CS-118-02 Week 11

Varun Narravula

2022-04-11

***Note**: There are some standalone C files in the examples folder for each snippet in this document that you can compile, test, and benchmark to see how slow and fast each algorithm is. Try it out for yourself.*

Sorting

Note: assume all references to the `swap` function in code examples in this section have this implementation:

```c
void swap(int *x, int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}
```

There are many different ways to sort ordered collections, and they are quite a popular concept in computer science. Quite a lot of computer science classes touch on them, and here you'll be looking at three specific algorithms for sorting: bubble sort, selection sort, and merge sort.

Bubble Sort

What is a bubble sort? This is a sorting algorithm that is the easiest to understand, but is also the least efficient for larger workloads. It repeatedly iterates over the collection that needs to be sorted, and compares two elements with each other. If the second element is less than the first element, then they are positionally swapped, and the algorithm continues. When it finishes iterating over the collection, then it checks if anything was swapped. If any elements were swapped, then it repeats the process again. If no elements were swapped, that means the collection is now sorted.

Let's see an example of this in action visually. Take a list as such:

```c
int collection[] = {5, 1, 4, 2, 8};
```

A bubble sort of this list would follow this process:

```
First iteration:
{ 5, 1, 4, 2, 8 } => { 1, 5, 4, 2, 8 } // 1 < 5 -> swapped
{ 1, 5, 4, 2, 8 } => { 1, 4, 5, 2, 8 } // 4 < 5 -> swapped
{ 1, 4, 5, 2, 8 } => { 1, 4, 2, 5, 8 } // 2 < 4 -> swapped
{ 1, 4, 2, 5, 8 } => { 1, 4, 2, 5, 8 } // 8 > 5 -> not swapped
Elements were swapped -> another iteration
Second iteration:
{ 1, 4, 2, 5, 8 } => {1, 4, 2, 5, 8} // 4 > 1 -> not swapped
{ 1, 4, 2, 5, 8 } => {1, 2, 4, 5, 8} // 2 < 4 -> swapped
{ 1, 2, 4, 5, 8 } => {1, 2, 4, 5, 8} // 5 > 4 -> not swapped
{ 1, 2, 4, 5, 8 } => {1, 2, 4, 5, 8} // 8 > 5 -> not swapped
Elements were swapped -> another iteration
Third iteration:
{ 1, 2, 4, 5, 8 } => { 1, 2, 4, 5, 8 } // 2 > 1 -> not swapped
{ 1, 2, 4, 5, 8 } => { 1, 2, 4, 5, 8 } // 4 > 2 -> not swapped
{ 1, 2, 4, 5, 8 } => { 1, 2, 4, 5, 8 } // 5 > 4 -> not swapped
{ 1, 2, 4, 5, 8 } => { 1, 2, 4, 5, 8 } // 8 > 5 -> not swapped
Elements were not swapped -> sorted; end
```

Here's an implementation of a bubble sort in C:

```c
void bubble_sort(int collection[], size_t size) {
  size_t i, j;
  for (i = 0; i < size - 1; i++) {
    bool swapped = false;
    for (j = 0; j < size - i - 1; j++) {
      if (collection[j] > collection[j+1]) {
        swap(&collection[j], &collection[j+1]);
        swapped = true;
      }
    }

    if (swapped == false) {
      break;
    }
  }
}
```

Do you see anything wrong with this? It does work. However, it's extremely slow. Like extremely slow. It has a time complexity of $O(n^2)$ so it gets slow fast. Imagine trying to sort a collection of a million elements with this algorithm. If the list were scrambled, you have a high likelihood of iterating over the same sequence more than 1 million times; that's horribly inefficient. Even though bubble sort is straightforward in terms of code, it does not work out for large collections.

