Algorithms: An overview

Josh Felmeden

June 3, 2019

Contents

1 Heap sort and Heapify **1**
1.1 Trees . . . 1
1.1.1 More definitions . . . 1
1.1.2 Binary trees . . . 2
1.2 Priority queues . . . 2
1.3 Heap property . . . 2
1.3.1 Constructing a heap . . . 3
1.3.2 Runtime of Heapify() . . . 3
1.3.3 Closer inspection of Heap Construction . . . 3
1.4 The complete algorithm . . . 4

2 Quicksort **4**
2.1 The Partition Step . . . 5
2.2 Runtime of quicksort . . . 5
2.2.1 Worst case . . . 5
2.2.2 Best case . . . 6
2.3 Splits . . . 6

3 Recurrences **7**
3.1 Solving recurrences . . . 7
3.2 The substitution method . . . 8
3.2.1 Guess good upper bound . . . 8
3.2.2 Substitute into the recurrence . . . 8
3.2.3 Verify the base case . . . 9
3.2.4 When substitution gets weird . . . 9

1 Heap sort and Heapify

1.1 Trees

Tree definition

A tree $T = (V, E)$ of a size n is a tuple consisting of:

$$V = \{v_1, v_2, \ldots, v_n\} \text{ and } E = \{e_1, e_2, \ldots, e_{n-1}\}$$

with $|V| = n$ and $|E| = n - 1$ with $e_i = \{v_j, v_k\}$ for some $j \neq k$ such that for every node v_i there is at least a single edge e_j such that $v_i \in e_j$. V are the nodes or vertices and E are the edges of T.

A **rooted tree** is a special kind of tree, such that $v \in V$ is a designated node that we call the root of T. This means that there's only a single node at the top most level of the tree. A *leaf* is a node that has only one incident edge. Basically, it has no children. Nodes that are not leaves are called *internal nodes*.

1.1.1 More definitions

Some more definitions of trees:

- The *parent* of a node is the node closest on a path from the node to the root. The root has no parents.
- The *children* of a node v are all of the neighbours except from its parent.
- The *height* of a tree is the length of the longest root-to-leaf path in the tree.
- The *degree* of a node is the number of incident edges to v. Since every edge is incident to two vertices, we end up with:

$$\sum_{v \in V} deg(v) = 2 \cdot |E| = 2(n-1)$$

- The *level* of a vertex v is the length of the unique path from the root to v plus 1.

Trees, by definition, need to have at least 2 leaves. This is because if we don't have this, we end up with:

$$\sum_{v \in V} deg(v) = \sum_{v \in L} deg(v) + \sum_{v \in V/L} deg(v)$$
$$\geq |L| \cdot 1 + (|V| - |L|) \cdot 2 = 2|V| - |L| \geq 2n - 1$$

Which is a contradiction to the above formula for a tree (and degrees).

1.1.2 Binary trees

Binary trees definition

A (rooted) tree is k-ary if every node has at most k children. Therefore, if $k = 2$ then the tree is binary. A k ary tree is:

- **Full** if every internal node has exactly k children
- **Complete** if all levels (except maybe the last one) is entirely filled, and the last one is filled from left to right.
- **Perfect** if every level is entirely filled.

The height of k-ary trees are special, because the number of nodes of a perfect k-ary tree with height $i - 1$ is

$$\sum_{j=0}^{i-1} = \frac{k^i = 1}{k - 1}$$

In other words, a perfect k-ary tree on n nodes has a height of

$$\log_k(n(k-1)+1) = O(\log_k n)$$

Similarly, a complete k-ary tree has a height of $O(\log_k n)$. The runtime of loads of algorithms that are using tree data structures depends on the height of the trees. Therefore, we're interested in using complete or perfect trees.

1.2 Priority queues

A priority is a data structure that allows us to access to a number of operations. These include:

- Build – Create data structure given a set of items.
- Extract-Max – Remove the maximum element from the data structure

Sorting with one of these is super easy because we can just extract-max over and over again.

1.3 Heap property

Before we start looking at the heap property, it's important to understand how we represent (complete binary) trees from arrays. Simply:

- The **parent** of the element at position i will be the element at position $\lfloor i/2 \rfloor$

- The left and right children of i are $2i$ and $2i + 1$ respectively

The **heap property** are that the parents of nodes are bigger than that of their children. That is to say that the values that the nodes hold are larger than that of their children's.

1.3.1 Constructing a heap

Given some binary tree, we're able to transform it into one that fulfils the Heap Property. The steps are really simple:

1. Traverse our tree with right to left array ordering.
2. If the node doesn't satisfy our heap property, then we run **Heapify()**

Heapify() Leg p be the key of a node and let c_1, c_2 be the values of its children.

- Let $c = \max\{c_1, c2\}$
- If $c > p$ then exchange the two nodes
- Call **Heapify()** at the node with the value c

1.3.2 Runtime of Heapify()

Exchanging the nodes requires a time of $O(1)$. The number of recursive calls is concretely bounded by the height of the tree (which we know is $\log n$), so $O(\log n)$. Therefore, the runtime of **Heapify** is $O(\log n)$.

