**Introduction to Algorithms fourth edition Notes**

Contents

[Collected Asymptotic Equations/inequalities 2](#_Toc114671431)

[PART 1: 2](#_Toc114671432)

[Chapter 1: Intro to Algorithms 2](#_Toc114671433)

[Chapter 2: Getting Started, insertion sort, divide and conquer (merge sort) 3](#_Toc114671434)

[Chapter 2.1 Insertion Sort 3](#_Toc114671435)

[2.2 Analysing Algorithms. 5](#_Toc114671436)

[2.3 Designing Algorithms/Divide and Conquer introduction 5](#_Toc114671437)

[Chapter 3: Characterizing runtimes, asymptotic notation Θ,Ω, O 6](#_Toc114671438)

[3.1 O-,-,- notation 6](#_Toc114671439)

[3.2 Formal definition of Ω, O, and Θ. 6](#_Toc114671440)

[3.3 Math Terms 8](#_Toc114671441)

[Chapter 4: Divide and Conquer 11](#_Toc114671442)

[4.1 Multiplying Square Matrices 12](#_Toc114671443)

[4.2 Strassen’s algorithm 13](#_Toc114671444)

[4.3 Substitution method for solving recurrences 14](#_Toc114671445)

[4.4 Recurrence Trees 15](#_Toc114671446)

[4.5 Master Theorem 18](#_Toc114671447)

[4.6 Proof of Master Theorem 19](#_Toc114671448)

[4.7 Akra-Bazzi recurrence and method 19](#_Toc114671449)

[Chapter 5 Probabilistic Analysis and Random Algorithms 20](#_Toc114671450)

[Chapter 5.1 The Hiring Problem 20](#_Toc114671451)

[5.2 Indicator Random Variables 21](#_Toc114671452)

[5.3 Randomizing Algorithms 22](#_Toc114671453)

[5.4.2 Further applications of indicator random variables – tossing balls into bins 23](#_Toc114671454)

[Part II Sorting and Ordering Statistics 24](#_Toc114671455)

[Chapter 6 Heapsort 25](#_Toc114671456)

[6.1 Heaps 25](#_Toc114671457)

[Chapter 6.2 Max-Heapify 25](#_Toc114671458)

[6.3 Building a Heap 26](#_Toc114671459)

[6.4 Heapsort algorithm 27](#_Toc114671460)

[6.5 Priority Queues 28](#_Toc114671461)

[Chapter 7 QuickSort 30](#_Toc114671462)

[7.1 Description of QuickSort 30](#_Toc114671463)

[7.2 Intuition about the runtime of QuickSort 31](#_Toc114671464)

[7.3 Randomized QuickSort 32](#_Toc114671465)

[7.4 Analysis of QuickSort 32](#_Toc114671466)

[Chapter 8 Sorting in Linear Time 35](#_Toc114671467)

[8.1 Lower Bound for Comparison Sorts 35](#_Toc114671468)

[8.2 Counting Sort 35](#_Toc114671469)

[8.3 Radix Sort 36](#_Toc114671470)

[8.4 Bucket Sort 36](#_Toc114671471)

[Chapter 9, Medians and Order Statistics 37](#_Toc114671472)

[9.1 Minimum and Maximum 37](#_Toc114671473)

[9.2 Selection in Expected Linear Time 38](#_Toc114671474)

[9.3 Worst Case Linear Selection 39](#_Toc114671475)

[Part 3 – Data Structures 40](#_Toc114671476)

# Collected Asymptotic Equations/inequalities

# PART 1:

Chapter Summary

Chapter 1: Intro to Algorithms

Chapter 2: Sorting algorithms, divide-and-conquer (merge sort), insertion sort

Chapter 3: asymptotic notation and math notation

Chapter 4: Divide and conquer algorithms, Strassen’s algorithm, multiplying square matrices, solving recursions, master method, substitutions

Chapter 5: Complexity and probability distributions, distribution of input data and randomized actions in the algorithm itself.

## Chapter 1: Intro to Algorithms

Algorithm, sequence of computational steps that turn input into output in finite amount of time. Tool for solving *computational problem*. A computational problem defines the desired input/output relationship.

E.g. of computational problem, sorting problem, specific set of numbers is called an *instance* of a sorting problem.

An algorithm for a problem is *correct*  if it halts in finite time and outputs the correct answer. Incorrect algorithms can be useful if you can control their error rate.

Here is what to look forward to:

Chapter 22, graph sorts

Chapter 20, topological sorting (parts made of other parts, sorting a library of parts based on dependency of the parts

Chapter 33, clustering

Chapter 15, compression, Huffman encoding.

Chapter 14, dynamic programming, DNA mapping

Chapter 22, finding routes on which data travels

Chapters 11 and 32, search engines

Chapter 31, public key cryptography, number theory.

Chapter 29, cost function minimization, linear equations

Chapter 30, fast fourier transform.

Techniques for developing algorithms: Chapter 2 and 4, divide-and-conquer. Chapter 14 dynamic programming, chapter 16, amortized analysis.

Chapter 34, NP-complete problems

Chapter 35, approximation algorithms (for approximating solutions to NP-Complete problems).

Chapter 26, parallel computing algorithms.

Chapter 27, online algorithms.

Exercise 1.1-4, difference between shortest path and travelling salesman. Finding shortest distance. Visiting all points vs visiting only necessary ones.

## Chapter 2: Getting Started, insertion sort, divide and conquer (merge sort)

### Chapter 2.1 Insertion Sort

Insertion-Sort, the part of the array that has already been sorted, A[1,i-1], is *loop-invariant*, it retains its sort throughout all iterations of the outer for loop.

Loop invariance help us show an algorithm is *correct*. To do this we rely on three things:

1. Initialization, proposition is true before first iteration of the loop
2. Maintenance, If proposition is true before iteration of loop it is also true before the next iteration.
3. Termination, the loop terminates and the invariant (usually also the reason it terminates) gives us a useful property that shows the algorithm is correct.

If (1) and (2) then the loop invariant is true before every iteration of the loop. This is similar to induction. However, induction usually doesn’t terminate. In loop invariance we use termination to show something about the algorithm e.g. that it is correct.

Proof of invariance of A[1,i-1] in insertion sort’s outer for-loop:

1. Initialization, Before the first loop i = 2 so the array is just A[1], this is one element and is sorted.
2. Maintenance, loosely. At A[i] we assume A[1]…A[i-1] is sorted. In this loop A[i-k] 🡪 A[i-(k+1)] if A[I-k] < A[i] and A[i] 🡪 A[i-(j+1)] where A[i-j] < A[i]. This preserves the established order and puts A[i] in the correct place. Therefore, when we start the next iteration at A[i+1], A[i] is sorted and we’ve shown maintenance.  
     
   Note that to show this properly, you’d have to show a loop invariant for the inner while loop.
3. Termination. Loop terminates when i = n+1. The loop invariant states that at step A[i], A[i-1] will be sorted so at termination if we consider A[n+1] then we see that A[n] is sorted.

Details of PseudoCode

An object’s attributes are accessed with “.” E.g. x.f to get attribute f of object x. If y is an object of the same class and y.f = x.f then if x.f changes so does y.f

On the other hand, methods get their own copies of variables. Arrays and objects passed to methods are passed as pointers.

Exercise 2.1-5, loop invariance for linear search.

Linear-Search(A, x, n)

for i = 1 to n

#Check if A[i]==x and if so terminate and return I

#Terminate and return NIL

Invariant: at ith iteration of loop, none of values in A[1:i-1] == x.

Initialization: Start at i = 1, A[1:i-1] is NIL.

Maintenance: At i nothing in A[1:i-1] == x, in this loop, check A[i]==x. If false move onto i+1 and now A[1:i] doesn’t have x.

Termination: If A[i] == x is True for any i terminate and return i, the correct answer. If nothing in the Array is x then halt at iteration n + 1. At the beginning of this iteration we know x is not in A[1:n]

### 2.2 Analysing Algorithms.

Analyze the speed of an algorithm to determine best one. To analyse speed we use the RAM model which says every instruction and data access on the machine take the same, constant, amount of time.

We confine ourselves to instructions common to computers: add, subtract, load, store, copy, conditional/unconditional branch, subroutine call and return. These are arithmetic, data movement, and command respectively. Data types are INT, FLOAT and CHAR.

When we analyse the speed of an algorithm we will consider the input size as the main variable. This is the size of the array in insertion sort but is the total number of bits needed to represent the input in an addition problem. For a graph there may be two numbers, one for vertices, one for edges.

Running time is the number of instructions and data accesses carried out.

When assessing run time we’ll simplify by usually just considering worst case for input size n. We do this because it gives an upper bound, the worst case is sometimes the most common (e.g. searching for something that isn’t there), and the average case is usually pretty similar to worst case (true for search).

### 2.3 Designing Algorithms/Divide and Conquer introduction

If a problem is simple you can solve a base case directly and this solves the problem. If its more complicated may be useful to use divide and conquer to solve the *recursive case*. 3 steps to this:

1. Divide the problem into sub problems
2. Conquer each subproblem recursively
3. Combine subproblem solutions to solve the main problem

Merge sort is exemplary of this. Let’s merge sort A[p:r]

1. Divide sorting A[p:r] into A[p:q] and A[q+1:r] where q is the midpoint of p and r.
2. Merge Sort A[p:q] and A[q+1:r]
3. Merge A[p:q] and A[q+1:r]

Note that step 2. recursively applies Merge Sort until the sets under consideration have just one element (and therefore are already sorted). The heavy lifting is done by the Merge command of step 3.

The steps of Merge are to compare the two sorted piles starting with their smallest elements and choose the smaller of the two elements, continue until one of the piles is done, then put the rest of the one pile on the fully sorted one.

**Analysing Divide and Conquer Algorithms**

In general algorithms with a recursive call can be analysed by considering the behaviour of the algorithm on a problem of size n in terms of the running time of it on smaller inputs. For example, with divide and conquer algorithms:

For a small enough input (n < n­0 for some n0 > 0) the runtime is constant. Otherwise, the run time is the length of divide [D(n)] + length of merge[M(n)] + solve. Solve is given by this: if the full problem takes T(n) to solve in worst case and we divide it into *a* sub problems of size n/*b*, then the solve takes a\*T(n/b). In the case of merge sort a = b = 2 but sometimes a != b.

