Algorithms

# Backtracking

Hee-Kap Ahn
Graduate School of Artificial Intelligence
Dept. Computer Science and Engineering
Pohang University of Science and Technology (POSTECH)

Textbooks: Algorithms by Jeff Erickson /Algorithms by Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani.

# Backtracking

**Construct a solution to a computational problem incrementally.**

- Compute candidates to the solution incrementally, one small piece at a time.
- For multiple alternatives to the solution,
  - evaluate every alternative recursively and,
  - abandon it (backtrack) immediately if it cannot belong to the solution.

Consider an instance of SAT consisting of 6 clauses from 4 variables $w, x, y, z$.

$$\phi(w,x,y,z) = (w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z}).$$

All assignments with $w = 0$ and $x = 1$ can be instantly eliminated.

$$\begin{tikzcd}
 & \boxed{(w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})} \arrow[dl, "w = 0"] \\
\boxed{(x \vee y \vee z)(\overline{x})(x \vee \overline{y})(y \vee \overline{z})} \arrow[dr, "x = 1"] & \\
 & \text{REJECT}
\end{tikzcd}$$

## Backtracking

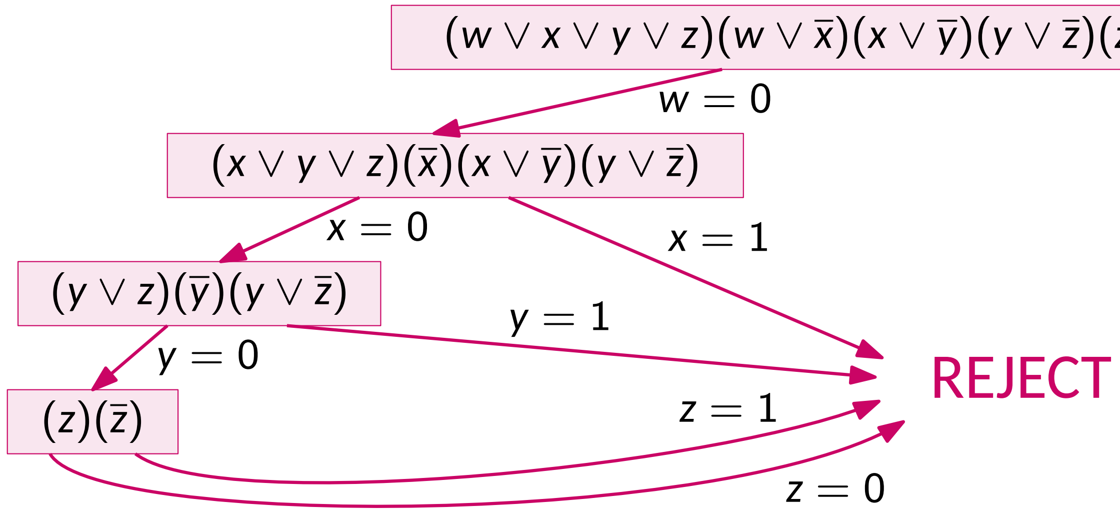
**Construct a solution to a computational problem incrementally.**

- Compute candidates to the solution incrementally, one small piece at a time.
- For multiple alternatives to the solution,
  - evaluate every alternative recursively and,
  - abandon it (backtrack) immediately if it cannot belong to the solution.

Consider an instance of SAT consisting of 6 clauses from 4 variables $w, x, y, z$.

$$\phi(w,x,y,z) = (w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z}).$$

All assignments with $w = 0$ and $x = 1$ can be instantly eliminated.



$(w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})$
$w = 0$
$(x \vee y \vee z)(\overline{x})(x \vee \overline{y})(y \vee \overline{z})$
$x = 0$
$x = 1$
$(y \vee z)(\overline{y})(y \vee \overline{z})$
$y = 1$
$y = 0$
REJECT
$(z)(\overline{z})$
$z = 1$
$z = 0$

## Backtracking

**Construct a solution to a computational problem incrementally.**

- Compute candidates to the solution incrementally, one small piece at a time.
- For multiple alternatives to the solution,
  - evaluate every alternative recursively and,
  - abandon it (backtrack) immediately if it cannot belong to the solution.

