CSE 6140 / CX 4140

Computational Science & Engineering (CSE) Algorithms

Homework 1

1. Suppose you have five algorithms that have the following running times. (Assume they are the exact running times). n denotes input size. n^2
n^3
$100n^2$
$nlogn$
2^n

Consider three different input sizes 10, 100, 1000. (a) (5 points) What is the running time of each algorithm for each input size? Use a calculator and provide your answers in scientific notation; i.e. write $10,000$ as 10^4.

Solution:

Input size (n)	n^2	n^3	$100n^2$	$nlogn$	2^n
10	1×10^2	1×10^3	1×10^4	1×10^1	1.024×10^3
100	1×10^4	1×10^6	1×10^6	2×10^2	1.268×10^{30}
1000	1×10^6	1×10^9	1×10^8	3×10^3	1.072×10^{301}

(b) (5 points) The running time of another algorithm is $n!$. Consider input sizes of 10, 50, and 100. What is the running time of the algorithm for each input size?

Solution:

Input size (n)	10	50	100
$n!$	3.6288×10^6	3.041×10^{64}	9.333×10^{157}

2. (10 points) Consider a for loop that runs for n steps. There's a function $g(.)$ inside that for loop. How would you describe the running time of the loop (in O-notation) if $g(.)$ completes in:
n steps
$logn$ steps
k steps, where k is a constant

Solution: Let us get the number of steps for general running time $T(n)$ for $g(.)$. The for loop runs n steps. In each step $g(.)$ routine is executed and this routine takes $T(n)$ steps. Hence, overall running time is $nT(n)$ steps.

$T(n)$	n	$logn$	k
$nT(n)$	$O(n^2)$	$O(nlogn)$	$O(kn)$

3. The worst-case running time of five algorithms are given below.
$O(n^{2.5})$
$O(\sqrt{2n})$
$O(n+10)$
$O(10^n)$
$O(100^n)$
a. (2 points) Which running time grows the fastest with increasing n?

Solution: The exponential functions grow the fastest. So, among the five functions, $O(100^n)$ grows the fastest.

b. (2 points) Which running time grows the slowest with increasing n?

Solution: For $x \geq 4$, we can see that $\sqrt{2x} \leq (x+10) \leq x^{2.5}$. Hence, by this analysis, $O(\sqrt{2n})$ grows the slowest.

c. (2 points) Arrange the algorithms in ascending order of running time.

Solution: From the above analysis, we get that $O(\sqrt{2n}) \leq O(n+10) \leq O(n^{2.5})$. Also, the exponential grows the fastest. For $x \geq 1$, $10^x \leq 100^x$, so we have $O(10^n) \leq O(100^n)$.
Also, for $x \geq 3$, $x^{2.5} \leq 10^x$. So, we get $n^{2.5} \in O(10^n)$.
Thus, we have:

$$O(\sqrt{2n}) \leq O(n+10) \leq O(n^{2.5}) \leq O(10^n) \leq O(100^n)$$

d. (2 points) In your ascending order, where would you place an algorithm that has worst-case running time of $O(n!)$?

Solution: First, we note that for all $x \geq 5$, $x^{2.5} \leq x!$. So, we have $O(n^{2.5}) \leq O(n!)$.
Similarly, we note that for all $x \geq 1$, $x! \leq 10^x$. So, we have $O(n!) \leq O(10^n)$.
Thus, we have

$$O(\sqrt{2n}) \leq O(n+10) \leq O(n^{2.5}) \leq \mathbf{O(n!)} \leq O(10^n) \leq O(100^n)$$

e. (2 points) Where would you place an algorithm that has worst-case running time of $O(n^2 logn)$ time?

Solution: Let us look at the following table and infer

From the above table, we can say that for all $x \geq 5$, $(x+10) \leq x^2 logx \leq x^{2.5}$.
Hence, we have

$$O(\sqrt{2n}) \leq O(n+10) \leq \mathbf{O(n^2 logn)} \leq O(n^{2.5}) \leq O(n!) \leq O(10^n) \leq O(100^n)$$

4. (10 points) Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$? Use the formal definition of O-notation to justify your answer?

