# Compsci 330 Design and Analysis of Algorithms
# Assignment 2, Spring 2024 Duke University

Example Solution

Due Date: Thursday, February 1, 2024

**How to Do Homework.** We recommend the following three step process for homework to help you learn and prepare for exams.

1. Give yourself 15-20 minutes per problem to try to solve on your own, without help or external materials, as if you were taking an exam. Try to brainstorm and sketch the algorithm for applied problems. Don't try to type anything yet.
2. After a break, review your answers. Lookup resources or get help (from peers, office hours, Ed discussion, etc.) about problems you weren't sure about.
3. Rework the problems, fill in the details, and typeset your final solutions.

**Typesetting and Submission.** Your solutions should be typed and submitted as a single pdf on Gradescope. Handwritten solutions or pdf files that cannot be opened will not be graded. LaTeX[1] is preferred but not required. You must mark the locations of your solutions to individual problems on Gradescope as explained in the documentation. Any applied problems will request that you submit code separately on Gradescope to be autograded.

**Writing Expectations.** If you are asked to provide an algorithm, you should clearly and unambiguously define every step of the procedure as a combination of precise sentences in plain English or pseudocode. If you are asked to explain your algorithm, its runtime complexity, or argue for its correctness, your written answers should be clear, concise, and should show your work. Do not skip details but do not write paragraphs where a sentence suffices.

**Collaboration and Internet.** If you wish, you can work with a single partner (that is, in groups of 2), in which case you should submit a single solution as a group on gradescope. You can use the internet, but looking up solutions or using large language models is unlikely to help you prepare for exams. See the course policies webpage for more details.

**Grading.** Theory problems will be graded by TAs on an S/U scale (for each sub-problem). Applied problems typically have a separate autograder where you can see your score. The lowest scoring problem is dropped. See the course assignments webpage for more details.

[1] If you are new to LaTeX, you can download it for free at latex-project.org or you can use the popular and free (for a personal account) cloud-editor overleaf.com. We also recommend overleaf.com/learn for tutorials and reference.

**Problem 1 (Binary Search Trees).** You are given two binary search trees $A$ and $B$, each storing a set of $n$ integer elements. Each node in the trees contains the integer element and a left and right child node reference; a NULL reference indicates there is no child.

Describe an $O(n)$ runtime algorithm that returns a binary search tree with height $O(\log(n))$ containing the union of the elements in $A$ and $B$. Briefly explain the runtime complexity of your algorithm. Prove the correctness of the algorithm.

*Hint.* Consider breaking the algorithm down into three steps: (i) traverse $A$ and $B$ to get their elements in sorted order, (ii) merge the elements, and (iii) construct a balanced binary search tree of the merged elements. If you use the merge procedure from mergesort, you can assume its correctness without proof.

**Solution 1.** The algorithm proceeds in the following steps:

(i) First, we do in-order traversals of $S$ and $R$, using the recursive procedure InOrder described below. We call InOrder($root_1$, $L_1$) and InOrder($root_2$, $L_2$) where $root_1$ and $root_2$ are roots of $S$ and $R$, respectively, and return the keys in two sorted lists $L_1$ and $L_2$ respectively.

(ii) Next, we apply the merge procedure from the MergeSort algorithm to merge $L_1$ and $L_2$ into a single sorted list $L$, but skipping over any duplicates.

(iii) Finally, we recursively build a balanced binary search tree from $L$ using the procedure BuildBST($L$, Length($L$)).

