3. Complete search, divide & conquer, greedy

CPCFI

UNAM's School of Engineering

2021

Based on: Halim S., Halim F. *Competitive Programming 3*. Handbook for ACM ICPC and IOI Contestants. 2013

Table of Contents

Background

In general, we have two types of algorithmic problems:

1. **Optimization**: we'd like to find a minimum or maximum within a *search space*
2. **Decision**: we'd like to answer 'yes' or 'no' for a problem within a *search space*

Algorithmic examples

For example:

- **Sudoku**: given a sudoku board with some numbers, decide whether or not a solution exists
- **Bipartite graph**: given a graph G, decide if a partition of its vertices exists such that G is bipartite
- **Robotic arm**: given some points in a plane, we'd like to find a Hamiltonian cycle of minimum length that goes through all points

Search Space

Definition: the state's space for a problem (optimization or decision) is the set of all possible configurations or inputs we must consider in order to answer the problem.

- Numbers from 1 to n
- Permutations of n elements
- Sets of n elements
- For a given k, all subsets of size k from n possible elements
- Vectors of m elements taken from a set of n elements

Example #0

Given a list of n integers, we'd like the following:

- Decide if two numbers exist such that its sum equals 1000
- Decide which two numbers have the minimum sum

*The first problem is a decision problem, while the second is an optimization problem.

Algorithmic heuristics

Once we know the type of algorithmic problem (optimization or decision) and the search space S, we can start looking for the optimal solution (or correct solution) within S. For this, we can use the following heuristics:

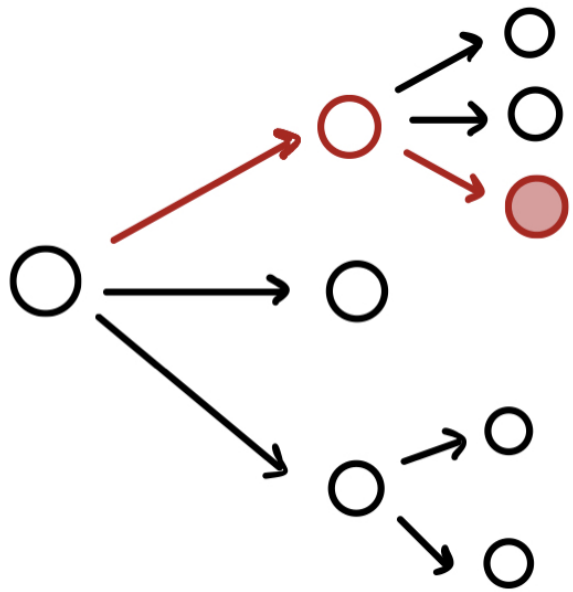
- Explore the search space S
  - Complete Search (Brute Force): explore all the elements of S
  - Pruning: explore only the elements of S that we know have a possibility of being the optimal or correct solution
- Divide and Conquer
- Greedy algorithms
- Dynamic Programming [1]

[1] We look into this heuristic in slides #4 and #5

Table of Contents

Complete Search

The idea:



Complete Search

- Heuristic for solving a problem by traversing through the entire (or part of) the search space to obtain a correct solution
- Also known as *brute force* or *recursive backtracking*
- During the search, one could prune the search space (Pruning technique) by only traversing through possible solutions and ignoring solutions that have no possibility of containing the correct solution

Complete Search

One should develop a complete search solution when:

1. No other clear algorithm exists
2. A better algorithm exists but are too complex for a small input size [2]

In ICPC, Complete Search should be the first solution considered as it is usually a simple technique to code and debug.

[2] For example, Range Minimum Queries on static arrays with $N < 100$ is solvable with $O(N)$ loop for each query

Complete Search on ICPC

- Complete Search should never get the Wrong Answer (WA) veredict since it always finds the correct solution. However, time limit exceeded (TLE) is very common to occur since it's a slow technique
- One should do a proper analysis before implementing Complete Search

Recall the following table:

n	Worst AC Algorithm	Comment
$\leq [10..11]$	$O(n!), O(n^6)$	e.g. Enumerating permutations (Section 3.2)
$\leq [15..18]$	$O(2^n \times n^2)$	e.g. DP TSP (Section 3.5.2)
$\leq [18..22]$	$O(2^n \times n)$	e.g. DP with bitmask technique (Section 8.3.1)
≤ 100	$O(n^4)$	e.g. DP with 3 dimensions + $O(n)$ loop, ${}_nC_{k=4}$
≤ 400	$O(n^3)$	e.g. Floyd Warshall's (Section 4.5)
$\leq 2K$	$O(n^2 \log_2 n)$	e.g. 2-nested loops + a tree-related DS (Section 2.3)
$\leq 10K$	$O(n^2)$	e.g. Bubble/Selection/Insertion Sort (Section 2.2)
$\leq 1M$	$O(n \log_2 n)$	e.g. Merge Sort, building Segment Tree (Section 2.3)
$\leq 100M$	$O(n), O(\log_2 n), O(1)$	Most contest problem has $n \leq 1M$ (I/O bottleneck)

3.2.1 Iterative Complete Search - Example #1

- UVa 725 - Division
- C++ code: `/CompleteSearch/275.cpp`

Problem description

Write a program that finds and displays all pairs of 5-digit numbers that between them use the digits 0 through 9 once each, such that the first number divided by the second is equal to an integer N, where $2 \leq N \leq 79$. That is,

$$\frac{abcde}{fghij} = N$$

where each letter represents a different digit. The first digit of one of the numerals is allowed to be zero.