T(n) = D(n) + a\*T(n/b) + C(n)

For merge sort, C(n) is given by the Merge subroutine which is Θ(n). D(n) is constant. So:

T(n) = 2\*T(n/2) + Θ(n).

The “Master Theorem” of chapter 4 lets us show that T(n) = Θ(n lg(n)).

## Chapter 3: Characterizing runtimes, asymptotic notation Θ,Ω, O

### 3.1 O-,-,- notation

O(n) notation is the upper bound on the runtime (or space requirements etc). It says that the runtime grows no faster than O(n). If you show an algorithm runs in O(n2) for example then you’ve shown it runs in O(n3) O(n4) etc.

Ω(n) is an lower bound, it shows the runtime grows no slower than O(n). if you show Ω(n2) then you’ve shown Ω(n) as well.

If you show Ω(n2) and O(n2) then Θ(n2) is true.

Precisely, for Ω(n) we show that for every input size n above a certain threshold, there is at least one specific input of size n for which the runtime is cn

E.g. insertion sort. We can see that the inner while loop dominates the runtime. This runs every time the outer for loop does which happens n-1 times. The while loop executes at most n-1 times for any iteration of the for loop. The inside of the while loop operates in constant time so the time is less than or equal to (n-1)(n-1) or O(n2).

For Ω we just need to show that there is configuration of the input of size n that will operate at a certain speed, then we know that the worst case is at least that speed (if not higher). We know that to move an entry k spaces takes k steps. If we consider a situation where the top third of the highest values are in the bottom third of the Array, each of these entries will take at least n/3 steps to move, and this will have to be done for n/3 entries, necessitating at least n2/9 steps. Therefore, we have an input of size n with runtime cn2 therefore Ω(n2) or “the worst case is at least n2.

Since we have Ω(n2) and O(n2) we have Θ(n2).

### 3.2 Formal definition of Ω, O, and Θ.

Ω(g(n)) is the set of functions f such that for some c and n0 0 <= cg(n) <= f(n) for all n > n0

O(g(n)) is the set of functions f such that for some c and n0 0 <= f(n) <= cg(n) for all n > n0

Θ(g(n)) is the set of functions f such that for some c1, c2, and n0 0 <= c1g(n) <= f(n) <= c2g(n) for all n > n0

Note that in a slight, justifiable perversion we say f(n) = Θ(g(n)) instead of f(n) belongs to the set Θ(g(n)).

Theorem 3.1: f(n) = Θ(g(n)) iff f(n) = Ω(g(n)) and f(n) = O(g(n)). This follows pretty directly from the definitions.

In terms of notation and language let’s consider insertion sort. We can say **the worst case run time** is O(n2), Ω(n2), and—due to Theorem 3.1—Θ(n2). We can say **the best case run time** is O(n), Ω(n), and Θ(n). In both cases it’s preferable to use Θ because this is most precise.

But we can’t say the **run time** is Θ(n2) because for some cases it’s better than this. We can say the run time is O(n2). Likewise we can’t say the run time is Θ(n) but we can say the run time is Ω(n).

On the other hand, merge sort always runs in Θ(n lg n) so we can and should say merge sort’s running time is Θ(n lg n). In general we should use the most precise and simplest notation.

We use the equality sign with asymptotic notation because we sometimes like to write equations like:

n2 + 2n + 4 = n2 + Θ(n)

We use Θ(n) in this case as a stand in for some function f(n) that belongs to Θ(n). Similarly we used Θ(n) in a recurrence formula in chapter 2:

T(n) = 2T(n/2) + Θ(n)

In some cases, asymptotic notation appears on the left-hand side of an equation, as in

2n2 + Θ(n) = Θ(n2).

Interpret such equations using the following rule: No matter how the anonymous functions are chosen on the left of the equal sign, there is a way to choose the anonymous functions on the right of the equal sign to make the equation valid. Thus, our example means that for any function f (n) ∈ Θ(n), there is some function g(n) ∈ Θ(n2) such that 2n2 + f (n) = g(n) for all n. In other words, the right-hand side of an equation provides a coarser level of detail than the left-hand side.

**Little o notation**

If a O(n) bound is not tight then we can use the o(n) notation. By example n2 = O(n2) is tight but n = O(n2) is not tight.

Formally: o(g(n)) is the set of functions f such that *for any* constant c > 0 there exists an n0 s.t. 0 <= f(n) <= cg(n) for all n > n0

The difference between O(n) and o(n) is that the o(n) holds *for any* c whereas O(n) only states that there exists a c. You can see how this works if you consider the fact that 5\*n2 != o(n2). You can choose a c (any 0 < c < 5) such that 5\*n2 is greater than cg(n).

There is an analogous lower omega notation as well.

**Relational properties**

All asymptotic notations (including little-o and little-omega) have transitivity:

f(n) = O(g(n)) and g(n) = O(h(n)) imply f(n) = O(h(n))

UPPER CASE asymptotic sets have reflexivity:

f(n) = O(f(n)) [it’s pretty clear why f(n) != o(f(n)), choose c less than 1]

Symmetry applies for Θ(n):

f(n) = Θ(g(n)) iff g(n) = Θ(f(n))

Finally there is transpose symmetry:

f(n) = O(g(n)) iff g(n) = Ω(f(n))

f(n) is *asymptotically smaller* than g(n) if f(n) = o(g(n)) and *asymptotically larger* than g(n) if f(n) = ω(g(n))

Finally, the trichotomy a < b, a > b, OR a = b does not hold for asymptotic comparisons. For example, if f(n) = n and g(n) = n1 + sin n then neither f(n) = O(g(n)) nor f(n) = Ω(g(n)) holds.

### 3.3 Math Terms

Modular arithmetic, a mod n is the remainder you get when dividing a / n.

We say a = b (mod n) if (a mod n) = (b mod n) or we can say a and b are equivalent modulo n.

Equivalently, a = b (mod n) iff n is a divisor of a – b. Consider the fact that if a = b (mod n) you can think of a and b as being separated by a “cycle” of n.

A function f(n) is *polynomially bounded* if f(n) = O(nk) for some constant k.

nb is strictly bounded above by an for real constants a > 1. That is nb = o(an)

ex = 1 + x + x2/2! + …

1 + x <= ex

If |x| <= 1 then 1 + x <= ex <= 1 + x + x2

Finally, ex = limn->inf(1 + x/n)n

ln(1 + x) = x – x2/2 + x3/3 - … (where |x| < 1)

Also where x > -1:

x / (x+1) <= ln(1+x) <= x

A function f(n) is polylogarithmically bounded if f(n) = O(lgk n) for some constant k.

lg2 n = lg(lg(n))

For all a > 0 and b

lgb n = o(na)

**Factorials**

n! is weakly bounded above by nn.

Stirling’s approximation:

n! = sqrt(2\*pi\*n) (n/e)n (1 + Θ(1/n))

This gives a tighter upper bound to n! than nn.

n! = o(nn)

n! = ω(2n)

lg(n!) = Θ(n lg n)

n! = sqrt(2\*pi\*n) (n/e)n eα(n)

where 1 / 12n + 1 < = α(n) <= 1/12n

**Functional Iteration**

f(i)(n) = { n if i = 0

f(f(i-1)(n)) if i > 0

**Iterated lg function**

lg\*n is the number of times we apply the lg function iteratively to n before getting to 1 or less. E.g. lg\*2 is 1, lg\*4 =2, lg\*16 = 3, lg\*65536= 4 etc…

**Fibonacci**: Fi = Φi – Φ’I / sqrt(5)

where Φ is the golden ratio [1 + sqrt(5)] / 2

and Φ’ is the conjugate of the golden ratio: [1 - sqrt(5)] / 2

## Chapter 4: Divide and Conquer

1. Divide the problem into smaller subproblems
2. Conquer the subproblems
3. Combine the solutions of the subproblems into the full solution.

Division happens using recurrences. These are equations that describe a function in terms of the same function applied to a smaller input. These allow us to assess the run time of a problems recursively.

A recurrence T(n) is *algorithmic* if for a sufficiently large n0

1. For all n < n0 T(n) = Θ(1)
2. For all n = n0, all paths of recursion terminate at a base case in a finite number of recursive invocations.

We can prove that a recurrence T(n) that represents a correct divide and conquer algorithm satisfies the above conditions and so is algorithmic.

For 1. note that it says 0 < c1 <= T(n) <= c2 for n < n0. We know that the algorithm outputs a solution in finite time for every legal input n. We can choose c1 to be the minimum time taken and the c2  to be the maximum time taken on all inputs of size n < n0.

2. holds if the algorithm is correct, if it didn’t hold than the algorithm wouldn’t be correct.

Divide and conquer recurrences usually don’t change if we drop floors or ceilings. 4.7 formalizes this but usually we just elide floors/ceilings when stating recurrences.

Divide a conquer algorithms don’t necessarily divide the problem into equal sizes. The matrix multiplication algorithm in section 4.2 (Strassen’s algorithm) divides the problem into seven subproblems of n/2. The recurrence is:

T(n) = 7T(n/2) + Θ(n2)

This turns out to be:

T(n) = Θ(nlg 7)

We may divide a problem into one problem of n/3 and another of 2n/3. If the division and combination take Θ(n), the recurrence would look like:

T(n) = T(n/3) + T(2n/3) + Θ(n)

This turns out to be:

T(n) = Θ(n lg n)

**Solving Recurrences**

We will learn the *substitution method* (guess recurrence and prove with induction), *recursion tree method* (represent recurrence as a tree, find the costs at each level, add them up, may need techniques for bounding summations found in A.2), the *master method* (provides bounds for recurrences of the form T(n) = aT(n/b) + f(n), *Akra-Bazzi method* (general method using calculus, widely applicable).

These can be found in 4.3, 4.4, 4.5-4.6, and 4.7 respectively.

### 4.1 Multiplying Square Matrices

Solve A \* B = C

We divide the nxn matrices into 4 n/2xn/2 matrices.

