Algorithms and Data Structures

Degree in Bioinformatics, UPF

José L. Balcázar

With help from Ramon Ferrer i Cancho, Jordi Delgado, Jordi Petit, Salvador Roura, and others

Dept. CS, UPC

Winter 2023–24

We start with halfgroups

Hence, course presentation deferred to Monday.

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.
  - Example: decorators (not studied in depth, take initiatives later on).

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.
  - Example: decorators (not studied in depth, take initiatives later on).
  - Example: the `pytokr` function returns one, or two, functions that handle an iterator (iterators not studied in depth either, take initiatives later on).

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.
  - Example: decorators (not studied in depth, take initiatives later on).
  - Example: the `pytokr` function returns one, or two, functions that handle an iterator (iterators not studied in depth either, take initiatives later on).
- And we start programming.

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.
  - Example: decorators (not studied in depth, take initiatives later on).
  - Example: the `pytokr` function returns one, or two, functions that handle an iterator (iterators not studied in depth either, take initiatives later on).
- And we start programming.
- There is a key programming notion behind every course on Algorithms and Data Structures:

Today

- Jutge courses: main one is Algorithms and Data Structures Winter 2023-24 ESCI.
- Most recent version of the `pytokr` package, version 0.1, which requires some discussion of Higher Order programming:
  - We say higher order functions to refer to functions that have, either as parameters, results, or both, other functions.
  - Example: the `key` parameter for `sorted`.
  - Example: decorators (not studied in depth, take initiatives later on).
  - Example: the `pytokr` function returns one, or two, functions that handle an iterator (iterators not studied in depth either, take initiatives later on).
- And we start programming.
- There is a key programming notion behind every course on Algorithms and Data Structures:

recursion!


SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING RECURSION
SAFELY ENDANGERED
SWEET JESUS, POOH! THAT'S NOT HONEY
YOU'RE EATING RECURSION

Recursion, I

A must

Half or more of every course like this one is based on recursive programs and recursive structuring of data.

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

How to think and conceive recursive structures

Recall the induction principle.

Recursion, II

How to think and conceive recursive structures

Recall the induction principle.

You must

- Reach an extremely clear notion of what the program achieves,
- write it down in as much formality as you are capable to,
- and solve the problem while relying on that notion to reason inductively about the recursive calls.

Never let your thinking slip down through what happens at the recursive calls themselves.

The `Python tutor` may be helpful to reach further understanding.

Recursion, III

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...

Today: We refresh and introduce a few notions of graphs and trees, and practice recursive programming on trees.

`https://en.wikipedia.org/wiki/Gallery_of_named_graphs`

Trees as Graphs

A couple of important adjectives

Easy to see a tree as a graph

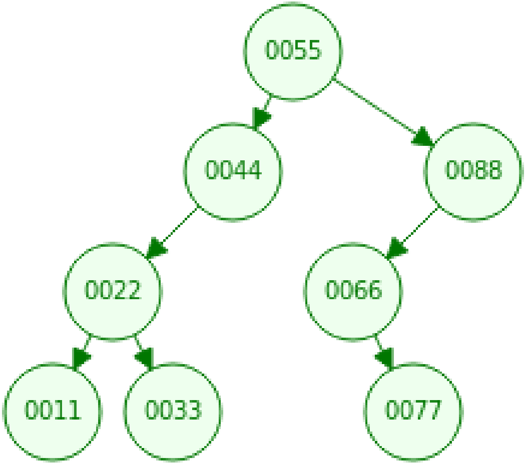
but one needs to pay a bit of attention to a couple of things.

- 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



Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. “recorregut”, cast. “recorrido”): passing through all the elements in a data structure.

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. "recorregut", cast. "recorrido"): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. "recorregut", cast. "recorrido"): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;
- postorder: similar, but leave root for the end;

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. "recorregut", cast. "recorrido"): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;
- postorder: similar, but leave root for the end;
- inorder: similar, but traverse root between subtrees;

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. “recorregut”, cast. “recorrido”): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;
- postorder: similar, but leave root for the end;
- inorder: similar, but traverse root between subtrees;
- levelwise (we will return to this in due time).

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. “recorregut”, cast. “recorrido”): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;
- postorder: similar, but leave root for the end;
- inorder: similar, but traverse root between subtrees;
- levelwise (we will return to this in due time).

Tree Traversals

Example traversals on the example tree

Traversal (a.k.a. Full scan, cat. “recorregut”, cast. “recorrido”): passing through all the elements in a data structure.

On binary trees

Several traversal schemes possible:

- Preorder: traverse root, then (recursively) the left subtree, then (recursively) the right subtree;
- postorder: similar, but leave root for the end;
- inorder: similar, but traverse root between subtrees;
- levelwise (we will return to this in due time).

On multiway trees

Preorder and postorder have natural generalizations.

Logistics, I

- José Luis Balcázar

  `jose.luis.balcazar@upc.edu`
  Please do not employ `jose.balcazar@prof.esci.upf.edu`
- Meeting according to the general schedule:

  Theory and (*compulsory!*) Lab sessions.
- Additional personal conversations as needed:
  - Normally, I can be available around our room before and after the theory session.
  - Alternative slots for longer appointments, by email if necessary.
- Slides available from
  `https://www.cs.upc.edu/~balqui/slides_ADS_24.pdf`
  (link also available through `http://aula.esci.upf.edu`).
- The slide deck is "live"; update often your copy!
- Mandatory Jutge course (for exams) and optional additional ones.

Course Contents

- Recalling **recursion** and related topics.
- Combinatorial Search Schemes:
  - Backtracking algorithms,
  - **Dynamic Programming**.
- Linear data structures.
- Tree traversals and graph traversals revisited.
- Fundamental notions of **heaps**, balanced trees, and hashing.
- Dynamic memory: basics of pointer programming.
- Optional: `C++` programming — mostly on your own.
- Optional: `STL`, the standard template library of `C++` — mostly on your own.

