound. Obviously, the tighter an upper bound is, the more information is conveyed by the statement that it is an upper bound. That is, saying that $n \rightarrow n^2$ is an upper bound conveys more information than saying that $n \rightarrow n^3$ is.

For convenience, it is common to omit the argument part of functional expressions when talking about growth rates. Thus n^3 would actually stand for the function $n \rightarrow n^3$. We will be taking this approach from here on in the discussion, except at points where it is useful to make it clear that we are talking about a function rather than just an expression.

11.9 Asymptotic Growth-Rate

A coarse, but useful, measure for comparing algorithms is based on asymptotic growth-rate. This measure has the benefit of being relatively easy to derive, since it is impervious to the making of many approximations in the derivation process. Asymptotic growth rate is a measure of goodness of the time taken by an algorithm as the value of the input measure *n* grows without bound. In computing asymptotic growth rate, we often ignore multiplicative constants in the complexity function and focus on the "rate" itself. Thus, while an execution time measure of n^2 (i.e. the function $n \to n^2$) is obviously better than one of n^3, the asymptotic comparison would also rank $1000n^2$ (i.e. the function $n \to 1000n^2$) as being better than n^3, even though the latter is better (i.e. lower) for values of $n < 1000$, called the *crossover point*. The reason to prefer $1000n^2$ is that n^3 is only better than it for a *finite* number of values of n (assuming the input measure is an integer). For the remaining infinite number of inputs, $1000n^2$ is better. Of course, this sort of reasoning is most meaningful when the crossover point is within the range of values of n to which the algorithm is actually going to be applied.

We can simplify the task of asymptotic comparisons by observing that, in a function the value of which is a sum of terms, the sum is often **asymptotically dominated** by one of those terms. Consider for example, the function

$$n \to 1000n^2 + n^3$$

For large n, the second term dominates, in the sense that the first term becomes relatively insignificant the larger n becomes. Thus, for purposes of comparing this function to another, we can simply neglect the term $1000n^2$ in the limit. The first function, now approximated by

$$n \to n^3$$

is clearly seen to grow faster than the second function.

11.10 The "O" Notation

For purposes of comparing asymptotic growth rates, the "O" (for "order") notation has been invented[†]. In considering a function such as

$$n \to 1000n^2 + n^3$$

it is natural to indicate that the growth rate of that function is "on the order of" the growth-rate of the function $n \to n^3$, or for short, the function is "order of" n^3. A simple way of accomplishing this is to define a set of functions, the growth rate of each of which is no more than a certain metric times an arbitrary constant. For example,

$$O(n^3)$$

means the set of functions growing no faster than does the function $n \to cn^3$, where c is an arbitrary constant.

[†] The "O" notation is due to P.G.H. Bachmann, *Zahlentheorie*, vol. 2: *Die analytische Zahlentheorie*, B.G. Teubner, Leipzig, 1894.

If f and g are two functions,

$$f \in O(g)$$

means that f is bounded from above by g times a constant.

It is also common to see in the literature

$$f = O(g)$$

which is a slight abuse of notation, but one having the same meaning as $f \in O(g)$. It is also common to use **expressions in place of functions**. Thus, one often sees something like

$$n^2 \in O(n^3)$$

when what is really meant is the following relationship:

$$(n \rightarrow n^2) \in O(n \rightarrow n^3)$$

Examples

We have already seen that $n^2 \in O(n^3)$. We also mentioned that $1000n^2 \in O(n^3)$. As will be seen, $cn^r \in O(n^s)$ whenever $r < s$, for any constant c. The rationale for all of these can be seen by looking at the slopes of the curves as n increases without limit. Even if c is very large, cn^r will eventually be overtaken by n^s for large enough n if $r < s$. The following diagram show the case where $f \in O(g)$ even though for low values of n, *f*'s value is a significant multiple of *g*'s value.

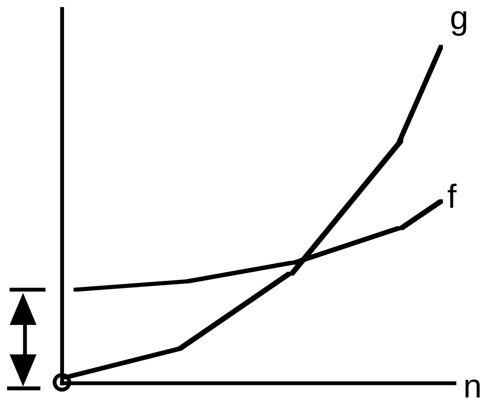


Figure 173: f ∈ O(g), assuming the indicated trends in f and g continue

We can use *g*'s algorithm for small values of n and *f*'s algorithm for large values to get the best of both worlds. We would choose between the two algorithms depending on whether $n < n_0$ where n_0 is the breakpoint, and the resulting execution time would then appear as in the following diagram.

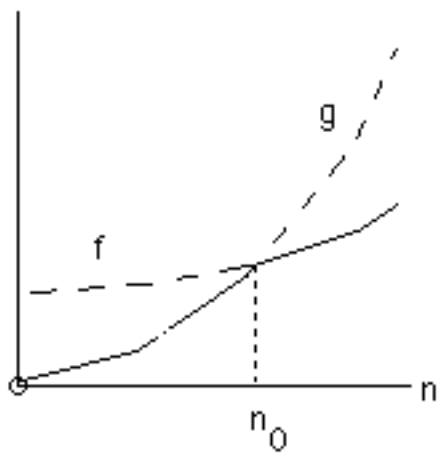


Figure 174: Combining two algorithms to get a superior execution time function

Simplified Definition of "O"

We give a definition for the **special case of functions with natural number arguments**, which allows most resource measures to be modeled, due to the fact that we almost always base the measure on the size of some input facet and the size is in integral units. Later (in the exercises) we give the more general definition, and indicate that the two definitions are consistent on the domain of natural numbers.

Let *f*: N $\rightarrow$ R and *g*: N $\rightarrow$ R be two functions with domain being the natural numbers and range being the positive real numbers. Then

$f \in O(g)$

[typically read "f is oh of g" or "f is big-oh of g"]

means

$(\exists c)(\forall n)\ f(n) \leq cg(n)$

This says: "there exists a constant c such that for all n, f(n) is less than or equal to c times g(n).

For the case of R as a domain, we would need to use a more complex definition for $f \in O(g)$:

$$(\exists\ c)(\exists\ n_0)(\forall n > n_0)\ \ f(n) \le cg(n)$$

For the case of N as a domain, the two definitions are equivalent.

If we are given *f* and *g*, then in order to show that $f \in O(g)$, we need only to exhibit an appropriate c. We show this in the following examples.

Examples

$n^2 \in O(n^3)$	Take c = 1. Obviously $(\forall\ n)\ n^2 \le 1n^3$.
$1000n^2 \in O(n^3)$	Take c = 1000. Obviously $(\forall\ n)\ 1000n^2 \le 1000n^3$.
$n^2 + 10^6 n \in O(n^2)$	Take c $=2*10^6$. We have

$$\begin{aligned}(\forall\ n)\ n^2 + 10^6 n &\le 10^6 n^2 + 10^6 n\\ &\le 10^6 n^2 + 10^6 n^2 = 2*10^6 n^2\end{aligned}$$

since $(\forall\ n)\ n \le n^2$.

11.11 O Notation Rules

Fortunately, it is not necessary to return to first principles for every comparison of two functions. We can establish some rules that help us reason about relationships between asymptotic growth rates:

Transitivity Rule

If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$.

Proof: Suppose that $f \in O(g)$ and $g \in O(h)$. Then by definition, we know that for some constants *c* and *d*:

$$(\forall\ n)\ \ f(n) \le cg(n)$$

and

$$(\forall\ n)\ \ g(n) \le dh(n)$$

i.e. the existence of *c* and *d* is guaranteed by our supposition and the definition of O. We must then show

$$(\exists\ e)(\forall\ n)\ \ f(n) \le eh(n)$$

We can do this by exhibiting a constant *e* that makes this statement true. It appears that choosing *e* to be the product of *c* and *d* will do the job: We need to show that, for arbitrary *n*,

$$f(n) \le cdh(n)$$

But we already have

$$f(n) \le cg(n)$$

and

$$g(n) \le dh(n)$$

Putting these two inequalities together gives exactly what we need.

Sum Rule

$$\boxed{\text{If } f \in O(h) \text{ and } g \in O(k), \text{ then } f+g \in O(\max(h, k)).}$$

Here we use f + g as an abbreviation for the *function* $n \rightarrow f(n) + g(n)$ and max(h, k) as an abbreviation for the function $n \rightarrow \max(h(n), k(n))$.

Proof: For convenience, define *m* to be the function $n \rightarrow \max(h(n), k(n))$. That is, for all n, $m(n) = \max(h(n), k(n))$. Assume that $f \in O(h)$ and $g \in O(k)$, to show that $(n \rightarrow f(n) + g(n)) \in O(m)$. Let c and d be constants such that

$$(\forall n)\ \ f(n) \le ch(n)$$

and

$$(\forall n)\ \ g(n) \le dk(n)$$

Then we have

$$(\forall n)\ \ f(n) + g(n) \le \max(ch(n), dk(n))$$

Thus

$$(\forall n)\ \ f(n) + g(n) \le \max(c, d)\ m(n)$$

Therefore we have found a constant, namely max(c, d), which establishes what we want to show.

