Analysis and Design of Algorithms

Carlos Esteban Guerrero Robles

06 April 2019

1 Warm up

Lets modify the classic merge sort algorithm a little bit. What happens if instead of splitting the array in 2 parts we divide it in 3? You can assume that exists a three-way merge subroutine. What is the overall asymptotic running time of this algorithm?

Answer:

If we divide the array in 3 parts it will result in the reduction of the height of the division tree. But the steps of the three-way merge subroutine will be the same as the two-way merge subroutine, because they need necessarily to pass through all the array; however, the number of comparison in the sub-routine will me more (at most three).

Based on that the running time will be:
$c * sizeArray(heightDivisionTree) + c * sizearray$
$c * n(log_3 n) + c * n$

So the overall asymptotic running time will be:
$\Theta(nlog_3 n)$

Even if looks like the three way method is more efficient than the normal merge sort, the actual time will be higher because the number of comparison in the sub-routine is more as mentioned before.

BONUS: Implement the three-way merge sort algorithm.

Answer:

Three-way Merge Sub-routine:

```
#include <limits>
#include <vector>

void merge(std::vector<int> &A, int p, int q, int r, int s) {
  int n1 = q-p+1;
  int n2 = r-q;
  int n3 = s-r;
  int R[n1+1], M[n2+1], L[n3+1];
  for (size_t i = 0; i < n1; i++) {
    R[i] = A[p+i];
  }
  for (size_t i = 0; i < n2; i++) {
    M[i] = A[q+1+i];
  }
  for (size_t i = 0; i < n3; i++) {
    L[i] = A[r+1+i];
  }
  R[n1] = std::numeric_limits<int>::max();
  M[n2] = std::numeric_limits<int>::max();
  L[n3] = std::numeric_limits<int>::max();
  int i = 0, j = 0, k = 0;
  for (size_t n = p; n <= s; n++) {
    if (R[i] <= M[j]) {
      if (R[i] <= L[k]) {
        A[n] = R[i];
        i++;
      }else{
        A[n] = L[k];
        k++;
      }
    }else if (M[j] <= L[k]) {
      A[n] = M[j];
      j++;
    }else{
      A[n] = L[k];
      k++;
    }
  }
}
```

Three-way Merge Sort:

```
#include "tWayM.h"

void threeWayMergeSort(std::vector<int> &A, int p, int s){
```

```
  if (p < s) {
    int q, r;
    q = p+(s-p+1)/3-1;
    r = q+(s-p)/3+1;
    threeWayMergeSort(A,p,q);
    threeWayMergeSort(A,q+1,r);
    threeWayMergeSort(A,r+1,s);
    merge(A,p,q,r,s);
  }
}
```

Main test:

```
#include <iostream>
#include "tWayMS.h"

int main(int argc, char const *argv[]) {
  std::vector<int> A = {7,6,5,4,3,2,1,0};
  threeWayMergeSort(A, 0, A.size()-1);
  for (size_t i = 0; i < A.size(); i++) {
    std::cout << A[i] << ' ';
  }
  std::cout << '\n';
  return 0;
}
```

2 Competitive programming

Welcome to your first competitive programming problem!!!

- Sign-up in Uva Online Judge (`https://uva.onlinejudge.org`) and in CodeChef if you want (we will use it later).
- Rest easy! This is not a contest, it is just an introductory problem. Your first problem is located in the "Problems Section" and is **100 - The 3n + 1 problem**.

