Stony Brook Puzzle Society

Stony Brook Puzzle Society

Classic Puzzles

Dedicated to
the thrill and joy of
solving classic puzzles

Preface

Drug addicts get to the highest level of infatuation by injecting the drugs into their blood. Puzzle addicts get to the highest level of the two feelings thrill and joy, by injecting challenging puzzles and problems into their brain.

The major benefits are the eternal and heavenly feelings of thrill and joy. Other benefits include:

- The only input to puzzles is thinking, the output given by puzzles are happiness, wisdom, and satisfaction.
- The art of thinking what we learn from the solutions of these puzzles helps us to tackle real life problems in a brilliant and beautiful way.
- Puzzles can be our only best friend when we are sad, feeling lonely, traveling, and waiting.
- Puzzles help in making friends, as I used to start talking to new beautiful girls by asking challenging puzzles and the next day they would run behind me for solutions. I would give the solution and ask another tough problem. The process would continue indefinitely.
- Many of these puzzles are proper interview questions of elite companies like Microsoft, and Google. So, learning these puzzles gives us a solid training to attack general puzzles.

The general way to solve a puzzle is summarized in the algorithm. Go through the algorithm very carefully and follow the steps given there. The solution to every puzzle will have a core idea. If we get that idea, we can solve the puzzle easily. Our aim is to get that one idea which is the crux of the solution through time tested beautiful techniques and practice.

Our aim should not be solving a puzzle. Our aim should be nailing a puzzle. In the latter, we are not interested in just one way of solving the puzzle but multiple ways of attacking the problem and possibly creating many more new problems and recursively nailing them.

Algorithm : NailingAPuzzle(Puzzle)
Input: An interesting puzzle
Output: Different solutions to the puzzle and a couple of new problems

Phase I: Pre-Solving Phase
Read the puzzle completely multiple times.
Understand the puzzle perfectly.
Write down carefully what is given, what is not given and what is to be computed.

Phase II: Solving Phase
Find the domain of the problem like probability, logic, divide-and-conquer, geometry, calculus, dynamic programming, graph theory, etc.
Relate the problem to a previous known concept (or idea) in the problem domain.
Solve the puzzle for smaller instances. Many a times we can find a general formula from the pattern.

Draw neat diagrams, graphs, tables, charts, etc wherever possible.
Write your thoughts on paper (or board) instead of trying to solve the puzzle completely in mind.
Discuss the problem with friends, classmates, teachers and anyone.
Write computer programs and analyze the results.
Try to find that one simple idea on which the entire solution rests upon.
Think, think, think and solve.

Phase III: Post-Solving Phase
Extensively test and verify whether the solution is flawless.
Generalize the solution so that it works for any instance of the puzzle.
Once the solution works, unlearn it, go to Phase II and think of a different way of solving the puzzle.
Analyze different solutions from books, Internet, discussions etc.
If bored of all solutions, go ahead and create new related puzzles.
At the point of saturation, leave the puzzle and take a new one.

return [Solutions NewProblems].

Tis a lesson you should heed,
Try, try, try again.
If at first you don't succeed,
Try, try, try again.

If you have any comments, suggestions, advices, criticisms, new problems, alternate solutions, logical errors, typo errors or literally anything, feel free to discuss them on the Facebook group. Cherish life with puzzles. Enjoy!

February 7, 2014

Pramod Ganapathi
Stony Brook Puzzle Society

Contents

Problems & Solutions

k^{th} smallest element in a sorted matrix.

Find the k^{th} smallest element in a $m \times n$ sorted matrix. A sorted matrix is a matrix with sorted rows and columns.

Solution.

The algorithms summary is given in the following table.

No	Algorithms	Complexity
1	Naive linear sorting method	$O(mn \cdot \log(mn))$
2	Naive merging method	$O(mn \cdot \min(m,n))$
3	Improved merging method	$O(k \cdot \min(m,n))$
4	Improved merging using heap method	$O(k \cdot \log(k))$
5	Succinct description method	$O(k)$

Table 1: k^{th} smallest element in sorted matrix algorithms.

The following algorithms can be tweaked slightly to get the above complexities.

Naive linear sorting method.

Algorithm : NAIVELINEARSORTING$(a[1\ldots m][1\ldots n], k)$ $O(mn \cdot \log(mn))$

Input: Sorted matrix a of size $m \times n$
Output: k^{th} smallest element in the matrix a

1: Convert matrix $a[1\ldots m][1\ldots n]$ to array $b[1\ldots mn]$	$O(mn)$
2: Sort the array $b[1\ldots mn]$	$O(mn \cdot \log(mn))$
3: **return** $b[k]$	$O(1)$

Naive merging method.

Algorithm : NAIVEMERGING$(a[1\ldots m][1\ldots n], k)$ $O(mn \cdot m)$

Input: Sorted matrix a of size $m \times n$
Output: k^{th} smallest element in the matrix a

1: Denote the rows of the matrix $a[1\ldots m][1\ldots n]$ as $l_1, l_2 \ldots, l_m$	
2: Merge the arrays $l_1, l_2 \ldots, l_m$ to array $b[1\ldots mn]$	$O(mn \cdot m)$
3: **return** $b[k]$	$O(1)$

Improved merging method.

Algorithm : ImprovedMerging$(a[1\ldots m][1\ldots n], k)$ — $O(k \cdot m)$

Input: Sorted matrix a of size $m \times n$
Output: k^{th} smallest element in the matrix a

1: Denote the rows of the matrix $a[1\ldots m][1\ldots n]$ as $l_1, l_2 \ldots, l_m$
2: Merge the arrays $l_1, l_2 \ldots, l_m$ until k elements to array $b[1\ldots k]$ — $O(k \cdot m)$
3: **return** $b[k]$ — $O(1)$

Improved merging using heap method.

Algorithm : ImprovedMergingUsingHeap$(a[1\ldots m][1\ldots n], k)$ — $O(k \log k)$ or $O(k \cdot \log m)$

Input: Sorted matrix a of size $m \times n$
Output: k^{th} smallest element in the matrix a

1: Denote the rows of the matrix $a[1\ldots m][1\ldots n]$ as $l_1, l_2 \ldots, l_m$
2: **if** $k < m$ **then**
3: Create a heap of k elements from first k rows — $O(k)$
4: Merge the first k rows until k elements are popped — $O(k \log k)$
5: **else**
6: Create a heap of m elements from all rows — $O(m)$
7: Merge all the rows until k elements are popped — $O(k \log m)$
8: **return** k^{th} popped element from the heap

Succinct description method.

