Administrant Details

This course is being taught by Daniel Panario, who can be reached by e-mail at *daniel@cs.toronto.edu*, by phone at *946-3510*, or at his office in the Sanford Flemming Building, room 3302c. The course will convene on Thursdays from 2:00pm till 4:00pm in Room 1078 of the Sidney Smith building.

1 Introduction

This year the course CSC 2414F, Topics in Applied Discrete Mathematics, will be focusing on *Mathematics for the Analysis of Algorithms*. The course will emphasize the development of mathematical tools for the analysis of algorithms by studying simple algorithms, in which the Big-Oh worst case resource bounds are easy to derive. This will permit us to focus on the techniques used in the analyses of the algorithms in question.

We must first understand what is meant by the term *analysis of algorithms*. We shall define it as a process which enables us to quantify and predict the behaviour and expected performance of algorithms under suitable probabilistic models. Examples of metrics which quantify performance include:

- worst case resource bounds,
- expected values,
- variances and other higher degree moments.

Metrics which quantify behaviour include:

- limit distributions,
- concentration results around the expected values.

Note: These terms will be defined later in the course, for the reader who is unfamiliar with them.

Topics which some people might associate with this course, but which will not be covered include:

Average-Case Complexity focuses on the innate computational difficulty of solving a problem on the average case input, and thus it is truly part of the field of computational complexity. People interested in this topic might be interested in the URL: *http://www.uncg.edu/mat/avg.html*.

Randomized Algorithms deal with algorithms which incorporate random coin tosses into their description. There will be some small intersection with this field because some of the algorithms which we look at will be randomized, but it is not the focus of this course. A very good introduction to this field is the book *Randomized Algorithms* by Motwani and Raghavan, by the Cambridge University Press; for more details see the URL: *http://www.cup.org/Reviews&Blurbs/RanAlg/RanAlg.html*

The Probabilistic Method is used to prove the existence of certain combinatorial structures in a non-constructive fashion, using probability theory. We will not discuss the probabilistic method in this course. However, many of the techniques in this course and in the probabilistic method seem very complimentary. The probabilistic method has traditionally been applied to problems in graph theory with great success; the techniques from this course have been applied to areas such as strings, permutations, words, and trees with great success. In addition, there are many cases in which one method has been successfully applied to problems traditionally in the domain of the other. Parties interested in the probabilistic method should contact Professor Mike Molloy and inquire about his course on the probabilistic method.

This course is about estimating characteristics related with time and space of particular algorithms. We study expectations, variances, limit distributions. Thus, some basic mathematical tools which are necessary to perform the analysis of algorithms include: *discrete probability, combinatorics, real and complex asymptotics.*

Historically, the field was started by Knuth in 1962/1963 when he studied the expected total displacement of linear probing hashing. As indicated by Knuth [7], this analysis was fundamental in his career. Knuth not only originated the field, but guided its development significantly through the publication of his series entitled *The Art of Computer Programming (TAOCP)*. Since the fields inception it has grown and matured considerably, and there are now many more tools available to computer scientists which permit for more accurate estimates of the behaviour of algorithms.

1.1 Steps in the Analysis of an Algorithm

Remark. The material of this section is from Vitter and Flajolet [10].

The first step in the analysis of an algorithm is to define an *input model* and a *measure of complexity*. If we let A be an algorithm, we can define the *input model* as datum represented in a specific format which has an associated notion of size. With each set of inputs of size n we associate a probabilistic distribution which models a real world input distribution for the algorithm. Examples of formatted input with an associated size include: arrays, where size is represented by the number of elements in the array; files and the number of records needed to store it; character strings and their length, etc.

The canonical *complexity measures* are the usage of space and time, and they can range from very specific to abstract. Examples of specific measures are the number of bits required to store the variables on a machine, in which the smallest accessible memory unit is the 32-bit word, or the number nanoseconds required to perform a certain comparison operation on a machine. More abstract measures include counting the number of comparisons required in a sorting routine, or the number of disk pages required to perform a database query.

