1. Case Study

1.1. Performance Optimization

We show several examples to demonstrate how EDITLORD improve performance optimization.

Given slow code

```cpp
#include <bits/stdc++.h>

using namespace std;

long long n;long long k;
vector<long long> vec;
long long meme[100007];

long long solve(long long index){
  if(index==n-1){
    return 0;
  }
  if(meme[index]!=-1){
    return meme[index];
  }
  long long ret = 10000000000000000;
  long long sol = 10000000000000000;
  for(long long i = 1 ;i<=k ; i++){
    if(index+i<n){
      ret = abs (vec[index]-
       ↪ vec[index+i]) +
       ↪ solve(index+i);
    }
    sol = min(sol,ret);
  }
  return meme[index] = sol;
}

int main(){
  cin>>n;
  cin>>k;
  memset(meme,-1,sizeof(meme));
  long long num;
  for(long long i = 0 ; i<n ; i++){
    cin>>num;
    vec.push_back(num);
  }
  cout<<solve(0)<<endl;
  return 0;
}
```

EDITLORD output

```
The given code describe the following
 ↪ problem: Find the minimum cost to reach
 ↪ the last index of a sequence of integers,
 ↪ where the cost is defined as the absolute
 ↪ difference between consecutive integers.
The input specification is: The programs take
 ↪ two integers n and k as input, where n is
 ↪ the number of integers in the sequence
 ↪ and k is the maximum jump length. Then, n
 ↪ integers are read, representing the
 ↪ sequence.
The output specification is: Output the
 ↪ minimum cost to reach the last index from
 ↪ the first index.
Following editing rules should be applied:
switch from multiple nested loops to single
 ↪ linear iterations
switch from recursive function calls to
 ↪ optimized iterative data handling methods
The corresponding fast code is:
[FAST CODE]
```

```cpp
#include<bits/stdc++.h>

using namespace std;

long long int dp[100007],a[100007];

int main(){
  long long int n,k,i,j,x,y,z;
  cin>>n>>k;
  for(i=0;i<n;i++){
    cin>>a[i];
  }
  dp[0]=0;
  for(i=1;i<n;i++){
    x=10000000000007;
    for(j=i-1;j>=i-k&&j>=0;j--){
      x=min(x,dp[j]+abs(a[i]-a[j]));
    }
    dp[i]=x;
  }
  cout<<dp[n-1];
}
```

```
[/FAST CODE]
```

Figure 1: Performance optimization example 1.

Given slow code

```
#include <bits/stdc++.h>

using namespace std;

int n ,k;
vector<int>vi;
long long mem[200005];
bool vis[200005];

long long solve(int indx ){
  if(indx == n-1)
    return 0;
  if(vis[indx])
    return mem[indx];
  vis[indx] = true;
  mem[indx] = 1e13;
  for(int i = indx+1; i <=min(indx+k
  ↪  , n-1) ; i++){
    mem[indx] = min(mem[indx] ,
     ↪  solve(i ) + abs(vi[indx] -
     ↪  vi[i]));
  }
  return mem[indx];
}

int main(){
  cin>>n>>k;
  vi.resize(n);
  for(int i = 0; i <n ; i++)
    cin>>vi[i];
  cout<<solve(0);
  return 0;
}
```

