

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. Otherwise, we can start the subarray again from the next non-negative element, until all elements are iterated. We record the current largest 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:    $\maxTillHere \leftarrow 0$ 
5:    $\maxSoFar \leftarrow 0$ 
6:   for  $0 < i \leq \text{len}(A)$  do
7:      $\maxTillHere \leftarrow \max(0, \maxTillHere + A_i)$ 
8:      $\maxSoFar \leftarrow \max(\maxSoFar, \maxTillHere)$ 
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:    $\maxTillHere \leftarrow \max(0, \maxTillHere + A_1)$ 
10:  return  $\max(\maxSum(A_2...A_n, \maxTillHere), \maxTillHere)$ 
```

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 array A with non-negative elements, the algorithm's correctness is established by the conclusion: "let $S = A_m \dots A_n$ be the largest contiguous subarray in A , $\sum_{i=m}^k A_i \geq 0$ holds for $\forall k \in [m, n]$ ". The conclusion can be proved by contradiction: if a k exists such that $\sum_{i=m}^k A_i < 0$, then the subarray $S' = A_{k+1} \dots A_n$ 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:    $queue \leftarrow []$ 
3:    $largest \leftarrow maxWithEnqueue(A, queue)$ 
4:    $2^{nd}largest \leftarrow max(queue)$ 
5:   return  $largest, 2^{nd}largest$ 
6: function MAXWITHENQUEUE( $A, queue$ )
7:   if  $|A| = 2$  then
8:     return  $max(A_1, A_2)$ 
9:    $n \leftarrow len(A)$ 
10:   $left \leftarrow maxWithEnqueue(A_1 \dots A_{\frac{n}{2}}, queue)$ 
11:   $right \leftarrow maxWithEnqueue(A_{\frac{n}{2}+1} \dots A_n, queue)$ 
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.

Algorithm 4 Convex hull divide and conquer

```
1: function CONVEX(P)
2:   convexHull(P.sortByX())
3: function CONVEXHULL(P)
4:    $n \leftarrow \text{len}(P)$ 
5:   if  $n \leq 3$  then
6:     return connectAll(P)
7:    $\text{left} \leftarrow \text{convexHull}(P_1 \dots P_{\frac{n}{2}})$ 
8:    $\text{right} \leftarrow \text{convexHull}(P_{\frac{n}{2}+1} \dots P_n)$ 
9:    $p_l \leftarrow \text{rightMost}(\text{left})$ 
10:   $p_r \leftarrow \text{leftMost}(\text{right})$ 
11:  while  $\text{line}(p_l, p_r).isTangent(\text{left}) = \text{false}$  do
12:    discard( $p_l$ )
13:     $p_l \leftarrow p_l.\text{nextClockwise}()$ 
14:  while  $\text{line}(p_l, p_r).isTangent(\text{right}) = \text{false}$  do
15:    discard( $p_r$ )
16:     $p_r \leftarrow p_r.\text{nextCounterClockwise}()$ 
17:  addLine( $p_l, p_r$ )
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 each time). 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).

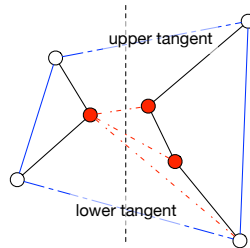


Figure 2: The merging process