Binary Search

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

- Surprisingly powerful technique! You should have seen binary search in the context of searching an array before. For us, the power comes from binary searching on non-obvious functions instead.
- **Key problem:** Given a monotone function, find the largest/smallest x such that $f(x)$ is less than/greater than/equal to/... y.

Aside: Binary Search Implementation

- Hands up if you've ever messed up a binary search implementation.
- I think binary search is notorious for having annoying off-by-1s and possible infinite loops.
- Many ways to implement so pick one you're confident you can code with no thought. I'll present the one I use which I find avoids all these annoying corner cases.

Aside: Binary Search Implementation


```cpp
#include <bits/stdc++.h>
using namespace std;


// Find the smallest X such that f(X) is true;
int binarysearch(function<bool(int)> f) {
    int lo = 0;
    int hi = 100000;
    int bestSoFar = -1;
    // Range [lo, hi];
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (f(mid)) {
            bestSoFar = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    return bestSoFar;
}
```

- Decision problems are of the form

  Given some parameters including X, can you ...

- Optimisation problems are of the form:

  What is the smallest X for which you can ...

- An optimisation problem is typically *much* harder than the corresponding decision problem, because there are many more choices
- Can we reduce (some) optimisation problems to decision problems?

- Let $f(X)$ be the outcome of the decision problem for a given X, so f is an integer valued function with range $\{0, 1\}$.
- It is sometimes (but not always) the case in such problems that increasing X does not make it any harder for the condition to hold (i.e: that if the condition holds with X then it also holds with $X + 1$).
- Thus f is all 0's up to the first 1, after which it is all 1's.
- This is a monotonic function, so we can use binary search!
- This technique of binary searching the answer, that is, finding the smallest X such that $f(X) = 1$ using binary search, is often called *discrete* binary search.
- Overhead is just a factor of $O(\log A)$ where A is the range of possible answers.

- **Problem Statement:** You have a bar of chocolate with N squares, each square has a tastiness t_i. You have K friends. Break the bar into K contiguous pieces. The overall happiness of the group is the minimum total tastiness of any of these K pieces. What's the maximum overall happiness you can achieve?
- **Input Format:** First line, 2 integers, N, K with $1 \leq K \leq N \leq 1,000,000$. The next line will contain N integers, t_i, the tastiness of the ith piece. For all i, $1 \leq i \leq 100,000$.

Example: Gravy Surprise

- **Sample Input:**

  ```
  5 2
  9 7 3 7 4
  ```

- **Sample Output:**

  ```
  14
  ```

- **Explanation:** Break the bar into the first 2 squares and the last 3 squares.

Example: Gravy Surprise

- It is worth trying to approach the minimization problem directly, just to appreciate the difficulty.
- The problem is there's no greedy choices you can make. It's impossible to determine where the first cut should end. You can try a DP but the state space is large.
- We are asked to maximize the minimum sum of the K pieces.
- Let's turn this into asking about a decision problem.
- Define $b(X)$ to be True iff we can split the bar into K pieces, each with sum at least X.
- Then the problem is asking for the largest X such that $b(X)$ is True.
- **Note:** We define it to be *at least* X. This makes it monotone. If we instead defined it as *exactly* X then the function is too messy to be useful.

- **Rephrased Problem:** Define $b(X)$ to be True iff we can split the bar into K pieces, each with sum at least X. What is the largest X such that $b(X)$ is True?
- **Key(and trivial) Observation:** $b(X)$ is non-increasing.
- So we can binary search over $b(X)$. Hence to find the maximum such X, it suffices to be able to calculate $b(X)$ quickly.
- **New Problem:** Can I split the bar into K pieces, each with sum at least A?

- **New Problem:** Can I split the bar into K pieces, each with sum at least A?
- Note that we can rephrase this into a maximization question. Given each piece has sum at least A, what is the maximum number of pieces I can split the bar into?
- Let's try going one piece at a time. What should the first piece look like?
- **Key Observation:** It should be the minimum length possible while having total $\geq A$.
- This applies for all the pieces.
- So to get the maximum number of pieces needed, we sweep left to right making each piece as short as possible.

Example: Gravy Surprise


```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 1000000;
int N, K; long long bar[MAXN];

bool canDo(long long A) {
    long long cPiece = 0;
    int nPieces = 0;
    for (int i = 0; i < N; i++) {
        cPiece += bar[i];
        if (cPiece >= A) {
            nPieces++;
            cPiece = 0;
        }
    }
    return nPieces >= K;
}
```

```
int main() {
    scanf("%d %d", &N, &K);
    for (int i = 0; i < N; i++) scanf("%lld", &bar[i]);
    long long lo = 1;
    long long hi = 1e12;
    long long ans = -1;
    while (lo <= hi) {
        long long mid = (lo + hi) / 2;
        // Trying to find the highest value that is feasible:
        if (canDo(mid)) {
            ans = mid;
            lo = mid + 1;
        } else {
            hi = mid - 1;
        }
    }
    printf("%lld\n", ans);
}
```

- **Complexity?** $O(N \log A)$ where A is max answer.
- This problem and solution is very typical of binary search problems.
- To start with, you are asked to maximize a value.
- But we can rephrase it into maximizing a value that satisfies a decision problem! In forming the decision problem, you ask if the answer could be *at least* A, not just exactly A.
- Now with the minimum tastiness of each bar fixed, you now switch to trying to maximize the number of pieces you can make. And this can be greedied since we know how small we can make each piece.
- Notice why fixing A made the problem easier. Because we had one less parameter influencing our choices and we could make greedy decisions now.

