Algorithms Week 4

Ljubomir Perković, DePaul University

Backtracking

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,

Backtracking

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,

Backtracking

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.

Text Segmentation

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Text Segmentation

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord(w)* that returns *True* if sequence of characters w is a word, *False* otherwise.

Text Segmentation

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord(w)* that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord(w)* is True if w is a word in English, then:

Text Segmentation

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord(w)* that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord(w)* is True if w is a word in English, then:

- Output *True* because
  BOTH·EARTH·AND·SATURN·SPIN
  is a valid segmentation of A

Text Segmentation

Input: A sequence of characters stored in $A[1..n]$.

Output: True if A can be segmented into a sequence of words, False otherwise.

Given: Function *IsWord(w)* that returns *True* if sequence of characters w is a word, *False* otherwise.

Example: If sequence A consists of characters

BOTHEARTHANDSATURNSPIN

and *IsWord(w)* is True if w is a word in English, then:

- Output *True* because
  BOTH·EARTH·AND·SATURN·SPIN
  is a valid segmentation of A
- By the way,
  BOT·HEART·HANDS·AT·URNS·PIN
  is another valid segmentation of A.

Backtracking algorithm idea

We consider a process that consumes the input characters in order from left to right

Backtracking algorithm idea

We consider a process that consumes the input characters in order from left to right and produces the output words in order from left to right.

Backtracking algorithm idea

We consider a process that consumes the input characters in order from left to right and produces the output words in order from left to right.

So, at any point in time of the process we might have the following situation:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word.

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Which option do we choose?

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Which option do we choose? We try all of them, using backtracking and letting the recursion fairy do the work.

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Which option do we choose? We try all of them, using backtracking and letting the recursion fairy do the work.

If any of the recursive calls returns True, we return True.

Backtracking algorithm idea

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

We need to decide on the next word. Options:

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEARTH	ANDSATURNSPIN

Which option do we choose? We try all of them, using backtracking and letting the recursion fairy do the work.

If any of the recursive calls returns True, we return True.
If none do, we return False.

Backtracking algorithm strategy

We start with:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

Backtracking algorithm strategy

We start with:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

We tentively accept HE as the next word and let the recursion fairy make the rest of the decisions:

BLUE	**STEM**	**UNIT**	**ROBOT**	**HE**	ARTHANDSATURNSPIN

Backtracking algorithm strategy

We start with:

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

We then tentively accept HEAR as the next word and let the recursion fairy make the rest of the decisions:

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

Backtracking algorithm strategy

We start with:

BLUE	STEM	UNIT	ROBOT	HEARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HE	ARTHANDSATURNSPIN

BLUE	STEM	UNIT	ROBOT	HEAR	THANDSATURNSPIN

We then tentively accept HEART as the next word and let the recursion fairy make the rest of the decisions:

BLUE	STEM	UNIT	ROBOT	HEART	HANDSATURNSPIN

Backtracking algorithm strategy

We start with:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

BLUE	**STEM**	**UNIT**	**ROBOT**	**HE**	ARTHANDSATURNSPIN

BLUE	**STEM**	**UNIT**	**ROBOT**	**HEAR**	THANDSATURNSPIN

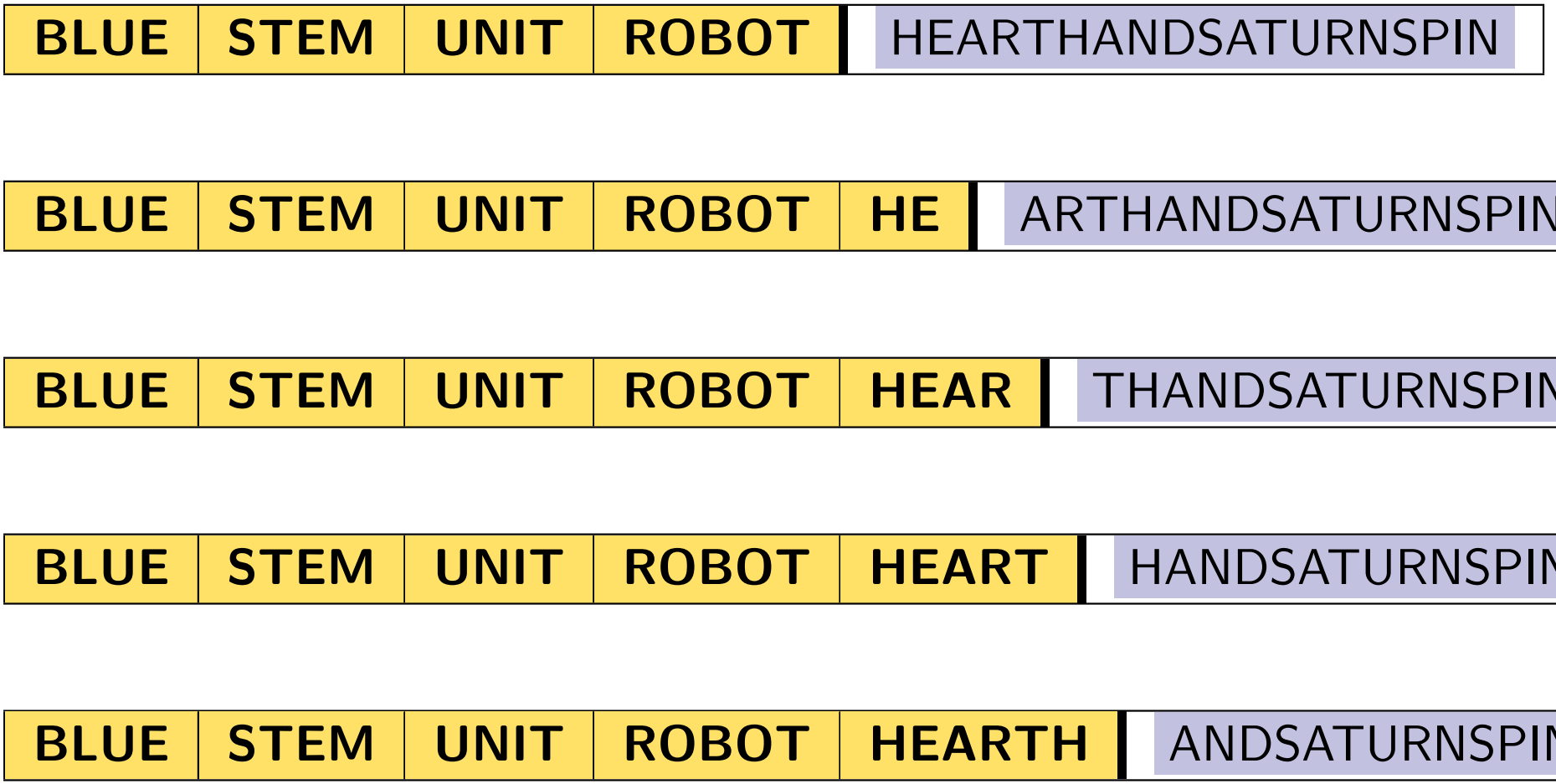
BLUE	**STEM**	**UNIT**	**ROBOT**	**HEART**	HANDSATURNSPIN

