Heaps and Balanced Trees

For in-class use only in CSC 311 course

Dr. Marek A. Suchenek ©

April 28, 2014

1 Heaps and Balanced Trees

1.1 Binary representations of positive natural numbers

How many bits are needed to represent a number $M > 0$ in binary?

Let's say it's n. The least binary number that needs n bits for its representation is one 1 followed by $n-1$ 0s. The greatest binary number that may be represented on n bits is n 1s. Thus we have:

$$2^{n-1} = 1\underbrace{00\ldots0}_{n-1} = \underbrace{100\ldots0}_{n} \leq M \leq \underbrace{11\ldots1}_{n}.$$

Also,

$$\underbrace{11\ldots1}_{n}+1 = 1\underbrace{00\ldots0}_{n} = 2^n.$$

So,

$$\underbrace{11\ldots1}_{n} = 2^n - 1.$$

Therefore,

$$2^{n-1} \leq M \leq 2^n - 1 \tag{1}$$

or

$$2^{n-1} \leq M < 2^n$$

or, since $\log_2$ is a growing function,

$$\log_2 2^{n-1} \leq \log_2 M < \log_2 2^n$$

that is,

$$n - 1 \leq \log_2 M < n.$$

Therefore, $n-1$ is the largest integer not larger than $\log_2 M$. By the definition of floor function,

$$n - 1 = \lfloor \log_2 M \rfloor$$

or

$$n = \lfloor \log_2 M \rfloor + 1. \tag{2}$$

So, $\lfloor \log_2 M \rfloor + 1$ bits are needed to represent number $M > 0$ in binary.

We will derive a different form of the formula (2). Adding 1 to all sides of inequality (1) we obtain

$$2^{n-1} + 1 \leq M + 1 \leq 2^n$$

that is,

$$2^{n-1} < M + 1 \leq 2^n$$

or, since $\log_2$ is a growing function,

$$\log_2 2^{n-1} < \log_2(M + 1) \leq \log_2 2^n$$

that is,

$$n - 1 < \log_2(M + 1) \leq n.$$

Therefore, n is the least integer not lesser than $\log_2(M+1)$. By the definition of ceiling function,

$$n = \lceil \log_2(M + 1) \rceil. \tag{3}$$

So, $\lceil \log_2(M + 1) \rceil$ bits are needed to represent number M in binary. Interestingly, the formula (3) holds for any integer $M \geq 0$, unlike the formula (2) that only holds for $M \geq 1$ (because $\log_2 0$ does not exist).

Combining the equalities (2) and (3) yelds an important equality that I recommend you try to memorize:

$$\lfloor \log_2 M \rfloor + 1 = \lceil \log_2(M + 1) \rceil, \text{ for any integer } M \geq 1. \tag{4}$$

Note. The argument we presented above carries on for any integer base $b \geq 2$, not just for base 2; just replace 2 with b where appropriate. It yields the following generalization of (4):

$$\lfloor \log_b M \rfloor + 1 = \lceil \log_b (M+1) \rceil, \text{ for any integer } M \geq 1 \text{ and } b \geq 2. \qquad (5)$$

Example 1.1.1 Let $M = 7$. Binary representation of 7 is 111. It uses 3 significant bits. On the other hand, $\lfloor \log_2 7 \rfloor + 1 = \lfloor 2.807354922... \rfloor + 1 = 2 + 1 = 3$. Also, $\lceil \log_2 (7+1) \rceil = \lceil \log_2 8 \rceil = \lceil 3 \rceil = 3$. So far so good!

1.1.1 An exercise with the imperial ruler

Do you see a set of binary sequences on Figure 1? Do you see a complete binary tree there? If so then explain why.

Figure 1: Do you see a set of binary sequences and a complete binary tree here?

A sophisticated carpenter will see each notch on the imperial ruler of Figure 1 as a binary fractional number that represents a trace of binary search of an end point the distance to which he needs to measure. In the case of the interval $(44, 45)$, such a trace may be computed this way.

Beginning with $k = 1$, he will add to or subtract $2^{-k} = \frac{1}{2^k}$ from the starting value (44 in this case), depending whether he has to go to the right

or to the left of the current notch on the scale, incrementing the value of k after each step.

For instance, to measure the distance to the red dot on Figure 2,

Figure 2: A red point to which the distance needs to be measured.

he will follow binary search right-left-right-left visualized on Figure 3.

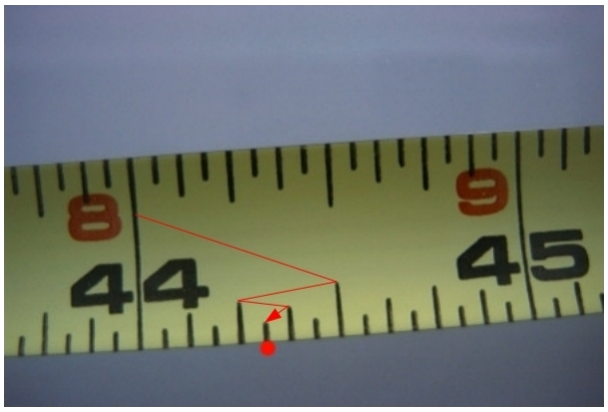

Figure 3: Binary search of the red point on the imperial ruler.