Read the paper *Generalized selection and ranking: sorted matrices* by Greg N Frederickson and Donald B Johnson. They give a method that uses a succinct representation of the matrix to give an $O(k)$ algorithm.

Exercise.

1. How can we solve the problem in 3-D or d-D?
2. (Andrey Gorlin) Can we find the k^{th} smallest element without enumerating all k smallest elements?

Search in a sorted matrix.

Search for an element k in a $m \times n$ sorted matrix. A sorted matrix is a matrix with sorted rows and columns.

Solution.

The algorithms summary. For more detailed info, refer
`http://leetcode.com/2010/10/searching-2d-sorted-matrix.html`.

No	Algorithms	Complexity
1	Linear search	$O(mn)$
2	Binary search	$O(\min(m,n) \cdot \log(\max(m,n)))$
3	Diagonal binary search	$O(\log(n!))$
4	Quad partition	$O\left((\min(m,n))^{\log 3}\right)$
5	Improved quad partition	$O(m+n)$
6	Left down search	$O(m+n)$

Table 2: Search in sorted matrix algorithms.

Comment. Diagonal binary search works only for square matrices of size $n \times n$.

Linear search.

Algorithm : LinearSearch$(a[1\ldots m][1\ldots n], k)$ $O(mn)$

Input: Sorted matrix a of size $m \times n$
Output: Whether the search element k exists or not

1: Linear search for the element k in matrix $a[1\ldots m][1\ldots n]$ $O(mn)$

Binary search.

Algorithm : BinarySearch$(a[1\ldots m][1\ldots n], k)$ $O(m \log n)$

Input: Sorted matrix a of size $m \times n$
Output: Whether the search element k exists or not

1: **for** each of the m rows in the sorted matrix **do**
2: Binary search for the element k in the row $O(\log n)$

Diagonal binary search.

This algorithm works only for square matrices of size $n \times n$.

Observations.

1. In a sorted matrix, the primary diagonal elements are always sorted and the secondary diagonal elements may be unsorted.
2. If the matrix is a square matrix of size $n \times n$, then $a[i][i]$, where $i \in [1,n]$ are the primary diagonal elements. If the matrix of size $m \times n$ is not a square matrix, then $a[i][i]$ might not be the primary diagonal elements. A similar argument holds for secondary diagonal.

Algorithm : DiagonalBinarySearch$(a[1\ldots n][1\ldots n], k)$ $O(\log(n!))$

Input: Sorted matrix a of size $n \times n$
Output: Whether the search element k exists or not
Require: Works only for square matrices of size $n \times n$

1: **for** each of the secondary diagonal element, say $a[i][j]$, from top right to bottom left **do**
2: **if** $k = a[i][j]$ **then**
3: **return true**
4: **else if** $k < a[i][j]$ **then**
5: Binary search for k in the row $a[i][1\ldots j-1]$ $O(\log n)$
6: **else**
7: Binary search for k in the column $a[i+1\ldots n][j]$ $O(\log n)$

Quad partition.

The recurrence relation is $T(n) = 3T(n/2) + O(1)$, which leads to $T(n) = O\left(n^{\log 3}\right)$.

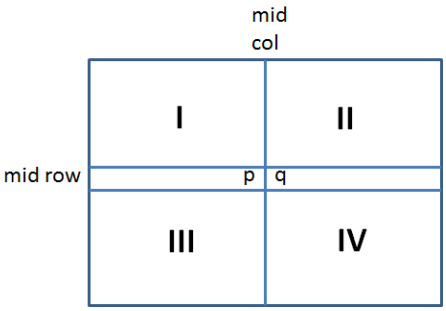

Figure 1: Quad partition

Algorithm : QuadPartition$(a[1\ldots m][1\ldots n],\ k)$ $O\left(\min(m,n)^{\log 3}\right)$

Input: Sorted matrix a of size $m \times n$
Output: Whether the search element k exists or not

```
1: Split the matrix into four almost identical quadrants I, II, III, IV
2: Let the elements across the split in the middle row be p and q
3: if k < p then
4:     QuadPartition(Quadrant I, k)
5:     QuadPartition(Quadrant II, k)
6:     QuadPartition(Quadrant III, k)
7: else if k > q then
8:     QuadPartition(Quadrant II, k)
9:     QuadPartition(Quadrant III, k)
10:    QuadPartition(Quadrant IV, k)
11: else
12:    QuadPartition(Quadrant II, k)
13:    QuadPartition(Quadrant III, k)
```

Improved quad partition.

The recurrence relation is $T(n) = 2T(n/2) + O(\log n)$, which leads to $T(n) = O(n)$.

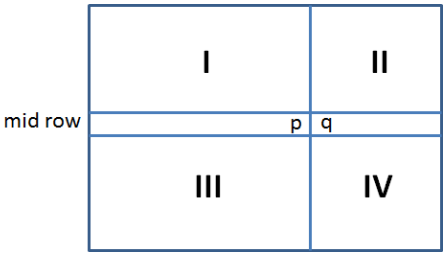

Figure 2: Improved quad partition

Algorithm : ImprovedQuadPartition$(a[1\ldots m][1\ldots n], k)$ $O(m+n)$

Input: Sorted matrix a of size $m \times n$
Output: Whether the search element k exists or not

```
1: Binary search for k in the middle row
2: if k is found then
3:     return true
4: else
5:     Let elements after binary search be p and q such that p < k < q
6:     Name the four quadrants as I, II, III, IV
7:     ImprovedQuadPartition(Quadrant II, k)
8:     ImprovedQuadPartition(Quadrant III, k)
```

Left down search.

Algorithm : LeftDownSearch$(a[1\ldots m][1\ldots n], k)$ $O(m+n)$

Input: Sorted matrix a of size $m \times n$
Output: Whether the search element k exists or not

```
1: row ← 1, col ← n {start from the top right element}
2: while row ≤ n and col ≥ 1 and a[i][j]! = k do
3:     if k < a[i][j] then
4:         j ← j − 1 {go left}
5:     else
6:         i ← i + 1 {go down}
```

Exercise.

1. How can we search for an element in a sorted matrix in 3-D or d-D?

Random Maze Generation.

Generate a random maze of order $m \times n$.

Solution.

The algorithms summary. For detailed explanations, refer
`http://weblog.jamisbuck.org/2011/2/7/maze-generation-algorithm-recap`.

No	Algorithms	Complexity
1	Recursive backtracking	–
2	Kruskal's algorithm	$O(mn)$
3	Prim's algorithm	$O(mn)$
4	Recursive division	$O(mn)$
5	Binary tree algorithm	$O(mn)$
6	Sidewinder algorithm	$O(mn)$
7	Growing tree algorithm	$O(mn)$
8	Hunt and kill algorithm	$O\left((mn)^2\right)$
9	Wilson algorithm	$O(mn)$
10	Aldous Broder algorithm	$O(mn)$
11	Eller algorithm	$O(mn)$

Table 3: Random maze generation algorithms.

