CSL351: Analysis and Design of Algorithms
Assignment No. 3

Sahil
2016UCS0008

September 12, 2018

Ans-1 We have been given an array of 2n elements

$\{a_1, a_2, a_3, a_4, \ldots, a_n, b_1, b_2, b_3, \ldots, b_n\}$

To shuffle the array to

$\{a_1, b_1, a_2, b_2, \ldots a_n, b_n\}$

without using any extra space, we notice that we must perform some swap operations.

(a) Brute force approach

- we first keep swapping b_1 to its left position till it reaches next to a_1.
- Similarly, we do for $b_2, b_3, \ldots, b_n$.

At the end, we will have the required shuffled array.

* Pseudo Code —

```
// consider array arr[0, 1, ---, n-1]
// we need to swap b1 till it reaches 1st index
// b2 till it reaches 3rd index
// so the inner loop will run till next odd no.
last ← 1
for i ← n/2 to n-1
    // swap arr[i] till it reaches at last index (j)
    j ← i
    while j > last
        swap(arr[j-1], arr[j]);
        j--;
    last ← last + 2
end
```

→ The complexity of this algorithm can be computed as:

when $i = \frac{n}{2}$, j will go from i till 2
$\Rightarrow \frac{n}{2} - 1$ swaps

when $i = \frac{n}{2} + 1$, j will go from i till 4
$\Rightarrow \frac{n}{2} - 2$ swaps

⋮

$i = \frac{n}{2} + \frac{n}{2} \Rightarrow$ 1 swap.

so, in total $T(n) = \sum_{i=1}^{(n/2-1)} \frac{n}{2} - i = \frac{n}{2}\left(\frac{n}{2} - 2 + 1\right) - \frac{\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right)}{2}$

$$\Rightarrow T(n) = \frac{\left(\frac{n}{2}-1\right)\left(\frac{n}{2}\right)}{2} = \frac{n^2-2n}{4}$$

Thus, $\boxed{T(n) \in \Theta(n^2)}$

(b) Divide and Conquer Approach

Let us take eg $arr = [a_1, a_2, b_1, b_2]$

Here, we must swap a_2 & b_1 somehow to get resultant array $= [a_1, b_1, a_2, b_2]$

So, we can divide the array into 2 parts

left $= [a_1, a_2]$, right $= [b_1, b_2]$

so, we must swap right half of left with left half of right

$\Rightarrow$ left $= [a_1, b_1]$, right $= [a_2, b_2]$

In general case, we have

$a_1, a_2, \dots, a_{\frac{n}{2}}, a_{\frac{n}{2}+1}, \dots a_n,$

$b_1, b_2, \dots, b_{\frac{n}{2}}, b_{\frac{n}{2}+1}, b_{\frac{n}{2}+2} \dots b_n$

We divide into 2 halves and swap right half of first with left half of second.

$\Rightarrow a_1, a_2, \dots a_{\frac{n}{2}}, b_1, b_2, \dots b_{\frac{n}{2}}$

$b_{\frac{n}{2}+1}, \dots a_n, b_{\frac{n}{2}+1} \dots b_n$

Thus, the problem can now be solved independently for the 2 halves.

In the base case, when we have only two elements, we simply return.

Pseudo-code –

```
// arr[0,1,---n-1] is the input array
shuffle (arr, low, high) =
        if (high-low == 1)
            return;
        mid ← (low + high)/2;
        left-mid ← (low + mid)/2;
        i ← left_mid;
        j ← mid + 1;
        while (i <= mid)
            swap(a[i], a[j]);
            i++;
            j++;
        shuffle (arr, low, mid);
        shuffle (arr, mid+1, high);
```

- It can be run by using shuffle (arr, 0, n-1) call.
- Since for a problem of size n, we divide it into 2 problems of size n/2 and do $\frac{n}{2}$ swaps

Thus,

$$T(n) = 2T\left(\frac{n}{2}\right) + \frac{n}{2} \quad , \; T(2) = 1$$

Using master theorem

$a = 2, \; b = 2, \; d = 1$

$b^d = 2$

since $a = b^d \Rightarrow T(n) = \Theta(n^d \log n)$

$\Rightarrow \boxed{T(n) = \Theta(n \log n)}$

ms-2 The assignment that minimizes the total cost in the given problem is

Person 0 — Job 3 ⇒ cost = 3
Person 1 — Job 2 ⇒ cost = 5
Person 2 — Job 0 ⇒ Cost = 5
Person 3 — Job 1 ⇒ Cost = 6
Total = ~~29~~ 19

When n people are to be assigned to n jobs, there can be n! total assignments possible. We can use exhaustive search method by recursion to ~~do~~ find all assignments and ~~check~~ find what is the minimum cost.

