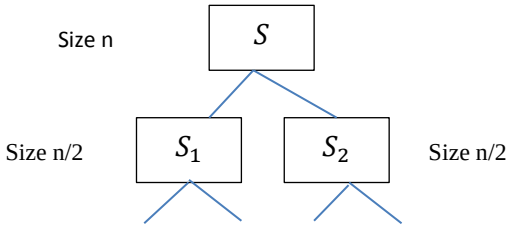
Recurrences – Contd.

Merge Sort problem:

Given $< a_1, a_2, \dots, a_n >$, we would like to sort them to a new sequence: $< a'_1 \le a'_2 \le \cdots \le a'_n >$

The algorithm is a **divide and conquer** algorithm:

- divide the problem into 2 subprobelms half the size: $S \to S_1, S_2, |S_1| \cong |S_2| \cong \frac{|S|}{2}$
- Solve the sub problems recursively.
- Merge the results back up.



The merge function is simple:

- Look at the two sorted lists, each time advancing with the one with the smallest value at top.
- Stop when got to the end of the two lists.

The recurrence of the **Merge-Sort** algorithm is:

Substitution method:

$$T(n) = 2T\left(\frac{n}{2}\right) + n = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = 2^2 T\left(\frac{n}{2^2}\right) + 2n = \cdots = 2^k T\left(\frac{n}{2^k}\right) + kn \underset{\frac{n}{2^k}=1 \Rightarrow k=\lg n}{=} 2^{\lg n} T(1) + n \cdot \lg n$$

$$= \Theta(n \cdot \lg n)$$

Note: The reason log bases don't matter is: $\lg_b n = \frac{\lg_c n}{\lg_c b} = \underbrace{\frac{1}{\lg_c b}}_{constant} \cdot \lg_c n \Rightarrow \lg_b n = \Theta(\lg_c n)$

Recursion Tree method:

Level 0: n - $cost = n$

Level 1: $T\left(\frac{n}{2}\right)$, $T\left(\frac{n}{2}\right)$, $cost = n$

Level 2: $T\left(\frac{n}{4}\right), T\left(\frac{n}{4}\right), T\left(\frac{n}{4}\right), T\left(\frac{n}{4}\right)$, $cost = n$

…

Until the bottom of the tree which has $\frac{n}{2}$ **pairs**.

The **total work** is $n \times (depth\ of\ recursion\ tree) = c \cdot \lg n$

Generalized Form:

$$\boxed{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

The recursion tree would look like:

Level 0: $T(n)$. Cost to go out: $f(n)$

Level 1: a problems of size $T\left(\frac{n}{b}\right)$. To go up would cost: $f(n)$

Level 2: a^2 problems of size $T\left(\frac{n}{b^2}\right)$. To go up would cost: $a \cdot f\left(\frac{n}{b}\right)$

Level 3: a^3 problems of size $T\left(\frac{n}{b^3}\right)$. To go up would cost $a^2 f\left(\frac{n}{b^2}\right)$ [this level costs $a^3 f\left(\frac{n}{b^3}\right)$

The last level has a^k problems, each of size $\frac{n}{b^k}$. To go up you pay: $a^{k-1} f\left(\frac{n}{b^{k-1}}\right)$

At the **last** layer each operation is trivial, as there are n nodes with $\Theta(1)$ work.

The depth of the tree is $\boxed{\lg_b n}$ because each layer you divide the problem into problems of size divided by b. After $\lg_b n$ steps you get to a problem of size 1.

To go from the bottom layer back up, costs: $a^l \cdot f(1)$ where $a^l = a^{\lg_b n} = n^{\lg_b a}$

When $a = b$ the cost to come up from one level to the one above it is n.

But, if for instance $a = 4, b = 2$ the work per level would be $n^{\lg 4} = n^2$ (!) so if $a \geq b$ you number of nodes is greater than linear, and the problem is broken down to sub-problems inefficiently.

The **work** is done only when you get to the trivial case and fold the tree back up:

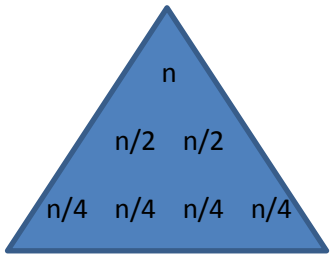
$$\boxed{T(n) = n^{\lg_b a} + \sum_{k=0}^{(\lg_b n)-1} a^k \cdot f\left(\frac{n}{b^k}\right)}$$

The bound of the sum is $\lg_b n - \mathbf{1}$ because we don't care about the bottom level (that's already calculated by $n^{\lg_b a} \cdot T(1) = n^{\lg_b a}$).

The sum is actually measuring the **area** of the triangle that is formed from expanding the recursion tree.