The sum rule is frequently applied in program analysis. If a program consists of two parts in sequence, P; Q, with the complexity of each in terms of the input measure being represented by functions *f* and *g*, respectively, and *h* and *k* are known upper bounds on *f* and *g*, then the function $n \to \max(h(n), k(n))$ is an upper bound on the overall program complexity. Put another way, the complexity of the combination P; Q is dominated by the part with the greater complexity. Quite often, the same part dominates for all values of the input measure.

Polynomial Rule

By applying the sum rule inductively, we can get the following:

> Let f(n) be any polynomial in n, with k being the highest exponent.
> Then $f \in O(n^k)$.

Caution: We need to be aware that polynomials have a *fixed* number of terms, i.e. the set of terms cannot be a function of n, as in the following:

Example of a Fallacious Argument: $n^3 \in O(n^2)$

Bogus Proof: $n^3 = n^2 + n^2 + \ldots + n^2$ where the sum is taken over n terms. By the polynomial rule, since the highest exponent is 2, we have $n^3 \in O(n^2)$.

Constant Absorption Rule

It is never necessary to write $f(n) \in O(dg(n))$ where *d* is a constant. It is always considered preferable to write this as $f(n) \in O(g(n))$. The argument here is that the constant d can be "absorbed into" the constant that exists by definition of $f(n) \in O(g(n))$.

Proof: Assume that $f(n) \in O(dg(n))$ where *d* is a constant, to show $f(n) \in O(g(n))$. By supposition, there is a *c* such that

$$(\forall n)\ \ f(n) \leq cdg(n)$$

But letting e = cd, *e* is also a constant, so

$$(\exists e)(\forall n)\ \ f(n) \leq eg(n)$$

Therefore $f(n) \in O(g(n))$.

Meaning of O(1)

When we say that $f \in O(1)$, we mean that f is O of the function $n \rightarrow 1$, i.e. the constant function 1. By the constant absorption rule, any function bounded above by a constant is O(1). So **saying $f \in O(1)$ is just another way of saying that f is bounded by a constant.** To say that $f \in O(c)$ where c is some other constant is the same as saying $f \in (1)$, so we *always* write the latter.

For example, the **linear addressing principle** says that any element within an array can be accessed, given its index, in time O(1). This is an important advantage of arrays over linked lists, for which the access can only be bounded by O(n) where n is the number of elements in the list.

Multiplication Rule

The proper form of argument when the number of terms is a function of n is given by the following:

$$\boxed{\text{If } f \in O(g)\text{, then } m(n)*f(n) \in O(m(n)*g(n)).}$$

In terms of programs, if g provides an upper bound on the execution of the body of a loop, where n is the value of the input measure, and the loop executes m(n) times, then the function $n \rightarrow m(n)*g(n)$ provides an upper bound for the overall program.

Example – The following program has O(ng(n)) as an upper bound, assuming that the loop body does not change i.

```
for( i = 0; i < n; i++)
        .... some O(g(n)) computation ....
```

Here m(n) = n.

11.12 Analyzing Growth of Exotic Functions

The rules above give us ability to analyze some basic functions, but it does not help us handle cases that will be encountered frequently, such as ones involving *log* n [all *log*s will be assumed base 2 unless otherwise noted. However, since *log*s of different bases differ by constant factors, it would not be worthwhile differentiating them in the O notation anyway, due to the constant absorption rule.]

As an overview, it is worth establishing a framework of a few functions in a "hierarchy" of functions that often come up in algorithms:

$$1 < \log n < .... < n^{1/4} < n^{1/3} < n^{1/2} < n < n \log n < n^2 < n^3 < n^4 < .... < 2^n < n\,!$$

Each "function" in this chain is O of the next function, but not conversely. We will establish some of these relations in this section. First we show a few comparative plots to remind ourselves of how some of these functions behave.

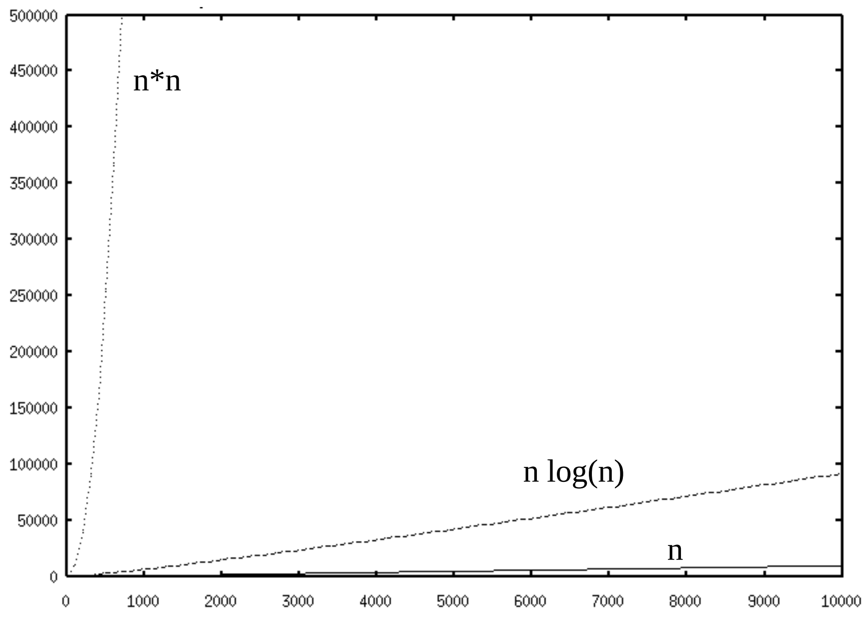


Figure 175: n^2 *vs*. n log n *vs*. n

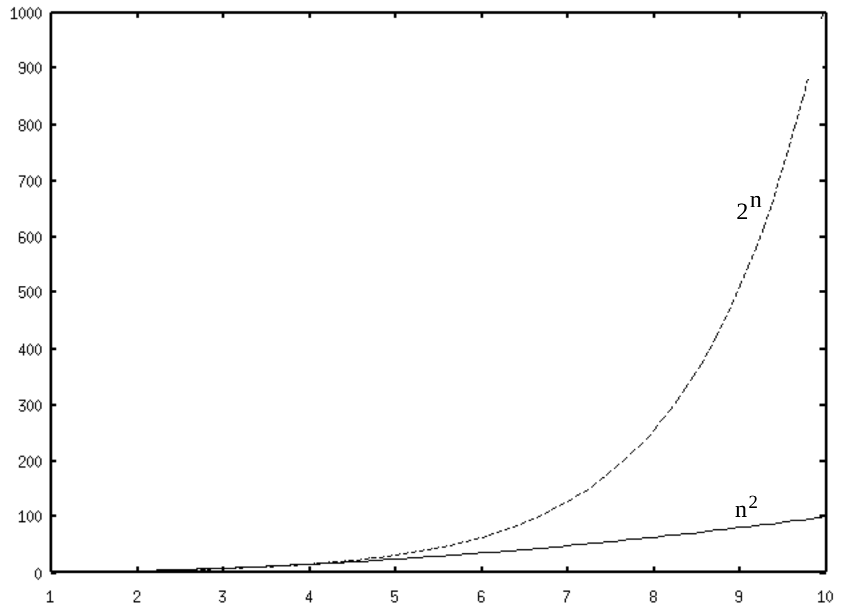


Figure 176: 2^n *vs*. n^2

A convenient way to approach analysis of some functions is through the *derivative*. Suppose we are trying to establish $f \in O(g)$. Even though we are working with functions on a natural number domain, suppose that each function has an analytic counterpart *F* and *G* on the real domain. If *G* maintains a greater derivative than *F* for sufficiently large n, then at some point the slope of the curve for *F* will stay less than the slope of the curve for *G*. By extrapolating from this point, we can see that *G* will ultimately overtake *F*. This is illustrated in the following diagram.

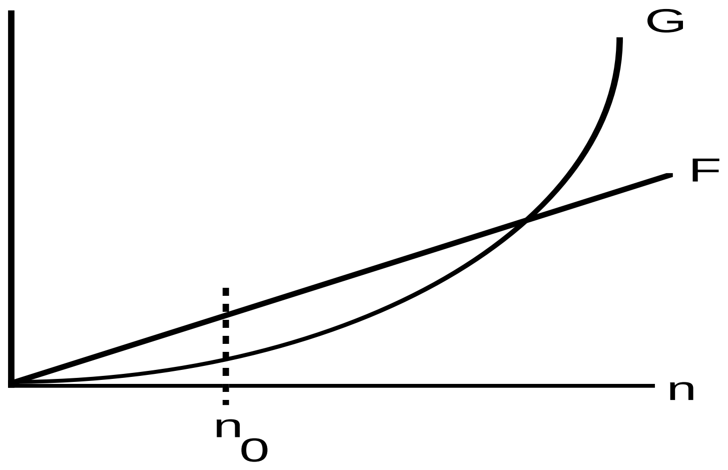


Figure 177: Showing that f ∈ O(g) through knowledge of the derivatives of corresponding analytic functions. At point n_0 the derivative of G becomes greater than that of F.

