PROJECT 1 (A)

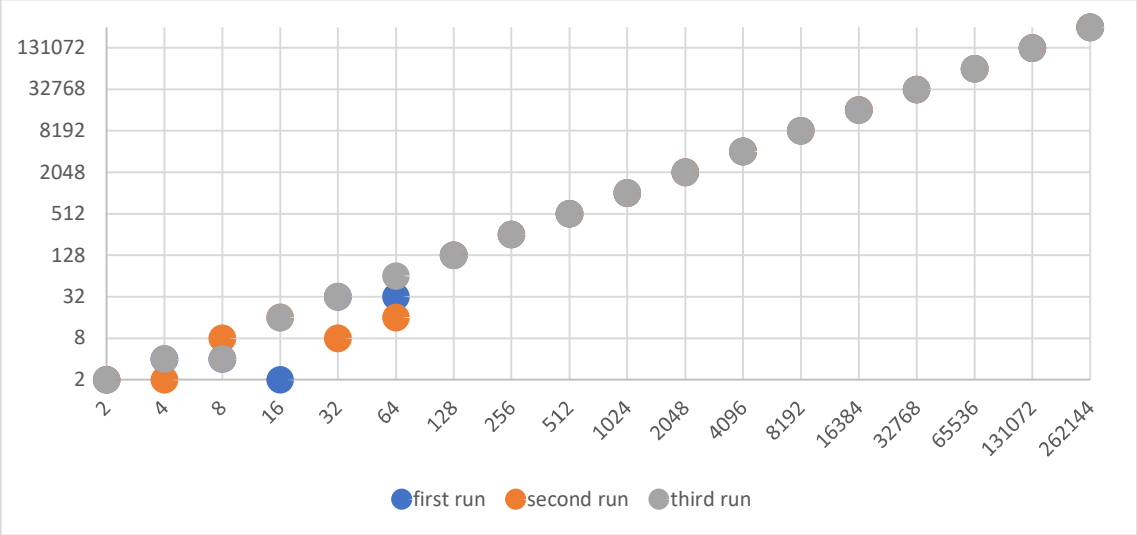
Before we begin, I would like to dwell on a general preamble to this project. We chose to work in C language as a challenge since writing the code Python would have been relatively trivial. Additionally, since C is a general-purpose compiled language, its programs tend to execute faster than interpreted programs. In the code (Appendix A), hybridSort takes the place of the original mergeSort function. On top of the original mergeSort, another else-if clause is added to check for whether the length of the subarray is greater than some threshold S. If so, the original mergeSort takes place. Else, we call insertionSort.

By iterating through various values of n and S through nested loops (all increments are in power of 2), we are able to extract the ideal S value by checking looking for the lowest runtime. To ensure a fairer outcome that is more consistent, we take 3 executions of each input case (Best, Average, Worst).

BEST CASE:

Our input is a strictly ascending array.

	IDEAL S OBTAINED		
n	first run	second run	third run
2	2	2	2
4	4	2	4
8	4	8	4
16	2	16	16
32	32	8	32
64	32	16	64
128	128	128	128
256	256	256	256
512	512	512	512
1024	1024	1024	1024
2048	2048	2048	2048
4096	4096	4096	4096
8192	8192	8192	8192
16384	16384	16384	16384
32768	32768	32768	32768
65536	65536	65536	65536
131072	131072	131072	131072
262144	262144	262144	262144