In the case of merge-sort:

The triangle is:



Binary search: $T(n) = T\left(\frac{n}{2}\right) + 1$

Linear search: $T(n) = T(n-1) + 1$

What about the sum of tree for $T(n) = 4T\left(\frac{n}{2}\right) + n$:

$$T(n) = n^2 + 4^0 \cdot \frac{n}{2^0} + 4^1 \cdot \frac{n}{2^1} + 4^2 \cdot \frac{n}{2^2} + \cdots + 4^{\lg n - 1} \cdot \frac{n}{2^{\lg n}} = n^2 + n\left(1 + 2 + 2^2 + 2^3 + \cdots + \underbrace{2^{\lg n}}_{=n}\right) = n^2 + n \cdot (n-1) =$$

$$= 2n^2 - n = \boxed{\Theta(n^2)}$$

Note that the work at the bottom level (the trivial level) is n^2 and that's the same work you pay going up – the triangle costs also n^2 (minus n).

For $T(n) = T\left(\frac{n}{2}\right) + n^2 = T\left(\frac{n}{4}\right) + \frac{n^2}{2^2} + n^2 = \cdots = T\left(\frac{n}{2^k}\right) + \left(\frac{n}{2^k}\right)^2 + \cdots + n^2 = T\left(\frac{n}{2^k}\right) + n^2 \sum_{k=0}^{\lg n - 1} \left(\frac{1}{2^k}\right)^2 = c \cdot n^2$

So there are three cases:

- $a = b$: a triangle
- $a > b$: all work is at the bottom of the tree
- $b > a$: all work is at the beginning of the tree

And that's the **Master Theorem**.

Master Theorem:

Given recurrence $T(n) = \begin{cases} \Theta(1), & n = 1 \\ aT\left(\frac{n}{b}\right) + f(n), & n = b^k \end{cases}$

We'll define the following for the 3 instances:

- The work at the bottom of the tree is: $Q(n) \coloneqq n^{\lg_b a}$ – number of the leaves of the tree.
- The work of the triangle is: $\sum_{k=0}^{\lg_b n} a^k f\left(\frac{n}{b^k}\right)$

The cases:

1) If $f(n) = O\left(n^{\lg_b a - \epsilon}\right), \epsilon > 0$, that is $f(n)$ is **polynomially smaller than** $n^{\lg_b a}$. Thus: $\boxed{T(n) = \Theta\left(n^{\lg_b a}\right)}$
2) If $f(n) = \Theta\left(n^{\lg_b a}\right)$, that is $f(n)$ is equivalent to $n^{\lg_b a}$. That's the cost of the bottom times the height of the triangle. Thus $\boxed{T(n) = \Theta\left(n^{\lg_b a}\right)}$
3) If $f(n) = \Omega\left(n^{\lg_b a + \epsilon}\right), \epsilon > 0$, **and** $n \geq b \Rightarrow a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ $(c \geq 0)$, that is $f(n)$ is **polynomially larger than** $n^{\lg_b a}$. Thus $\boxed{T(n) = \Theta\left(f(n)\right)}$

Example for (2): $T(n) = 4T\left(\frac{n}{2}\right) + n^2 \Rightarrow T(n) = O(n^2 \lg n)$

Example for (3): $T(n) = 2T\left(\frac{n}{4}\right) + n^2\sqrt{n} \Rightarrow Q(n) = n^{\lg_4 2} = \sqrt{n}, f(n) = n\sqrt{n} \Rightarrow T(n) = \Theta\left(n\sqrt{n}\right)$

Note: (1) and (2) are Θ for some relation between a and b.

Back to Algorithms:

Quick Sort:

After every partition you have a pivot that you know all the left side of it is less than the pivot, and the right – bigger than the pivot. That means that every step you "earn" 1 bit of information.

The recurrence for quick sort is:

$T(n) = T(q) + T(n - q) + \Theta(n)$

- The $\Theta(n)$ at the level is for the partition of the array into one side $\leq x$ and the other side $> x$ (x is the pivot).
- The partition would be into q array and $n - q$ array that are sorted recursively.

The case of $q = n - q = \frac{n}{2}$ is good, but the **worst case** can be **insertion sort**: $T(n) = T(1) + T(n - 1) + n$, that would end up being $T(n) = \Theta(n^2)$.

And the case of $k = \frac{n}{2}, n - k = \frac{n}{2}$:

$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \lg n)$

This is preferable over **merge-sort** since it **sorts in place**, unlike merge sort. The space complexity of quick sort is better than that of merge sort.