Evaluation, I

Part individual, part in small teams

Deliverable: Documented programming project in pairs, on assigned topics.

Lab tasks: individual, unless negotiated differently with me.

All exams: individual.

Evaluation, II

Deliverable: 25% of the grade.

Lab tasks: 25% of the grade, taken jointly.

Midterm: 25% of the grade, Monday February 12th.

Final exam: 25% of the grade; as of today, March 18th 15h seems likely.

Recovery: date and time TBD.

Evaluation, III

Topics for programming projects

Topic is rather free but the project must include algorithmic or data structures content beyond what is covered in the course, such as:

- Applications of alternative or more sophisticate algorithm schemes.
- Pointer-based list and tree implementations in C++.
- Balanced search trees: AVL BSTs, B+ trees.
- Hashing.
- Advanced usage of graph libraries.
- Graph drawing software.
- ...

Index

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, quite closely, and many more do if we relax the interpretations a bit.

Spanning Trees, I

Review and algorithmic ideas

Also called "minimal connectors"

Given a connected graph with edge weights, find a subgraph that is still connected and has weight as small as possible. (Variants. . .)

Spanning Trees, I

Also called "minimal connectors"

Given a connected graph with edge weights, find a subgraph that is still connected and has weight as small as possible. (Variants. . .)

Properties:

1. The answer is always a (free, unordered) tree. Why?
2. The edge of least cost, if unique, must belong to the answer.

Spanning Trees, I

Review and algorithmic ideas

Also called "minimal connectors"

Given a connected graph with edge weights, find a subgraph that is still connected and has weight as small as possible. (Variants. . .)

Properties:

1. The answer is always a (free, unordered) tree. Why?
2. The edge of least cost, if unique, must belong to the answer.

Algorithmic approaches:

Kruskal: keep adding the lowest-cost edge, unless it creates a cycle, in which case it can be discarded.

Prim: keep expanding the current tree by the lowest-cost edge that keeps it a tree; if it does not, can be discarded.

Examples of the Framework, I

Or: Spanning Trees, II

Two examples on spanning trees:

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

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

Examples of the Framework, I

Or: Spanning Trees, II

Two examples on spanning trees:

Given a connected graph with edge weights, find 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 the Framework, I

Or: Spanning Trees, II

Two examples on spanning trees:

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

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

(b) that connects all the vertices with the minimum total weight.

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 the Framework, I

Or: Spanning Trees, II

Two examples on spanning trees:

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

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

(b) that connects all the vertices with the minimum total weight.

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 the Framework, II

Or: Knapsack, I

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

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 the Framework, III

Or: Knapsack, II

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”.

We study optimization cases later, and decisional versions first.

Knapsack, III

Decisional version

Given:

- objects $i \in \{0, \ldots, N-1\}$
- with weights $w[i]$ and values $v[i]$,
- maximum capacity of knapsack W,
- desired total value V:

find a set of objects to take for the knapsack:

- total weight does not exceed the maximum capacity,
- total value is at least the desired total value.

Example:

Maximum weight $W = 26$, desired value $V = 45$ with objects of:

Weight:	9	8	12	11	7
Value:	16	15	24	23	13

Knapsack, IV

Why not, simply, scan the whole powerset? “Brute force” (perebor)

```
def slow_knapsack(objects, W, V):
    for candidate in powerset(objects):
        if (totalweight(candidate) <= W
            and totalvalue(candidate) >= V):
            return candidate
```

Knapsack, IV

Why not, simply, scan the whole powerset? “Brute force” (perebor)

```
def slow_knapsack(objects, W, V):
    for candidate in powerset(objects):
        if (totalweight(candidate) <= W
            and totalvalue(candidate) >= V):
            return candidate
```

You have a choice of options for the `powerset` iterable:

- Start by trying your hand without any other inspiration: P18957.
- Maybe you saw it as a basic recursion example of previous quarter.
- Recipe in the Python documentation, chapter on itertools, section “itertools recipes” (provides subsets ordered by size).
- My favorite: a recursive generator (learn on yourself about generators and come up with it, ask me if necessary).

Knapsack, IV

Why not, simply, scan the whole powerset? “Brute force” (perebor)

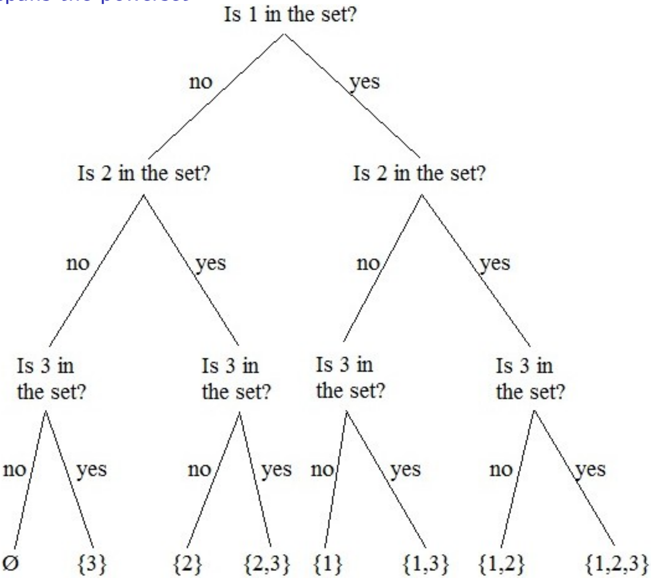
```
def slow_knapsack(objects, W, V):
    for candidate in powerset(objects):
        if (totalweight(candidate) <= W
            and totalvalue(candidate) >= V):
            return candidate
```

You have a choice of options for the `powerset` iterable:

- Start by trying your hand without any other inspiration: P18957.
- Maybe you saw it as a basic recursion example of previous quarter.
- Recipe in the Python documentation, chapter on itertools, section “itertools recipes” (provides subsets ordered by size).
- My favorite: a recursive generator (learn on yourself about generators and come up with it, ask me if necessary).
- Too slow for all practical purposes.