We provide pseudocode for the first and third parts – see the text or lecture for pseudocode for the merge procedure.

```
procedure InOrder($u$, $L$)
    if $u$ is not null then
        InOrder($u$.left, $L$)
        $L$.add($u$.key)
        InOrder($u$.right, $L$)
```

```
procedure BuildBST($L$, $\ell$)
    if $L$ is empty then
        return NULL
    else
        $T \leftarrow$ new Node($L[\lceil \frac{\ell}{2} \rceil]$)
        $T$.left $\leftarrow$ BuildBST($L[1, \ldots, \lceil \frac{\ell}{2} \rceil - 1], \lceil \frac{\ell}{2} \rceil - 1$)
        $T$.right $\leftarrow$ BuildBST($L[\lceil \frac{\ell}{2} \rceil + 1, \ldots, \ell], \ell - \lceil \frac{\ell}{2} \rceil$)
        return $T$
```

*Runtime Analysis.* In-order traversal on a tree of size $n$ takes $O(n)$ time since each element is visited exactly once. The standard merge procedure of mergesort is also $O(n)$. Finally, let $T(n)$ be the runtime of BuildBST on a list of size $n$. Then, we have the following recurrence relation (ignoring the ceiling):

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

whose solution is $T(n) = O(n)$. Therefore, the total running time is $O(n)$.

*Correctness.* Here, we argue for two properties: (i) that the tree returned by BuildBST$(L, n)$ is a binary search tree (BST) and (ii) that the tree returned by BuildBST$(L, n)$ has height $O(\log(n))$. We prove these properties of BuildBST$(L, n)$ by induction.

- Base Case: where $n = 1$, where $n$ is the number of elements in $L$, BuildBST$(L, n)$ returns a tree with a single node, which is a BST and has height 1.

- Inductive hypothesis (IH): suppose that $\textsc{BuildBST}(L, \ell)$ returns a BST of height at most $\log_2(\ell) + 1$ for lists with $\ell$ elements for any list $L$ of length $\ell$ and $\ell \leq n$.
- Inductive Step: Consider a list with $n + 1$ elements. The algorithm selects the median as the root. The elements in the first recursive call on line 6 are less than the median, and these form the left subtree, which is a BST by the IH. The elements in the second recursive call on line 7 are greater than the median, and these form the right subtree, which is a BST by the IH. So the resulting tree is a binary search tree. To see the second point, note that the left and right subtrees have height at most $\log_2(n/2) + 1 = \log_2(n) - \log_2(2) + 1 = \log_2(n)$ by the IH, so connecting them to a root node one level up results in a tree of height at most $\log_2(n) + 1$.

**Problem 2 (Median).** You are given an array $A$ of $n$ unique integers, not necessarily sorted.

Describe an efficient randomized algorithm to compute the median value of the elements of $A$. By efficient, the algorithm should have an *expected* runtime that is $O(n)$. Analyze the expected runtime complexity of your algorithm. Prove the correctness of the algorithm.

*Hint.* Remember the median is the element that would be "in the middle" of an array if it were sorted (the element itself or the average of two if $n$ is even). Think about adapting the quicksort algorithm. Solving a recurrence relation might *not* be the easiest way to analyze the runtime. If you use the same PARTITION procedure from lecture/book, you can assume its correctness.

**Solution 2.** We adapt the quicksort algorithm. Recall that the partition procedure chooses a pivot and puts all smaller elements to the left (smaller indices) and larger elements to the right (larger indices) in the array. We will do the same, but then *only recurse on the side that could hold the median.* For example, if on the first split 60% of the elements are on the left side, we would recurse there. The randomness will come from our random choice of the pivot.

The outer QUICKMEDIAN procedure checks the two cases of parity of input: if there are an odd number of elements then the median is the index $(n-1)/2$ in sorted order, but if $n$ is even then it is the average of the $n/2-1$ and $n/2$ index elements (in sorted order). The QUICKHELP procedure is the divide and conquer algorithm that actually solves the more general problem of searching for the $k$'th smallest (0-indexed, with $k=0$ implying minimum) element of $A[l], A[l+1], \dots, A[r]$. The PARTITION procedure is the same as in regular quicksort but is provided for completeness.

```
1: procedure QUICKMEDIAN($A, n$)
2:     if $n$ is odd then
3:         return QUICKHELP($A, 0, n-1, (n-1)/2$)
4:     else
5:         return $\frac{1}{2}$(QUICKHELP($A, 0, n-1, n/2-1$) + (QUICKHELP($A, 0, n-1, n/2$)))
```

```
1: procedure QUICKHELP($A, l, r, k$)
2:     $p$ = PARTITION($A, l, r$)
3:     if $l+k<p$ then
4:         return QUICKHELP($A, l, p-1, k$)
5:     else if $l+k>p$ then
6:         $k' = k-(p-l+1)$
7:         return QUICKHELP($A, p+1, r, k'$)
8:     else
9:         return $A[p]$
```

```
1: procedure PARTITION($A, l, r$)
2:     $d \sim \{l, l+1, \dots, r\}$ uniformly at random
3:     Swap $A[d]$ and $A[r]$
4:     $p = l$
5:     for $i = l$ to $r-1$ do
6:         if $A[i] < A[r]$ then
7:             Swap $A[i]$ and $A[p]$
8:             $p = p+1$
9:     Swap $A[p]$ and $A[r]$
10:    return $p$
```

