CS180 Homework 7

Zhehao Wang 404380075 (Dis 1B)

May 18, 2016

1 Largest sum of contiguous subarray

The iterative algorithm's given in alg 1 (Kadane's algorithm). The algorithm uses the first non-negative element as the starting point, and iterates through the given array. Each time it considers adding the iterated element to the subarray, if adding the element does not cause the sum of the subarray to fall below 0. If otherwise, we can start the subarray again from the next non-negative element, until all elements are iterated. We record the current longest subarray as we go through the list, and return it after the traversal's done.

Algorithm 1 Kadane's algorithm

```
1: function subarraySumIterative(A)
2:     if all elements in A < 0 then
3:         return max(A)
4:     maxTillHere ← 0
5:     maxSoFar ← 0
6:     for 0 < i ≤ len(A) do
7:         maxTillHere ← max(0, maxTillHere + A_i)
8:         maxSoFar ← max(maxSoFar, maxTillHere)
9:     return maxSoFar
```

The recursive algorithm's given in alg 2, and is initiated by calling $subarraySumRecursive(A)$. This uses the same idea as above, only difference being that the input array A is traversed using tail recursion, with $maxSoFar$ variable returned in each recursion.

Algorithm 2 Kadane's algorithm (tail recursion)

```
1: function subarraySumRecursive(A)
2:     if all elements in A < 0 then
3:         return max(A)
4:     maxSum(A, 0)
5:     return maxSoFar
6: function maxSum(A, maxTillHere)
7:     if |A| = 1 then
8:         return max(0, A_1)
9:     maxTillHere ← max(0, maxTillHere + A_1)
10:    return max(maxSum(A_2...A_n, maxTillHere, maxSoFar), maxTillHere)
```

Time complexity: for both algorithms, initial check for if input array's all negative takes $O(n)$. Afterwards, each element in the input array is considered once. And in each loop body or recursive call the number of operations is constant. So both algorithms are $O(n)$.

Correctness: for all-negative array, the algorithm's correctness is obvious. For arrays with non-negative elements A, the algorithm's correctness is established by the conclusion: "let $S = A_m...A_n$ be the largest sum contiguous subarray in A, $\Sigma_{i=m}^{k} A_i \geq 0$ holds for $\forall k \in [m, n]$". The conclusion can be proved by contradiction: if a k exists such that $\Sigma_{i=m}^{k} A_i < 0$, then the subarray $S' = A_{k+1}...A_n$ would have a larger sum than S, which contradicts with S being the largest subarray.

Given this conclusion, the algorithm can carve out each satisfying subarray in one iteration, and record the largest.

2 Two largest numbers in an array

The algorithm's given in alg 3, and is initiated by calling $twoLargest(A)$. The idea is that given the input array, we construct a binary tree by each time comparing two elements a and b, and copy the larger one up (node labeled $max(a, b)$ being the parent of nodes labeled a and b), until there is only one element left. The largest element is at the top of the binary tree, and can be traced down to the bottom. The second largest element is the maximum among the sibling elements of those in the trace from top to bottom (for example, if the left child is copied up as the largest element, then the right child could be a candidate for the second largest).

Algorithm 3 Two largest numbers in an array

```
1:  function twoLargest(A)
2:      queue ← []
3:      largest ← maxWithEnqueue(A, queue)
4:      2^nd largest ← max(queue)
5:      return largest, 2^nd largest
6:  function maxWithEnqueue(A, queue)
7:      if |A| = 2 then
8:          return max(A_1, A_2)
9:      n ← len(A)
10:     left ← maxWithEnqueue(A_1...A_{n/2}, queue)
11:     right ← maxWithEnqueue(A_{n/2+1}...A_n, queue)
12:     if left > right then
13:         queue.enqueue(right)
14:         return left
15:     else
16:         queue.enqueue(left)
17:         return right
```

Number of steps: number of steps counts the number of comparisons. Initial construction of the binary tree takes n steps to find out the largest, and puts height of the tree, $\log n$, elements in the queue. The second largest is the largest element in the queue, and finding it takes $\log n$ steps. Thus the overall algorithm takes $n + \log n$ steps.

Fig 1 illustrates the number of steps: elements in the box are the input array; suppose the red element's the largest, then the second largest element could only be among the blue ones.

