Section 2: Divide-and-Conquer Algorithms (Continued)

Amy Zhao

February 02, 2024

1 Recapping Divide and Conquer Algorithms

You can find supplemental slides posted on Canvas which recaps divide and conquer algorithms covered in lecture.

2 Array Balance Point

We define the balance point of a sorted list of positive integers $A[1, \ldots, n]$ as the greatest index p such that:

$$\sum_{i=1}^{p-1} A[i] \leq \sum_{i=p}^{n} A[i]. \tag{1}$$

Suppose the array that we are given is out of order, but we want to find the balance point if the array were sorted. We could sort the list first before the balance point, but this may take $\Omega(n \log n)$ steps. Let's try to find an algorithm that finds the balance point of A in $O(n)$ steps.

We can compute the balance point via a modified binary search algorithm. We first compute the total sum T of the elements in A. The general idea is to maintain an interval of indices containing the balance point and modify A so that $A[\ell, \ldots, r]$ contains the ℓ-th to r-th smallest elements. We also keep track of the partial sum S of the first to the $(\ell - 1)$-th smallest elements of A.

In iteration i, we compute the median element m_i of $A[\ell, \ldots, r]$ and partition around m_i. This means that the elements before m_i are at most m_i, and the elements after m_i are at least m_i. We then compute the sum S_i of elements that precede m_i which

takes $O(r - \ell)$ computation time. We halve our search area by comparing $S + S_i$ with $\frac{T}{2}$ to ensure we retain the balance point. S is updated accordingly, and the algorithm terminates when the interval $[\ell, \ldots, r]$ contains a single index.

```
   Input  : Array A of unsorted positive integers
   Output: Balance point index
 1 Function BalancePoint(A):
 2     ℓ ← 1, r ← n ;
 3     S ← 0 ;
 4     T ← Sum(A[1, ..., n]) ;
 5     while ℓ < r do
 6         m_i ← GetMedian(A, ℓ, r) ;
 7         PartitionPivot(A, ℓ, r, m_i) ;
 8         mid ← ⌊(ℓ+r)/2⌋ ;
 9         S_i ← Sum (A[ℓ, ..., mid]) ;
10         if S + S_i ≤ T/2 then
11             S ← S + S_i ;
12             ℓ ← mid + 1;
13         else
14             r ← mid
15     end
16     return ℓ
```

The proof of correctness follows a loop invariant. We claim that at each iteration, $[\ell, r]$ contains the balance point, $A[\ell, \ldots, r]$ are the ℓ-th to the r-th smallest elements of A, and S is the sum from the first to the $(\ell - 1)$-th smallest elements. Evidently, this is true before the start of the loop.

Suppose the loop variant holds prior to the start of the iteration on $A[\ell, \ldots, r]$. The algorithm computes the median m_i and modifies $A[\ell, \ldots, r]$ so that every element left of m_i is less than or equal to m_i, and everything right of m_i is greater than or equal to m_i. Using the calculated S_i (sum of values before and including the median), the algorithm accurately "binary searches" on the correct portion of the array.

- If $S + S_i \leq \frac{T}{2}$, the balance point occurs after the median.
- If $S + S_i > \frac{T}{2}$, the balance point occurs before the median.

When $[\ell, r]$ contains multiple indices, it is still feasible to "check" and see if it's possible to add more to S and still maintain the balance required by the problem.

At the end of the loop, the invariant properties stated above still hold. When the interval contains a single index, we can't continue searching, so we return the balance point of A.

Each iteration computes the median which takes $O(r - \ell)$ time, and partitions the elements around the median in $O(r - \ell)$ operations. With binary search, there are at most $O(\log n)$ iterations, so the total runtime is bounded by:

$$T(n) \leq T\left(\frac{n}{2}\right) + O(n) \leq O(n). \tag{2}$$

3 Searching Matrices

Consider an integer-valued matrix where the values in each row are sorted in ascending order from left to right, and values in each column are sorted in ascending order from top to bottom. One such matrix is:

$$M = \begin{bmatrix} 1 & 4 & 7 & 11 & 15 \\ 2 & 5 & 8 & 12 & 19 \\ 3 & 6 & 9 & 16 & 22 \\ 10 & 13 & 14 & 17 & 24 \\ 18 & 21 & 23 & 26 & 30 \end{bmatrix}.$$

We would like to create a divide-and-conquer algorithm to find a target element t because searching the entire array is too time consuming. The overall idea is to partition the matrix into four submatrices. We claim that the target element is possibly contained in two of the four submatrices.

