Week 7: Wednesday

EECS 281

MAXIM ALEKSA

maximal.io

Agenda

Algorithm Design

Brute force

Greedy

Divide and conquer

Backtracking

Branch and Bound

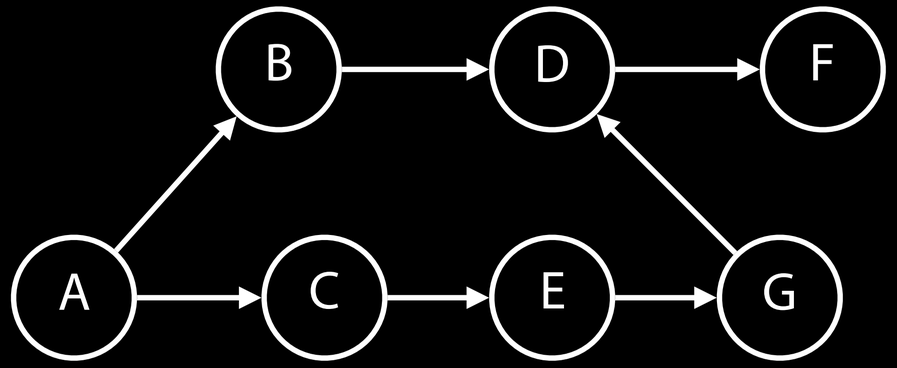
Dynamic programming

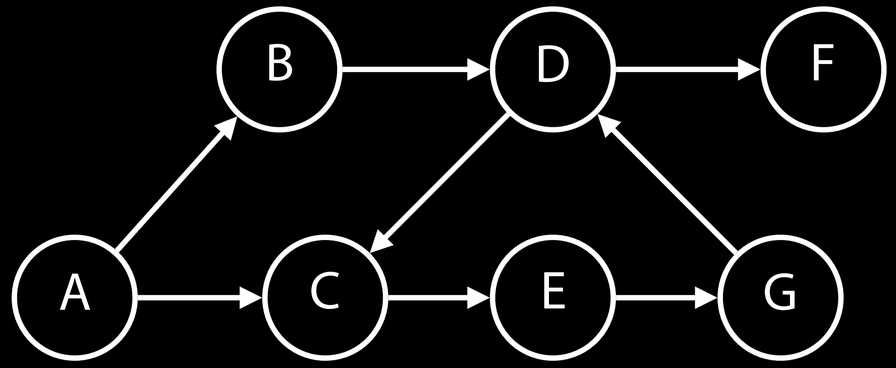
Detecting Cycles

Detecting cycles

Given a *directed* graph represented with an adjacency matrix, determine if the graph has a cycle.

```
bool isCyclic(const vector<vector<bool>> &graph);
```





Detecting cycles

```
bool isCyclic(const vector<vector<bool>> &graph) {
    vector<bool> isVisited(graph.size(), false);
    vector<bool> isOnCurrentPath(graph.size(), false);

    for (size_t i = 0; i < graph.size(); i += 1) {
        if (dfsVisit(i, graph, isVisited, isOnCurrentPath)) {
            return true;
        }
    }

    return false;
}
```

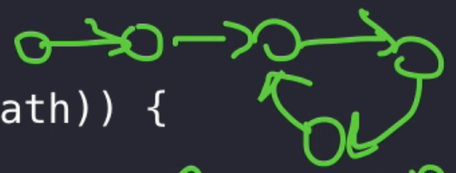
Detecting cycles

```
bool dfsVisit(size_t vertexIndex, const vector<vector<bool>> &graph,
                    vector<bool> &isVisited, vector<bool> &isOnCurrentPath) {
    if (!isVisited[vertexIndex]) {
        isVisited[vertexIndex] = true;
        isOnCurrentPath[vertexIndex] = true;

        for (size_t adjacentIndex = 0; adjacentIndex < graph.size(); adjacentIndex += 1) {
            if (graph[vertexIndex][adjacentIndex]) {
                if (!isVisited[adjacentIndex] &&
                    dfsVisit(adjacentIndex, graph, isVisited, isOnCurrentPath)) {
                    return true;
                } else if (isOnCurrentPath[adjacentIndex]) {
                    return true;
                }
            }
        }
    }
    isOnCurrentPath[vertexIndex] = false;
    return false;
}
```

Detecting cycles

```
bool dfsVisit(size_t vertexIndex, const vector<vector<bool>> &graph,
                    vector<bool> &isVisited, vector<bool> &isOnCurrentPath) {
    if (!isVisited[vertexIndex]) {
        isVisited[vertexIndex] = true;
        isOnCurrentPath[vertexIndex] = true;

        for (size_t adjacentIndex = 0; adjacentIndex < graph.size(); adjacentIndex += 1) {
            if (graph[vertexIndex][adjacentIndex]) {
                if (!isVisited[adjacentIndex] &&
                    dfsVisit(adjacentIndex, graph, isVisited, isOnCurrentPath)) {
                    return true;
                } else if (isOnCurrentPath[adjacentIndex]) {
                    return true;
                }
            }
        }
    }
    isOnCurrentPath[vertexIndex] = false;
    return false;
}
```

later cycle



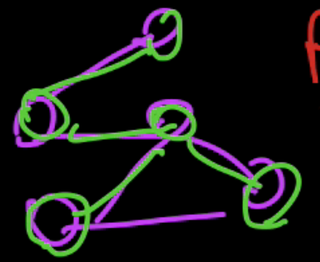
visited + onCurrentPath
THIS IS A CYCLE

Dijkstra’s Algorithm

Problems for weighted graphs

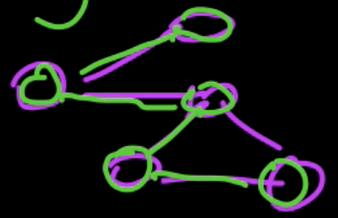
min / max

1. Min (max) spanning tree



Prim's
Kruskal's

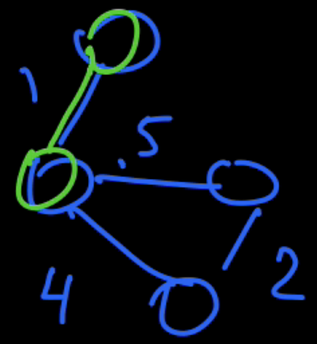
2. Shortest Path
can have cycles

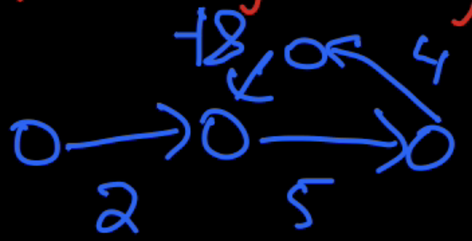


Dijkstra's
Bellman-Ford
Floyd-Warshall's

Dijkstra's Algorithm

↳ works well for graphs with non-negative edges





Dijkstra’s Algorithm

Visit vertices in order of best-known distance from source, relaxing each edge from the visited vertex.