Pseudo-code -

```
// cost[n][n] is the cost matrix
// cost[i][j] denotes the cost to assign job "j" to "ith" person
min_cost ← +∞ ;
done[n] ← false; // boolean array of size n
                 // to see if job already assigned
assign ( person_index, current_cost ) =
    if (person_index == n) then
        if (current_cost < min_cost)
            min_cost ← current_cost;
        return;
    ~~for job ← 0 ← n~~
    for job ← 0 to n-1 do
        if (not done[job])
            done[job] = true;
            assign( person_index + 1, current_cost + cost[i][j]);
            done[job] = false;
end
```

Here, in recursive call to assign
cost[i][j] = cost[person-index][job].

The above algorithm has a time-complexity O(n!).

Ans-3 We are given a knapsack of capacity B.
There are infinite objects of similar type and each object of same type has the same weight wi and cost ci.
- We need to pack the objects in knapsack to maximize the total cost of the objects.
So, it might not be the case that we should pick the costly objects first since they might have large weight but picking lighter objects with lesser cost might lead to larger cost.
- Thus, we need to try picking all objects which can be picked and solve a problem with reduced capacity. While solving any smaller subproblem, we need to take the maximum cost over all possibilities.
- So, we can write a recursive knapsack procedure.

We also need to print the counts of the chosen objects of different types, so, during each recursive call, we can simply find the index of object that gave the maximum cost and increment its count by 1.

Also, to make sure that we don't solve the same subproblems again, we can store the answer as we solve it and always check if already solved.

Pseudo-code

```
// wt[n] represents the weights
// c[n] represents the costs
count[n] = new array initialized with all zeroes
solved[W+1] = new array initialized with -1
knapsack(W) =
  if (solved[W] != -1)
      return solved[W];
  maxcost = 0;
  chosen = 0; // index of chosen item
  for i <- 0 to n-1
    if (wt[i] <= W)
      if ( c[i] + knapsack(W - wt[i]) > maxcost)
          maxcost = c[i] + knapsack(W - wt[i]);
          chosen = i;
  end
  solved[W] = maxcost;
  count[chosen]++;
  return maxcost;
```

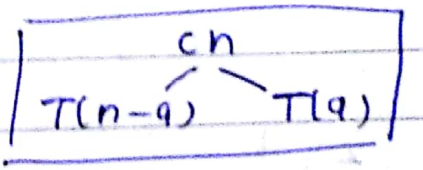
There can be atmost W subproblems and since each will be solved only once and time taken for one subproblem = O(n)

Thus, total time complexity = O(nW)

Ans-4 $T(n) = T(n-a) + T(a) + cn$

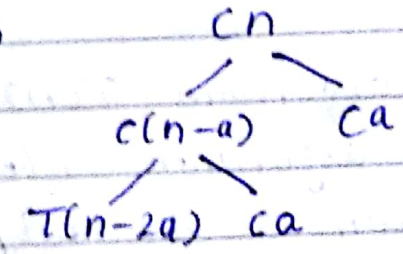
where $a >= 1$ and $c > 0$ are constants.

The recursion tree can be built in the ~~follow~~ manner shown below:

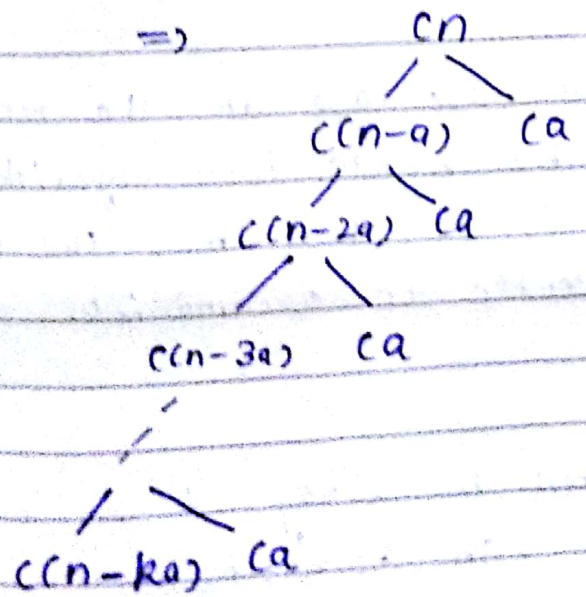


Since a is constant, we can consider $T(a) = ca$

=>



=>



At the last level, we have $T(n-ka) + ca$

So, considering $T(0) = 1$

$\Rightarrow n - ka = 0$

$\Rightarrow n = ka$

$\Rightarrow k = \frac{n}{a}$

We can add complexity at each level in the tree

$$T(n) = \sum_{i=0}^{n/a} (c(n-ia) + ca)$$

$$= \sum_{i=0}^{n/a} [c(n+a) - cai]$$

$$= c(n+a)\left(\frac{n}{a}+1\right) - ca \sum_{i=0}^{n/a} i$$

$$= \frac{cn^2}{a} + (c+1)n + a - \frac{c\cancel{a}\left(\frac{n}{\cancel{a}}\right)\left(\frac{n}{a}+1\right)}{2}$$

