1. Assume we have a list of nuts $N = [n_1 .. n_k]$ and a list of bolts $B = [b_1 .. b_k]$, where each nut and bolt is of unique size. Each nut matches exactly one bolt and vice versa. The nuts and bolts are visually indistinguishable, therefore you cannot directly compare a pair of bolts or a pair of nuts. However, you can compare a bolt to a nut by trying to fit them, at which point you'll find if the pair is too loose, too tight, or perfectly fit. In addition, you are also given an $O(1)$ oracle $M(S)$ which will return the median of S, where $S \subseteq N$. Describe an $O(k \log k)$ algorithm that matches each nut to each bolt.

Solution: We can utilize the oracle to select the median $\tilde{n}$ of the nuts, and compare $\tilde{n}$ to every bolts. After the comparisons, we will get a set B^+ of the bolts that were bigger than $\tilde{n}$, a set B^- of the bolts that were smaller than $\tilde{n}$, and the bolt $\tilde{b}$ that matches with $\tilde{n}$ (which is the median of B). Then, we can use $\tilde{b}$ to divide $N \setminus \{\tilde{n}\}$ into N^+ and N^- in the same way. Note that every nuts in N^+ would have its matching bolt in B^+, and the same applies for N^-. Therefore we can recursively apply the above method to (N^+, B^+) and (N^-, B^-).

The runtime of the above algorithm is represented with the following recurrence:

$$T(k) = 2T(k/2) + k$$

Since the amount of work at each level is k and there are $\log k$ levels, the total amount of work is $O(k \log k)$.

The following is the pseudo code for the algorithm. Note that in the following pseudo code, the operator $+$ represents list appending while maintaining the order. When applied on two pairs of lists, it represents pairwise appending, so $(A_1, B_1) + (A_2, B_2) = (A_1 + A_2, B_1 + B_2)$.

```
MATCH(N[n_1 .. n_k], B[b_1 .. b_k]):
  if N = []                          ⟨⟨Base case⟩⟩
      return ([], [])
  N^+, N^-, B^+, B^- ← []            ⟨⟨Initialize as empty lists⟩⟩
  ñ ← M(N[n_1 .. n_k])               ⟨⟨Get median using oracle⟩⟩
  for i ← 1 to k
      if b_i > ñ, then B^+ ← B^+ + [b_i]
      else if b_i < ñ, then B^- ← B^- + [b_i]
      else, b̃ ← b_i
  for i ← 1 to k
      if n_i > b̃, then N^+ ← N^+ + [n_i]
      else if n_i < b̃, then N^- ← N^- + [n_i]
  N, B ← MATCH(N^-, B^-) + ([ñ], [b̃]) + MATCH(N^+, B^+)
  return (N, B)
```

■

2. There is a group of n dogs labeled from 1 to n where each dog has a different level of loudness and a different level of smartness. You are given an array, denoted as clever, where clever$[i] = [a_i, b_i]$ indicates that a_i is smarter than b_i and an integer array where quite$[i]$ denotes the quietness of the i^{th} dog. You can assume all the input data is correct.

   Describe and analyze an algorithm that will output an integer array where ret$[x] = y$ if y is the least quiet dog among all dogs who have equal or more intelligence than dog x.

Solution: Let's write some notations first.

- Let n be the number of dogs, labeled from 1 to n.
- clever$[i] = [a_i, b_i]$ indicates that dog a_i is smarter than dog b_i.
- quiet$[i]$ denotes the quietness level of the i^{th} dog.
- dp$[i]$ will store the index of the least quiet dog among all dogs that are at least as intelligent as dog i.

First, we construct a directed graph G, where each vertex represents a dog. Then, for each clever$[i] = [a_i, b_i]$, we add an edge (a_i, b_i). Notice that G is a DAG.

The recurrence to fill the dynamic programming table is:

$$\text{dp}[i] = \begin{cases} i & \text{if } i \text{ is a source;} \\ \min(\text{quiet}[i], \min\limits_{i \text{ reachable from } j} \text{quiet}[\text{dp}[j]]) & \text{otherwise.} \end{cases}$$

This function essentially states that the least quiet dog among those as smart as or smarter than dog i is determined by considering i itself and all dogs that can be reached via directed edges to i. If there is no dog smarter than i (meaning that i is a source in the constructed DAG), then dp$[i] = i$.