Input size (n)	$(n+10)$	$n^2 logn$	$n^{2.5}$	$n!$
1	11	0	1	1
2	12	1.204	5.657	2
3	13	4.294	15.588	6
4	14	9.633	32	24
5	15	17.474	55.901	120
6	16	28.013	88.182	720
7	17	41.410	129.642	5040
8	18	57.798	181.020	40320
9	19	77.294	243	36280
10	20	100	316.228	362800

Solution: Let $f(x) = 2^{x+1}$ and $g(x) = 2^x$. So, we note that $f(x) = 2g(x)$, and both are strictly positive functions. We could write it in the required form as:

$$\exists c_0 = 2, \forall x \geq 1 \text{ s.t. } f(x) \leq c_0 g(x)$$

Thus, by the definition of O-notation, $f(n) = O(g(n))$ i.e., $2^{n+1} = O(2^n)$.

Let $f(x) = 2^{2x}$ and $g(x) = 2^x$. Let's assume $2^{2n} = O(2^n)$ for the purpose of contradiction. So, there exists positive real number c_0 and N_0 such that

$$\begin{aligned} & f(x) \leq c_0 g(x) \forall x \geq N_0 \\ & \Rightarrow 2^{2x} \leq c_0 2^x \forall x \geq N_0 \\ & \Rightarrow 2^x \leq c_0 \forall x \geq N_0 \qquad \text{(Dividing both sides by } 2^x) \end{aligned}$$

This is a contradiction since 2^x is unbounded. Thus, we conclude that $2^{2n} \neq O(2^n)$.

5. (10 points) The Stable Matching Problem, as discussed in class, assumes that all men and women have a fully ordered list of preferences. In this problem we will consider a version of the problem in which men and women can be indifferent between certain options. As before we have a set M of n men and a set W of n women. Assume each man and each woman ranks the members of the opposite gender, but now we allow ties in the ranking. For example (with $n = 4$), a woman could say that m_1 is ranked in first place; second place is a tie between m_2 and m_3 (she has no preference between them); and m_4 is in last place. We will say that w prefers m to m' if m is ranked higher than m' on her preference list (they are not tied).

With indifferences in the rankings, there could be two natural notions for stability. And for each, we can ask about the existence of stable matchings, as follows.

(a) A **strong instability** in a perfect matching S consists of a man m and a woman w, such that each of m and w prefers the other to their partner in S. Does there always exist a perfect matching with no strong instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a strong instability; or give an algorithm that is guaranteed to find a perfect matching with no strong instability.

(b) A **weak instability** in a perfect matching S consists of a man m and a woman w, such that their partners in S are w' and m', respectively, and one of the following holds:

- m prefers w to w', and w either prefers m to m' or is indifferent between these two choices; or
- w prefers m to m', and m either prefers w to w' or is indifferent between these two choices

In other words, the pairing between m and w is either preferred by both, or preferred by one while the other is indifferent. Does there always exist a perfect matching with no weak instability? Either give an example of a set of men and women with preference lists for which every perfect matching has a weak instability; or give an algorithm that is guaranteed to find a perfect matching with no weak instability.

Solution:

(a) We modify the Gale-Shapley algorithm as follows to get a perfect matching with no strong instability:

1. If a man m is has multiple choices for his next proposal (many women are tied in the highest rank), then he breaks the tie arbitrarily and proposes to one of them at random.
2. If a woman w is already engaged to m' and then receives a proposal from m, and she is indifferent between m and m', then she chooses to stay engaged with m'.

Now, we prove that this algorithm ensures a perfect matching with no strong instability. Let M be the perfect matching returned by the algorithm.

Let's assume for the sake of contradiction that there is a strong instability in M (i.e.,) there exists m and w with partners w' and m' in M respectively and they prefer each other over their current partners.

So, we have $\{(m, w'), (m', w)\} \subseteq M$, with $m > m'$ for w and $w > w'$ for m. Since m prefers w he would have proposed to her first. They didn't get engaged because of either one of these:

- w has a better partner m^*, but eventually got engaged to m'. Here, preference is: $m' \geq m^* > \cdots > m$ (here we write $m' \geq m^*$ since m^* could have been m' in which case preferences match). However, this violates that w prefers m over m' $\Rightarrow$ **Contradiction**.
- w accepts m but eventually ends up with m'. As per the algorithm, this is possible only when w prefers m' over m, which clearly contradicts that

$m > m'$ for $w \Rightarrow$ **Contradiction**.

