Brute Force Overview

while(candSolution hasNext)
    if(candSolution.next isDesiredSolution)
        [do whatever]

depending on the problem, what are the candidate solutions needed to be generated? and how do we check whether that solution is good

Brute force relies on the computer's brute force computing power.

Advantages:

- Straightforward, simple to code, test and understand.
- Directly based on the problem's definition which makes it quick to implement.
- Applicable to a wide variety of problems.

It can be used in a hybrid approach:

- Create a brute force solver and start it running while you try to think of a more advanced approach.

But a brute force solver may only work reasonably for small instances of a problem.

Brute Force: Exhaustive Search

Example: Print the divisors of a positive integer

divisors come in pairs, so you can chop in half (twice as efficient, but this is not efficient for large n, it doesn't matter about large n but it does for small n)

```
PrintDivisors (n) {
    for (i = 1; i <= n/2; i++)
        if (i divides n) print i, n/i
}
```

starting with the first candidate input divisor to the end, systematically generate all the candidates of things that might be divisors, as they come up, check them and if it works do what you need to do with them

after k iterations, we have a sorted portion of size k, and look at the unsorted portion, find the largest value, and swap it into the last position in the unsorted portion, growing the sorted portion by 1

Example: Selection Sort

- Find the largest value in the unsorted portion of the array and swap it into it’s spot
- Repeat.

systematically generate/explore each candidate-solution (for whatever the problem is, like who is the max? for this one), + as each one arises (from systematic generation) and simply "check it" (like check to see if it really is a divisor, check for mod, and this ex. to compare it to the max so far)

Brute Force: Exhaustive Search

Example: On an n x n chessboard, place n queens in such a way that they can't attack each other

// notice that a solution MUST place exactly one queen in each column of the chessboard,

// so solving this boils down to *which row should we choose for the queen in column c?*

// for each column

imagine search space as a decision tree

Systematically generate every array a of n integers between 1 and n {

for each column c, place the queen that goes in column c in row a[c]

if (the placement of the n queens is attack free)

print a and exit

if sol is an array, where the indices represent columns and the values represent rows

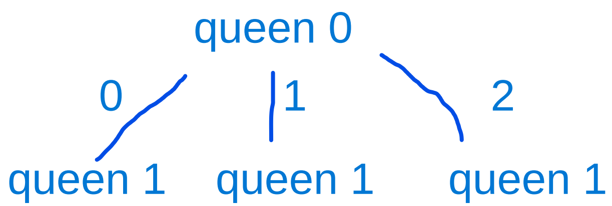
```
// print solution
for (i = 1; i < n; i++)
    print("queen" i "goes in row" sol[i]);
```

// generate all candidate solutions
when n is 3

generating sol arrays one at the time
with the same format of the sol array
we talked about earlier, so

[0 0 0] means there's a queen in the
0 row for every column

[0 2 1], [1 0 0], etc.



[each queen1 has 0, 1, 2
branches and so forth for queen 2]

to design a solution, things to consider:

- which decisions does the solver need to make? (ex. where to put the ith queen)
- how many decisions does the solver need to make? (ex. n queens)
- what are the options for each decision? (ex. first decision: where to put queen 0? options: something between 0 and n - 1 inclusive)
- how many decision combos would the solver consider? think about the search space (where the solver searches for the solution).

search space for n queens with n = 3; we're going to imagine this as a decision tree, doesn't mean the solver builds a tree

[queen 0] each decision is a node in this search space. you can put it in row 0, 1, or 2. whichever the solver chooses, they also need to make a decision for queen 1. the last row (leaves) are candidate solutions/complete decision combos (# of combos is n^n).

-- imagine example: with n decisions and 2 options per decision, we can imagine this as a binary tree gives us a total of 2^n decision combos

-- this algorithm is worse than polynomial; the total # of combos is n^n even if it takes less time to go from one combo to the next

thus, this is n^n for the last question on the things to consider (worse case)

brute force hops from 1 decision combo to the next systematically.

we can create a decision tree similar to n queens

item 0
take leave
item 1 item 1

Brute Force: Exhaustive Search

inputs: number of items in a museum (n)
how much profit that item generates (vi)
the weight of that item (wi)
the weight capacity of the backpack (W)

Brute-force approach to combinatorial problems.

- Involve permutations or combinations or subsets of a set.

goal: maximize the total profit for the job (max summation of vi of the stolen items subject to the max summation of wi < W of the stolen items)

Example: 0-1 Knapsack for each item, the thief has only 2 choices: take or leave it (hence 0 and 1, can't cut in half or reduce value)

A museum has n precious items. A thief robs the museum and either grabs or leaves each precious item, subject to the max weight they can carry.

on a map, we have a depot where the packages are filled in the truck. there are some packages that need to be dropped off at a bunch of locations. doesn't matter if we start at the depot or not (we can really start anywhere)

$O(2^n)$ possible subsets of items to consider.