This will result of this sequence of additions and subtractions to 44:

$$+\,2^{-1} - 2^{-2} + 2^{-3} - 2^{-1} = 0.0101$$

Thus each point on the interval $(44, 45)$ is identified by a (not necessarily finite) binary sequence the does not end with 0. However, some of these points have two different binary sequences that do not end with 0, for instance, the red dot is given by 0.0101 and by $0.0100\overline{1}$ (the latter being an abbreviation for an infinite sequence 0.0100111...1...), which peculiarity is characteristic to any positional system of counting (for instance, base 10).

Another way of seeing a set of binary sequences on Figure 1 is to ascribe 0 for each move to the left in the binary search outlined above and 1 for each move to the right. In the search visualized on Figure 3, the binary sequence that identifies the red dot is 1010. Like before, some points of the interval $(44, 45)$ have many (up to three) different binary sequences, for instance the red dot is identified by 1010, $1010\overline{1}0$ and $1010\overline{0}1$.

We will utilize the latter method in analysis of heaps in the sequel of this paper.

Interestingly, both ways described above have one thing in common. If we drop the initial digit 0, the binary dot, and the last digit 1 from the fractional identification 0.0101 of the red point we obtain a sequence 010 that encodes the navigation information how to get there from the point $44\frac{1}{2}$ (go to the left, go to the right, and go to the left). And that coincides with the result of dropping the first digit 1 in the binary-search identification 1010 of the red dot. This, not surprisingly, defines a $1-1$ function from the set of fractional identifications onto the set of binary-search identifications of the notches in the interval $(44, 45)$ of the imperial ruler: erase everything from the left of the string up to and including the binary dot, and move the last digit 1 to the front of the remainder sequence.

We conclude this section with an observation that the part that the above two interpretations of the imperial ruler have in common results in the set of binary sequences that is closed under truncation, and as such is a binary tree. In particular, Figure 4 visualizes a complete binary tree given by the binary sequences corresponding to the notches of the imperial ruler.

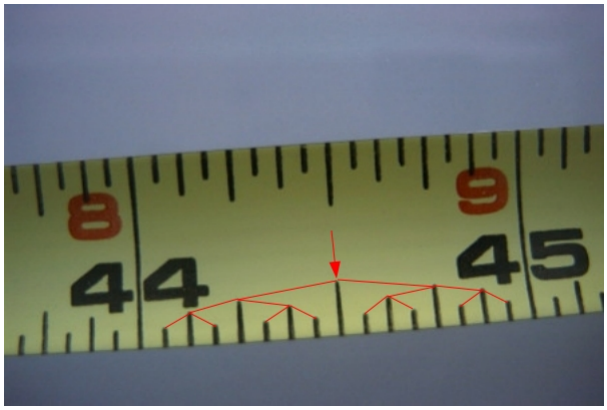

Figure 4: A complete binary tree.

1.2 Heaps

A heap is a contiguous, partially ordered binary tree. *Contiguous* means that all levels of the tree in question, except, perhaps, for the last level, contain the maximum number of nodes, and if the last level of the tree contains a lesser number of nodes then they are flushed all the way to the left.

Figure 5 visualizes an example of heap with 17 nodes. It shows nodes' ordinal numbers in decimal. Their binary representations are of the form: 1 followed by a sequence of edges' labels along the path from the root to the node in question. For instance, the sequence of labels along the path from the root to node number 17 is 0001 and the binary representation of 17 is 10001. The depth of the node number 17, defined as the length of path from the root to that node, is 4 and may be computed as one less than the length of the binary representation of 17, that is, $\lfloor \log_2 17 \rfloor$. Since it is the last node of the heap, it is also the depth of the heap. (We will comment more on this later.)

In addition to providing navigation information, labels of the edges indicate orientation of children: an edge labeled with 0 points to the left child and one labeled with 1 points to the right child. It so happens that the children of node i are $2i$ and $2i + 1$, as it has been visualized on Figure 7.

This important fact may be easily established by looking at binary representations of nodes' ordinal numbers. Since each such representation is a

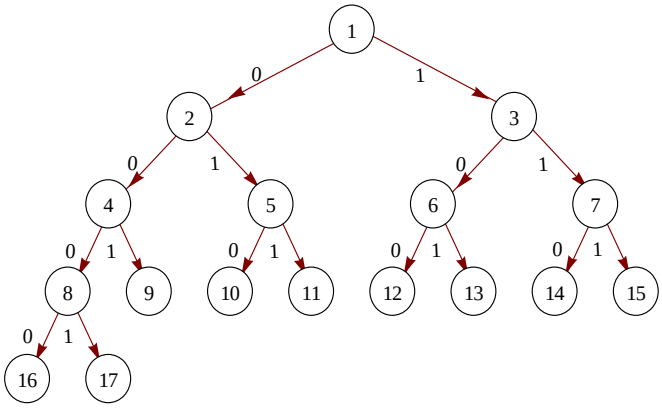

Figure 5: A heap with 17 nodes showing nodes' ordinal numbers in decimal.

