CMSC 351
Algorithms

Alex Reustle
Dr. Clyde Kruskal • Spring 2016 • University of Maryland
http://www.cs.umd.edu/class/spring2016/cmsc351-0101

Last Revision: May 9, 2016

Table of Contents

1 Maximum Subarray Problem

Given a sequence of integers, find the largest collection which is a sum of adjacent values.

$$3\ -5\ \underbrace{4\ 3\ -6\ 4\ 8}_{Sum=13}\ -5\ 3\ -7\ 2\ 1$$

Brute Force solution

Brute force solution: try every possible sum. (I'd guess it's n^3) Sum variable M set to zero. We will allow empty sums (no elements, sum = 0).

```
M ← 0;
for i=1 to n do
    for j=i to n do
        S ← 0;
        for k=i to j do
            S = S + A[k];
            M = max(M, S);
        end
    end
end
```

Algorithm 1: Matrix Subarray Brute Force

Insight: Loops in programs are like summation in Mathematics.(for i=1) to n == $\sum_{i=1}^{n}$
Nested loops are nested summations. $\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1$
Simplify summation using summation algebra from the inside out. $\sum_{i=1}^{n}\sum_{j=i}^{n} j-i+1$
Tangent: since j is the variable, i and 1 are constants. So it can be separated into two sums.

Analysis of the Algorithm

Actual progress: Change a variable to manage sum. Like integrals in calculus. Every technique in calculus for integrations are matched by similar techniques in summations. Change of variable by example.

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j} 1 &= \sum_{i=1}^{n}\sum_{j=i}^{n} j-i+1 && \text{By adding 1 } j-i \text{ times, and fenceposting}\\
&= \sum_{i=1}^{n}\sum_{j=1}^{n-i+1} j && \text{By change of variable Method}\\
&= \sum_{i=1}^{n}\frac{(n-i+1)(n-i+2)}{2} && \text{Gausses sum}\\
&= \frac{1}{2}\times\sum_{i=1}^{n}(n-i+1)(n-i+2) && \text{Expanding is error prone. Change variable instead}\\
&= \frac{1}{2}\times\sum_{n-i+1=1}^{n} i(i+1) && \text{substitute i=n-i+1}\\
&= \frac{1}{2}\times\sum_{i=1}^{n} i(i+1) && \text{simplify bounds}\\
&= \frac{1}{2}\times\frac{n(n+1)(n+2)}{3} && \text{magic.}\\
&= \frac{n(n+1)(n+2)}{6}
\end{aligned}$$

This first method is $\Theta(n^3)$

Methods to improve integer list summation algorithm.

Remember previous sums, do not recalculate known data.

```
M ← 0;
for i=1 to n do
    S ← 0;
    for j = i to n do
        S = S + A[i];
        M = max(M, S)
    end
end
```

Algorithm 2: n^2 Subarray

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

This new method is Θn^2.
Loosely speaking the maximum sum is just the previous maximum sum, plus the current value.

```
M ← 0, S ← 0;
for i = 1 to n do
    S ← max(S + A[i], 0);
    M = max(M, S);
end
```

Algorithm 3: Linear Subarray

Correctness of the algorithm stems from proof by induction on $S = \max(S + A[i], 0)$. This is the Loop Invariant.

$$\sum_{i=1}^{n} 1 = n$$

2 Timing analysis

Lower Bound
No algorithm is Faster than this boundary condition, the lower bound time.
Best Case scenario, very hard to prove in the general case.
Upper Bound
Some Algorithm can obtain upper bound time.
Worst case Scenario.

Classes of algorithms

$\Theta(n^2)$

Bubble sort
Insertion Sort
Selection Sort

$\Theta(n\log(n))$

Merge Sort
Heap Sort
Quick Sort

Others

Radix Sort

3 Bubble Sort Analysis

```
for i = n downto 2 do
    for j = 1 to i-1 do
        if $A[j] > A[j+1]$ then
            swap (A[j],A[j+1]);
        end
    end
end
```

Algorithm 4: Bubble Sort

Major Growth Elements of Bubble sort will be Comparisons and Exchanges.