How about $T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$ – the work tree will be **unbalanced** and the deepest side would be $\lg_{10} n$. that's why the order of growth is still $\Theta(n \lg n)$:

- Level 0: n
- Level 1: $\frac{n}{10}, \frac{9n}{10}$
- Level 2: $\frac{n}{100}, \frac{9n}{100}, \quad \frac{9n}{100}, \frac{9^2}{100} n$
- Level 3: …… the last one: $\left(\frac{9}{10}\right)^3 n$

And the last level would have $\left(\frac{9}{10}\right)^k n = 10 \Rightarrow k = \lg_{\frac{10}{9}} n = \lg_c n, c > 1 \Rightarrow$ that's $\Theta(\lg n)$

So the total work is: $T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n) = O(n \lg n)$

As long as you put a portion of the total on one side and another on the other side, you stay in $O(n \lg n)$, i.e. stay in logarithmic depth tree. In general:

$$\boxed{T(n) = T\left(\frac{n}{\alpha}\right) + T\left(\frac{\alpha - 1}{\alpha} n\right) + \Theta(n), \alpha \geq 2 \Rightarrow T(n) = O(n \lg n)}$$

Same logic apply for binary-search-like vriations: $T(n) = T\left(\frac{99}{100} n\right) + \Theta(1) = \Theta(\lg n)$, with the base of the log being almost 1 in this case: $\lg_{\frac{100}{99}} n$.

<u>So back to quick sort</u>:

The worst case scenario for quick sort is if the array is already sorted.

Every level you partition right at the beginning, and that's $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

Another case as bad is this is when every pivot is a few (constant) places away from its position, you get also $T(n) = T(n-c) + \Theta(n) = \Theta(n^2)$.

Experimentally, the worst cases will not happen often. The general form is: for partition $(k, n-k-1)$, the formula is $T(n) = T(k) + T(n-k-1) + \Theta(n)$.

You have many possibilities and you're interested in the **expected** running time, which will be a weighted sum of running times:

$$Exp. = \sum_{i=1}^{N} E_i \times \Pr(E_i)$$

So the expected running time is the sum of products of event probability with the event's running time:

$T(n) = \sum_k \Pr[(k, n-k-1)\ split] \cdot T(n|(k, n-k-1)\ split) =$ The probability for each event is $\frac{1}{n}$

The cost for event k is $T(k) + T(n-k-1) + \Theta(n)$

$\frac{1}{n}\sum_k [T(k) + T(n-k-1) + \Theta(n)] =$ The transition is explained in (*)

$$\frac{2}{n}\sum_{k=1}^{n-1}[T(k) + \Theta(n)]$$

(*) The sum is:

$T(1) + T(n-2) + \Theta(n)$

$T(2) + T(n-3) +$ "

…

$T(n-3) + T(2) +$ "

$T(n-2) + T(1) +$ "

⇒ There are 2 copies of all $T(k)|_{k=1}^{n-1}$

The above concludes to that the average running time of quick sort is $\Theta(n \lg n)$.

<u>Proof</u>:

$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n) \leq \frac{2}{n}\sum_{k=1}^{n-1}[ak \lg k + b + \Theta(n)] = \frac{2a}{n}\boxed{\sum_{k=1}^{n-1} k \lg k} + \frac{2}{n}nb + \Theta(n) \leq (*)$

$$\left[\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\left|\frac{n}{2}-1\right|} k \lg k + \sum_{\left|\frac{n}{2}\right|}^{n-1} k \lg k \leq 2 \lg n \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\left\lceil\frac{n}{2}-1\right\rceil} k\right) \leq \lg n \cdot \frac{n(n-1)}{2} - \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2} \leq \frac{1}{2}n^2 \lg n - \frac{n^2}{8}\right]$$

$$(*) \leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{n^2}{8}\right) + 2b + \Theta(n) = an \lg n + b + \left(\Theta(n) + b - \frac{an}{4}\right)$$

Heaps, Priority Queues and Heap Sort

Priority Queue:

Priority queue is designed to allow extraction of elements with highest priority. A queue is a kind of priority queue, where the priority is time of arrival.

This is a dynamic data structure, so it should handle insertions and removals efficiently. If you maintain a sorted data, the extraction is constant – get the top. But, the insertion costs more. Generally, there's a tradeoff between insertion and removal times. If you don't know the frequency of each operation type, you want to optimize both – binary heap.

Binary Heap:

- Initialize: initialize the structure.
- Insert(key): insert a new key.
- Remove Max: remove the largest key.

The heap is defined as a data structure that supports the above operations, and satisfies:

- Binary tree
- At every node:
  - Partial order: $key(child) < key(parent)$
  - Left-filled levels: the last level is left filled, the levels above are full.

That means that:

- At every node, that node is the largest from the tree of which that node is its root.
- You don't know anything of the relation between any left child and right child. Otherwise that's a **binary search tree**.

The relationship above is called **heap-order**.

The left-filling character means you can put the elements physically in an array, so for any node i:

- Left child sits at $2i$
- Right child sits at $2i + 1$
- Parent = $i \ div \ 2$

The height of the heap is $\lg n$ so any operation takes $\lg n$ time: $H(n) = \lg_2 n$.