11.13 Derivative Rule

A sufficient condition for $f \in O(g)$, where *f* and *g* are restrictions of analytic functions *F* and *G* to the natural number domain, is that

$$(\exists\, n_0)(\forall\, n > n_0)\ \ F'(n) \le G'(n)$$

where F' and G' denote the first derivatives of F and G respectively.

Caution: The condition above is only sufficient for $f \in O(g)$. It is not necessary. For example, one can easily construct examples using functions where each function is O of the other, yet there are no corresponding analytic functions.

Example: log n ∈ O(n)

Here we are comparing two functions: *log*, and the identity function $n \rightarrow n$. Let us call the analytic counterparts *F* and *G*. Then from calculus the derivatives have the property that

$F'(n) = c / n$ where c is an appropriate constant[†]

$G'(n) = 1$

Thus if we choose n_0 to be the next integer above c, we have the conditions set forth in the derivative rule: $(\forall\ n > n_0)\ c / n \leq 1$.

Example: $\log n \in O(n^{1/2})$

$n^{1/2}$ is, of course, another way of writing the square root of n. From calculus, the derivative of this function is $1/(2n^{1/2})$. This derivative will overtake the derivative of log n, which is c / n. Equality occurs at the point where

$$c / n = 1/(2n^{1/2})$$

i.e. $n = \text{ceiling}(4c^2)$.

11.14 Order-of-Magnitude Comparisons

Below is a table of some common functions and their approximate values as n ranges over 6 orders of magnitude.

log n	3.3219	6.6438	9.9658	13.287	16.609	19.931
$\log^2 n$	10.361	44.140	99.317	176.54	275.85	397.24
sqrt n	3.162	10	31.622	100	316.22	1000
n	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
n log n	33.219	664.38	9965.8	132877	$1.66*10^6$	$1.99*10^7$
$n^{1.5}$	31.6	10^3	$31.6*10^4$	10^6	$31.6*10^7$	10^9
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	1000	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	1024	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}
n!	3 628 800	$9.3*10^{157}$	10^{2567}	10^{35659}	10^{456573}	$10^{5565710}$

Values of various functions vs. values of argument n.

Such tables can give hints to the growth rate of functions, although are by no means to be considered a proof. For such things we should rely on analytic methods. In any case, such tables are instructive. For example, the table above shows that we can run a problem with a factor of 10^6 larger in its input measure using an O(log n) algorithm in only 20 times

[†] Recall that for any two bases, a and b, $\log_b x = \log_b a\ \log_a x$.

longer to execute. For an O(n log n) algorithm, we would require only $20*10^6$ times longer, as compared to 10^{12} times longer for an $O(n^2)$ algorithm, a factor of $5*10^4$.

An "inverted" version of the table can be used determine the relative sizes of problem that can be run in a *fixed* time using algorithms of various orders.

Time Multiple	**10**	**100**	**1000**	**10000**	**100000**	**1000000**
log n	1024	10^{30}	10^{300}	10^{3000}	10^{30000}	10^{30000}
$\log^2 n$	8	1024	$3*10^9$	$1.2*10^{30}$	$1.5*10^{95}$	$1.1*10^{301}$
sqrt n	100	10^4	10^6	10^8	10^{10}	10^{12}
n	10	10^2	10^3	10^4	10^5	10^6
n log n	4.5	22	140	1000	$7.7*10^3$	$6.2*10^4$
$n^{1.5}$	4	21	100	210	2100	10000
n^2	3	10	32	100	320	1000
n^3	2	4	10	21	46	100
2^n	3	6	9	13	16	19
n!	3	4	6	7	8	9

Increase in size of problem that can be run based on increase in allowed time, assuming algorithm runs problem size 1 in time 1.

This table tells us, for example, that if we have 1000 times more time, if our algorithm is O(n!) we can only run a problem 6 times as large. On the other hand, if we have an n log n algorithm, we could run a problem 140 times as large in the same time.

11.15 Doubling Comparisons

Perhaps a handier way to remember how various functions grow is to consider what happens if we double the input size. If the function is O(n), then doubling the input size will at most double the execution time. If the function is log(n), then doubling the input size will only add a constant to the execution time, and so on. We can summarize these sorts of observations in the following table, where k is a constant.

Complexity	**Doubling the input causes execution time to**
O(1)	stay the same
O(log n)	increase by an additive constant
$O(n^{1/2})$	increase by a factor of sqrt(2)
O(n)	double
O(n log n)	double, plus increase by a constant factor times n
$O(n^2)$	increase by a factor of 4
$O(n^k)$	increase by a factor of 2^k
$O(k^n)$	square

This table can be used to help intuition in algorithm design. For example, if the algorithm is observed to double in time plus add a constant factor times the input, then we can infer that the algorithm is O(n log n). An example of this kind of behavior is quicksort, under ideal circumstances.

11.16 Estimating Complexity Empirically

Given access to the code of a program, the complexity can often be determined by analysis. However, it may be desirable to check our analysis empirically, or we might wish to estimate the complexity of a program the code of which we do not have. A way to proceed experimentally is to run the program on inputs of a variety of values of the input measure and record the time in each case. This will give us a set of size-time pairs (S_i, T_i). A good choice would be to have input sizes differ by successive powers of two. Of course we do not know that our chosen inputs achieve the maximum time among inputs of that size; we are just assuming that all will be about the same, an assumption that is not always valid. If this assumption is in question, we can run the program with multiple inputs of each size.

The size-time pairs derived above constitute an approximation to the time complexity function T of the program. Now we form a hypothesis that T is O(*f*) for some function *f*. For example, if we were analyzing a sorting program, we might hypothesize that T is $O(n^2)$. If our hypothesis is correct, then there must be a constant c such that

$$\text{(for all S)} \qquad T(S) \leq cf(S).$$

We can compute the ratios T(S)/f(S). If there is a noticeable upper bound on this ratio, then that would make a possible value of c. If the ratios seem to hover around c, especially as S gets larger, then there is a good chance that our hypothesis is correct. On the other hand, if the ratios tend to decrease with S, our hypothesis is probably correct, but it might have been too conservative. That is, there might be a tighter bound, such as *n log n* in the case of the sorting program. The third possibility is that there is no bound evident. In this case, our hypothesis is probably incorrect and we should try again with a new hypothesis of a faster growth rate.

Empirical Comparison of Sorting Procedures

The various sorts described were tested on our local computer with varying sizes of arrays, doubling the sizes from 16 through 4096. The algorithms themselves will be described in a later section. As we timed each run, we also computed the time divided by n, n^2, and n log n, in an effort to ascertain the asymptotic performance and compare it to our analytic results. The results are shown below. The reader should examine each table and conclude:

(a) which bound best describes the performance of that particular algorithm

(b) an appropriate multiplicative constant for each bound

(c) an estimate of the time each algorithm would take for one million elements

The reader should also try to get a sense of how a good asymptotic bound does not necessarily indicate the best algorithm for small data sizes.

```
minsort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00085938   0.00005371   0.00000336   0.00001937
      32   0.00304688   0.00009521   0.00000298   0.00002747
      64   0.01109375   0.00017334   0.00000271   0.00004168
     128   0.03687500   0.00028809   0.00000225   0.00005937
     256   0.14187500   0.00055420   0.00000216   0.00009994
     512   0.56750000   0.00110840   0.00000216   0.00017768
    1024   2.24750000   0.00219482   0.00000214   0.00031665
    2048   8.98000000   0.00438477   0.00000214   0.00057508
    4096  35.89000000   0.00876221   0.00000214   0.00105343

insertSort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00058594   0.00003662   0.00000229   0.00001321
      32   0.00210938   0.00006592   0.00000206   0.00001902
      64   0.00781250   0.00012207   0.00000191   0.00002935
     128   0.03000000   0.00023437   0.00000183   0.00004830
     256   0.11875000   0.00046387   0.00000181   0.00008365
     512   0.47125000   0.00092041   0.00000180   0.00014754
    1024   1.88000000   0.00183594   0.00000179   0.00026487
    2048   7.50500000   0.00366455   0.00000179   0.00048062
    4096  34.86000000   0.00851074   0.00000208   0.00102320

quicksort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00082031   0.00005127   0.00000320   0.00001849
      32   0.00171875   0.00005371   0.00000168   0.00001550
      64   0.00328125   0.00005127   0.00000080   0.00001233
     128   0.00906250   0.00007080   0.00000055   0.00001459
     256   0.01625000   0.00006348   0.00000025   0.00001145
     512   0.03500000   0.00006836   0.00000013   0.00001096
    1024   0.07500000   0.00007324   0.00000007   0.00001057
    2048   0.17000000   0.00008301   0.00000004   0.00001089
    4096   0.35000000   0.00008545   0.00000002   0.00001027
```

```
heapsort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00144531   0.00009033   0.00000565   0.00003258
      32   0.00242187   0.00007568   0.00000237   0.00002184
      64   0.00640625   0.00010010   0.00000156   0.00002407
     128   0.01218750   0.00009521   0.00000074   0.00001962
     256   0.02875000   0.00011230   0.00000044   0.00002025
     512   0.06375000   0.00012451   0.00000024   0.00001996
    1024   0.14250000   0.00013916   0.00000014   0.00002008
    2048   0.31000000   0.00015137   0.00000007   0.00001985
    4096   0.68000000   0.00016602   0.00000004   0.00001996

radixSort
elements   time (sec)   time/n       time/(n*n)   time/(n*log n)
      16   0.00335937   0.00020996   0.00001312   0.00007573
      32   0.00546875   0.00017090   0.00000534   0.00004931
      64   0.01093750   0.00017090   0.00000267   0.00004109
     128   0.02187500   0.00017090   0.00000134   0.00003522
     256   0.04312500   0.00016846   0.00000066   0.00003038
     512   0.08500000   0.00016602   0.00000032   0.00002661
    1024   0.18000000   0.00017578   0.00000017   0.00002536
    2048   0.35000000   0.00017090   0.00000008   0.00002241
    4096   0.69000000   0.00016846   0.00000004   0.00002025
```

