Analysis of Algorithms
Sorting

Andres Mendez-Vazquez

January 15, 2018

Contents

1 Introduction **2**

2 Heapsort **2**
2.1 Finding Parents and Children 3
2.2 Max/Min Heap 4
2.3 Max-Heapify 4
2.3.1 The Complexity of the Max-Heapify Algorithm 5
2.4 Build Max-Heap 7
2.5 Excercises 7

3 Quick Sort Loop Invariance **7**
3.1 Quicksort Analysis 8
3.1.1 Worst case analysis of QS 8
3.1.2 Expected Running Time of QS 8
3.2 Excercises 9

4 The Bounds of Sorting **9**

1 Introduction

The process of sorting has been one of those problems in computer science that have been around almost from the beginning of time. For example, the tabulating machine (IBM, 1890's Census) was the first early data processing unit able to sort data cards for people in the USA (Remember the Babbage's Machine was only a dream). After all the first census took around 7 years to be finished which makes the entire effort worthless. Therefore, the need to obtain efficient algorithms and machines for sorting data on those times, and in these times. Furthermore, studying different techniques of sorting allows for a more precise introduction to the algorithm concept.

2 Heapsort

Instead of going directly to the Quicksort idea, requiring probabilistic analysis, let us stop in a simpler version of sorting, heap sort. This is an earlier algorithm from 1964 invented by J.W.J Williams based in the max/min heap property.

Definition 1. A binary heap data structures is an array A that can be viewed as a nearly complete binary tree (Figure 1). This data structures has the following attributes:

1. $length\,[A]$ is the size of the storage array for the heap.

2. $heap\text{-}size\,[A]$ is how many elements are stored in the array.

Thus, having $0 \leq heap\text{-}size\,[A] \leq length\,[A]$.

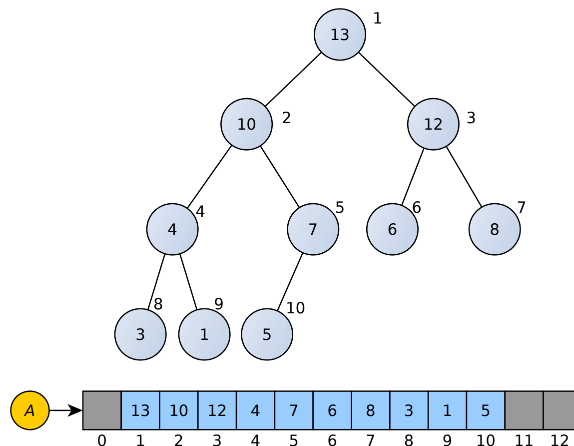

Figure 1: Nearly Complete Binary Tree

Now imagine numbering the nodes in the nearly complete binary tree in the following way (Figure 1):

1. First, number the nodes starting at the root with the number one.
2. Then, we increment the count each time we move horizontally to the next node (Level Order Walking).
3. Until, all the nodes are numbered.

This allows to map the nearly complete binary tree into the array A. Therefore, The heap A has a really regular structure where walking around is quite simple.

2.1 Finding Parents and Children

It is clear that the numbering allows to look, in a quite simple way given the position i, for the children and parent of the node at position i. For example given (Figure) a nearly complete tree of branching of three, it is clear that we can go to the left, middle or right child using simple regular functions. For example, as we can see in the figure, if you are at a parent, it is possible to move to the right child by using (Eq. 1).

$$w(x) = 3x + 1 \tag{1}$$

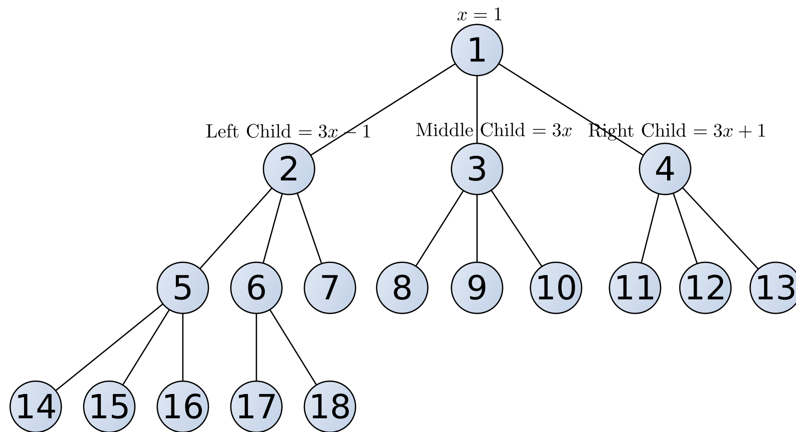

Figure 2: A Tree with Branching Factor of three

In the case of binary heaps the This is done as follow given the node i:

1. The left child position can be found by the function $Left[i] = 2i$.
2. The left child position can be found by the function $Right[i] = 2i + 1$.
3. The parent can be found by the function $Parent[i] = \left\lfloor \frac{i}{2} \right\rfloor$.

