Scientific Programming: Algorithms (part B)

Programming paradigms - continued -

Luca Bianco - Academic Year 2019-20
luca.bianco@fmach.it
[credits: thanks to Prof. Alberto Montresor]

Greedy algorithms

Greedy

- Greedy approach: select the choice which appears "locally optimal"
- Area of application: optimization problems

Independent intervals

Input

Let $S = \{1, 2, \ldots, n\}$ be a set of interval of the real line. Each interval $[a_i, b_i[$, with $i \in S$, is closed on the left and open on the right.

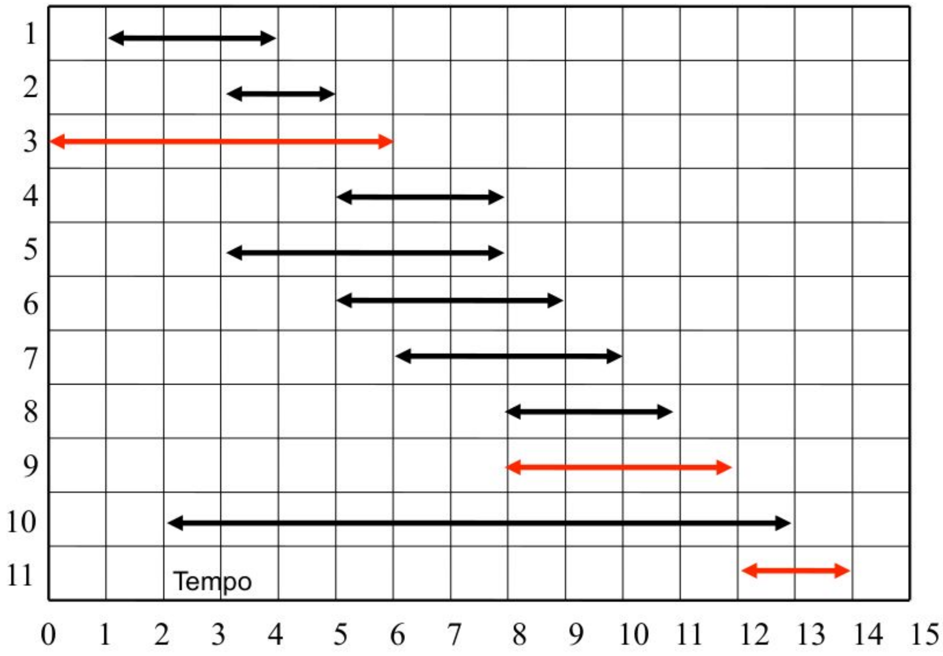
- a_i: starting time
- b_i: finish time

Problem definition

Find a maximal independent subset, i.e. a subset that has maximal cardinality and it is composed by completely disjoint intervals.

i	a_i	b_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Independent intervals



Input

Let $S = \{1, 2, \ldots, n\}$ be a set of interval of the real line. Each interval $[a_i, b_i[$, with $i \in S$, is closed on the left and open on the right.

- a_i: starting time
- b_i: finish time

Problem definition

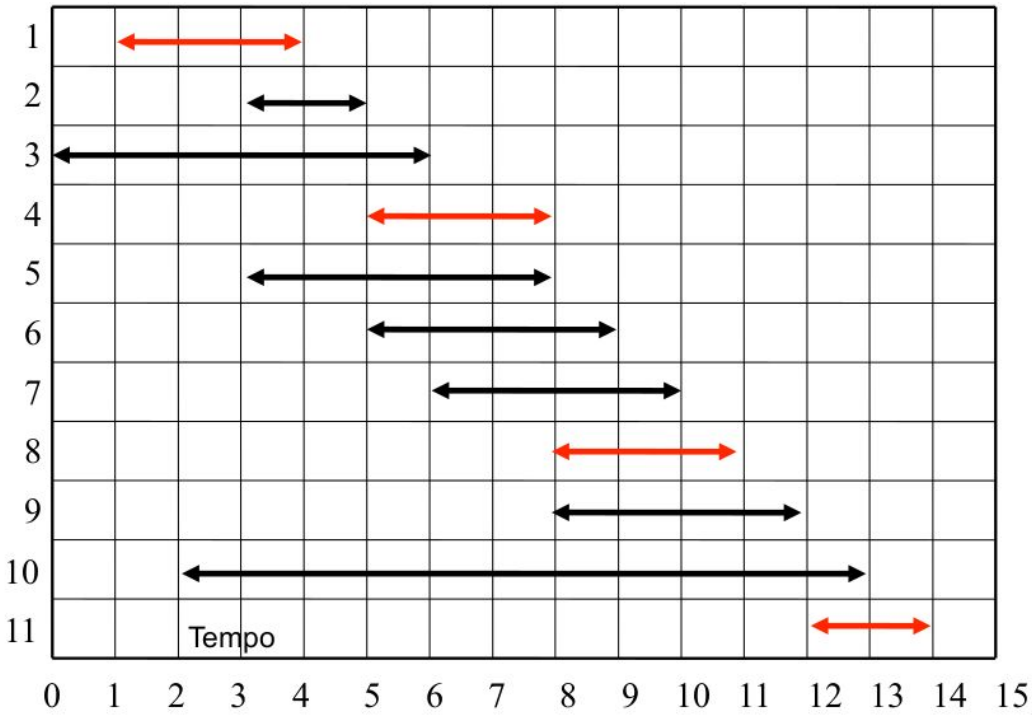
Find a maximal independent subset, i.e. a subset that has maximal cardinality and it is composed by completely disjoint intervals.

i	a_i	b_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

these three intervals are not maximal!

intervals are open on the right, hence these are disjoint

Independent intervals



Input

Let $S = \{1, 2, \ldots, n\}$ be a set of interval of the real line. Each interval $[a_i, b_i[$, with $i \in S$, is closed on the left and open on the right.

- a_i: starting time
- b_i: finish time

Problem definition

Find a maximal independent subset, i.e. a subset that has maximal cardinality and it is composed by completely disjoint intervals.

i	a_i	b_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Path to the solution

We start with dynamic programming

- Let's define the problem in a mathematical way
- Let's define the recursive definition

We move to greedy

- Let's search for a greedy choice
- Let's prove that the greedy choice is optimal
- Let's write an iterative algorithm

Optimal substructure

- Assume that the intervals are sorted by finish time:

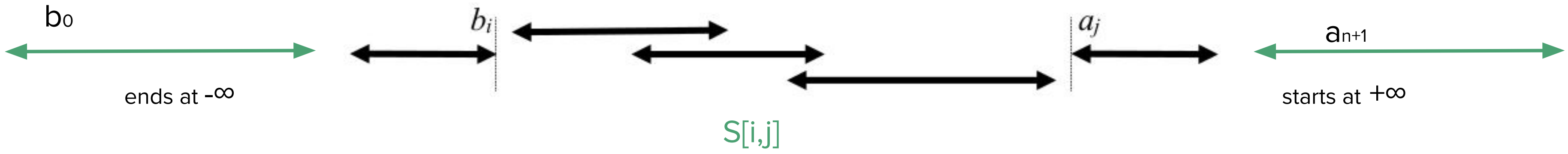
$$b_1 \leq b_2 \leq \ldots \leq b_n$$

- Let the subproblem $S[i,j]$ be the set of intervals that start after the end of i and finish before the start of j:

$$S[i,j] = \{k | b_i \leq a_k < b_k \leq a_j\}$$

- Let's add two "dummy" intervals
  - Interval 0: $b_0 = -\infty$
  - Interval $n+1$: $a_{n+1} = +\infty$

- The initial problem corresponds to problem $S[0, n+1]$

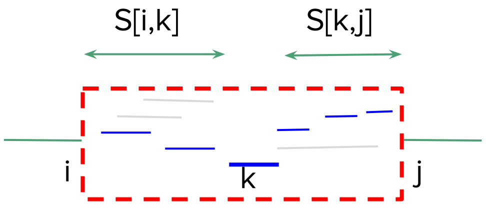


Optimal substructure

Theorem

Let $A[i,j]$ be an optimal solution of $S[i,j]$ and let k be an interval belonging to $A[i,j]$; then

- The problem $S[i,j]$ is subdivided in two subproblems
  - $S[i,k]$: the intervals of $S[i,j]$ that finish before k
  - $S[k,j]$: the intervals of $S[i,j]$ that start after k
- $A[i,j]$ contains the optimal solutions of $S[i,k]$ e $S[k,j]$
  - $A[i,j] \cap S[i,k]$ is an optimal solution of $S[i,k]$
  - $A[i,j] \cap S[k,j]$ is an optimal solution of $S[k,j]$



optimal solution A[i,j]

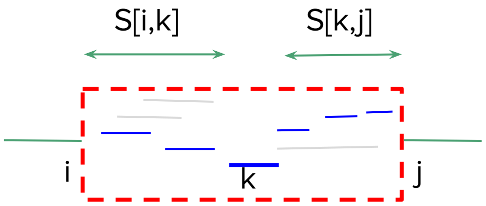
once found k that belongs to the optimal solution A[i,j], we need to solve the two smaller intervals

Optimal substructure

Theorem

Let $A[i,j]$ be an optimal solution of $S[i,j]$ and let k be an interval belonging to $A[i,j]$; then

- The problem $S[i,j]$ is subdivided in two subproblems
  - $S[i,k]$: the intervals of $S[i,j]$ that finish before k
  - $S[k,j]$: the intervals of $S[i,j]$ that start after k
- $A[i,j]$ contains the optimal solutions of $S[i,k]$ e $S[k,j]$
  - $A[i,j] \cap S[i,k]$ is an optimal solution of $S[i,k]$
  - $A[i,j] \cap S[k,j]$ is an optimal solution of $S[k,j]$

