9 Algorithm Design. Divide and Conquer

9.1 Objectives

The learning objectives of this laboratory session are:

- To understand the challenges of implementing *divide and conquer* algorithm development method.
- To get hands-on experience with implementing divide and conquer algorithms.
- To develop critical thinking concerning implementation decisions for algorithms which use a divide and conquer approach.

The outcomes for this session are:

- Improved skills in C programming.
- A clear understanding of divide and conquer algorithms.
- The ability to make implementation decisions concerning divide and conquer algorithms.
- The ability to decide when divide and conquer is suitable for solving a problem.

9.2 Brief Theory Reminder

Divide and Conquer Strategy

Divide and conquer means repeatedly dividing a problem into two or more subproblems of the same type, and then recombining the solved subproblems in order to get a solution of the problem.
Let $A = (a_1, a_2, \ldots, a_n)$ be a vector whose components are processed. Divide an conquer is applicable if for any p and q, natural numbers such that $1 \leq p < q \leq n$ we can find a number $m \in [p+1, q-1]$ such that processing the sequence $a_p, a_{p+1}, \ldots, a_q$ can be achieved by processing sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_m, a_{m+1}, \ldots, a_q$, and then combining the results. Briefly, divide and conquer may be sketched as illustrated by Listing 9.1

Listing 9.1: A sketch of Divide and Conquer strategy.

```
void DivideAndConquer(int p, int q, SolutionT α)
/* p and q are indices in the processed sequence; α is the solution */
{
        int ϵ, m;
        SolutionT β, γ; // intermediate results

        if ( abs(q - p) ≤ϵ) process(p, q, α);
        else
        {
                Divide(p, q, m);
                DivideAndConquer(p, m, β);
                DivideAndConquer(m + 1, q, γ);
                Combine( β, γ, α);
        }
}
```

Invocation: `DivideAndConquer(1, n, α)`

In Listing 9.1:

ϵ is the maximum length of a sequence $a_p, a_{p+1}, \ldots, a_q$ which can be directly processed;
m is the intermediate index where we can split the sequence $a_p, a_{p+1}, \ldots, a_q$;
β and γ are the intermediate results obtained by processing sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_{m+1}, \ldots, a_q$, respectively;
α is the result of combining intermediate solutions β and γ;
`Divide` splits the sequence $a_p, a_{p+1}, \ldots, a_q$ into sequences $a_p, a_{p+1}, \ldots, a_m$ and $a_{m+1}, \ldots, a_q$;
`Combine` combines solutions β and γ obtaining the final solution, α.

A Divide and Conquer Example. Mergesort A Vector of n Elements

Mergesorting a vector of n elements is implemented in Listing 9.2. Mergesort uses divide and conquer as follows:

1. **Divide** by finding the index mid of the position midway between $lBound$ and $rBound$: add $lBound$ and $rBound$, divide by 2, and round down.
2. **Conquer** by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray $A[lBound..mid]$ and recursively sort the subarray $A[mid+1..rBound]$.
3. **Combine** by merging the two sorted subarrays back into the single sorted subarray $A[lBound..rBound]$]. In Listing 9.2, the combine step is achieved by `merge`.

