V00034442

Kobel Iragaoly

$T(n) = 1+2+\cdots n$

a_0

$a_1 \times x$ [illegible]

$a_2 \times x^2$ [illegible]

1. An n-degree polynomial $p(x)$ is an equation of the form

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

where x is a real number and each a_i is a real constant, with $a_n \neq 0$.

(a) Describe a naive $O(n^2)$-time method for computing $p(x)$ for a particular value of x. Justify the runtime.

(b) Consider now the nested form of $p(x)$, written

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + xa_n)\ldots))).$$

Using the big-Oh notation, characterize the number of multiplications and additions this method of evaluation uses.

2. In the 3SUM problem, we are given as input an array A of n distinct integers and the goal is to check if there are three integers in A that sum to 0. Design a $O(n^2)$ running time algorithm for 3SUM. Describe your algorithm in pseudocode and show that its running time is $O(n^2)$.

3. Recall the LinearSelect algorithm we learnt in the class. Suppose that we modify the algorithm to use groups of size 3 instead of 7. Show that the modified algorithm does not run in $O(n)$ time. You may follow the lecture slide examples from lecture 3 when developing your recurrence equation $T(n)$ for this version of the algorithm. That is, you can use upper bounds to get rid of the ceiling notation and assume an in-place implementation which costs nothing to separate into subsequences. [Note: For a subsequence of 3 elements, it takes at most 3 comparisons to sort them.]

1, General form of a polynomial of degree n is :

$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \cdots a_1 x^1 + a_0$ where a_n is non-zero for all $n >= 0$

$$p(x) = a_n \times \underbrace{x \times x \times x \cdots x}_{n \text{ times}}$$
$$+ a_{n-1} \underbrace{\times x \times x \times \text{------} x}_{} \quad n-1 \text{ times}$$
$$+ a_{n-2} \underbrace{\times x \times x \times \text{-----} x}_{} \quad n-2 \text{ times}$$
$$\vdots$$
$$+ a_2 \underbrace{\times x \times x}_{} \quad 2 \text{ times}$$
$$+ a_1 \underbrace{\times x}_{} \quad 1 \text{ time}$$
$$+ a_0$$

So the number of multiplications is

$$T(n) = n + (n-1) + (n-2) + \cdots 2 + 1$$

To calculate this, I sum n terms in forward and backward.
and sum two T(n) and divide by 2.

$$T(n) = 1 + 2 + \cdots (n-2) + (n-1) + n \qquad \text{forward}$$
$$+\big)\ T(n) = n + n-1 + \cdots\ 3 + 2 + 1 \qquad \text{backward}$$
$$2T(n) = (n+1) + (n+1) + \cdots (n+1) + (n+1) + (n+1)$$
$$2T(n) = n(n+1)$$
$$T(n) = \frac{n(n+1)}{2}$$

+n multiplications

Therefore $T(n) = \frac{n^2}{2} + \frac{n}{2}$, so this polynomial runtime is $O(n^2)$.

(b) $p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 \cdots + x(a_{n-1} + x a_n))\cdots)$

$\downarrow$

$$\left.\begin{array}{ll} = a_0 & \\ + & \\ x(a_1 + & 1 \text{ times multiplication} \\ x(a_2 + & 1 \text{ times} \cdots \\ \vdots & \vdots \\ x(a_{n-1} + & 1 \text{ times} \cdots \\ x(a_n))))\cdots) & 1 \text{ times} \cdots \end{array}\right] \rightarrow n \text{ times multiplication and addition}$$

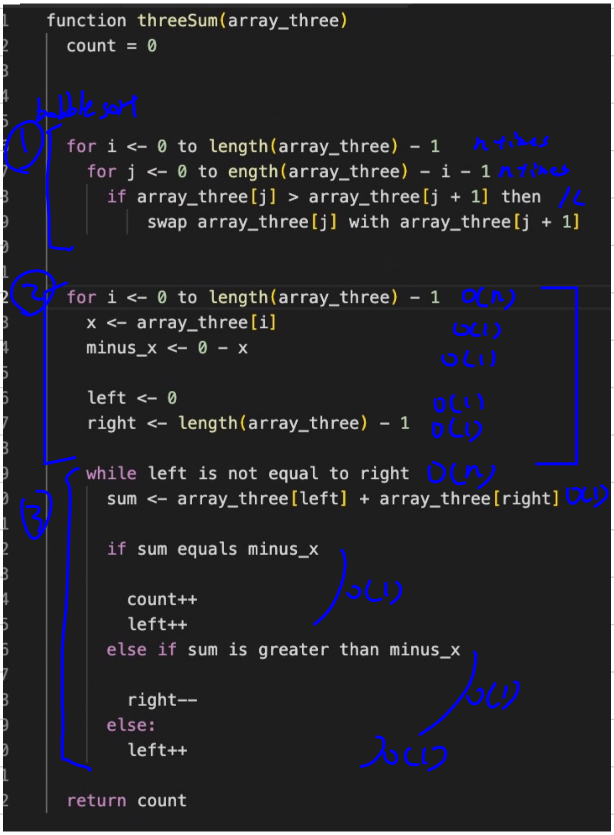
$$\therefore T(n) = \underset{\text{multiplications}}{n} + \underset{\text{additions}}{n} = 2n \rightarrow O(n)$$

