Lecture Notes
Kenrick Mock
CS 411

Chapter 1-3 : Basic Concepts, Recurrence with Substitution

Algorithms – so what is an algorithm, anyway?

The dictionary definition: An algorithm is a well-defined computational procedure that takes input and produces output. This class will deal with how to design algorithms and understand their complexity in terms of runtime, space, and correctness.

Examples: data compression, path-finding, game-playing, scheduling, bin packing

Example algorithm: Insertion Sort

Insertion sort is how people often sort a hand of cards. Starting from the left, go towards the right and make sure that everything on the left hand side is correctly sorted. For example:

Original:	5	3	8	2	10	7
Pass 1:	3	5|	8	2	10	7
Pass 2:	3	5	8|	2	10	7
Pass 3:	2	3	5	8|	10	7
Pass 4:	2	3	5	8	10|	7
Pass 5:	2	3	5	7	8	10|

Here is the pseudocode for the Insertion sort algorithm.
Typically we will use pseudocode and not actual implementation code.

```
Insertion-Sort(A)          ; Assume our arrays start at index 1 and not at 0
    For j← 2 to length(A)
    Do
        key←A[j]
        i← j-1
        while i>0 and A[i] > key
            do
                A[i+1]← A[i]
                i← i-1
        A[i+1]←key
```

Run through algorithm on our input, 5 3 8 2 10 7

```
J=2
    A=[5 3 8 2 10 7] Key=3
    I=1
    A[1]>3 so A[2]=A[1]
    A[1]=3
J=3
    A=[3 5 8 2 10 7] Key=8
    I=2
    A[2] < 8
    A[3]=8
J=4
    A=[3 5 8 2 10 7] Key=2
    I=3
    A[3]>2 so A[4]=A[3]
```

A[2]>2 so A[3]=A[2]
A[1]>2 so A[2]=A[1]
A[1]=2
J=5
A=[2 3 5 8 10 7]
…
The final two passes are left as an exercise for you to verify.

It looks like this algorithm works to sort the input. But how do we analyze it? There are also certainly many other ways to sort the data (and we'll look at lots of them!) What makes one algorithm better than another?

Let's make an assumption that we will use throughout most of this class.

RAM model. This is a uniprocessor, random access machine with no parallelism (PRAM).

Run time and space will generally depend on the size of the input.

Input Size 'n' : Definition of n depends on the problem. It may be bits, bytes, but is usually the number of items in the input. For sorting, n=# of items to sort. For a graph, n could be the number of nodes or the number of links.

Running time: defined as the number of steps that are executed in the program.

Lets count up the runtime for this algorithm. This can be a little tricky since the loop doesn't always run the same number of times, but varies depending on the input.

Let t_j = the # of times the inner while loop is examined for the current value of j. This includes checking for the exit condition of the loop.

Code	Cost	Times
For j← 2 to length(A)	C1	n
Do		
key←A[j]	C2	n-1
i← j-1	C3	n-1
while i>0 and A[i] > key	C4	$\sum_{j=2}^{n} t_j$
do		
A[i+1]← A[i]	C5	$\sum_{j=2}^{n} (t_j - 1)$
i← i-1	C6	$\sum_{j=2}^{n} (t_j - 1)$
A[i+1]←key	C7	n-1

The statement for the outer j loop will run n times (once for each element, once to check the exit condition). The inner statements run n-1 times. Since we've defined t_j as the number of times the inner loop is executed for a value of j, to get the total times it is run we just add up the sum or all values of j.

Our total runtime is then:

$$C1(n)+C2(n-1)+C3(n-1)+C4(\sum_{j=2}^{n} t_j)+C5(\sum_{j=2}^{n} t_j - 1)+C6(\sum_{j=2}^{n} t_j - 1)+C7(n-1)$$

Doing a little math:

$$C1(n)+C2(n)-C2+C3(n)-C3+C4(\sum_{j=2}^{n} t_j)+(C5+C6)(\sum_{j=2}^{n} t_j - 1)+C7(n)-C7$$

$$(C1+C2+C3+C7)(n) - (C2+C3+C7) + (C4)(\sum_{j=2}^{n} t_j) + (C5+C6)(\sum_{j=2}^{n} (t_j - 1))$$

What's the best case? If the input is already sorted. In this case, $\forall j, t_j = 1$. That is the inner loop only gets executed once since we'll immediately find the right spot for the last item.