Selection Sort

Now, what is a selection sort? This is a sorting algorithm that still relies on swapping, but also has the idea of a sorted part and an unsorted part within the same collection. What does this mean? The way selection sort works is that it finds the smallest element in the unsorted part, and then moves it to the sorted part. This explanation sounds confusing, but it is much clearer with the visual process. Let's take a look; say you have the following collection:

```
int collection[] = {64, 25, 12, 22, 11};
```

A selection sort of this collection would go like this (the `::` denotes the boundary between the sorted :: unsorted parts of this collection):

```
First iteration (sorted part is empty):
{ :: 64, 25, 12, 22, 11 } => { 11 :: 25, 12, 22, 64 } // 64 <-> 11
Second iteration:
{ 11 :: 25, 12, 22, 64 } => { 11, 12 :: 25, 22, 64 } // 25 <-> 12
Third iteration:
{ 11, 12, :: 25, 22, 64 } => { 11, 12, 22 :: 25, 64 } // 25 <-> 22
Fourth iteration:
{ 11, 12, 22 :: 25, 64 } => { 11, 12, 22, 25 :: 64 } // 25 <-> 25 (swap with itself)
```

When each minimum element of the unsorted part was found, it was swapped with the first element of the unsorted part, and the boundary of the sorted and unsorted parts was advanced by one.

Here's an implementation of a selection sort in C:

```
void selection_sort(int collection[], size_t size) {
  size_t i, j;

  for (i = 0; i < size - 1; i++) {
    size_t minimum_index = i;

    for (j = i + 1; j < size; j++) {
      if (collection[j] < collection[minimum_index]) {
        minimum_index = j;
      }
    }

    swap(&collection[minimum_index], &collection[i]);
  }
}
```

This also has its own drawbacks, but it does have some benefits over a bubble sort. For one, it also has $O(n^2)$ complexity in the worst scenarios, the same as bubble sort. However, it is great for decreasing the amount of stress that it puts on memory and is less write-intensive than a bubble sort because it never makes more than n swaps. Versus bubble sort, selection sort is much better. However, it does fall victim to the fact that it is unstable (does not preserve the order of equal elements with differing keys) and can still use the same time complexity as a bubble sort. Other sorts, as you will see in the next subsection, do not fall victim to this.

Merge Sort

Merge sort is perhaps the trickiest of all of these sorting algorithms. Instead of iterating over a collection repeatedly like bubble sorts and selection sorts do, it recursively splits a collection into two halves and sorts them using the same method; it then merges these results together into a sorted list. A diagram is pretty confusing to write in Markdown, and frankly I'm lazy, so I'll link to a video that shows this process quite well.

Raw link: [https://www.youtube.com/watch?v=JSceec-wEyw]

Let's say you have the following list:

Here's an implementation of merge sort in C:

```c
void merge(int collection[], int left_index, int middle_index, int r) {
  int i, j, k;
  int left_size = middle_index - left_index + 1;
  int right_size = r - middle_index;

  int left[left_size], right[right_size];

  for (i = 0; i < left_size; i++) {
    left[i] = collection[left_index + i];
  }

  for (j = 0; j < right_size; j++) {
    right[j] = collection[middle_index + 1 + j];
  }

  i = 0;
  j = 0;
  k = left_index;
  while (i < left_size && j < right_size) {
    if (left[i] <= right[j]) {
      collection[k] = left[i];
      i++;
    }
    else {
      collection[k] = right[j];
      j++;
    }
    k++;
  }

  while (i < left_size) {
    collection[k] = left[i];
    i++;
    k++;
  }

  while (j < right_size) {
    collection[k] = right[j];
    j++;
    k++;
  }
}

void merge_sort(int arr[], int left, int right) {
  if (left < right) {
    int middle = left + (right - left) / 2;

    merge_sort(arr, left, middle);
    merge_sort(arr, middle + 1, right);

    merge(arr, left, middle, right);
  }
}
```

While this involves much more code, it is more efficient at sorting than the other two algorithms in worst-case scenarios. However, this suffers from the fact that it is recursive and it can quickly overwhelm the stack. There is an iterative way to implement merge sort, but it takes much more code and is less understandable.

