Here's another demonstration of the Branch and Bound method.

Suppose we have n persons and n tasks – each person is to be assigned to one of the tasks. Because the people have different skill sets, the time required to complete the tasks can be different for each person. Our goal is to assign the tasks in such a way that the total time required is minimized.

For example, suppose we have 5 persons $\{A, B, C, D, E\}$ and 5 tasks $\{T1, T2, T3, T4, T5\}$. We can represent the time requirements with a matrix:

	T1	T2	T3	T4	T5
A	10	15	4	12	9
B	7	18	3	10	16
C	4	5	14	12	10
D	17	2	18	6	21
E	21	18	2	10	25

At this point I have to point out that there exists a polynomial-time algorithm to solve this problem! It is called the Hungarian Method – if we had an extra week in the course I would teach it to you because it is a brilliant and widely applied algorithm. I recommend it to you as an independent learning activity.

However it's worth looking at how we can solve this problem using Branch and Bound because it introduces some clever techniques that we can apply to other problems that do not have polynomial-time algorithms.

Step by step:

Objective function: we are trying to minimize the total time – ie the sum of the times required for the assignments we choose.

Decision sequence: Our i^{th} decision will be to assign a task to the i^{th} person. Note that we could turn this around and make our i^{th} decision to assign a person to the i^{th} task.

Initial Global Upper Bound:

To find this initial bound we can simply generate a solution and use its total cost – the optimal solution's cost must be $\leq$ this.

One way to do this is to assign the first task to the first person, etc. In the example above this gives a total of 73. This is a very quick and easy way to get an initial upper bound.

If we are willing to do a bit more work we can improve this (remember, we always want to make our upper bounds lower and our lower bounds higher). We can apply a simple greedy heuristic: give person A the task they can do the fastest, then give B the remaining task they can do the fastest, etc. This results in this assignment:

	T1	T2	T3	T4	T5
A	10	15	4	12	9
B	7	18	3	10	16
C	4	5	14	12	10
D	17	2	18	6	21
E	21	18	2	10	25

This has a total cost of 47 – much better than the 73 we had before. The trade-off is that it takes $O(n)$ time to get the trivial bound, and $O(n^2)$ to get the improved bound … but if this results in a significant speed-up of the Branch and Bound algorithm it may be worth it.

There are more things we can try – for example, we could apply the Greedy heuristic n times, starting with each one of the persons. This raises the time requirement to $O(n^3)$ … but it may result in a very good initial upper bound.

But here comes the main reason for introducing this problem: there's a technique we can apply to extract "hidden" information from the matrix that can help us with our bounds. Notice that person A takes at least 4 hours for each task. We can subtract 4 from every value in A's row without changing the optimal task assignment – we just have to remember to add

the 4 back in to get the actual cost. And … we can do the same for every person: we subtract their smallest time requirement from all the values in their row. This gives this matrix:

	T1	T2	T3	T4	T5
A	6	9	0	8	5
B	4	15	0	7	13
C	0	1	10	8	6
D	15	0	16	4	19
E	19	16	0	8	23

The total extracted costs are 4+3+4+2+2 = 15

But we're not done yet: Notice that all the values in the column for T4 are positive. In fact we can see that no matter who does task T4 it will take at least 4 hours. So we can subtract 4 from every value in this column and add 4 to our accumulated costs … and by the same logic we can extract 5 from the column for T5.

This gives

	T1	T2	T3	T4	T5
A	6	9	0	4	0
B	4	15	0	3	8
C	0	1	10	4	1
D	15	0	16	0	14
E	19	16	0	4	18

The total extracted cost is now 15 + 4 + 5 = 24

Very clever, but what's the point? Look what happens when we apply the Greedy heuristic to get an initial upper bound:

	T1	T2	T3	T4	T5
A	6	9	0	4	0
B	4	15	0	3	8
C	0	1	10	4	1
D	15	0	16	0	14
E	19	16	0	4	18

0+3+0+0+18 = 21

When we add the extracted costs we get 21+24 = 45 as the cost of this solution … and that is LOWER than the total cost we got by just applying the Greedy heuristic to the original matrix. So we have found a better initial upper bound.

(Note that if we try starting the Greedy heuristic with B, C, D or E we get even better results.)

Now we turn to computing the L and U values for partial solutions. As before we will compute CostSoFar (CSF) , GuaranteedFutureCosts (GFC) and FeasibleFutureCosts (FFC) for each partial solution, and let L = CSF + GFC, U = CSF + FFC

CSF: This is easy – it is just the sum of the costs of the decisions made so far, plus the extracted costs as defined above.

GFC: We can look at the remaining persons and the remaining tasks. For each remaining person, we can find the remaining task that takes the least time – this is a guaranteed future cost because any other task for this person would cost at least as much.