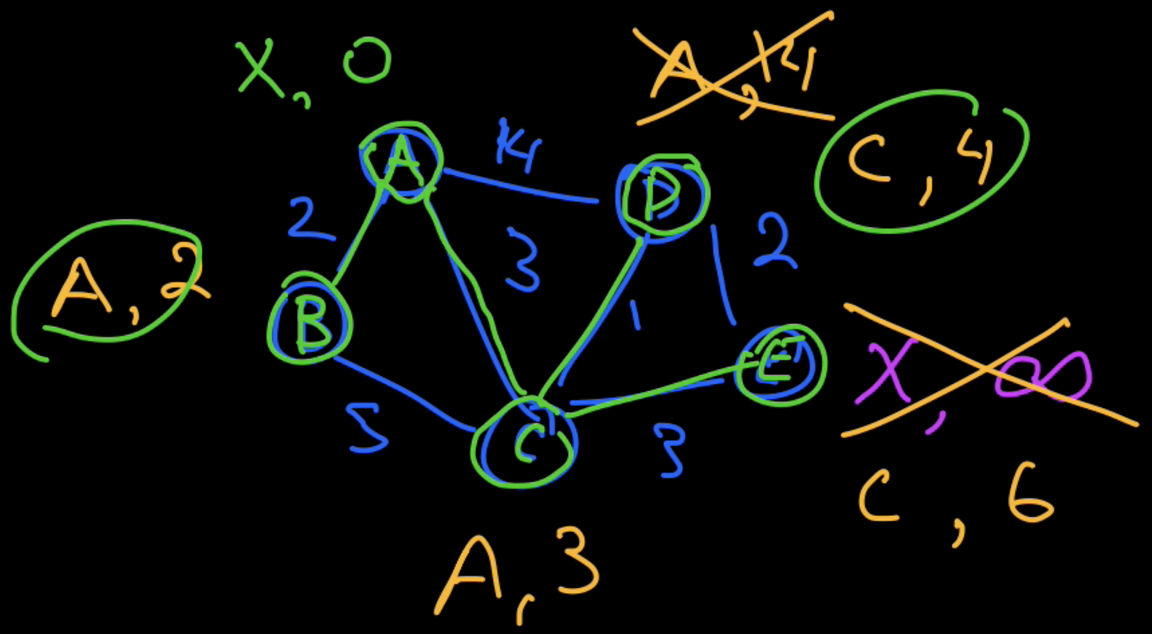
Relaxing an edge means to add to SPT if better distance.

Dijkstra's Algorithm

Visit vertices in order of best-known distance from source, relaxing each edge from the visited vertex.

Relaxing an edge means to add to SPT if better distance.