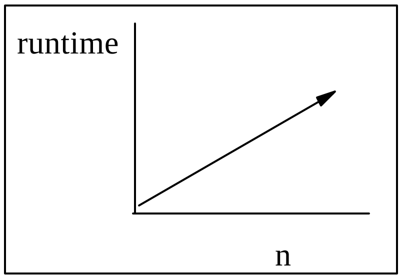
Runtime for this case:

$$(C1+C2+C3+C7)(n) - (C2+C3+C7) + (C4)(\sum_{j=2}^{n} 1) + (C5+C6)(0)$$
$$(C1+C2+C3+C7)(n) - (C2+C3+C7) + (C4)(n-1)$$

$$(C1+C2+C3+C4+C7)(n)-(C2+C3+C7+C4)$$

$$= (CONSTANT1)n - CONSTANT2$$

This is a linear function of n ; y=ax+b. The runtime will increase linearly as the size of the input increases
.



What's the worst case? Lets say that the input conspires to provide the worst runtime. This would happen if the array was in reverse order, because then we have to compare with every other number in the inner loop.

	5	4	3	2	1
j=2	t_j = 2 compares				
j=3	t_j = 3 compares				
j=4	t_j = 4 compares				
…					

So we know that $t_j = j$

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

$$\sum_{j=2}^{n} t_j - 1 = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1$$

$= \frac{n(n+1)}{2} - 1$ - (n-1)

$= \frac{n^2+n}{2} - \frac{2n}{2}$

$= \frac{n(n-1)}{2}$

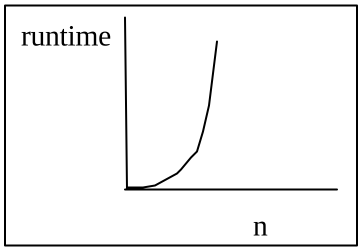
Plug these into our formula:

(C1+C2+C3+C7)(n) – (C2+C3+C7) + (C4)($\sum_{j=2}^{n} t_j$) + (C5+C6)($\sum_{j=2}^{n} t_j - 1$)

(C1+C2+C3+C7)(n) – (C2+C3+C7) + (C4)($\frac{n(n+1)}{2} - 1$) + (C5+C6)($\frac{n(n-1)}{2}$)

Of the form (Constant1)(n^2) + (Constant2)(n) + Constant3

This is a quadratic; an^2+bn+c; the runtime will increase based on the square of the input.



Average case is also quadratic.

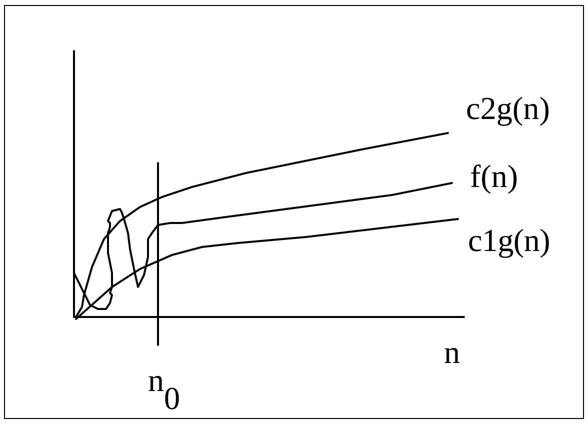
How much space does this take? Sort is done in place, so just a few variables.
This requires constant space.

