EXPERIMENT NO.4:- Optimal Merge Pattern

Reg no.:-24141045

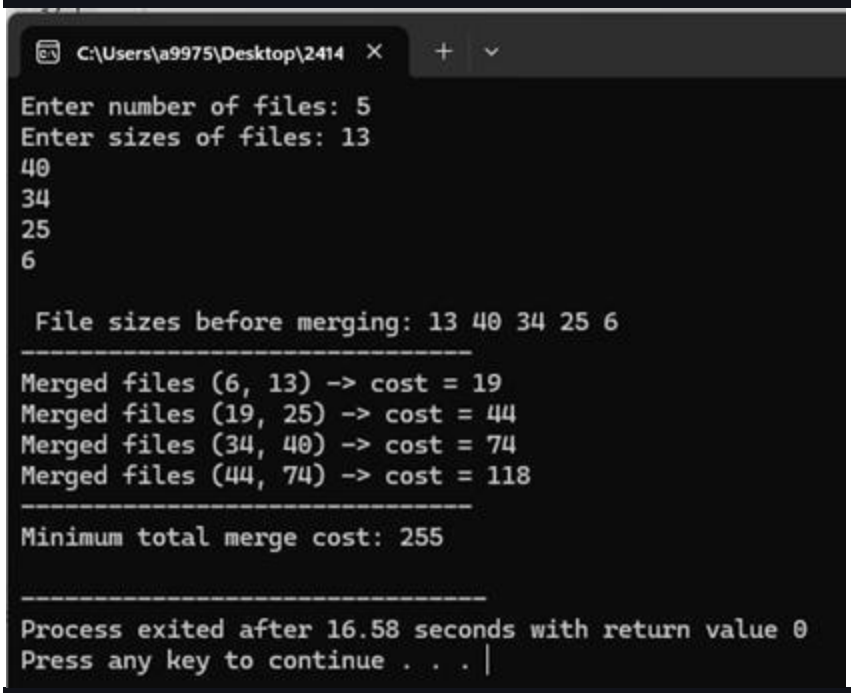
Program:-

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
void sort(int arr[], int n) {
            int i,j;
   for ( i = 0; i < n - 1; i++) {
      for ( j = 0; j < n - i - 1; j++) {
         if (arr[j] > arr[j + 1]) {
            swap(&arr[j], &arr[j + 1]);
         }
      }
   }
}
int optimalMerge(int files[], int n) {
   int totalCost = 0;
   while (n > 1) {
            int i;
      sort(files, n);
      int first = files[0];
      int second = files[1];
      int merged = first + second;
```

```
        totalCost += merged;
        printf("Merged files (%d, %d) -> cost = %d\n", first, second, merged);
        files[0] = merged;
        for ( i = 1; i < n - 1; i++) {
            files[i] = files[i + 1];
        }

        n--;
    }
    return totalCost;
}
int main() {
    int n,i;
    printf("Enter number of files: ");
    scanf("%d", &n);
    int files[n];
    printf("Enter sizes of files: ");
    for ( i = 0; i < n; i++) {
        scanf("%d", &files[i]);
    }
    printf("\n File sizes before merging: ");
    for ( i = 0; i < n; i++) printf("%d ", files[i]);
    printf("\n-------------------------------\n");
    int minCost = optimalMerge(files, n);
    printf("-------------------------------\n");
    printf("Minimum total merge cost: %d\n", minCost);
    return 0;
}
```

Output:-



Application:-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
typedef struct {
   char name[50];
   int duration;
} Song;
void swapInt(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
void swapSong(Song *a, Song *b) {
   Song temp = *a;
   *a = *b;
   *b = temp;
```

```
}
void heapify(Song arr[], int n, int i) {
    int smallest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;
    if (left < n && arr[left].duration < arr[smallest].duration)
        smallest = left;
    if (right < n && arr[right].duration < arr[smallest].duration)
        smallest = right;
    if (smallest != i) {
        swapSong(&arr[i], &arr[smallest]);
        heapify(arr, n, smallest);
    }
}
Song extractMin(Song arr[], int *n) {
    Song root = arr[0];
    arr[0] = arr[*n - 1];
    (*n)--;
    heapify(arr, *n, 0);
    return root;
}
void insertMinHeap(Song arr[], int *n, Song key) {
    (*n)++;
    arr[*n - 1] = key;
    int i = *n - 1;
    while (i != 0 && arr[(i-1)/2].duration > arr[i].duration) {
        swapSong(&arr[i], &arr[(i-1)/2]);
        i = (i-1)/2;
    }
}
int mergePlaylists(Song playlists[], int n) {
```

```
    int totalMergeDuration = 0;
    int i;
    for ( i = n/2 - 1; i >= 0; i--)
        heapify(playlists, n, i);
    while (n > 1) {
        Song first = extractMin(playlists, &n);
        Song second = extractMin(playlists, &n);
        Song merged;
        sprintf(merged.name, "%s + %s", first.name, second.name);
        merged.duration = first.duration + second.duration;
        totalMergeDuration += merged.duration;
        insertMinHeap(playlists, &n, merged);
    }
    printf("Merged Playlist: %s\n", playlists[0].name);
    return totalMergeDuration;
}
int main() {
    Song playlists[] = {
        {"Rock Hits", 120},
        {"Pop Vibes", 80},
        {"Chill Beats", 50},
        {"Jazz Classics", 30},
        {"EDM Party", 70}
    };
    int n = sizeof(playlists)/sizeof(playlists[0]);
    int cost = mergePlaylists(playlists, n);
    printf("Total merge duration cost: %d seconds\n", cost);
    return 0;
}
```

Output:-

List Of Applications:-

1. File merging in external sorting
2. Data compression (Huffman coding)
3. Optimal binary search tree construction
4. Tape or disk scheduling
5. Document or log file merging
6. Multiway merge sort
7. Compilation and linking (symbol/object file merging)
8. Parallel and distributed data merging
9. Database system sorting and merging operations
10. Merging sorted runs in big data processing