UPES
UNIVERSITY OF TOMORROW

Name -Aditya Thakur

Sapid-590012258

Course- Design and analysis of algorithms

Submitted to – Aryan gupta

github repository link

https://github.com/aditya0001-cyber/DAA-LAB_-aditya-Thakur-_590012258.git

1) C source code :

```
#define _POSIX_C_SOURCE 199309L
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <stdint.h>
#include <string.h>
#include <math.h>

int binary_search(int *arr, int n, int target) {
 int left =0,right =n-1;
   while (left <= right) {
      int mid = left + (right - left) / 2;
      if (arr[mid] == target) return mid;
      else if (arr[mid] < target) left = mid + 1;
      else right = mid - 1;
```

```c
    }
    return -1;
}

static inline uint64_t timespec_to_ns(const struct timespec *t) {
    return (uint64_t)t->tv_sec * 1000000000ULL + (uint64_t)t->tv_nsec;
}

int main() {
 const int sizes[5]={1000,5000,10000,50000,100000};
    srand(123456789);

    printf("case_type,case_id,n,reps,time_ns\n");

    int case_id = 1;

    for (int category = 0; category < 3; ++category) {
        for (int si = 0; si < 5; ++si) {
            int n = sizes[si];
            int *arr = (int*)malloc(sizeof(int)*n);
            if (!arr) {
                fprintf(stderr, "malloc failed for n=%d\n", n);
                return 2;
            }
            for (int i = 0; i < n; ++i) {
                arr[i] = (i/3) - (n/2);
            }

            int target;
            if (category == 0) {
                int mid = n/2;
                target = arr[mid];
            } else if (category == 1) {
                target = arr[0] - 1;
            } else {
                int idx = (rand() % n);
                target = arr[idx];
            }

            double logn = (n>1) ? log((double)n)/log(2.0) : 1.0;
            long reps = (long)(10000000.0 / (logn + 1.0));
            if (reps < 1) reps = 1;
            if (reps > 20000000) reps = 20000000;
```

```
        struct timespec t0, t1;
        clock_gettime(CLOCK_MONOTONIC, &t0);
        volatile int sink = 0;
        for (long r = 0; r < reps; ++r) {
            int found = binary_search(arr, n, target);
            sink ^= found;
        }
        clock_gettime(CLOCK_MONOTONIC, &t1);
        uint64_t dt = timespec_to_ns(&t1) - timespec_to_ns(&t0);

        const char *catname = (category==0) ? "best" : (category==1) ? "worst" :
"average";
        printf("%s,%d,%d,%ld,%llu\n", catname, case_id, n, reps, (unsigned long
long)dt);

        free(arr);
        case_id++;
      }
    }

    return 0;
}
```

2) Summary of the 15 test cases

All arrays are sorted and generated to include negative numbers and duplicates (arr[i] = (i/3) - (n/2)), so duplicates and negative values are present.

Input sizes used: n = {1000, 5000, 10000, 50000, 100000}.

Cases are run in this order:

Best-case (5):

- case 1: best, n=1000 — target at middle index (fastest)
- case 2: best, n=5000
- case 3: best, n=10000
- case 4: best, n=50000
- case 5: best, n=100000

Worst-case (5):

- case 6: worst, n=1000 — target **absent** (value less than min), forces full search path
- case 7: worst, n=5000
- case 8: worst, n=10000
- case 9: worst, n=50000
- case 10: worst, n=100000

Average-case (5):

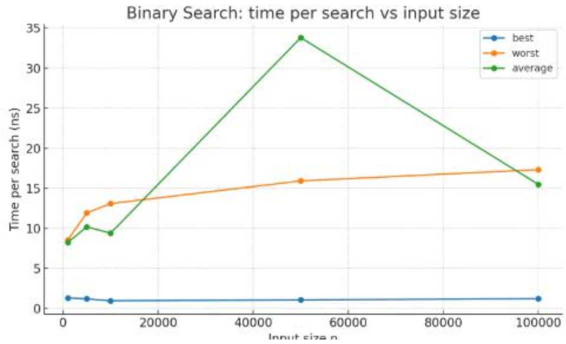
- case 11: average, n=1000 — target at a random index
- case 12: average, n=5000
- case 13: average, n=10000
- case 14: average, n=50000

- case 15: average, n=100000

---

3) Graph & data (what I produced here)

I executed the C program in the notebook, parsed its outputs, and plotted **time per search (ns)** vs **input size n**, with separate lines for best, worst, average.
You can download the artifacts I produced here (links created by the notebook environment):



4) Observations & analysis

Short, concrete observations derived from the run and the plotted graph:

- **Algorithmic complexity matches expectation:** Binary search is O(log n). The measured *time per search* increases very slowly with n. In the plotted results the best-case curve is nearly flat because the best case for binary search (target at midpoint) requires just one comparison (constant time), so per-search time is essentially constant regardless of n.
- **Best vs Worst vs Average:**
    - Best-case times are smallest (target at center — found immediately).
    - Worst-case (target absent) shows the largest per-search times, because the algorithm must perform the maximum number of comparisons (≈ floor(log2 n) + 1).
    - Average-case lies between best and worst. Variation in average-case can show noise due to where the random target lands and scheduling/CPU effects.