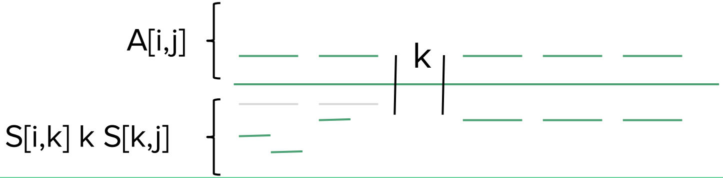


optimal solution A[i,j]

once found k that belongs to the optimal solution A[i,j], we need to solve the two smaller intervals

Proof

We want to prove that if A[i,j] contains the optimal solution of S[i,j] and k is in A[i,j] then it optimally solves S[i,k] and S[k,j]. By contradiction:



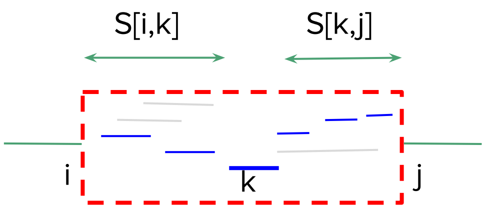
ex. if S[i,k] is better than the corresponding intervals in A[i,j] → A[i,j] is not optimal

Optimal substructure

Theorem

Let $A[i,j]$ be an optimal solution of $S[i,j]$ and let k be an interval belonging to $A[i,j]$; then

- The problem $S[i,j]$ is subdivided in two subproblems
  - $S[i,k]$: the intervals of $S[i,j]$ that finish before k
  - $S[k,j]$: the intervals of $S[i,j]$ that start after k
- $A[i,j]$ contains the optimal solutions of $S[i,k]$ e $S[k,j]$
  - $A[i,j] \cap S[i,k]$ is an optimal solution of $S[i,k]$
  - $A[i,j] \cap S[k,j]$ is an optimal solution of $S[k,j]$

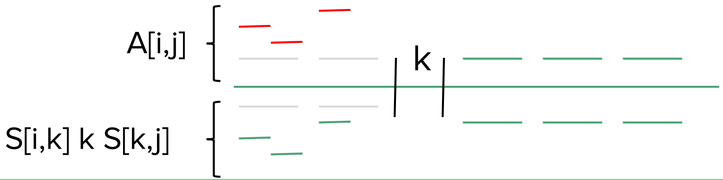


optimal solution A[i,j]

once found k that belongs to the optimal solution A[i,j], we need to solve the two smaller intervals

Proof

We want to prove that if A[i,j] contains the optimal solution of S[i,j] and k is in A[i,j] then it optimally solves S[i,k] and S[k,j]. By contradiction:



ex. if S[i,k] is better than the corresponding intervals in A[i,j] → A[i,j] is not optimal

Recursive formula

Recursive definition of the solution

$$A[i,j] = A[i,k] \cup \{k\} \cup A[k,j]$$

Recursive definition of the cost

- How to identify k? By trying all the possibilities
- Let $D[i,j]$ the size of the largest subset $A[i,j] \subseteq S[i,j]$ of independent intervals

$$D[i,j] = \begin{cases} 0 & S[i,j] = \emptyset \\ \max_{k \in S[i,j]} \{D[i,k] + D[k,j] + 1\} & \text{otherwise} \end{cases}$$

because we chose interval K

Dynamic programming

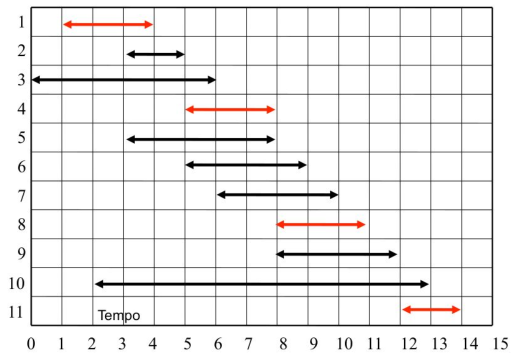
$$D[i,j] = \begin{cases} 0 & S[i,j] = \emptyset \\ \max_{k \in S[i,j]} \{D[i,k] + D[k,j] + 1\} & \text{otherwise} \end{cases}$$

```python
import math

#gets intervals within startI (the interval) and endI
def S(intervals, startI, endI):
    return [x for x in intervals
            if x[0]>=startI[1] and x[1] < endI[0]]

def disjointInt(intervals, i, j, DP):

    s = S(intervals, intervals[i], intervals[j])
    if len(s) == 0:
        return 0
    else:
        if (i,j) not in DP:
            m = 0
            start = intervals.index(s[0])
            end = intervals.index(s[-1])
            for k in range(start,end+1):
                if (i,k) not in DP:
                    DP[(i,k)] = disjointInt(intervals, i, k, DP)
                if (k, j) not in DP:
                    DP[(k, j)] = disjointInt(intervals,k, j, DP)

                m = max(m, DP[(i,k)] + DP[(k, j)] + 1)
            DP[(i,j)] = m

        return DP[(i,j)]


def disjoint_intervals(intervals):
    D = dict()
    return  disjointInt(intervals, 0, len(intervals)-1, D)
```

top-down: DP[0,n]

```python
intervals = [(-math.inf,0), (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
             (8,12), (2,13), (12,14), (15,math.inf)]


print(S(intervals, (1,4), (12,14)))
print(S(intervals, (3,5), (12,14)))
print(S(intervals, intervals[0], intervals[-1]))
print(disjoint_intervals(intervals))
```

```
[(5, 8), (5, 9), (6, 10), (8, 11)]
[(5, 8), (5, 9), (6, 10), (8, 11)]
[(1, 4), (3, 5), (0, 6), (5, 8), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13), (12, 14)]
4
```



Complexity

Dynamic programming

- The definition allows us to write an algorithm based on dynamic programming or memoization
- Complexity $O(n^3)$: we need to solve all potential problems with $i < j$, and it costs $O(n)$ for each subproblem in the worst case.

Can we do better?

- Are we sure that we need to analyze all the values of k?

```python
import math

#gets intervals within startI (the interval) and endI
def S(intervals, startI, endI):
    return [x for x in intervals
            if x[0]>=startI[1] and x[1] < endI[0]]

def disjointInt(intervals, i, j, DP):

    s = S(intervals, intervals[i], intervals[j])
    if len(s) == 0:
        return 0
    else:
        if (i,j) not in DP:
            m = 0
            start = intervals.index(s[0])
            end = intervals.index(s[-1])
            for k in range(start,end+1):
                if (i,k) not in DP:
                    DP[(i,k)] = disjointInt(intervals, i, k, DP)
                if (k, j) not in DP:
                    DP[(k, j)] = disjointInt(intervals,k, j, DP)

                m = max(m, DP[(i,k)] + DP[(k, j)] + 1)
            DP[(i,j)] = m

        return DP[(i,j)]


def disjoint_intervals(intervals):
    D = dict()
    return  disjointInt(intervals, 0, len(intervals)-1, D)
```

Greedy choice

Input

Let $S = \{1, 2, \ldots, n\}$ be a set of interval of the real line. Each interval $[a_i, b_i[$, with $i \in S$, is closed on the left and open on the right.

- a_i: starting time
- b_i: finish time

Theorem

Let $S[i, j]$ a non-empty subproblem, and let m be the interval of $S[i, j]$ that has the smallest finish time, then:

1. the subproblem $S[i, m]$ is empty
2. m is included in some optimal solution of $S[i, j]$

Proof 1

We know that:	$a_m < b_m$	(Interval definition)
We know that:	$\forall k \in S[i, j] : b_m \leq b_k$	(m has smallest finish time)
Then:	$\forall k \in S[i, j] : a_m < b_k$	(Transitivity)

If no interval in $S[i, j]$ terminates before a_m, then $S[i, m] = \emptyset$

Greedy choice

Input

Let $S = \{1, 2, \ldots, n\}$ be a set of interval of the real line. Each interval $[a_i, b_i[$, with $i \in S$, is closed on the left and open on the right.

- a_i: starting time
- b_i: finish time

Theorem

Let $S[i, j]$ a non-empty subproblem, and let m be the interval of $S[i, j]$ that has the smallest finish time, then:

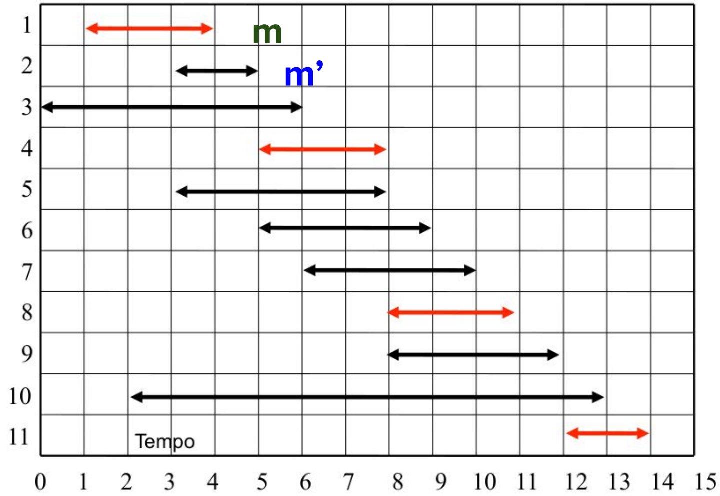
1. the subproblem $S[i, m]$ is empty
2. m is included in some optimal solution of $S[i, j]$

Proof 2

- Let $A'[i, j]$ an optimal solution of $S[i, j]$
- Let $m' \in A'[i, j]$ be the interval with smallest finish time $A'[i, j]$
- Let $A[i, j] = A'[i, j] - \{m'\} \cup \{m\}$ be a new solution obtained by removing m' from and adding m to $A'[i, j]$
- $A[i, j]$ is an optimal solution that contains m, because it has same size of $A'[i, j]$

Greedy choice

- Let $A'[i,j]$ an optimal solution of $S[i,j]$
- Let $m' \in A'[i,j]$ be the interval with smallest finish time $A'[i,j]$
- Let $A[i,j] = A'[i,j] - \{m'\} \cup \{m\}$ be a new solution obtained by removing m' from and adding m to $A'[i,j]$
- $A[i,j]$ is an optimal solution that contains m, because it has same size of $A'[i,j]$

