homework01written

Class	CPSC 3120
Due Date	@February 5, 2025
Status	Done
Created time	@January 22, 2025 11:52 AM

R-1.7 Order the following list of functions by the big-Oh notation. Group together (for example, by underlining) those functions that are big-Theta of one another.

$$\begin{array}{lllll} 6n\log n & 2^{100} & \log\log n & \log^2 n & 2^{\log n} \\ 2^{2^n} & \lceil\sqrt{n}\rceil & n^{0.01} & 1/n & 4n^{3/2} \\ 3n^{0.5} & 5n & \lfloor 2n\log^2 n\rfloor & 2^n & n\log_4 n \\ 4^n & n^3 & n^2\log n & 4^{\log n} & \sqrt{\log n} \end{array}$$

Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

$$2^{100}, \log\log n, \sqrt{\log n}, log^2(n), n^{0.01}, \underline{3\sqrt{n}, \lceil\sqrt{n}\rceil}, \underline{5n, \lfloor 2n\log^2 n\rfloor}, \underline{6n\log n, n\log_4 n}, \underline{4n^{3/2}}, \underline{n^2\log n}, n^3, 2^n, \underline{4^n, 4^{\log n}}, 2^{\log n}, 2^{2^n}$$

R-1.8 For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t assuming that the algorithm to solve the problem takes $f(n)$ microseconds. Recall that $\log n$ denotes the logarithm in base 2 of n. Some entries have already been completed to get you started.

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$\approx 10^{300000}$			
$\sqrt{n}$				
n				
$n\log n$				
n^2				
n^3				
2^n				
$n!$		12		

solving	function	1 Second = 1,000,000us	1 Hour = 3,600s = 3,600,000,000us	1 Month = 2,628,000s = 2,628,000,000,000us	1 Century = 3,153,600,000s = 3,153,600,000,000,000us
2^(linear)	log n	10^300000=2^10^6	2^(3.6*10^9)	2^(2.628*10^12)	2^(3.1536*10^15)
linear^2	sqrt(n)	10^12	3.6*10^18	2.628*10^24	3.1536*10^30
linear :)	n	10^6	3.6*10^9	2.628*10^12	3.1536*10^15

:(expect between n and n^2 time (10E6 and 10E3)	n log n	62746	133378058	72830422138	68619394449811
sqrt(linear)	n^2	1000	6*10^4	1.621*10^6	5.6156*10^7
cbrt(linear)	n^3	100	1532	13799	146645
log_2(linear)	2^n	19	31	41	51
:(expect less than 2^n	n!	9	12	15	17

for nlogn and n! we have to do trial and error:

1000000	n	2^n	nlogn	nlogn tester referencing 2^n	n!	n! tester referencing n
1000000	1	2	2	OK	1	OK
1000000	2	4	8	OK	2	OK
1000000	3	8	24	OK	6	OK
1000000	4	16	64	OK	24	OK
1000000	5	32	160	OK	120	OK
1000000	6	64	384	OK	720	OK
1000000	7	128	896	OK	5040	OK
1000000	8	256	2048	OK	40320	OK
1000000	9	512	4608	OK	362880	OK
1000000	10	1024	10240	OK	3628800	BAD
1000000	11	2048	22528	OK	39916800	BAD
1000000	12	4096	49152	OK	479001600	BAD
1000000	13	8192	106496	OK	6227020800	BAD
1000000	14	16384	229376	OK	87178291200	BAD
1000000	15	32768	491520	OK	1307674368000	BAD
1000000	16	65536	1048576	BAD	20922789888000	BAD
1000000	17	131072	2228224	BAD	355687428096000	BAD