Iterative Complete Search - Example #2

- UVa 441 - Lotto
- C++ code: `/CompleteSearch/441.cpp`

Problem description

Given $6 < k < 13$ integers, enumerate all possible subsets of size 6 of these integers in sorted order. Since the size of the required subset is always 6 and the output has to be sorted lexico-graphically (the input is already sorted), the easiest solution is to use six nested loops. Even when $k = 12$, we only have:

$$C_6(12) = \binom{12}{6} = \frac{12!}{6! \cdot (12-6)!} = 924$$

combinations

Iterative Complete Search - Example #3

- UVa 11565 - Simple Equations
- C++ code: /CompleteSearch/11565.cpp

Problem Description

Given three integers A, B, C ($1 \leq A, B, C \leq 10000$), find three other **distinct** integers x, y, z such that $x + y + z = A$, $x \times y \times z = B$ and $x^2 + y^2 + z^2 = C$.

Hint: We must avoid three nested loops from $1 \rightarrow 10000$?

3.2.2 Recursive Complete Search

Before we look into these examples, we must study **Backtracking**

Index

Backtracking - Definition

Definition: backtracking is a systematic way of running through all possible configurations of a search space.

- Regarding the type of search space, all search spaces must generate each possible configuration **exactly once**
- In order to avoid repetitions and missed configurations, we must define a systematic generation order

Backtracking - The Idea

- Backtracking models the combinatorial solution by using a vector $a = (a_1, a_2, \ldots, a_n)$ where each element a_i is selected from a finite ordered set S_i
- The vector a represents the solution
- At each step in the backtracking algorithm, we try to extend a by adding another element at the end
- Backtracking constructs a tree of partial solutions

Backtracking - Algorithm

```
1  Backtrack(a,k)
2  if a=(a1,a2,...,an) is a solution
3    Report it
4  else
5    k=k+1
6    construct Sk //set of candidates for position k of a
7    while Sk has elements
8      ak = an element in Sk
9      Sk = Sk - {ak}
10     Backtrack(a,k)
```

Backtracking - Examples: Subsets

Problem: Find the subsets of an n-element set. For example, the integers $\{1, \ldots, n\}$

For $n = 1$:
2^1 subsets: $\{\}, \{1\}$
For $n = 2$:
2^2 subsets: $\{\}, \{1\}, \{2\}, \{1, 2\}$
For $n = 3$:
2^3 subsets: $\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}$
$\vdots$

For $n = k$:
2^k subsets

C++ code: `/CompleteSearch/backtracking_subsets.cpp`

Backtracking - Examples: Permutations

Problem: Generate all permutations of n integers: $\{1, \ldots, n\}$

For $n = 1$: $\{0\}$
For $n = 2$: $\{01, 10\}$
For $n = 3$: $\{012, 021, 102, 120, 201, 210\}$

$\vdots$

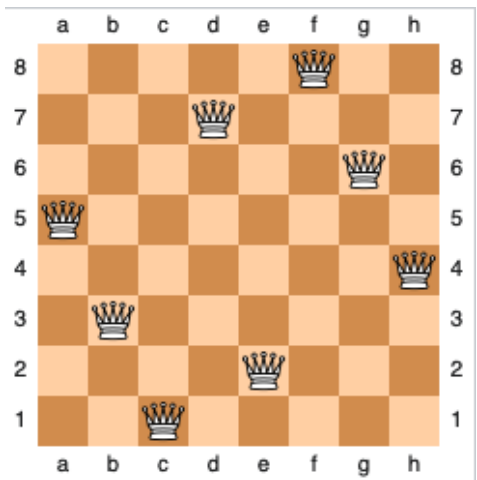
C++ code:
/CompleteSearch/backtracking_permutations.cpp

Recursive Complete Search - Example #4

- UVa 750 - 8 Queens Chess Problem
- C++ code: `/CompleteSearch/750.cpp`

Problem Description

In chess (with an 8×8 board), it is possible to place eight queens on the board such that no two queens attack each other.
Determine all such possible arrangements given the position of one of the queens (i.e. coordinate (a, b) must contain a queen).



CPCFI

Recursive Complete Search - Example #5

- UVa 11195 - Another n-Queen Problem
- `C++ code: /CompleteSearch/11195.cpp`

Problem Description

Given an $n \times n$ chessboard ($3 < n < 15$) where some of the cells are bad (queens cannot be placed on those bad cells), how many ways can you place n queens in the chessboard so that no two queens attack each other? Note: Bad cells cannot be used to block queens' attack.

Recursive Complete Search - Example #5

- The recursive backtracking algorithm presented for the 8-Queen problem is not fast enough for $n = 14$
- It's too slow when checking whether the position of a new queen is valid since we compare the new queen's position with the previous $c - 1$ queens' positions
- Therefore, we must store the same information with three boolean arrays
  - `rw`: represents the n rows
  - `ld`: $2 \times n - 1$ left diagonals
  - `rd`: $2 \times n - 1$ right diagonals

Recursive Complete Search - Example #5

- When a queen is placed at (r, c) we flag `rw[c] = true` to prevent this **row** from being used again
- In addition to this, we need to prevent its left and right **diagonals** from being used:
  - If another queen (a, b) is placed in either a right or left diagonal of the original queen (r, c), then the following equation is satisfied:

$$\text{abs}(r - a) = \text{abs}(c - b)$$

$$r - c = a - b \qquad \text{or} \qquad r + c = a + b$$

  - Since $r - c$ can be negative, we must add an offset of $n + 1$
  - Therefore, we flag `ld[r-c+n+1]=true` and `rd[r+c]=true` if a queen is placed at (r, c)

3.2.3 Tips

- One of the biggest problems with Complete Search is whether your solution will pass in the right amount of time or will get a Time Limit Exceeded (TLE) verdict
- Now, we'll review some tips to consider when designing a Complete Search solution

Tip #1: Filtering vs. Generating

Filters: programs that examine most of the candidate solutions and choose the correct ones

Generators: programs that gradually build the solutions and immediately prune invalid or partial solutions

- Filters are easier to code but run slower since it's much harder to prune the search space iteratively
- Before programming a generator, do an algorithm analysis to know if a filter is good enough