EDITLORD output

```
The given code describe the following
 ↪  problem: Calculate the minimum cost to
 ↪  reach the last index in a list of
 ↪  integers, where each step can jump up to
 ↪  k positions.
The input specification is: The programs take
 ↪  two integers n and k as input, where n is
 ↪  the number of integers in the list and k
 ↪  is the maximum jump length. Then, n
 ↪  integers are read, representing the cost
 ↪  associated with each index.
The output specification is: Output the
 ↪  minimum cost to reach the last index from
 ↪  the first index.
Following editing rules should be applied:
switch from multiple nested loops for
 ↪  counting elements to direct arithmetic
 ↪  calculation
switch from multiple nested loops to a
 ↪  compact function
switch from cout to printf
switch from cin to scanf
switch from dynamic memory allocation to
 ↪  static memory allocation
The corresponding fast code is:
[FAST CODE]
#include<bits/stdc++.h>

using namespace std;

int n,k;
long long dp[100005];
long long arr[100005];

int main(){
  scanf("%d%d",&n,&k);
  for(int i=1;i<=n;i++)
  ↪  scanf("%lld",&arr[i]);
  for(int i=2;i<=n;i++){
    dp[i]=1e18;
    for(int j=max(i-k,1);j<i;j++){
      dp[i]=min(dp[i],
       ↪  dp[j]+abs(arr[i]-arr[j]));
    }
  }
  printf("%lld",dp[n]);
}
[/FAST CODE]
```

Figure 2: Performance optimization example 2.

1.2. Decompilation

We show several examples to demonstrate how EDITLORD improves the readability of machine-decompiled code.

Given machine decompiled code

```
void func0(char *param_1,long
↪  param_2)
{
  char cVar1;
  long lVar2;
  ulong uVar3;
  char *pcVar4;
  int iVar5;
  bool bVar6;

  lVar2 = -1;
  pcVar4 = param_1;
  do {
    if (lVar2 == 0) break;
    lVar2 = lVar2 + -1;
    cVar1 = *pcVar4;
    pcVar4 = pcVar4 + 1;
  } while (cVar1 != '\0');
  iVar5 = ~(uint)lVar2 - 1;
  if (0 < iVar5) {
    uVar3 = 0;
    do {
      cVar1 = param_1[uVar3];
      if ((byte)(cVar1 + 0x9fU) <
      ↪  0x1a) {
        cVar1 = cVar1 + -0x20;
      }
      else if ((byte)(cVar1 + 0xbfU)
      ↪  < 0x1a) {
        cVar1 = cVar1 + ' ';
      }
      *(char *)(param_2 + uVar3) =
      ↪  cVar1;
      bVar6 = uVar3 != ~(uint)lVar2 -
      ↪  2;
      uVar3 = uVar3 + 1;
    } while (bVar6);
  }
  *(undefined *)(param_2 + iVar5) =
  ↪  0;
  return;
}
```

EDITLORD output

```
The given code describe the following
↪  problem: Convert a string from one case
↪  (upper or lower) to another, handling
↪  special characters appropriately.
The input specification is: The programs take
↪  a string as input, which may contain
↪  uppercase and lowercase letters, and a
↪  buffer to store the converted string.
The output specification is: Output the
↪  converted string, where all letters are
↪  converted to lowercase, while preserving
↪  the case of special characters.
Following editing rules should be applied:
switch from arbitrary pointer checks to
↪  clearer termination conditions
switch from complex pointer arithmetic to
↪  clear variable assignments
switch from indirect variable assignments to
↪  direct return statements
switch from complex multi-step declarations
↪  to clear single declarations
switch from complex logic to simple loops
switch from complex complex if-else structure
↪  to simplified conditional logic
switch from implicit return values to
↪  explicit return values
switch from cryptic variable names to
↪  descriptive variable names
The corresponding source code is:
[ORIGINAL SOURCE CODE]
static void
func0 (char *str, char *buf)
{
  int i, len = strlen (str);
  for (i = 0; i < len; i++) {
    if (str[i] >= 'A' && str[i] <= 'Z')
      buf[i] = str[i] + ' ';
    else if (str[i] >= 'a' && str[i] <= 'z')
      buf[i] = str[i] - ' ';
    else
      buf[i] = str[i];
  }
  buf[len] = '\0';
}
[/ORIGINAL SOURCE CODE]
```

Figure 3: Decomilation example 1.