Recursive backtracking.

Algorithm : RecursiveBacktracking()

Input: Size $m \times n$

Output: Maze matrix with paths between the cells

```
Mark all cells as unvisited
Choose a random starting point
Mark the starting point as visited
Mark the starting point as current cell
Until the algorithm ends at starting point
repeat
    if all neighbors of current cell are visited then
        Go back to previous cell
    else
        Randomly choose an unvisited neighbor of current cell and mark it as current cell
until the algorithm ends at starting point
```

Kruskal's algorithm.

Algorithm : KruskalAlgorithm() $O(mn)$

Input: Size $m \times n$

Output: Maze matrix with paths between the cells

```
Make a set of a all edges
while edge set is non empty do
    Pull out an edge at random
    if edge connects two disjoint trees then
        Join the trees
    else
        Discard the edge
```

Prim's algorithm.

Algorithm : PrimAlgorithm() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

1: Choose an arbitrary starting cell and add it to a tree
2: $m \leftarrow$ number of cells
3: **for** $(m-1)$ times **do**
4: Choose an edge randomly that connects a vertex in the tree to a vertex not in the tree
5: Add that edge to the trees

Recursive division.

Algorithm : RecursiveDivision() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

1: Begin with an empty field
2: **while** the maze has not reached the desired dimension **do**
3: Add a wall somewhere, either horizontally or vertically
4: Add a single passage through the wall
5: Recursively solve the two sub-problems

Binary tree algorithm.

Algorithm : BinaryTreeAlgorithm() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

1: **for** each cell in the grid **do**
2: Randomly carve a passage either north or east

Sidewinder algorithm.

Algorithm : SidewinderAlgorithm() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

1: *set* $\leftarrow$ empty
2: Work through the grid row-wise
3: Start with the first cell
4: **repeat**
5: **repeat**
6: *set* $\leftarrow$ *set* + current cell
7: for the current cell, randomly decide whether to carve east or not
8: if a passage was carved, make the new cell the current cell
9: **until** you decide not to carve east or in the last column
10: choose any one of the cells in the set and carve a passage north
11: *set* $\leftarrow$ empty
12: current cell $\leftarrow$ next cell
13: **until** you finish off all rows

Growing tree algorithm.

Algorithm : GrowingTreeAlgorithm() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

```
1: let set be a list of cells
2: set ← a random cell from the grid
3: repeat
4:     Choose a cell from set
5:     if the cell has unvisited neighbors then
6:         set ← set + a random unvisited neighbor
7:         Carve passage to that unvisited neighbor
8:         Mark that neighbor as visited
9:     else
10:        Remove the cell from set
11: until set is empty
```

Hunt and kill algorithm.

Algorithm : HuntAndKillAlgorithm() $O\left((mn)^2\right)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

```
1: choose a starting cell
2: while there are unvisited cells scanned in hunt mode do
3:     Perform a random walk, carving passages to unvisited neighbors, until the current cell has no
       unvisited neighbors
4:     Enter hunt mode, where you scan the grid looking for an unvisited cell that is adjacent to a
       visited cell
5:     If found, carve a passage between the two and let the formerly unvisited cell be the new starting
       location
```

Wilson algorithm.

Algorithm : WilsonAlgorithm() $O(mn)$

Input: Size $m \times n$
Output: Maze matrix with paths between the cells

```
1: Choose any vertex at random and add it to the uniform spanning tree (UST).
2: repeat
3:     Select any vertex that is not already in the UST and perform a random walk until you encounter
       a vertex that is in the UST.
4:     Add the vertices and edges touched in the random walk to the UST.
5: until all vertices have been added to the UST.
```

Aldous Broder algorithm.

Algorithm : AldousBroderAlgorithm() $O(mn)$

Input: Size $m \times n$

Output: Maze matrix with paths between the cells

1: Choose a vertex. Any vertex.
2: **repeat**
3: Choose a connected neighbor of the vertex and travel to it. If the neighbor has not yet been visited, add the traveled edge to the spanning tree.
4: **until** all vertexes have been visited.

Eller's algorithm.

Algorithm : EllerAlgorithm() $O(mn)$

Input: Size $m \times n$

Output: Maze matrix with paths between the cells

1: Initialize the cells of the first row to each exist in their own set.
2: **repeat**
3: Now, randomly join adjacent cells, but only if they are not in the same set. When joining adjacent cells, merge the cells of both sets into a single set, indicating that all cells in both sets are now connected (there is a path that connects any two cells in the set).
4: For each set, randomly create vertical connections downward to the next row. Each remaining set must have at least one vertical connection. The cells in the next row thus connected must share the set of the cell above them.
5: Flesh out the next row by putting any remaining cells into their own sets.
6: **until** the last row is reached.
7: For the last row, join all adjacent cells that do not share a set, and omit the vertical connections, and youâĂŹre done!

Find the recurrence.

[Pramod Ganapathi] The recurrence

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

has a complexity $O(\log n)$. *The recurrence*

$$T(n) = T\left(\sqrt{n}\right) + 1$$

has a complexity $O(\log \log n)$.

1. *Which recurrence has a complexity of* $O(\log \log \log n)$?
2. *Generalizing the problem, which recurrence has a complexity of* $O\left(\underbrace{\log \log \log \ldots \log}_{k} n\right)$?

Solution.

[Sourabh Daptardar] Answers below. The solution is left as an exercise to the reader.

(1) The recurrence

$$T(n) = T\left(2^{\sqrt{\log n}}\right) + 1$$

has the complexity of $O(\log\log\log n)$.

(2) Let,

$$f(k,l) = 2^{2^{2^{\cdots^{2^l}}}} \qquad (2 \text{ is repeated } k \geq 1 \text{ times})$$
$$g(k,l) = \underbrace{\log\log\log\cdots\log}_{k} l$$

then the recurrence

$$T(n) = T\left(f\left(k-1, \frac{g(k-1,n)}{2}\right)\right) + 1$$

has the complexity of $O\left(\underbrace{\log\log\log\ldots\log}_{k} n\right)$.

Game of Life.

George Conway, a mathematician invented a game called Game of Life in 1970. It is a 0-player game. Try to understand the game and implement it.

Solution.