Consider an instance of SAT consisting of 6 clauses from 4 variables $w, x, y, z$.

$$\phi(w,x,y,z) = (w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z}).$$

All assignments with $w = 0$ and $x = 1$ can be instantly eliminated.

$$\begin{tikzcd}
 & & \boxed{(w \vee x \vee y \vee z)(w \vee \overline{x})(x \vee \overline{y})(y \vee \overline{z})(z \vee \overline{w})(\overline{w} \vee \overline{z})} \arrow[dl, "w=0"] \arrow[dr, "w=1"] & \\
 & \boxed{(x \vee y \vee z)(\overline{x})(x \vee \overline{y})(y \vee \overline{z})} \arrow[dl, "x=0"] \arrow[ddr, "x=1"] & & \boxed{(x \vee \overline{y})(y \vee \overline{z})(z)(\overline{z})} \arrow[ddl, bend right, "z=0"'] \arrow[ddl, bend left, "z=1"] \\
\boxed{(y \vee z)(\overline{y})(y \vee \overline{z})} \arrow[d, "y=0"] \arrow[drr, "y=1"] & & & \\
\boxed{(z)(\overline{z})} \arrow[rr, bend right, "z=1"] \arrow[rr, bend right=40, "z=0"'] & & \text{REJECT} &
\end{tikzcd}$$

# Backtracking

Backtracking explores the space of assignments, growing the tree only at nodes where there is uncertainty about the output.

But, which subproblem to expand next? Which branching variable to use?

> Choose the subproblem that contains the shortest clause and then branch on a variable in that clause.

The backtracking procedure has the following format.

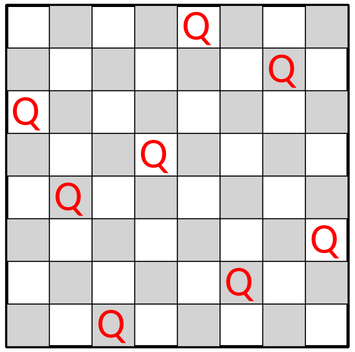
```
Start with some problem $P_0$
Let $\mathcal{S} = \{P_0\}$, the set of active subproblems.
while $\mathcal{S}$ is nonempty:
    choose a subproblem $P \in \mathcal{S}$ and remove it from $\mathcal{S}$
    expand it into smaller subproblems $P_1, P_2, \ldots, P_k$
    for each $P_i$:
        if test($P_i$) succeeds: halt and announce this solution
        if test($P_i$) fails: discard $P_i$
        Otherwise: add $P_i$ to $\mathcal{S}$
Announce that there is no solution
```

recursive calls to subproblems

Uncertain

# *n* Queens Problem

**Problem.** Place $n$ queens on an $n \times n$ chessboard so that no two queens are attacking each other (no two queens are in the same row, the same column, or the same diagonal).



**Strategy.**
Place queens one row at a time, starting with the top row.
At $r$-th iteration, place the $r$-th queen by trying all squares in row $r$ from left to right.

- If a particular square is attacked by an earlier queen, ignore that square.
- Otherwise, place a queen on that square tentatively, and recursively search for consistent placements of the queens in later rows.