[[C11 C12]

[C21 C22]]

[[A11 A12]

[A21 A22]]

[[B11 B12]

[B21 B22]]

C11 = A11 \* B11 + A12\* B21

The other submatrices of C have similar equations for a total of 8 matrix multiplications you have to perform. So the algorithm is:

if n is 1 then then C11 = C11 + A11\*B11

Otherwise partition the matrices and run the algorithm on the 8 matrix multiplications needed to calculate the four submatrices and add those into the C matrix directly. The C matrix is initialized to 0s.

The recurrence is:

T(n) = 8T(n/2) + Θ(1)

The 8T(n/2) is clear enough. The Θ(1) comes from the fact that the matrices are partitioned in constant time (divide) and C is constructed as the recursions bottom out. There is no copying values from one matrix to another, we just keep a running tally of what each value of C is as we go through the recursions, so there is no combine step. By convention we just omit the constant runtime of the base case.

4.1-1: Generalize the algorithm so n doesn’t have to be a power of 2.

If at any step n isn’t divisible by two, divide on n-1 and then manually solve the values of row and column left out.

The number of elements that must be solved this way is bounded above by 2n 4 lg n = 8 n lg n. If the first recurrence is odd, there will be 2n -1 elements to solve for. After this division there are 4 matrices of size less than n that may require the outer row and column to be solved. This can happen at most lg n times (number of levels to the tree).

The number of steps needed to solve each element is bounded above by (3/2)\*n (n multiplications and n/2 additions). So this part of the algorithm = O(n2 lg n)

T(n) = Θ(n3) + O(n2 lg n) = Θ(n3).

OR, **pad the matrices to the next largest power of 2** and use the basic algorithm. Say the next largest power is m, this is less than 2n and more than n so:

m3 < (2n)3 = 8n3 = O(n3)

m3 > n3 = Ω(n3)

Therefore, runtime is Θ(n3)

### 4.2 Strassen’s algorithm

Uses the idea that you can turn something that looks like 2 multiplications and one addition into 2 additions and one multiplication to reduce the bushiness of the recursion tree from 8 nodes per level to 7 nodes. Going from multiplication to addition is good in the case of matrices.

Strassen’s algorithm computes the four submatrices of C in four steps:

1. If n==1 then perform the base step of basic recursive matrix multiplication (Θ(1)) or partition the matrices as was done in basic matrix multiplication (Θ(1)).
2. Create 10 n/2Xn/2 matrices S1 … S­10, these are all the sum or difference of each of the matrices created in step one. Create 7 n/2Xn/2 matrices, P1…P7 these are initialized to zero and represent 7 matrix products. This can be done in Θ(n2) time.
3. Recursively compute P1…P7 using the matrices from step 1 and S1…S10. This takes 7T(n/2) time.
4. Update C11…C22 by adding P1…P7 which takes Θ(n2) time.

This results in the recurrence:

T(n) = 7T(n/2) + Θ(n2)

Using the master method (section 4.5) we can show T(n) = Θ(nlg 7) = O(n2.81).

These are how the matrices are broken down:

S1 = B12 – B22,

S2 = A11 + A12,

S3 = A21 + A22,

S4 = B21 – B11,

S5 = A11 + A22,

S6 = B11 + B22,

S7 = A12 – A22,

S8 = B21 + B22,

S9 = A11 – A21,

S10 = B11 + B12.

|  |  |  |  |
| --- | --- | --- | --- |
| *P*1 | = | *A*11 · *S*1 | (= *A*11 · *B*12 – *A*11 · *B*22), |
| *P*2 | = | *S*2 · *B*22 | (= *A*11 · *B*22 + *A*12 · *B*22), |
| *P*3 | = | *S*3 · *B*11 | (= *A*21 · *B*11 + *A*22 · *B*11), |
| *P*4 | = | *A*22 · *S*4 | (= *A*22 · *B*21 – *A*22 · *B*11), |
| *P*5 | = | *S*5 · *S*6 | (= *A*11 · *B*11 + *A*11 · *B*22 + *A*22 · *B*11 + *A*22 · *B*22), |
| *P*6 | = | *S*7 · *S*8 | (= *A*12 · *B*21 + *A*12 · *B*22 – *A*22 · *B*21 – *A*22 · *B*22), |
| *P*7 | = | *S*9 · *S*10 | (= *A*11 · *B*11 + *A*11 · *B*12 – *A*21 · *B*11 – *A*21 · *B*12). |
|  |  |  |  |

Note that the expansion of the S matrices on the rightmost side of the P equations are for convenience only, they are not computed.

The four C submatrices are computed from the Ps as follows:

C11 = C11 + P5 + P4 – P2 + P6.

C12 = C12 + P1 + P2

C21 = C21 + P3 + P4

C22 = C22 + P5 + P1 – P3 – P7

By expanding the values of P to their constituent parts and adding them out, you return the multiplication and addition pattern of the basic calculation of the C submatrices e.g.

C1,1 = A1,1 B­1,1 + A1,2 B2,1

**4.2-7** If we can square a matrix in Θ(n2) time, show we can multiply two different nXn matrices in Θ(n2) time.

From <https://inst.eecs.berkeley.edu/~cs170/fa18/assets/dis/dis01-sol.pdf>

Make a 2nX2n matrix A with A12 = X A21 = Y and the rest zero. Squaring A will give us XY at A211. This is done in Θ((2n)2) time which is Θ(n2).

This is an example of reduction. It reduces the complexity of multiplying two different matrices to the complexity of squaring one matrix.

### 4.3 Substitution method for solving recurrences

Guess what the form of the function is and prove by induction. Better to prove upper bound O and lower bound Ω separately.

Consider T(n) = 2T(floor(n/2)) + Θ(n)

This looks similar to the recurrence for merge sort which has Θ(n lg n) so lets guess O(n lg n) for our recurrence here.

The inductive hypothesis should be T(n) <= c(n lg n) for all n > n0 and some constant c. We need the constants later so don’t make the hypothese T(n) = O(n lg n)

Assume the hypothesis holds for all numbers at least as big as n0 up and less than n.

Say n = 2n0 then by assumption we know that T(floor(n/2)) <= c(floor(n/2) lg floor(n/2))

Our recurrence is T(n) = 2T(floor(n/2)) + Θ(n)

So T(n) <= 2\*c(floor(n/2) lg floor(n/2)) + Θ(n)

<= 2\*c(n/2 lg (n/2)) + Θ(n)

= cn (lg n – lg 2) + Θ(n)

= cn\*lg n – cn + Θ(n)

<= c\*n\*lg n (where c is sufficiently large s.t. cn > Θ(n))

Most proofs stop here and omit proving the base cases.

In this case the base cases are those where n0 <= n < 2n0

Since lg 1 = 0, choose n0 = 2. By convention the base cases of the recurrence were omitted because they run in constant time. So T(2) and T(3) run in constant time.

We choose our constant c to be max(T(2), T(3)). Meaning T(2) <= c < (2 lg 2)\*c AND T(3) <= c < (3 lg 3)\*c

The very simple statements c < c \* 2 lg 2 and c < c \* 3 lg 3 prove the base cases.

Making a good guess. If the recurrence looks similar to one you’ve seen try that. Recursion trees can help generate guesses. Can start with a loose lower and upper bound and work both upwards and downwards until they converge.

If your proof isn’t working, it might help to subtract a lower term from the hypothesis. E.g. it might be good to subtract n from the hypothesis when proving O(n2). When it comes time to sub in the assumption this will pay dividends.

### 4.4 Recurrence Trees

If you’re very precise in drawing and adding up the trees they can be a proof of the solution to the recurrence. However, it might be easier to be a bit sloppy with the trees, just generate a guess from them, and then prove with substitution.

On the page below is the recursion tree for T(n) = 3T(n/4) + Θ(n2):

![Diagram

Description automatically generated]()

Adding up all the costs on the right side and finessing the summation of (3/16)i using the fact that the geometric series goes to 1 / (1 – a) for 0 < a < 1 as I 🡪 inf, you can make a pretty good guess that the upper bound is n2, this is the cost generated at the first step of the recursion so this means that n2 is also probably a lower bound. Can prove this with substitution.

Consider T(n) = T(n/3) + T(2n/3) + Θ(n)

This is irregular in that the different paths to the leaves will have different lengths. E.g. the path that always follows T(n/3) is shorter than any other. Also there is the n0 for the Θ(n) and the n0 for the recurrence. We can choose the n0  we work with to be the larger of the two. The diagram is below.

![Diagram

Description automatically generated]()

Following the right path (always taking T(2n/3)), which is the longest path down, we can find the length is floor(log3/2(n/n0)) + 1, meaning leaves are on at most that level. To show this consider:

(2/3)h n <= n0 <= (2/3)h-1 n

(2/3)h <= n0/n

h <= log2/3(n0/n) < floor(log2/3(n0/n)) + 1

So the height of the tree is Θ(lg n). Each level of the tree costs cn so O(n lg n) looks like a reasonable guess. Considering the base case runtime we have to figure out how many leaves there are. If we choose the number of nodes assuming binary splitting down to the level given by the 2n/3 splits we find there are 2h leaves at the bottom, this is 2^(log3/2n) + 1 = 2n^(log3/22) leaves which is bounded above by n1.71. O(n1.71) is too loose a bound. The branches of the tree have O(n lg n), if we use substitution we can show that this is a good upper bound for the recurrence, however, this is dominated by O(n1.71). So this is a good example of why trying to prove an initial guess from the tree using substitution could be useful, it would save us the time of working through the tree more thoroughly to get the leaves bound down to n, which it should be.

But let’s try and work out the number of leaves properly. It might be the case in other situations that the number of leaves is large enough to dominate the recurrence. We will do this by writing a recurrence for the number of leaves in the tree L(n):

If n < n0L(n) = 1

If n >= n0 L(n) = L(n/3) + L(2n/3)

This is similar to the time recurrence but without the Θ(n) term which makes it easier. Let’s solve it with substitution assuming L(n) < dn (Θ(n)).

For the inductive step: L(n) = L(n/3) +L(2n/3) <= dn/3 + d2n/3 = dn

For the base case, we’re looking at 0 < n <= n0:

L(n) = 1, n is 1🡪n0 so L(n) <= dn if d >=1

So this shows L(n) = O(n). Therefore the cost of the leaves is O(n). Since the internal nodes are O(n lg n) T(n) = O(n lg n) + O(n) = O(n lg n).

