More Sorting

2024 Winter APS 105: Computer Fundamentals
Jon Eyolfson

Lecture 33
1.0.0

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License

We'll Touch on One More Sorting Algorithm

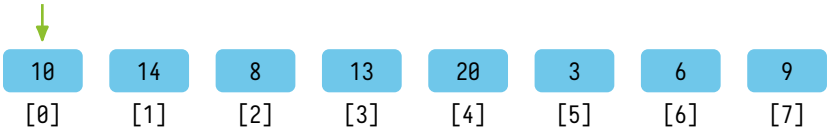
The sorting algorithms in this lecture are NOT testable

Quicksort is a $\mathcal{O}(n \log n)$ sorting algorithm

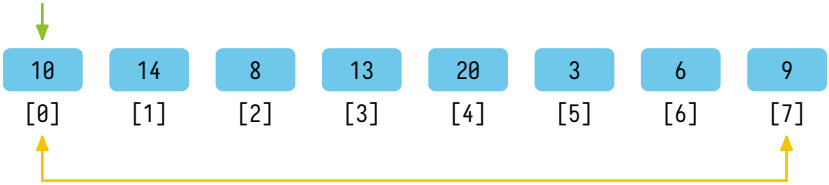
We select an element as a "pivot", and place elements less than the pivot to the left, and greater than to the pivot the right (called partitioning)

After this, we recursively sort both sides

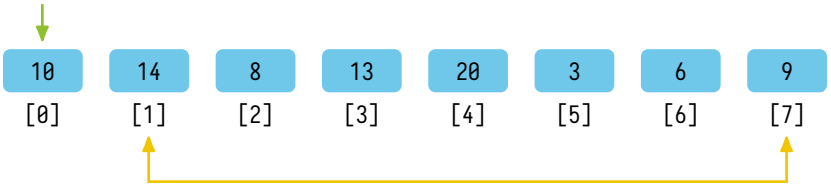
Partitioning an Array Using the Last Element (9) as a Pivot



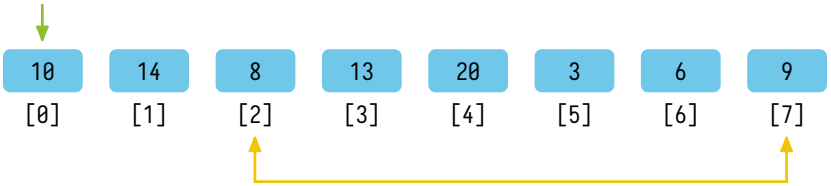
Partitioning an Array Using the Last Element (9) as a Pivot



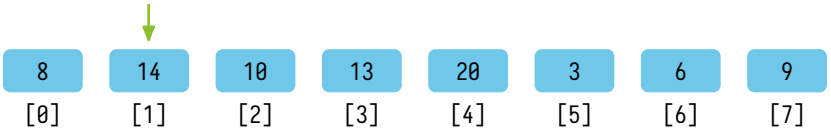
Partitioning an Array Using the Last Element (9) as a Pivot



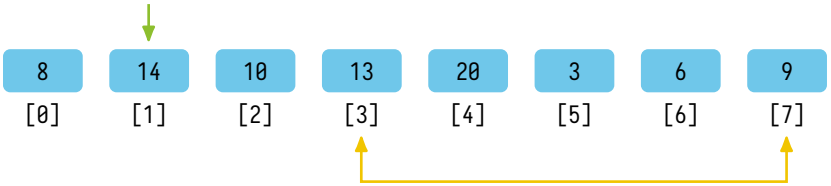
Partitioning an Array Using the Last Element (9) as a Pivot



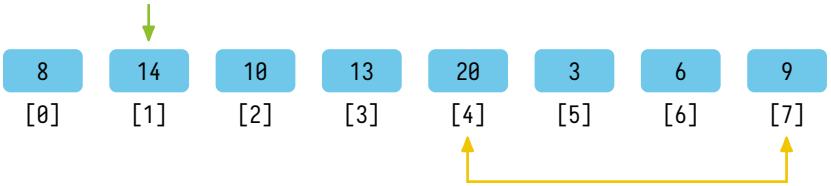
Partitioning an Array Using the Last Element (9) as a Pivot



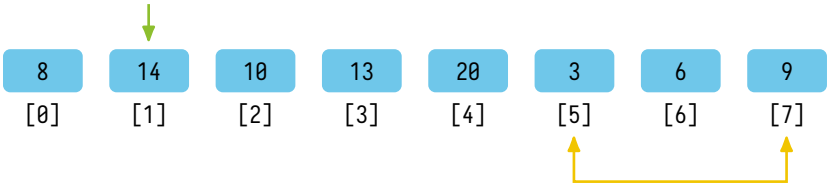
Partitioning an Array Using the Last Element (9) as a Pivot



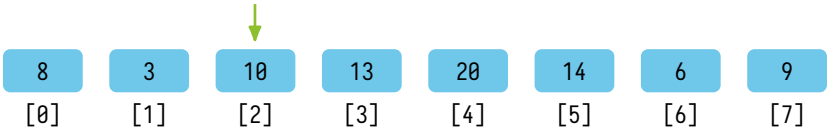
Partitioning an Array Using the Last Element (9) as a Pivot



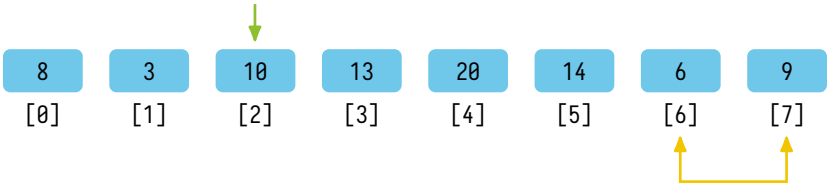
Partitioning an Array Using the Last Element (9) as a Pivot