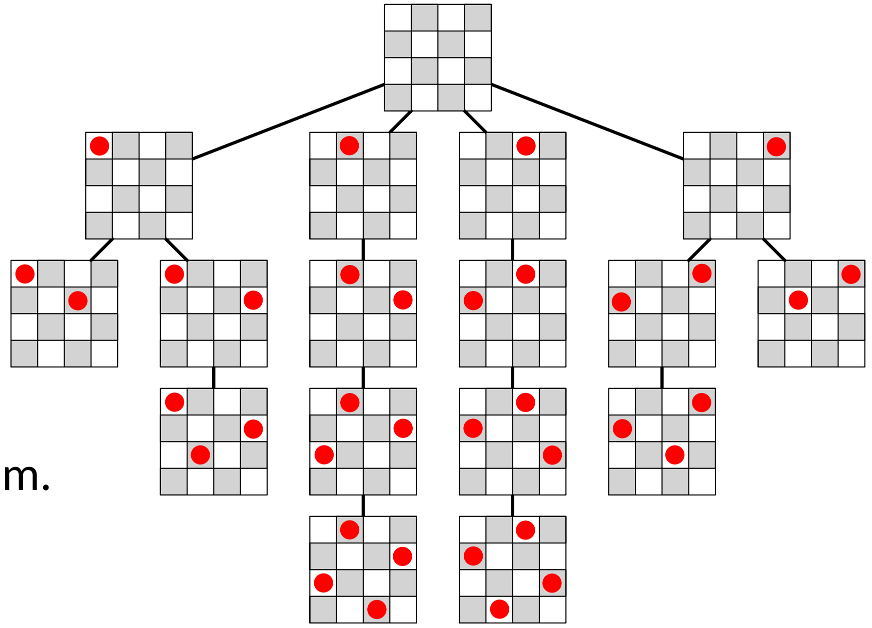


Illustration using a recursion tree.
- Each node corresponding to a recursive subproblem.
- Each edge corresponding to a recursive call.

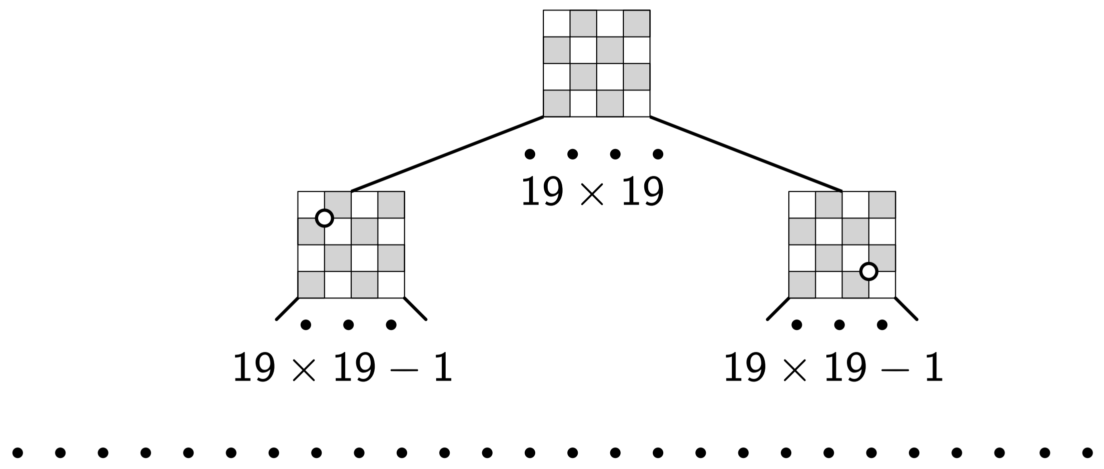
## $n$ Queens Problem

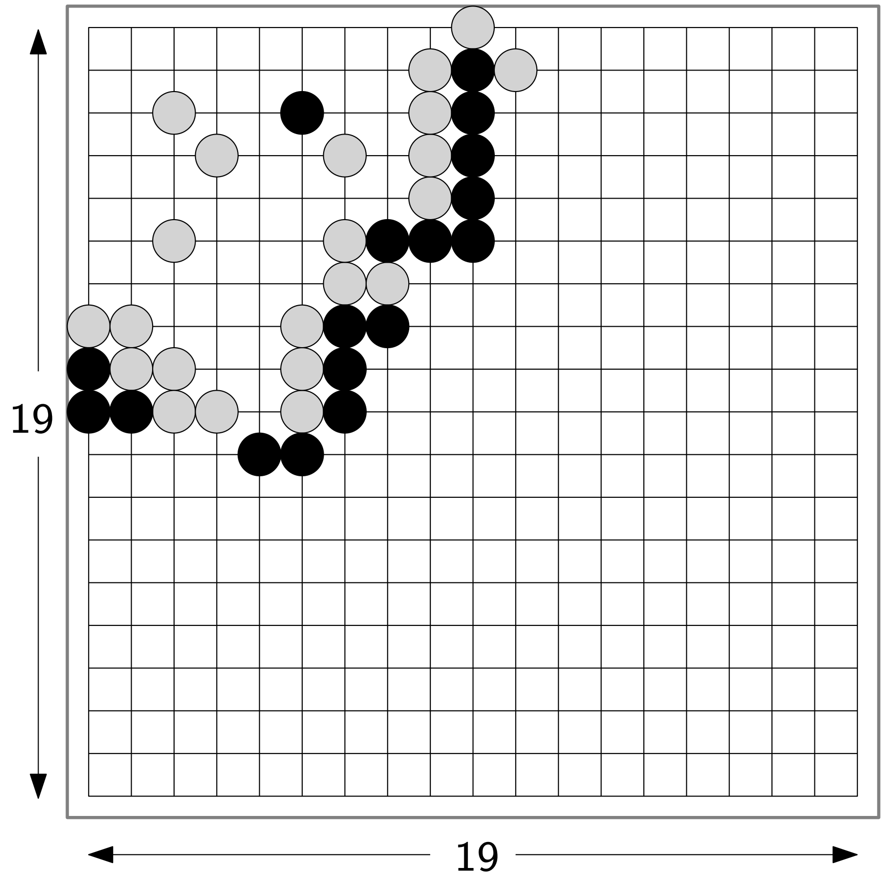
```
PlaceQueens($Q[1:n], r$)
if $r = n + 1$ then
    print $Q[1:n]$
else
    for $j \leftarrow 1$ to $n$ do
        legal $\leftarrow$ True
        for $i \leftarrow 1$ to $r - 1$ do
            if $(Q[i] = j)$ or $(Q[i] = j + r - i)$ or $(Q[i] = j - r + i)$ then
                legal $\leftarrow$ False
        if legal then
            $Q[r] \leftarrow j$
            PlaceQueens($Q[1:n], r + 1$)
```

