

Algorithms and Data Structures

Degree in Bioinformatics, UPF

Ramon Ferrer-i-Cancho

Dept. CS, UPC

2021–22 (Winter-Spring)

These materials are borrowed from José L. Balcázar, their creator.

Personnel (current)

Current professor

► Ramon Ferrer-i-Cancho

`ramon.ferrer@upc.edu` (primary)

`ramon.ferrer@prof.esci.upf.edu` (secondary)

<https://cqlab.upc.edu/people/rferrericancho/>
theory and lab sessions (also as coordinator of the whole course).

Personnel (past)

Former professors

- ▶ José Luis Balcázar

`jose.balcazar@prof.esci.upf.edu`

`jose.luis.balcazar@upc.edu`

most or even all theory sessions;
contributing as coordinator to the whole course;

- ▶ Edelmira Pasarella and Emma Rollón

`edelmira.pasarella@prof.esci.upf.edu`

`erollon@cs.upc.edu`

most lab contribution and preparation, most or even all lab sessions.

Logistics, I

- ▶ Meeting according to the general schedule:
Theory and Lab sessions.
- ▶ Additional personal conversations as needed:
 - ▶ Normally, I can be available at the lecture room, or in my office at ESCI, sometime before and after each seminar (lab) session.
 - ▶ Alternative slots for appointments, by email if necessary.
 - ▶ Easy to schedule a virtual meeting with Meet.
- ▶ Slides available from <http://aula.esci.upf.edu>.
If that fails [try](#).

Logistics, II

In more detail: theory

Session concepts

You know them by now.

Theory:

- ▶ lecturer mostly speaks and sometimes (not too often) demonstrates,
- ▶ students ask questions (as often as possible please),
- ▶ generally refraining from opening your laptops (except if you use them just for taking your notes).

Lab:

- ▶ lecturer briefly introduces the topic and instructions of the day,
- ▶ students work on their own and ask questions when necessary.

Course Contents

- ▶ C++ programming (for the seminar, practical and exams).
- ▶ Trees and graphs.
- ▶ Combinatorial Search Schemes:
 - ▶ Backtracking algorithms,
 - ▶ Dynamic Programming.
- ▶ Linear data structures and applications:
 - ▶ variations tree traversal and graph traversal algorithms.
- ▶ Fundamental notions of hashing, heaps, and balanced trees.
- ▶ Dynamic memory: basics of pointer programming.
- ▶ STL (the standard template library of C++).

Evaluation, I

Partly different from last year

Programming project: 25% of the grade, in some weeks.

Midterm: 25% of the grade, by the mid of the lectures (to be announced).

Presentation: 25% of the grade, last week of lectures.

Final exam: 25% of the grade, pending date and time

Recovery: pending date and time

Evaluation, II

Part individual, part in small teams

Programming project: in pairs, on assigned topics.

Presentation: with a second written task, also in (different!) pairs, on assigned topics.

All exams: individual.

Evaluation, III

Topics for presentations to negotiate with me (list to be updated)

- ▶ the “greedy” algorithmic scheme,
- ▶ the “divide-and-conquer” algorithmic scheme,
- ▶ advanced combinatorial search algorithms,
- ▶ pointer-based implementations of linear data structures,
- ▶ deques,
- ▶ pointer-based tree implementations,
- ▶ balanced AVL BST trees, balanced B+ trees,
- ▶ heap algorithms,
- ▶ implementations of hashing,
- ▶ implementations of graphs, DFS versus BFS,
- ▶ spanning trees, shortest paths,
- ▶ ...

Recursion, I

A must

About half of every course on Algorithms and(/or) Data Structures is based on recursive programs.

A recursive function

includes necessarily, among whatever else is necessary,

Base cases: solved without recursion.

Recursive cases: solved by calling the function itself, either directly or indirectly.

Test to distinguish between them (most likely an alternative instruction).

It is **crucial** that the recursive call(s) send parameters that are “in some sense smaller” than the value received:

they must **progress** towards the base cases.

Recursion, II

Indirect recursion

There is recursion whenever there is a cycle of function calls.

- ▶ Function f calls itself, or
- ▶ function f calls function g and, in turn, g calls f ,
- ▶ function f calls g , which calls h , which calls f ...

Suggestion: Find a few spare hours in the next few days to review everything you know about recursive programming and do some exercises, in order to refresh the notion.

One Word of Warning

On homonyms

Hitting with the right meaning of a word requires to take into account the context.

Example: **object**.

(Contest: who provides further words with two different meanings in different contexts, but both within computing?)

The most prominent case in our course: **heap**.

Graphs

Conceptual review

We elicit the key notions from the audience
and have a bit more fun than by mere listening.

- ▶ Graph, undirected or directed (**digraph**).
- ▶ vertex (or sometimes node), vertex weight / label,
- ▶ arc or edge, their weight / label, walk, path, closed walk, cycle (or circuit), multiple edges, self-loop,
- ▶ the handshaking lemma,
- ▶ sparse graph, dense graph, degree, regularity, connectivity, distance, diameter,
- ▶ subgraph, spanning tree, minor,
- ▶ free tree, rooted tree, ordered tree,
- ▶ bipartite graph, directed acyclic graph, topological sort, DFS,
- ▶ graph isomorphism, subgraph isomorphism,
- ▶ specific graph examples with proper names: C_3 ; C_4 ; K_5 ; $K_{3,3}$; S_4 ; S_5 ; W_3 ; W_4 ; fullerenes; Petersen; Dürer...
(check out the gallery of named graphs on Wikipedia).

Trees as Graphs

A couple of important adjectives

Easy to see a tree as a graph

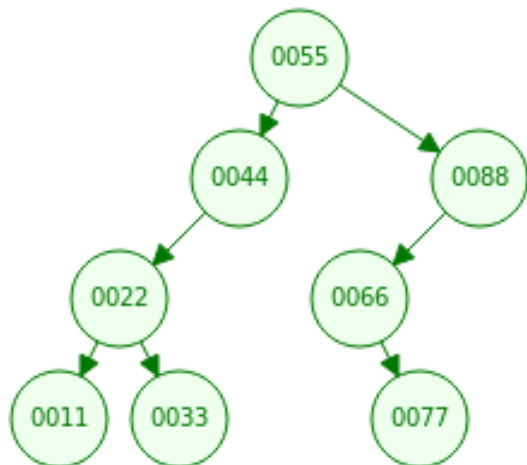
but one needs to pay a bit of attention to a couple of things.

