In [1]:

```
%%html
<style>
    p,ul {

          font-size: 18px;
          font-family: "Segoe Print","Georgia", "Verdana", "Lucida Console", "Courier N
ew";
          line-height: 150%;
    }

    .text_cell_render p {
    text-align: justify;
    text-justify: inter-word;
    }
</style>
```

In [4]:

```
from IPython.display import Image
```

Standard Recurrence relationships

In [3]:

```
Image(filename = "srf.png", width = "100%")
```

Out[3]:

Standard Recurrence Format

Base case: $T(n)$ is at most a constant for all sufficiently small n.[4]

General case: for larger values of n,

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + O(n^d).$$

Parameters:

- $a =$ number of recursive calls
- $b =$ input size shrinkage factor
- $d =$ exponent in running time of the "combine step"

- The base case of a standard recurrence asserts that once the input size is so small that no recursive calls are needed, the problem can be solved in $O(1)$ time.
- The general case assumes that the algorithm makes a recursive calls, each on a subproblem with size a b factor smaller than its input, and does $O(n^d)$ work outside these recursive calls.
- In general, a can be any positive integer, b can be any real number bigger than 1 (if $b \leq 1$ then the algorithm won't terminate), and d can be any nonnegative real number, with $d = 0$ indicating only constant ($O(1)$) work beyond the recursive calls.
- Parameters a, b, and d should be constants—numbers that are independent of the input size n.
- One restriction in standard recurrences is that every recursive call is on a subproblem of the same size.
- For example, an algorithm that recurses once on the first third of an input array and once on the rest would lead to a non-standard recurrence.
- Most (but not all) natural divide-and-conquer algorithms lead to standard recurrences.

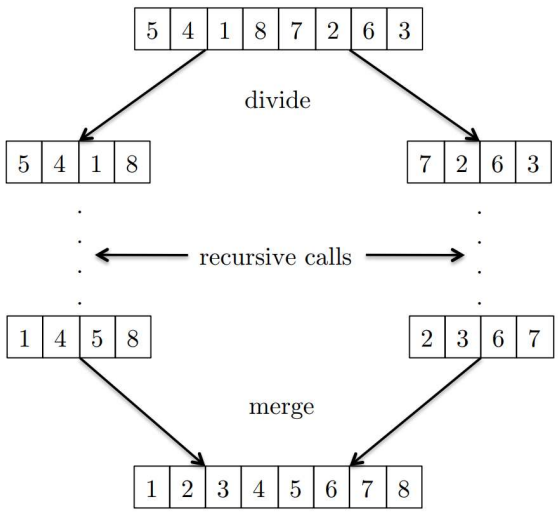
Merge Sort

- MergeSort is an ideal introduction to the divide-and-conquer paradigm.
- The basic idea is to break the problem into smaller subproblems, solve the subproblems recursively, and finally combine the solutions to the subproblems into one for the original problem.

In [5]:

```
Image(filename = "merge.png", width = "50%")
```

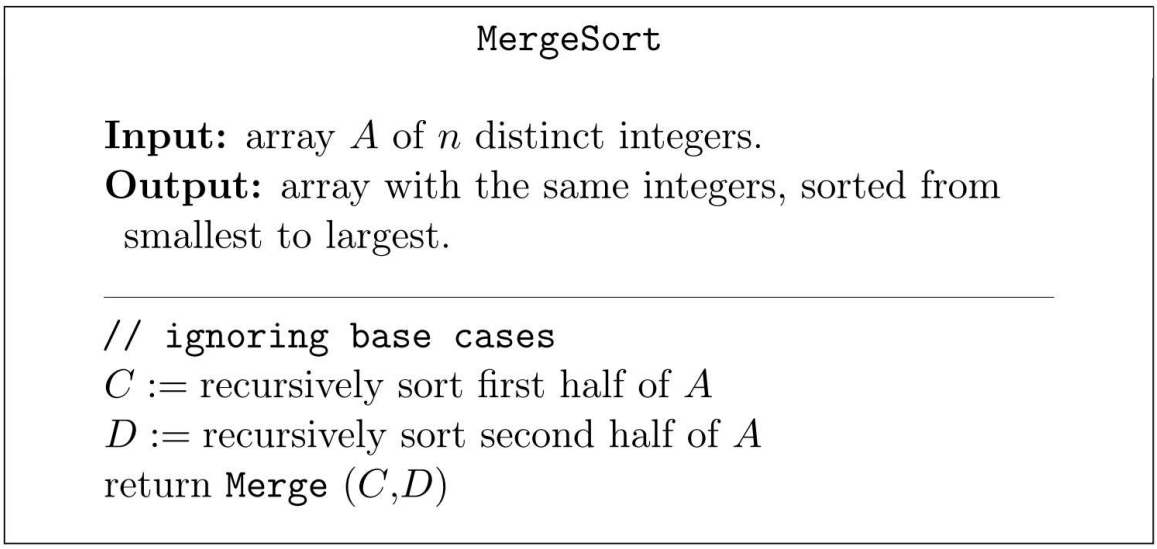
Out[5]:



In [7]:

```
Image(filename = "merge2.png", width = "100%")
```

Out[7]:



Merge algorithm

In [8]:

```
Image(filename = "merge3.png", width = "100%")
```

Out[8]:

Merge

Input: sorted arrays C and D (length $n/2$ each).
Output: sorted array B (length n).
Simplifying assumption: n is even.

```
i := 1
j := 1
for k := 1 to n do
    if C[i] < D[j] then
        B[k] := C[i]          // populate output array
        i := i + 1                          // increment i
    else                                     // D[j] < C[i]
        B[k] := D[j]
        j := j + 1
```

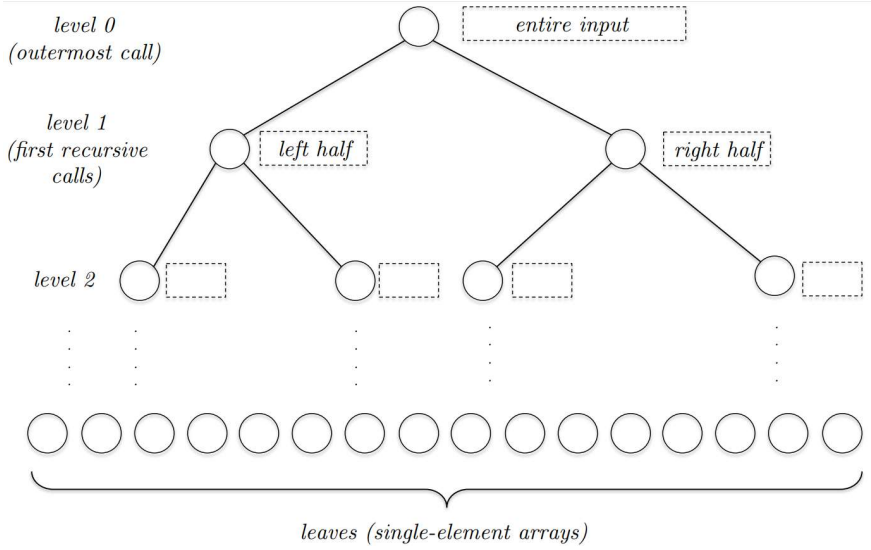
- The Merge subroutine performs at most $4l + 2$ operations to merge two sorted arrays of length $l/2$ each.
- For $l \geq 1, 4l + 2 \leq 6l$. That is, $6l$ is also a valid upper bound on the number of operations performed by the Merge subroutine.
- For every pair of sorted input arrays C, D of length $l/2$, the Merge subroutine performs at most $6l$ operations.

Running time of Merge Sort

In [5]:

```
Image(filename = "merge1.png", width = "75%")
```

Out[5]:



The total work done by level-j recursive calls (not counting later recursive calls) as

In [6]:

```
Image(filename = "merge4.png", width = "75%")
```

Out[6]:

$$\underbrace{\#\text{ of level-}j\text{ subproblems}}_{=2^j} \times \underbrace{\text{work per level-}j\text{ subproblem}}_{=6n/2^j}.$$

Thus atmost $6n$ operations are performed across all the recursive calls at the $j-$th recursion level.

The recursion tree has $log_2 n + 1$ levels (levels 0 through $log_2 n$, inclusive). Using our bound of $6n$ operations per level, we can bound the total number of operations by

In [7]:

```
Image(filename = "merge5.png", width = "75%")
```

Out[7]:

$$\underbrace{\text{number of levels}}_{=\log_2 n+1} \times \underbrace{\text{work per level}}_{\leq 6n} \leq 6n \log_2 n + 6n$$

Master Method

- It is used for analyzing recursive algorithms.
- It takes as input the recurrence relationship for the time complexity of an algorithm and produces as output an upper bound on the running time of the algorithm.
- If $T(n)$ is defined by a standard recurrence, with parameters $a \geq 1$, $b > 1$, and $a \geq 0$, then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad [\text{Case 1}] \\ O\left(n^d\right) & \text{if } a < b^d \quad [\text{Case 2}] \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \quad [\text{Case 3}]. \end{cases}$$