Oh, but this is familiar! If any remaining person has positive costs for all remaining tasks we can subtract the smallest value from every value in the row, and add the value to the accumulated extracted costs for this partial solution. And we can do the same for each column! As we proceed, more and more elements of the matrix are reduced to 0, and that has a payoff as we shall see below.

FFC: For the feasible future cost we can use the same approach as we did for the initial upper bound: apply the Greedy heuristic one or more times. And now think about those 0's. The

more 0's there are in the matrix, the more likely we are to find a feasible solution with a total future cost of 0. And since that obviously cannot be improved on, it means we have found the best possible expansion of the current partial solution. So we can replace this partial solution with the full expansion we just found. This accelerates the algorithm by eliminating all the iterations required to explore other possible expansions of this partial solution.

This will probably make more sense if we do an example. Consider the first step of the algorithm. We generate a partial solution for each of the possible initial choices: assigning A to each of T1, T2, … T5

Here's the matrix again

	T1	T2	T3	T4	T5
A	6	9	0	4	0
B	4	15	0	3	8
C	0	1	10	4	1
D	15	0	16	0	14
E	19	16	0	4	18

Our first partial solution assigns T1 to A. This pair has a cost of 6. Thus the CSF for this partial solution is 24 + 6 = 30. This assignment reduces the matrix to this:

	T2	T3	T4	T5
B	15	0	3	8
C	1	10	4	1
D	0	16	0	14
E	16	0	4	18

The row for C has all values > 0 so we can extract a cost of 1 from this row, giving

	T2	T3	T4	T5
B	15	0	3	8
C	0	9	3	0
D	0	16	0	14
E	16	0	4	18

We can consider the 1 we extracted as the GFC, but in practical terms it is easier to recognize it as part of the CSF since it results from the decisions made up to this point. Thus we now have CSF = 30 + 1 = 31

Now every row and column contains a 0 so no further reductions are possible at this time.

Having made this reduction, our GFC = 0 . We end up with L = 31 for this partial solution.

For the FFC we can apply the Greedy heuristic – I'll just do it once, starting with B. We get a FFC of 18. This gives U = 31 + 18 = 49

I won't do all 5 of the initial set of partial solutions, but one more may clarify things. Let's look at the partial solution resulting from assigning A to T3

This decision has a direct cost of 0. The reduced matrix is

	T1	T2	T4	T5
B	4	15	3	8
C	0	1	4	1
D	15	0	0	14
E	19	16	4	18

We can reduce the row for B by 3, the row for E by 4, and the column for T5 by 1. The result is

	T1	T2	T4	T5
B	1	12	0	4
C	0	1	4	0
D	15	0	0	13
E	15	12	0	13

CSF = 24 + 0 + 3 + 4 + 1 = 32

GFC = 0

FFC = 13 (Note that if we start the Greedy heuristic at E, we get FFC = 1 !!)

L= 32

U=45

And so we carry on in standard B&B fashion until we find an optimal solution. The only tricky bit is making sure we keep track of the extracted costs properly, which is actually pretty easy: each partial solution inherits all the extracted costs from its parent, and possibly adds more – all of which are passed on to its children.

Now that we have seen this "matrix reduction" idea, let's consider a really difficult problem: the Traveling Salesperson Problem, or TSP. In this problem we imagine a salesperson who has to choose a route that takes them around a cycle of n cities, visiting each city once and returning to where they started. All the cities are connected by direct roads and all the roads have different costs. The goal is to find the route with the smallest total cost. A small example might look like this:

	Burgerburgh	Pizzapolis	Nachoville	Timbitton
Burgerburgh	∞	15	13	8
Pizzapolis	15	∞	4	11
Nachoville	13	4	∞	14
Timbitton	8	16	14	∞

Note the ∞ entries on the diagonal – this is just a simple way of making sure we never try to go from a city to itself. Basically any solution that tries to do this ends up with an infinitely large cost so it is rejected immediately – we don't need to code anything extra to handle this.

We can think of the row for Burgerburgh as representing the different costs of going from Burgerburgh to any of the other cities … and we can reduce the row by subtracting off the smallest of them. Of course we do the same for all the rows. Similarly we can think of the columns as representing the costs to arrive at the cities. If any column is all > 0 after the rows are reduced we can reduce that column as well.

As we progress through the algorithm we lose rows and columns of the matrix as we saw in the task assignment problem. For example if we choose to go from Pizzapolis to Timbitton we can delete the row for Pizzapolis and the column for Timbitton.

This is important because we absolutely do not have a polynomial-time algorithm for the TSP problem. There are some restricted cases where we can find the optimal solution easily, and some cases where we can get a good approximation to the optimal answer quite quickly, but the general TSP problem is NP-Hard … which means it is at least as hard as all the NP-Complete problems. In other words we really don't expect to ever find a polynomial-time algorithm for it. The best algorithm to find the optimal solution to a general case TSP problem is – you guessed it – Branch and Bound.