Algorithm Efficiency

1 Introduction

The objective of this chapter is to present a method of evaluating and comparing the ***performance of algorithms***. This begs the question, what do we mean by performance? Computer scientists usually refer to two different performance metrics: time and space, time being how much time an algorithm needs to solve a problem, and space meaning how much memory it needs.

Note that we are not interested in measuring the performance of *programs*, but of *algorithms*. For this reason, we are not interested in the performance differences between two implementations of the same algorithm, nor are we concerned with precise calculations of running times, since these depend on the implementation, the compiler, the operating system, and the machine architecture. Instead, we devise a means of evaluating the performance of an actual algorithm, independent of its implementation, and to do this we must work at a higher level of abstraction.

Although one can actually "time" the execution of a program, this does not always help us to understand the cost of running the program with larger-sized input data. But on the other hand, one cannot "time" an algorithm, since an algorithm is not executed on hardware. Instead of timing an algorithm, we mathematically analyze its running time to assess its efficiency.

The running time of an algorithm depends on the input it is given. For some inputs it might be faster than for others. We are usually interested in knowing its ***worst case behavior***, because then we can plan conservatively. Sometimes though, we want to know the "average" running time. The idea of "average" is a bit vague and also a bit useless, because "average" running time treats all inputs as equally likely to occur, which is never true. What is more useful is the ***expected running time***, which is a weighted average, i.e., each input is weighted by its probability of occurrence. This is also a little impractical, as we rarely are able to give weights to the inputs realistically, because we do not know their respective probabilities of occurring. Therefore, people generally use the average running time, knowing that it is only an approximation. Because the expected running time is sometimes possible to derive, we define it here.

Definition 1. Let D denote the set of all possible inputs to an algorithm. (D stands for domain.) Let x be any element of the domain. Let $C(x)$ be a cost function that assigns a cost, such as the running time, to each input x, and let $p(x)$ be the probability that x will be chosen as an input to the algorithm on an arbitrary run of it. Then

$$E[C] = \sum_{x \in D} p(x) \cdot C(x)$$

is the *expected value of the cost function* $C(x)$ for the given probability function p.

Because we do not have the tools yet to describe the running time of algorithms, we will use a program to illustrate how to apply this definition.

Example 2. Suppose an interactive program has a set of eight possible input commands, and market analysis has shown that their probabilities of being issued to the program by a large class of users are given in the table below. Suppose that cost function for each command has also been determined experimentally as the number of seconds it takes to execute the command; their values are in the table below. For simplicity, the commands are labeled c1, c2, ..., c8.

x	c1	c2	c3	c4	c5	c6	c7	c8
$p(x)$	0.1	0.05	0.1	0.05	0.2	0.05	0.25	0.2
$C(x)$	5.0	2.0	3.0	4.0	8.0	2.0	1.0	3.0

Then the expected running time in seconds for this program with the given input distribution is

$$\begin{aligned}
E[C] &= \sum_{x \in D} p(x) \cdot C(x) \\
&= (0.1 \cdot 5.0) + (0.05 \cdot 2.0) + (0.1 \cdot 3.0) + (0.05 \cdot 4.0) + (0.2 \cdot 8.0) + (0.05 \cdot 2.0) + (0.25 \cdot 1.0) + (0.2 \cdot 3.0) \\
&= 0.5 + 0.1 + 0.3 + 0.2 + 1.6 + 0.1 + 0.25 + 0.6 \\
&= 3.65
\end{aligned}$$

There are actually three different measures of the running time of any algorithm:

Best case: The running time under the best of all possible conditions.

Expected case: The running time averaged over all possible conditions, using some probability distribution of the input domain.[1]

Worst case: The running time in the worst of all possible conditions.

For each of these, the measure is expressed as a function of the size of the input, the idea of which will be explained shortly. Best case analysis is rarely used because one usually does not want to know how an algorithm will behave in the best possible situation. The expected case is useful only when the following three conditions are true:

1. It is possible to postulate a probability distribution of the input domain that reflects actual usage.
2. It is mathematically tractable to calculate.
3. Nothing catastrophic can result from using the algorithm under worst case conditions without having planned for it.