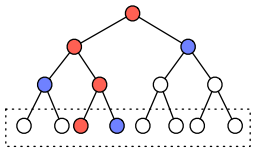

Figure 1: Sample binary tree for 8 elements

Correctness: the correctness for getting the largest element is obvious. The reason why the 2^{nd} largest element can only be among the same-trace siblings (blue elements in fig 1) is that each of the same-trace sibling is the largest in a sub tree, and all such subtrees combined makes the entire input array without the largest element. So the largest among these subtrees is the 2^{nd} largest in the original.

3 Element uniqueness lower bound and closest pair

Proof by contradiction: assume that there's an algorithm S for unbounded closest pairs that's $o(n \log n)$. We can then build an $o(n \log n)$ unbounded Element Uniqueness algorithm in the following way:

Consider an input of n elements $x_1...x_n$ for the unbounded Element Uniqueness problem, run S on the pairs $(x_1, x_1), ..., (x_n, x_n)$ takes $o(n \log n)$, and gives back pairs $(x_i, x_i), (x_j, x_j)$ as being closest. Then we calculate the distance between (x_i, x_i) and (x_j, x_j), if 0, not all elements in $x_1...x_n$ are distinct; otherwise all elements in $x_1...x_n$ are distinct. And we've an $o(n \log n)$ unbounded Element Uniqueness algorithm.

This contradicts with the given conclusion that unbounded Element Uniqueness is at least $O(n \log n)$, thus there's no algorithm for unbounded closest pair that's $o(n \log n)$.

4 Convex hull divide and conquer

The algorithm's given in alg 4. The idea's that we first sort the points by their X-axis, and divide them into two groups with about the same number of points in each. If either group has less than 4 points, we just connect all the points in the group. Otherwise, we call divide recursively (line 4 - 8).

Now assume that we have the convex hull of both the left and the right group, we want to merge the two by finding the top and bottom tangents[1] for both groups and discarding all points whose X-axis value falls in between. We find the tangents by starting with a line between pr, the leftmost point in the right group, and pl, the rightmost point in the left group. To find the lower tangent (line 9 - 17), we check if pr's counter-clockwise neighbor is below this line, if so, we drop pr and move pr to its counter-clockwise neighbor. Similarly, we check and move pl clockwise. Finding the upper tangent (line 19 - 20) is symmetric with the lower tangent.

With the tangents added and nodes in between dropped, we then return the remaining nodes in P in order, and we have a convex hull for the left and right groups combined.

[1] A tangent is a line connecting a point in the left group and a point in the right group, and crosses both the left and right convex hulls exactly once

Algorithm 4 Convex hull divide and conquer

```
1:  function CONVEX(P)
2:      convexHull(P.sortByX())
3:  function CONVEXHULL(P)
4:      n ← len(P)
5:      if n ≤ 3 then
6:          return connectAll(P)
7:      left ← convexHull(P_1...P_{n/2})
8:      right ← convexHull(P_{n/2+1}...P_n)
9:      pl ← rightMost(left)
10:     pr ← leftMost(right)
11:     while line(pl, pr).isTangent(left) = false do
12:         discard(pl)
13:         pl ← pl.nextClockwise()
14:     while line(pl, pr).isTangent(right) = false do
15:         discard(pr)
16:         pr ← pr.nextCounterClockwise()
17:     addLine(pl, pr)
18:
19:     while ... do
20:         ...similar with previous two loops, find upper tangent...
21:
22:     return those P that are not discarded
```

Time complexity: given the description, at worst we need to visit all n nodes exactly once in a merge of n nodes. During each visit, the check $isTangent$ is constant time (as we only need to check one neighbor). So the complexity is represented as $T(n) = 2T(\frac{n}{2}) + n$. And by master theorem (also, the same complexity representation as mergesort), the algorithm is $O(n \log n)$.

Correctness: the algorithm's correctness is obvious per the description of the merging process and definition of upper / lower tangents. As an example, figure 2 illustrates the correctness of finding the lower tangent and the merging process. (left 3 and right 4 nodes are being merged. When finding the lower tangent, the lines we tried are denoted as red dotted lines, and red nodes are discarded as we try those lines. The blue dotted lines are the tangents we find eventually; and all the blue lines, along with the remaining nodes, are outputted as a convex hull).

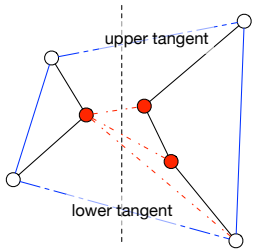

Figure 2: The merging process