Figure 6: A really large heap on a small picture.

sequence of bits that determine the path from the root to the node in question, the binary representation of a parent node is a result of truncating the last bit from the binary representation of any of its children. Truncating the last bit yields the same result as the `shiftright` operation, that performs the integer division by 2. So, if j is the ordinal number of a child then the ordinal number i of its parent is

$$i = \lfloor \frac{j}{2} \rfloor,$$

or, in other words,

$$j = \begin{cases} 2i \text{ if } j \text{ is the left child of } i \\ \\ 2i + 1 \text{ if } j \text{ is the right child of } i. \end{cases} \tag{6}$$

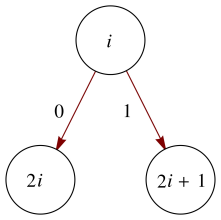

Figure 7: Ancestral information translated onto ordinal numbers.

For example, the path from the root to node 13 in the heap on Figure 5, is 101 which can be obtained from the binary representation of 13, that is, from 1101, by dropping its first digit 1. So the path to the parent of 13 is 10 and the ordinal number of the node at the end of that path (the parent of 13) is 110, or in 6 in decimal. So, 6 is the parent of 13. Of course, $6 = \lfloor \frac{13}{2} \rfloor$.

Also, the children of 6, if it has any, must have ordinal numbers that in binary read 1100 and 1101 since these are the only numbers that when divided by 2 will yield 110 or 6. These are 12 and 13.

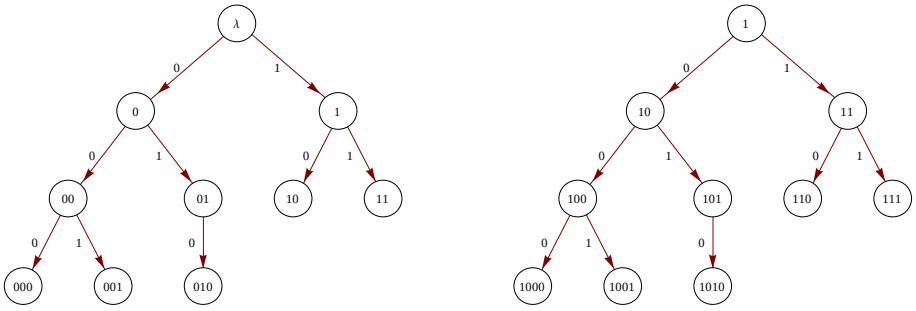

Figure 8: Two visualizations of a heap with 10 nodes: on the left showing binary sequences that represent the nodes (with λ being the empty sequence), on the right showing their ordinal numbers in binary, and edges' labels on both showing suffixes (the last digit that makes the difference between a child and its parent).

In a similar fashion one can determine if a node i has a child of children

by comparing $2i$ to n. If $2i \leq n$ then i has a child or children and if $2i > n$ then it has not (is a leaf, that is). If $2i = n$ then $2i + 1 > n$ and so node i does not have the right child. If $2i < n$ then $2i + 1 \leq n$ and so node i does have the right child.

Figures 8 and 9 visualize an example of a contiguous binary with 10 nodes with the values of the nodes shown as binary sequences, binary numbers, and decimal numbers.

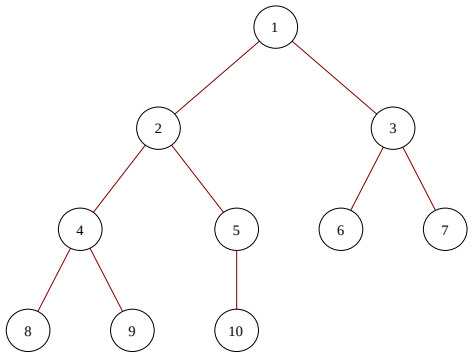

Figure 9: A heap with 10 nodes showing their ordinal numbers in decimal.

Partially ordered means that every sequence of nodes along a path from the root to a leaf in the tree is ordered in a non-increasing order. Or, in other words, that children, if any, of a node are not larger than their parent. Figure 10 visualizes an example of a heap with 10 nodes of Figure 9 with the values of the nodes shown instead of their ordinal numbers.

One of the most amazing things about contiguous trees is how cleverly are they represented with one-dimensional arrays. An array stores the nodes of the tree according to their level-by-level order. The root of the tree is stored at index 1, and the children, if any, of a node stored at index i are stored at indices $2i$ (the left child) and $2i + 1$ (the right child). A node i has a child j (left or right) if, and only if, $j \leq N$, where N is the number of nodes of the tree.