Because Condition 1 is unlikely to be true, a *uniform distribution* is often assumed. A uniform distribution is one in which every input is equally likely to occur. This makes the analysis less meaningful, since it may not reflect actual conditions. Condition 3 is usually the determining factor; if an algorithm absolutely must perform within a fixed running time, as when it is being used in an embedded system with real-time constraints, one cannot base an assessment on any kind of averaging of the running times.

Worst case analysis is almost always what people use to decide on the running time of an algorithm, because it provides an upper bound on how "bad" it can be.

2 Input Size

Every input is assumed to have an integer size that depends on the particular problem to be solved. For example, the input to an algorithm that sorts a list of items is a list. The number of elements in the list is the size of the input, not the lengths of the items to be sorted, or their combined lengths, because usually the sorting algorithm will depend on the number of elements, not how big each element might be. The input to an algorithm that searches for the occurrence of one string in a second string, would be the two strings, and so there may be two integers that influence the running time: the length of the first string and the length of the second. Different algorithms may depend on their sum, or product, or some other function of the two sizes.

Every algorithm has a particular set of input data, and the size of that data will depend on the particular problem. In any case, the running time will always be viewed as a function of the size of the input.

[1] If you have not yet had a course in probability, then the way to view this is as follows. Imagine that the set of all possible inputs to the algorithm is finite. Suppose that S is the name of the set, for argument's sake. Imagine that the program will be run in all kinds of environments, by all differ types of users. Some users are more likely to input a particular value than others. This leads to the idea that certain elements of S are more "likely" to be inputs to the program than others. The "likeliness" of these elements is called the probability distribution of S.

Examples

- For an algorithm that sorts an array of items, the input size is the number of items in the array.
- For an algorithm that searches for a given item within an array, the input size is the number of items in the array.
- For an algorithm that merges the lines of two different files in a particular order, the input size is two values, the number of lines in each file.
- For an algorithm that evaluates a polynomial, the input size is the number of nonzero terms of the polynomial.
- For an algorithm that must determine whether a number is prime or not, the input size is the number of bits in the number (not the actual number!)

3 Operator Counts as a Cost Function

The running time of an algorithm is obtained by assuming that each primitive operation takes the same amount of execution time and counting the total number of operations that the algorithm requires for a given input. For example, the `retrieve()` list member function may, in the worst case have to look at every element of the list in order to complete its task. The list is the input in this case, and the number of operations will be proportional to the length of the list.

Typically, all arithmetic operators, logical operators, control-flow operators, array subscripting operators, pointer dereferencing operators, and other operators similar to these take a single "step." Function calls, returns, and parameter transmission each count as a single step, even though they actually involve more instructions than simple assignment operators. The simplest way to approximate the number of steps is to count the operators, and the number of operands of each function call. Treat the parentheses of all conditions as a single operator.

Example. Consider the code

```
if ( z == f(x) )
    y = 0;
else if ( z > f(x) )
    y = 1;
else
    y = -1;
```

Executing this code, if `z == f(x)`, will take 6 operations: if-statement (1), relational operator (1), function call with one parameter passed and returned (3), assignment operator (1).

Executing the code when `z < f(x)` will take 11 operations: two if-statements (2), two relational operators (2), two function calls with one parameter passed and returned (6), one assignment (1).

There are simple algebraic rules for counting the number of steps in an algorithm.

Rule 1. Loops

The running time of a loop is at most the running time of the statements inside the loop times the number of iterations of the loop. It can be less if the loop body has conditional statements within it and the number of statements within the branches varies.