Tip #2: Prune Death Ends Early

- When generating solutions using recursive backtracking (recall UVa 750 - 8-queens problem) we might encounter partial solutions that we know will never produce a correct solution
  - For example, suppose we have an initial queen at `row[0]=2`. Knowing this, we know that any board configuration with queens at `row[1] = 1` or `row[1]=0` will never produce correct solutions, thus, we must stop at this point and focus on the next possible solutions (`row[1] != {0,1,2}`)
- We need to identify this paths as early as possible and stop the search at that point
- The earlier you prune the search space, the better

Tip #3: Exploit symmetries

- We must exploit problem domain symmetries to reduce execution time
- Using symmetries could increase code complexity and thus time debugging
- Exploit symmetries only if the reduced time is significant

Tip #4: Optimize Your Source Code I

1. If possible, use C++ instead of Java since it's faster
2. Use `scanf/printf` instead `cin/cout`
3. Use C++ STL `algorithm::sort`
4. Access a 2D array row by row; it's faster
5. Bit manipulation on the built-in integer data types is more efficient that index manipulation in a boolean array
6. Use lower level data structures (or types) at all times if you do not need the extra functionality of the higher level ones. For example, use an `array` with a slightly larger size than the maximum input size instead of a resizable array (`vector`)

Tip #4: Optimize Your Source Code II

7. Declare most data structures once by placing them in a global scope and allocate enough memory to deal with the largest input of the problem. This way we do not need to pass data structures around as function arguments
8. If possible, write your code iteratively instead of recursively
9. If you have an array `A` and you frequently access the value `A[i]` within nested loops, then, a better option would be to declare a variable `temp = A[i]` and work with `temp` instead
10. Appropriate usage of macros or inline functions can reduce time
11. Using C-style character arrays will perform better than using the C++ STL `string` class

Tip #5: Iterative vs. Recursive Complete Search I

If a problem is solvable by Complete Search, it will also be clear when to use the iterative or recursive backtracking approaches:

1. **Iterative** approaches are used when it's easy to derive the different states or all states have to be checked
   - *Subsets, permutations, etc*
2. **Recursive** approaches are used when it's hard to derive states with a simple index or when one wants to prune the search space
   - *n-queens problem, etc*

Tip #5: Iterative vs. Recursive Complete Search II

If the search space of a problem - that can be solved using Complete Search - is large, then recursive backtracking approaches that allow early pruning of infeasible sections are usually better

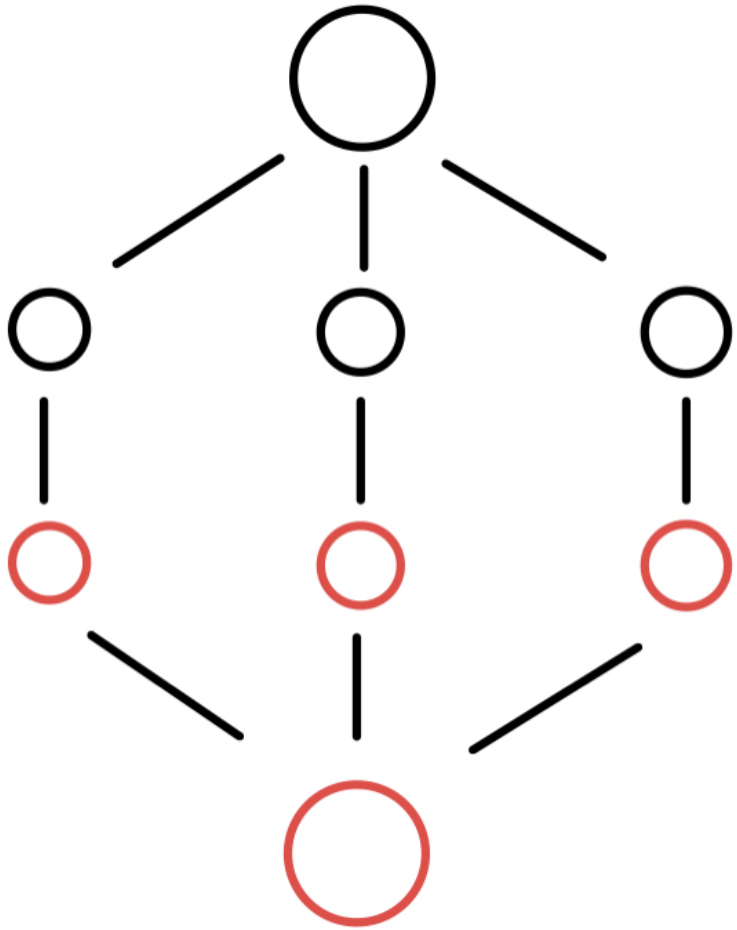
Index

UVa - Online Judge

- Complete Search Problems: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=639
- If **UVa Online Judge** is not working, please refer to [Halim]: page 80

Table of Contents

Divide and Conquer - The Idea



Divide and Conquer - Definition

Divide and Conquer (DQ) is a problem solving paradigm in which a problem is divided into smaller parts and then each part is solved (conquered). The general steps are the following:

1. **Divide** the original problem into sub-problems - usually in halves
2. **Solve** each of these sub-problems - which are now easier
3. **Combine** these solutions to obtain a complete solution of the problem

Divide and Conquer - Examples

Some examples of DQ:

- Quick Sort
- Merge Sort
- Heap Sort
- Binary Search

3.3.1 Interesting Usages of Binary Search

Binary Search algorithm is an example of the Divide and Conquer paradigm; however, there are other ways in which this algorithm could be used:

- Ordinary Usage
- On Uncommon Data Structures
- Bisection Method
- Binary Search the Answer

Binary Search - Ordinary Usage

- The canonical way of using *Binary Search* is searching for an item in a *static sorted array*
- We check the middle element of the sorted array to determine if it contains the element we are searching
- If not, we can decide whether the answer is to the left or the right of the middle element
- On each iteration, we are halving the space search, thus the complexity is $O(\log n)$

Binary Search On Uncommon Data Structures - Problem

Problem description[3]: Given a weighted tree of up to N vertices ($N \leq 80K$) where vertex values increase from root to leaves on each level of the tree, find the ancestor vertex closest to the root from a starting vertex v that has weight at least P. Find these ancestors for Q queries ($Q \leq 20K$); each query contains a vertex v.