Algorithm : GameOfLife()

Input: Size $m \times n$ grid, rules
Output: Simulation of the game

1: Each cell can have at most eight alive neighbors (four orthogonal and four diagonal).
2: Initially, some cells are alive and the rest are dead.
3: To create each generation, the following genetic rules are applied to all the cells simultaneously.
4: **for** each cell c of the grid **do**
5: **if** if c is alive and has < 2 neighbors **then**
6: it dies due to under-population.
7: **else if** c is alive and has 2 or 3 neighbors **then**
8: it lives.
9: **else if** c is alive and has > 3 neighbors **then**
10: it dies due to over-population.
11: **else if** c is dead and has exactly 3 neighbors **then**
12: it becomes alive.

Day of the Week.

Find the day of the week on a given date in Gregorian calendar, the calendar we use now.

Solution.

There are many algorithms to solve the problem and they majorly use the concept of odd days, the number of days past the complete weeks. The following solution appeared in the book *Quantitative Aptitude* by R S Aggarwal.

Algorithm : DayOfTheWeek(d, m, y)

Input: d date, m month, y year.
Output: Day of the week on the given date in Gregorian calendar.

1: // Most commonly used variables
2: $years = y - 1$
3: $fourcenturies = years/400$
4: $years = years - 400 \times fourcenturies$
5: $centuries = years/100$
6: $years = years - 100 \times centuries$
7: $leapyears = years/4$
8: $normalyears = years - leapyears$
9: // Initialize the number of odd days for different months depending on whether the year is a leap year or a normal year
10: $OD\_monthsnormal[0 \ldots 12] = \{0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5, 1\}$
11: $OD\_monthsleap[0 \ldots 12] = \{0, 3, 4, 0, 2, 5, 0, 3, 6, 1, 4, 6, 2\}$
12: $dayoftheweek[0 \ldots 6] = \{$Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday$\}$
13: // Find the odd days in $y - 1$ years, $m - 1$ months and d days
14: $OD\_years = (fourcenturies \times 0 + centuries \times 5 + leapyears \times 2 + normalyears \times 1)\%7$
15: **if** IsLeapYear(y) **then**
16: $\quad OD\_months = OD\_monthsleap[m - 1]$
17: **else**
18: $\quad OD\_months = OD\_monthsnormal[m - 1]$
19: $OD\_days = d\%7$
20: // Compute the total number of odd days
21: $OD = (OD\_years + OD\_months + OD\_days)\%7$
22: The day of the week on the given date is $dayoftheweek[OD]$

Algorithm : IsLeapYear(y)

Input: y year.
Output: True if y is a leap year, else not, as per Gregorian calendar.

1: **if** $y\%400 = 0$ or $(y\%100 \neq 0$ and $y\%4 = 0)$ **then**
2: $\quad$ **return true**
3: **else**
4: $\quad$ **return false**

We discussed about other questions about time and calendar.
(1) How did ancients measure an year?
Ans: Seasons.
(2) How did they come up with a month of approx 30 days?
Ans 1: Full moon to new moon and new moon to full moon is a month.
Ans 2: Change in the constellations.
(3) How did they come up with a week of 7 days?
Ans 1: Majorly due to religious reasons. Babylonians used 7 days (6 days of work and 1 day rest).
Ans 2: They are named based on Sun, Moon and five planets that are visible to naked eye.
(4) How did they come up with a day?

Ans: Day and night.
(5) How did they come up with a hour, min, sec?
Think about it...

People in every civilization tried to come up with calendars to measure time. There are more than 30 calendars in the world. Here are some of them: Gregorian, ISO, Egyptian, Julian, Coptic, Ethiopic, Islamic, Persian, Bahai, Hebrew, Mayan, Aztec, Balinese Pawukon, French Revolutionary, Chinese, Japanese, Korean, Vietnamese, Old Hindu, Hindu, Hindu Astronomical, Tibetan.

Refer the following book for more details *Calendrical Calculations* by Nachum Dershowitz and Edward M. Reingold.

---

100 Prisoners and 100 Boxes.

A room has 100 boxes labeled 1 through 100. The numbers of 100 prisoners from 1 to 100 have been placed in these boxes by the warden in a random order. The prisoners shall visit the room one by one. Each prisoner is allowed to inspect the contents of at most 50 boxes, one after the other and leave the room with no communication with other prisoners. If the prisoner discovers his own number in the max 50 boxes he inspects, he is released. If at least one of the 100 prisoners fail to identify their own names in the max 50 boxes they inspect, they will all be executed.
The prisoners are allowed to collude before hand and devise a strategy to maximize their survival (each prisoner should find his number in the boxes he inspects). What is their strategy?

Solution.

The problem appeared in Gurmeet Singh Manku's website.
`http://gurmeet.net/puzzles/100-prisoners-and-100-boxes/`
The detailed solution can be found here by Peter Taylor.
`http://www.mast.queensu.ca/~peter/inprocess/prisoners.pdf`

Algorithm : PrisonersAndBoxes()
Input: *maxprisoners*, the number of prisoners
Output: Strategy to maximize the probability of survival of the prisoners

```
1:  maxprisoners ← 100
2:  maxboxes ← maxprisoners/2
3:  box[1...maxprisoners] ← RandomPermutation(1...maxprisoners)
4:  for i ← 1 to maxprisoners do
5:      flagprisoner ← 0
6:      nextbox ← i
7:      for j ← 1 to maxboxes do
8:          Open nextbox
9:          if nextbox has prisoner number i then
10:             Prisoner has found his number
11:             flagprisoner ← 1
12:             Prisoner exits
13:         nextbox ← box[nextbox]
14:     if flagprisoner = 0 then
15:         All prisoners are executed                    < 60% probability
16: All prisoners are freed                               > 30% probability
```

Change Making.

We have n coins of denominations $c_1, c_2, \ldots, c_n$. We need to make a change for m amounts of money in minimum number of coins. We need to find the number of coins of each denomination that makes a change for m. There are four versions of the problem.

1. *Assume unlimited number of coins of each denomination. Use greedy approach.*
2. *Assume unlimited number of coins of each denomination. Use dynamic programming.*
3. *Assume 1 coin of each denomination. Use dynamic programming. This is called as 01 change making problem.*
4. *Assume limited number of coins of each denomination, say $k_1, k_2, \ldots, k_n$. Use dynamic programming.*

Solution.

The solid algorithm for limited number of coins is given in
`http://www.or.deis.unibo.it/kp/Chapter3.pdf`

Unlimited Greedy.

