Algorithm design strategies

Many successful algorithms are based on a similar technique

Most well-known are:

- *Exhaustive Search*
- *Greedy Approach*
- *Divide and Conquer*
- *Dynamic Programming*
- *Iterative Improvement*

Main algorithm design strategies

- ***Exhaustive Computation.*** Generate every possible candidate solution and select an optimal solution.
- ***Greedy*.** Create next candidate solution one step at a time by using some greedy choice.
- ***Divide and Conquer*.** Divide the problem into non-overlapping subproblems of the same type, solve each subproblem with the same algorithm, and combine sub-solutions into a solution to the entire problem.
- ***Dynamic Programming*.** Start with the smallest subproblem and combine optimal solutions to smaller subproblems into optimal solution for larger subproblems, until the optimal solution for the entire problem is constructed.
- ***Iterative Improvement.*** Perform multiple iterations of the algorithm, at each iteration moving closer to the optimal solution, until no further improvement is possible.

Main algorithm design strategies

- *Exhaustive Computation*
- *Greedy Algorithms*
- *Divide and Conquer Algorithms*
- *Dynamic Programming*
- *Iterative Improvement*

There are many more design techniques

Many successful algorithms are designed using the combination of several techniques

There are also surprising interesting solutions based on original insights - which do not use any of these strategies

Main algorithm design strategies

- *Exhaustive Computation*
- *Greedy Algorithms*
- *Divide and Conquer Algorithms*
- *Dynamic Programming*
- *Iterative Improvement*

Thus, this classification should be used just for inspiration and not to constrain your creativity by forcing you to stay within a certain paradigm

Main design strategies

- ***Exhaustive Computation.*** Generate every possible candidate solution and select an optimal solution.
- ***Greedy.*** Create next candidate solution one step at a time by using some greedy choice.
- ***Divide and Conquer.*** Divide the problem into non-overlapping subproblems of the same type, solve each subproblem with the same algorithm, and combine sub-solutions into a solution to the entire problem.
- ***Dynamic Programming.*** Start with the smallest subproblem and combine optimal solutions to smaller subproblems into optimal solution for larger subproblems, until the optimal solution for the entire problem is constructed.
- ***Iterative Improvement.*** Perform multiple iterations of the algorithm, at each iteration moving closer to the optimal solution, until no further improvement is possible.

First - the baseline technique:

the Exhaustive Computation

Exhaustive Search

Brute-Force

Generate-and-Test

Exhaustive Search:

a straightforward way to solve a problem, based on the definition of the problem itself

- In many problems an optimal solution belongs to some **finite set** of **candidate solutions**
- An algorithm based on *Exhaustive Computation* generates all candidate solutions and then evaluates each candidate in turn to select an optimal solution

Enumerating all candidates: search space

When designing exhaustive algorithms we must first evaluate the size of the candidate set in order to avoid "combinatorial explosion"

- **Polynomial search space:** if the total number of candidates is polynomial in *n*, we may think of applying the exhaustive search
- **Exponential search space:** if the total number of candidates is exponential in *n*, we may apply the exhaustive search to very small problem sizes only: we must consider other techniques

Exhaustive algorithm consists of two parts:

- Generating all candidate solutions
- Checking each candidate solution

Generating and checking candidates **should be efficient**.

Checking usually is, generating usually isn’t

Example: Maximum Sublist problem

Suppose that we have a list of integers, and we want to find **a sublist with the maximum sum**.

(A *sublist* must be contiguous within the list)

We call such a sublist a ***maximum sublist***

Problem instance:

What is the maximum sublist in the following list?

-2	1	-3	4	-1	2	1	-5	4

Maximum Sublist: real-life applications

1. Stock Analysis: suppose we have some insider information on the future stock prices of a company like Apple. We want to maximize our profit while making only one buy and one sell transaction to avoid suspicion.

2. Used in Bioinformatics: identify highly scored regions of sequences ([Example]())

3. Used in Image Analysis: identify brightest regions within the image (2D version of the problem - find the contiguous *submatrix* with the largest sum in a matrix) Example: astronomical *imaging problem*

Maximum Sublist: variations

Two variations of *maximum sublist problem* are formalized below:

Problem: maximum sublist (value)	
Input:	array A of n integers
Output:	The sum of the sublist A[i...j] with the maximum possible value.

Problem: maximum sublist (sublist)	
Input:	array A of n integers
Output:	A sublist A[i...j] with the maximum possible value

Maximum Sublist = Subarray

What is the maximum subarray in the following array?

1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

Search space estimation:

How many different sublists in total?

