Solutions to Homework Four

CSE 101

1. Algorithm A has running time
$$T_A(n) = 3T_A(n/3) + O(n),$$
to which we can apply the general recurrence formula, giving $T_A(n) = O(n \log n)$.

   Algorithm B has running time
$$T_B(n) = T_B(n-1) + O(n).$$
To solve this one, let's expand it out:
$$\begin{aligned} T_B(n) &= T_B(n-1) + n \\ &= T_B(n-2) + (n-1) + n \\ &= T_B(n-3) + (n-2) + (n-1) + n \end{aligned}$$
and so on. Eventually we get $T_B(n) = T_B(1) + (2 + 3 + 4 + \cdots + n) = O(n^2)$.

   Algorithm C has running time
$$T_C(n) = 2T_C(n/3) + O(n^2),$$
for which the general recurrence formula gives $T_C(n) = O(n^2)$.

   Algorithm D has running time
$$T_D(n) = 5T_D(n/4) + O(n),$$
which, by the general recurrence formula, comes out to $T_D(n) = n^{\log_4 5}$.

   Of the four, Algorithm A is the quickest.

2. Let $L(n)$ be the number of lines. Then $L(n) = 3L(n/2) + 1$ so that $L(n) = O(n^{\log_2 3})$.

3. *Textbook problem 2.24(a,b).*

   (a) Here's the `quicksort` procedure.

   <u>`function quicksort`</u>$(S[1\cdots n])$
   Input: `array of numbers`
   Output: `sorted array`

   `If` $n \leq 1$: `return` S
   `Pick` v `at random from` S
   `Split` S `into three pieces:`
   $S_L =$ `elements less than` v
   $S_v =$ `elements equal to` v
   $S_R =$ `elements greater than` v
   `Return quicksort`$(S_L) \circ S_v \circ$ `quicksort`(S_R)

   (b) Each iteration through the recursive procedure takes linear time, but the number of recursive calls varies according to the particular split elements chosen.

   A particularly unlucky scenario is when the elements of S are distinct, and the split element is always the largest element of S. Then S_R is always empty, S_v contains a single element, and S_L has everything else. We thus get a running time
$$T(n) = T(n-1) + O(n),$$
which works out to $O(n^2)$.

4. *Randomized binary search.*

(a) Here's the algorithm, given an array $S[1 \ldots n]$ and a number x.

```
Let ℓ = 1, r = n // current search interval is [ℓ, r]
While r ≥ l:
   Pick p at random from {ℓ, ℓ+1, ..., r}
   If S[p] = x:  halt and output ``yes''
   If S[p] > x:  let r = p − 1
   If S[p] < x:  let ℓ = p + 1
Output ``no''
```

(b) Let $T(n)$ denote the expected running time on an array of size n. On any given iteration, a constant amount of work is done, and there is a 1/2 probability that the randomly chosen position p lies in the central half of the search interval $[\ell, r]$. If this happens, the search interval shrinks to at most 3/4 its size on that iteration. If not, then at the very worst the interval doesn't shrink at all. We can thus write

$$T(n) \quad \leq \quad \frac{1}{2} T\left(\frac{3n}{4}\right) + \frac{1}{2} T(n) + O(1)$$

which means $T(n) \leq T((3/4)n) + O(1)$ and thus $T(n) = O(\log n)$.

5. *Textbook problem 2.23.*

(a) *Solving the problem in $O(n \log n)$ time.*

Suppose we divide array A into two halves, A_L and A_R. Then:

$$\begin{aligned} A \text{ has a majority element } x \quad &\Longleftrightarrow \quad x \text{ appears more than } n/2 \text{ times in } A \\ &\Longrightarrow \quad x \text{ appears more than } n/4 \text{ times in either } A_L \text{ or } A_R \text{ (or both)} \\ &\Longleftrightarrow \quad x \text{ is a majority element of either } A_L \text{ or } A_R \text{ (or both)} \end{aligned}$$

