
Lecture 1:

ANALYZING ALGORITHMS
Two Goals:
1. Algorithm is correct
Let f be a function that takes input x and produces output y.
f is correct if and only if y has the correct properties for every possible x.
EX. If f is a sorting algorithm that sorts in ascending order, the algorihm always produces a sorted result.

2. Algorithm is efficient
A correct algorithm can be inefficient. An effective algorithm uses as few resources as possible- we want to minimize time and space.
We determine resource use from atomic operations.

3. Algorithm is elegant
The first pass at developing an algorithm is rarely elegant.

ATOMIC OPERATIONS
A good algorithm is machine independant. In evaluating the efficiency of an algorithm, we want to evaluate the algorithm, not it's implementation.

Random Access Memory (RAM) model

RAM PROPERTIES
Atomic Operations: Simple math operations (+, -, *, /), and simple functional operations (if, call, store)
-take on unit of time

Higher Order Operations: Loops (for, while) and subroutines are not atomic. They are composed of atomic operations.
Within the loop, there are atomic operations. To find the cost of the loop, you multiply the sum of the atomic operations by the number of excecutions of the loop.

ATOMIC DATA TYPES
Simple data types: bool, int, char, float
Each takes up one unit of space

HIGHER ORDER DATA TYPES
Complex data types: Objects, arrays, lists, trees
Are not atomic; are composed of atomic data types

ALGORITHM RUNTIME ANALYSIS
1. Worst Case Analysis:
Provide a robust guarentee of algorithm efficiency by examining worst case behavior. Behavior is no worse than x, and what input produces that behavior.

Big O notation.

2. Average, best case analysis:
Average performance may or may not be better than worst case.
Best case is the best case we can hope for. Big Omega (Ω) notation.

Lecture 2:

ASYMPTOTIC ANALYSIS
Resource use as size of input n -> infinity.
Analysis is always a function of the input size for determining time and space resources.

Why asymptotic analysis?
Assume two algorithms: One has a high start up cost but it is very efficient. The other has a low start up cost, but is very inefficient. With a small input, the second algorithm will by more efficient, but as the size of the input increases, runtime of the first algorithm will be lower than the second.

If T(n) is the runtime of an algorithm with input size n, we want to relate T(n) to a known growth form, g(n), such that:
The lim as n -> infinity of T(n) = Γ(g(n)), where Γ is the asymptotic relationship.

O = upper bound, worst case, Big O analysis
Ω = lower bound, best case, Big Omega analysis
θ - tight bound, occurs when best case = worst case

Examples of g(n):
c - constant resource usage, independant of n
log(n) - sublinear, specifically logarithmic resource usage
n^c, where c<1 - sublinear resource usage
n - linear resource usage
n^c for c>1 - polynomial resource usage, super-linear
c^n for c>1 - exponential resource usage
n^n - extremely fast growing resource usage, seriosuly time to redesign your code

Asymptotic upper bound may or may not be asymptotically tight
Zn^2 = O(n^2) is a tight bound
Zn =)(n^2) is technically correct, but is a very loose bound

We use litte o, little omega for a lower bound that is not asymptotically tight notation for bounds that are not asymptotically tight

DID NOT TYPE OUT ALL OF EARLIER LECTURES AS THEY'RE PTRETTY BASIC

Divide and Conquer

Steps: Divide, Conquer, and Combine
Fundamentally, divide and conquer is a recursive approach.

Balanced and Unbalanced trees:

A full balanced tree has depth $\theta(\log n)$
A fully unbalanced tree has depth $\theta(n)$

Some Tree Definitions:

Full binary tree- a tree in which every node other than the leaves has 2 children
Complete binary tree- a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
Balanced binary tree- a tree in which for each node it holds that the number of inner nodes in the left subtree and the number of inner nodes in the right subtree differ at most by one. In other words, a binary tree is balanced if for any two leaves the difference of depth is at most 1.