Rule 2 – Nested Loops

The total running time of a statement inside nested loops is the running time of the statement multiplied by the product of the sizes of the loops.

```
for ( i = 0; i < n; i++ )
    for ( j = 0; j < m; j++ )
        k = k+1;
```

The body of the nested loops is 1 step if we assume the increment takes place in a register, so the running time is approximated by $n * m$. If we count more accurately, each iteration of a for-loop adds the comparison and the increment/assign, or 2 steps, and the final comparison that fails is 1 step, and the initialization is 1 step. Therefore, each iteration of the inner loop uses $1 + 3m + 1 = 3m + 2$ steps, and the total running time is $1 + n \cdot (3m + 2) + 2n = 3mn + 4n + 1$. We normally do not care about the added constants, nor the constant factors, so it is sufficient to say that the running time is $c_1 mn + c_2 n$ for some constants $c_1$and c_2.

```
for ( i = 0; i < n; i++ )
    for ( j = i; j < n; j++ )
        k = k+1;
```

The body of the nested loops is 1 step if we assume the increment takes place in a register. The inner loop executes $n - i$ times for each i, so the running time is approximated by $(1 + 2 + 3 + ... + n = n(n + 1)/2$. Again, we would state that the running time is $cn(n + 1) = cn^2 + cn$ for some constant c.

Rule 3 – Consecutive Statements

The total running time of a sequence of statements is the sum of the running times of the statements. This makes sense. However, when we introduce order notation, the running time of a sequence of statements will be defined as the maximum of the running times of the individual statements. If, for example, a loop is one of the statements, then its running time may "dominate" the total time.

Rule 4 – Conditional Statements

The running time of an if/else statement

```
if (condition)
    S1
else
    S2
```

is at most the running time of the evaluation of the condition plus the maximum of the running times of `S1` and `S2`. Because we do not know which branch is taken in general, when we are interested in worst case analysis, we have to use the maximum, unless we can prove one of the branches is never executed with any inputs.

Rule 5 – Function Calls

If a sequence of statements contains function calls, the running time of the calls must be determined first. For recursive functions, the analysis can get difficult. This is a topic that we cover in the sequel to this class.

```
long fib(int n)
{
    if (n <= 1)
        return 1;
    else
        return fib(n-1) + fib(n-2);
}
```

This is a recursive function. The running time for any $n <= 1$ is the cost of the parameter passing (1), the single if-test (1), the comparison (1), and the return (1), for a total of 4 steps. However, the cost for $n > 1$ is a recursively defined function. Let $T(n)$ be the running time of this function given input n. Then $T(0) = T(1) = 4$. For $n > 1$, the total is

$$T(n) = T(n-1) + T(n-2) + 9$$

because there 9 steps in addition to the two recursive calls. We will not solve this recurrence relation here. However, it is worth knowing that the running time is an exponential function of n.

3.1 Weakness of Counting Operations as a Measure

The idea of using operator counts is at this point questionable. In a modern processor, the dominant cost is not the time to execute the instructions, but the time to fetch data from primary memory. This can be an order of magnitude larger than the time to execute an instruction. However, there are many factors that can influence the number of times that data must be fetched from memory, because of caching within processors. The way in which the particular algorithm accesses its data, the relationship between data sizes and cache block sizes and the type of cache, and the amount of cache in general, all influence the overall time spent accessing memory.

4 Big "O" and Asymptotic Rates of Growth

First, we define a method of measuring how "fast" real-valued functions of a single variable can grow. Use your intuition here. Consider the non-decreasing functions

$$f(x) = x^2$$
$$g(x) = x^3$$
$$h(x) = 5x^2$$

Your intuition should tell you that as x gets larger and larger, the function $g(x)$ grows faster and faster than the others. Furthermore, for any value of x, $h(x)$ will always be exactly $5f(x)$. So whatever the rate of growth of $f(x)$, $h(x)$ is growing at the same rate as $f(x)$. They stay in a kind of lock step, with h and f proportionally the same as x marches towards infinity. On the other hand, as x increases, clearly $g(x)$ gets larger and larger than both $f(x)$ and $h(x)$. To see this, look at the first six integer cubes: 1, 8, 27, 64, 125, 216 in comparison to the first six integer squares: 1, 4, 9, 16, 25, 36. Even the first six values of $h(x)$ are overtaken by the faster growing $g(x)$: 5, 20, 45, 80, 125, 180. The point is that whatever means we use to measure the relative rates of growth of functions, it ought to ignore constant factors such as the 5 above, and must rank functions like cubics ahead of quadratics.