Given machine decompiled code

```
undefined8 func0(char *param_1,char
↪ *param_2)
{
  char cVar1;
  int iVar2;
  uint uVar3;
  char *pcVar4;
  char *pcVar5;
  long lVar6;
  size_t __n;
  bool bVar7;

  lVar6 = -1;
  pcVar4 = param_2;
  do {
    if (lVar6 == 0) break;
    lVar6 = lVar6 + -1;
    cVar1 = *pcVar4;
    pcVar4 = pcVar4 + 1;
  } while (cVar1 != '\0');
  uVar3 = ~(uint)lVar6;
  iVar2 = uVar3 - 1;
  pcVar4 = (char
   ↪ *)malloc((long)(int)(uVar3 +
   ↪ iVar2));
  if (0 < iVar2) {
    lVar6 = (long)iVar2;
    __n = 0;
    do {
      strncpy(pcVar4,param_2 +
       ↪ __n,lVar6 - __n);
      strncpy(pcVar4 + (lVar6 -
       ↪ __n),param_2,__n);
      pcVar4[lVar6] = '\0';
      pcVar5 =
       ↪ strstr(param_1,pcVar4);
      if (pcVar5 != (char *)0x0) {
        free(pcVar4);
        return 1;
      }
      bVar7 = __n != uVar3 - 2;
      __n = __n + 1;
    } while (bVar7);
  }
  free(pcVar4);
  return 0;
}
```

EDITLORD output

```
The given code describe the following
↪ problem: Determine if a string is a
↪ rotation of another string by checking if
↪ any substring of the second string is a
↪ substring of the first string.
The input specification is: The programs take
↪ two strings as input: the first string is
↪ the original string, and the second
↪ string is the potential rotation.
The output specification is: Return a boolean
↪ value indicating whether the second
↪ string is a rotation of the first string.
Following editing rules should be applied:
switch from ambiguous function signatures to
↪ clear function signatures
switch from complex pointer arithmetic to
↪ clear variable assignments
switch from indirect variable assignments to
↪ direct return statements
switch from complex logic to simple loops
switch from complex loops to clear
↪ single-responsibility sections
switch from complex nested conditionals to
↪ clearer sequential logic
switch from complex variable operations to
↪ direct assignments
switch from redundant checks to
↪ straightforward boolean comparisons
The corresponding source code is:
[ORIGINAL SOURCE CODE]
static int
func0(const char *s1, const char *s2)
{
  char *s;
  size_t len, i;
  len = strlen(s2);
  s = malloc(len * 2 + 1);
  if (s == NULL) return (0);
  for (i = 0; i < len; i++) {
    strncpy(s, s2 + i, len - i);
    strncpy(s + len - i, s2, i);
    s[len] = '\0';
    if (strstr(s1, s) != NULL) {
      free(s);
      return (1);
    }
  }
  free(s);
  return (0);
}
[/ORIGINAL SOURCE CODE]
```

Figure 4: Decomilation example 2.

1.3. Security Hardening

We show several examples to demonstrate how EDITLORD hardens the vulnerable code.

Given vulnerable code

```
#include <stdlib.h>
#include <string.h>

char*
↪ allocate_buffer_and_write(size_t
↪ buf_size) {
    char* buf =
    ↪ (char*)malloc(buf_size);
    memset(buf, 'A', 10);
    memset(buf + buf_size - 10, 'A',
    ↪ 10);
    return buf;
}
```

EDITLORD output

```
The program defines a function that allocates
↪ a buffer of a specified size and
↪ initializes its contents by writing 'A'
↪ characters at both the start and end of
↪ the buffer. The goal is to ensure that
↪ the buffer is filled correctly without
↪ exceeding its allocated size, thereby
↪ preventing potential buffer overflows.
The input specification is: The function
↪ takes a single input parameter, buf_size,
↪ which specifies the size of the buffer to
↪ be allocated. This size determines how
↪ much memory will be reserved for the
↪ buffer.
The output specification is: The function
↪ returns a pointer to the allocated and
↪ initialized buffer. If the allocation is
↪ successful, the buffer will contain 'A'
↪ characters at the specified positions;
↪ otherwise, it may return NULL if memory
↪ allocation fails.
Following editing rules should be applied:
switch from no checks on function return
↪ values to check function return value
The corresponding secure code is:
[SECURE CODE]
char* allocate_buffer_and_write(size_t
↪ buf_size) {
    char* buf = (char*)malloc(buf_size);
    if (!buf) {
        return 0;
    }
    memset(buf, 'A', 10);
    memset(buf + buf_size - 10, 'A', 10);
    return buf;
}
[/SECURE CODE]
```