We then tentively accept HEARTH as the next word and let the recursion fairy make the rest of the decisions:

BLUE	**STEM**	**UNIT**	**ROBOT**	**HEARTH**	ANDSATURNSPIN

Backtracking algorithm strategy

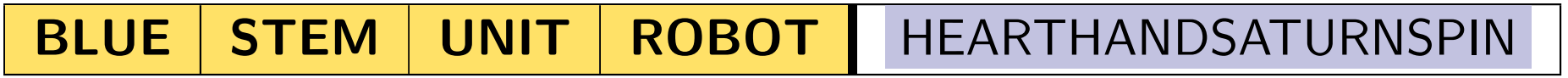
We start with:

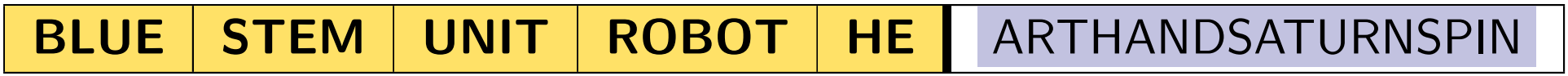


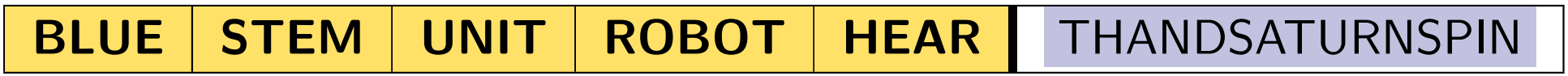
If the recursion fairy reports success on any of these, we report success.

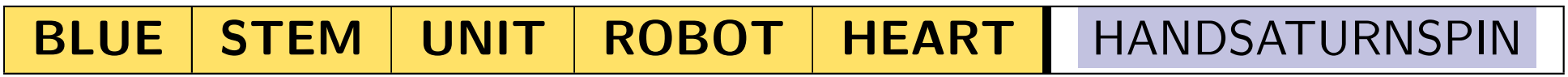
Backtracking algorithm strategy

We start with:









If, on the other hand, the recursion fairy does not report success then we report failure.

Backtracking algorithm strategy, simplified

Past decisions do not affect the choices we have available now:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN

Backtracking algorithm strategy, simplified

Past decisions do not affect the choices we have available now:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN

So the recursive process can be simplified by ignoring the past:

HEARTHANDSATURNSPIN

Backtracking algorithm strategy, simplified

Past decisions do not affect the choices we have available now:

BLUE	**STEM**	**UNIT**	**ROBOT**	HEARTHANDSATURNSPIN

BLUEST	**EMU**	**NITRO**	**BOT**	HEARTHANDSATURNSPIN

So the recursive process can be simplified by ignoring the past:

HEARTHANDSATURNSPIN

Backtracking stategy: Select the first output word, and recursively segment the rest of the input string.

Base case?

A recursive algorithm needs a base case.

Base case?

A recursive algorithm needs a base case.

We stop when the input sequence is of length 0.

Base case?

A recursive algorithm needs a base case.

We stop when the input sequence is of length 0. What is returned then? Success or Failure?

Segmentation backtracking algorithm

```
Splittable(A[1 .. n]):
  if n = 0
    return True
  for i ← 1 to n
    if IsWord(A[1 .. i]) and Splittable(A[i + 1 .. n])
        return True
  return False
```

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

It is better to treat the original input array as a global variable and reformulate the problem and the algorithm in terms of array indices.