Use of Limits

Sometimes a quick way to check whether $f \in O(g)$, $g \in O(f)$, etc. is to look at the limit of f(n)/g(n) as *n* increases without bound. If this limit exists (i.e. is finite), then $f \in O(g)$. This follows from the definition of "limit":

$$\lim_{n \to \infty} h(n) = c$$

means that

$$(\forall\, \varepsilon > 0)(\exists\, n_0)\ (\forall\, n > n_0)\ \ | h(n) - c | < \varepsilon$$

If this limit exists, where the h(n) of interest is f(n)/g(n), then

$$(\forall\, n > n_0)\ | f(n)/g(n) - c | < \varepsilon$$

Thus (since functions of interest to us are positive)

$$(\forall\, n > n_0)\ f(n) < (c + \varepsilon)g(n)$$

By letting d be the maximum of the values of f(n) for $n \le n_0$, we get

$$(\forall\, n)\ f(n) < \max(d, (c + \varepsilon))g(n)$$

By choosing the constant in the definition of O to be max(d, (c + ε)), we have shown

Limit Rule

If a finite limit c of the ratio of f(n)/g(n) exists

$$\lim_{n \to \infty} f(n)/g(n) = c$$

then $f \in O(g)$.

L'Hopital's Rule

Sometimes the limit may exist but is not easy to derive. This happens when the ratio f(n)/g(n) cannot be simplified in an obvious way. An example is log(n) / sqrt(n). Both the numerator and denominator go to ∞ as n increases. In such cases, the notion of derivative can again be used:

$$\lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} f'(n)/g'(n)$$

where f' and g' denote the first derivatives of f and g, provided that

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

We can continue applying this rule iteratively until a reducible form is obtained, since

$$\lim_{n \to \infty} f'(n)/g'(n) = \lim_{n \to \infty} f''(n)/g''(n)$$

Additional Complexity Notation

Here is some additional notation that is used in the literature (see the references by Knuth and by Brassard).

f ∈ Ω(g) is used to designate that $g \in O(f)$. That is, g is a *lower bound* on f, within the confines of some multiplicative constant.

f ∈ Θ(g) is used to abbreviate that $f \in O(g)$ *and* $g \in O(f)$, i.e. the two functions have the *same* growth-rate.

f ∈ o(g) [*f* is "little-oh" of *g*] means that $f \in O(g)$ and that the limit of f(n)/g(n) is 0. In other words, f(n) becomes insignificant compared to g(n) as n gets large.

In particular, use of Θ notation indicates that the given bound is *tight*.

Exercises

1 •• Show that for any positive constant ε, $\log n \in O(n^{\varepsilon})$.

2 •• Show that $\sum_{i=1}^{n} i^2 \in O(n^3)$

3 ••• Derive a closed form expression for $\sum_{i=1}^{n} i^2$. Prove that your expression is correct by induction. (Hint: From the preceding problem, it might be reasonable to try a 3rd order polynomial. If there is such a polynomial, you could solve for its coefficients by constructing 4 equations with the coefficients as unknowns.)

4 ••• Show that for any fixed k, and any c > 1, $n^k \in O(c^n)$.

5 •• Which of the following are true, and why? $2^n \in O(2^{n+1})$. $2^{n+1} \in O(2^n)$.

6 • Suppose that a and b are positive integers with a < b. Which of the following are true, and why? $n^a \in O(n^b)$. $n^b \in O(n^a)$.

7 •• Suppose that a and b are positive constants with 1 < a < b. Which of the following are true, and why? $a^n \in O(b^n)$. $b^n \in O(a^n)$.

8 •••• Suppose that f and g are functions such that $f(n) \in O(g(n))$. Let c be a positive constant. Is it necessarily true that $f(cn) \in O(g(n))$? Justify your answer.

9 ••• Let b be a positive integer. Show that

$$(1 + b + b^2 + b^3 + \ldots + b^n) \in O(b^n).$$

[Hint: There is a closed form for the left-hand side: It is the sum of a geometric series.]

10 •• Show that for any positive integer k, $(\log n)^k \in O(n)$.

11 •• Prove the multiplication rule: If $f \in O(g)$, then $n \to n*f(n) \in O(n \to n*g(n))$.

12 ••• As mentioned earlier, our definition of $f \in O(g)$ applies to the restricted case that the domains are natural numbers. For the more general case of real domains, the definition needs to be modified. In fact, the standard definition of $f \in O(g)$ is

$$(\exists\, c)(\exists\, n_0)(\forall\, n > n_0)\ \ f(n) \le cg(n)$$

That is, there exist constants c and n_0 such that for all n beyond n_0, $f(n) \le cg(n)$. Show that on the domain of natural numbers, this definition is equivalent to the definition given in these notes. [Hint: For a given n_0, the set of natural numbers less than n_0 is finite. Thus one can use the maximum of the values of f(n) over this set in the construction of the constant c.]

13 •••• Using the general definition in the previous exercise, re-prove each of the rules for O that have been derived in the notes.

14 ••• For each of the following program outlines, provide the best "O" upper bound on the time complexity growth rate as a function of parameter N, where P(); is some constant-time computation. In all cases `i` and `j` are declared as `int`. Unless otherwise stated, assume single arithmetic operations are O(1).

a.
```
for( i = N; i > 0; i-- )
      P();
```

b.
```
for( i = N; i > 0; i-- )
      for(j = 0; j < i; j++ )
            P();
```

c.
```
for( i = N; i > 0; i-- )
      for( j = i; j < i+1000000; j++ )
            P();
```

d.
```
for( i = N; i > 0; i-- )
      for( j = i; j > 1; j = j/2 )
            P();
```

e.
```
for( i = N; i > 0; i = i/2 )
      for( j = i; j > 1; j = j/2 )
            P();
```

f.
```
for( i = N; i > 0; i /= 2 )
  for( j = i; j > 0; j-- )
      P();
```

g.
```
int i, j;
for( j = N; j > 0; j-- )
  for( i = j; i > 0; i /= 2 )
      P();
```

h.
```
double F = 1;
```

```
            for( i = N; i > 0; i-- )
                  F *= N;
```

15 ••• Rank the following functions in such a way that *f* precedes *g* in the ranking whenever *f*(n) $\in$ O(*g*(n)). Show your rationale.

$a(n) = n^{1.5}$

$b(n) = n * \log n$

$c(n) = n/ \log n + n^3$

$d(n) = (\log n)^2 + \log(\log n)$

$e(n) = n + 10^9$

$f(n) = n!$

$g(n) = 2^n$

11.17 Case Studies Using Sorting

As already mentioned, the problem of arranging the elements of a sequence into increasing order is called *sorting*. Because the problem is easy to understand and there is a large number of methods, sorting provides a good set of examples for analyzing complexity. Earlier we saw that sorting by repeatedly selecting the minimum element from an array gives an upper bound of O(n^2) for an n-element array. Here we look at other methods for sorting with hopes that good algorithm design can improve this bound. This is important when n gets large. For a sequence of size 10^6, if the complexity of sorting were n^2 microseconds, it would take 10^6 seconds to sort the sequence, i.e. it would take over 11.5 days. If we had a supercomputer that ran 100 times this fast, then the time still might be prohibitive, over 2.7 hours. If we were able to improve the algorithm to n log n microseconds, then a sequence of the same size would take less than 12 milliseconds, or less than .12 milliseconds on the supercomputer.

Although they may be expressed using numbers as data elements, a typical sorting application actually sorts *structs*, data objects consisting of several components or *fields*. Typically only one or two fields comprise the values upon which records are compared. The aggregate of these fields is called the *key* of the record. For example, records representing people might have a combination of last name and first name as the key. If the keys consist only of integers in a relatively small range, the best algorithm is apt to be quite different than in a case where the key is a chemical formula of arbitrary size.

Bucket Sort

For sorting elements chosen from a small range of integers, a **bucket sort** is a good choice. Suppose the range is 0 to N-1 where N is fixed. Then we can sort by constructing an array of N items into which the data are placed. The elements of this array are thought of as *buckets*. The general principle here is called **distribution sorting**, since we distribute the items into buckets. If the data items are just numbers, then since all numbers are alike, the buckets can just be counters of the number of times each number occurs. If the data items are general records, then the buckets would need to be containers for a sufficient number of records. A linked list of records would be a good candidate implementation.