The range of accuracy in the complexity measures implies that there are two approaches to analyzing algorithms. Firstly, one can consider the case in which the algorithm is implemented on a specific machine, where the complexity measures are dictated by the machine in question. Secondly, one can consider more abstract measures, which are independent of any particular implementation. Although we will be concentrating on the second type of analysis in this course, we should not forget that the first type has its uses. People in industry often perform a detailed analysis on an algorithm, for a specific implementation, *if the analysis is worth the effort.*

Let A be an algorithm, I_n the set of inputs of size n over an associated probability distribution, and a measure function $\mu : \bigcup_{i=j}^{\infty} I_j \to \mathbb{N}$. We define two types of complexity:

Worst Case Complexity:

$$\mu_n^{\text{Worst}}[A] = \max\{\mu_n[\alpha] | \alpha \in I_n\}.$$

Average Case Complexity:

$$\mu_n^{\text{Average}}[A] = E_{\alpha \in I_n}[\mu_n(\alpha)] = \sum_{k=1}^{\infty} k \cdot \Pr_{\alpha \in I_n}(\mu_n(\alpha) = k).$$

Where μ_n is μ restricted to the domain I_n.

At an abstract level, many average-case analyses of algorithms trace the steps demonstrated in Figure 1.

Initially a study of the algorithm first reveals which variables are of interest and of use to explain its behaviour. Traditionally, after these variables have been determined they are represented in a recurrence which relates the variables of interest, on the algorithm of size n, to the same variable for the algorithm when run on inputs of a smaller size. We can then solve the recurrence directly or through the use of generating functions. This gives an explicit expression for the variable, which can then be asymptotically classified using real analysis techniques. For instance, sums can be estimated by integrals (Euler-Maclaurin formula).

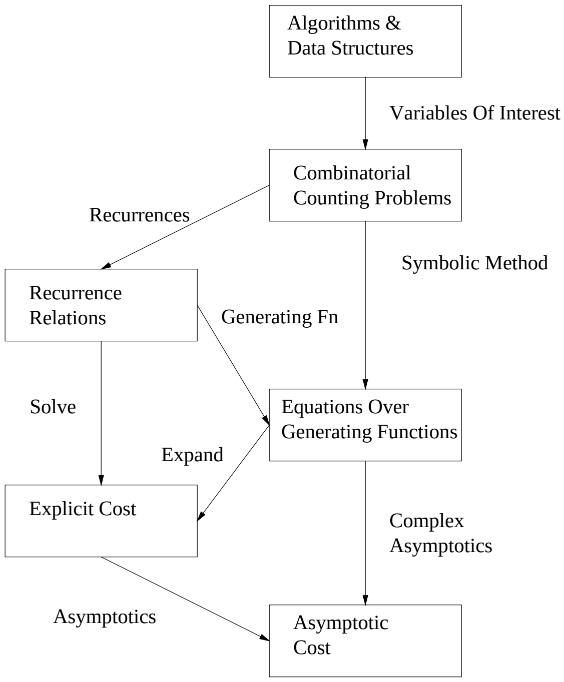

Figure 1: An outline of different routes to take in the Analysis of Algorithms

Sometimes a shortcut will be possible. Methods have been developed so that we can progress directly from determining the variables of interest to identifying an equation involving generating functions which represents the variable in question. This process is known as the *symbolic method.* Finally, we can now jump directly from an equation involving generating functions to an asymptotic classification by using complex asymptotics. The Cauchy integral formula plays a central role in developing this process.

1.2 Analysis of Quicksort

Remark. The material of this section is taken from Sedgewick and Flajolet [9], Chapter 1, and from Knuth [6], Section 5.2.2. Other sources and material for further study are [2, 3, 5, 8].