# Game Trees

**Two-player games.** 바둑(Baduk, Go, Weiqi, Weichi, 棋, 圍棋, 碁).

- An abstract strategy board game for two players in which the aim is to surround more territory than the opponent.
- Standard boards with a $19 \times 19$ grid.
- $\#$. legal board positions $\approx 2.1 \times 10^{170}$.



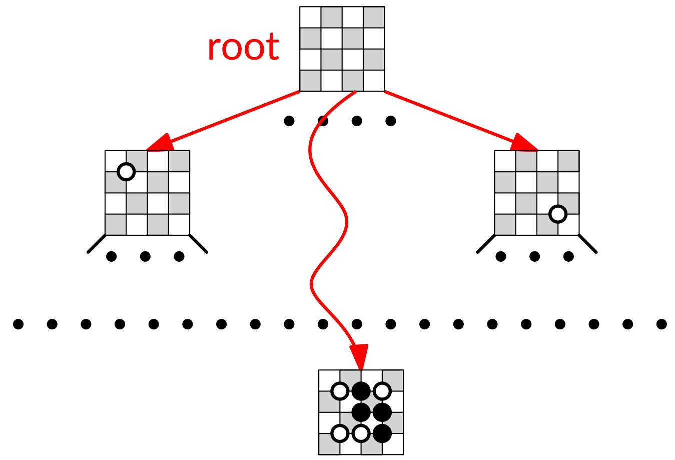


# Game Trees

**Backtracking algorithm** for any two-player game without randomness or hidden information that ends after a finite number of moves.

**State** $=$ locations of all the pieces $+$ identity of the current player.
**Game tree** $=$ graph consisting of states (nodes) and edges connecting them.

- Edge from state $x$ to state $y$ iff the current player in state $x$ can legally move to state $y$.
- Root is the initial position of the game.
- Every path from root to a leaf is a complete game.



A game state is

- **good** if either the current player has already won, or if the current player can move to a bad state for the opponent.
- **bad** if either the current player has already lost, or if every available move leads to a good state for the opponent.

A nonleaf node is **good** if it has at least one bad child (for the opponent), and **bad** if all its children are good (for the opponent).

## Game Trees

By induction, a player in a good state can win the game even if the opponent plays perfectly. A player in a bad state can win only if the opponent makes a mistake.

A recursive backtracking algorithm.

```
PlayAnyGame($X$, player)
if player has already won in state $X$ then
    return Good
if player has already lost in state $X$ then
    return Bad
for all legal moves $X \longrightarrow Y$ do
    if PlayAnyGame($Y$, $\neg$player)=Bad then
        return Good
return Bad
```

Most games have an enormous number of states. Instead of traversing the entire game tree, game programs employ other heuristics.

- **Prune** the tree by ignoring (skipping) states that are obviously good or bad, or at least better or worse than other states.
- **Cut off** the tree at a certain depth (or ply) and use more efficient heuristics to evaluate the leaves.

## Subset Sum

Given a set $X$ of positive integers and a target integer $T$,
decide (Yes/No) if there is a subset of elements in $X$ that add up to $T$.