Function Growth

q -notation : Gives asymptotically tight bound on a functions growth

Definition:

$q(g(n)) = \{ f(n): \exists c_1, c_2, n_0 : 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \}$ for all $n \ge n_0$

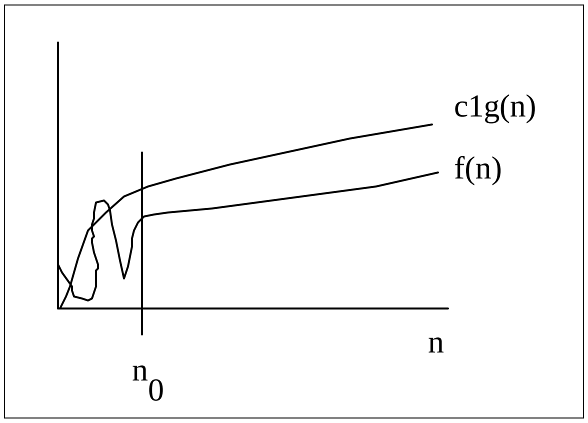


F(n)=3n is q (n) since we can find g(n)=n, c1=1, c2=4.

Insertion sort is not q (n^2) since it is linear in the best case.
However, average and worst case insertion sort is q (n^2).

O-notation: Big-O notation. This gives an asymptotic upper bound.
Definition:

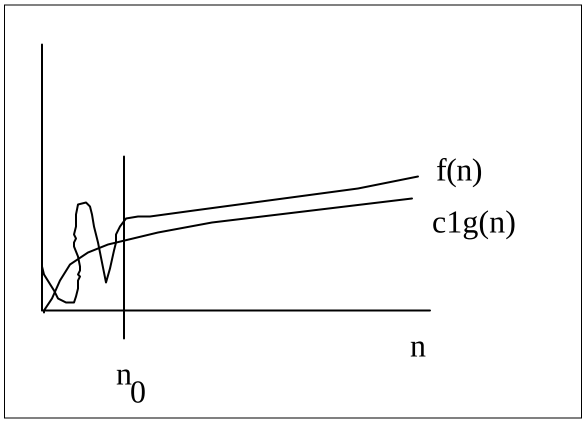
$$O\big(g(n)\big) = \{\, f(n) : \exists c_1, n_0 : 0 \le f(n) \le c_1 g(n) \} \text{ for all } n \ge n_0$$



Insertion sort is O(n^2). Tight upper bound in the worst case, loose in the best case.
Looser: O(n^3), O(2^n).
Use little-o notation if you know that it is a loose bound.
o(n^3) if f=n^2.

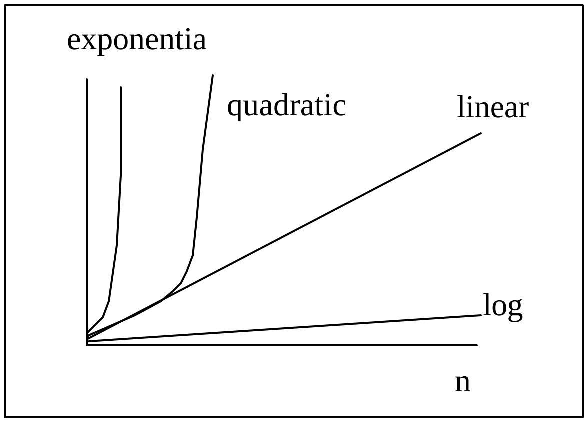
Ω - Omega notation. Omega provides an asymptotic lower bound.
Definition:

$$\Omega\big(g(n)\big) = \{\, f(n) : \exists c_1, n_0 : 0 \le c_1 g(n) \le f(n) \} \text{ for all } n \ge n_0$$



Ω (1) – Constant time, this is a trivial lower bound for most cases.
For insertion sort, Ω (n) : linear time is the best possible
Running Time Comparison:

q	n=10	n=100	n=1000	n=10000
lg(n)	2	5	7	9
n	10	100	1000	10000
n(lg(n))	20	500	7000	90000
n^2	100	10000	1,000,000	100,000,000
2^n	1024	1.3×10^{30}	huge	very huge



For comparison: estimate of the number of atoms in the universe is about 10^{80} (i.e. about 2^{270}).

Tool we will use for analysis and solutions: Divide and Conquer

Recursive approach to solve problems. The idea is to break the problem up into sub-problems with the same solution, but smaller size. Solve these problems recursively, the combine the sub-solutions to solve the original problem.

