Solutions of Introduction to Algorithms: A Creative Approach

Saman Saadi

Contents

Chapter 1

Mathematical Induction

1.1 Counting Regions in the Plane

A set of lines in the plane is said to be in **general position** if no two lines are parallel and no three lines intersect at a common point.

Guess: Adding one more line to $n-1$ lines in general position in the plane **increases** the number of regions by n. In other words $T(n) = T(n-1) + n$.

The base cases is trivial

- $T(0) = 1$
- $T(1) = T(0) + 1 = 2$
- $T(2) = T(1) + 2 = 2 + 2 = 4$
- $T(3) = T(2) + 3 = 4 + 3 = 7$

So we assume $T(n)$ is correct, now we want to prove $T(n+1)$ is also correct. Let's remove line n^{th}. According to induction hypothesis Adding line $(n+1)^{th}$ add n new regions. If we add line n^{th} again, it intersect with line $(n+1)^{th}$ at exactly one point p. This point is located in region R.

In the absence of line n^{th}, line $(n+1)^{th}$ adds only one new region when it passes R. But in presence of line n^{th}, it adds 2 new regions when it passes R. For other regions line $(n+1)^{th}$ adds $n-1$ new regions with or without the presence of line n^{th}. So line $(n+1)^{th}$ adds $n-1+2 = n+1$ new regions when n^{th} is presented.

So instead of proving the number of regions by adding a new line, we proved how many new regions are added when we have line $(n+1)^{th}$. So It's easy to prove the number of regions. Starting with one line we have $2+2+3+4+\cdots+n = 1+1+2+\cdots+n = 1+\frac{n\times(n+1)}{2}$.

1.2 Euler's Formula

Consider a connected planar map with V vertices, E edges and F faces. A face is an enclosed region. The outside region is counted as one face. So for example, a square has four vertices, four edges and two faces.

Theorem The number of vertices (V), edges (E), and faces (F) in an arbitrary connected planar map are related by the formula $V + F = E + 2$.

Proof It's clear the formula doesn't hold if the planar map is not connected. So we cannot just simply remove an edge. So the base case should be a tree.

Theorem In a tree with V vertices, the number of edges E is $E = V - 1$.

Proof The base case is trivial. Suppose it's true for all trees with V vertices. Now consider a tree with $V+1$ vertices. There should be at least one vertex connected to only one edge. If we don't have such a vertex, then we can start from an arbitrary vertex v and try to visit other vertices. Since each vertex has at least 2 edges, we can easily enter and exit other vertices and visit edges at most once. Since the number of vertices are limited, then we should revisit a vertex. It implies a cycle which is a contradiction. So we have at least one vertex that is connected to only one edge. If we remove that vertex and that edge, the tree is still connected so we can use hypothesis so $E = V - 1$. So by adding one vertex and one edge, the formula is also correct for $V + 1$ vertices.

So the base case is tree. A tree only has one face. So we have $V + 1 = V - 1 + 2$.

Now consider a planar map which is not tree. In other words, it has at least one cycle. If we remove one edge from that cycle, It's still connected. By removing that edge, the inner face will be combined with outer face. So the number of faces also reduced by 1. So the formula is correct. The textbook choose faces as induction parameter but it actually remove an edge. So I don't see any different between choosing edge or face as induction parameter.

1.3 Gray Codes

Gray codes are strings of 0s and 1s in such a way that two neighbours only differ in one digit. For example 100 and 101. If the last string respect the condition with the first one, the code is closed; otherwise it's open. For example 00, 01, 11 and 10 is closed but 00, 01, 11 is open.

Hypothesis: There exists gray codes of length $\lceil \log_2 k \rceil$ for all values $k < n$. If k is even, then the code is closed; if k is odd, then the code is open.

Proof: There are two scenarios:

- If $n = 2m$ We use the hypothesis and assume $s_1, s_2, \dots, s_m$ are gray codes. It can be open or closed. We can create gray codes of size n which is closed like $0s_1, 0s_2, \dots, 0s_m, 1s_m, \dots, 1s_2, 1s_1$.
  Based on the hypothesis the length of s_i $(1 \le i \le m)$ is $\lceil \log_2 m \rceil$. So the length of $0s_i$ is $\lceil \log_2 m \rceil + 1$:

$$\begin{aligned} \lceil \log_2 m \rceil + 1 &= \lceil log_2 \frac{n}{2} + 1 \rceil \\ &= \lceil \log_2 n - log_2 2 + 1 \rceil \\ &= \lceil \log_2 n \rceil \end{aligned}$$

- If $n = 2m+1$ We use the hypothesis and assume $s_1, s_2, \dots, s_m, s_{m+1}$ are gray codes. It can be opne or closed. We can create gray codes of size n which is open like $0s_1, 0s_2, \dots, 0s_m, 0s_{m+1}, 1s_{m+1}, 1s_m, \dots, 1s_2$.
  Based on the hypothesis the length of s_i $(1 \le i \le m{+}1)$ is $\lceil \log_2 (m+1) \rceil$. So the length of $0s_i$ is $\lceil \log_2 (m+1) \rceil + 1$:

$$\begin{aligned} \lceil \log_2 (m+1) \rceil + 1 &= \lceil log_2 (\frac{n-1}{2} + 1) + 1 \rceil \\ &= \lceil \log_2 (\frac{n+1}{2}) + 1 \rceil \\ &= \lceil \log_2 (n+1) - log_2 2 + 1 \rceil \\ &= \lceil \log_2 (n+1) \rceil \end{aligned}$$

  Since n is odd we have $\lceil \log_2 (n) \rceil = \lceil \log_2 (n+1) \rceil$.
  Note that we cannot find another list that is closed. Because we need to find a list that stars with $0s_1$ and ends with $1s_1$. We generate s_{i+1} by flipping exactly one bit of s_i. So $0s_1$ bits are flipped in total m times to generate $0s_{m+1}$. Then we put $1s_{m+1}$ to the second list. We need to flip $1s_{m+1}$ bits $m-1$ times to generate the rest of the list. Therefore s_1 bits are flipped $m + m - 1 = 2m - 1$ which is an odd number. So it's impossible the list starts with $0s_1$ and ends with $1s_1$.