### 4.5 Master Theorem

Solves recurrences of the form:

T(n) = aT(n/b) + f(n)

f(n) is the *driving function,* a recurrence of form aT(n/b) is a *master recurrence*.

Three cases:

1. Where there exists an ϵ such that f(n) = O(n^(logb(a)- ϵ)) then the solution to the recurrence is:

E.g. f(n) = n2 and 8(T(n/2)) so compare n2 to n^(log2(8)) = n3. If f(n) is nx compare logb(a) to x, if x < logb(a) then this case applies and T(n) = Θ(nlogb(a))

1. There exists a k >=0 such that, f(n) = Θ(n^logb(a) \* lgk(n)), then T(n) = Θ(nlogb(a) lgk+1(n))  
   E.g. the merge sort algorithm T(n) = 2T(n/2) + Θ(n).
2. The second line is called the *regularity condition*.

is called the *watershed function* and is compared to the *driving function* f(n).

Case 1 occurs when the watershed function grows polynomially faster than the driving function. That is it grows faster by a factor of . If we consider the recurrence tree, this occurs when the cost of leaves dominates the total cost, when the cost of each level increases geometrically.

Case 2 occurs when the driving function is greater than the watershed function by . In practice, we usually see this when the driving function equals , and k = 0. Like with merge sort.

Case 3 has the regularity condition which holds for most polynomially bounded functions. Looking at the recurrence tree, the cost per level drops geometrically and the total cost is dominated by the root.

Let’s look at one case involving the regularity condition.

Case 3 applies because so

The regularity condition states for some c < 1.

### 4.6 Proof of Master Theorem

See the text book…

### 4.7 Akra-Bazzi recurrence and method

Akra-Bazzi recurrences look like this:

This is a more coarse master recurrence. The divisions of the initial problem can be different sizes.

The Akra-Bazzi method solves the recurrences above where f(n) satisfies the polynomial growth condition.

*Polynomial Growth Condition*

f(n) satisfies this condition if

where

Polynomials satisfy this of course but exponentials (e.g. 2n) and “weird” functions don’t.

Akra Bazzi method shows the following solution:

Where p is the solution to:

P can be tough to calculate but sometimes the integral is of a form that we don’t actually have to calculate p to get the bound. For example, if f(n) is a polynomial, it will probably be possible and easy to apply the calculus rule and then find the bound after subtracting out the interval.

**Exercise 4-3 Change of Variables to solve recurrences**

Can solve: by substituting m = lg(n) and

Note that S(m) = T(n) when you change m back to lg(n). The substitution works by changing the formula to: which can be solved by the master method and gives:

After subbing lg n in for m you get.

## Chapter 5 Probabilistic Analysis and Random Algorithms

### Chapter 5.1 The Hiring Problem

We’re trying to hire a new person. We interview n people. If person i is more qualified than person i-1 then we fire i-1 and hire i.

Best = 0

for i = 1 to n:

interview i

if i is better than i-1:

best = i

hire i

There is a cost ci for interviewing, the total cost will always be cin.

There is a cost for hiring/firing, ch, this depends on the order of candidates.

E.g. if candidates come in increasing order of qualification, the total hiring cost will be chn.

In general, we’ll say hiring cost is chm where m is the number of people hired.

We want to get an expected value for chm and will use probability to do so.

Note that for this algorithm we are assessing a different kind of cost, not runtime but the money paid for the hiring process. The principles of assessing this cost are the same though.

Using probability to assess cost will result in *average-case­ runtime/cost*.

We have to make assumptions about the distribution of the problem. In the hiring problem case we can assume that there is a total order of candidates (all candidates can be given a rank that orders them from worst to best), there are n! permutations of the orders candidates can come to us, and that each permutation is equally likely to occur (*uniform distribution*).

There is a small difference between having inputs that naturally follow a certain distribution and randomizing inputs ourselves using a RANDOM function. The latter we call a randomized algorithm and the runtime is *expected runtime.* The former’s runtime we call *average-case runtime*.

### 5.2 Indicator Random Variables

Let S be a sample space with A1…k­ events in it. Each event has P(Ai) of occurring. Our indicator random variable .

We have where E[X] is the expected value of X. This is Lemma 5.1 in the book.

If we want the Expected number of heads in n coin flips we can define:

Where where the ith flip is heads and 0 otherwise.

which because of the linearity of Expected values

If we consider the hiring problem with indicator random variables we simplify the analysis of how many people we’re expected to hire. Say X is how many people we hire.

Where Xi is 1 if the ith person is hired and 0 otherwise.

For person i, there is a 1/i chance that they will be hired since the candidates are randomized so all i of them have an equal chance of being the best out of i.

so

So the expected value of the cost of hiring will be O(CH \* ln(n))

This is a big improvement over the worst case hiring cost and since we control the input by randomizing it, we can accept that the expected value/average cost is a good bound to work with.

**Exercise 5.2-1**

1/n chance you hire once because it’s the same as the chance of the best person being the first candidate.

1/n! chance you hire n times. Every interview you need to get just the right person, the worst is first, second worst is second and so on. The chances of this are 1/n, then 1/n-1, and so on. By nm rule this is 1/n!

### 5.3 Randomizing Algorithms

As discussed above this is when we shuffle the inputs to make them random regardless of how they come to us. The application of the Expected value function on the Indicator random variable to find the runtime applies the same as if the algorithm was not randomized and the inputs just followed a uniform distribution. However, we now refer to it as the *expected runtime* as opposed to the average runtime.

To make the input random we can use the Randomly-Permute algorithm that creates a random permutation of an array. The conceit is that every element will have a 1/n chance to end up in any of slots.

***Randomly-Permute(A,n)***

for i 1 to n:

Swap A[i] with any element A[i:n]

A[1:i-i] is invariant. For i=1 there is 1/n chance it will “swap” with itself, so the initial A[1] has a 1/n chance to end up at A[1]. For all other elements, there is a (n-1)/n chance a there will be a swap and a 1/n-1 chance that it will be any one of the n-1 elements.

By the mn rule: Probability that any element A[2:n] will end up at A[1]. There’s some intuition, here’s the proof.

**Lemma 5.4** Randomly-Permute creates a permutation <x1…xn> with likelihood 1/n!, that is it computes a uniform random permutation.

The invariant of this loop is: at the beginning of loop i, the likelihood that A[1:i-1] has permutation <x1…xi-1> is (n – (i-1))! / n! =

This invariance we are trying to prove comes from the number of permutations of n-choose-k elements. Namely there are n! / (n-k)! ways to choose k elements from n.

*Initialization***:** at the first iteration of the loop, the subarray under consideration has zero elements. Each permutation of 0 elements has Pr(1) and this is true of A[1:0].

*Maintenance*: At the start of loop i, A[1:i-1] are in permutation <x1…xi-1> with probability (n-i+1)! / n! (by assumption). We’ll call A[1:i-1] being in permutation <x1…xi-1> E­1. Line 2 will swap A[i] with any element from A[i:n] randomly. We will call the probability of xi ending up at position A[i] E2.

We are interested in the joint probability =

so

Therefore at the start of loop n+1, we have permuation <x1…xi> with probability (n – (i+1) +1)!/n!

This completes maintenance.

*Termination*

We complete the nth loop and go to the beginning of the n+1 loop. At this point A[1:n] has permutation <x1…xn> with probability (n-(n+1)-1)!/n! = 1/n!

### 5.4.2 Further applications of indicator random variables – tossing balls into bins

Imagine you randomly toss balls in b bins. The chance of a ball ending up in a given bin is 1/b. If we imagine the process as a Bernoulli trial (one bin is success, the others failure) the chance of success is 1/b. This is a useful structure when studying hashing (chapter 11).

How many balls fall into a given bin follows the binomial distribution b(k;n, 1/b), k balls in given bin after

For n tosses expected number of balls in a given bin is n/b.

Expected tosses until a given bin has a ball in it is b.

How many tosses will it take until every bin has a ball in it (*coupon collector problem*)?

Thinking about the probability that a toss will land in an empty bin. We want E[n] where n is the number of tosses it takes to get one in every bin.

Can divide n into where ni is the number of tosses it takes to go from i-1 bins having balls to i bins having balls.

This is the geometric distribution so E[ni] = 1/p:

Then:

[Solution to harmonic series]

This is also known as the coupon collector problem because it describes how many coupons you need to collect in order to get k distinct coupons.

# Part II Sorting and Ordering Statistics

We’ll look at various sorting algorithms in this section. Sorting is important because:

Many problems require sorting directly

Many programs require sorting as a subroutine

The sorting algorithms that have been developed represent a wide array of algorithm design techniques that are useful in other contexts

We can prove a nontrivial lower bound for sorting. This can be used to make lower bounds for other problems.

Sorting presents interesting engineering issues that are best addressed at the algorithmic level.

We’ll look at heap sort in chapter 6 , quick sort in chapter 7 ( In chapter 8 we’ll see the worst case lower bound for comparison sorts (insertion, merge, heap, and quick) is n lg n and then consider sorting techniques that could be the lower bound if we assume information about the array.

So in chapter 8 we’ll see counting sort, radix sort, and bucket sort. The first two are deterministic and run in and respectively. The last is assumes a probability distribution of the input and has average runtime of .

The ith *order statistic* is the ith smallest element in the array. In chapter 9 we’ll see how to find this in O(n) time.

## Chapter 6 Heapsort

Heapsort runs in n lg n *and* it sorts in place, which means it uses only constant memory (merge sort creates arrays as it divides so it uses n lg n memory).

### 6.1 Heaps

We use the (binary) heap data structure in heap sort. A binary heap is an *almost complete binary tree*. Almost complete because only the bottom level is missing nodes.

A heap is an array with attribute *A.heapsize* where A[1:A.heapsize] are elements of the heap. Heaps have the following methods

For any i from 1:A.heapsize:

Parent(i):

return

Left(i):

return 2i

Right(i):

return 2i +1

For *max heaps*, for every node i other than the root:

For *min heaps* the opposite holds.

Heapsort uses max heaps whereas priority queues use min heaps.

The *height* of a node in a heap is the number of edges on the longest simple path from node to leaf. The height of the heap is the height of the root node. This is lg n.