Recurrence Relations:

Equation that recursively defines a sequence using previous terms to define the next term. Divide and conquer algorithms written as recurrence relation with them form:

$T(n) = aT(g(n)) + f(n)$
Where:

$T(n)$ is the runtime for a problem of size n
a is a constant denoting the number of subproblem
$g(n)$ is the size of the subproblems as a function of n
$f(n)$ is the time to combine the smaller problems, or divide the problem into smaller instances

Solve recursive relation to find the asymptotic behavior of an algorithm

EXAMPLE: SOLVING THE RECURRENCE RELATION FOR MERGE SORT

Step 1: Cost to solve subproblems

$\theta(1)$, if $n <= c$, for some constant c
$T(n) = aT(n/b) + D(n) + C(n)$, where $D(n)$ is the cost of the divide and $C(n)$ is the cost of the combine.

Step 2: Find D(n) and C(n) for merge sort

The divide in this algorithm is the midpoint calculation
so, $D(n) = \theta(1)$ <- because it happens once, regardless of input size

The conquer step is the merge step
C(n) = θ(n), because merge takes θ(n) on an n-element subarray

Step 3: Conquer
a=b=2 so equation is 2T(n/b), because the algorithm divides the problem into 2 subproblems, each half the size of the original problem.

Thus, the recurrence relation for merge sort is:

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 1 \\ 2T(n/2) + \theta(n), & \text{if } n>1 \end{cases}$$

EXAMPLE: SOLVING FOR THE RUNTIME USING THE RECURRENCE RELATION FOR MERGE SORT

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 1 \\ 2T(n/2) + cn, & \text{if } n>1 \end{cases}$$

The root is the cost of the top level recurrence (which in this problem is cn).
T(n/2) contribute to cost cn.
At the root, the cost is cn. At each step when the problem is divided, the subproblem will have half of the previous cost. However, the sum of the costs of all of the subproblems on that level will add up to cn.

Therefore, the total cost of the algorithm is the number of level * the cost at each level.
The number of levels is log(n) + 1 //the one is to account for the root
The cost of a level is cn.
So, we have cn*log(n) + cn.
When you remove lower order terms and constants, we find the runtime is:

θ(n*log(n)

Quicksort Algorithm

This algorithm is a divde and conquer approach for sorting

Worst case: O(n^2)
Average Case: O(n*log(n))

Steps:
Divide - pick an element A[q] in array A and use to partition the array into A1 and A2, such that A1 <= A[q] and A2 > A[q]
for example, if A = [15, 20, 5, 6, 2], and A[q] = 6, then A1 = [5,6,3] and A2 = [15, 20]

Conquer - Recursively call Quicksort() on A1 and A2.

Combine- Return A1 concatenated with A2, which will be a sorted array A.

Pseudocode for Quicksort():

Preconditions: A is an array to be sorted, p >= 0, r <= the last index of A
//p is beginning of array passed, r is end

```
Quicksort(A, p, r)
{
    if (p < r)
    {
    q= partition(A, p, r)
    Quicksort(A, p, q - 1)
    Quicksort A, q + 1, r
    }
}
```

Pseudocode for partition():

The partition() method is the meat of the algorithm. The runtime of quicksort is strongly influenced by the chosen partition. If the partition is correct, we can assume that Quicksort() is correct.

Preconditions: A[p... r] is the array to partition, r>=0 and r<= last index of A

Post-condition: All elements A[p... q-1] < A[q] and A[q+1... r] > A[q]

NOTICE: A sorted array is NOT a post-condition of partition()

Worst case input (O(n^2)) occurs when the pivot chosen is the last or firstst element in the array. It will return an empty array and the original array.

```
partition(A, p, r)
{
    x=A[r] //x is the pivot value
    i = p-1
    for (j = p to r-1)
    {
        if (A[j] <= x)
        {
            i = i + 1
            swap (A[i], A[j])
        }
        swap (A[i+1], A[j])
        return i+1
    }
}
```

Example of Quicksort running:

A = [2, 6, 4, 1, 5, 3]