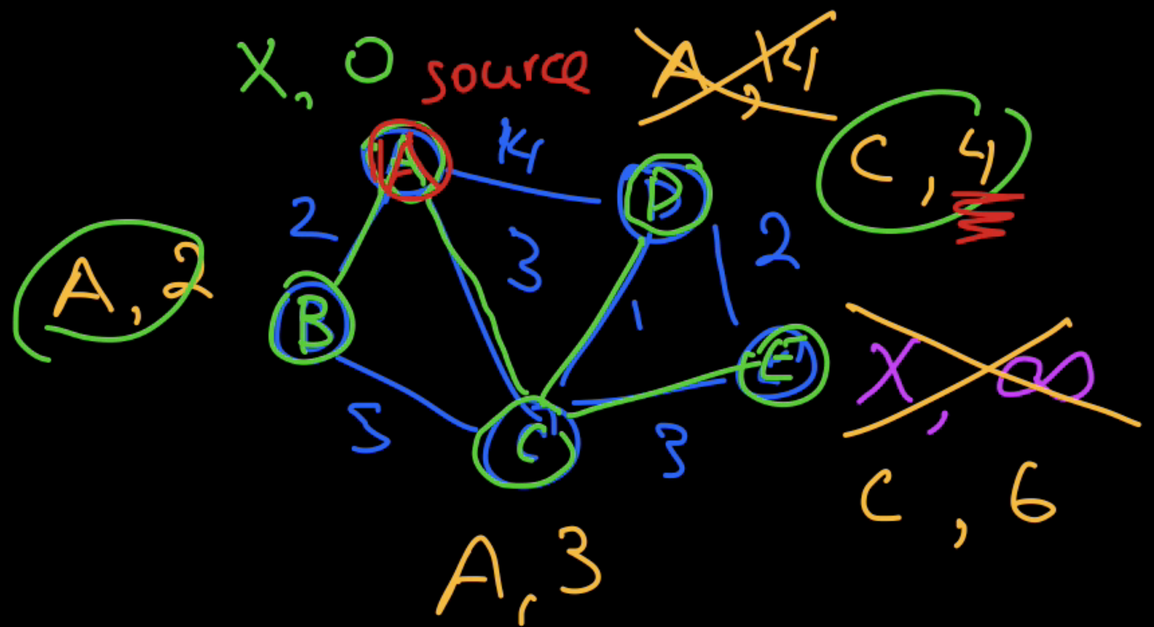
shortest path tree

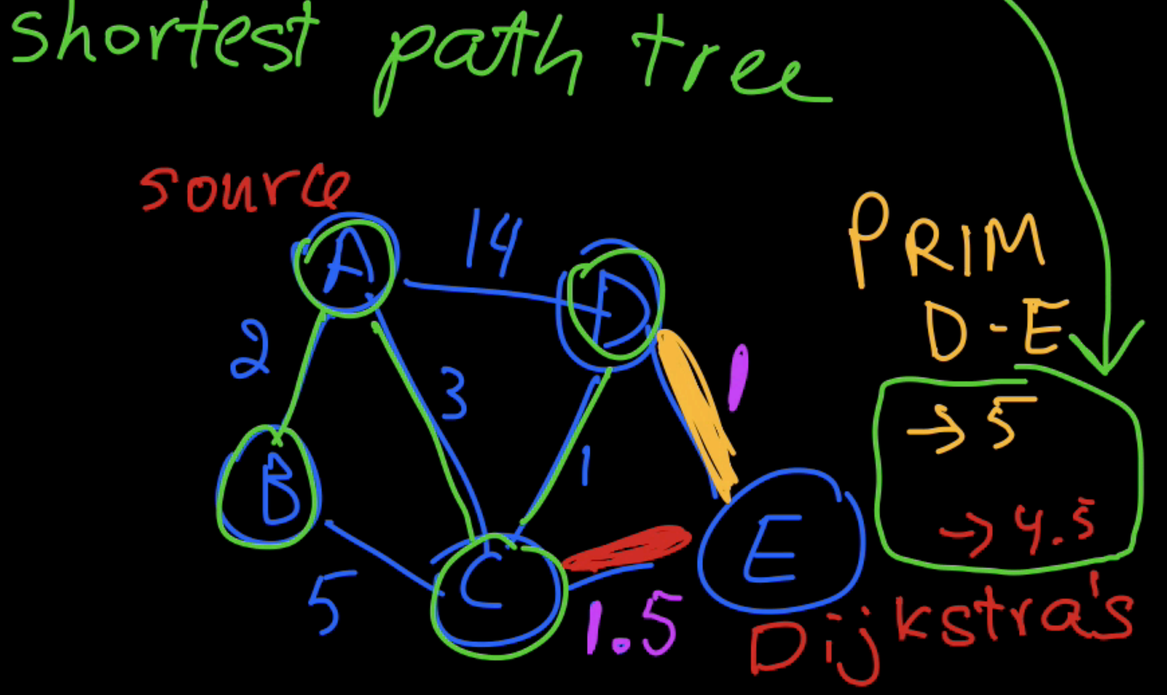


Dijkstra's Algorithm

Visit vertices in order of best-known distance from source, relaxing each edge from the visited vertex.

Relaxing an edge means to add to SPT if better distance.



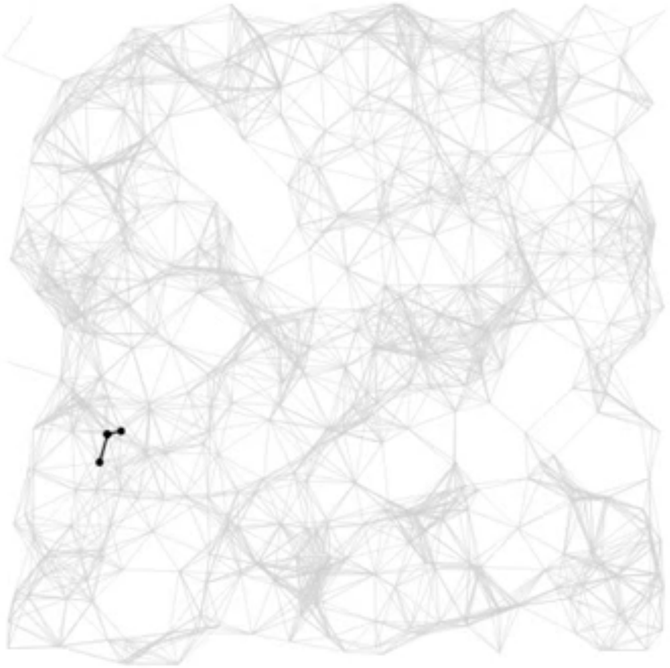


total weight
PRIM 7
Dijkstra 7.5

Prim's vs. Dijkstra's

	Prim's	Dijkstra's
Visit order	in order of distance from the MST under construction	in order of distance from the source
Visiting a vertex: relax all edges	under the metric of distance from tree	under the metric of the distance from the origin

Prim’s vs. Dijkstra’s



Prim’s

Dijkstra’s

Algorithm Design

Algorithm Design

Brute force

Greedy

Divide and conquer

Backtracking

Branch and Bound

Dynamic programming

Brute force

Generates every possible answer and selects only the valid ones

Usually runs in exponential time

Requires a generation of each answer such as with a permutation-generation function or a tree-traversal function

Examples:
Pick all subsets of numbers and see which add up to a given sum
Pick all routes in the travelling salesman problem and choose the best.

Tree Questions for Tree Points

3 points. For each question, select all correct answers.

Question 1 — 1 pts

Consider the *Tree Sort* algorithm in which sorting is achieved by placing all elements to be sorted into a binary search tree and then taking the in-order traversal of the tree. Which of the following statements are true about Tree Sort?

- [ ] Tree Sort performs well on input that is already in nearly sorted order.
- [ ] The worst case time complexity of Tree Sort is $\Theta(n \log n)$.
- [ ] Using an AVL tree (i.e. "AVL Sort") instead of a regular binary search tree improves the worst case complexity.
- [ ] Tree Sort is stable, as long as the right policy is used for equally valued elements.
- [ ] None of the above.

Greedy

Picks the “best” next partial solution from the current partial solution at every step, by some meaning of “best”.

Usually visits each node once (often this translates to linear complexity, but not always).

May not always produce an optimal solution.

Examples: MST algorithms like Prim’s and Kruskal’s, making change using U.S. coins

Greedy

locally optimal solution

Picks the "best" next partial solution from the current partial solution at every step, by some meaning of "best".

Usually visits each node once (often this translates to linear complexity, but not always).

May not always produce an optimal solution.