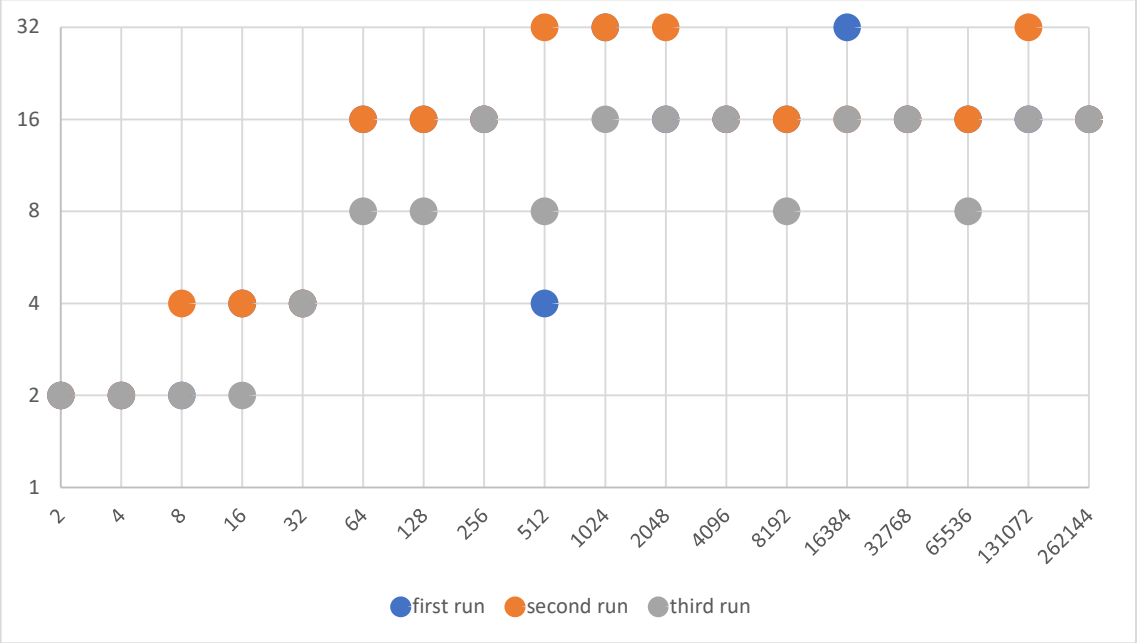
Best-Case Graph (Horizontal Axis: n-value, Vertical Axis: Optimal S-value)

As n increases, S values begin to follow n. This is expected since insertionSort is ideal for almost-sorted arrays [O(n)]. By taking the best fit of the graph, we are able to get S = n.

AVERAGE CASE:

Inputs in this array are chosen by random.

	IDEAL S OBTAINED		
n	first run	second run	third run
2	2	2	2
4	2	2	2
8	2	4	2
16	4	4	2
32	4	4	4
64	16	16	8
128	16	16	8
256	16	16	16
512	4	32	8
1024	32	32	16
2048	16	32	16
4096	16	16	16
8192	16	16	8
16384	32	16	16
32768	16	16	16
65536	16	16	8
131072	16	32	16
262144	16	16	16



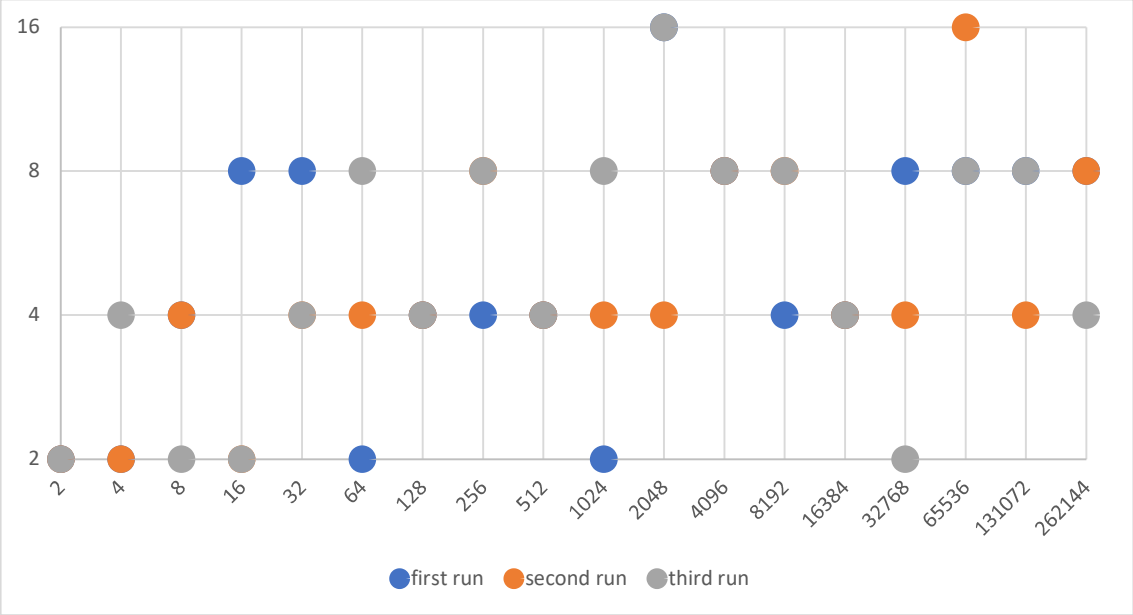
Average-Case Graph (Horizontal Axis: n-value, Vertical Axis: Optimal S-value)

