Another surprising and satisfying application of the D&C method is the following algorithm for finding the closest pair of points in a set of points in the x-y plane.

Assume we have n points, each identified by its x and y coordinates. Our goal is to find the two points with the minimum distance between them. Obviously we can solve this problem in $O(n^2)$ time by computing the distance between all pairs of points. It may seem unlikely that we can reduce this, but by clever design we can eliminate enough of the pairwise distance calculations that we can achieve a lower complexity.

For reasons that will become clear later, we start by creating a list or array of all the points sorted by their x-coordinate and another list of all the points sorted by their y-coordinate. This is a pre-processing phase that sets up the rest of the algorithm.
In this figure we see the points labelled in left-to-right order. This makes it very easy to split the set into a Left half and a Right half, which is exactly what we do. We then recursively solve the problem on the left side and the right side. As always with a D&C algorithm, when the problem size is \( \leq \) some specific number we solve the problem directly. In this case we might decide that when the number of points is \( \leq 3 \) (or 5, or 10 – it really doesn’t matter what cut-off value we pick so long as we stick with it), we will compute all the distances between the points. This takes \( O(1) \) time, which means we can treat it as a constant.

However, after we split the point set into a left side and a right side and solve the problem recursively on both sides, we are not done. We still have to deal with the possibility that the two points that are closest together are on opposite sides of the dividing line. If we compute all the distances between points on the left and points on the right, we are back with \( O(n^2) \) complexity.
Fortunately we are able to avoid most of the potential computations. Let $d_L$ be the minimum distance on the left side, and $d_R$ be the minimum distance on the right side - we get these values from the recursive applications of the algorithm to the left and right halves of the set. Let $\delta = \min(d_L, d_R)$. 
We need to determine if there are two points, one on each side of the dividing line, that have distance less than $\delta$ from each other. We can eliminate all points that have distance more than $\delta$ from the dividing line, since they cannot be less than $\delta$ from any point on the other side of the line.

Imagine a vertical panel or strip, $2 \times \delta$ wide, centred on the dividing line between the left and right sides. The only points we need to consider in this stage of the algorithm are within this panel.
Take these points in ascending order by y-coordinate (oh, so this is why we sorted the points on their y-coordinates before we started!). For each point \( p \), compute its distance to the points in the panel above it that might possibly be less than \( \delta \) away. We can imagine a box \( 2 \times \delta \) wide and \( \delta \) high with \( p \) on its bottom edge. Any point above \( p \) in the vertical panel that could be \( < \delta \) away from \( p \) must be in this box.

The following 3 figures show the magical box of holding for each of the lowest three points in the vertical panel in our example.
By a simple geometric argument there cannot be more than 7 such points (see my note 1 below). Thus for each point in the vertical panel, we need to compute no more than 7 distances to other points in the panel. Even if all n points are in the panel, the complexity of computing the necessary distances within the panel is in O(n).

Thus the complexity of the algorithm is given by the recurrence relation

\[
\begin{align*}
T(n) &= 2 \cdot T(n/2) + c \cdot n & \text{when } n > 3 \\
T(n) &= \text{constant} & \text{when } n \leq 3
\end{align*}
\]

(Remember, the "3" is our arbitrarily chosen cut-off value for the recursion. It

1Note: I have followed the text-book here by saying that there are no more than 7 points to consider. In class I presented an argument that there can be no more than 5 such points. The argument for 7 such points depends on allowing identical points in the set, and having the identical pairs on "opposite sides" of the dividing line. I think this situation can be resolved as a special case. Either way, the number of point-combinations that must be considered is strictly linear.)
can be replaced with any other constant without changing the complexity)

We know this recurrence relation - it is the same one that describes the complexity of merge-sort. We know that it works out to $O(n \log n)$. This is the same as the complexity of the pre-processing step, which means that the pre-processing step is effectively free.

