CS5050 Advanced Algorithms

Spring Semester, 2018

Assignment 2: Divide and Conquer

Philip Nelson: worked with Raul Ramirez, Ammon Hepworth, Kyle Hovey

Due Date: 3:00 p.m., Thursday, Feb. 8, 2018 (**at the beginning of CS5050 class**)

1. Let $A[1 \cdots n]$ be an array of n *distinct* numbers (i.e., no two numbers are equal). If $i < j$ and $A[i] > A[j]$, then the pair $(A[i], A[j])$ is called an *inversion* of A.

   (a) All of the inversions are: $\{(4,2),(4,1),(2,1),(9,1),(9,7)\}$

   (b) What array with elements from the set $\{1, 2, \ldots, n\}$ has the most inversions?

   - The array $\{n, \ldots, 2, 1\}$ has the most inversions.
   - It would have $\frac{n(n-1)}{2}$ inversions.

   (c) Give a divide-and-conquer algorithm that computes the number of inversions in array A in $O(n \log n)$ time.

   1. Algorithm Description The *count_inversions* algorithm counts the number of inversions of the array A in $O(n \log n)$ making use of a divide and conquer technique similar to merge sort. The divide step is exactly the same as merge sort. Each time the *mid* is calculated and the functions is recursively called from *start* to *mid* and *mid+1* to *end*. The inversions are counted during the combine step. When $A[i] < A[j] \wedge i > j$, an inversion is counted. However, since the array is being sorted, we can take advantage of the sorted property. If $A[i] < A[j] \wedge i > j$ then all the elements, $\{A[i], A[i+1], \cdots, A[mid]\}$ are also inversions; that is $(mid - i + 1)$ inverisons. Therefore each time we encounter the situation $A[i] < A[j] \wedge i > j$, we add $mid - i + 1$ to the inversion count. This allows the combine phase to complete in $O(n)$ time.

   2. Pseudocode Count Inversions

```
int mergeList(std::vector<int>& A, int start, int mid, int end)
{
  int inversions = 0;
  std::vector<int> merged;
  int i = start;
  int j = mid + 1;

  while (i <= mid && j <= end)
  {
    if (A[i] < A[j])
      merged.push_back(A[i++]);
      else
      {
        if (i < j)
          inversions += mid - i + 1;

        merged.push_back(A[j++]);
      }
    }

    while (i <= mid)
      merged.push_back(A[i++]);

    while (j <= end)
      merged.push_back(A[j++]);

    for (unsigned int x = start, y = 0; y < merged.size(); ++x, ++y)
      A[x] = merged[y];

    return inversions;
  }

  int mergeSort(std::vector<int>& A, int start, int end)
  {
    if (end - start == 0)
      return 0;

    auto mid = (start + end) / 2;

    return mergeSort(A, start, mid) +
      mergeSort(A, mid + 1, end) +
      mergeList(A, start, mid, end);
  }
```

3. **Time Analysis** The algorithm *count_inversions* counts the inversions of matrix A in $O(n \log n)$ time. It divides the problem $\log n$ times then sorts A. Inorder to count the inversion, it must do a constant ammout of work. If there is ever the case where $A[i] < A[j] \wedge i > j$. This can happen at most n times, if the array is in reverse sorted order. Here the algorithm will do a constant amount of extra work n times which is still n. Therefore the algorithm executes in $n \log n$.

2. To find the number of elements of in A that are less than each element of B we will employ divide and conquer algorithm. However we will not divide A in half until there is only one element in each sub array as this would by $\log n$ operations. We will divide A m times and each element of B is associated with one of the sub arrays of A. Each a_i is either larger or smaller than the element of B it is associated with and the relationship is kept track of. This first m work is only done once, so far m work has been done. Now we begin to sort the elements of B with a merge sort. This involves doing $\log m$ work to divide B. During the conquer step, if elements b_i and b_j are swapped, the larger elements of b_i are compared with b_j and the smaller elements of b_j are compared to b_i. The associations are changed so at the end, all elements of A are sorted relative to the elements of B, and B is sorted. Therefore, to count the number of elements in A that are less than any b_i, all that is required is to count the elements less than b_i and the elements associated with the smaller elements of B.

   This algorithm is $O(m + n \log m)$ which is $O(n \log m)$.

   Note: You will receive the full 30 points if you give an $O(a \log b)$ time algorithm directly for (c) without giving any algorithms for (a) or (b).

3. **(20 points)** Solve the following recurrences (you may use any of the methods we studied in class). Make your bounds as small as possible (in the big-O notation). For each recurrence, $T(n)$ is constant for $n \le 2$.

   (a) $T(n) = 2 \cdot T(\frac{n}{2}) + n^3$.
   $a = 2,\ b = 2,\ f(n) = n^3$
   $n^3 = n^{\log_2 2}$
   case 3:
      i. $n^3 = \Omega(n^{1+\epsilon}),\ \epsilon > 2$
      ii. $2 \cdot \left(\frac{n}{2}\right)^3 \le cn^3 \Rightarrow \frac{1}{4}n^3 \le cn^3,\ \frac{1}{4} \le c < 1$

   $$\boxed{T(n) = \Theta\left(n^3\right)}$$

   (b) $T(n) = 4 \cdot T(\frac{n}{2}) + n\sqrt{n}$.
   $a = 4,\ b = 2,\ f(n) = n^{\frac{3}{2}}$
   $n^{\frac{3}{2}} = n^{\log_2 4}$
   case 1:
   $n^{\frac{3}{2}} = O\left(n^{\frac{4}{2}-\epsilon}\right),\ 0 < \epsilon < \frac{1}{2}$

   $$\boxed{T(n) = \Theta\left(n^2\right)}$$

   (c) $T(n) = 2 \cdot T(\frac{n}{2}) + n \log n$.
   $a = 2,\ b = 2,\ f(n) = n \log n$
   $n \log n = \Theta(n^{\log_2 2} log^k n)$
   If $f(n) = \Theta(n^{\log_b a} \log^k n),\ k \ge 0 \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
   $n \log n = \Theta(n^{\log_2 2} log^k n),\ k = 1$

$$\boxed{T(n) = n\log^2 n}$$

(d) $T(n) = T(\frac{3}{4} \cdot n) + n.$

$a = 1,\ b = \frac{4}{3},\ f(n) = n$

$n = n^{\log_{\frac{4}{3}}(1)}$

case 3:

i. $n = \Omega(n^{0+\epsilon}),\ 0 < \epsilon < 1$

ii. $1 \cdot \frac{n}{4/3} \leq cn \Rightarrow \frac{3}{4} \cdot n \leq cn,\ \frac{3}{4} < c < 1$

$$\boxed{T(n) = \Theta(n)}$$

4. **(20 points)** The maximum gains can be found by using a modified maximum sub-array, which is a divide and conquer algorithm. The major difference is that the algorithm looks at the difference between the upper and lower values instead of the sum. The algorithm begins by dividing the array of days into two sub-arrays, then calling itself on both sub arrays. The base case is when there is only one day in the array. This case is trivial, the gains are zero. On the combine step, the algorithm will check the gains fo the right and left to determine wheter the max gains are on the left or right side. However the max gains could be across the span of the two sets. To check the spanning gains, it will find the max of the right array and the min of the left. The max of the right, left and spanning gains will be returned. Since this process is executed recursively, it return the max gains of the whole array.

   The max gains algorithm is $O(n \log n)$. It divides the array of days $\log n$ times and it does up to n work on the combine step. In the worst case, the max and min could be spanning and at the end and beginning of the array, respectively. This would require n work to find.