Introduction to Competitive Programming

Instructor: William W.Y. Hsu

CONTENTS

- Complete search
- Iterative: (nested) loops, permutations, subsets
- Recursive backtracking (N queens), from easy to (very) hard
- State-space search
- Meet in the middle (bidirectional search)
- Some tips to speed up your solution
- Divide and conquer (D&C) algorithms
- Greedy algorithms

Before we begin…

Analyzing practice

- Given an array A containing $n \leq 10K$ small integers $\leq 100K$
  - $A = \{10, 7, 3, 5, 8, 2, 9\},\ n = 7$

- Find the largest and the smallest element of A.
  - *10 and 2 for the given example.*
- Find the *k*th smallest element in A.
  - *if* $k = 2$*, the answer is 3 for the given example.*
- Find the largest gap g such that $x, y \in$ A and $g = |x - y|$.
  - *8 for the given example.*
- Find the longest increasing subsequence of A.
  - *{3, 5, 8, 9} for the given example.*

Iterative complete search

Iterative complete search - Loops

- **UVa 725 – Division**
  - Find two 5-digits number s.t. → $\boldsymbol{abcde} / \boldsymbol{fghij} = N$.
  - $\boldsymbol{abcdefghij}$ must be all different, $2 \leq N \leq 79$.
- Iterative complete search solution (nested loops):
  - Try all possible $\boldsymbol{fghij}$ (one loop).
  - Obtain $\boldsymbol{abcde}$ from $\boldsymbol{fghij} \times \boldsymbol{N}$.
  - Check if $\boldsymbol{abcdefghij}$ are all different (*another* loop).
- More challenging variants:
  - 2- 3- 4- ⋯ - K nested loops
  - Some pruning are possible.
    - e.g. using "continue", "break", or if-statements

Iterative complete search – Nested loops

- Problems that are solvable with a *single* loop are usually considered *easy*!
- Problems which require doubly-nested iterations like UVa 725 - Division above are more challenging but they are not necessarily considered difficult.
- Competitive programmers must be comfortable writing code with ***more than two*** nested loops.
- **UVa 441 – Lotto**
  - Generating all possible permutations.
  - Can be solved with nested loops.

Iterative complete search – Loops + pruning

- **UVa 11565 - Simple Equations**
  - The third equation $x^2 + y^2 + z^2 = C$ is a good starting point.
  - Assuming that C has the largest value of 10000 and y and z are one and two (x, y, z have to be distinct), then the possible range of values for $x \in [-100 \ldots 100]$.
  - Use the same reasoning to get a similar range for y and z.
  - Write a triply-nested iterative solution below that requires $201 \times 201 \times 201 \approx 8M$ operations per test case.
  - **Can be solved with nested loops!**

Analysis

- Short circuit **AND** was used to speed up the solution by enforcing a *lightweight* check on whether x, y, and z are all different *before* we check the three formulas.
- We can also use the second equation $x \times y \times z = B$ and assume that $x = y = z$ to obtain $x \times x \times x < B$ or $x < \sqrt[3]{B}$.
  - The new range of $x \in [-22 \ldots 22]$.
- We can also prune the search space by using if statements to execute only some of the (inner) loops, or use `break` and/or `continue` statements to stop/skip loops.
- Try UVa 11571 - Simple Equations - Extreme!!

Iterative complete search - Permutations

- **UVa 11742 – Social Constraints**
  - There are $0 < \boldsymbol{n} \leq 8$ movie goers.
  - They will sit in the front row with $\boldsymbol{n}$ consecutive open seats.
  - There are $0 \leq \boldsymbol{m} \leq 20$ seating constraints among them, i.e. $\boldsymbol{a}$ and $\boldsymbol{b}$ must be at most (or at least) $\boldsymbol{c}$ seats apart.
  - How many possible seating arrangements are there?
- Iterative complete search solution (permutations):
  - Set `counter=0` and then try all possible $\boldsymbol{n}$! **permutations**.
  - Increase counter if a permutation satisfies all $\boldsymbol{m}$ constraints.
  - Output the final value of counter.

