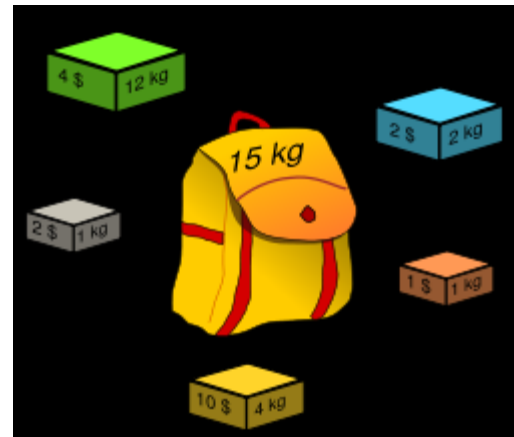


## O Notation (Big Oh)

- We want to give an *upper bound* on the amount of time it takes to solve a problem. **Why would we settle for an upper bound instead of a tight bound?**
- Definition:  $v(n) = O(f(n)) \leftrightarrow \exists$  constant  $c$  and  $n_0$  such that  $|v(n)| \leq c|f(n)|$  whenever  $n \geq n_0$ .
  - The growth rate of  $T(N)$  is less than or equal to that of  $F(N)$
- **Is an  $O(n^2)$  algorithm always better than an  $O(n^3)$  algorithm?**
  - No, it depends on the specific constants. For small  $n$ ,  $c$  may dominate.
  - But if  $n$  is large enough, an  $O(n^3)$  algorithm is always slower than an  $O(n^2)$  algorithm. **Can you prove that?**
- The different complexity classes are (in order):  $O(1)$  (constant),  $O(\log n)$  (logarithmic),  $O(n)$  (linear),  $O(n \log n)$  ( $n \log n$ ),  $O(n^2)$  (quadratic),  $O(n^3)$  (cubic),  $O(2^n)$  (exponential),  $O(n!)$  (factorial).
- Intractable – all known algorithms to solve a problem are of exponential complexity.
- **What kinds of problems have an exponential solution?** What about the knapsack problem? *Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.*



*For iterative algorithms, it is often easy to see the complexity – but recursive algorithms are a bit more difficult.*

## Recursive Algorithms often have the same general forms

$T(n) = aT(n/b) + O(n^k)$  - each call reduces the problem size by a factor of  $b$  and makes “ $a$ ” recursive calls. The last term indicates the amount of work per call. (as a function of a power of  $n$ ). Note, in terms of complexity, we don’t care whether the work is done before or after (or in between) the recursive calls.

```
void doit(int n)
{ if (n==1) return;
  for (i=0; i<n; i++)
    x = x + i;
  doit(n/2);
  doit(n/2);
}
```

```
void doit(int n)
{ if (n==1) return;
  for (i=0; i<n; i++)
    x = x + i;
  doit(n/2);
}
```

```
void doit(int n)
{ if (n==1) return;
  x++;
  doit(n/2);
  doit(n/2);
}
```

```
void doit(int n)
{ if (n==1) return;
  for (i=0; i<n; i++)
    x = x + i;
  doit(n-1);
}
```

## A Formula Approach

- Mathematicians have developed a formula approach to determine complexity.
- Theorem: Assume  $T(n) = aT(n/b) + O(n^k)$  is the time function.
  - If  $a > b^k$ , the complexity is  $O(n^{\log_b a})$ .
  - If  $a = b^k$ , the complexity is  $O(n^k \log n)$ .
  - If  $a < b^k$  the complexity is  $O(n^k)$ .
- Nice as it reinforces what we compute by drawing AND solves additional problems.
- Note, only divide and conquer type of problems work for this method.

## Determining Complexity from Experimental Evidence

n	T(n)
2	10
4	10
8	10
16	11
32	8

n	T(n)
2	10
4	17
8	32
16	66
32	130

n	T(n)
2	10
4	10
8	15
16	20
32	25

n	T(n)
2	8
4	32
8	500
16	65,600
32	$4 \times 10^9$

n	T(n)
2	2
4	8
8	25
16	64
32	161

- How do you figure out the complexity from experimental data when you know neither the constant nor the complexity? I “eyeball it” to come up with a good guess. Then I figure out the constant for one entry. Next I see if that constant works for all the entries. This is basically my “approximate-then-finalize” approach. How do I guess?
  - $O(1)$  is basically constant.
  - $O(\log n)$  grows slowly, by a constant between entries.
  - $O(n)$  doubles between entries.
  - $O(n \log n)$  slightly more than doubles between entries .
  - $O(n^2)$  quadruples between entries.
  - $O(2^n)$  grows exponentially .

Another way to check if the complexity is correct is to divide the experimental time by the complexity. Below represents three attempts at finding the complexity. The first is an overestimate. Notice how the ratio goes to zero. The second is correct, showing the constant is approximately 4.3. The third is an underestimate. Notice how the ratio increases.

n	Actual time	$n^3$	ratio	$n^2$	ratio	$n \log n$	ratio
4	75	64	1.17	16	4.7	8	9.4
8	219	512	0.43	64	3.4	24	9.1
16	1104	4096	0.27	256	4.3	64	17.2
32	4398	32768	0.13	1024	4.3	160	27.5
64	17632	262144	0.07	4096	4.3	384	45.9