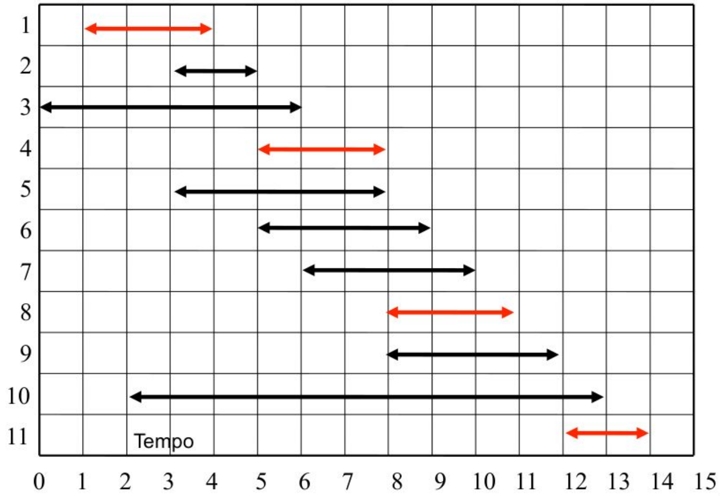


Consequences of the theorem

- It's not necessary to analyze all values of k
  - Let's do a "greedy" choice: let's select the activity m with the smallest finish time
- It is not necessary to analyze two subproblems
  - Remove all the activities that are not compatible with the greedy choice
  - We only get a subproblem: $S[m,j]$

Greedy algorithm

```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]              #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)    #other greedy choices
            last = i
    return S
```



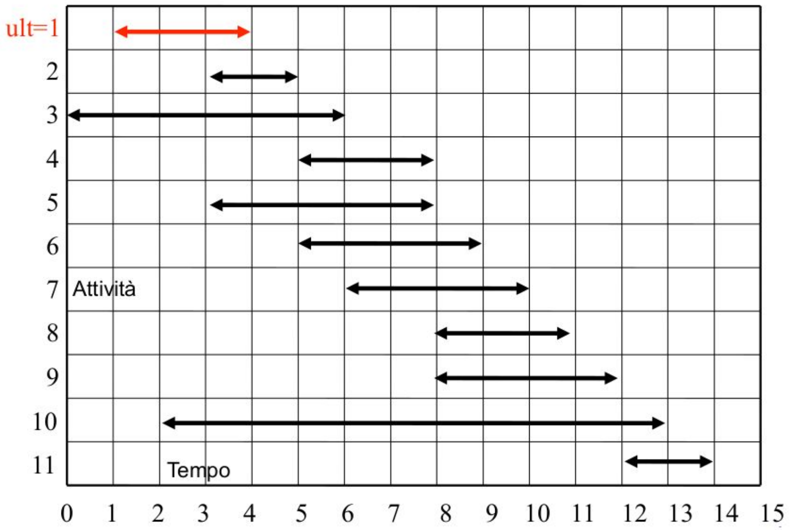

```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity?

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm




```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]                #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)    #other greedy choices
            last = i
    return S
```

```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
             (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

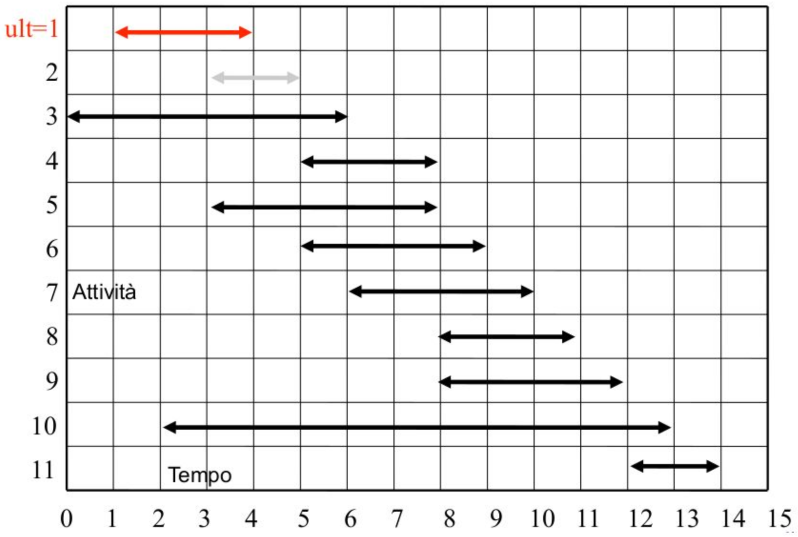
```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]            #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)    #other greedy choices
            last = i
    return S
```




```
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

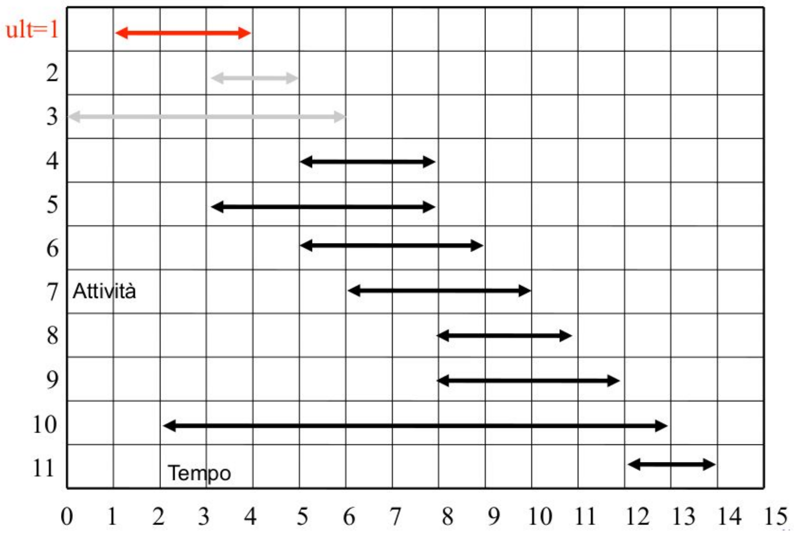
If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)
            last = i
    return S
```

#first greedy choice

#other greedy choices



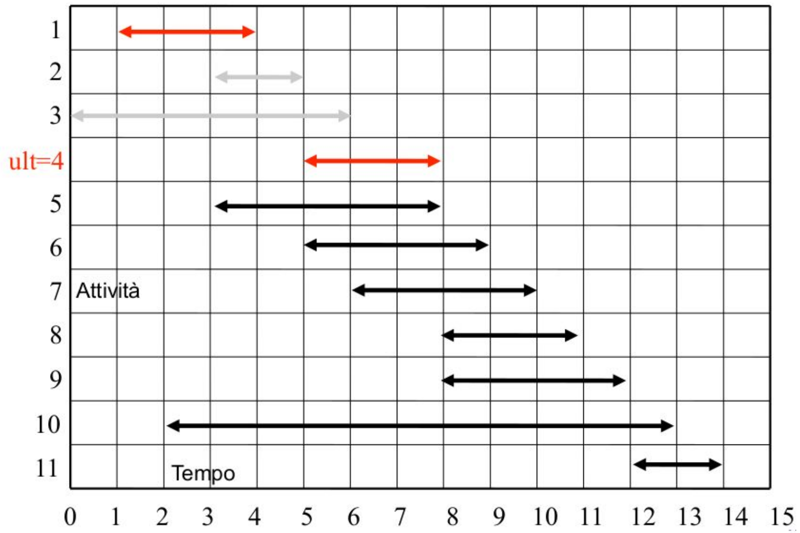

```
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm




```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]          #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)   #other greedy choices
            last = i
    return S
```

```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

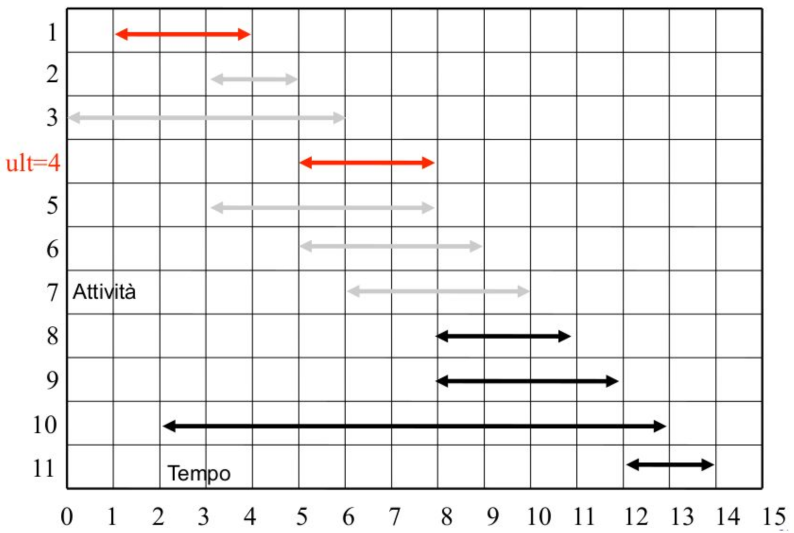
```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]          #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)   #other greedy choices
            last = i
    return S
```




```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

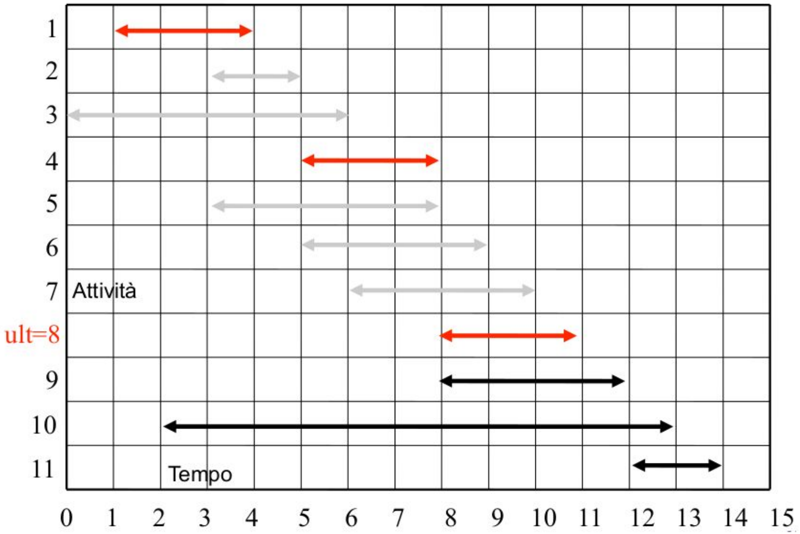
```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]                              #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)                  #other greedy choices
            last = i
    return S
```