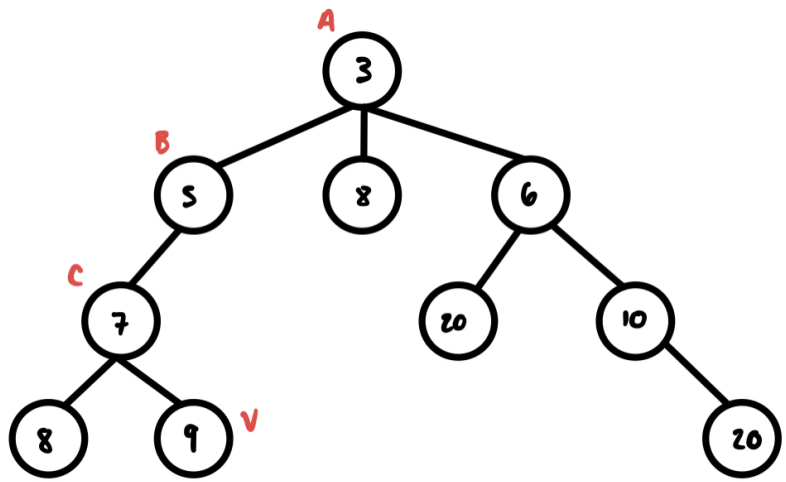


Figure: My Ancestor problem

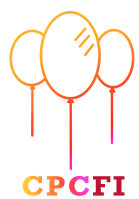
[3] 'My Ancestor' was used in the Thailand ICPC National Contest 2009

Binary Search On Uncommon Data Structures - Naive Solution

Naive Solution: perform linear scan per query in $O(N)$. Starting from a vertex v we move up in the tree until we reach the first vertex u whose direct parent has value $< P$ or until we reach the root. If vertex u has value $\geq P$ and $u \neq v$, we have found the solution.

Since there are Q queries, we have $O(QN)$ complexity which will get a TLE verdict since $80,000 \cdot 20,000 = 1,600,000,000$

*Recall that modern computers can run up to $100M$ operations per second.



Binary Search On Uncommon Data Structures - Better Solution

Better solution:

1. Store the Q queries[4]
2. Traverse the tree just once using preorder tree traversal and store the partial *root-to-current-vertex sequence*
3. When we land on a queried vertex v, we can perform a binary search in $O(\log N)$ to obtain the ancestor closest to the root with a value of at least P and store the solution
4. Perform a $O(Q)$ iteration to output the results; the overall complexity of this solution is $O(Q \log N)$

$$Q \log N = 20,000 \log(80,0000) \approx 98,000$$
$$98,000 << 1,600,000,000$$

[4] The important element to store is the v vertex from which we're going to start

Binary Search On the Bisection Method

- Binary Search principle can also be used to find the root of a function that may be difficult to compute directly
- The bisection method takes $O(\log_2(\frac{b-a}{\epsilon}))$ to get an answer that is good enough

Binary Search the Answer - Problem

Problem description[5]: Imagine that you are an explorer trying to cross a desert. You use a jeep with a large enough fuel tank – initially full. You encounter a series of events throughout your journey such as *drive*, *gas leak*, *encounter gas station*, *encounter a mechanic* or *reach your destination*. You need to determine the smallest possible fuel tank capacity for your jeep to be able to reach the goal. The answer must be precise to three digits after decimal point.

[5] UVa 11935

Binary Search the Answer - Solution

- From the problem description, the range of possible answers is between $[0, \ldots, 10000]$, with 3 digits of precision. However, there are $10M$ such possibilities. Trying each value sequentially will get us a TLE verdict
- However, this problem has a property we can exploit. Suppose the correct answer is X:
  - Setting the jeep's fuel capacity to any value between $[0, \ldots, X - 0.001]$ will not bring the jeep to the goal event
  - Setting the jeeps' fuel capacity to any value between $[X, \ldots, 10000]$ will bring the jeep to the goal event
- C++ code: `/DivideAndConquer/11935.cpp`

Divide and Conquer - In Programming Contests

- The most common use case of the Divide and Conquer paradigm, is the Binary Search Principle

Index

UVa - Online Judge

- Divide and Conquer problems: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=660
- If this link is not working, please check [Halim]: page 88 for DQ Problems

Table of Contents

Greedy - The Idea

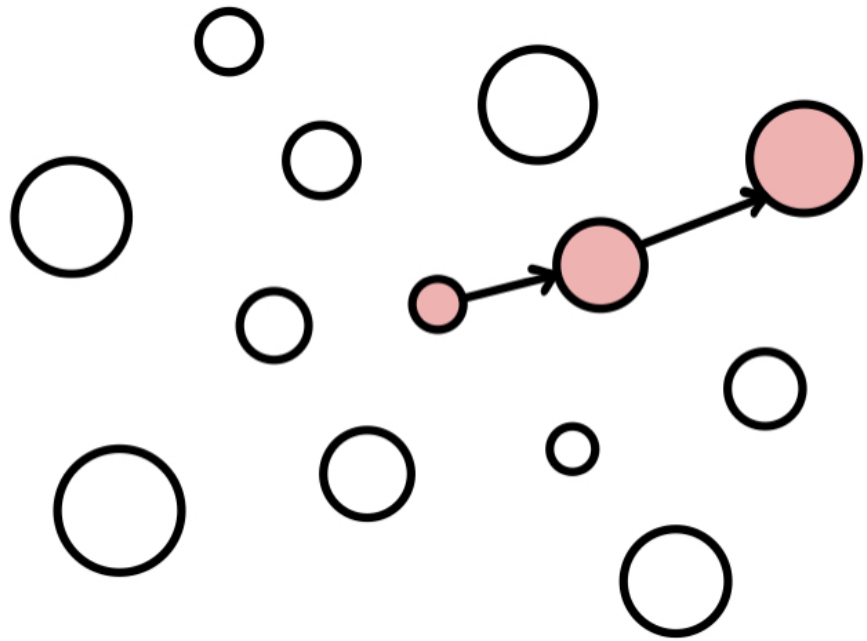


Figure: Choosing the best local option hoping to reach the best global solution

Greedy - Definition

An algorithm is said to be greedy if it makes the locally optimal choice at each step hoping to eventually reach the globally optimal solution. In some cases greedy works but in many others, it does not.

