Chapter 3: Algorithm Strategies

Contents

- Brute-force
- The Greedy method
- Divide-and-conquer
- Backtracking
- Branch-and-bound
- Heuristics

Brute-force

Brute-force

- A straightforward approach to solving a problem, usually directly based on the problem statement and definition
- Systematically enumerates all possible candidates for the solution and checks whether each candidate satisfies the problem statement

Brute-force algorithms

Selection sort

Based on sequentially finding the smallest elements

Bubble Sort

Based on consecutive swapping adjacent pairs. This causes a slow migration of the smallest elements to the left of the array.

Brute-force algorithms

Bubble Sort

Based on consecutive swapping adjacent pairs. This causes a slow migration of the smallest elements to the left of the array.

```
procedure BUBBLESORT(A[0 ... n − 1])
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if  A[j + 1] < A[j] then
                swap A[j+1] and A[j]
            end if
        end for
    end for
end procedure
```

Brute-force algorithms

Sequential Searching

```
procedure SequentialSearch(A[0 ... n-1], K)        ▷ K is the search key
    i ← 0
    while i < n and A[i] ≠ K  do
        i ← i + 1
    end while
    if i < n then
        return i
    else
        return -1
    end if
end procedure
```

Brute-force algorithms

Brute-Force String Matching: Searching for a pattern, P[0...m-1], in text, T[0...n-1]

```
procedure BruteForceStringMatch(T[0...n − 1], P[0...m − 1])
    for i ← 0 to n − m do
        j ← 0
        while j < m and P[j] == T[i + j] do
            j ← j + 1
        end while
        if  j == m then
            return i
        end if
    end for
    return -1
end procedure
```

The Greedy method

The greedy method

- Builds up a solution piece by piece, always choosing the next piece that looks best at the moment
- The main idea is to make locally optimal choice in the hope that this choice will lead to a globally optimal solution
- Greedy algorithms do not always yield optimal solutions, but for many problems they do

Huffman coding

Coding: Assigning binary codewords to (blocks of) source symbols

Huffman coding is a lossless data compression algorithm.

Idea:

- Assign variable-length codes to input characters, based on the frequencies of corresponding characters.
- Instead of using ASCII codes, store the more frequently occurring characters using fewer bits and less frequently occurring characters using more bits.

Huffman coding

There are mainly two major parts in Huffman Coding

1. Build a **Huffman Tree** from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Huffman coding

Building a Huffman tree:

1. Organize the entire character set into a row, ordered according to frequency from highest to lowest (or vice versa). Each character is now a node at the leaf level of a tree
2. Find two nodes with the smallest combined frequency weights and join them to form a third node, resulting in a simple two-level tree. The weight of the new node is the combined weights of the original two nodes.
3. Repeat step 2 until all of the nodes, on every level, are combined into a single tree.

Huffman coding example

Input character string: "**datastructures**"

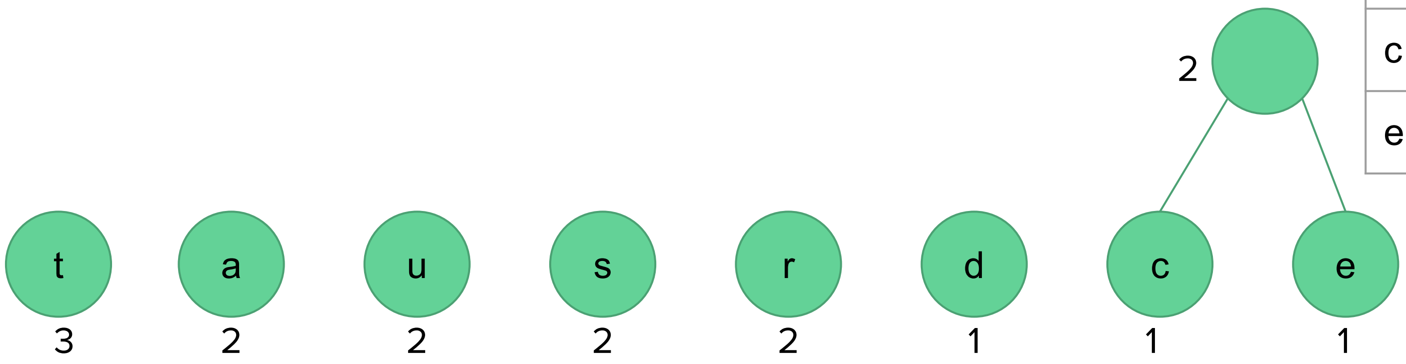
We first build the frequency table

character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

Build a Huffman tree:

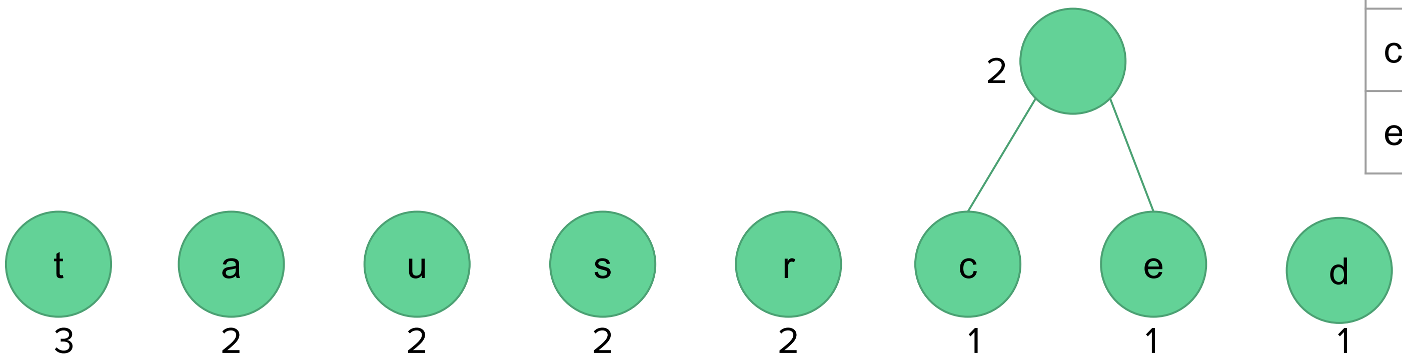
character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1



Huffman coding example

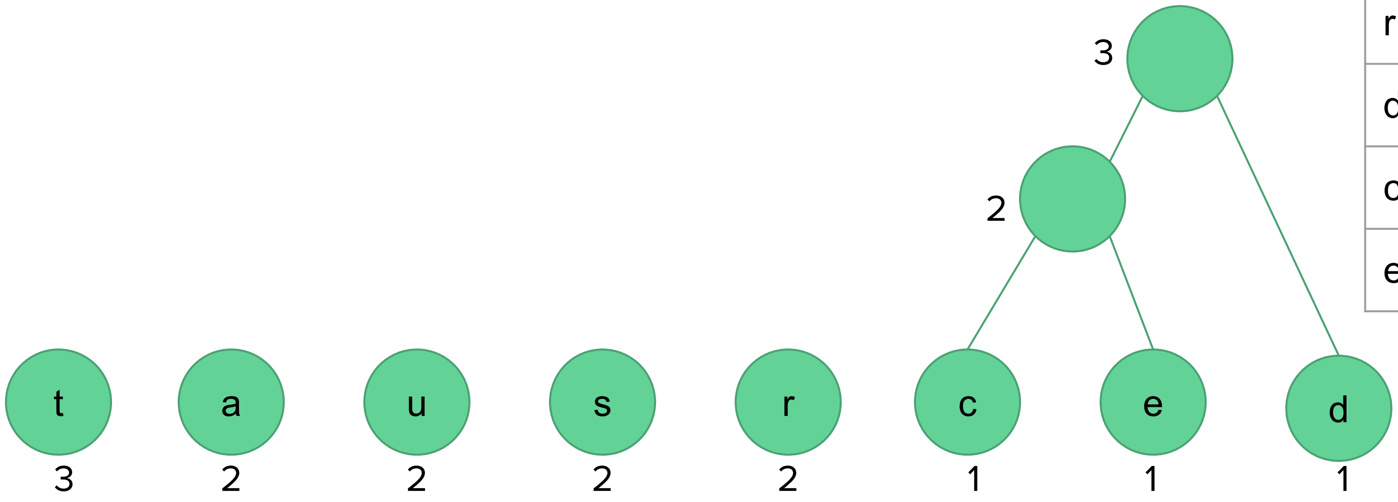
Build a Huffman tree:

character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1



Huffman coding example

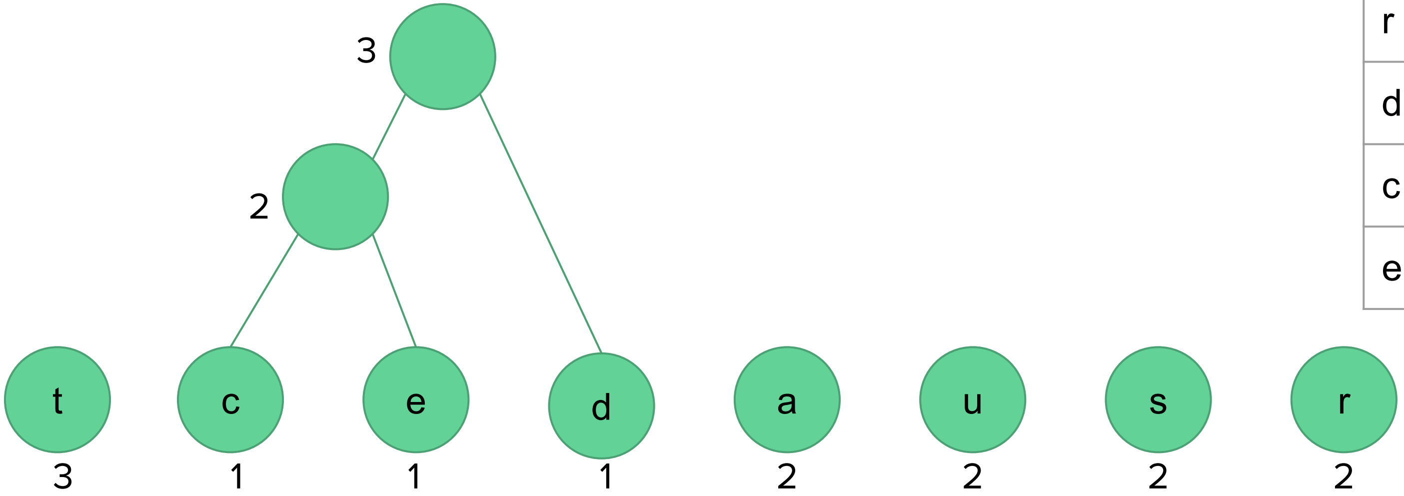
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

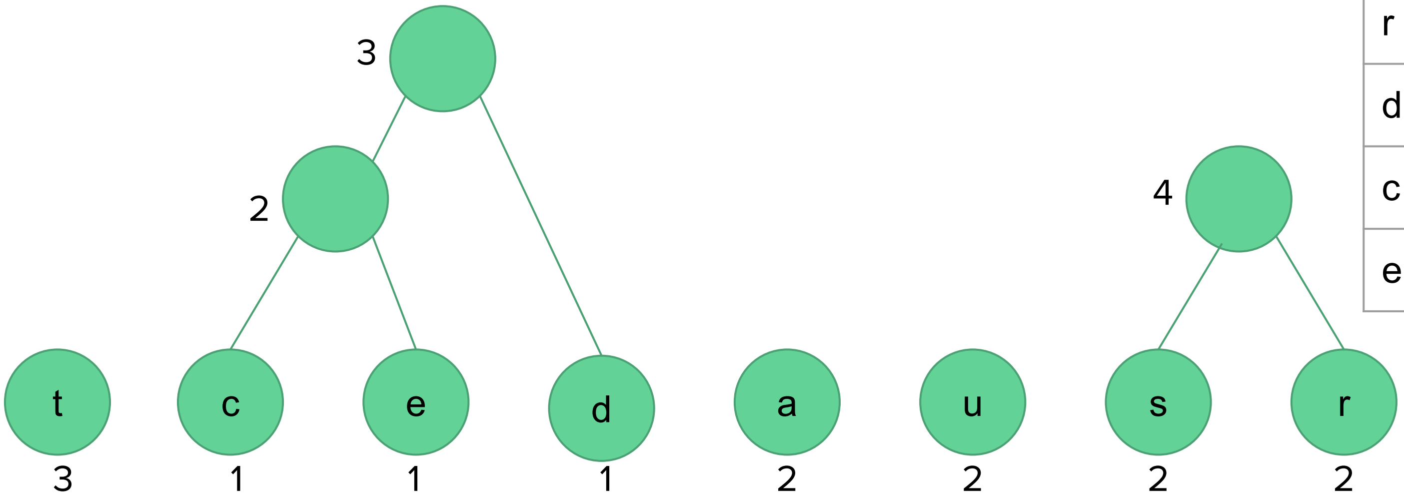
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

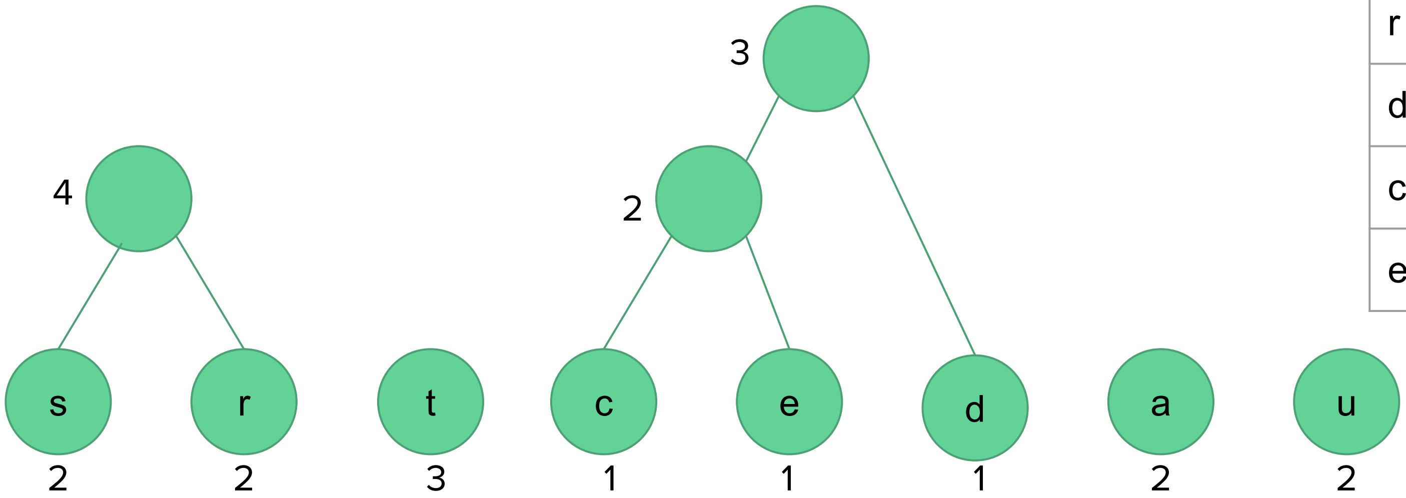
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

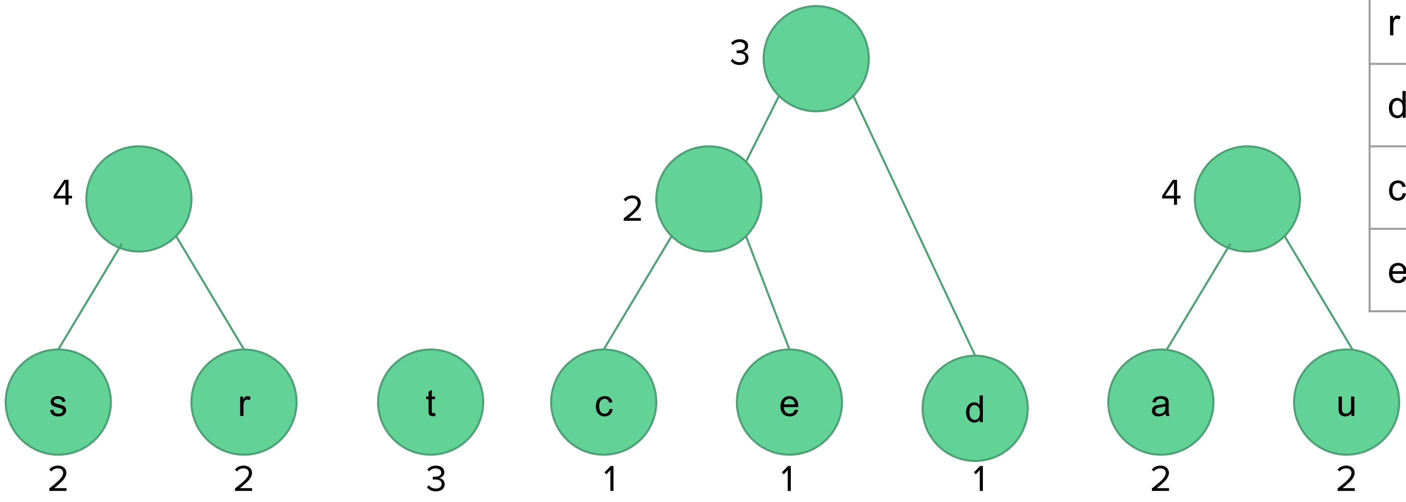
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

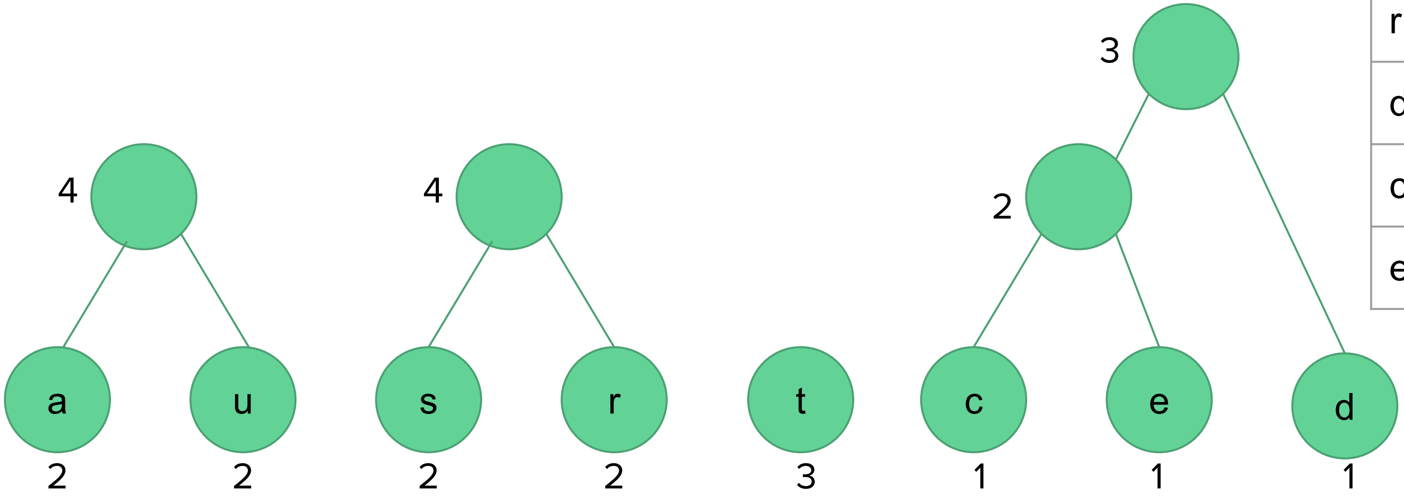
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

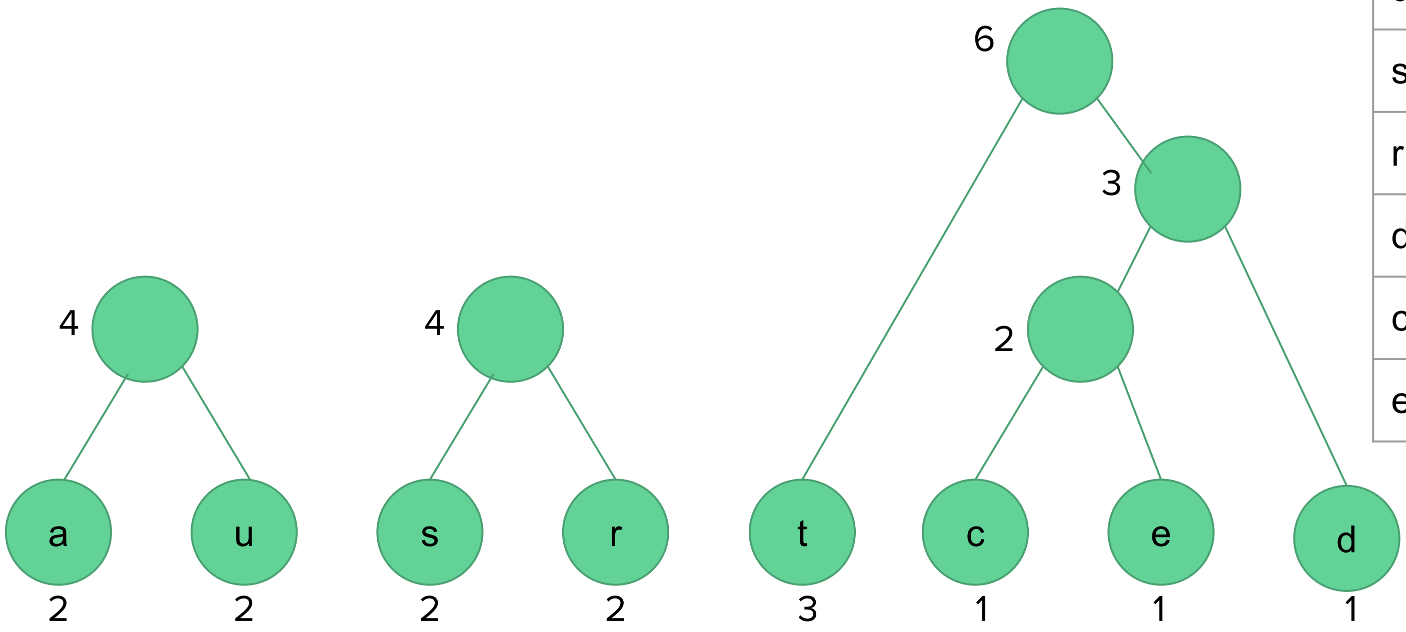
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