The following are important procedures to heaps:

1. Max-Heapify, runs in lg n, key to maintaining the *max heap property*
2. Build-Max-Heap, runs in O(n) and produces a max heap from an unordered array
3. HeapSort, runs in O(n lg n) and sorts an array in place.
4. Max-Heap-Insert, Max-Heap-Extract-Max, Max-Heap-Increase-Key, Max-Heap-Maximum run in O(lg n) and are used to make priority queues.

6.1-1) Min size of heap of height h is 2h, max size is 2h+1 -1

6.1-2) n = 2m – 1 + k has m-1 full layers and k leaves. This is height m for 1<=k<=2m, so m =

### Chapter 6.2 Max-Heapify

Max-Heapify takes a sub-array/sub-tree of a heap and floats the root down into its correct position.

Max-Heapify(A, i):

L = Left(i)

R = Right(i)

If L<=A.heap-size and A[i]<A[L]:

Largest = L

Else:

Largest = i

If R<=A.heap-size and A[Largest]<A[R]:

Largest = R

If Largest!=i:

Swap A[i] and A[Largest]

Max-Heapify(A, Largest)

The recursion for MaxHeapify is:

Because the largest a subtree can be is two thirds of the whole tree. We can see this if we consider a tree where the last level is half full with all entries of the last level on one side. If the tree is height h, the larger subtree will have half of the completely full tree = (2h – 1) / 2 and all of the 2h / 2 leaves in the bottom layer. This is 2h – ½. The full tree has 2h – 1 + 2h-1 = 3\*2h-1 – 1. The ratio goes to 2/3 as h goes to infinity.

By case 2 of the master method

### 6.3 Building a Heap

BuildMaxHeap(A, n):

A.heapsize = n

for

MaxHeapify(A, i)

We start at n/2 because anything greater than this is a leaf and so is trivially already a max heap.

The loop invariant is: at the start of each loop i: i+1, i+2, … n, are the roots of max-heaps.

Initialization: at i = n/2 the nodes at i+1 etc are leaves so are trivially the roots of max-heaps

Maintenance: MaxHeapify is applied to the subtree with root at node i. By assumption all subtrees starting at i+1 are MaxHeaps. MaxHeapify will place the largest of node I, Left(i), and Right(i) at the root of the subtree. Max-Heapify will float the value that starts at node i to its proper place in the sub-subtree and so will keep the MaxHeap property of the subtrees. At the beginning of the next loop (i-1), since the largest value will be in position i, subtree with root i will be a MaxHeap.

Termination: Terminates at i = 0 so subtrees with roots at 1...n are maxheaps. Especially important is the fact that the tree with root at 1 is a MaxHeap.

Worst case runtime can be found by adding up the number of nodes at each level and multiplying by the number of times MaxHeapify will be applied to each node.

We know there are at most nodes at each level h (exercise 6.3-4). MaxHeapify will be applied at most lg n = h times to a node at height h. Therefore the time is less than:

Based on what we know of ceilings.

### 6.4 Heapsort algorithm

Heapsort(A, n):

BuildMaxHeap(A, n)

For i = n down to 2:

Exchange A[1] and A[n]

A.heapsize = A.heapsize -1

MaxHeapify(A, 1)

First you construct the max heap in O(n) time. Then you store the max value in the last place of the array, you protect that part of the array by decreasing the heapsize by one, then you run Max-Heapify to float the exchanged value back into its proper place in the heap which is one element smaller.

In terms of maintenance of the MaxHeap, note that after you exchange the top element, call it s, for the bottom and then remove s from the heap, the subtrees with root i>1 are max heaps. When you float A[1] back to its proper place using MaxHeapify, the subtrees remain maxHeaps and you reestablish MaxHeapedness for the tree with root at 1 (based on established properties of MaxHeapify).

In terms of maintenance of n down to i+1 being in the proper position at the start of loop i, note that the top element of the max heap is the Max Element. This is added to position i of the array and so at the start of loop i-1, i to n is in the proper position. At the start of the last loop, 2 to n are in proper position so 1 is as well.

The for loop will execute n-1 times and MaxHeapify will run at O(lg n) time for each loop. Therefore the runtime is O(n) [construction of the max heap] + O(n lg n) = O(n lg n)

### 6.5 Priority Queues

HeapSort is usually beaten by quicksort in practice but heaps are useful structures in their own right. One reason is that they can be used to make priority queues.

A priority queue is a structure of a set of elements S that are key:value pairs. A Max-Priority Queue supports the following operations:

Insert(S, x, k), inserts element x with key k into S. Equivalent to

Maximum(S) returns element with largest Key.

ExtractMax(S), removes and returns elements of S with largest key

Increase-Key(S, x, k) increases element x’s key to k. Assumes k is at least as large as x’s original key.

Min Priority queues have Minimum(S), ExtractMin, and Decrease-Key functions.

Max-Priority queues can be used as task schedulers on a shared computer, min-priority queues can be event simulators where the key represents the time of the event and simulating one event can lead to the creation of new events that have to be inserted into the queue.

A max heap has the key (priority) values arranged in max heap order. We need to be able to link these values to the objects of the application that they represent. There are different ways of doing this, handles, hash tables etc. Handles seem like the suggested method because of speed and they allow for greater abstraction in the code, but I’m not sure what handles are exactly…

Here is pseudocode for implementing the max priority queue operations with a max heap.

***Max-Heap-Maximum(A):***

If A.heapsize < 1

Error underflow

Return A[1]

***Max-Heap-Extract-Max(A):***

max = Max-Heap-Maximum(A)

A[1] = A[A.heapsize]

A.heapsize = A.heapsize-1

MaxHeapify(A, 1)

Return max

***Max-Heap-Increase-Key(A, x, k***):

If x.key > k:

Error k is less than existing key

x.key = k

i = index of the array at which object x occurs (this depends on how you map between priority queue and application objects).

While i > 1 and A[Parent(i)].key < A[i].key:

Exchange A[Parent(i)] and A[i] and update mapping from index to object

i = Parent(i)

***Max-Heap-Insert(A, x, n):***

If A.heapsize == n:

error overflow

A.heapsize = A.heapsize + 1

k = x.key

x.key = -inf

A[A.heapsize] = x

Map object x to index heapsize in the array

Max-Heap-Increase-Key(A, x, k)

Max-Heap-Maximum runs in O(1) the rest run in O(lg n). It’s pretty clear that worst case, the loops will only run for the height of the tree = lg n.

**Exercise 6.5-10**

Max-Heap-Delete(A, x):

i = index for object x

if A[i].key > A[A.heaplength].key:

A[i] = A[A.heaplength]

MaxHeapify(A, i)

Else:

Max-Heap-Increase-Key(A, A[A.heaplength], x.key)

A.heaplength = A.heaplength - 1

Max-Heap-Delete calls either MaxHeapify or Max-Heap-Increase-Key, both are O(lg n) so Max-Heap-Delete is O(lg n).

## Chapter 7 QuickSort

We’ll explore the QuickSort algorithm. It has slow worst case but a good average case O(n lg n), like heap sort, it sorts in place so it uses linear space. Also the constants hidden in the asymptotic notation are small so in practice it’s often faster than heap sort.

### 7.1 Description of QuickSort

It’s a divide and conquer algorithm like merge sort.

Divide: for a subarray A[p:r] choose an index q such that all values in A[p:q-1] are less than A[q] and all values in A[q+1:r] are greater than A[q].

Conquer: Keep applying QuickSort to subarrays created in previous step until they’re sorted (this requires more explanation).

Combine: Nothing, A[p:q-1] and A[q+1:r] are sorted, since the values in the former are all less than A[q] and vice versa for the latter, the subarray is already sorted.

The algorithm is:

***QuickSort(A, p, r):***

If p < r:

q = Partition(A, p, r)

QuickSort(A, p, q-1)

Quicksort(A, q+1, r)

The Partition subroutine is as follows:

***Partition(A, p ,r):***

x = A[r]

i = p-1

for j = p to r-1:

if A[j] <= A[r]

i += 1

Swap A[i] with A[j]

Swap A[r] with A[i+1]

q = i+1

Return q

Partition creates a subarray from p to i that is less than A[r] and another from A[i+1] to A[r-1] that is greater than A[r], it then swaps A[r] for A[i+1] effectively making i+1=q (the index that partitions the array).

Specifically the invariants at the start of loop j are as follows:

1. Values in A[p:i] are less than or equal to A[r] [x in the pseudocode]
2. Values in A[i+1:j-1] are greater than A[r]
3. If k == r then A[k] = x

Initialization: for 1 and 2 you have null sets so it holds trivially. For 3, A[k] = A[r] = x.

Maintenance: At loop j, j >= i,

1. A[p:i] is less than x, if A[j] is less than x, i = i+1 and A[j] is swapped to that spot, at loop j+1, A[p:i] are less than x (with A[p:i] being one element larger than before. If A[j] is greater than x, i = i and A[p:i] is the same at the start of loop j+1 so the elements are the same and less than x.
2. A[i+1:j-1] are greater than x.   
     
   If A[j] is less than x, i = i+1 and A[j] is swapped there. At the start of loop j+1 A[(i+1) + 1:j] are greater than x because A[i+2:j-1] is the same as A[i+2:j-1] from the previous loop, which are greater than x by assumption and A[j] is the just A[i+1] from the previous loop, which is greater than x by assumption  
     
   If A[j] > x, then nothing happens, at j+1 A[i+1:j-1] is greater than x by assumption and we showed A[j] is greater than x.
3. Trivial.

Termination: A[p:i] is less than or equal to x, A[i+1:r-1] is greater than A[r]. So if we swap A[r] and A[i+1], A[i+2:r] will be greater than A[i+1] and A[p:i] will be less than A[i+1] so q can be i+1.

### 7.2 Intuition about the runtime of QuickSort

Worst case happens when the split makes a subarray of length n-1 and another of length 0 at every step. By the arithmetic series this gives a worst case runtime .

Best Case occurs when the split is even, then we have the recurrence which by case 2 of the master theorem is

If the split is seemingly uneven, like say T(n/10) and T(9n/10) then the recursion will bottom out at level in fastest case and in the slowest case. Each level of the recursion tree has cost at most and after level , the cost will be less than n. So the total cost is which is so no matter how unbalanced the split is, the total cost is bound above by n lg n.