The table in Figure 11 shows an array that represents the heap of Figure 10 with the indicies of the array shown in the top row of the table.

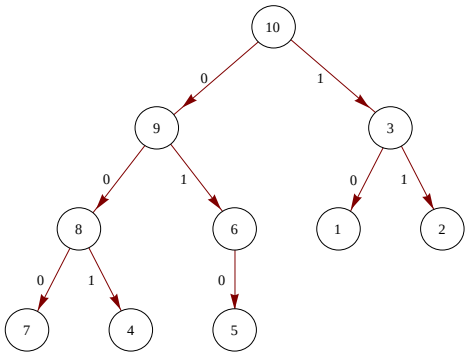

Figure 10: A heap with 10 nodes showing their values and not the ordinal numbers.

1	2	3	4	5	6	7	8	9	10
10	9	3	8	6	1	2	7	4	5

Figure 11: Array representation of the heap of Figure 10.

1.3 The height (or depth) of a heap

Each node of of a heap with n nodes is represented by a binary sequence (a path from the root of the heap to that particular node). The depth of the heap is equal to the maximal (over all nodes of the heap) length of such a path. Since the last node in the heap belongs to the last level of the heap, the length of the path from the root to that last node is maximal.

Let p be the path from the root to the last node of the heap. As we noticed before, the binary representation of that node's ordinal number (n, that is) is 1 followed by p.

In other words, the depth D_n of the heap with n nodes, which is equal to the length of p, is one less than the number of bits needed to represent n. So

$$D_n = \lfloor \log_2 n \rfloor .$$

Exercise Show that a heap with l leaves (not nodes) has a depth D that

satisfies

$$\lceil \log_2 l \rceil \leq D \leq \lfloor \log_2 l \rfloor + 1.$$

Note Since the depth of a heap with n nodes is also the level of node n, it follows that the level of node i is:

$$level(i) = \lfloor \log_2 i \rfloor.$$

To see why, remove from the heap all the nodes after i. The resulting heap will have i nodes so its height is $\lfloor \log_2 i \rfloor$, which (by the definition of the height of a heap) happens to be the same as the level of i.

1.4 The running time of H.remove() and H.insert(x)

Let H be a heap with n nodes. The number of comparisons $C_{remove}(n)$ done in the worst case by H.remove() is less equal to 0 if $n = 1$ or equal to $2 \times D_{n-1}$ or $2 \times D_{n-1} - 1$ otherwise, where D_{n-1} is the depth of the heap with $n - 1$ nodes (after removal, that is), and the number of comparisons $C_{insert}(n)$ done in the worst case by H.insert(x) is less than or equal to D_{n+1}, where D_{n+1} is the depth of the heap with $n + 1$ nodes (after inserting of x, that is).

So,

$$2 \times \lfloor \log_2(n-1) \rfloor - 1 \leq C_{remove}(n) \leq 2 \times \lfloor \log_2(n-1) \rfloor \text{ for } n > 1,$$

and

$$C_{insert}(n) = \lfloor \log_2(n+1) \rfloor.$$

Therefore,

$$C_{remove}(n) \in \Theta(\log n) \text{ and } C_{insert}(n) \in \Theta(\log n). \tag{7}$$

1.5 The running time of PriorityQueueSort

Because of (7), taking into account that the entire sorting requires n insertions followed by n removals, each of them performed on a heap with no

more than n elements, the worst-case number $C_{sort}(n)$ of comparisons by PriorityQueueSort is

$$O(n \max\{\log n, \log n\}) = O(n \lg n).$$

We will calculate a more accurate estimate of $C_{sort}(n)$

1.6 More accurate estimate of the running time of PriorityQueueSort

Assume that the array to be sorted has n elements. The number of comparisons in the first for-loop is

$$\Sigma_{i=0}^{n-1} C_{insert}(i) = \Sigma_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor$$

and the number of comparisons in the second for-loop is

$$\Sigma_{i=2}^{n}(2 \times \lfloor \log_2(i-1) \rfloor - 1) \leq \Sigma_{i=2}^{n} C_{remove}(i) \leq \Sigma_{i=2}^{n} 2 \times \lfloor \log_2(i-1) \rfloor$$

or

$$\Sigma_{i=2}^{n} 2 \times \lfloor \log_2(i-1) \rfloor - (n-1) \leq \Sigma_{i=2}^{n} C_{remove}(i) \leq \Sigma_{i=2}^{n} 2 \times \lfloor \log_2(i-1) \rfloor$$

so that the total number of comparison in both loops is

$$\Sigma_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor + \Sigma_{i=2}^{n} 2 \times \lfloor \log_2(i-1) \rfloor - (n-1) \leq C_{sort}(n) \leq$$

$$\leq \Sigma_{i=0}^{n-1} \lfloor \log_2(i+1) \rfloor + \Sigma_{i=2}^{n} 2 \times \lfloor \log_2(i-1) \rfloor$$

that is,

$$\Sigma_{i=1}^{n} \lfloor \log_2 i \rfloor + 2\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor - (n-1) \leq C_{sort}(n) \leq$$

$$\leq \Sigma_{i=1}^{n} \lfloor \log_2 i \rfloor + 2\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor$$

or

$$3\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor - n + \lfloor \log_2 n \rfloor + 1 \leq C_{sort}(n) \leq 3\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor + \lfloor \log_2 n \rfloor. \quad (8)$$

