Algorithms Week 3

Ljubomir Perković, DePaul University

Consider the problem of navigating a maze. In general, one does not know the structure of the maze when starting. In fact, a maze may be constructed not to have any particular structure. So there is typically no way to break down the problem into smaller subproblems.

Consider the problem of navigating a maze. In general, one does not know the structure of the maze when starting. In fact, a maze may be constructed not to have any particular structure. So there is typically no way to break down the problem into smaller subproblems.

One way to solve a maze is to try every potential solution in the search space, i.e. every possible sequence of steps from the maze entrance until the maze exit is found.

Consider the problem of navigating a maze. In general, one does not know the structure of the maze when starting. In fact, a maze may be constructed not to have any particular structure. So there is typically no way to break down the problem into smaller subproblems.

One way to solve a maze is to try every potential solution in the search space, i.e. every possible sequence of steps from the maze entrance until the maze exit is found.

Backtracking systematizes the search using the following rule: when we hit a dead end, we back up until we find a place where a choice we have not tried can be made and we make that choice.

The basic strategy of backtracking is to *systematically* explore the space of all potential solutions. It does so by:

- extending a partial solution in some feasible way

The basic strategy of backtracking is to *systematically* explore the space of all potential solutions. It does so by:

- extending a partial solution in some feasible way
- trying another extension if the extension fails,

The basic strategy of backtracking is to *systematically* explore the space of all potential solutions. It does so by:

- extending a partial solution in some feasible way,
- trying another extension if the extension fails,
- backing up to a smaller partial solution when the options for extending a partial solution are exhausted.

Classic problem to illustrate backtracking.

- Input: An integer $n \geq 4$.
- Output: A placement of n queens on an $n \times n$ chessboard so that no two can attack each other.

Classic problem to illustrate backtracking.

- Input: An integer $n \geq 4$.
- Output: A placement of n queens on an $n \times n$ chessboard so that no two can attack each other.

Recall that a queen may move any length along a diagonal or any length along a row or column.

Classic problem to illustrate backtracking.

- Input: An integer $n \geq 4$.
- Output: A placement of n queens on an $n \times n$ chessboard so that no two can attack each other.

Recall that a queen may move any length along a diagonal or any length along a row or column.

A solution for $n = 4$:

n-Queens problem

Classic problem to illustrate backtracking.

- **Input:** An integer $n \geq 4$.
- **Output:** A placement of n queens on an $n \times n$ chessboard so that no two can attack each other.

Recall that a queen may move any length along a diagonal or any length along a row or column.

A solution for $n = 4$:

This solution can be represented as $[(1,3),(2,1),(3,4),(4,2)]$.

In order to use backtracking for this problem, we need to describe what a potential solution looks like.

As seen in the previous example, a potential solution is n points in an $n \times n$ array:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

In order to use backtracking for this problem, we need to describe what a potential solution looks like.

As seen in the previous example, a potential solution is n points in an $n \times n$ array:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

Note that an exhaustive search will check $(n^2)^n$ possible solutions, one for each set of n coordinates.

In order to use backtracking for this problem, we need to describe what a potential solution looks like.

As seen in the previous example, a potential solution is n points in an $n \times n$ array:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n).$$

Note that an exhaustive search will check $(n^2)^n$ possible solutions, one for each set of n coordinates.

Let us use backtracking to narrow down the search space. To do this, we first need to design a more efficient representation of potential solutions and to develop tests that check whether a partial solution can be extended to a complete solution.

We can make several observations that eliminate some of the possible partial solutions:

1. There must be exactly one queen per row; this means that the solutions must have the form:

$$(1, y_1), (2, y_2), \ldots, (n, y_n).$$

We can make several observations that eliminate some of the possible partial solutions:

1. There must be exactly one queen per row; this means that the solutions must have the form:

$$(1, y_1), (2, y_2), \ldots, (n, y_n).$$

2. There must be exactly one queen per column; this means that $(y_1, y_2, \ldots, y_n)$ must be a permutation of $(1, 2, \ldots, n)$. Note that this decreases the search space to "only" $n!$ permutations.

We can make several observations that eliminate some of the possible partial solutions:

1. There must be exactly one queen per row; this means that the solutions must have the form:

$$(1, y_1), (2, y_2), \ldots, (n, y_n).$$

2. There must be exactly one queen per column; this means that $(y_1, y_2, \ldots, y_n)$ must be a permutation of $(1, 2, \ldots, n)$. Note that this decreases the search space to "only" $n!$ permutations.