$$= \frac{cn^2}{a} + (c+1)n + a - \frac{cn^2}{2a} - \frac{cn}{2}$$

$$\boxed{T(n) = \frac{cn^2}{2a} + \left(c + \frac{1}{2}\right)n + a}$$

Thus, $\boxed{T(n) = \Theta(n^2)}$

Ans-5 The recursive algorithm ($T(n) = n \log n$) beats the brute force algorithm ($T(n) = n^2$) at $n_0 = 30$ as shown in the output displayed when the given code was run.

If we change the base case of the recursive algorithm to use the brute-force algorithm when the problem size is less than n_0, the crossover point changes a little to around 20.

Q.6 (a) Recurrence relation for the Strassen's algorithm is given by

$$T(n) = 7T(n/2) + \Theta(n^2), \quad T(1) = \Theta(1)$$

Comparing with the general form to apply master's theorem, i.e. $T(n) = aT(n/b) + \Theta(n^d)$

$a = 7, \; b = 2, \; d = 2$

$\Rightarrow b^d = 2^2 = 4$

Since $a > b^d \Rightarrow T(n) = \Theta(n^{\log_b a})$

$= \Theta(n^{\log_2 7})$

The recurrence for algorithm developed by Prof. Caesar is

$$T(n) = aT(n/4) + \Theta(n^2)$$

Here, a = unknown, b = 4, d = 2

So, $b^d = 4^2 = 16$

Let $a > b^d$ So, $T(n) = \Theta(n^{\log_4 a})$

For it to beat ~~Gaesar's~~ Strassen's algorithm,

$$n^{\log_4 a} < n^{\log_2 7}$$

$$\Rightarrow \frac{1}{2}\log_2 a < \log_2 7$$

$$\Rightarrow \log_2 a - \log_2 7^2 < 0$$

$$\Rightarrow \log_2\left(\frac{a}{49}\right) < 0 \quad \Rightarrow \underline{a < 49} \rightarrow (1)$$

Not Considering cases $a = b^d$ and $a < b^d$, because the time complexity in these cases would be $\Theta(n^d \log n)$ and $\Theta(n^d)$ which is better already than $\Theta(n^{\log_2 7})$ for $d = 2$ and also we want the largest value for a,

Thus, largest integer value for $\boxed{a = 48}$ from (1).

(b) $T(n) = aT(n/2) + \Theta(1)$

$b = 2$, $d = 0 \Rightarrow b^d = 1$

since we have $\boxed{a = 1}$ for binary search because we reject one half after comparing the value to be searched with middle element,

thus $a = b^d = 1$

$\therefore$ by Master's Theorem

$$T(n) = \Theta(n^d \log n) = \boxed{\Theta(\log n)}$$

Ans-7 (a) The total no. of operations performed in the given algorithm are

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=i}^{n} 1$$

$$= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} (n-i+1)$$

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

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

$$= (n+1)(n-1) + n(n-2) + (n-1)(n-3) + \cdots + 3*1$$

$$= \sum_{j=1}^{n-1} (j+2)j = \sum_{j=1}^{n-1} j^2 + \sum_{j=1}^{n-1} 2j$$

$$= \frac{(n-1)(n)(2n-1)}{6} + 2\frac{(n-1)n}{2}$$

$$= \frac{n(n-1)(2n+5)}{6} = \frac{2n^3+3n^2-5n}{6}.$$

Thus, $T(n) = \Theta(n^3)$

So, the above algorithm has cubic complexity.

(b) In the innermost loop, we are multiplying by $\frac{A[j,i]}{A[i,i]}$ each time but j and i are fixed for inner loop. Thus, instead of computing the same quantity each time, we can precompute it before inner loop and after storing in a variable, use it directly.

Thus, the modified algorithm is:

```
for i ← 0 to n-2 do
    for j ← i+1 to n-1 do
        val = A[j,i]/A[i,i]
        for k ← i to n do
            A[j,k] ← A[j,k] - A[i,k] * val
```

Ans-8 We have to determine whether there exist two points such that distance between them is less than 1. This can be done using the ~~closet~~ closest pair of points algorithm.

So, we first divide the list Lx into 2 equal subsets of size n/2 each by finding a vertical line that does not pass through any of the points. The Lx and Ly can be computed for both the left and right halves.

Computing Lx is straight forward, we simply divide Lx at the x-coordinate of the line. For Ly, we simply check if the current point has x-coordinate less than that of line, then keep it on left part otherwise on right. We do it for all points in Ly, ultimately the 2 lists obtained will be sorted ~~on~~ since originally Ly is sorted.