```
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

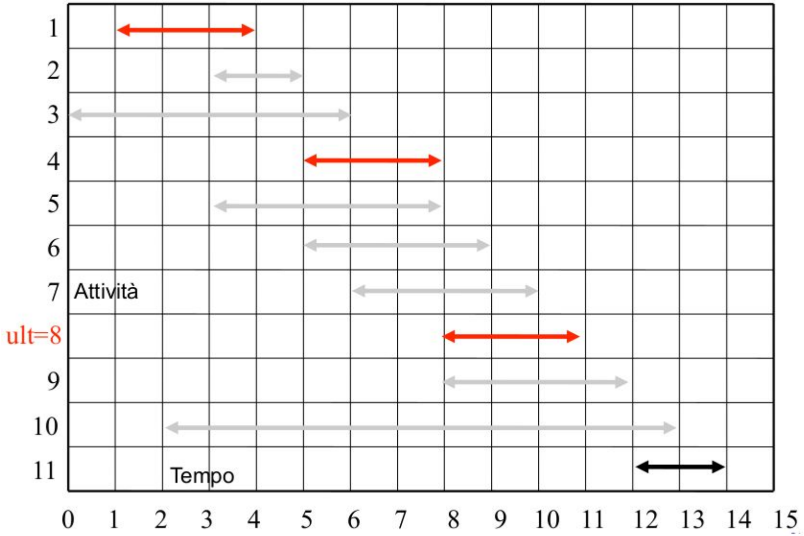
```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]         #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)    #other greedy choices
            last = i
    return S
```




```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

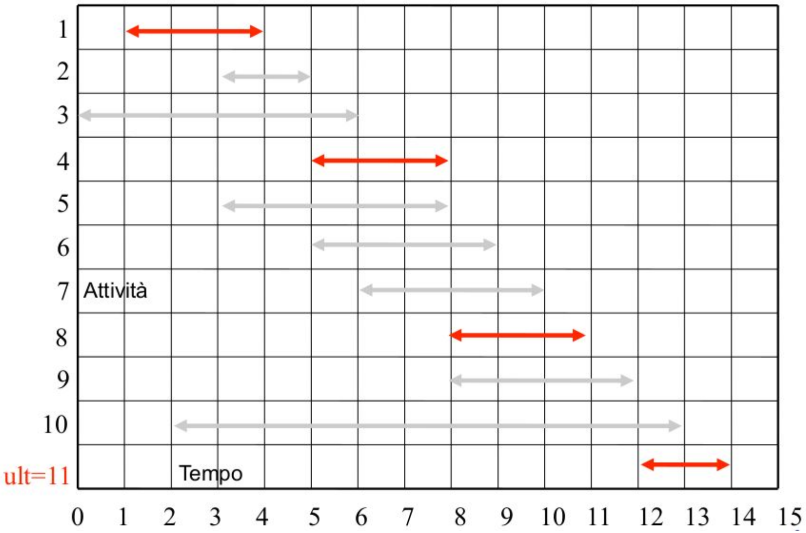
```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Greedy algorithm

```python
def disjoint_greedy(intervals):
    #sort pairs by finishing time
    #if not sorted
    intervals.sort(key = lambda x : x[1])
    S = [0]          #first greedy choice
    last = 0
    for i in range(1,len(intervals)):
        if intervals[i][0] >= intervals[last][1]:
            S.append(i)   #other greedy choices
            last = i
    return S
```



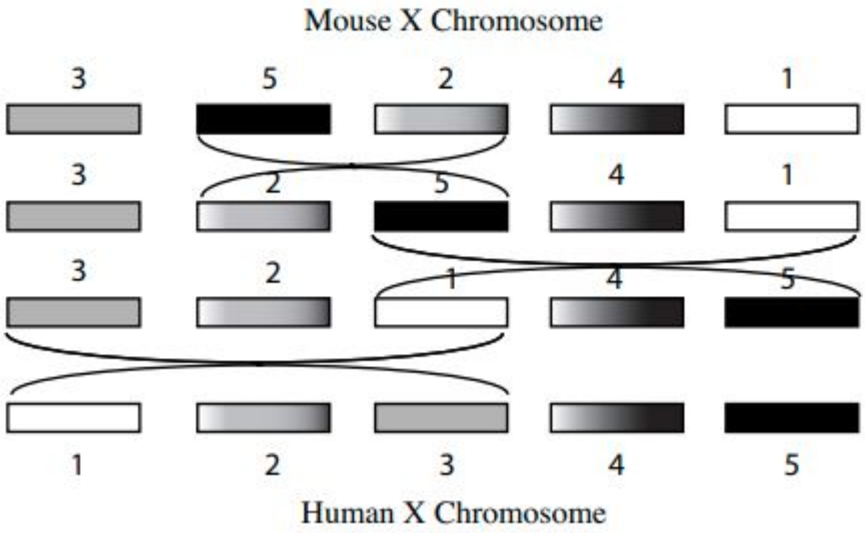

```python
intervals = [ (1,4),(3,5), (0,6), (5,8), (3,8), (5,9), (6,10),(8,11),
              (8,12), (2,13), (12,14)]
DI = disjoint_greedy(intervals)
print(DI)
for i in DI:
    print(intervals[i], end = " ")
```

```
[0, 3, 7, 10]
(1, 4) (5, 8) (8, 11) (12, 14)
```

Complexity

If input not sorted: **O(n log n + n)** = **O(n log n)**
If input sorted: **O(n)**

Genome rearrangements



[3, 5, 2, 4, 1]

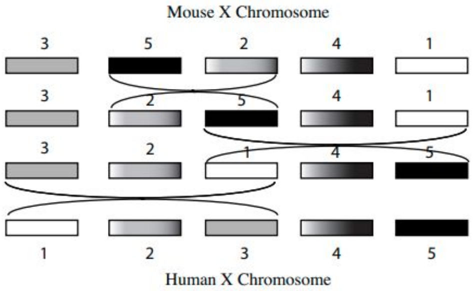
[3, 2, 5, 4, 1]

[3, 2, 1, 4, 5]

[1, 2, 3, 4, 5]

Transformation of mouse gene order into human gene order on Chr X
(biggest synteny blocks)

Genome rearrangements



- Syntheny blocks (for a computer scientist: substrings)
- Re-arrangement: reversing the order of a group of syntheny block
- $\pi = \pi_1 \pi_2 \ldots \pi_{i-1} \overrightarrow{\pi_i \pi_{i+1} \ldots \pi_{j-1} \pi_j} \pi_{j+1} \ldots \pi_{n-1} \pi_n$
- $\pi \cdot \rho(i, j) = \pi_1 \pi_2 \ldots \pi_{i-1} \overleftarrow{\pi_j \pi_{j-1} \ldots \pi_{i+1} \pi_i} \pi_{j+1} \ldots \pi_{n-1} \pi_n$
- Example: $\pi = 1\,2\,\overrightarrow{4\,3\,7\,5}\,6$, $\pi \cdot \rho(3, 6) = 1\,2\,\overleftarrow{5\,7\,3\,4}\,6$

Reversal Distance Problem

Given two permutations, find a shortest series of reversals that transforms one permutation into another

Greedy solution

Reversal Distance Problem

Given two permutations, find a shortest series of reversals that transforms one permutation into another

- We define $prefix(\pi)$ to be the number of already-sorted elements of π
- A sensible strategy for sorting by reversals is to increase $prefix(\pi)$ at every step.
- This leads to an algorithm that sorts a permutation by repeatedly moving its ith element to the ith position.

Greedy solution

Reversal Distance Problem

Given two permutations, find a shortest series of reversals that transforms one permutation into another

```python
def simple_reversal_sorting(L):
    n= len(L)
    for i in range(0,n-1):
        j = L.index(i)
        if j != i:
            L[i:j+1] = L[i:j+1][::-1] # rho(i,j)
            print(L)
```

```python
L = [5,0,1,2,3,4]
print("In list:\n{}\n".format(L))
simple_reversal_sorting(L)

L1 = [2, 4, 1, 3, 0]
print("\nIn list:\n{}\n".format(L1))
simple_reversal_sorting(L1)
```

```
In list:
[2, 4, 1, 3, 0]

[0, 3, 1, 4, 2]
[0, 1, 3, 4, 2]
[0, 1, 2, 4, 3]
[0, 1, 2, 3, 4]
```

```
In list:
[5, 0, 1, 2, 3, 4]

[0, 5, 1, 2, 3, 4]
[0, 1, 5, 2, 3, 4]
[0, 1, 2, 5, 3, 4]
[0, 1, 2, 3, 5, 4]
[0, 1, 2, 3, 4, 5]
```

```
In list:
[5, 0, 1, 2, 3, 4]

[4, 3, 2, 1, 0, 5]
[0, 1, 2, 3, 4, 5]
```

Simple but not optimal!

Approximated algorithms exist...