Algorithm : ChangeMaking-Unlimited-Greedy$(n, m, c[1 \ldots n])$

Input: n number of denominations, m money to be changed for, $c[1 \ldots n]$ coin denominations
Output: $a[1 \ldots n]$ number of coins used, $mincoins[nn][mm]$ minimum number of coins
Require: $c[1] > c[2] > \cdots > c[n]$ and $a[1] + a[2] + \cdots + a[n]$ is minimized

```
{ Compute the number of coins of each denomination and the minimum number of coins. }
money ← m
for i ← 1 to n do
    a[i] ← ⌊money / c[i]⌋
    mincoins ← mincoins + a[i]
    money ← money − a[i] × c[i]
if money > 0 then
    Sorry, not possible to make a change
else
    return a[1...n], mincoins
```

Unlimited DP.

Algorithm : ChangeMaking-Unlimited-DP(n, m, $c[1\ldots n]$)

Input: n number of denominations, m money to be changed for, $c[1\ldots n]$ coin denominations
Output: $a[1\ldots n]$ number of coins used, $mincoins[nn][mm]$ minimum number of coins
Require: $a[1]+a[2]+\cdots+a[n]$ is minimized

```
1: { Compute the minimum number of coins. }
2: mincoins[0] ← 0
3: for i ← 1 to m do
4:    mincoins[i] ← ∞
5:    for j ← 1 to n do
6:       if c[j] ≤ i and mincoins[i − c[j]] + 1 < mincoins[i] then
7:          mincoins[i] ← mincoins[i − c[j]] + 1
8:          parent[i] ← j
9: { Compute the number of coins of each denomination. }
10: if mincoins[m] = ∞ then
11:    Sorry, not possible to make a change
12: else
13:    a[1...n] ← ∅
14:    i ← m
15:    while i ≥ 1 do
16:       a[parent[i]] ← a[parent[i]] + 1
17:       i ← i − c[parent[i]]
18:    return a[1...n], mincoins[m]
```

Limited 01 DP.

Algorithm : ChangeMaking-Limited-01-DP(n, m, $c[1\ldots n]$)

Input: n number of denominations, m money to be changed for, $c[1\ldots n]$ coin denominations
Output: $a[1\ldots n]$ number of coins used, $mincoins[nn][mm]$ minimum number of coins
Require: $a[1]+a[2]+\cdots+a[n]$ is minimized

```
1: { Compute the minimum number of coins. }
2: mincoins[0][0...m] ← ∞
3: mincoins[0...n][0] ← ∅
4: for i ← 1 to n do
5:    for j ← 1 to m do
6:       mincoins[i][j] ← mincoins[i − 1][j]
7:       parent[i][j] ← 0
8:       if c[i] ≤ j and mincoins[i − 1][j − c[i]] + 1 < mincoins[i][j] then
9:          mincoins[i][j] ← mincoins[i][j − c[i]] + 1
10:         parent[i][j] ← 1
11: { Compute the number of coins of each denomination. }
12: if mincoins[n][m] = ∞ then
13:    Sorry, not possible to make a change
14: else
15:    a[1...n] ← ∅
16:    j ← m
17:    for i ← n to 1 do
18:       if parent[i][j] = 1 then
19:          j = j − c[i]
20:          a[i] ← a[i] + 1
21:    return a[1...n], mincoins[n][m]
```

Limited DP.

Algorithm : ChangeMaking-Limited-DP(n, m, $c[1\ldots n]$, $k[1\ldots n]$)

Input: n number of denominations, m money to be changed for, $c[1\ldots n]$ coin denominations, $k[1\ldots n]$ number of coins available
Output: $a[1\ldots n]$ number of coins used, $mincoins[nn][mm]$ minimum number of coins
Require: $a[1]+a[2]+\cdots+a[n]$ is minimized

```
1:  { Convert the limited change making problem to 0-1 change making problem. }
2:  mm ← m, nn ← 0
3:  for i ← 1 to n do
4:      if k[i] > 0 then
5:          count ← 0, power ← 1
6:          repeat
7:              if count + power > k[i] then
8:                  power ← k[i] − count
9:              nn ← nn + 1
10:             cc[nn] ← power × c[i]
11:             d[nn] ← i, v[nn] ← power
12:             count ← count + power
13:             power ← 2 × power
14:         until count = k[i]
15: { Compute the minimum number of coins. }
16: mincoins[0...nn][0] ← ∅, mincoins[0][0...nn] ← ∅
17: for i ← 1 to nn do
18:     for j ← 1 to mm do
19:         mincoins[i][j] ← mincoins[i−1][j]
20:         parent[i][j] ← 0
21:         if cc[i] ≤ j and mincoins[i−1][j−cc[i]] + v[i] < mincoins[i][j] then
22:             mincoins[i][j] ← mincoins[i−1][j−cc[i]] + v[i]
23:             parent[i][j] ← 1
24: { Compute the number of coins of each denomination. }
25: if mincoins[nn][mm] = ∞ then
26:     Sorry, not possible to make a change
27: else
28:     a[1...nn] ← ∅, j ← mm
29:     for i ← nn to 1 do
30:         if parent[i][j] = 1 then
31:             j ← j − cc[i]
32:             a[d[i]] ← a[d[i]] + v[i]
33:     return a[1...n], mincoins[nn][mm]
```

Switches and Bulbs.

There are n switches $s_1, s_2, \ldots, s_n$ in one room and n bulbs $b_1, b_2, \ldots, b_n$ in another room. We do not know which switch is connected to which bulb. Initially all switches are off and hence all bulbs are turned off. We can toggle any number of switches and change their states and then we visit the bulb room to observe the states of the bulbs. What is the strategy that identifies which switch is connected to which bulb minimizing the number of visits to the bulb room.

Solution.

The question appeared in Minko Markov's book *Problems with Solutions in the Analysis of Algorithms*.

The solution is explained through an example. Let there be 8 switches and 8 bulbs. We do not know which switch is connected to which bulb. The 8 switches are $S_1, \ldots, S_8$ and the 8 bulbs are $B_1, \ldots, B_8$. Assume that the bulbs are lighted as in the given table 4.

Iteration 1				Iteration 2				Iteration 3			
S_1	✓	B_1	✓	S_1	✓	B_1	✗	S_1	✓	B_1	✗
S_2	✓	B_2	✓	S_2	✓	B_2	✓	S_2	✗	B_2	✗
S_3	✓	B_3	✗	S_3	✗	B_3	✓	S_3	✓	B_3	✗
S_4	✓	B_4	✓	S_4	✗	B_4	✗	S_4	✗	B_4	✓
S_5	✗	B_5	✗	S_5	✓	B_5	✗	S_5	✓	B_5	✗
S_6	✗	B_6	✗	S_6	✓	B_6	✗	S_6	✗	B_6	✓
S_7	✗	B_7	✓	S_7	✗	B_7	✓	S_7	✓	B_7	✓
S_8	✗	B_8	✗	S_8	✗	B_8	✓	S_8	✗	B_8	✓

Table 4: The bulbs that get lighted up when we turn on the switches. Three iterations are enough to solve the problem.