Listing 9.2: Merge sorting a vector of n elements.

```
#include <stdio.h>
#define MAXN 100
int A[MAXN]; /* vector to sort */

void printVector(int n)
/* print vector elements - 10 on one line */
{
  int i;

  printf( "\n");
  for(i = 0; i < n; i++)
  {
     printf("%5d", A[i]);
     if((i + 1) % 10 == 0)
       printf("\n");
  }
  printf("\n");
}
void merge(int lBound, int mid, int rBound)
{
  int i, j, k, l;
  int B[MAXN]; /* B = auxiliary vector */

  i = lBound;
  j = mid + 1;
  k = lBound;
  while(i <= mid && j <= rBound)
  {
    if(A[i] <= A[j])
    {
      B[k] = A[i];
      i++;
    }
    else
    {
      B[k] = A[j];
      j++;
    }
    k++;
  }
  for ( l = i; l <= mid; l++)
  { /* there are elements on the left */
    B[k] = A[l];
    k++;
  }
  for ( l = j; l<= rBound; l++)
  { /* there are elements on the right */
    B[k] = A[l];
    k++;
  }
  /* sequence from index lBound to rBound is now sorted */
  for(l = lBound; l <= rBound; l++)
    A[l] = B[l];
}

void mergeSort(int lBound, int rBound)
{
  int mid;

  if(lBound < rBound)
  {
    mid= ( lBound + rBound) / 2;
    mergeSort(lBound, mid);
    mergeSort(mid + 1, rBound);
```

```
    merge(lBound, mid, rBound);
  }
}
int main()
{
  int i, n;

  printf("\nNumber of elements in vector=");
  scanf( "%d", &n);
  while ( '\n' != getchar());
  printf("\nPlease input vector elements\n");
  for(i = 0; i < n; i++)
  {
    printf( "a[%d]=", i);
    scanf( "%d", &A[i]);
  }
  printf("\nUnsorted vector\n");
  printVector(n);
  mergeSort(0, n-1);
  printf("\nSorted vector\n");
  printVector(n);
  while ( '\n' != getchar());
  return 0;
}
```

Another Example of Divide and Conquer Algorithm. Finding the 2D Closest Points

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

$$||pq|| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The Brute force solution is $O(n^2)$, compute the distance between each pair and return the smallest. We can calculate the smallest distance in $O(n \log^2 n)$ time using Divide and Conquer strategy.

1. Sort the input array of points by their x-coordinates
2. Find the middle point in the sorted array, we can take $P[n/2]$ as middle point.
3. Divide the given array in two halves. The first subarray contains points from $P[0]$ to $P[n/2]$. The second subarray contains points from $P[n/2 + 1]$ to $P[n - 1]$.
4. Recursively find the smallest distances in both subarrays. Let the distances be d_l and d_r. Find the minimum of d_l and d_r, i.e., $d = \min(d_l, d_r)$. Figure 9.1 illustrates this.

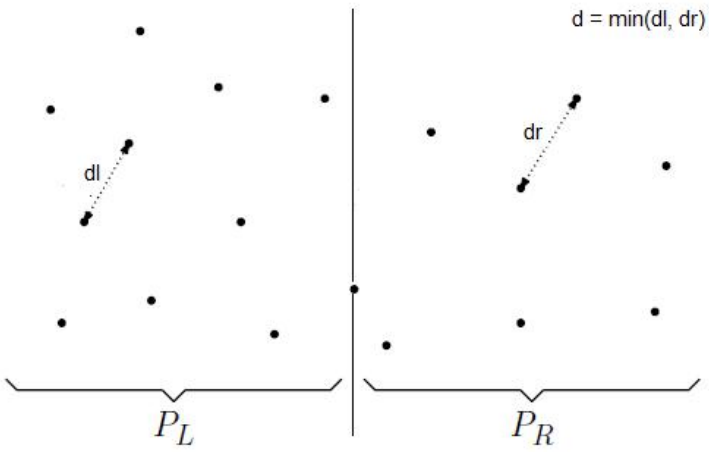

Figure 9.1: Closest points. Dividing the plane.

5. From above 4 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from left half and other is from right half. Consider the vertical line passing through $P[n/2]$,

and find all points whose x-coordinate is closer than d to the middle vertical line. Build an array `strip[]` of all such points. (See Figure 9.2.)

6. Sort the array `strip[]` according to y coordinates. This step is $O(n \log n)$. It can be optimized to $O(n)$ by recursively sorting and merging.