The advantage of bucket sort is that it runs in O(n) time, because we need only make one pass through the input data to put all of the data in buckets, then a pass over the buckets to create the sorted data. This analysis assumes that most buckets are non-empty. If there is a much larger number of buckets than records and many of the buckets are empty, then the time to scan the buckets could dominate. Our O(n) figure assumes that almost all buckets have something in them.

Radix Sort

Radix sort is a variant of bucket sorting which uses fewer buckets by exploiting radix representations of the integer keys. The reason that the number of buckets is of concern is that if there are many more buckets than items to be sorted, and many buckets end up empty, handling the number of buckets could dominate the sorting time.

If the range of numbers is very large, we can conduct the distribution sort recursively, by dividing up the range into sub-ranges, performing an initial distribution, then sorting within the buckets. A variation on this idea is to use the **radix principle**. It is applicable when the values are represented as integer numerals in some base (radix). We sort on each digit of the numerals, starting with the least-significant. If the radix is b, then there are b buckets. We repeat this process, progressing toward the most-significant digit. After each distribution, we regroup the items anew, taking care to preserve their order from the previous distribution. After the last regrouping, the items are sorted.

The radix sorting principle was used on automatic punch-card sorters, and can also be used in hand punch-card sorting. The following illustrates how a home-made indexing system amenable to radix sorting can be constructed using cards. Obtain a deck of cards. Estimate n, the maximum range of values to be used in sorting. Punch log n holes along a specific edge of each of the cards, as suggested by the following diagram. For a card numbered m, cut a channel from the hole to the edge of the card for each hole corresponding to a 1 bit in the binary representation of m.

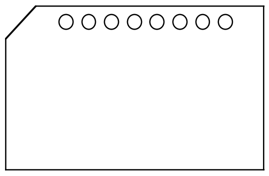


Figure 178: The card for number 0 in the indexing system

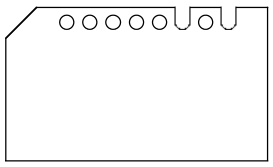


Figure 179: The card for number 5 in the indexing system

To sort the cards, insert a spindle into the holes representing the lowest-order bit. Lift the spindle, separating the cards without channels at that bit to those with channels. Restack the cards with the non-channel cards in front. Repeat this process on the next significant bit, and so on, up to the most significant bit. At the conclusion, the cards will be sorted.

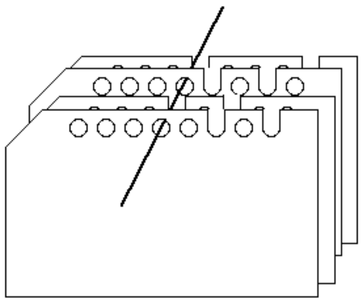


Figure 180: Using a spindle to select the cards having their fifth bit 0, a step in the radix sorting process

The following Java code simulates the sorting process outlined above.

```
class radixSort
{
/**
  *  Calling radixSort constructor on array of floats sorts the array.
  *  Parameter N is the number of elements to be sorted.
  */

// radixSort works using binary representation of numbers being sorted.
// radixSort first sorts on the least-significant bit, then the next least,
// and so on until there are no more bits which have 1 as a value.
// On each pass, it counts the number of words with a 0 in the current
// bit position. It then copies the elements from the array into a
// buffer so that all words with a 0 precede all with a one. It then
// copies the buffer back to the array for the next pass.
```

```
radixSort(int array[], int N)
  {
  int buffer[] = new int[N];    // place to put result of one pass

  boolean done = false;         // indicates whether sorting completed

  for( int shiftAmount = 0; !done; shiftAmount++ )
    {
    // one pass consists of the following:

    int count = 0;              // count of number of 0 bits

    done = true;

    // first phase: determine number of words with 0 bit

    for( int i = 0; i < N; i++ )
      {
      int shifted = (array[i] >> shiftAmount);  // move bit to low-order

      if( shifted > 0 )                         // is anything left?
        done = false;

      if( shifted % 2 == 0 )
        count++;                                // count this 0
      }

    if( done )
      break;


    // second phase: redistribute words with 0 vs. 1

    int lower = 0, upper = count;       // positions for redistribution

    for( int i = 0; i < N; i++ )
      {
      int shifted = (array[i] >> shiftAmount);

      if( shifted % 2 == 0 )
        {
        buffer[lower++] = array[i];
        }
      else
        {
        buffer[upper++] = array[i];
        }
      }

    for( int i = 0; i < N; i++ )
      {
      array[i] = buffer[i];
      }
    }
  }
```

The time to sort a set of numerals using radix sort is proportional to the number of numerals times the number of digits in the longest numeral. If the latter number is bounded by a constant K, then n numerals can be sorted in time proportional to Kn, i.e.

O(n) time. Again, this sort works only in the case that data can be represented as numerals in some radix.

We now turn to sorting methods that work on general keys, using only the assumption that two keys can be compared, but nothing further. A number of obvious sorting algorithms repeat the bound of O(n^2) given by **minsort** discussed earlier. These include:

Simple insertion sort:

Repeat the following process: Begin with a prefix of the array containing just one element as a sorted array. From the remaining elements, choose the next one and find its position in the sorted array. Insert the element by moving the higher elements upward one. The algorithm is expressed in Java is shown below. As before, calling the constructor is what causes the array to be sorted, in place.

```
class insertSort
  {
  private double array[];     // The array being sorted
  int N;                      // The length of the prefix to be sorted

  // Calling insertSort constructor on array of doubles sorts it.
  // Parameter N is the number of elements to be sorted (which might
  // be fewer than are in the array itself).

  insertSort(double array[], int N)
    {
    this.array = array;
    this.N = N;

    for( int i = 1; i < N; i++ )
      {
      insert(i, findPosition(i));
      }
    }


   //  insert(i, j) inserts array[i] into an array at position j,
   //  shifting to the right the elements
   //  array[j+1], array[j+2], ...., array[i-1]

  void insert(int i, int j)
    {
    double hold = array[i];
    for( int k = i; k > j; k-- )
      {
      array[k] = array[k-1];
      }
    array[j] = hold;
    }


   //  findPosition(i) finds the position at which to insert array[i]
   //  in array[0] .... array[i-1]
   //
  int findPosition(int i)
    {
    double item = array[i];
    for( int k = i-1; k >= 0; k-- )
      {
      if( array[k] <= item )
        return k+1;
      }
    return 0;
    }
 }
```

That the above insertion sort is O(n^2) can be seen by analyzing the programs using step counting. Intuitively, minsort and insertion sorts get their O(n^2) linear nature of their attack on the problem: we have an outer loop that runs n steps, and the cost of that loop ranges from 1 to n. If we are to break through O(n^2) to a lower upper bound, we must find an approach that is not so linear. Here is where the following principle suggests itself:

Divide-and-Conquer Principle: Try to break the problem in half, rather than paring off one element at a time.

Perhaps the most obvious divide-and-conquer sorting algorithm is **Quicksort**. At least, it is obvious that the approach is correct. Quicksort is easiest to state recursively:

Quicksort: Sorting by Divide-and-Conquer

Basis: If the sequence consists of at most 1 element, it is sorted.

Recursion:

Break a sequence of more than 1 element into two, as follows:

Chose an element from the sequence as a pivot value. All elements less than the pivot value are selected as subsequence L and all elements greater than or equal the pivot value are selected as subsequence R.

Sort the subsequences L and R (recursively). Then form the sequence consisting of L (sorted) followed by the pivot, followed by R (sorted).

In Java this could be expressed as follows:

```
class quicksort
  {
  float a[];

   // Calling quicksort constructor on array of floats sorts the array.
   // Parameter N is the number of elements to be sorted.
```

```
quicksort(float array[], int N)
  {
  a = array;
  divideAndConquer(0, N-1);
  }

// sort the segment of the array between low and high

void divideAndConquer(int low, int high)
  {
  if( low >= high )
    return;                          // basis case, <= 1 element

  float pivot = a[(low + high)/2];    // select pivot

  // shift elements <= pivot to bottom
  // shift elements >= pivot to top

  int i = low-1;
  int j = high+1;

  for( ; ; )
    {                             // find two elements to exchange
    do { i++; } while( a[i] < pivot );  // slide i up
    do { j--; } while( a[j] > pivot );  // slide j down

    if( i >= j ) break;           // break if sliders meet or cross

    swap(i, j);                   // swap elements and continue
    }

  divideAndConquer(low, i-1);     // sort lower half
  divideAndConquer(j+1, high);    // sort upper half
  }


 // swap(i, j) interchanges the values in a[i] and a[j]

void swap(int i, int j)
  {
  float temp = a[i];
  a[i] = a[j];
  a[j] = temp;
  }
```

Under ideal circumstances, the dividing phase of quicksort will split the elements into two equal-length subsequences. In this case, there will be *log* n levels of recursive calls to quicksort. At each level, O(n) steps must be done to split the array. So the running time is on the order of n *log* n. Unfortunately this is only in ideal circumstances, although by a probabilistic argument that we do not present, it also represents an *average* case performance under reasonable assumptions. The worst case performance however causes the array to split very unevenly, resulting in a worst case of O(n^2), which is the worst case for the other sorting algorithms presented. In a worst-case sense, we have made no progress.