Throughout all 3 runs, there is interestingly a ceiling of S = 32. We take the mode throughout the runs of S = 16.

WORST CASE:

Our input is a strictly descending array.

	IDEAL S OBTAINED		
n	first run	second run	third run
2	2	2	2
4	2	2	4
8	4	4	2
16	8	2	2
32	8	4	4
64	2	4	8
128	4	4	4
256	4	8	8
512	4	4	4
1024	2	4	8
2048	16	4	16
4096	8	8	8
8192	4	8	8
16384	4	4	4
32768	8	4	2
65536	8	16	8
131072	8	4	8
262144	8	8	4



Worst-Case Graph (Horizontal Axis: n-value, Vertical Axis: Optimal S-value)

Now there is a cap of S = 16. Nevertheless we take the mode of S = 4.

Overall, we have

BEST CASE → optimal S = n

AVERAGE CASE → optimal S = 16

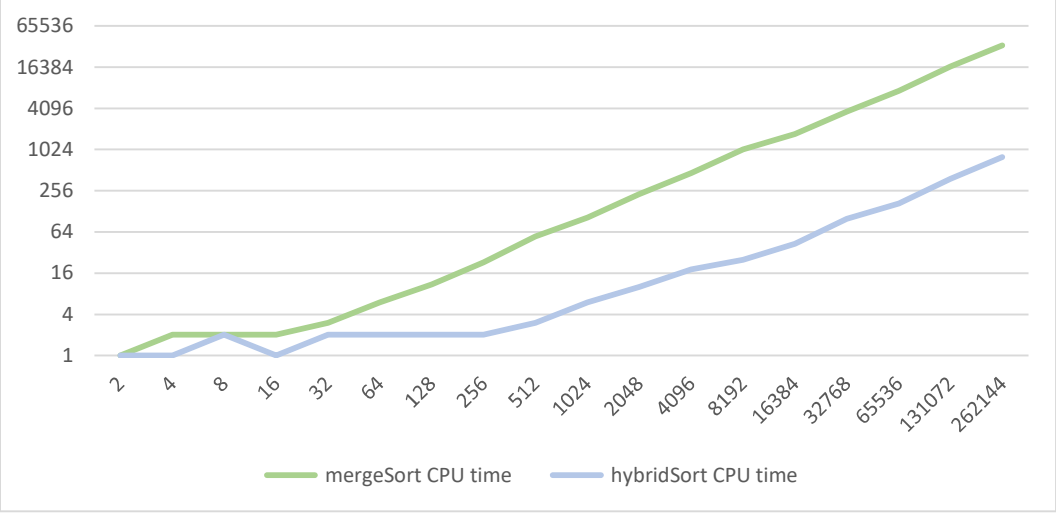
WORST CASE → optimal S = 4

PROJECT 1 (B)

Using the above values of S obtained, we compare the speeds of mergeSort and hybridSort. All CPU timings are in microseconds.