Following directly from this, the runtime of constructing a heap has runtime $O(n \log n)$ because at worst case we'd need to run **Heapify** n times.

1.3.3 Closer inspection of Heap Construction

The runtime of Heapify is in the order $O(\log n)$. This is because at worst case, we need to continually change the nodes down to the depth bottom node, which is of course $\log n$.

It's important to note that most nodes are close to the bottom. There are more leaves than internal nodes in a perfect binary tree also.

We only need to run Heapify on the internal nodes. Let's let i be the largest integer such that $n' := 2^i - 1$ and $n' < n$. There are at most n' internal nodes that are candidates for Heapify because of this. These nodes are contained in a perfect binary tree. The perfect tree has height of $i - 1$.

We now need to sum over the relevant levels, and count the number of nodes per level, and multiply the depth of their subtrees (because we might need to rerun it for each of the children).

The sum of all the work we need to do is that we sum all of the nodes in level i, and subtract the depth of the subtree j.

$$\sum_{j=1}^{i} 2^{i-j} \cdot j = 2^i \cdot \sum_{j=1}^{i} \frac{j}{2^j} = O(2^i) = O(n') = O(n)$$

As seen here, the time complexity is really more like $O(n)$ rather than $O(n \log n)$

1.4 The complete algorithm

Firstly, we need to build the heap, and repeat this n times: $O(n)$

- Swap root with the last element $O(1)$
- Decrease size of heap by 1 $O(1)$
- Heapify(root) $O(\log n)$

Therefore the runtime is $O(n \log n)$.

Heapsort is **not stable** because the first step is to swap the root with the last element, therefore switching the order. If the heap was entirely one value, then they would be swapped.

2 Quicksort

Quicksort is a speedy little boy, but it can be really pretty naff. It's basically just an in place version of merge sort, but it also has a worst case runtime of $O(n^2)$. If we look at the two side by side, we can see there are a few subtle differences:

- *Mergesort:* First, solve the subproblems recursively and then merge the solutions.
- *Quicksort:* First partition the problem into two subproblems in a clever way so that no more work is needed when combining the subproblems, then solve the subproblems recursively.

To divide and conquer quicksort, we need to:

- **Divide**: Choose a good *pivot* ($A[q]$). Rearrange A such that every element $\leq A[q]$ is left of $A[q]$ in the resulting ordering, and every element $> A[q]$ is right of $A[q]$ in the resulting ordering. We let p be the new position of $A[q]$.
- **Conquer**: Sort $A[0, p-1]$ and $A[p+1, n-1]$ recursively.

- **Combine**: We don't need to do any work to combine them.

The issues that can arise from this is that we need to be able to rearrange the elements around the pivot in $O(n)$ time. What is a good pivot though? Ideally we want subproblems of similar sizes, so we don't get really really big sub problems to deal with.

2.1 The Partition Step

In this step, we give it an array A of length n, and we get out a partition around the pivot $A[n-1]$.

```
x = A[n-1]
i = -1
for j = 0 to n-1
    if A[j] <= x then
        i = i + 1
        exchange A[i] with A[j]
    end if
next
return i
```

The algorithm assumes that the pivot is $A[n-1]$. If we wanted a different pivot, we just switch the right pivot into position $n-1$.

2.2 Runtime of quicksort

The worst case runtime of quicksort of an input of length n would consist of:

$$T(1) = O(1)$$
$$T(n) = O(n) + T(n_1) + T(n_2)$$

Where n_1, n_2 are the lengths of the two resulting subproblems. Observe that $n_1 + n_2 = n - 1$.

In worst case, the pivot will be the largest element, in which case: $n_1 = n - 1, n_2 = 0$.

In the best case, the pivots will split the array exactly evenly, in which case: $n_1 = \lfloor \frac{n-1}{2} \rfloor$.

2.2.1 Worst case

Suppose the partition function runs in time C_n for some constant C. The recurrence looks like this:

$$T(n) \leq C_n + T(n-1)$$

And therefore the total runtime is:

$$\begin{aligned} T(n) &\leq \sum_{i=1}^{n} Ci = C\sum_{i=1}^{n} i \\ &= C\frac{(n+1)n}{2} \\ &= \frac{C}{2}(n^2+n) = O(n^2) \end{aligned}$$

2.2.2 Best case

In the best case, $n_1, n_2 \leq \frac{n}{2}$

The number of levels is I:

- Last level: n = 1

$$\begin{aligned} \frac{n}{2^{I-1}} &\leq 1 \\ \log(n) + 1 &\leq I \end{aligned}$$

- The penultimate level is n = 2:

$$\frac{n}{2^{I-2}} > 1 \text{ which implies } \log(n) + 2 > I$$

- Hence there are $I = \lceil \log(n) \rceil + 1$ levels.

The total runtime (taking into account the total runtime of the partition function in a level is $O(n)$) is $I \cdot O(n) = O(n \log n)$.