Knapsack, V

Implicit tree that spans the powerset



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

Knapsack, VI

Tree-based traversal of the powerset, also “brute force”

```
def knapsack(weights, values, current_item,
        max_w, min_v, cand, ...):
  if current_item == -1:
    if ("candidate value" >= min_v and
        "candidate weight" <= max_w): return cand
    else: return list()
  else:
    "current_item >= 0"
    sol = knapsack(weights, values, current_item - 1,
              max_w, min_v, cand, ...)
    if sol: return sol
    else:
      return knapsack(weights, values, current_item - 1,
                 max_w, min_v,
                 cand + [ current_item ], ...)
```

Knapsack, VII

Tree-based traversal of the powerset, also "brute force" – extra details

```
def knapsack(weights, values, current_item,
      max_w, min_v, cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return cand
    else: return list()
  else:
    sol = knapsack(weights, values, current_item - 1,
            max_w, min_v, cand, cand_w, cand_v)
    if sol: return sol
    else:
      return knapsack( weights, values, current_item - 1,
                max_w, min_v,
                cand + [ current_item ],
                cand_w + weights[current_item],
                cand_v + values[current_item] )
```

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 still need a bit more conceptualization

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 refered to as "global");

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered to as "global");
- a function that tells us whether a sequence of decisions is

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered to as “global”);
- a function that tells us whether a sequence of decisions is
  (a) already “unacceptable”

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered 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 still need a bit more conceptualization

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 refered 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" so far but still "incomplete"

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered 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" so far but still "incomplete"
  (the subproblem might be solvable, we must go on) or

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered 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" so far but still "incomplete"
  (the subproblem might be solvable, we must go on) or
  (c) a "complete" candidate solution

Framework for Algorithmic Schemes, II

We still need a bit more conceptualization

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 refered 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" so far but still "incomplete"
  (the subproblem might be solvable, we must go on) or
  (c) a "complete" candidate solution
  (that is, a solved subproblem that actually provides a solution to the original instance).

Examples of the Framework, IV

Or: Spanning Trees, III

Spanning tree (with or without connectivity?)

Subproblem:

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

Examples of the Framework, IV

Or: Spanning Trees, III

Spanning tree (with or without connectivity?)

Subproblem:

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

Examples of the Framework, IV

Or: Spanning Trees, III

Spanning tree (with or without connectivity?)

Subproblem:

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

Examples of the Framework, V

Or: Knapsack, VIII

Knapsack:

Objects with weights and values.

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

Examples of the Framework, V

Or: Knapsack, VIII

Knapsack:

Objects with weights and values.

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.

Often there is more than one way to set up the scheme. This is but one.

Backtracking, I

Traversal of the implicit tree

What is it?

Related to tree preorder traversal, except that the tree remains implicit.

- Imagine each subproblem as a vertex in a (very large) tree.
- Edges correspond to decisions that change a subproblem into another one.

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 (dead ends).

Backtracking, II

The optimization case will come later

Three options:

1. Search-like scheme: stop the exploration if we are satisfied with only one solution.
2. Full traversal: we want all solutions and must complete the traversal.
3. Optimization: want to see all solutions to pick the "best" one according to some criterion.

Implicit tree:

- Binary tree in set-based backtracking where we construct a set through binary membership decisions.
- General tree when decisions are not binary (to be studied soon too).

Knapsack, IX

Set-based backtracking

Subproblem: What do we consider as a subproblem?
Knapsack restricted to the first k objects.

Infeasibility: When do we backtrack from a subproblem?
Insufficient value? Adding objects might solve it.
Excess weight?
Adding objects will not make it feasible!

Alternative: Work it out "the other way around"!
Start by taking everything,
keep discarding objects until fitting the weight,
declare infeasibility when value drops below threshold.

Knapsack, X

```
def knapsack(weights, values, current_item, max_w, min_v,
      cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v:
      return cand
    else:
      return list()
  else:
    "current_item >= 0"
    sol = knapsack(weights, values, current_item - 1,
            max_w, min_v, cand, cand_w, cand_v)
    if not sol and weights[current_item] <= max_w:
      sol = knapsack(weights, values, current_item - 1,
              max_w - weights[current_item], min_v,
              cand + [ current_item ],
              cand_w + weights[current_item],
              cand_v + values[current_item])
    return sol
```

Knapsack, XI (closer to the tree-based exploration)

```
def knapsack(weights, values, current_item, max_w, min_v,
      cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return cand
    else:
      return list()
  else:
    "current_item >= 0"
    sol = knapsack(weights, values, current_item - 1,
            max_w, min_v, cand, cand_w, cand_v)
    if not sol and weights[current_item] <= max_w:
      sol = knapsack(weights, values, current_item - 1,
              max_w, min_v,
              cand + [ current_item ],
              cand_w + weights[current_item],
              cand_v + values[current_item])
    return sol
```

Knapsack, XII

Avoid making copies! Example on "all solutions"

```
def knapsack(weights, values, current_item, max_w, min_v,
      cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand ]
    ...
  else:
    ...
    if weights[current_item] <= max_w:
      cand.append(current_item)
      ... knapsack(weights, values, current_item - 1,
              max_w, min_v,
              cand,
              cand_w + weights[current_item],
              cand_v + values[current_item])
      ...
    return sol
```

Knapsack, XII

Avoid making copies! Example on “all solutions”

```
def knapsack(weights, values, current_item, max_w, min_v,
      cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand ]
    ...
  else:
    ...
    if weights[current_item] <= max_w:
      cand.append(current_item)
      ... knapsack(weights, values, current_item - 1,
              max_w, min_v,
              cand,
              cand_w + weights[current_item],
              cand_v + values[current_item])
      cand.pop() # backtrack!
    return sol
```

Knapsack, XII

Avoid making copies! Example on "all solutions"

```
def knapsack(weights, values, current_item, max_w, min_v,
        cand, cand_w, cand_v):
  if current_item == -1:
    if cand_v >= min_v and cand_w <= max_w:
      return [ cand.copy() ]
    ...
  else:
    ...
    if weights[current_item] <= max_w:
      cand.append(current_item)
      ... knapsack(weights, values, current_item - 1,
              max_w, min_v,
              cand,
              cand_w + weights[current_item],
              cand_v + values[current_item])
      cand.pop() # backtrack!
    return sol
```

Knapsack, XIII

Yet another option

Instead of carrying around the candidates as we proceed down the tree. . .

