Analysis and Design of Algorithms

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?

In this version of merge sort, the recursive step of the algorithm would be called 3 times and then the sub-arrays would be merged to form the sorted original array. With this in mind, the runtime of merge sort can be calculated as follows:

$$T(n) = 3T(n/3) + \Theta(n)$$

where $3T(n/3)$ represents the call to merge sort as the recursive step and $\Theta(n)$ the merging of the three sorted sub-arrays. By expanding the recursive step in the definition of the runtime function, we obtain the following result:

$$\begin{aligned} T(n) &= 3T(n/3) + \Theta(n) \\ &= 3(3T(n/9) + n/3) + \Theta(n) \\ &= 9T(n/9) + 2\Theta(n) \\ &= 27T(n/27) + 3\Theta(n) \end{aligned}$$

which shows how each level cost is equivalent to $\Theta(n)$. As the recursive division of three sub-arrays from the original array leads to a maximum level of $\log_3 n$, we can conclude that the asymptotic running time of three-way merge sort is $\Theta(n \log_3 n)$.

BONUS: Implement the three-way merge sort algorithm.

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**.

```
#include <iostream>

int cycleLength(int n) {
   if (n == 1) {
      return 1;
   }

   return 1 + cycleLength(n % 2 ? 3 * n + 1 : n / 2);
}

using namespace std;

int main (void) {
   int i, j;

   while (cin >> i) {
      cin >> j;

      int iMin, iMax;

      if (i < j) {
         iMin = i;
         iMax = j;
      } else {
         iMin = j;
         iMax = i;
      }
```

```
      int max = 0;

      for (int k = iMin; k <= iMax; ++k) {
         int current = cycleLength(k);

         if (current > max) {
            max = current;
         }
      }

      cout << i << ' '<< j << ' ' << max << endl;
   }

   return 0;
}
    }
}
```

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

```
#include <iostream>

using namespace std;

int main(void) {
   char c = cin.get();

   while (!cin.eof()) {
      if (c != '\n') {
         cout << (char) (c - 7);
      } else {
         cout << endl;
      }

      c = cin.get();
   }
```

```
    return 0;
}
```

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

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

```
#include <vector>

#include "insertionSort.h"

void insertionSort(std::vector<int>& v) {
   // Current index of element to be sorted
   int curIndex;

   for (curIndex = 1; curIndex < v.size(); ++curIndex) {
      // Target index in which v[curIndex] will be inserted
      int tarIndex = curIndex - 1;

      // Current element to be sorted
      int curElement = v[curIndex];

      while (tarIndex >= 0 && v[tarIndex] > curElement) {
         v[tarIndex + 1] = v[tarIndex];
         --tarIndex;
      }

      v[tarIndex + 1] = curElement;
   }
}
```

- Implement the merge sort algorithm

```cpp
void merge(std::vector<int>& v, int left, int mid, int right)
 {
  std::vector vLeft(v.begin() + left, v.begin() + mid + 1);
  std::vector vRight(v.begin() + mid + 1, v.begin() + right +
     1);

  int nLeft = vLeft.size(), nRight = vRight.size();
  int vIndex = left, lIndex = 0, rIndex = 0;

  while (lIndex < nLeft && rIndex < nRight) {
    v[vIndex++] = vLeft[lIndex] < vRight[rIndex] ?
       vLeft[lIndex++] : vRight[rIndex++];
  }

  while (lIndex < nLeft) {
    v[vIndex++] = vLeft[lIndex++];
  }

  while (rIndex < nRight) {
    v[vIndex++] = vRight[rIndex++];
  }
}

void mergeSort(std::vector<int>& v, int left, int right) {
  if (left < right) {
    int mid = (right + left) / 2;

    mergeSort(v, left, mid);
    mergeSort(v, mid + 1, right);

    merge(v, left, mid, right);
  }
}

void mergeSort(std::vector<int>& v) {
  mergeSort(v, 0, v.size() - 1);
}
```

- 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.

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

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

2. $n^2 log(n)$

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

4. 2^n

5. 2^{2^n}

Explanation:

First of all, both n^2 and $n^2 \log(n)$ share the same factor n^2. If both expressions are divided by this factor, the remaining expressions would be 1 and $\log(n)$ respectively. This would suggest that $n^2 \log(n)$ has a higher growth rate than n^2.

Using the same approach, $n^2 \log(n)$ and $n^{\log(n)}$ can be compared by finding common factors. For this reason, $n^{\log(n)}$ can be transformed to $n^2 n^{\log(n)-2}$ to meet this requirement. After removing n^2 from the equation, we are left with $\log(n)$ from the first expression and $n^{\log(n)-2}$ from the second one. It can be easily confirmed that

$$\begin{aligned} log(n) &= O(n) \\ n &= O(n^{\log(n)-2}) \end{aligned}$$

By the property of transitivity, we can conclude that $log(n) = O(n^{\log(n)-2})$ which implies $log(n) = O(n^{\log(n)})$.

The comparison of functions $n^{\log(n)}$ and 2^n can be addressed with logarithmic transformations.

$$\begin{aligned} n^{\log(n)} &= (2^{log_2(n)})^{log(n)} \\ &= 2^{log_2(n) log_2(n)} \\ &= 2^{log_2(n)^2} \end{aligned}$$

As both $2^{log_2(n)^2}$ and 2^n share the same base, the difference in growing factors can be found in exponents $log_2(n)^2$ and n. When replacing n by $\sqrt{n}^2$, the expressions remaining (after removing the common 2 exponent) are $log_2(n)$ and $\sqrt{n}$, from which $\sqrt{n}$ is well known to have a higher growing rate than $log_2(n)$. From this result we can conclude that $n^{\log(n)} = O(2^n)$.

Finally, it is trivial to show that $2^n = O(2^{2^n}$. The exponent of the first expression has a linear growing rate, while the exponent of the second one has an exponential growing rate. In this way, we can confirm that 2^{2^n} has a higher growing rate than 2^n.