Backtracking

Problem classes (decisional, search, optimization)

- Definition bases on the concept of admissible solution: a solution that satisfies a given set of criteria

Typical problems

- Build one or all admissible solution
- Counting the admissible solutions
- Find the admissible solution "largest", "smallest", in general "optimal"

Typical problems

Enumeration

- List algorithmically all possible solutions (search space)
- Example: list all the permutations of a set

Build at least a solution

- We use the algorithm for enumeration, stopping at the first solution found
- Example: identify a sequence of steps in the Fifteen game

we explore all possible solutions building/enumerating them and counting or stopping when we find one

Typical problems

Count the solutions

- In some cases, it is possible to count in analytical way
- Example: counting the number of subsets of k elements taken by a set of n elements

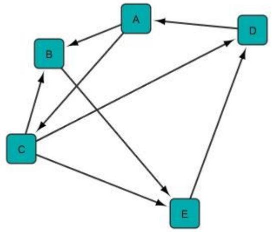
$$\frac{n!}{k!\,(n-k)!}$$

- In other cases, we build the solutions and we count them
- Example: number of subsets of a integer set S whose sum is equal to a prime number

Typical problems

Find optimal solutions

- We enumerate all possible solutions and evaluate them through a cost function
- Only if other techniques are not possible:
  - Dynamic programming
  - Greedy
- Example: Hamiltonian circuit (Traveling salesman)



Build all solutions

To build all the solutions, we use a "brute-force" approach

- Sometimes, it is the only possible way
- The power of modern computer makes possible to deal with problems medium-small problems
  - $10! = 3.63 \cdot 10^6$ (permutation of 10 elements)
  - $2^{20} = 1.05 \cdot 10^6$ (subsets of 20 elements)
- Sometimes, the space of all possible solutions does not need to be analyzed entirely

Backtracking

Approach

- Try to build a solution, if it works you are done else undo it and try again
- “keep trying, you’ll get luckier”

Needs a systematic way to explore the search space looking for the admissible solution(s)

General scheme

General organization

- A solution is represented by a list S
- The content of element $S[i]$ is taken from a set of choices C that depends on the problem

Examples

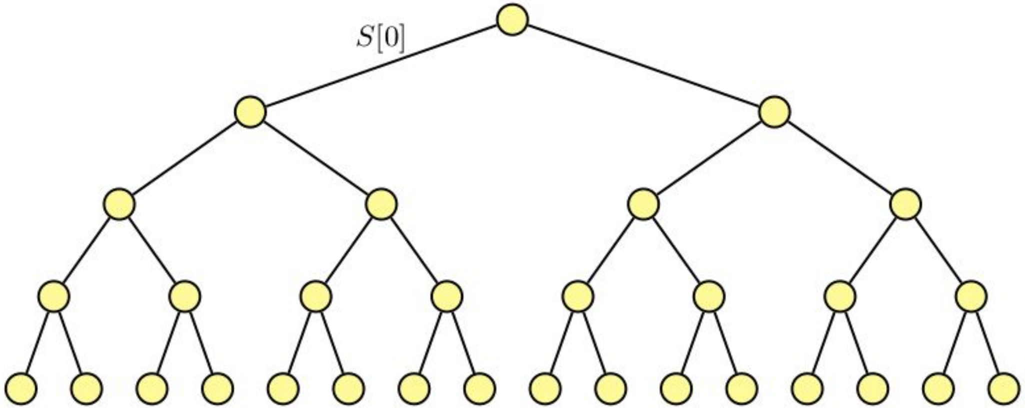
- C generic set, possible solutions permutations of C
- C generic set, possible solutions subsets of C
- C game moves, possible solutions a sequence of moves
- C edges of a graph, possible solutions paths

Partial solutions

- At each step, we start from a partial solution S where $k \geq 0$ choices have been already taken
- If $S[0:k]$ is an admissible solution, we "process" it
  - E.g., we can print it
  - We can then decide to stop here or keep going by listing/printing all solutions
- If $S[0:k]$ is not a complete solution:
  - If possible, we extended solution $S[0:k]$ with one of the possible choices to get a solution $S[0:k+1]$
  - Otherwise, we "cancel" the element $S[k]$ (backtrack) and we go back to to solution $S[0:k-1]$

Decision tree

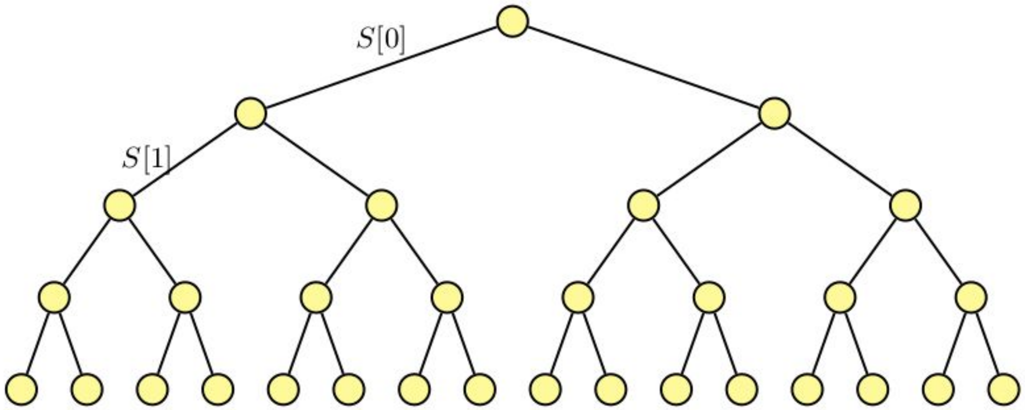
- Decision tree ≡ Search space
- Root ≡ Empty solution
- Internal nodes ≡ Partial solutions
- Leaves ≡ Admissible solutions



Note: the decision tree is “virtual” we do not need to store it all…

Decision tree

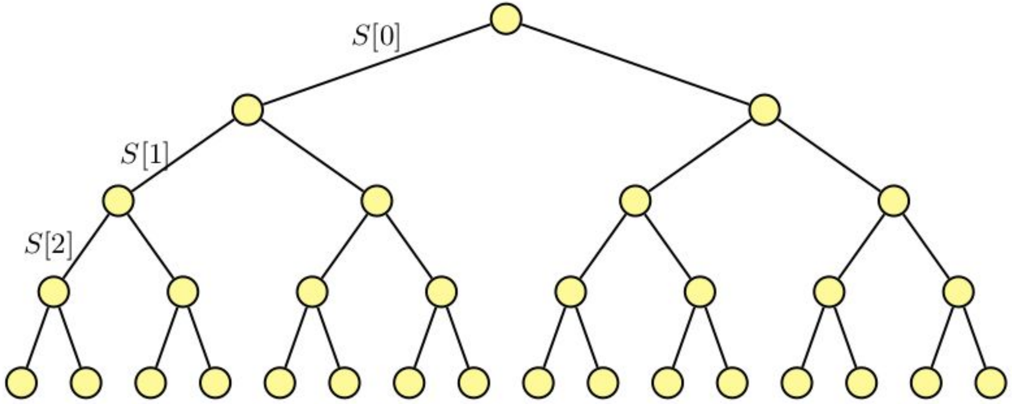
- Decision tree $\equiv$ Search space
- Root $\equiv$ Empty solution
- Internal nodes $\equiv$ Partial solutions
- Leaves $\equiv$ Admissible solutions



Note: the decision tree is "virtual" we do not need to store it all…

Decision tree

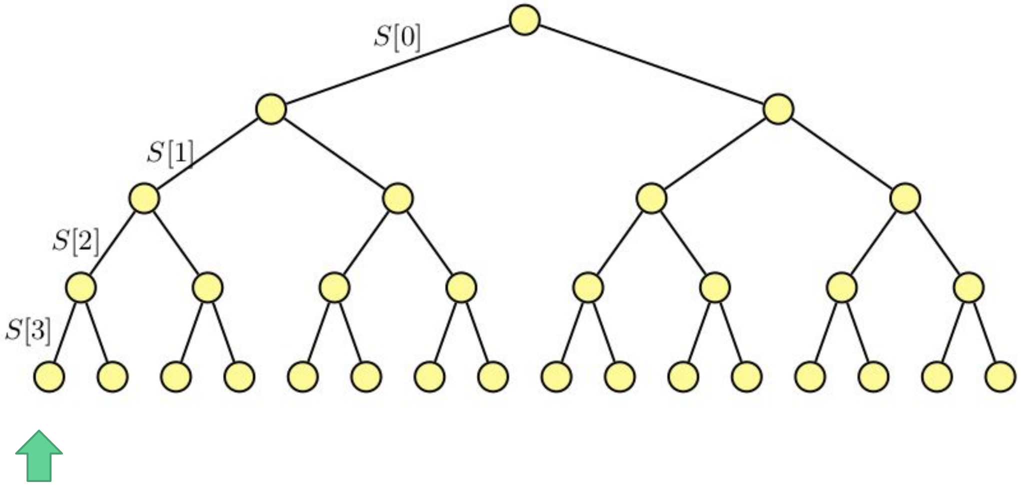
- Decision tree ≡ Search space
- Root ≡ Empty solution
- Internal nodes ≡ Partial solutions
- Leaves ≡ Admissible solutions



Note: the decision tree is "virtual" we do not need to store it all...