- ▶ Rooted trees versus free trees:
 - ▶ in a **rooted tree**, there is a distinguished node that we call **root**;
 - ▶ if we do not distinguish a root, then we call it a **free tree**.
 - ▶ In a rooted tree, there is a unique path from each node to the root.
 - ▶ Then, **subtrees** (or **children**) of a node are those appearing in the direction opposite to the root.
- ▶ Ordered or unordered trees:
 - ▶ in a rooted tree, the subtrees can form a *set* (**unordered** trees) or a *sequence* (**ordered** trees).
- ▶ Very important variant: **binary trees**.

Careful: very often, you find written “tree” and must figure out the adjectives on your own, from the context.

A Binary Tree

Actually, a BST as we will study in due course



Framework for Algorithmic Schemes, I

An intuitive framework for developing algorithms and comparisons among them:
combinatorial search

There are many strategies for designing algorithms; several of them exhibit a particularly successful record.

Intuitive context to explain them

and discuss their similarities and differences:

- ▶ Notion of “instance” of a computational problem,
- ▶ notion of “candidate solutions” for each instance,
- ▶ notion of “solutions aimed at”, in two possible ways:
 - (a) mere existence (one solution? or all of them?),
 - (b) optimality (maximization? minimization?).

Of course not all computing problems fit this framework; but many do, very closely, and many more if we relax the interpretations a bit.

Examples of this Framework, I

More detail in a minute

Two examples on spanning trees:

Given a connected graph with edge weights, find in it a connected subgraph

(a) that connects all the vertices without creating cycles;

Examples of this Framework, I

More detail in a minute

Two examples on spanning trees:

Given a connected graph with edge weights, find in it a connected subgraph

- (a) that connects all the vertices without creating cycles; or
- (b) that connects all the vertices with the minimum total weight.

Examples of this Framework, I

More detail in a minute

Two examples on spanning trees:

Given a connected graph with edge weights, find in it a connected subgraph

- (a) that connects all the vertices without creating cycles; or
- (b) that connects all the vertices with the minimum total weight.

Three examples on knapsacks:

Given numbers V and W and a set of objects, each with a weight and a value, find a subset of these objects

- (a) that reaches total value at least V but weighs at most W ;
- (b) that reaches the highest possible value but weighs at most W ;
- (c) that reaches total value at least V but weighs as little as possible.

Examples of this Framework, II

About the first examples

Spanning tree:

- ▶ Notion of “instance” of a computational problem, like:
“given a connected graph with weights in the edges...”
- ▶ notion of “candidate solutions” for each instance, like:
“find in it a connected subgraph that...”;
- ▶ notion of “solutions aimed at”, in two possible ways:
 - (a) mere existence, like:
“connects all the vertices without creating cycles”;

Examples of this Framework, II

About the first examples

Spanning tree:

- ▶ Notion of “instance” of a computational problem, like:
“given a connected graph with weights in the edges...”
- ▶ notion of “candidate solutions” for each instance, like:
“find in it a connected subgraph that...”;
- ▶ notion of “solutions aimed at”, in two possible ways:
 - (a) mere existence, like:
“connects all the vertices without creating cycles”;
 - (b) optimality (maximization or minimization), like:
“connects all the vertices with the minimum total weight”.

Examples of this Framework, III

About the subsequent examples

Knapsack:

- ▶ Notion of “instance” of a computational problem, like:
“given numbers V and W and a set of objects, each with a weight and a value. . .”
- ▶ notion of “candidate solutions” for each instance, like:
“find a subset of these objects that. . .”;
- ▶ notion of “solutions aimed at”, in two possible ways:
 - (a) mere existence, like:
“reaches total value at least V but weighs at most W ”;
 - (b) optimality (maximization or minimization), like:
“reaches the highest possible value but weighs at most W ”,
or:
“reaches total value at least V but weighs as little as possible”.

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
(a) already “unacceptable”

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
 - (a) already “unacceptable”
(that is, the subproblem is **unsolvable**) or

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
 - (a) already “unacceptable”
(that is, the subproblem is **unsolvable**) or
 - (b) “acceptable” but still “incomplete”

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
 - (a) already “unacceptable”
(that is, the subproblem is **unsolvable**) or
 - (b) “acceptable” but still “incomplete”
(the subproblem might be solvable, we must go on) or

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
 - (a) already “unacceptable”
(that is, the subproblem is **unsolvable**) or
 - (b) “acceptable” but still “incomplete”
(the subproblem might be solvable, we must go on) or
 - (c) a “complete” candidate solution

Framework for Algorithmic Schemes, II

We need a bit more in our setting

Additionally:

The solution candidates are structured into

- ▶ A notion of “subproblem”, obtained through a “sequence of decisions” towards reaching candidate solutions
(subproblems are often termed “local problems”, and then the original problem is referred to as “global”);
- ▶ a function that tells us whether a sequence of decisions is
 - (a) already “unacceptable”
(that is, the subproblem is **unsolvable**) or
 - (b) “acceptable” but still “incomplete”
(the subproblem might be solvable, we must go on) or
 - (c) a “complete” candidate solution
(that is, a solution to the subproblem provides a solution to the original instance).

Framework for Algorithmic Schemes, III

Existence versus optimization

In the case of optimization problems

(either maximization or minimization) we need as well:

an **objective function** to optimize,

- ▶ defined on candidate solutions but
- ▶ in such a way that it naturally extends to local subproblems (sequences of decisions).

Framework for Algorithmic Schemes, IV

Often there is **more than one** way to set up the scheme

Spanning tree (with or without connectivity?)

Subproblem:

- ▶ find the spanning tree of a subgraph, or, alternatively,
- ▶ complete a single spanning tree of the graph given
 - ▶ one partial tree already constructed, or

Framework for Algorithmic Schemes, IV

Often there is **more than one** way to set up the scheme

Spanning tree (with or without connectivity?)

Subproblem:

- ▶ find the spanning tree of a subgraph, or, alternatively,
- ▶ complete a single spanning tree of the graph given
 - ▶ one partial tree already constructed, or
 - ▶ a set of partial trees already constructed (spanning forest)...

Framework for Algorithmic Schemes, IV

Often there is **more than one** way to set up the scheme

Spanning tree (with or without connectivity?)