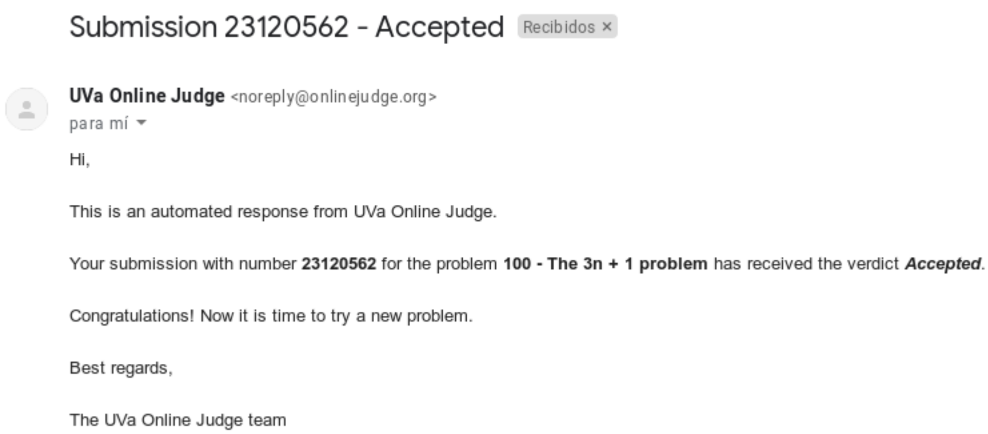
Answer:

```
#include <iostream>

void orderPair(int &iPair, int &jPair){
```

```
  if (iPair > jPair) {
    int temp;
    temp = iPair;
    iPair = jPair;
    jPair = temp;
  }
}

int main(int argc, char const *argv[]) {
  int maxCycle = 0, cycle = 0;
  int iPar = 0, jPar = 0, cValue = 0;
  while (std::cin >> iPar >> jPar) {
    std::cout << iPar << ' ' << jPar << ' ' ;
    orderPair(iPar, jPar);
    maxCycle = 0;
    for (int i = iPar; i <= jPar; i++) {
      cValue = i;
      cycle = 1;
      while (cValue != 1) {
        if (cValue%2 == 1) {
          cValue = 3*cValue+1;
        }else{
          cValue = cValue/2;
        }
        cycle++;
      }
      if (cycle > maxCycle) {
        maxCycle = cycle;
      }
    }
    std::cout << maxCycle << '\n';
  }
  return 0;
}
```



Submission 23120562 - Accepted Recibidos ×

UVa Online Judge <noreply@onlinejudge.org>
para mí

Hi,

This is an automated response from UVa Online Judge.

Your submission with number **23120562** for the problem **100 - The 3n + 1 problem** has received the verdict ***Accepted***.

Congratulations! Now it is time to try a new problem.

Best regards,

The UVa Online Judge team

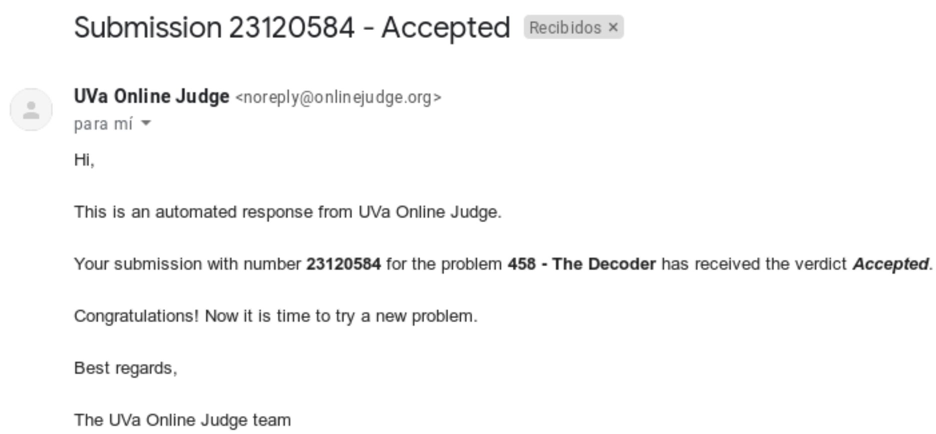
Figure 1: Problem 100 accepted by Uva

- Once that you finish with that problem continue with **458 - The Decoder**. Again, this problem is just to build your confidence in competitive programming.

  Answer:

```
#include <iostream>

int main(int argc, char const *argv[]) {
  std::string line;
  while (getline(std::cin, line)) {
    for (size_t i = 0; i < line.size(); i++) {
      line[i] = line[i]-7;
    }
    std::cout << line << '\n';
  }
}
```


Submission 23120584 - Accepted Recibidos ×

UVa Online Judge <noreply@onlinejudge.org>
para mí

Hi,

This is an automated response from UVa Online Judge.

Your submission with number **23120584** for the problem **458 - The Decoder** has received the verdict ***Accepted***.

Congratulations! Now it is time to try a new problem.