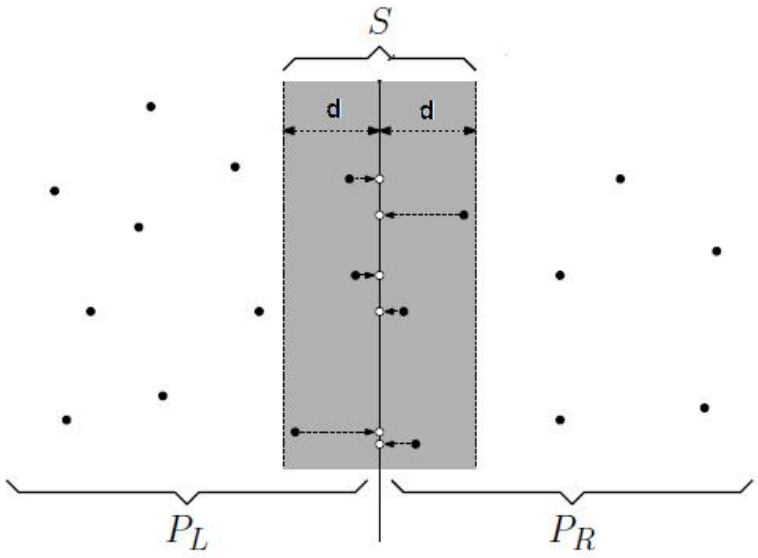

Figure 9.2: Closest points. Building an array `strip` (S).

7. Find the smallest distance in `strip[]`. It can be proved geometrically that for every point in strip, we only need to check at most 7 points after it (note that strip is sorted according to y coordinate).
8. Return the minimum of d and distance calculated in above step.

Listing 9.3 shows an implementation of this algorithm.

Listing 9.3: A 2D closest points implementation.

```
#include <stdio.h>
#include <stdlib.h>

// A divide and conquer program in C/C++ to find the smallest distance from a
// given set of points.

#include <stdio.h>
#include <float.h>
#include <stdlib.h>
#include <math.h>

// A structure to represent a PointT in 2D plane
typedef struct
{
    int x, y;
} PointT;

// Needed to sort array of points according to X coordinate
int compareX(const void* a, const void* b)
{
    return ((PointT *)a)->x - ((PointT *)b)->x;
}
// Needed to sort array of points according to Y coordinate
int compareY(const void* a, const void* b)
{
    return ((PointT *)a)->y - ((PointT *)b)->y;
}
// Returns the distance between two points
float dist(PointT p1, PointT p2)
{
    return sqrt( (p1.x - p2.x)*(p1.x - p2.x) +
                 (p1.y - p2.y)*(p1.y - p2.y)
           );
```

```
}
// Returns the smallest distance between two points in array P[] of size n
float bruteForce(PointT P[], int n)
{
    float min = FLT_MAX;
    for (int i = 0; i < n; ++i)
        for (int j = i+1; j < n; ++j)
            if (dist(P[i], P[j]) < min)
                min = dist(P[i], P[j]);
    return min;
}
// Find minimum of two float values
float min(float x, float y)
{
    return (x < y)? x : y;
}
// Find the distance beween the closest points of a strip of given size.
// All points in strip[] are sorted accordint to their y coordinate.
// They all have an upper bound on minimum distance as d.
float stripClosest(PointT strip[], int size, float d)
{
    float min = d;  // Initialize the minimum distance as d

    qsort(strip, size, sizeof(PointT), compareY);

    // Pick all points one by one and try the next points till the difference
    // between y coordinates is smaller than d.
    // This is a proven fact that this loop runs at most 6 times
    for (int i = 0; i < size; ++i)
        for (int j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)
            if (dist(strip[i],strip[j]) < min)
                min = dist(strip[i], strip[j]);

    return min;
}
// Find the smallest distance. The array P contains
// all points sorted according to x coordinate
float closestUtil(PointT P[], int n)
{
    // If there are 2 or 3 points, then use brute force
    if (n <= 3)
        return bruteForce(P, n);

    // Find the middle PointT
    int mid = n/2;
    PointT midPoint = P[mid];

    // Consider the vertical line passing through the middle PointT
    // calculate the smallest distance dl on left of middle PointT and
    // dr on right side
    float dl = closestUtil(P, mid);
    float dr = closestUtil(P + mid, n-mid);

    // Find the smaller of two distances
    float d = min(dl, dr);

    // Build an array strip[] that contains points close (closer than d)
    // to the line passing through the middle PointT
    PointT strip[n];
    int j = 0;
    for (int i = 0; i < n; i++)
        if (abs(P[i].x - midPoint.x) < d)
            strip[j] = P[i], j++;

    // Find the closest points in strip.  Return the minimum of d and closest
    // distance is strip[]
    return min(d, stripClosest(strip, j, d));
}

// The main function that finds the smallest distance
// This method mainly uses closestUtil()
float closest(PointT P[], int n)
{
    qsort(P, n, sizeof(PointT), compareX);
```

```
    // Use recursive function closestUtil() to find the smallest distance
    return closestUtil(P, n);
}
int main()
{
    PointT P[] = {{2, 3}, {12, 30}, {40, 50}, {5, 1}, {12, 10}, {3, 4}};
    int n = sizeof(P) / sizeof(P[0]);
    printf("The smallest distance is %f ", closest(P, n));
    return 0;
}
```