Even if you alternate between a bad split (say 9/10 and 1/10) and the worst case where the split is n-1 and 0, the cost is still bounded by n lg n. In Exercise 7.2-6 we show that if all permutations of the input are equally likely and all values are distinct, then the likelihood of 9-1 split or better is 80%. So if you have distinct elements and all permutations are equally likely, then the expected runtime is probably n lg n.

### 7.3 Randomized QuickSort

We can get splits of expected balance if we randomize the inputs. Unlike the randomized hire assistant algorithm we don’t have to completely randomize the order of inputs though. We can use a RandomPartition subroutine instead, that simply swaps A[r] with a random element from A[p:r]. This is enough to give pretty random balances of partitions.

### 7.4 Analysis of QuickSort

#### 7.4.1 Worst Case Runtime

We’ve developed a pretty good guess that worst case runtime is so we should prove this with substitution. So prove . In the worst case:

By assumption:

where we choose large c to dominate

By example we’ve shown that the lower bound is n2 as well so we have

#### 7.4.2 Expected Runtime

We have some intuition that the expected runtime for the randomized quicksort is n lg n. By getting a better understanding of RandomPartition, we can prove O(n lg n). We also have that best case runtime is , which proves expected runtime is (since we have a best case lower bound being n lg n.

*Lemma 7.1, total time for QuickSort is O(n +X) where X is the number of comparisons made in the for loop in Partition.*

On every call of Quicksort the last element is removed to be the pivot. This can happen at most n times so Partition can only be called n times. Furthermore, there are two calls to Quicksort in the recursive step so at most there are 2n calls to Quicksort itself. Outside the for loop, Partition takes O(1), inside it takes as many loops as there are comparisons of elements. Therefore, quicksort takes O(n) for the n calls to the constant part of Partition plus O(X) for the total number of comparisons. In total O(n + X)

So the task becomes to compute the expected value of the number of comparisons performed throughout QuickSort E[X]. To make this easier we will index values by their placement in the sorted array so z1…zn has the property , the inequality is strict because all elements are distinct. zi, …, zj is denoted by zij.

*Lemma 7.2,*

During quicksort two elements zi and zj ­(with j > i) are compared iff one of the is chosen as a pivot before any element in the set zij. No two elements are compared more than once.

If an element from zij is chosen it may be zi, zj or some x zi < x < zj. Consider this third case. x will become the pivot, zi and zj will fall on different sides of it and so will never be compared. If either zi or zj are chosen first, they will be compared to all zij and so will be compared with each other. Elements are only compared with pivots and pivots are removed from future recursions so no two elements can be compared twice.

*Lemma 7.3*

Consider RandomizedQuicksort being done on . The probability two elements zi  and zj  will be compared is:

Randomized quicksort divides the elements z1n into sets with each choice of pivot. zij will remain together until an element from z­­ij is chosen. Since RandomPartition chooses the pivot from zij based on a uniform distribution, any element of zij is equally likely as any other to be chosen. Specifically, the likelihood of one being chosen is .

So, using Lemma 7.2 we have:

***Theorem 7.4***

The Expected Runtime of QuickSort is n lg n.

We will use the indicator random variable Xij which is 1 if i and j are compared and 0 otherwise. Then the total number of comparisons X is given by:

From Lemma 7.1

So expected runtime is

**Exercise 7-5**

Can replace the second recursion call with a while loop like this:

TRE-QuickSort(A, p, r):

While p < r:

q = Partition(A, p, r)

TRE-QuickSort(A, p, q)

P = q+1

This is called tail recursion elimination. Good compilers will do this automatically, not sure how…

Pretty clear how this works, does the first half of every partition first, then covers the second half of any recursion by making p = q+1, the partition will run on A, q+1, r like it would in the second recursion of a normal Quicksort call.

## Chapter 8 Sorting in Linear Time

### 8.1 Lower Bound for Comparison Sorts

Any sorting algorithm that relies exclusively on comparing values to determine order has a worst case runtime bounded below by n lg n. So merge sort and heapsort are asymptotically optimal as comparison sorts.

*Theorem 8.1*: Lower bound for worst case of any comparison sort is

We can represent any comparison sort as a decision tree where the nodes represent comparisons between elements of an array and the leaves are all permutations of the elements. A sorting algorithm simply follows a simple path of comparisons to the correct permutation leaf. Therefore, the worst case runtime is the height of the tree (the longest simple path).

If the number of reachable leaves is l and the height of the tree is h. n! is the number of permutations of the elements, each permutation is at least one leaf so n! <= l. A binary tree of height h has at most 2h leaves so . Taking the lg of both sides:

(By an inequality in chapter 3/a proof using the Stirling Approximation)

*Corollary 8.2*: Heap and merge sort are asymptotically optimal.

Proof: They are both which matches the lower bound .

### 8.2 Counting Sort

We use a temp array C who’s relevant indexes are the values of in Array A[1:n] that we’re trying to sort. C is C[0:k] meaning we assume there is no value in A greater than k.

Count-Sort(A, n, k):

Let B be an array 1…n and C be an array 0 to k

For i = 0 to k:

C[i] = 0

For j = 1 to n:

C[A[j]] = C[A[j]] +1 //C[i] is number of values equal to i

For I = 1 to k:

C[i] = C[i] + C[i-1] //C[i] now has number of values less than or equal to i.

For j = n down to 1:

B[C[A[j]]] = A[j]

C[A[j]] = C[A[j]] – 1 //Handles duplicate values

Return B

Counting sort is *stable* meaning that equal values in the input appear in the same order in the output. This is important if there is satellite data attached to the elements being sorted. It’s also important because Counting sort is a subroutine of radix sort and radix sort only works if countin sort is stable.

### 8.3 Radix Sort

Sorts numbers by sorting from least significant digit to most using a stable sort algorithm. E.g. if sorting [153,125,311] first you get [311,153,125], then [311,125,153] then [125,153,391]. Say d is the number of digits that n numbers have and k is the max value of those digits. Digit d is the most significant and digit 1 is the least significant. A is an array of the n numbers.

RadixSort(A, n,d):

For i = 1 to d:

Stable sort A according to digit i

CountingSort is an obvious choice for the stable sort, especially when we’re dealing with a digits who’s values go from 0 to 9.

Radix sort correctly sorts in time if we use counting sort in the for loop. Furthermore, if we have data stored in b bits, we can choose an r <= b s.t. run time is . The idea is we divide the b-bit data into d r-sized pieces. If b = O(lg n) and we choose r = lg n approximately, then the running time is constant.

Although this appears better than our comparison sorts and QuickSort’s expected runtime, the constants make lengths of the actual runtimes depend on the machine and input data. Furthermore, counting sort does not sort in place so if space is a concern, an inplace algorithm might be better.

### 8.4 Bucket Sort

This algorithm assumes a uniform distribution of the elements of the set across the interval [0,1).

BucketSort(A, n):

Let B be an empty array of B[0:n-1]

For i = 0 to n-1:

Let B[i] = an empty list

For i = 1 to n:

Insert A[i] into list

For i = 0 to n-1:

Sort B[i] with insertion sort

Concatenate lists B[0] to B[n-1]

Return concatenated lists

It’s pretty clear that this is correct. It’s harder to see that the expected runtime is linear.

All the outer for loops are so we should focus on the time it takes to run the n loops of insertion sort. This depends on the length of the bucket and we know insertion sort runs in quadratic time so:

Where ni is the length of bucket i.

We can find the expected runtime just be finding the expected length of the buckets.

The ni’s all have the same expected value because we assume the input is uniformly distributed.

Ni can be seen as a Bernoulli trial with n attempts and 1/n chance of success (p = 1/n). We know that the expected value of a Bernoulli trial is and

So:

## Chapter 9, Medians and Order Statistics

The ith order statistic is the ith smallest element in a set of n elements. For i=1 you have the minimum, for i=n you have the maximum. The median is (n+1)/2 if n is odd. If n is even the lower median is n/2 and the upper median is n/2+1. The term median will generally refer to the lower median.

These chapters will be concerned with finding the ith order statistic, meaning, given a set of n distinct elements and an integer , the algorithm will find the number such that i-1 numbers are smaller than it. We can do this in n lg n time if we sort with merge or heapsort and then return the ith element. However, we will explore other algorithms that do it in constant time.

### 9.1 Minimum and Maximum

Can find either minimum or maximum by choosing first element as temp min/max, then going through the array and updating the min/max when needed. The n-1 time this takes is optimal for min/max finding algorithms.

We can find both min and max in 3n/2 time by comparing pairs of elements and then measuring the higher one against the temp max and the lower one against temp min. This means we do 3 comparisons for two elements as opposed to four if we did it the naïve way (doubling the work of the single min/max finder).

### 9.2 Selection in Expected Linear Time

We can select the ith order statistic in Expected Linear Time. This relies on a divide and conquer algorithm similar to Quicksort. We use RandomPartition to split the input but we only have to consider the part that contains the ith order statistic.

For intuition, imagine always getting the pretty good split of half and half. The recursion looks like by the master theorem case 3, the driving function dominates the recurrence so . Remember for the master theorem case three to apply, the driving function has to be bounded below by and satisfy the regularity condition for some c for all sufficiently large n. This is met in this case. For further intuition about why the splits are expected to be about as good as this, see 7.2 and 7.4.

**RandomizedSelect**(**A, p, r, i):**

If p==r:

Return A[p] //Since 1<=i<=p-r+1 so if p-r=0 then i = 1

q = RandomPartition(A,p,r)

k = q – p +1

if i == k:

return A[q] //the pivot value is the ith order statistic

elif i < k:

return RandomizedSelect(A,p,q-1,i)

else:

return RandomizedSelect(A, q+1,r, i – k)

Proof that the expected runtime of RandomizedSelect is relies on the lemma (Lemma 9.1) that there is at least a 50% chance that a single loop of partitioning and recurrence removes at least a quarter of the elements. We call a recurrence that does this *helpful*.

We can view A(j) as the jth partition, we know that every partition removes at least one element so

We will divide our partitions into generations such that each is a helpful partition. Generation i consists of . We define a random variable for i = 0, …, m-1. This is the number of recurrences in generation i. Because of lemma 9.1 and the formula that gives the expected value of a geometric distribution, .

