Arrays and performance in C

Algorithms and data structures ID1021

Johan Montelius

Spring 2026

Introduction

In this assignment you should explore the efficiency of different operations over an array of elements. We take for granted that you have been working with arrays before so this should not be a problem, focus will be on performance measurement, presenting numbers and realizing that there is a fundamental difference between what is called $O(1)$, $O(n)$ and $O(n^2)$ (more on this later).

You should benchmark three different operations and determine how the execution time differs with the size of the array. The three operations are:

- Random access : reading or writing a value at a random location in an array.
- Search : searching through an array looking for an item.
- Duplicates : finding all duplicates in an array.

The implementation of these operations are quite straight forward, this is not problem. The problem is to do the benchmark, present the results and describe the general behavior for these operations.

Random access

When trying to measure how long time an operation takes, one is limited by the resolution of the clock. If we have a clock that measures microseconds then obviously it will be very hard to measure an operation that only takes a few nanoseconds.

the clock

In order to measure the performance we need a method to measure time. You first task is to figure out the accuracy of this clock. Run the following code and present your conclusion in the report.

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>

long nano_seconds(struct timespec *t_start, struct timespec *t_stop) {
    return (t_stop->tv_nsec - t_start->tv_nsec) +
           (t_stop->tv_sec - t_start->tv_sec)*1000000000;
}

int main() {
  struct timespec t_start, t_stop;
  for(int i = 0; i < 10; i++) {
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    long wall = nano_seconds(&t_start, &t_stop);
    printf("%ld ns\n", wall);
  }
}
```

The above code is for a POSIX system i.e. MacOS, Linux etc. If you're using Windows the code will not compile. There are multiple ways how to solve but why not use the Windows equivalent.

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

long long nano_seconds(LARGE_INTEGER *start, LARGE_INTEGER *stop) {
  LARGE_INTEGER frequency;
  QueryPerformanceFrequency(&frequency);
  return (stop->QuadPart - start->QuadPart) * 1000000000 / frequency.QuadPart;
}

int main() {
  LARGE_INTEGER t_start;
  LARGE_INTEGER t_stop;
  for (int i = 0; i < 10; i++) {
    QueryPerformanceCounter(&t_start);
```

```
    QueryPerformanceCounter(&t_stop);
    long long wall = nano_seconds(&t_start, &t_stop);
    printf("%lld ns\n", wall);
  }
}
```

In the following examples the POSIX version will be used so you need to adapt it to the Windows equivalent.

Will the clock be accurate enough to measure how long time it takes to perform a single array access? Let's try the following:

```
int main() {
  struct timespec t_start, t_stop;

  int given[] = {1,2,3,4,5,6,7,8,9,0};
  int sum = 0;

  for(int i = 0; i < 10; i++) {
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    sum += given[i];
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    long wall = nano_seconds(&t_start, &t_stop);
    printf("one operation in %ld ns\n", wall);
  }
}
```

As you probably notice the time to access a value in an array is very small in comparison to the resolution of the clock. In order to get a better understanding of the time it takes to perform an access operation we would have to do a couple of hundred and measure the time it takes to do them all.

We preferably also want to make the access random (at random locations in the array) to prevent caching from playing a role (if we read from the same location or consecutive locations then the value will definitely be in the cache and give us a false impression). To do random read operations we could use a library procedure but we need to understand what we measure.

```
int main() {

  struct timespec t_start, t_stop;

  int array[] = {0,1,2,3,4,5,6,7,8,9};
  int sum = 0;
```

```
  clock_gettime(CLOCK_MONOTONIC, &t_start);
  for (int i = 0; i < 1000; i++) {
    sum += array[rand()%10];
  }
  clock_gettime(CLOCK_MONOTONIC, &t_stop);

  long wall = nano_seconds(&t_start, &t_stop);
  printf("%ld ns\n", wall/1000);
}
```

What is it that we now are measuring? The time it takes to create a random number is much longer than accessing an element in an array so we are measuring the wrong thing. In order to measure the time it takes to do a number of random access operation in an array of size n we could do some thing like this:

```
  long bench(int n, int loop) {

      int *array = (int*)malloc(n*sizeof(int));
      for (int i = 0; i < n; i++) array[i] = i;

      int *indx = (int*)malloc(loop*sizeof(int));
      for (int i = 0; i < loop; i++) indx[i] = rand()%n;

      int sum = 0;
      clock_gettime(CLOCK_MONOTONIC, &t_start);
      for (int i = 0; i < loop; i++) sum += array[indx[i]];
      clock_gettime(CLOCK_MONOTONIC, &t_stop);
      long wall = nano_seconds(&t_start, &t_stop);
      return wall;
    }
```

In this program we first create the `array`, of size n, that we are to access but then also create an array `indx`, of size *loop*, of random indices that we are going to use to access the array. The actual loop now first selects the random index from the `indx` array and then use the index to access the target `array`.