Example: Traveling Salesperson Problem (TSP) (FedEx Delivery Problem)

minimizing cost of tour (also optimization)

In a graph with n locations, is there a sequence for visiting each location exactly once and return to the start?

what is a minimum cost tour/circuit/loop where we visit each location exactly once?

Or it's more intuitive version: What is cheapest way to visit each location exactly once and return to the start?

without replacement (no revisits/duplicates)

O(n!) possible sequences to consider //permutations of [1..n]

loc0
1 2 3
0-1 0-2 0-3

the options after decrease by 1 (node means from loc 0 to 1) because the location has been visited (last one has only 1 option) notice how for knapsack and sudoku, # decisions/options are related to find complete decision combos

To design/analyze a brute force ennumeration algorithm:	N queens	0-1 Knapsack	TSP	Sudoku
Which decision to be made	where to place Queen i	what to do about item i	where to go after location i	what value to assign to a blank cell
For each decision, what are the options/alternatives	row [0...(n-1)]	take/leave	the unvisited locations (varies as we travel)	[1...(w*h)] (w and h mean width and height for this)
# decisions	n	n (# of items)	O(n) b/c it's a linear # of decisions, n locations to choose to visit	b (decision for each blank cell on what to fill it as)
# candidate solutions (complete decision combos)	n^n	2^n (2 is the outcomes of an item)	(n - 1)! b/c you're not making decision for last since no choice	(w*h)^b

generating the candidate solution (knapsack) -- [0,0,0] is the case where you leave everything, [1,0,0] where the thief takes item0 and nothing else, [0,1,0], [0,1,1], [1,0,0], [1,1,0], [1,1,1]. as each candidate solution comes up, we need to check by verifying the solution is feasible (does it meet all the requirements/constraints of the problem? like here is the weight) and how profitable it is (does it beat the max profit so far?)

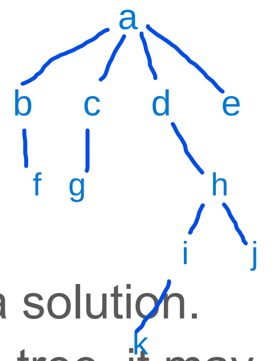
for this problem, you have to go through all 2^n solutions (global optimization) vs n queens where if you just find one solution you can stop. therefore, this is worse than polynomial.

tree traversals
- preorder (root - children)
- inorder (root left child)
- postorder (children - root)

Brute Force: Exhaustive Search

Example tree

(heaps)



An exhaustive search solver explores (all) the possible combinations of values that may lead to a solution. This solver might build a tree of its **search space**. Even when the solver doesn't explicitly build a tree, it may be helpful for us humans to imagine the search space as a tree. There are several ways of traversing a tree.

- Depth-First Search

```
Depth-first(Node root)
  print root
  for each child c of root
    Depth-first(c)
```

a, b, f, c, g, d, h, i, k, j, e (backtracking traversal)

  - Start at the root and explore a path all the way to a leaf, then backtrack to explore another path.

- Breadth-First Search
  by levels

```
Breadth-first(Node root)
  Q = new Queue
  Q enq(root)
  while(! Q.isEmpty)
    node = Q deq
    print node.label
    for each child c of node
      Q.enq(c)
```

the queue is entered in from the back [. . . *]
[k j i h g f e d c b a]
print: a b c d e f g h i j k

notice that enqueue means to add the node to the queue at the back

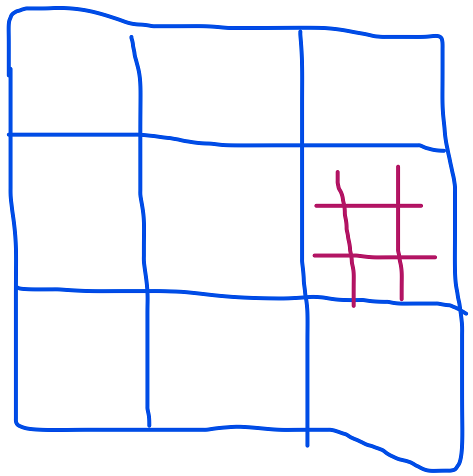
  - Start at the root and explore all nodes in a level before moving down a level and repeating.
  - Iterative implementation using a queue: while (more to go) {pop, process, enqueue children}

- Best-First Search

priority queue (some kind of heuristic, "evaluation function")

  - Rate (evaluate) each node in the search space by how **promising** it *seems*
  - Implement like Breadth-First Search, with a queue **sorted** by promising value

Brute Force: Exhaustive Search



notice both are 3x3

Example: Sudoku

Given a puzzle with

boxes of width w and height h

don't have to be squares, can be rectangles (still has same number of cells in each row and column though)

and b blank cells

there are $(w * h)^b$ possible assignments to consider.

grid is ((3x3) x (3x3)), each box has the purple in it
each cell is assigned a value in [1...(3x3)]

each value in [1...(3x3)] appears exactly once in each row, column, box
-- true out of 9 rows, columns, boxes

what is different in each puzzle are the preset values.
the job of the solver is to decide which value to place in each blank cell.

is there a way to cut down the # of solutions within the spirit of brute force so no partial solutions, backtracking?