Quicksort(A, 0, 5);

```
partition(A, 0, 5)
{
    x = 3
    i = -1
    for (j = 0 to 4)
    {
        //first A[0] <= 3 is true so:
            i = 0
            swap(A[0], A[0])
        //second A[1] is not less than 2
        //third a[2] is not less than 3
        //fourth A[3] <= 3 so
            i=1
            swap(A[1], A[3])
    }
    //At the end f this for loop, A = [2, 1, 4, 6, 5, 3]
    swap([A[i+1], A[j])          //swaps A[2] and the last element in the array, our pivot value
    //At the end of the method, A = [2, 1, 3, 6, 5, 4]
    return i+1              //i+1 in this cae is 2

}
```

Correctness and Runtime:

Correctness- We'll show that partition is correct. Partition drives the p,r inputs to Quicksort, so if partition is correct, so is Quicksort().

There are 4 regions in array in partition

1.
   At the beginning of a loop iteration, values <= x, where x is the pivot.
   If p <= k <= i, A[k] <= x
2.
   Values > x
   If i + 1 <= k <= j-1, then A[k] > x
3.
   Pivot value, A[r]
   If k=r, then A[k]=x

4. Unprocessed region of A, between j and r-1 in A

The region statements 1 through 3 in the above are the loop invariant. No need to do full proof out.

Runtime-
Recall the general form of a recurrence relation is $T(n) = aT(g(n)) + f(n)$
The total runtime of Quicksort depends on whether the partitioning (the recursive step) is balanced or not, which depends solely on which element is used for the pivot.

Worst case performance: $\theta(n^2)$

//we use θ because we're assuming worst case input and this is a tight bound on the worst case

Worst case partitioning occurs when Partition produces 2 subproblems, one which has n-1 elements and one that has zero elements.

Example: A = [1, 2, 3, 4, 5]
Partition returns 4
Quicksort(A=[1, 2, 3, 4], 0, 3)
Quicksort(A=[], 4, 4)

To get worst case behavior, we need worst case partitioning on some constant number of partitions.

Best case performance: θ(n*log(n))

Best case behavior occurs when partition produces an even split into 2 subproblems, with size no more than n/2

Recurrence of best case is the same as merge sort...
T(n) = 2T(n/2) + θ
This meets case 2 of the Master's Theorem
T(n) = θ(n*log(n))

Average case performance: θ(n*log(n))

Average case is much closer to the best case than the worst.

For example, assume a partition that produces a 9:1 split.
(i.e. if n = 10, n1 = 9 and n2 = 1)
Recurrence:
T(n) = T((9/10)n) + T(n/10) + θ(n)
Replace θ(n) with cn

As with merge sort, the sum of the costs of each subproblem on a level is cn.

So, the recursion terminates at θ(log(n)) <- the cost of each level is log(n)
Number of levels * cost of each level is O(n*log(n)

In the actual implementation of quicksort, we can get a mix of good and bad partitions. If we alternate good and bad splits, the cost of a bad split is absorbed the the cost of a good split, so behavior is still O(n*log(n).

Randomized Algorithms

A randomized algorithm is one that employs a degree of randomness by design. It is designed to achieve good performance in the average case.

Ex. hash tables, cryptography, etc.

Hiring Problem Example:

You want to hire a new assistant. There is a small cost to interview a candidate and a larger cost to hire. You want the best person at all times. If after the interview, the candidate is better than the current employee, fire the current employee and hire and the candidate.

Pseudo-code:

```
hireAssistant()
{
    best = 0;
    for (i=1 to n) //candidates # 1 to n
    {
        interviewCandidate();
        if (candidate[i]>best)
        {
            best = i
            hireCandidate(i);
        }
    }
}
```

The cost in this case is not about the runtime but about the cost of hiring.
Let Ci be the cost to interview.
Let Ch be the cost to hire.
Let m be the number of people hired, and n be the total number of candidates.

Worst case:

Hire ever candidate (they arrive in increasing quality). Cost O(Ch n)

Average case:

ordering unknown

Probabalistic analysis:

Average case cost by taking the average over the dist of all possible inputs. Assume candidates arrive in a random order, and each candidate has rank 1...n. Thus, there are n! orderings of candidates, and each ordering is equally likely. Uniform randomness permutations.