Table 1 shows the approximate values of selected functions with varying rates of growth. A good method of measuring the rates of growth of functions would rank these functions in the order they appear in the table, because each successive row is growing faster than the preceding one.

Order of magnitude analysis is used for comparing the relative rates of growth of the running times of algorithms. Order of magnitude will ignore constant multiples and differences of functions. Instead of making a statement such as

	n				
$f(n)$	10	100	1000	10000	10^6
$log_2(n)$	3	6	9	13	19
n	10	100	1000	10000	10^6
$n \log_2 n$	30	664	9965	10^5	10^7
n^2	10^2	10^4	10^6	10^8	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{18}
2^n	10^3	10^{30}	10^{301}	10^{3010}	10^{301030}

Table 1: Approximate values of selected functions of n

"Algorithm A runs in $5n^2$steps for inputs of size n."

we will say that

"Algorithm A runs in time proportional to $n^2$for inputs of size n."

Suppose algorithm A solves a problem in time proportional to $n^2$and algorithm B solves the exact same problem in time proportional to n for inputs of size n. Then even if B's more accurate running time were $100n$, as n increases, A's running time would overtake B's.

What matters most is how fast the function grows as n increases. In the above example, the rate of growth of A's running time, n^2, is greater than the rate of growth of B's running time, n.

Definition 3. Algorithm A is order $f(n)$, denoted $O(f(n))$, if for any implementation of algorithm A, there is a positive constant c and a constant n_0 such that for all $n \geq n_0$, A requires no more than $c \cdot f(n)$ time units to solve a problem of size n.

The requirement that $n \geq n_0$ in the definition is the mathematical way to say that, for inputs that are sufficiently large, the running time is bounded by a constant multiple of $f(n)$. You pronounce the notation $O(f(n))$ as "big-O of f(n)". This is called ***order notation*** and the "big-O" is a way to characterize the *order of magnitude* of the rate of growth of the function.

Example 4. Suppose the running time of an algorithm with input size n is $3n^2+5n-6$. Then the algorithm is $O(n^2)$ because there is a constant $c = 8$ and an integer $n_0 = 0$ such that for all $n \geq n_0$, $3n^2+5n-6 \leq cn^2$. If you substitute the value of c, you will see that

$$3n^2 + 5n - 6 \leq 8n^2 \quad iff \quad 0 \leq 5n^2 - 5n + 6$$

which is true for any value of n greater than or equal to 0.

Example 5. Suppose the running time of an algorithm with input size n is $n^4 + 100n^3 + 100$. Then the algorithm is $O(n^4)$ because for all $n \geq 2$, $n^4 + 100n^3 + 100 \leq 101n^4$. To see this, note that

$$\begin{aligned} n^4 + 100n^3 + 100 &\leq 101n^4 \qquad iff \\ 0 &\leq 100n^4 - 100n^3 - 100 \qquad iff \\ 0 &\leq n^4 - n^3 - 1 \end{aligned}$$

and since for any $n > 1$, $n^4 - n^3 - 1 \geq 0$. Therefore, the constant $c = 101$ and the constant $n_0 = 2$.

Although we have applied the definition strictly in these examples, there are rules that make it easy in many cases to show that the running is big-O of some function. For example, you should see that if an algorithm is $O(f(n))$ for some function $f(n)$, then it is $O(k \cdot f(n))$ for any constant k.

Proof. Suppose that the algorithm is $O(f(n))$ for some function $f(n)$, and let k be a positive constant. By the definition of big-O there is a positive constant c such that the algorithm runs in at most $cf(n)$ steps for sufficiently large n. Let $b = c/k$. Since $b \cdot k \cdot f(n) = (c/k) \cdot k \cdot f(n) = c \cdot f(n)$, there is a constant b such that the algorithm runs in at most $b \cdot kf(n)$ steps for all sufficiently large n, or stated in terms of big-O, the algorithm is $O(kf(n))$. $\square$

Another way to state the preceding observation is that, for any function $f(n)$ and any positive constant k,

$$O(f(n)) = O(k \cdot f(n))$$

In short, constant multiples of functions have the same order of magnitude as each other, so there is never a need to multiple a function by a constant greater than 1.