-- checking how often each value occurs (it should occur a certain # of time)

sample solutions like [1,1,1,1,1,....,1], [1,1,1,1,1,...2]
as soon as we find the first solution, we're out of here

blank id
yellow - a list of all values yellow can have; initially it'll be 0 to 9
as the puzzle is read, eliminate values that it can't be (according to rows, cols, box)
so we get 7 (static based on input)

green - 1 2 3 4 5 6 7 8 9 --> 2 and 6 are the only options

we can then cut out some decision trees (significant runtime), like "preprocessing" before we go into the big loop

Application of the Brute Force Strategy to a tough problem: Design and Analyze a Brute Force Sudoku Solver

- The input to the Sudoku solver is a Sudoku puzzle, specified by
- w, the width of each box
- h, the height of each box
- b, the number of blank cells in the puzzle
- ($(w * h)^2$ - b) preset values in [1 . . $w * h$]

- We could represent the board with a 2-D array

Application of the Brute Force Strategy to a tough problem: Design and Analyze a Brute Force Sudoku Solver

- Algorithm:

```
for (each possible combination c of values for the b blank cells) {
        plug c into the blank cells of the puzzle;
        if (puzzle satisfies all the Sudoku rules) return true; // and print the puzzle
}
// nothing worked because we are still here
return false;
```

- What is the runtime complexity of this algorithm?

Application of the Brute Force Strategy to a tough problem: Design and Analyze a Brute Force Sudoku Solver

- Let's look at this algorithm more closely:

```
for (each possible combination c of values for the b blank cells) {
        plug c into the blank cells of the puzzle;
        if (puzzle satisfies all the Sudoku rules) return true; // and print the puzzle
}
return false; // no combination of values worked
```

- How would you generate the sequence of all possible combinations *c* of values for b blank cells, plug each successive combination into the puzzle, and repeatedly verify satisfaction of the rules?
- Is your design better than naively inefficient?
- What is the runtime complexity of this algorithm?

Recurring themes

- **Trade off** run**time for space** in memory

  (takes time)

  - instead of recomputing, compute once and store

  anytime you need something more than once (so we just fetch things when we need them, which takes memory)

- Select your **data structures** mindfully

  ex. heaps

- **Precompute**

  - Example 1: first sort, later use binary search instead sequential search

  - Example 2: parse the Sudoku input **instance** and **prune** surely **infeasible** options

  - Example 3: Horspool's string matching algorithm

Horspool’s string matching algorithm

- Find occurrences of a pattern in a text (a long string)
- Ex: Find the pattern BARBER in a text T
- Compare the pattern with the text at position i. If no match, advance to the next position in the text
- Naïve brute force: i++

  b/c it checks every single position (time consumption)
- Smarter brute force: we want to be able to **skip ahead** in the text **many positions** before making the next comparison

notice in terms of big o is the same amount of work, brute force only works for relatively small things like this

Horspool’s string matching algorithm

- Horspool’s algorithm precomputes a table with the shifts (skips)

  m is the length of the pattern P

  sizeAlphabet is the number of characters in the alphabet used in the pattern and the text

  table is an array of integers that is indexed by the alphabet’s characters and

  table [c] = the distance to the last character of P of the right-most occurrence of c in P

- table [0.. *sizeAlphabet*-1] **ComputeShiftTable** (P [0 .. *m* -1])

  O(sizeAlphabet) for (i=0; i < *sizeAlphabet*; i++) table[i] = m; // initialize table for all chars to the width of the pattern

  O(m) for (j=0; j < *m -1*; j++) table[P[j]] = m – 1 - j; // scan P and update table entries for its chars

- Naïve brute force: i++

BARBER (1st beginning), scan R to L, found mismatch at A so, let's slide the pattern so that the A lines up
The next mismatch is Y since there's no Y in the pattern

space complexity ~ O(sizeAlphabet) -> size of table, each int is constant

- For pattern BARBER, table[A] is 4, table[B] is 2, table[C] is 6, … table[R] is 3.

