Counting basic steps —and other things

We call a step of an algorithm a *basic step* if the time it takes to evaluate it (or execute it) does not depend on the size of the data on which it operates but is bounded above by some constant. We say that execution of a basic step takes *constant time*.

Suppose x and y are variables of type **int**. Then expressions (1) and (2) in the box to the right are basic steps. We don't know the exact time it takes to evaluate (1) and (2), but we do know that the time does not depend on the values of the operands and is bounded above by some number of microseconds, depending of course on the computer that is executing it.

```
(1) x + y
(2) x * y
(3) x + x*y
```

In fact, all basic operations on the primitive types are basic steps, including these:

casting, like `(int) 'g'` and `(int) 5.2` , and
relations like `5 < 3` and `'c' >= 'd'`

The definition of a basic step allows us to consider expression (3) as either one or two basic steps. This is done deliberately. We will see later on that because of the way in which we use basic steps, it won't matter which choice we make. This may confuse you at first, but go along with this idea for now. If you are asked to count the basic steps in an algorithm, if there is ambiguity, write down what you are considering the basic steps.

Consider the if-statement to the right. Evaluation of the if-condition is a basic step. Evaluation of `x+y` is a basic step, as is assignment of its value to `x`. Therefore, execution of this if-statement executes either 1 or 3 basic steps. But we could also consider the whole if-statement as a basic step, because, since `x` and `y` are **int** variables, the time to execute the if-statement is bounded above by some number of nanoseconds, no matter what the values of `x` and `y` are.

```
if (x < y) {
   x= x+y;
}
```

We calculate the number of basic steps in executing the loop with initialization that appears to the right, assuming that `n ≥ 0`.

```
// Store in s the sum of 1..n.
int s= 0;
for (int k= 1; k <= n; k++)
    s= s + k;
```

The basic step `s= 0;` is executed once.
The basic step `k= 1;` is executed once.
The basic step `k ≤ n` is evaluated `n+1` times (it is true `n` times and false once).
The basic step `k++` (which is shorthand for `k= k+1;`) is executed `n` times.
The basic step `s= s+k;` is executed `n` times.

We add these together and see that execution of this code requires `3n + 3` basic steps. The number of steps is *linear in* `n`; it is *proportional to* `n`.

Suppose we count the statement `s= s+k;` as two basic steps —the addition is one basic step and the assignment is the second basic step. Then, execution requires `4n+3` basic steps, not `3n+3`. The important point is that the number of basic steps is still linear in `n`, it is proportional to `n`.

Example of a nested loop

There is a tendency to look at nested loops in which both have `n` as the upper limit in the loop-condition and to immediately say that execution must require n^2 basic steps. One must be more careful and study the code to know what it is doing.

We look at an example in which the term `log n` arises. Remember that if $m = 2^n$, then `log m = n`. We use logarithms to the base 2. Below, assume `2 ≤ n`.

```
for (int k= 1; k <= n; k++) {
    t= 2;
    while (t <= n) {
        t= 2*t;
    }
}
```

In the code to the right, `k` is not used in the body of the outer loop. Therefore, the number of basic steps in executing the body is the same at each iteration.

When executing the outer-loop body, `t` takes on the values 2, 4, 8, …, 2^h where $2^h \le n < 2^{h+1}$. Write this as; $2^1, 2^2, \ldots, 2^h$ where $2^h \le n < 2^{h+1}$, and we see that the statement `t= 2*t;` is executed h = `floor(log n)` times. We calculate the number of basic steps in executing the body of the outer loop:

`t= 2;`	1 time
`t ≤ n`	1 + `floor(log n)` times (it is found false once)
`t= 2*t;`	`floor(log n)` times

Counting basic steps —and other things

In total, then, execution of the body executes `2 * floor (log n) + 2` basic steps.

We calculate the number of basic steps in executing the whole algorithm:

`k= 1;`	`1` time
`k ≤ n`	`n+1` times
`k++`	`n` times

basic steps in body of outer loop: `n * (2 floor (log n) + 2)` (since the body is executed `n` times)

We add these together and rearrange to get this many basic steps:

```
2n floor(log n) + 4n + 2
```

We would now say that the number of basic steps performed is proportional to `n log n`.

Valuable tip: In the box to the right, the value of `t` is doubled at each iteration until it gets greater than `n`. Whenever you see a loop that doubles a value (starting with 1 or 2) until it becomes greater than a value `n`, recognize immediately that the number of iterations is proportional to the base-2 log of `n`.

```
t= 2;
while (t <= n) {
    t= 2*t;
}
```

Counting only certain steps

The method body shown to the right returns the index of the first occurrence of `'c'` in `String s`, or –1 if `'c'` doesn't occur in `s`. Suppose `s[m]` is the first occurrence of `'c'`. The basic steps performed are given to the right, and we see that 3`m` + 4 basic steps are performed in this case.

The number comparisons with `'c'` is `m+1`.

Note that formulas 3`m` + 4 and `m+1` are both proportional to `m`, they are both *linear* in `m`. Therefore, if we are interested *only* in determining that the time of execution is proportional to `m`, we need to count only the comparisons with `'c'`.

```
for (int k= 0; k < s.length(); k= k+1)
    if (s.charAt(k) == 'c') return k;
return -1;
----------------------------------------
k= 0;                  1 time
k < s.length()         m+1 times:
k= k+1;                m times:
s.charAt(k) == 'c'     m+1 times
return k;              1 time:
```

With experience, we can eyeball such a loop, determine that the important point is the number of comparisons with `'c'`, which is `m` +1 (if `s[m]` is the first occurrence of `'c'`), understand that nothing else in the method body makes the time worse than linear in `m`, and stop further calculations. We will be doing this often in the future.

Executing a method call

Executing a method call `p(…)` consists of

(1) Pushing a frame for the call on the call stack. This takes constant time.
(2) Evaluating the arguments and storing their values in the parameters. Time depends on the arguments.
(3) Executing the method body. This is at least constant time.
(4) Popping the frame for the call off the stack and, if a function, pushing the return value on the call stack. This takes constant time.

Suppose evaluating the arguments takes constant time. Then, as a simplification, in considering a method call like `s.charAt(n)` or `s.indexOf('c')`, we can count (1) and (2) and (4) together as a single basic step, so the number of basic steps is 1 + (the number of basic steps in executing the method body). We can also simply throw away the 1 and just consider the number of basic steps in executing the method body.

Here are three examples:

`s.charAt(n)`: a basic step, taking constant time, since referencing element `n` of a **char**[] takes constant time.

`s.length()`: a basic step, taking constant time.

`s.indexOf('c')`: We can envision that, for this call, the body of method `indexOf` is equivalent to the code in the box above. Assume that the first occurrence of `'c'` in `s` is `s[m]`. Based on the discussion in the previous section *Counting only certain steps*, we know both the number of basic steps in this call and the number of comparisons with `'c'` is linear in `m`. Having the code in a method call instead of inline does not change this.