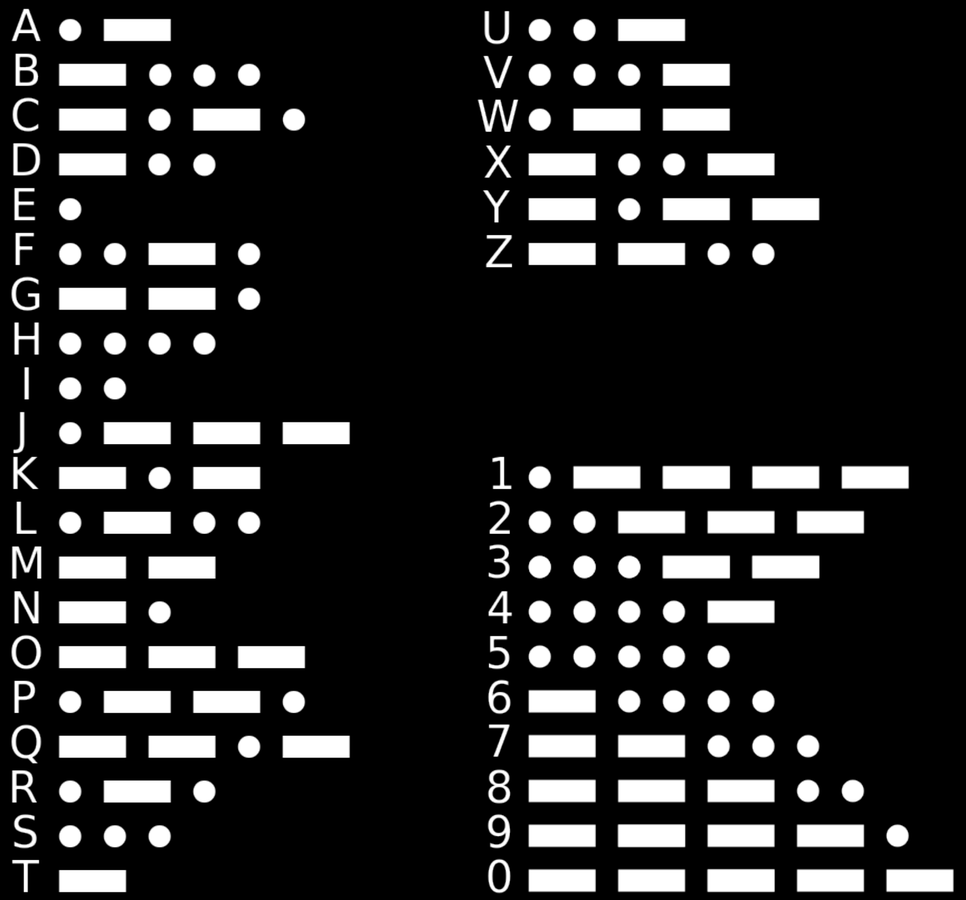
Examples: MST algorithms like Prim's and Kruskal's, making change using U.S. coins

Huffman coding

ASCII

Character	Decimal	Binary	Hexadecimal
A	65	1000001	41
B	66	1000010	42
C	67	1000011	43
D	68	1000100	44
E	69	1000101	45