This simple functions allow to find the relationship between $A[i]$, $A[Left[i]]$ and $A[Right[i]]$ in time $\Theta(1)$.

Now, it is necessary to add an extra property to be able to do the sorting by this data structure.

2.2 Max/Min Heap

Once we have the heap structure, we add an extra field called *key* with the two possible properties:

1. We have a Max Heap if and only if $A\left[Parent(i)\right] \geq A[i]$ for all i such that $0 < i \leq heap\text{-}size\left[A\right]$.

2. We have a Min Heap if and only if $A\left[Parent(i)\right] \leq A\left[i\right]$ for all i such that $0 < i \leq heap\text{-}size\left[A\right]$.

An example of a Max Heap can be seen in the figure (Figure 1) where the nodes have the *key* values.

2.3 Max-Heapify

Given the fact that we want to maintain the Max/Min Heap structure, it is desirable to have a procedure that achieves that. Thus, What kind of scenarios do we have where the Max/Min Heap structure changes? Making necessary to re-establish the Max/Min Heap properties, when decreasing or increasing the keys.

Given that we only need to keep those properties, we have the following interesting facts:

1. If you decrease the key in a child's Max heap the parent's key is still larger than the child's key.

2. If you increase the key in a child's Min heap, the parent's key is still smaller than the child's key.

Thus, What key should you put in place of the key that has decreased? Clearly the ones at the left and right of such child by means of comparing them with the key at the child. Furthermore, if that change violates the Max/Min property, it is possibler to use a recursion procedure going down the binary heap to fix the problem. This simple idea is the one at the center of the Max-Heapify (Min-Heapify) algorithm (Algorithm 1).

Algorithm 1 A trickle down algorithm

Max-Heapify(A, i)

1. $l = Left(i)$
2. $r = Right(\text{i})$
3. If $l \leq heap - size\,[A]$ and $A\,[l] > A\,[i]$
4. $\quad largest = l$
5. else $largest = i$
6. If $r \leq heap - size\,[A]$ and $A\,[r] > A\,[largest]$
7. $\quad largest = r$
8. if $largest \neq i$
9. $\quad$ exchange $A[i]$ with $A[largest]$
10. $\quad$ **Max-Heapify**$(A, largest)$

2.3.1 The Complexity of the Max-Heapify Algorithm

Here, the recursion used to calculate the complexity is not so simple to obtain because the amount of work that is passed down the recursion. However, it is possible to prove an upper bound for that work by proving that the size children's subtrees is $\frac{2n}{3}$. For this, look at the following example

1. First for $n = 1$, we have that the size of children's subtrees is 0.
2. For $n = 2$, we have that the size of children's subtrees is at most $1 < \frac{4}{3}$.
3. For $n = 3$, we have that the size of children's subtrees is at most $1 < \frac{6}{3} = 2$.
4. For $n = 4$, we have that the size of children's subtrees is at most $2 < \frac{8}{3}$.
5. etc...

Thus, we can use a full binary tree to try to obtain a bound for the number of nodes in each child (Fig.) by recognizing that

1. The number of nodes between the root and the last level of the full binary tree is equal to

$$2^1 + 2^2 + ... + 2^{\lceil \log_2 n \rceil - 2} \tag{2}$$

2. The number of nodes in the last level of the full binary tree

$$\tag{3}$$

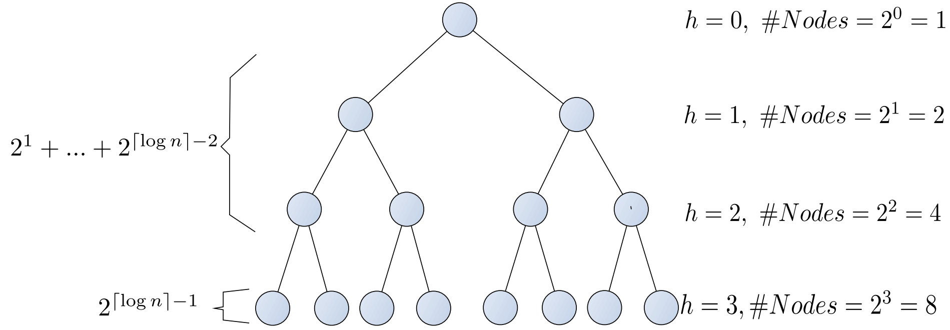

Figure 3: Given n nodes , we can use the *ceil* function to calculate the total number of nodes below the root.