Can we just obtain the existing solutions below each successor and combine them?

Knapsack, XIII

Yet another option

Instead of carrying around the candidates as we proceed down the tree. . .

Can we just obtain the existing solutions below each successor and combine them?

Yes, of course. The program is even a bit faster but (in my humble opinion) harder to understand.

We will apply that strategy later on today for optimization.

Graph Colorability

Beyond the binary decisions of set-based backtracking

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`

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`

We focus on edge coloring today, see `Wikipedia link`.

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different at every vertex.

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.
- Decisions: the same graph with one more edge colored.

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.
- Decisions: the same graph with one more edge colored.

  But... which one?

Example: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.
- Decisions: 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: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.
- Decisions: 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: Edge Colorability, I

The implicit tree: graph with more and more colored edges

Given a graph G and a natural number k of different colors, assign colors to the edges so that all colors are different 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.
- Subproblems: the same graph with part of the edges already colored.
- Dead ends: the coloring is already incompatible.
- Decisions: the same graph with one more edge colored.

  But... which one?

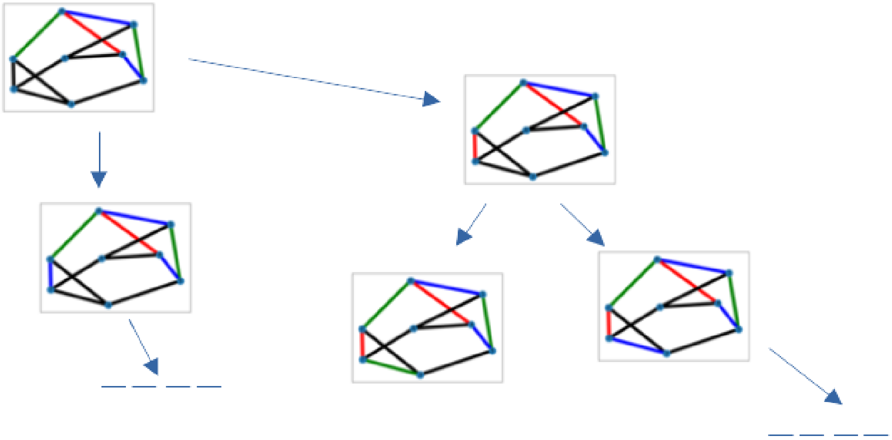
  To keep the implicit graph a tree, we must be a bit careful.

Examples and demos:

mostly on 3-regular graphs (all vertices of degree 3) and $k = 3$.

Example: Edge Colorability, II

The implicit tree (a fragment)



Example: Edge Colorability, III

Finding one solution

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.

Example: Edge Colorability, III

Finding one solution

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, stop 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: Edge Colorability, IV

Finding one solution

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

Example: Edge Colorability, V

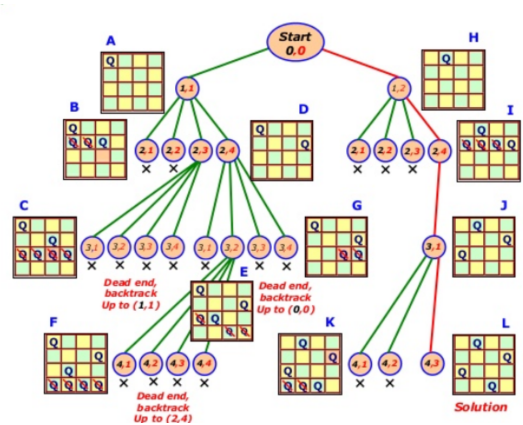
Can we do better?

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.
- . . .

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

Finding all solutions

```
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, III

Finding one solution

```
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, III

Finding one solution

```
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, IV

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").
- ...

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).

Objective function on spanning trees:

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

Objective function on graph coloring:

- Color the edges of the given graph with as few colors as possible.

Knapsack, XIV

Towards backtracking for optimization

Given:

- objects $i \in \{0, \ldots, N-1\}$
- 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.

Examples of the Framework, VI

Or: Knapsack, XV

Alternative:

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

Objective function:

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

Examples of the Framework, VI

Or: Knapsack, XV

Alternative:

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

Objective function:

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

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

(Complete the scheme on your own.)

Knapsack, XVI

Scan the whole powerset again?

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

Too slow for all practical purposes.

Knapsack, XVII

Backtracking for optimization

```
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 or weight available, emptyset 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, XVIII

If we want to go beyond, it may be worthwhile to embrace OO

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?

Knapsack, XVIII

If we want to go beyond, it may be worthwhile to embrace OO

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?
- . . .

Knapsack, XVIII

If we want to go beyond, it may be worthwhile to embrace OO

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.

Factorial Function Growth, I

Exhaustive search ("try all possible solutions") is unlikely to be acceptable

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")...

Factorial Function Growth, I

Exhaustive search ("try all possible solutions") is unlikely to be acceptable

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?

Factorial Function Growth, I

Exhaustive search ("try all possible solutions") is unlikely to be acceptable

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 Growth, I

Exhaustive search ("try all possible solutions") is unlikely to be acceptable

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?