Then we can find the correspondence between the switches and bulbs as given in Table 5

Iteration 1	Iteration 2	Iteration 3
$S_1 = \{B_1, B_2, B_4, B_7\}$	$S_1 = \{B_2, B_7\}$	$S_1 = \{B_7\}$
$S_2 = \{B_1, B_2, B_4, B_7\}$	$S_2 = \{B_2, B_7\}$	$S_2 = \{B_2\}$
$S_3 = \{B_1, B_2, B_4, B_7\}$	$S_3 = \{B_1, B_4\}$	$S_3 = \{B_4\}$
$S_4 = \{B_1, B_2, B_4, B_7\}$	$S_4 = \{B_1, B_4\}$	$S_4 = \{B_1\}$
$S_5 = \{B_3, B_5, B_6, B_8\}$	$S_5 = \{B_3, B_8\}$	$S_5 = \{B_8\}$
$S_6 = \{B_3, B_5, B_6, B_8\}$	$S_6 = \{B_3, B_8\}$	$S_6 = \{B_3\}$
$S_7 = \{B_3, B_5, B_6, B_8\}$	$S_7 = \{B_5, B_6\}$	$S_7 = \{B_6\}$
$S_8 = \{B_3, B_5, B_6, B_8\}$	$S_8 = \{B_5, B_6\}$	$S_8 = \{B_5\}$

Table 5: The correspondence between switches and bulbs are found in three iterations.

If there are n switches it can be shown that the number of iterations required is $\lceil \log n \rceil$. The solution can be extended if we did not know the initial states of switch whether they were on or not and also the bulbs were not all turned off initially.

Coin Tossing Problems: Fundamentals.

- A coin has two faces: head (H) and tail (T).
- We solve various problems for a biased (or unfair) coin assuming that the probability of occurrence of a head is p and the probability of occurrence of a tail is $q = (1 - p)$.
- Tossing a coin n times is same as tossing n coins once, if they have the same probability of success (or heads).
- The results can be obtained for an unbiased (or fair) coin by just plugging in the values $p = q = \frac{1}{2}$.

k heads.

A coin is tossed n times. What is the probability of getting:

(*a*) *Exactly k heads?*

(*b*) *At most k heads?*

(*c*) *At least k heads?*

Solution.

The binomial (or Bernoulli) distribution is given by

$$\begin{aligned}(p+q)^n &= {}^nC_0 p^0 q^n + {}^nC_1 p^1 q^{n-1} + \cdots + {}^nC_n p^n q^0 \\ &= \sum_{i=0}^{n} {}^nC_i p^i q^{n-i}\end{aligned}$$

We use the above formula to solve the problem.

(*a*) The probability of getting exactly k heads in n tosses is given by

$$P = {}^nC_r p^k q^{n-k}$$

(*b*) The probability of getting atmost k heads in n tosses is the sum of all probabilities of getting 0 heads till k heads, given by

$$P = \sum_{i=0}^{k} {}^nC_i p^i q^{n-i}$$

(*c*) The probability of getting at least k heads in n tosses is the sum of all probabilities of getting k heads till n heads, given by

$$P = \sum_{i=k}^{n} {}^nC_i p^i q^{n-i}$$

---

k continuous heads.

A coin is tossed n times. What is the probability of getting:

(*a*) *Exactly k continuous heads?*

(*b*) *At most k continuous heads?*

(*c*) *At least k continuous heads?*

Solution.

(*a*) The probability of getting k heads among n tosses $= p^k q^{n-k}$.
The number of ways to get k $(k \in [1,n])$ continuous heads among n tosses $= n-k+1$.
Hence, the probability of getting k continuous heads is given by

$$P = \begin{cases} (n-k+1)p^k q^{n-k} & \text{if } k \in [1,n] \\ q^n & \text{if } k = 0 \end{cases}$$

(b) The probability of getting at most k continuous heads is given by

$$P = q^n + \sum_{i=1}^{k} (n-i+1)p^i q^{n-i}$$

(c) The probability of getting at least k continuous heads is given by

$$P = \begin{cases} \sum_{i=1}^{k} (n-i+1)p^i q^{n-i} & \text{if } k \in [1,n] \\ q^n + \sum_{i=1}^{k} (n-i+1)p^i q^{n-i} & \text{if } k = 0 \end{cases}$$

Tosses to get head w.h.p.

How many times should we toss a coin to get a head with high probability?

Solution.

To get a head with high probability, the probability of not getting a head should be $\frac{1}{n^c}$ for some variable n and constant $c \geq 1$.

Probability of not getting a head in 1 toss $= q$.
Probability of not getting a head in $k \log n$ tosses $= q^{k \log n} = n^{k \log q} = n^{-k \log \frac{1}{q}} = n^{-c}$.
Not getting a head in $k \log n$ tosses is with high probability when $k \log \frac{1}{q} \geq 1$ or $k \geq \frac{1}{\log \frac{1}{q}}$.

Hence, we should toss a coin greater than or equal to $\log_{\log \frac{1}{q}} n$ times to get a head with high probability.

Similar problem
If we toss a coin n times, then there will be $O(\log n)$ heads with high probability.

Average tosses to get head.

On an average, how many times should be toss a coin until we get a head?

Solution.

Let the random variables X_i and X be defined as

$$X_i = \begin{cases} 1 & \text{if } i^{\text{th}} \text{ coin toss is heads} \\ 0 & \text{if } i^{\text{th}} \text{ coin toss is tails} \end{cases} \qquad X = \begin{cases} 1 & \text{if we get heads} \\ 0 & \text{if we get tails} \end{cases}$$

Then, $X = \sum_{i=1}^{n} X_i$. Therefore,

Expected value of $X_i = E[X_i] = i \cdot pq^{i-1}$
Expected value of $X = E[X] = \sum_{i=1}^{\infty} E[X_i] = \sum_{i=1}^{\infty} i \cdot pq^{i-1} = \frac{1}{p}$ (after simplification)

Hence, on an average, we should toss a coin $\frac{1}{p}$ times to get a head.

Simulate biased coin and unbiased coin.

How can we simulate:

(*a*) *a biased coin with an unbiased coin?*

(*b*) *an unbiased coin with a biased coin?*

Solution.

(*a*) We want to simulate a biased coin with success probability p with an unbiased coin. Let, $p : q = r : s$, where $r, s \in \mathbb{Z}^+$.