Morse code



Prefix Property

```
A   0
B   1
C   01
D   11
```

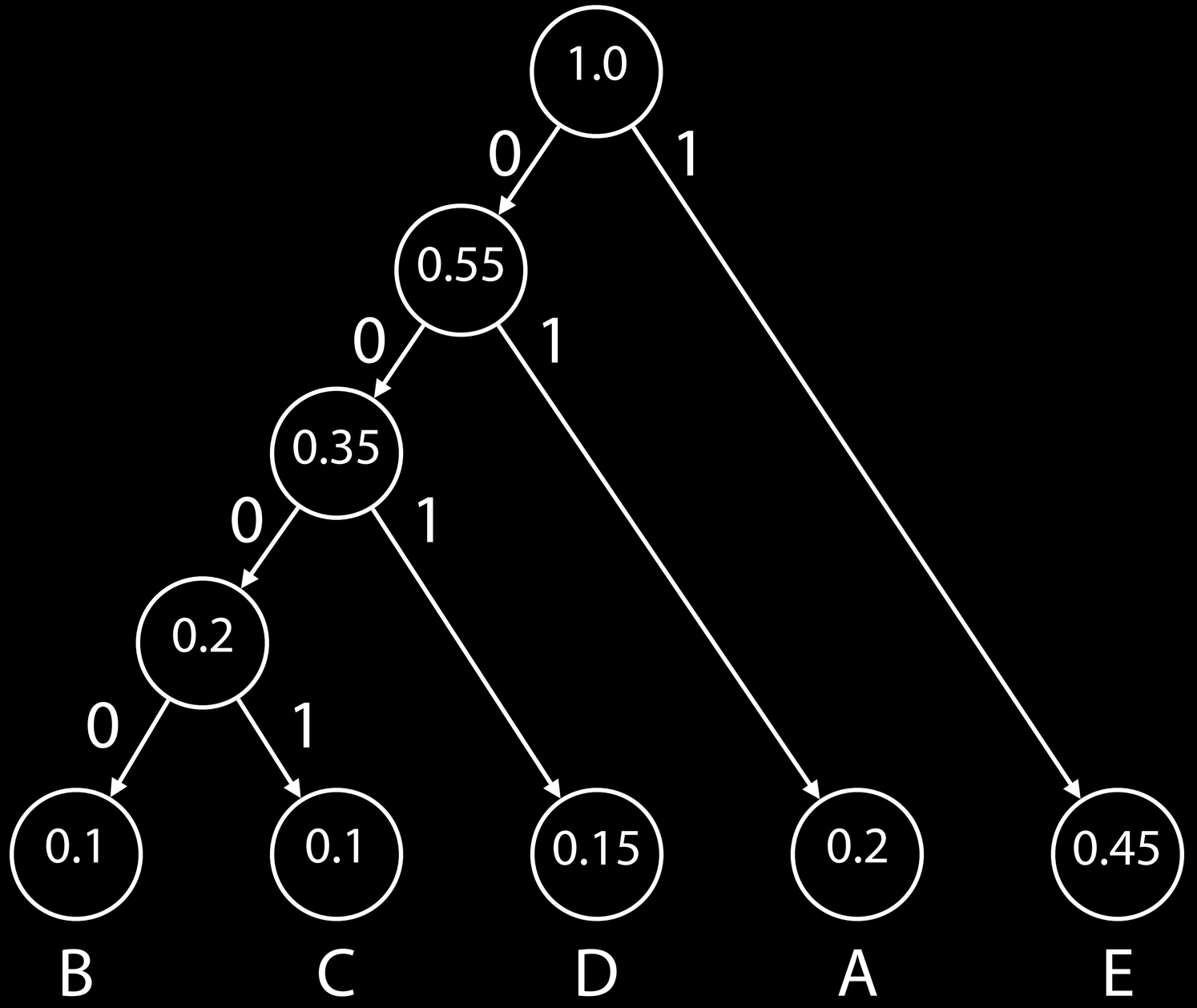
Prefix Property

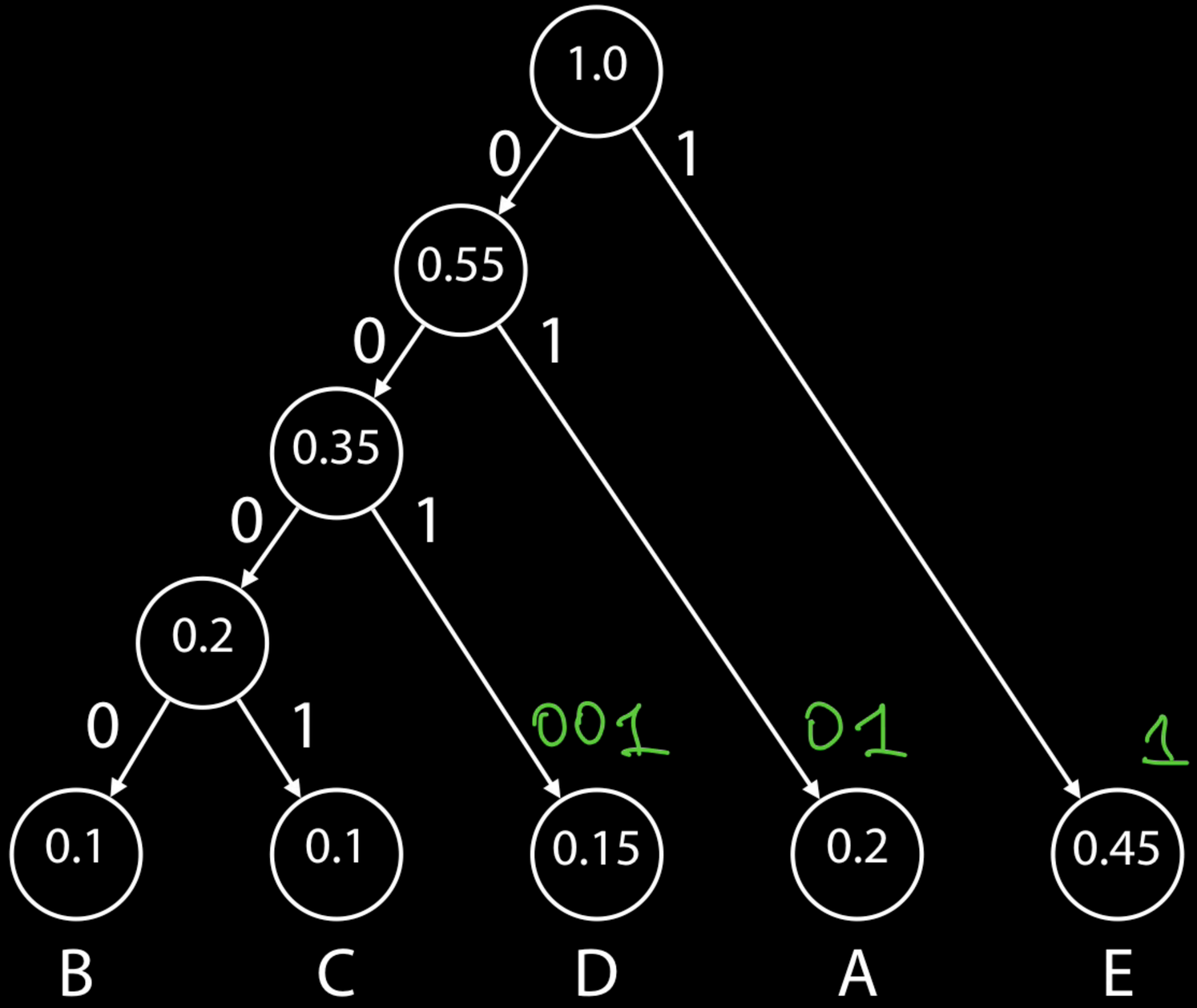
A	0
B	1
C	01
D	11

A	0
B	10
C	110
D	111

ECEABEADCAEFEEEECEADEEEEEDBAAEABDB
BAAEAAACDDCCEABEEDCDEEDEAEEEEEAEED
BCEBEEADEAEEDAEBCDEDEAEEDCEEAEEE

character	A	B	C	D	E
frequency	0.2	0.1	0.1	0.15	0.45


1.0
0
1
0.55
0
1
0.35
0
1
0.2
0
1
0.1
0.1
0.15
0.2
0.45
B
C
D
A
E


1.0
0
1
0.55
0
1
0.35
0
1
0.2
0
1
001
01
1
0.1
0.1
0.15
0.2
0.45
B
C
D
A
E

Huffman coding

Exploit redundancy and existing order inside the sequence.

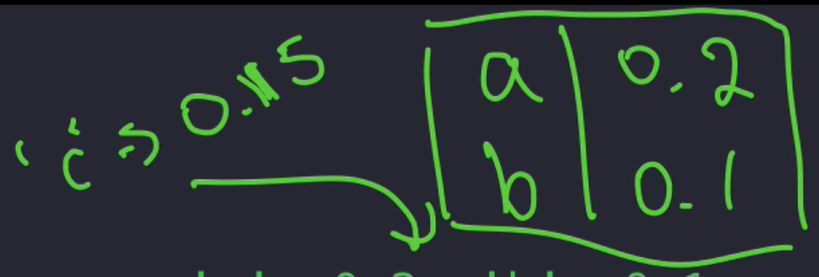
Sequences with no existing redundancy or order may get expanded.

