* Homework 3 - 20010400403y - Yağım Yalçın

(1)

a-) The algorithm needs to run for n-1 sized array each time it runs.
→ It does not divide the problem.
→ The array length becomes n-1 each time.
→ The basic operation is the if statement inside which does a comparison. It is $\Theta(1)$.

$$\rightarrow T(n) = T(n-1) + \Theta(1)$$

b-)
→ It divides the problem into 2 parts.
→ The array length is halved for each problem in each iteration.
→ The basic operation is the if statement inside which does a comparison. It is $\Theta(1)$.

$$\rightarrow T(n) = 2T(\tfrac{n}{2}) + \Theta(1)$$

* $T(n) = T(n-1) + \Theta(1)$

* Solving by substition

$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
$T(n-2) = T(n-3) + 1$
⋮
$T(n) = T(n-3) + 1 + 1 + 1$
This part goes on
At the end
$n \times 1 = n$

$$\rightarrow T(n) \in \Theta(n)$$

$T(n) = 2T(\frac{n}{2}) + \Theta(1)$

* Solving with Master Theorem:

$a = 2$
$b = 2$
$\log_b^a = 1$

$\log_b^a = n^1$
$f(n) = \Theta(1)$
$f(n) \in O(n)$

$$\rightarrow T(n) \in \Theta(n)$$

Both of the algorithms have $\Theta(n)$ time complexity therefore both can be chosen over the other. Does not matter. The first one uses a bit less space. If we really need to choose one, that can be preffered.

(2) Brute force algorithm to compute a polynomial at a given point x_0.

↳ $p(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$

* A brute force method of finding the result can be achieved by simply calculating all results of a_n's with the given x and then calculating it.
  - ↳ Calculate x^n
  - ↳ Multiply with a_n
  - ↳ Repeat for all
  - ↳ Sum them up

  Calculating x^n is $O(n)$ time because n multiplications are needed. Then multiply it with a_n, Then repeat for all a_n meaning an $O(n) \Rightarrow$ ↳ $O(n^2)$ //

* Calculating x^n by multiplying x n times is not efficient. This can be done in less time if we write a special power operation.
  ↳ We can calculate $x^{\frac{n}{2}}$ then work with it:
  if n is even then
  $x^{\frac{n}{2}} \cdot x^{\frac{n}{2}} = x^n$ → Here we need to calculate $x^{\frac{n}{2}}$ then need only 1 more extra multiplication
  if n is odd then:
  $x^{\frac{n-1}{2}} \cdot x^{\frac{n-1}{2}} \cdot x$ → Here we need to calculate $x^{\frac{n-1}{2}}$ then need only 2 more extra multiplications.
  ↳ This halves the operation time for each input making it $O(\log n)$
  ↳ In the end evaluating polynomial becomes $O(n \log n)$ //

* If we want to think about a solution aside from brute force algorithms, we can use Horner's Method. It is $O(n)$ time.

* The implementation of $O(n^2)$ algorithm is in polynomial_[2001040034].py file.

(3) Brute force algorithm to count the number of substrings that start with a specific letter and ends with a specific letter.

* A brute force method of finding the result can be achieved by trying each letter to form the requested substring.
  - ↳ Start traversing
  - ↳ Analyze a letter
  - ↳ Skip if it does not match start letter
  - ↳ Continue if it matches the start letter
  - ↳ Continue till the end
  - ↳ If not found return back to the following letter of the analyzed letter
  - ↳ If found, increment the count for each found
  - ↳ Return back to the following letter of the analyzed letter
  - ↳ Repeat till the end.

Checking will be done for each letter. $O(n)$. During checks, in the worst case, there is no substrings and checks continue till the end. This makes $(n-x)$ checks for all elements. (x = place of the analyzed letter.) For this to happen all letters must be the start letter.

$$\hookrightarrow \sum_{i=0}^{n-2} (n-i) \Rightarrow n\sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$\Rightarrow n(n-1) - \frac{(n-2)(n-1)}{2}$$

$$\Rightarrow \frac{n^2-n}{2} \in O(n^2)$$

* In the best case; There is no start letter so only a simple traversal is done $\Rightarrow O(n)$

* The implementation is in count-str-[20010400409].py file.

(4) Brute force algorithm to find the closest pair in space.

* A brute force method of finding the result can be achieved by calculating the distance between each pair. Storing the min result. Changing the min result if a smaller result is found.

```
* d ← ∞
  for i ← 1 to n-1 do
      for j ← i+1 to n do
          d ← min( d, Euclidean_distance (points[i], points[j], k))
      end for
  end for
  return d
end
```

* The run time of the euclidean_distance method depends on k which is the space dimension. Because in this function k times substractions will be done. Euclidean_distance function is the basic operation of the loops.

$$A(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} k \Rightarrow \sum_{i=1}^{n-1} \left(k \sum_{j=i+1}^{n} \right) \Rightarrow \sum_{i=1}^{n-1} (k(n-i)) \Rightarrow$$

$$\sum_{i=1}^{n-1} (kn - ki) \Rightarrow \sum_{i=1}^{n-1} kn - \sum_{i=1}^{n-1} ki \Rightarrow kn \underbrace{\sum_{i=1}^{n-1} 1}_{i=1} - k \underbrace{\sum_{i=1}^{n-1} i}_{i=1}$$

$$\underbrace{kn(n-1) - \frac{k(n-1)n}{2}}$$

$$kn^2 - kn - \frac{kn^2 - kn}{2} \in \theta(kn^2)$$

* k here can be any value from 2 to ∞.

(5)

a-> Brute force algorithm that can find the most profitable cluster.

* A brute force method of finding the result can be achieved by calculating the sum of every possible subarray. Storing the start and end points of the subarray that gives the largest sum.
  - ↳ Start with the first element
  - ↳ Calculate the sum of its size 2 subarray's sum.
  - ↳ Repeat till size n subarray
  - ↳ Save the start and end points that gives the max sum during the process
  - ↳ Repeat for all elements.

This operation will be done for all element and for all element n-i subarays will be checked.

$$A(n) = \sum_{i=0}^{n-1} (n-i) \Rightarrow$$

$$n \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i \Rightarrow$$

$$n \times n - \frac{(n-1)n}{2} \Rightarrow$$

$$n^2 - \frac{(n^2-n)}{2} \in \Theta(n^2)$$

b-) A divide and conquer method of finding the profit result can be achieved by dividing the array into 2 parts and calculating their max separetly. In the end calculate if there is a better subarray that is in the intersection.
  - ↳ Find max profit of left half
  - ↳ Find max profit of right half
  - ↳ Find max profit of intersection

* By doing this we halve the problem each time. The main problem here is finding max profit of the intersection which can be done in $\Theta(n)$ time.

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

* Solving with master's theorem

$a=2$, $b=2$, $\log_b^a = 1$

$f(n) = n$, $n \in \Theta(n)$

$n^{\log_b^a} = n$

$\Rightarrow T(n) \in \Theta(n \log n)$