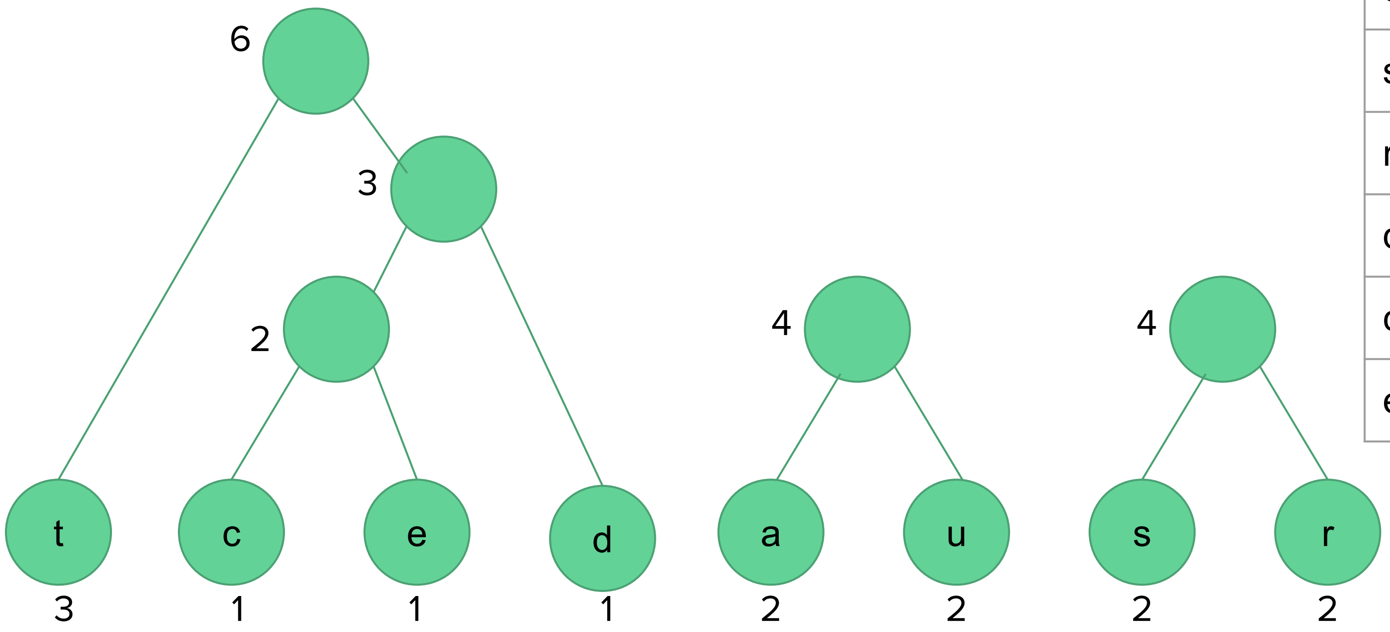
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

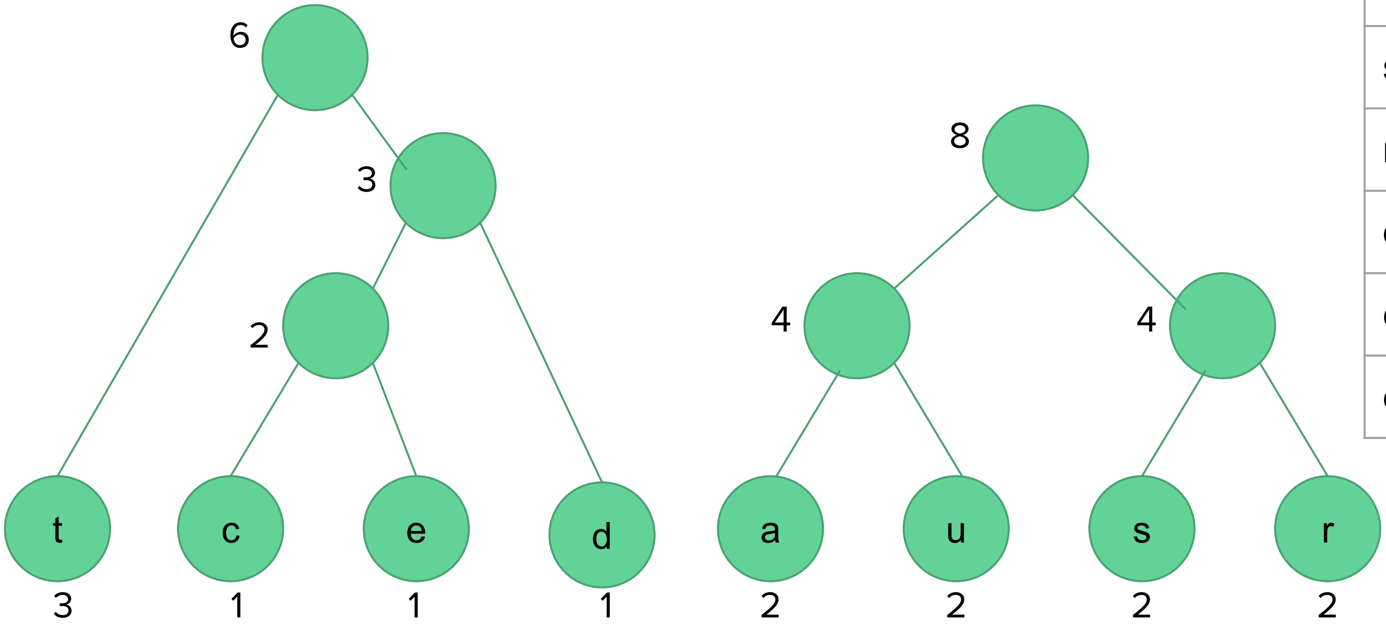
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

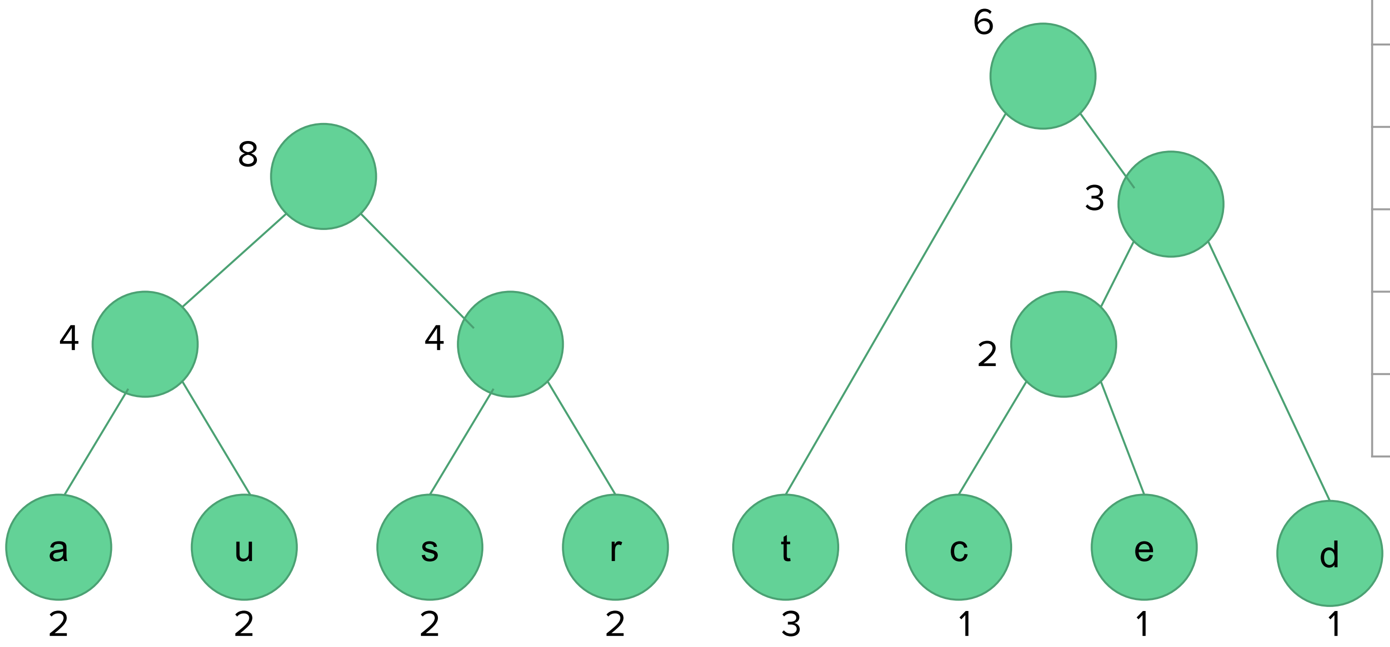
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

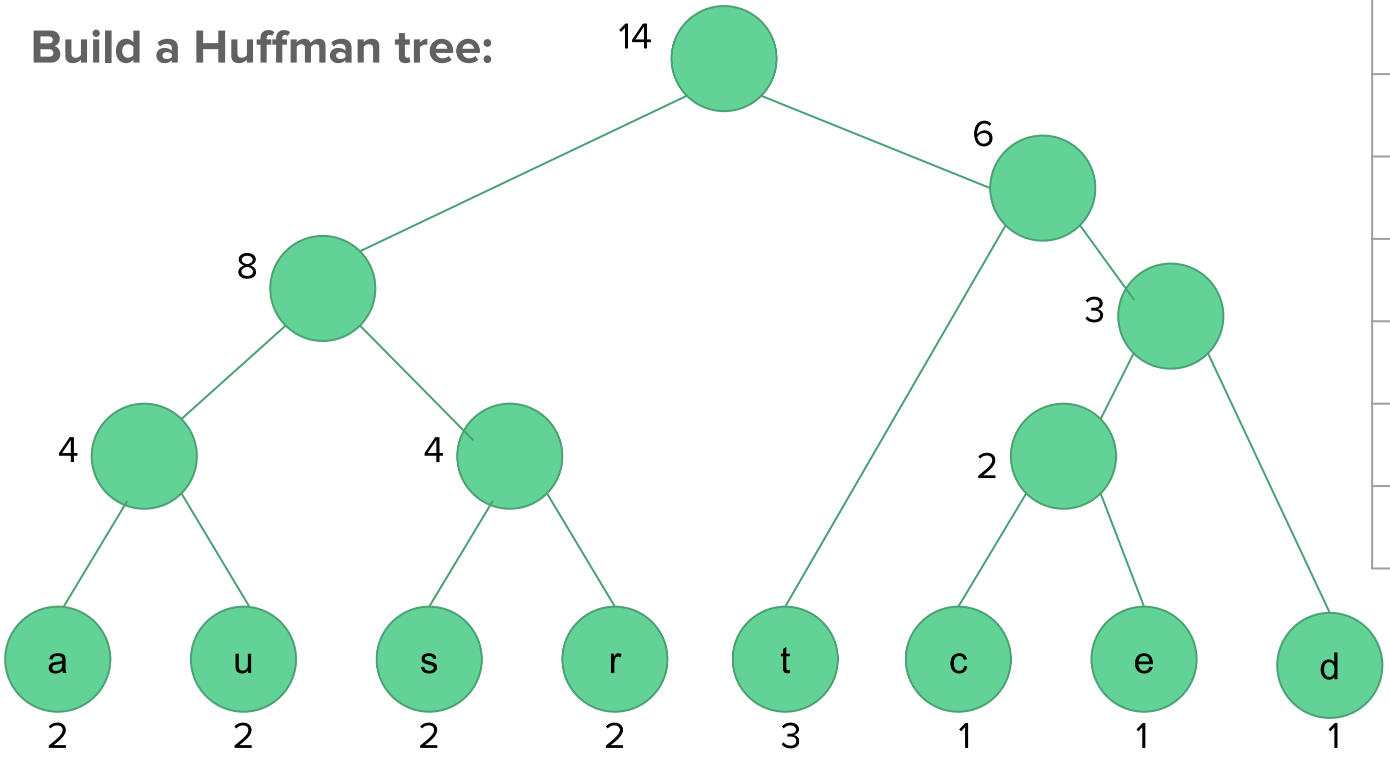
Build a Huffman tree:



character	frequency
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

Build a Huffman tree:



character	**frequency**
t	3
a	2
u	2
s	2
r	2
d	1
c	1
e	1

Huffman coding example

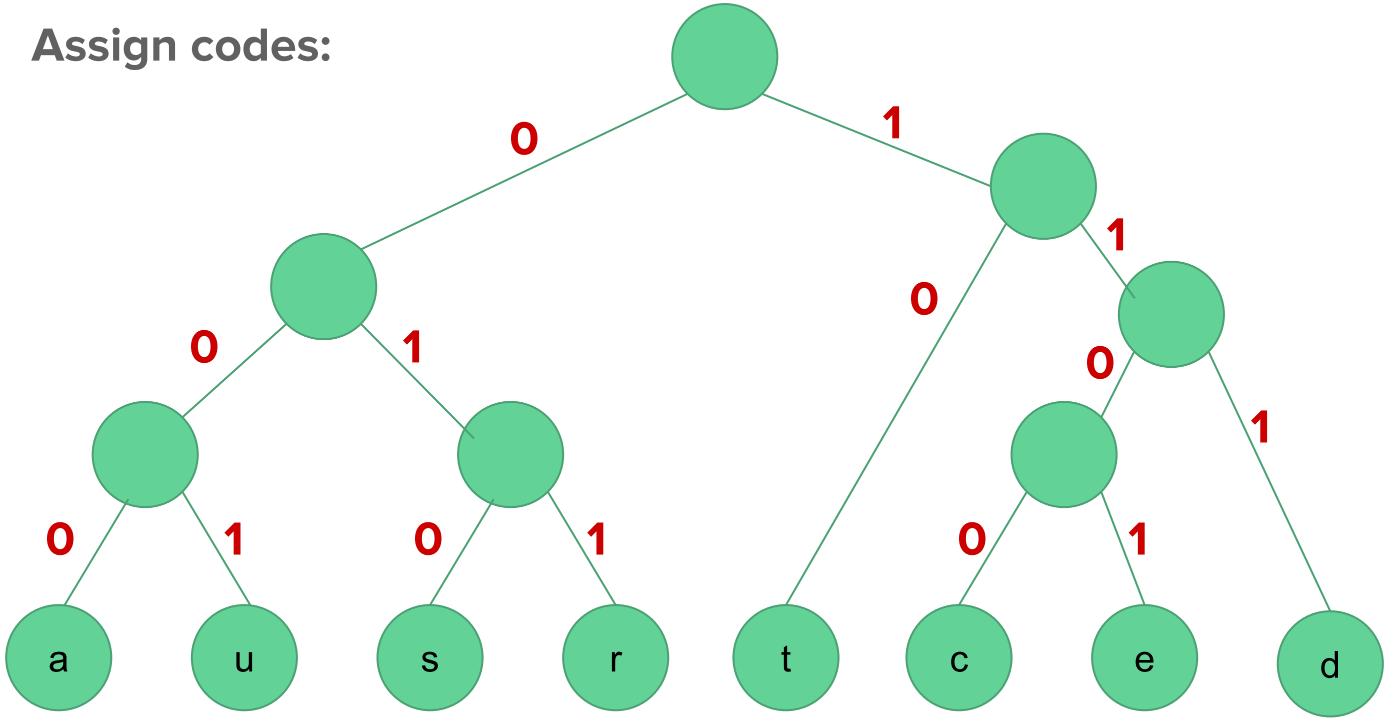
Now we **assign codes** to the tree by **placing a 0 on every left branch and a 1 on every right branch**

A traversal of the tree from root to leaf give the Huffman code for that particular leaf character

These codes are then used to encode the string

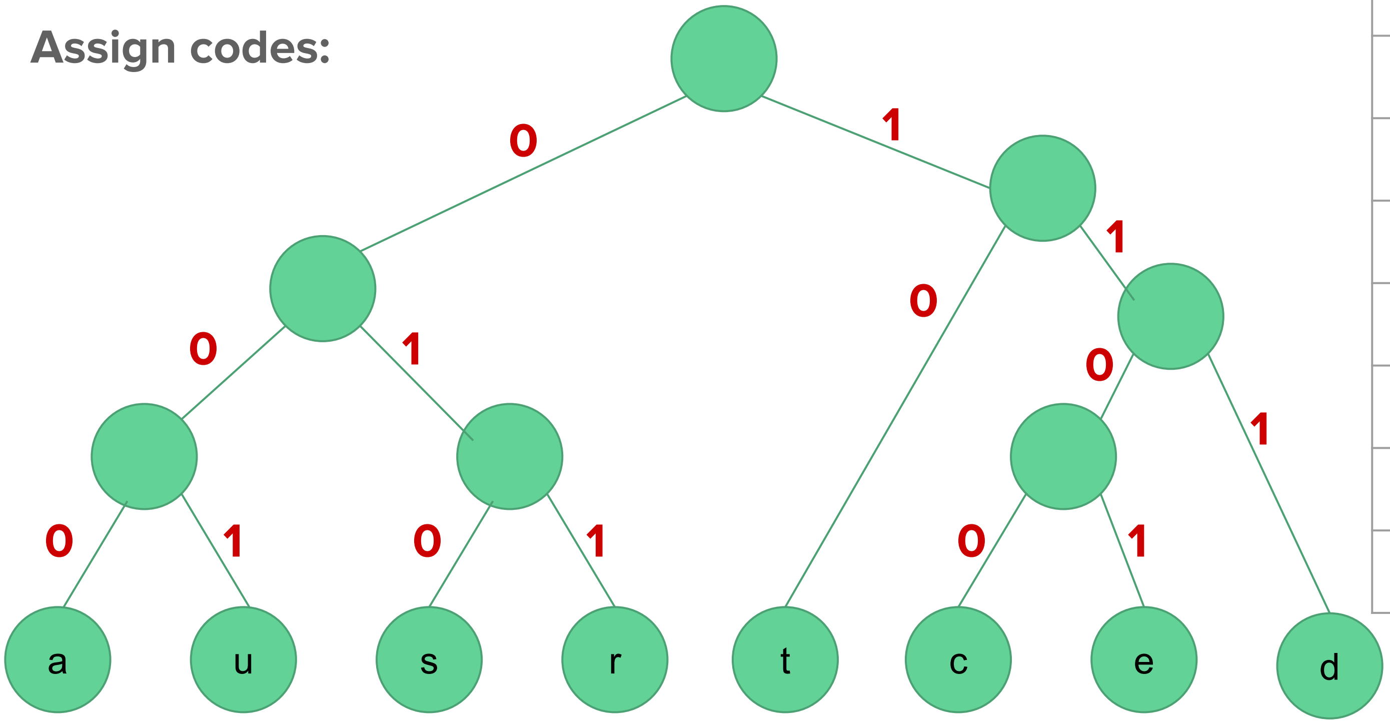
Huffman coding example

Assign codes:



Huffman coding example

Assign codes:



character	Huffman code
t	10
a	000
u	001
s	010
r	011
d	111
c	1100
e	1101

Huffman coding example

Thus “datastructures” turns into

11100010000010100110011100100010111101010

If 8-bit ASCII code had been used instead of Huffman coding, “datastructures” would have been

0110010001100001011101000110000101110011
0111010001110010011101010110001101110100
01110101011100100110010101110011

character	Huffman code	ASCII code
t	10	01110100
a	000	01100001
u	001	01110101
s	010	01110011
r	011	01110010
d	111	01100100
c	1100	01100011
e	1101	01100101

Huffman coding

Uncompression:

Read the file bit by bit

1. Start at the root of the tree
2. If a 0 is read, head left
3. If a 1 is read, head right
4. When a leaf is reached, decode that character and start over again at the root of the tree

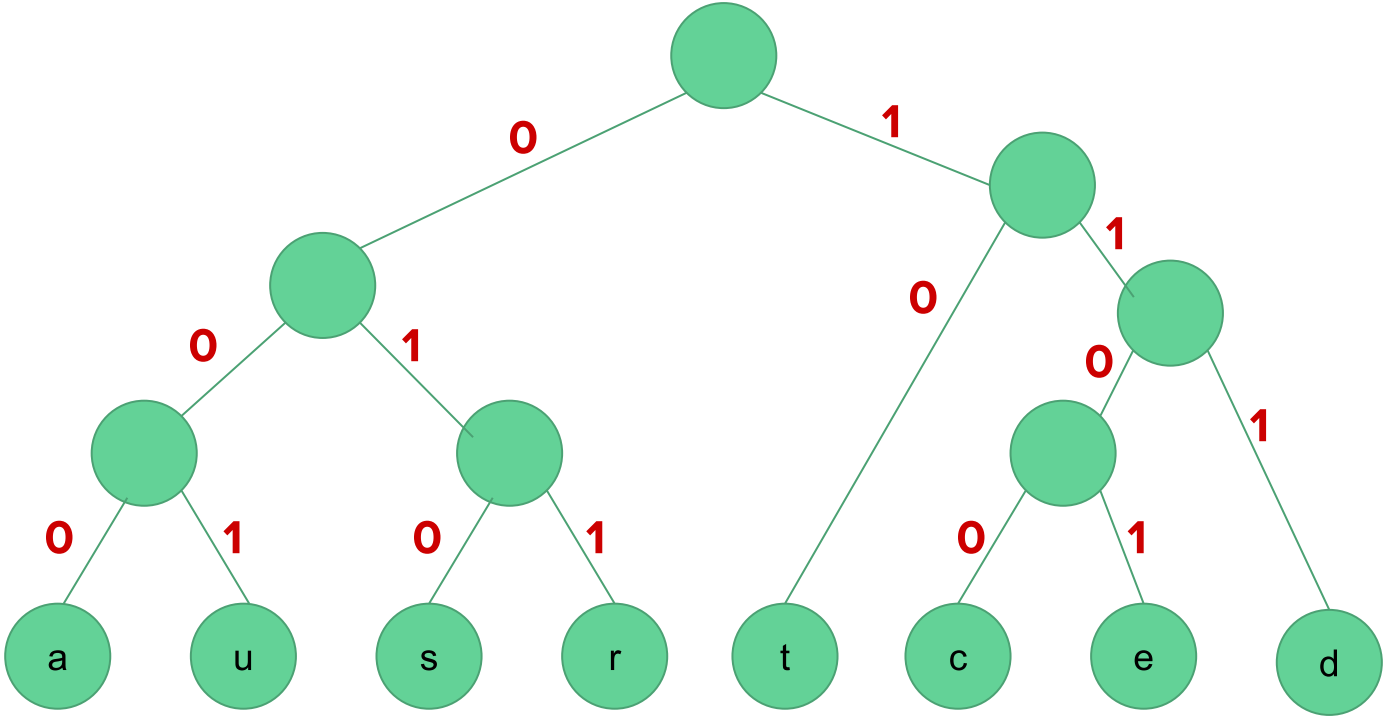
Huffman coding

Uncompression example:

Decode 11100010000010100110011100100010111101010 using the previous Huffman tree

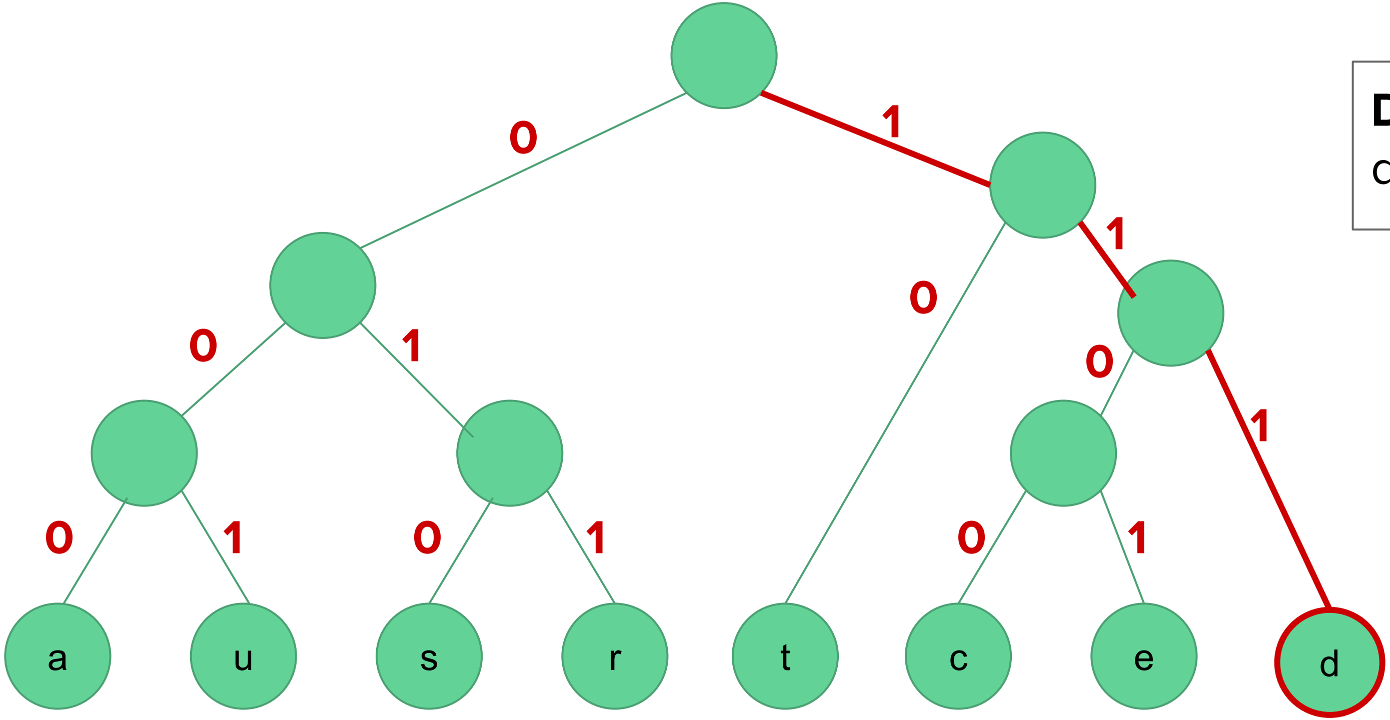
Uncompression example

Decode 11100010000010100110011100100010111101010



Uncompression example

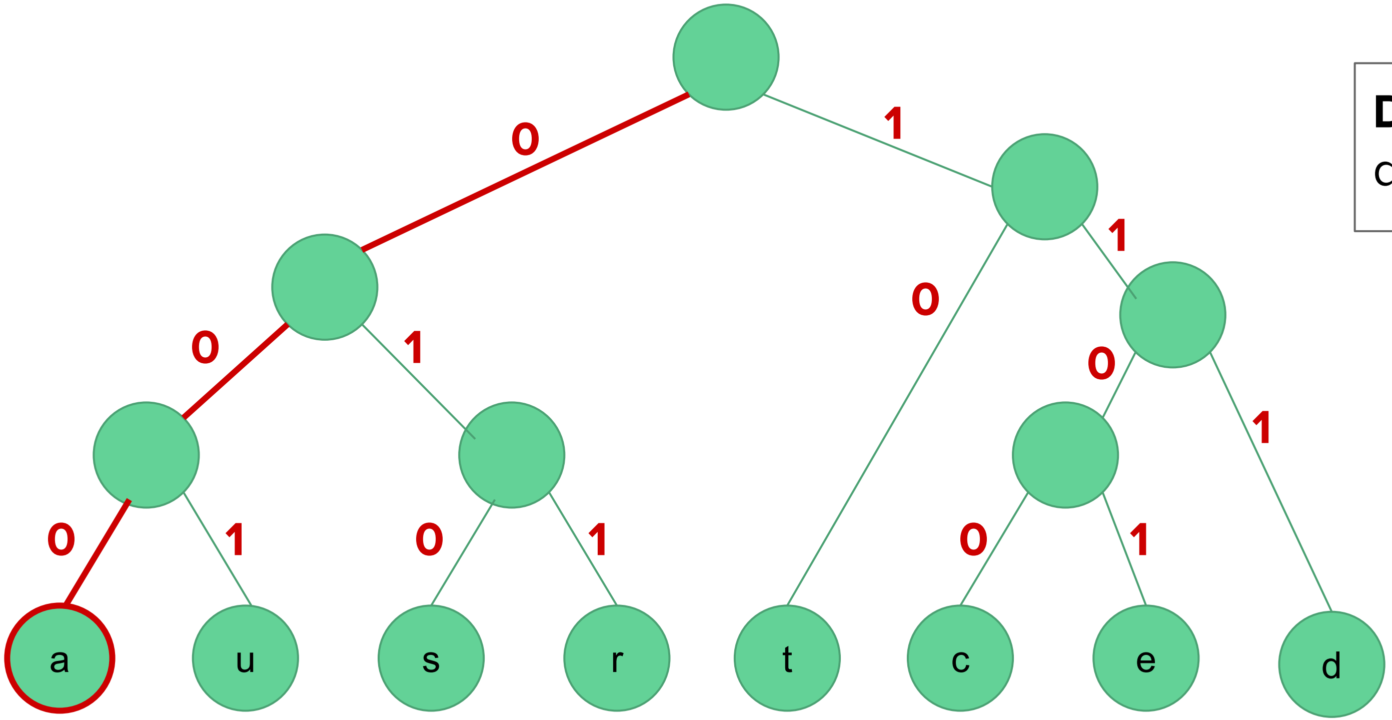
Decode **111**00010000010100110011100100010111101010



Decoded string:
d

Uncompression example

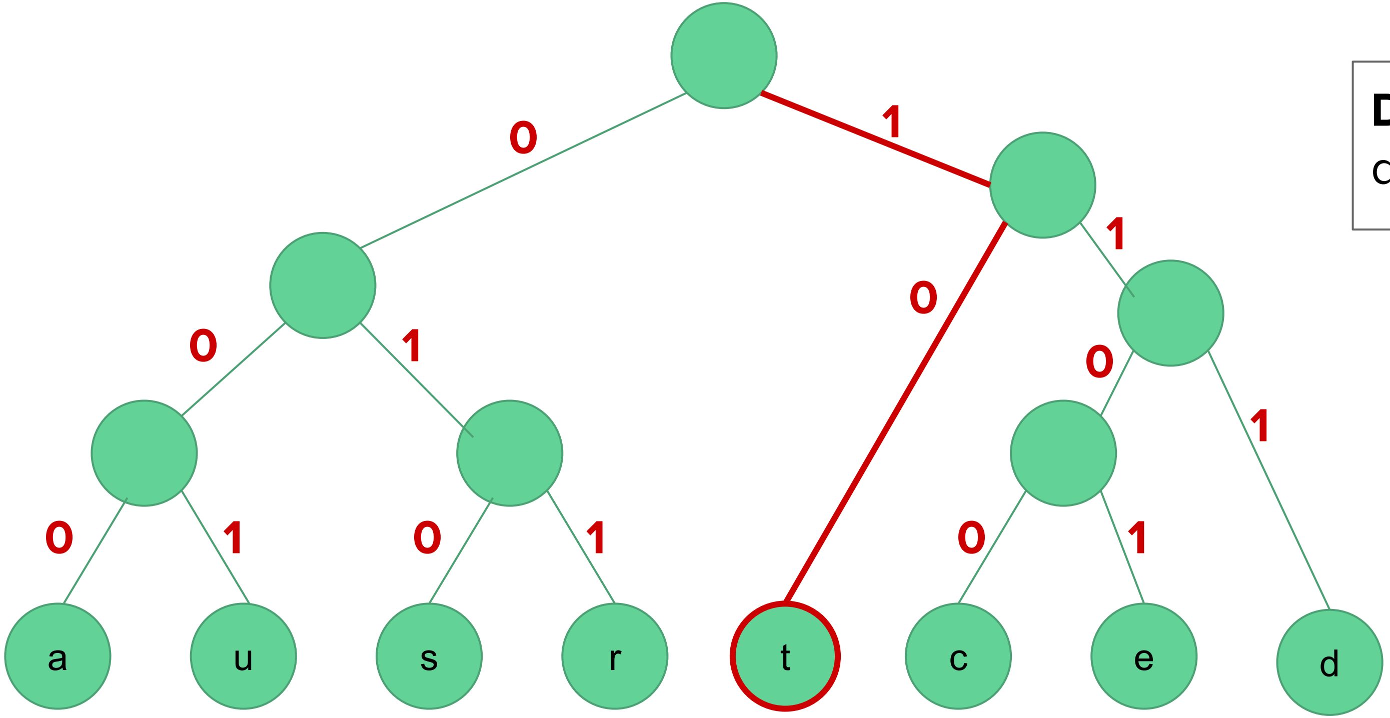
Decode 111**000**10000010100110011100100010111101010



Decoded string:
da

Uncompression example

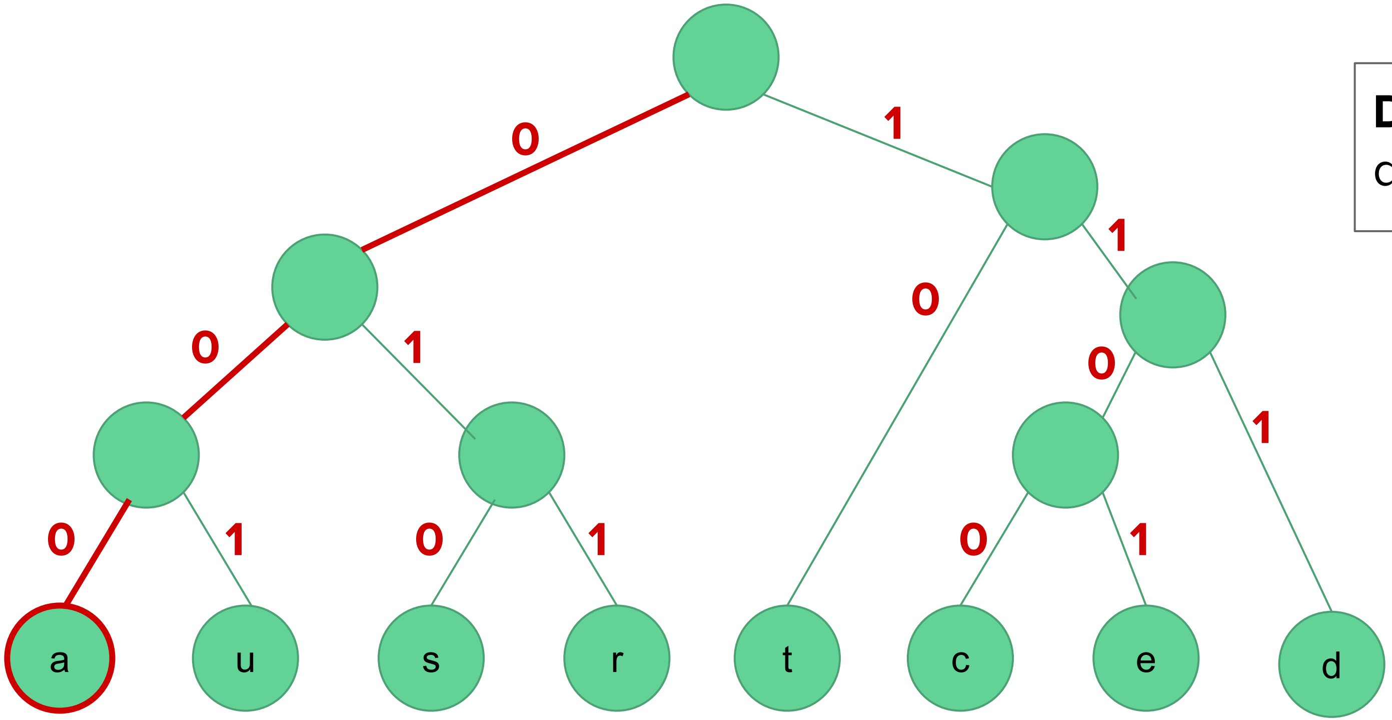
Decode 111000**10**0000010100110011100100010111101010



Decoded string:
dat

Uncompression example

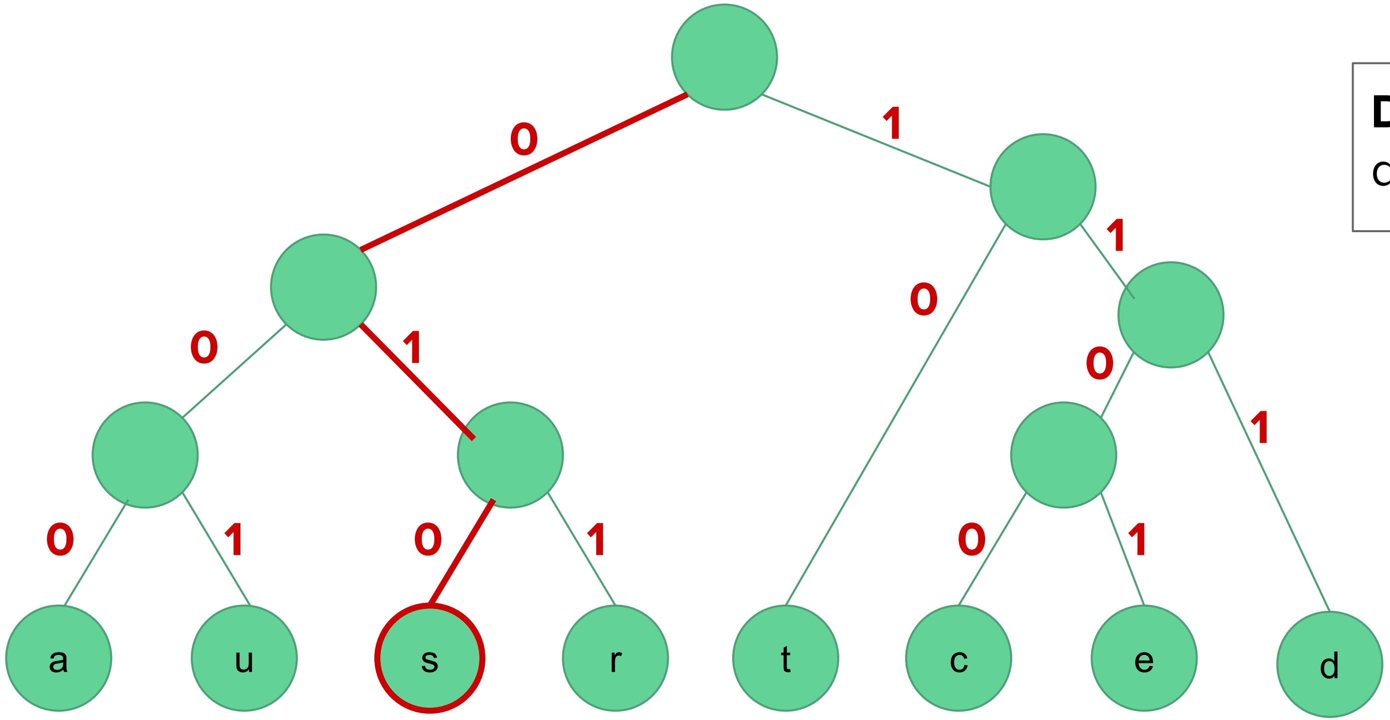
Decode 11100010**000**010100110011100100010111101010



Decoded string:
data

Uncompression example

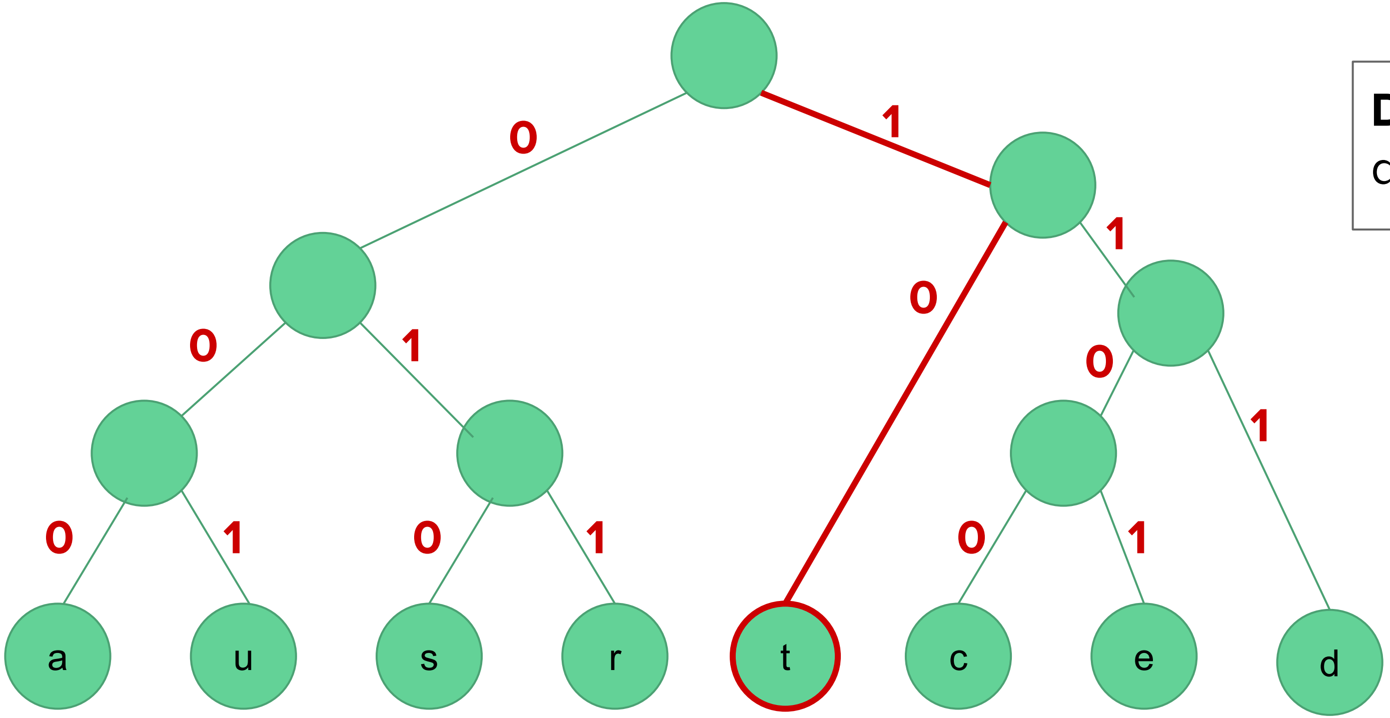
Decode 11100010000**010**10011001110010001011101010