The runtime of RandomizedSelect is driven by the number of comparisons it makes in the Partition phase. At each recurrence it will make less comparisons than the size of the partition. So it is bound above by:

Where n0 is the size of the nought partition (aka the size of the original set of elements).

Taking the expected value of this:

This means the expected runtime is bounded above by the size of the input. We also know that the very first call of partition has n comparisons so the lower bound is n. So expected runtime is .

### 9.3 Worst Case Linear Selection

This is a somewhat cumbersome algorithm that is mostly of academic interest apparently. The basic idea is that you choose a pivot that is guaranteed to get a good split. You choose this pivot with a recursive call to 1/5 of the initial input organized in a specific way.

First, go through the input 0 to 4 times, removing the smallest element successively each time until the size of the input is divisible by 5 (r – p +1 mod 5 = 0). If you encounter the order statistic you’re looking for while doing this, just return that. I think this happens everytime where i is the order statistic you’re looking for.

Second, find g, the number of times the new input size can be divided by 5. We will use this number to divide the input into g groups of five.

Third, create sorted groups of 5 as follows:

for j = p to p + g -1:

sort <nj, nj + g, nj + 2g, nj + 3g, nn + 4g> in place in A.

That is, for example, if nj > nj+g swap them in A.

Fourth, we now have the median of each group in the middle 5th of A. Select the median of these medians with a recursive call to Select(A, p+2g, p+3g-1, ). We know by how we constructed it that this median is less than or equal to at least 3g/2 elements and more than or equal to at least 3g/2 elements.

Fifth, use the median of medians as your partition in a modified partition function that partitions around a specific value.

Sixth, run Select on the correct part of the partition, worst case, this has size 7n/10.

The recurrence is: . We can show by substitution that this is . T(n/5) comes from the call to Select to find the median of medians (the pivot). T(7n/10) comes from the worst case partition where the median of medians divides the set of elements as poorly as it’s able and the target is in the larger set. comes from the time everything outside the recurrences takes. It is the summation of: (1) the time to get the input to a size divisible by 5 (<5n), (2) the time it takes to sort the groups (O(1) \* g), and (3) the time it takes to partition around the custom pivot ().

**9.3-6 With black box linear time median function, write a simple linear select function**

***Select(A, p, r, i)*:**

If p==r:

Return A[p]

x = Median(A, p, r)

q = Partition-Around(A,p,r,x)

k = q – p +1

if i==k:

return A[q]

elif I<k:

return Select(A,p,q-1,i)

else:

return Select(A, q+1,r,i-k)

# Part 3 – Data Structures

On computers Data Structures are dynamic sets that support certain operations. At a minimum they support, insert elements into, delete elements from, and test membership, this is called a *dictionary*. We’ve already seen a more complicated structure in Priority-Queue, which supports: return-min/max. The best type of dynamic set for a given problem depends on the operations you need.

**Elements in Dynamic Sets**

Elements are represented by an object whose attributes can be examined using a pointer. Some sets assume the objects have a unique key attribute that can be used to identify the object. If so, the set can be regarded as a set of key-values. The other data of the object is satellite data and is not used in set operations (although it may be altered by them). It may also be the case that the attributes have pointers to other objects in the set.

Sometimes the keys are drawn from a totally ordered set. This gives us a minimum element of a set and allows to speak of the next largest element in a set given some element.

**Operations in dynamic sets**

There are two types of operations: queries and modifying operations. Queries return information about elements of a set and modifying operations alter the set. Here are some examples.

Select(S, k), a query returns a pointer x to an element in set S such that x.key = k or nil if no such element belongs to s.

Insert(S, x), adds element pointed to by x into set S. We assume attributes needed by the Set implementation have already been initialized.

Delete(S, x) deletes element pointed to by x.

Minimum(S), Maximum(S), returns pointer x such that x.key is less than (greater than) all other keys in S. Assumes the keys are totally ordered.

Successor(S, x), returns pointer to next larger element in x or nil if x points to the largest element. Assumes x.key is from the totally ordered set S of all keys.

Predecessor(S,x) reverse of successor.

We study different data structures to improve the time of operations. For example, arrays are good data structures and make algorithms easy to write. However, Search, Min/Max, Successor/Predecessor all take . If we maintain a sorted list then Min/Max/Succ/Pred take O(1) but insert and delete take . Search takes .

We will see structures in this section that allow all operations to happen in O(lg n) time.

Chapter 10 covers basic structures, arrays, linked lists, matrices, stacks, queues, and rooted trees

Chapter 11 covers hash tables.

Chapter 12 covers binary trees

Chapter 13, covers red-black trees. These guarantee operations happen in O(lg n).

## Chapter 10: Simple data structures, arrays, matrices, linked lists, stacks queues, rooted trees

### 10.1 Arrays, matrices, stacks queues

**Arrays**

Arrays tend to be stored contiguously in memory. If a is the location in memory of the start of the array, each element occupies b bytes of data, and the first index of the array is s. Then the ith element of the array in memory begins at a + b(i – s) and ends at a + b(i – s + 1) – 1. If s=1 as it often does in out algorithms then the ith element of the array starts at a + b(i-1) and ends at a + b(i) – 1.

If the objects stored in the array are different sizes then the above formulas don’t work. Usually in this case we’ll store pointers in the array that point to the relevant object. Pointers are the same size regardless of what they point to.

**Matrices**

Matrices can be stored in row major or column major pattern and can either be stored as one dimensional arrays which have either the number of columns as an attribute (in the case of row-major storage) or the number of rows (in the case of column major storage).

A = [[1,2,3],

[4,5,6]]

Is A= [1,2,3,4,5,6] stored row major and [1,4,2,5,3,6] stored column major.

You can also store row/column major using a pointer for each row/column. Then you index A[1,2] with A[1][2] (for row major).

Single array storage is usually more efficient but multiarray storage allows for ragged arrays that have rows/columns of different lengths.

**Stacks/Queues**

Both prespecify the element targeted by Delete. Stacks are LIFO (last-in-first-out) and queues are FIFO (first-in-first-out).

*Stack*

Stacks can be implemented with special Push, Pop, and IsEmpty commands. They also require a .top and .size attribute.

isEmpty(S):

if S.top==0

return True

else return False

Push(S, x):

If S.top==S.size:

Return error Overflow

Else:

S.top+=1

S[S.top]=x

Pop(S):

If isEmpty(S):

Return error underflow

Else:

s.top = s.top-1

return S[S.top +1]

*Queues*

Queues work with Enqueue and Dequeue operations to add and remove elements and head and tail attributes to track the beginning and end of the queue.

Enqueue(S, x):

If S.tail+1 == S.head or if S.tail==size AND S.head==1:

Return error Overflow

S[S.tail] = x

If S.tail==S.Size:

S.tail=1

Else:

S.tail= S.tail +1

Dequeue(S):

If S.head==S.Tail

Return error underflow

X = S[S.head]

If s.Head+1 == s.Size

S.Head = 1

Else:

S.head = S.head + 1

Return X

### 10.2 Linked Lists

Linked lists are structures where the elements have a key value attribute, an attribute next that points to next element. It also has a head element, L.head points to the first element of the list. If an element’s next attribute is nil, then it is the tail of the list.

A doubly linked list has a previous pointer attribute as well. A circular linked list, the tail’s next attribute points to the first element and the first element’s previous attribute points to the last element. Lists can also be sorted or unsorted based on the key attribute.

Lists have the List-Search method which takes a list and a value and returns the first element in the list with that value.

List-Search(L, k):

x = L.head

while x != Nil and x.key != k:

x = x.next

return x

The Prepend function adds an element to the start of the list in constant time. Notice that L.head is a pointer to what was previously the first element in the list. Setting L.head.prev is setting the previous attribute of the first element to be the inserted element.

***Prepend(L,x):***

x.next = L.head

x.prev = Nil

if L.head != Nil

L.head.prev = x

L.head = x

We can insert x anywhere into the list (e.g. after y) in constant time without even referencing the list

***Insert(x,y)***:

x.next = y.next

x.prev = y

if y.next!=Nil:

y.next.prev = x

y.next = x

Deleting from a list given a pointer happens in constant time. Deleting an element given its key takes linear time because you have to run search to find the element, then run delete.

ListDelete(L, x):

If x.next!=Nil

x.next.prev = x.prev

if x.prev!=Nil

x.prev.next = x.next

else L.head = x.next

Inserting and deleting is faster for doubly linked lists than for arrays because for arrays you have to move a certain number of elements (<n) over after inserting or deleting. However, lookup is faster for arrays (constant instead of linear).

We can simplify the code of some of these if we use a circular doubly linked list with a sentinel. A sentinel is an empty element pointed to by L.Nil. L.Nil.next points to the first element and L.Nil.prev points to the last. The last element’s next points to L.Nil and vice versa for the first element. When initialized without elements, L.Nil.next and L.Nil.prev are L.Nil.

ListDelete\*(L,x):

x.next.prev = x.prev

x.prev.next = x.next

We don’t need to check for Nil anymore. If you pass L.Nil as x then the whole list is deleted.

**Exercises 10.2-2 and 10.2-3**

Can implement stacks and queues of arbitrary size using singly linked lists. Even though Delete takes O(n) time in singly linked list, pop can happen in constant time because you pop the first element of the list.

Queues require adding an L.Tail attribute in order to be able to add elements to the end in constant time. Dequeue is the same as Pop and Enqueue simply adds the element to the end of the list instead of the beginning. You also need a IsEmpty function to test for underflow.

### 10.3 Representing Rooted Trees

The elements of the tree are Nodes with a value, a p attribute for their parent and some child attribute. For binary trees you have N.L, N.R for left and right children. If N.p is Nil then N is the root of the tree. We’ll also represent the root with T.root. If T.root is Nil then the tree is empty.

Trees whose nodes have a set number of children k are similarly represented but something different is required if the number of children are unbounded.

We use the L-Child and R-Sibling notation for this. For each layer of the tree, if N.R-Sibling is Nil then that N is the last Node of the layer or the last child of the parent node. Each R-Sibling has the same parent attribute as all its siblings. You can imagine that this means it might take longer for a parent to access a child but any child can immediately access its parent.

As we saw in Chapter 6 on heaps, we can represent trees in other ways. In chapter 6, heaps, which are complete binary trees were represented with an arrays and an attribute giving the last index of the heap (.heapsize).