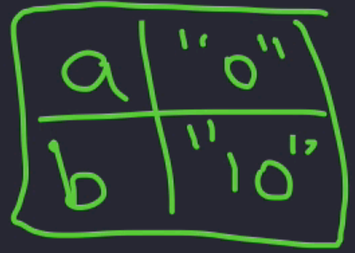
Huffman coding

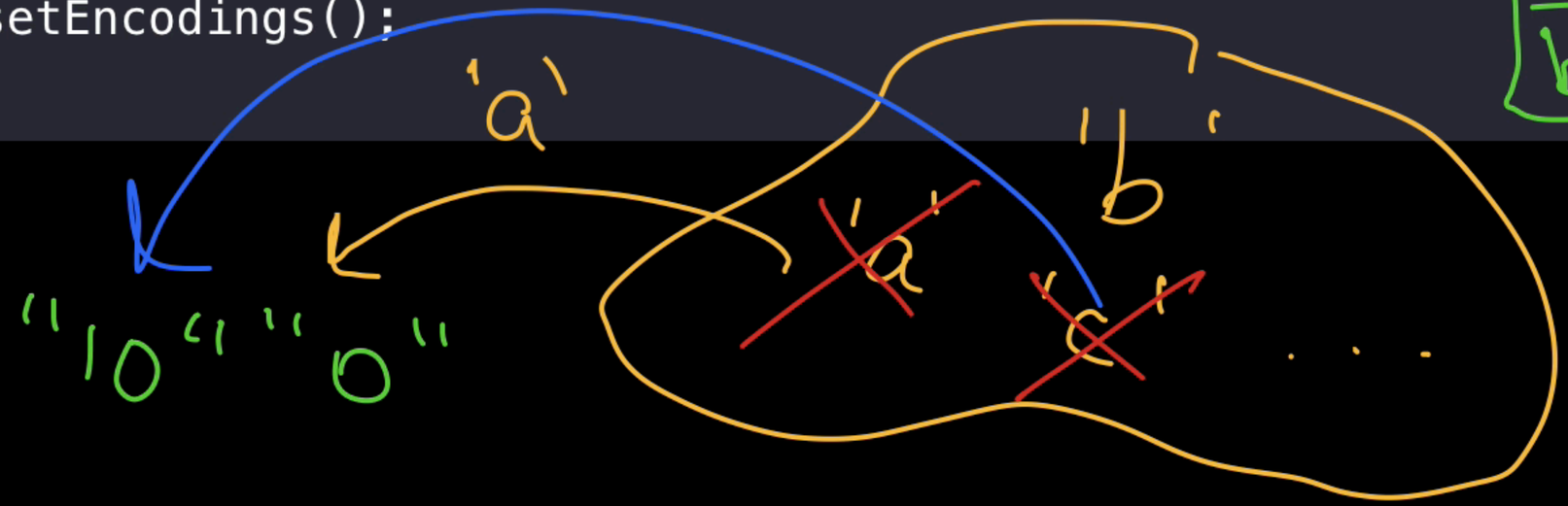
```
class Encoder {
public:
    // ...
private:
    // probabilites for each supported character, e.g. 'a'->0.2, 'b'->0.1
    unordered_map<char, double> probabilities;
    // encodings for each supported character, e.g. 'a'->"0", 'c'->"10"
    unordered_map<char, string> encodings;
    // ...
    void setEncodings();
};
```

Huffman coding

```
class Encoder {
public:
    // ...
private:
    // probabilites for each supported character, e.g. 'a'->0.2, 'b'->0.1
    unordered_map<char, double> probabilities;
    // encodings for each supported character, e.g. 'a'->"0", 'c'->"10"
    unordered_map<char, string> encodings;
    // ...
    void setEncodings();
};
```







Huffman coding

```
void Encoder::setEncodings() {
    auto cmp = [this](char a, char b) {
        return probabilities[a] < probabilities[b];
    };
    priority_queue<char, deque<char>, decltype(cmp)> characters(cmp);
    for (auto it = probabilities.begin(); it != probabilities.end(); ++it) {
        char character = it->first;
        characters.push(character);
    }
    string prefix = "";
    while (!characters.empty()) {
        char character = characters.top();
        characters.pop();
        string encoding = prefix + '1';
        encodings[character] = encoding;
        prefix += '0';
    }
}
```

Huffman coding

```
void Encoder::setEncodings() {
    auto cmp = [this](char a, char b) {
        return probabilities[a] < probabilities[b];
    };
    priority_queue<char, deque<char>, decltype(cmp)> characters(cmp);
    for (auto it = probabilities.begin(); it != probabilities.end(); ++it) {
        char character = it->first;
        characters.push(character);
    }
    string prefix = "";
    while (!characters.empty()) {
        char character = characters.top();
        characters.pop();
        string encoding = prefix + '1';
        encodings[character] = encoding;
        prefix += '0';
    }
}
```

instance

lambda

figure out type (keyword)

var

put all chars in PQ

= string("0") + prefix

Divide and Conquer

Divides a problem into multiple non-overlapping subproblems to construct a solution.

Typically only top-down and recursive.

Subproblems constructed by partitioning the current problem, then the subproblems are solved and joined together somehow.

Examples: quicksort, merge sort.

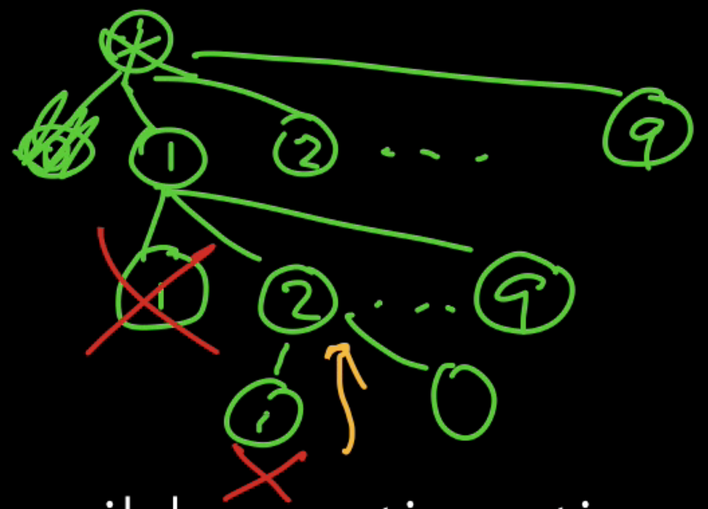
Backtracking

Find a solution satisfying a constraint.