Figure 5: Secure hardening example 1.

Given vulnerable code

```c
#include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>
#include <string.h>

char *get_user(sqlite3 *db, const
↪ char *username, const char
↪ *password) {
    sqlite3_stmt *stmt;
    char query[256];
    char *userInfo = NULL;

    snprintf(query, sizeof(query),
    ↪ "SELECT userInfo FROM users
    ↪ WHERE username = '%s' AND
    ↪ password = '%s'", username,
    ↪ password);

    if (sqlite3_prepare_v2(db, query,
    ↪ -1, &stmt, NULL) !=
    ↪ SQLITE_OK) {
        fprintf(stderr, "Failed to
        ↪ prepare statement: %s\n",
        ↪ sqlite3_errmsg(db));
        return NULL;
    }

    if (sqlite3_step(stmt) ==
    ↪ SQLITE_ROW) {
        const unsigned char *result
        ↪ =
        ↪ sqlite3_column_text(stmt,
        ↪ 0);
        userInfo = strdup((const char
        ↪ *)result);
    }

    sqlite3_finalize(stmt);
    return userInfo;
}
```

EDITLORD output

```
The program aims to retrieve user information
↪ from a SQLite database based on a
↪ provided username and password. It
↪ connects to the database, executes a
↪ query to find the user, and returns the
↪ user's information if found. The program
↪ ensures that the database interactions
↪ are secure and that user input is handled
↪ appropriately to prevent SQL injection
↪ vulnerabilities.
The input specification is: The program takes
↪ three command-line arguments: the
↪ database filename, a username, and a
↪ password. These inputs are used to query
↪ the database for matching user
↪ credentials.
The output specification is: The program
↪ outputs the user's information if a
↪ matching user is found in the database,
↪ or an error message if no user matches
↪ the provided credentials or if an error
↪ occurs during database operations.
Following editing rules should be applied:
switch from direct SQL string interpolation
↪ to use of parameterized logic
The corresponding secure code is:
[SECURE CODE]
char *get_user(sqlite3 *db, const char
↪ *username, const char *password) {
  sqlite3_stmt *stmt;
  char *userInfo = NULL;
  const char *query = "SELECT userInfo FROM
  ↪ users WHERE username = ? AND password =
  ↪ ?";
  if (sqlite3_prepare_v2(db, query, -1,
  ↪ &stmt, NULL) != SQLITE_OK) {
    fprintf(stderr, "Failed to prepare
    ↪ statement: %s\n",
    ↪ sqlite3_errmsg(db));
    return NULL;
  }
  sqlite3_bind_text(stmt, 1, username, -1,
  ↪ SQLITE_STATIC);
  sqlite3_bind_text(stmt, 2, password, -1,
  ↪ SQLITE_STATIC);
  if (sqlite3_step(stmt) == SQLITE_ROW){
    const unsigned char *result =
    ↪ sqlite3_column_text(stmt, 0);
    userInfo = strdup((const char *)result);
  }
  sqlite3_finalize(stmt);
  return userInfo;
}
[/SECURE CODE]
```

Figure 6: Secure hardening example 2.