For $X = \{8, 6, 7, 5, 3, 10, 9\}$,
$T = 15$ / Yes. $T = 54$ / No. $T = 0$ / Yes. $T = -1$ / No.

**Backtracking.** For an arbitrary element $x \in X$, there is a subset of $X$ that sums to $T$ if and only if

- there is a subset of $X$ that includes $x$ and whose sum is $T$, or
- there is a subset of $X$ that excludes $x$ and whose sum is $T$.

```
SubsetSum($X, T$)
if $T = 0$ then
    return True
else if $T < 0$ or $X = \emptyset$ then
    return False
else
    $x \leftarrow$ any element of $X$
    with $\leftarrow$ SubsetSum($X \setminus \{x\}, T - x$)
    wout $\leftarrow$ SubsetSum($X \setminus \{x\}, T$)
    return (with $\vee$ wout)
```

When $X$ is given as an array,
does any subset of $X[1 \dots i]$ sum to $T$?

```
SubsetSum($X, i, T$)
if $T = 0$ then
    return True
else if $T < 0$ or $i = 0$ then
    return False
else
    with $\leftarrow$ SubsetSum($X, i - 1, T - X[i]$)
    wout $\leftarrow$ SubsetSum($X, i - 1, T$)
    return (with $\vee$ wout)
```

## Subset Sum

**Correctness.** By induction.

- If $T = 0$, $T < 0$, or $X = \emptyset$, the output is correct.
- If there is a subset that sums to $T$, then either it contains $X[n]$ or it doesn't. Each of those possibilities is checked recursively. Thus the output is correct.

**Analysis.** When $X$ is given as an array,

- $T(n) \leq 2T(n-1) + O(1)$.
- In the worst case, the recursion tree is a complete binary tree with depth $n$.
  The algorithm considers all $2^n$ subsets of $X$.
- $T(n) = O(2^n)$.

**Variants.** Construct a subset of $X$ that sums to $T$ if one exists, or return an error value if no such subset exists.

- Count subsets that sum to $T$, or
- Choose the best subset (according to some other criterion) that sums to $T$.

```
ConstructSubset($X, i, T$)
if $T = 0$ then
    return $\emptyset$
else if $T < 0$ or $i = 0$ then
    return None
$Y \leftarrow$ConstructSubset($X, i-1, T$)
if $Y \neq$ None then
    return $Y$
$Y \leftarrow$ConstructSubset($X, i-1, T-X[i]$)
if $Y \neq$ None then
    return $Y \cup \{X[i]\}$
return None
```

## General Pattern

Backtracking builds a recursively defined structure satisfying certain constraints by making a *sequence of decisions*. Often this goal structure is a sequence.

- structure: subset $Y$ of $X[1 \ldots i]$
- constraints: $Y$ sums to $T$
- decision: $Y$ or `None`

Each recursive call makes **exactly one decision** which is consistent with all previous decisions.
Each recursive call requires

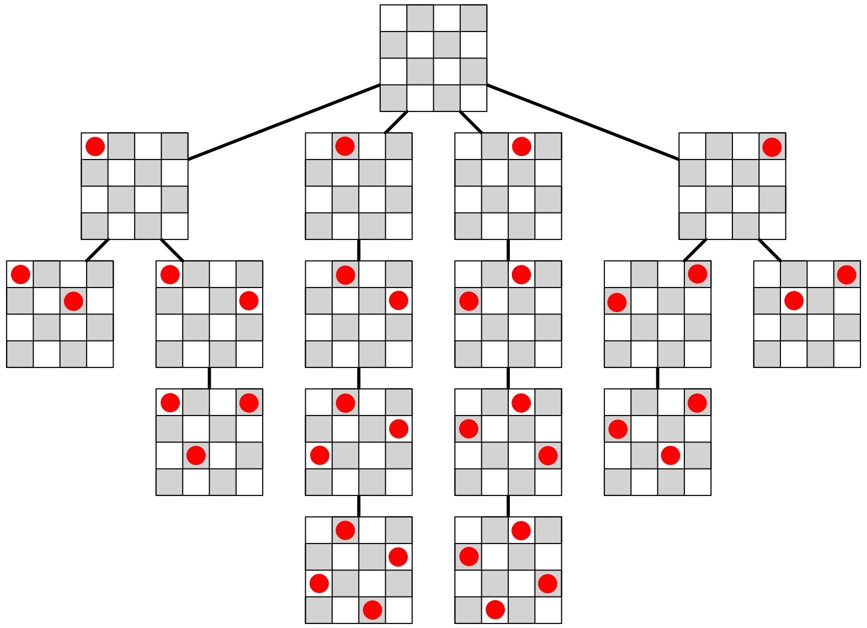
- the portion of input data we have not yet processed, and
- a *suitable* summary of past decisions.