Divide – into sub-problems
Conquer – Sub-problems recursively
Combine – Sub-problem solutions to original problem

Let's look at Merge Sort:
Divide n elements into two subsequences to be sorted of size n/2
Conquer – sort subsequences recursively with merge sort
Combine – merge sorted subsequences into big sorted answer
Need termination criteria for recursion –
Quit if sequence to sort is length 1

Pseudocode:
```
Merge_Sort(A,p,r)
    If p<r then
        q ← $\left\lfloor \frac{p+r}{2} \right\rfloor$
        Merge_Sort(A,p,q)
        Merge_Sort(A,q+1,r)
        Merge(A,p,q,r)
```

Call with Merge_Sort(A,1,length(A))
q gets the median, merge left, right

A[1…5…10] becomes Merge_Sort(A,1,5), Merge_Sort(A,6,10)
How do we merge?

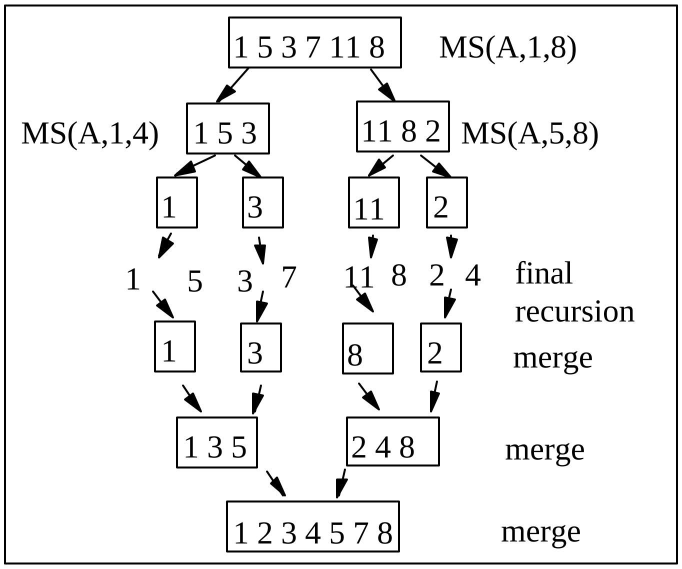
A1:	1	5	8	30	31	50
A2:	2	10	11	12	15	

Combined: 1 2 5 8 10 11 12 15 30 31 50

Easy to do; just start at the front of each array, compare the pointers, and put the smallest into a new array and then increment the pointer that was the smallest.

If n is the number of elements to merge, then it takes *q* (n) time to merge.

Example: A= 1 5 3 7 11 8 2 4



How much work is done? It's the cost to split + the cost to merge.
Let's define T(n) to be the runtime for a problem of size(n).

Recurrence: $$T(n) = \left\{ \begin{array}{c} \Theta(1), n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), n > 1 \end{array} \right\}$$

The 2T(n/2) comes from the divide, and the *q* (n) comes from the merge.
Later we will show that Merge Sort is *q* (nlgn) in runtime.

Math Review

$\lfloor x \rfloor$ - floor of x, round down
$\lceil x \rceil$ - ceiling of x, round up

$\lg(x) = \log_2 x$
$\ln x = \log_e x$

$$a = b^{\log_b a}$$

$\log_c(ab) = \log_c(a) + \log_c(b)$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$\log_b (1/a) = -\log_b a$

$\log_b a = 1 / (\log_a b)$

$$a^{\log_b n} = n^{\log_b a}$$

iterated log : $\log(\log n) = \log^{(2)} n$
This grows very slowly! Almost the same as constant time.

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + 4 + \ldots n = \frac{1}{2} n(n+1) = \Theta(n^2)$$

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{n} = \ln(n) + O(1) = \Theta(\ln n)$$ Harmonic series

$$\sum_{k=0}^{n} a_k - a_{k-1} = a_n - a_0$$ Telescoping

There are some others in chapter 3, but these are the ones you will use most frequently for this course.

