EE3980 Algorithms

HW4 Trading Stock, II

104061212 馮立俞

2018/4/1

Introduction

In this assignment, we’re required to improve the brute-force approach for maximum sum array problem in HW03 to reach $O(n^2)$ time complexity. Then, we’ll discuss whether it’s possible to device an algorithm having lower complexity than $O(n \log n)$.

Approach

Recap: Brute-force Approach

```
 Algorithm MaxSubArrayBF(A, n, low, high) // Find low and high to
                         maximize ΣA[i ], low≦i ≦high.
 {
     max: = 0;low: = 1;high: = n;
     for j: = 1 to n do { // Try all possible ranges: A[j : k ].
         for k: = j to n do {
                 sum  = price[high] - price[low];  // n^2 complexity version

                 sum: = 0;                      //
                 for i: = j to k do {           //
                         sum: = sum + A[i];     // n^3 complexity version,
                 }                              // could choose either one

                 if (sum > max) then {
                    // Record the maximum value and range.
                     max = sum;
                     low = j;
                     high = k;
                 }
         }
     }
  return max;
}
```

Since we've known the prices of stock in the given time, the price change over a certain period can be obtained by subtracting the high price with low one. Doing so would save us a loop; thus improve the overall complexity from $O(n^3)$ to $O(n^2)$. Space complexity would remain the same nonetheless, i.e. $O(n)$.

Linear-Complexity Approach

```
Algorithm LinearApproach(Prices[], N) {
    max_sum: = 0,
    sum: = 0;start: = 0,
    end: = 0,
    temp_start: = 0;
    for i from 0 to N - 1: {
        sum += a[i];
        if (sum < 0) {
            sum: = 0;temp_start: = i + 1;
        } //restart record
        if (sum > max_sum) {
            max_sum = sum;
            start = temp_start;
            end = i;
        } //update record
    }
    if (start >= end)
        do something // array all negative, error handling
    return max_sum, start, end;
}
```

In this implementation, only one single loop is used. Obviously the time complexity is $O(n)$. Plus, no other large memory space is required, so the space complexity is $O(n)$.

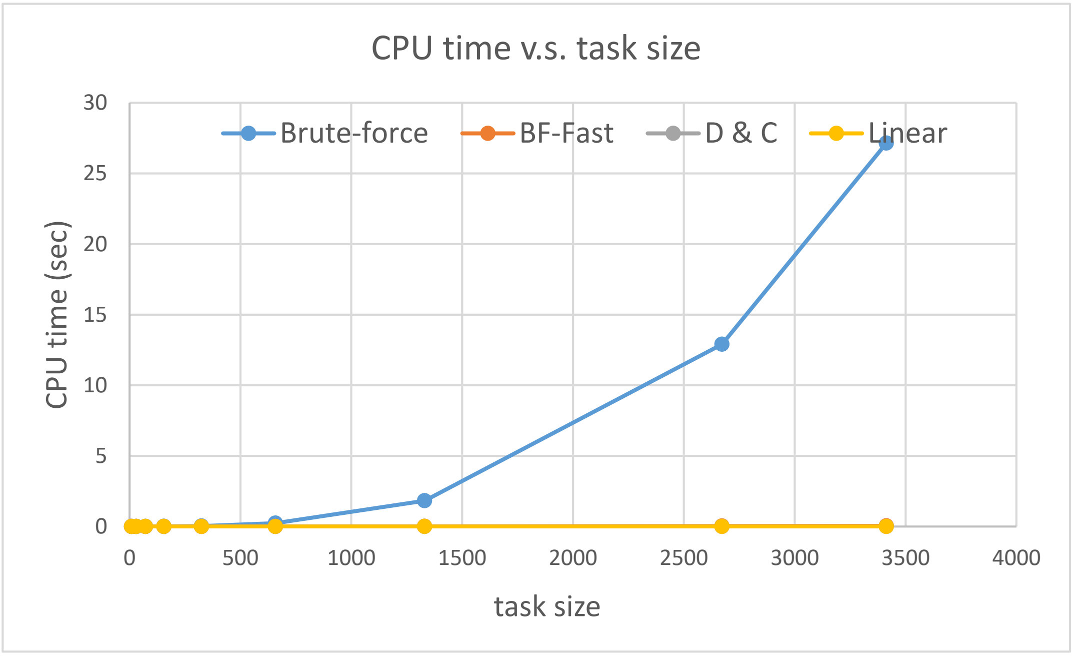
In the algorithm, we keep adding new term to current sum, restart our record if current sum is less than zero, and update `max_sum` and `start` & `end` if current sum is greater than `max_sum`. Simple as that, yet it works (at least intuitively and empirically for this homework)!