The implementation in C++:

```
string str;
void grayCodes(size_t index, size_t listLength)
{
  if (listLength == 1)
  {
    cout << str << endl;
    return;
  }
  const auto m = listLength / 2;
  if ((listLength % 2 ) == 0)
    grayCodes(index + 1, m);
  else
    grayCodes(index + 1, m + 1);
  str[index] = (str[index] == '0' ? '1' : '0');
  grayCodes(index + 1, m);
}

int main()
{
  const size_t n = 9;
  const size_t codeLen = [n]()
  {
    size_t res = 0;
    //ceil(lg(m)):
    for (size_t m = 1; m < n; m *= 2)
      ++res;
    return res;
  }();
  str = string(codeLen, '0');
  grayCodes(0, n);
}
```

Note that this is not the exact implementation of the proof. For example for $n = 6$ we have:

$$\begin{gathered}
0\ 00 \to 0s_1\\
0\ 01 \to 1s_2\\
0\ 11 \to 1s_3\\
\vdots\\
1\ 11 \to 1s_3\\
1\ 10 \to 1s'_2\\
1\ 00 \to 1s_1
\end{gathered}$$

How this implementation always generate closed gray codes when n is even? Suppose $n = 2m$. The generated code should have the following structure to consider it as closed:

$$\begin{gathered}
0s_1\\
0s_2\\
\vdots\\
0s_m\\
1s_m\\
1s'_{m-1}\\
\vdots\\
1s'_2\\
1s_1
\end{gathered}$$

Suppose $s = (b_{k-1} \dots b_1 b_0)_2$ represents a code. Based on definition $k = \lceil \log_2 m \rceil$. For generating s_1 to s_m we flip s digits $m - 1$ times. We define c_i as the number of times bit ith is flipped. It's obvious that:

$$m - 1 = \sum_{i=0}^{k-1} c_i$$

For the second half of the list which starts with $1s_m$ we use the same pattern to flip bits exactly $m - 1$ times. So in total we have:

$$2 \times (m - 1) = \sum_{i=0}^{k-1} 2 \times c_i$$

So the ith bit is flipped $2 \times c_i$ which is an even number. In other words if we flip the ith bit of s_0, c_i times for all $0 \leq i \leq k-1$ we get s_m. If we flip the ith bit of s_m, c_i times for all $0 \leq i \leq k-1$ we get s_0. So the list starts with $0s_1$ and ends with $1s_1$.

1.3.1 Implementing $\lfloor \log_2 n \rfloor$ and $\lceil \log_2 n \rceil$

Note that calculating $\lceil \log_2 n \rceil$ can be tricky. According to definition we have:

$$\lceil \log_2 n \rceil = r \implies 2^{r-1} < n \leq 2^r$$
$$\lfloor \log_2 n \rfloor = r \implies 2^r \leq n < 2^{r+1}$$

So for $\lceil \log_2 n \rceil$ we are looking for the maximum 2^i which is smaller than n. When we find it the answer is $i + 1$. In each iteration as long as $2^i < n$, we can assume the result is at least $i + 1$:

```
int ceilLogarithm(int n)
{
  int r = 0;
  for (int m = 1; m < n; m *= 2)
    ++r;
  return r;
}
```

For calculating $\lfloor \log_2 n \rfloor$ we are looking for the minimum 2^i which is bigger than n. When we find it the answer is $i - 1$. In each iteration as long as $2^i \leq n$, we can assume the result is at least i:

```
int floorLogarithm(int n)
{
  int r = 0;
  for (int m = 2; m <= n; m *= 2)
    ++r;
  return r;
}
```

1.4 Website Questions

1.4.1 SRM 784 - Division II, Level One: Scissors

You are in charge of N other people. You have a pair of scissors. But none of your N helpers do.

You purchased N pairs of scissors which are wrapped in plastic. Getting scissors out of the plastic wrap requires having another pair of scissors (that's not in plastic) and it takes 10 seconds. Assume that everything other than opening the packages happens instantly.

Calculate and return the shortest amount of time (in seconds) in which it is possible to release all the scissors from their plastic wraps.

For more information visit this website.

Solution There are two methods which are similar to each other. Note that how using different approaches to break down the problem can make it easier to understand.

Method 1 Suppose that we have k knives:

Hypothesis: We know the solution for $n < N$ and $1 \leq k \leq N + 1$

Proof: Note that we must prove that it also correct for $n = N$ and $1 \leq k \leq N + 1$. So if we have N unwrapped packages and k knives ($1 \leq k \leq N + 1$), we can unwrap at most k packages. After it we have at most $k + k$ knives.

$$T(n,k) = \begin{cases} T(n-k, 2k) + 10 & k \leq n \wedge n \neq 0 \\ 0 & n = 0 \\ T(0, k+n) + 10 & k > n \wedge n \neq 0 \end{cases}$$

The solution is $T(N, 1)$. Because at the beginning we only have one unwrapped knife.

Method 2 Unlike the previous one, the hypothesis only has one parameter.

Hypothesis: We know the solution for $n < N$ pairs of wrapped scissors.

Proof: Suppose $n = N$. For an optimal solution we want to use as many pairs of scissors as possible to unwrap the remaining ones. Suppose $T(n)$ is the minimum time required to unwrap n pairs of scissors. Of course after we do that we have $n + 1$ pairs of scissors (remember you have a pair of unwrapped scissors). For edge cases we consider two possible scenarios:

1. $n = N = 2k+1$ We use the hypothesis and find the answer for the first k wrapped pairs of scissors ($T(k)$). Then using those k ones plus the first one we can unwrapped the remaining $k + 1$ ones:

$$T(2k+1) = T(k) + 10$$

2. $n = N = 2k$ We use the hypothesis and unwrapped the first k pairs of scissors. Besides those k pairs of scissors, we have another one which is unwrapped from the beginning but we only have k pairs of wrapped scissors. We don't use one of those $k+1$ ones:

   $$T(2k) = T(k) + 10$$

   Note that the following equation is not always correct:

   $$T(2k) = T(k-1) + 10 \times 2$$

   Because by unwrapping the first $k-1$ ones, we have k pairs of unwrapped scissors and $k+1$ wrapped ones. On the other hand, $T(k) = T(k-1) \vee T(k) = T(k-1) + 10$. So $T(2k) = T(k) + 10$ finds the optimal solution.