For efficiency, will usually generate only possible continuations of the solution.

Examples: *n*-queens, class scheduling.

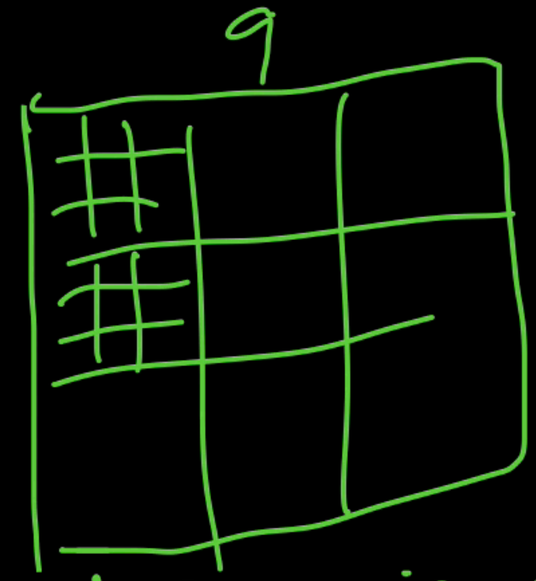
Backtracking



Find a solution satisfying a constraint.

For efficiency, will usually generate only possible continuations of the solution.

Examples: *n*-queens, class scheduling.



Sudoku brute force
9x9 gen. all poss. permutations
then check solutions

Backtracking
- constraints: can't repeat 1-9 unique
- Solve w/ solution tree

Branch-and-Bound

Find the best solution possible.

Branch pruning. Bound estimation.

Any number of solutions may be ‘valid’ solutions but only a few will be optimal.

Examples: travelling salesman problem, chess engine AI.

Dynamic Programming

Divides a problem into multiple overlapping subproblems and builds a solution (either bottom-up or top-down).

Only ever needs to calculate the solution to a given solution once.

Often involves some kind of cache or table to store intermediate solutions.

Examples: Fibonacci, text-justification, knapsack problem.

Dynamic Programming

Divides a problem into multiple overlapping subproblems and builds a solution (either bottom-up or top-down).

Only ever needs to calculate the solution to a given solution once.

Often involves some kind of cache or table to store intermediate solutions.

Examples: Fibonacci, text-justification, knapsack problem.

Knapsack problem

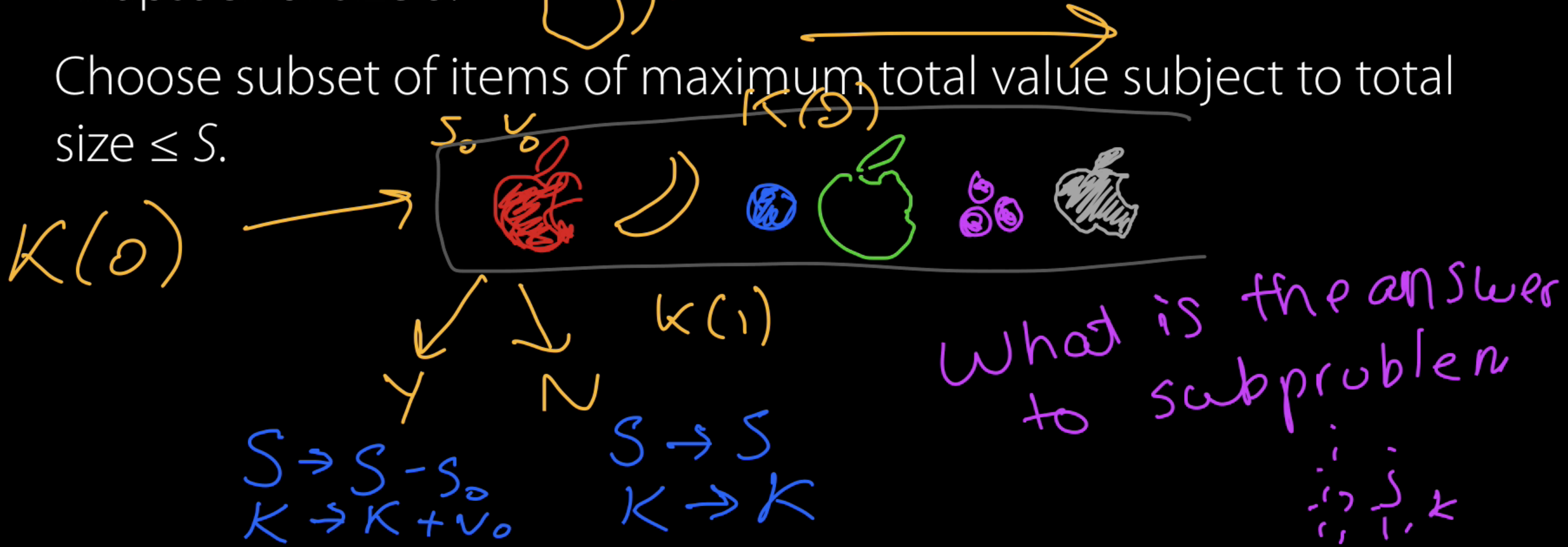
weight
cost

benefit
profit

List of n items, each with size s_i and value v_i.

Knapsack of size S.

Choose subset of items of maximum total value subject to total size $\leq S$.



Algorithm Design

Brute force

Greedy

Divide and conquer

Backtracking

Branch and Bound

Dynamic programming

Fibonacci sequence

```
fib(n) = fib(n – 1) + fib(n – 2)

fib(0) = 0, fib(1) = 1
```

Knapsack problem

List of n items, each with size s_i and value v_i.