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

It is better to treat the original input array as a global variable and reformulate the problem and the algorithm in terms of array indices.

For the string segmentation problem, the argument of any recursive call is always a suffix $A[i..n]$ of the original input array $A[1..n]$.

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

It is better to treat the original input array as a global variable and reformulate the problem and the algorithm in terms of array indices.

For the string segmentation problem, the argument of any recursive call is always a suffix $A[i..n]$ of the original input array $A[1..n]$.

So if we treat the input array $A[1..n]$ as a global variable, we can reformulate our recursive problem as follows:

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

It is better to treat the original input array as a global variable and reformulate the problem and the algorithm in terms of array indices.

For the string segmentation problem, the argument of any recursive call is always a suffix $A[i..n]$ of the original input array $A[1..n]$.

So if we treat the input array $A[1..n]$ as a global variable, we can reformulate our recursive problem as follows:

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

Index formulation

Passing arrays as input parameters to recursive functions is inefficient.

It is better to treat the original input array as a global variable and reformulate the problem and the algorithm in terms of array indices.

For the string segmentation problem, the argument of any recursive call is always a suffix $A[i..n]$ of the original input array $A[1..n]$.

So if we treat the input array $A[1..n]$ as a global variable, we can reformulate our recursive problem as follows:

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

Note that function *IsWord(i, j)* now takes two indices of array A.

Segmentation backtracking algorithm, v2.0

```
// Is the suffix A[i..n] Splittable?
Splittable(i):
  if i > n
    return True
  for j ← i to n
    if IsWord(i, j) and Splittable(j + 1)
        return True
  return False
```

Longest Increasing Subsequence

For any sequence S,

2	6	1	4	2	4	2	9	5	3	5	7	8	3

Longest Increasing Subsequence

For any sequence 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 Subsequence

For any sequence 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.

Problem: Given a sequence of integers S find the Longest Increasing Subsequence (LIS) of S.

Longest Increasing Subsequence

For any sequence 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.

2	6	**1**	4	**2**	**4**	2	9	**5**	3	5	**7**	**8**	3

Problem: Given a sequence of integers S find the Longest Increasing Subsequence (LIS) of S.

Input: Integer array $A[1..n]$.

Output: Longest possible sequence of indices $1 \leq i_1 < i_2 < \cdots < i_l \leq n$ such that $A[i_1] < A[i_2] < \cdots < A[i_l]$.

Example: See above.

Backtracking approach

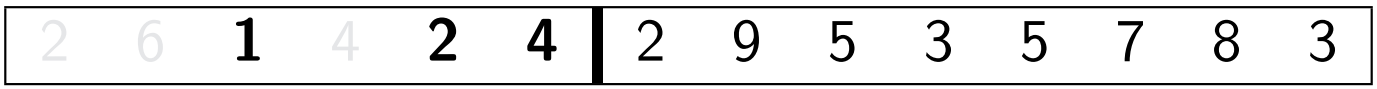
We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

Backtracking approach

We use the same approach as we used for Subset Sum:

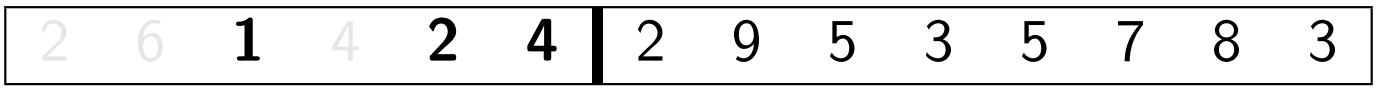
We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.



Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

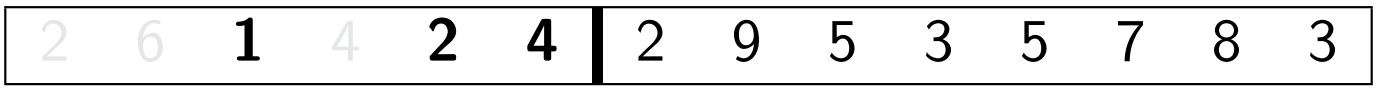


Do we include $A[7] = 2$?

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

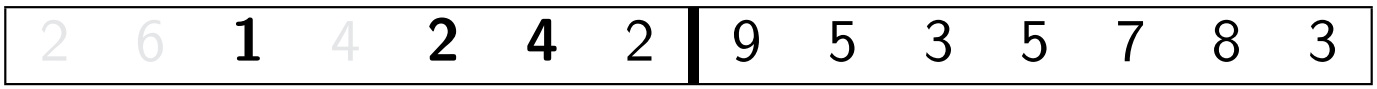


Do we include $A[7] = 2$? No, we can't.

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

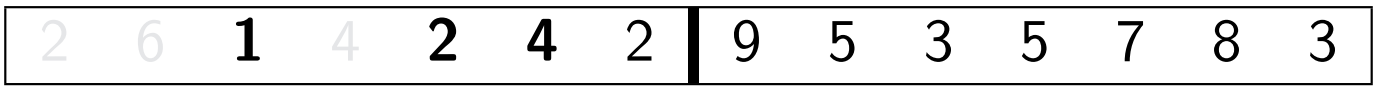


Do we include $A[8] = 9$?

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.



Do we include $A[8] = 9$? We can, but should we?