2. In the 3SUM problem, we are given as input an array A of n distinct integers and the goal is to check if there are three integers in A that sum to 0. Design a $O(n^2)$ running time algorithm for 3SUM. Describe your algorithm in pseudocode and show that its running time is $O(n^2)$.

This is my pseudo code on the right.
In the algorithm, I use two-pointers to search for the desired triplets in the sorted array. By compareing the sum of elements with the target value, it adjustes pointers depends on different combinations

1. Sort the array called "array_three". by using bubble sort algorithm. Let's say N is the length of the array. The running time of this bubble sort is $O(N^2)$ The worst case running time is also $O(N^2)$ since in the worst case, it requires $N-1$ passes over the array.

2. After sorting, I use for loop to calculate negation of 'x', and store it in the variable called 'minus_x' Also it initialized left = 0 and right = the end of the array.



```
function threeSum(array_three)
  count = 0


  for i <- 0 to length(array_three) - 1
    for j <- 0 to ength(array_three) - i - 1
      if array_three[j] > array_three[j + 1] then
        swap array_three[j] with array_three[j + 1]


  for i <- 0 to length(array_three) - 1
    x <- array_three[i]
    minus_x <- 0 - x

    left <- 0
    right <- length(array_three) - 1

    while left is not equal to right
      sum <- array_three[left] + array_three[right]

      if sum equals minus_x

        count++
        left++
      else if sum is greater than minus_x

        right--
      else:
        left++

  return count
```

3. Uses while loop which continues until left = right. and put the sum of values of left and right into a variable called sum. if sum = minus_x, this means we found a valid triplet. So, I increment count variable and move the left pointer one step to the right to find next triplet.

If sum is greater than minus_x, it means the current sum is too large.
So I decrement right pointer.
If sum is less than minus_x, it means current sum is too small,
So I increment left pointer.
After the while loop ends, I return the variable called "count" which is the number of triplets.

Since outer loop has running time $O(n)$ and the inner while loop has running time $O(n)$ from step 2 to 3, the running time of steps 2 to 3 is $O(n^2)$.
Also for step 1, the running time is $O(n^2)$, so overall running time is $O(n^2)$.

① Divide S into equal-sized [illegible] groups of 3 [illegible]

$\lceil \frac{n}{3} \rceil$ elements $O(n)$

② Sort each groups

[illegible] $\lceil \frac{n}{3} \rceil = \frac{3 + 3}{2} \cdot \frac{n}{3} = 2n$

$O(n)$ use linear search [illegible] [illegible]

[illegible] and [illegible] $O(n)$

3. Recall the LinearSelect algorithm we learnt in the class. Suppose that we modify the algorithm to use groups of size 3 instead of 7. Show that the modified algorithm does not run in $O(n)$ time. You may follow the lecture slide examples from lecture 3 when developing your recurrence equation $T(n)$ for this version of the algorithm. That is, you can use upper bounds to get rid of the ceiling notation and assume an in-place implementation which costs nothing to separate into subsequences. [Note: For a subsequence of 3 elements, it takes at most 3 comparisons to sort them.]

1. Divide S into equal-sized groups of 3 elements
   $\rightarrow \lceil \frac{n}{3} \rceil$ groups of size 3. This takes $O(1)$

2. Sort each group of size 3 completely. $\frac{3\times2}{2\times1}$
   Using 2 comparisons takes $\lceil \frac{n}{3} \rceil \times \binom{3}{2} = \lceil \frac{n}{3} \rceil \times 3 = \underline{n}$

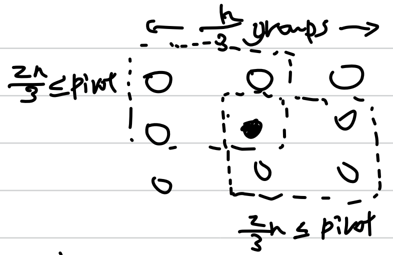
3. Determine the median of each group
   pick the middle element of each group $O(1)$.
   Gather all medians in a sequence ... $\underline{\text{takes } n}$

4. Use LinearSelect recursively to determine the median of medians
   If the running time of LinearSelect is $T(n)$,
   the median of $\lceil \frac{n}{3} \rceil$ medians takes about $\underline{T(\frac{n}{3})}$

$\therefore$ The time complexity of clever pivot computation is $2n + T(\frac{n}{3})$

For partitioning, it takes n times.