Knapsack of size S.

Choose subset of items of maximum total value subject to total size $\leq S$.

Knapsack problem

Subproblem: what is the value of items starting at index *i* ?

Recurrence:

$$K(i) = \max(K(i + 1), v_i + K(i + 1) \text{ if } s_i \leq S)$$

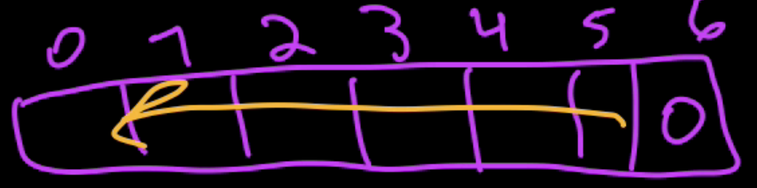
Better recurrence:

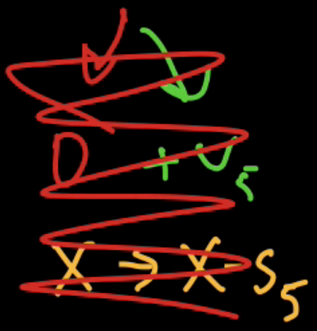
$$K(i, X) = \max(K(i + 1, X), v_i + K(i + 1, X - s_i) \text{ if } s_i \leq X)$$

Base case:

$$K(n, X) = 0$$

Knapsack problem





Subproblem: what is the value of items starting at index *i* ?

Recurrence:

don't choose choose

```
K(i) = max(K(i + 1), vi + K(i + 1) if si ≤ S)
```

Better recurrence:

don't choose

K(0, S) = max(K(1, S), v0 + K(1, S − s0))

solution

```
K(i, X) = max(K(i + 1, X),
              vi + K(i + 1, X − si) if si ≤ X)
```

remaining size

current size

taken by items

Base case:

```
K(n, X) = 0
```

after n items, no items remain → Total value is 0

Pseudopolynomial Time

Polynomial in the problem size AND the numbers in input.

Polynomial = good

Exponential = bad

Pseudopolynomial = okay

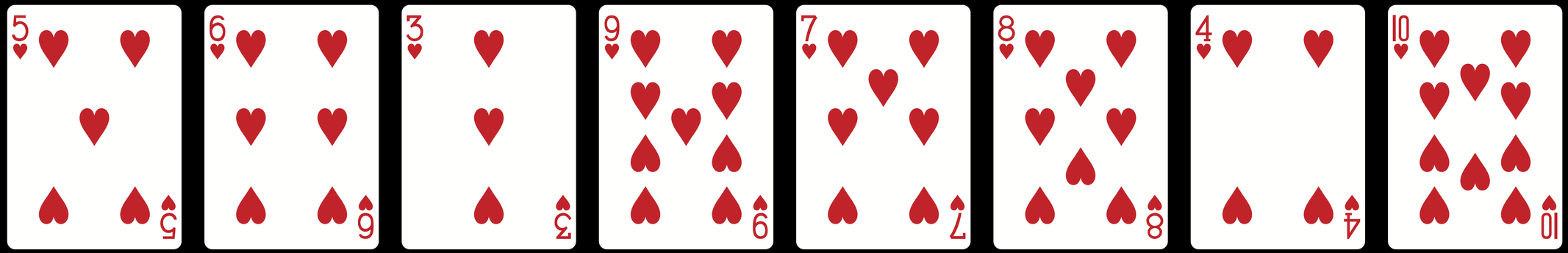
Subset Sum

Given a set of *n* integers *S* and a value *W*, determine if there is a subset of integers that sum to W.

Winning games

A simple card game

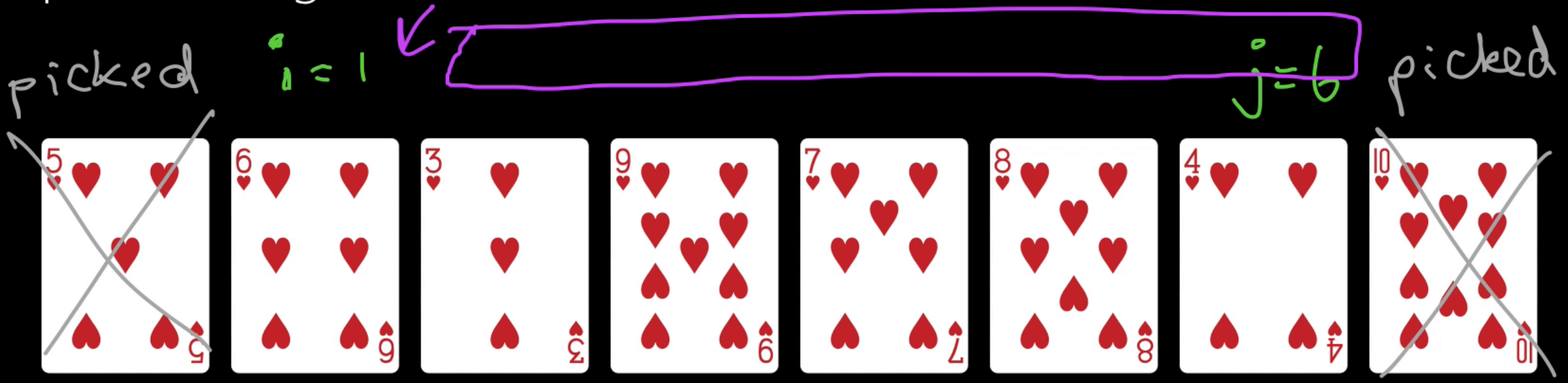
Deal *n* cards. Each player picks a card from the left end or from the right end until no cards remain. The player whose cards sum up to the highest number wins.



$S(i,j) = \max(v[i] + \min(S(i+2,j), S(i+1,j-1)),$
$v[j] + \min(S(i+1,j-1), S(i,j-2)))$

A simple card game

Deal *n* cards. Each player picks a card from the left end or from the right end until no cards remain. The player whose cards sum up to the highest number wins.



vector<int> = {5, 6, 3, 9, 7, 8, 4, 10}

Text justification