This suggests a divide-and-conquer algorithm:

```
function majority (A[1...n])
if n = 1:  return A[1]
let A_L, A_R be the first and second halves of A
M_L = majority(A_L) and M_R = majority(A_R)
if M_L is a majority element of A:
   return M_L
if M_R is a majority element of A:
   return M_R
return ``no majority''
```

Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

(b) *A linear-time algorithm.*

```
function majority (A[1...n])
x = prune(A)
if x is a majority element of A:
   return x
else:
   return ``no majority''

function prune (S[1...n])
if n = 1:  return S[1]
if n is odd:
   if S[n] is a majority element of S:  return S[n]
   n = n − 1
```

```
S' = [ ] (empty list)
for i = 1 to n/2:
   if S[2i − 1] = S[2i]:  add S[2i] to S'
return prune(S')
```

Justification: We'll show that each iteration of the **prune** procedure maintains the following invariant: if x is a majority element of S then it is also a majority element of S'. The rest then follows.

Suppose x is a majority element of S. In an iteration of **prune**, we break S into pairs. Suppose there are k pairs of Type One and l pairs of Type Two:

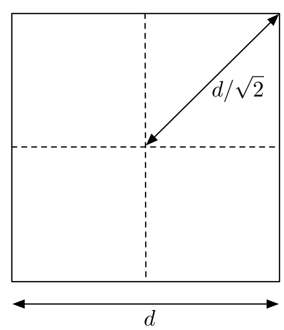
- Type One: the two elements are different. In this case, we discard both.
- Type Two: the elements are the same. In this case, we keep one of them.

Since x constitutes at most half of the elements in the Type One pairs, x must be a majority element in the Type Two pairs. At the end of the iteration, what remains are l elements, one from each Type Two pair. Therefore x is the majority of these elements.

Running time. In each iteration of **prune**, the number of elements in S is reduced to $l \leq |S|/2$, and a linear amount of work is done. Therefore, the total time taken is $T(n) \leq T(n/2) + O(n) = O(n)$.

6. *Closest pair.*

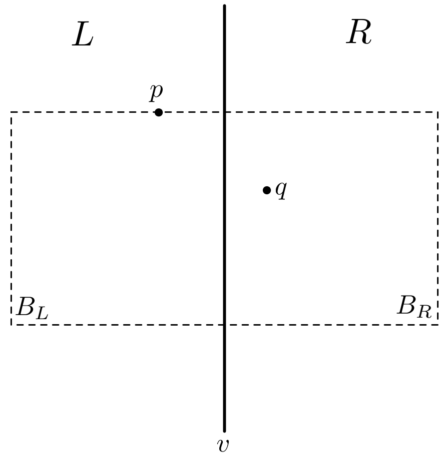
(a) We know that any two points in L are at distance $\geq d$ from each other. Now consider any $d \times d$ square, and divide it into four smaller squares of side length $d/2$:



Each smaller square can contain at most one point of L, since any two points in the smaller square are at distance $\leq d/\sqrt{2}$. Therefore the $d \times d$ square contains at most four points of L.

(b) To show correctness of the algorithm, we only need to show that if the closest pair p, q has $p \in L$ and $q \in R$, then this pair will be found.

So assume this is the case. By construction, the distance between $p = (x_p, y_p)$ and $q = (x_q, y_q)$ is less than d. Suppose $y_p < y_q$ (the other case is symmetric). Then the configuration is as shown below:



In the picture, B_L and B_R are squares of side-length d whose top-right and top-left corners, respectively, are at the point (v, y_p). Since p, q are within distance d of each other, point q must lie within B_R. We know from part (a) that B_L and B_R each contain at most four points. In short, y_q must be one of the 7 y-values closest to y_p.

(c) The steps involved in solving the problem on n points are:

- Finding the median x value: $O(n)$.
- Recursing on two subproblems of size $n/2$: this is $2T(n/2)$.
- Discarding some points: $O(n)$.
- Sorting y values: $O(n \log n)$.
- Iterating through a list of y values and doing seven computations for each: $O(n)$.

This gives $T(n) = 2T(n/2) + O(n \log n)$.