Note: finding a greedy solution is an art, just as finding good Complete Search solutions requires creativity !

Greedy - When does it work?

A problem must exhibit these two properties for a greedy approach to work:

1. It has optimal sub-structures
   - Optimal solution to the problem contains optimal solutions to the sub-problems
2. It has the greedy property
   - We reach the optimal solution if we make a choice that seems like the best at the moment and proceed to solve the remaining sub-problems

Greedy - Example #1: Coin Change

Problem description

Given a target amount V cents and a list of denominations of n coins, what is the minimum number of coins that we must use to represent the amount V? Assume that we have unlimited supply of coins of any type.

For example:

If $n = 4$, `coinValue = {25, 10, 5, 1}` cents[6] and $V = 42$, what is the minimum number of coins?

$25 + 10 + 5 + 1 + 1 = 42$, thus, **5 coins**

Algorithm: select the largest coin denomination which value is not greater that the remaining amount.

[6] The 1-cent coin ensures that we can always obtain any value V

Greedy - Example #1: Coin Change

Why did the past algorithm worked on this problem?

1. **It has optimal sub-structures**
   - 5-coin solution is optimal
   - Optimal solutions to the sub-problems are contained within the 5-coin solution

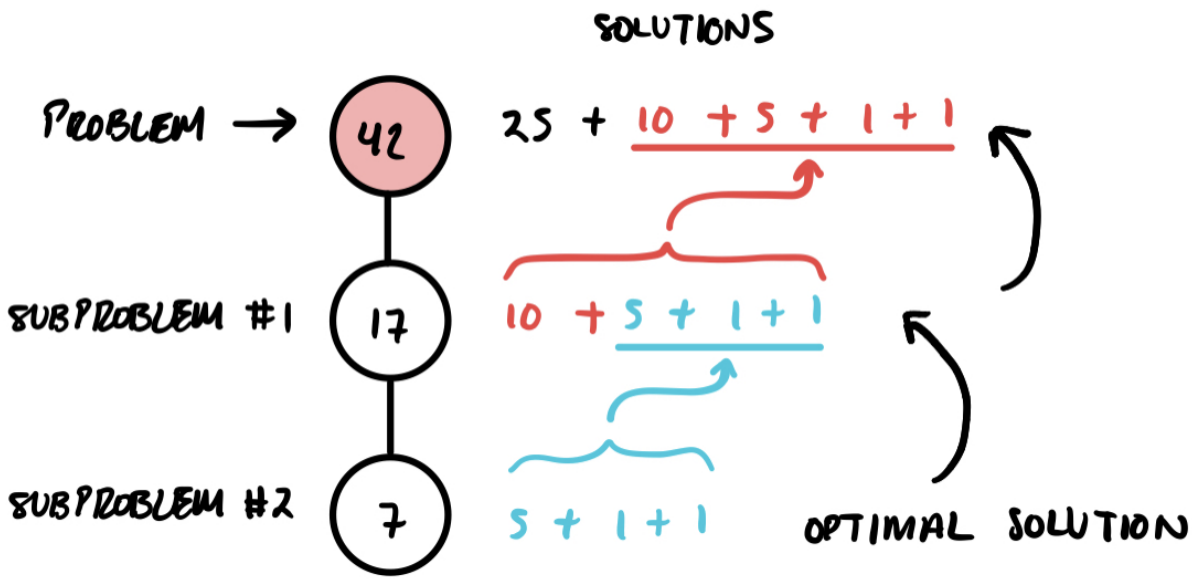


Figure: Optimal solutions to the sub-problems are contained within the global optimal solution

CPCFI

Greedy - Example #1: Coin Change

2. **It has the greedy property**
   - For this set of coin denominations (it can be proven) that substracting the largest coin denomination from every amount V that is not greater that the remaining amount, we will always obtain the minimum number of coins (optimal solution)

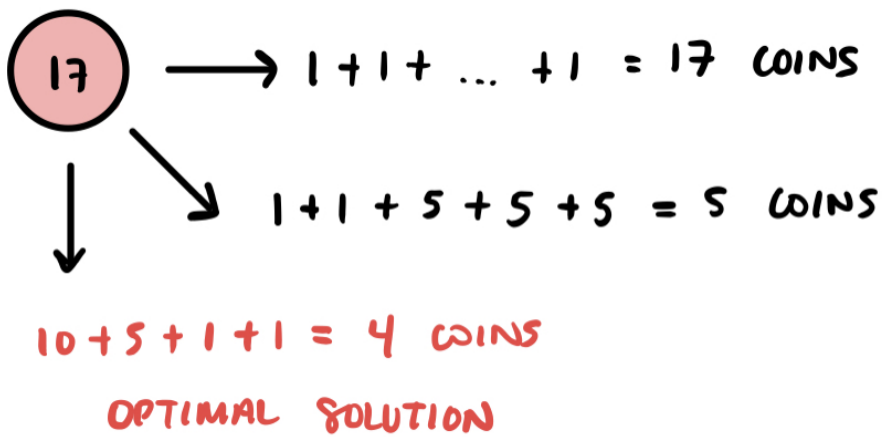


Figure: Substracting the largest coin denomination that is not greater that V will always result in the optimal solution for this set of coins: $\{25, 10, 5, 1\}$

CPCFI

Greedy - Example #1: Coin Change

- If we change the set of coins, then the same algorithm might not always result in the optimal solution
- For example, take the following set of coins: `{4,3,1}` and $V = 6$ cents
- Using the previous algorithm, we will obtain that the answer is $4 + 1 + 1$, **3 coins**
- Instead of the obvious optimal solution $3 + 3$, **2 coins**

* The general solution to this problem will be found in **Dynamic Programming** section

Greedy - Example #2: UVa 410 - Station Balance (Load Balancing)

Problem description

Given C chambers ($1 \leq C \leq 5$) which can store S specimens ($1 \leq S \leq 2C$) and a list of M masses (one mass value i per specimen S_i), determine which chamber should store each specimen in order to minimize **imbalance**.