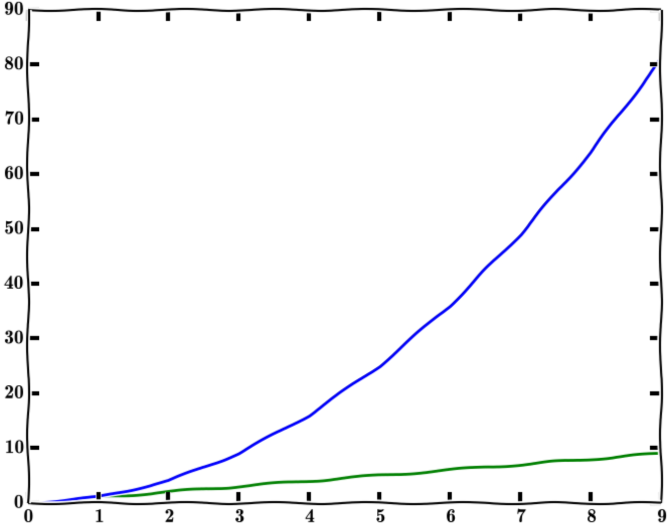
Shall we explore all possibilities?

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

Risk of combinatorial explosion, see `Wikipedia link`.

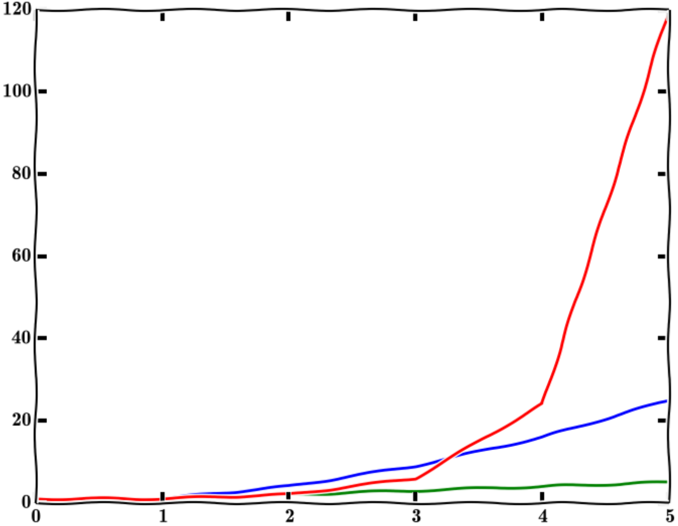
Factorial Function Growth, II

Thou shalt not take the name of the Exponential in vain



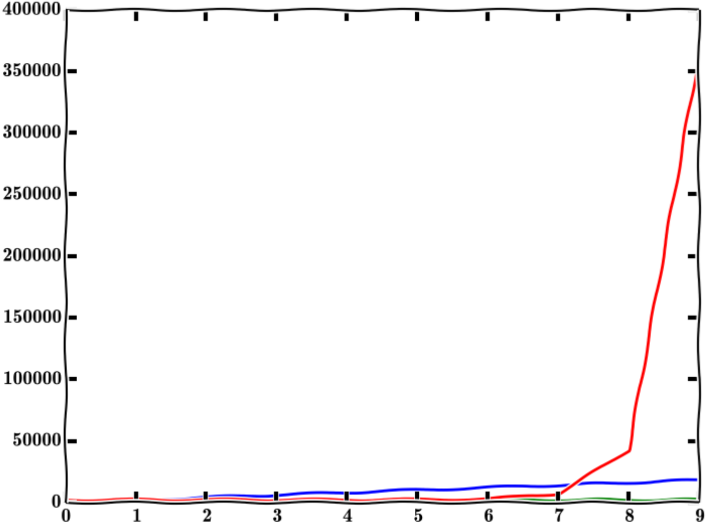
Factorial Function Growth, II

Thou shalt not take the name of the Exponential in vain



Factorial Function Growth, II

Thou shalt not take the name of the Exponential in vain



Factorial Function Growth, III

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});

Factorial Function Growth, III

$N!$ 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 Growth, III

$N!$ 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 Growth, III

$N!$ 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 Growth, III

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 Growth, III

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. . .

Factorial Function Growth, IV

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!, see `Wikipedia link`.)

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).

Dynamic Programming, I

Recommended reading: `nice account` of the origins by Richard Bellman himself

For a given combinatorial search problem, is the following true?

Bellman's Principle of Optimality:

> the part of the optimal global solution that corresponds to any subproblem is itself a locally optimal solution.

Dynamic Programming, I

Recommended reading: `nice account` of the origins by Richard Bellman himself

For a given combinatorial search problem, is the following true?

Bellman's Principle of Optimality:

> the part of the optimal global solution that corresponds to any subproblem is itself a locally optimal solution.

This condition does not hold always.

Dynamic Programming, I

Recommended reading: `nice account` of the origins by Richard Bellman himself

For a given combinatorial search problem, is the following true?

Bellman's Principle of Optimality:

> the part of the optimal global solution that corresponds to any subproblem is itself a locally optimal solution.

This condition does not hold always.

But... does it hold often enough!

Dynamic Programming, I

Recommended reading: `nice account` of the origins by Richard Bellman himself

For a given combinatorial search problem, is the following true?

Bellman's Principle of Optimality:

> the part of the optimal global solution that corresponds to any subproblem is itself a locally optimal solution.

This condition does not hold always.

But... does it hold often enough!

Rather good and instructive article on Wikipedia:
`https://en.wikipedia.org/wiki/Dynamic_programming`

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.

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.

- Once the table is conceived, be careful in implementing it efficiently.
- For instance, you may need only one row in order to construct the next, or. . .
- Dynamic Programming is particularly efficient when many repeated subproblems keep appearing while solving the original problem.
- May look like inefficient. . . until one finds out how many (possibly repeated) subproblems are solved by a backtracking scheme.