Best case number of comparisons require you to view every element unconditionally.

$$\begin{aligned}\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 &= \sum_{i=2}^{n} i-1 \\ &= \sum_{i=1}^{n-1} (i+1)-1 \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)n}{2}\end{aligned}$$

Comparisons will always occur, regardless of state of list.

Best Case Exchanges occur when the list is sorted. Zero exchanges.
Worst Case Exchanges occur when the list is reverse sorted. Same number of exchanges as comparisons.

Definition 3.1 (Random)**.** Each permutation (ordering) of the list is equally likely.
Rotations of list permutations are equally likely to be in any position, but are not random because they ignore other possible permutations.

Average Case Analysis

Count Transpositions. Two elements that are out of order related to each other.

In the best case there are no transpositions.
In the worst case there are n choose 2 transpositions. Every element is out of order with the number of elements below it. So the number of transpositions is the number of unique pairs. Which is n choose 2 pairs.
In a randomized sample the permutations are equally likely to be out of order greater or lesser, so the number of average case exchanges will be half the number of comparisons. $\frac{n(n-1)}{4}$.

4 Insertion Sort

Every 0 to [i-1] index during the loop will be sorted.
ith term is stored in tmp. Values in the sorted subarray which are larger than the tmp value are brought forward individually until tmp > A[j].

```
$A[0] \leftarrow -\infty$;
for i = 2 to n do
    $t \leftarrow A[i]$;
    $j = i - 1$;
    while $A[j] \geq t$ do
        $A[j+1] \leftarrow A[j]$;
        $j--$;
    end
    $A[j+1] \leftarrow t$;
end
```

Algorithm 5: Insertion Sort

Analysing Number of Comparisons

Best Case Comparisons

In the case where the array is already correctly sorted, there will be only 1 comparison for each iteration of the other for loop. It will always evaluate false. So the number of best case comparisons is just the number of for loop iterations.

$$\begin{aligned}\sum_{i=2}^{n} 1 &= (n-2)+1 && \text{Already sorted}\\ &= n-1\end{aligned}$$

Worst Case Comparisons

In the worst case the array is reverse sorted, and every element must be moved. The while loop always decrements j to zero to compare against the sentinel value.

$$\begin{aligned}\sum_{i=2}^{n} i &= \left(\sum_{i=1}^{n} i\right) - 1 && \text{Reverse Sorted}\\ &= \frac{n(n+1)}{2} - 1 && \text{must remove i=1}\\ &= \frac{(n+2)(n-1)}{2}\end{aligned}$$

Average Case Comparisons

In the average case we must determine the probability that a given element will move. So for evaluation we want the expected value over the set $\sum_{x \in X} P(x) \cdot V(x)$.
Starting at location i, go until reach location j-1. Probability that 1 element (A[i]) will end up in any location in the subarray is $\frac{1}{i}$ Value is number of moves. Which is (i-j+1).

$$\begin{aligned}\sum_{i=2}^{n}\sum_{j=1}^{i}\frac{1}{i}\cdot(i-j+1) &= \sum_{i=2}^{n}\frac{1}{i}\sum_{j=1}^{i}(i-j+1) && \text{Pull out 1/i}\\ &= \sum_{i=2}^{n}\frac{1}{i}\sum_{j=1}^{i} j\\ &= \sum_{i=2}^{n}\frac{1}{i}\cdot\frac{i(i+1)}{2}\\ &= \sum_{i=2}^{n}\frac{(i+1)}{2}\\ &= \frac{1}{2}\left[\sum_{i=2}^{n} i+1\right]\end{aligned}$$

$$= \frac{1}{2}\left[\sum_{i=2}^{n} i + \sum_{i=2}^{n} 1\right]$$
$$= \frac{1}{2}\left[\sum_{i=1}^{n} i - 1 + (n-1)\right]$$
$$= \frac{1}{2}\left[\frac{(n)(n+1)}{2} - 1 + \frac{2(n-1)}{2}\right]$$
$$= \frac{1}{2}\left[\frac{(n^2+n-2)}{2} + \frac{2n-2}{2}\right]$$
$$= \frac{(n^2+3n-4)}{4}$$
$$= \frac{(n+4)(n-1)}{4}$$

Analyzing Exchanges

Best Case Exchanges

Best case number of moves $= 2n - 1$. There is 1 move in in the $-\infty$ assignment at the top, then in the for loop there are 2 moves per iteration, moving A[i] into t, and moving it back into the same spot. $1 + 2(n-1) = 2n - 1$.