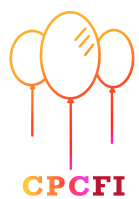
Average total mass in each chamber:

$$A = \frac{\sum_{i=1}^{S} M_i}{C}$$

Imbalance:

$$I = \sum_{j=1}^{C} |X_j - A|$$

is the sum of differences between the total mass in each chamber X_j with respect to the average total mass A



Greedy - Example #2: UVa 410 - Station Balance

Consider the following values:

- $C = 3$
- $S = 4$
- $M = \{5, 1, 2, 7\}$
- $A = \frac{5+1+2+7}{3} = 5$

Greedy - Example #2: UVa 410 - Station Balance

Some possible accomodations:

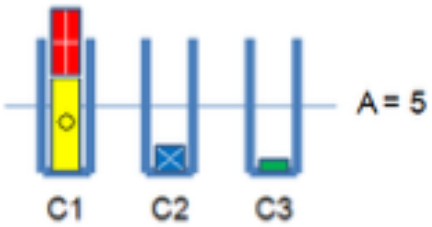


Figure: $I = 14$

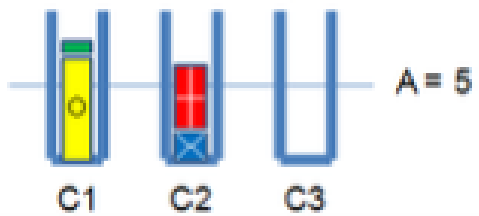


Figure: $I = 10$

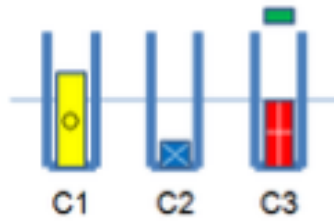


Figure: $I = 6$

- $M_{\text{yellow}} = 7$
- $M_{\text{red}} = 5$
- $M_{\text{blue}} = 2$
- $M_{\text{green}} = 1$

Greedy - Example #2: UVa 410 - Station Balance

- Observation 1: If there exists an empty chamber, it's usually beneficial since its worse to have an empty chamber ($I = A$) and its easy to move one specimen with mass X_j to the empty chamber and will have a less imbalance ($I = |A - X_j|$)
- **Observation 2**: If $S > C$, then $S - C$ specimens must be paired with a chamber already containing other specimens[7]

[7] Pigeonhole principle

Greedy - Example #2: UVa 410 - Station Balance

Therefore, the solution to the problem can be simplified by sorting:

1. If $S < 2C$, add $2C - S$ dummy specimens with mass 0
2. Sort the list M
3. Pair the specimens with masses M_1 and M_{2C} into C_1
4. Pair the specimens with masses M_2 and M_{2C-1} into C_2
5. $\vdots$

This algorithm is called: **load balancing**

Greedy - Example #2: UVa 410 - Station Balance

Returning to the original values:

- $C = 3$
- $S = 4$
- $M = \{5, 1, 2, 7\}$
- $A = 5$

Load balancing algorithm:

1. $M = \{5, 1, 2, 7, 0, 0\}$
2. $M = \{0, 0, 1, 2, 5, 7\}$
3. 0 and 7 goes in chamber C_1
4. 0 and 5 goes in chamber C_2
5. 1 and 2 goes in chamber C_3

Figure: Imbalance $I = 4$ is optimal

UVa 10382 - Watering Grass (Interval Covering)

Problem Description

n sprinklers ($n \leq 10000$) are installed in a horizontal strip of grass L meters long and W meters wide. Each sprinkler is centered vertically in the strip. For each sprinkler, we are given its position as the distance from the left end of the center line and its radius of operation. What is the minimum number of sprinklers that should be turned on in order to water the entire strip of grass?

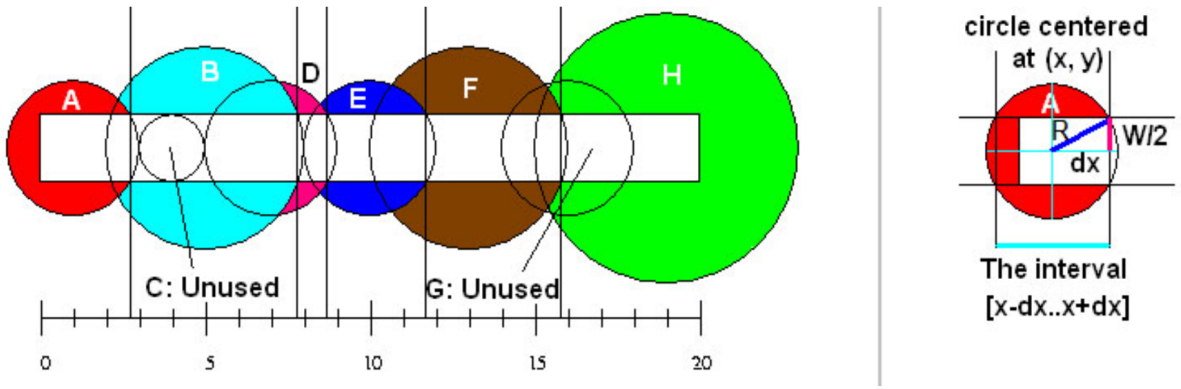
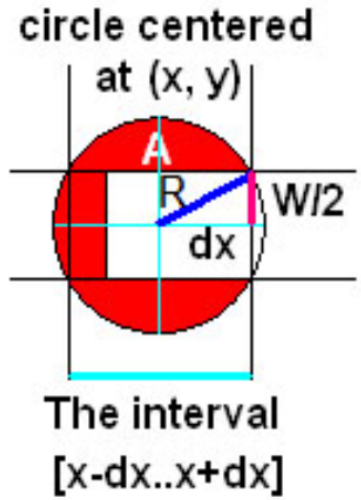