"The gold ones are Galleons. Seventeen silver Sickles to a Galleon and twenty-nine Knuts to a Sickle, it's easy enough."

J. K. Rowling
Harry Potter and the Philosopher's Stone
(1997)

Giving change, I

https://jutge.org/problems/P81009

Given goal and available coin denominations:

- goal quantity M to reach by adding up coins
- of denominations $d_1, \ldots, d_n$.

Giving change, I

https://jutge.org/problems/P81009

Given goal and available coin denominations:

- goal quantity M to reach by adding up coins
- of denominations $d_1, \ldots, d_n$.

Related both to Knapsack and to Subset Sum.

- Similar to Subset Sum more than to Knapsack: the goal is to be reached exactly.
- Difference with Subset Sum: "unboundedly available" coins/objects, we can take several of each, repeated as necessary.

Giving change, I

https://jutge.org/problems/P81009

Given goal and available coin denominations:

- goal quantity M to reach by adding up coins
- of denominations $d_1, \ldots, d_n$.

Related both to Knapsack and to Subset Sum.

- Similar to Subset Sum more than to Knapsack: the goal is to be reached exactly.
- Difference with Subset Sum: "unboundedly available" coins/objects, we can take several of each, repeated as necessary.

(Some coin denominations allow for substantially faster solutions. Back to that after the midterm.)

Giving change, II

Dynamic Programming: tabulate partial solutions

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…

Giving change, II

Dynamic Programming: tabulate partial solutions

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...

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

Giving change, II

Dynamic Programming: tabulate partial solutions

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...

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

...provided that...

Giving change, II

Dynamic Programming: tabulate partial solutions

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...

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

...provided that...

$$d_i \leq h \text{ (o/w?)}$$

Giving change, II

Dynamic Programming: tabulate partial solutions

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...

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

... provided that...

$$d_i \leq h \text{ (o/w?)}$$

The Principle of Optimality holds:

A fragment of an optimal solution is itself an optimal solution.

Giving change, III

"The details are the jungle in which the devil hides." [N. Wirth]

- Do we need an indexing of the denominations?

  Consider using only positions 1 on of a list, `denoms`, for $d_1, \ldots, d_n$, avoiding d_0.

  "Placeholder" value in `denoms[0]`.
- (We simplify that indexing in a more efficient subsequent version.)
- Precise meaning of each of the rows?
- $T[0, h]$? Particularly $T[0, 0]\ldots$

Giving change, III

"The details are the jungle in which the devil hides." [N. Wirth]

- Do we need an indexing of the denominations?

  Consider using only positions 1 on of a list, `denoms`, for $d_1, \ldots, d_n$, avoiding d_0.

  "Placeholder" value in `denoms[0]`.
- (We simplify that indexing in a more efficient subsequent version.)
- Precise meaning of each of the rows?
- $T[0, h]$? Particularly $T[0, 0]\ldots$

  $$T[0, 0] = 0$$

Giving change, III

"The details are the jungle in which the devil hides." [N. Wirth]

- Do we need an indexing of the denominations?

  Consider using only positions 1 on of a list, `denoms`, for $d_1, \ldots, d_n$, avoiding d_0.

  "Placeholder" value in `denoms[0]`.
- (We simplify that indexing in a more efficient subsequent version.)
- Precise meaning of each of the rows?
- $T[0, h]$? Particularly $T[0, 0] \ldots$

  $T[0, 0] = 0$
- $T[0, h]$ for $h > 0$ must say somehow: "impossible".

Giving change, III

"The details are the jungle in which the devil hides." [N. Wirth]

- Do we need an indexing of the denominations?

  Consider using only positions 1 on of a list, `denoms`, for $d_1, \ldots, d_n$, avoiding d_0.

  "Placeholder" value in `denoms[0]`.
- (We simplify that indexing in a more efficient subsequent version.)
- Precise meaning of each of the rows?
- $T[0, h]$? Particularly $T[0, 0]\ldots$

  $T[0, 0] = 0$
- $T[0, h]$ for $h > 0$ must say somehow: "impossible".

  $T[0, h] =$ `float("inf")` — why?

Giving change, IV

```
def minchange(goal, denoms):
 dptable = {}
 for quantity in range(goal + 1):
  dptable[0, quantity] = float("inf")
 for coin in range(len(denoms)):
  dptable[coin, 0] = 0
 for quantity in range(1, goal + 1):
  for coin in range(1, len(denoms)):
   if denoms[coin] <= quantity:
    dptable[coin, quantity] = min(
        dptable[coin - 1, quantity],
        1 + dptable[coin, quantity - denoms[coin]])
   else:
    dptable[coin, quantity] = dptable[coin - 1, quantity]
 return (-1 if dptable[len(denoms) - 1, goal] > goal
        else dptable[len(denoms) - 1, goal])
```

Dynamic Programming, III

But. . . we want to know how to achieve the quantity!

In the table of this specific problem we can read whether each denomination was actually employed: keep testing whether

```
dptable[coin, quantity] != dptable[coin - 1, quantity]
```

and keep updating downwards `quantity` accordingly and collecting the used coins.

Dynamic Programming, III

But... we want to know how to achieve the quantity!

In the table of this specific problem we can read whether each denomination was actually employed: keep testing whether

```
dptable[coin, quantity] != dptable[coin - 1, quantity]
```

and keep updating downwards `quantity` accordingly and collecting the used coins.

In general: The basic Dynamic Programming scheme leads often only to the cost of the best solution, and needs a bit of extra work to find out the best solution itself.

Namely: at the time of updating a cell on the main table, record in a secondary table the key information that motivated the update.