Activity 8

Exhaustive search

Maximum Sublist: Algorithm 1

We start with exhaustive algorithm which evaluates the sum of all $O(n^2)$ possible sublists.

We will use this opportunity to practice writing good pseudocode that is easy to read and has no off-by-one errors.

Maximum Sublist: Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for i from 1 to n-1:
    for j from i+1 to n:
        sum = 0
        for k from i to j:
            sum ← sum + num_list[k]
            if sum > max then max ← sum

return max
```

Any <u>logical errors</u> in this code?
Watch for *boundary* and *off-by-one* errors.

Maximum Sublist: Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for i from 1 to n-1:
    for j from i+1 to n:
        sum = 0
        for k from i to j:
            sum ← sum + num_list[k]
            if sum > max then max ← sum

return max
```

i is the start of a current sublist. So, our sublist never starts at num_list[n]!

Any <u>logical errors</u> in this code?
Watch for *boundary* and *off-by-one* errors.

Maximum Sublist: Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for i from 1 to n-1:
    for j from i+1 to n:
        sum = 0
        for k from i to j:
            sum ← sum + num_list[k]
            if sum > max then max ← sum

return max
```

j is the end of a current sublist. So, we never consider sublists of length 1!

Any <u>logical errors</u> in this code?
Watch for *boundary* and *off-by-one* errors.

Maximum Sublist: Algorithm 1

```
Algorithm max_sublist1(num_list, n)

max ← - ∞
for i from 1 to n:
   for j from i to n:
      sum = 0
         for k from i to j:
            sum ← sum + num_list[k]
               if sum > max then max ← sum

return max
```

Fixed

Maximum Sublist: Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for i from 1 to n:
    for j from i to n:
        sum = 0
            for k from i to j:
                sum ← sum + num_list[k]
                if sum > max then max ← sum

return max
```

It is easy to get confused about meaning of i and j

			Left			Right									
1	-5	4	2	-7	3	6	-1	2	-4	7	-10	2	6	1	-3

So let's replace them with left/right

Maximum Sublist: Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for left from 1 to n:
   for right from left to n:
      sum = 0
      for k from left to right:
         sum ← sum + num_list[k]
         if sum > max then max ← sum

return max
```

Full and correct exhaustive algorithm

Running time of Algorithm 1

How many different sublists are in a list of length *n*?

- *There are $O(n^2)$ sublists.*

How much time does it take to evaluate the sum of each sublist with Algorithm 1?

- *Summing up a sublist of length k takes $O(k)$-time.*
  In the worst-case this is $O(n)$-time since the longest sublist has length n.

What is the total runtime of Algorithm 1?

- *There are $O(n^2)$ sublists, and evaluating each sublist takes $O(n)$-time.*
  Therefore, the algorithm takes $O(n^3)$-time.

Can we do better?

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

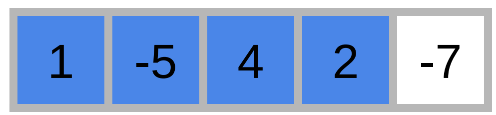
Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.



Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

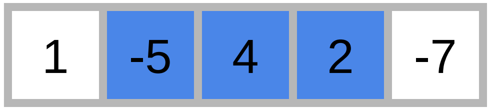
Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.



Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

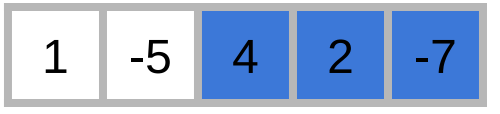
Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.



Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Meditating on Algorithm 1

The order of the sublists in Algorithm 1 is illustrated below.

1	-5	4	2	-7

Revisiting Algorithm 1

Algorithm max_sublist1(num_list, n)

```
max ← - ∞
for left from 1 to n:
    for right from left to n:
        sum = 0
        for k from left to right:
            sum ← sum + num_list[k]
                if sum > max then max ← sum

return max
```

Look at this summation.
It repeats the same computation

$$\text{sum} = x_1 + x_2 + x_3 + x_4$$

Is immediately followed by:

$$\text{sum} = x_1 + x_2 + x_3 + x_4 + x_5$$

We could avoid recomputing this sum at every iteration of the inner loop

Maximum Sublist: Algorithm 2

```
Algorithm max_sublist2(num_list, n)

max ← - ∞
for left from 1 to n:
    sum = 0
    for right from left to n:
        sum ← sum + num_list[right]
        if sum > max then max ← sum

return max
```

This should be faster than Algorithm 1, since we got rid of the third nested loop.

What is the time complexity of Algorithm 2?

Maximum Sublist: Algorithm 2

Algorithm max_sublist2(num_list, n)