*Runtime Analysis.* The PARTITION procedure has linear runtime in the number of elements dominated by the for loop on line 5. The QUICKHELP procedure then makes at most one recursive call to a subproblem whose size is an integer chosen uniformly at random from 1 to $n-1$ based on the random choice of a pivot.

Let $n_0, n_1, \dots$ denote random variables equal to the number of elements in the range from $l$ to $r$ at each level of the recursion. Let $T(n)$ denote the runtime of QUICKHELP on an input of size

$n = r - l + 1$, also a random variable. The linear runtime of PARTITION implies that $T(n) \propto n_0 + n_1 + \cdots + \ldots$. So the expected runtime can be bound as

$$\begin{aligned}\mathbb{E}[T(n)] &= \mathbb{E}\left[c\,(n_0 + n_1 + \ldots)\right] \\ &= c\,(\mathbb{E}[n_0] + \mathbb{E}[n_1] + \ldots)\end{aligned}$$

for some constant $c$, by the linearity of expectation. Note that by choice of the random pivot, $\mathbb{E}[n_k] \leq n(1/2)^k$. Summing:

$$\begin{aligned}\mathbb{E}[T(n)] &\leq c\,(n + n/2 + n/4 + n/8 + ... + 1) \\ &= c \cdot n \left(\sum_{i=0}^{\log_2(n)} \frac{1}{2^i}\right) \text{ is } O(n)\end{aligned}$$

where the last line follows from the geometric series formula.

*Correctness.* We prove that for $k \leq r - l$, QUICKHELP$(A, l, r, k)$ returns the $k$'th smallest value (0-indexing where $k = 0$ implies the minimum) from $A[l], A[l{+}1], \ldots, A[r]$. Correctness of QUICKMEDIAN$(A, n)$ follows from this claim and the definition of median.

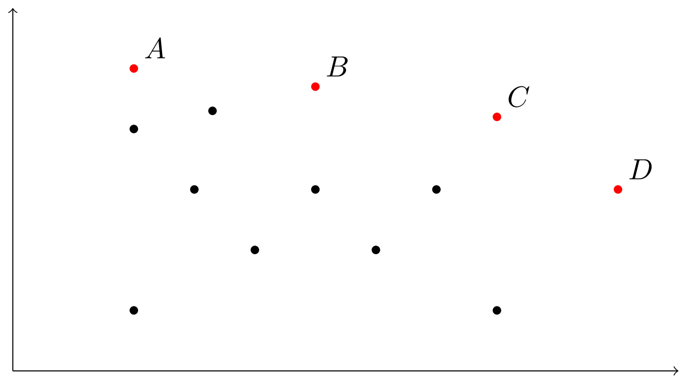
We argue by induction on $n = r - l + 1$, the size of the range considered in a subproblem. In the base case $r = l$ and $k = 0$. Then $p = l = r$ follows from the PARTITION procedure. So $l + k = l = p$ and $A[p]$ is returned correctly.

For the inductive hypothesis (IH), assume QUICKHELP$(A, l, r, k)$ returns the $k$'th smallest value from $A[l], A[l + 1], \ldots, A[r]$ for $r - l + 1 \leq n - 1$. Consider an arbitrary subproblem with $r - l + 1 = n$. The PARTITION procedure guarantees that after line 2, we have the following property: $A[i] < A[j]$ for all $i < p$ an $j \geq p$. Call this the partition property.