We can squeeze both equations into one:

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + 10 & n > 0 \\ 0 & n = 0 \end{cases}$$

1.5 Exercises

Exercise 1 For $n = 1$ it's trivial. We assume it's true for n. Now we want to extend it to $n+1$:

$$\begin{array}{c|l} x^{n+1} - y^{n+1} & x - y \\ \cline{2-2} x^{n+1} - x^n y & x^n \\ \cline{1-1} x^n y - y^{n+1} & \end{array}$$

We use hypothesis. Assume that $\frac{x^n - y^n}{x-y} = z$ So we can write:

$$\begin{aligned} \frac{x^{n+1} - y^{n+1}}{x-y} &= \frac{x^n(x-y) + x^n y - y^{n+1}}{x-y} \\ &= \frac{x^n(x-y) + y(x^n - y^n)}{x-y} \\ &= \frac{x^n(x-y)}{x-y} + \frac{y(x^n - y^n)}{x-y} \\ &= x^n + yz \end{aligned}$$

Exercise 7 For $n = 1$ The answer is trivial. Assume that it's correct for $2n$. We want to prove that it also corrects for $2n+2$. Assume that the name of the set is S. Based on the problem definition $|S| = n+2$. We want to prove there exists pair (a, b) in which $a \mod b = 0$. There are three cases:

1. All members in S are less than or equal to $2n$. We can use induction hypothesis to find a and b.
2. Either $2n+1$ or $2n+2$ is in S. Then we need to choose $n+1$ numbers from 1 to $2n$. Based on hypothesis we can find a and b.
3. Both $2n+1$ and $2n+2$ are in S. There are two cases:
   (a) $n+1 \in S$: We can choose $a = 2n+2$ and $b = n+1$ and we are done.
   (b) $n+1 \notin S$: We define a new set:
   $$S' = \{s \in S \mid s \neq 2n+1 \land s \neq 2n+2\} \cup \{n+1\}$$
   Since all members of S' are less than or equal to $2n$ and $|S'| = n+1$, based on hypothesis there should be a and b in S'. It's obvious that $b \leq \frac{2n}{2} = n$. If $a = n+1$, then $2n+2 \mod b = 0$ so we are done. If $a \neq n+1$ we are also done.

Exercise 14 We assume $a_1 = 1, a_2 = 2, a_3 = 3, a_4 = 4, a_5 = 5, a_6 = 10, a_7 = 20, \ldots$. For $n = 1$ it's trivial. We assume it's correct for $k < n$. Suppose $S(n)$ has the answer. There should be at least one a_i in which $n - a_i < a_i$ so we are sure $a_i \notin S(n - a_i)$:

$$S(n) = \begin{cases} S(n-a_i) \cup \{a_i\} & \lfloor \frac{n}{2} \rfloor < a_i < n \\ \{a_i\} & n = a_i \end{cases}$$

For $n < 5$ this a_i exists. Since $a_{i+1} = 2a_i$ for $i \geq 5$, there is at least one a_i in that range.

Exercise 15 The example in problem statement is wrong: $81 = 54 + 24 + 3$. Anyway, we cannot use the proof of exercise 14. Because we assume there exists at least one $\lfloor \frac{n}{2} \rfloor < a_i < n \lor a_i = n$ which doesn't exist for $n = 48$. On the other hand it's impossible to find an answer for $n = 49$.

By changing the hypothesis we can solve this problem. We add the dummy member $a_0 = 0$ to the series so we have:

$$A = \{0, 1, 2, 3, 6, 12, 24, 54, 84, 114, \ldots\}$$

We assume the answer is $S(n,k)$ which is the unique list of $a_i \in A$ in such a way $S(n,k) = \{a_i \in A \mid a_i \neq 0 \land a_i \leq k\}$

$$S(n,k) = \begin{cases} S(n-a_i, a_{i-1}) \cup \{a_i\} & \max(\{a_i \in A \mid a_i \leq k\}) \\ \emptyset & k = 0 \lor n = 0 \end{cases}$$

The final solution is $S(n,n)$. Unlike exercise 14 solution, this one can handle a case like $S(48,24) = \{\underbrace{1,2,3,6,12}_{24}, 24\}$

Exercise 17 For $n = 3$ it's correct. We need to prove adding 1 line, increase the number of triangles by at least 1. Suppose it's true for n lines. Now consider line $n+1$. If we remove line n, we can use hypothesis so line $n+1$ create at least one triangle. Consider one of them and we call it T. If we add line n again there are two cases:

1. Line n does not cross triangle T. So line $n+1$ creates at least one triangle.

2. Line n crosses triangle T. Note that since lines are in general positions, when a line crosses a triangle, it intersects with exactly two edges of triangle. There are two cases

   (a) Line n crosses one edge of triangle which line $n+1$ creates. So T is not a triangle anymore but it contains a smaller triangle T' which consists of lines n, $n+1$ and another line that creates T. We call it n'. Besides T, which is not a triangle anymore, line $n+1$ should create at least another triangle. If we remove n', Both T and T' lose one edge. Based on hypothesis line $n+1$ should create another triangle named T''. Since two lines in general positions intersect each other in exactly one point, line n' doesn't cross triangle T''. So we can safely add it again.

   (b) Line n intersects line $n+1$ outside T. So n crosses two edges of T that $n+1$ creates none of them. T is not triangle anymore but it contains a smaller triangle T' which n and two other lines that create T also creates T'. We call them n' and n''. If we remove n' or n'', Both T and T' loses one edge. Based on hypothesis, line $n+1$ should have another triangle T'' that neither n' nor n'' cross it.

Exercise 23 This is Pick's theorem. Since base case is more complicated we prove it later! Suppose the theorem is true for polygon P and triangle T which only shares 1 edge with it. We want to prove it's also true for PT. In other words, since we can triangulate every simple polygon, we create PT just by adding T to P. Suppose P and T share C boundary points on that common edge (including two vertices). So we can calculate the boundary points of PT:

$$\begin{aligned} p_{pt} &= p_p + p_t - 2C + 2 \\ \implies p_p + p_t &= p_{pt} + 2C - 2 \end{aligned}$$

We can also calculate its interior points:

$$\begin{aligned} q_{pt} &= q_p + q_t + C - 2 \\ \implies q_p + q_t &= q_{pt} - C + 2 \end{aligned}$$

Now we can calculate the areas of PT:

$$\begin{aligned}
A_{pt} &= A_p + A_t \\
&= \frac{p_p}{2} + q_p - 1 + \frac{p_t}{2} + q_t - 1 \\
&= \frac{p_p + p_t}{2} + q_p + q_t - 2 \\
&= \frac{p_{pt} + 2C - 2}{2} + q_{pt} - C + 2 - 2 \\
&= \frac{p_{pt}}{2} + q_{pt} - 1
\end{aligned}$$

So the formula is also correct for PT. Since for an arbitrary polygon, we remove a triangle that shares an edge with it in each step, the base case is a triangle. So we need to prove this theorem is true for all triangles. We prove it in multiple steps. First note the following facts:

1. we can break every rectangle into two right triangles. Those two triangles share exactly two edges with the rectangle.
2. We can break every rectangle into one triangle which is not necessary right and two right triangles.