All three sorting algorithms have their strengths and weaknesses, and can be used in different situations. The most important thing is to evaluate your use cases for each sort and figure out what the best sorting algorithm for your situation is. More often than not, it will be a builtin function, and you will likely not have to implement it yourself.

Time Complexity

What is this "time complexity" I've been going on about? Time complexity, in the simplest way put, is the number of times a statement is performed; this is a metric of how efficient a statement or algorithm can be. It is represented using something called *Big O notation*, which is essentially a way of representing where a mathematical function tends to (which could be a finite value, or even infinity). Examples of common time complexities for algorithms include:

- $O(1)$ :: constant time (i.e. indexing an array, printing to console)
- $O(n)$ :: linear time (i.e. linear search, naive summation of array)
- $O(\log n)$ :: logarithmic time (i.e. binary search)
- $O(n \log n)$ :: linearithmic time (i.e. merge sort, Fourier transforms)
- $O(n^2)$ :: quadratic time (i.e. bubble sort, insertion sort)
- $O(n!)$ :: factorial time (i.e. naive traveling-salesperson problem solution)

The way you can tell if an algorithm is more efficient than another is to compare their time complexities in the worst scenarios; if a time complexity grows faster on a graph (compare their rates of growth or derivatives), then it is less efficient. Think about attempting to solve the traveling-salesperson problem by brute-forcing every possible solution; such an approach has a factorial time complexity. A factorial grows extremely fast; if you have 20 cities for the salesperson to travel to, then you have to repeat your loop 20! times, which is an incredibly large number; 2432902008176640000, to be exact. This is completely impractical with the computing resources you have now.

Algorithms that have smaller time complexities are almost always preferred over ones with slower time complexities. However, there are certain exceptions. Hashing passwords in a database is one such application. Hashing algorithms like `bcrypt` and `argon2` often have parameters for how many times you want to run the hashing algorithm on an input, and are quite slow themselves in order to deter any hacking attempts by rendering them impractical. Evaluating your use cases is important here, but more often than not, you will want to prefer time complexities that have slower growth rates than others.

Binary Sarch

Well, there's a lot of sorting algorithms, but what about searching ones? The simplest, and also the most naive way of searching would be to iterate through the collection and see if it matches the search term. However, this has a time complexity of $O(n)$ time in the worst case scenario (when the element searched for is located at the end of the collection). Is there a better way? There is! Binary search has a time complexity of $O(\log n)$, which is much more efficient on larger lists. However, it comes with a caveat: it can only work on sorted collections; otherwise, your result will be undefined.

How does binary search work? It divides the collection searched in half, and checks to see what the element in question is in relation to the middle. If it is greater, then do the same to only the upper half, and so on, and so on. If it is less, then do the same to only the lower half. If somehow it is the same, then you've already found it; this last scenario is unlikely, though.

Let's see how this works visually; say you have the following collection:

```
int collection[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
```

A binary search of this collection for the element 23 would be as follows:

```
First iteration:
Middle: 16
23 > 16 -> upper half {23, 38, 56, 72, 91}
Second iteration:
Middle: 56
23 < 16 -> lower half {23, 38}
Third iteration:
38 found! Return index 5
```

Here's an implementation of a recursive binary search in C:

```c
int binary_search(int collection[], int left, int right, int element) {
  if (right >= left) {
    int middle = left + (right - left) / 2;

    if (collection[middle] == element) {
      return middle;
    }

    if (collection[middle] > element) {
      return binary_search(collection, left, middle - 1, element);
    }

    return binary_search(collection, middle + 1, right, element);
  }

  return -1;
}
```

While binary searching can be implemented in an iterative manner, it's more difficult to understand. This splits the collection and performs a binary search on each subcollection. When a base case is reached (aka the element searched for), then it will wind down and return the value. If a case is not reached (aka the element is not found), then it will wind down after returning -1.