Quicksort was introduced by Hoare [4]. The analysis is due to Knuth, Sedgewick and others in the 70's. It is well known that the worst case behaviour of the algorithm is $O(n^2)$, which occurs when the input is initially sorted, and the first element in the array is used as a pivot. However, the average case analysis is $O(n \log n)$. Further, several variants of Quicksort have been proposed by Hoare and other researchers, such as partitioning around the median of the first, last and middle element being sorted; stopping the recurrence when the array size hits a predetermined value M, and then switching to another sorting routine; randomly selecting the partition element from the array, and countless others. Many of these variants have been shown to have smaller constants in the $O(n \log n)$ average-case analysis, than did Hoare's original algorithm (see [6] for references). Below the Quicksort algorithm is defined.

Algorithm 1 Quicksort(l , r : integer)

```
var val, aux, i, j: integer;
begin
    if r ≥ l then
    begin
        val←a[l];i←l;j←r+1;
        repeat
            repeat
                i←i+1;
            until a[i] ≥ val;
            repeat
                j←j-1;
            until a[j] ≤ val;
            aux←a[j];a[j]←a[i];←a[i]←aux;
        until j≤ i;
        a[i]←a[j];a[j]←a[l];a[l]←aux;
        Quicksort(l,i-1);
        Quicksort (i+1,r);
    end
end
```

We leave it to the readers to trace the algorithm and convince themselves that it is correct; we will show a simple analysis of this algorithm. We make two assumptions in this analysis. Firstly, we assume that the keys are distinct, which permits the assumption of a unambiguous ordering of keys. Secondly, we assume that after the partitioning process the elements in both sub-arrays remain in a random order (algorithms which break this assumption have been shown to be slower; see [6], p. 119).

The first step in the analysis of an algorithm is to determine which operations performed can be used to properly quantify the cost of the algorithm. For the Quicksort algorithm we recognize the following quantities:

A - the number of partitioning steps,

B - the number of exchanges,

C - the number of comparisons.

A similar argument to the one in Knuth's Vol 3. of TAOCP and on Sedgewick and Flajolet adapted to the version of Quicksort in Algorithm 1 above implies that on the MIX computer the running time of Quicksort is about $4C + 11B + 35A$. Given that the MIX machine supports an instruction set similar to that on most modern CPUs, it can be argued that this probably is close to the running time on most modern machines.

The problem now is one of estimating the values A, B, and C.

Theorem 1 *To sort an array of n randomly ordered, distinct elements Quicksort uses on the average:*

- *n partitioning stages,*
- $2(n+1)(H_{n+1} - 1) \approx 2n \ln n - 0.846n$ *comparisons,*
- $\frac{(n+1)}{3}(H_{n+1} - \frac{5}{2}) + \frac{1}{2} \approx \frac{1}{3} n \ln n - 1.346n$ *exchanges.*

Where $H_n = \sum_{k=1}^{n} \frac{1}{k}$ is the nth harmonic number.

Proof.

The *partition case* is trivial since in this implementation of Quicksort each element is partitioned once.

To determine the number of comparisons we count the number of them in the first stage, and add it to the expected number of comparisons in the recursive process. Let C_i represent the expected number of comparisons on an input of size i. The terminal case is described by $C_0 = 0$ and is made obvious by looking at the pseudo-code. The recurrence $C_n = n + 1 + \frac{1}{n} \sum_{1 \leq j \leq n} (C_{j-1} + C_{N-j})$

(for $n > 0$) can be derived as follows: the $n+1$ term follows from the fact that each element but the first in the array is compared once when the index i or j is pointing to its position in the array. Index i starts on the second element in the array and moves towards the middle, whereas j starts at the last element of the array and moved towards the middle. The index i and j are updated until i is greater than j, at which point there are two elements which have been compared twice. Summing the number of comparisons gives $(n-1)+2 = n+1$.