First we need to prove it for rectangles. We assume the theorem is true for all rectangles $n \times m$ in which $n \leq N$ and $m \leq M$. We want to prove it also correct for $N \times M + 1$ and $N + 1 \times M$. The base case is a unit square which is trivial. Now consider rectangle $R = N \times M + 1$. We can consider it as two rectangles $R_1 = N \times M$ and $R_2 = N \times 1$. Based on hypothesis the formula is correct for T_1 and T_2. Assume $T1$ and $T2$ has C boundary points in common.

$$\begin{aligned}
p_r &= p_{r_1} + p_{r_2} - 2C + 2 \\
\implies p_{r_1} + p_{r_2} &= p_r + 2C - 2 \\
q_r &= q_{r_1} + q_{r_2} + C - 2 \\
\implies q_{r_1} + q_{r_2} &= q_r - C + 2
\end{aligned}$$

So we have

$$\begin{aligned}
A_r &= A_{r_1} + A_{r_2} \\
&= \frac{p_{r_1} + p_{r_2}}{2} + q_{r_1} + r_{r_2} - 2 \\
&= \frac{p_r}{2} + q_r - 1
\end{aligned}$$

Similarly we can break $N + 1 \times M$ into $T_1 = N \times M$ and $T_2 = 1 \times M$ and prove it.

Now we know it's true for all rectangles, we want to prove it for all right triangles that share exactly two edges with surrounding rectangle. As mentioned before we can break every rectangle R into two right triangles T_1 and T_2. Assume those two triangles have C common boundary points:

$$\begin{aligned} p_r &= p_{t_1} + p_{t_2} - 2C + 2 \\ q_r &= q_{t_1} + q_{t_2} + C - 2 \end{aligned}$$

$$\begin{aligned} A_{t_1} = A_{t_2} &= \frac{A_r}{2} \\ &= \frac{\frac{p_r}{2} + q_r - 1}{2} \\ &= \frac{\frac{p_{t_1}+p_{t_2}-2C+2}{2} + q_{t_1} + q_{t_2} + C - 2 - 1}{2} \\ &= \frac{\frac{p_{t1}+p_{t_2}}{2} + q_{t_1} + q_{t_2} - 2}{2} \\ &= \frac{\underbrace{\frac{p_{t_1}}{2} + q_{t_1} - 1}_{A_{t_1}} + \underbrace{\frac{p_{t_2}}{2} + q_{t_2} - 1}_{A_{t_2}}}{2} \end{aligned}$$

Now we need to prove the theorem is correct for case 2 which have 3 triangles. As mentioned before we can break rectangle R into two right triangles T_1 and T_2 that shares two edges with R and triangle T which is not necessary right and shares 1 edge with T. Assume T_1 and T have C_1 common boundary points. Also assume T_2 and T have C_2 boundary points:

$$\begin{aligned} p_r &= p_{t_1} + p_{t_2} + p_t - 2C_1 - 2C_2 + 2 \\ \implies p_r - p_{t_1} - p_{t_2} &= p_t - 2C_1 - 2C_2 + 2 \\ q_r &= q_{t_1} + q_{t_2} + q_t + C_1 - 2 + C_2 - 1 \\ \implies q_r - q_{t_1} - q_{t_2} &= q_t + C_1 + C_2 - 3 \end{aligned}$$

$$\begin{aligned} A_t &= A_r - A_{t_1} - A_{t_2} \\ &= \frac{p_r}{2} + q_r - 1 - \frac{p_{t_1}}{2} - q_{t_1} + 1 - \frac{p_{t_2}}{2} - q_{t_2} + 1 \\ &= \frac{p_r - p_{t_1} - p_{t_2}}{2} + q_r - q_{t_1} - q_{t_2} + 1 \\ &= \frac{p_t - 2C_1 - 2C_2 + 2}{2} + q_t + C_1 + C_2 - 3 + 1 \\ &= \frac{p_t}{2} + q_t - 1 \end{aligned}$$

Exercise 24 Assume $k = \lceil \log_2 n \rceil$. For k we can only have $n = 2$. But for $k-1$, we can create n objects. Suppose $n = 2m$. We use gray code algorithm to create $s_1, s_2, \ldots, s_m$. It's obvious s_i has $k-1$ bits. We define s'_i as the complement of s_i (each bit is flipped). If s_i and s_{i+1} differ only in 1 bit, then s_i and s'_{i+1} differ in $k-1$ bits. Based on these facts we can extend it to n. We assume m is even. On the first line you can see the first m codes and on the second line you can see the rest:

$$\begin{array}{cccccc} 0s_1 & 1s'_2 & 0s_3 & 1s'_4 & \dots & 0s_{m-1} & 1s'_m \\ 1s_m & 0s'_{m-1} & 1s_{m-2} & 1s'_{m-3} & \dots & 1s_2 & 0s'_1 \end{array}$$

If m is odd:

$$\begin{array}{ccccccc} 0s_1 & 1s'_2 & 0s_3 & 1s'_4 & \dots & 0s_{m-1} & 0s_m \\ 0s'_m & 1s_{m-1} & 0s'_{m-2} & 1s_{m-3} & \dots & 1s_2 & 0s'_1 \end{array}$$

Note that adding a new bit is necessary. To make sure we avoid duplication we must use $1s'_i$ and not $0s'_i$. For an example consider $n = 8$.