Code

```
#include <algorithm> // next_permutation is inside this C++ STL
// the main routine
int i, n = 8, p[8] = {0, 1, 2, 3, 4, 5, 6, 7}; // the first permutation
do {
  // try all possible O(n!) permutations, the largest input 8! = 40320
  ...
  // check the given social constraint based on 'p' in O(m)
} // the overall time complexity is thus O(m * n!)
while (next_permutation(p, p + n)); // this is inside C++ STL <algorithm>
```

Iterative complete search - Subsets

- **UVa 12455 – Bars**
- We can try all $2n$ possible subsets of integers, sum the selected integers for each subset in $O(n)$, and see if the sum of the selected integers equals to X
- The overall time complexity is thus $O(n \times 2n)$.
  - For the largest test case when $n = 20$, this is just $20 \times 220 \approx 21M$.
  - This is 'large' but still viable (for reason described below).
- An easy solution is to use the *binary representation* of integers from 0 to $2n - 1$ to describe all possible subsets.
  - Bit manipulation operations are (very) fast, the required $21M$ operations for the largest test case are still doable in under a second.

Iterative complete search - Subsets

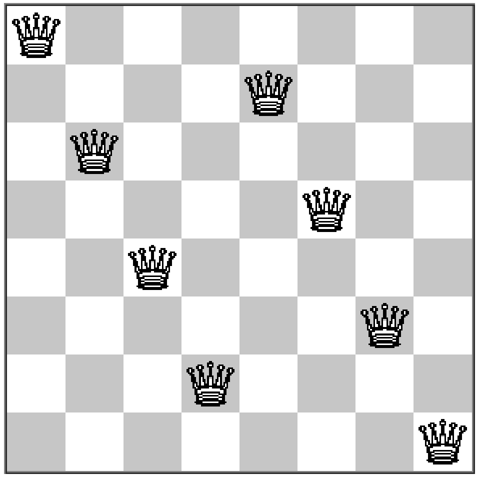
- **UVa 12346 – Water Gate Management**
  - A dam has $1 \leq \boldsymbol{n} \leq 20$ water gates to let out water when necessary, each water gate has **flow rate** and **damage cost.**
  - Your task is to manage the opening of the water gates in order to get rid of *at least* the specified **total flow rate** condition that the **total damage cost** is minimized!
- Iterative complete search solution (subsets):
  - Try all possible $\mathbf{2}\boldsymbol{n}$ subsets of water gates to be opened.
  - For each subset, check if it has sufficient flow rate:
    - If it is, check if the total damage cost of this subset is smaller than the overall minimum damage cost so far.
    - – If it is, update the overall minimum damage cost so far.
  - Output the minimum damage cost.

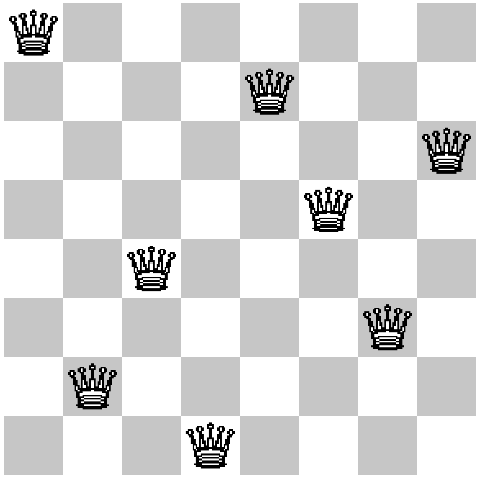
Recursive backtracking

N Queens, from easy to (very) hard

Recursive backtracking

- **UVa 750 – 8 Queens Chess Problem**
  - Put 8 queens in 8x8 Chessboard.
  - No queen can attack other queens.
- Naïve ways (Time Limit Exceeded)
  - Choose 8 out of 64 cells.
  - C_8^{64} = 4,426,165,368 possibilities!
- Insight 1: Put one queen in each column
  - 8^8 = 16,777,216 possibilities.





Recursive backtracking

- Better way, recursive backtracking
  - Insight 2: **all-different constraint** for the rows too
    - We put one queen in each column **AND each row.**
    - Finding a valid permutation out of **8**! possible permutations.
    - Search space goes down from $8^8 \cong 17M$ to $8! = 40320!$
  - Insight 3: **main** diagonal **and secondary diagonal** check:
    - Another way to prune the search space.
    - Queen $A(i, j)$ attacks Queen $B(k, l)$ iff $abs(i - k) = abs(j - l)$.
- Scrutinize the sample code of recursive backtracking!