Decoded string:
datas

Uncompression example

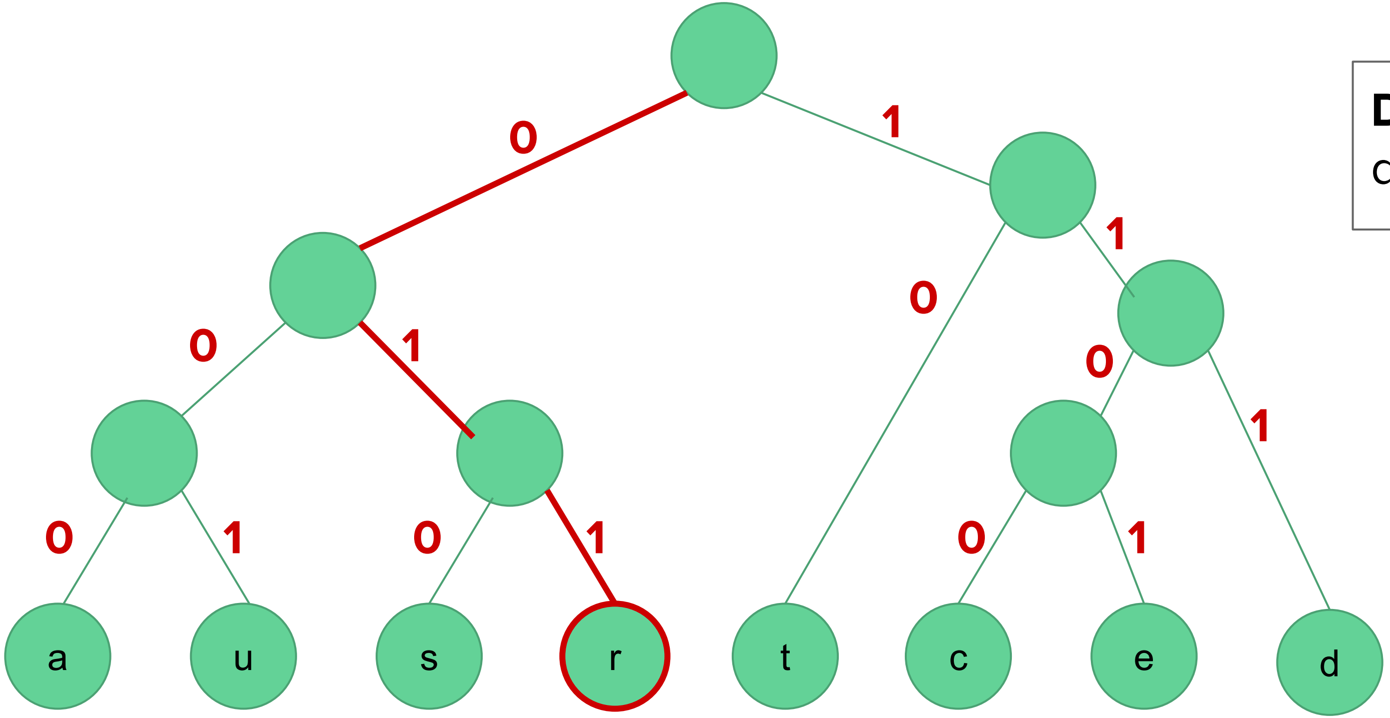
Decode 111000100000101**10**0110011100100010111101010



Decoded string:
datast

Uncompression example

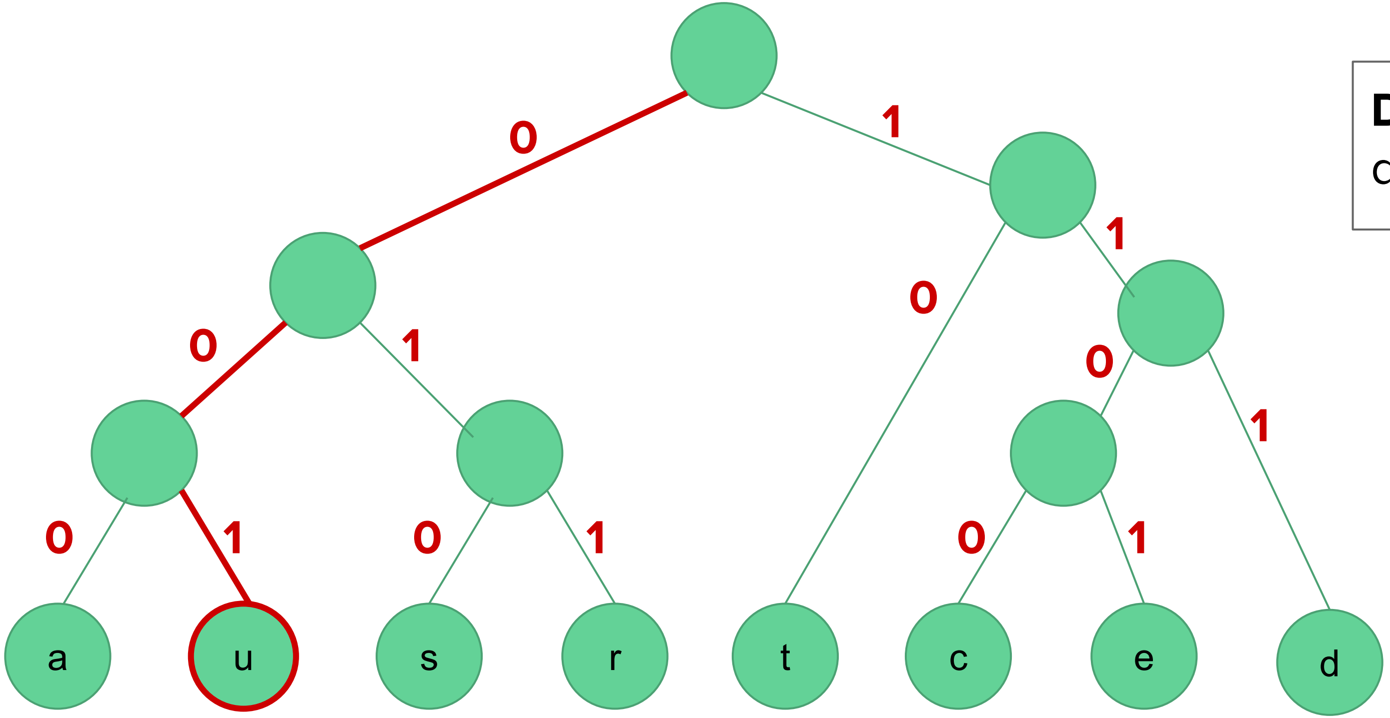
Decode 1110001000001010**011**0011100100010111101010



Decoded string:
datastr

Uncompression example

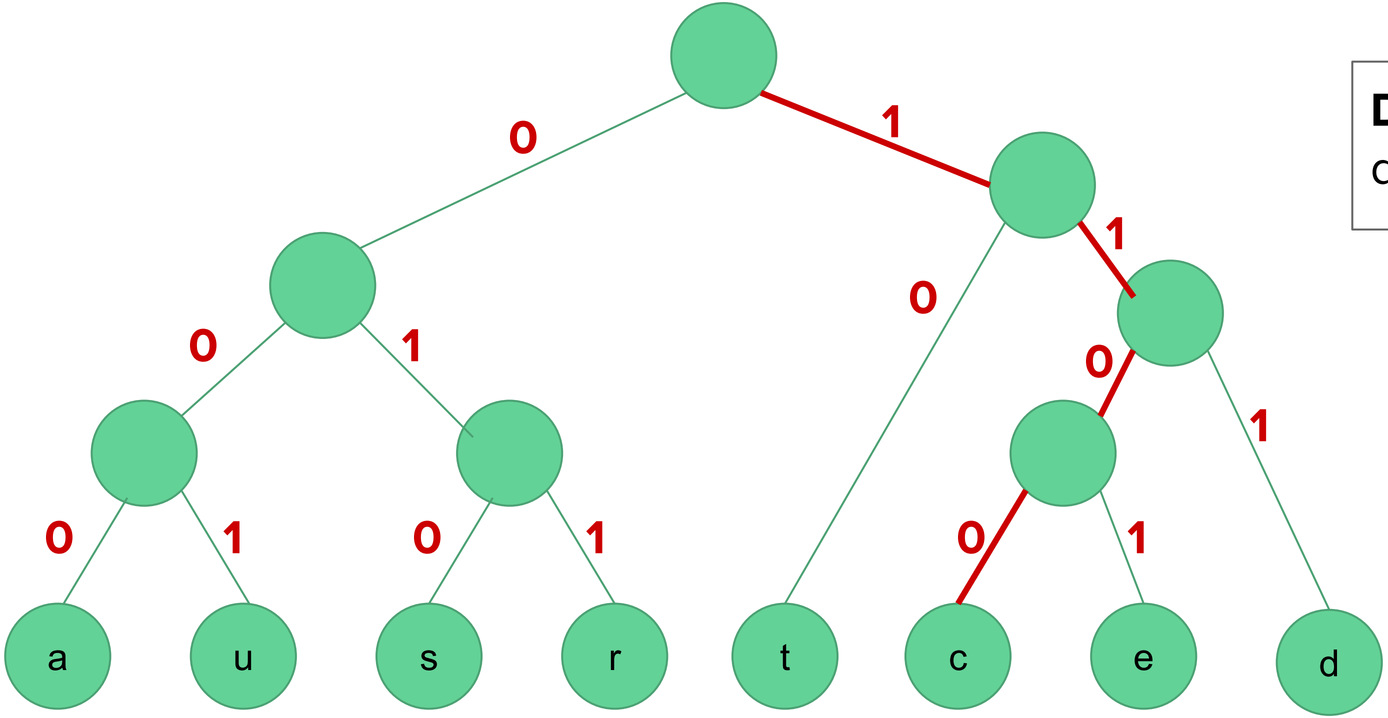
Decode 111000100000101001**1001**11001000101111101010



Decoded string:
datastru

Uncompression example

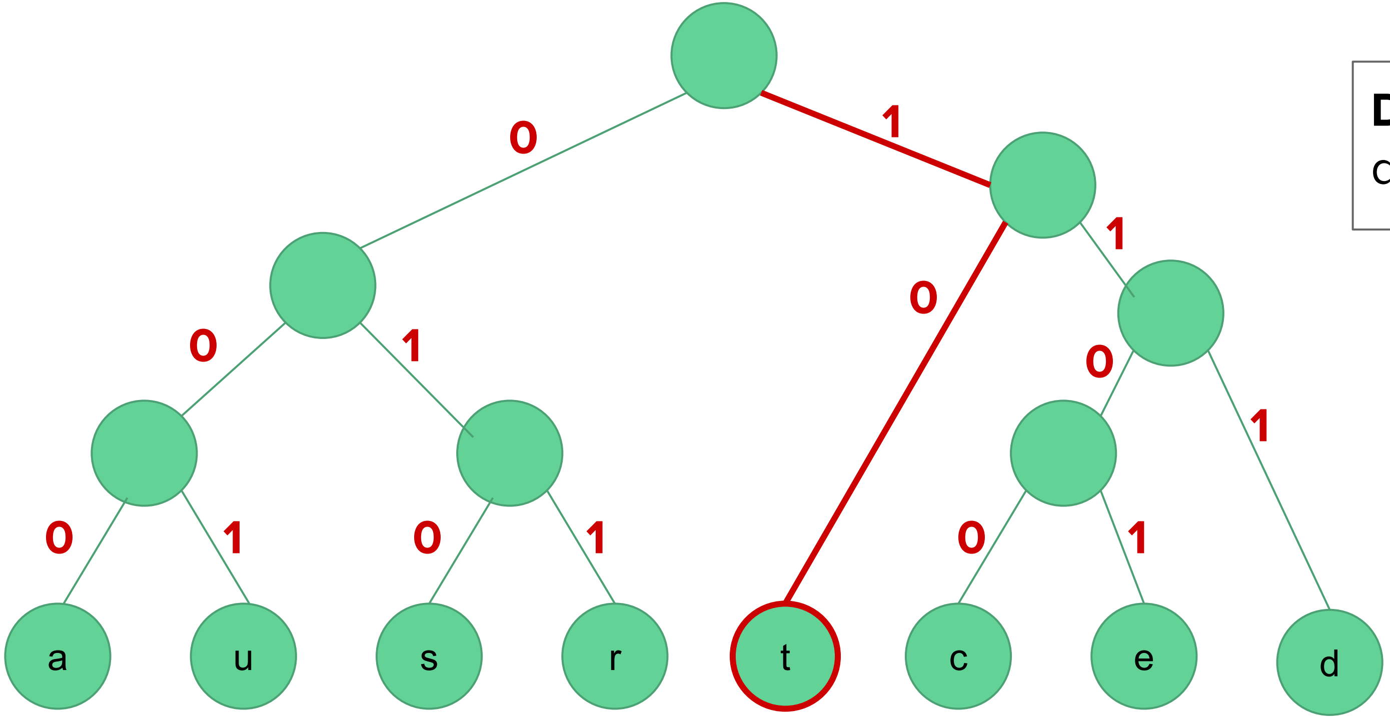
Decode 11100010000010100110011**1100**10001011101010



Decoded string:
datastruc

Uncompression example

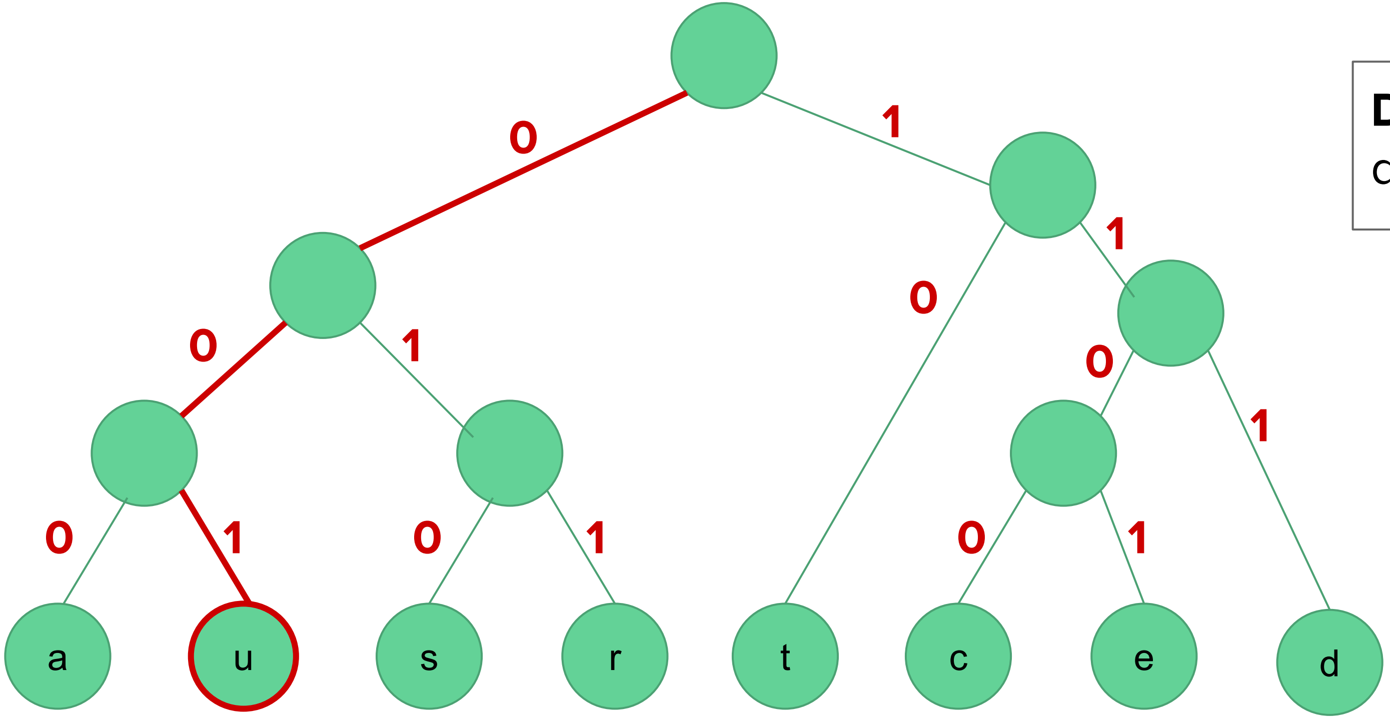
Decode 11100010000010100110011100**10**0010111101010



Decoded string:
datastruct

Uncompression example

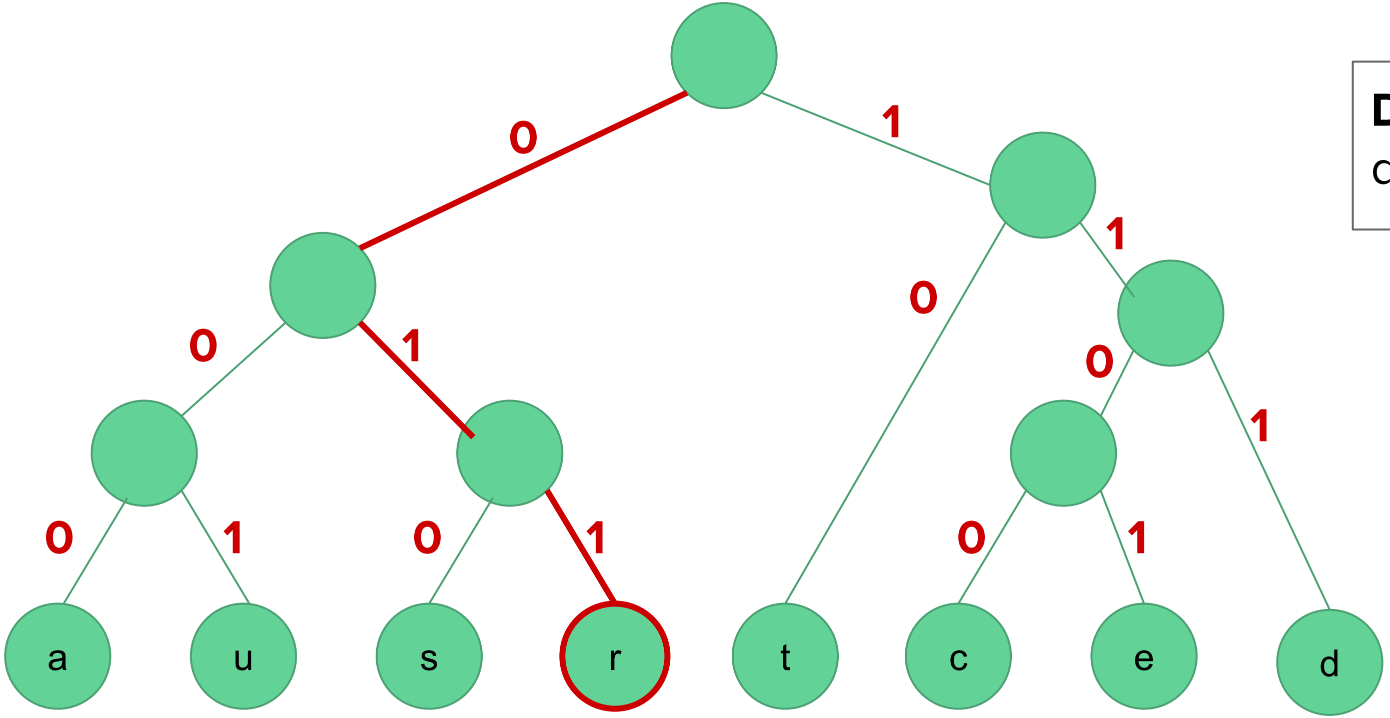
Decode 1110001000001010011001110010**001**0111101010



Decoded string:
datastructu

Uncompression example

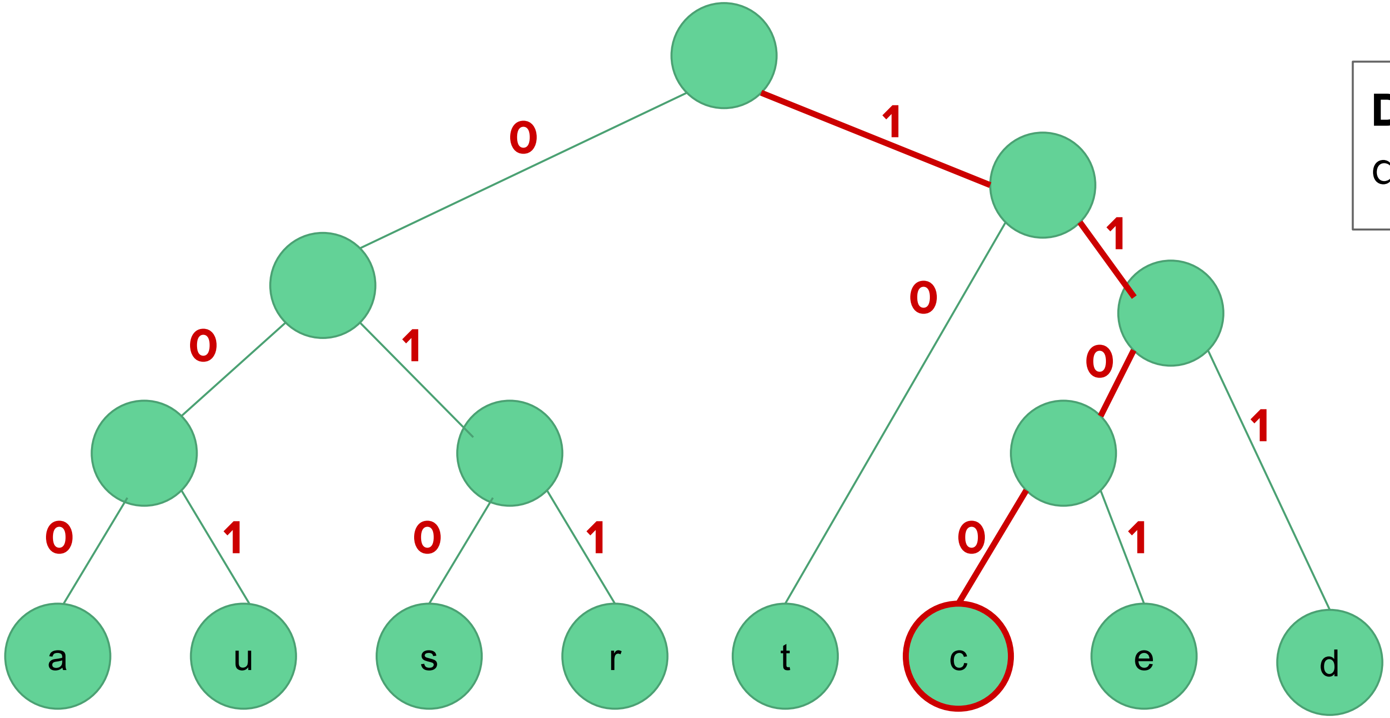
Decode 111000100000101001100111001000**1011**1101010



Decoded string:
datastructur

Uncompression example

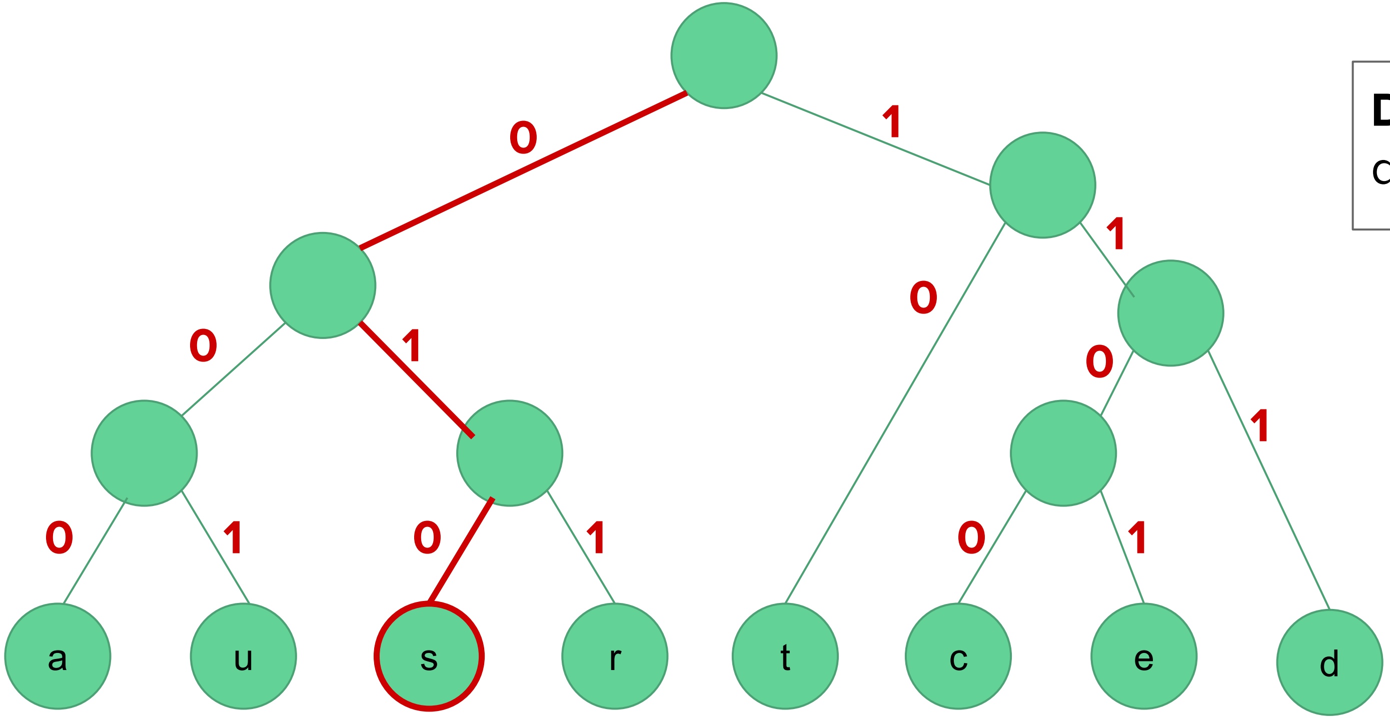
Decode 1110001000001010011001110010001011**1101**010



Decoded string:
datastructure

Uncompression example

Decode 111000100000101001100111001000101111101**010**



Decoded string:
datastructures

Activity-Selection Problem

The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

Activity-Selection Problem

Input: A set of activities that we wish to use a resource (such as classroom) which can serve only one activity at a time. Each activity a_i in the set $S = \{a_1, a_2, .., a_n\}$ has a start time s_i and a finish time f_i, where $0 \le s_i < f_i < \infty$.