We can extend this idea to find gray codes that differ in exactly $k - i$ bits. We assume $n = 2^i m$. Then we find two set of gray codes $T = \{t_1, t_2, \dots, t_{2^i}\}$ and $S = \{s_1, s_2, \dots, s_m\}$. We need to concatenate these two codes. Consider $t_i s_j$ and $t_{i+1} s'_{j+1}$. t_i and t_{i+1} differ only in 1 bit. Also s_j and s'_{j+1} differ in $k - i - 1$ bits. So in total $t_i s_j$ and $t_{i+1} s'_{j+1}$ differ in $k - i - 1 + 1 = k - i$ bits. We assume m is even. In the first line you can see the first $2^{i-1}m$ codes and in the second line you can see the rest:

$$\begin{array}{cccccccc} t_1 s_1 & t_2 s'_2 & t_3 s_1 & \dots & t_{2^i} s'_2 & t_1 s_3 & \dots & t_{2^i} s'_m \\ t_{2^i} s_m & t_{2^{i-1}} s'_{m-1} & t_{2^{i-2}} s_m & \dots & t_1 s'_{m-1} & t_{2^i} s_{m-2} & \dots & t_1 s'_1 \end{array}$$

If m is odd:

$$\begin{array}{cccccccc} t_1 s_1 & t_2 s'_2 & t_3 s_1 & \dots & t_{2^i} s'_2 & t_1 s_3 & \dots & t_{2^{i-1}} s_m \\ t_{2^{i-1}} s'_m & t_{2^{i-2}} s_{m-1} & t_{2^{i-3}} s'_m & \dots & t_1 s'_m & t_{2^i} s'_{m-1} & \dots & t_1 s'_1 \end{array}$$

For finding the codes that differ in exactly $k-i$ bits, $1 \le i \le k-1$. If $i = k-1$, $n = 2^{i-1}m$. Since $m = \lceil \frac{n}{2^i} \rceil$, $m = \lceil \frac{n}{2^{k-1}} \rceil = 2$ which is gray code algorithm itself.

Also note that if $n \mod 2^i \neq 0$, then the last code is not $t_1 s'_1$, so the codes are not closed. This is the general form of gray codes when n is odd. We can say, if n is a multiple of 2^i, then we can find gray codes with exactly $k - i$ different bits that is closed.

Exercise 25 Suppose we have $T = (V, E)$. If $T_1 = (V_1, E_1)$ is a subtree of T, then $V_1 \subseteq V \land E_1 \subseteq E$. T_1 should have all tree properties such as $E_1 = V_1 - 1$ and there should be exactly one path between every $u, v \in V_1$. Suppose T_2 is another subtree of T. We define $T_1 \cap T_2 = (V_1 \cap V_2, E_1 \cap E_2)$ which we can call it $T' = (V', E')$. So based on this definition $V' \subseteq V \land E' \subseteq E$. We claim $T_1 \cap T_2$ is also a subtree of T. For proving this assume that it's not a subtree of T. It means there are at least two vertices $u, v \in V'$ that aren't reachable from each other. Those two vertices should be in T_1 and T_2. Based on definition there should be exactly one path $v_{i_1}, v_{i_2}, \dots, v_{i_k}$ in which $u = v_{i_1}$ and $v = v_{i_k}$. Since both T_1 and T_2 are subtrees of T, they should have vertices $v_{i_1}, v_{i_2}, \dots, v_{i_k}$ and edges $(v_{i_i}, v_{i_{i+1}})$ for all $1 \le i \le k-1$. Therefore T' should also have those vertices and edges which is contradiction. So $T_1 \cap T_2$ is also a subtree of T. Note that $T_1 \cup T_2$ is not necessary a tree of T.

Now we use induction to prove it. For $k = 2$ it's trivial. Assume it's correct for $\leq k$ subtrees. We want to prove it for $k+1$ subtrees. We can conclude from hypothesis:

$$\underbrace{T_1 \cap T_2 \cap \cdots \cap T_{k-1}}_{T'} \cap T_{k+1} = S \neq \emptyset$$
$$\underbrace{T_1 \cap T_2 \cap \cdots \cap T_{k-1}}_{T'} \cap T_k = S' \neq \emptyset$$

As we proved it before, S and S' are also subtrees of T. Since $S \cap S' \neq \emptyset$ (both of them have at least one $v \in T'$), we can use hypothesis for $k = 2$ for S and S'. So it also holds for $k+1$.

Exercise 27 For more information you can read this Wikipedia article. Suppose $T(n)$ is the number of regions with n points on the circle. Let's try to solve the question for small values:

n	$T(n)$
0	$1 = 2^0$
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
5	$16 = 2^4$
6	$31 = 2^5 - 1$

As you can see $T(n) \neq 2T(n-1)$ when $n = 6$. So we don't know the difference between $T(n)$ and $T(n-1)$. Suppose we know the answer for $T(n-1)$. Now consider point n. We need to add $n-1$ new segments. Consider segment i which connects point n to point i. Since no three segments intersect each other on one point, The number of new regions that it produce is the number of segments it intersects plus one. We know that on one side of segments i there are $i-1$ points and on the opposite side there are $n-2-(i-1)$ points. Therefore segment i produce $(i-1) \times (n-i-1) + 1$ new regions. We know that (For more information read this Wikipedia article):

$$\begin{aligned}\sum_{i=1}^{n} i^2 &= \frac{n(n+1)(2n+1)}{6} \\ &= \frac{2n^3 + 3n^2 + n}{6} \\ &= \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}\end{aligned}$$

Also we know that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. So we have:

$$\begin{aligned}
T(n) &= T(n-1) + \sum_{i=1}^{n-1} ((i-1) \times (n-i-1) + 1) \\
&= T(n-1) + \sum_{i=1}^{n-1} (ni - i^2 - n + 2) \\
&= T(n-1) + \sum_{i=1}^{n-1} (n(i-1)) - \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} 2 \\
&= T(n-1) + n\frac{(n-2)(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} + 2(n-1) \\
&= T(n-1) + (n-1)(\frac{n(n-2)}{2} - \frac{n(2n-1)}{6} + 2) \\
&= T(n-1) + (n-1)\frac{n^2 - 5n + 12}{6} \\
&= T(n-1) + \frac{n^3}{6} - n^2 + \frac{17n}{6} - 2
\end{aligned}$$

So the answer is $T(n) = T(n-1) + \frac{n^3}{6} - n^2 + \frac{17n}{6} - 2$. We can find an answer for this recursive function. Suppose $D(n) = \frac{n^3}{6} - n^2 + \frac{17n}{6} - 2$. $D(0) = -2$ but for $n \geq 1$, $D(n)$ is correct. Remember that $T(0) = 1$. Also according to this artichle we have:

$$\begin{aligned}
\sum_{i=1}^{n} i^3 &= (\sum_{i=1}^{n} i)^2 \\
&= (\frac{n(n+1)}{2})^2 \\
&= \frac{n^4}{4} + \frac{n^3}{3} + \frac{n^2}{2}
\end{aligned}$$

So we have:

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} (\frac{i^3}{6} - i^2 + \frac{17i}{6} - 2) + 1 \\
&= \frac{n}{24}(n^3 - 6n^2 + 23n - 18) + 1
\end{aligned}$$

Exercise 30 Suppose it's true for a complete tree of height h. In other words $S(h) = 2^{h+1} - h - 2$. We want to prove that $S(h+1) = 2^{h+2} - h - 3$. Based on definition a complete tree of height $h+1$ consists of two complete trees of height h plus a new root. We define the height of node v as $H(v) = h - d(v)$. We know that $d(v)$ is the distance of v from the root of tree. When we add a new root to the tree we have $H(v) = h + 1 - (d(v) + 1) = h - d(v)$. In other words, its height doesn't change. So we have:

$$\begin{aligned} S(h+1) &= 2S(h) + h + 1 \\ &= 2(2^{h+1} - h - 2) + h + 1 \\ &= 2^{h+2} - h - 3 \end{aligned}$$

Now let's define another problem. Let's find the number of nodes in a complete tree of height h. We called it $V(h)$. We know that $V(0) = 1$. Based on definition we have:

$$\begin{aligned} V(h) &= 2V(h-1) + 1 \\ &= 2(2(V(h-2)+1)) + 1 \\ &= \underbrace{2(2(2}_{2^3}(V(h-3) + \underbrace{1}_{2^2}) + \underbrace{1}_{2^1})) + 1 \\ &= 2^h + \sum_{i=0}^{h-1} 2^i \\ &= 2^h + \frac{1 \times 2^h - 1}{2 - 1} \\ &= 2^{h+1} - 1 \end{aligned}$$

Exercise 31 We assume $F(2k+1) = F(k)^2 + F(k+1)^2$ is correct for $k \le n$. We know that $2k+1$ is an odd number. Let's find the formula for $F(2k)$. We know that $F(2k+1) = F(2k) + F(2k-1)$. So we have:

$$\begin{aligned} F(2k) &= F(2k+1) - F(2k-1) \\ &= F(k)^2 + F(k+1)^2 - (F(k-1)^2 + F(k)^2) \\ &= F(k+1)^2 - F(k-1)^2 \end{aligned}$$

So based on hypothesis for $1 \le k \le n$ (we assume $F(0) = 0$):

$$\boxed{\begin{aligned} F(2k) &= F(k+1)^2 - F(k-1)^2 \\ F(2k+1) &= F(k)^2 + F(k+1)^2 \end{aligned}}$$

For $k = 1$ it's trivial. Now we want to prove:

$$\boxed{\begin{aligned} F(2n+2) &= F(n+2)^2 - F(n)^2 \\ F(2n+3) &= F(n+1)^2 + F(n+2)^2 \end{aligned}}$$

Let's start with $F(2n+2)$. We know that $F(n+2)^2 = (F(n+1)+F(n))^2$. So $F(n)^2 + F(n+1)^2 = F(n+2)^2 - 2F(n)F(n+1)$:

$$\begin{aligned}
F(2n+2) &= F(2n+1) + F(2n) \\
&= F(n)^2 + F(n+1)^2 + F(n+1)^2 - F(n-1)^2 \\
&= F(n+2)^2 - 2F(n)F(n+1) + F(n+1)^2 - F(n-1)^2 \\
&= F(n+2)^2 + \underbrace{F(n+1)}_{F(n)+F(n-1)}(-2F(n) + \underbrace{F(n+1)}_{F(n)+F(n-1)}) - F(n-1)^2 \\
&= F(n+2)^2 + (F(n-1)+F(n))(F(n-1)-F(n)) - F(n-1)^2 \\
&= F(n+2)^2 + F(n-1)^2 - F(n)^2 - F(n-1)^2 \\
&= F(n+2)^2 - F(n)^2
\end{aligned}$$

Now we can easily prove $F(2n+3)$:

$$\begin{aligned}
F(2n+3) &= F(2n+2) + F(2n+1) \\
&= F(n+2)^2 - F(n)^2 + F(n)^2 + F(n+1)^2 \\
&= F(n+1)^2 + F(n+2)^2
\end{aligned}$$

Note that based on this equation we can solve Fibonacci numbers in $O(\log_2 n)$:

```
unordered_map<int, uint64_t> calc;

uint64_t fibonacci(int n) //O(logn)
{
  if (n == 0 || n == 1)
    return n;
  if (n == 2)  //To avoid stack overflow when n = 2
    return 1;
  auto& res = calc[n];
  if (res != 0)
    return res;
  int k = n / 2;
  if ((n % 2) == 0)
  {
    auto res1 = fibonacci(k + 1);
    auto res2 = fibonacci(k - 1);
    res = res1 * res1 - res2 * res2;
  }
  else
  {
    auto res1 = fibonacci(k);
    auto res2 = fibonacci(k + 1);
    res = res1 * res1 + res2 * res2;
  }
  return res;
}
```

Exercise 32 For simplicity $z = n^2 - m(n+1) + 2n + m^2$. We prove this in two steps:

1. We assume inequality is true for m and using that we prove it's correct for $m+1$. For the base case $m = 1$ we have $n^2 - (n+1) + 2n + 1 = n^2 + n \leq n^2 + n$. We assume $z \leq n^2 + n$. Now consider $m + 1$:

$$\begin{aligned}
&n^2 - (m+1)(n+1) + 2n + (m+1)^2 \leq n^2 + n \\
\implies &n^2 - m(n+1) - n - 1 + 2n + m^2 + 2m + 1 \leq n^2 + n \\
\implies &z + 2m - n \leq n^2 + n \\
\implies &2m - n \leq 0 \\
\implies &m \leq \frac{n}{2} \\
&= \lfloor \frac{n}{2} \rfloor
\end{aligned}$$

So the inequality is true for $1 \leq m \leq \lfloor \frac{n}{2} \rfloor$

2. We assume inequality is true for m and using that we prove it's correct for $m - 1$. For the base case $m = n$ we have $n^2 - n(n+1) + 2n + n^2 = n^2 - n^2 - n + 2n + n^2 = n^2 + n$. We assume $z \leq n^2 + n$. Now consider $m - 1$:

$$\begin{aligned}
&n^2 - (m-1)(n+1) + 2n + (m-1)^2 \leq n^2 + n \\
\implies &n^2 - m(n+1) + n + 1 + 2n + m^2 - 2m + 1 \leq n^2 + n \\
\implies &z - 2m + n + 2 \leq n^2 + n \\
\implies &-2m + n + 2 \leq 0 \\
\implies &m \geq \lceil \frac{n+2}{2} \rceil
\end{aligned}$$

Note that n is an integer. To cover the edge case $\frac{n}{2}$:

1. $n = 2k$: For the first induction $m \leq \frac{2k}{2} = k$. For the second induction $m \geq \frac{2k+2}{2} = k + 1$. So we covered $1 \leq m \leq n$.