For conquer and recursive call,
by selecting the pivot this way, we know that $2 \times \lceil \frac{\frac{n}{3}}{2} \rceil$
are smaller than the pivot. (also larger than the pivot)



$\therefore$ In the worst case $\frac{n}{3}$ elements at partitioning are in L and $\frac{2}{3}n$ are in G
we continue searching for the k^{th} element in $\frac{2}{3}n$ elements
$\rightarrow$ Conquer part takes $\underline{T(\frac{2}{3}n)}$

- Clever pivot selection $2n + T(\frac{n}{3})$
- partition n
- Conquer recursive call $T(\frac{2}{3}n)$

$\rightarrow$ linearSelect $T(n) \le 3n + T(\frac{n}{3}) + T(\frac{2}{3}n)$

Now, Proof.

Guess $T(n) \le Cn$

$T(n) = 3n + T(\frac{n}{3}) + T(\frac{2}{3}n)$

$\rightarrow 3n + c(\frac{n}{3} + \frac{2}{3}n] \le Cn$

$\rightarrow 3n + Cn \le Cn$

$3n \le 0$

$\therefore n \notin O(n)$ ■

D

1. **The Master Theorem**

$$\begin{aligned} T(n) &= c \text{ if } n < d \\ &= aT\left(\frac{n}{b}\right) + \Theta(n^c) \text{ if } n \geq d \end{aligned}$$

(a) If $c < \log_b a$, then $T(n)$ is $\Theta(n^{\log_b a})$.

(b) If $c = \log_b a$, then $T(n)$ is $\Theta(n^c \log n)$.

(c) If $c > \log_b a$, then $T(n)$ is $\Theta(n^c)$.

Solve the following recurrence equations using the Master Theorem given above.

(a) $T(n) = 16T(n/4) + n^4$

(b) $T(n) = 125T(n/5) + n^2$

(c) $T(n) = 64T(n/8) + n^2$

(a) $T(n) = 16T(n/4) + n^4$

$a = 16 \quad b = 4 \quad c = 4$

$\log_b a = \log_4 16 = \log_4 4^2 = 2$

$(\log_b a = 2) < (c = 4)$

Therefore $T(n) \in \Theta(n^4)$

(b) $T(n) = 125T(\frac{n}{5}) + n^2$

$a = 125 \quad b = 5 \quad c = 2$

$\log_b a = \log_5 125 = \log_5 5^3 = 3\log_5 5 = 3$

$(3 = \log_b a) > (c = 2)$

Therefore $T(n) \in \Theta(n^{\log_b a})$

$\in \Theta(n^3)$

(c) $T(n) = 64T(\frac{n}{8}) + n^2$

$a = 64 \quad b = 8 \quad c = 2$

$\log_b a = \log_8 64 = \log_8 8^2 = 2\log_8 8 = 2$

Since $(\log_b a = 2 = c)$

$T(n) \in \Theta(n^c \log n)$

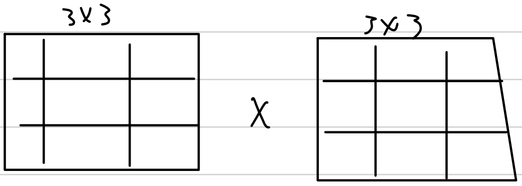
$\in \Theta(n^2 \log n)$

5. Suppose that your goal is to come up with a new algorithm for matrix multiplication whose running time is better than the $O(n^{2.807})$ running time of Strassen's matrix multiplication algorithm.

Your plan to achieve this by coming up with a new method for multiplying two 3×3 matrices using as few multiplications as possible. Show that in order to beat Strassen's algorithm, your method must use 21 multiplications or less. (Hint: Write down the recurrence equation for your method similar to the equation for Strassen's algorithm in the slides and use Master Theorem to solve your recurrence).

First, The running of Strassen's matrix multiplication is $O(n^{2.807})$ and we would like to beat this algorithm by using 21 multiplications or less.

Now, Let's think about two 3X3 matrices.



From the master theorem, I know that

$$T(n) = aT(\tfrac{n}{b}) + \Theta(n^c) \qquad a \geq 1,\ b \geq 2,\ c \geq 0$$

with 3 cases:

Case 1 : If $c > \log_b a$, then $T(n) = \Theta(n^c)$

Case 2 : If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$

Case 3 : $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$

I think about the case of 21 multiplications now with 3X3 matrix.

then $\underset{\text{number of sub problems}}{a = 21}$, and $\underset{\text{the factor which the subproblem size decreases}}{b = 3}$

$$\Theta(n^{\log_3 21}) = \Theta(n^{2.771}) < \Theta(n^{2.807})$$

and if "a" (number of multiplications) decreases $\Theta(n^{\log_b a})$ also decreases.

Therefore, we must use 21 multiplications or less to beat Strassen's algorithm.