Because $\lfloor \log_2 i \rfloor$ is a non-decreasing function and because $i < n$ in summation $\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor$, we have:

$$\Sigma_{i=1}^{n-1} \lfloor \log_2 i \rfloor \leq \Sigma_{i=1}^{n-1} \lfloor \log_2 n \rfloor = (n-1) \lfloor \log_2 n \rfloor.$$

This and the right-hand side of (8) yield the following estimate (an upper bound) on $C_{sort}(n)$:

$$C_{sort}(n) \leq 3(n-1) \lfloor \log_2 n \rfloor + \lfloor \log_2 n \rfloor = (3n-2) \lfloor \log_2 n \rfloor. \tag{9}$$

1.7 Even more accurate estimate of the running time of PriorityQueueSort

Let's compute accurately the sum $S_M = \Sigma_{i=1}^{M} \lfloor \log_2 i \rfloor = \Sigma_{i=1}^{M} level(i)$. This sum is adding the levels of all nodes of the heap with M nodes together, so it can be split on the sum of all levels of the nodes that are in the first D_M levels (ranging from 0 to $\lfloor \log_2 M \rfloor - 1$) plus the sum of the levels of the nodes that are in the last level $D_M = \lfloor \log_2 M \rfloor$).

For the example of heap on Figure 5 ($M = 17$ in this case), Figure 12 shows how to split $\Sigma_{i=1}^{17} \lfloor \log_2(i) \rfloor$ on $\Sigma_{i=1}^{\lfloor \log_2 17 \rfloor - 1} i \times 2^i$ plus $(17 - 2^{\lfloor \log_2 17 \rfloor} + 1) \lfloor \log_2(17) \rfloor$.

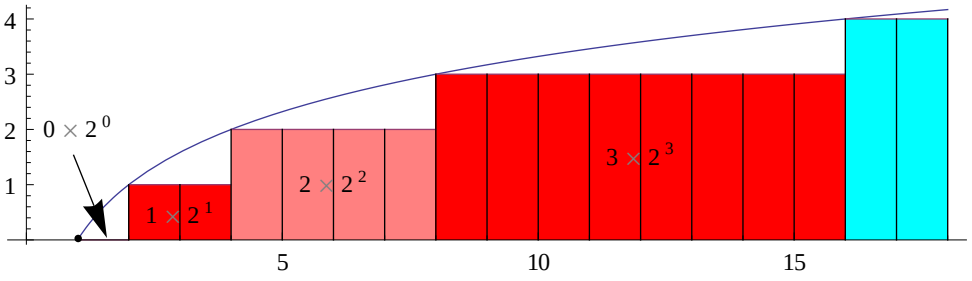

Figure 12: Computation of $\Sigma_{i=1}^{17} \lfloor \log_2(i) \rfloor$ (the colored area) as $\Sigma_{i=1}^{\lfloor \log_2 17 \rfloor - 1} i \times 2^i$ (the reddish area) + $(17 - 2^{\lfloor \log_2 17 \rfloor} + 1) \lfloor \log_2(17) \rfloor$ (the cyan area).

Clearly, there are

$$\Sigma_{j=0}^{\lfloor \log_2 M \rfloor - 1} 2^j = 2^{\lfloor \log_2 M \rfloor} - 1$$

nodes in the first $\lfloor \log_2 M \rfloor - 1$ levels of the heap. (Note that $2^{\lfloor \log_2 M \rfloor}$ is the largest power of 2 that is not greater than M.) So, the last level must contain $M - (2^{\lfloor \log_2 M \rfloor} - 1) = M - 2^{\lfloor \log_2 M \rfloor} + 1$ nodes. Therefore:

$$S_M = \Sigma_{i=1}^{M} level(i) = \Sigma_{i=1}^{2^{\lfloor \log_2 M \rfloor}-1} level(i) + \Sigma_{i=2^{\lfloor \log_2 M \rfloor}}^{M} level(M) =$$

$$\Sigma_{i=1}^{2^{\lfloor \log_2 M \rfloor}-1} \lfloor \log_2 i \rfloor + \Sigma_{i=2^{\lfloor \log_2 M \rfloor}}^{M} \lfloor \log_2 M \rfloor =$$

$$\Sigma_{i=1}^{\lfloor \log_2 M \rfloor - 1} i \times 2^i + \lfloor \log_2 M \rfloor \times (M - 2^{\lfloor \log_2 M \rfloor} + 1) =$$