So, our initial assumption that M had a strong instability was wrong. **Thus, the modified algorithm ensures a perfect matching with no strong instability.**

(b) Consider the following example with $M = \{m_1, m_2\}$ and $W = \{w_1, w_2\}$ and the preferences are as follows:

$$m_1 : w_1 = w_2 \text{ and } w_1 : m_1 > m_2$$

$$m_2 : w_1 = w_2 \text{ and } w_2 : m_1 > m_2$$

Now consider each of the perfect matching:

1. $\{(m_1, w_1), (m_2, w_2)\}$: Here w_2 prefers m_1 over her current partner but m_1 is indifferent, thus forming a **weak instability**
2. $\{(m_1, w_2), (m_2, w_1)\}$: Here w_1 prefers m_1 over her current partner but m_1 is indifferent, thus forming a **weak instability**

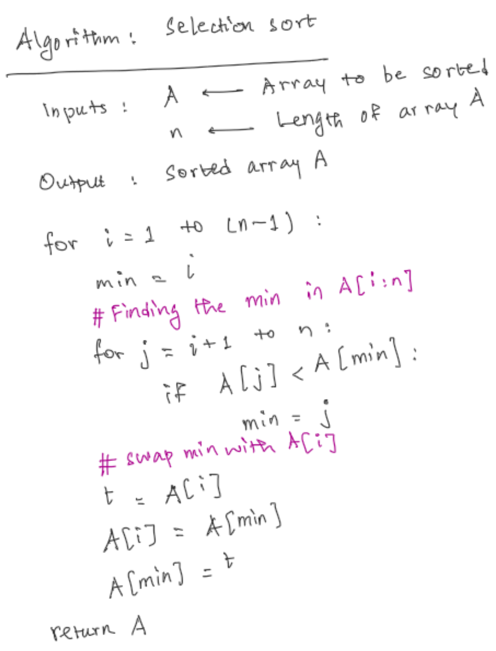
Thus, every perfect matching results in a weak instability. So, there can be cases where no perfect matching without weak stability exists.

6. (20 points) Consider sorting n numbers stored in array $A[1 : n]$ by first finding the smallest element of $A[1 : n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2 : n]$ and exchange it with $A[2]$. Then find the smallest element of $A[3 : n]$, and exchange it with $A[3]$. Continue in this manner for the first $(n - 1)$ elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. Why does it need to run for only the first $(n - 1)$ elements, rather than for all n elements? Give the worst-case running time of selection sort in O-notation. Is the best-case running time any better?

Solution: The pseudocode is in Figure 1.

From the pseudocode we note that the outer loop considers the sub-array $A[i : n]$, finds the smallest element in this subarray, and swaps it with $A[i]$. So, as i increases, we have $A[1 : i]$ to be sorted. If that wasn't the case there would have been $m < n < i$ with $A[m] > A[n]$. However, this would have been resolved in $i = m$ iteration and so we would have the subarray $A[1 : i]$ to be sorted. So, after we are done with $i = n - 1$, we have the whole array sorted, else the last element is smaller than one of the previous ones. However, this is not possible since we find the minimum element in the subarray at each outer loop execution. Thus, after $i = (n - 1)$, sorting ends.

For time complexity, we see that the outer loop runs for $(n - 1)$ steps and the inner loop runs for $(i + 3)$ steps (i steps for the loop and 3 steps for the swap).



```
Algorithm : Selection sort
Inputs :  A <-- Array to be sorted
          n <-- Length of array A
Output :  Sorted array A

for i = 1 to (n-1) :
    min = i
    # Finding the min in A[i:n]
    for j = i+1 to n :
        if A[j] < A[min] :
            min = j
    # swap min with A[i]
    t = A[i]
    A[i] = A[min]
    A[min] = t
return A
```

Figure 1: Selection sort pseudocode

So, the total steps in our algorithm is:

$$T(n) = \sum_{i=1}^{(n-1)} (i+3) = (1+2+\cdots+(n-1)) + 3(n-1) = \frac{(n-1)(n-2)}{2} + 3n - 3$$

$$\Rightarrow T(n) = \frac{n^2-3n+2}{2} + 3n - 3 = \frac{(n^2+3n-4)}{2} = O(n^2)$$