Decision tree

- Decision tree $\equiv$ Search space
- Root $\equiv$ Empty solution
- Internal nodes $\equiv$ Partial solutions
- Leaves $\equiv$ Admissible solutions

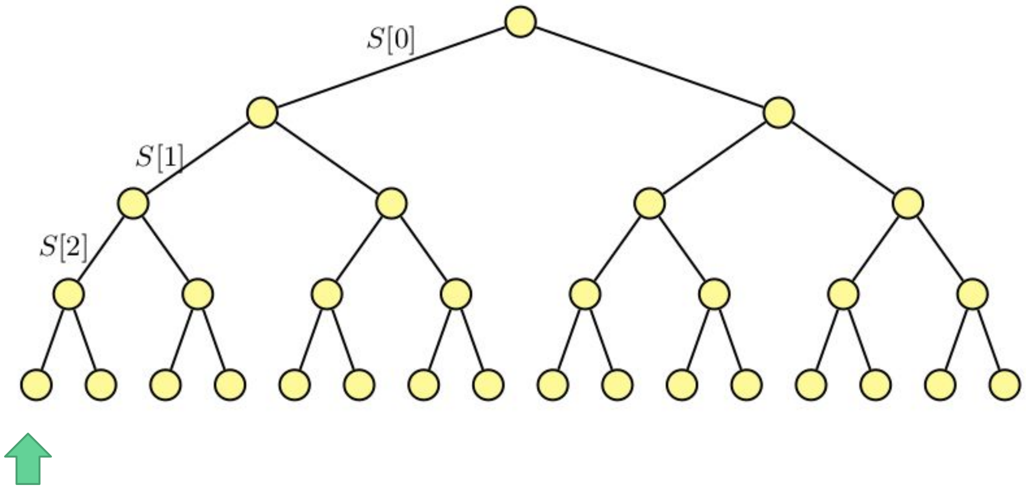


process or ignore the solution

Note: the decision tree is “virtual” we do not need to store it all...

Decision tree

- Decision tree $\equiv$ Search space
- Root $\equiv$ Empty solution
- Internal nodes $\equiv$ Partial solutions
- Leaves $\equiv$ Admissible solutions

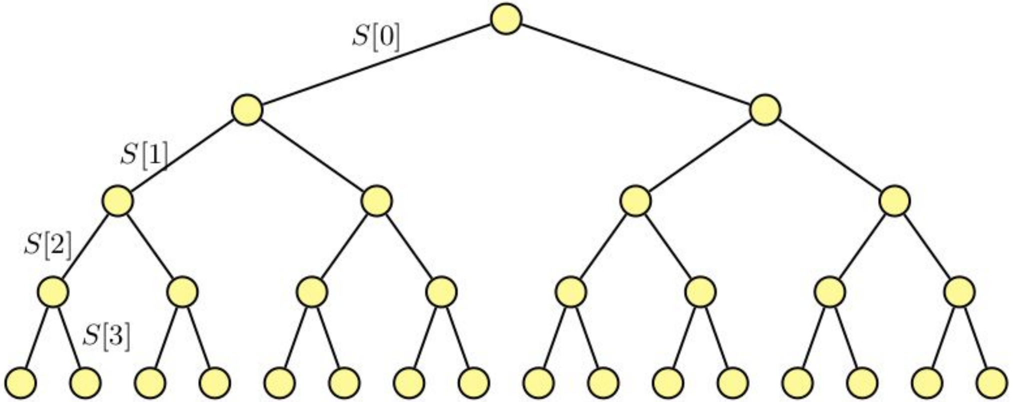


solution ignored

Note: the decision tree is “virtual” we do not need to store it all...

Decision tree

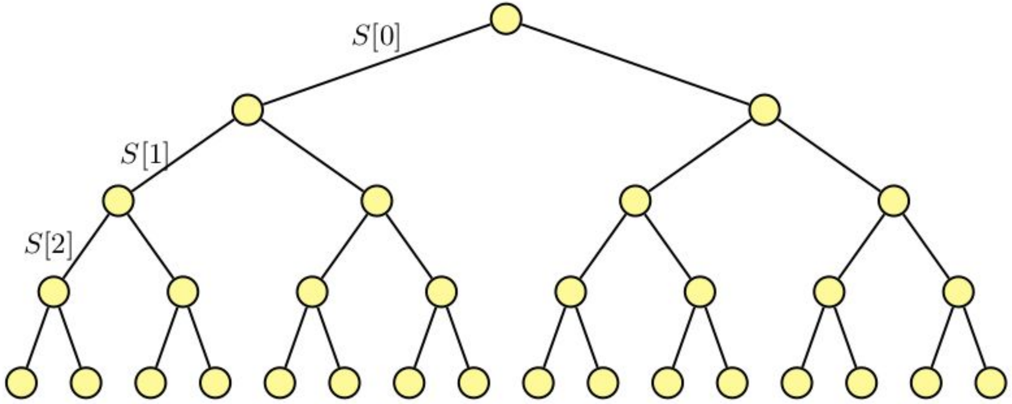
- Decision tree ≡ Search space
- Root ≡ Empty solution
- Internal nodes ≡ Partial solutions
- Leaves ≡ Admissible solutions



Note: the decision tree is “virtual” we do not need to store it all...

Decision tree

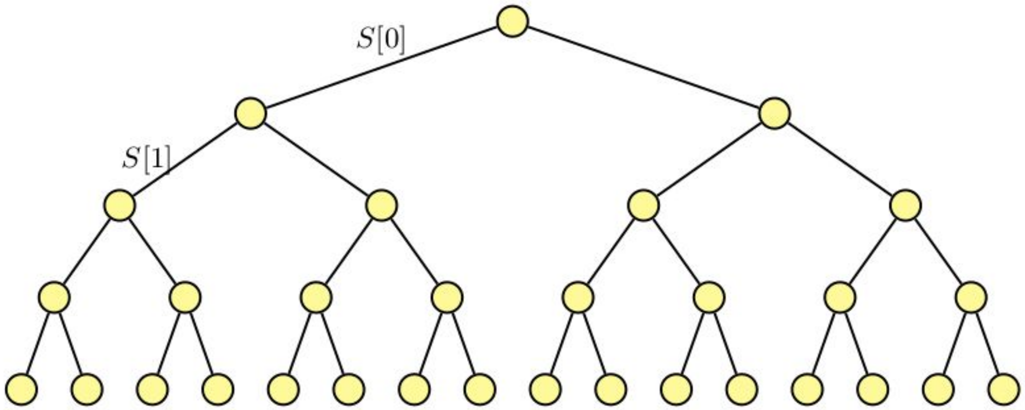
- Decision tree ≡ Search space
- Root ≡ Empty solution
- Internal nodes ≡ Partial solutions
- Leaves ≡ Admissible solutions



Note: the decision tree is “virtual” we do not need to store it all…

Decision tree

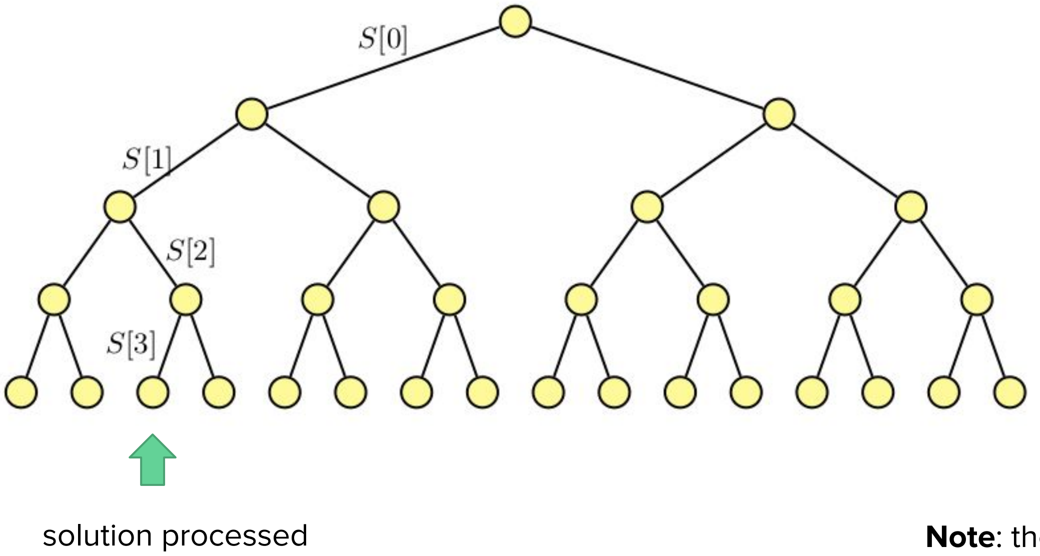
- Decision tree $\equiv$ Search space
- Root $\equiv$ Empty solution
- Internal nodes $\equiv$ Partial solutions
- Leaves $\equiv$ Admissible solutions



Note: the decision tree is “virtual” we do not need to store it all…

Decision tree

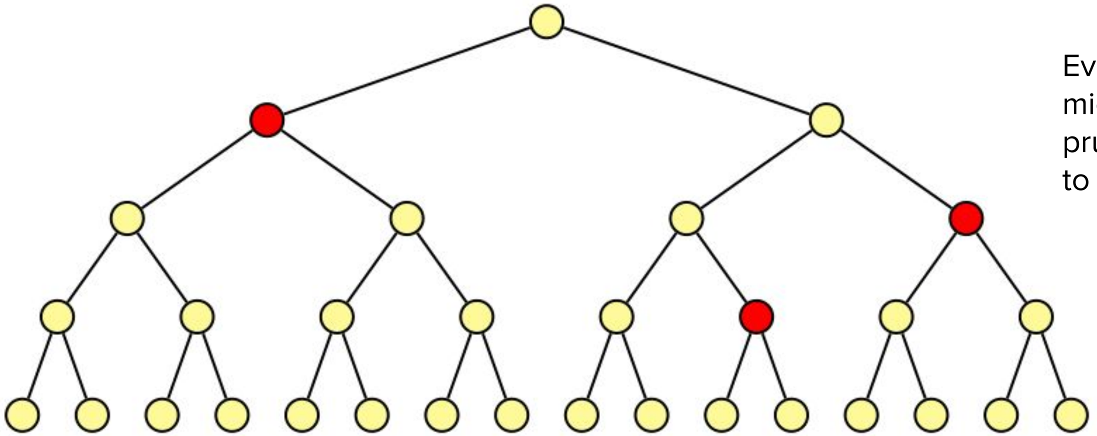
- Decision tree ≡ Search space
- Root ≡ Empty solution
- Internal nodes ≡ Partial solutions
- Leaves ≡ Admissible solutions



Note: the decision tree is “virtual” we do not need to store it all...