If selected, activity a_i takes place during the time interval $[s_i, f_i)$

Output: A maximum-size subset of **mutually compatible** activities.

Two activities are compatible if and only if their intervals do not overlap.

Activity-Selection

Example

Consider the following set S of activities

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Activity-Selection

Greedy approach

- Choose an activity that leaves the resource available for as many other activities, i.e.

  Choose an activity with the earliest finish time

Activity-Selection

Greedy algorithm:

We assume that n input activities are already ordered by monotonically increasing finish time:

$f_1 \leq f_2, \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$

1. Select the activity with the earliest finish time
2. Eliminate the activities that could not be scheduled / incompatible activities
3. Repeat

Activity-Selection: Recursive greedy algorithm

Input: start times s, finish times f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem

Activity-Selection: Recursive greedy algorithm

Input: start times s, finish times f, the index k that defines the subproblem S_k it is to solve, and the size n of the original problem

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
1  m = k + 1
2  while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
3      m = m + 1
4  if m ≤ n
5      return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6  else return Ø
```

We start with k = 0 and a fictitious activity a0 with f0 = 0

Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

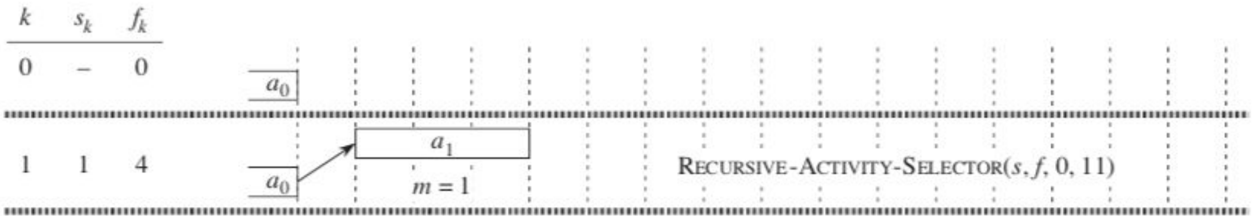
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```

k	s_k	f_k
0	–	0

Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

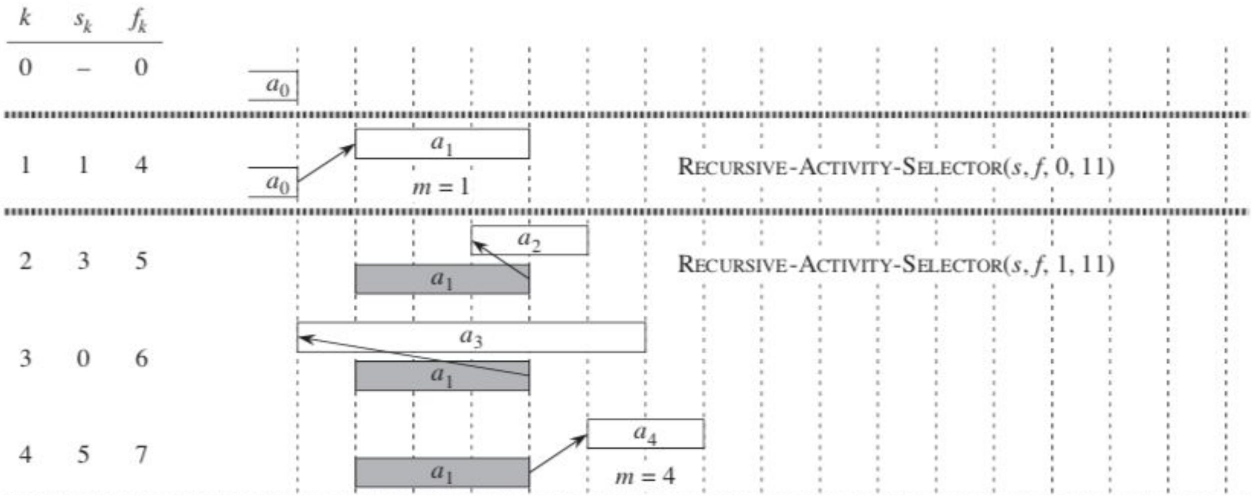
```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]        // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```



Activity-Selection: Recursive greedy algorithm

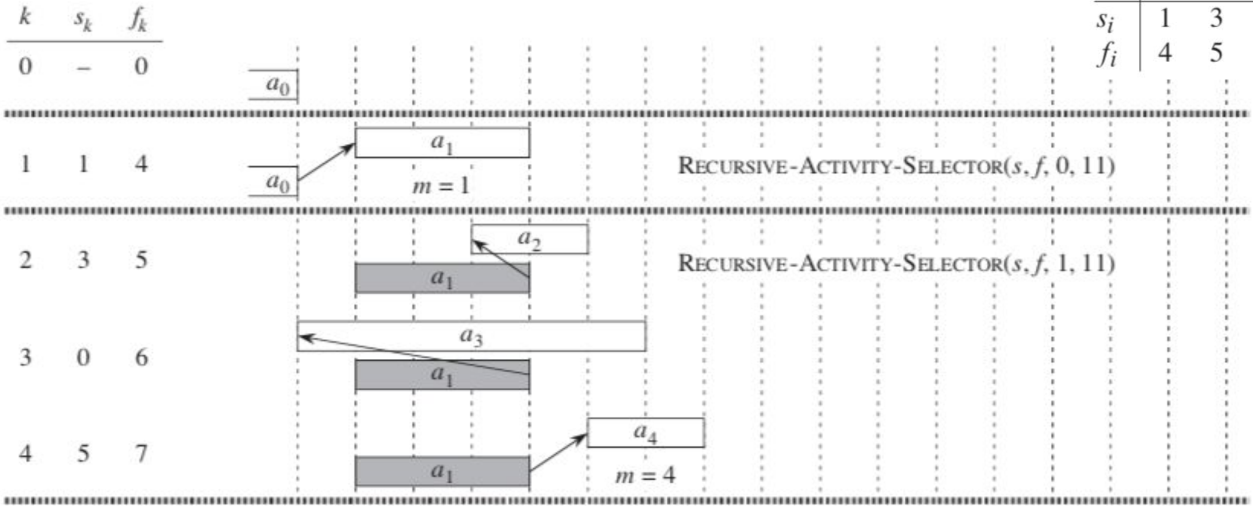
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

```
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
m = k + 1
while m ≤ n and s[m] < f[k]      // find the first activity in S_k to finish
    m = m + 1
if m ≤ n
    return {a_m} ∪ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
else return Ø
```



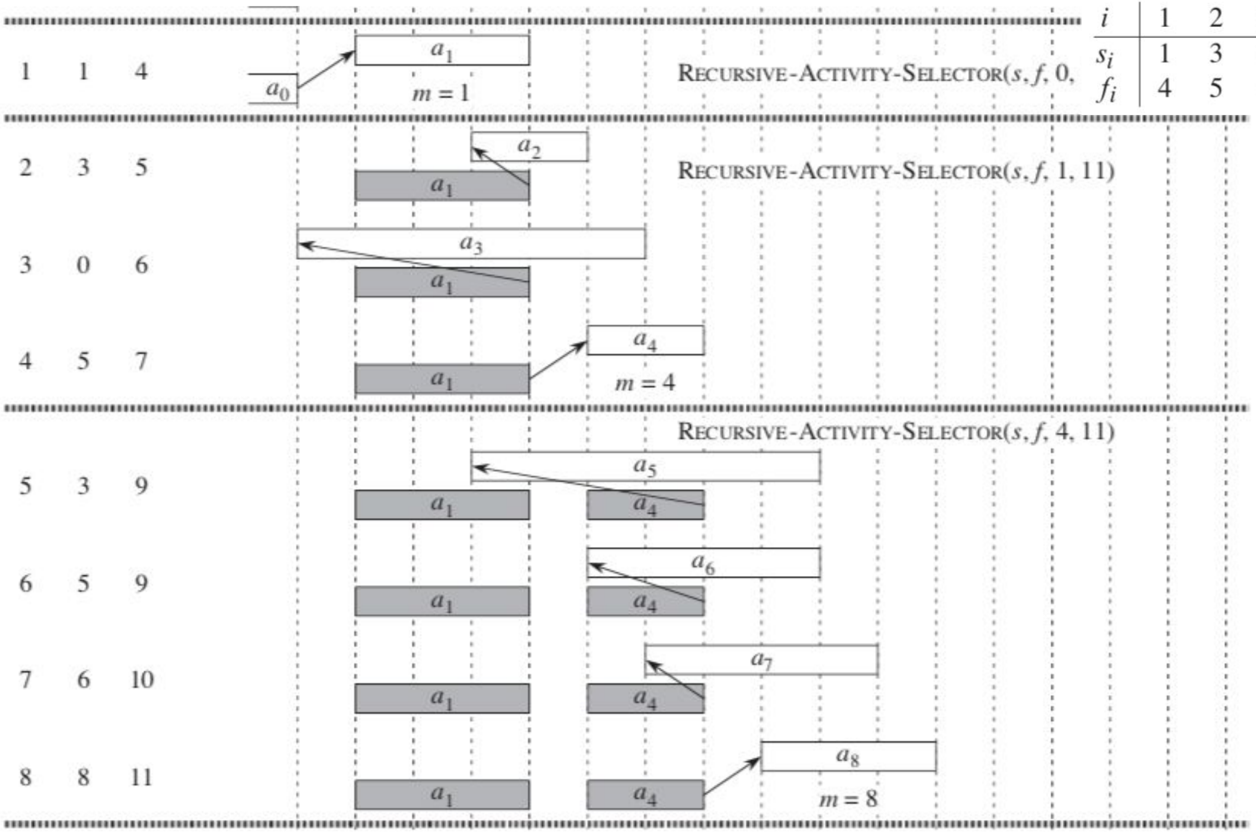
Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



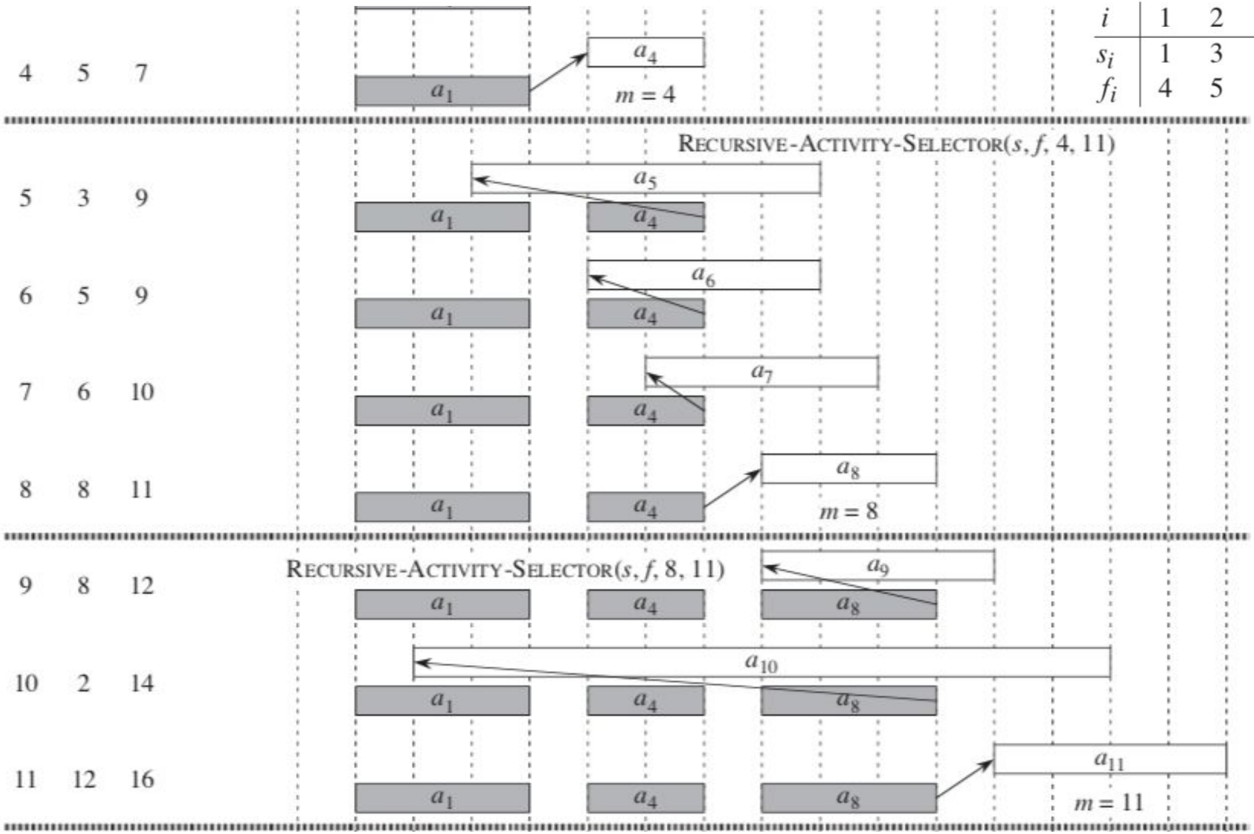
Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



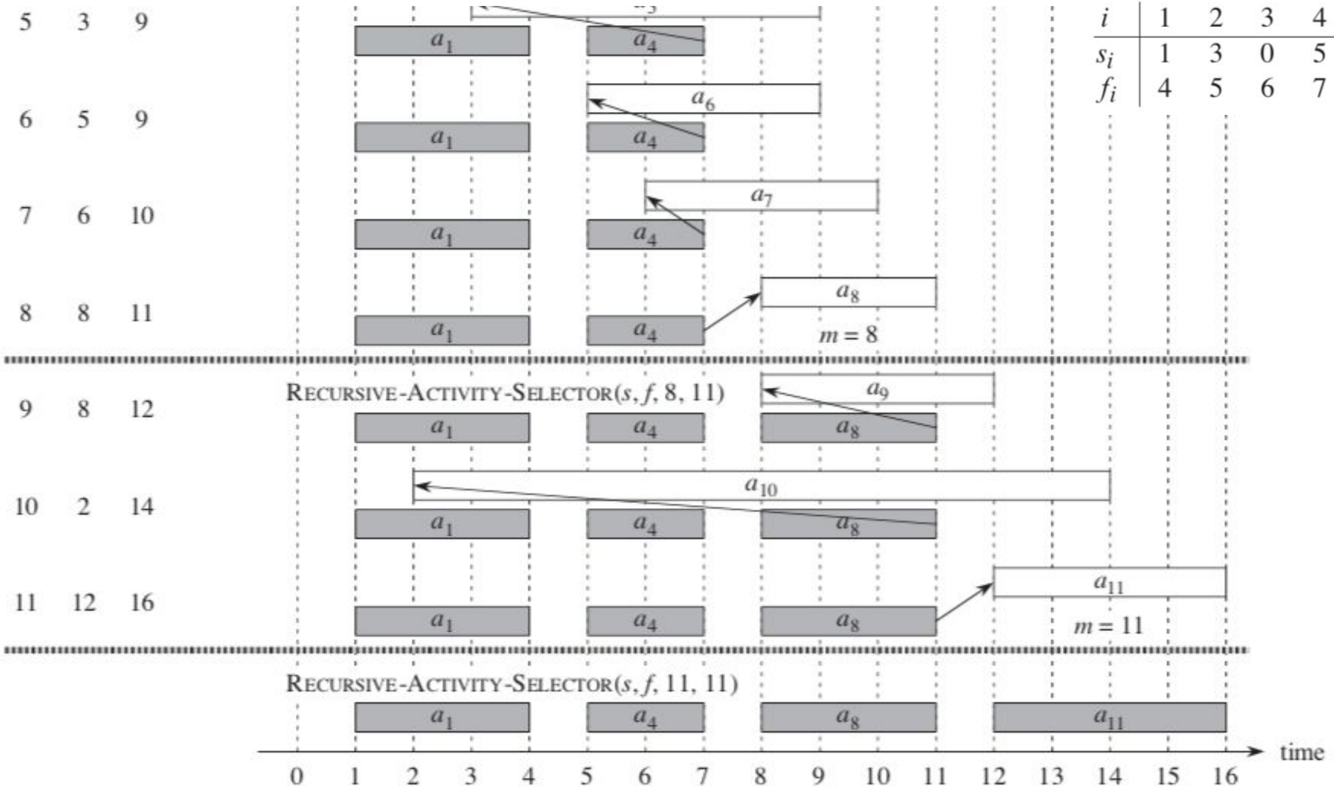
Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity-Selection: Recursive greedy algorithm

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16



Activity-Selection: Iterative greedy algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n = s.length
A = {a_1}
k = 1
for m = 2 to n
    if s[m] ≥ f[k]
        A = A ∪ {a_m}
        k = m
return A
```

Divide-and-conquer

Divide-and-conquer

The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

Divide-and-conquer examples

- Merge sort (already studied)
- Quick sort (already studied)
- Power algorithm:

  Compute b^n as $b^{n/2} * b^{n/2}$

  $b^n = 1$ if $n = 0$

  $b^n = b^{n/2} * b^{n/2}$ if $n > 0$ and n is even

  $b^n = b * b^{n/2} * b^{n/2}$ if $n > 0$ and n is odd

Power algorithm using divide-and-conquer strategy