Remark. You will see in almost every context that authors will write statements such as

$$T(n) = O(f(n))$$

or

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

to mean that the function $T(n)$ is big-O of $f(n)$ or that $g(n)$ is big-O of $f(n)$. The use of the equality symbol is misleading and technically incorrect. The notation $O(f(n))$ is actually describing a set. It is the set of all functions $g(n)$ such that there are constants c and n_0 such that $g(n) \leq c \cdot f(n)$ for $n \geq n_0$. A better way to write the preceding statements, one that does not mislead you, is

$$T(n) \in O(f(n))$$

or

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

If you realize that $O(f(n))$ is a set of functions that grow no faster than $f(n)$, then it will make the following statements easy to understand.

1. If $g(n) \in O(f(n))$, then if $h(n) \in O(f(n))$ it is also true that $h(n) \in O(f(n) + g(n))$. Another way to say this is that adding a slower growing function to $f(n)$ does not change the set of functions that grow no faster than it.
2. A corollary is that adding any constant to$f(n)$ does not change the set of functions in $O(f(n))$, since all constants are $O(f(n))$ for any function $f(n)$. (Why?)
3. For any constant c, $O(c) = O(1)$. This is the set of constant functions.
4. $O(f(n)) \bigcup O(g(n)) = O(f(n) + g(n))$. In other words, the set of functions that grow no faster than the sum of f(n) and g(n) is the union of the sets of functions that grow no faster than each. (Why?)

4.1 A Hierarchy of Functions

We can define a binary operation on two orders of magnitude, $O(f(n))$ and $O(g(n))$ as follows. We will write

$$O(g(n)) < O(f(n))$$

if and only if

$$O(g(n)) \subsetneq O(f(n))$$

In other words, if $O(g(n)) < O(f(n))$ then all functions that grow no faster than $g(n)$ grow no faster than $f(n)$, but there are functions that grow faster than $g(n)$ that do not grow faster than $f(n)$. For example, $O(n^2) < O(n^3)$. If you need a function that acts like a "witness" to this, notice that if $h(n) = 2n^3$ then $h(n) \in O(f(n))$ but $h(n) \notin O(g(n))$. It should be clear that any function that is in $O(n^2)$ must be in $O(n^3)$. (Why?)

More generally, $O(n^k) < O(n^m)$ whenever $k < m$. This can also be proved easily.

This "less-than" relation is a transitive relation. The following is a hierarchy of functions of successively greater rate of growth:

$$O(1) < O(\log_2(n)) < O(\log^c(n)) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(n^b) < O(2^n) < O(3^n) < \cdots < O(n!)$$

where c is any number greater than 1 and b is any number greater than 3.

4.2 Practical Matters

You will find, when we start to analyze the various algorithms that we will encounter, that certain rates of growth will be prevalent, among which are

$O(\log n)$**.** Algorithms such as binary search have ***logarithmic*** running time.

$O(n)$**.** An algorithm that whose running time is $O(n)$ is said to have ***linear*** running time.

$O(n \log n)$**.** Algorithms that use divide and conquer strategies often have running times that are $O(n \log n)$. We will see that certain sorting algorithms have worst case behavior that is $O(n \log n)$.

$O(n^2)$. An algorithm whose running time is $O(n^2)$ is said to have ***quadratic*** running time. Many inefficient sorting algorithms have quadratic running time in their worst cases.

$O(c^n)$**.** Algorithms whose running time is $O(c^n)$ for some $c > 1$have ***exponential*** running time. There are classes of problems for which it is generally believed that any algorithm that solves them must have exponential running time. Such problems are called ***intractable*** problems.

5 Application Example: Efficiency of Search Algorithms

Unless otherwise stated, the size of the input to all problems will be denoted by n. For search algorithms, this means that they search through a container (an array, a vector, a list, etc.) that contains n items.