2.3 Splits

It's really important that the subproblems are *roughly* balanced. It's actually alright if $n_1 = \frac{1}{1000}$ and $n_2 = n - 1 - n_1$ to get $n \log n$ runtime. In practice, this happens the majority of the time, so quicksort is normally pretty quick(!).

To select a decent pivot, we have $O(n)$ time to spare to select a good pivot. Thankfully, there are $O(n)$ time algorithms to select such a pivot (the median). They're pretty complicated and not very efficient, but they do work.

In practice, selecting a pivot at complete random works the majority of the time.

A **bad split** is defined as if $\min\{n_1, n_2\} \leq \frac{1}{10}n$. If we select the pivot randomly, then there's a 20% chance we get a bad split (because we choose the split twice).

3 Recurrences

As we've previously seen, we make use of a lot of divide and conquer algorithms. Just to reiterate:

- **Divide**: We split the problem into a number of subproblems that are just smaller instances of the same problem
- **Conquer**: We tackle the subproblems by solving them recursively until the subproblems have constant size, in which case we solve them *directly*.
- **Combine**: At the end, the solutions to the subproblems are combined to be the solution to the original problem.

Examples of this method are: quicksort, mergesort, binary search, ...

If we look back and remember merge sort, we split the array A into n subarrays. We then sort the two sub arrays recursively using the same algorithm and combine the results. The runtime (assuming that n is a power of two) is:

$$T(1) = O(1)$$
$$T(n) = 2T(n/2) + O(n)$$

What this means is that if the input is of size 1, then the runtime will trivially be $O(1)$ since it has to be sorted. Otherwise, the runtime of an input of size n would be the same as 2 times the runtime of the algorithm on input of size $n/2$ (since we will split the input into two), plus the runtime of the algorithm on size n (for the final sorting step).

3.1 Solving recurrences

Divide-and-conquer algorithms lend themselves naturally to recurrences. To solve them, we're normally only interested in *asymptotic upper bounds*.

There are a few methods we can take for solving them:

- Substitution method
    1. Guess solution
    2. Verify
    3. Induction
- Recursion tree method (what we see for merge sort)
    1. Might have a lot of awkward details

- Master theorem
    1. Very powerful, but cannot always be applied.

3.2 The substitution method

The steps of the substitution method are as follows:

1. Guess the form of the solution
2. Use mathematical induction to find the constants and show that the solution works
3. Method provides an upper bound on the recurrence.

Let's look at an example:

$$T(1) = c_1$$
$$T(n) = 2T(n/2) + c_2 n$$

3.2.1 Guess good upper bound

We eliminate O-notation in recurrence. This means that where we see $O(1)$, we can just replace it with some constant. Also, if we have $O(n)$ would be some constant multiplied by n. This is how we achieved the above equations. The first step is to guess a good upper bound. This time, we'll be using:

$$T(n) \leq Cn \log n$$

3.2.2 Substitute into the recurrence

Next, we substitute into the recurrence. We can assume that our guess of $T(n) \leq Cn \log n$ is correct for every value smaller than n ($n' < n$). It also corresponds to the induction step of a proof by induction:

$$\begin{aligned} T(n) &= 2T(n/2) + c_2 n \leq 2C\frac{n}{2}\log(\frac{n}{2}) + c_2 n \\ &= Cn(\log(n) - \log(2)) + c_2 n \\ &= Cn\log n - Cn + c_2 n \leq Cn \log n \end{aligned}$$

If we chose $C \geq c_2$, then it's correct, because then the whole thing would be bounded by $Cn \log n$. Otherwise, the upper bound wouldn't hold.

Our goal is to show that $T(n)$ is **at most** $C \cdot n \log n$. To do this, we use the definition that we previously discussed ($T(n)$ is at most $2T(n/2) + c_2 n$). Since we're doing a proof by induction, we can apply the upper bound for this term. So we replace the $T(n/2)$ and we put in our guess, but applied to $n/2$. Therefore, we have achieved our goal because all we want to achieve is an upper bound. If $c_2 n - Cn$ and $C \geq c_2$, then we're going to be subtracting from $C_n \log n$, leaving us with the desired result.

3.2.3 Verify the base case

The base case is:

$$T(1) \leq C \cdot \log(1) = 0 \ngeq c_1$$

This is a problem, because we don't get a consistency. Luckily, we can just choose a different base case ($n = 2$).

$$\begin{aligned} T(2) &= 2T(1) + 2c_2 = 2c_1 + 2c_2 = 2(c_2 + c_1) \\ C2\log 2 &= 2C \end{aligned}$$

Hence for every $C \geq c_2 + c_1$, our guess holds for $n = 2$.

$$T(2) \leq C2\log 2$$

We've now proved that $T(n) \leq Cn \log n$ for every $n \geq 2$ when choosing $C \geq c_1 + c_2$.

This implies that $T(n) \in O(n \log n)$. This is important and remember this.

3.2.4 When substitution gets weird

Let's look at an example where substitution gives us a weird output.