Consider three cases.

- If $l + k < p$ then the $k$'th smallest element cannot be at an index greater than or equal to $p$ by the partition property. It must be at an index less than $p$, which the recursive call on the smaller range correctly returns by the IH.
- Similarly, if $l + k > p$ then the $k$'th smallest element cannot be at an index less than or to $p$ by the partition property. It must instead be the $k - (p - l + 1)$ smallest element among $A[p + 1], \ldots, A[r]$, noting that $p - l + 1$ is simply the number of elements between $l$ and $p$ inclusive. The correct index is returned by the recursive call on a smaller range by the IH.
- Otherwise, $l + k = p$, in which case $A[p]$ is the $k$'th smallest by the partition property. The algorithm returns the value.

**3. Maximal Points.** You are given an array $P = [(x_0, y_0), \dots, (x_{n-1}, y_{n-1})]$ of $n$ unique points in the two-dimensional Euclidean plane. A point $(x_i, y_i)$ *dominates* another point $(x_j, y_j)$ if $x_i > x_j$ and $y_i \geq y_j$ or $x_i \geq x_j$ and $y_i > y_j$ (that is, if it is up and to the right). A point is *maximal* if it is not dominated by any other point. See below for an illustration where the red points $A, B, C, D$ are all maximal points in the diagram.



Describe an efficient algorithm to compute all maximal points in $P$. Derive and solve a recurrence relation to analyze the runtime complexity of the algorithm. Prove the correctness of the algorithm.

*Hint.* The input is not sorted, though you could sort it in $O(n\log(n))$ time using, for example, mergesort, if that would be helpful for your algorithm (you do not need to repeat how mergesort works if you use it for this problem and you can assume its correctness).

**Solution 3.** The following algorithm begins by sorting $P$ by $x$-coordinates with ties broken by $y$-coordinates. The divide and conquer algorithm then partitions $P$ into two halves by $x$-coordinates and recursively calculates the maximal points in each half. Then the algorithm filters out points in the first half (lower $x$-coordinates) with $y$-coordinates less than the maximum $y$-coordinate in the second half (greater $x$-coordinates).

```
1: procedure MaximalPoints($P$)
2:     Sort $P$ by $x$-coordinates with ties broken by $y$-coordinate (that is, $x_1 \leq x_2 \leq \cdots x_n$, and if $x_i = x_{i+1}$ then $y_i < y_{i+1}$)
3:     $n = P$.length
4:     return MaximalHelp($P, 0, n-1$)
```

```
1: procedure MaximalHelp($P, i, j$)
2:     if $i = j$ then
3:         return $[P[i]]$    ▷ An array with a single element $P[i]$
4:     $m \leftarrow \lfloor (i+j)/2 \rfloor$
5:     $L \leftarrow$ MaximalHelp($P, i, m$)
6:     $R \leftarrow$ MaximalHelp($P, m+1, j$)
7:     $y_R \leftarrow \max_{(x_i,y_i)\in \mathrm{R}} y_i$
8:     $L' = [\,]$    ▷ make an empty list
9:     for $(x_i, y_i) \in L$ do
10:        if $y_i > y_R$ then
11:            Append $(x_i, y_i)$ to $L'$
12:    return $L' + R$    ▷ concatenate $R$ to end of $L'$
```

*Runtime Analysis.* Sorting is done once by the MaximalPoints wrapper, which can be done in $O(n\log(n))$ time with, for example, mergesort. The MaximalHelp procedure makes two recursive calls on input of half the size (lines 5 and 6). The remainder of the procedure takes linear time (dominated, for example, by the for loop on line 9). Therefore the recurrence $T(n) \leq 2T(n/2) + n$ characterizes the runtime complexity, the solution of which is $O(n \log(n))$.

We conclude that the overall runtime complexity is $O(n \log(n))$.