- consistent: index $i$ and (remaining) $T$ value
- portion: $X[1 \ldots i]$
- summary: previously chosen elements and their sum

## General Pattern

Designing new recursive backtracking algorithms,

- Figure out what information about past decisions we will need in the **middle** of the algorithm.
- Solve the recursive problem by **recursive brute force** by trying **all possibilities for the next decision** that are **consistent** with past decisions.



# Text Segmentation

Given a string of letters, break it into its individual constituent words.

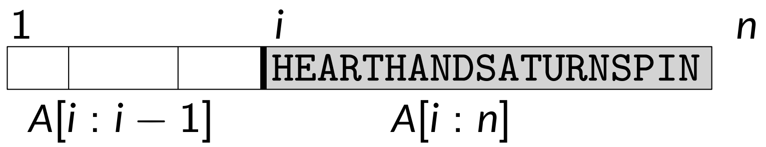
```
POSTECHISTHEBESTUNIVERSITYINTHEWORLD
```

```
POSTECH IS THE BEST UNIVERSITY IN THE WORLD
```

Input string of length $n$ is stored in an array $A[1:n]$.

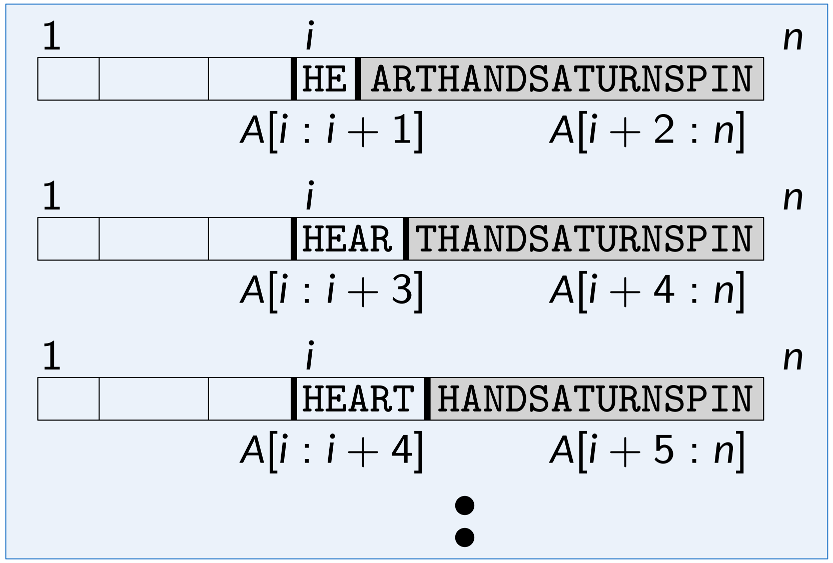
Assume that we have processed a **prefix** $A[1:i-1]$ of $A$ into words, maybe multiple solutions.

Now solve the problem for $A[i:n]$.



Given an index $i$,
find a segmentation of the **suffix** $A[i:n]$.

There are **multiple possibilities** for $A[i:n]$:



## Text Segmentation

$$\text{Splittable}(i) = \begin{cases} \texttt{True} & \text{if } i > n \\ \bigvee_{j=i}^{n} \left(\text{IsWord}(i,j) \wedge \text{Splittable}(j+1)\right) & \text{otherwise} \end{cases}$$

- IsWord($i, j$) = `True` iff $A[i : j]$ is a word.
- Splittable($i$)=`True` iff $A[i : n]$ can be split into words.