Now, having divided the problem into 2 halves, we have to define a base case. We can consider $n \leq 3$ as the base case and simply find ~~the~~ whether any pair has distance < 1, If yes, we return it.

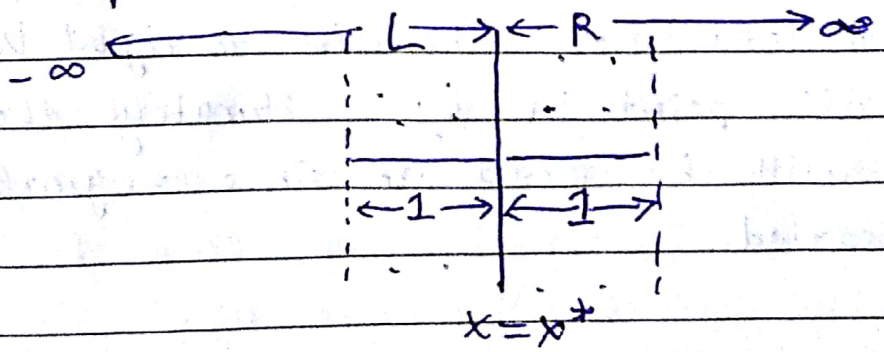
Now, considering the merge operation.

If we have already found a pair either on the left side or right side of line, we simply return it.

Otherwise, we need to check whether $\exists\ p_L \in L$ and $p_R \in R$ s.t. distance between point P_L and $P_R < 1$. where L is the set of left half of points and R is the set of right half of points.

So, if $\delta = \min(\delta_L, \delta_R)$ where δ_L is the min. distance in left half and δ_R is the min. distance in right half, then we have $\delta \geq 1$ in the above case.

Thus, to have 2 such points p_L and p_R, we must have p_L and p_R atmax 1 units apart from the vertical line.

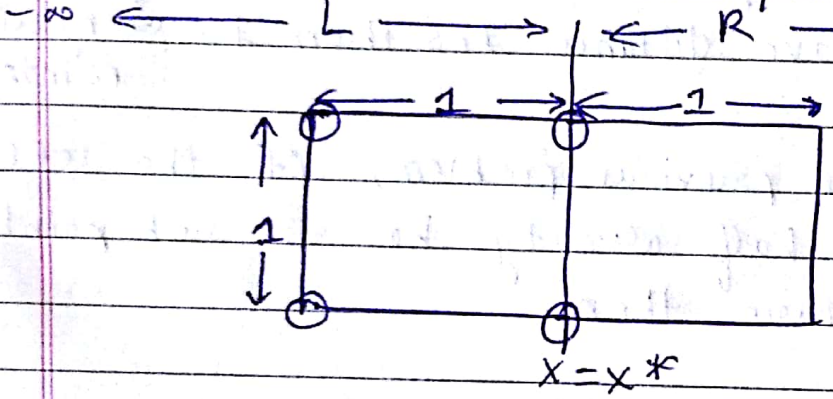


Thus, they must be in a stripe of width 2 centered on the boundary between the halves.

Also, since the vertical distance between them must also be less than 1, they must be in a 2x2 rectangle centered at boundary line. (height x width)

We can show that atmost 8 points can reside within this 2×1 rectangle.
(width × height)

Consider the 1×1 square forming the left half of this rectangle, since all points within L and R are atleast 1 units apart, atmost 4 points can reside within this square.



The points in R, similarly, atmost 4 can reside in 1×1 square.

So, in the 2 units wide strip, we can simply put the points in list in the sorted order of y-coordinates and check for each point, the next 7 points, if any pair has distance < 1, we simply return it.

Here, merging operation thus takes O(n) time and also since lists were sorted initially, to divide Lx and Ly into 2 halves is a constant time operation.

$$T(n) \leq 2T(n/2) + cn$$

$$\Rightarrow \boxed{T(n) = \Theta(n \log n)}$$

(b) We now need to check whether three points exist each having a distance < 1 between them.

In the base case of previous algorithm, we checked whether any pair out of 3 points had less than 1 distance. Now, we can check all $3C_2 = 3$ pairs and return true if all pairs have distance less than 1. (along with 3 points)

So, similar to previous problem, if the left half or right half already has 3 such points, we can return them.

Now, we need to make modification in the merge operation. So, the 2 points would be ~~one~~ in one of the half and the other point in the opposite half.

Thus, instead of just finding one point in the next 7 points above the current point in the list of points ~~over~~ in 2 units width around boundary line, we have to find 2 such points and also the distance between those 2 must also be less than 1.

So, we have to check $7C_2 = 21$ pairs with the current point ~~i.e. 7 points~~

If the condition holds true for any of the 3 points, simply return these 3 points.

Thus, again $T(n) \leq 2T(n/2) + cn$

$\Rightarrow T(n) = O(n \log n)$