Using Master Method for analysis of Merge Sort algorithm

- In MergeSort, there are two recursive calls, so a = 2.
- Each recursive call receives half of the input array, so b = 2 as well.
- The work done outside these recursive calls is dominated by the Merge subroutine, which runs in linear time, and so d = 1. Thus the parameters fulfill the first case of the master method.

$$a = 2 = 2^1 = b^d,$$

- Plugging in the parameters, gives us the running time of MergeSort as $O(n^d \log n) = O(n \log n)$.

Using Master Method for analysis of Binary Search algorithm

- Binary search recurses on either the left half of the input array or the right half (never both), so there is only one recursive call ($a = 1$).
- The recursive call is on half of the input array, so b is again equal to 2.
- Outside the recursive call, all binary search does is a single comparison (between the middle element of the array and the element being searched for) to determine whether to recurse on the left or the right half of the array. This translates to $O(1)$ work outside the recursive call, so $d = 0$.
- Since $a = 1 = 2^0 = b^d$, we are again in the first case of the master method, and we get a running time bound of $O(n^d log n) = O(log n)$.

Proof of the Master Method

Base Case: for $n = 1$

$$T(n) \leq c$$

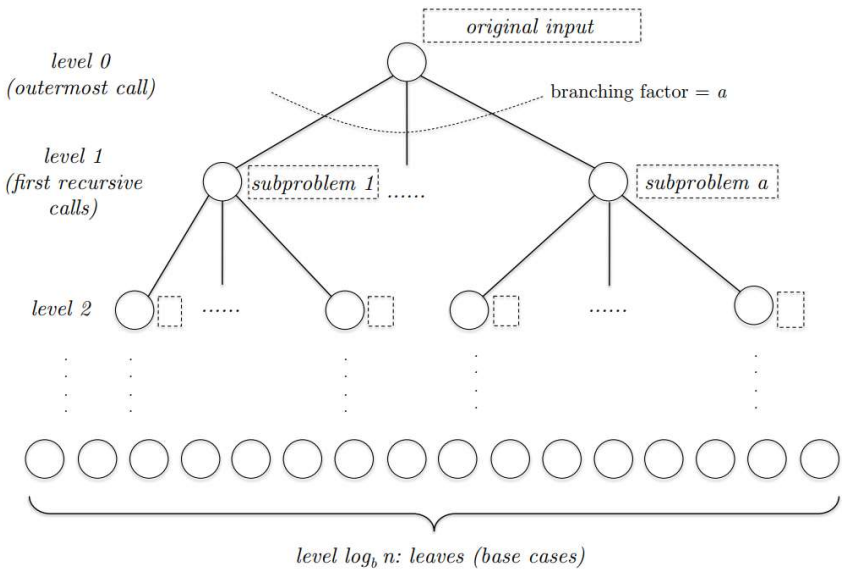
General Case: for $n > 1$

$$T(n) \leq a \cdot T\left(\frac{n}{b}\right) + cn^d.$$

In [8]:

```
Image(filename = "rtg.png", width = "75%")
```

Out[8]:



There are a^j different subproblems at level j, each with an input size of n/b^j

In [10]:

```
Image(filename = "wpl.png", width = "65%")
```

Out[10]:

$$\text{work at level } j \leq \underbrace{a^j}_{\#\text{ of subproblems}} \cdot \overbrace{c \cdot \underbrace{\left\lceil \frac{n}{b^j} \right\rceil}_{\substack{\text{input}\\ \text{size}}}{}^{d}}^{\text{work per subproblem}}$$

In [11]:

```
Image(filename = "wpls.png", width = "35%")
```

Out[11]:

$$\text{work at level } j \leq cn^d \cdot \left[\frac{a}{b^d}\right]^j.$$

Summing over all the levels $j = 0, 1, 2, \ldots, log_b n$ we obtain the following upper bound on the running time (using that n^d is independent of j): and can be

In [12]:

```
Image(filename = "tw.png", width = "35%")
```

Out[12]:

$$\text{total work} \leq cn^d \cdot \sum_{j=0}^{\log_b n} \left[\frac{a}{b^d}\right]^j.$$

Evil is represented by a, the rate of subproblem proliferation (RSP)—with every level of recursion, the number of subproblems explodes by an a factor, and this is a little scary.