Now, the total number of nodes in both children, assuming a full tree with n nodes

$$2^1 + 2^2 + ... + 2^{\lceil \log n \rceil - 2} + 2^{\lceil \log n \rceil - 1} \tag{4}$$

Now, Thus, we look at the limiting case in the structure of the

Then, we have that in the worst case the total number of elements in a children's subtree is:

$$\begin{aligned}
\frac{2^1 + 2^2 + ... + 2^{\lfloor \log n \rfloor}}{2} &= 1 + 2^2 + ... + 2^{\lfloor \log n \rfloor - 1} \\
&= \frac{1 - 2^{\lfloor \log n \rfloor}}{1 - 2} \\
&= 2^{\lfloor \log n \rfloor} - 1 \\
&< \frac{4}{3} 2^{\lfloor \log n \rfloor} - \frac{2}{3} - \frac{1}{3} \\
&< \frac{4}{3} 2^{\lfloor \log n \rfloor} - \frac{2}{3} \\
&< \frac{2}{3} \left[2^{\lfloor \log n \rfloor} - 1 \right] \\
&= \frac{2}{3} [n - 1] \\
&< \frac{2n}{3}
\end{aligned}$$

Therefore, we have that $n > -6$ is true for $n > 0$. Then, we have the following recursion $T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$, and from the Master Theorem $T(n) = O(\log n)$.

2.4 Build Max-Heap

The Build algorithm keeps the properties of the max heap. This can be seen through the following induction

Loop-Invariance At the start of each iteration of the for loop of lines 2–3, each node $i+1, i+2, ..., n$ is the root of a max-heap.

Initialization Each node $\lfloor \frac{n}{2} \rfloor + 1, \lfloor \frac{n}{2} \rfloor + 2, ..., n$ is a leaf, therefore is a trivial max-heap

Maintenance It is easy to see that the children of root i are higher in value than the parent. Then, you call Max-heapify preserves the property that nodes $i+1, i+2, ..., n$ are all roots of max heaps. Once that call is terminated i is the root of a max heap, then when i is decremented we keep the property of loop-invariance.

Termination Once you reach $i = 0$. Then, each node $1, 2, 3, ..., n$ is the root pf a max heap!!! In particular node 1.

Using the algorithm and a naive counting we can assume that the cost of building the max-heap is $O(n \log n)$. A more tight solution is giving by the following facts:

1. Height of the heap $\lfloor \log n \rfloor$.
2. You have at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$nodes of any height, where the height h is the number of edges on the longest simple downward path from the node to a leaf, and it is being measured from the bottom to the top.
3. The time required of Max-Heapify when called on height h is $O(h)$ to trickle down to leaf level.

Then, we have that, first $\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ work for level and $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$ as total work.

2.5 Excercises

3 Quick Sort Loop Invariance

While looking to the loop invariance at the slides, we can device the following proof for loop invariance

Initialization Prior to the first equation, $i = p - 1$ and $j = p$. Then, no value lies in $[p, i]$ or $[i+1, j-1]$.

Maintenance
1. Case I - $A[j] > x$ then j is incremented and condition 2 holds for $A[j-1]$.
2. Case II - $A[j] \leq x$ then i is incremented, swap $A[i]$ and $A[j]$ and j is incremented. Now, $A[i] \leq x$ and condition 1 is satisfied. In addition, $A[j-1] > x$.

Termination At termination $j = r$. The array is partitioned in three sets: those less or equal to x, a singleton containing x and those greater than x.

3.1 Quicksort Analysis

As with any other algorithm, at least two analysis need to be performed: The Worst Case and the Average Case. The first one tells us how bad the Quicksort can be, but strangely the quick sort really requires a bad selection of a pivot, and this happens when:

1. Array is already sorted in same order.
2. Array is already sorted in reverse order.
3. All elements are same (special case of case 1 and 2)

Thus, if the array elements are coming form a uniform distribution, this should happen quite rarely.

3.1.1 Worst case analysis of QS

Using the substitution and the guess $T(n) \leq cn^2$, we can prove that $T(n) = max_{0 \leq q \leq n-1} \{T(q) + T(n-q-1) + \Theta(n)\} = O(n^2)$.

$$T(n) \leq max_{0 \leq q \leq n-1} \left\{ cq^2 + c(n-q-1)^2 \right\} + \Theta(n)$$

$$T(n) \leq c * max_{0 \leq q \leq n-1} \left\{ q^2 + (n-q-1)^2 \right\} + \Theta(n)$$

But we have that $\frac{d^2[q^2+(n-q-1)^2]}{d^2q} = 4$. Then, $q^2 + (n-q-1)^2]$ is a convex function with maximum at either point 0 or $n-1$. Thus,

$$T(n) \leq c * max_{0 \leq q \leq n-1} \left\{ q^2 + (n-q-1)^2 \right\} + \Theta(n) \leq c(n-1)^2 + \Theta(n).$$

This implies $T(n) \leq cn^2$.