Due to our previous assumptions, the probability of partitioning on an element j is $\frac{1}{n}$, and this leads us to the term $\frac{1}{n}\sum_{1\leq j\leq n}(C_{j-1} + C_{N-j})$ which is the expected number of comparisons in the recursive calls to the two sub-arrays which are created.

Observe that $\sum_{j=1}^{n}(C_{j-1} + C_{N-j}) = 2\sum_{j=1}^{n} C_{j-1}$, and so we can rewrite the recurrence as

$$C_n = (n+1) + \frac{2}{n}\sum_{j=1}^{n} C_{j-1}.$$

This type of recurrence is called a *full history recurrence* since the value of C_n is dependent on the entire recursive history, $C_0, C_1, ..., C_{n-1}$, and can normally be quite difficult to solve (see [3]). Hopefully, in this case there is a telescopic cancellation which can be shown subtracting

$$nC_n = (n+1)n + 2\sum_{j=1}^{n} C_{j-1}$$

from

$$(n-1)C_{n-1} = n(n-1) + 2\sum_{j=1}^{n-1} C_{j-1}$$

that is,

$$nC_n - (n-1)C_{n-1} = 2n + 2C_{n-1} \qquad \text{(for } n > 1\text{)}.$$

This gives

$$nC_n = 2n + (n+1)C_{n-1},$$

and dividing by $(n+1)n$ results in:

$$\frac{C_n}{n+1} = \frac{C_{n-1}}{n} + \frac{2}{n+1} \qquad \text{(for } n > 1\text{)}.$$

Iterating gives:

$$\frac{C_n}{n+1} = \frac{C_1}{2} + \sum_{k=3}^{n+1}\frac{2}{k},$$

and by the definition of the recurrence we know that $C_1 = 2$, and thus:

$$\frac{C_n}{n+1} = 2\sum_{k=2}^{n+1}\frac{1}{k} = 2\left[\sum_{k=1}^{n+1}\frac{1}{k} - 1\right].$$

Which implies that:

$$C_n = 2(n+1)(H_{n+1} - 1).$$

We will see later in the course that an application of the Euler-Maclaurin formula gives:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O\left(\frac{1}{n^4}\right),$$

where $\gamma = 0.5772156649....$ is Euler's constant (introduced by Euler in 1734; see [1]).

Substituting in with the approximation we arrive at:

$$\begin{aligned} C_n = 2(n+1)(H_{n+1} - 1) &\approx 2n(\ln n + \gamma - 1) \\ &= 2n\ln n + 2(\gamma - 1)n \\ &\approx 2n\ln n - 0.846n. \end{aligned}$$

In a similar way we can estimate the number of exchanges B_N. We have the following recurrence

$$B_N = \frac{N-2}{6} + \frac{1}{N}\sum_{j=1}^{N}(B_{j-1} + B_{N-j}).$$

This recurrence can be solved in a similar fashion to the way the recurrence for C_N. The number of exchanges in the partitioning part, $(N-2)/6$, can be deduced from the equality below (this requires more justification; see Assignment 1)

$$\frac{1}{N}\sum_{j=2}^{N-1}\frac{(j-1)(N-j)}{N-1} = \frac{N-2}{6}.$$

This equality can be derived as follows

$$\begin{aligned} \frac{1}{N}\sum_{j=2}^{N-1}\frac{(j-1)(N-j)}{N-1} &= \frac{1}{N-1}\sum_{j=2}^{N-1}(j-1) - \frac{1}{N(N-1)}\sum_{j=2}^{N-1} j(j-1) \\ &= \frac{1}{N-1}\cdot\frac{(N-2)(N-1)}{2} - \frac{1}{N(N-1)}\cdot\left(\frac{N^2}{3} - N + \frac{2}{3}\right)N \\ &= \frac{N-2}{2} - \frac{1}{N-1}\cdot\frac{N^2-3N+2}{3} \\ &= \frac{N-2}{6}. \end{aligned}$$