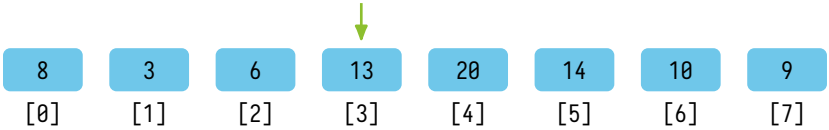
Partitioning an Array Using the Last Element (9) as a Pivot



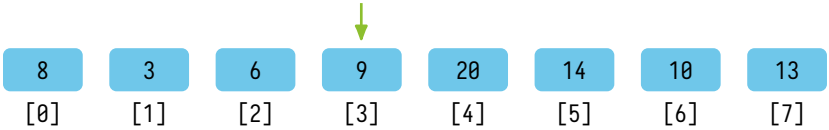
Partitioning an Array Using the Last Element (9) as a Pivot



Partitioning an Array Using the Last Element (9) as a Pivot



Partitioning an Array Using the Last Element (9) as a Pivot



Quicksort Recursively Sorts Both Sides of the Partition

```
void quickSortHelper(int array[], int low, int high) {
    if (low >= high) {
        return;
    }
    int pivot = partition(array, low, high);
    quickSortHelper(array, low, pivot - 1);
    quickSortHelper(array, pivot + 1, high);
}

void quickSort(int array[], int arrayLength) {
    quickSortHelper(array, 0, arrayLength - 1);
}
```

Partition Code as Seen from a Search

```
int partition(int array[], int low, int high) {
    int pivot = array[high];
    int i = low - 1; /* Temporary pivot index */
    for(int j = low; j <= high; j++) {
        if(array[j] < pivot) {
            /* Move the pivot one element forward */
            ++i;
            swap(&array[i], &array[j]);
        }
    }
    ++i;
    swap(&array[i], &array[high]);
    return i;
}
```

Partition Code Re-written for Clarity

```
int partition(int array[], int low, int high) {
    int pivot = array[high];
    int i = low;
    for(int j = low; j < high; j++) {
        if(array[j] < pivot) {
            if (i != j) {
                swap(&array[i], &array[j]);
            }
            ++i;
        }
    }
    swap(&array[i], &array[high]);
    return i;
}
```

There’s Also Joke Sorting Algorithms

There’s a sorting algorithm called bogosort, which “work” but NEVER use

If you want to use bogosort to sort a deck of cards:
Throw them in the air, pick them up randomly, if they’re not sorted repeat

There's Also Joke Sorting Algorithms

There's a sorting algorithm called bogosort, which "work" but NEVER use

If you want to use bogosort to sort a deck of cards:
Throw them in the air, pick them up randomly, if they're not sorted repeat

We can do even worse, called bozosort:
randomly switch two cards and see if it's sorted yet, if not repeat

This is a Sorting to NEVER Use

```
bool inOrder(int array[], int arrayLength) {
    for (int i = 1; i < arrayLength; ++i) {
        if (array[i - 1] > array[i]) {
            return false;
        }
    }
    return true;
}
void bozoSort(int array[], int arrayLength) {
    while (!inOrder(array, arrayLength)) {
        int i = rand() % arrayLength;
        int j = rand() % (arrayLength - 1);
        if (j >= i) {
            ++j;
        }
        swap(&array[i], &array[j]);
    }
}
```

Quicksort is Part of the C Standard Library

The function prototype for quicksort is:

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void*, const void*));
```

The arguments are:

- `base`: the starting address of the array to sort
- `nmemb`: the length of the array (number of elements)
- `size`: the size (in bytes) of each element
- `compar`: a function that takes a pointer to two elements and returns a result
  - `-1` if the first argument is less than the second
  - `1` if the first argument is greater than the second
  - `0` if the arguments are equal
  - (Note: this the same as `strcmp`)

Using qsort to Sort an Array of Integers

```
int compare(const void *a, const void *b) {
    int x = *((const int *) a);
    int y = *((const int *) b);
    if (x < y)      { return -1; }
    else if (x > y) { return 1; }
    else            { return 0; }
}

int main(void) {
    int array[] = {10, 14, 8, 13, 20, 3, 6, 9, 4};
    int arrayLength = ARRAY_LENGTH(array);
    qsort(array, arrayLength, sizeof(int), compare);
    printArray(array, arrayLength);
    return EXIT_SUCCESS;
}
```

We Could Use qsort to Sort Program Arguments

```
int compare(const void *a, const void *b) {
    const char **x = (const char **) a;
    const char **y = (const char **) b;
    return strcmp(*x, *y);
}

int main(int argc, const char *argv[]) {
    if (argc < 2) {
        return EXIT_FAILURE;
    }
    qsort(argv + 1, argc - 1, sizeof(const char *), compare);
    for (int i = 1; i < argc; ++i) {
        printf("%s\n", argv[i]);
    }
    return EXIT_SUCCESS;
}
```

You'll Use Sorting Algorithms Instead of Writing Them

However, writing sorting algorithms are excellent C practice,
small errors produce the wrong result, or memory errors

`qsort` is the most difficult sorting function to use since
you really have to understand memory, and the limitations of C
C has to use `void*` to be general and support different types

The primary design goal of C++ is to make operations
such as sorting easier to use and more efficient