If we are now picky we are measuring more than one array read operation. In each loop we: compare i to *loop*, access the array *indx*, access the array *array*, increment *sum* and increment i. You can remove the two read operations and simply increment the sum by one to see how much time the loop overhead is. As you will probably realize we're measuring the loop more than the read operations.

We ignore this for now since our goal is to determine how the access time differs when we change the size of the array. In the future we will

measure the time to perform more complex operations that will take very much longer time so the time to do the loop will be negligible.

Try the following benchmark a couple of times.

```
int main() {
  for (int i = 0; i < 10; i++) {
    long wall = bench(n, loop);
    printf("time : %ld ns\n", wall);
  }
}
```

As you see, the measurements we get vary and the question is how we should report this. You could of course report the average or median of a hundred measurements . . . or you could report the minimum value. Let's try this to see what we are talking about:

Include the file `<limits.h>` to get access to the pre-defined values such as `LONG_MAX`.

```
int main(int argc, char *argv[]) {

  int n = 1000;
  int loop = 1000;
  int k = 10;

  long min = LONG_MAX;
  long max = 0;
  long total = 0;

  for (int i = 0; i < k; i++) {
    long wall = bench(n, loop);
    if (wall < min) min = wall;
    if (wall > max) max = wall;
    total += wall;
  }
  printf("maximum time: %0.2f ns/operation \n", (double)max/loop);
  printf("minimum time: %0.2f ns/operation \n", (double)min/loop);
  printf("average time: %0.2f ns/operation \n", (((double)total)/loop)/k);
}
```

If you run this a couple of times you will probably see that the maximum time varies quite a lot. This is understandable since you're running on top of an operating system that has a lot of things to do and once in a while it decides to pause our program. The minimum time is, in comparison, almost carved in stone; there is an exact minimum value and the more times we try you will see that we more often hit this value.

The average value might be of interest but since it is very influenced by the worst case number it only adds noise in what we want to show. The median is better for describing the behavior but why not go for the most predictable one - the minimum value. If the minimum value is all that we are after we can of course simplify the program.

Now we only add some code where we get the minimum time of the different sizes of an array.

```
int main(int argc, char *argv[]) {

  int sizes[] = {1000,2000,4000,8000,16000,32000};

  int k = 10;
  int loop = 1000;

  for (int i = 0; i < 6; i++) {
    int n = sizes[i];
    long min = LONG_MAX;
    for (int i = 0; i < k; i++) {
      long wall = bench(n, loop);
      if (wall < min)
        min = wall;
    }
    printf("%d  %0.2f ns\n", n, (double)min/loop);
  }
```

One thing that you should try is to compile the code with optimizations. Depending on which C compiler that you use the options are a bit different but typically accept options like `-O`, `-O2` or even `-O4`.

When we measure performance we of course want to do our best so why not compile using the best optimizations. Compile the program you now have with the best optimization and you will see that it makes a huge difference... or, are the new numbers too good to be true?

If you look at your implementation of `bench()` you will see that we are summing all the values in the variable `sum`. This value is then never used and the compiler will detect this; why calculate a value that is not used, let's remove it. With the best optimizations turned on it is likely that you're measuring nothing. In order to fix this we must trick the compiler into thinking that the value `sum` is actually needed. Try the following change:

```
    int sum = 0;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < loop; i++) sum += array[indx[i]];
```

```
clock_gettime(CLOCK_MONOTONIC, &t_stop);

if (sum == 0)
  return 0;
:
```

The reason why you should go through these steps is not because I want you to memorize the one and only way of measuring performance. The reason is that I want you to realize that measuring something so trivial as the time it takes to access an element is quite difficult.

