# CMPT 125

# Introduction to Computing Science and Programming II

October 20, 2021

# Midterm – Oct 25

- Midterm will be on October 25, during the Monday class.
- The exam will be pen and paper.
- Format: 4 question each with several items
- Closed books. Only pens/pencils are allowed.
- Please bring your student IDs
- The material includes everything learned before the midterm (including Oct 18-20)
- For previous exams go to https://www.cs.sfu.ca/~ishinkar/teaching/fall21/cmpt125/exams.html
  (In some offerings the midterm was a bit later, and covered more material)
- Solve all practice problems on piazza: The links are under Resources

# Quick Sort

# Quick Sort

- Given an array,
  - Choose an element in the array, call it the ***pivot***.
  - Rearrange the elements in the array so that:
    - All elements < pivot are to the left of pivot.
    - All elements >= pivot are to the right of pivot.
  - *Recursively* sort to the left of the pivot.
  - *Recursively* sort to the right of the pivot.

# Quick Sort

- Q1: how should we choose the pivot?
  - At random
  - Let pivot = a[mid point]
  - Maybe we know something about the array…

# Quick Sort

Q2: how do we rearrange the elements?

Example: Input = [4,1,8,**7**,10,6,12,3]

- Set pivot = ***7***
- Swap pivot with the a[start] - [7,1,8,4,10,6,12,3]
- Have two pointers - [7,***1***,8,4,10,6,12,***3***]
- We are going to move the left pointer to the right,
  making all elements to its left to be smaller than pivot.
- Similarly, we move the right pointer to the left,
  making all elements to its right to be larger than pivot.

# Quick Sort

Input = [7,1,8,4,10,6,12,3], pivot = ***7***

We'll have two pointers - [7,***1***,8,4,10,6,12,***3***]

Move left pointer - [7,1,***8***,4,10,6,12,***3***]

8>pivot -- stop

Move the right pointer - [7,1,**8**,4,10,6,12,***3***]

3<pivot -- stop

Swap 3 and 8 - [7,1,***3***,4,10,6,12,***8***]

Move left pointer 3->4->10 - [7,1,3,4,***10***,6,12,***8***]

10>pivot -- stop

Move right pointer 8->12->6 - [7,1,3,4,***10***,***6***,12,8]

6<pivot -- stop

Swap 10 and 6 - [7,1,3,4,***6***,***10***,12,8]

Swap ***pivot*** with **6** - [**6**,1,3,4,***7***,10,12,8]

# Quick Sort

- Running time – *$O(n \log(n))$* for good pivots
- Q: What is a good pivot?
- A:The one that splits the array into equal halves
    - Indeed: $T(n) = \boxed{O(1)} + \boxed{O(n)} + \boxed{2*T(n/2)}$

      Choose a pivot | Rearranging | Recursion

    - <u>Fact:</u> $T(n) = O(n \log(n))$
    - Saw in the analysis on Merge sort

# Quick Sort

- A different analysis for good pivot:
- Let’s count how many times each element it touched.

  1. Each element can be a pivot at most once.
  2. Each element is swapped at most $\log_2(n)$ times.
- Proof:
  - Each element is swapped *at most once* in each rearrangement procedure.
  - Q: How many times does an element appear in a rearrangement procedure?
  - A: The size of the array containing the element is divided by two each time.
  - So each element appears in at most $log_2(n)$ rearrangement procedure.
  - Therefore, each element is touched/swapped at most $log_2(n)$ times.
- Therefore, the total running time is $O(n*log(n))$.

# Quick Sort

- Homework: What if the array is not split evenly, but say n/3 – 2n/3?
  $T(n) = O(n) + T(n/3) + T(2n/3)$

  Prove that $T(n) = O(n \log(n))$

- Homework: What if pivot is the maximal element?
  $T(n) = O(n) + O(1) + T(n-2)$

  Then $T(n) = n+(n-2) + (n-4) + (n-6) + \ldots + 2 = O(n^2)$

# Quick Sort

- How can we choose a good pivot?

- <u>Heuristics:</u>

1. Choose a random element
2. Choose 3 random elements and pick the median of the three
3. Compute the median
    - How? ~~Sort the array, and take the median.~~ NO, WAIT, WHAT?
    - There is a linear time algorithm for computing the median… But!
    - The running time is O(n), but the constant is quite large.
    - Not used in practice.

# Quick Sort

- Running time – *O(n log(n))* for good pivots.
- More efficient than quadratic sorts (selection sort, insertion sort).
- It is very efficient for arrays sets that are already substantially sorted.
- In-place comparison sort, i.e., requires O(1) additional memory.

# Comparing to MergeSort

- Running time – O(n log(n)) **worst case.**
- It is very efficient for arrays sets that are already substantially sorted.
- **Requires O(n) additional memory.**

# Homework

- Implement each of the algorithms in C.

# qsort()

# qsort() in C

*#include <stdlib.h>*

*void qsort(void *array, int n, size_t size, int (*compar)(const void *, const void*))*

- *The function gets an array size n of any type, and sorts it.*
- *size specifies the size of the elements of the array*
- *compar() is a <u>pointer to a function</u> used to compare two elements.*
  *Returns <0 if the first argument is smaller*
  *Returns 0 if the arguments are equal*
  *Returns >0 if the first argument is greater*

# qsort() in C - example

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

// returns a positive number if *a > *b, returns negative if *a < *b
int cmpr_ints (const void * a, const void * b)
{
	return ( *(int*)a - *(int*)b );
}

int main () {
	int values[] = { 88, 56, 100, 2, 25 };


	qsort(values, 5, sizeof(int), cmpr_ints);

	for( int i = 0 ; i < 5; i++ )
		printf("%d  ", values[i]);

	return 0;
}
```

# qsort() in C - example

```
typedef struct student_info {
	char* first_name;
	char* last_name;
	int ID;
	int grades[5];
} student;



// compares students by ID
int cmpr_students (const void * a, const void * b)
{
	return  ( (student*)a)->ID – ((student*)b)->ID;
}

int main () {
	student arr[] = …;

	qsort(arr, 180, sizeof(student), cmpr_students);
}
```

# Lower bounds

# Can we do better than n log(n)?

**Theorem**: In the *comparison model* any sorting algorithm
runs in time at least *N log(N)*.

**(not a) Proof**: There are n! different permutations, in each comparison we get only “one bit of information”, and $\log_2(n!) > n\log(n)/2$.

*Comparison model*:

- no prior information on the data.
- Can only compare elements: which one of the two is smaller?

Examples that are **not** in the comparison model:

- Array of length N containing each of the number *1…N* once.
- Array contains only number *1…100*

Questions?

Comments?