While a binary search is much faster than a linear search, since it only works on sorted lists and gives undefined results otherwise, attempting to sort and then perform a binary search for performance gains instead of using a linear search is counterintuitive and is actually an antipattern. Since a naive linear search will always have $O(n)$ time complexity in its worst-case scenario, and a sort will almost always be much slower than this (the fastest merge sort still has a complexity of $O(n \log n)$), merge sorting and binary searching will have a time complexity of $O(n \log n + \log n)$, which is much slower. If you know your list is sorted beforehand, though, then it's'always better to use a binary search.

Multithreading Introduction

Now for real, what is a thread? Before that, remember what a process is? It's a stack, heap, and your instructions. When a process is started, it has a few things given to it by the kernel, including a set of file handles it is using, kernel resources it is using, among other things. No process can overtake another process's resources without something going haywire. All of this information is called a "task struct" in Linux kernel terms, and this is all the data for a process.

Based on this information about processes, what is a thread? It's something simpler; it is the path of execution for your code. When a process is loaded, it starts executing code from the entrypoint (i.e. `main` in C/C++, `_start` in Linux x86 assembly). But what is starting at that location? The starting path of execution itself, or the beginning of the instructions in a sequence, is where the kernel will start this thread from. This starting path is a thread called the *primary thread*; other paths of execution spawned from this primary thread are called *worker threads*. It usually continues on this single thread, without using other threads. But how would you get multiple functions to execute at the same time? And how would they communicate with each other if they were on different threads? These are all challenges of multithreading. You can't directly decide what threads you run on; the kernel does that for you in a process called "SMP", or **S**ymmetric **M**ulti-**P**rocessing. In the kernel, there are task queues that contain tasks that are put to sleep, and only woken up when they are needed through a process called *interrupts* or other means. These are all components of threading abstracted away from you. There are different types of queues, such as run queues, but you don't need to know about them for now at least.

Think about a web server. Web servers serve content, but only when they are needed. When someone makes an HTTP request to a server, it wakes up, serves the content requested, and then the server goes back to sleep. It does this millions and millions of times; it's not constantly awake. Task queues work in a similar way; each task is only woken up when it needs something. A mouse's tasks are woken up when you move it, for example.

Sharing Information Between Threads

Each thread has their own stack, so it's not possible to read another thread's stack. How would you share information between threads then? The one common resource that all of these threads share is the process's heap space, so you would need to allocate a shared resource on the heap.

To take advantage of multithreading, you need to split up work between threads using shared resources on the heap, and then recombine these results. This is the primary principle behind it all; split your workloads, run them in parallel, and recombine the results. And bam; there you have it. Imagine you need to process a super large image from NASA of a far-away galaxy; you split up the workload 16 different ways, and when you are done, it will have taken 1/16th of the time. However, not all problems can benefit from this; event loops, for example, are not great at multithreading. Again, evaluate your use cases. Not everything needs or can benefit from multithreading and the extra cognitive overhead that comes with implementing it.

How do you pass information when processes are not done, though? You can make threads sleep while waiting for information required to operate. How do you make a thread sleep, though? You might have used this function before in C: it's called `sleep` in the `unistd.h` header. The common misconception behind this function is that it pauses execution of the entire program; in reality, it only ceases execution of the current thread for a certain number of seconds. The entire OS is a great example of this concept of sleeping while waiting for resources. The kernel itself is running constantly, but it needs to give priority to other programs when it is not invoked by itself.

In practice, you will not usually wait for a certain amount of time in order to let resources be created, changed, or destroyed before the thread can access it; instead, you would use something called locking.

Since memory is shared, you have to make sure that other threads are not reading your information while it's being written; this is locking. If threads could read information before it is finished, then it's essentially the same as reading garbage data from a register that has not been zeroed out. This actually has a name: such a situation is called a *race condition*. Race conditions are horrible; this is when two or more threads attempt to read/write to the same resource at the same time; each thread essentially races against each other to get to the same result, and there is no telling what will happen; even something like a print statement can influence which thread accesses a resource first, because this print statement is synchronous and will block the thread. In race conditions, timing is everything; in fact, using `sleep` to wait for a call to finish is an example of something that can be called a race condition; a thread would be racing against the clock for access to a resource quite literally.