If we can't assume randomness, we need to impose it. You can impose randomness by generating a random number between 1 and n, and selecting that candidate from the list to interview.

Indicator Random Variable:

A special type of random variable.

Given space S and event A, independant random variable I[A] associates with event A defined as:

$$I[A] = \begin{cases} 1 \text{ if event A happens} \\ 0 \text{ if event A doesn't happen} \end{cases}$$

Incorporating IRV with the hiring problem:

We want to find the expected number of times that we hire a new assistant.

Let Xi = I{candidate i is hired}
= {1 if candidate i is hired; 0 if candidate i is not hired}

X is the number of times we hire an assitant, where X = X1 + X2 + ... + Xn.

E[Xi] = Probability candidate i is hired.

if candidate i is hired, he was better than candidates 1 through i-1. Since order is random, any of the first i candidates is equally likely to be the best so far. Candidate i has probability (1/i) of being the most qualified candidate/ being hired.

E[Xi] = 1/i
Compute E[x]:
E[x] = E[summation of Xi from 1 to n]
= the summation from 1 to n of E[Xi]
= the summation from 1 to n of 1/i
= ln(n)

Therefore, if we interview n people, we expect to hire approaximately ln(n) of them on average.

Master's Therorem

The general form of the Master's Theorem is T(n) = aT(n/b) + f(n), where:
n is the size of the problem
a is the number of subproblems in the recursion
n/b is the size of each subproblem
f(n) is the cost of the work done outside the recursive calls

It has 3 cases:
Case 1: If f(n) = θ(n^c), where c<log(base b)a (using big O notation), then:
T(n) = θ(n^(log(base b)a))

Example:
T(n) = 8T(n/2) + 1000n^(2)
We can se that:
a = 8, b = 2, f(n) = 1000n^2
f(n)= θ(n^c), where c=2
log(base b)a = log(base 2)8 = 3 > c
So, we can see from Case 1 of the Master's theorem that:
T(n) = θ(n^(log(base b)a) = n^3

Case 2: If it is true, for some constant k>= θ, that:
f(n) = θ(n^c log^(k) n), where c= log(base b)a, then:
T(n) = θ(n^(c)*log^(k+1)*n)

Example:
T(n) = 2T(n/2) + 10n
a=2, b=2, c=1, and f(n) = 10n
log(base b)a = log(base 2)2 = 1, so c = log(base b)a, so case 2 is satisfied.
Therefore:
T(n) = θ(n^(log(base b)a) * log^(k+1) *n) = θ(n^(1) log^(1) n) = θ(n*log(n)).

Case 3: If it is true that f(n) = θ(n^c), where c > log(base b)a, then:
T(n) = θ(f(n))

Example:
T(n) = 2T(n/2) + n^2
As we can see:
a = 2, b = 2, f(n) = n^2 so c = 2
log(base b)a = log(base 2)2 = 11. c > 1.
So it follows from the third case of the master's theorem that:
T(n) = θ(f(n)) = θ(n^(2))

Universal Hashing

Multiplication Method:
This is a way to create hash functions. The functions take the form:
h(k) = [m(kA mod 1)], where:
0 < A < 1 and kA mod 1 refers to the fractional part of kA. Since 0 < (kA mod 1) < 1, the range of h(k) is from 0 to m.
The advantage of the multiplication method is that it works equally well for any size m. A should be chosen carefully; rational numbers should not be chosen for a.
The advantage of this method is that the value of m is not critical.

Universal Hashing:
Chose hash function randomly from a family of hash functions, independant of keys being stored.
Guaranteed good performance average.
Behaves differently on each excecution.
Let H be a family of hash functions, and h ϵ H be a hash function.
H is universal if Pr(h(j) = h(k)) < 1/n, where:
j and k are keys, and n is the table size
Two keys in universe of keys will collide with a probability of 1/n

Creating a set of universal hash functions:
One option:
1. Choose table size m to be prime
2. Decompose key X into r + 1 components, such that X = <X0, X1, ..., Xr>, where the maximum value of any Xi < m.