Best regards,

The UVa Online Judge team

Figure 2: Problem 458 accepted by Uva

- *BONUS:* **10855 - Rotated squares**

  Answer:

```
#include <iostream>
#include <vector>
#include <string>

typedef std::vector<std::string> square;

bool compareSquares(square bSquare, square sSquare, int dH, int dV) {
  for (size_t i = 0; i < sSquare.size(); i++) {
    for (size_t j = 0; j < sSquare.size(); j++) {
      if (bSquare[i+dV][j+dH] != sSquare[i][j]) {
        return false;
      }
    }
  }
  return true;
}

square turnSquare(square sSquare) {
  square tSquare;
  for (size_t i = 0; i < sSquare.size(); i++) {
    std::string line = "";
    for (size_t j = sSquare.size(); j > 0; j--) {
      line += sSquare[j-1][i];
```

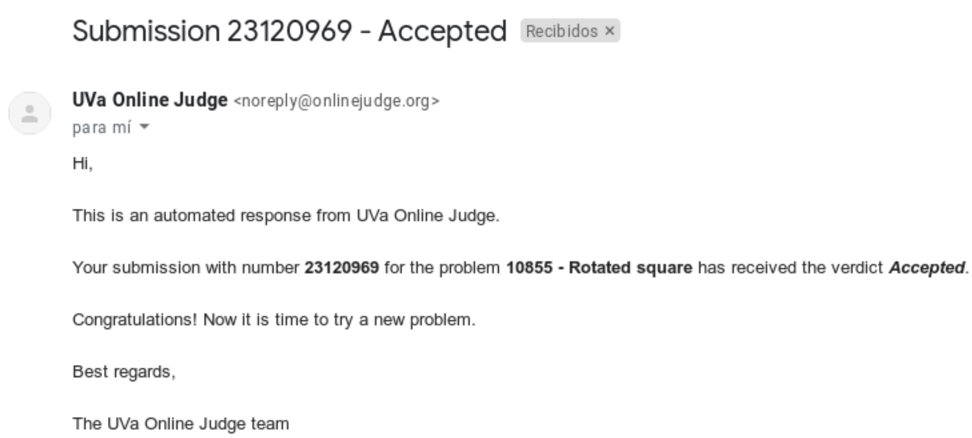
```
    }
    tSquare.push_back(line);
  }
  return tSquare;
}

void fitInSquare(square bSquare, square sSquare, int dH, int dV, std::vector<int> &nCondidences) {
  for (size_t i = 0; i < 4; i++) {
    if (compareSquares(bSquare, sSquare, dH, dV)) {
      nCondidences[i]++;
    }
    sSquare = turnSquare(sSquare);
  }
}

void scanSquares(square bSquare, square sSquare, std::vector<int> &nCondidences){
  int desplacements = bSquare.size()-sSquare.size()+1;
  for (size_t dV = 0; dV < desplacements; dV++) {
    for (size_t dH = 0; dH < desplacements; dH++) {
      fitInSquare(bSquare, sSquare, dH, dV, nCondidences);
    }
  }
}

int main(int argc, char const *argv[]) {
  int n = 0, N = 0, tests = 0;
  std::cin >> N >> n;
  std::string line;
  std::vector<std::vector<int>> output;
  while (N != 0 && n != 0) {
    tests++;
    square bSquare, sSquare;
    std::vector<int> nCondidences = {0,0,0,0};
    for (size_t i = 0; i < N; i++) {
      std::cin >> line;
      bSquare.push_back(line);
    }
    for (size_t i = 0; i < n; i++) {
      std::cin >> line;
      sSquare.push_back(line);
    }
    scanSquares(bSquare, sSquare, nCondidences);
```

```cpp
    output.push_back(nCondidences);
    std::cin >> N >> n;
  }
  for (size_t i = 0; i < tests; i++) {
    for (size_t j = 0; j < 4; j++) {
      std::cout << output[i][j];
      if (j != 3) {
        std::cout << ' ';
      }
    }
    std::cout << '\n';
  }
}
```


Submission 23120969 - Accepted Recibidos ×

UVa Online Judge <noreply@onlinejudge.org>
para mí

Hi,

This is an automated response from UVa Online Judge.

Your submission with number **23120969** for the problem **10855 - Rotated square** has received the verdict ***Accepted***.

Congratulations! Now it is time to try a new problem.

Best regards,

The UVa Online Judge team

Figure 3: Problem 10855 accepted by Uva

3 Simulation

Write a program to find the minimum input size for which the merge sort algorithm always beats the insertion sort.