5.1 Linear Search

Linear search is a method of searching a list of key-value pairs one after the other, starting at the beginning of the list, until either the key is found or the end of the list is reached. It does not matter whether the list is a linked list or an array; the algorithm is the same. The only difference is in how the elements are accessed and how the structure is traversed.

We can use the term *iterator* to refer to an object that, at any time, references a particular element in the structure. So, for example, if the list is stored in an array, then the array index will be the value of the iterator. If the list is stored in a linked list, then a node pointer will be the value of the iterator. With this in mind, linear search is of the form

```
list.iterator = list.reference_to_first_value;
while ( list.iterator refers to a valid item ) {
    if ( list.iterator.key matches search_key )
        return list.iterator.value;
    else
        list.iterator++;
}
return item_not_in_list;
```

Basically this starts at the beginning of the list and compares the search key to the key in the list. On a match it quits, otherwise it continues to the next item, until the entire list is searched.

What is the running time of this algorithm as a function of the list size, in the worst case?

The worst case is when no item in the list matches the key. In this case, it has to compare the search key to every item's key. Thus, for searches that fail it performs n comparisons and the worst case is therefore $O(n)$.

What about the average case running time?

The number of comparisons made if the search key matches the first item is 1, if it matches the second item, 2, if it matches the k^{th} item, k. If it is equally likely to stop at any of these nodes, or not be there at all, then the expected case running time is the average, which is $(1+2+3+\cdots+n)/(n+1) = n(n+1)/(2n+2) = n/2$, which is $O(n)$. We divide by $n+1$ instead of n because there are $n+1$ cases – matching any of the n list items, and not matching any of them.

5.2 Binary Search

Binary search can only be applied to data that is sorted and that is in a structure with efficient random access, such as an array. There is no advantage to performing binary search on a data structure that only provides sequential access, since each access requires traversing the structure repeatedly. Therefore we will assume the data is in an array. Binary search of an array is of the form

```
Let the search region be the entire array.
while the item is not found and the search region is not empty {
    let middle be the index at the middle of the search region;
    if the key is smaller than the middle
        make the search region the lower half and search in there;
    else if the key is larger than the middle
        make the search region the upper half and search in there;
    else
        return middle;
}
return item_is_not_found;
```

What is the worst case running time as a function of n?

The worst case will occur when the item is not in the array, or when, as luck will have it, it is not in a computed middle element until the search region is size 1. Each iteration of the loop divides the size of the search region in half, approximately. In the worst case, it stops when the search region reaches size 1. At that point , if the item is not found in the middle element, the search region will become empty (that is in the code details, which are omitted in the above pseudo-code), and the loop will exit. If it is found, the loop will exit anyway.

In each iteration, at least one key comparison is made. (It could be two, but we will ignore that fact for the moment.) How many iterations will it take for the size of the search region to become 1 if it is initially of size n, which we will first assume is a power of 2, and is repeatedly divided in half? The size starts out at $n = n/2^0$, then becomes $n/2$, then $n/(2^2)$, then $n/(2^3)$ and so on until it has been divided k times and $n/(2^k) = 1$, which implies that $n = 2^k$, or that $k = \log_2(n)$. Therefore, there are $\log_2(n) + 1$ iterations. Because we make a comparison for each value of k from 0 to $\log_2(n)$, we make $\log_2(n) + 1$ comparisons. If n is not a power of 2, k would not be an integer, and the more accurate answer is that we make $\lfloor \log(n) \rfloor + 1$ comparisons, or equivalently, $\lceil \log_2(n + 1) \rceil$ comparisons. This shows that the number of comparisons will be proportional to $\log_2(n)$ or that it is $O(\log_2 n)$. Because we ignore constants of proportionality, the fact that two comparisons might be made in each iteration does not affect this answer.

In conclusion, binary search is much faster than linear search, but it requires that the data be maintained in sorted order. Therefore, one has to factor in the cost of keeping the array sorted to have a meaningful comparison of efficiency.