Results and analysis

Table 1. Algorithm CPU time (in seconds) vs. task size

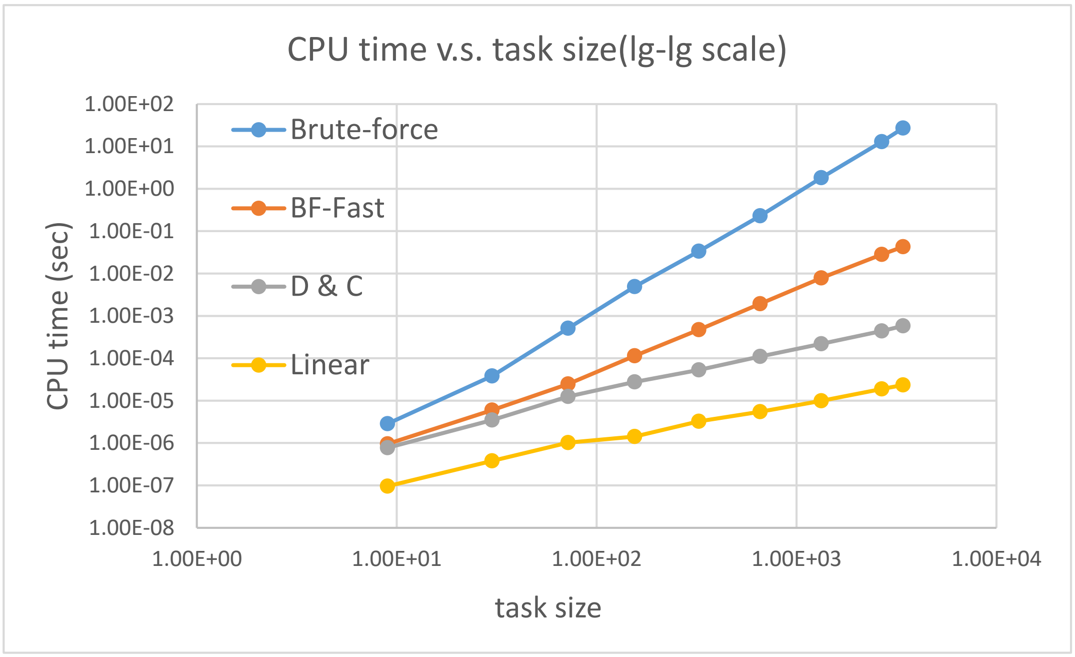
task size	9	30	72	155	325	658	1331	2672	3414
Brute-force	2.86E-06	3.81E-05	0.000509	0.0049	0.0336	0.23	1.82	12.9	27.1378
BF-Fast	9.54E-07	5.96E-06	2.48E-05	1.14E-04	4.73E-04	1.93E-03	7.84E-03	2.83E-02	4.27E-02
D & C	7.72E-07	3.48E-06	1.25E-05	2.76E-05	5.33E-05	1.10E-04	2.19E-04	4.41E-04	5.79E-04
Linear	9.61E-08	3.78E-07	1.02E-06	1.42E-06	3.26E-06	5.46E-06	9.93E-06	1.88E-05	2.35E-05

We can plot above table as follows

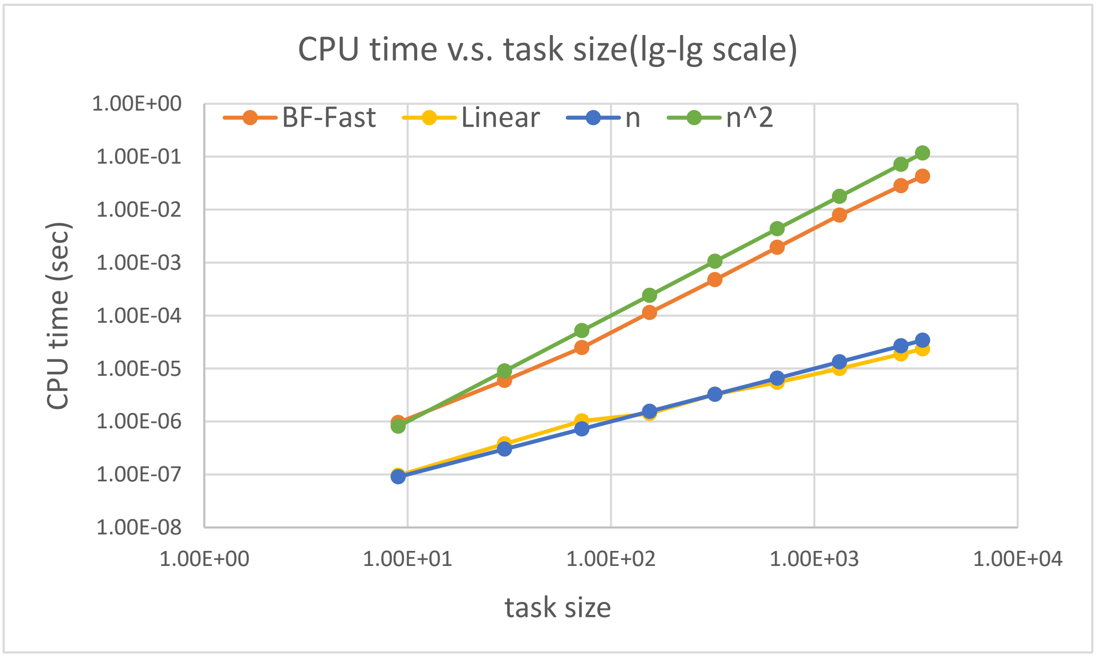


Well, the complexity of the four algorithms are basically not on the same league.

It's pretty hard to plot them on linear scale without some curves being suppressed.



We expect their complexity to be $O(n^3)$, $O(n^2)$, $O(n \log n)$, $O(n)$ respectively. And the curves above are quite fit.



We can further examine the complexity by plotting $O(n^2)$ and $O(n)$ curves with them, and they fit well too.

Observations and Conclusion

“Brevity is the soul of wit”, and the same philosophy applies for algorithms too.

With some ingenuity, a great amount of time could be saved.