BEST CASE: S = n

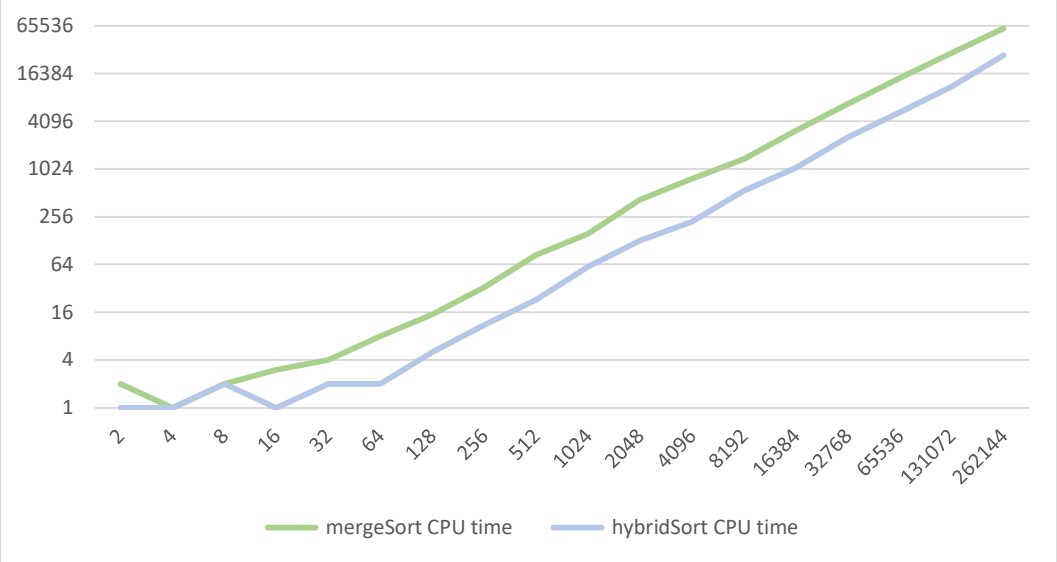
n	mergeSort CPU time	hybridSort CPU time	faster algorithm
2	1	1	hybridSort
4	2	1	hybridSort
8	2	2	hybridSort
16	2	1	hybridSort
32	3	2	hybridSort
64	6	2	hybridSort
128	11	2	hybridSort
256	23	2	hybridSort
512	55	3	hybridSort
1024	104	6	hybridSort
2048	226	10	hybridSort
4096	466	18	hybridSort
8192	1022	25	hybridSort
16384	1726	43	hybridSort
32768	3646	100	hybridSort
65536	7403	168	hybridSort
131072	16698	384	hybridSort
262144	34091	795	hybridSort



Comparison between mergeSort and hybridSort for Best-Case Array (Horizontal Axis: n-values, Vertical Axis: CPU Time)