Is that the best n-Queens solution?

- Maybe not!
- See **UVa 11195 – Another n-Queen Problem**
  - Several cells are forbidden
    - Do this helps?
- n can now be as large as $n = 14$??
  - How to run 14! algorithm in a few seconds?

Speeding up diagonal checks

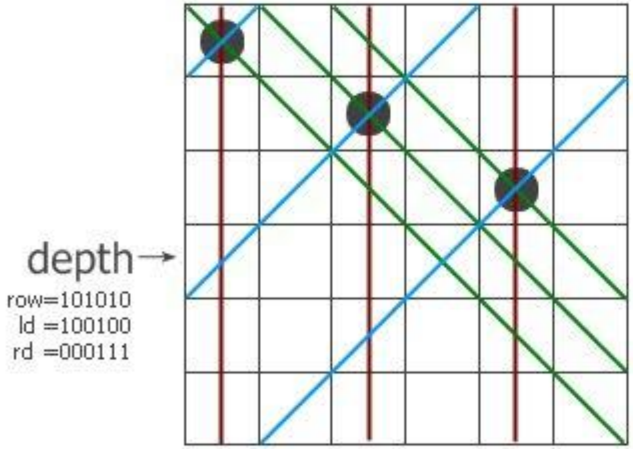
- This check is slow:

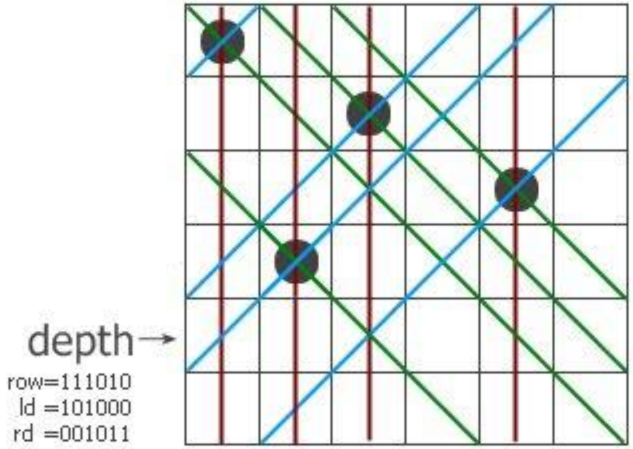
```
bool place(int r, int c) {
  for(int prev = 0; prev < c; prev++) // check previously placed queens
    if(rw[prev] == r || (abs(rw[prev] - r) == abs(prev - c)))
      return false; // share same row or same diagonal -> infeasible
  return true;
}
```

- We can speed up this part by using $2 \times n - 1$ boolean arrays (or bitset) to test if a certain left/right diagonal can be used.

Speeding up diagonal checks

- The ***queen*** function takes three parameters, **`row`**, **`ld`**, **`rd`** representing the forbidden places of current row, left diagonal and right diagonal respectively.
- The **`row | ld | rd`** combines all invalid positions.
  - **~** is the boolean **not** operation which gives the valid `position.`
  - `p&-pos` equals to the right-most one. i.e. `-pos=~pos+1`





State-space search

UVa 11212 – Editing a book

- Given n equal-length paragraphs numbered from 1 to n.
- Arrange them in the order of $1, 2, \ldots, n$
- With the help of a clipboard, you can press **Ctrl-X** (cut) and **Ctrl-V** (paste) several times.
  - You cannot cut twice before pasting, but you can cut several contiguous paragraphs at the same time - they'll be pasted in order.
- The question: What is the minimum number of steps required?
- Example 1: In order to make {2, 4, (1), 5, 3, 6} sorted, you can
  - Cut 1 and paste it before 2→{**1,** 2, 4, 5, (3), 6}
  - then cut 3 and paste it before 4 →{1, 2, **3,** 4, 5, 6}.
- Example 2: In order to make {(3, 4, 5), 1, 2} sorted, you can
  - Cut {3, 4, 5} and paste it after {1, 2} →{1, 2, **3, 4, 5**}
  - or cut {1, 2} and paste it before {3, 4, 5} →{**1, 2**, 3, 4, 5}