In the future we will measure things that are a bit easier to measure, mostly because they take longer time, but you should always question your method and make user that you measure what you think you're measuring.

presenting execution time

When you report execution times it is quite ridiculous to report something like 123456789 ns - even if this is actually the result that you got. The reason that it is ridiculous is that the execution time will most likely change very much if you run the benchmark again. If the next run results in 124356789 ns then it is rather pointless to present the runtime using nine significant figures.

Your choice of the number of significant figures should reflect how stable your measurements are. If you run your benchmarks on a regular laptop there is a lot of things going on in the background that will effect the execution time. My guess is that you will not be able measure anything with more than three figures of accuracy and in most cases two will do fine.

Even if you can make a stable measurement with three significant figures it might not be how you should report it. If you want to show that a sequence of measurements increase with a factor 2 for each measurement, large numbers will only be in the way.

Take a look at the table 1. The conclusion might be correct but it is not "clearly seen" at all.

n	100	200
prgm1	12345678 μs	22345678 μs
prgm2	14325678 μs	56213478 μs

Table 1: As clearly seen - prgm1 double the execution time whereas prgm2 almost increase with a factor four.

Always think about the message that you want to deliver and then use as few numbers as possible to convey that message. If the doubling of execution time was the message, then Table 2 might be a better idea.

n	100	200
prgm1	12 ms	22 ms
prgm2	14 ms	56 ms

Table 2: Much better

When you present numbers also think about readability. To say 23000000 μs is of course only using two significant figures but is very hard to compare this measurement with 1200000 μs . . . is that half the execution time or . . . ? Also refrain from using mathematical notations such as 1.23×10^7 ns, especially when you want someone to quickly see the relationship with 2.4×10^6 ns, the information is of course all there but you don't make the connection without thinking - how about 12 ms compared to 2.4 ms.

Search for an item

Once we know how to set up a nice benchmark we can explore a simple search algorithm and see how the execution time varies with the size of the array.

When we setup the benchmark we want to capture the estimated time it would take to search for a random key in an array of unsorted keys. We assume that the number of keys is larger the size of the array - if we only had the keys 1, 2 and 3 we would probably find one very quickly even if the array was very large. So when we search for a given key it mighty be that we search through the whole array without finding it.

If we follow the pattern in the previous section we could try the following:

```
long search(int n, int loop) {

    int *array = (int*)malloc(n*sizeof(int));
    for (int i = 0; i < n; i++) array[i] = rand()%(n*2);

    int *keys = (int*)malloc(loop*sizeof(int));
    for (int i = 0; i < loop; i++) keys[i] = rand()%(n*2);

    int sum = 0;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < loop; i++) {
      int key = keys[i];
      for (int j = 0; j < n; j++) {
        if (key == array[j]) {
          sum++;
          break;
```

```
        }
      }
    }
    clock_gettime(CLOCK_MONOTONIC, &t_stop);
    long wall = nano_seconds(&t_start, &t_stop);
    return wall;
  }
```

How would you present the figures you get and how would you explain the result? Can you find a simple polynomial that roughly describes the execution time?

Search for duplicates

So now for the final task, finding the number of duplicated numbers in an arrays of size n. If a number occurs three times it is counted as two doubles, four times as three etc. For example, in the array:

```
{4, 2, 5, 2, 1, 3, 2, 1}
```

there are three doubles: the number 1 and the number 2 counted as two doubles.

This task is very similar to the search exercise and we will use the same strategy but now using the array it self to know what we are looking for. For each key in the array try to find it in the rest of the array, stop as soon as you have found it.

The only difference is that now as the size of the array grows we need to do more search operations. A small change it might seam but it makes a huge difference. If the size of the array is large enough (> 100 elements) we could skip the loop parameter; the time to find the duplicates will be long enough to hide the uncertainty of the clock.

```
long duplicates(int n) {

    int *array = (int*)malloc(n*sizeof(int));
    for (int i = 0; i < n; i++) array[i] = rand()%(n*2);

    int sum = 0;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
    for (int i = 0; i < n; i++) {
      int key = array[i];
      for (int j = i+1; j < n; j++) {
        if (key == array[j]) {
          sum++;
```

```
            break;
          }
        }
      }
      clock_gettime(CLOCK_MONOTONIC, &t_stop);
      long wall = nano_seconds(&t_start, &t_stop);
      return wall;
    }
```

How would you present the figures you get and how would you explain the result? Can you find a simple polynomial that roughly describes the execution time for finding duplicates in two arrays of size n?