Project 2 - Schreiber, Matt

Monday, February 03, 2014 12:05 AM

(1) Pseudocode and asymptotic running time:

```
A = array
i = length

algo1(A[1..n]):
    max <- 0
    for i <- 1 to n
        for j <- i to n
                sum <- 0
                for k <- i to j
                    sum <- sum + A[k]
                if sum > max
                    max <- sum
    return max
```

Analysis:

The outer loop will iterate n times, the middle loop will iterate at most n times, and the innermost loop will iterate at most n times. Because the loops are nested, the number of operations is at most n * n * n = n^3, which is bounded above by n^3, and the time complexity of algo1 is therefore O(n^3).

```
algo2(A[1..n])
    max <- 0
    for i <- 1 to n
        h <- A[i]
        for j <- i+1 to n
            h <- h + A[j]
            if h > max
                max <- h
    return max
```

Analysis:

The outer loop will iterate n times, and the inner loop will iterate at most n times (well, n-1 times). Because the loops are nested, the number of operations is at most n * (n-1) = n^2 -n, which is bounded above by n^2, and the time complexity of algo2 is therefore O(n^2).

```
algo3(A[1..n])
    max <- 0
// If the array is one unit long,
    // return that unit or 0, whichever
    // is larger.
    if n = 1
        if A[1] > 0
            return A[1]
else
        lmax <- 0
        rmax <- 0
        m <- floor(n/2)
// Get largest subarray starting
        // at index m-1 and heading toward
        // index 1
        lsum <- 0
        for i <- m-1 to 1
            lsum <- lsum + A[i]
            if lsum > lmax
                lmax <- lsum
// Get largest subarray starting
        // at index m and heading toward
        // index n (the final index)
        rsum = 0
        for j <- m to n
            rsum <- rsum + A[j]
            if rsum > rmax
                rmax <- rsum
// set mmax ('mid-max')
        // to lmax + rmax
        mmax <- lmax + rmax
// set lmax to the value returned
        // by calling algo3 recursively on
        // the first half of the array
        lmax <- algo3(A[1..m-1])
// set rmax to the value returned
        // by calling algo3 recursively on
        // the second half of the array
        rmax <- algo3(A[m..n])
// Find the largest of mmax, lmax,
        // and rmax; assign to max
        if mmax > lmax
            max <- mmax
        else if lmax > rmax
            max <- lmax
        else
            max <- rmax
return max
```

Analysis:
We see that the execution time T(n) for this algorithm fits the general form T(n) = aT(n/b) + f(n)^c where a and b are constants with a >= 1 and b > 1, and f(n) is a function of n. The base case (n = 1) running time is T(n) = theta(1). For the recursive case, the division of the array into two pieces is theta(1), the recursive 'conquer' step takes 2T(n/2) (since there are two recursive calls, each on an input size of n/2), and the two for-loops jointly take theta(n) (since each iterates over an input size of n/2). Finally, the comparison operations at the end take theta(1). Therefore, for all n > 1, T(n) = theta(1) + 2T(n/2) + theta(n) + theta(1) = 2T(n/2) + theta(n)^1. In this case, a = 2, b = 2, and f(n)^c = theta(n)^1. Because log2(2) = 1 = c, it follows by the Master Theorem that T(n) = theta(n^c * log(n)) = theta(n log(n)).

(2) Proof of Correctness for algo3

Let the sum of a contiguous subarray A[x..y] of an array of real numbers A[1..n] be denoted S(x, y), and let S(x, y) equal sum(p = x through y, A[p]). The maximum contiguous subarray of an array of real numbers A[1..n] is therefore the subarray S(x, y) such that, for all pairs of indices(i, j), S(i, j) <= S(x, y). Let P(k) denote the proposition that is true when algo3 locates the largest
contiguous subarray sum of an array A[1..k] of k real numbers.

To be proven (using strong induction): P(n) for all positive integers n

(Note: my algorithm does not solve for arrays of size 0, that is, arrays whose sole member is the empty set, and I do not consider empty subarrays. In addition, my algorithm disallows negative maximum sums; that is, if the largest contiguous subarray sum is < 0, the algorithm returns 0. This is in keeping with approaches/definitions I found while researching the problem; I hope it
was not mistaken. Finally, all references to 'subarray' should be understood to mean 'contiguous subarray' unless otherwise noted).