Good takes the form of b^d, the rate of work shrinkage (RWS)—the good news is that with every level of recursion, the amount of work per subproblem decreases by a factor of b^d

The three cases of the master method correspond exactly to the three possible outcomes of this tug-of-war: a draw (RSP = RWS), a victory for good (RSP < RWS), or a victory for evil (RSP > RWS).

There are three fundamentally different types of recursion trees—with the workper-level staying the same, decreasing, or increasing—and the relative sizes of a (the RSP) and b^d (the RWS) determine the recursion tree type of a divide-and-conquer algorithm.

Case-I

- Consider the first case, when $a = b^d$ and the algorithm performs the same amount of work at every level of its recursion tree.
- We certainly know how much work is done at the root, in level 0 — $O(n^d)$, as explicitly specified in the recurrence.
- With $O(n^d)$ work per level, and with $1 + log_b n = O(logn)$ levels, we should expect a running time bound of $O(n^d logn)$ in this case

In [13]:

```
Image(filename = "case_one.png", width = "70%")
```

Out[13]:

$$cn^d \cdot \underbrace{\sum_{j=0}^{\log_b n} \underbrace{\left[\underbrace{\frac{a}{b^d}}_{=1}\right]^j}_{=1 \text{ for each } j}}_{} = cn^d \cdot \underbrace{(1 + 1 + \cdots + 1)}_{1 + \log_b n \text{ times}},$$

which is $O(n^d \log n)$.[17]

Case-II

- In the second case, $a < b^d$ and the forces of good are victorious— the amount of work performed is decreasing with the level.
- Thus more work is done at level 0 than at any other level.
- The simplest and best outcome we could hope for is that the work done at the root dominates the running time of the algorithm.
- Since $O(n^d)$ work is done at the root, this best-case scenario would translate to an overall running time of $O(n^d)$
- Set $\frac{a}{b^d}$; since a, b, and d are constants (independent of the input size n), so is r.
- The upper bound becomes:

In [17]:

```
Image(filename = "case_two.png", width = "35%")
```

Out[17]:

$$cn^d \cdot \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O(1)} = O(n^d),$$

Case-III

- In the third case, when subproblems proliferate even faster than the work-per-subproblem shrinks, the amount of work performed is increasing with the recursion level, with the most work being done at the leaves of the tree.
- Again, the simplest- and best-case scenario would be that the running time is dominated by the work done at the leaves.
- A leaf corresponds to a recursive call where the base case is triggered, so the algorithm performs only O(1) operations per leaf.
- We know that there are a^j nodes at each level j.
- The leaves are at the last level $j = log_b n$, so there are $a^{log_b n}$ leaves. Thus, the best-case scenario translates to a running time bound of $O(a^{log_b n})$.

In [22]:

```
Image(filename = "logi.png", width = "55%")
```

Out[22]:

$$\underbrace{a^{\log_b n}}_{\text{more intuitive}} = \underbrace{n^{\log_b a}}_{\text{easier to apply}}$$

In [21]:

```
Image(filename = "case_three.png", width = "65%")
```

Out[21]:

$$cn^d \cdot \underbrace{\sum_{j=0}^{\log_b n} r^j}_{=O(r^{\log_b n})} = O(n^d \cdot r^{\log_b n}) = O\left(n^d \cdot \left(\frac{a}{b^d}\right)^{\log_b n}\right).$$

Geometric Series

- Expressions of the form $1 + r + r^2 + \cdots + r^k$ for some real number r and nonnegative integer k are called Geometric Series
- For us, r will be the critical ratio $\frac{a}{b^d}$
- When $r \neq 1$, there is a useful closed-form formula for a geometric series:

$$1 + r + r^2 + \cdots + r^k = \frac{1 - r^{k+1}}{1 - r}$$

- when $r < 1$,

$$1 + r + r^2 + \cdots + r^k \leq \frac{1}{1 - r}$$

- Thus every geometric series with r < 1 is dominated by its first term — the first term is 1 and the sum is only O(1). For example, it doesn't matter how many powers of 1 2 you add up, the resulting sum is never more than 2.
- when $r > 1$,

$$1 + r + r^2 + \cdots + r^k = \frac{r^{k+1} - 1}{r - 1} \leq \frac{r^{k+1}}{r - 1} = r^k \cdot \frac{r}{r - 1}$$

- Thus every geometric series with $r > 1$ is dominated by its last term — the last term is r^k while the sum is at most a constant factor ($\frac{r}{r-1}$) times this. For example, if you sum up the powers of 2 up to 1024, the resulting sum is less than 2048.