Master Theorem

The master method provides a simple method for solving recurrences of the form

$T(n) = a(n/b) + f(n)$,

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

The master method can be broken down into three cases depending on how the function $f(n)$ compares with the function $n^{log_b a}$. The three cases can also be interpreted as different weightings of the recursion tree associated with the recursion.

1. **Case 1:** If $f(n) = \Theta(n^{\log_b a - \epsilon})$, then the amount of work per level geometrically increases as we go down the tree. The work at the leaf level dominates and $T(n) = \Theta(n^{\log_b a})$.
2. **Case 2:** If $f(n) = \Theta(n^{\log_b a} \log^k n)$, then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = \Theta(n^{\log_b a} \log^k n \log n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. **Case 3:** If $f(n) = \Theta(n^{\log_b a + \epsilon})$, then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$. **Note:** the recursion must additionaly satisfy the "regularity" condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

To use the master method, simply determine which case (if any) of the master theorem applies, by comparing $f(n)$ with $n^{log_b a}$. If the function $n^{log_b a}$ is the larger, consider case 1. If $f(n)$ and $n^{log_b a}$ are the same, consider case 2. And if $f(n)$ is the larger, consider case 3.

Heaps

A heap is an array object that can be represented as a nearly complete binary tree. "Nearly complete" means that the tree is completely filled at each level except for possibly the lowest, which is filled from left to right.

In an array A representing a heap, the root of the tree is $A[1]$. For a given index i of a node, the indices of its parent, left child, and right child can be determined as follows:

1. `Parent(i)` $= \lfloor i/2 \rfloor$
2. `Left(i)` $= 2i$
3. `Right(i)` $= 2i + 1$

There are two kinds of binary heaps: max-heaps and min-heaps. Max-heaps have the property that A[`Parent(i)`] $\geq A[i]$, and min-heaps have the property that A[`Parent(i)`] $\leq A[i]$.

Max-heaps have the following operations:

`Max-heapify(A,i)`
Description: Assuming that binary trees rooted at the left and right children of i are max-heaps (but the tree rooted at i may not be), return A such that the binary tree rooted at i is a max-heap.
Approach: Compare the value $A[i]$ with its left and right children. If it is smaller than either of its children, exchange it with the larger of its two children. Continue comparing the value $A[i]$ with its left and right children and making exchanges with the larger of its children, until the value $A[i]$ is greater or equal to both of its children.
Analysis: Since $A[i]$ may need to be checked with each value in some branch of the tree, the runtime is $O(h) = O(lgn)$.

`Build-max-heap(A)`
Description: Given an array $A[1..n]$, build a max-heap.
Approach: Starting from the parent of the last leaf and traversing backwards in the array (traversing left and up in the binary heap), call `Max-heapify` on each index of the array.
Analysis: Using the recurrence $T(n) = 2T(n/2) + lgn$ to model the operation, the runtime is $O(n)$.

`Heapsort(A)`
Description: Given an unordered array $A[1..n]$, sort A using a max-heap.
Approach: Build a max-heap from A. Remove the maximum element by swapping it with the last leaf and placing it at the end of our sorted array. Calling `Max-heapify` on the root will result in a max-heap with the second largest element at the root. Continue the process of exchanging the root with the last leaf, removing the current maximum,placing it in our sorted array, and calling `Max-heapify` on the root.
Analysis: We make $n - 1$ calls to `Max-heapify` (one call after each element in A that is visited), which has running time $O(lgn)$, so the total runtime of `Heapsort` is $O(n \lg n)$.

`Heap-maximum(A)`
Description: Return just the value of the maximum element of the max-heap A.
Approach: The maximum element is the root of the max-heap or $A[1]$.
Analysis: $O(1)$.

`Heap-extract-max(A)`
Description: Return the value of the maximum element of the max-heap and also remove the element from the max-heap.
Approach: The maximum element is the root of the max-heap. To remove it, we exchange it with the last leaf, remove the element, and `Max-heapify` the new root.

Analysis: The runtime is the same as `Max-heapify`, which is $O(\lg n)$.

`Heap-increase-key(A,i,key)`
Description: Increase the key of $A[i]$ to the new key key.
Approach: Check to see if the new key is actually greater than the current key. If it is, update the key $A[i]$ to the new key. This might violate the max-heap property of i's parent. If it does, exchange the key $A[i]$ with its parent. This in turn might violate the max-heap property again. So we traverse up the binary heap, making key exchanges with ancestor nodes until the max-heap property is again satisfied.
Analysis: In the worst case, we will have to traverse all the way up a binary heap from a leaf to the root, so the runtime of `Heap-increase-key` is $OO(h) = O(lgn)$.

`Max-heap-insert(A,key)`
Description: Insert key into the max-heap A.
Approach: Add a new leaf to the max-heap with an initial key ∞, then call `Heap-increase-key` on that leaf with the actualy key to be inserted.
Analysis: The runtime is the same as `Heap-increase-key`, which is $O(\lg n)$.

Comparison sort

Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.