It is easy to see that using an unbiased coin, we can generate a random integer in:

- range $[1, 2]$ with 1 toss.
- range $[1, 2^2]$ with 2 tosses.
- range $[1, 2^3]$ with 3 tosses, and so on.

Find a positive integer k such that $2^{k-1} < r + s \leq 2^k$. We toss the unbiased coin k times. If the random number generated is in the range $[1 \ldots r]$, consider it a head of the biased coin. If the random number generated is in the range $[r+1 \ldots r+s]$, consider it a tail of the biased coin. If the random number generated is in the range $[r+s+1 \ldots 2^k]$, then repeat the process.

Unbiased Coin (k tosses)	**Probability**	**Biased Coin**
$[1 \ldots r]$	$\frac{r}{2^k}$	head
$[r+1 \ldots r+s]$	$\frac{s}{2^k}$	tail
$[r+s+1 \ldots 2^k]$	$1 - \frac{r+s}{2^k}$	repeat again

Table 6: Simulate a biased coin with an unbiased coin.

(*b*) Toss the biased coin twice. If we get a head and a tail on the first and second toss respectively, consider it a head of the unbiased coin. If we get a tail and a head on the first and second toss respectively, consider it a tail of the unbiased coin. If we get two heads or two tails, then repeat the process.

Biased Coin (2 tosses)	**Probability**	**Unbiased Coin**
HT	pq	head
TH	pq	tail
HH	p^2	repeat again
TT	q^2	repeat again

Table 7: Simulate an unbiased coin with a biased coin.

Exercise.

1. (Rob Johnson) The above method of simulating unbiased coin to biased coin with heads probability p works only for rational p. Find a way to simulate an unbiased coin to biased coin where p can be irrational like $1/e$, or $1/\pi$ etc.

Probability bounds.

A coin is tossed n times. What are the probability bounds of getting at least 90% heads using:

(a) *Markov's inequality?*

(b) *Chebyshev's inequality?*

(c) *Chernoff bounds?*

Solution.

Let the random variables X_i and X be defined as

$$X_i = \begin{cases} 1 & \text{if } i^{\text{th}} \text{ coin toss is heads} \\ 0 & \text{if } i^{\text{th}} \text{ coin toss is tails} \end{cases} \qquad X = \begin{cases} 1 & \text{if we get heads} \\ 0 & \text{if we get tails} \end{cases}$$

Then, $X = \sum_{i=1}^{n} X_i$. Therefore,

$$\text{Expectation} = \mu = E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \frac{1}{2} = \frac{n}{2}.$$

$$\text{Variance} = Var[X] = \sum_{i=1}^{n} Var[X_i] = \sum_{i=1}^{n} (E[X_i^2] - E[X_i]^2) = \sum_{i=1}^{n} \frac{1}{4} = \frac{n}{4}.$$

$$90\% \text{ heads} = 90\% \cdot n = \frac{9n}{10}.$$

For a list of all probability bounds, please refer Table 8. To solve the problem we need to find,

$$P\left[X \geq \frac{9n}{10}\right] \text{ or } P\left[Y \leq \frac{n}{10}\right]$$

where $Y = n - X$ and hence, $E[Y] = E[X]$.

Name	Probability Bound	Constraint
Markov's Inequality	$P[X \geq a] \leq \frac{\mu}{a}$	$X \geq 0,\ a > 0$
Chebyshev's Inequality	$P[\lvert X - \mu \rvert \geq a] \leq \frac{Var[X]}{a^2}$	$X \geq 0,\ a > 0$
Chernoff Bound 1	$P[X \geq (1+a)\mu] \leq \left(\frac{e^a}{(1+a)^{(1+a)}}\right)^{\mu}$	$X \geq 0,\ a \in (0,1)$
Chernoff Bound 2	$P[X \geq (1+a)\mu] \leq e^{-\frac{\mu a^2}{3}}$	$X \geq 0,\ a \in (0,1)$
Chernoff Bound 3	$P[X \geq \mu + a] \leq e^{-\frac{a^2}{3\mu}}$	$X \geq 0,\ a \in (0,\mu)$
Chernoff Bound 4	$P[X \leq (1-a)\mu] \geq \left(\frac{e^{-a}}{(1-a)^{(1-a)}}\right)^{\mu}$	$X \geq 0,\ a \in (0,1)$
Chernoff Bound 5	$P[X \leq (1-a)\mu] \geq e^{-\frac{\mu a^2}{2}}$	$X \geq 0,\ a \in (0,1)$
Chernoff Bound 6	$P[X \leq \mu - a] \geq e^{-\frac{a^2}{2\mu}}$	$X \geq 0,\ a \in (0,\mu)$

Table 8: Probability Bounds.

(a) Applying Markov's inequality, we have,

$$P\left[X \geq \frac{9n}{10}\right] \leq \frac{n/2}{9n/10} = \frac{5}{9}$$

(b) Applying Chebyshev's inequality, we have,

$$P\left[X \geq \frac{9n}{10}\right] \leq P\left[|X - E[X]| \geq \frac{4n}{10}\right] \leq \frac{n/4}{(4n/10)^2} = \frac{25}{16n}$$

(c) Applying Chernoff bound 1, we have,

$$P\left[X \geq \frac{9n}{10}\right] = P\left[x \geq \left(1+\frac{4}{5}\right)\frac{n}{2}\right] \leq \left(\frac{e^{\frac{4}{5}}}{\left(1+\frac{4}{5}\right)^{\left(1+\frac{4}{5}\right)}}\right)^{\frac{n}{2}} = e^{\frac{2n}{5}} \cdot \left(\frac{5}{9}\right)^{\frac{9n}{10}}$$

Applying Chernoff bound 2, we have,

$$P\left[X \geq \frac{9n}{10}\right] = P\left[X \geq \left(1+\frac{4}{5}\right)\frac{n}{2}\right] \leq e^{\frac{-8n}{75}}$$

Applying Chernoff bound 3, we have,

$$P\left[X \geq \frac{9n}{10}\right] = P\left[X \geq \left(1+\frac{4}{5}\right)\frac{n}{2}\right] \leq e^{\frac{-8n}{75}}$$

Applying Chernoff bound 4, we have,

$$P\left[Y \leq \frac{n}{10}\right] = P\left[Y \leq \left(1-\frac{4}{5}\right)\frac{n}{2}\right] \geq \left(\frac{e^{-\frac{4}{5}}}{\left(1-\frac{4}{5}\right)^{\left(1-\frac{4}{5}\right)}}\right)^{\frac{n}{2}} = e^{-\frac{2n}{5}} \cdot 5^{\frac{n}{10}}$$

Applying Chernoff bound 5, we have,

$$P\left[Y \leq \frac{n}{10}\right] = P\left[Y \leq \left(1-\frac{4}{5}\right)\frac{n}{2}\right] \geq e^{\frac{-4n}{25}}$$

Applying Chernoff bound 6, we have,

$$P\left[Y \leq \frac{n}{10}\right] = P\left[Y \leq \left(1-\frac{4}{5}\right)\frac{n}{2}\right] \geq e^{\frac{-16}{25}}$$

Applications of coin tossing.