- Example with Text: `I_WEAR_MY_BEST_TO_THE_BARBERSHOP`

  Pattern: `BARBER`

anything that isn't in the pattern, slide the entire width of the pattern (6 in this example); for things that do occur, scan the string. at first it's B, update the entry of the shift table to 5 (5 - j)

you can update letters if they occur more than once; don't count the last symbol because there can never be a mismatch there

Horspool’s string matching algorithm

```
// returns the position in T where the first occurrence of P begins, or -1 if there is no match
// Scans T from left to right. When checking for a match, compare the chars in P from right to left

HorspoolMatching (P [0 .. m -1] , T [0 .. n -1] ) {
   Table = ComputeShiftTable (P)
   i = m - 1; // position of first possible occurrence of P’s last char
   while ( i < n - 1) {    as long as there are more symbols to check

      numMatchedChars = 0    needs to equal m for it to be found
      // scan T from right to left, beginning at position i, until a mismatch or a complete match
      while ( (numMatchedChars < m) and (P[m-1-numMatchedChars] == T[i-numMatchedChars]) )
         numMatchedChars ++    check the last char of the pattern that wasn't found yet w/ what was
                               found with that same char in the table of the text
                                                 leftmost
      if (numMatchedChars == m) return i – m + 1 // match found
      // else
      i = i + Table [ T[i] ]; // skip ahead depending on the char in position i    what did we find at the mismatch
   } // while (i …                                                                 move ahead depending on the alphabet

   return -1 // no match    not found
}
```

Horspool’s string matching algorithm

Runtime is $O(nm)$

But performs faster on average than naïve brute force.

Brute Force: Exhaustive Search

Example: SAT (Satisfiability problem)

Given a Boolean formula in **CNF** form, is it satisfiable. In other words,

is there an assignment of Boolean values to the variables in the formula that makes the formula evaluate to true.

Application of the Brute Force Strategy to a tough problem: SAT

An example formula with 4 Boolean variables **x1**, **x2**, **x3** and **x4** is

(**x1** or **x2**) and (not-**x2** or **x3** or $\boxed{\textbf{x4}}$) and ($\boxed{\text{not-}\textbf{x1}}$ or not-**x3** or not-**x4**)

A ***formula*** **in conjunctive normal form** is a conjunction (and) of clauses, or a single clause.

A ***clause*** is a disjunction (or) of literals, or a single literal.

A ***literal*** is a Boolean variable or the negation of a Boolean variable.

do we ever get a true? (truth table) if we do, it's satisfiable with some assignment (some mixture of T and F of what x1 and x2 can be, etc.)

non satisfiable example: x1 and not x1

one solution (assignment): T, F, T, F --> the formula is satisfiable

the most work is the unsatisfiable formulas

Application of the Brute Force Strategy to a tough problem: SAT

Definition of *assignment*

An assignment for a formula is a setting of truth values (either **true** or **false**) for the variables in the formula. An example assignment for the example formula given above is: **x1**, **x2** and **x4** are set to **true** and **x3** is set to **false**. We will write this assignment as (**x1, x2, x3, x4**) = (**true**, **true**, **false**, **true**).

Definition of *satisfying assignment* for a formula

An assignment that makes a formula evaluate to true is a satisfying assignment. Observe that the assignment (**x1, x2, x3, x4**) = (**true**, **true**, **false**, **true**) makes each clause in the example formula evaluate to true, and thus makes the example formula evaluate to true and is a satisfying assignment. On the other hand, the assignment (**x1, x2, x3, x4**) = (**false**, **false**, **true**, **true**) makes the above formula evaluate to **false** since it does not satisfy the formula's first clause, and therefore it is not a satisfying assignment.

Application of the Brute Force Strategy to a tough problem: SAT

Definition of *satisfiable formula*

A Boolean formula is satisfiable when it has a satisfying assignment.

The input to a SAT Solver is a CNF, and its output is True or False (whether the input CNF is satisfiable).

As a side effect, a solver might also print a satisfying assignment for a satisfiable input.

for this problem:

decision	what value to give to boolean variable i
# decisions	n
options per decision	because boolean, T/F
# candidate solutions	2^n

Application of the Brute Force Strategy to a tough problem: SAT

Things to think about:

How many different satisfying assignments can you find for the given example formula?

(**x1** or **x2**) and (not-**x2** or **x3** or **x4**) and (not-**x1** or not-**x3** or not-**x4**)

Can you write a CNF for which there is no satisfying assignment?

How many possible assignments exist for a CNF?

Application of the Brute Force Strategy: Design of a Brute Force SAT Solver

The Brute Force algorithm design strategy will systematically generate all possible variable assignments, checking each one as it goes along to determine whether it satisfies the input formula or not.

- Provide a high-level pseudocode for your Brute Force SAT Solver.
- How would you *systematically generate all possible variable assignments?* How efficient is your solution? Clearly explain how your solution is better than naively inefficient.
- How would you check a (the current) variable assignment to *determine whether it satisfies the input formula or not?* How efficient is your solution? Clearly explain how your solution is better than naively inefficient.
- What is the runtime complexity of your Brute Force SAT Solver?