Algorithms: The Notes

Josh Felmeden

April 29, 2019

Contents

1 Peak Finding

Let $A = a_0, a_1, \cdots, a_{n-1}$ be an array of integers of length n.

0	1	2	3	4	5	6	7	8	9
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9

A **peak** is an integer a_i where the adjacent integers are not larger than a_i. That is to say, if we had the array:

4	3	9	10	14	8	7	2	2	2

The problem we're faced with is that we need an algorithm to find the peaks, when we give it some array of arbitrary length. For example:

```
int peak(int *A, int len) {
    if A[0] >= A[1] then
        return 0
    end if
    if A[len - 1] >= A[len - 2] then
        return len - 1
    end if
    for (int i = 1, i < len - 1, i++)
        if A[i] >= A[i-1] AND A[i] >= A[i+1] then
            return i
        end if
    next
    return -1
}
```

What we *can* say is that every integer array has at least **one peak**. This is the same as saying 'is peak finding well defined'. The proof is that if we let A be an integer array of length n, then suppose that A does not have a peak (for the sake of contradiction). It must be the case that $a_1 > a_0$ because otherwise a_0 would be a peak. But then, $a_2 > a_1$ because otherwise a_1 is a peak. This would continue until $a_i > a_{i-1}$ and then we're out of options so a_n must be a peak. This is a contradiction so therefore every array has to have a peak.

Going back to the above algorithm, this has runtime $O(n)$, or more precisely $4(n-1)$ because it runs both $A[0] \text{and} A[n-1]$ twice, and $A[1] \cdots A[n-2]$ 4 times.

1.1 Fast peak finding

We can do much better than the initial example algorithm through **recursion**:

- `if (A.length == 1)return 0`
- `if (A.length == 2)return (A[0] > A[1])? A[0] : A[1]`

- `if (A[n/2].isPeak())return A.length / 2`
- `else if (A[n/2 - 1] >= A[n/2])return fastpeak(A[0,n/2] - 1)`
- `else return n/2 + 1 + fastpeak(A[n/2 + 1, n - 1])`

It's good because right at the end it calls itself, so this makes it more effective.

Without the recursive calls, the algorithm looks at the array elements at most **5 times**. If we let $R(n)$ be the number of calls to the fast peak finding algorithm, and the input array has length n, then we end up with:

$$R(1) = R(2) = 1$$
$$R(n) \leq R(\lfloor n/2 \rfloor) + 1, \text{for } n \geq 3$$

Solving the recurrence (see later on), we get:

$$\begin{aligned} R(n) &\leq R(\lfloor n/2 \rfloor) + 1 \leq R(n/2) + 1 = R(\lfloor n/4 \rfloor) + 2 \\ &\leq R(n/4) + 2 = \cdots \leq \lceil \log n \rceil \end{aligned}$$

1.2 Why does it work?

Well, if we look at the steps of the algorithm:

1. if A is of length 1, then we return 0
2. if A is of length 2, then we return the position of the larger element ($A[0]$ or $A[1]$)
3. if $A[\lfloor n/2 \rfloor]$ is a peak, then we return $\lfloor n/2/rfloor$.
4. Otherwise, if $A[\lfloor n/2 \rfloor - 1] \geq A[\lfloor n/2 \rfloor]$ then we call the algorithm again with A from 0 to $\lfloor n/2 \rfloor - 1$.
5. If this is not the case, then we call the algorithm again with A from $\lfloor n/2 \rfloor + 1$ to $n - 1$, and we add $\lfloor n/2 \rfloor + 1$ to this answer.

It's pretty obvious that steps 1-3 are correct. However, why is step 4 correct? (step 5 follows from 4).

- We need to prove that a peak in $A[0, \lfloor n/2 \rfloor - 1]$ is a peak in A.
- The critical case is that $\lfloor n/2 \rfloor - 1$ is a peak in $A[0, \lfloor n/2 \rfloor - 1]$.
- The condition in step 4 actually guarantees that $A[0, \lfloor n/2 \rfloor - 1] \geq A[\lfloor n/2 \rfloor]$ and therefore $\lfloor n/2 \rfloor - 1$ is a peak in A as well. This is a really important fact so make sure you remember it.

2 O notation

The runtime of an algorithm is the function that maps the input length n to the number of simple operations.

The general order of functions is as follows:

$$\log n \leq n \leq n \log n \leq n! \leq n^n$$

For a large enough n value, constants seem to matter less, but for smaller values of n, most of the algorithms are fast anyway (not *all* the time though).