Subproblem:

- ▶ find the spanning tree of a subgraph, or, alternatively,
- ▶ complete a single spanning tree of the graph given
 - ▶ one partial tree already constructed, or
 - ▶ a set of partial trees already constructed (spanning forest)...

Sequence of decisions: grow a current tree by one further edge.

- (a) already unacceptable: the new edge creates a cycle;
- (b) complete candidate solution: connects all vertices;
- (c) acceptable but still incomplete: rest of cases.

Framework for Algorithmic Schemes, IV

Often there is **more than one** way to set up the scheme

Spanning tree (with or without connectivity?)

Subproblem:

- ▶ find the spanning tree of a subgraph, or, alternatively,
- ▶ complete a single spanning tree of the graph given
 - ▶ one partial tree already constructed, or
 - ▶ a set of partial trees already constructed (spanning forest)...

Sequence of decisions: grow a current tree by one further edge.

- (a) already unacceptable: the new edge creates a cycle;
- (b) complete candidate solution: connects all vertices;
- (c) acceptable but still incomplete: rest of cases.

Objective function:

- ▶ weight of the current partial tree?

Framework for Algorithmic Schemes, IV

Often there is **more than one** way to set up the scheme

Spanning tree (with or without connectivity?)

Subproblem:

- ▶ find the spanning tree of a subgraph, or, alternatively,
- ▶ complete a single spanning tree of the graph given
 - ▶ one partial tree already constructed, or
 - ▶ a set of partial trees already constructed (spanning forest)...

Sequence of decisions: grow a current tree by one further edge.

- (a) already unacceptable: the new edge creates a cycle;
- (b) complete candidate solution: connects all vertices;
- (c) acceptable but still incomplete: rest of cases.

Objective function:

- ▶ weight of the current partial tree?
- ▶ best possible weight for a complete spanning tree that extends the given one?

Framework for Algorithmic Schemes, V

Often there is **more than one** way to set up the scheme

Knapsack:

Reach the highest possible value with weight at most W .

Subproblem: given a set of objects already selected, add further objects to it.

Framework for Algorithmic Schemes, V

Often there is **more than one** way to set up the scheme

Knapsack:

Reach the highest possible value with weight at most W .

Subproblem: given a set of objects already selected, add further objects to it.

Sequence of decisions: consider one further object; either **pick** it or **discard** it.

- (a) already unacceptable: the new total weight is over W ;
- (b) complete candidate solution: no further undecided objects remain;
- (c) acceptable but still incomplete: rest of cases.

Framework for Algorithmic Schemes, V

Often there is **more than one** way to set up the scheme

Knapsack:

Reach the highest possible value with weight at most W .

Subproblem: given a set of objects already selected, add further objects to it.

Sequence of decisions: consider one further object; either **pick** it or **discard** it.

- (a) already unacceptable: the new total weight is over W ;
- (b) complete candidate solution: no further undecided objects remain;
- (c) acceptable but still incomplete: rest of cases.

Objective function:

- ▶ current value?
- ▶ best possible value attainable by expanding the current choice?

Framework for Algorithmic Schemes, VI

Often there is **more than one** way to set up the scheme

Knapsack again:

Reach the lowest possible weight with a value of at least V .

Framework for Algorithmic Schemes, VI

Often there is **more than one** way to set up the scheme

Knapsack again:

Reach the lowest possible weight with a value of at least V .

Subproblem: given a set of objects not yet discarded, discard further objects from it.

(Complete the scheme on your own.)

Exhaustive Search, I

Might we as well “try all possible solutions”?

We concentrate first on existence cases.

When a new (to us) problem comes:

How do we proceed?

1. Formalize it following the combinatorial search scheme.
2. Maybe explore alternative ways to cast the problem into the scheme, until finding some very smart ad-hoc algorithm that works (e.g. so-called “greedy schemes”). . .

Exhaustive Search, I

Might we as well “try all possible solutions”?

We concentrate first on existence cases.

When a new (to us) problem comes:

How do we proceed?

1. Formalize it following the combinatorial search scheme.
2. Maybe explore alternative ways to cast the problem into the scheme, until finding some very smart ad-hoc algorithm that works (e.g. so-called “greedy schemes”) . . .
or giving up.
3. What then?

Exhaustive Search, I

Might we as well “try all possible solutions”?

We concentrate first on existence cases.

When a new (to us) problem comes:

How do we proceed?

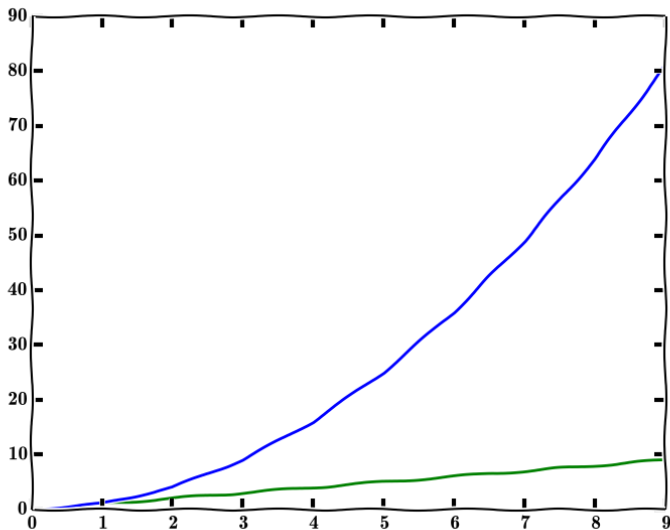
1. Formalize it following the combinatorial search scheme.
2. Maybe explore alternative ways to cast the problem into the scheme, until finding some very smart ad-hoc algorithm that works (e.g. so-called “greedy schemes”) . . .
or giving up.
3. What then?

Shall we explore all possibilities?

- ▶ All subsets (the whole powerset) . . . 2^N cases.
- ▶ All permutations . . . $N!$ cases.