```
max ← - ∞
for left from 1 to n:
    sum = 0
    for right from left to n:
        sum ← sum + num_list[right]
            if sum > max then max ← sum

return max
```

The sublists are generated in the same order as in Algorithm 1, but we apply
Optimization: avoiding recomputation between successive candidates

Maximum Sublist: Algorithm 2

```
Algorithm max_sublist2(num_list, n)

max ← - ∞
for left from 1 to n:
    sum = 0
    for right from left to n:
        sum ← sum + num_list[right]
            if sum > max then max ← sum

return max
```

The algorithm is now $O(n^2)$

Can we do better?

Maximum Sublist: Algorithm 3??

Recall that in both Algorithm 1 and Algorithm 2 we generated all n^2 possible sublists

1	-5	4	2	-7

Is there a smaller natural set of candidate solutions?

Maximum Sublist: Algorithm 3??

Recall that in both Algorithm 1 and Algorithm 2 we generated all n^2 possible sublists

1	-5	4	2	-7

The candidate set has exactly one non-empty max sublist ending at each position of the list.

Each sublist must end at position 1, 2, 3, 4, or 5. We can generate 5 different non-empty sublists, **one for each end position**, with the maximum sum among all sublists ending at this position.

Finally, the algorithm will select the max of all candidate sums.

Exhaustive Search: optimizations

1. Avoid recomputation between successive candidates (Max-sublist 2)
2. Reduce the size of the candidate set (Max-sublist 3, Euclidean GCD)

Search space: polynomial vs. exponential

- **Polynomial search space:** the max sublist problem had a polynomial search space, and exhaustive search proved useful, especially after applying some optimizations techniques to bring down the degree of the polynomial

- **Exponential search space:** If the total number of candidates is exponential in *n*, the exhaustive search becomes not feasible - especially for large values of *n*

Introducing the Thief Problem

The **Thief** Problem:

- There are *n* different items in a store
- Item *i* weighs w_i pounds and is worth $\$v_i$
- A thief breaks in. He can carry up to *W* pounds in his ***knapsack***
- What should he take to maximize the profit of his haul?

Knapsack motivations

- Least wasteful way to use raw materials
- Selecting capital investments and financial portfolios
- Generating keys for the Merkle-Hellman cryptosystem
- …

Exhaustive solution for the Thief Problem

- Consider **every possible subset of items**
- Calculate total value and total weight of each subset and discard if more than *W*