Figure: UVa 10382 - Watering Grass

UVa 10382 - Watering Grass

- This problem cannot be solved using brute force since $n \leq 10000$ and to solve it we would need to analyze 2^{10000} subsets
- This problem is a modification of the *interval covering problem*
- It's possible to transform the circles into the interval covering problem by computing $dx = \sqrt{R^2 - (W/2)^2}$



CPCFI

UVa 10382 - Watering Grass

Solution

1. Sort the intervals by increasing left endpoint and decreasing right endpoint if ties arise
2. Process the intervals one at a time, taking the interval with the largest right endpoint

*This will produce uninterrupted coverage from the leftmost side to the rightmost side of the horizontal strip of grass, therefore, the minimum number of sprinklers

UVa 11292 - Dragon of Lootwater (Sort the Input First)

Problem description

There are n dragon heads with a specified *diameter* D and m knights with a specified *height* H ($1 \leq n, m \leq 20000$). A knight can only chop **one** dragon head iff $H \geq D$. Given a list of heights and a list of diameters:

- Is it possible to chop off all the dragon heads?
- If so, what is the minimum total height of the knights used to chop off the dragons' heads?

UVa 11292 - Dragon of Lootwater

Solution

- This could be thought as a bipartite matching problem
- However, this problem can also be solved by sorting the input
  - Each dragon head should be chopped by a knight with the shortest height that is at least as tall as the diameter of the dragon's head
  - Since the input is given in arbitrary order, we could sort both the list of dragon head diameters and knight heights in $O(n \log n + m \log m)$
  - Then, we could use the following $O(\min(n, m))$ scan to determine the correct answer

`C++ code: /Greedy/11292.cpp`

Greedy in Programming Contests

- For the classical greedy problems (Coin Change, Load Balancing and Interval Covering), it's helpful to memorize their solutionos
- When dealing with a greedy problem, it's usually helpful to sort data to highlight hidden greedy strategies
- There are two other classical examples of Greedy algorithms:
  1. Kruskal's (and Prim's) algorithm for the Minimum Spanning Tree (MST)
  2. Dijkstra's algorithm for the Single-Source Shortest Paths (SSSP)
- A Greedy algorithm won't - normally - encounter TLE verdict, but is more prone to encountering WA verdict
- Rule of thumb:
  - If the input size is small enough, use either a Complete Search or Dynamic Programming (DP) approach
  - Otherwise, use the Greedy approach

Index

UVa - Online Judge

- Greedy problems: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=656
- If site is not working, please refer to [Halim]: page 93 and 94 for Greedy problems

Table of Contents

More Advanced Search Techniques

We have already seen iterative and recursive backtracking Complete Search techniques. However, some problems require other search techniques to avoid the TLE[8] verdict.

[8] Time Limit Exceeded

Index

Backtracking wit Bitmask

Recall that a bitmask can be used to model a small set of Boolean. Bitmask operations are very lightweight and we could use them to speed up a Complete Search solution.

UVa 11195 - Another n-Queen Problem

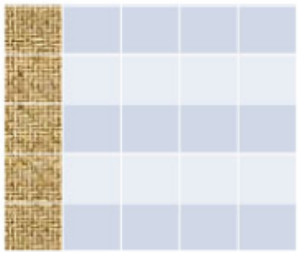
- UVa 11195 would still get a TLE even when using the three bitsets `rw, ld, rd` from the previous formulation (section 3.2 - Recursive Complete Search)
- First, let's use bitmasks instead of bitsets for representing the three sets of Booleans
- These bitmasks represent which rows are attacked (or, not available to place another queen) in the next column, left and right diagonal from previously placed queens

UVa 11195 - Another n-Queen Problem

```
1  int ans = 0, OK = (1 << 5) - 1;
2  void backtrack(int rw, int ld, int rd) {
3    // if all bits in rw are on
4    if (rw == OK) { ans++; return; }
5    // the ones in pos mean availability
6    int pos = OK & (~(rw | ld | rd));
7    while (pos) {
8      // p will be the Least Significant One
9      int p = pos & -pos;
10     // turn off the LSO
11     pos -= p;
12     backtrack(rw | p, (ld | p) << 1, (rd | p) >> 1);
13   }
14 }
15 int main() {
16   backtrack(0,0,0);
17   printf("\%d\n", ans);
18 }
```

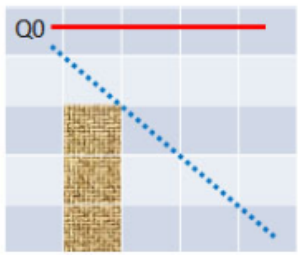
UVa 11195 - Another n-Queen Problem

Let's see how this works:



```
pos = 11111 & ~00000 = 11111 (p = 1)
```

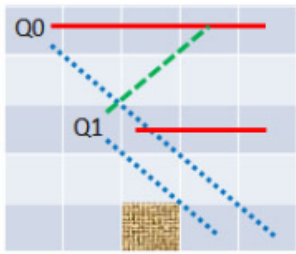
Figure: Initial state




```
rw                   = 00001 (1)
ld = 00001 << 1 = 00010 (2)
rd = 00001 >> 1 = 00000|1 (0, the LSB is removed)
---------------------- OR
                     00011 -> NEGATE -> 11100
pos = 11111 & 11100 = 11100 (p = 4)
```

