## Measuring an Input's Size

We measure an algorithm's input size as a function of some parameter $n$.

For measuring input sizes that are just one number, and the numbers magnitude that determines the input size. It is usually preferable to measure size by the number $b$ of bits in the $n$'s binary representation:

$$b = \lfloor \log_2 n \rfloor + 1$$

## Units for Measuring Running Time

One approach of a metric for measuring an algorithm's Running time is to identify the **basic operation**. The basic operation is the operation contributing the most to the total running time, and compute the number of times the basic operation is executed.

The framework for analysis of an algorithm's time efficiency suggests measruing it by counting the number of times the algorithm's basic operation is executed on inputs of size $n$.

Let $c_{op}$ be the execution time of an algorithm's basic operation on a particular computer, and let $C(n)$ be the number of times this operation needs to be executed for the algorithm. Then we can estimate the running time of a program implementing this algorithm on that computer by the formula

$$T(n) \approx c_{op}\, C(n)$$

$C(n)$ does not contain any information about operations that are not basic. $c_{op}$ is also an approximation. Unless $n$ is very large or very small, the formula can give a reasonable estimate of the algorithm's running time. It also makes it possible to answer such questions as "How much faster would this algorithm run on a machine that is 10 times faster than the one we have?"

Let $C(n) = \frac{1}{2}n(n-1)$

$$\begin{aligned} C(n) &= \frac{1}{2}n(n-1) \\ &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &\approx \frac{1}{2}n^2 \end{aligned}$$

and therefore

$$\begin{aligned} \frac{T(2n)}{T(n)} &\approx \frac{c_{op}C(2n)}{c_{op}C(n)} \\ &\approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} \\ &= 4 \end{aligned}$$

## Orders of Growth

| $n$ | $\log_2 n$ | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| 10 | 3.3 | $10^1$ | $3.3 \cdot 10^1$ | $10^2$ | $10^3$ | $10^3$ | $3.6 \cdot 10^6$ |
| $10^2$ | 6.6 | $10^2$ | $6.6 \cdot 10^2$ | $10^4$ | $10^6$ | $1.3 \cdot 10^{30}$ | $9.3 \cdot 10^{157}$ |
| $10^3$ | 10 | $10^3$ | $1.0 \cdot 10^4$ | $10^6$ | $10^9$ | | |
| $10^4$ | 13 | $10^4$ | $1.3 \cdot 10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | 17 | $10^5$ | $1.7 \cdot 10^6$ | $10^{10}$ | $10^{15}$ | | |
| $10^6$ | 20 | $10^6$ | $2.0 \cdot 10^7$ | $10^{12}$ | $10^{18}$ | | |

**Worst-Case, Best-Case, and Average Case Efficiencies**

```
SequentialSearch($A[0 \ldots n-1], K$)
/* Searches for a given value in a given array by sequential search
* Input: An array $A[0 \ldots n-1]$ and a search key $K$
* Output: The index of the first element in $A$ that matches $K$
* or $-1$ if there are no matching elements */

$i \leftarrow 0$
while $i < n$ and $A[i] \neq K$ do
        $i \leftarrow i + 1$
if $i < n$ return $i$
else return $-1$
```

In the worst case, when there are no matching elements or the first matching element happens to be the last one on the list, the algorithm makes the largest number of key comparisons among all possible inputs of size $n : C_{worst}(n) = n$

The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size $n$, which is an input of size $n$ for which the algorithm runs the longest among all possible inputs of that size. The way to determine the worst-case efficiency of an algorithm is straightforward: anaylze the algorithm to see what kind of inputs yield the largest value of the basic operation's count $C(n)$ among all possible inputs of size $n$ and then compute this worst-case value $C_{worst}(n)$. The worst-case analysis bounds an algorithm's running time from above.

The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size $n$, which is an input of size $n$ for which the algorithm runs the fastest among all possible inputs of that size. We can analyze the best-case efficiency by first, determining the kind of inputs for which the count $C(n)$ will be the smallest among all possible inputs of size $n$. Then we find the value of $C(n)$ on these most convient inputs.

The **average-case efficiency** yields information about an algorithm's behvaior on a "typical" or "random" input. To analyze the algorithm's average-case efficiency, we must make some assumptions about possible inputs of size $n$:

(a) the probability of a successful search is equal to $p (0 \leq p \leq 1)$ and

(b) the probability of the first match occurring in the $i$th position of the list is the same for every $i$.

Under these assumptions, we can find the average number of key comparisons $C_{avg}(n)$. In this case the probability of a successful search is $\frac{p}{n}$ for every $i$, and the number of comparisons made by the algorithm in such a situation is $i$. In the case of an unsucessful search, the number of comparisons will be $n$ with the probability being $(1-p)$. Therefore

$$\begin{aligned} C_{avg}(n) &= \left[1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}\right] + n \cdot (1-p) \\ &= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p) \\ &= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) \\ &= \frac{p(n+1)}{2} + n(1-p) \end{aligned}$$

This general formula yields reasonable answers. For example if $p = 1$ (the search is successful), the average number of key comparisons made by the sequential search is $\frac{(n+1)}{2}$; that is, the algorithm will inspect, on average, about half the elements. If $p = 0$ (the search is unsucessful), the average number of key comparisions will be $n$ because the algorithm will inspect all $n$ elements on all such inputs.

**Recapitulation of the Analysis Framework**

- Both time and space efficiencies are measured as functions of the algorithm's input size.

- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.

- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and the best-case efficiencies.

- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.