- Then choose from remaining subsets the one with maximum total value

Takes **$\Omega(2^n)$** time (for generating subsets)

Knapsack Example

- item 1: 7 lbs, $42
- item 2: 3 lbs, $12
- item 3: 4 lbs, $40
- item 4: 5 lbs, $25

- W = 10

subset	total weight	total value
Ø	0	$0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1,2}	10	$54
{1,3}	11	infeasible
{1,4}	12	infeasible
{2,3}	7	$52
{2,4}	8	$37
etc. ...		

We have to check 2^4=16 possibilities

Enumerating all subsets of a set with *n* items

1. Loop from 0 to 2^n -1
2. For each number get the binary representation of the number, e.g. 3 = 0011 (easy in Python: *bin(n)*)
3. Determine from the binary representation whether or not to include an item from the set, e.g. 0011 = [exclude, exclude, include, include]

This generates a *lexicographic ordering* of bits.

0	0000	{}
1	0001	{a}
2	0010	{b}
3	0011	{a, b}
4	0100	{c}
5	0101	{a, c}
6	0110	{b, c}
7	0111	{a, b, c}
8	1000	{d}
9	1001	{a, d}
10	1010	{b, d}
11	1011	{a, b, d}
12	1100	{c, d}
13	1101	{a, c, d}
14	1110	{b, c, d}
15	1111	{a, b, c, d}

Setting on different bits indicating subsets of *n=4* items

Subset generation: bottleneck of the Exhaustive Knapsack

There are more efficient ordering algorithms (see paper):

- Gray Codes
- Banker's Sequence

For **n = 100**, there are 2^{100} subsets, about 10^{30}.

Assuming a computer capable of checking 10^8 subsets per second, we would require about 10^{22} seconds to generate all subsets of n=100 items, or about $\mathbf{4 \times 10^{14}}$ years.

If we use Gray codes - the time is half of that. Does it help?

Can we do better?

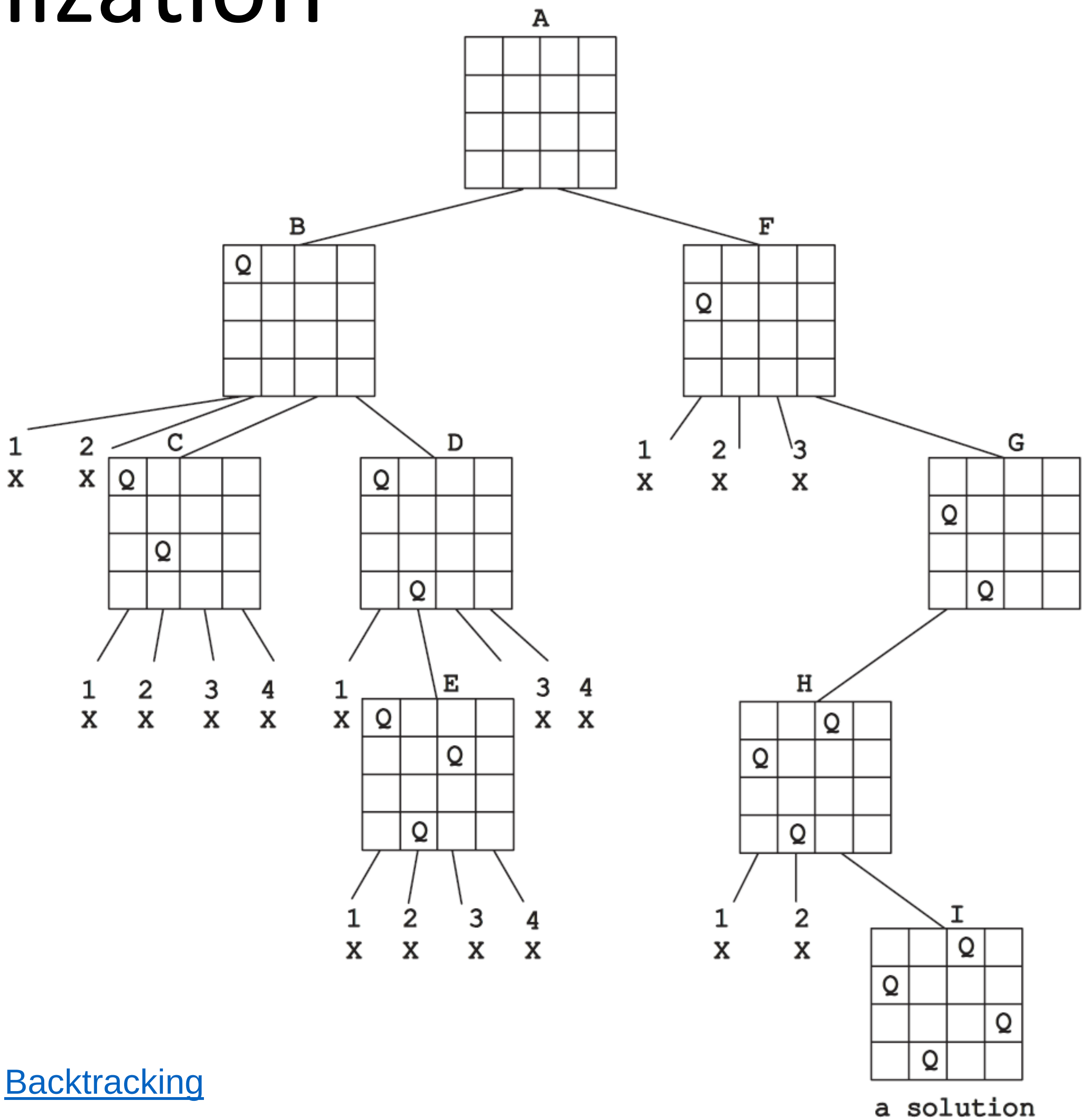
Yes, see *dynamic-programming*

The n-Queens Puzzle

Place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same column, row, or diagonal.

Solve

Example of backtracking optimization



Backtracking

Backtracking algorithm

- Construct solutions systematically - by adding one component at a time
- Evaluate candidate partial solution:
  - If it can be developed further without violating problem constraints:
    - add the next element in order
  - Else
    - any remaining component does not need to be considered
    - backtrack and replace the last component of the partial solution with the next option

Three optimization techniques for Exhaustive Computation

1. Avoid recomputation between successive candidates (Max-sublist 2)
2. Reduce the size of the candidate set (Max-sublist 3, Euclidean GCD)
3. Eliminate non-promising candidates during the search: this technique is called backtracking (n-Queens problem)