Loose upper bound

- Answer: $k-1$
  - Where k is the number of paragraphs initially the wrong positions.
- Trivial but wrong algorithm:
  - Cut a paragraph that is in the wrong position.
  - Paste that paragraph in the correct position.
  - After $k-1$ such cut-paste, we will have a sorted paragraph.
    - The last wrong position will be in the correct position at this stage
  - But this may not be the shortest way.
- • Examples:
  - {(3), 2, 1}→{(2), 1, **3**}→{1, **2**, 3}→2 steps
  - {(5), 4, 3, 2, 1}→{(4), 3, 2, 1, **5**}→{(3), 2, 1, **4**, 5}→{(2), 1, **3**, 4, 5}→{1, **2**, 3, 4, 5}→4 steps

The correct answer

- {3, 2, 1}
  - Answer: 2 steps, e.g.
    - {(3), 2, 1}→{(2), 1, **3**}→{1, **2**, 3}, or
    - {3, 2, (1)}→{**1**, (3), 2,}→{1, 2, **3**}
- {5, 4, 3, 2, 1}
  - Answer: Only **3** steps, e.g.
    - {5, 4, (3, 2), 1}→{**3, (2**, 5), 4, 1}→{3, 4, (1, **2), 5**}→{**1, 2**, 3, 4, 5}
- How about {5, 4, 9, 8, 7, 3, 2, 1, 6}?
  - Answer: 4, but very hard to compute manually.
- How about {9, 8, 7, 6, 5, 4, 3, 2, 1}?
  - Answer: 5, but very hard to compute manually.

Some analysis

- There are at most $n!$ permutations of paragraphs
  - With maximum $n = 9$, this is $9! = 362880$.
  - The number of vertices is not that big actually.
- Given a permutation of length n (a vertex)
  - There are C_2^n possible cutting points (index $i, j \in [1..n]$)
  - There are n possible pasting points (index $k \in [1..(n\text{-}(j - i + 1))]$)
  - Therefore, for each vertex, there are about $O(n^3)$ branches.
- The worst case behavior if we run a single BFS on this State-Space graph is: $O(V + E) = O(n! + n! \times n^3) = O(n! \times n^3)$.
  - With $n = 9$, this is $9! \times 93 = 264539520 \cong 265\ M$
  - TLE (or maybe MLE…)!

Possible solution

› Iterative deepening A* (IDA*)

› Prune condition: $3d + h > 3\text{maxd}$

  – 3*depth + current height should be smaller than 3 * max depth

Divide and conquer

Divide and conquer

- A problem-solving paradigm in which a problem is made *simpler* by ‘dividing’ it into smaller parts and then conquering each part.
  1. Divide the original problem into *sub*-problems—usually by half or nearly half.
  2. Find (sub)-solutions for each of these sub-problems—which are now easier.
  3. If needed, combine the sub-solutions to get a complete solution for the main problem.