3. There is at most one queen on each diagonal; this means that if two queens are at positions (a, b) and (c, d) then $|a - c| \neq |b - d|$.

We can make several observations that eliminate some of the possible partial solutions:

1. There must be exactly one queen per row; this means that the solutions must have the form:

$$(1, y_1), (2, y_2), \ldots, (n, y_n).$$

2. There must be exactly one queen per column; this means that $(y_1, y_2, \ldots, y_n)$ must be a permutation of $(1, 2, \ldots, n)$. Note that this decreases the search space to "only" $n!$ permutations.

3. There is at most one queen on each diagonal; this means that if two queens are at positions (a, b) and (c, d) then $|a - c| \neq |b - d|$.

We say that we prune a partial solution if we do not extend it any further because it cannot be extended to a complete solution.

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

Step 1: Choose a column for the queen in row 1.

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

Step 1: Choose a column for the queen in row 1.

Step i: Given the positions of the first $i-1$ queens $(1, y_1), (2, y_2) \ldots, (i-1, y_{i-1})$, choose the position for (i, y_i).

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

Step 1: Choose a column for the queen in row 1.

Step i: Given the positions of the first $i-1$ queens $(1, y_1), (2, y_2), \ldots, (i-1, y_{i-1})$, choose the position for (i, y_i).

If you cannot complete Step i for some choice of previous positions $(1, y_1), (2, y_2), \ldots, (i-1, y_{i-1})$ then backtrack and re-choose the position for $(i-1, y_{i-1})$.

We will generate all $n!$ permutation until we find a solution. We do this by building up the solutions one position at a time.

Step 1: Choose a column for the queen in row 1.

Step i: Given the positions of the first $i-1$ queens $(1, y_1), (2, y_2) \ldots, (i-1, y_{i-1})$, choose the position for (i, y_i).

If you cannot complete Step i for some choice of previous positions $(1, y_1), (2, y_2), \ldots, (i-1, y_{i-1})$ then backtrack and re-choose the position for $(i-1, y_{i-1})$.

We will prune a partial solution if it forces the placement of two queens on the same column or diagonal.

Let $Q[1 \dots n]$ store the positions of the n queens, i.e. $Q[i]$ will be the column of the queen in row i.

```
PlaceQueens(Q[1 .. n], r):
  // extends a partial solution of size r-1
  // to one of size r,
  // unless the partial solution
  // is a complete solution, in
  // which case it is output;
  // the initial call is PlaceQueens(Q, 1)
  // with Q storing a partial solution of size 0







.
```

Let $Q[1 \ldots n]$ store the positions of the n queens, i.e. $Q[i]$ will be the column of the queen in row i.


```
PlaceQueens(Q[1 .. n], r):
  if r = n + 1
    print Q[1 .. n]
  else
    // extends a partial solution of size r-1
    // to one of size r
```


.

Let $Q[1 \dots n]$ store the positions of the n queens, i.e. $Q[i]$ will be the column of the queen in row i.

```
PlaceQueens(Q[1 .. n], r):
  if r = n + 1
    print Q[1 .. n]
  else
    for j ← 1 to n
      Q[r] ← j
      if Feasible(Q, r)
        PlaceQueens(Q, r + 1)

Feasible(Q[1 .. n], r)
  // Return true if
  // the placement of the queen
  // in row r is feasible,
  // false otherwise
```

n-Queens backtracking algorithm

Let $Q[1 \ldots n]$ store the positions of the n queens, i.e. $Q[i]$ will be the column of the queen in row i.


```
PlaceQueens(Q[1 .. n], r):
  if r = n + 1
    print Q[1 .. n]
  else
    for j ← 1 to n
      Q[r] ← j
      if Feasible(Q, r)
        PlaceQueens(Q, r + 1)

Feasible(Q[1 .. n], r)
  for i ← 1 to r-1
    if Q[i] = Q[r] or |Q[i] - Q[r]| = |i-r| then
      return false
  return true
```

Backtracking is a very general-purpose algorithm design technique.

Most game-playing programs use a version of bactracking to decide on the next move

For example, one can develop a relatively simple backtracking algorithm that can play a two-player game perfectly (assuming a game that has no randomness or hidden information and that ends after a finite number of moves)

Given the current state of the game, this backtracking algorithm can tell you whether it is possible to win against another perfect player and also tell you how to win.