$$(\lfloor \log_2 M \rfloor - 2) 2^{\lfloor \log_2 M \rfloor} + 2 + \lfloor \log_2 M \rfloor \times (M - 2^{\lfloor \log_2 M \rfloor} + 1) =$$

$$(\lfloor \log_2 M \rfloor - 2) \times 2^{\lfloor \log_2 M \rfloor} + \lfloor \log_2 M \rfloor \times M - \lfloor \log_2 M \rfloor \times 2^{\lfloor \log_2 M \rfloor} + \lfloor \log_2 M \rfloor + 2 =$$

$$\lfloor \log_2 M \rfloor \times 2^{\lfloor \log_2 M \rfloor} - 2 \times 2^{\lfloor \log_2 M \rfloor} + \lfloor \log_2 M \rfloor \times M - \lfloor \log_2 M \rfloor \times 2^{\lfloor \log_2 M \rfloor} + \lfloor \log_2 M \rfloor + 2 =$$

$$-2 \times 2^{\lfloor \log_2 M \rfloor} + \lfloor \log_2 M \rfloor \times M + \lfloor \log_2 M \rfloor + 2 =$$

$$M \lfloor \log_2 M \rfloor - 2^{\lfloor \log_2 M \rfloor + 1} + \lfloor \log_2 M \rfloor + 2 =$$

$$(M+1) \lfloor \log_2 M \rfloor - 2^{\lfloor \log_2 M \rfloor + 1} + 2.$$

Hence,

$$S_M = \Sigma_{i=1}^{M} \lfloor \log_2(i) \rfloor = (M+1) \lfloor \log_2 M \rfloor - 2^{\lfloor \log_2 M \rfloor + 1} + 2$$

.

In particular,

$$S_{n-1} = S_n - \lfloor \log_2 n \rfloor = n \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2,$$

that is,

$$\Sigma_{i=1}^{n-1}\lfloor\log_2 i\rfloor = n\lfloor\log_2 n\rfloor - 2^{\lfloor\log_2 n\rfloor+1} + 2. \quad (10)$$

Combining (8) and (10), we conclude that the total number $C_{sort}(n)$ of comparisons is:

$$C_{sort}(n) \leq 3 \times S_{n-1} + \lfloor\log_2 n\rfloor = 3(n\lfloor\log_2 n\rfloor - 2^{\lfloor\log_2 n\rfloor+1}) + \lfloor\log_2 n\rfloor + 6. \quad (11)$$

Let $x = \log_2 n - \lfloor\log_2 n\rfloor$, that is,

$$\lfloor\log_2 n\rfloor = \log_2 n - x. \quad (12)$$

Applying (12) to (11) we obtain:

$$C_{sort}(n) \leq 3(n(\log_2 n - x) - 2^{\log_2 n+1-x}) + \log_2 n - x + 6 =$$
$$3(n(\log_2 n - x) - n2^{1-x}) + \log_2 n - x + 6 = 3n(\log_2 n - (x + 2^{1-x})) + \log_2 n - x + 6 =$$
$$(3n+1)\log_2 n - 3(x + 2^{1-x})n - x + 6 = (3n+1)\log_2 n - 3\alpha n + \beta,$$

where $\alpha = x + 2^{1-x}$ and $\beta = 6 - x$, that is, $1.91 < \alpha \leq 2$ and $5 < \beta \leq 6$.

The same way we compute that

$$(3n+1)\log_2 n - 3\alpha n + \beta - (n-1) \leq C_{sort}(n),$$

or

$$(3n+1)\log_2 n - (3\alpha+1)n + \beta + 1 \leq C_{sort}(n).$$

Hence,

$$(3n+1)\log_2 n - 7n + 5 < C_{sort}(n) < (3n+1)\log_2 n - 5.73n + 6. \quad (13)$$

Of course,

$$C_{sort}(n) \in \Theta(n \log n) \quad (14)$$

as both

$$(3n+1)\log_2 n - 7n + 6 \in \Theta(n \log n)$$

and

$$(3n+1)\log_2 n - 5.73n + 6 \in \Theta(n \log n).$$

Figure 13 visualizes $C_{sort}(n)$ and its upper and lower bounds $(3n+1)\log_2 n - 7n + 6$ and $(3n+1)\log_2 n - 5.73n + 6$.