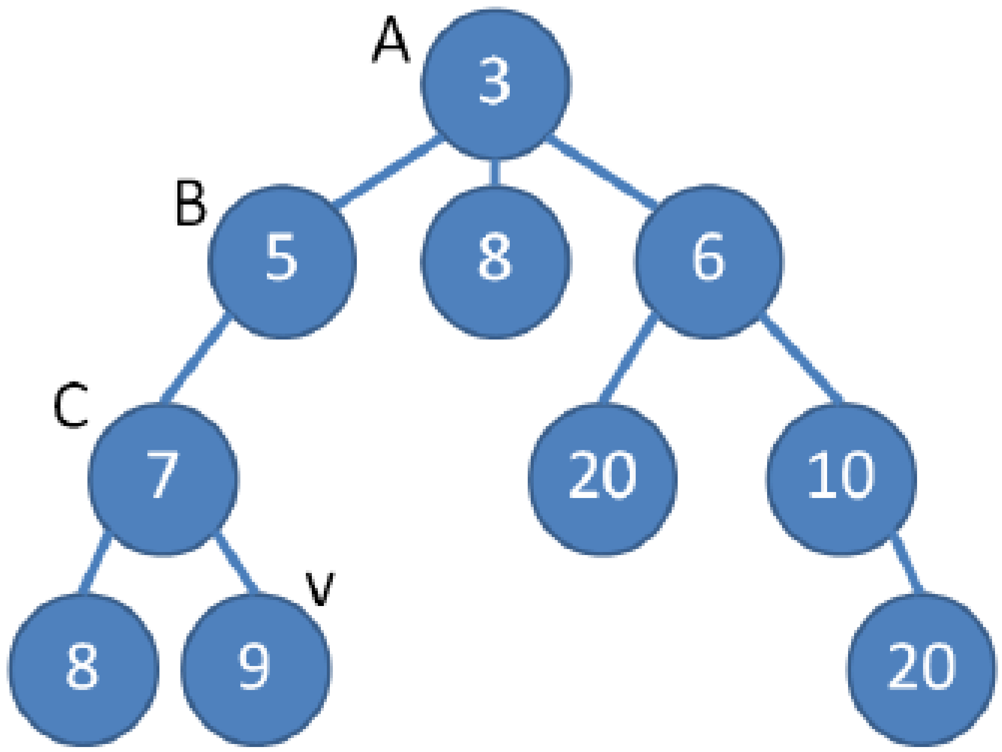
Binary search – Ordinary usage

- The *canonical* usage of Binary Search is searching for an item in a *static sorted array*.
  - We check the middle of the sorted array to determine if it contains what we are looking for.
  - If it is or there are no more items to consider, stop.
  - Otherwise, we can decide whether the answer is to the left or right of the middle element and continue searching.
- There are built-in library routines for this algorithm, e.g. the C++ STL `algorithm::lower_bound` (and the Java `Collections.binarySearch`).

Binary search - Uncommon data structures

- Thailand ICPC National Contest 2009. 'My Ancestor'
- Given a weighted (family) tree of up to $N \leq 80K$ vertices with a special trait: *Vertex values are increasing from root to leaves*. Find the *ancestor* vertex closest to the root from a starting vertex *v* that has weight at least P. There are up to $Q \leq 20K$ such *offline* queries.
- If $P = 4$, then the answer is the vertex labeled with 'B' with value 5 as it is the ancestor of vertex *v* that is closest to root 'A' and has a value of ≥ 4. If $P = 7$, then the answer is 'C', with value 7. If $P \geq 9$, there is no answer.

Binary search - Uncommon data structures

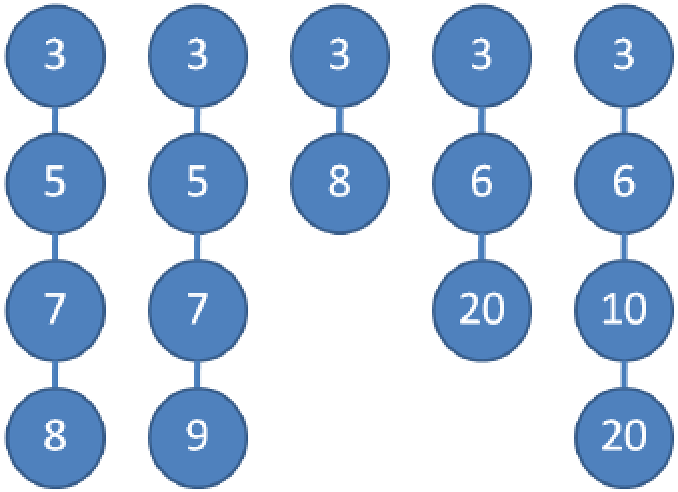


Binary search - Uncommon data structures

- The naive solution is to perform a linear $O(N)$ scan per query
- As there are Q queries, this approach runs in $O(QN)$ (the input tree can be a sorted linked list, or rope, of length N) and will get a TLE as $N \leq 80K$ and $Q \leq 20K$.

Binary search - Uncommon data structures

- A better solution is to store all the $20K$ queries!
- Traverse the tree *just once* starting from the root using the $O(N)$ preorder tree traversal algorithm.
- The array is always sorted because the vertices along the root-to-current-vertex path have increasing weights.