This algorithm takes as input the state of the game X and one of the two players and returns Good or Bad depending on whether player can win or not.


```
PlayAnyGame(X, player):
  if player has already won in state X
    return Good
  if player has already lost in state X
    return Bad
  for all legal moves X ⇝ Y
    if PlayAnyGame(Y, ¬ player) = Bad
      return Good
  return Bad
```

Subset sum is a classic problem:

Input: A set S of positive integers and an integer T.

Output: A subset X of S such that $\sum_{x \in X} x = T$.

Subset sum is a classic problem:

Input: A set S of positive integers and an integer T.

Output: A subset X of S such that $\sum_{x \in X} x = T$.

Example: Let $S = \{4, 7, 6, 3, 1\}$ and $T = 10$.

Subset sum is a classic problem:

Input: A set S of positive integers and an integer T.

Output: A subset X of S such that $\sum_{x \in X} x = T$.

Example: Let $S = \{4, 7, 6, 3, 1\}$ and $T = 10$.

- $4 + 6 = 10$, so $X = \{4, 6\}$ is a valid solution.

Subset sum is a classic problem:

Input: A set S of positive integers and an integer T.

Output: A subset X of S such that $\sum_{x \in X} x = T$.

Example: Let $S = \{4, 7, 6, 3, 1\}$ and $T = 10$.

- $4 + 6 = 10$, so $X = \{4, 6\}$ is a valid solution.
- $6 + 3 + 1 = 10$, so $X = \{6, 3, 1\}$ is a valid solution.

Subset sum is a classic problem:

Input: A set S of positive integers and an integer T.

Output: A subset X of S such that $\sum_{x \in X} x = T$.

Example: Let $S = \{4, 7, 6, 3, 1\}$ and $T = 10$.

- $4 + 6 = 10$, so $X = \{4, 6\}$ is a valid solution.
- $6 + 3 + 1 = 10$, so $X = \{6, 3, 1\}$ is a valid solution.
- $4 + 3 + 1 = 8$, so $X = \{4, 3, 1\}$ is **not** a valid solution.

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Now consider a particular number e of S. Note that either

$$e \in X \qquad \text{or} \qquad e \notin X.$$

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Now consider a particular number e of S. Note that either

$$e \in X \qquad \text{or} \qquad e \notin X.$$

- If $e \in X$ then it follows that there is a subset $(X \setminus \{e\})$ of $S \setminus \{e\}$ that adds up to $T - e$.

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Now consider a particular number e of S. Note that either

$$e \in X \qquad \text{or} \qquad e \notin X.$$

- If $e \in X$ then it follows that there is a subset $(X \setminus \{e\})$ of $S \setminus \{e\}$ that adds up to $T - e$.

- If $e \notin X$ then there is a subset (X) of $S \setminus \{e\}$ that adds up to T.

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Now consider a particular number e of S. Note that either

$$e \in X \qquad \text{or} \qquad e \notin X.$$

- If $e \in X$ then it follows that there is a subset $(X \setminus \{e\})$ of $S \setminus \{e\}$ that adds up to $T - e$.

- If $e \notin X$ then there is a subset (X) of $S \setminus \{e\}$ that adds up to T.

In both cases we reduce the problem with inputs S and T to two smaller problem instances: one with inputs $S \setminus \{e\}$ and $T - e$ and one with inputs $S \setminus \{e\}$ and T.

Suppose that X is a solution. In other words, $\sum_{x \in X} x = T$.

Now consider a particular number e of S. Note that either

$$e \in X \quad \text{or} \quad e \notin X.$$

- If $e \in X$ then it follows that there is a subset $(X \setminus \{e\})$ of $S \setminus \{e\}$ that adds up to $T - e$.

- If $e \notin X$ then there is a subset (X) of $S \setminus \{e\}$ that adds up to T.

In both cases we reduce the problem with inputs S and T to two smaller problem instances: one with inputs $S \setminus \{e\}$ and $T - e$ and one with inputs $S \setminus \{e\}$ and T.

In other words the solution for inputs S and T can be constructed from the solutions to the problem with inputs $S \setminus \{e\}$ and $T - e$ and to the problem with inputs $S \setminus \{e\}$ and T.

What about the base case of the recursion?

What about the base case of the recursion?

- If $T = 0$, the solution is $X = \emptyset$

What about the base case of the recursion?

- If $T = 0$, the solution is $X = \emptyset$
- If $T < 0$, there is no solution.

What about the base case of the recursion?

- If $T = 0$, the solution is $X = \emptyset$
- If $T < 0$, there is no solution.
- If $T > 0$ and $S = \emptyset$, there is no solution.

