Analysis of Algorithms

Introduction and Mergesort

Assoc. Prof. Dr. Burkay Genç

Spring 2023

Introduction

Syllabus

The course syllabus can be found at my website

You can also download all lecture slides from the same web page.

Grading

There will be one midterm and one final exam.

- 40% Midterm
- 60% Final Exam

or

- 2x30% Assignments
- 40% Final Exam

Algorithm

- An *algorithm* is a set of instructions to be followed to solve a problem.
  - There can be *more than one solution* (more than one algorithm) to solve a given problem.
  - An algorithm can be implemented using *different programming languages* on different platforms.
- An algorithm must be **correct**. It should correctly solve the problem.

Why Analyze An Algorithm?

- To discover its characteristics
  - to evaluate its suitability for various applications
  - to compare it with other algorithms
- The characteristics to be discovered are
  - time : how long does it take to run the algorithm on a given input
  - space : how much memory do we need to run the algorithm on a given input

Independent of Implementation

- We try to keep the analysis independent of the implementation
  - quality of implementation
  - properties of compilers
  - machine architectures
  - other facets of programming (oop vs procedural, interpreted vs compiled)

Other Concerns

- On a mobile device battery consumption of an algorithm
- On a numerical problem the accuracy may be a concern
- Sometimes multiple concerns maybe handled
  - A fast algorithm that requires a lot of space
  - A slow algorithm that requires much less space

Two Types of Analysis

- Aho, Hopcroft, Ullman, Cormen, Leiserson, Rivest, Stein
  - determine the growth of the worst-case performance of the algorithm
  - referred to as “theory of algorithms”
- Knuth
  - precise characterizations of the best, worst and average-case performances
  - build mathematical models to describe the performance of real-world algorithm implementations

Theory of Algorithms

Notation

Definition Given a function $f(N)$,

- $O(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from *above* as $N \to \infty$
- $\Omega(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from *below* by a (strictly) positive number as $N \to \infty$
- $\Theta(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from *both above and below* as $N \to \infty$

Knuth (1976)

Notation

- $O(f(N))$ denotes an upper bound
- $\Omega(f(N))$ denotes a lower bound
- $\Theta(f(N))$ denotes matching upper and lower bounds

Big-Oh

Typically used to

- hide constants
- express a *small* error term
- bound the worst case

Hiding Constants

When you hide the constants derivations become simpler:

- natural logarithm $\ln N \equiv \log_e N$
- binary logarithm $\lg N \equiv \log_2 N$
- $\ln N = O(\log N)$ and $\lg N = O(\log N)$
  - that is because $\lg N = \frac{\log N}{\log 2} = \frac{1}{\log 2} \log N = c \log N$

Mergesort

- Sorting is a fundamental problem in computer science
  - Given a list of numbers in an array, sort them in ascending order
- Mergesort is a well-known efficient algorithm for sorting
  - divide the array in the model
  - sort each half separately (recursively)
  - merge the two halves

Mergesort

```
private void mergesort(int[] a, int lo, int hi)
{
  if (hi <= lo) return;
  int mid = lo + (hi - lo) / 2;
  mergesort(a, lo, mid);
  mergesort(a, mid + 1, hi);
  for (int k = lo; k <= mid; k++)
    b[k-lo] = a[k];
  for (int k = mid+1; k <= hi; k++)
    c[k-mid-1] = a[k];
  b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
  int i = 0, j = 0;
  for (int k = lo; k <= hi; k++)
    if (c[j] < b[i]) a[k] = c[j++];
    else a[k] = b[i++];
}
```

Three-way Merge Sort

If we were to split the array into three parts, sort each and then do a three-way merge, would it make a difference?

Mergesort Compares

Theorem (Mergesort Compares) Mergesort uses $N \lg N + O(N)$ compares to sort an array of N elements.

Proof. Let C_N be the number of compares for N elements. Then, the first half of the array requires C_{N-1} compares, as well as the second half. For the merge, we make N more compares. Hence,

$$C_N = C_{N/2} + C_{N/2} + N$$

Assume, $N = 2^n$. Then,

$$C_{2^n} = 2C_{2^{n-1}} + 2^n$$

...

Mergesort Compares

$$C_{2^n} = 2C_{2^{n-1}} + 2^n$$

Divide both sides by 2^n:

$$\frac{C_{2^n}}{2^n} = \frac{C_{2^{n-1}}}{2^{n-1}} + 1 = \frac{C_{2^{n-2}}}{2^{n-2}} + 2 = \frac{C_{2^{n-3}}}{2^{n-3}} + 3 = \ldots = \frac{C_{2^0}}{2^0} + n = n$$

Therefore,

$$C_{2^n} = 2^n n \implies C_N = N \lg N$$

We will later look into the general case (where $N \neq 2^n$)

Upper Bound for Sorting

- Ignoring details for now, we can assume a reasonable implementation of mergesort will result in a running time of a constant factor of $N \lg N$.
- From a theoretical point of view, mergesort provides an upper bound on sorting:

There exists an algorithm that can sort any N-element file in time proportional to $N \log N$.

- So, we can say that "time complexity of sorting is $O(N \log N)$"

Lower Bound for Sorting

Theorem (Complexity of Sorting) Every compare based sorting program uses at least $\lceil \lg N! \rceil > N \lg N - N/(\ln 2)$ compares for some input.

Proof. We will not do a full proof, but the idea is as follows. Consider all permutations of the given array. That is, we have $N!$ different arrangements. With each comparison, the best you can do is get rid of half of the remaining arrangements (the ones that conflict the comparison result). Then, you need at least $\lceil \lg N! \rceil$ comparison to reach a single remaining arrangement, that is the sorted array. Using Stirling's approximation one can show that $\lceil \lg N! \rceil > N \lg N - N/(\ln 2)$

Lower Bound for Sorting

- From a theoretical point of view, this result provides a lower bound on sorting:

All compare-based sorting algorithms require time proportional to $N \log N$ to sort some N-element input file.

- Therefore, sorting is $\Omega(N \log N)$.
- Having the same upper and lower bounds is a very significant thing.
  - We can now claim sorting is $\Theta(N \log N)$.

Exercise

Suppose that it is known that each of the items in an N-item array has one of two distinct values. Give a sorting method that takes time proportional to N.

Exercise

Answer the previous exercise for three distinct values.