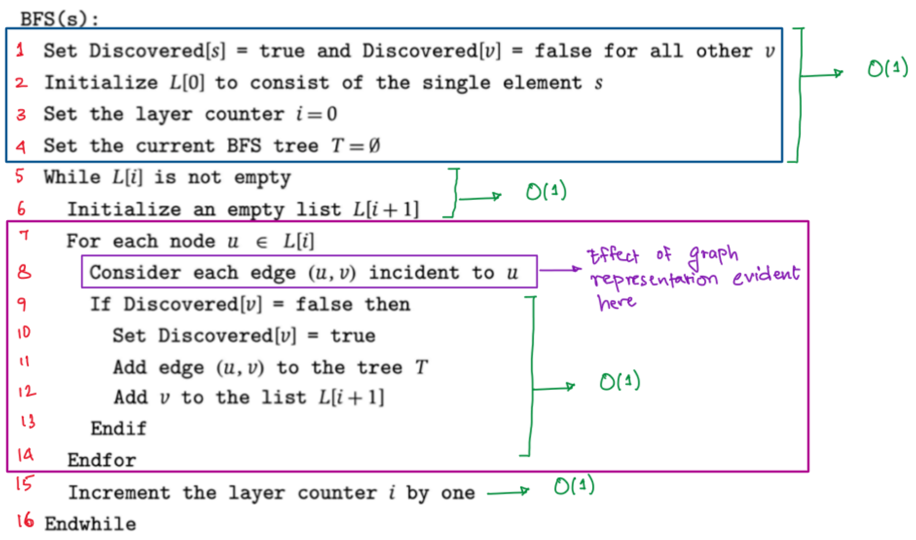
Note that, even in the best case when array A is already sorted, the number of comparisons remains the same and the swap steps are also executed. So, $T(n)$ does not change for the best case either. So, we have same time-complexity of $O(n^2)$ for the best case as well.

7. (10 points) What is the running time of breadth-first search if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

Solution:

We look at the time complexity step by step as follows:

1. Lines 1-4 are just initializations and thus take constant time ($O(1)$)



```
BFS(s):
1  Set Discovered[$s$] = true and Discovered[$v$] = false for all other $v$
2  Initialize $L[0]$ to consist of the single element $s$
3  Set the layer counter $i=0$
4  Set the current BFS tree $T=\emptyset$
5  While $L[i]$ is not empty
6    Initialize an empty list $L[i+1]$
7    For each node $u \in L[i]$
8      Consider each edge $(u,v)$ incident to $u$
9      If Discovered[$v$] = false then
10       Set Discovered[$v$] = true
11       Add edge $(u,v)$ to the tree $T$
12       Add $v$ to the list $L[i+1]$
13     Endif
14   Endfor
15   Increment the layer counter $i$ by one
16 Endwhile
```

Figure 2: Pseudocode for BFS. Note that the line 8 is where the effect of graph representation is evident

2. Lines 5 and 6 are constant time operations ($O(1)$)

3. Line 7 (outer loop) visits every node in the layer once. So, at the end of one run of the algorithm we would have visited all the nodes in the largest connected component. In the worst case, when we have a connected graph we have $O(|V|)$ visits.

4. Line 8 is the key line where the graph representation matters. In line 8, we consider every edge of node u. However, in adjacency matrix to visit all edges of a vertex, we have to go through the entire row or column corresponding to that node. Hence, for each node u, lines 8-14 which forms the inner loop runs for $O(|V|)$ times

5. Line 15 is just a constant time operation ($O(1)$)

Thus, we see that the lines 7-14 are the ones that take majority of the time and they take $O(|V|).O(|V|)$ steps. So, due to adjacency matrix representation, the algorithm's time complexity becomes $O(|V|^2)$ where V is the vertex set of the graph G.

8. (20 points) Consider a two-dimensional piece of land consisting of a number of towns. Consider a virus spreading across these towns. We are given an $m \times n$ array that represents this land. Each element in the array has one of three values:

0 represents empty land

1 represents an uninfected town

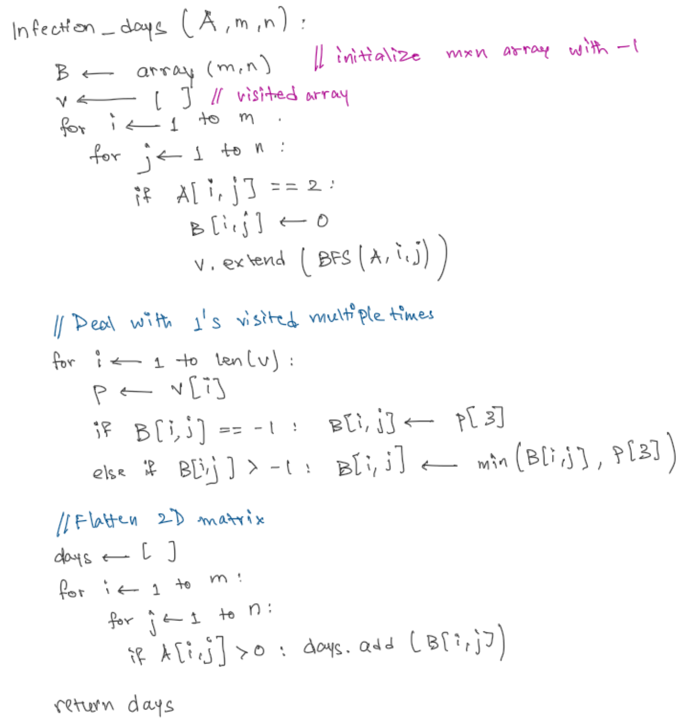
2 represents a town infected by the virus