Pruning

- "Branches" of the trees that do not bring to admissible solutions can be "pruned"
- The evaluation is done in the partial solutions corresponding to internal nodes

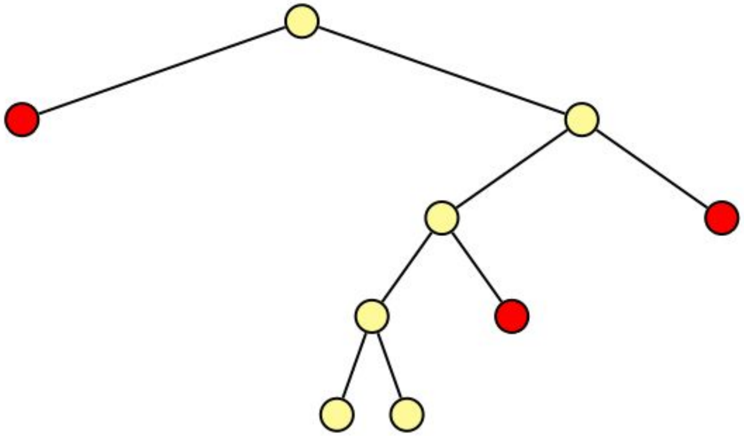


Even though the tree might be exponential, with pruning we might not need to explore it all

Note: the decision tree is “virtual” we do not need to store it all...

Pruning

- "Branches" of the trees that do not bring to admissible solutions can be "pruned"
- The evaluation is done in the partial solutions corresponding to internal nodes



Even though the tree might be exponential, with pruning we might not need to explore it all

General schema to find a solution (modify as you like)

boolean enumeration(OBJECT[] S, **int** n, **int** i, $\ldots$)

SET C = choices($S, n, i, \ldots$) % Compute C based on $S[0 : i-1]$

foreach $c \in C$ **do**

  $S[i] = c$

  if isAdmissible(S, n, i) **then**

    if processSolution($S, n, i, \ldots$) **then**

      return True

  if enumeration($S, n, i+1, \ldots$) **then**

    return True

return False

S is the list of choices
n is the maximum number of choices
i is the index of the choice I am currently making
... other inputs

The recursive call will test all solutions unless they return true

1. We build a next choice with choices(...) based on the previous choices S[0:i-1]: the logic of the code goes here
2. For each possible choice, we memorize the choice in S[i]
3. If S[i] is admissible then we process it and we can either stop (if we needed at least one solution) or continue to the next one (return false)
4. In the latter case we keep going calling enumeration again to compute choice i+1

Enumeration

```
boolean enumeration(OBJECT[] S, int n, int i, ...)
SET C = choices(S, n, i, ...)        % Compute C based on S[0 : i − 1]
foreach c ∈ C do
    S[i] = c
    if isAdmissible(S, n, i) then
        if processSolution(S, n, i, ...) then
            return True
    if enumeration(S, n, i + 1, ...) then
        return True
return False
```

- S: list containing the partial solutions
- i: current index
- $\ldots$: additional information
- C: the set of possible candidates to extend the current solution
- isAdmissible(): returns **True** if $S[0:i]$ is an admissible solution
- processSolution(): returns
  - **True** to stop the execution at the first admissible solution
  - **False** to explore the entire tree

Subsets problem

boolean enumeration(OBJECT[] S, **int** n, **int** i, $\dots$)

SET C = choices($S, n, i, \dots$)   % Compute C based on $S[0 : i-1]$
foreach $c \in C$ **do**
  $S[i] = c$
  if isAdmissible(S, n, i) **then**
    if processSolution($S, n, i, \dots$) **then**
      return True
  if enumeration($S, n, i+1, \dots$) **then**
    return True
return False

List all subsets of $\{0, \dots, n-1\}$

```python
def process_solution(S):
    for i in range(len(S)):
        print(S[i], end = " ")
    print("")
    return False

def subsets(S,n,i):
    #print("subsets({},{},{})".format(S,n,i))
    C = [1, 0] if i<n else []
    for c in C:
        S[i] = c
        if i == n-1:
            #print("\t\tS:{} c:{} i:{}".format(S,c,i))
            if process_solution(S):
                return True
        else:
            #print("\tCalling:  subsets({},{},{})".format(S,n,i+1))
            subsets(S,n,i+1)
    return False

n = 5
S = [0]*n
subsets(S,n,0)
```

False: we want all solutions

choice: keep or discard element

an admissible solution has decided if to keep or discard all elements

```
1 1 1 1 1
1 1 1 1 0
1 1 1 0 1
1 1 1 0 0
1 1 0 1 1
1 1 0 1 0
1 1 0 0 1
1 1 0 0 0
1 0 1 1 1
1 0 1 1 0
1 0 1 0 1
1 0 1 0 0
1 0 0 1 1
1 0 0 1 0
1 0 0 0 1
1 0 0 0 0
0 1 1 1 1
0 1 1 1 0
0 1 1 0 1
0 1 1 0 0
0 1 0 1 1
0 1 0 1 0
0 1 0 0 1
0 1 0 0 0
0 0 1 1 1
0 0 1 1 0
0 0 1 0 1
0 0 1 0 0
0 0 0 1 1
0 0 0 1 0
0 0 0 0 1
0 0 0 0 0
```

```
subsets([0, 0, 0, 0, 0],5,0)
        Calling:  subsets([1, 0, 0, 0, 0],5,1)
subsets([1, 0, 0, 0, 0],5,1)
        Calling:  subsets([1, 1, 0, 0, 0],5,2)
subsets([1, 1, 0, 0, 0],5,2)
        Calling:  subsets([1, 1, 1, 0, 0],5,3)
subsets([1, 1, 1, 0, 0],5,3)
        Calling:  subsets([1, 1, 1, 1, 0],5,4)
subsets([1, 1, 1, 1, 0],5,4)
                S:[1, 1, 1, 1, 1] c:1 i:4
1 1 1 1 1
                S:[1, 1, 1, 1, 0] c:0 i:4
1 1 1 1 0
        Calling:  subsets([1, 1, 1, 0, 0],5,4)
subsets([1, 1, 1, 0, 0],5,4)
                S:[1, 1, 1, 0, 1] c:1 i:4
1 1 1 0 1
                S:[1, 1, 1, 0, 0] c:0 i:4
1 1 1 0 0
        Calling:  subsets([1, 1, 0, 0, 0],5,3)
subsets([1, 1, 0, 0, 0],5,3)
...
```

Subsets problem

```python
def process_solution(S):
    for i in range(len(S)):
        print(S[i], end = " ")
    print("")
    return False

def subsets(S,n,i):
    #print("subsets({},{},{})".format(S,n,i))
    C = [1, 0] if i<n else []
    for c in C:
        S[i] = c
        if i == n-1:
            #print("\t\tS:{} c:{} i:{}".format(S,c,i))
            if process_solution(S):
                return True
        else:
            #print("\tCalling:  subsets({},{},{})".format(S,n,i+1))
            subsets(S,n,i+1)
    return False

n = 5
S = [0]*n
subsets(S,n,0)
```

List all subsets of $\{0, \ldots, n-1\}$

- There is no pruning. All the possible space is explored. But this is required by the definition of the problem
- Computational complexity $O(n2^n)$ (→ i.e. 2^n sets, printing each costs n)
- In which order sets are printed?
- Is it possible to think to an iterative version, ad-hoc for this problem? (non-backtracking)

Subsets problem

List all subsets of $\{0, \ldots, n-1\}$

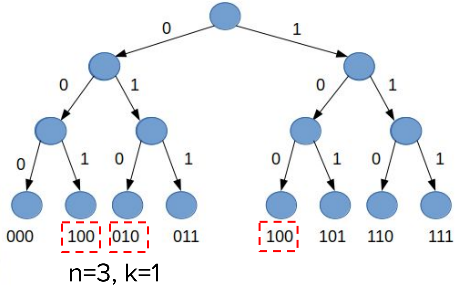
```python
def process_solution(S):
    for i in range(len(S)):
        print(S[i], end = " ")
    print("")
    return False

def subsets(S,n,i):
    #print("subsets({},{},{})".format(S,n,i))
    C = [1, 0] if i<n else []
    for c in C:
        S[i] = c
        if i == n-1:
            #print("\t\tS:{} c:{} i:{}".format(S,c,i))
            if process_solution(S):
                return True
        else:
            #print("\tCalling:  subsets({},{},{})".format(S,n,i+1))
            subsets(S,n,i+1)
    return False

n = 5
S = [0]*n
subsets(S,n,0)
```

- There is no pruning. All the possible space is explored. But this is required by the definition of the problem
- Computational complexity $O(n2^n)$ (→ i.e. 2^n sets, printing each costs n)
- In which order sets are printed? (→ 11111 first and then values decrease...)
- Is it possible to think to an iterative version, ad-hoc for this problem? (non-backtracking)

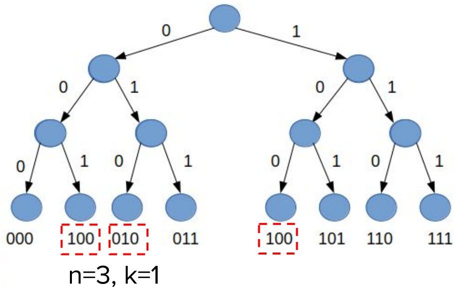
```python
def subsets(n):
    for i in range(0,2**n):
        #i is a bit mask!
        print("{0:05b}".format(i))

subsets(5)
```

Subsets problem:iterative code

List all possible subsets of size k of a set $\{0, \ldots, n-1\}$



Subsets problem:iterative code



List all possible subsets of size k of a set $\{0, \ldots, n-1\}$

```
def subsets(n, k):
    for i in range(0,2**n):
        #i is a bit mask!
        b = "{0:05b}".format(i)
        sets = [x for x in range(len(b)) if int(b[x]) == 1]

        if len(sets) == k:
            print("{} --> subset: {}".format(b,sets))

subsets(5,3)
```

all subsets
(cost: O(2^n))

creation of the
subsets (cost: O(n))

printing subsets
(cost: O(n))

```
00111 --> subset: [2, 3, 4]
01011 --> subset: [1, 3, 4]
01101 --> subset: [1, 2, 4]
01110 --> subset: [1, 2, 3]
10011 --> subset: [0, 3, 4]
10101 --> subset: [0, 2, 4]
10110 --> subset: [0, 2, 3]
11001 --> subset: [0, 1, 4]
11010 --> subset: [0, 1, 3]
11100 --> subset: [0, 1, 2]
```

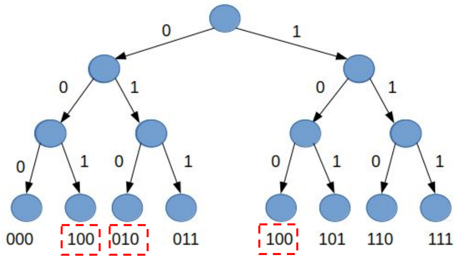
What is the complexity of this iterative code?

$$O(n \cdot 2^n)$$

How many solutions are we testing?