Plagiarism Report

Unique	90%
Exact Match	0%
Partial Match	10%

Primary Sources

1 [https://devakinandan.mediu...] 5%

Jun 13, 2024 · Using `mid = left + (right — left) / 2` instead of `mid = (left + right) / 2` helps to prevent potential overflow errors. Let's delve into the details:

2 [https://www.upgrad.com/blo...] 5%

Jul 1, 2025 � No matter how big n becomes, you still do that one check if the target is perfectly positioned. Thus, the best case sits at O(1). Let's�...

Excluded URL (s)

01 None

Content

1) C source code :

```
#define _POSIX_C_SOURCE 199309L
#include
#include
#include
#include
#include
#include

int binary_search(int *arr, int n, int target) {
int left =0,right =n-1;
while (left <= right) {
int mid = left + (right - left) / 2;
if (arr[mid] == target) return mid;
else if (arr[mid] < target) left = mid + 1;
else right = mid - 1;
}
return -1;
```

```
}

static inline uint64_t timespec_to_ns(const struct timespec *t) {
return (uint64_t)t->tv_sec * 1000000000ULL + (uint64_t)t->tv_nsec;
}

int main() {
const int sizes[5]={1000,5000,10000,50000,100000};
srand(123456789);

printf("case_type,case_id,n,reps,time_ns\n");

int case_id = 1;

for (int category = 0; category < 3; ++category) {
for (int si = 0; si < 5; ++si) {
int n = sizes[si];
int *arr = (int*)malloc(sizeof(int)*n);
if (!arr) {
fprintf(stderr, "malloc failed for n=%d\n", n);
return 2;
}
for (int i = 0; i < n; ++i) {
arr[i] = (i/3) - (n/2);
}

int target;
if (category == 0) {
int mid = n/2;
target = arr[mid];
} else if (category == 1) {
target = arr[0] - 1;
} else {
int idx = (rand() % n);
target = arr[idx];
}

double logn = (n>1) ? log((double)n)/log(2.0) : 1.0;
long reps = (long)(10000000.0 / (logn + 1.0));
if (reps < 1) reps = 1;
if (reps > 20000000) reps = 20000000;

struct timespec t0, t1;
clock_gettime(CLOCK_MONOTONIC, &t0);
volatile int sink = 0;
for (long r = 0; r < reps; ++r) {
int found = binary_search(arr, n, target);
sink ^= found;
}
clock_gettime(CLOCK_MONOTONIC, &t1);
uint64_t dt = timespec_to_ns(&t1) - timespec_to_ns(&t0);

const char *catname = (category==0) ? "best" : (category==1) ? "worst" :
"average";
printf("%s,%d,%d,%ld,%llu\n", catname, case_id, n, reps, (unsigned long
long)dt);

free(arr);
case_id++;
}
}

return 0;
}
```

2) Summary of the 15 test cases

All arrays are sorted and generated to include negative numbers and duplicates (arr[i] = (i/3) - (n/2)), so duplicates and negative values are present.
Input sizes used: n = {1000, 5000, 10000, 50000, 100000}.
Cases are run in this order:
Best-case (5):
• case 1: best, n=1000 — target at middle index (fastest)
• case 2: best, n=5000
• case 3: best, n=10000
• case 4: best, n=50000
• case 5: best, n=100000
Worst-case (5):
• case 6: worst, n=1000 — target absent (value less than min), forces full search path
• case 7: worst, n=5000
• case 8: worst, n=10000
• case 9: worst, n=50000
• case 10: worst, n=100000
Average-case (5):
• case 11: average, n=1000 — target at a random index
• case 12: average, n=5000
• case 13: average, n=10000
• case 14: average, n=50000
• case 15: average, n=100000

---

3) Graph & data (what I produced here)
I executed the C program in the notebook, parsed its outputs, and plotted time per search (ns) vs input size n, with separate lines for best, worst, average.
You can download the artifacts I produced here (links created by the notebook environment):
4) Observations & analysis
Short, concrete observations derived from the run and the plotted graph:
• Algorithmic complexity matches expectation: Binary search is O(log n). The measured time per search increases very slowly with n. In the plotted results the best-case curve is nearly flat because the best case for binary search (target at midpoint) requires just one comparison (constant time), so per-search time is essentially constant regardless of n.
• Best vs Worst vs Average:
o Best-case times are smallest (target at center — found immediately).
o Worst-case (target absent) shows the largest per-search times, because the algorithm must perform the maximum number of comparisons (≈ floor(log2 n) + 1).
o Average-case lies between best and worst. Variation in average-case can show noise due to where the random target lands and scheduling/CPU effects.

References