3. Let a= <a0, a1, ..., ar> denote the sequence of r+1 elements chosen randomly such that ai ϵ {0, 1, ... m-1}

There are m^(r+1) combinations for a

EXAMPLE: ai = {0,1}, r=4, then m=2
a = < a0, a1, a2, a3, a4 >
m^(r+1) = 2^5 = 32 combinations for a

4. Define a hash function ha, where ha(x) = the summation from 0 to r of (aiXi % m)

5. h = U{ha}, universe of hash functions with m^(r+1) members, one for each sequence of a.

Load factor:

Assuming we have a hash function with low collisions. It the load on the hash table is low, then operations are O(1).

Load factor a is the average size of a bucket.

The expected time for operations is O(1 + a), where a = entries/bucket

For m entries in the table, how large does n have to be before every bucket has at least one element in it?

This can be described by the coupon collector problem.

See her dynamic programming pdf for this problem

How large does n need to be before we start seeing buckets with multiple elements (collisions)?

This can be described by the birthday problem.

See her dynamic programming pdf for this problem

Dynamic Programming

This is a general strategy for solving problems
Approach:

1. Define the value of an optimal solution in terms of overlapping subproblems, with the same properties as the original problem.
2. Store results of subproblems for later use
3. Reconstruct optimal solution from stored information

Not every problem can be solved by dynamic programming. For example, sorting problems cannot.

Memoize - Term for storing solved subproblems

Example:

Use dynamic programming to count the number of paths in a directed, acyclic graph directed means edges have direction, and acyclic means there are no cycles

```
2             > 5
^ \          /  |
|  v        /   v
1-> 3 ---->4 --->6
```

Number of paths from 1 to 6?
In general terms:
Let i and j be nodes in the graph. How many paths are there from i to j?
x = the number of paths from i to j
Let s(j) be the set of nodes a that connect to j

```
a1  \
a2 -> j
a3  /
```

Each of the x paths must pass through neighbour ai of j

The total number of path x = the number of paths i to a1 + number of paths from i to a2 ... + number of paths from i to an

Xi,j = the summation of a's in s(j) (Xi, a)

Xi,j counts paths i to j
a is the index of nodes in s(j), which point to j.

There is 1 path from x1 to x2, X1,2 + 1 paths from x1 to x3, X1, X3 paths from x1 to x4, X4 paths from X1 to X5, and X4 + X5 paths from X1 to X6, so there are four total paths to X6 from X1.

Another Example is the rod cutting problem:
Given a rod of length n inches and a table of prices pi for i = 1, 2, ..., n, determine the maximum revenue rn obtainable by cutting rods and selling the pieces.

Ex.
{Length, price}
{1, 1}
{2, 5}
{3, 8}
{4, 9}

n=4

With no cuts, p4 = 9
With one cut, p2 + p2 = 10
p1 + p3 = 9

Thus the maximum profit is by cutting it in half.
We can cut the rod into 2^(n-1) different ways.

Using dynamic programming, we can calculate the solution in polynomial time (n^2).

Pseudocode:

```
//p is an array of prices, n is the length of the rod
    cutRod(p.,n)
    {
    initialize r[o...n]
    r[0] = 0 //there is no profit for a rod of length 0
    for (j = 0 to n):
        q = min_ int; //smallest machine value
        for (i=1 to j):
            q = max(q, p[i] + r[j-i])
        r[j] = q
        return r[n]
    }
```

{Length,	price}
{1,	1}
{2,	5}
{3,	8}
{4,	8}
{5,	10}

n=5

Steps algo is following:

```
    j = 1
    p[i] = p[i] + r[0]
    r[1] = 1
    i=1
    q = max(q, p[i] + r[2-1]) //solution for 2 pieces of length 1
    q = 2
    r[2] = 5
```

This is not super efficient though. We want to modify the lgorithm to store the size of the first cut.

Start with the above cutRod() algorithm.
Then add an array s[0...n] that stores optimal first cut length.

Then modify the inner loop:

```
        for (i=0 t0 j):
            if (q < p[i] + r[j-1]):
                q = p[i] + r[j-i]
                s[j] = i
        r[j] = q
    return both r and s
```