IsWord($1, n$)=`True` iff $A[1 : n]$ is a single word.
Splittable(1)=`True` iff the $A[1 : n]$ can be segmented.

```
Splittable(i)
if $i > n$ then
    return True
for $j \leftarrow i$ to $n$ do
    if IsWord(i, j) then
        if Splittable(j + 1) then
            return True
return False
```

**Analysis.** Splittable calls IsWord on every prefix, and possibly calls itself recursively on every suffix of the output string. #. of calls to IsWord is

$$T(n) = \sum_{i=0}^{n-1} T(i) + cn$$

Thus, $T(n) = 2T(n-1) + c = O(2^n)$.

$$T(n-1) = \sum_{i=0}^{n-2} T(i) + c(n-1)$$

$$\Longrightarrow T(n) - T(n-1) = T(n-1) + c$$

## Text Segmentation

Find the best segmentation according to some criterion (Score function).

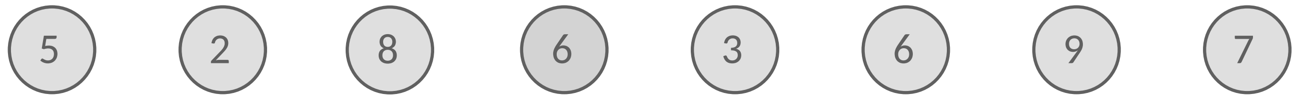
$$\text{MaxScore}(i) = \begin{cases} 0 & \text{if } i > n \\ \max_{i \le j \le n} \big(\text{Score}(i,j) + \text{MaxScore}(j+1)\big) & \text{otherwise} \end{cases}$$

Find a segmentation that maximizes the sum of the scores of the segments.

## Longest Increasing Subsequences

**Input** : a sequence $S$ of numbers $a_1, \ldots, a_n$.
**Goal** : find a longest increasing subsequence of $S$.

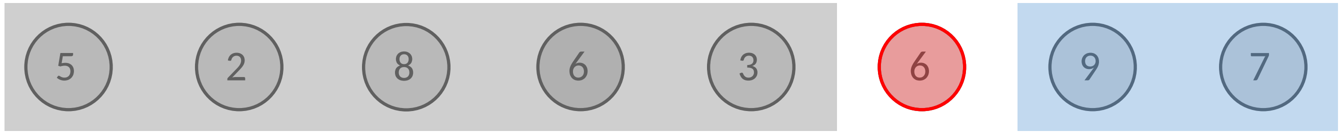
A subsequence of $S$ is another sequence obtained from $S$ by deleting zero or more elements, without changing the order of the remaining elements.



# Longest Increasing Subsequences

**Input** : a sequence $S$ of numbers $a_1, \dots, a_n$.
**Goal** : find a longest increasing subsequence of $S$.

A subsequence of $S$ is another sequence obtained from $S$ by deleting zero or more elements, without changing the order of the remaining elements.



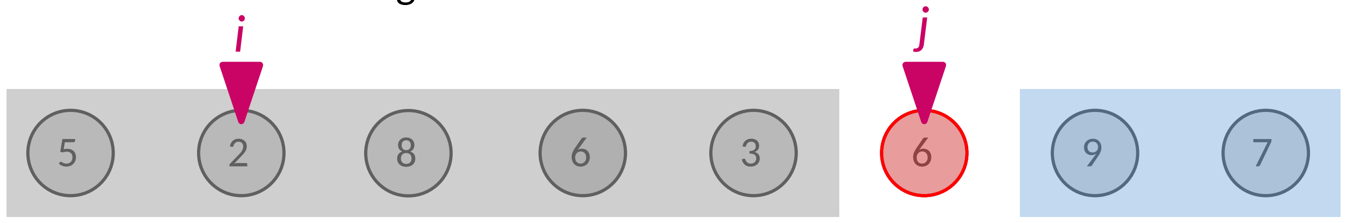
Backtracking algorithm has two choices:

- Include 6 tentatively, and proceed to the rest of the decisions. (take)
- Exclude 6 tentatively, and proceed to the rest of the decisions. (skip)

# Longest Increasing Subsequences

**Input** : a sequence $S$ of numbers $a_1, \dots, a_n$.
**Goal** : find a longest increasing subsequence of $S$.

A subsequence of $S$ is another sequence obtained from $S$ by deleting zero or more elements, without changing the order of the remaining elements.



Backtracking algorithm has two choices:

- Include 6 tentatively, and proceed to the rest of the decisions. (take)
- Exclude 6 tentatively, and proceed to the rest of the decisions. (skip)

To include 6, we need to remember the last number selected in the past.