Heapsort: Using a Tree to Sort

Now that we have a hint from Quicksort that O(n log n) *might* be achievable in the best case, we seek an algorithm that has this performance. Whenever we are trying to make improvements over algorithms that deal with linear sequences, the following approach is worth trying:

Tree Structuring Principle:

Rather than dealing with the sequence linearly, try to employ a tree structure to cut sequence traversal needs from n to *log* n.

How about doing insertions within a tree rather than in a linear array, as is done by the simple insertion sort? If there are n elements and we can do each insertion in O(log n) time, we might be able to achieve our goal. Of course there are details to be worked out concerning how we can do the insertions so as to maintain the balance of the tree.

A linear array, as used in *select_min* sort and simple insertion sort, maintains the items sorted thus far in a strict order. By relaxing this condition, we can keep the information "partially ordered" and gain a faster insertion. The original sort of this nature, as presented by Williams, 1964, was called heapsort, reflecting a "heap" structure, a particular type of tree structure.

Note: The heap in this section should not be confused with the *heap* used for general storage of dynamic data structures.

The Heap Invariant

An example of a heap is shown below. A heap has the following defining property (or invariant)

The children of any node cannot be greater than the node itself.

heap invariant

This means that there is a tendency for increasing values as we move toward the root. As a corollary, we see that the progeny of a node cannot be greater than the node itself, and further the root must be the greatest element in the tree.

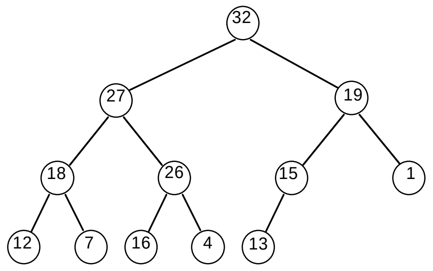


Figure 181: A heap

The following diagram shows the standard implementation of a heap using an array. The array is viewed as being indexed starting at 1. A node can have 0, 1, or 2 children. The children of a node with index p have indices 2*p and 2*p+1.

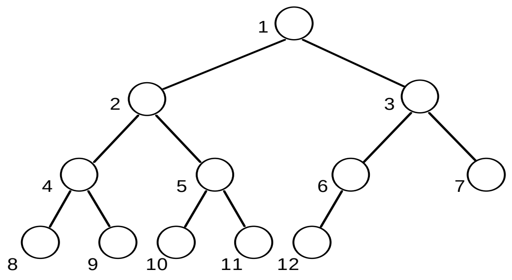


Figure 182: A heap mapped onto the locations of an array.

The numbers show indices of locations, not data values.

The reasoning for choosing 1-origin indexing is just so the relationship between parent and child indices is simple arithmetically. If we use 0-origin instead, it becomes slightly more complicated.

We can get a rough idea of how sorting is done using a heap by first explaining another structure that can be implemented as a heap: a **priority queue**. This term was introduced earlier. Recall that in a priority queue discipline, the *largest* item is always the next to be removed. Evidently, the largest item in a heap is always the root, so it is easy enough to locate. However, in removing the root, we must fill the vacancy thus left in such a way that the result is still a heap. Our goal will be to show that the heap can be formed in time O(n log n) and reformed, after removing an element, in time O(log n). Given this, the following code indicates how a heap can be used to achieve an O(n log n) sorting algorithm.

```
.... create an empty heap ....
for( i = 0; i < n; i++ )
     .... add a[i] to the heap ....

for( i = n-1; i >= 0; i-- )
     .... remove a[i] from the heap ....
```

The heapsort algorithm also uses a space-saving trick: using the vacated locations in the heap for storage of the final sorted array. This means that no additional memory space is needed.

Deleting the Maximum from a Heap

The ability to find the maximum quickly within a heap is based on the "partial ordering" of the node values. To preserve this property, it is not enough to simply remove the root.

When removing the max from a heap, we must adjust the tree immediately afterward so that the max so that the heap invariant once again holds.

preservation of the heap invariant

Below we show our original heap example after removing the max. Obviously there is a "hole" at the root that needs to be filled. Also obviously, the value that must fill this hole is 27, since it is the maximum of the remaining nodes.

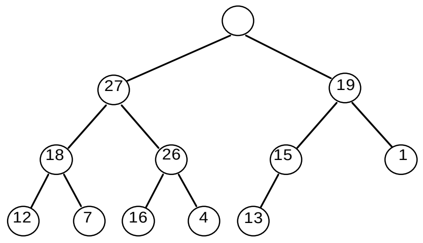


Figure 183: Original heap example after removing the maximum,

but before restoring the heap invariant

However, if we move 27 to the root, that leaves another hole, etc. Continuing this process, we would eventually have a hole in the top row. This situation is undesirable, for it means that the heap can no longer be represented as a contiguous array.

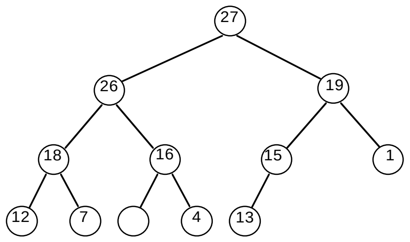
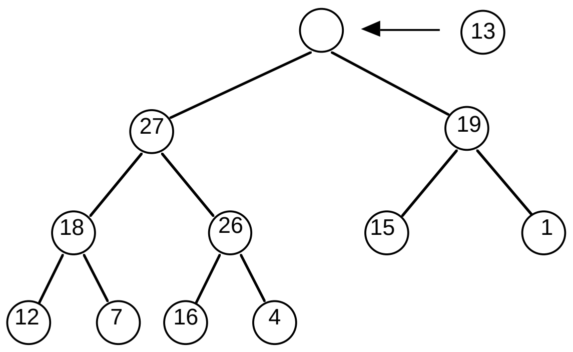


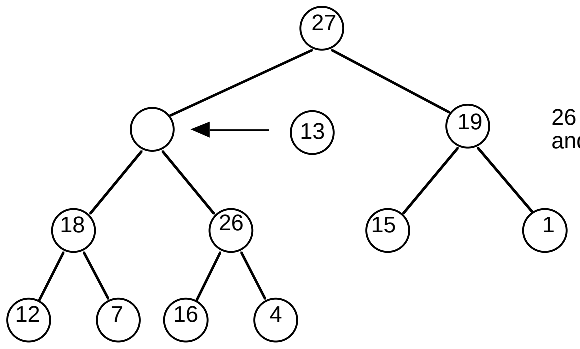
Figure 184: Undesirable situation: after filling the hole at the root with the maximum, and continuing this process down the tree, we have left a hole at the bottom level. This heap can no longer be represented as a contiguous array.

In order to avoid this situation, we shall have to plan in advance to preserve the contiguity of the heap. We can do this by removing that *last* item in the heap, i.e. the one that appears in the lower right node of the tree, 13 in this case. So we temporarily make an orphan out of 13, to find a new home somewhere in the array.

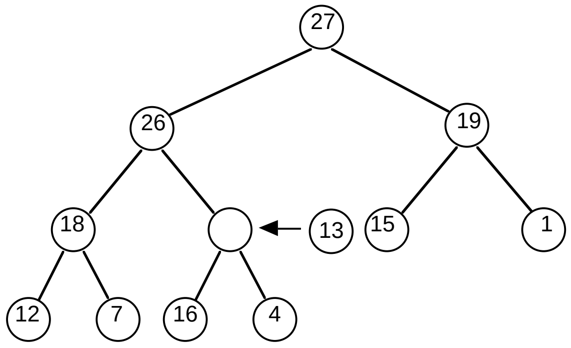
We fill the hole at the root with this former leaf value (13), then adjust the array by "bubbling up" that value. In this case, bubbling up means to repeatedly adjust a combination of three nodes so that the maximum is the parent, as shown in the following sequence.



27 is the maximum of {13, 27, 19} and will "bubble up"



26 is the maximum of {13, 18, 26} and will "bubble up"



13 continues its descent as others bubble up

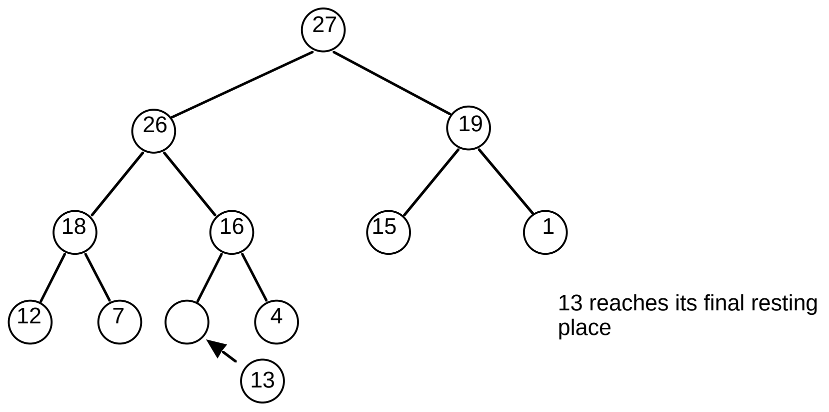


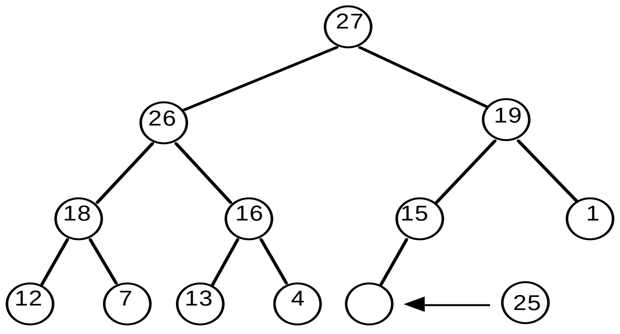
Figure 185: Restoring the heap invariant

At this point, the heap invariant is restored and the corresponding heap is again contiguous. The algorithm for delete_max then is a mechanization of this process. We notice that the algorithm for deleting max runs in O(*log* n), since all accesses to nodes are made along a single path from root to some leaf. We only look at nodes directly on that path and single nodes to one side or the other.

