For **Best first**, we need to think about how to manage the set of “live” partial solutions so that we can quickly choose the one with the lowest lower bound.

One method is to store the partial solutions in a **min-heap**. Each new item can be inserted in $O(\log t)$ time, and each choice for the next partial solution to be expanded can be extracted in $O(\log t)$ time, where $t$ is the number of solutions in the heap. Since $t$ may be $O(2^n)$ where $n$ is the number of decisions to be made, this gives us $O(n)$ time for selecting the next partial solution and for inserting new partial solutions.
The more accurate the lower and upper bounds for each partial solution are, the more effectively the bad branches can be pruned.

In some applications, it may be worth using quite complex algorithms to compute good bounds.
The more accurate the lower and upper bounds for each partial solution are, the more effectively the bad branches can be pruned.

In some applications, it may be worth using quite complex algorithms to compute good bounds.

For a partial solution $P$, the lower bound consists of two parts:

- **Cost so far**: the cost of decisions already made
- **Guaranteed future cost**: unavoidable costs from future decisions

The upper bound also consists to two parts:

- **Cost so far**: same as above
- **Feasible future cost**: the cost of any extension of $P$ to a complete solution
The quality of the initial upper bound can be critically important.

Rather than randomly choosing a solution to give the initial upper bound, it is sometimes worthwhile to invest the time to find a fairly good solution for this purpose.

This can be done with an heuristic algorithm that runs in polynomial time but doesn't always find the optimal answer.

For example, we might be solving a problem for which there is no greedy algorithm solution. However, we might use a greedy algorithm to get the initial upper bound, and then use branch and bound to find the optimal solution.
Let’s do an example!

The 0-1 Knapsack Problem: We have a collection of objects, each with a known volume and a known value. We have a knapsack with a known capacity. We want to choose the most valuable set of objects that will fit in the knapsack.

This is an NP-Complete problem.

With n objects to choose from, there are potentially $2^n$ possible solutions to be considered (every subset of the set of objects).

But with a branch and bound algorithm, we can try to cut this down a bit.
First, what is our objective function?

The obvious one is to compute the value of items chosen, and try to maximise it ...
First, what is our objective function?

The obvious one is to compute the value of items chosen, and try to maximise it …

… except that we have developed the algorithm in terms of minimisation.
First, what is our objective function?

The obvious one is to compute the value of items chosen, and try to maximise it …

… except that we have developed the algorithm in terms of minimisation.

So let’s compute the value of the objects not chosen - minimising this will maximise the value of the set of objects we choose.
Second, how can we conceptualise this as a sequence of decisions?
Second, how can we conceptualise this as a sequence of decisions?

Easy - list the objects in some order. At each stage, we make the decision to include the next object or not.
Choosing a solution to get an initial upper bound:

We can use a simple greedy algorithm, based on choosing the object with the maximum ratio of value to volume.
Computing lower and upper bounds for partial solutions:

Each partial solution contains a “cost so far” - the value of all items already excluded. This certainly works as a lower bound on the cost of all extensions of the partial solution … but we can do better. How?
Computing lower bounds for partial solutions:

Each partial solution contains a “cost so far” - the value of all items already excluded. This certainly works as a lower bound on the cost of all extensions of the partial solution … but we can do better. How?

We can exclude all objects yet to be considered which will not fit in the knapsack on top of the objects already chosen.
Computing upper bounds for partial solutions:

The “cost so far” obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...
Computing upper bounds for partial solutions:

The “cost so far” obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...

Remember that any solution gives us an upper bound on the cost of an optimal solution ...
Computing upper bounds for partial solutions:

The “cost so far” obviously contributes to the upper bound, and a simple and valid extension is to imagine that we will also leave out of the knapsack all the objects not yet considered. But we can do better than that ...

Remember that any solution gives us an upper bound on the cost of an optimal solution ...

so applying the greedy heuristic to the remaining objects will give us a better upper bound for the current partial solution.
Why Use “Best First”?

Unless there is good reason not to, “best first” is the normal method for selecting which partial solution to expand first. (Recall that “best first” means “choose the element of S with the lowest lower bound”).

This could be called optimistic choice, since it suggests that hope to actually achieve the lower bound on the partial solution we select. It is important to remember that for any given partial solution, the best expansion may have a total cost anywhere between its bounds. So the “best first” choice may not lead to the optimal solution ...
Best choice version of Branch and Bound

while True:
    Choose P in S with minimum $L_p$
    if P is a complete solution:
        Break
    else:
        $S = S \setminus \{P\}$
        for each $P'$ that can be built from P with one more step,
            compute $(L_{p'}, U_{p'})$
            if $L_{p'} > U$, discard $P'$
            else
                $S = S + \{P'\}$
                if $U_{p'} < U$, $U = U_{p'}$
        end for each
end while
Return P
The big differences between this version of the algorithm and the original are:

- now we toss full solutions right back into S, instead of keeping track of the best one we have seen
- As soon as a full solution in S is selected, we stop (even if S is still full of partial solutions)

This is valid because when a full solution P is generated, its lower bound and upper bound are equal (there is no more uncertainty), and when a full solution is selected, its cost is $\leq$ the best possible expansion of all other items in S (this is how P is chosen). Thus there can be no other solution that beats this one.