AVERAGE CASE: S = 16

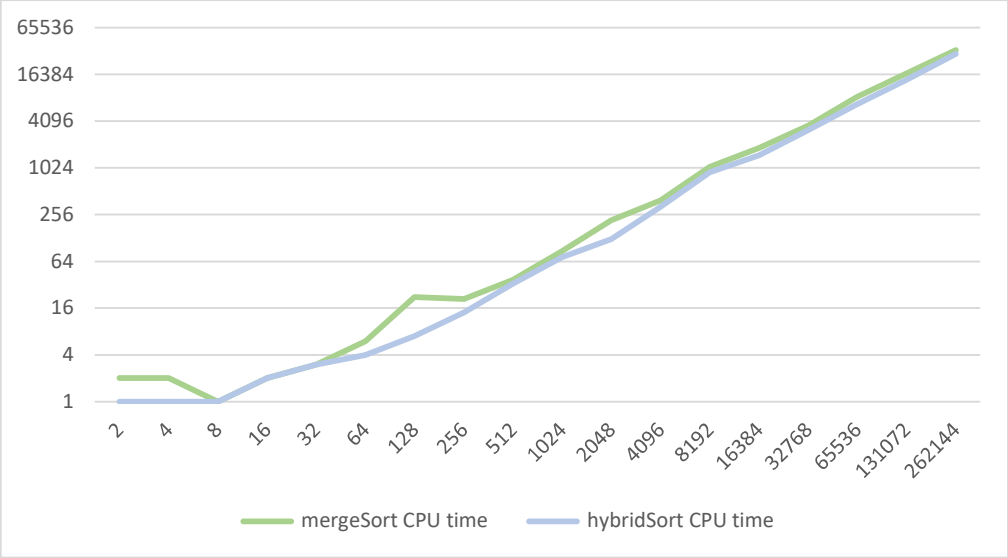
n	mergeSort CPU time	hybridSort CPU time	faster algorithm
2	2	1	hybridSort
4	1	1	hybridSort
8	2	2	hybridSort
16	3	1	hybridSort
32	4	2	hybridSort
64	8	2	hybridSort
128	15	5	hybridSort
256	33	11	hybridSort
512	84	23	hybridSort
1024	156	60	hybridSort
2048	418	128	hybridSort
4096	774	221	hybridSort
8192	1380	546	hybridSort
16384	3086	1059	hybridSort
32768	6817	2569	hybridSort
65536	14403	5335	hybridSort
131072	29793	11227	hybridSort
262144	60880	27719	hybridSort



Comparison between mergeSort and hybridSort for Average-Case Array (Horizontal Axis: n-values, Vertical Axis: CPU Time)

WORST CASE: S = 4

n	mergeSort CPU time	hybridSort CPU time	faster algorithm
2	2	1	hybridSort
4	2	1	hybridSort
8	1	1	hybridSort
16	2	2	hybridSort
32	3	3	hybridSort
64	6	4	hybridSort
128	22	7	hybridSort
256	21	14	hybridSort
512	37	33	hybridSort
1024	87	73	hybridSort
2048	216	124	hybridSort
4096	392	325	hybridSort
8192	1050	895	hybridSort
16384	1845	1492	hybridSort
32768	3585	3139	hybridSort
65536	8450	6763	hybridSort
131072	16800	13899	hybridSort
262144	33790	30186	hybridSort



Comparison between mergeSort and hybridSort for Worst-Case Array (Horizontal Axis: n-values, Vertical Axis: CPU Time)

In all 3 input cases, hybridSort provides a better alternative than mergeSort. This is expected; we are indeed removing a large portion of recursion and replacing it with iteration, which is, well, known to have a more desirable time complexity.

The difference in runtime between both methods is least obvious in the Worst Case, and most obvious in the Best Case. The likely reasoning for the Best Case could be traced back to the iteration-versus-recursion idea as previously mentioned, while a possible explanation for the Worst Case could be that our S value was not sufficiently precise (considering we are incrementing in powers of 2), due to the relatively low value of S = 4, insertionSort played little role in sorting out Worst Case arrays.

Nevertheless, in a real-world situation, it is difficult to determine how 'sorted' any array is, and by extension, which S value to choose. To sort any general case array of size n however, it could be sufficient to use $2^3 = 8$ as it is a common value within the neighbourhoods of 2^2 and 2^4.

There are many things that could be done to improve on the accuracy of this experiment, and one of them could definitely be to enlarge the sample size of our runs for each input case in (a). Perhaps then we could ensure that we have a more representative solution for S to be used in (b). Regardless, I believe what has been presented here is an acceptable amount of analysis of the efficiency of this hybrid algorithm.