Initial Creation of a Heap

A heap can be created by starting with an empty heap and repeatedly adding new elements. As with removal, we must show how to maintain the heap invariant when inserting, as well as making sure that insertions are also O(*log* n).

Let us work with the heap above as an example, and suppose that the value 25 is to be added. As before, to maintain contiguity as a heap, we will need to put into play the node shown vacant below. We use a technique similar, but not identical, to the one used for removing the maximum: We put the new element 25 into the vacancy, then let nodes above it change places until an appropriate level is reached. This is simpler than deleting the maximum, because it only involves a 2-way comparison, rather than a 3-way one.



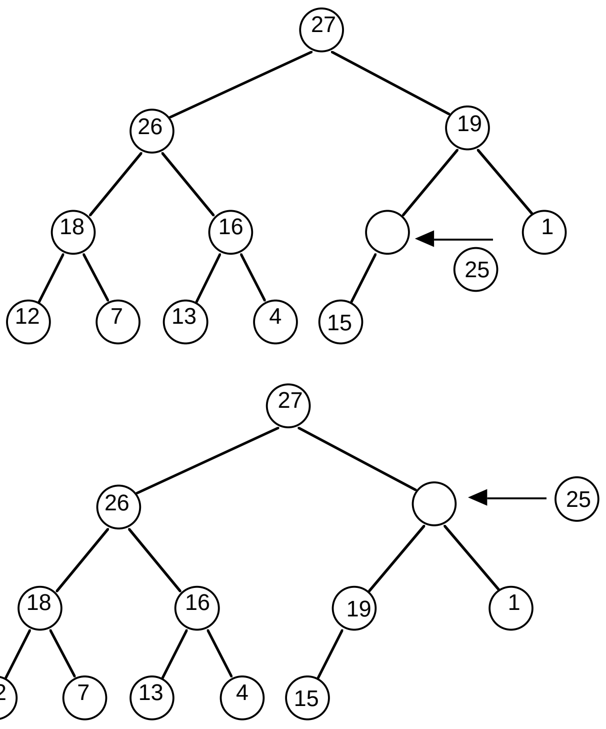


Figure 186: Bubbling up in the initial creation of a heap

At this point, the proper position for the new element has been found, so bubbling stops, leaving us with a new heap. Again we notice that the heap modification is accomplished in O(log n), since only nodes on the path from the created vacancy to the root are examined.

The following code shows the combination of these ideas in the heapsort procedure, which sorts an array in place:

```
class heapsort
  {
  private float array[];        // The array being sorted

  // Calling heapsort constructor on array of floats sorts the array.
  // Parameter N is the number of elements to be sorted.

  heapsort(float array[], int N)
    {
    this.array = array;
    int Last = N-1;

    // A heap is a tree in which each node is smaller than either of its
    // children (and thus than any of its descendants). All sub-trees
    // of a heap are also heaps. In this program, a heap is stored as
    // an array, with the root at element 0. In general, if a node is
    // at element I, its children are at elements 2*I+1 and 2*I+2.
    // phase 1: form heap
    // Construct heap bottom-up, starting with small trees just above
    // leaves and coalescing into larger trees near the root.

    for( int Top = Last/2; Top >= 0; Top-- )
      {
      adjust(Top, Last);
      }


    // phase 2: use heap to sort
    // Move top element (largest) out of heap, swapping with last
    // element and changing the heap boundary, until only one element
    // remains.

    while( Last > 0 )
      {
      swap(0, Last);
      adjust(0, --Last);
      }
    }


  //  adjust(Top, Last) adjusts the tree between Top and Last

  void adjust(int Top, int Last)
    {
    float TopVal = array[Top];                  // Set aside top of heap
    int Parent, Child;

    for( Parent = Top; ; Parent = Child )       // Iterate down tree
      {
      Child = 2*Parent+1;                       // Child is left child

      if( Child > Last )
        break;                                  // No left child exists

      if( Child+1 <= Last                       // Right child exists
           && array[Child] < array[Child+1] )   // and is larger
        Child++;                                // Child is larger child

      if( TopVal >= array[Child] )
        break;                                  // Location for TopVal

      array[Parent] = array[Child];             // Move larger child up
      }

    array[Parent] = TopVal;                     // Install TopVal
    }


   // swap(i, j) interchanges the values in array[i] and array[j]

  void swap(int i, int j)
    {
    float temp = array[i];
    array[i] = array[j];
    array[j] = temp;
    }
  }
```

Merge Sort

One natural method of sorting is to sort by repeated merging. A set of rules for this form of sort was given in the chapter *Low-Level Functional Programming*. Our analysis of merge_sort as given in that chapter follows:

a. The time to merge a pair of lists to get an *N* element list is O(N) since each recursive call "retires" an element to the result list and each element of the result gets retired only once.

b. Thus, the time to *merge_pairs* on a list of lists, the summed total length of which is N, is O(N), since from a. the time to merge a single pair is proportional to the number of elements in the result.

c. The time to use *repeat* on a list of *N* 1-element lists is the number of times *repeat* is called recursively times O(N), from b.

d. The number of times *repeat* is called is O(log N), since each call sees a list that is about half as long as the previous call.

e. The time to merge_sort an *N* element list is O(N) + time to repeat an *N* element list, since mapping an *N* element list is O(N). From c and d, the time to repeat is O(N log N).

Therefore the overall time to *merge_sort* an *N* element list is O(N log N).

11.18 Complexity of Set Operations and Mappings

A recurring theme in computer problem solving is the need to deal with various kinds of sets and mappings. Frequently we have need for sets of integers, strings, and more complex data types, as well as mappings from a wide variety of types to integers, etc. Thus it is worthwhile using these as one of the foci in our discussion of complexity. A wide variety of different representations for these abstractions is available and they differ in the types of data they handle, the complexity, space requirements, and coding difficulty. It is helpful to taxonomize the methods based on these points.

Thinking of a set as a class, the following operations are typical:

find	Find out whether a member is in the set. If so, return a locator for it. A locator is a kind of reference to the member. It is used in deleting the member or changing it.
add	Add a new member to the set (assuming it is not present already)
delete	Given a locator for a member, remove the member from the set.

new	Create a new set.

Set Implementation Using an Unordered Array

Perhaps the simplest way to represent a set is as an array, the elements of which are in no particular order. The implementation is similar to that for a stack using an array, with an index indicating the last element in the set. Adding to the set (assuming there is space available) is trivial, essentially the same as a push onto the stack. Finding a member requires a search through the array up until the point the element is found or the end is reached. In the worst case, all elements in the array are examined. Delete, given a locator, is simple: since order is not important, we can fill the hole left by deletion by putting the last element in its place (unless, of course, we wish to maintain the order of insertion, but once order becomes important, we no longer have a set).

Letting *N* represent the current set size, we can see from the above discussion that the following upper bounds hold on the set operations:

find	O(N)
add	O(1)
delete	O(N)
new	O(1)

Here we assume that the delete accounts only for the time to do the deletion, not to find the element being deleted as well. A similar discussion and set of bounds holds for using a **linked-list** to represent a set.

Binary Search Principle

Some improvement can be obtained, at the expense of slightly greater coding complexity, by keeping the members of the set *in order*. This requires that an order be available for the domain of the set, which is not always the case. The ordering assumption is not at odds with the definition of a set being unordered, since its use is for internal purposes.

By keeping the set elements in order, the technique of **binary search** becomes available for the find operation. Here is how binary search works for find: Using index computation to find the index of the mid-point of the array, we take a "stab" at that point and compare it with the element to be found. If we have equality, we return the index as the locator. If the element to be found is less than the mid-point, then we search in the half-array below the mid-point; if it is greater, we search in the half-array above. This process is repeated until the element is either found, or we are left with an empty array to search.

Each step of the binary search procedure reduces the number of elements still under consideration by half. Therefore, the number of steps is proportional to the number of halvings it takes to reduce the original number of elements, *N*, to less than 1. This number of steps is, of course, *log N*.

The superiority of binary search for find in an ordered array is tempered somewhat by the costs of addition and deletion. In order to maintain the ordering, we have to shift elements one way or the other when we insert or delete. In the worst case, we might have to shift all of the elements. The complexity for the ordered array case would thus appear as follows:

find	O(log N)
add	O(N)
delete	O(N)
new	O(1)

If most of the activity involving the set is of the *find* type, with few additions or deletions, then binary search on an ordered set is preferred over using an unordered set.