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

2	6	**1**	4	**2**	**4**	2	9	5	3	5	7	8	3

Do we include $A[8] = 9$? We can, but should we?

Instead of trying to be smart, we apply the backtracking strategy:

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.

Do we include $A[8] = 9$? We can, but should we?

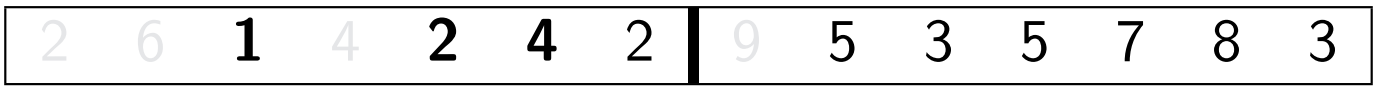
Instead of trying to be smart, we apply the backtracking strategy:

- We first tentatively include 9, and let the recursion fairy make the rest of the decisions.

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.



Do we include $A[8] = 9$? We can, but should we?

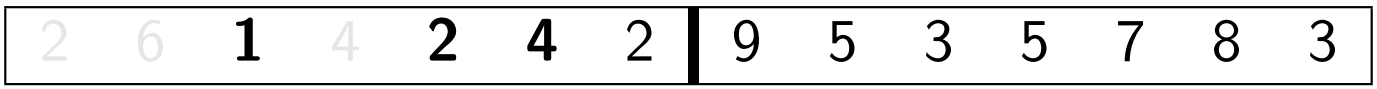
Instead of trying to be smart, we apply the backtracking strategy:

- We first tentatively include 9, and let the recursion fairy make the rest of the decisions.
- We then tentatively exclude 9, and let the recursion fairy make the rest of the decisions.

Backtracking approach

We use the same approach as we used for Subset Sum:

We decide, for each index j in order from left to right, whether or not to include $A[j]$ in the subsequence.



Do we include $A[8] = 9$? We can, but should we?

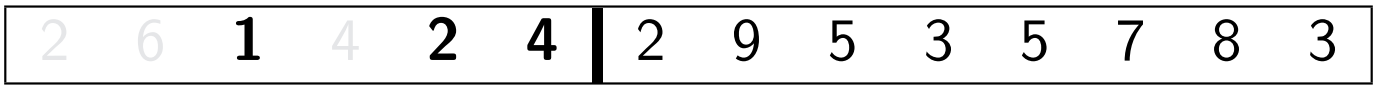
Instead of trying to be smart, we apply the backtracking strategy:

- We first tentatively include 9, and let the recursion fairy make the rest of the decisions.
- We then tentatively exclude 9, and let the recursion fairy make the rest of the decisions.

Whichever choice leads to a longer increasing subsequence is the right one.

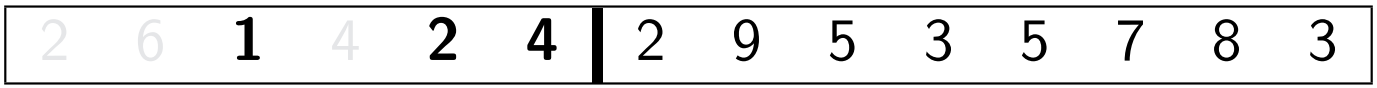
Past decisions?

How do previous decisions affect future choices?



Past decisions?

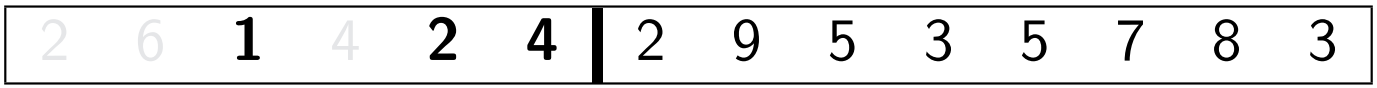
How do previous decisions affect future choices?



In other words, what do we need to remember about our past decisions?

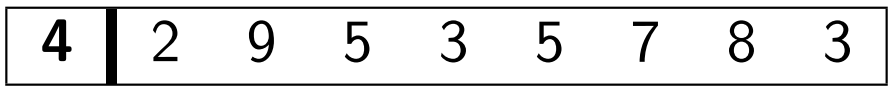
Past decisions?

How do previous decisions affect future choices?



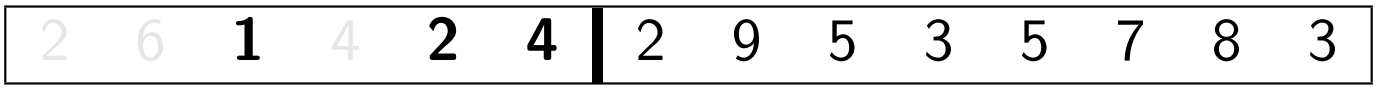
In other words, what do we need to remember about our past decisions?

The only information we need about the past is the last number selected so far:



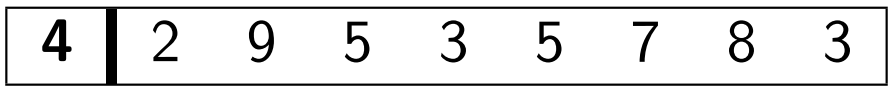
Past decisions?

How do previous decisions affect future choices?



In other words, what do we need to remember about our past decisions?