```
power(b, n)
{
    if (n == 0)
        return 1;
    else {
        int p = power(b, n/2);
        if (n % 2 == 0)
            return p * p;
        else
            return b * p * p;
    }
}
```

Optimal substructure

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

Assignment

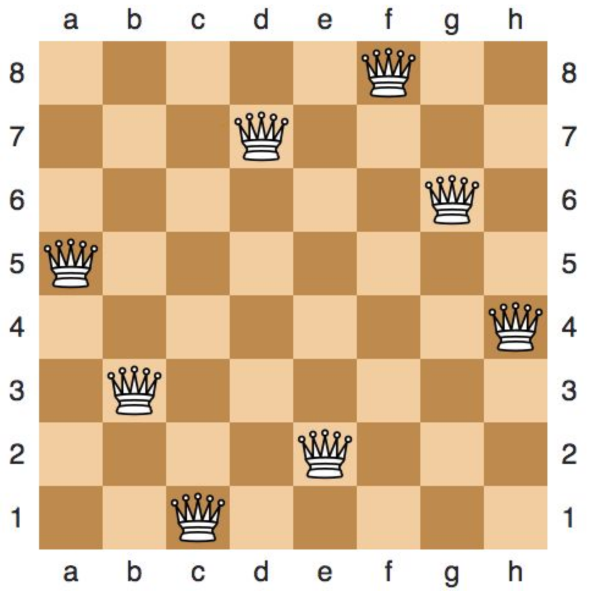
1. Improve the power algorithm to work with negative powers, i.e. it should be able to calculate a^{-n}.

Backtracking

Backtracking

Examples of problems which can be solved using backtracking:

1. **N-queens problem**
   The problem of placing N chess queens on an NxN chessboard so that no two queens attack each other.



Source: https://en.wikipedia.org/wiki/Eight_queens_puzzle

Backtracking

Examples of problems which can be solved using backtracking (Contd.):

2. **Sudoku**
   The problem of filling a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 subgrids that compose the grid contains all of the digits from 1 to 9.

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

Source: https://en.wikipedia.org/wiki/Sudoku

Backtracking

Examples of problems which can be solved using backtracking (Contd.):

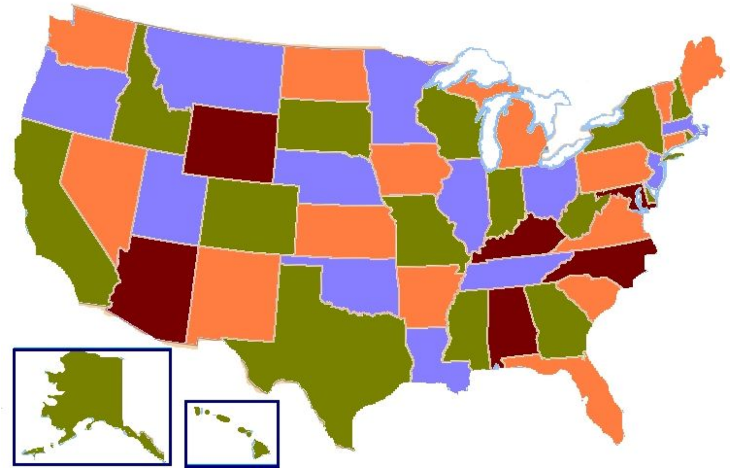
3. **The Knight's tour problem**
   Moving a knight on a chessboard such that the knight visits every square only once.

Source: https://en.wikipedia.org/wiki/Knight%27s_tour

Backtracking

Examples of problems which can be solved using backtracking (Contd.):

4. **Map coloring**
   Coloring each country/state/entity in a map with a color from the given set of colors, such that no two adjacent countries/states/entities have the same color.



Source: https://en.wikipedia.org/wiki/Four_color_theorem

Backtracking

Examples of problems which can be solved using backtracking (Contd.):

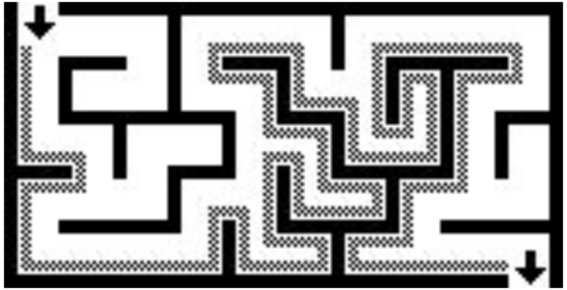
5. **Maze solving**

   Given a maze, find a path from start to finish.

   At each intersection, one has to decide between

   Three or fewer choices:

   a. Go straight
   b. Go left
   c. Go right



Source: https://en.wikipedia.org/wiki/Maze_solving_algorithm

Backtracking

An approach to solving problems which requires that a series of decision, among various choices, be made where

- We don't have enough information to know what to choose
- Each decision leads to a new set of choices
- Some sequence of choices may be a solution to our problem

Backtracking

An approach to solving constraint-satisfaction problems without trying all possibilities.

Constraint-satisfaction problems require that all the solutions satisfy a complex set of constraints.

Constraints may be explicit or implicit.

Backtracking

The desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$ where the x_i are chosen from some finite set S_i.

Often the problem to be solved calls for finding one vector that maximizes (minimizes/satisfies) a criterion function $P(x_1, \ldots, x_n)$.

Example:

All solutions to the N-queens problem can be represented as n-tuples $(x_1, \ldots x_n)$, where x_i is the column on which each queen i is placed.

Backtracking

All solutions to the N-queens problem can be represented as n-tuples $(x_1, \ldots x_n)$, where x_i is the column on which each queen i is placed.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

One solution to the 4-queens problem is (3, 1, 4, 2).

Backtracking

Explicit constraints are rules that restrict each x_i to take on values only from a given set.

Example:

In the N-queens problem, explicit constraints are:

$S_i = \{1, 2, \ldots, n\}$

$1 \leq x_i \leq n$

Backtracking

Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function.

They describe the way in which the x_i must relate to each other.

Example:

In the N-queens problem, the implicit constraints are

1. No two x_i's can be the same.
2. No two queens can be on the same diagonal.

N-Queens problem

- N queens are to be placed on an N x N chessboard without attacking each other.
- All solutions to the N-queens problem can be represented as n-tuples (x1, ... xn), where xi is the column on which each queen i is placed.
- Explicit constraints:
  - $S_i = \{1, 2, ..., n\}$
  - $1 \leq x_i \leq n$
- Implicit constraints:
  - No two xi's can be the same.
  - No two queens can be on the same diagonal

N-Queens problem

Possible solutions / Solution space:

(1, 2, 3, 4)

(1, 3, 2, 4)

(1, 3, 4, 2)

and so on.

That is, the solution space consists of all n! Permutations of the n-tuple (1, 2, ... , n).

Permutation tree

Searching the solution space is facilitated by using a tree organization.

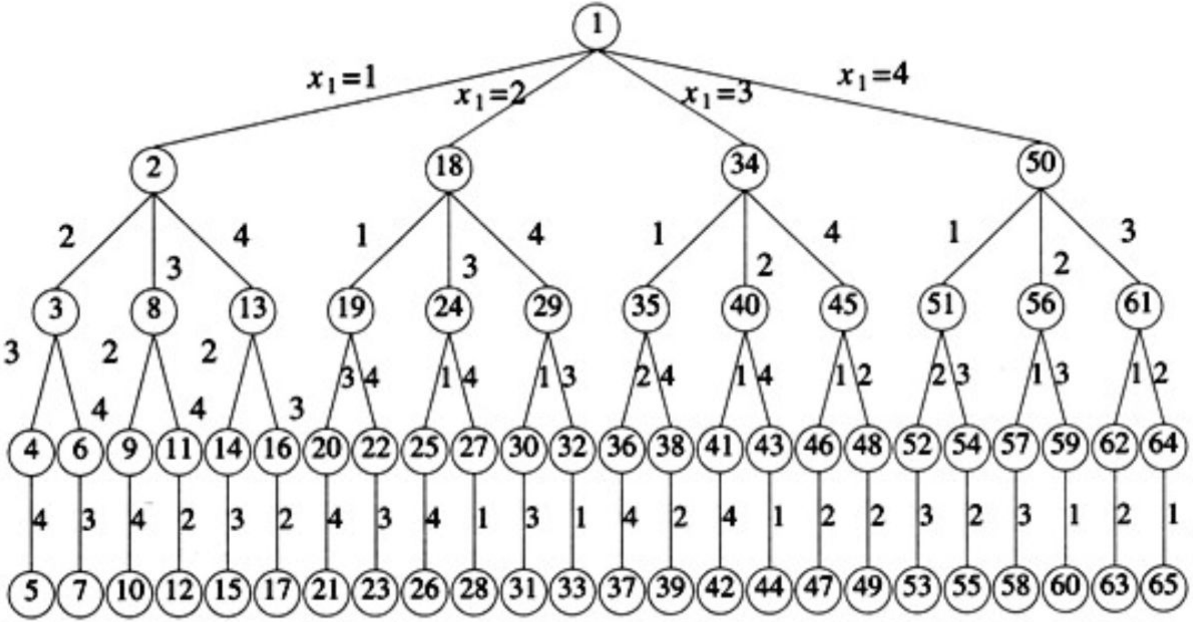
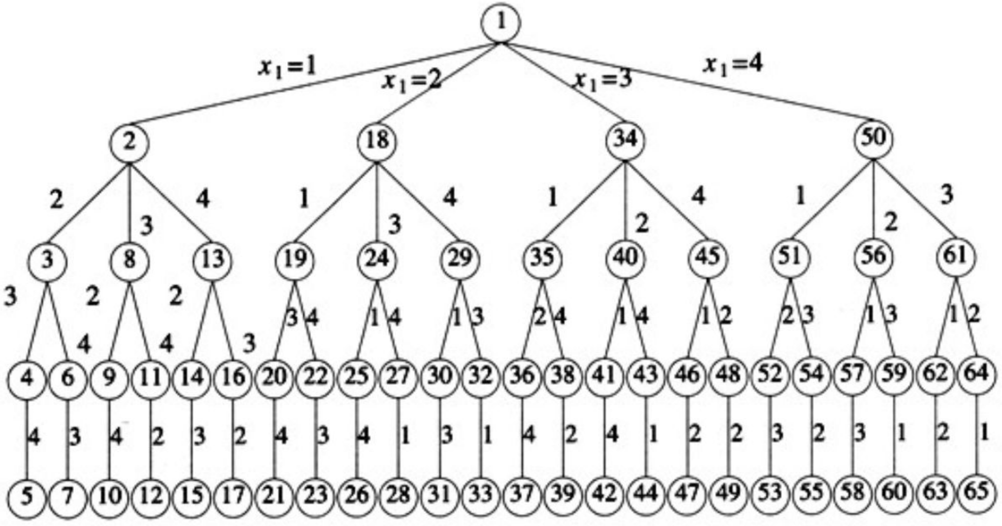


Fig: State space tree of 4-queens problem. Nodes are numbered as in depth first search.

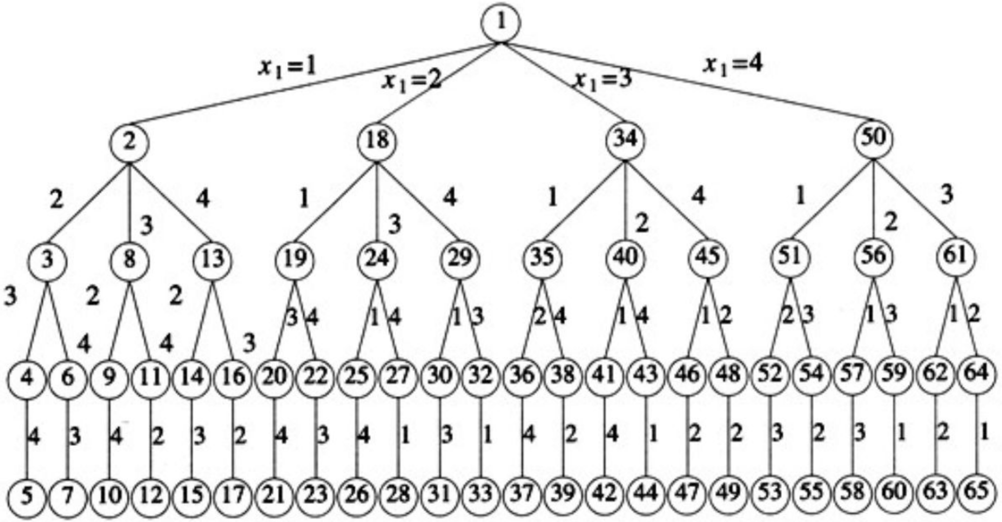
Permutation tree

A brute-force algorithm searches the whole tree, but with backtracking, we got to throw away massive parts of the tree (prune the tree) when we discover a partial solution cannot be extended to a complete solution.



Permutation tree

- **Edges** from level i to i+1 : possible values of xi
- Each **node** is a **partial solution**
- **Solution space** is defined by all paths from the root node to a leaf node.



N-queens problem: Backtracking algorithm

1. Place a queen on the first available square in row 1.
2. Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).
3. Continue in this fashion until either:
    a. you have solved the problem, or
    b. you get stuck.
    When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.

q_1			
		q_2	

N-queens problem: Backtracking algorithm

The problem can be solved by systematically generating nodes of the state space tree, determining which of the nodes are the solutions.

While exploring the tree, the tree is pruned if the partial solution does not satisfy the constraints.

Backtracking begins with the root and generate other nodes in depth-first manner.

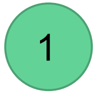
N-queens problem: Backtracking algorithm

Terminologies

- **Live node** : a node which has been generated and all of whose children have not yet been generated.
- **E-node** : A live node whose children are currently being generated
- **Dead node** : A generated node which is not to be expanded further or all of whose children have been generated.

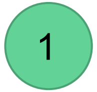
N-queens problem: Backtracking algorithm

We start with the root node as the only live node. This becomes the E-node and the path is ().

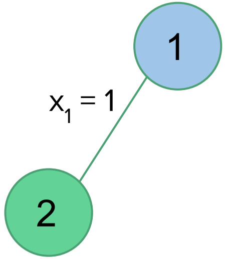


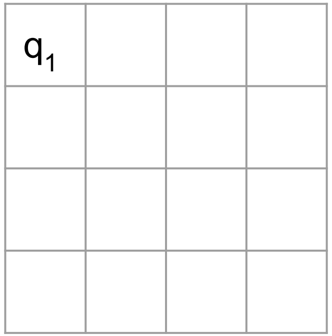
N-queens problem: Backtracking algorithm

We start with the root node as the only live node. This becomes the E-node and the path is ().



We generate one child (Node 2) and the path is (1). This corresponds to placing a queen 1 on column 1.

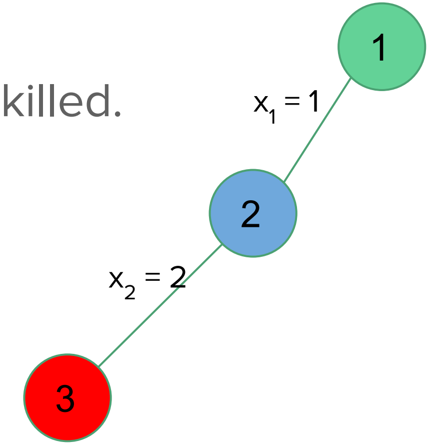


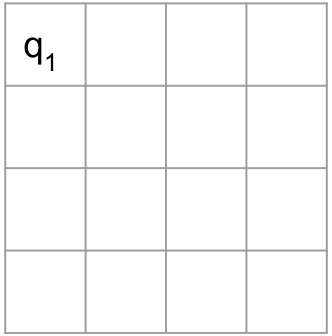


N-queens problem: Backtracking algorithm

Node 2 becomes the E-node.

Node 3 is generated an immediately killed.

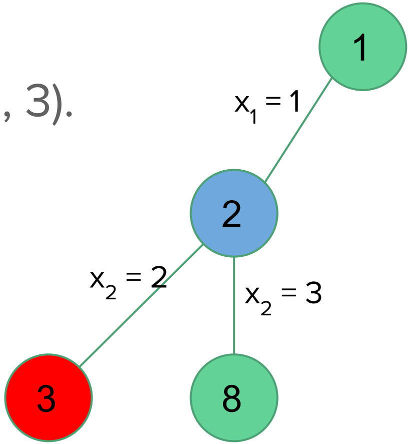


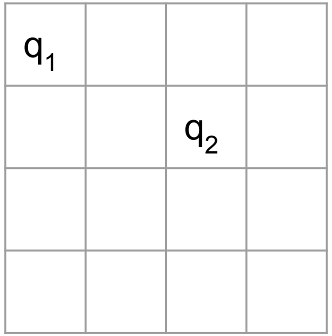


N-queens problem: Backtracking algorithm

Node 2 is still the E-node.

Node 3 is generated and the path is (1, 3).

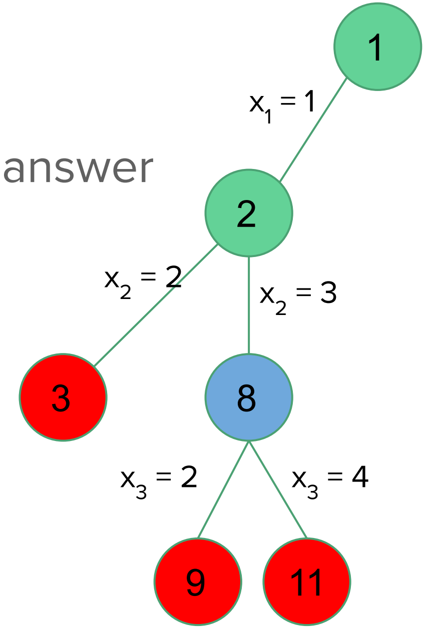


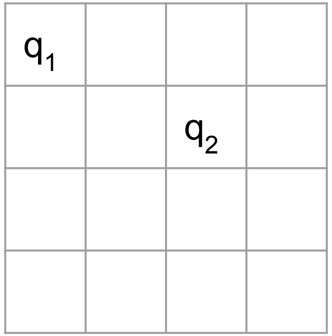


N-queens problem: Backtracking algorithm

Node 8 becomes the E-node.

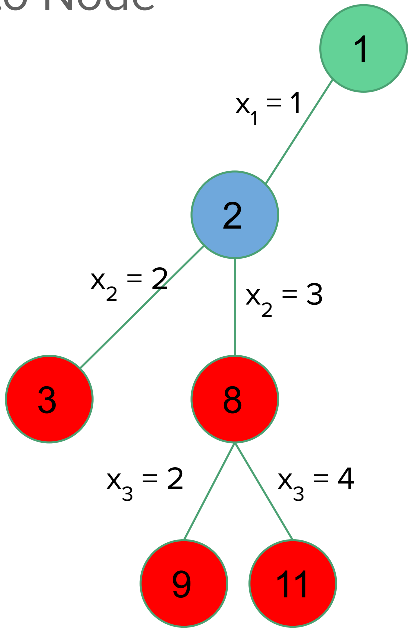
All of its children represent board configurations that cannot lead to an answer node.





N-queens problem: Backtracking algorithm

Node 8 gets killed and we backtrack to Node 2.

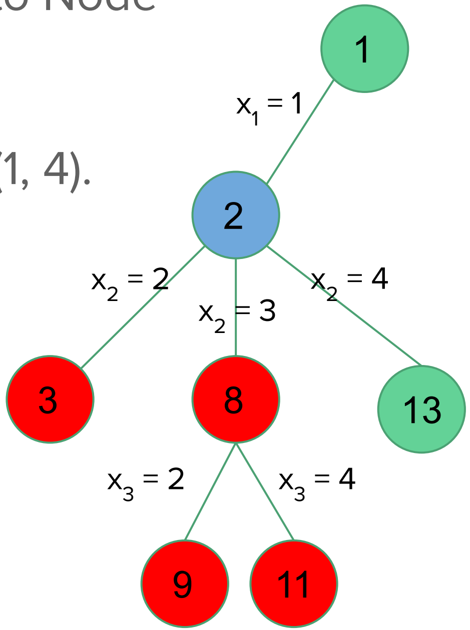


q_1			

N-queens problem: Backtracking algorithm

Node 8 gets killed and we backtrack to Node 2, which then becomes the E-node.

Node 13 is generated and the path is (1, 4).

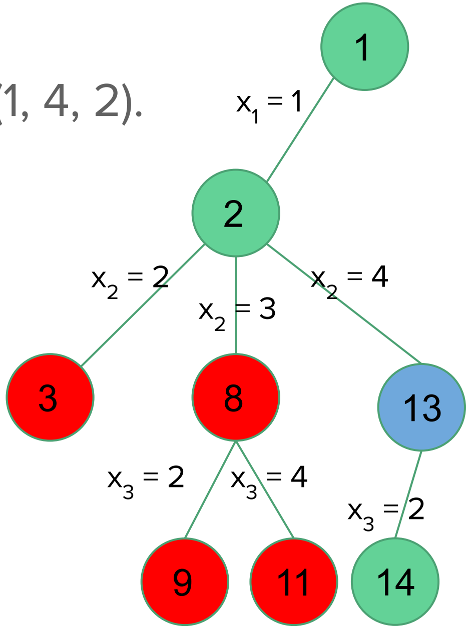


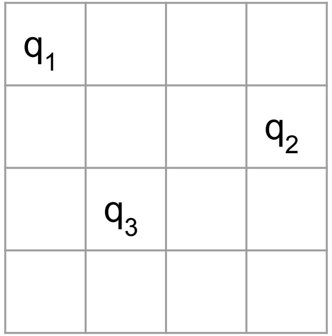
q_1			
			q_2

N-queens problem: Backtracking algorithm

Node 13 becomes the E-node.

Node 14 is generated and the path is (1, 4, 2).

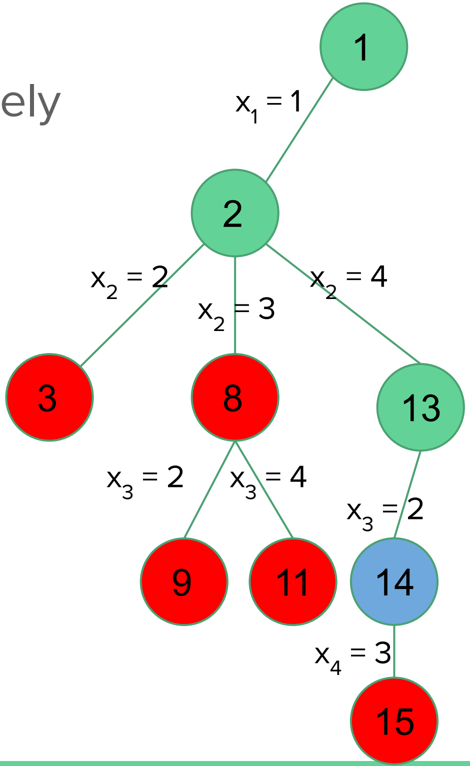




N-queens problem: Backtracking algorithm

Node 14 becomes the E-node.

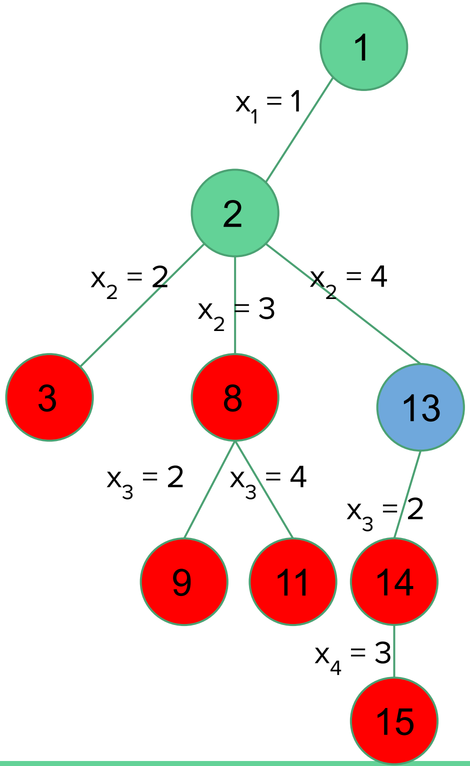
Node 15 is generated and is immediately killed.



q_1			
			q_2
	q_3		

N-queens problem: Backtracking algorithm

We backtrack to Node 13.

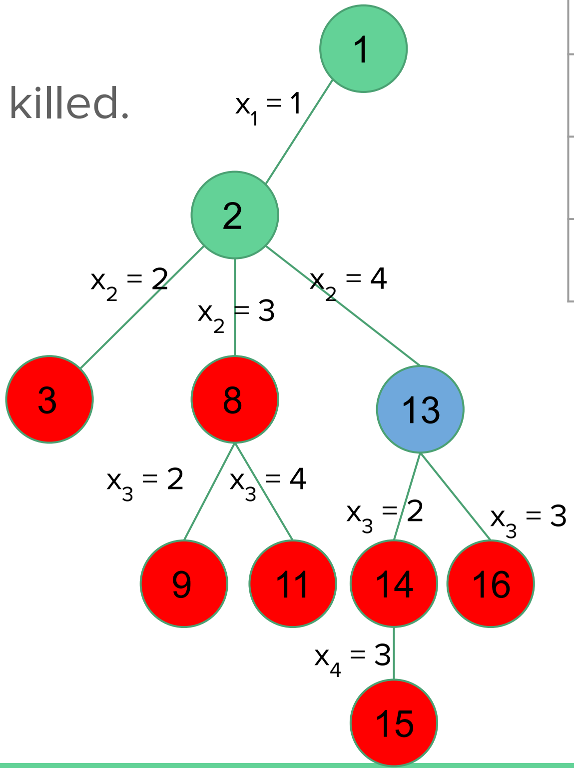


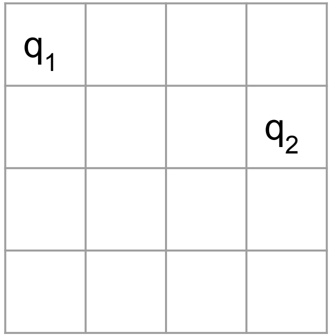
q_1			
			q_2

N-queens problem: Backtracking algorithm

Now the E-node is Node 13.

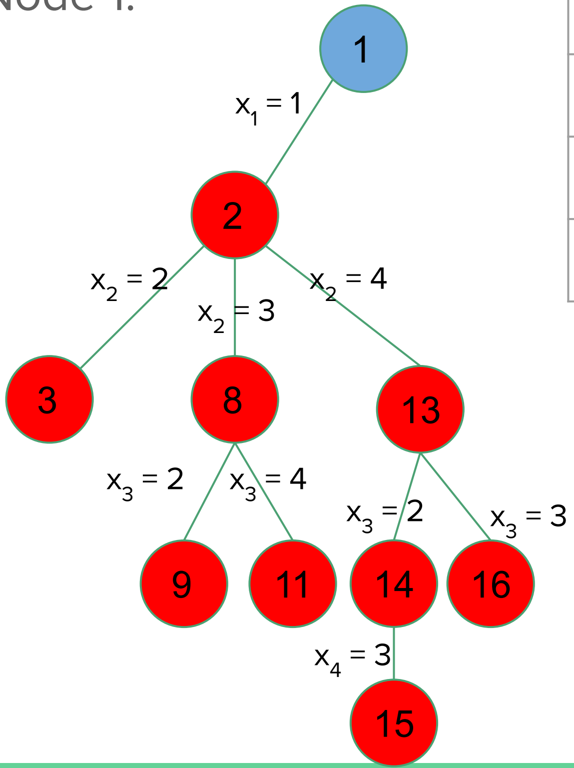
Its another child, Node 16, will also be killed.

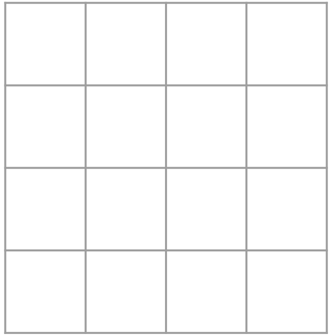




N-queens problem: Backtracking algorithm

We backtrack to Node 2 and then to Node 1.

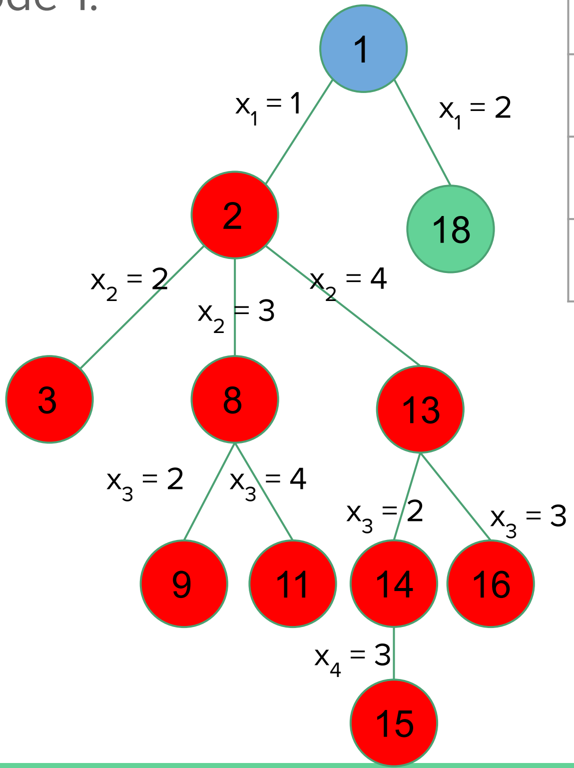


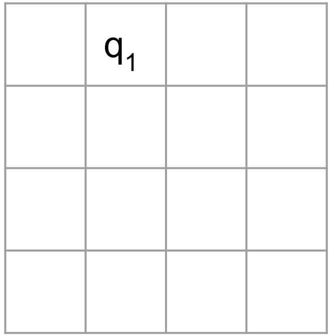


N-queens problem: Backtracking algorithm

Now, we generate another child of Node 1.

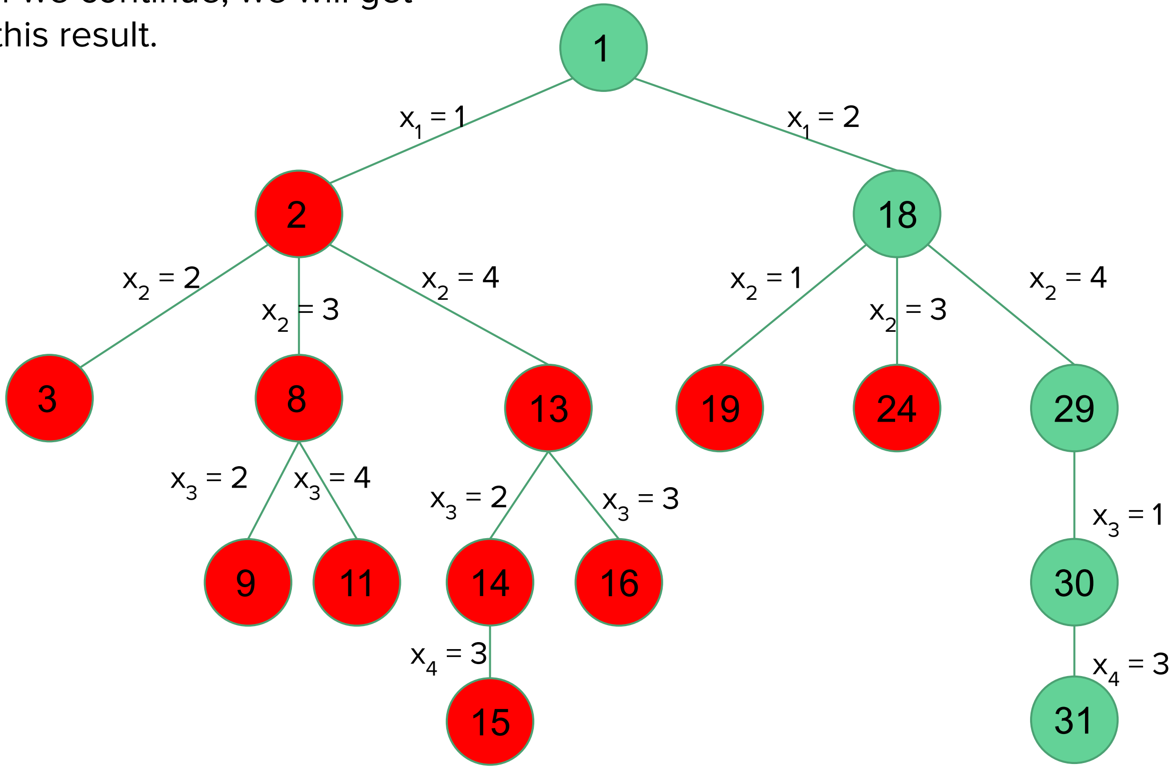
And the path will be (2).





N-queens problem: Backtracking algorithm

If we continue, we will get this result.



	q_1		
			q_2
q_3			
		q_4	

The solution is (2, 4, 1, 3).

N-queens problem: Backtracking algorithm