- One of the most common places binary search appears is in problems that ask us to maximize the minimum of something (or minimize the maximum of something).
- Another way to see if it's useful is just to see if the quantity you are minimizing is monotone.
- And this is very common! Usually, you are told to minimize a value because the problem only gets easier if it increases.
- Until you get the hang of it, it's worth just always trying to apply it.
- At worst, the decision problem can't be any harder than the optimization problem. (though it may lead you down a dead end).

Aside: Ternary Search

- Ternary search also exists. It applies to finding the maximum of a function that *strictly* increases to a peak, stays the same, then *strictly* decreases. Note the strictlys.
- Instead of splitting the range in 2, we instead now split it into 3 by querying 2 points. At each step we discard one of the thirds based on comparison of the 2 points.
- Alternatively, we can usually binary search the derivative. Usually this is the discrete form of the derivative (binary search on $h(x) := f(x+1) - f(x)$).
- Appears much less often so won't talk about it more but it is a useful thing to know exists.
- Exercise left to the reader to figure it out!

- **Problem Statement:** You have just created a robot that will revolutionize RoboCup forever. Well 1D RoboCup at least.
  The robot starts at position 0 on a line and can perform three types of moves:
  - **L**: Move left by 1 position.
  - **R**: Move right by 1 position.
  - **S**: Stand still.

  Currently the robot already has a loaded sequence of instructions.
  You need to get the robot to position X. To do so, you can replace a single **contiguous** subarray of the robot's instructions. What is the shortest subarray you can replace to get the robot to position X?

- **Input Format:** First line, 2 integers, N, X, the length of the loaded sequence and the destination. $1 \leq |X| \leq N \leq 200,000$. The next line describes the loaded sequence.
- **Sample Input:**

  ```
  5 -4
  LRRLR
  ```

- **Sample Output:**

  ```
  4
  ```

- **Explanation:** You can replace the last 4 instructions to get the sequence `LLLLS`.

- How would one do the problem directly?
- There is an $O(N^2)$ by trying all subsegments but we can't do better if we need to try all subsegments.
- Okay, well we can try binary searching now. How?
- **Key Observation:** If we can redirect the robot correctly by replacing M instructions, then we can also do so by replacing $M + 1$ instructions. Why?
- Let's turn this into a decision problem. $b(M)$ is true if...?

- $b(M)$ is true if we can correctly redirect the robot by replacing a subsegment of size M.
- We need to do this in around $O(N)$ now. How? It's worth considering how to do it in $O(1)$ if I tell you exactly what subsegment to replace.
- Reduces to, given a list of $N - M$ instructions, can I add M more instructions to get the robot to position X.

- **Key Observation:** In M instructions, the robot can move to every square within distance M.
- So we are reduced to finding if there is a subsegment of size M such that its removal leaves the robot within distance M of X.
- Now we just need to find where the robot is after the removal of each subsegment of size M.
- For this, we precompute a cumulative sum array from the front and back, where `L` is -1, `S` is 0 and `R` is 1.
- Then the position of the robot after removing the segment $[L, L+M)$ is `sum[0,..,L-1] + sum[L+M,..,N-1]`.

Example: Robot Moves


```
#include <bits/stdc++.h>
using namespace std;

const int MAXN = 200000;
int N, X;
char moves[MAXN+5];
// cumFront[i] = sum moves[1..i]
int cumFront[MAXN+5];
// cumBack[i] = sum moves[i..N]
int cumBack[MAXN+5];

void precomp() {
    vector<int> moveDeltas(N+5, 0);
    for (int i = 1; i <= N; i++) {
        if (moves[i] == 'L') moveDeltas[i] = -1;
        if (moves[i] == 'S') moveDeltas[i] = 0;
        if (moves[i] == 'R') moveDeltas[i] = 1;
    }
    for (int i = 1; i <= N; i++)
        cumFront[i] = cumFront[i-1] + moveDeltas[i];
    for (int i = N; i >= 1; i--)
        cumBack[i] = cumBack[i+1] + moveDeltas[i];
}
```

Example: Robot Moves


```
bool canDo(int A) {
    for (int i = 1; i+A-1 <= N; i++) {
        // try replacing [i, i+A-1]
        int posAfterCut = cumFront[i-1] + cumBack[i+A];
        if (abs(posAfterCut - X) <= A) return true;
    }
    return false;
}

int main() {
    scanf("%d %d", &N, &X);
    for (int i = 1; i <= N; i++) scanf(" %c", &moves[i]);
    precomp();
    int lo = 0;
    int hi = N;
    int ans = -1;
    while (lo <= hi) {
        int mid = (lo + hi) / 2;
        // Trying to find the lowest value that is feasible:
        if (canDo(mid)) {
            ans = mid;
            hi = mid - 1;
        } else {
            lo = mid + 1;
        }
    }
    printf("%d\n", ans);
    return 0;
}
```

Example: Robot Moves

- **Complexity:** $O(N \log N)$.
- Hopefully you can see the similarities between this example and the earlier example.
- Again, we started with a problem where approaching it directly was too slow.
- But the problem naturally could be rephrased as finding the minimum M such that a decision problem $b(M)$ was true.
- So from that point onwards we only consider the decision problem.
- This still required some work but was more direct. The idea of trying all subsegments of length M is relatively straight forward. From that point on it was just trying to optimize this problem with data structures.