Recursive solution

What about the base case of the recursion?

- If $T = 0$, the solution is $X = \emptyset$
- If $T < 0$, there is no solution.
- If $T > 0$ and $S = \emptyset$, there is no solution.

So, recursion can be used to implement an algorithm.

What about the base case of the recursion?

- If $T = 0$, the solution is $X = \emptyset$
- If $T < 0$, there is no solution.
- If $T > 0$ and $S = \emptyset$, there is no solution.

So, recursion can be used to implement an algorithm.

Backtracking makes use of the recursion to more systematically search the space of potential solutions and prune the search when we find partial solutions that cannot be extended to a complete valid solution.

To solve the problem via backtracking we need a way to represent potential partial solutions. We start by assuming that the numbers in set S are stored in an array $S[1..n]$.

To solve the problem via backtracking we need a way to represent potential partial solutions. We start by assuming that the numbers in set S are stored in an array $S[1..n]$.

We then use array $X[1..r]$ containing zeros and ones to represent a partial potential solution that incorporates our decisions whether to include numbers $S[1], S[2], \ldots, S[r]$, where

- $X[i] = 1$ if number $S[i]$ is included in the partial solution.
- $X[i] = 0$ if number $S[i]$ is not included in the partial solution.

To solve the problem via backtracking we need a way to represent potential partial solutions. We start by assuming that the numbers in set S are stored in an array $S[1..n]$.

We then use array $X[1..r]$ containing zeros and ones to represent a partial potential solution that incorporates our decisions whether to include numbers $S[1], S[2], \ldots, S[r]$, where

- $X[i] = 1$ if number $S[i]$ is included in the partial solution.
- $X[i] = 0$ if number $S[i]$ is not included in the partial solution.

When $r = n$ then the partial potential solution is a complete potential solution.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

- $X = []$ is the starting partial solution that does not incorporate any decisions about including numbers in $S[1..n]$; it represents an empty set.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

- $X = []$ is the starting partial solution that does not incorporate any decisions about including numbers in $S[1..n]$; it represents an empty set.
- $X = [0]$ represents the partial solution that incorporates our decisions on number $S[1]$; the number is not included and X represents an empty set as well.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

- $X = []$ is the starting partial solution that does not incorporate any decisions about including numbers in $S[1..n]$; it represents an empty set.
- $X = [0]$ represents the partial solution that incorporates our decisions on number $S[1]$; the number is not included and X represents an empty set as well.
- $X = [1, 0]$ represents the partial solution that incorporates our decision on numbers $S[1]$ and $S[2]$; $S[1]$ is in and $S[2]$ is out.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

- $X = []$ is the starting partial solution that does not incorporate any decisions about including numbers in $S[1..n]$; it represents an empty set.
- $X = [0]$ represents the partial solution that incorporates our decisions on number $S[1]$; the number is not included and X represents an empty set as well.
- $X = [1, 0]$ represents the partial solution that incorporates our decision on numbers $S[1]$ and $S[2]$; $S[1]$ is in and $S[2]$ is out.
- $X = [1, 0, 1, 0, 0]$ represents a partial solution that incorporates our decisions on all numbers in $S[1..5]$; it happens to be a complete and valid solution.

For example. Let $S = [4, 7, 6, 3, 1]$ and $T = 10$.

- $X = []$ is the starting partial solution that does not incorporate any decisions about including numbers in $S[1..n]$; it represents an empty set.
- $X = [0]$ represents the partial solution that incorporates our decisions on number $S[1]$; the number is not included and X represents an empty set as well.
- $X = [1, 0]$ represents the partial solution that incorporates our decision on numbers $S[1]$ and $S[2]$; $S[1]$ is in and $S[2]$ is out.
- $X = [1, 0, 1, 0, 0]$ represents a partial solution that incorporates our decisions on all numbers in $S[1..5]$; it happens to be a complete and valid solution.
- $X = [1, 0, 0, 1, 1]$ represents a partial solution that incorporates our decisions on all numbers in $S[1..5]$; it happens to be a complete but invalid solution.