It is worth noting that if we had to re-sort the points at the beginning of each recursive call, the complexity would be higher. Fortunately we don't - each reduced set of points is just a subset of the set at the previous level and the relative order of the points does not change.
Suppose we are planning a trip from Kingston to Charlottetown (and suppose there is only one road so we don’t have a choice of which route to take). Our car will need several stops to fill up the gas tank. We know the location of all the service stations on the highway between Kingston and Charlottetown, and we know exactly how far we can travel on a tank of gas. We want to minimize the number of stops we make along the way.

The intuitive solution, of course, is to go as far as possible on each tank - that is, stop at a service station iff we can't make it to the next one. It turns out that in this case intuition can be trusted - this is exactly the right solution.

But how can we ...

- prove that this algorithm always finds the correct solution?
- determine the computational complexity of this algorithm?

We can formalize this idea into an algorithm as follows. I'll show a recursive formulation first – here we will define a function RT(i) which is supposed to find an optimal solution when we are leaving station s_i with a full tank of gas ... RT(0) will give us the complete solution.

We will assume that the problem has a solution (ie that there is no gap between consecutive stations that exceeds the distance we can travel on one tank of gas ... so when we fill up, we know we can reach the next station). It is easy to test to make sure this condition is satisfied.
1. Sort the stations according to how far they are from Kingston
   Let \( S = \{s_0, s_1, s_2, \ldots, s_n\} \) be the sorted set, with \( s_0 = \) Kingston
   and \( s_n = \) Charlottetown

2. \( \text{RT}(i) \)  
   \# i is the number of the station we are leaving with
   \# a full tank

   \# We know we have enough gas to reach the next station
   \[ t = i+1 \]

   \[ \textbf{while } t \neq n \text{ and } s_{t+1} \text{ is reachable} \]
   \[ \# \text{ ignore station } t \]
   \[ t++ \]

   \[ \textbf{if } t == n: \quad \text{STOP} \]

   \[ \textbf{else :} \]
   \[ \text{fill up at } s_t \]
   \[ \text{RT}(t) \]

We find the solution by calling \( \text{RT}(0) \)

To complete our consideration of this problem, we need to do several things:

1. Show that the problem does have an optimal solution (some problems don't, either because there is no solution or there is no bound on the value of the potential solutions).

2. Show that the greedy algorithm finds an optimal solution

3. Determine the complexity of the algorithm.

First, once the stations are sorted we can easily determine if there are any feasible solutions: the trip is possible iff there is no gap between stations too long for us to cover on one tank of gas. Assuming then that there are feasible solutions, it is clear that there is at least one optimal solution since we can rank all the feasible solutions according to their cardinality and choose one with minimum cardinality.

It is also clear that the algorithm will find some solution to the problem: it never lets us run out of gas, and it gets us to the destination.
Let's also dispose of the complexity issue: sorting the n stations takes $O(n \times \log n)$ time. Choosing the stations where we stop takes $O(n)$ time since we look at each station exactly once, and decide in constant time whether or not to stop there. Thus the entire algorithm takes $O(n \times \log n)$ time ... or just $O(n)$ if the stations are given to us already sorted.

Now the proof of correctness. This is the interesting part. We use Proof by Induction.

**Base Case:** If $n = 1$, the only possible stop is Charlottetown. If there is any feasible solution (i.e. if we can reach Charlottetown on a single tank), then the algorithm's action is correct (it makes no stops before Charlottetown). Thus for $n = 1$ the algorithm finds an optimal solution.

**Inductive Hypothesis**

We assume that the algorithm finds an optimal solution when we have a full tank and the number of service stations still ahead of us is $\leq k$, for some $k \geq 1$.

**Inductive Step**

Suppose there are $k+1$ service stations still ahead of us and we have a full tank when we leave Kingston.

Let the algorithm’s solution be $S_A = \{s_i, s_j, \ldots, s_n\}$ ... the solution lists the stations chosen for stops. So $s_i$ is the first stop and $s_n$ (Charlottetown) is the last.