## 11 Hash Tables

Hash tables can perform the dictionary operations (Search, Delete, Insert) in expected constant time. They also use an array that is close to the size of the set of keys/indices. Using simple arrays may require allocating a lot more space than there are actual keys.

Hash tables compute the array index from a key and use techniques (chaining) to handle overlap in keys.

Python uses hash tables to implement Dictionaries.

### 11.1 Direct-Address Tables

There is a universe U of keys but only some of these are actually used in the elements. The Table has 0:m-1 slots and each slot corresponds to a key in the universe of keys. Elements are accessed through the table with the key value.

11.2 Hash Tables

When the universe of keys is much larger than the set of used keys, direct access tables are inefficient. Hash tables have 0:m-1 slots where m is much smaller the size of the universe of keys (|U|). m is proportional to k where k is the number of keys in use.

Hash tables use a hash function h to go from key k to a hash value which is a slot of the hashtable. The function is chosen to allow the array representing to the hash table to have significantly less slots than |U|. An example of a simple but bad hash function h(k) is k mod m. This creates collisions which we have to develop techniques to resolve.

Collisions always occur where |U| > m but “random looking” hash functions can minimize this. Independent uniform hashing is a simple implementation of “random looking” hash functions. An ideal hash function has the property that for any k from U h(k) has an equal chance of being any element from 1…m and once h(k) results in a value, it will always result in that value. This is a good theoretical goal but isn’t achievable in practice. We call it independent uniform hashing.

*Collision Resolution by Chaining*

We can resolve collisions by having hash value j be a pointer to the head of a doubly linked list where for every element x, h(x.key) = j. We can then use list functions to add, remove, and find elements in the table:

***ChainedHashInsert(T, x):***

ListPrepend(T[h(x.key)], x)

***ChainedHashDelete(T,x):***

ListDelete(T[h(x.key)], x)

***ChainedHashSearch(T,k):***

Return ListSearch(T[h(k)], k)

*Analysis of Expected Runtime of Chained Hash Table*

Delete and insert happen in constant time regardless of the length of any doubly linked list. Search depends on the length of the list being searched. So the expected time taken to lookup a value in a hash table depends on the expected length of the lists.

We’ll say = n/m = the average length of a list for an independent uniform hash function with chaining. n is the number of elements and m is the slots in the table. nj is the length of the list at T[j] so . . We assume the time it takes to run the hashing function and the time it takes to access the table is constant.

*Theorem 11.1 An unsuccessful search of a chained hash table with independent uniform hashing has an average runtime of .*

We are searching for key k. The search is unsuccessful so every list is equally likely to be accessed by h(k). The expected length of any list is . Searching a list of length takes . Accessing the hash table take constant time so time taken is:

If the key is in the table it is more likely to be in a longer list, so you might expect to the time taken to be longer. This is not true though. We will prove the expected time is still ,

*Theorem 11.2 A successful search of a chained hash table with independent uniform hashing has an average runtime of .*

The time it takes to search a list for element x is one more than the number of elements that occur before x in the list. We prepend elements to the HashTable lists so if we’re looking for x, the time depends on how many collisions occur after x has been inserted. That is, how many xj’s are inserted into the table after x such that h(xj.key) = h(x.key).

Let xi be the ith element inserted into the table where i = 1…n and ki by xi.key.

We will use indicator random variables to represent the statistics we need to assess.

Let Xijq be an indicator random variable such that

Because there are n elements, m slots in the table, and this is a independent uniform hashing:

Since this is an indicator random variable, the expected value is: 1/nm2.

Next we define Yj to be 1 if xj appears earlier in a list that the element being searched for.

Note that at most, only one of the Xijq’s will be one and possibly none of them will be.

Finally, Z is the number of elements that appear in the list before the element being searched for:

When we search a list we will look at the element being searched for as well so the searching is expected to take: E[Z + 1] = E[Z] + 1

Since = n/m, if n = O(m) then ChainedHashSearch runs in O(1) on average.

**Exercise 11.2-1**

Number of Collisions is Y.

(Some notes on mistakes I made. Initially I thought E[Xij] = 1/m2. I was incorrectly importing the probability from the proof: Pr(h(xi) = q AND h(xj) = q). Because we set a specific value of the hash table these have to equal we have 1/m2. For E[Xij], h(xi) can be any value and you just have the probability that h(xj) equals that, which is 1/m. Second mistake, I thought you could just multiply by *m*. This doesn’t work because n/m is the expected length of the table at a certain hash value. This does not equal the number of collisions.)

### 11.3 Hash Functions

Nowadays the best hash functions choose randomly from a set of hash functions at runtime. This introduces randomness into the algorithm that isn’t reliant on the distribution of keys. It can be shown to get close enough to independent uniform hashing that the expected runtimes are O(1).

Sometimes, we may know the distribution of the keys. If keys are uniformly distributed between 0 and 1, then h(k) = km is a good hash function. Often we don’t know the distribution of the keys though.

#### 11.3.1 Static Hashing

There is also static hashing. Static hashing by division is just taking the mod m of the key. This might work okay if m is a prime not too close to a power of 2 but isn’t recommended.

*Multiplication Method*

Choose an A between 0 and 1. . This lets you choose m to be whatever you want.

A variation of the multiplication is the multiply shift method. It only works where m = 2l. Instead of A we have a = A\*2w, where again A is 0 < A < 1. a is a w-bit word and so is k. So a\*k is a 2w-bit word. If we think of a\*k as two w-bit words concatenated (r1 and r2), we will only consider the lower order w-bit word, r2 (same as taking mod 2w). Finally we’ll shift r2 right by w – l bits to generate an l-bit integer, this will be our hash value. Simply:

Where >>> X means shift X bits to the left. The benefit of this is it can be executed quickly and efficiently by a computer: multiply, subtract, shift. I’m not sure how one would implement this though. For example, it seems like you multiply the two w-bit numbers together then just ignore the first w-bits, not sure how that is done…

The multiply shift method doesn’t guarantee independent uniform hashing. If we choose a to be a random odd integer less than m, then the family of multiply shift functions given by all the values of a is *­2/m-universal* (defined below) and runs very quickly. In practice, this is probably the best way to choose hash functions.

#### 11.3.2 Random Hashing

If we have a static hash function then malicious actors can exploit it by generating keys that always result in worst behaviour. Random hashing fixes this by randomly choosing from a family of hash functions at runtime.

*Universal Hashing* is one type of random hashing that works well. A set of hash functions H is *universal* if for each distinct pair of keys k1 and k2 , the number of , such that is less than |H|/m. Meaning the probability that is 1/m.

**Corollary 11.3**

If we use universal hashing and chaining any sequence of s dictionary operations (Insert, Delete, Search) containing n = O(m) Insert operations will be executed in .

*Proof*

Insert and delete happen in constant time. For search, since the chance of collision for any two keys is at most 1/m, we know from theorem 11.2 that the expected time for search is constant as well. Therefore the time for s operations, regardless of what they are, is O(s). Each operation takes so in total we have .

Other terminology:

An -*universal* family of hash functions is a family H where for any two keys k1, k2 and any . So a universal family is also *1/m-universal*. As stated above, the family of hash functions given by the multiply-shift function is *2/m-universal*.

A d-independent family of hash functions H is a family such that for any d distinct keys (k1…kd), and any slots (q1…q­d) not necessarily distinct, (ki) = qi) for all is 1/md.

#### 11.3.4 Designing a Universal Hash Function Family

The book presents two options, constructing a universal family with number theory and constructing one using the multiply shift method mentioned above.

The number theory family is as follows:

Where and , where p is some prime number greater than the largest possible key. Also:

Each hash function hab­­ maps to . It contains p(p-1) hash functions since |a| = p-1 and |b| = p. Furthermore, m does not have to be prime.

***Theorem 11.4***

The family is universal.

*Proof*

For k1 and k2 where consider r1 and r­2 where and . With and

We know a few things about r1­ ­ and r2:

(1) , consider . This is not 0 because neither mod p nor mod p are 0 and p is prime. Chapter 31 has the number theory theorems needed to show this, specifically theorem 31.6.

(2) There are the same number of possible (a,b) pairings as there are possible (r1,r2) pairings. First note, that every different (a,b) pairing results in a different (r1,r2) pairing. We can see this if we invert the formulas:

If we set r2 then a and b are different for every . Same if we set r1. Finally, for all p possible values of r1 we have p-1 possible values of r2 since . So the number of possible pairs is p(p-1), same as (a,b).

Therefore, if we uniformly at random choose a pair (a,b), the (r1, r2) is uniformly chosen at random as well. Any value for r1 and r2 are equally likely.

Therefore, the probability that is the same as the probability that . If we set r1, of the (p-1) remaining possible values for r2, at most cause a collision.

Since there are p choices for setting r1, there are at most

Furthermore, the chance of a collision for any two distinct keys k1 and k2 is 1/m.

**Randomized Multiply Shift**

Theorem 11.5 (not proven in the book): is *2/m-universal*.

Having the very efficient multiply shift method and only being *2/m-universal* means drawing hashing using this family is very efficient and recommended.

#### 11.3.5 Cryptographic Hash Functions (Handling Inputs of Variable Lengths)

Sometimes input keys will be too big to represent with a standard 16 or 32 bit integer. Or they may be something else entirely that is of variable length, like a vector of 8-bit bytes (commonly used for strings in programming languages. One could extend the universal hash functions described just above. But one could also just use any number of cryptographic functions. E.g. SHA-256.

We could make a hash to a table of size m by saying

We can even create a family of hash functions by prepending a salt vector .

And we randomly choose from a set of strings and means we concatenate to the end of .

One benefit of using cryptographic functions is that some hardware is built to run these functions extremely efficiently

### 11.4 Open Addressing

Each key has a list of preferred addresses and is stored in the first available preferred address. To generate this you use a probe number U and the function is: where and is a permutation of

***HashInsert(T,k):***

i = 0

repeat

q = h(k, i)

if T(q) is empty:

T(q) = k

Return q

Else i = i +1

Until i == m

Return overflow

***HashSearch(T,k):***

i = 0

repeat

q = h(k, i)

if T(q) = k:

return q

i = i+1

until T(q)== Nil OR i == m

return Nil