APPENDIX A

Code for (a)

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
void merge(int n, int m, int arr[]); void mergeSort(int lst[], int n, int m); void
insertionSort(); void hybridSort(); void randomArray(); void bestArray(); void
worstArray();
#define INT_MAX 2147483647

int main(void) {

  FILE *f = fopen("results.csv", "w");
  fprintf(f, "n, best S, CPU time\n");

  srand(time(NULL));
  clock_t start, end; double cpu_time;
  for (int n = 2; n <= pow(2, 18); n *= 2){
    double cpu_threshold = INT_MAX; int best_s = 1;

    for (int s = 2; s <= n; s *= 2){
      int *arr = malloc(n * sizeof(int));
      worstArray(n, arr); // to be toggled yourself
      start = clock();
      hybridSort(0, n-1, arr, s);
      end = clock();
      cpu_time = (double) ((end - start) * 1000000 / CLOCKS_PER_SEC); // multiplied
for microseconds
      if (cpu_time < cpu_threshold){
        cpu_threshold = cpu_time;
        best_s = s;
      }
      free(arr);
    }
    printf("n = %d, best S = %d, CPU time = %f \n", n, best_s, cpu_threshold);
    fprintf(f, "%d, %d, %f\n", n, best_s, cpu_threshold);

  }
  fclose(f);
}

////////////// ARRAY MAKING ///////////////

void randomArray(int num, int *arr){
  srand(time(NULL));
  for (int i = 0; i < num; i++){
    arr[i] = rand() % 20000;
  }
}

void bestArray(int num, int *arr){
  for (int i = 0; i < num; i++){
    arr[i] = i;
  }
}

void worstArray(int num, int *arr){
  for (int i = 0; i < num; i++){
    arr[i] = num - i;
  }
}

////////////// METHODS ///////////////
```

```
void hybridSort(int n, int m, int *arr, int S){
  if (m - n <= 0) return;
  else if (m - n >= 1 && m - n + 1 <= S){
    insertionSort(arr, n, m);
  }
  else if (m - n >= 1 && m - n + 1 > S){
    int mid = (n + m)/2;
    hybridSort(n, mid, arr, S);
    hybridSort(mid + 1, m, arr, S);
    merge(n, m, arr);
  }
}

void merge(int n, int m, int *arr){
  int mid = (n + m)/2;
  int len1 = mid - n + 1;
  int len2 = m - mid;
  int L[len1], R[len2];
  for (int i = 0; i < len1; i++){
    L[i] = arr[n + i];
  }
  for (int j = 0; j < len2; j++){
    R[j] = arr[mid + j + 1];
  }

  int ind = n, l = 0, r = 0;
  // printf("mergeSort \n");

  while (l < len1 && r < len2){
    if (L[l] < R[r]){
      arr[ind] = L[l];
      l++;
    }
    else {
      arr[ind] = R[r];
      r++;
    }
    ind++;
  }