Monte Carlo Methods

Throughout this class we'll focus primarily on analytic methods to characterize algorithms. However, there is another class of statistical/probabilistic methods that are commonly referred to as Monte Carlo algorithms. From http://www.qpsf.edu.au/mirrors/csep/mc/node1.html:

Numerical methods that are known as Monte Carlo methods can be loosely described as statistical simulation methods, where statistical simulation is defined in quite general terms to be any method that utilizes sequences of random numbers to perform the simulation. Monte Carlo methods have been used for centuries, but only in the past several decades has the technique gained the status of a full-fledged numerical method capable of addressing the most complex applications. The name "Monte Carlo" was coined during the Manhattan Project of World War II, because of the similarity of statistical simulation to games of chance, and because the capital of Monaco was a center for gambling and similar pursuits.

Monte Carlo is now used routinely in many diverse fields, but has gained some of the greatest popularity in computer science and physics. For example, consider a complex physical system (solar system, nuclear reaction, quantum chromodynamics, traffic patterns, etc.) that has been modeled through a series of pdf's (probability density functions). The behavior and interaction of the model with the input is often too complex to determine deterministically. Instead, the system is simulated by running many trials using random inputs selected from a proper range. The resulting output helps us learn about the outputs of the system and the complexity of the system.

As another example, the merge sort example we previously covered could be analyzed using a monte carlo method by randomly selecting inputs, running the algorithm, and measuring the runtime (or the number of comparisons). By averaging out the average runtime, we could determine that merge sort is a q (nlgn) algorithm.

Recurrence Relations

Earlier we said that:

Recurrence: $$T(n) = \begin{Bmatrix} \Theta(1), n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n), n > 1 \end{Bmatrix}$$

Is q (nlgn).

The next section will show how to compute these bounds. There are generally three methods we will discuss to solve the problems:

Substitution
Iteration
Master Method

Note that we will typically drop out the lower order terms.
For example, if an algorithm runs in q (n) + q (1) time, the q (n) will dominate so we can just drop the q (1) term comp letely.

Substitution Method

The substitution method consists of the following steps:

1. Guess the form of the solution
2. Use mathematical induction to find the constants of the solution, assume the solution works for up to n. We need to then show the solution works for n+1; this is a weak form of proof by induction.
3. Show that the solution works for the new case
4. This method works best if its easy to guess the form of the answer
5. Solve the base case

Example: Given	T(n)= K , i.e. q (1)	for n<=1	; Where K is a constant often not specified and left implicit
	T(n)=2T(n/2) + n	for n>1	; similar to what we had for merge sort

Let's guess the form of the solution is O(nlgn).

Next we need to prove that T(n) $\leq$ c n lg(n) for c>0 (recall definition of O(nlgn)

Substitute back in to original guess:

$T(n) \le 2(c(n/2)\lg(n/2) + n$
$T(n) \le cn \lg(n/2) + n$
$T(n) \le cn \lg(n) - cn\lg 2 + n$
$T(n) \le cn \lg(n) - cn + n$
$T(n) \le cn \lg n$ if $c \ge 1$

Looks good so far, we must also show the boundary condition (or base case) fits our guess by showing that T(constant) from the original recurrence $\le$ guess condition.

We guessed that $T(n) \le c\, n \lg(n)$
If $T(1)=K$ then

$T(1) \le c(1)\lg 1$
$T(1) \le c*1*0$

In the original however, $T(1) = K$.

Is $K \le c(0)$? This does not work for ANY value of c. We know it takes more time than 0!

But we don't have to make the $T(1)=K$ solution fit, we can use a higher number of n. Remember we only need to show that the solution is true for $n > n_0$ and can ignore transients early on.

Try $T(2) \le c2\lg 2$
$T(2) \le c2$

Orig: $T(2) = 2T(n/2)+2$
$T(2) = 2T(1) + 2$
$T(2) = 2K + 2$

Is $2K+2 \le c2$? The guess is true for $c \ge K+1$ so we are done.

In the above example, it is common to "cheat" a little bit and instead assume that $T(1) = 1$ instead of $T(1) = q(1)$.