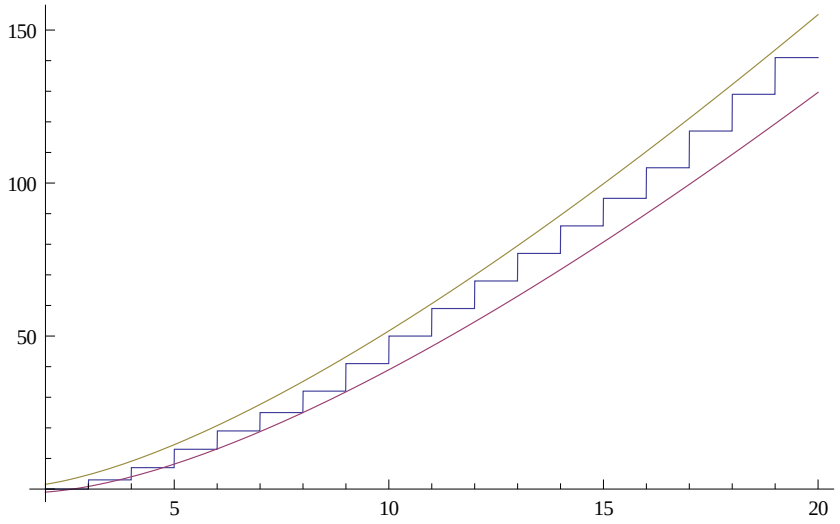

Figure 13: Graph of $C_{sort}(n)$ and its upper and lower bounds.

1.8 The running time of MakeHeap

Assume that there are n elements placed at random in an array A[1..n], so that A, although a contiguous tree, may or may not have the heap property. The method MakeHeap will re-heapfy A by scanning it from its last parent $\lfloor \frac{n}{2} \rfloor$ to its first element 1, treating each of them as the root of a contiguous subtree of A and demoting it in order re-heapify that subtree. Once this process is complete, the resulting A has a heap property so it is a heap.

First, let's note that

$$\lfloor \log_2 \lfloor \frac{n}{2} \rfloor \rfloor = \lfloor \log_2 n \rfloor - 1.$$

(To see that, let's note that $\lfloor \log_2 n \rfloor$ is the level of node n while $\lfloor \log_2 \lfloor \frac{n}{2} \rfloor \rfloor$ is the level of n's parent, so it must be 1 less.)

Let's compute $S_{\lfloor \frac{n}{2} \rfloor}$. We have:

$$S_{\lfloor \frac{n}{2} \rfloor} = \Sigma_{i=1}^{\lfloor \frac{n}{2} \rfloor} \lfloor \log_2 i \rfloor =$$

$$(\lfloor \frac{n}{2} \rfloor + 1) \lfloor \log_2 \lfloor \frac{n}{2} \rfloor \rfloor - 2^{\lfloor \log_2 \lfloor \frac{n}{2} \rfloor \rfloor + 1} + 2 =$$

$$(\lfloor\frac{n}{2}\rfloor+1)(\lfloor\log_2 n\rfloor-1)-2^{\lfloor\log n\rfloor-1+1}+2=$$

$$\lfloor\frac{n}{2}\rfloor\lfloor\log_2 n\rfloor+\lfloor\log_2 n\rfloor-\lfloor\frac{n}{2}\rfloor-1-2^{\lfloor\log n\rfloor}+2=$$

$$\lfloor\frac{n}{2}\rfloor\lfloor\log_2 n\rfloor+\lfloor\log_2 n\rfloor-\lfloor\frac{n}{2}\rfloor-2^{\lfloor\log n\rfloor}+1.$$

We note that the following formula gives a lower bound and an upper bound on the number of comparisons that MakeHeap needs, in a worst case, to re-heapify A:

$$C_{MakeHeap}\leq\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}2\times rank(i)$$

where $rank(i)$ is the depth of the subheap H_i that has i as its root. Indeed, the root i of the subheap H_i will participate in at most $2\times rank(i)$ comparisons while being demoted during re-heapification of the subheap H_i (at most two comparisons per each level of the said subheap except for the lowest level that will admit no comparisons since there are no nodes left to compare the demoted i with) and $rank(i)>0$ if, and only if, i is a parent or, in other words, $i\leq\frac{n}{2}$, that is, $i\leq\lfloor\frac{n}{2}\rfloor$.

Since $rank(i)=D_n-level(i)$ (if subheap H_i has any leaves at level D_n) or $rank(i)=D_n-level(i)-1$ (if subheap H_i has leaves only at level D_n-1), or - in other words - $rank(i)\leq D_n-level(i)$, we have:

$$C_{MakeHeap}\leq\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}2\times rank(i)\leq\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}2(D_n-level(i))=$$

$$2\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}(\lfloor\log_2 n\rfloor-\lfloor\log_2 i\rfloor)=$$

$$2\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}\lfloor\log_2 n\rfloor-\Sigma_{i=1}^{\lfloor\frac{n}{2}\rfloor}\lfloor\log_2 i\rfloor=$$

$$2\lfloor\frac{n}{2}\rfloor\lfloor\log_2 n\rfloor-2\times S_{\lfloor\frac{n}{2}\rfloor}=$$

$$2\lfloor\frac{n}{2}\rfloor\lfloor\log_2 n\rfloor-2\times(\lfloor\frac{n}{2}\rfloor\lfloor\log_2 n\rfloor+\lfloor\log_2 n\rfloor-\lfloor\frac{n}{2}\rfloor-2^{\lfloor\log n\rfloor}+1)=$$

