Alin Tomescu | Week 1, Tuesday, February 4th, 2014 | Lecture 1
6.006 Intro to Algorithms | Prof. Srinivas Devadas | Prof. Nancy Lynch | Prof. Vinod Vaikuntanathan

Lecture 1: Peak finding

Course overview

- Efficient procedures for solving problems on large inputs.
- Asymptotic complexity
- Scalability (algorithm A is faster than algorithm B for an input size of 1 million, but not for 1 billion)
- Classic data structures
  - Binary balanced search trees
- Python programming language
- Flexible collaboration policy

Content

8 modules, each of them has a problem set associated with it

1. Algorithmic thinking (document distance)
2. Sorting, binary search trees (computing baseball statistics)
3. Hashing
4. Numerics
5. Graphs
6. Shortest paths
7. Dynamic programming
8. Advanced topics, complexity theory

Peak finding

Table 1: 1D peak finding illustration

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
a	b	c	d	e	f	g	h	i

Example of peaks:

- Position 2 is a peak iff $b \geq a$ and $b \geq c$.
- Position 9 is a peak iff $i \geq h$.
- Position 1 is a peak iff $a \geq b$.

Will a peak always exist? Yes. But, if we use $>$ instead of $\geq$, no (consider an array of all one's).

Straightforward algorithm

Just go through the array one by one and check that element $a[i]$ is a peak by comparing with $a[i-1]$ and $a[i+1]$.

$\Theta(n)$ complexity for "scan" algorithm regardless of where you start (left or right).

Divide and conquer algorithm

How do you decide when to go left or right when you are in the middle of the array?

- If $a\left[\frac{n}{2}\right] < a\left[\frac{n}{2}-1\right]$ then only look at the left half.
- Else if $a\left[\frac{n}{2}\right] < a\left[\frac{n}{2}+1\right]$ only look at the right half.
- Else $\frac{n}{2}$ position is a peak, because $a\left[\frac{n}{2}-1\right] \leq a\left[\frac{n}{2}\right] \geq a\left[\frac{n}{2}+1\right]$.

We always "climb the hill" when trying to find a peak, because we are guaranteed to find one. If we go downhill, we might not find a peak (there could be a lake at the bottom of the hill).

Algorithm correctness:

- Termination (algorithm won't go into an infinite loop)
- Safety (algorithm returns the correct result)

Algorithm complexity:

The recurrence for the amount of time $T(n)$ it takes to find a peak in an array of n elements is:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

Solve recurrence using your favorite technique:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1) = T\left(\frac{n}{8}\right) + \Theta(1) + \Theta(1) + \Theta(1) = T\left(\frac{n}{2^4}\right) + 4 \cdot \Theta(1) = \cdots$$
$$= T\left(\frac{n}{2^i}\right) + i \cdot \Theta(1)$$

Note that $T(1) = \Theta(1)$ because for a one-element array the peak is exactly that one element (trivial case).

We can see that for $i = \log_2 n$, $T\left(\frac{n}{2^i}\right)$ becomes $T\left(n/2^{\log_2 n}\right) = T\left(\frac{n}{n}\right) = T(1) = \Theta(1)$. So if we replace i in the above recurrence, we get:

$$T(n) = T\left(\frac{n}{2^i}\right) + i \cdot \Theta(1) = T\left(\frac{n}{2^{\log_2 n}}\right) + \log_2 n \cdot \Theta(1) = T(1) + \Theta(\log_2 n) = \Theta(1) + \Theta(\log_2 n) = \Theta(\log_2 n)$$

$$T(n) = \Theta(\log_2 n)$$

Two-dimensional peak finding

Table 2: 2D peak finding illustration

	b	
e	a	c
	d	

Definition: a is a $2D$ peak iff. $a \geq b \wedge a \geq c \wedge a \geq d \wedge a \geq e$.

Greedy ascent

Informal description: Pick a direction $(\leftarrow, \uparrow, \downarrow, \rightarrow)$ and if you see something greater, you follow it. If all values around are less, then you found the peak.

What is the complexity? $\Theta(nm)$

Table 3: An example of an n by m worst case scenario

1	2	3	4
0	0	0	5
13	14	0	6
12	0	0	7
11	10	9	8

Alin Tomescu | Week 1, Tuesday, February 4th, 2014 | Lecture 1
6.006 Intro to Algorithms | Prof. Srinivas Devadas | Prof. Nancy Lynch | Prof. Vinod Vaikuntanathan

Divide and conquer approach

Case 1: Extend 1D to 2D (no recursion)

Algorithm:

- Pick middle column $j = \frac{m}{2}$ and find a 1D peak at position (i, j).
- Use (i, j) as a start on row i to find a 1D peak on row i.

It's not going to work. Informally, this is because, after finding a peak at (i, j) on column j, when we look for a peak on row i, we will not compare it to the elements above it (the ones on row $i - 1$ and $i + 1$). We just *blindly* declare it a peak. But this peak on row i could not be peak.

To convince yourself, in the example below, we find **3** as the column peak and 4 as the row peak. But 4 is not a peak at all in the matrix (it has 7 above it).

		j		
	0	**1**	0	0
	0	**2**	7	0
i	*1*	***3***	*4*	*2*
	0	**1**	8	0

Case 2: Split on 1D peak (recursion)

Algorithm:

- Pick middle column $j = \frac{m}{2}$
- Find 1D peak on column j at position (i, j)
- Compare (i, j) with the two neighbours on the adjacent columns and decide which half to look into.
  - If $(i, j - 1) > (i, j)$ recurse into the left side
  - If $(i, j + 1) > (i, j)$ recurse into the right side
  - Else, (i, j) is a peak.

It's not going to work either because splitting on a column peak is a bad idea. You cannot guarantee that there will be a peak on the side you recurse on.

Table 4: Counter example to splitting on 1D peaks

20	16	10	0	0
22	21	20	0	0
23	24	10	0	0
24	25	40	30	0
26	27	28	*50*	40

Here we find 20 as the peak on the middle column then we find 26 as the peak on the next column. But 26 is not a peak in the matrix because it's surrounded by 24 and 27.

Case 3: Split on 1D maximum (recursion)

This is the same algorithm as in *Case 2* above, except on column j you find the *maximum*, not the peak. This will work.