Binary search - Uncommon data structures

- During the preorder traversal, when we land on a queried vertex, we can perform a $O(logN)$ **binary search** (to be precise: `lower_bound`) on the partial root-to-current-vertex weight array to obtain the ancestor closest to the root with a value of at least P, recording these solutions.
- Finally, we can perform a simple $O(Q)$ iteration to output the results.
- The overall time complexity of this approach is $O(QlogN)$.

Binary search - Bisection method

- Used to find the root of a function that may be difficult to compute directly.
- Bisection method only requires $O(\log_2(b-a)/\epsilon)$ iterations to get an answer that is good enough.

Binary search the answer

- UVa 11935 - Through the desert
  - If we know the jeep's fuel tank capacity, then this problem is just a simulation problem.
  - The problem is that we do not know the jeep's fuel tank capacity—this is the value that we are looking for.
  - From the problem description, we can compute that the range of possible answers is between $[0.000..10000.000]$, with 3 digits of precision. However, there are $10M$ such possibilities. (TLE!)
  - Binary search!
    - Setting your jeep's fuel tank capacity to any value between $[0.000..X - 0.001]$ will *not* bring your jeep safely to the goal event.
    - On the other hand, setting your jeep fuel tank volume to any value between $[X..10000.000]$ will bring your jeep safely to the goal event, usually with some fuel left.

Greedy

Greedy

- An algorithm is said to be greedy if it makes the locally optimal choice at each step with the hope of eventually reaching the globally optimal solution.
- Two properties in order for a greedy algorithm to work:
  - It has optimal sub-structures.
  - It has the greedy property (difficult to prove in time-critical contest environment!).

Coin change - The greedy version

- Given a target amount V cents and a list of denominations of n coins, i.e. we have `coinValue[i]` (in cents) for coin types $i \in [0..n-1]$, what is the minimum number of coins that we must use to represent amount V?
  - If $n = 4$, `coinValue`={25, 10, 5, 1} cents[6], and we want to represent $V = 42$ cents, we can use this Greedy algorithm:
    - Select the largest coin denomination which is not greater than the remaining amount.
    - 42-25 = 17 → 17-10 = 7 → 7-5 = 2 → 2-1 = 1 → 1-1 = 0, a total of 5 coins.

Coin change - The greedy version

- The problem above has the two ingredients required for a successful greedy algorithm:
  - It has optimal sub-structures.
  - It has the greedy property.

Coin change - The greedy version

- This greedy algorithm does *not* work for *all* sets of coin denominations.
- Consider $\{4, 3, 1\}$ cents.
  - To make 6 cents with that set, a greedy algorithm would choose 3 coins $\{4, 1, 1\}$ instead of the optimal solution that uses 2 coins $\{3, 3\}$.

Try these problems

- UVa 410 - Station Balance (Load Balancing)
- UVa 10382 - Watering Grass (Interval Covering)
- UVa 11292 - Dragon of Loowater (Sort the Input First)

Meet in the middle

Bidirectional search

More search algorithms…

- Depth Limited Search (DLS) + Iterative DLS
- A* / Iterative Deepening A* (IDA*) / Memory Bounded A*
- Branch and Bound (B&B)

Remarks

- The biggest gamble in writing a **Complete Search** solution is whether it will or will not be able to pass the time limit.
- Tweaking `critical code' may not be as efficient.
  - A saying is that every program spends most of its time in only about 10% of its code — the critical code.

Remarks

- Tip 1: Filtering versus generating
  - Programs that examine lots of (if not all) candidate solutions and choose the ones that are correct (or remove the incorrect ones) are called ‘filters’.
  - Programs that gradually build the solutions and immediately prune invalid partial solutions are called ‘generators’.
  - Generally, filters are easier to code but run slower, given that it is usually far more difficult to prune more of the search space iteratively.
  - Do the **math** (complexity analysis) to see if a filter is good enough or if you need to create a generator.

Remarks

- Tip 2: Prune infeasible/inferior search space early
  - When generating solutions using recursive backtracking (see the tip no 1 above), we may encounter a partial solution that will never lead to a full solution.
  - We can prune the search there and explore other parts of the search space.
  - In other problems, we may be able to compute the ‘potential worth’ of a partial (and still valid) solution.
    - A* algorithm.