Giving change, V

```
def minchange(goal, denoms):
  dptable = {}
  best = {}
  ...
    if (denoms[coin] <= quantity and
          1 + dptable[coin, quantity - denoms[coin]] <
              dptable[coin - 1, quantity]):
      dptable[coin, quantity] = (1 +
                     dptable[coin, quantity - denoms[coin]])
      best[quantity] = denoms[coin]
  ...
```

Giving change, V

```
def minchange(goal, denoms):
  dptable = {}
  best = {}
  ...
    if (denoms[coin] <= quantity and
        1 + dptable[coin, quantity - denoms[coin]] <
            dptable[coin - 1, quantity]):
      dptable[coin, quantity] = (1 +
                  dptable[coin, quantity - denoms[coin]])
      best[quantity] = denoms[coin]
  ...

def trace(best, goal):
  coins = []
  while goal:
    use = best[goal]
    coins.append(use)
    goal -= use
  return coins
```

Giving change, VI

Do we need the whole table?

Once we have that solution in place:

- Can we simplify the data structures?
- Often, simplifying the data structures helps simplifying the code.

We maintain only one row of the table and initialize it as if with a coin of “denomination zero”.

Giving change, VII

```
dptable = [ float("inf") ] * (goal + 1)
dptable[0] = 0
for quant in range(1, goal + 1):
  "dptable[quant]: how many coins needed to add up to it"
  for coin in denoms:
    "try using it - actual value, not just an index"
    if coin <= quant:
      dptable[quant] = min(
          dptable[quant],
          1 + dptable[quant - coin])
```

At the end, `dptable[goal] > goal` means again that it is not possible.

Shortest Paths, I

Are there costs? Can they be negative?

Shortest path problems in graphs

They are extremely common: many practical problems boil down to finding shortest paths.

We consider first the "single-source" case: find paths that start at a given vertex.

- Do edges have various associated costs?
- Are there edges with a negative cost?
- If yes... is there a cycle of total negative weight?
- Then the problem is pretty complicated and only recently solved.

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:
  the answer is breadth-first search.

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:
  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

  start from Dijkstra's Algorithm and then consider adding sophistications like A* and beyond (more on this after the midterm).

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

  start from Dijkstra's Algorithm and then consider adding sophistications like A* and beyond (more on this after the midterm).
- So, assume that there are weights, some can be negative, but there is no cycle of total negative weight.

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

  start from Dijkstra's Algorithm and then consider adding sophistications like A* and beyond (more on this after the midterm).
- So, assume that there are weights, some can be negative, but there is no cycle of total negative weight.
- Observe that this only makes sense for directed graphs.

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

  start from Dijkstra's Algorithm and then consider adding sophistications like A* and beyond (more on this after the midterm).
- So, assume that there are weights, some can be negative, but there is no cycle of total negative weight.
- Observe that this only makes sense for directed graphs.
- And observe that... a fragment of a shortest path is a shortest path!,

Shortest Paths, II

Algorithmic options

- When the cost of traversing edges is constant:

  the answer is breadth-first search.
- If edge costs are not constant, but are non-negative:

  start from Dijkstra's Algorithm and then consider adding sophistications like A* and beyond (more on this after the midterm).
- So, assume that there are weights, some can be negative, but there is no cycle of total negative weight.
- Observe that this only makes sense for directed graphs.
- And observe that... a fragment of a shortest path is a shortest path!,
- even in the presence of negative weights!

Shortest Paths, III

Dynamic Programming to the rescue

Let u be the fixed source vertex: tabulate the distance to vertex v in a maximum of i steps.

The solution is when i is at most 1 less than the number of vertices.

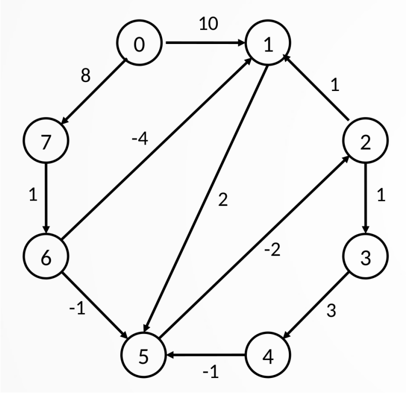


Image source: Jordi Delgado, UPC

Shortest Paths, IV

Identifying a notion of "subproblem"

Graph of n vertices; source is vertex s; `dist[v, i]` is the distance from s to v in at most i steps.

If `dist[v, i]` is optimal and the last edge is (u, v), then `dist[u, i-1]` must be optimal.

Shortest Paths, IV

Identifying a notion of "subproblem"

Graph of n vertices; source is vertex s; `dist[v, i]` is the distance from s to v in at most i steps.

If `dist[v, i]` is optimal and the last edge is (u, v), then `dist[u, i-1]` must be optimal.

```
for all v in V:
    dist[v, 0] = 'infinity' # discussed in a minute
dist[s, 0] = 0

for i in range(1, n):
    for all the edges (u, v):
        dist[v, i] = min(
            dist[v, i-1],
            dist[u, i-1] + cost[u, v] )
```

Shortest Paths, V

Bellman-Ford algorithm

```
for all v in V:
    dist[v] = float('inf') # or any other 'big' value


dist[s] = 0

for i in range(1, n):
    for all the edges (u, v):
        if dist[v] > dist[u] + cost[u,v]:
            dist[v] = dist[u] + cost[u, v]
```

Shortest Paths, VI

How to reconstruct the shortest paths?

```
for all v in V:
    dist[v] = float('inf') # or any other 'big' value
    prev[v] = None

dist[s] = 0

for i in range(1, n):
    for all the edges (u, v):
        if dist[v] > dist[u] + cost[u,v]:
            dist[v] = dist[u] + cost[u, v]
            prev[v] = u
```

Shortest Paths, VII

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:

- Subproblems defined by: up to which vertices are allowed as intermediate steps?
- Initialiation means 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?

