HW1 Solutions

2. O(n2), O(n log n), O(n)

3a. 100n grows the fastest
3b. sqrt(2n) grows slowest
3c. sqrt(2n), n+10, n2.5, 10n, 100n 3d. n! goes at the end
3e. Between n+10 and n2.5

4.

$2^{n+1} = O(2^n)$, but $2^{2n} \neq O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$0 \leq 2^{n+1} \leq c \cdot 2^n$ for all $n \geq n_0$.

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$0 \leq 2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$.

Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

5.

(a) The answer is Yes. A simple way to think about it is to break the ties in some fashion and then run the stable matching algorithm on the resulting preference lists. We can for example break the ties lexicographically — that is if a man m is indifferent between two women w_i and w_j then w_i appears on m's preference list before w_j if $i < j$ and if $j < i$ w_j appears before w_i. Similarly if w is indifferent between two men m_i and m_j then m_i appears on w's preference list before m_j if $i < j$ and if $j < i$ m_j appears before m_i.

Now that we have concrete preference lists, we run the stable matching algorithm. We claim that the matching produced would have no strong instability. But this latter claim is true because any strong instability would be an instability for the match produced by the algorithm, yet we know that the algorithm produced a stable matching — a matching with no instabilities.

(b) The answer is No. The following is a simple counterexample. Let $n = 2$ and m_1, m_2 be the two men, and w_1, w_2 the two women. Let m_1 be indifferent between w_1 and w_2 and let both of the women prefer m_1 to m_2. The choices of m_2 are insignificant. There is no matching without weak stability in this example, since regardless of who was matched with m_1, the other woman together with m_1 would form a weak instability.

6.

```
SELECTION-SORT(A, n)
 for i = 1 to n − 1
     smallest = i
     for j = i + 1 to n
         if A[j] < A[smallest]
             smallest = j
     exchange A[i] with A[smallest]
```

The algorithm maintains the loop invariant that at the start of each iteration of the outer **for** loop, the subarray $A[1 : i - 1]$ consists of the $i - 1$ smallest elements in the array $A[1 : n]$, and this subarray is in sorted order. After the first $n - 1$ elements, the subarray $A[1 : n - 1]$ contains the smallest $n - 1$ elements, sorted, and therefore element $A[n]$ must be the largest element.

The running time of the algorithm is $\Theta(n^2)$ for all cases.

\

7.

BFS runs in $O(V^2)$ time with an adjacency-matrix representation of a graph because there are $O(V^2)$ potential edges to check.

8.

Use breadth-first search with multiple sources. Source nodes are towns that are initially infected.

Steps

Initialization:

- Initialize a queue *q*
- Initialize output array *days*: *days[i]* will contain the day town *i* gets infected
- Initialize dictionary *town_ids*: keys are 2D locations, values are town ids;
- Call the initial *m x n* array *A*.
- Scan *A*; i.e. iterate through all the rows and columns
  - * If a (row_id, col_id) is a town:
    - * town_ids[(row_id, col_id)] = id;
    - * If town is infected:
      - * days[id] = 0
      - * Insert (row_id, col_id) into q
    - * Else: days[id] = -1;
- Insert (-1, -1) into q. This will indicate the elapse of a day
- Initialize *days_elapsed = 0*

Runtime:

- * Pop from queue:
- * Every element popped is an infected town *T* or (-1, -1)
- * If popped item is *T*:
  - * Look up the adjacent elements of *T* (from array *A)*
    - * Adjacent elements of (row_id, col_id) are:
      - * Left: (row_id, col_id-1)
      - * Right: (row_id, col_id+1)
      - * Up: (row_id+1, col_id)
      - * Down: (row_id-1, col_id),
  - * If adjacent elements are uninfected towns:
    - * Set them as infected in array A
    - * Insert them into q
    - * id = *town_ids[T]*
    - * *days[id] = days_elapsed*
- * If popped item is (-1, -1):
  - * *days_elapsed = days_elapsed + 1*
  - * If *q* is empty, there are no more infected towns.
    - * Return *days_elapsed*
    - * You can also return *town_ids* in addition to *days_elapsed*
  - * If *q* is not empty, insert (-1, -1) into *q*.