9.3 Laboratory Assignments

Mandatory Assignments

Implement modular C programs having their input and output stored in files, with names given as command line arguments, which solve the following problems using divide and conquer:

9.3.1. Given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.
I/O description. Input. One line of space-separated integers, where the first integer is the number of elements in the sequence.
Output. The value of maximum subarray sum.
For example, if the given array is $\{-2, -5, 6, -2, -3, 1, 5, -6\}$, then the maximum subarray sum is 7, and the elements of the sybarray are $\{6, -2, -3, 1, 5\}$.

9.3.2. Given n days' stock price $P[1..n]$, you want to make the most profit by buying a fixed number of shares at day b and later selling at day s. That is, you desire to get the maximum profit that you can earn by buying and selling within the period of $[b, s]$. Note that you must first buy then sell.
I/O description. Input. One line of space-separated values, where the first is an integer, n, the number of stock prices in the sequence. The rest of the values are `double`s.
Output. The value of maximum profit.

Extra Credit Assignments

9.3.3. A sequence $A = \langle a_1, ..., a_n \rangle$ of length $n \geq 3$ is called *unimodal* if $\exists p$ with $1 \leq p \leq n$ such that $a_1 < a_2 < \ldots < a_p$ and $a_p > a_{p+1} > \ldots > a_n$; in other words, the sequence increases up to an index p and then decreases. The element a_p is called its *top*. Write and implement a divide and conquer algorithm that, given a vector that stores a unimodal sequence, finds its *top* with cost $O(\log n)$.
I/O description. Input. One line of space-separated integers, where the first integer is the number of elements in the sequence.
Output. Value of the *top*.

Optional Assignments

9.3.4.
The Towers of Hanoi. There are three pegs: A, B and C. Initially, peg A has on it a number of disks, the largest at the bottom and the smallest at the top, as the figure 9.3 shows. The task is to move the disks, one at a time, from peg to peg, never placing a larger disk on top of a smaller one, ending with all disks on B.
I/O description. Input: a positive integer, the initial number of disks on peg A.
Output: consecutive configurations. **E.g.** for 6 disks, the initial configuration would be:

```
A: 1 2 3 4 5 6
B:
C:
```

Here the numbers represent disk sizes, the lowest number indicating the smallest disk.

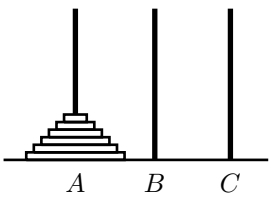

Figure 9.3: Initial position in "towers of Hanoi".

9.3.5. An n-element vector of integers, sorted in ascending order is given. Write a function to return the position of any element.
I/O description. Input: The elements of a vector, separated by spaces on the first line; the value of the element which we are looking for, on the second line. Output: number indication position (in the range $1..n$) or 0 if not found.