An important fact to remember is that an increasing function f grows *asymptotically* at least as fast as an increasing function g if there exists an $n_0 \in N$ such that for every $n \geq n_0$ it holds. What this means is that the function f grows at least as fast as function g. For example:

$$f(n) = 2n^3, \ g(n) = \frac{1}{2} \cdot 2^n$$

From this, $g(n)$ grows asymptotically at least as fast as $f(n)$ since for every $n \geq 16$, we have $g(n) \geq f(n)$. How do we prove this? In the following way.

Firstly, we need to find values for n of which the following statements hold true:

$$\begin{aligned}
\frac{1}{2} \cdot 2^n &\geq 2n^3 \\
2^{n-1} &\geq 2^{3 \log n + 1} \text{ (using n} = 2^{\log n}) \\
n - 1 &\geq 2 \log n + 1 \\
n &\geq 3 \log n + 2
\end{aligned}$$

These statements do indeed hold for every $n \geq 16$ (which follows from the racetrack principle (2.1))

2.1 The Racetrack Principle

Racetrack principle: Let f, g be functions and k be an integer. Also suppose that the following hold:

1. $f(k) \geq g(k)$
2. $f'(n) \geq g'(n)$ for every $n \geq k$

Then, for every $n \geq k$, it holds that $f(n) \geq g(n)$

If we take an example where $n \geq 3 \log n + 2$ holds for every $n \geq 16$, we see that

- $n \geq 3 \log n + 2$ holds for $n = 16$

- We then have: $(n)' = 1$ and $(3 \log n + 1)' = 3/(n \ln 2)$ The result follows:

If we take $\geq$ to mean *grows asymptotically at least as fast* then we end up with:

$$5 \log n \leq 4(n-1) \leq n \log(n/2) \leq 0.1n^2 \leq 0.01 \cdot 2^n$$

2.2 Big O Notation

Definition: O-Notation (sometimes called Big O)

Let $g : \mathbb{N} \to \mathbb{N}$ be a function. Then, $O(g(n))$ is the set of functions:

$O(g(n)) = \{f(n) :$ There exists positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ $\}$

Don't forget that $f(n) \in O(g(n))$ means that 'g grows asymptotically at least as fast as f up to any constant'.

The way that we calculate Big O is to use the formula above, and then see if we can choose some c and some n_0 to make sure that $p \leq f(n) \leq cg(n)$ is matched.

Some other properties of O notation is that you can't apply proofs n times. For example, say we wanted to prove that $n^2 \in O(n)$, what we couldn't do is:

$$\begin{aligned} n^2 &= n + n + \underbrace{n + \cdots n}_{n-2 \text{ times}} = O(n) + O(n) + \underbrace{n + \cdots n}_{n-2 \text{ times}} \\ &= O(n) + \underbrace{n + \cdots n}_{n-2 \text{ times}} = O(n) + O(n) + \underbrace{n + \cdots n}_{n-3 \text{ times}} \\ &= O(n) + \underbrace{n + \cdots n}_{n-3 \text{ times}} = O(n) \end{aligned}$$

For the runtime of an algorithm, we express it with O-notation. This means that we can compare the runtimes of algorithms. It is important that we find the slowest growing function f so that the runtime is in $O(f)$. (Side note, most algorithms have a runtime of $O(2^r)$)

Rules for analysis of algorithms

- Composition of instructions

$$f \in O(h_1) \; g \in O(h_2) \to f + g \in O(h_1 + h_2)$$

- Loops: (repetition of instructions)

$$f \in O(h_1), g \in O(h_2) \to f \cdot g \in O(h_1 \cdot h_2)$$

3 Θ, Big-Ω and the RAM model

O-notation is an **upper bound** for the runtime of an algorithm. What this means is that on any input of length n, the runtime is bounded by some function $O(f(n))$. If we say an algorithm has a runtime of $O(n^2)$, then the actual runtime could also be anything from $O(\log n), O(n)$ etc.

This is good because we get a worst case runtime, but to avoid ambiguities there are also other cool notations such as Θ-notation, where growth is precisely determined and Ω-notation, which gives us a lower bound.

Definition: Θ-notation.

Let $g : \mathbb{N} \to \mathbb{N}$ be a function. $\Theta(g(n))$ is the set of functions:

Θ(g(n)) $= \{f(n) :$ There exists positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$