Every day, an uninfected town that is adjacent to an infected town becomes infected. Here, adjacent means left, right, up or down (not diagonal).

Provide pseudocode of an algorithm that finds the number of days until each town becomes infected.

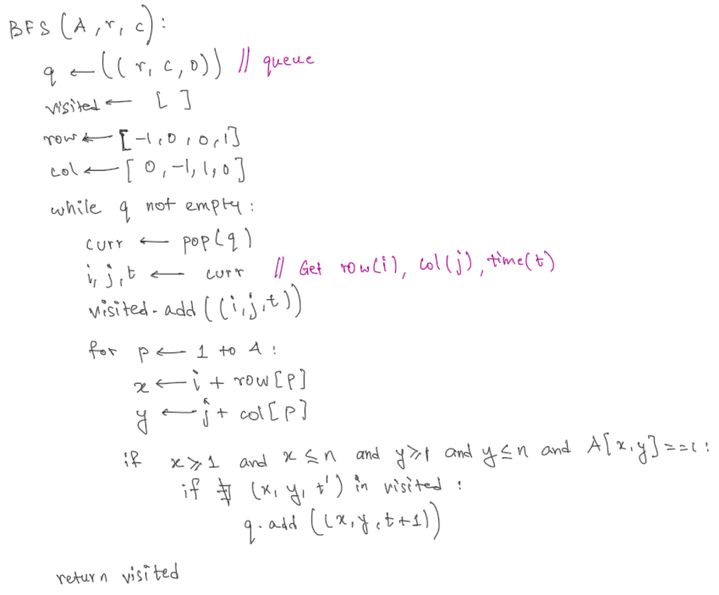
The algorithm should return an array *days* where *days*[i] is the number of days until town i becomes infected. If town i never gets infected, i should equal -1.

Solution: We approach this using **BFS Traversal of 2D matrix**. We start a BFS at every infected town and the days after which an uninfected town is infected is assigned as the layer in which it was discovered in the BFS. If it is revisited, then we update based on the shorted duration. The pseudocode is as follows (Refer to Figures 3 and 4):



```
Infection_days (A, m, n) :

    B ← array (m, n)    // initialize mxn array with -1
    v ← [ ] // visited array
    for i ← 1 to m .
        for j ← 1 to n :
            if A[i, j] == 2 :
                B[i, j] ← 0
                v.extend ( BFS (A, i, j) )

    // Deal with 1's visited multiple times

    for i ← 1 to len(v) :
        p ← v[i]
        if B[i, j] == -1 :  B[i, j] ← p[3]
        else if B[i, j] > -1 :  B[i, j] ← min (B[i, j], p[3])

    // Flatten 2D matrix
    days ← [ ]
    for i ← 1 to m :
        for j ← 1 to n:
            if A[i, j] > 0 : days.add (B[i, j])

    return days
```

Figure 3: The pseudocode for the algorithm that gives the *days* array. Here, we use 1-indexing for all arrays



```
BFS(A, r, c):
    q ← ((r, c, 0))  // queue
    visited ← [ ]
    row ← [-1, 0, 0, 1]
    col ← [0, -1, 1, 0]
    while q not empty:
        curr ← pop(q)
        i, j, t ← curr  // Get row(i), col(j), time(t)
        visited.add((i, j, t))
        for p ← 1 to 4:
            x ← i + row[p]
            y ← j + col[p]
            if x ≥ 1 and x ≤ n and y ≥ 1 and y ≤ n and A[x, y] == 1:
                if ∄ (x, y, t') in visited:
                    q.add((x, y, t+1))
    return visited
```

Figure 4: The pseudocode for the BFS function used in the $Infection_days$