```
Algorithm NQueens(k, n)
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × n
// chessboard so that they are nonattacking.
{
    for i := 1 to n do
    {
        if Place(k, i) then
        {
            x[k] := i;
            if (k = n) then write (x[1 : n]);
            else NQueens(k + 1, n);
        }
    }
}
```

```
Algorithm Place(k, i)
// Returns true if a queen can be placed in kth row and
// ith column. Otherwise it returns false. x[ ] is a
// global array whose first (k − 1) values have been set.
// Abs(r) returns the absolute value of r.
{
    for j := 1 to k − 1 do
        if ((x[j] = i) // Two in the same column
            or (Abs(x[j] − i) = Abs(j − k)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Branch-and-bound

Recall

A **state space tree** consists of

- a set of nodes representing each state of the problem,
- arcs between nodes representing the legal moves from one state to another,
- an initial state and
- a goal state

In state space tree methods of problem solving, we first represent the problem as a state space tree and then search the tree to find the solution to the problem by systematically generating nodes of the tree.

Recall

Terminologies

- A node which has been generated and all of whose children have not yet been generated is called a **live node**
- The live node whose children are being generated is called the **E-node**
- A **dead node** is a generated node which is not to be expanded further or all of whose children have generated

Graph search strategies

- Depth-first search
- Breadth-first search

Branch and bound

In **backtracking**, nodes are generated in depth-first manner, i.e. as soon as a new child of the current E-node is generated, this child will become the new E-node

In **branch-and-bound**, nodes are generated in breadth-first manner, i.e. E-node remains the E-node until it is dead.

Branch-and-bound

- A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node
- Common search techniques
  - FIFO search
  - LIFO search
  - Least-cost search

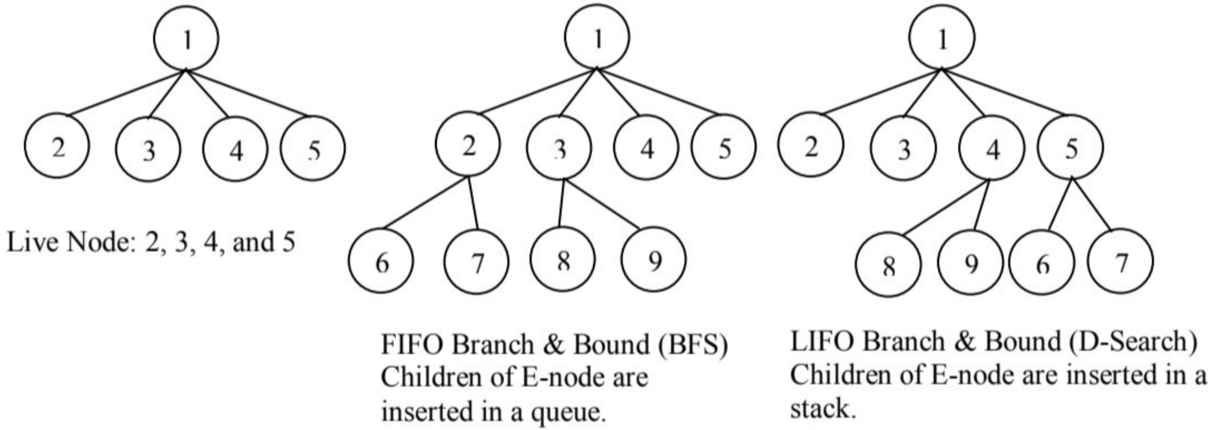
State space tree search

FIFO search

The list of live nodes is a FIFO list

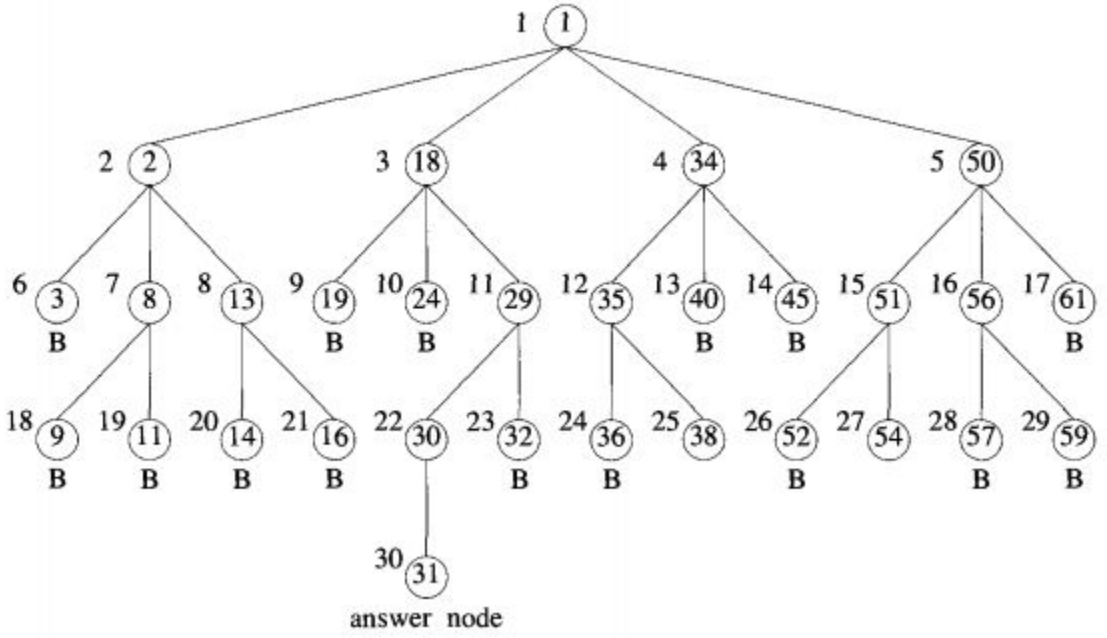
LIFO search

The list of live nodes is a LIFO list



FIFO search for 4-queens problem

FIFO search for 4-queens problem



State space tree search

Least cost (LC) search

- In FIFO and LIFO branch and bound, the selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer quickly
- The search for an answer node can often be speeded by using an "intelligent" ranking function, also called an **approximate cost function,** $\hat{c}$, for live nodes
- In **LC search**, we expand the node with the best cost

Branch-and-bound

- A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node
- We assume that each answer node x has a cost c(x) associated with it
- The goal is to find the minimum-cost answer node

Branch-and-bound

- A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node
- Requirements
  - **Branching**: A set of solutions, which is represented by a node, can be partitioned into mutually exclusive sets. Each subset in the partition is represented by a child of the original node.
  - **Lower bounding**: An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

0/1 Knapsack problem

We are given n objects and a knapsack or bag. Object i has a weight w_i and value p_i. The knapsack has a capacity m. If an object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

0/1 Knapsack problem

In **0/1 Knapsack problem**, each object is either included or excluded in the knapsack, i.e. the object cannot be divided.

In **fractional knapsack problem**, fractions of objects can be included in the knapsack.

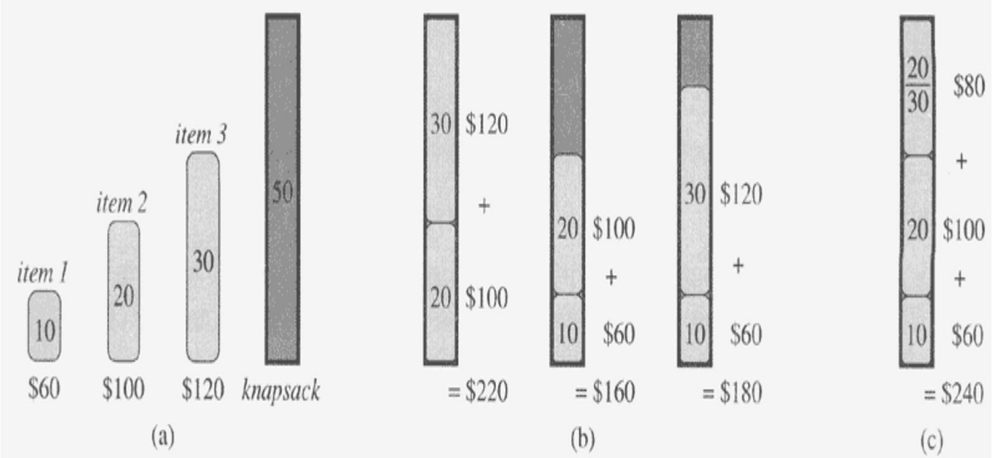


Fig:
(a) A knapsack problem
(b) Solutions for 0/1 knapsack problem
(c) A solution for fractional knapsack problem

0/1 Knapsack problem

We are given n objects and a knapsack or bag. Object i has a weight w_i and value p_i. The knapsack has a capacity m. If an object i is placed into the knapsack, then a profit of $p_i x_i$ is earned. The objective is to obtain a filling of the knapsack that maximizes the total profit earned.

Since LC BB deals with minimization problems, we transform this maximization problem into a minimization problem as follows:

$$\text{minimize} \ -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \ \sum_{i=1}^{n} w_i x_i \leq m$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

0/1 Knapsack problem

Consider the following knapsack problem:

4 objects are to be put in a knapsack of capacity 16. The objects have the following values and weights

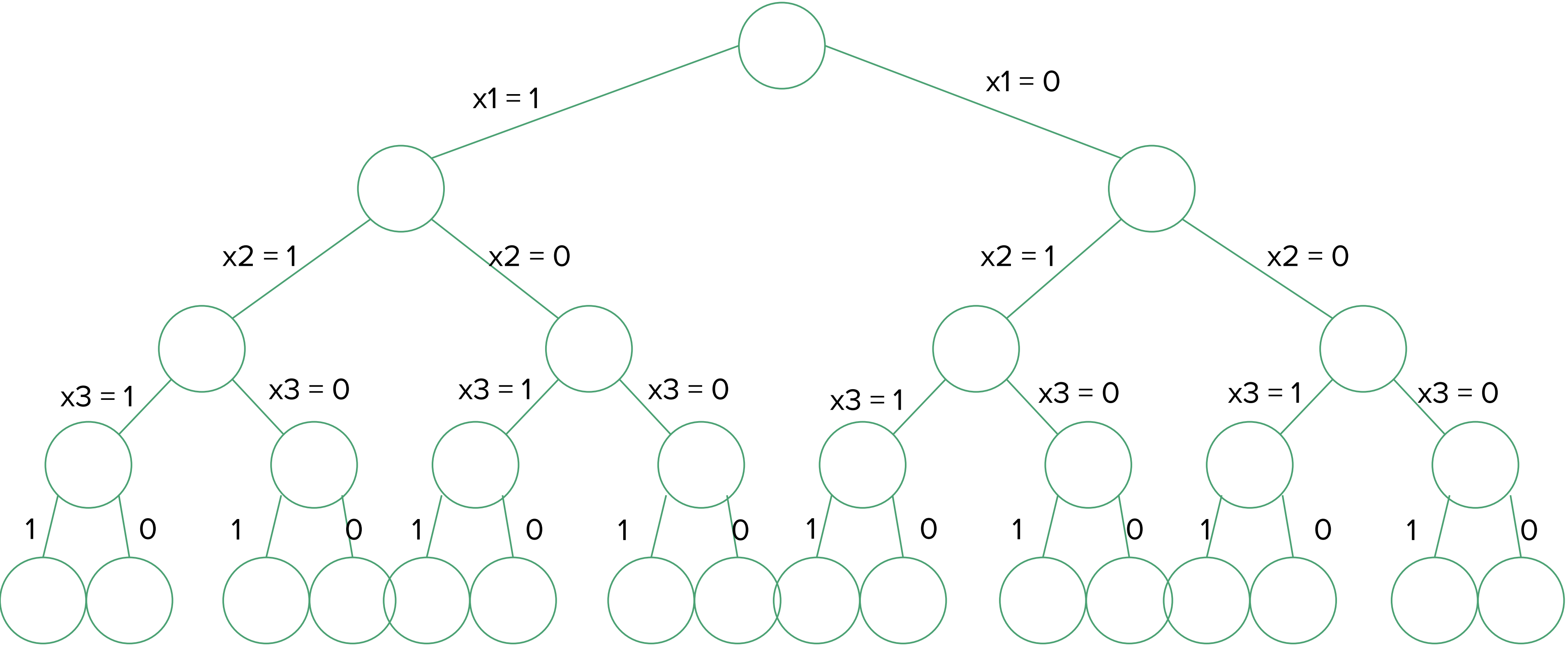
p	45	30	45	10
w	3	5	9	5

Let the solution of this problem be an n-tuple $(x_1, x_2, \ldots x_n)$ where $x_i \in \{0, 1\}$

$x_i = 0$ if object i is not taken

$x_i = 1$ if object i is taken

State space tree



LC branch-and-bound solution

Every leaf node in the state space tree representing an assignment for which $\sum_{i=1}^{n} w_i x_i \leq m$ is an **answer node**

For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{i=1}^{n} p_i x_i$ for every answer node