Shortest Paths, VII

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:

- Subproblems defined by: up to which vertices are allowed as intermediate steps?
- Initialiation means 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?

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

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

What would be appropriate table dimensions?

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

What would be appropriate table dimensions?

Then, how does the corresponding Bellman equation look like?

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

What would be appropriate table dimensions?

Then, how does the corresponding Bellman equation look like?

$$T[i,h] = \min(\ T[i-1,h],\ v_i + T[i-1,h-w_i]\ )$$

... provided that...

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

What would be appropriate table dimensions?

Then, how does the corresponding Bellman equation look like?

$$T[i, h] = \min(\ T[i-1, h],\ v_i + T[i-1, h-w_i]\ )$$

. . . provided that. . .

$$w_i \leq h$$

Knapsack, XIX

Just a couple of words

How do we approach Knapsack by Dynamic Programming?

What would be appropriate table dimensions?

Then, how does the corresponding Bellman equation look like?

$$T[i, h] = \min(\ T[i-1, h],\ v_i + T[i-1, h-w_i]\ )$$

. . . provided that. . .

$$w_i \leq h$$

Complete it by thinking carefully about the boundary conditions.

Knapsack, XX

Implementation options

- Indexing: start at zero? reserve zero for bookkeeping? ...
- "Spelled out" 2-dimensional table?
  - Easier to connect visually the code with the Bellman equation.
  - Simpler reconstruction of the solution:
    - 2-dimensional table storing reasons for changes;
    - simple alternative (shared by some similar problems): test `dptab[obj, weight] != dptab[obj - 1, weight]` to see whether `obj` is used for `weight`.

Knapsack, XX

Implementation options

- Indexing: start at zero? reserve zero for bookkeeping? ...
- "Spelled out" 2-dimensional table?
  - Easier to connect visually the code with the Bellman equation.
  - Simpler reconstruction of the solution:
    - 2-dimensional table storing reasons for changes;
    - simple alternative (shared by some similar problems): test `dptab[obj, weight] != dptab[obj - 1, weight]` to see whether `obj` is used for `weight`.
- Flatten it instead into a 1-dimensional list?
  - More efficient in memory and (slightly) in time.
  - Crisper code, less error-prone.
  - Still two dimensions in the secondary table.

Supersequences, I

Connection to Bioinformatics not that far away

Given two strings, find (one of) the shortest string(s) that is a supersequence of both.

(Invent your examples with the letters `A`, `T`, `G`, `C`.)

Supersequences, I

Connection to Bioinformatics not that far away

Given two strings, find (one of) the shortest string(s) that is a supersequence of both.

(Invent your examples with the letters `A`, `T`, `G`, `C`.)

Again: we start by just computing how long it is, then add later code to trace an actual solution.

Supersequences, I

Connection to Bioinformatics not that far away

Given two strings, find (one of) the shortest string(s) that is a supersequence of both.

(Invent your examples with the letters `A`, `T`, `G`, `C`.)

Again: we start by just computing how long it is, then add later code to trace an actual solution.

If `r` is one shortest common supersequence of `s` and `t`, what can we find out about it?

Supersequences, II

Some considerations

1. Case of empty input sequences?

Supersequences, II

Some considerations

1. Case of empty input sequences?
2. Case that both input sequences start with the same letter?
   - Outcome too.
   - Rest of result is. . .

Supersequences, II

Some considerations

1. Case of empty input sequences?
2. Case that both input sequences start with the same letter?
   - Outcome too.
   - Rest of result is... a subproblem with two shorter strings.
3. Case that the initial letters of the input sequences are different?
   - Outcome must start with one of them.
   - Rest of result is...

Supersequences, II

Some considerations

1. Case of empty input sequences?
2. Case that both input sequences start with the same letter?
   - Outcome too.
   - Rest of result is... a subproblem with two shorter strings.
3. Case that the initial letters of the input sequences are different?
   - Outcome must start with one of them.
   - Rest of result is... a subproblem with one string kept and the other shorter.

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

If `s[i] == t[j]`:

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

If `s[i] == t[j]`: `1 + S[i+1, j+1]`.

If `s[i] != t[j]`:

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

If `s[i] == t[j]`: `1 + S[i+1, j+1]`.

If `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

If `s[i] == t[j]`: `1 + S[i+1, j+1]`.

If `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Boundary conditions: when `s[i:]` and/or `t[j:]` are empty.

Supersequences, III

How do we set up our table

`S[i, j]`: length of a shortest supersequence of `s[i:]` and `t[j:]`.

If `s[i] == t[j]`: `1 + S[i+1, j+1]`.

If `s[i] != t[j]`: `1 + min(S[i+1, j], S[i, j+1])`.

Boundary conditions: when `s[i:]` and/or `t[j:]` are empty.

Careful! This time, entries in the table depend on other entries with larger indices!

Supersequences, IV

Hint at code

```
initialize
for i in reversed(range(len(s))):
    for j in reversed(range(len(t))):
        if s[i] == t[j]:
            S[i, j] = 1 + S[i+1, j+1]
        else:
            S[i, j] = 1 + min(S[i+1, j], S[i, j+1])
return S[0,0]
```

Supersequences, V

Tracing back the solution

Usual method: secondary table that says, for each `i` and `j`, what letter do we take for the result.

If `s[i] == t[j]`, the same letter.

If `s[i] != t[j]`, implement an inequality test instead of the `min` operation and store the letter that corresponds to the better value.

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!

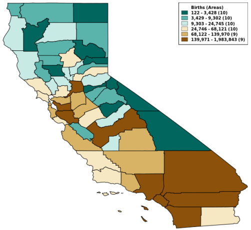
Unsupervised Discretization

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, II

Demo available

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, III

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, III

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 $c_{j,i}$,

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

where

$c_{j,i} = \frac{1}{i-j+1} \sum_{j \le \ell \le i} x_\ell$ and

$h = m$.

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.)