The only information we need about the past is the last number selected so far:



The problem can be restated as:

Given an integer *prev* and array $A[1..n]$, find the Longest Increasing Subsequence of A in which every element is larger than *prev*.

Backtracking algorithm LISbigger

This function returns the length of the Longest Increasing Subsequence of $A[1..n]$:

```
LISbigger(prev, A[1 .. n]):
  if n = 0
    return 0
  else if A[1] ≤ prev
    return LISbigger(prev, A[2 .. n])
  else
    skip ← LISbigger(prev, A[2 .. n])
    take ← LISbigger(A[1], A[2 .. n]) + 1
    return max{skip, take}
```

Index formulation

Recall that passing arrays as input parameters to recursive functions is expensive.

Index formulation

Recall that passing arrays as input parameters to recursive functions is expensive.

We will restate the algorithm in terms of array indices, assuming that the array $A[1..n]$ is a global variable.

Index formulation

Recall that passing arrays as input parameters to recursive functions is expensive.

We will restate the algorithm in terms of array indices, assuming that the array $A[1..n]$ is a global variable.

The integer *prev* is *typically* an array element $A[i]$, and the remaining array is always a suffix $A[j..n]$ of the original input array.

Index formulation

Recall that passing arrays as input parameters to recursive functions is expensive.

We will restate the algorithm in terms of array indices, assuming that the array $A[1..n]$ is a global variable.

The integer *prev* is *typically* an array element $A[i]$, and the remaining array is always a suffix $A[j..n]$ of the original input array.

The problem to be solved is then:

Given two indices i and j, where $i < j$, find the Longest Increasing Subsequence of $A[j..n]$ in which every element is larger than $A[i]$.

LISbigger v2.0

```
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j + 1)
  else
    skip ← LISbigger(i, j + 1)
    take ← LISbigger(j, j + 1) + 1
    return max{skip, take}
```

LISbigger v2.0

```
LISbigger(i, j):
  if j > n
    return 0
  else if A[i] ≥ A[j]
    return LISbigger(i, j + 1)
  else
    skip ← LISbigger(i, j + 1)
    take ← LISbigger(j, j + 1) + 1
    return max{skip, take}
```

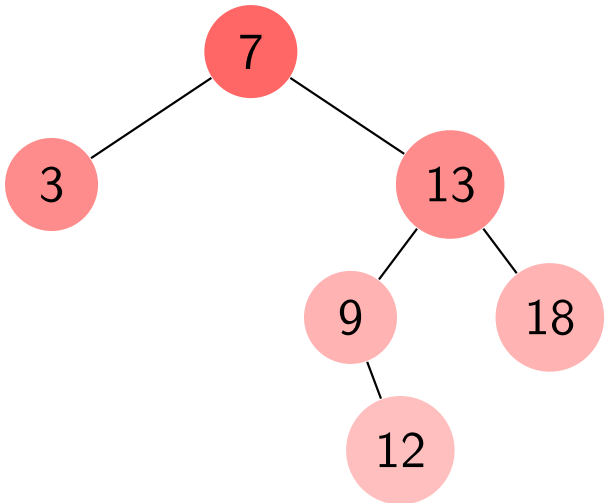
What is the initial call on $A[1..n]$? There is not index i initially...

LIS backtracking algorithm wrapper function

```
LIS(A[1 .. n]):
  A[0] ← −∞
  return LISbigger(0, 1)
```

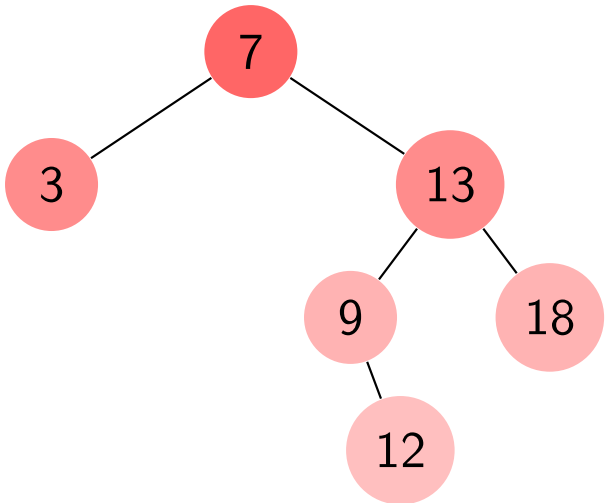
Optimal Binary Search Trees

The running time for a successful search in a binary search tree is proportional to the depth of the target node.



Optimal Binary Search Trees

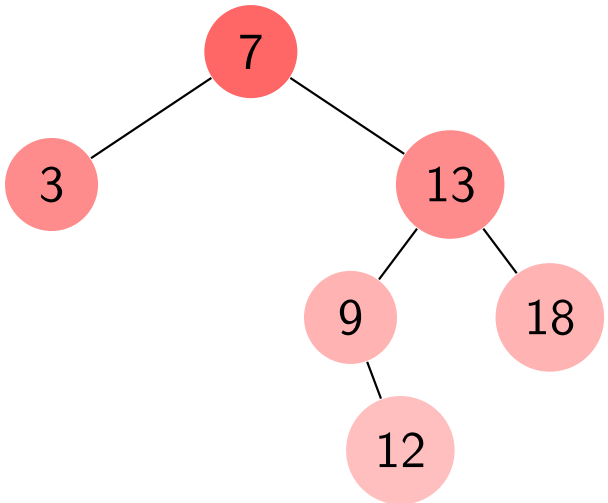
The running time for a successful search in a binary search tree is proportional to the depth of the target node.