testing for 1 second

but i found this to be too manual and tedious so i made a quick and dirty program that found it for me:

```
#include <iostream>
#include <cmath>
#include <limits>
using namespace std;
#define ull unsigned long long

// helper function to calculate the time for f(n) = nlogn
// speed up using a binary search approach
// test at each 2^n until we exceed the time limit
// then do a binary search between the failed and the last successful n
ull find_n_nlogn(ull time_limit,
                                ull start = 1)
{
    unsigned long long n = start;
    while (true)
    {
        unsigned long long time = n * log2(n);
        if (time > time_limit)
        {
            break;
        }
        n *= 2;
    }
```

```
    // do a binary search between the last successful n and the failed n
    unsigned long long left = n / 2;
    unsigned long long right = n;
    while (left < right)
    {
        unsigned long long mid = (left + right) / 2;
        unsigned long long time = mid * log2(mid);
        if (time > time_limit)
        {
            right = mid;
        }
        else
        {
            left = mid + 1;
        }
    }
    return left-1;
}

// helper function to show that the previous n is too low, the current n is just right,
void print_nlogn_surrounding_values(ull n,
                                    ull time_limit)
{
 // ... excluded for brevity
}

// helper function to calculate the time for f(n) = n!
// because of how slow n! already is, trial and error is extremely quick, so we don't ne
ull find_n_factorial(ull time_limit)
{
    unsigned long long n = 1;
    while (true)
    {
        double time = tgamma(n + 1); // tgamma is the gamma function, which is equivaler
        if (time > time_limit)
        {
            break;
        }
        n++;
    }
    return n;
}

// helper function to show that the previous n is too low, the current n is just right,
void print_factorial_surrounding_values(ull n,
                                        ull time_limit)
{
 // ... excluded for brevity
}
```

```
int main()
{

    // only keeping the 1 centuries for brevity

    // 1 century
    cout << "1 century" << endl;
    // 3.154e+9 seconds in 1 century, 1 century = 3.154e+9 * 1E6 microseconds
    n = find_n_nlogn((3154E6) * 1E6, n);
    cout << "f(n) = nlogn: " << n - 1 << endl;
    print_nlogn_surrounding_values(n, (3154E6) * 1E6);

    // 1 century
    cout << "1 century" << endl;
    n = find_n_factorial(3154E6 * 1E6);
    cout << "f(n) = n!: " << n - 1 << endl;
    print_factorial_surrounding_values(n, 3154E6 * 1E6);

    return 0;
}
```

C-1.1 Describe how to modify the description of the MaxsubFastest algorithm so that, in addition to the value of the maximum subarray summation, it also outputs the indices j and k that identify the maximum subarray $A[j : k]$.

j will always be the index of the last zero before the M_t chain with the maximum value, so we can traverse M backwards, and upon finding m, store it as our k value (last value in the maximum). Continue reverse traversing M until the first 0 is found. This entry + 1 is our j value.

Alternatively, we can also track j and k as we iterate, maintaining the (index of the last zero + 1) as a "tempStart" variable, and updating "realStart" and "realEnd" variables every time we hit a new maximum in the "if M > m" block. (realEnd being the current index of the maximum and realStart being tempStart)

C-1.2 Describe how to modify the MaxsubFastest algorithm so that it uses just a single loop and, instead of computing $n + 1$ different M_t values, it maintains just a single variable M.

I actually already did this by accident while experimenting with my code because I saw it was essentially the same loop twice. Just take the All we have to do when combining loops is to turn the vector M into an int and then we have a single loop and a single variable M, and turn the M[t] reference into M in m= max(m,M)

C-1.3 What is the amortized running time of the operations in a sequence of n operations $P = p_1 p_2 \ldots p_n$ if the running time of p_i is $\Theta(i)$ if i is a multiple of 3, and is constant otherwise?

cost of operations where i%3==0: Θ(i)

$$\begin{aligned}\text{Cost} &= \Theta(3) + \Theta(6) + \cdots + \Theta(3k) \\ &= 3(1 + 2 + \cdots + k)\end{aligned}$$

$$\text{Sum} = \frac{k(k+1)}{2} = \frac{\lfloor n/3 \rfloor(\lfloor n/3 \rfloor + 1)}{2}$$

$$\text{Cost} = \Theta(3) \cdot \frac{\lfloor n/3 \rfloor(\lfloor n/3 \rfloor + 1)}{2} = \Theta(n^2)$$

cost of operations where i%3!=0: Θ(1)

$$\text{Cost} = \Theta(\frac{2n}{3}) = \Theta(n)$$

$$T(n) = \Theta(n^2) + \Theta(n) = \Theta(n^2)$$

$$\text{Amortized Time} = \frac{T(n)}{n} = \frac{\Theta(n^2)}{n} = \Theta(n)$$

C-1.8 Al and Bill are arguing about the performance of their sorting algorithms. Al claims that his $O(n \log n)$-time algorithm is *always* faster than Bill's $O(n^2)$-time algorithm. To settle the issue, they implement and run the two algorithms on many randomly generated data sets. To Al's dismay, they find that if $n < 100$, the $O(n^2)$-time algorithm actually runs faster, and only when $n \geq 100$ is the $O(n \log n)$-time algorithm better. Explain why this scenario is possible. You may give numerical examples.

I actually had a very similar problem to this recently with a friend and myself, where we both approached a problem I had come up with in two unique ways. My solution was rapid with small data sets (less than 50 inputs) and ran in O(n^2) time, but rapidly increased as n inputs increased. My solution was based on every possible combination of the problem, so it was a strict O(n^2) but was incredibly fast with small inputs. My friend's implementation was closer to O(nlogn), but speed was determined based on min and max values in the entire input. So if the difference between the max and min values in the set was too great, my problem would always beat his where it logically shouldn't because mine wasn't based on values from the input set. A similar case could be happening with Al and Bill, which is pretty interesting to see a real problem a friend and I had come up as an example problem for homework.