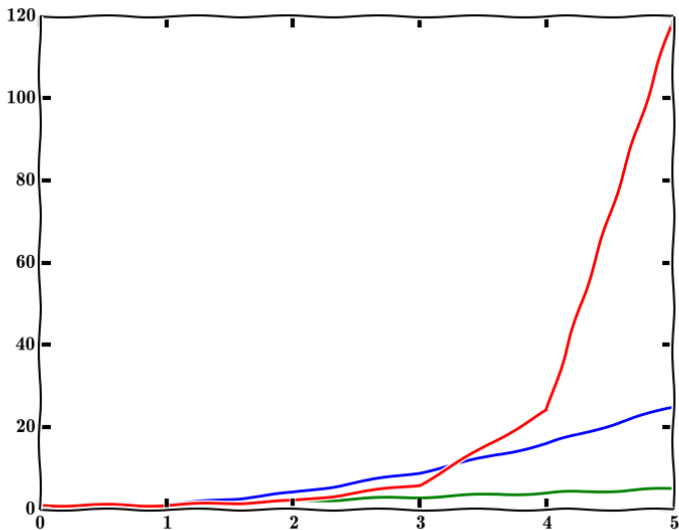
Factorial Function and Exponential Growth, I

Thou shalt not take the name of the Exponential in vain



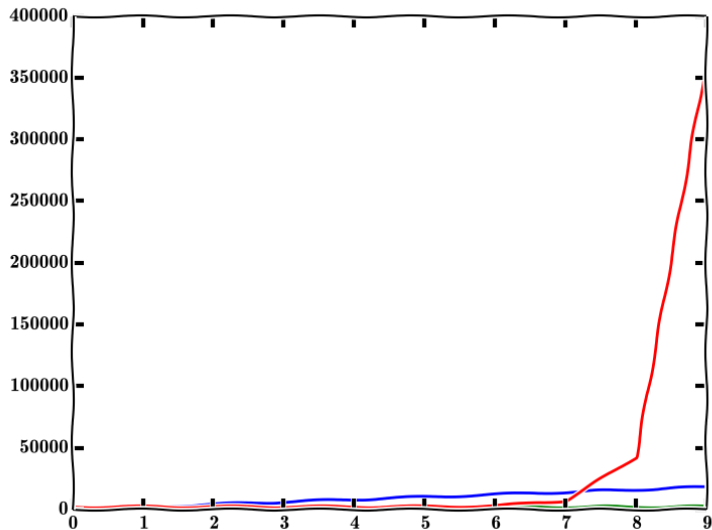
Factorial Function and Exponential Growth, I

Thou shalt not take the name of the Exponential in vain



Factorial Function and Exponential Growth, I

Thou shalt not take the name of the Exponential in vain



Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dicit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do 13000 trillion operations per second (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dixit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do **13000 trillion operations per second** (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

for $N = 15$: 8 thousandths of a second (8×10^{-3});

Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dixit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do **13000 trillion operations per second** (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

for $N = 15$: 8 thousandths of a second (8×10^{-3});

for $N = 18$: half a second;

Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dixit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do **13000 trillion operations per second** (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

for $N = 15$: 8 thousandths of a second (8×10^{-3});

for $N = 18$: half a second;

for $N = 21$: one hour;

Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dixit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do **13000 trillion operations per second** (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

for $N = 15$: 8 thousandths of a second (8×10^{-3});

for $N = 18$: half a second;

for $N = 21$: one hour;

for $N = 24$: one and a half years;

Factorial Function and Exponential Growth, II

$N!$ grows exponentially, Stirling dixit

Suppose:

- ▶ we only need one elementary operation for each of $N!$ configurations, and
- ▶ we could do **13000 trillion operations per second** (13×10^{15}).

Then we would spend

for $N = 12$: billionths of a second (10^{-9});

for $N = 15$: 8 thousandths of a second (8×10^{-3});

for $N = 18$: half a second;

for $N = 21$: one hour;

for $N = 24$: one and a half years;

for $N = 27$: over 250 centuries. . .

Exhaustive Search, II

Try all possible solutions **only** if you must — and, then, do it well

Confronted with a case that seems to need exhaustive search
(Don't forget to ask around whether anybody has proved
NP-hardness!)

1. Focus on **existence** first, leave optimization for the subsequent stage;
2. throw in a quick-and-dirty, **exponentially slow** but fast-to-program algorithm to make sure that you need to do better, and to get some counts on quantities of different subproblems involved;
3. go for a **backtracking** solution;
4. frequent repeated subproblems?, consider trying a **dynamic programming** approach (maybe after backtracking, or maybe directly head-on).

Exhaustive Search, III

Keep learning about algorithmics!

Once the backtracking solution is in place, expand your solution into the optimization case, if needed:

- ▶ Best-first search?

(That is, A^* and cousins:

https://en.wikipedia.org/wiki/Best-first_search.)

- ▶ Branch-and-bound?

- ▶ Extra sophistications? (branch-and-cut, alpha-beta pruning. . .)

All the time, paying attention to learn more about algorithmics and checking whether they can be put to work on your specific problem (greedy schemes, divide-and-conquer, linear programming, quadratic/semidefinite programming. . .)

Backtracking

A basic intuition behind many other variants

What is it?

Akin to Depth-First Search, except that the graph remains **implicit**.

- ▶ Imagine each subproblem as a **vertex** in a (very large) graph.
- ▶ **Edges** in that imaginary graph correspond to decisions that change a subproblem into another one.
- ▶ In this course, we only consider the case of an **implicit tree**.

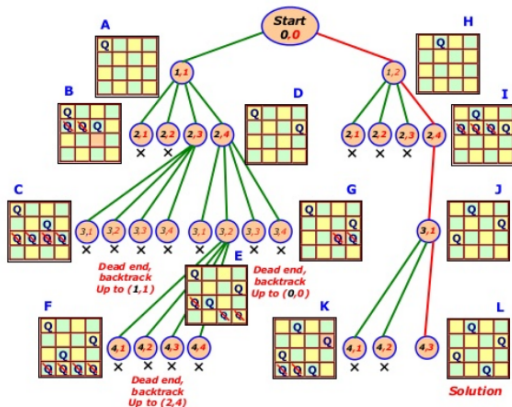
The main idea is:

If we identify that a subproblem is not feasible, we spare ourselves traversing all the solutions that include attempts to solve that subproblem.

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

Example: N-queens, I

The implicit tree: generated and explored part up to the first solution



Source: <https://www.slideshare.net/praveenkumar33449138/02-problem-solvingsearchcontrol>

Example: N-queens, II

Traversal of the implicit tree

Akin to depth-first search

where vertices can be already unacceptable (**dead ends**) and, hence, no subtrees are visited anymore.

Lab2***

Three options:

1. Simplest: we want all solutions and must complete the traversal.
2. Search-like scheme: stop the exploration if we are satisfied with only one solution.
3. Optimization case: want to see all solutions to pick the “best” one according to some criterion (does not apply to N-queens).