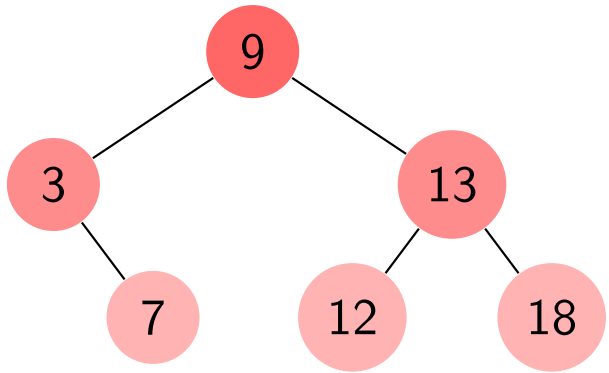


- The worst-case search time is proportional to the depth of the tree.

Optimal Binary Search Trees

The running time for a successful search in a binary search tree is proportional to the depth of the target node.





- The worst-case search time is proportional to the depth of the tree.
- To minimize the worst-case search time, the tree should be balanced.

Optimal Binary Search Trees

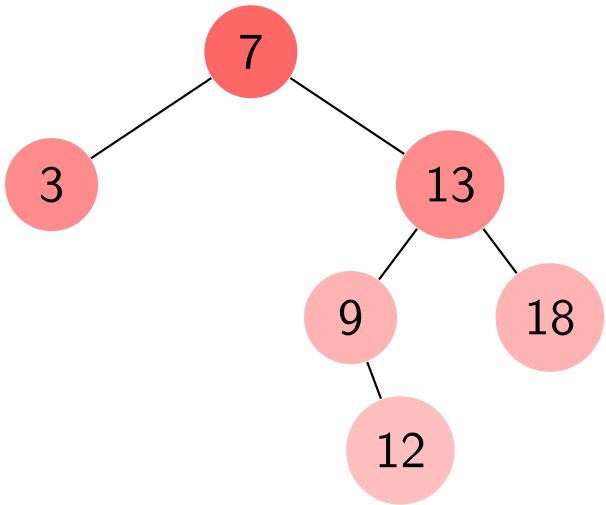
It is often more important to minimize the total cost of several searches

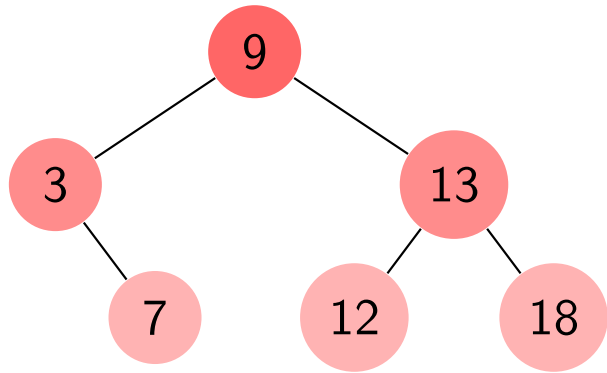
- Rather than the worst-case cost of a single search.

Optimal Binary Search Trees

It is often more important to minimize the total cost of several searches

- Rather than the worst-case cost of a single search.

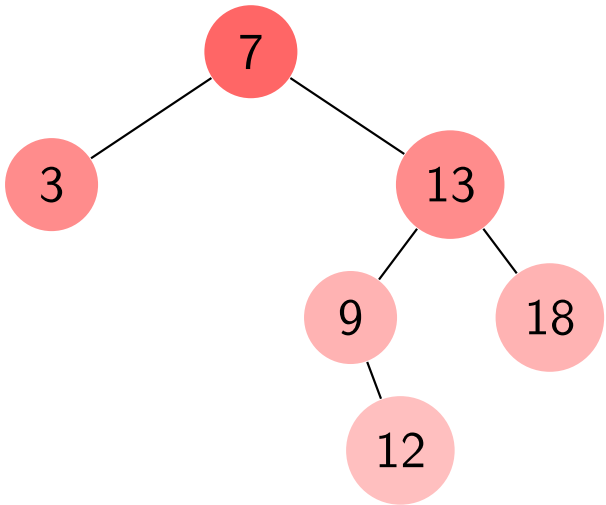


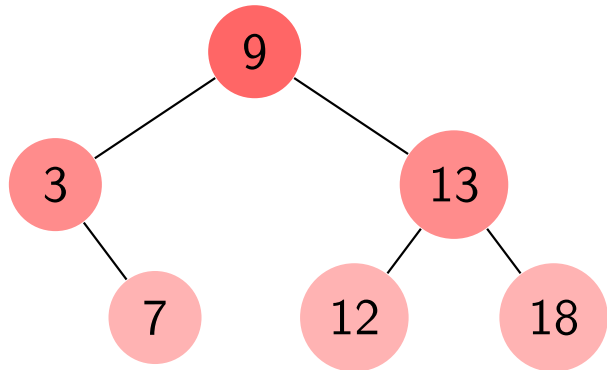


Optimal Binary Search Trees

It is often more important to minimize the total cost of several searches

- Rather than the worst-case cost of a single search.