3.1.2 Expected Running Time of QS

First, we have the following Lemma to describe the behavior of the Quick Sort.

Lemma 7.1

X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

Proof: Imagine the following, you partition the array, first one time at height zero, then two times at height one, etc. The interesting thing is that the partition is always done, even in the case where the array has the worst possible partition given the random partition. Therefore, we have the following count for the number of partitions or pivot selection

$$2^0 + 2^1 + 2^2 + ...2^{\log n} = \frac{1 - 2^{\log n + 1}}{1 - 2} = \frac{1 - 2n}{-1} = 2n - 1 = O(n)$$

Thus, we have that the number of partitions is bounded by $O(n)$. Now, we have work done at the level of the loop in line 4. For now, the amount of work done at that level will be called X. Then, the QUICKSORT is bounded by $O(n + X)$

Now, we need to calculate the quantity X. For this, we will assume the following definitions and constraint:

1. $A = \langle z_1, z_2, ..., z_n \rangle$ the original array.
2. $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$.
3. The fact that two elements are compared only if one of them is a pivot.

Then if we define $X_{ij} = I\{z_i \text{ comparess with } z_j\}$. It is possible to define $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \Rightarrow E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$, but $E[X_{ij}] = Pr\left\{z_i \text{ is compared to } z_j\right\}$.The final magic is as follows

$$Pr\left\{z_i \text{ is compared to } z_j\right\} = Pr\left\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\right\}$$

$$= Pr\left\{z_i \text{ is choosen as pivot chosen from } Z_{ij}\right\} + Pr\left\{z_j \text{ is choosen as pivot chosen from } Z_{ij}\right\}$$

$$= \frac{1}{j - i + 1} + \frac{1}{j - i + 1} = \frac{2}{j - i + 1}.$$

Then, $E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1}$ which gives us $E[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$. From this, we have that QUICKSORT is bounded by $O(n \log n)$.

3.2 Excercises

4 The Bounds of Sorting

In this section, we are going to analyze the bound of sorting based in the fact that the main sorting algorithms are based in the comparisons. Then, imagine

the following decision-tree as a way to find all possible permutations for an set of three elements (3!).

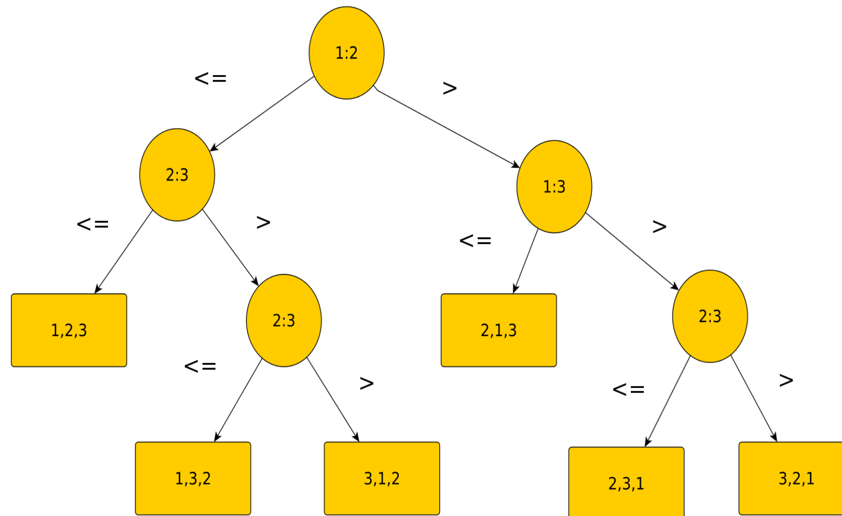

Figure 4: Decision three for three elements

This is the basis for the proof of the following theorem.

Theorem 8.1

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

Proof:

Consider a decision three of height h with l reachable leaves where each leave is an end decision or sort. We know that for n possible elements we have $n!$ possible ways of sorting those element using comparisons. Then, we have that $n! \leq l \leq 2^h$. After all when taking a full tree with height h, any binary tree full or not full should have less leaves. This implies that $\log n! \leq h \Rightarrow h = \Omega(n \log n)$. QED

From this theorem the corollary 8.2 is evident.