Example: N-queens, III

Finding all solutions

Lab2****

```
def attempt(row, board, size):  
    if row == size:  
        board.draw()  
    else:  
        for column in range(size):  
            if board.free(row, column):  
                board.put_q(row, column)  
                attempt(row + 1, board, size)  
                board.remove_q(row, column)
```

Initial call:

```
board = Board()  
size = int(input("How many queens? "))  
attempt(0, board, size)
```

Example: N-queens, IV

Finding one solution

Lab2*****

```
def attempt(row, board, size):
    if row == size:
        return True
    else:
        for column in range(size):
            if board.free(row, column):
                board.put_q(row, column)
                s = attempt(row + 1, board, size)
                if s:
                    return True
            else:
                board.remove_q(row, column)
        return False
```

Example: N-queens, IV

Finding one solution

Lab2*****

```
def attempt(row, board, size):
    if row == size:
        return True
    else:
        for column in range(size):
            if board.free(row, column):
                board.put_q(row, column)
                s = attempt(row + 1, board, size)
                if s:
                    return True
                else:
                    board.remove_q(row, column)
        return False
```

Initial call: declare board, get size, and call thus:

```
if attempt(0, board, size):
    board.draw()
```

Example: N-queens, V

Can we **do better**?

Ideas to follow up:

- ▶ Find ways to avoid exploring a partial solution that is symmetrical to one already explored and failed.
- ▶ Explore each row in a different order:
 - ▶ for every square in the current row, compute how many squares in subsequent rows would be lost from there,
 - ▶ then explore squares that leave as much freedom as possible before those that are more restrictive (“best-first search”).
- ▶ ...

Graph Colorability

Many practical problems can be reduced to these

Vertex coloring:

Given a graph, assign a color to each vertex in such a way that no edge connects two vertices of the same color.

[http://mathworld.wolfram.com/images/eps-gif/
VertexColoring_750.gif](http://mathworld.wolfram.com/images/eps-gif/VertexColoring_750.gif)

Edge coloring:

Given a graph, assign a color to each edge in such a way that no vertex shows two meeting edges of the same color.

[http://mathworld.wolfram.com/images/eps-gif/
EdgeColoring_850.gif](http://mathworld.wolfram.com/images/eps-gif/EdgeColoring_850.gif)

We focus on **edge coloring** today.

Example: 3-colorability of 3-regular graphs, I

The implicit tree: graph with more and more colored edges

Restriction:

we consider today only 3-regular graphs (all vertices of degree 3).

That is, our problem is: given a 3-regular graph G , color edges with three colors so that all three meet at every vertex.

Ideas for the backtracking scheme:

- ▶ One vertex of the implicit graph corresponds to the whole graph G with part of the edges colored.

Example: 3-colorability of 3-regular graphs, I

The implicit tree: graph with more and more colored edges

Restriction:

we consider today only 3-regular graphs (all vertices of degree 3).

That is, our problem is: given a 3-regular graph G , color edges with three colors so that all three meet at every vertex.

Ideas for the backtracking scheme:

- ▶ One vertex of the implicit graph corresponds to the whole graph G with part of the edges colored.
- ▶ Dead ends: the coloring is already incompatible.

Example: 3-colorability of 3-regular graphs, I

The implicit tree: graph with more and more colored edges

Restriction:

we consider today only 3-regular graphs (all vertices of degree 3).

That is, our problem is: given a 3-regular graph G , color edges with three colors so that all three meet at every vertex.

Ideas for the backtracking scheme:

- ▶ One vertex of the implicit graph corresponds to the whole graph G with part of the edges colored.
- ▶ Dead ends: the coloring is already incompatible.
- ▶ Immediate subtrees: the same graph with one more edge colored.

Example: 3-colorability of 3-regular graphs, I

The implicit tree: graph with more and more colored edges

Restriction:

we consider today only 3-regular graphs (all vertices of degree 3).

That is, our problem is: given a 3-regular graph G , color edges with three colors so that all three meet at every vertex.

Ideas for the backtracking scheme:

- ▶ One vertex of the implicit graph corresponds to the whole graph G with part of the edges colored.
- ▶ Dead ends: the coloring is already incompatible.
- ▶ Immediate subtrees: the same graph with one more edge colored.

But... which one?

Example: 3-colorability of 3-regular graphs, I

The implicit tree: graph with more and more colored edges

Restriction:

we consider today only 3-regular graphs (all vertices of degree 3).

That is, our problem is: given a 3-regular graph G , color edges with three colors so that all three meet at every vertex.

Ideas for the backtracking scheme:

- ▶ One vertex of the implicit graph corresponds to the whole graph G with part of the edges colored.
- ▶ Dead ends: the coloring is already incompatible.
- ▶ Immediate subtrees: the same graph with one more edge colored.

But... which one?

To keep the implicit graph a tree, we must be a bit careful.

Example: 3-colorability of 3-regular graphs, II

One possible approach

Force an order on the edges

and maintain it strictly, so that if a path in the implicit graph colors first edge e_1 and later edge e_2 , then the same happens to all paths.

Then, the implicit graph is effectively a tree.

Example: 3-colorability of 3-regular graphs, II

One possible approach

Force an order on the edges

and maintain it strictly, so that if a path in the implicit graph colors first edge e_1 and later edge e_2 , then the same happens to all paths.

Then, the implicit graph is effectively a tree.

Demo

based on NetworkX and GraphViz:

- ▶ we fix an order of the edges as given by the NetworkX `graph.edges()` method;
- ▶ we keep a list of available colors at each vertex;
- ▶ we try using in turn each available color for the current edge and launch the recursive call, break the loop and finish the recursions if successful.

Additional paraphernalia to keep reporting and to draw the graph (e. g. the dict `gd` with the GraphViz layout).

Example: 3-colorability of 3-regular graphs, III

Finding one solution

```
def tricolor(g, edgelist):
    if not edgelist: return True
    else:
        u, v = edgelist.pop()
        possib = g.node[u]['free'] & g.node[v]['free']
        for c in possib:
            g.edge[u][v]['color'] = c
            g.node[u]['free'].remove(c)
            g.node[v]['free'].remove(c)
            success = tricolor(g, edgelist)
            if success: return True
            # else, free again the colors, try next possib
            g.edge[u][v]['color'] = noncolor
            g.node[u]['free'].add(c)
            g.node[v]['free'].add(c)
        edgelist.append((u, v)) # backtrack!
    return False
```

Example: 3-colorability of 3-regular graphs, IV

Can we **do better**?