Remarks

- Tip 3: Utilize symmetries
  - Some problems have symmetries and we should try to exploit symmetries to reduce execution time!
  - In the 8-queens problem, there are 92 solutions but there are only 12 unique (or fundamental/canonical) solutions as there are rotational and line symmetries in the problem.
  - It is true that sometimes considering symmetries can actually complicate the code.

Remarks

- Tip 4: Pre-computation a.k.a. Pre-calculation
  - Generate tables or other data structures that accelerate the lookup of a result prior to the execution of the program itself.
  - However, this technique can rarely be used for recent programming contest problems.

Remarks

- Tip 5: Try solving the problem backwards
  - Some contest problems look far easier when they are solved 'backwards' (from a *less obvious* angle) than when they are solved using a frontal attack.
- **UVa 10360 - Rat attack**
  - Imagine a 2D array (up to 1024×1024) containing rats. There are $n \leq 20000$ rats spread across the cells.
  - Determine which cell (x, y) should be gas-bombed so that the number of rats killed in a square box $(x - d, y - d)$ to $(x + d, y + d)$ is maximized. The value d is the power of the gas-bomb ($d \leq 50$),

First try

- Bomb each of the 1024^2 cells and select the most effective location.
- For each bombed cell (x, y), we can perform an $O(d^2)$ scan to count the number of rats killed within the square bombing radius.
- For the worst case, when the array has size 1024^2 and $d = 50$, this takes $1024^2 \times 502 = 2621M$ operations.

Inverse problem

- Another option is to attack this problem **backwards**:
  - Create an array `int killed[1024][1024]`.
  - For each rat population at coordinate (x, y), add it to `killed[i][j]`, where $|i - x| \leq d$ and $|j - y| \leq d$.
  - This is because if a bomb was placed at (i, j), the rats at coordinate (x, y) will be killed.
  - This pre-processing takes $O(n \times d^2)$ operations.
  - To determine the most optimal bombing position, find the coordinate of the highest entry in array killed, which can be done in 1024^2 operations.
  - This approach only requires $20000 \times 50^2 + 1024^2 \cong 51M$ operations for the worst test case $(n = 20000, d = 50)$
  - $\approx$ 51 times faster than the frontal attack!

Remarks

- Tip 6: Optimizing your source code
- Understanding computer hardware and how it is organized, especially the I/O, memory, and cache behavior, can help you design better code.
- Some examples (not exhaustive) are shown below:
  - A biased opinion: Use C++ instead of Java.
    - An algorithm implemented using C++ usually runs faster than the one implemented in Java in many online judges, including UVa.
    - Some programming contests give Java users extra time to account for the difference in performance.
  - For C/C++ users, use the faster C-style `scanf/printf` rather than `cin/cout`. For
  - Java users, use the faster `BufferedReader/BufferedWriter` classes.

Remarks

- Use the *expected* $O(n \log n)$ but cache-friendly quicksort in C++ STL `algorithm::sort` (part of '**introsort**') rather than the true $O(n \log n)$ but non cache-friendly **heapsort**.
- Access a 2D array in a row major fashion (row by row) rather than in a column major fashion
- Bit manipulation on the built-in integer data types (up to the 64-bit integer) is more **efficient** than index manipulation in an array of Booleans.

Remarks

- Use lower level data structures/types at all times if you do not need the extra functionality in the higher level (or larger) ones.
- For Java, use the faster `ArrayList` (and `StringBuilder`) rather than `Vector` (and `StringBuffer`).
  - Java Vectors and StringBuffers are *thread safe* but this feature is not needed in competitive programming.
- Declare most data structures (especially the bulky ones, e.g. large arrays) once by placing them in global scope.

Remarks

- When you have the option to write your code either iteratively or recursively, choose the iterative version.
  - The iterative C++ STL `next_permutation` and iterative subset generation.
- Array access in (nested) loops can be slow.
- In C/C++, *appropriate* usage of macros or inline functions can reduce runtime.
- For C++ users: Using C-style character arrays will yield faster execution than when using the C++ STL string.
- For Java users: Be careful with String manipulation as Java String objects are **immutable**.

Remarks

- Tip 7: Use better data structures & algorithms
  - Using better data structures and algorithms will always outperform any optimizations.