- Implement the insertion sort algorithm

  Answer:

```cpp
#include <vector>

```

```
void insertionSort(std::vector<int> &array) {
  int temp;
  int j;
  for (size_t i = 1; i < array.size() ; i++) {
    temp = array[i];
    j = i-1;
    while (j+1 > 0 && array[j] > temp) {
      array[j+1] = array[j];
      j--;
    }
    array[j+1] = temp;
  }
};
```

- Implement the merge sort algorithm

  Answer:

```
#include <limits>
#include <vector>

void merge(std::vector<int> &A, int p, int q, int r){
  int n1 = q-p+1;
  int n2 = r-q;
  int R[n1+1], L[n2+1];
  for (size_t i = 0; i < n1; i++) {
    R[i] = A[p+i];
  }
  for (size_t i = 0; i < n2; i++) {
    L[i] = A[q+1+i];
  }
  R[n1] = std::numeric_limits<int>::max();
  L[n2] = std::numeric_limits<int>::max();
  int i = 0, j = 0;
  for (size_t n = p; n <= r; n++) {
    if (R[i] <= L[j]) {
      A[n] = R[i];
      i++;
    }else{
      A[n] = L[j];
      j++;
    }
  }
}

```

```
void mergeSort(std::vector<int> &A, int p, int r){
  if (p < r) {
    int q;
    q = (p+r)/2;
    mergeSort(A,p,q);
    mergeSort(A,q+1,r);
    merge(A,p,q,r);
  }
}
```

- Just compare them? No !!! Run some simulations or tests and find the average input size for which the merge sort is an asymptotically "better" sorting algorithm.

  Answer:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include "iSort.h"
#include "mSort.h"

void randomIG (std::vector<int> &A, std::vector<int> &B,
    int size) {
  srand (time(NULL));
  int n;
  for (size_t i = 0; i < size; i++) {
    n = rand() % size + 1;
    A.push_back(n);
    B.push_back(n);
  }
}

void testPS(double &pmIS, double &pmMS, int testSize, int
      test) {
  double cValue = 1000000; // microseconds
  pmIS = 0;
  pmIS = 0;
  for (size_t i = 0; i < testSize; i++) {
    std::vector<int> A, B;
    randomIG(A, B, test);
    auto start = std::chrono::high_resolution_clock::now
        ();
```

```
    insertionSort(A);
    auto finish = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeIS = finish - start;
    start = std::chrono::high_resolution_clock::now();
    mergeSort(B, 0, B.size()-1);
    finish = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeMS = finish - start;
    pmIS += (timeIS.count()*cValue);
    pmMS += (timeMS.count()*cValue);
  }
  pmIS /= double(testSize);
  pmMS /= double(testSize);
}

int main(int argc, char const *argv[]) {
  int test = 1, bTest = 0;
  double pmIS, pmMS;
  int testSize = 100;
  std::fstream file;
  file.open("tests_1.txt", std::ios::out);
  do {
    test++;
    testPS(pmIS, pmMS, testSize, test);
    file << test << " " << pmIS << " " <<pmMS << '\n';
  } while(pmIS < pmMS);
  bTest = test;
  do {
    for (size_t i = test+1; i < test+test/2; i++) {
      testPS(pmIS, pmMS, testSize, i);
      file << i << " " << pmIS << " " <<pmMS << '\n';
      if (pmIS < pmMS) {
        bTest = 0;
        test = i;
        break;
      }else if (bTest == 0) {
        bTest = i;
      }
    }
  } while(bTest == 0);
  file.close();
  std::cout << "Average input size for which the merge sort is an asymptotically better sorting algorithm
```

```
        is: " << bTest << '\n';
    return 0;
}
```

Comments:
In order to have a well designed experiment I implement a random vector generator ramdomIG() (the vector is the same for both algorithms), a function to obtain a mean execution time of both algorithms testPS() (the size of the sample was 100). Also in the main function, there is a two loops; the first one is to find the firs input size for which the merge sort is asymptotically better, and in order to be sure that for higher input size merge sort continues being better than insertion exist the second loop. That loop demand that for an input size n where merge sort is better than insertion sort, all the $n/2$ higher values of input size need to maintain the condition, if it does not accomplish continues searching a correct n value.
Also to have the chance of plot the experiment, all the input sizes and mean times of both algorithms are saved in a "test_1.txt" file. There is its content:

```
test    mInsert mMerge
2       0.29912 0.52946
3       0.28821 0.810435
4       0.43106 1.13158
5       0.62363 1.63252
6       0.97234 2.18851
7       1.10116 2.57968
8       1.09117 2.97235
9       1.20502 2.67161
10      0.98755 3.76077
11      1.52512 3.72463
12      1.86994 4.51613
13      1.61423 3.16789
14      1.75589 3.35821
15      1.28199 2.82997
16      1.29542 2.57779
17      1.60352 2.78484
18      1.75757 3.02688
19      1.94531 2.55167
20      1.36198 2.64129
21      1.95244 2.79974
22      1.48315 2.87772
23      2.24076 2.64798
```

24	24	2.26248	2.70462
25	25	2.69371	3.12635
26	26	2.02033	2.93626
27	27	1.93963	2.59128
28	28	2.15301	2.71888
29	29	2.35708	2.80084
30	30	2.23122	3.06361
31	31	2.76962	2.89246
32	32	2.41553	2.75936
33	33	2.87736	2.92254
34	34	3.00783	3.00098
35	35	3.34942	3.09189
36	36	3.41011	3.04258
37	37	2.7227	3.35784
38	38	2.95964	3.26476
39	39	3.79422	3.33778
40	40	3.5966	4.1331
41	41	3.66891	3.75605
42	42	3.91243	3.69796
43	43	4.5314	3.70874
44	44	4.03838	3.79609
45	45	4.12697	4.01143
46	46	4.47982	4.03359
47	47	4.87174	4.21944
48	48	4.68329	4.6458
49	49	5.09107	4.77527
50	50	5.74342	5.31022
51	51	5.39877	4.70753
52	52	4.91341	4.65284
53	53	5.77932	4.70943
54	54	6.8531	5.18637
55	55	6.48779	5.11443
56	56	6.68693	4.98157
57	57	7.41279	5.20209
58	58	6.27407	5.3983
59	59	7.06712	5.20902
60	60	7.69631	5.76043

Note: Include (.tex) and attach(.cpp) your source code and use a dockerfile to interact with python and plot your results.

BONUS: Compare both algorithms against any other sorting algorithm

Answer:

I chose the selection sort as other algorithm to be compare against the inser-

tion and merge sort.
Here is it's implementation:

```
#include <vector>

void swapElements(std::vector<int> &array, size_t posA,
    size_t posB) {
  int temp = 0;
  temp = array[posA];
  array[posA] = array[posB];
  array[posB] = temp;
};

void selectionSort(std::vector<int> &array) {
  int size = array.size();
  int minIndex;
  for (size_t i = 0; i < size-1; i++) {
    minIndex = i;
    for (size_t j = i; j < size; j++) {
      if (array[minIndex] > array[j]) {
        minIndex = j;
      }
    }
    swapElements(array, i, minIndex);
  }
}
```

Also was necessary to modify our experiment to work with the selection sort, here is the new implementation:

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <chrono>
#include "iSort.h"
#include "mSort.h"
#include "sSort.h"

void randomIG (std::vector<int> &A, std::vector<int> &B,
    std::vector<int> &C, int size) {
  srand (time(NULL));
  int n;
  for (size_t i = 0; i < size; i++) {
```

```
    n = rand() % size + 1;
    A.push_back(n);
    B.push_back(n);
    C.push_back(n);
  }
}

void testPS(double &pmIS, double &pmMS, double &pmSS, int
    testSize, int test) {
  double cValue = 1000000; // microseconds
  pmIS = 0;
  pmIS = 0;
  pmSS = 0;
  for (size_t i = 0; i < testSize; i++) {
    std::vector<int> A, B, C;
    randomIG(A, B, C, test);
    auto start = std::chrono::high_resolution_clock::now();
    insertionSort(A);
    auto finish = std::chrono::high_resolution_clock::now()
        ;
    std::chrono::duration<double> timeIS = finish - start;
    start = std::chrono::high_resolution_clock::now();
    mergeSort(B, 0, B.size()-1);
    finish = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeMS = finish - start;
    start = std::chrono::high_resolution_clock::now();
    selectionSort(C);
    finish = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> timeSS = finish - start;
    pmIS += (timeIS.count()*cValue);
    pmMS += (timeMS.count()*cValue);
    pmSS += (timeSS.count()*cValue);
  }
  pmIS /= double(testSize);
  pmMS /= double(testSize);
  pmSS /= double(testSize);
}

int main(int argc, char const *argv[]) {
  int test = 1, bTest = 0;
  double pmIS, pmMS, pmSS;
  int testSize = 100;
  std::fstream file;
  file.open("tests_2.txt", std::ios::out);
  do {
```

```
    test++;
    testPS(pmIS, pmMS, pmSS, testSize, test);
    file << test << " " << pmIS << " " <<pmMS << " " <<pmSS
        << '\n';
  } while(pmIS < pmMS || pmSS < pmMS);
  bTest = test;
  do {
    for (size_t i = test+1; i < test+test/2; i++) {
      testPS(pmIS, pmMS, pmSS, testSize, i);
      file << i << " " << pmIS << " " <<pmMS << " " <<pmSS
          << '\n';
      if (pmIS < pmMS || pmSS < pmMS) {
        bTest = 0;
        test = i;
        break;
      }else if (bTest == 0) {
        bTest = i;
      }
    }
  } while(bTest == 0);
  file.close();
  std::cout << "Average input size for which the merge sort
      is an asymptotically better sorting algorithm is: "
     << bTest << '\n';
  return 0;
}
```