Assuming the matrix is non-empty, it's obvious that it doesn't contain the target element t if t is smaller than the value in the top-left corner or larger than the value in the bottom-right corner. If this isn't the case, target t is potentially contained within the matrix. We search along the matrix's middle column for a row index such that:

$$M[\text{row} - 1][\text{mid}] < t < M[\text{row}][\text{mid}].$$

If we don't find the target in this process, we partition the matrix into four submatrices. The top-left and bottom-right submatrices cannot contain t, so we exclude them from the search space before recursively applying the algorithm to the bottom-left and top-right submatrices.

```
Input  : Matrix M and target t
Output: True if t exists in M; False otherwise
Function SearchMatrix(M, t):
    Function RecursiveSearch(left, up, right, down):
        if left > right or up > down then
            return False
        else if t < M[up][left] or t > M[down][right] then
            return False
        mid ← ⌊(left+(right−left))/2⌋;
        row ← up;
        while row ≤ down and M[row][mid] ≤ t do
            if M[row][mid] == t then
                return True
            row ← row + 1;
            return RecursiveSearch(left, row, mid − 1, down) or
             RecursiveSearch(mid + 1, up, right, row − 1)
        end
    return RecursiveSearch(0, 0, len(M[0]) − 1, len(M) − 1)
```

We can see pretty clearly that there are two recursive calls in each iteration, and each recursive call searches on a quarter of the original matrix. The work required outside of these recursive calls is the work done to find a partition point along a $O(n)$ length column; the runtime is $\sqrt{n}$. Thus, the total runtime is given by:

$$T(n) \leq 2T\left(\frac{n}{4}\right) + \sqrt{n} \Longrightarrow O(n \log n). \tag{3}$$

4 Application of FFT: String Matching

Suppose we have a substring $p = p_{m-1}p_{m-2}\dots p_0$, and we want to return the indices where it matches within a larger string $s = s_{n-1}s_{n-2}\dots s_0$. For example, if $s = ababaababbaaababababaaab$ and $p = baba$, the output should tell us that the substring occurs three times at starting positions 1, 13, and 15. With a brute force method, we can compare the m-character substring against all possible $n - (m-1)$ starting positions in the longer string. The problem is that this is extremely efficient, with a time complexity of $O(m(n-m+1))$. We can probably do better…

Define the mapping $f : \{a, b\} \to \{-1, 1\}$ where $f(a) = 1$ and $f(b) = -1$. For a

substring of s of length m, the inner product:

$$f(s_k \dots s_{k+m-1}) \cdot f(p_0 \dots p_{m-1}) = \sum_{i=1}^{m-1} f(s_{k+i} f(p_i))$$

is equal to m if and only if $s_k \dots s_{k+m-1} = p_0 \dots p_{m-1}$.

We set up two polynomials:

$$\begin{aligned} s(x) &= f(s_0) + f(s_1)x + \cdots + f(s_{n-1})x^{n-1}, \\ p(x) &= f(p_{m-1}) + f(p_{m-2})x + \cdots + f(p_0)x^{m-1}. \end{aligned}$$

The product of the polynomials will have coefficients $C_k = x^{k+m-1}$ equal to the sum $\sum_{i=0}^{m-1} s_{k+i} p_i$; $C_k = m$ if and only if $s_k = p_0$, $s_{k+1} = p_1$, $\dots$, $s_{k+m-1} = p_{m-1}$. Places where $C_k = m$ will indicate positions where the substring matches the input string.

Continuing the example presented previously, the two polynomials are:

$$\begin{aligned} s(x) = 1 - x + x^2 - x^3 + x^4 + x^5 - x^6 + x^7 - x^8 - x^9 + x^{10} + x^{11} \\ + x^{12} - x^{13} + x^{14} - x^{15} + x^{16} - x^{17} + x^{18} + x^{19} + x^{20} - x^{21}, \\ p(x) = 1 - x + x^2 - x^3. \end{aligned}$$

The x terms where the coefficients are equal to 4 are x^4, x^{16}, and x^{18}.

FFT affords us an efficient manner to calculate the product of polynomials, yielding a much better algorithm than the brute force method discussed first.