We will use the following backtracking strategy:

We will use the following backtracking strategy:

1. Start with $X = []$.

We will use the following backtracking strategy:

1. Start with $X = []$.
2. Given a partial solution $X[1..i-1]$ (of length $i-1$) that includes our choices which numbers in $S[1..i-1]$ to include, try including number $S[i]$ and extending the resulting partial potential solution $X[1..i]$

We will use the following backtracking strategy:

1. Start with $X = []$.
2. Given a partial solution $X[1..i-1]$ (of length $i-1$) that includes our choices which numbers in $S[1..i-1]$ to include, try including number $S[i]$ and extending the resulting partial potential solution $X[1..i]$
3. Given a partial solution $X[1..i-1]$ (of length $i-1$) that includes our choices which numbers in $S[1..i-1]$ to include, try excluding number $S[i]$ and extending the resulting partial potential solution $X[1..i]$

We will use the following backtracking strategy:

1. Start with $X = []$.
2. Given a partial solution $X[1..i-1]$ (of length $i-1$) that includes our choices which numbers in $S[1..i-1]$ to include, try including number $S[i]$ and extending the resulting partial potential solution $X[1..i]$
3. Given a partial solution $X[1..i-1]$ (of length $i-1$) that includes our choices which numbers in $S[1..i-1]$ to include, try excluding number $S[i]$ and extending the resulting partial potential solution $X[1..i]$
4. Backtrack if both choices fail to lead to a feasible solution.

Let us implement this strategy on input $S = [4, 7, 6, 3, 1]$ and $T = 10$.

```
SubsetSum(S[1..n], T, X[1..n], r)
  // extends a partial solution X[1..r-1]
  // of size r-1 to one of size r,
  // unless the partial solution
  // is a complete solution, in
  // which case it is output;
  // the initial call is SubsetSum(S, T, X, 1)
  // with X storing a partial solution of size 0

.
```

```
SubsetSum(S[1..n], T, X[1..n], r)
  if r = n+1 then
    output X
    return
  X[r] ← 0
  if Feasible(S, T, X, r) then
    SubsetSum(S, T, X, r+1)
  X[r] ← 1
  if Feasible(S, T, X, r) then
    SubsetSum(S, T, X, r+1)
```

When is partial solution $X[1..r]$ infeasible?

When is partial solution $X[1..r]$ infeasible?

We can clearly prune X if

$$\sum_{i=1}^{r} X[i]\dot{S}[i] > T$$

When is partial solution $X[1..r]$ infeasible?

We can clearly prune X if

$$\sum_{i=1}^{r} X[i]S[i] > T$$


```
Feasible(S[1..n], T, X[1..n], r)
  S1 ← ∑_{i=1}^{r} X[i]S[i]
  if S1 > T then
    return false
  return true
```

Furthermore, X can also be pruned if

$$\sum_{i=1}^{k} X[i]S[i] + \sum_{i=r+1}^{n} S[i] < T.$$

Furthermore, X can also be pruned if

$$\sum_{i=1}^{k} X[i]S[i] + \sum_{i=r+1}^{n} S[i] < T.$$


```
Feasible(S[1..n], T, X[1..n], r)
  S1 ← ∑_{i=1}^{k} X[i]S[i]
  if S1 > T then
    return false
  S2 ← ∑_{i=r+1}^{n} S[i]
  if S1 + S2 < T then
    return false
  return true
```

```
Solve(partial solution)
  if partial solution is a solution then
    // if goal is to find a solution
    output solution and quit
  for all possible extensions of partial solution
    if extension is feasible then
      Solve(extension)
```

```
Solve(partial solution)
  if partial solution is a solution then
    // if goal is to find all solutions
    output solution and return
  for all possible extensions of partial solution
    if extension is feasible then
      Solve(extension)
```

```
Solve(partial solution)
  if partial solution is a solution then
    // if goal is to find optimal solution
    keep solution if best so far and return
  for all possible extensions of partial solution
    if extension is feasible then
      Solve(extension)
```

- Backtracking is a very general-purpose algorithm design technique.

Backtracking is a very general-purpose algorithm design technique.

The actual implementation of a backtracking algorithm involves engineering decisions that are not typically made explicit in a more theoretical description of an algorithm.

Backtracking is a very general-purpose algorithm design technique.

The actual implementation of a backtracking algorithm involves engineering decisions that are not typically made explicit in a more theoretical description of an algorithm.

Algorithm engineering is an area of computer science that goes beyond the design and analysis of algorithms and includes their *implementation, optimization, profiling and experimental evaluation*.

Backtracking is a very general-purpose algorithm design technique.

The actual implementation of a backtracking algorithm involves engineering decisions that are not typically made explicit in a more theoretical description of an algorithm.

Algorithm engineering is an area of computer science that goes beyond the design and analysis of algorithms and includes their *implementation, optimization, profiling and experimental evaluation*.

To illustrate this, I develop next a backtracking solution for the Kattis problem Class Picture.