To avoid repeated recalculations, we can implement this in a dynamic programming approach by processing the dogs in a topologically sorted order.

```
FindLeastQuietDogs(clever, quiet):
  Construct the directed graph G.
  topOrder ← TopologicalSort(G)
  Initialize dp[i] ← i for all i in 1, ..., n
  for each node b in topOrder:
      for each outgoing edge (b, a):
          if quiet[dp[a]] > quiet[dp[b]]:
              dp[a] ← dp[b]
  return dp
```

- Time Complexity:
  - Topological sorting takes $O(n+m)$, where m is the number of edges in the graph (the size of clever).
  - Filling the dp table requires $O(n+m)$ as each node and edge is processed once.

  - Therefore, the total time complexity is $O(n+m)$.
- Space Complexity:
  - The graph representation (adjacency list) requires $O(n+m)$ space.
  - The dp and quiet arrays each require $O(n)$ space.
  - Hence, the overall space complexity is $O(n+m)$.

■

3. You are given a directed graph $G = (V, E)$ with positive length edges, as well as two vertices s and t. An edge $e \in E$, is considered bad if the cost of all walks from s to t that uses e costs at least 3β, where β is the length of the shortest path from s to t. Describe an algorithm that computes all the *bad* edges in G. Slower algorithms would earn 60% of the total points.

Solution: Let's write some notations first.

- l_e is the length of edge $e \in E$. It is given that $l_e > 0$.
- The cost of a path from s to t through edges $e_1, e_2, \ldots e_n$ can be written as $\sum_{i=1}^{n} l_{e_i}$.
- DIJKSTRA(G, s, t) returns the shortest path between vertices s and t of a graph G. Assume that it is given to us, and it returns $-\infty$ if there is no path from s to t.

First, let's find β. We can use Dijkstra's algorithm to find the shortest path between s and t, and its cost would be β. We can use DIJKSTRA(G, s, t).

One way to solve the problem is to check shortest paths from s and t through all e's in E. If $e = (u, v)$, the shortest path from s to t through e can be computed as

$$\text{DIJKSTRA}(G, s, u) + l_e + \text{DIJKSTRA}(G, v, t)$$

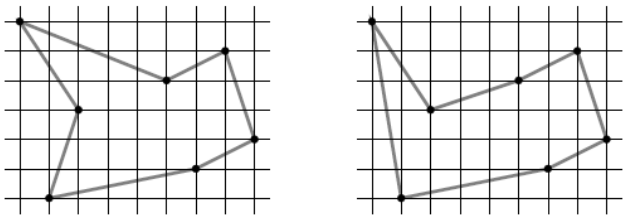
All the edges through which the shortest path has a cost $> 3\beta$ would be the bad edges of G.

```
FindBadEdges(G, s, t):
  es ← φ
  β ← Dijkstra(G, s, t)
  for each e in G.Edges:
      α ← Dijkstra(G, s, e.u) + e.length + Dijkstra(G, e.v, t)
      if α > 3β:
          es.add(e)
  return es
```

The run time of this solution is $O(|E| \cdot (|E| + |V| \cdot \log |V|))$, assuming Dikstra's algorithm has a time complexity of $O(|E| + |V| \cdot \log |V|)$.

Can we improve the time complexity? Yes! Notice that while computing β, we can store the shortest path from s to $v \in V$ in a lookup table, which means DIJKSTRA$(G, s, e.u)$ may be replaced by a table lookup which can be done in $O(1)$. Creating a similar lookup table for DIJKSTRA$(G, e.v, t)$ is slightly tricky. We can create another graph $G' = (V, E')$ with $E' = \{(v, u) \mid (u, v) \in E\}$. We can run Dijkstra's algorithm as DIJKSTRA(G, t, s) and create a lookup table to store the length of the shortest paths from t to $v \in V$. This would reduce the time complexity of the solution to $O(|E| + |V| \cdot \log |V|)$ with an additional space complexity of $O(|V| + |E|)$. ■

4. In the euclidean traveling-salesman problem, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. The figure above shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (we'll learn more about this in the third part of the course but for right now , this is just a bit of flavor).



J. L. Bentley has suggested that we simplify the problem by restricting our attention to bitonic tours, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.

Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. (Hint: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

Solution: We sort the points by increasing x-coordinate, labeling them as $(p_1, p_2, \ldots, p_n)$.

Let $b[i,j]$ be the length of the shortest bitonic path $P_{i,j}$ that starts from point p_i and goes strictly left to the leftmost point p_1, and then goes strictly right to the rightmost point in the bitonic path p_j. $P_{i,j}$ should contain all the points $p_1, p_2, \ldots, p_j$. The final result, representing the shortest bitonic tour, is $b[n,n]$.