Finally, for partitions we have the similar recurrence

$$A_N = 1 + \frac{1}{N}\sum_{j=1}^{N}(A_{j-1} + A_{N-j}).$$

If we wanted to know the total cost for this Quicksort implementation on the particular machine we described above, we use the Quicksort Theorem and the equation $4C + 11B + 35A$ to obtain $11.667N \ln N - 16.810N$ steps. We see that "comparisons" is the essential component of the cost with "exchanges" contributing to the main cost and "partitioning" only contributing to the second order term, i.e., the term in N. Note carefully that the constants are machine dependent. We are more concerned about the values for A, B and C which are *not* machine dependent. Indeed, such quantities will be the focus in other algorithms that we will study.

One can consider many improvements to the given quicksort algorithm. Indeed these improvements have been successfully analyzed. For instance, if we alter the algorithm to run Insertion sort for sizes less than or equal to M, then we get the following recurrence for the number of comparisons

$$C_N = \begin{cases} (N+1) + \frac{1}{N}\sum_{j=1}^{N}(C_{j-1} + C_{N-j}) & \text{for } N > M; \\ \frac{N(N-1)}{4} & \text{for } N \le M. \end{cases}$$

This recurrence can be solved in a similar way. The new closed-form result for C_N will have expressions like $H_{N+1} - H_M$ instead of H_{N+1}.

For the "median-of-three" variant the resulting recurrence is

$$C_N = (N+1) + \sum_{j=1}^{N}\frac{(N-j)(j-1)}{\binom{N}{3}}(C_{j-1} + C_{N-j}), \qquad N > 3,$$

where $\binom{N}{3}$ is the number of ways to choose 3 out of N items, and the probability that the jth smallest element is the partitioning element is

$$\frac{(N-j)(j-1)}{\binom{N}{3}}$$

instead of $1/N$. For solving this kind of recurrence, we will need more powerful mathematical tools such as generating functions. Moreover, the symbolic method will allow us to derive other moments. For instance, we will see that the variance of Quicksort is

$$7N^2 - 4(N+1)^2\sum_{k=1}^{N}\frac{1}{k^2} - 2(N+1)H_{N+1} + 13N \approx \left(7 - \frac{2\pi^2}{3}\right)N^2.$$

(Note that this approximation makes use of the fact that $\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{1}{6}\pi^2$.) Hence, the standard deviation is approximately $\sqrt{7 - 2\pi^2/3}N$. This means that *we have concentration around the mean.* That is, the standard deviation is less than the expected value.

References

[1] L. Euler. De progressionibus harmonicis observationes. *Comentarii academiæ scientiarum imperialis Petropolitanæ*, 7:150–161, 1734. Reprinted in his Opera Omnia, series 1, volume 14, 87-100.

[2] R. Graham, D.E. Knuth, and O. Patashnik. *Concrete Mathematics.* Addison-Wesley, Reading, MA, 2 edition, 1994.

[3] D.H. Greene and D.E. Knuth. *Mathematics for the analysis of algorithms.* Birkhauser, Boston, 3 edition, 1990.

[4] C.A.R. Hoare. Quicksort. *Computer Journal*, 5:10–15, 1962.

[5] D.E. Knuth. *The art of computer programming, vol.1: fundamental algorithms.* Addison-Wesley, Reading MA, 3 edition, 1997.

[6] D.E. Knuth. *The art of computer programming, vol.3: sorting and searching.* Addison-Wesley, Reading MA, 2 edition, 1998.

[7] D.E. Knuth. Linear probing and graphs. To appear in *Algorithmica*, 1998.

[8] R. Sedgewick. *Quicksort.* Garland Publishing, New York, 1980.

[9] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, Reading MA, 1996.

[10] J.S. Vitter and P. Flajolet. Analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science A: Algorithms and Complexity*, pages 431–524. Elsevier, Amsterdam, 1990.