Ideas to follow up:

- ▶ Fix the names of the three colors of some particular vertex so as to avoid exploring subtrees that only differ in the names of the colors.
- ▶ Design carefully the edge ordering.
 - ▶ For instance, a depth-first search strategy may be a good idea.
 - ▶ Indeed, this ensures that, at each edge, at least one of the endpoints has already lost at least one color.
- ▶ ...

Knapsack, I

Towards backtracking for optimization

Given:

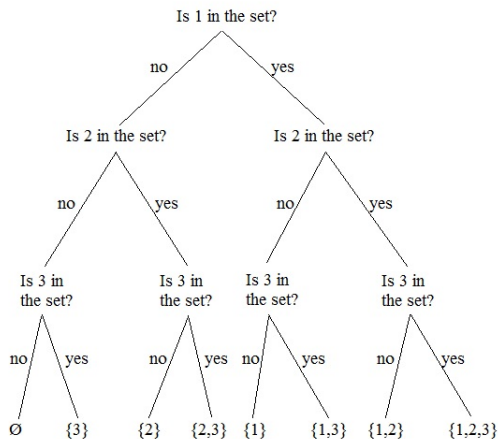
- ▶ objects $i \in \{0, \dots, N - 1\}$
- ▶ with values $v[i]$ and weights $w[i]$,
- ▶ maximum capacity of knapsack W :

report the best set of objects to take for the knapsack:

- ▶ total weight does not exceed the maximum capacity,
- ▶ total value is as large as possible.

Knapsack, II

The implicit tree spans the powerset



By: Brian M. Scott at math.stackexchange.com

Knapsack, III

Then, simply scan the whole powerset

Lab2*

```
def slow_knapsack(objects, W):  
    mx = 0  
    for cand in powerset(objects):  
        if totalweight(cand) <= W and totalvalue(cand) > mx:  
            best = cand  
            mx = totalvalue(best)  
    return best, mx
```

You have a choice of options for the powerset function:

- ▶ Recipe in the Python documentation, chapter on itertools, section “itertools recipes” (provides subsets ordered by size).
- ▶ Basic recursion example in Ricard’s slides of previous quarter.
- ▶ My favorite: a recursive generator (learn on yourself about generators and come up with it, ask me if you want to compare with mine).

Knapsack, IV

Simple backtracking in C++

Returns the best feasible value:

Lab2**

```
int knapsack(const vector<int>& v,
             const vector<int>& w,
             int i, int W){

    if (i == 0 or W == 0) return 0;
    else if (w[i-1] > W) return knapsack(v, w, i-1, W);
    else return max(v[i-1] + knapsack(v, w, i-1, W-w[i-1]),
                    knapsack(v, w, i-1, W));

}
```

Knapsack, V

Simple backtracking in Python (compare to similar solution in the previous quarter)

```
def best_knapsack(values, weights, itm, limw):
    "best knapsack under limw weight w/ items in range(itm)"
    if itm == 0 or limw == 0:
        "no items available, emptyset is only solution"
        return set(), 0
    else:
        "solve first excluding current item, itm-1"
        k0, v0 = best_knapsack(values, weights, itm-1, limw)
        if weights[itm-1] <= limw:
            "current item fits, so solve now including it"
            k1, v1 = best_knapsack(values, weights, itm-1,
                                    limw-weights[itm-1])
            if v0 < v1 + values[itm-1]:
                "second solution is better"
                k1.add(itm-1)
                return k1, v1 + values[itm-1]
    return k0, v0
```

Knapsack, VI

If we want to go beyond, I suggest embracing OO fully

A number of additional ideas can be implemented:

- ▶ Can we report the successive updates to the “best-so-far” solution?
- ▶ Should we explore the items in some specific order? (Careful with the way the recursive calls handle that!)
 - ▶ Weight?
 - ▶ Value?
 - ▶ Ratio value/weight?
 - ▶ Increasing or decreasing?
- ▶ ...

Adding functionality to the simple scheme is the wrong approach!

- ▶ Code gets complicated and sloppy very soon.
- ▶ At some early point the quantity of rig-like decisions makes the program go wrong with very little control for correcting it.
- ▶ Reach up to **object orientation**.

Knapsack, VII

```
class Itm:
    def __init__(self, ident):
        print("Item " + str(ident) + ":")
        self.v = int(input("    Value: "))
        self.w = int(input("    Weight: "))
        self.ident = ident

    def ratio(self):
        return float(self.v) / self.w

    def __str__(self):
        return str(self.ident)

[...]
```

```
for itm in range(itq):
    its.append(Itm(itm))
its = sorted(its, key = lambda x: x.ratio())
```

Knapsack, VIII

```
def best_knapsack(items, itm, curr, best):
    "best knapsack under limit weight w/ items below itm"
    if itm == 0:
        "no further items available, check for new best"
        if curr.v > best.v:
            best.mimic(curr)
            print("Current best knapsack is:", best)
    else:
        "two options: take (if it fits) or ignore"
        it = items[itm-1]
        if curr.w + it.w <= curr.mxw:
            curr.grab(it)
            best_knapsack(items, itm-1, curr, best)
            curr.trash(it)
        best_knapsack(items, itm-1, curr, best)
    [...]
best, curr = Knapsack(mxw), Knapsack(mxw)
best_knapsack(its, len(its), curr, best)
```

Suggested Tasks, I

On your own

- ▶ Watch for as long as necessary this visualization of the N-queens backtracking, using the pause and step-by-step and speed controls, so as to understand thoroughly how it works. (We will return often in the course to that algorithm and data structure [visualization webpage](#).)
- ▶ Complete various versions of knapsack backtracking:
 - ▶ Python and C++?
 - ▶ With and without OO?
 - ▶ Lowest weight with enough value instead of highest value under weight limit.
 - ▶ Other variations you come up with.
 - ▶ The problem of **Giving Change**:
 - ▶ Set it up in the framework of combinatorial search.
 - ▶ Complete a backtracking algorithm (and program) for it.

Giving change, I

A classical example

Lab3*

Given available coin denominations

and as much a supply of coins of each denomination as necessary:

- ▶ denominations $d_1, \dots, d_n$;
- ▶ goal quantity M to reach.