Base case: P(1).
Because there is only one element in the array, there is only one subarray; namely, the subarray whose sole member is that element. By definition, the largest subarray sum is therefore S(1, 1) = A[1]. Either the number at A[1] is non-negative, in which case it is returned by the algorithm, or it is not, in which case the algorithm returns 0.

Inductive hypothesis: P(k) for all k < n.
Inductive step:

Let m = floor(n/2)
There are three cases to consider:
(1) the maximum subarray sum is in the subarray of A[1..n] A' = A[1..m-1];
(2) the maximum subarray sum is in the subarray of A[1..n] A'' = A[m..n]; and
(3) the maximum subarray sum is in a subarray of A[1..n] which overlaps the two subarrays.
Because the first two cases concern arrays of length < n, we can conclude by the inductive hypothesis that algo3 will locate the maximum subarray sums of these subarrays when it recurses on them.

Lemma:
Let Q(m) denote the proposition that the second for-loop in algorithm three locates the largest subarray sum beginning at a given index and proceeding m indices toward the end of the array.
To be proven (by weak induction): Q(m) for all positive integers m.
Base case: Q(0).
Because the sequence consists of only one member, by definition the largest subarray sum is that member. Where s is an arbitrary index, if A[s+0] = A[s] > 0, the algorithm returns it; otherwise it returns 0.

Inductive Hypothesis: Q(m-1).
Inductive step:
By the inductive hypothesis, we can conclude that the algorithm has located the maximum subarray sum beginning at index s in subarray A[s..s+(m-1)]. Let this value be denoted j. By the law of the excluded middle, either j + A[s+m] > j or j + A[s+m] <= j. If the first, the algorithm returns j + A[s+m]; if the second, the algorithm returns j. Either way, it returns the largest subarray sum in A[s..s+m].

Mutatis mutandis, we can conclude that the lemma also holds for subarray sums calculated by descending rather than ascending the array's indices, i.e. those sums calculated by the first for-loop in algo3.

By the lemma, we can conclude that algo3 finds the largest subarray of A that begins at index m-1 and proceeds toward index 0, as well as the largest subarray that begins at index m and proceeds towards index n. Now, either (1) both of those subarray sums are positive, in which case the largest subarray sum which overlaps the two halves of A is the sum of those two positive sums, or (2) one of the subarray sums equals zero, in which case the subarray sums is the same as one of the subarrays found by recursing on A[1..m-1] and A[m..n], or (3) both are zero. Because algo3 finds the largest subarray in each of the three cases listed at the beginning of the inductive step, the comparison of those values at the end of algo3 which determines the largest of those three values is guaranteed to find the largest subarray in A[1..n]. This concludes the proof.

(3) Extrapolation and interpretation

Slopes

- algo1 - The results suggest a multiplicative constant of around 8.25 * 10^-7, so the slope should be about 2.50 * 10^-6 * x^2.
- algo2 - The results suggest a multiplicative constant of around 6.80 * 10^-6, so the slope should be about 1.36 * 10^-5 * x.
- algo3 - The results suggest a multiplicative constant of around 1.90 * 10^-4, so the slope should be about (1.90 * 10^-4)(log(x) + 1) (although my calculus is *seriously* rusty, so that -- or, for that matter, all of the foregoing -- could be way off).
- kadane - this is very hard to determine; it seems like the slope is effectively 0 for the input sizes I tried.

Maximum elements processed in one hour

- algo1(x) = 8.25 * 10^-7 * x^3 = 3600
    - x^3 = 3600 / (8.25 * 10^-7) = 4.3636E9

    - x = 1634
- algo2(x) = 6.80 * 10^-6 * x^2 = 3600
    - x^2 = 3600 / (6.80 * 10^-6) = 5.2941E8
    - x = 23009
- algo3(x) = 1.90 * 10^-4 * x * log(x) = 3600
    - x * log(x) = 3600 / (1.90 * 10^-4) = 1.8947E7
    - log(10^x) * log(x) = 1.8947E7
    - Aaaand here I'm stuck. :)