2. $n = 2k+1$: For the first induction $m \leq \frac{2k+1}{2} = k + \frac{1}{2} = k$. For the second induction $m \geq \frac{2k+3}{2} = \frac{2k+2+1}{2} = k+1+\frac{1}{2} = k+2$. As you can see when n is odd, the inductions don't cover $k+1$. So we need to consider this special case. Based on first induction we know $z = n^2 - k(n+1) + 2n + k^2 \leq n^2 + n$:

$$\begin{aligned}
&n^2 - (k+1)(n+1) + 2n + (k+1)^2 \leq n^2 + n \\
\implies &n^2 - k(n+1) - n - 1 + 2n + k^2 + 2k + 1 \leq n^2 + n \\
\implies &z + 2k - n \leq n^2 + n \\
\implies &2k - n \leq 0 \\
\implies &k \leq \frac{n}{2} \\
\implies &k \leq \frac{2k+1}{2} \\
\implies &k \leq k + \frac{1}{2}
\end{aligned}$$

Exercise 33 For more information read this CodeForces article and this Wikipedia article.

We choose the number of vertices as induction parameters. If $|V| = 1$ we don't have any edges so it's trivial. Suppose we know how to partition the edges of every undirected connected graph with less than V vertices, which doesn't have bridge edges, into ear decomposition. Now consider an arbitrary connected undirected graph $G = (V, E)$. We choose an arbitrary vertex u. We know that every edge in this graph should be in a cycle; otherwise by removing that edge we disconnect the graph which means we have a bridge which is contradiction. We remove vertex u and all its edges. We may introduce bridges.

Consider bridge (x, y). Since in presence of u, (x, y) is not a bridge, it means there is a path $x \to u \to y$ which doesn't have edge (x, y). Adding (x, y) to that path, makes it a cycle. Note that bridge (x, y) is only in one cycle. So removing (x, y) doesn't introduce new bridges. We call the set of all those edges that by u removal become bridges, D.

In summery we remove vertex u and all edges in D. Now we have connected components $C_1, \ldots, C_n$. Each of them doesn't have any bridges. We use the hypothesis to find ear decomposition for them. Now we add vertex u and its edges as well as those edges in D. Assume the degree of vertex u is d. So there are cycles $c_1, \ldots, c_d$ that include u. All edges in D are in these cycles. Now consider c_i. If all edges of c_i are not in previous ears (i.e. all edges are in D), We add c_i as a new ear but we should start from another vertex other than u to respect the conditions in the problem statements. If there is at least one edge that is not in previous ears, we break c_i into one or more paths. Each of these paths have edges that aren't in previous ears and its endpoints belong to previous ears.

Now consider an undirected graph which may not be connected and it may has bridges. We want to find ears and bridges in this graph. To implement this we use DFS to create DFS forest. We choose the root of one of DFS trees named u. We traverse all its back edges to find all cycles that include u and add those edges that are not in previous ears to the current ear. When we finished with u, we try the next vertex in DFS order.

Since the graph is undirected, $v \in G.adj[u]$ implies $u \in G.adj[v]$ which can make the implementation a little tricky.

```
function EEAR-DECOMPOSITION($G$)
    INIT($G$)                                                  ▷ $O(V)$
    for all $u \in G.V$ do
        if $u.color == WHITE$ then                             ▷ $O(V)$
            Let $CC$ be a new connected component              ▷ $O(CCs)$
            $CC.V.clear()$
            $CC.E = \emptyset$
            DFS($G, u, CC$)                                    ▷ $O(E)$
            FIND-EARS($CC, u$)                                 ▷ $O(V + E)$
            $G.CCs = G.CCs \cup CC$
        end if
    end for
end function
```

```
function INIT($G$)
    for all $u \in G.V$ do
        $u.color = WHITE$
        $u.\pi = NIL$
        $u.visited = False$                      ▷ belong to previous ears
    end for
    $G.CCs = \emptyset$                          ▷ connected components
end function
```

```
function DFS($G, u, CC$)
    $u.color = GRAY$
    $CC.V.insert(u)$                                           ▷ In DFS order
    for all $v \in V.adj[u]$ do
        $CC.E = CC.E \cup \{(u, v)\}$
        $CC.adj[u].insert(v)$
        if $v.color == WHITE$ then                             ▷ tree edge
            $v.\pi = u$
            DFS($G, u, CC$)
        end if
    end for
    $u.color = BLACK$
end function
```

```
function FIND-EARS($CC, u$)
    $CC.bridges = \emptyset$
    for all $(u, v) \in CC.E$ do
        $CC.bridges = CC.bridges \cup \{(u, v)\}$
    end for
    for all $u \in CC.V$ do                          ▷ the list is in DFS order
        for all $v \in CC.adj[u]$ do
            tree-edge = $u.\pi == v \vee v.\pi == u$
            back-edge = tree-edge == $False$
            if back-edge $\wedge (u, v) \in CC.bridges$ then
                Let $ear$ be a new ear
                $ear.clear()$
                $x = u$
                $y = v$
                repeat
                    $x.visited = True$
                    $ear.insert(x)$
                    $ear.insert(y)$
                    $CC.bridges = CC.bridges - \{(x, y), (y, x)\}$
                    $x = y$
                    $y = y.\pi$
                until $x.ear = True$
                $CC.ears.insert(ear)$
            end if
        end for
    end for
end function
```

Some notes from this Wikipedia article:

- An open ear decomposition or a proper ear decomposition is an ear decomposition in which the two endpoints of each ear after the first are distinct from each other

- A graph is k-vertex-connected if the removal of any (k − 1) vertices leaves a connected subgraph. A graph is 2-vertex-connected if and only if it has an open ear decomposition.