$$-2(\lfloor \log_2 n \rfloor - \lfloor \frac{n}{2} \rfloor - 2^{\lfloor \log n \rfloor} + 1) =$$

$$2\lfloor \frac{n}{2} \rfloor + 2^{\lfloor \log n \rfloor + 1} - 2\lfloor \log_2 n \rfloor - 2 \leq$$

$$n + 2n - 2\log_2 n - 2 = 3n - 2\log_2 n - 2 \in O(n).$$

One can also show that

$$C_{MakeHeap} \geq \Sigma_{i=1}^{\lfloor \frac{n}{2} \rfloor} 2(D_n - level(i) - 1),$$

or, using calculations similar to the above,

$$C_{MakeHeap} \geq 2^{\lfloor \log n \rfloor + 1} - 2\lfloor \log_2 n \rfloor - 2 > n - 2\log_2 n - 2 \in \Omega(n).$$

Thus

$$n - 2\log_2 n - 2 < C_{MakeHeap} \leq 3n - 2\log_2 n - 2,$$

that is,

$$C_{MakeHeap} \in \Theta(n).$$

The above approximations, visualized on Fig. 14 and Fig. 15, although rough were good enough to establish the big-Θ characterization of $C_{MakeHeap}$.

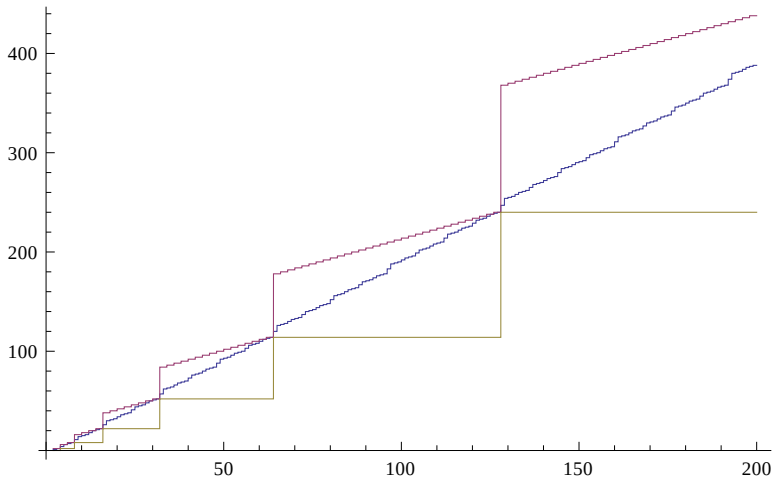

Figure 14: Graph of $2\lfloor\frac{n}{2}\rfloor+2^{\lfloor\log n\rfloor+1}-2\lfloor\log_2 n\rfloor-2$ (top), $C_{MakeHeap}$ (middle), and $2^{\lfloor\log n\rfloor+1}-2\lfloor\log_2 n\rfloor-2$ (bottom).

1.9 The speed-up while using MakeHeap instead of H.insert() in a for-loop

The speed-up is equal to approximately 33%. Additional speed-up cuts the C_{remove} roughly by 50%, thus resulting in the total speed-up of *HeapSort* by about a factor of two. This brings the worst-case number of comparisons of key of *HeapSort* down to $n\log_2 n + o(n)$, that is, in par with *MergeSort*.

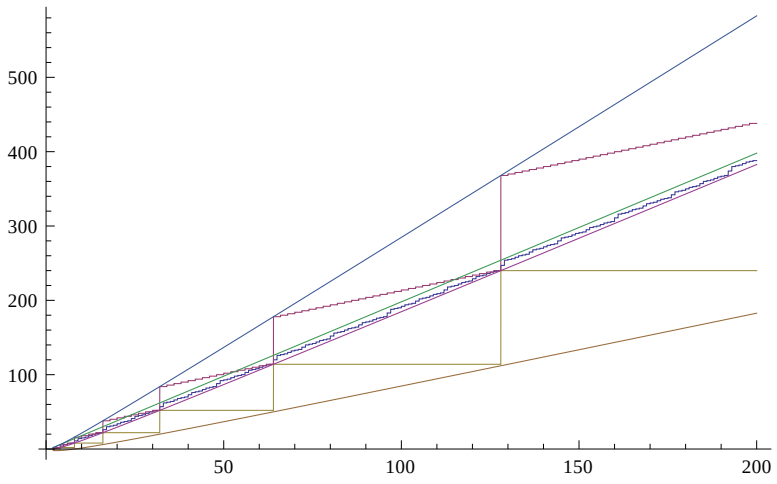

Figure 15: Graph of Fig. 14 with smooth bounds added (from the top to the bottom): $3n - 2\log_2 n - 2$, $2n - 2$, $2n - 2\log_2 n - 2$, $n - 2\log_2 n - 2$.