> Given two indices $i$ and $j$ with $i < j$,
> find LIS of $A[j:n]$ in which every element is larger than $A[i]$. (consistency)

$$\text{LIS}(i,j) = \begin{cases} 0 & \text{if } i > n \\ \text{LIS}(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{\underbrace{\text{LIS}(i, j+1)}_{\text{skip}}, 1 + \underbrace{\text{LIS}(j, j+1)}_{\text{take}}\} & \text{otherwise} \end{cases}$$

# Longest Increasing Subsequences

Another backtracking algorithm.

Given an index $i$, find LIS of $A[i:n]$ that begins with $A[i]$.

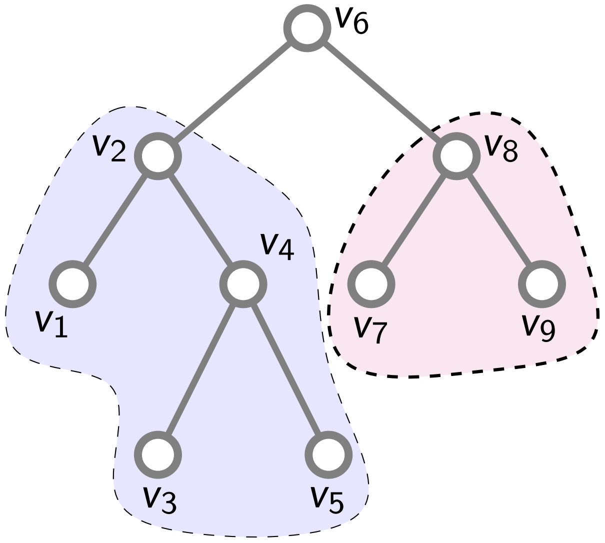
$$\text{LISfirst}(i) = 1 + \max\{\text{LISfirst}(j) \mid j > i \text{ and } \underbrace{A[j] > A[i]}_{\text{(consistency)}}\}$$

```
LISfirst(i)
best ← 0
for $j ← i + 1$ to $n$ do
    if $A[j] > A[i]$ then
        best ← max{best, LISfirst($j$)}
return 1 + best
```

# Optimal Binary Search Trees

**Input** : $n$ keys $v_1, \dots, v_n$ and their frequencies $f[1], \dots, f[n]$ of searches.
**Goal** : binary search tree of the keys with the minimum total cost of the searches.

$$\text{Cost}(T, f[1 \dots n]) = \sum_{i=1}^{n} f[i] \cdot \text{depth}(T, v_i)$$



If $f[i] > f[j]$, $\text{depth}(v_i) \leq \text{depth}(v_j)$ in any optimal binary search tree.

Suppose $v_r$ is the root of the tree. Then

$$\text{Cost}(T, f[1 \dots n]) = \sum_{i=1}^{n} f[i] + \boxed{\sum_{i=1}^{r-1} f[i] \cdot \text{depth}(\text{left}(T), v_i) \ \text{(consistency)}} + \boxed{\sum_{i=r+1}^{n} f[i] \cdot \text{depth}(\text{right}(T), v_i) \ \text{(consistency)}}$$

$$\text{OPT}(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \leq r \leq k} \{\text{OPT}(i, r-1) + \text{OPT}(r+1, k)\} & \text{otherwise} \end{cases}$$

## Optimal Binary Search Trees

$$\mathrm{OPT}(i,k) = \begin{cases} 0 & \text{if } i > k \\ \sum_{j=i}^{k} f[j] + \min_{i \le r \le k}\{\mathrm{OPT}(i, r-1) + \mathrm{OPT}(r+1, k)\} & \text{otherwise} \end{cases}$$

A recursive backtracking algorithm to compute $\mathrm{OPT}(1,n)$ takes time

$$T(n) = \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big) + O(n).$$

$$\begin{aligned} T(n) &= 2\sum_{t=0}^{n-1} T(t) + cn \\ T(n-1) &= 2\sum_{t=0}^{n-2} T(t) + c(n-1) \\ T(n) - T(n-1) &= 2T(n-1) + c \\ T(n) &= 3T(n-1) + c = O(3^n) \end{aligned}$$

#. binary search trees with $n$ vertices is

$$N(n) = \sum_{r=1}^{n-1} \big(N(r-1)\cdot N(n-r)\big) = \Theta(4^n/\sqrt{n})$$