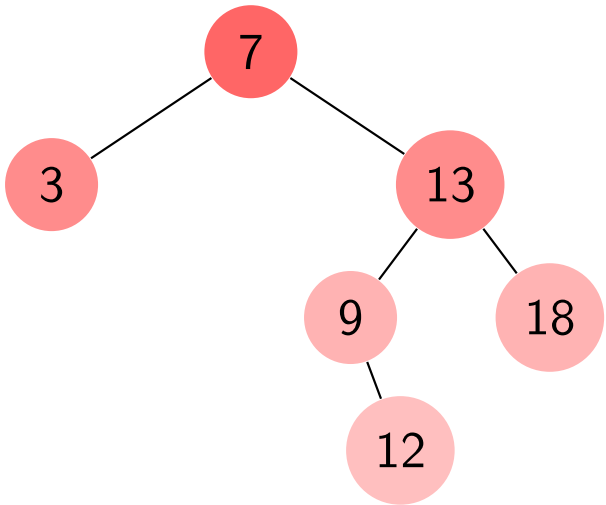


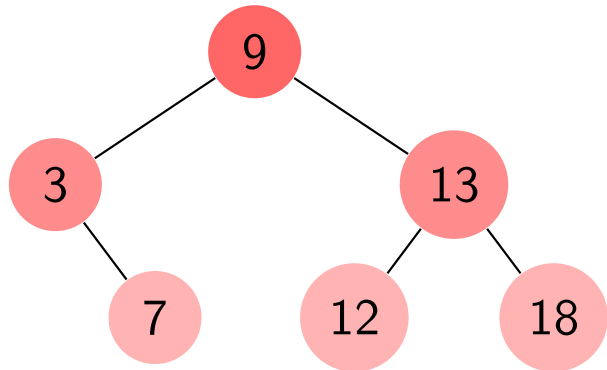
If x is a more frequent search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y

Optimal Binary Search Trees

It is often more important to minimize the total cost of several searches

- Rather than the worst-case cost of a single search.





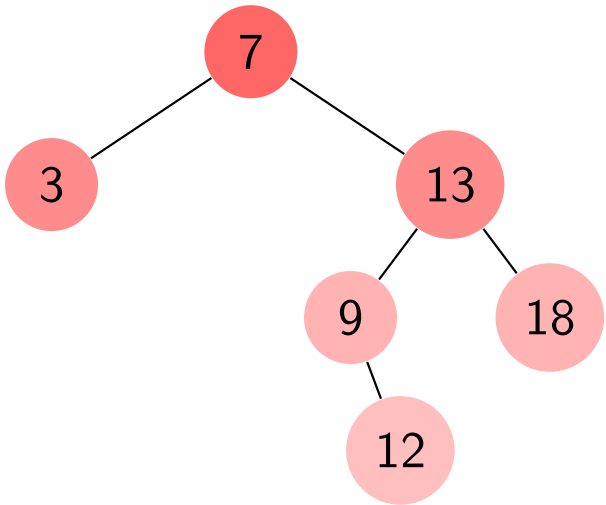
If x is a more frequent search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y

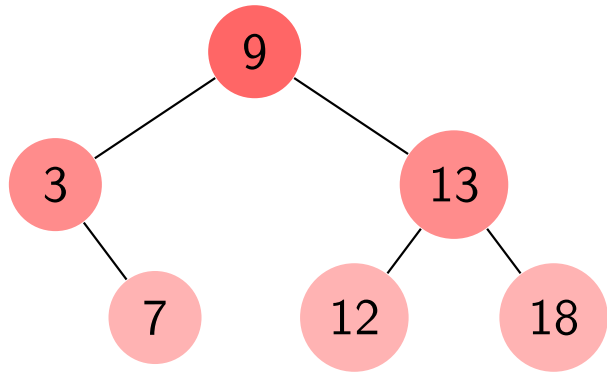
- Even if that means increasing the overall depth of the tree.

Optimal Binary Search Trees

It is often more important to minimize the total cost of several searches

- Rather than the worst-case cost of a single search.





If x is a more frequent search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y

- Even if that means increasing the overall depth of the tree.
- A perfectly balanced tree is not the best choice if some items are significantly more popular than others.

Optimal Binary Search Trees

This state of affairs suggests the following problem:

Given a sorted array of keys $A[1..n]$ and an array of corresponding access frequencies $f[1..n]$, build the binary search tree to store the keys and that minimizes the total search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

Optimal Binary Search Trees

This state of affairs suggests the following problem:

Given a sorted array of keys $A[1..n]$ and an array of corresponding access frequencies $f[1..n]$, build the binary search tree to store the keys and that minimizes the total search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

It is helpful to write a good recursive definition of the function we are trying to optimize.