LC branch-and-bound

We need two functions, ĉ(.) and u(), (for lower and upper bounds) such that

$$\hat{c}(x) \leq c(x) \leq u(x)$$

For 0/1 knapsack problem,

$$u(x) = -\sum_{i=1}^{n} p_i x_i$$

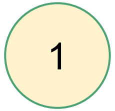
$$\hat{c}(x) = -\sum_{i=1}^{n} p_i x_i \quad \text{(with fraction)}$$

If `upper` is an upper bound on the cost of a minimum-cost solution, all live nodes with `ĉ(x) > upper` may be killed

The starting value of `upper` can be obtained by some heuristics or can be set to ∞

LC branch-and-bound solution

p	45	30	45	10
w	3	5	9	5



$u = -75$
$\hat{c} = -115$

LC BB starts with upper = ∞

$u(1) = -(45 + 30) = -75$
Remaining capacity = $16 - 3 - 5 = 8$
$\hat{c}(1) = -(45 + 30 + 45/9 \times 8) = -115$

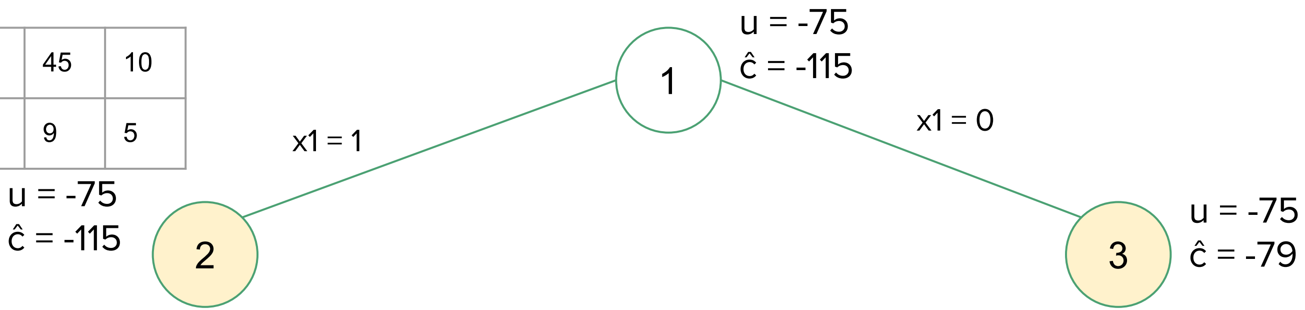
upper = -75

Live nodes: {1}

LC branch-and-bound solution

p	45	30	45	10
w	3	5	9	5

upper = -75



$u(2) = -(45 + 30) = -75$
Remaining capacity = 16 - 3 - 5 = 8
$\hat{c}(2) = -(45 + 30 + 45/9 \times 8) = -115$

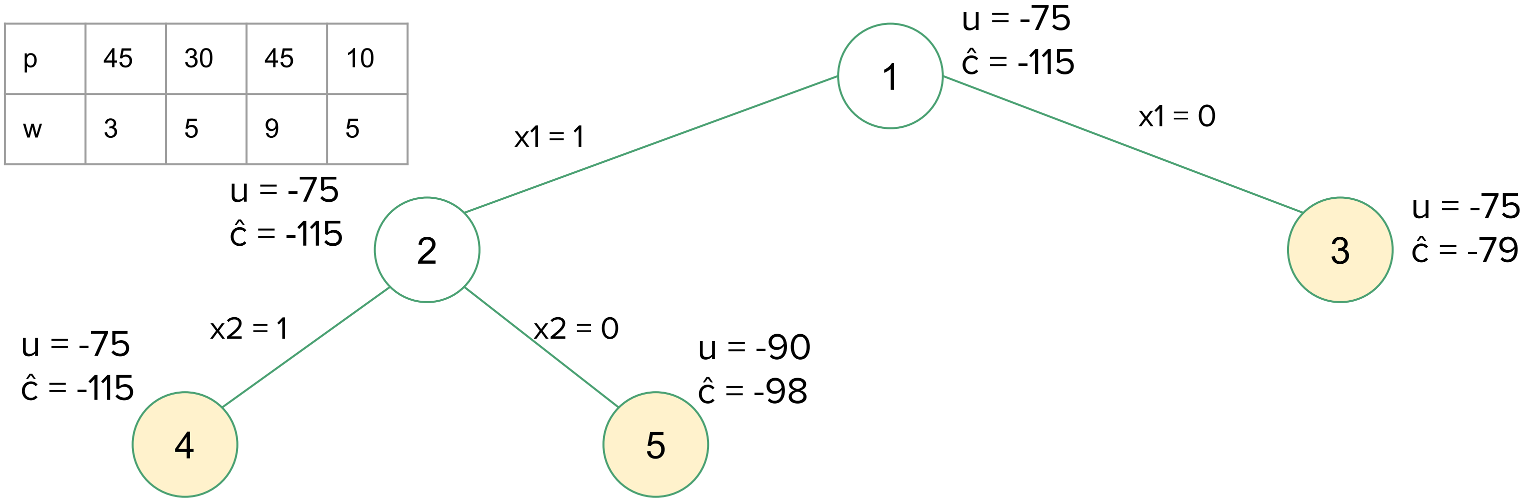
$u(3) = -(30 + 45) = -75$
Remaining capacity = 16 - 5 - 9 = 2
$\hat{c}(3) = -(30 + 45 + 10/5 \times 2) = -79$

Live nodes: {2, 3}
$\hat{c}(2)$ is the lowest

LC branch-and-bound solution

upper = -75

p	45	30	45	10
w	3	5	9	5



u(4) = -(45 + 30) = -75
Remaining capacity = 16 - 3 - 5 = 8
ĉ(4) = -(45 + 30 + 45/9x8) = -115

u(5) = -(45 + 45) = -90
Remaining capacity = 16 - 3 - 9 = 4
ĉ(5) = -(45 + 45 + 10/5x4) = -98

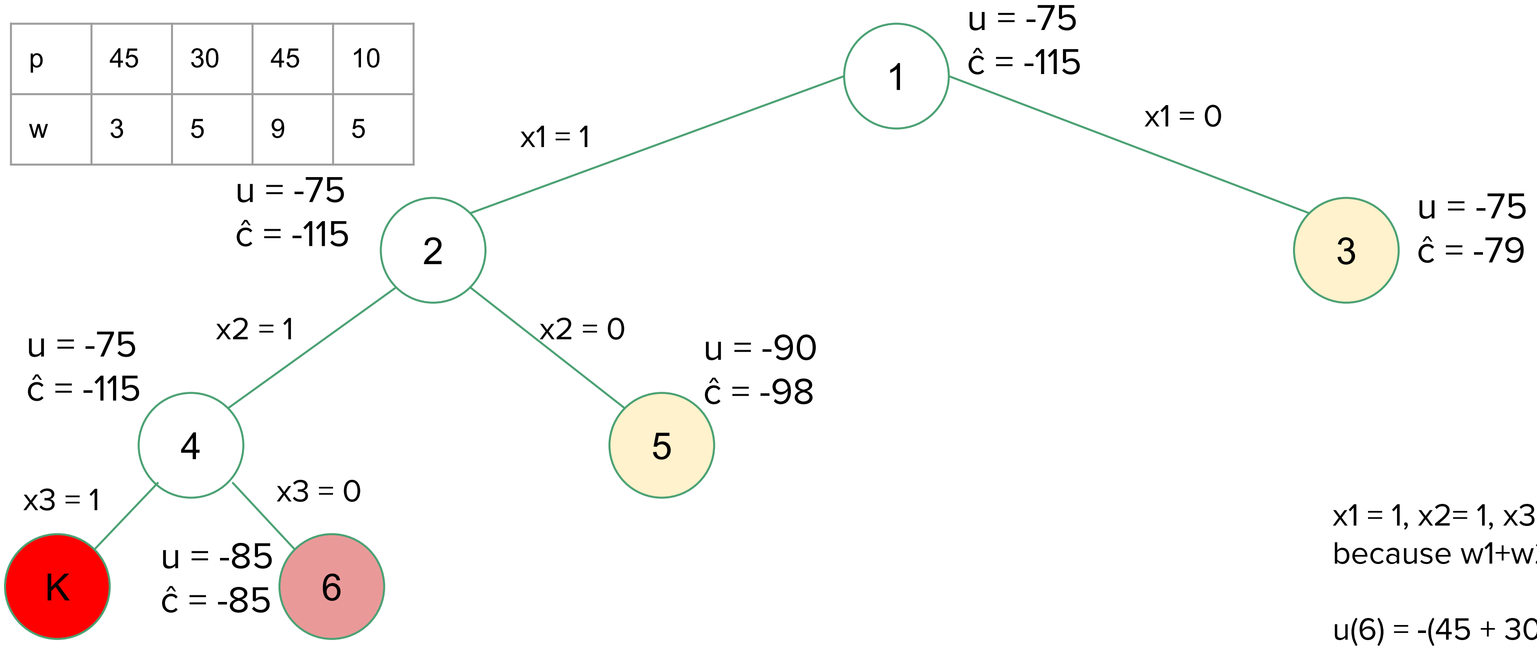
upper = -90

Live nodes: {3, 4, 5}
ĉ(4) is the lowest

LC branch-and-bound solution

p	45	30	45	10
w	3	5	9	5

upper = -90



x1 = 1, x2= 1, x3 = 1 is not possible because w1+w2+w3 exceeds 16.

u(6) = -(45 + 30 + 10) = -85
Remaining capacity = 16 - 3 - 5 = 8
ĉ(6) = -(45 + 30 + 10) = -85

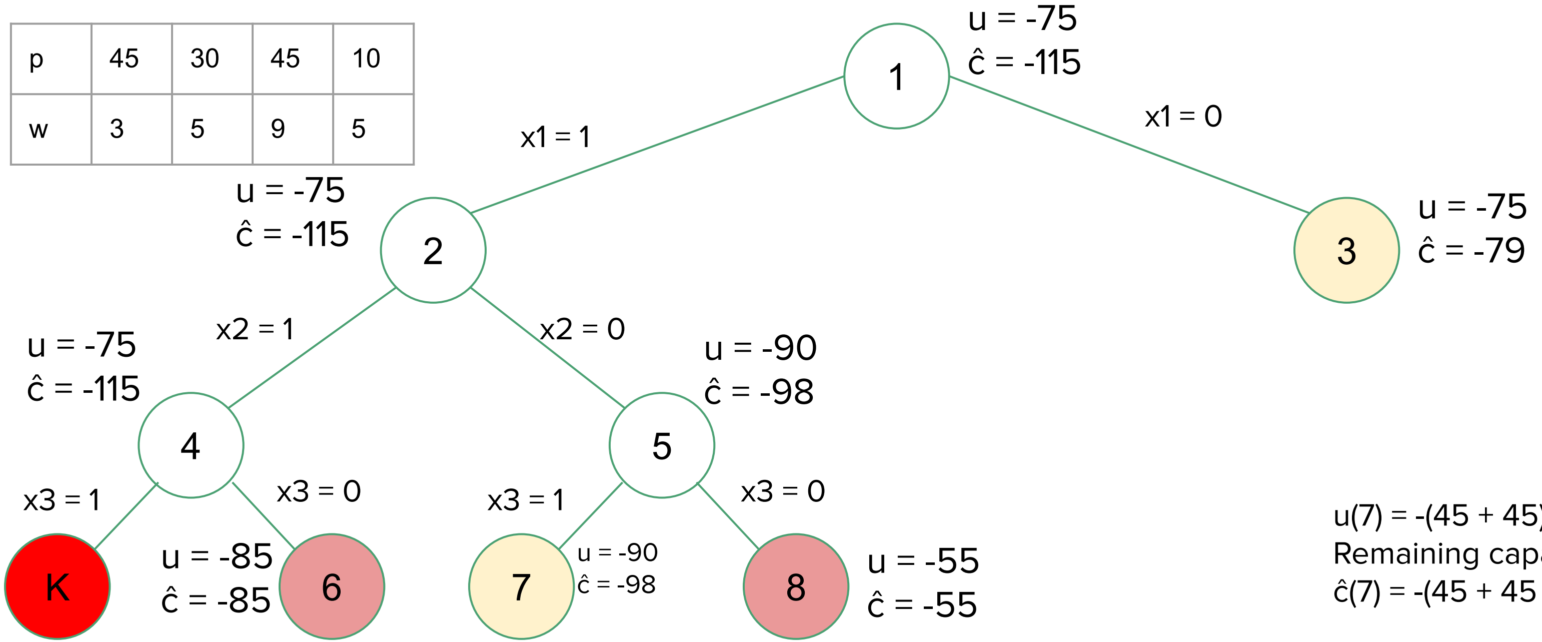
ĉ(6) > upper. So Node 6 is killed.

Live nodes: {3, 5}
ĉ(5) is the lowest

LC branch-and-bound solution

upper = -90

p	45	30	45	10
w	3	5	9	5



Live nodes: {3, 7}
ĉ(7) is the lowest

u(7) = -(45 + 45) = -90
Remaining capacity = 16 - 3 - 9 = 4
ĉ(7) = -(45 + 45 + 10/5x4) = -98

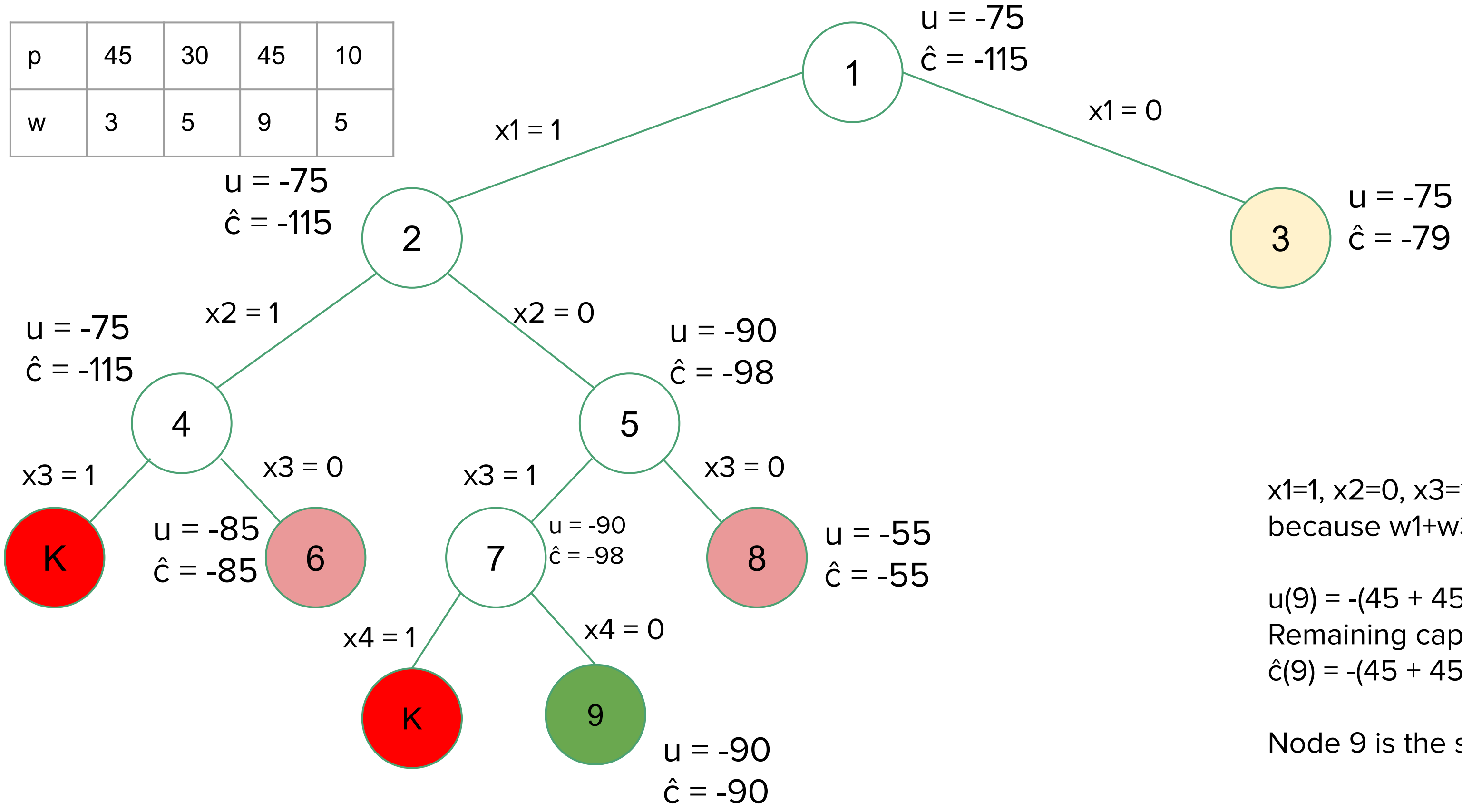
u(8) = -(45 + 10) = -55
Remaining capacity = 16 - 3 - 5 = 8
ĉ(8) = -(45 + 10) = -55

ĉ(8) > upper. So Node 8 is killed.

LC branch-and-bound solution

upper = -90

p	45	30	45	10
w	3	5	9	5

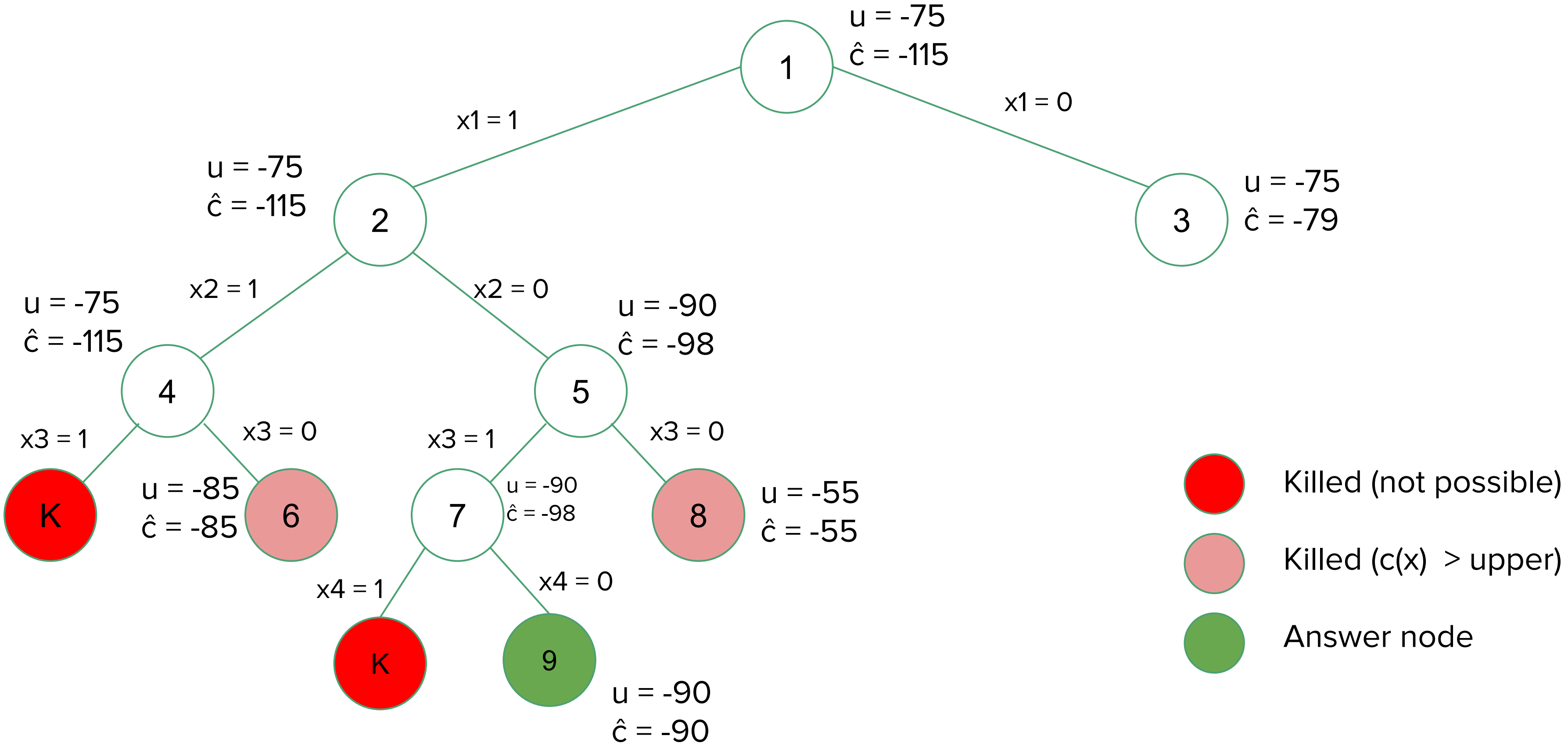


$x_1=1, x_2=0, x_3=1, x_4=1$ is not possible because $w_1+w_3+w_4 = 3 + 9 + 5 > 16$

$u(9) = -(45 + 45) = -90$
Remaining capacity = $16 - 3 - 9 = 4$
$\hat{c}(9) = -(45 + 45) = -90$

Node 9 is the solution.

LC branch-and-bound solution



Heuristics

Heuristics

- A **heuristic** is a technique designed or *solving a problem more quickly when classic methods are too slow,* or for finding *an approximate solution* when classic methods fail to find any exact solution
- It guides an algorithm to find good choices without exhaustively searching all possible solutions
- A **heuristic function** is a function that ranks alternatives in search algorithms at each branching step based on available information to decide which branch to follow.

Examples of heuristics

Travelling Salesman Problem (TSP):

- "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?"
- **The nearest-neighbor heuristic**: pick the **nearest** unvisited city as the next city on the path.

Knapsack problem:

- **Heuristics:** Sort the items by value/weight ratio, then select the next item with the highest ratio.

Examples of heuristics

Antivirus:

- **Heuristic scanning** looks for code and/or behavioral patterns common to a class or family of viruses, with different sets of rules for different viruses

A* search (pronounced A star)

- A search algorithm that finds the shortest path between some nodes in a graph
- (will be covered in Chapter Graph Algorithms)