Important terminology: There are two types of bounds for workloads: CPU-bound (waiting for CPU computation cycles), which would be things like games and mathematically intensive workouts, and IO-bound (waiting for external information, like web servers and networking) workloads. Both types of workloads can benefit from multithreading depending on the situation.

Introductory Multithreading Example

Let's see an example of a program utilizing threads in C using the `pthread` library that comes with Linux:

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

```
struct random {
  int x;
  int y;
};

void * func(void *arg) {
  struct random *thing = (struct random *)arg;

  printf("x: %d; y: %d\n", thing->x, thing->y);
  sleep(1);

  return NULL;
}

int main() {
  struct random a = {1, 2};
  struct random b = {3, 4};

  pthread_t threads[2] = {0};

  int err = pthread_create(&threads[0], NULL, &func, &a);
  if (err) {
    printf("pthread_create failed\n");
    return err;
  }

  err = pthread_create(&threads[1], NULL, &func, &b);
  if (err) {
    printf("pthread_create failed\n");
    return err;
  }

  for (size_t i = 0; i < 2; i++) {
    err = pthread_join(threads[i], NULL);
    if (err) {
      printf("pthread_join failed\n");
      return err;
    }
  }

  return 0;
}
```

This is a particularly useless program, but it shows two key functions that are used in Linux to create and manage threads, `pthread_create` and `pthread_join`. What do they do? `pthread_create` is pretty self-explanatory: it creates a thread. It requires a few arguments: the first one is a pointer to where to store the thread ID of type `pthread_t` (you will need this to communicate with the thread, similar to how you need a PID to interact with a process). The second one is passed `NULL` (defined as a constant for 0 on most platforms), but it can also contain a struct of type `pthread_attr_t` that specifies different attributes that the thread will have. The third looks weird; this is a reference to a function, or a function pointer. Since a thread is a path of execution, `pthread_create` needs to know where to start the thread from; it will in fact start it from the beginning of the function referred to by the function pointer, which in this case is `func`. This function needs to have a return type of `void *` and should take a single void pointer as an argument (this can point to anything). The last argument is what is passed to the function described by the function pointer. Since you can only pass in one argument, it's common practice to pass multiple arguments as a single struct.

This function that is called on those threads is simple: it casts from the void pointer type to the pointer type that it needs (in this case, to the `random` type), prints its values, and sleeps for 1 second. It then returns `NULL` (or 0). This function is called on two separate threads with two separate variaibles, `a` and `b`; the thread IDs are stored in a array of type `pthread_t[]` to refer to later.

How does the threads then finish executing? This is what `pthread_join` does. Since all threads need to finish in a program before it can exit successfully, `pthread_join` blocks the thread it is called from and waits for the thread corresponding to the thread ID that is passed to it to end, and cleans up any resources that were used in the process of using the thread. There is another way to stop a thread, which is a function called `pthread_kill`, but you almost *never* want to kill a thread using `pthread_kill`; this leaves resources undestructed or unfreed and essentially results in memory leaks but worse.

Interestingly enough, you do not necessarily need to join all threads together for the program to end successfully. If the pthread_join section is commented out, then this particular example program will end fine, because the kernel will forcibly deallocate the threads' resources when they are finished, and not perform any real cleanup routines. This is similar to how memory that is not freed at the end of a program is technically freed; however, this is bad practice, and is essentially the same behavior as executing `pthread_kill` on a thread before `main` ends. Threads that are not joined and then become inaccessible in such a way (their thread IDs are lost after getting allocated but are never joined) are called zombie threads, and these should be avoided like the plague (no pun intended).

Another situation where threads are not necessarily joined are when they are created and then detached explicitly using `pthread_detach`. If you were to do such a thing, then the kernel should be able to clean it up gracefully, b ut only use it when you do not care about the path of execution that happens in the background. In this class, you will likely only use `pthread_join` to syncrhonously end threads.