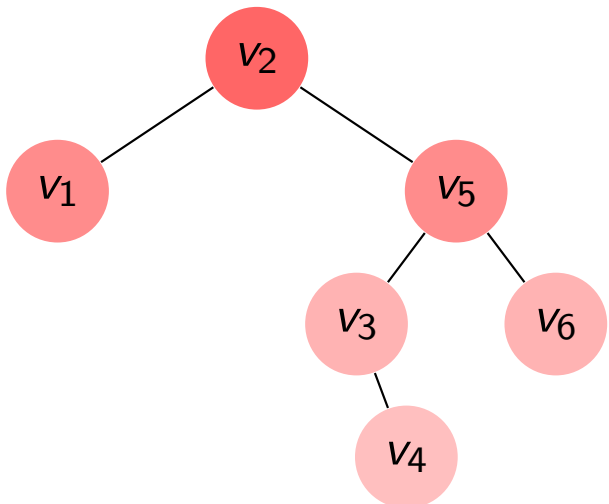
Optimal Binary Search Trees

This state of affairs suggests the following problem:

Given a sorted array of keys $A[1..n]$ and an array of corresponding access frequencies $f[1..n]$, build the binary search tree to store the keys and that minimizes the total search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

It is helpful to write a good recursive definition of the function we are trying to optimize.

Given a binary search tree T with n nodes, let $v_1, v_2, \ldots, v_n$ be the nodes of T, indexed in sorted order, so that each node v_i stores the corresponding key $A[i]$.



Optimization function

The total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \times d_T(v_i)$$

where $d_T(v_i)$ is the depth of v_i in T (where depth of root is 1).

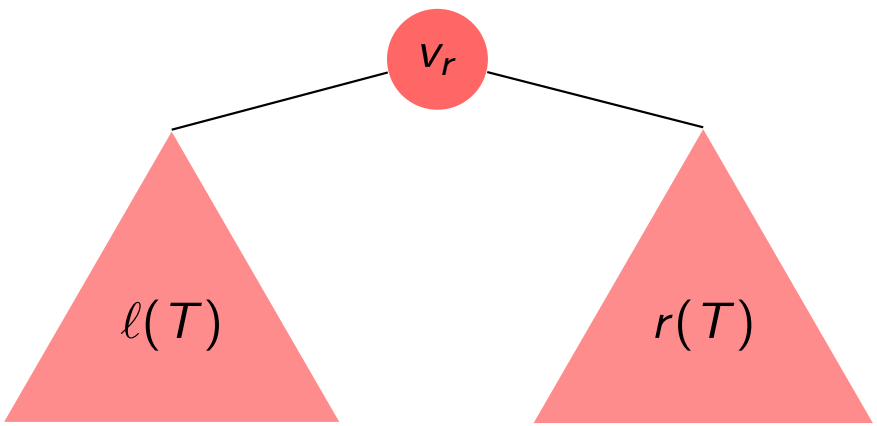
Optimization function

The total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \times d_T(v_i)$$

where $d_T(v_i)$ is the depth of v_i in T (where depth of root is 1).

Let v_r, where $1 \leq r \leq n$ be the root of T and let $\ell(T)$ and $r(T)$ be the left and right subtrees of v_r.



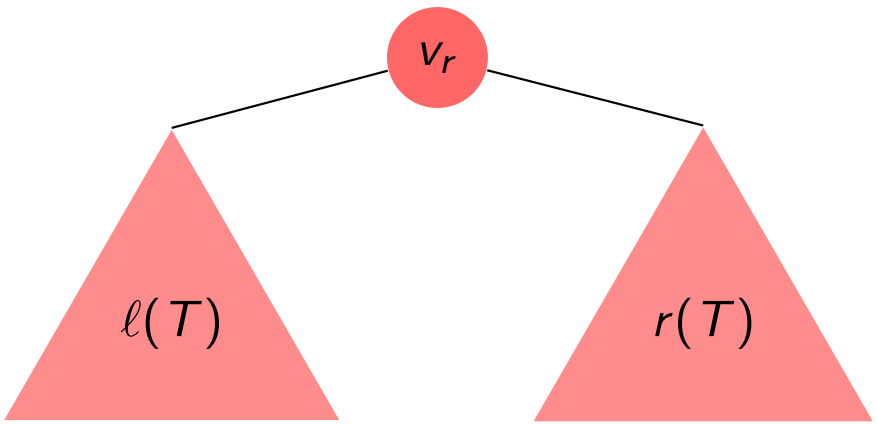
Optimization function

Then:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \times d_T(v_i)$$

$$= \sum_{i=1}^{n} f[i] + \sum_{i=1}^{r-1} f[i] \times d_{\ell(T)}(v_i) + \sum_{i=r+1}^{n} f[i] \times d_{r(T)}(v_i)$$

$$= \sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1..n])$$



Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

- the left subtree $\ell(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

- the left subtree $\ell(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$
- the right subtree $r(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$.

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

- the left subtree $\ell(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$
- the right subtree $r(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$.

So, if we choose the correct key to store at the root, the recursion fairy will construct the rest of the optimal tree.

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

- the left subtree $\ell(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$
- the right subtree $r(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$.

So, if we choose the correct key to store at the root, the recursion fairy will construct the rest of the optimal tree.

So how do we choose the correct key?

Optimization function

Goal is to compute optimal tree T_{opt} that minimizes:

$$Cost(T, f[1..n]) =$$

$$\sum_{i=1}^{n} f[i] + Cost(\ell(T), f[1..r-1]) + Cost(r(T), f[r+1\dot{n}])$$

Let v_r be the root of T_{opt}; the above definition implies that:

- the left subtree $\ell(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$
- the right subtree $r(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$.

So, if we choose the correct key to store at the root, the recursion fairy will construct the rest of the optimal tree.

So how do we choose the correct key? Try each one!