Give some algorithms that uses tossing a coin.

Solution.

Tossing a coin is used in randomized algorithms, where the decisions are made based on the outcomes of an unbiased (or fair) coin toss. Some examples of the algorithms are:

1. Randomized quick sort.
2. Randomized min-cut.
3. Random skip lists.

Largest subarray sum.

Given an array of size n, find the largest subarray sum efficiently.

Solution.

The summary of the five algorithms are as follows:

No	Algorithms	Complexity
1	Naive algorithm	$O(n^3)$
2	Standard algorithm	$O(n^2)$
3	Divide-and-conquer algorithm	$O(n \log n)$
4	Improved divide-and-conquer algorithm	$O(n)$
5	Kadane's algorithm	$O(n)$

Table 9: Largest subarray sum algorithms.

For more info, refer
`http://www.geeksforgeeks.org/divide-and-conquer-maximum-sum-subarray/`
`http://www.cs.cmu.edu/afs/cs/academic/class/15210-f11/www/lectures/all.pdf`
`http://www.ics.uci.edu/~goodrich/teach/cs161/notes/MaxSubarray.pdf`

Naive algorithm.

Algorithm : NaiveAlgorithm$(a[1 \ldots n])$ $O(n^3)$

Input: Array of size n
Output: Largest subarray sum

```
1: max ← −∞
2: for i ← 1 to n do
3:     for j ← i to n do
4:         sum ← 0
5:         for k ← i to j do
6:             sum ← sum + a[k]
7:         if sum > max then
8:             max ← sum
9: return max
```

Standard algorithm.

Algorithm : StandardAlgorithm$(a[1 \ldots n])$ $O(n^2)$

Input: Array of size n
Output: Largest subarray sum

```
1: max ← −∞
2: for i ← 1 to n do
3:     sum ← 0
4:     for j ← i to n do
5:         sum ← sum + a[j]
6:         if sum > max then
7:             max ← sum
8: return max
```

Divide-and-conquer algorithm.

The complexity of the algorithm can be computed as $T(n) = 2T(n/2) + \Theta(n)$

Algorithm : LargestSubarraySum$(a[1 \dots n])$ $O(n \log n)$

Input: Array of size n
Output: Largest subarray sum

```
1: mid ← (1+n)/2
2: S_l ← LargestSubarraySum(a[1 ... mid])
3: S_r ← LargestSubarraySum(a[mid + 1 ... n])
4: S_m ← LargestSubarraySumMerge(a[1 ... n])
5: return Max(S_l, S_r, S_m)
```

Algorithm : LargestSubarraySumMerge$(a[1 \dots n])$

Input: Array of size n
Output: Largest subarray sum that spans between the first and the second half

```
1: { find the maximum suffix in the first half }
2: suffixmax ← 0
3: sum ← 0
4: for i ← mid to 1 do
5:     sum ← sum + a[i]
6:     if sum > suffixmax then
7:         suffixmax ← sum
8: { find the maximum prefix in the second half }
9: prefixmax ← 0
10: sum ← 0
11: for i ← mid + 1 to n do
12:     sum ← sum + a[i]
13:     if sum > prefixmax then
14:         prefixmax ← sum
15: return suffixmax + prefixmax
```

Improved divide-and-conquer algorithm.

The complexity of the algorithm can be computed as $T(n) = 2T(n/2) + \Theta(1)$

Algorithm : LargestSubarraySum$(a[1\ldots n])$ $O(n)$

Input: Array of size n
Output: Largest subarray sum, total, prefixmax, suffixmax

```
1: if n = 1 then
2:     return ⟨a[1], a[1], a[1], a[1]⟩
3: mid ← (1+n)/2
4: [m1, t1, p1, s1] ← LargestSubarraySum(a[1...mid])
5: [m2, t2, p2, s2] ← LargestSubarraySum(a[mid+1...n])
6: M ← Max(s1 + p2, m1, m2)
7: T ← t1 + t2
8: P ← Max(p1, t1 + p2)
9: S ← Max(s1 + t2, s2)
10: return [M, T, P, S]
```

Kadane's algorithm.

Algorithm : KadaneAlgorithm$(a[1\ldots n])$ $O(n)$

Input: Array of size n
Output: Largest subarray sum

```
1: max ← a[1]
2: sum ← a[1]
3: for i ← 2 to n do
4:     sum ← Max(a[i], sum + a[i])
5:     max ← Max(max, sum)
6: return max
```

Longest increasing subsequence problem.

Given an array of size n, find the longest increasing subsequence efficiently. Only the number of elements in the longest increasing subsequence.

Solution.

The summary of the two algorithms are as follows:

No	Algorithms	Complexity
1	DP algorithm	$O(n^2)$
2	DP binary search algorithm	$O(n \log n)$

Table 10: Longest increasing subsequence algorithms.

For more details refer
http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/
http://en.wikipedia.org/wiki/Longest_increasing_subsequence
http://comscigate.com/Books/contests/icpc.pdf

DP algorithm.

Algorithm : DPAlgorithm$(a[1\ldots n])$ $O\left(n^2\right)$

Input: Array of size n
Output: Longest increasing subsequence

```
1:  { Initialize }
2:  for i ← 1 to n do
3:      lis[i] ← 1
4:  { DP recurrence }
5:  for i ← 2 to n do
6:      for j ← 1 to i do
7:          if a[j] < a[i] and lis[j] + 1 > lis[i] then
8:              lis[i] ← lis[j] + 1
9:  { Longest lis[i] }
10: max ← 0
11: for i ← 1 to n do
12:     if max < lis[i] then
13:         max ← lis[i]
14: return max
```

DP binary search algorithm.

Algorithm : DPBinarySearchAlgorithm$(a[1\ldots n])$ $O\left(n\log n\right)$

Input: Array of size n
Output: Longest increasing subsequence

```
1: lis ← 0
2: for i ← 1 to n do
3:     Binary search for largest positive j ≤ lis such that a[m[j]] < a[i] (or set j = 0 if no such value
       exists)
4:     previous[i] ← m[j]
5:     if j = lis or a[i] < a[m[j + 1]] then
6:         m[j + 1] ← i
7:         lis = Max(lis, j + 1)
```