Given vulnerable code

```
#include <string.h>

void trim_spaces(char* str) {
    int len = strlen(str);
    while (str[len - 1] == ' ') {
        str[len - 1] = '\0';
        len--;
    }
}
```

EDITLORD output

```
The program defines a function that trims
 ↪  trailing spaces from a given string. It
 ↪  modifies the string in place, ensuring
 ↪  that any whitespace characters at the end
 ↪  are removed, leaving the rest of the
 ↪  string intact.
The input specification is: The function
 ↪  takes a single string input, which is a
 ↪  character array that may contain trailing
 ↪  spaces that need to be removed.
The output specification is: The function
 ↪  does not return any value; instead, it
 ↪  modifies the input string by removing any
 ↪  trailing spaces, effectively shortening
 ↪  the string if necessary.
Following editing rules should be applied:
switch from unvalidated input handling to
 ↪  check for NULL pointers before access
switch from unvalidated input handling to
 ↪  check for buffer overflows on memory
 ↪  accesses
The corresponding secure code is:
[SECURE CODE]
void trim_spaces(char* str) {
    int len = strlen(str);
    while (len > 0 && str[len - 1] == ' ') {
        str[len - 1] = '\0';
        len--;
    }
}
[/SECURE CODE]
```

Figure 7: Secure hardening example 3.

Given vulnerable code

```c
#include <stdio.h>
#include <stdlib.h>

char* read_stdin_to_buffer(int
↪ buf_size) {
    char* buf =
     ↪ (char*)malloc(buf_size);
    gets(buf);
    return buf;
}
```

EDITLORD output

```
The program aims to read a line of input from
 ↪ the standard input (stdin) into a
 ↪ dynamically allocated buffer, ensuring
 ↪ that the input does not exceed the
 ↪ specified buffer size. It is designed to
 ↪ prevent buffer overflow vulnerabilities
 ↪ by limiting the number of characters read
 ↪ based on the buffer's capacity.
The input specification is: The function
 ↪ takes an integer input representing the
 ↪ maximum size of the buffer, which
 ↪ dictates how many characters can be
 ↪ safely read from stdin.
The output specification is: The function
 ↪ returns a pointer to the buffer
 ↪ containing the user input, ensuring that
 ↪ the input is properly null-terminated and
 ↪ does not exceed the allocated buffer
 ↪ size.
Following editing rules should be applied:
switch from unvalidated input handling to
 ↪ check for buffer overflows on memory
 ↪ accesses
switch from no error handling to check for
 ↪ NULL pointers after memory allocation
The corresponding secure code is:
[SECURE CODE]
char* read_stdin_to_buffer(int buf_size) {
    char* buf = (char*)malloc(buf_size);

    fgets(buf, buf_size, stdin);

    return buf;
}
[/SECURE CODE]
```

Figure 8: Secure hardening example 4.

Given vulnerable code

```
#include <stdlib.h>

int get_price_at_index(int* prices,
↪ int size, char* index_str) {
    int index = atoi(index_str);
    return prices[index];
}
```

EDITLORD output

```
The program defines a function that retrieves
↪ the price of an item at a specified index
↪ from an array of prices. The index is
↪ provided as a string, which is converted
↪ to an integer. The function returns the
↪ price at the corresponding index if it's
↪ valid, or an error if the index is out of
↪ bounds.
The input specification is: The function
↪ takes three inputs: an array of integers
↪ representing prices, the size of the
↪ array, and a string representing the
↪ index of the price to retrieve.
The output specification is: The function
↪ returns the price at the specified index
↪ if it is valid; otherwise, it returns an
↪ error message indicating that the index
↪ is out of bounds.
Following editing rules should be applied:
switch from direct parsing to implement
↪ comprehensive validation checks for
↪ character handling
The corresponding secure code is:
[SECURE CODE]
int get_price_at_index(int* prices, int size,
↪ char* index_str) {
    int index = atoi(index_str);
    if (index < 0 || index >= size) {
        return -1; // Error: index out of
        ↪  bounds
    }
    return prices[index];
}
[/SECURE CODE]
```

Figure 9: Secure hardening example 5.