Also the experiment demand that for an input size n where merge sort is better than insertion sort and selection sort, all the $n/2$ higher values of input size need to maintain the condition, if it does not accomplish continues searching a correct n value.

In a similar way all the data is saved in a file "tests_2.txt", here is its content:

```
test    mInsert mMerge  mSelect
2       0.28864 0.59875 0.40735
3       0.44428 0.983657        0.57721
4       0.53401 1.34328 0.73769
5       0.55964 1.92366 1.07169
6       0.66934 1.95819 1.1022
7       0.77118 2.28149 1.30734
8       0.86073 2.62316 1.56693
9       0.67268 2.14806 1.32959
10      1.22219 2.22212 1.39275
```

```
11       0.97173 2.04224 1.32683
12       1.10309 2.2445  1.54588
13       1.43514 2.79387 2.17615
14       1.49208 2.81191 2.09632
15       1.11288 2.34958 1.9649
16       0.94923 2.00526 1.60905
17       0.72052 1.82439 1.50107
18       1.14233 1.98769 1.6429
19       1.28919 2.10369 1.78716
20       1.06536 1.89724 1.85709
21       1.54839 2.15088 2.01874
22       1.52294 2.09781 2.04738
23       1.34492 2.86141 2.21219
24       1.45354 2.1239  2.17979
25       1.52659 2.20433 2.28197
26       1.67313 2.21623 2.41159
27       2.06157 2.37361 2.76331
28       1.74002 2.37142 2.96647
29       1.66257 2.42302 2.83489
30       2.18841 2.48184 3.04142
31       2.12096 2.70044 3.31905
32       2.42822 2.6978  3.52103
33       2.31161 2.70412 3.61412
34       2.12897 2.78981 3.77525
35       2.59469 2.96456 3.96166
36       2.84762 3.02683 4.239
37       3.15502 3.12043 4.5442
38       3.75368 3.68754 5.38054
39       3.4361  3.40396 5.31784
40       3.50281 3.49912 5.25371
41       3.24211 3.69861 5.96462
42       3.61037 3.66313 5.92637
43       4.3356  4.00051 6.43599
44       3.71553 3.62356 6.01534
45       4.09364 3.69508 6.06684
46       4.22994 3.82198 6.41221
47       4.45623 3.8874  6.86013
48       5.03368 4.32931 6.893
49       5.29145 4.57252 7.86239
50       5.40128 4.64894 7.97582
51       5.92832 4.94203 8.47919
52       5.41306 4.84887 8.44713
53       5.81993 4.76635 8.93447
54       5.67853 5.10462 9.21068
55       6.30353 5.63068 10.3473
```

```
56      6.48456 5.52111 9.97676
57      6.36249 5.2169  10.1028
58      6.98559 5.4439  10.6383
59      7.16111 5.471   11.1135
60      7.64822 5.64568 11.6087
61      8.75452 5.94485 12.6142
62      8.74199 5.82245 12.0604
```

4 Research

Everybody at this point remembers the quadratic "grade school" algorithm to multiply 2 numbers of k_1 and k_2 digits respectively.

Your assignment now is to compare the number of operations performed by the quadratic grade school algorithm and Karatsuba multiplication.