The recurrence relation is:

$$b[i,j] = \begin{cases} |p_1p_2| & \text{if } i = 1, j = 2 \quad (a) \\ b[i,j-1] + |p_{j-1}p_j| & \text{if } i < j-1 \quad (b) \\ \min\limits_{1 \le k < j-1} \left\{ b[k,j-1] + |p_kp_j| \right\} & \text{if } i = j-1 \quad (c) \end{cases}$$

(a) **Base Case:** When there are only two points, p_1 and p_2, the bitonic path consists solely of these two points. The path length is simply the Euclidean distance between them: $|p_1p_2|$.

(b) **Extending the Rightward Path:** If the point p_{j-1} lies on the right-going subpath, it immediately precedes p_j on this subpath. Since the rightward path continues, the shortest path to p_j from p_i must involve the shortest bitonic path from p_i to p_{j-1}. If not, we could replace the subpath with a shorter one.

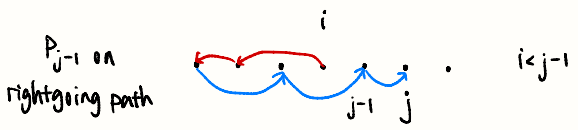


Figure 1. If p_{j-1} is on the rightgoing path, it immediately preceeds p_j.

(c) **On Leftward Path:** Given p_j is the final point on the right-going subpath, if the point p_{j-1} lies on the left-going subpath, and p_{j-1} must be the rightmost point on this subpath, so $i = j-1$. p_j must have an immediate predecessor point p_k, where $1 \le k < j-1$, on the rightgoing path. This case ensures that the subpath from p_k to p_j is the shortest possible, which also ensures the path from p_k to p_{j-1} must also be a shortest bitonic path. Otherwise, we could find a shorter bitonic path than $P_{i,j}$.

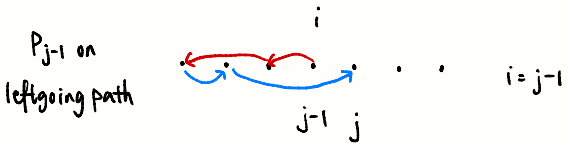


Figure 2. If p_{j-1} is on the leftgoing path, it must be the rightmost point on the leftgoing subpath.

In an optimal bitonic tour, one of the points adjacent to p_n must be in p_{n-1}, so we have $b[n,n] = b[n-1,n] + |p_{n-1}p_n|$. The overall time complexity is $O(n^2)$, where we compute $b[i,j]$ for all $1 \le i,j \le n$.

To reconstruct the points on the shortest bitonic tour, we define $r[i,j]$ to be the index of the immediate predecessor of p_j on the shortest bitonic path $P_{i,j}$. Since the immediate predecessor of p_2 on the path $P_{1,2}$ is p_1, we know that $r[1,2] = 1$.

```
EUCLIDEAN-TSP(p):
  Sort points (p_1, p_2, ..., p_n) by increasing x-coordinate
  Initialize two arrays b[1..n, 1..n] and r[1..n, 1..n]
  b[1,2] ← |p_1 p_2|
  for j ← 3 to n
      for i ← 1 to j-2
          b[i,j] ← b[i,j-1] + |p_{j-1} p_j|
          r[i,j] ← j-1
      b[j-1,j] ← ∞
      for k ← 1 to j-2
          q ← b[k,j-1] + |p_k p_j|
          if q < b[j-1,j]
              b[j-1,j] ← q
              r[j-1,j] ← k
  b[n,n] ← b[n-1,n] + |p_{n-1} p_n|
  return b and r
```

The `Print-Tour` function prints the entire bitonic tour recursively. It starts from the rightmost point p_n and traces backward along the left-going subpath until it reaches the starting point p_1. After completing the left-going subpath, it prints the remaining points in the right-to-left order, ensuring that no point is printed twice.

```
PRINT-TOUR(r, n):
  Print p_n
  Print p_{n-1}
  k ← r[n-1, n]
  PRINT-PATH(r, k, n-1)
  Print p_k
```

`Print-Path` is a helper function used to recursively print segments of the path between two points. The recursion allows us to first print the deeper right-to-left subpath, and then print the left-to-right subpath as the recursion unwinds.

```
PRINT-PATH(r, i, j):
  if i < j
      k ← r[i, j]
      if k ≠ i
          Print p_k
      if k > 1
          PRINT-PATH(r, i, k)
  else
      k ← r[i, j]
      if k > 1
          PRINT-PATH(r, k, j)
      Print p_k
```

The algorithm sorts the points in $O(n \log n)$ time and computes the bitonic path in $O(n^2)$ time. Tour reconstruction takes $O(n)$ time.

■