First we show that there exists at least one optimal solution that starts with the algorithm’s first choice. To do this, let $O$ be some optimal solution to the problem, i.e. $O = \{o_1, o_2, \ldots, o_t\}$ where each element is a station where we should fill up with gas, in order. Note that $o_t = s_n$ since the trip must end in Charlottetown.
Consider $s_i$: the algorithm's first stop. From the definition of the algorithm we know it is not possible to reach $s_{i+1}$ on the first tank of gas. Therefore $o_1 \leq s_i$

But now consider $o_2$ the second stop in $O$. It must be reachable from $o_1$ ... which means it is also reachable from $s_i$. Thus we can create a new feasible solution:

Let $O^* = \{s_i, o_2, ..., o_t\}$

Note that $|O^*| = |O|$, so $O^*$ is also optimal. Thus there is an optimal solution that contains $s_i$.

Now we know that the algorithm's first decision is 'safe' - ie there is at least one optimal solution that matches this decision.

When we make a decision about the first stop, the problem reduces in size to $\leq k$ (i.e. the number of remaining stations between us and Charlottetown is reduced, and our tank is full again). By the IH, the algorithm will find an optimal solution to this reduced problem.

The last step of our proof is to show that when we combine the algorithm's first choice with its solution to the reduced problem, we get an optimal solution to the original problem.

We know that there is at least one optimal solution that matches the algorithm's first choice $s_i$ – we identified $O^*$ above as one such, so let’s use that one again: $O^* = \{s_i, o_2, o_3, ..., o_t\}$. $|O^*| = t$

Let the algorithm’s solution be $S_A = \{s_i, s_j, ..., s_n\}$, as before. Let $|S_A| = m$

Due to the structure of the algorithm, we know $S_A$ is a feasible solution.
Observe that \( \{s_j, \ldots s_n\} \) and \( \{o_2, \ldots o_t\} \) are both solutions to the problem of getting from station \( s_i \) to Charlottetown, based on the decision made to fill up at station \( s_i \). In fact, we know \( \{s_j, \ldots s_n\} \) is an \textbf{optimal} solution to this reduced problem of getting from station \( s_i \) to Charlottetown.

Thus \( |\{s_j, \ldots s_n\}| \leq |\{o_2, \ldots o_t\}|. \) Since \( S_A \) and \( O^* \) both start with \( s_i \), it is clear that \( |S_A| \leq |O^*|. \) Thus \( S_A \) is also an optimal solution.

This completes the proof that \( \text{RT()} \) finds an optimal solution to the Road Trip problem.

Based on this example we can state the general Greedy Algorithm paradigm:

The fundamental principle of greedy algorithms is "take the best thing first".

More formally, suppose we are trying to solve an \textbf{optimization problem} that involves choosing objects from a set, subject to some feasibility constraint. We want to choose the objects that will maximize (or minimize) the total value of the solution, while satisfying the constraint. The \textbf{Greedy Paradigm} looks like this:

1. Sort the objects according to some criterion
2. repeat
   select the next item in the list if it does not violate the feasibility constraint
   until no more selections can be made
For the Road Trip problem, the algorithm can be phrased as

1. Sort the objects according to the distance from the start
2. repeat
   skip the next station if that doesn’t cause us to run out of gas
   until we reach the destination

Our original goal was to minimize the number of stations we stop at ... but this is exactly equivalent to maximizing the number stations we skip.

The Greedy Algorithm is such a simple idea that we cannot expect it to find optimal solutions for all problems (for example, it is very easy to define greedy algorithms that find solutions to some instances of NP-Complete problems ... but there will be instances of the problem for which the algorithm's solutions won't be optimal).

However, there is a huge class of problems for which simple greedy algorithms will always find the optimal solution. This means that our big job with respect to Greedy Algorithms will not be coming up with the algorithm, but finding a proof that the algorithm gives the optimal solution for all instances of the problem.