  while (l < len1){
    arr[ind] = L[l];
    l++;
    ind++;
  }

  while (r < len2){
    arr[ind] = R[r];
    r++;
    ind++;
  }
}

void insertionSort(int *lst, int n, int m){
  if ((m - n) < 1) return;
  int dummy;
  for (int i = n + 1; i < m + 1; i++){
    for (int j = i; j > n; j--){
      if (lst[j] < lst[j-1]){
        dummy = lst[j];
        lst[j] = lst[j-1];
        lst[j-1] = dummy;
      }
      else break;
    }
  }
}
```

APPENDIX B

Code for (b)

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>
void merge(); void mergeSort(); void insertionSort(); void hybridSort(); void
randomArray(); void bestArray(); void worstArray();
#define INT_MAX 2147483647

int main(void) {

  FILE *f = fopen("results.csv", "w");
  fprintf(f, "n, mergeSort CPU time, hybridSort CPU time, faster algorithm\n");

  srand(time(NULL));
  clock_t start_m, end_m, start_h, end_h; double cpu_time_m, cpu_time_h; int
choice;

  choice = 3; // toggle this yourself, 1 -> best, 2 -> ave, 3 -> worst

  for (int n = 2; n <= pow(2, 18); n *= 2){
    int *arr = malloc(n * sizeof(int));
    switch(choice){
      case 1:
      bestArray(n, arr);
      start_m = clock();
      mergeSort(0, n-1, arr);
      end_m = clock();
      start_h = clock();
      hybridSort(0, n-1, arr, n);
      end_h = clock();
      break;
      case 2:
      randomArray(n, arr);
      start_m = clock();
      mergeSort(0, n-1, arr);
      end_m = clock();
      start_h = clock();
      hybridSort(0, n-1, arr, 16);
      end_h = clock();
      break;
      case 3:
      worstArray(n, arr);
      start_m = clock();
      mergeSort(0, n-1, arr);
      end_m = clock();
      start_h = clock();
      hybridSort(0, n-1, arr, 4);
      end_h = clock();
      break;
      default:
      printf("Error\n");
      break;
    }
    cpu_time_m = (double) ((end_m - start_m) * 1000000 / CLOCKS_PER_SEC);
    cpu_time_h = (double) ((end_h - start_h) * 1000000 / CLOCKS_PER_SEC);

    if (cpu_time_m < cpu_time_h){
      printf("n = %d, mergeSort CPU time = %f, hybridSort CPU time = %f, faster =
mergeSort\n", n, cpu_time_m, cpu_time_h);
      fprintf(f, "%d, %f, %f, %s\n", n, cpu_time_m, cpu_time_h, "mergeSort");
    }
    else{
      printf("n = %d, mergeSort CPU time = %f, hybridSort CPU time = %f, faster =
hybridSort\n", n, cpu_time_m, cpu_time_h);
```

```
      fprintf(f, "%d, %f, %f, %s\n", n, cpu_time_m, cpu_time_h, "hybridSort");
    }


  }
  fclose(f);

}

////////////// ARRAY MAKING ///////////////

void randomArray(int num, int *arr){
  srand(time(NULL));
  for (int i = 0; i < num; i++){
    arr[i] = rand() % 20000;
  }
}

void bestArray(int num, int *arr){
  for (int i = 0; i < num; i++){
    arr[i] = i;
  }
}

void worstArray(int num, int *arr){
  for (int i = 0; i < num; i++){
    arr[i] = num - i;
  }
}

////////////// METHODS ///////////////

void hybridSort(int n, int m, int *arr, int S){

  if (m - n <= 0) return;
  else if (m - n >= 1 && m - n + 1 <= S){
    // printf("insertionSort \n");

    insertionSort(arr, n, m);

  }
  else if (m - n >= 1 && m - n + 1 > S){
    int mid = (n + m)/2;
    hybridSort(n, mid, arr, S);
    hybridSort(mid + 1, m, arr, S);
    merge(n, m, arr);
  }
}

void mergeSort(int n, int m, int *arr){
  int mid = (n + m)/2;
  if (m - n <= 0) return;
  else if (m - n > 1){
    mergeSort(n, mid, arr);
    mergeSort(mid + 1, m, arr);
  }
  merge(n, m, arr);
}

void merge(int n, int m, int *arr){
  int mid = (n + m)/2;
  int len1 = mid - n + 1;
  int len2 = m - mid;

  int L[len1], R[len2];

  for (int i = 0; i < len1; i++){
    L[i] = arr[n + i];
```

```
  }
  for (int j = 0; j < len2; j++){
    R[j] = arr[mid + j + 1];
  }

  int ind = n, l = 0, r = 0;
  // printf("mergeSort \n");

  while (l < len1 && r < len2){
    if (L[l] < R[r]){
      arr[ind] = L[l];
      l++;
    }
    else {
      arr[ind] = R[r];
      r++;
    }
    ind++;
  }

  while (l < len1){
    arr[ind] = L[l];
    l++;
    ind++;
  }

  while (r < len2){
    arr[ind] = R[r];
    r++;
    ind++;
  }

}

void insertionSort(int *lst, int n, int m){

  if ((m - n) < 1) return;

  int dummy;
  for (int i = n + 1; i < m + 1; i++){
    for (int j = i; j > n; j--){
      if (lst[j] < lst[j-1]){
        dummy = lst[j];
        lst[j] = lst[j-1];
        lst[j-1] = dummy;
      }
      else break;
    }
  }
}
// this is a cry for help. as much as i want to keep my values and ideas as accurate as possible this is a
one-man-show and i just slept for 1.5 hours
```