The reason is to simplify the analysis a bit. Let's look at what we would have if we used $T(1)=1$ instead and tried to solve for the base case:

Try $T(2) \le c2$
Orig: $T(2) = 2T(n/2)+2$
$T(2) = 2T(1) + 2$
$T(2) = 2(1) + 2$
$T(2) = 4$

Is $4 \le c2$? Yes, if $2 \le c$. We can still make the base case work for $T(1)= 1$. The difference is that there is a constant factor using the $T(1)=K$, that is missing when we assume $T(1)=1$. For a slightly easier analysis, we will often assume that $T(1)=1$ to simplify the process. However, remember that even if $T(1) = q(1)$, it is safe to assume $T(1)=1$ to prove the recurrence.

How to guess:

It is often useful to drop the asymptotically insignificant constants.

Ex: $T(n) = 2T(n/2 + 17) + n$

Ignore the constants, since for a large n they are insignificant. Guess O(nlgn)

Or, prove a loose upper and lower bounds, then try to reduce the bounds.
E.g., start with $O(n^3)$, $O(n^2)$, … assume only Ω (n) initially in recurrence.

Subtract Terms:

T(n)=T(n/2) + T(n/2) + 1

If you guess T(n) $\leq$ cn linear
Get T(n) $\leq$ cn/2 + cn/2 + 1
T(n) $\leq$ cn + 1
This is not $\leq$ cn for any value of n!

Instead try subtracting out the lower order term that is causing problems:

Try T(n) $\leq$ cn – b ; b>= 0
Get T(n) $\leq$ (cn/2)-b + (cn/2)-b + 1
T(n) $\leq$ cn – 2b + 1
T(n) $\leq$ cn –b ; if b>=1 then this works to satisfy the solution

Changing Variables:

Consider T(n) = 2T($\sqrt{n}$) + lgn ; looks hard!

Let m=lg n ; $n = 2^m$
Get $T(2^m) = 2T(n^{1/2}) + m$
$T(2^m) = 2T(2^{m/2}) + m$
Let $S(m) = T(2^m)$

→ S(m)=2S(m/2) + m ; We solved this already!

= O (m lgm) = O(lgn lglgn) ; This grows very slowly!

Second substitution example:

Given : T(n) = q (1) if n=1
T(n) = 4T(n/2) + n if n>1

Let's guess that the solution is O(n lgn)
So we need to show that T(n) $\leq$ cn lgn

Show: T(n) $\leq$ 4 (c(n/2)lg(n/2)) + n
T(n) $\leq$ 2cn lg(n) - 2cn lg2 + n
T(n) $\leq$ 2cn lgn - 2cn + n
T(n) $\leq$ 2cn lgn - (2c+1) n
This is NOT of the form cn lgn for any value of c! Must match the exact form of the induction hypothesis or the result is not valid. Additionally, this result is not $\leq$ cnlgn for all $n \geq n_0$.

New try: Guess solution is $O(n^2)$
So we need to show that T(n) $\leq$ cn^2

Show: $T(n) \leq 4(c(n/2)^2) + n$

$T(n) \leq 4c(n^2/4) + n$

$T(n) \leq cn^2 + n$

Does not work either! But its close.

Off by a factor of n, so lets try subtracting off that term.

Try: $T(n) \leq cn^2 - n$

$T(n) \leq 4((cn^2/4) - (n/2)) + n$

$T(n) \leq cn^2 - 2n + n$

$T(n) \leq cn^2 - n$

This is a match! c = any value > 0.

We also need to show the base case:

In our guess, $T(2) \leq 4c - 2$

In original, $T(2) = 4T(2/2) + 2$

$T(2) = 4+2$

$T(2) = 6$

So our guess is safe as long as c>1.

Be careful! The solution must match the EXACT form of the induction hypothesis.

Ex: Given $T(n) = 2T(n/2) + n$

Guess $T(n) \leq cn$

Show $T(n) \leq 2c(n/2) + n$

$T(n) \leq cn + n$

It looks like this works, since we have cn+n which is O(n) and something O(n) is ≤cn but it is not the exact same form as our guess. It has an extra n in there that the guess does not. (The ultimate solution may still be O(n) though! But this doesn't prove it. For this problem, a guess of $T(n) \leq c$ would work).