Binary Search Trees

Does using an ordered linked-list help in a similar way? If we have a linked structure, additions and deletions usually become less complex. Unfortunately a linked-list doesn't provide a good way to do the index computation required for binary search. That is, we'd like to have a way to find the mid-point of a list similar to what was used for an array. But without adding additional structure, there is no such way. The desire to have something approaching a linked structure on which binary search can be done has motivated the use of various tree structures, the most obvious of which is the **binary search tree**.

A binary search tree enables binary searching on a linked structure by approximating access to successive mid-points, similar to the way that binary search was described. To do this, each node in the structure has two links, one to the elements below the current one and one to the elements above. The figure shows a binary search tree that holds the set {1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15}

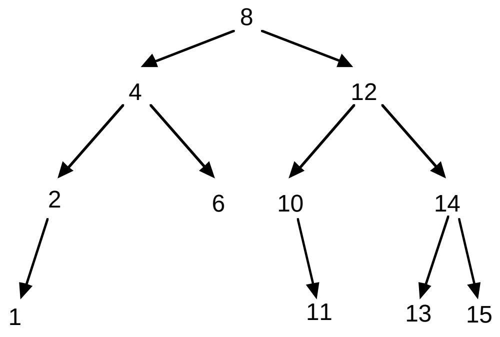


Figure 187: A binary search tree

The defining property of a binary search tree is that all elements in the left sub-tree of a node are less than the node itself, while all elements in the right sub-tree are greater than or equal to the node. If the binary search tree is ***balanced***, meaning that for a tree with N nodes, the length of the longest path is at most 1 + log N, then the search can be done in time O(*log N*). We can also insert and delete in time O(*log N*). However, it is not obvious that we can insert and delete in this amount of time and still maintain the balance needed for O(*log N*) search. In fact, several extensions of binary search trees have been developed that maintain the balance. These include AVL-trees, 2-3 trees, 2-3-4 trees, red-black trees, and B-trees. The reader might explore these interesting possibilities in future courses.

Bit Vectors

If the domain of a set's elements consists of integers, or can be easily mapped into integers (for example, character strings can be considered as large-radix numerals and are thus identifiable as integers), then a very fast method of set representation is available. A bit vector is an array with one array element per domain element. The value of an element is either 1, indicating that the corresponding domain element is a member of the set being represented, or 0 indicating that it is not. For example, if the domain is {0, ..., 15}, then a bit vector representation of the set {1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15} is

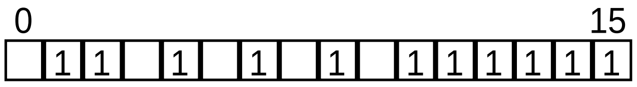


Figure 188: A bit vector representation of the set

{1, 2, 4, 6, 8, 10, 11, 12, 13, 14, 15}
with empty entries representing 0

The advantage of a bit vector is that *find* is extremely fast. We simply index the position corresponding to the element we are trying to find. If the value is 1, the element is in the set, otherwise it is not. By the linear addressing principle, *find* is O(1) time. Similarly, add and delete are also O(1) time. In a bit vector, the actual elements are not stored, so the representation can be compact in terms of space. However this compactness only occurs if the elements in the set are relatively **dense** in the range of possible elements. The amount of space required by a bit vector is O(M) where M is the domain size. If the set is sparse and M is large, much space will be wasted. Also, it requires time proportional to M to *initialize* a bit vector, compared to time proportional to the size of the set for the other representations studied so far. To summarize, **for a bit vector**:

find	O(1)
add	O(1)
delete	O(1)
new	O(M)

Analysis of Hashing

Hashing was introduced in the chapter *Implementing Information Structures*. Under nominal conditions, the time for *find* using hashing is best regarded as O(1). This assumes that

- the elements of the set are distributed relatively evenly into buckets,
- the size of any one bucket is bounded by a constant, and
- the time to compute the hashing function is bounded by a constant

Any of these assumptions can be violated, but there are ways to remedy any tendency to violate them. For example, the first assumption can be satisfied by devising an appropriate hashing function, such as *hash_pdg* presented earlier. The size of buckets are bounded, assuming the first assumption holds, by making the table large enough. If the number of elements in the set is not known in advance, there are ways to extend the table size dynamically. This is not a trivial problem, but it is a solvable one, using techniques such as "extendible hashing" (see Fagin, *et al*. 1979). The third assumption is true if there is a bounded number of digits in the representation of each element. Obviously this would not be true if we used arbitrarily-long strings, but it is approximately true for many common cases. Since we have to look at every character of the string to be matched anyway, and the time to compute a typical hashing function is bounded by a constant times the length of the string, this time can be considered to be part of the cost of looking at the elements to be found.

The Trie Principle

Trie representation is a form of tree demonstrated in *Information Structures*. Unlike binary search trees, but similar to radix sorting, tries can exploit situations where the data can be represented as numerals. Since the linear addressing principle applies at each level of a trie, the access time to any element of a trie is O(1) if the number of levels is bounded, or O(L) where L is the number of levels, in general.

Sets vs. Bags and Mappings

So far we have emphasized structures for storing sets. However, most of these structures can also be adapted to implement bags and mappings as well. Usually it is a matter of storing additional information with the element inside the representation. For example, with bags we store a number indicating the *multiplicity* of the element in the bag; the absence of an element implies multiplicity 0. With mappings, the elements stored in the set are the domain values, and with each element in the domain we store the corresponding range value.

Exercises

1 ••• Rewrite the radix-sort procedure using a queue abstract data type. In particular, store the initial data on one queue and dequeue each item, placing it in one of two other queues depending on the least significant bit. Repeat this process for bits of successively-increasing significance. (Note: You do not have to implement the queue data type; that was discussed in Computing Objectively.) Hopefully your algorithm is easier to understand now that more abstraction has been employed.

2 ••• Try to devise an O(n) sort for numbers of fixed precision using a trie.

3 ••• Design a data abstraction for *"bignums"*, integers of arbitrary precision, using internal arrays of fixed-precision items (such as `short int`). Implement the operations of addition, subtraction, and multiplication at a minimum. Give O bounds on the complexity of your operations as a function of argument size.

4 ••• Using your implementation in the previous problem, code the Russian peasants' method of raising a bignum to a power. Analyze the complexity of raising a fixed constant to a power. See how well your analysis agrees with empirical observation.

5 •••• Explore the possibility of speeding up bignum multiplication using the divide-and-conquer strategy.

6 ••• Empirically compare the performance of a spell checker using hashing against ones using (a) binary search, and (b) a trie. Assume that you do not count the time taken to create the ordered array or the trie.

7 ••• Conduct a literature search on methods for keeping binary search trees in balance, so as to ensure an O(log n) search time.

8 •• Write a program that will do a fast spelling check by using a dictionary stored as a hash table. Populate the table from a dictionary files, such as `/usr/dict/words` available in most UNIX® systems. Compare the speed of your program to one that searches the dictionary sequentially.

9 ••• Suppose you wish to treat arbitrary Polys as keys. Develop a hash function for this application. Use recursion and avoid converting the Poly into text first.

10 ••• Implement merge sort. Use linked lists. Verify empirically the *n log n* upper bound

11.19 Chapter Review

Define the following terms:

Amdahl's law
asymptotically dominated
binary search
binary search tree
bit vector
bucket sort
complexity
distribution sorting
divide and conquer
growth-rate
hashing
heapsort
insertion sort
L'Hopital's rule
merge sort
"O" notation
profiling
quicksort
radix sort
tight
trie
upper bound

Describe how you would estimate the complexity of a program empirically.

Describe the role of limits of sequences in interpreting complexity bounds.

11.20 Further Reading

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, 1983. [Moderate.]

Jon Bentley, *Programming Pearls*, Addison-Wesley, Reading Mass., 1986. [A series of articles on algorithms and programming. Easy to moderate.]

Jon Bentley, *More Programming Pearls*, Addison-Wesley, Reading Mass., 1988. [A continuation of Bentley, 1986. Easy to moderate.]

G. Brassard. *Crusade for a better notation*. ACM SIGACT News, **17**, 1, 60-64 (1985).

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms,* MIT Press, Cambridge, Mass., 1990. [Moderate.]

E. Fredkin, *Trie memory*, Comm. ACM, **3**, 4, 490-500, 1960.

R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong, *Extendible hashing – a fast access method for dynamic files*, ACM Trans. on Database Systems, **4**, 315-344, 1979. [Shows an interesting way to expand hashing tables by splitting buckets. Moderate to difficult.]

R.W. Floyd, *Algorithm 245: Treesort 3*, Comm. ACM, **7**, 12, 345, 1964 [Improvements on the original heapsort.]

C.A.R. Hoare, *Quicksort*, Computer Journal, 5, 1, 10-15, 1962.

D.E. Knuth. *Big omicron and big omega and big theta*. ACM SIGACT News, **8**, 2, 18-24 (1976).

D.E. Knuth, *The Art of Computer Programming*, 3 volumes, Addison-Wesley, Reading Mass., 1973, 1981. [Comprehensive reference on algorithms and analysis. Moderate to difficult.]

Williams, J.W.J., *Algorithm 232: Heapsort*, Comm. ACM, **7**, 6, 347-348, 1964 [Original paper on heapsort.]