- Define Karatsuba multiplication
- Implement grade school multiplication
- Implement Karatsuba multiplication
- Compare Karatsuba algorithm against grade school multiplication
- Use any of your implemented algorithms to multiply $a * b$ where:

  a: 3141592653589793238462643383279502884197169399375105820974944592

  b: 2718281828459045235360287471352662497757247093699959574966967627

Note: Include(.tex) and attach(.cpp) your source code, of course.

BONUS: How about Schönhage-Strassen algorithm ?

5 Wrapping up

Arrange the following functions in increasing order of growth rate with $g(n)$ following $f(n)$ if $f(n) = \mathcal{O}(g(n))$

1. $n^2 log(n)$

2. 2^n

3. 2^{2^n}

4. $n^{log(n)}$

5. n^2

Answer:
By plotting all the functions we can order it:

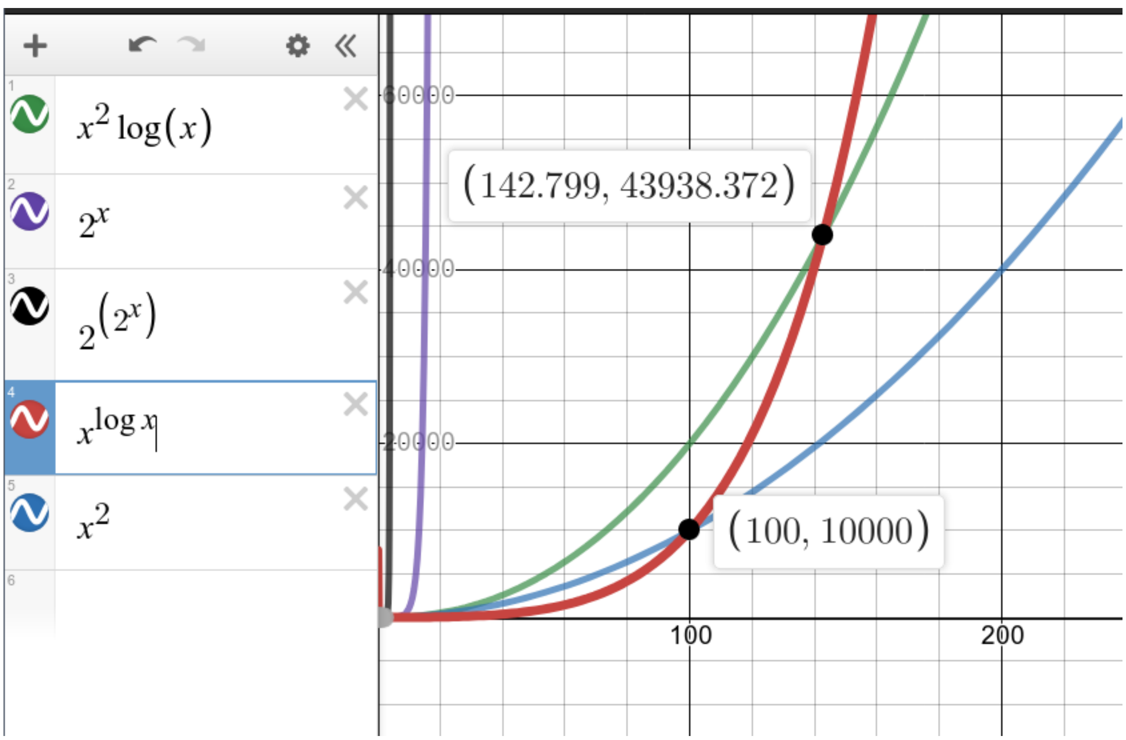


Figure 4: Plot of all functions

According to the graph the order will be: (Increasing order of grow rate)

1. n^2 (Lesser order of grow rate)

2. $n^2 log(n)$

3. $n^{log(n)}$

4. 2^n

5. 2^{2^n} (Higher order of grow rate)

Commentaries:

As we can see in the plot the function 2^{2^n} grow fastest passing 60000 with only $x \approx 8$ that mean it is $\mathcal{O}()$ of the rest of functions. After 2^{2^n} comes 2^n with high grow rate passing 60000 with only $x \approx 18$. The rest of functions require more analyses because on the beginning of the domain looks like $n^2 log(n)$ have more grow rate than the rest; however, as the value of x grow the situation changes. Since we use an x big enough to obtain the order of grow, we need to zoom out the plot. With a x big enough we can see the function $n^{log(n)}$ pass n^2 at $x = 100$ and pass $n^2 log(n)$ at $x = 142.799$. That explain the order aforementioned.