*Correctness.* We will argue the correctness of MaximalHelp by induction on $n = j - i + 1$, the number of points considered. In the base case of $n = 1$, the single point is necessarily undominated.

For the inductive hypothesis (IH) suppose MaximalPoints$(P, i, j)$ returns the set of undominated points in $P$ between $i$ and $j$ for all $j - i + 1 \leq n - 1$. The IH implies that on an input of size $j - i + 1 = n$, the recursive calls correctly calculate $L$ as the maximal points from $i$ to $m$, and $R$ as those from $m + 1$ to $j$.

No point in $L$ can dominate a point in $R$ by the sorting, as points in $R$ either have strictly greater $x$-coordinate or have equal $x$ and greater $y$-coordinate. However, all points in $L$ are dominated by the greatest $y$-coordinate point in $R$ *unless* they have greater $y$-coordinate. So the maximal points are precisely all of $R$ and those points from $L$ with greater $y$-coordinate than the greatest in $R$, which is what the algorithm returns.

**Problem 4 (Applied).** You are designing a piece of software for aircraft controllers that attempts to detect planes that are dangerously close. In particular, your software must calculate the minimum distance between any two (distinct) planes so that if there are any active flights less than some minimum allowable safe distance, an alarm can be raised. Plane locations will be represented by (x, y)-coordinate points (we will ignore height/altitude) and the distance to be used is the standard Euclidean distance (where $d((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$).

You **need to design and implement an algorithm that efficiently determines the minimum distance between any two distinct planes.** Your solution's efficiency will be checked by comparing if its empirical runtime is within constant factors of an $O(n \log^2(n))$ reference solution (where $n$ is the number of points). *Hint.* Remember this is the assignment on recursive divide and conquer algorithms. How can you divide the input spatially, in terms of their x or y coordinates? If you recurse on both divided sides, how can you combine the results? Finally, is there anything you need to check other than the two recursive results?

Language-specific details follow. You can use whichever of Python or Java you prefer. You will receive automatic feedback when submitting, and you can resubmit as many times as you like up to the deadline. **NOTE:** Unlike the theory problems, the applied problem grade **is the raw score shown on Gradescope**. See the course assignments webpage for more details.

- **Python.** You should submit a file called `closest.py` to the Gradescope item "Assignment 2 - Applied (Python)." The file should define (at least) a top level function with the signature `def minimum_distance(points)`. The input `points` will be a list of tuples of $(x, y)$ coordinate pairs, for example `[(0, 0), (2, 2.5), (3, 1.5)]`. Your function should return the minimum Euclidean distance between any pair of points; it would return $\sqrt{2} \approx 1.414$ on the example input above, the distance between `(2, 2.5)` and `(3, 1.5)`.

- **Java.** You should submit a file called `Closest.java` to the Gradescope item "Assignment 2 - Applied (Java).". The file should define a `public class Closest` and (at least) a method with the signature `public static double minimumDistance(List<double[]> points)`. The input `points` will be a list of length-2 arrays representing `x, y` coordinate pairs, for example `[{0, 0}, {2, 2.5}, {3, 1.5}]`. Your function should return the minimum Euclidean distance between any pair of points; it would return $\sqrt{2} \approx 1.414$ on the example input above, the distance between `{2, 2.5}` and `{3, 1.5}`.

**Solution 4.** We do not provide full code solutions, but we briefly sketch the idea of the algorithm here. Begin by sorting the points by their $x$-coordinates from least to greatest. Then use the following divide and conquer algorithm:

- Divide the points into a "left" and "right" half of equal sizes around the median $x$-coordinate.

- Recursively calculate the minimum distance between any two distinct points in the left and the right half. Let $d$ be the lesser of these two distances.

- In the merge step, consider the points within $\pm d$ by $x$-coordinate of the median line used to split the subproblems. For each such point, check all points within $\pm d$ by $y$-coordinate and update $d$ if a lesser distance is discovered.

- Return $d$

The implementation of the merge step that sorts by $y$-coordinates yields an overall runtime complexity of $O(n(\log(n))^2)$.