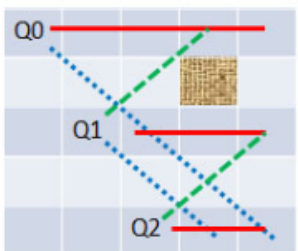
Figure: After positioning the first queen, **pos** only reflects the three available spots for the second queen in the second column

UVa 11195 - Another n-Queen Problem



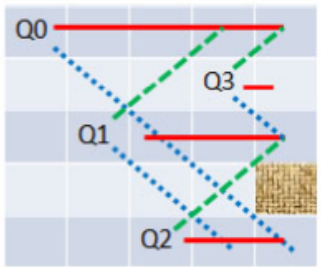

```
rw                  = 00101 (5)
ld = 00110 << 1 = 01100 (12)
rd = 00100 >> 1 = 00010 (2)
------------------------ OR
                      01111 -> NEGATE -> 10000
pos = 11111 & 10000 = 10000 (p = 16)
```

Figure: After positioning the second queen, **ld** reflects this change by turning on bits 2 and 3 (which corresponds to row 2 and 3 respectively). The same thing happens with **rd** but with row 1

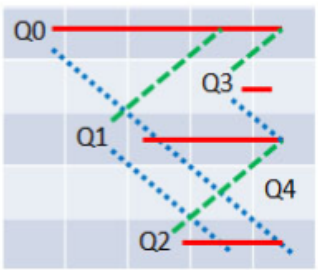



```
rw                  =  10101 (21)
ld = 11100 << 1 = 111000 (56, but the MSB is unused)
rd = 10010 >> 1 =  01001 (9)
------------------------ OR
                    111101 -> NEGATE -> 000010
pos = 11111 & 000010 = 00010 (p = 2)
```

UVa 11195 - Another n-Queen Problem




```
rw                    =    10111 (23)
ld = 111010 << 1 = 1110100 (116, two MSBs are unused)
rd =  01011 >> 1 =   00101 (5)
--------------------------- OR
                     1110111 -> NEGATE -> 0001000
pos = 11111 & 0001000 = 01000 (p = 8)
```




```
rw                    =    11111 (we get one solution)
```

Figure: Final solution when **rw** has all five bits turned on

Index

8.2.2 Backtracking with Heavy Prunning

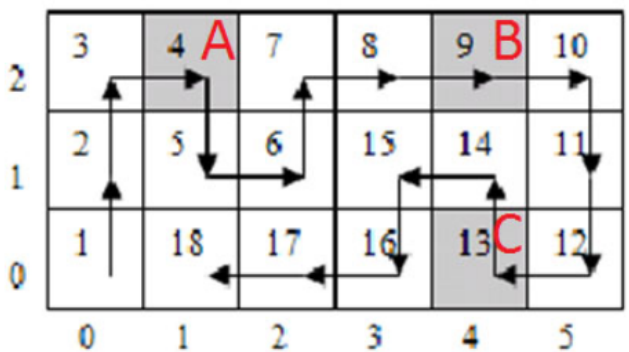
Problem description

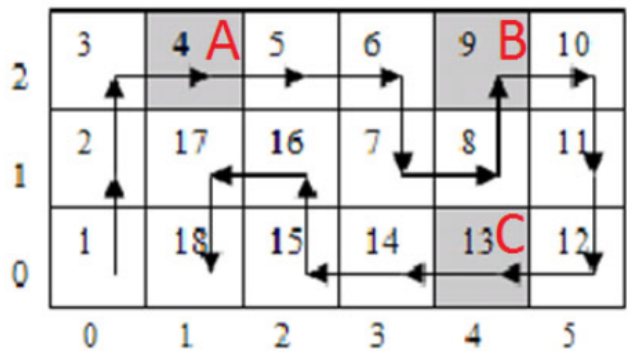
Given an $M \times N$ board with 3 check-in points $\{A, B, C\}$, find a Hamiltonian path[9] of length $(M \times N)$ from coordinate $(0, 0)$ to coordinate $(0, 1)$. This Hamiltonian path must hit the three check points: A, B, and C at one-quarter, one-half, and three-quarters of the way through its path, respectively. Constraints: $2 \leq M, N \leq 8$.

[9] Path in an undirected graph that visits each vertex exactly once

Backtracking with Heavy Prunning

For example, 3×6 board with $A = (2, 1)$, $B = (2, 4)$ and $C = (0, 4)$, we have two options:





Backtracking with Heavy Prunning

Naive solution: try with all paths and exit when found a Hamiltonian path. To implement this solution, we would need - in the worst case - to analyze $4^{64} = 3.4 \times 10^{38}$ paths, which is simply unfeasible.

Backtracking with Heavy Prunning

Heavy Prunning: prune the search space if the search:

1. Wanders outside the $M \times N$ grid
2. Does not hit the appropriate target check point at one-quarter, one-half and three-quarters distace
3. Hits target check point earlier than the target distance
4. Will not be able to hit certain coordinates as the current partial path self-blocks the access to those coordinates. This can be checked with a simple DFS/BFS starting from coordinates $(0, 1)$ and determine if there are coordinates in the grid that are not reachable from $(0, 1)$ and not yet visited by the current partial path, we can prune the current partial path

Index

UVa - Online Judge

- More Advanced Search problems: https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=765
- If site is down, please refer to [Halim]: page 310 and 311 for this section's problems

References

- Halim S., Halim F., *Competitive Programming 3*, Handbook for ACM ICPC and IOI Contestants. 2013
- Skiena S. *The Algorithm Design Manual*. Springer. 2020
- Martínez L. *Apuntes de Matemáticas Discretas y Algoritmos*. Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas. 2020

CPCFI