Relationship to knapsack:

- ▶ “Repeated” objects, we can take several of each.
- ▶ The goal is to be reached **exactly**.
- ▶ Various possibilities:
 - ▶ Do we see the denominations as values?
 - ▶ or as weights?
 - ▶ or as both?

Giving change, II

The **greedy** scheme

Greedy approach:

- ▶ Keep adding coins of as large a denomination as feasible.
- ▶ For some denomination sets, this greedy approach works: **canonical coin systems**, like the typical cases of many countries:
 - ▶ 1, 2, 5, 10, 20, 50;
 - ▶ 1, 5, 10, 25, 50, 100, 200;
 - ▶ ...
- ▶ For others, it does not; for instance:
get 15 units out of 1-unit, 5-units, and 8-units elements!

https://en.wikipedia.org/wiki/Change-making_problem.

The Greedy Algorithmic Schema

What is common between that solution to giving change and Kruskal?

Characteristics:

- ▶ Notion of “current optimality”;
- ▶ next choice is always the one that “looks best” given the current situation:
 - every locally optimal choice is globally optimal
(greedy-choice property);
- ▶ need to argue separately that it is a good idea to go for the best choice, given that the full picture is unknown.

https://en.wikipedia.org/wiki/Greedy_algorithm.

Suggested Tasks, II

More options to work on your own

- ▶ Does a **greedy scheme** work for the **knapsack** problem? Two options to consider:
 - ▶ with **divisible** objects?
 - ▶ with **indivisible** objects?(and ponder carefully about what it means to ‘look best’). Implement your approaches.
- ▶ Solve the giving change problem with a greedy approach and explore for suboptimal cases.

Dynamic Programming, I

Recommended reading: [nice](#) account of the origins by Richard Bellman himself

Key points:

- ▶ Most often, no single “locally optimal decision” with guarantee that it will be globally optimal:
the “greedy” mantra fails;

Dynamic Programming, I

Recommended reading: [nice account of the origins by Richard Bellman himself](#)

Key points:

- ▶ Most often, no single “locally optimal decision” with guarantee that it will be globally optimal:
 - the “greedy” mantra fails;
- ▶ but, sometimes, a close relative might still hold, namely, Bellman’s **Principle of Optimality**:
 - the part of the optimal global solution that corresponds to any subproblem is itself a locally optimal solution.
- ▶ Particularly efficient when many repeated subproblems keep appearing while solving the original problem.

Dynamic Programming, II

Tabulating partial solutions to local subproblems

Organize subproblems and partial solutions into a **table** form and devise a rule for **filling in each cell** of the table, on the basis of cells that you know you can fill before it.

Lab3**

In the example of giving change:

entry $T[i, h]$ of table T tells how many coins are used to get quantity h with only the first i denominations, so that

Dynamic Programming, II

Tabulating partial solutions to local subproblems

Organize subproblems and partial solutions into a **table** form and devise a rule for **filling in each cell** of the table, on the basis of cells that you know you can fill before it.

Lab3**

In the example of giving change:

entry $T[i, h]$ of table T tells how many coins are used to get quantity h with only the first i denominations, so that

$$T[i, h] = \min(T[i - 1, h], 1 + T[i, h - d_i]).$$

Dynamic Programming, III

Key step: invent the right notion of subproblem

All-pairs shortest paths

Given a directed or undirected graph, find the shortest distances between all pairs of vertices:

the Floyd(-Warshall(-Roy)) algorithm.

Dynamic Programming strategy (study on your own!):

- ▶ Vertex numbered 0 to $N - 1$, subproblems defined by an initial segment of the vertices: only those are allowed as intermediate steps.
- ▶ Init: using no vertices as intermediate steps: only direct, single-edge reachability.
- ▶ Assuming tabulated all shortest distances that only use intermediate vertices less than k , how do we find out shortest distances when vertex k is allowed as well?

Dynamic Programming, III

Key step: invent the right notion of subproblem

All-pairs shortest paths

Given a directed or undirected graph, find the shortest distances between all pairs of vertices:

the Floyd(-Warshall(-Roy)) algorithm.

Dynamic Programming strategy (study on your own!):

- ▶ Vertex numbered 0 to $N - 1$, subproblems defined by an initial segment of the vertices: only those are allowed as intermediate steps.
- ▶ Init: using no vertices as intermediate steps: only direct, single-edge reachability.
- ▶ Assuming tabulated all shortest distances that only use intermediate vertices less than k , how do we find out shortest distances when vertex k is allowed as well?

$$\text{dist}(i, j, k) = \min(\text{dist}(i, j, k-1), \\ \text{dist}(i, k, k-1) + \text{dist}(k, j, k-1))$$

K-Means: Goal

Minimize the squared error

Geometry (working hypothesis):

Euclidean distance on the reals.

- ▶ Data: n real vectors x_i , positive integer k ;
- ▶ want: to split them into k clusters C_j ;
- ▶ we will pick a real vector c_j representing each cluster C_j (its centroid);
- ▶ we want to minimize the average squared error:

$$\frac{1}{n} \sum_j \sum_{x_i \in C_j} d(x_i, c_j)^2$$

Note:

We do **not** require the c_j to be among the x_i .

K-Means: Goal

Minimize the squared error

Geometry (working hypothesis):

Euclidean distance on the reals.

- ▶ Data: n real vectors x_i , positive integer k ;
- ▶ want: to split them into k clusters C_j ;
- ▶ we will pick a real vector c_j representing each cluster C_j (its centroid);
- ▶ we want to minimize the average squared error:

$$\frac{1}{n} \sum_j \sum_{x_i \in C_j} d(x_i, c_j)^2$$

Note:

We do **not** require the c_j to be among the x_i .

Bad news: Utterly infeasible at dimension 2 and beyond;
complexity theorists say: *NP-hard*.

K-Means: Partial Approach

Let's think a bit more about it

If heavens would give us the centroids:

Then, constructing the clusters is **easy**: each point to its closest centroid, as otherwise the error increases.

K-Means: Partial Approach

Let's think a bit more about it

If heavens would give us the centroids:

Then, constructing the clusters is **easy**: each point to its closest centroid, as otherwise the error increases.

If heavens would give us the clusters:

Then, finding the centroids is **easy**: minimize $\sum_{x_i \in C} d(x_i, c)^2$, by forcing the derivative to zero; and that tells us that each centroid must be set at the mass center of its cluster, as otherwise the error increases.