2^n **no pruning... can we improve this?**

Subsets problem: bactracking



n=3, k=1

List all possible subsets of size k of a set $\{0, \ldots, n-1\}$

```
def process_solution(S):
    sets = []
    for i in range(len(S)):
        print(S[i], end = "")
        if S[i] == 1:
            sets.append(i)
    print(" -> {}".format(sets))
    return False


def subsets(S, k, n, i, count ):
    C = [1,0]
    for c in C:
        S[i] = c
        count = count + c
        if i == n-1:
            if count ==  k:
                #print(S)
                process_solution(S)
        else:
            subsets(S, k, n, i+1, count)
        #backtracking:
        #print(count)
        count = count -c

n = 5
k = 3
S = [0]*n
subsets(S, k, n, 0, 0)
```

we want all solutions

count how many 1s

admissible solutions have k 1s

```
11100 -> [0, 1, 2]
11010 -> [0, 1, 3]
11001 -> [0, 1, 4]
10110 -> [0, 2, 3]
10101 -> [0, 2, 4]
10011 -> [0, 3, 4]
01110 -> [1, 2, 3]
01101 -> [1, 2, 4]
01011 -> [1, 3, 4]
00111 -> [2, 3, 4]
```

Still generates 2^n subsets, for each it will count how many 1s are present and finally print only the ones having a correct number of 1s.

What is the complexity of this backtracking code?

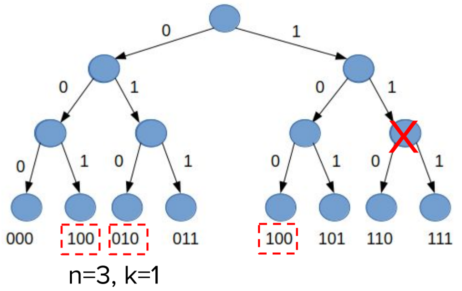
$O(n \cdot 2^n)$

How many solutions are we testing?

2^n

no pruning... can we improve this?

Subsets problem: bactracking & pruning



List all possible subsets of size k of a set $\{0, \ldots, n-1\}$

```python
#Pruning!
def subsets(S, k, n, i, count ):
    if count < k and count + (n-i) >= k:
        C = [1,0]
    else:
        C = []
    for c in C:
        S[i] = c
        count = count + c
        if count ==  k:
            #print(S)
            process_solution(S)
        else:
            subsets(S, k, n, i+1, count)
        #backtracking:
        #print(count)
        count = count -c
        S[i] = 0
n = 5
k = 3
S = [0]*n
subsets(S, k, n, 0, 0)
```

generate only solutions that can potentially be admissible!

```
11100 -> [0, 1, 2]
11010 -> [0, 1, 3]
11001 -> [0, 1, 4]
10110 -> [0, 2, 3]
10101 -> [0, 2, 4]
10011 -> [0, 3, 4]
01110 -> [1, 2, 3]
01101 -> [1, 2, 4]
01011 -> [1, 3, 4]
00111 -> [2, 3, 4]
```

What is the complexity of this iterative code?

$$O(n \cdot 2^n)$$

Sudoku

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

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

Sudoku: pseudocode

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

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

```
boolean sudoku(int[][] S, int i)

int x = i mod 9
int y = ⌊i/9⌋
SET C = Set()
if i ≤ 80 then
    if S[x, y] ≠ 0 then
        C.insert(S[x, y])
    else
        for c = 1 to 9 do
            if check(S, x, y, c) then
                C.insert(c)
int old = S[x, y]
foreach c ∈ C do
    S[x, y] = c
    if i = 80 then
        processSolution(S, n)
        return True
    if sudoku(S, i + 1) then
        return True
S[x, y] = old
return False
```

Sudoku: pseudocode

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

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

```
boolean check(int[][] S, int x, int y, int c)

for j = 0 to 8 do
    if S[x, j] = c then
        return False                                  % Column check
    if S[j, y] = c then
        return False                                  % Row check
int b_x = ⌊x/3⌋
int b_y = ⌊y/3⌋
for i_x = 0 to 2 do
    for int i_y = 0 to 2 do                           % Subtable check
        if S[b_x · 3 + i_x, b_y · 3 + i_y] = c then
            return False
return True
```

Sudoku: python code

```python
#This function prints the sudoku matrix
def process_solution(S):
    for i in range(0,9):
        if i > 0 and i % 3 == 0:
            print("--------------------------------------------------------------------")
        for j in range(0,9):
            if j % 3 == 0:
                print("|", end = "")
            print(S.get((i,j), "."), end = "\t")

        else:
            print("")
#Given a solution S, checks if c can go in (x,y)
def check_sudoku(S,x,y, c):
    for j in range(0,9):
        #column check
        if S.get((x,j),"") == c:
            return False
        #row check
        if S.get((j,y),"") == c:
            return False
    #diagonal check
    bx = x //3
    by = y //3
    for ix in range(0,3):
        for iy in range(0,3):
            if S.get((bx*3 + ix, by*3+iy),"") == c:
                return False
    return True
```

```python
#finds a backtracking solution to an input sudoku matrix S
#with brute force
def sudoku(S, i):
    x = i % 9
    y = i //9
    C = set()
    if i <= 81:
        if S[(x,y)] != 0:
            C.add(S[(x,y)])
        else:
            for c in range(1,10):
                if check_sudoku(S,x,y, c):
                    C.add(c)

    old = S.get((x,y), "")
    for c in C:
        S[(x,y)] = c
        if i == 80:
            process_solution(S)
            return True
        if sudoku(S,i+1):
            return True
        #print(old)
        if old != "":
            S[(x,y)] = old

    return False
```

```python
def initialize(S):
    for i in range(0,9):
        for j in range(0,9):
            S[(i,j)] = 0

mat = dict()
initialize(mat)
for i in range(0,9):
    mat[(i,i)] = i+1
print("Initial board:")
process_solution(mat)
print("\n\nSolution:")
sudoku(mat,0)
```

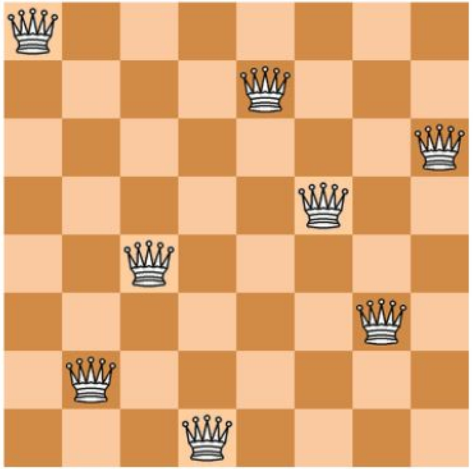
```
Initial board:
|1	0	0	|0	0	0	|0	0	0
|0	2	0	|0	0	0	|0	0	0
|0	0	3	|0	0	0	|0	0	0
--------------------------------------------------------------------
|0	0	0	|4	0	0	|0	0	0
|0	0	0	|0	5	0	|0	0	0
|0	0	0	|0	0	6	|0	0	0
--------------------------------------------------------------------
|0	0	0	|0	0	0	|7	0	0
|0	0	0	|0	0	0	|0	8	0
|0	0	0	|0	0	0	|0	0	9


Solution:
|1	8	6	|2	9	4	|3	5	7
|4	2	7	|1	3	5	|8	9	6
|5	9	3	|8	6	7	|2	4	1
--------------------------------------------------------------------
|2	1	9	|4	7	8	|6	3	5
|3	6	4	|9	5	2	|1	7	8
|8	7	5	|3	1	6	|9	2	4
--------------------------------------------------------------------
|9	3	8	|5	4	1	|7	6	2
|6	4	1	|7	2	9	|5	8	3
|7	5	2	|6	8	3	|4	1	9
```

8 queens puzzle

Problem

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other

- History:
  - Introduced by Max Bezzel (1848)
  - Gauss found 72 of the 92 solutions
- Let's start from the stupidest approach, and let's refine the solution step by step



8 queens puzzle

Idea: every column must contain exactly one queen

$S[0:n]$ coordinates in $\{0 \dots n-1\}$	permutations of $\{1 \dots n\}$
$\textsc{isAdmissible}()$	$i == n$
choices(S, n, i)	$\{0 \dots n-1\}$
pruning	removes diagonals
# Solutions for $n = 8$	$n! = 8! = 40320$

Comments

- Solutions actually visited = 15720

```
def queens(S, i, columns):
  for c in columns:
    S[i] = c
    columns.remove(c)
    if (diagonalsOK(S,i)):
      if len(columns)==0:
        printBoard(S)
      else:
        queens(S,i+1,columns)
    columns.add(c)
```