- If a graph is not 2-vertex-connected, there is at least one vertex that by removing it, the graph will become disconnected. such vertices are called **cut vertices** or **articulation points**

- A graph is k-edge-connected if the removal of any (k − 1) edges leaves a connected subgraph. A graph is 2-edge-connected if and only if it has an ear decomposition

cut vertices or **articulation points** are those vertices which are on the end-point of a cycle apart from E_1 or bridges.

```
function FIND-CUT-VERTICIES($G$)
    EEAR-DECOMPOSITION($G$)
    for all $CC \in G.CCs$ do
        CC.cut-verticies = $\emptyset$
        for all $ear \in \{CC.ears - \{CC.ears.front()\}\}$ do                ▷ $O(V)$
            cyle = $ear.front() == ear.back()$
            if cycle then
                CC.cut-verticies = CC.cut-verticies $\cup$ $ear.front()$
            end if
        end for
        visited-bridges = $\emptyset$
        for all $(u, v) \in CC.bridges \wedge (u, v) \notin$ visited-bridges do      ▷ $O(E)$
            if $CC.adj[u].size() > 1$ then                          ▷ $v \in CC.adj[u]$
                CC.cut-verticies = CC.cut-verticies $\cup$ $u$
            end if
            if $CC.adj[v].size() > 1$ then                          ▷ $u \in CC.adj[v]$
                CC.cut-verticies = CC.cut-verticies $\cup$ $v$
            end if
            visited-bridges = visited-bridges $\cup$ $\{(u, v), (v, u)\}$
        end for
    end for
end function
```

Chapter 2

Algorithms Involving Sequences and Sets

2.1 String Matching

Suppose we have $A = a_0a_1 \dots a_{n-1}$ and $B = b_0b_1 \dots b_{m-1}$ are two strings. We assume $m \leq n$. We want to determine whether B is a substring of A. We define $next[i] = j$ for all $0 \leq i \leq m-1$ as the maximum index j such as the first $j+1$ characters should be equal to the last $j+1$ ones:

$$next[i] = \begin{cases} \max\limits_{j=0}^{i-2} (j) \implies b_0b_1 \dots b_j = b_{i-j-1}b_{i-j} \dots b_{i-1} & i \geq 2 \\ -1 & i = 1 \vee \text{j doesn't exist} \\ -2 & i = 0 \end{cases}$$

Note that the minimum index of suffix is $x = i - j - 1$ which is calculated by $(i-1-x)+1 = (j-0)+1$. The maximum index for prefix string $(b_0 \dots b_j)$ can be $i-2$ and the minimum index for postfix string $(b_{i-j-1} \dots b_{i-1})$ can be 1.

Finding prefix and postfix strings of B is like finding substring $Q = B[1..m-2]$ in $P = B[0..m-2]$. So P represents prefix and Q represents suffix.

We use mathematical induction to calculate $next[i]$. The base case is $i = 1$ which is trivial. We assume $next[k]$ is true for all $k < i$. Now consider $next[i]$. Assume $j = next[i-1]$. So we have:

$$\boxed{b_0 \dots b_j = b_{i-j-2} \dots b_{i-2}}$$

There are two scenarios:

1. $b_{j+1} = b_{i-1}$ In that case $next[i] = next[i-1] + 1 = j + 1$ because $b_0 \dots b_jb_{j+1} = b_{i-j-2} \dots b_{i-2}b_{i-1}$

2. $b_{j+1} \neq b_{i-1}$ In that case we cannot extend $next[i-1]$ by 1. Suppose $k = next[j+1]$ which implies $b_0 \dots b_k = b_{j-k} \dots b_j$. According to above

equation in box $b_j = b_{i-2}, b_{j-1} = b_{i-3}, \ldots$ So we can say $b_0 \ldots b_k = b_{i-k-2} \ldots b_{i-2}$. This is like scenario 1. We need to compare b_{k+1} to b_{i-1} and if they are equal then $next[i] = next[j+1] + 1 = k + 1$

The implementation of above algorithm:

```
vector<int> calculateNext(const string& B)
{
  vector<int> next(B.length());
  next[0] = -2;
  next[1] = -1;
  int i, j;
  for (i = 2; i < B.length(); ++i)
  {
    j = next[i - 1] + 1;
    for (; j >= 0 && B[j] != B[i - 1]; j = next[j] + 1);
    next[i] = j;
  }
  return next;
}

int findSubstring(const string& A, const string& B)
{
  const auto next = calculateNext(B);
  int i, j;
  for (i = 0, j = 0; i < A.length() && j < B.length();)
  {
    if (A[i] == B[j])
      ++i, ++j;
    else if ((j = next[j] + 1) < 0)
      ++i, j = 0;
  }
  if (j == B.length())
    return i - B.length();
  return -1;
}
```

Notice the edge cases. For example if $A = ababc$ and $B = abc$. When $i = 2$ and $j = 2$, we know that $A[2] \neq B[2]$ so we must use $next[2] = -1$ in the next iteration by comparing $A[2]$ with $B[next[2] + 1] = B[0]$. In other words we shouldn't increase i in this iteration. On the other hand, if $A = abfoo$ and $B = abc$, when $i = 2$ and $j = 2$, in the next iteration we must compare $A[2]$ with $B[next[2] + 1] = B[0]$. Like previous example we shouldn't increase i in this iteration. But in the next iteration $A[2] \neq B[0]$ and $next[0] = -2$. So we must increase i and set $j = 0$.

Let's consider $findSubstring$. When $A[i] = B[j]$, we increase both i and j so we do comparison for $A[i]$ just once. If $A[i] \neq B[j]$, we must compare $A[i]$ more than once. Doing $j = next[j] + 1$ can be repeated at most j times. So we can say we do comparison for $A[i]$ at most j times. But $B[0..j]$ causes we forward A, j times to reach $A[i]$. We can use amortized analysis and put cost 2 for going foward in A and put cost 0 for going backward in B. So the total running time of $finSubstring$ is $O(2n) = O(n)$.

Now consider $calculateNext$. When we want to calculate $next[i]$. We know that $j = next[i-1]+1$; if $B[i-1] = B[j]$ we do comparison for $B[i-1]$ only once otherwise the number of comparisons for $B[i-1]$ is more than once and at most j times. If $j \geq 0$ we can say there are $j+1$ indices less than i such as k that for calculating $next[k]$ we went forward in B with cost 1. So we can assign cost 2 for going forward and cost 0 for going backward (when $B[i-1] \neq B[j]$). Note that if we consume all j ($j < 0$), the indices greater than i don't use this j. So the running time is $O(2m) = O(m)$.

For an application of this algorithm see TopCoder SRM 428, ThePalindrome. Basically we want to add the minimum number of characters to the end of a string to make it palindrome.