K-Means: HowTo

Stage-wise approximation

We alternate

among the two things we know how to do, starting from k initial centroid candidates:

- ▶ recompute the **clusters**,
- ▶ recompute the **centroids**,
- ▶ repeat.

Often advisable: try several runs!

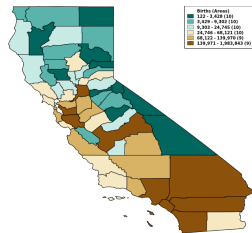
Unsupervised Discretization

Or: one-dimensional clustering

Given list of floats, organize them into just a few “bins” (or “buckets”, or “clusters”...)

Separate case of “supervised discretization”, not covered here.

Parallel research in so-called choropleth maps, a branch of Cartography, where the solution we describe here goes by the name of Jenks’ natural breaks.



(Source: Expert Health Data Programming, Inc (EHDP): Vitalnet)

One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

Tabulate: $C[i, m]$, cost of a clustering of x_1 to x_i into m clusters, for $m \leq k$ and $m \leq i$; solution given by $C[n, k]$.

One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

Tabulate: $C[i, m]$, cost of a clustering of x_1 to x_i into m clusters, for $m \leq k$ and $m \leq i$; solution given by $C[n, k]$.

Initialization: $C[i, m] = 0$ if $m = 0$.

One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

Tabulate: $C[i, m]$, cost of a clustering of x_1 to x_i into m clusters, for $m \leq k$ and $m \leq i$; solution given by $C[n, k]$.

Initialization: $C[i, m] = 0$ if $m = 0$.

Relate to “one cluster less” by identifying x_j , the smallest point in the last (m -th) cluster.

One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

Tabulate: $C[i, m]$, cost of a clustering of x_1 to x_i into m clusters, for $m \leq k$ and $m \leq i$; solution given by $C[n, k]$.

Initialization: $C[i, m] = 0$ if $m = 0$.

Relate to “one cluster less” by identifying x_j , the smallest point in the last (m -th) cluster.



One-Dimensional, Global-Optimum K-Means, I

Clustering floats with DP: the Wang and Song strategy, a.k.a. Jenks' Natural Breaks

Input: desired number of clusters k , and $n \geq k$ floats, x_1 to x_n , assumed given in increasing order (otherwise, do a sort first).

Tabulate: $C[i, m]$, cost of a clustering of x_1 to x_i into m clusters, for $m \leq k$ and $m \leq i$; solution given by $C[n, k]$.

Initialization: $C[i, m] = 0$ if $m = 0$.

Relate to “one cluster less” by identifying x_j , the smallest point in the last (m -th) cluster.



One-dimensional, Global-Optimum K-Means, II

Difference between m clusters and $m - 1$ clusters

For appropriately identified values

namely a lower limit h and a candidate to centroid of the m -th cluster $c_{j,i}$,

$$C[i, m] = \min_{h \leq j \leq i} (C[j - 1, m - 1] + \sum_{j \leq \ell \leq i} d(x_\ell, c_{j,i})^2)$$

One-dimensional, Global-Optimum K-Means, II

Difference between m clusters and $m - 1$ clusters

For appropriately identified values

namely a lower limit h and a candidate to centroid of the m -th cluster $c_{j,i}$,

$$C[i, m] = \min_{h \leq j \leq i} (C[j - 1, m - 1] + \sum_{j \leq \ell \leq i} d(x_\ell, c_{j,i})^2)$$

where

$$c_{j,i} = \frac{1}{i-j+1} \sum_{j \leq \ell \leq i} x_\ell \text{ and} \\ h = m.$$

One-dimensional, Global-Optimum K-Means, III

Demo available

For better understanding, find and run the existing R routine.

A visual demo of the process for each new point considered has been set up at:

<https://www.cs.upc.edu/~balqui/demoWSJ/>

Alpha stage!

- ▶ aesthetics fully postponed to later versions,
- ▶ usability at minimal levels. . .

Needs:

- ▶ the number of clusters,
- ▶ the points handled so far up to one specific pass,
- ▶ and the newcomer point,

Then, shows the computations made in order to account for the new point.

One-dimensional, Global-Optimum K-Means, IV

We can do it faster

Strategy leads to an $O(n^3)$ algorithm.

Acceleration: don't compute every $c_{j,i}$ individually but, instead, update $c_{j,i-1}$ to find it.

This spares a linear computation and reduces the cost to $O(n^2)$.

(Jenks' alternative: in Cartography you only need the cutpoints, not the centroids; work out an alternative formula by replacing the centroid by its definition in the minimization scheme.)

Dynamic Programming, IV

Finding the best solution cost versus finding the best solution proper

The main Dynamic Programming scheme leads often to the **cost** of the best solution, but needs a bit of extra work to find out the **best solution** itself.

- ▶ In the Giving Change program, see a (non-generalizable) hack.
- ▶ **In general**: at the time of updating a cell on the main table, record in a **second table** the key information that motivated the update.

Dynamic Programming, IV

Finding the best solution cost versus finding the best solution proper

The main Dynamic Programming scheme leads often to the **cost** of the best solution, but needs a bit of extra work to find out the **best solution** itself.

- ▶ In the Giving Change program, see a (non-generalizable) hack.
- ▶ **In general**: at the time of updating a cell on the main table, record in a **second table** the key information that motivated the update.
 - ▶ Floyd algorithm: if
$$\text{dist}(i, k, k-1) + \text{dist}(k, j, k-1) < \text{dist}(i, j, k-1),$$
then record in separate table that the best path from i to j must go through k , and use the table later (recursively!) to reconstruct paths (Wikipedia offers a different alternative).
 - ▶ In unsupervised discretization, keep recording the successive potential centroids of the successively considered clusters (see how in the Wang and Song paper).

Exercise: Solve Knapsack by Dynamic Programming.

Further Reading on Algorithmics

Some suggestions to learn about:

Several of these have been assigned to you or your colleagues, in some cases for a brief presentation in our last week.

- ▶ The “divide-and-conquer” scheme;
- ▶ graph traversals including the “best-first search” family:
 - A* (a variant of Dijkstra’s shortest paths algorithm, vaguely related also to backtracking) and evolutions;
- ▶ the AO* algorithm and alpha-beta pruning (behind many successful game-playing programs);
- ▶ branch-and-bound and branch-and-cut approaches to global optimization algorithms;
- ▶ randomized algorithms;
- ▶ linear programming, integer programming and semi-definite programming;
- ▶ approximation algorithms. . .

End of 1st part of the course