Worst Case Exchanges

Worst case number of moves is 2 for each iteration of outer loop plus worst case number of comparisons, minus the time when the comparison is false at the end, of the inner loop, plus 1 for the top assignment. $2(n-1)+COMP-(n-1)+1$. Note the short cut, at each iteration we're doing one more move than comparison. So just easily take the value already derived for Worst case comparisons, and add the number of loop iterations, plus 1 for the sentinel assignment.

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

Average Case Exchanges

In the average case, whenever there's a comparison, there will always be a move, except the 1 time per iteration it evaluates false. So the analysis method is the same as for the worse case. Since we know the number.

$$\frac{(n+4)(n-1)}{4} + n$$

Remark 4.1. An important trick is to recognize that the moves are related to the comparisons 1 to 1 and that there will be 1 time when the comparison evaluates false each iteration, so you subtract that from the final analysis.

5 Insertion Sort without Sentinel

Add another comparison in the algorithm so the A[0] term is unnecessary, by checking that you haven't gone past i=1.

```
for i=2 to n do
    $t \leftarrow A[i]$;
    $j = i - 1$;
    while $j > 0 \wedge A[j] > t$ do
        $A[j+1] \leftarrow A[j]$;
        $j--$;
    end
    $A[j+1] \leftarrow t$;
end
```

Algorithm 6: Insertion Sort w/o Sentinel

A note about comparisons. We don't count index value comparisons, because index values can easily be stored in registers and won't have the same cost to check as array value comparisons.

Best Case Comparisons

In the best case, the array is already sorted, so there will just be 1 comparison per iteration of the outer loop. $\sum_{i=2}^{n} 1 = n - 1$

Worst Case Comparisons

In the worst case the array is reverse sorted, and every ith iteration of the loop must compare against all previous $(i-1)$ elements.

$$\begin{aligned}\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 &= \sum_{i=2}^{n} i - 1 \\ &= \sum_{i=2}^{n} i - \sum_{i=2}^{n} 1 \\ &= \frac{n(n+1)}{2} - 1 - (n-1) \\ &= \frac{n(n+1)}{2} - n \\ &= \frac{n^2+n}{2} - \frac{2n}{2} \\ &= \frac{n^2-n}{2} \\ &= \frac{n(n-1)}{2}\end{aligned}$$

Average Case Comparisons

Remark 5.1. HARMONIC SERIES and brick problem. How far out can you stack bricks on top of one another? Arbitrarily far out (but not infinitely far out)

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots \approx \ln n + 1$$

$$H_n - 1 = \sum_{i=2}^{n} \frac{1}{i} = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots \approx \ln n$$

Harmonic Series come up again and again because of the idea that something has a $\frac{1}{i}$ chance of happening in an iteration.

What are we saving when we don't have a sentinel? In the event that the ith element is the smallest element in the A[1..i-1] sub-array, we will descend all the way to the 1st index of the list. If that happens, we won't have the 1 extra step of comparing against the sentinel. So only when the ith element is the smallest seen thus far do we save 1 step. Probability that current ith element is smallest in sub array is $\frac{1}{i}$. The value when that occurs is 1. Thus over all n elements of the array, on average the sentinel would have cost us $\sum_{i=2}^{n} \frac{1}{i} = H_n - 1$ comparisons.
So average Case comparisons without sentinel is average case with sentinel minus average cost of sentinel. $\frac{(n-1)(n+4)}{4} - (H_n - 1) \approx \frac{(n-1)(n+4)}{4} - \ln n$.

Exchanges

Removing the sentinel from insertion sort adds no new exchanges, nor does it alter when a change might have been done already. Best, worst and average cases are all the same, therefore.

6 Selection Sort

Summary: Find biggest element, put at bottom of list, recurse for sub array.

```
for i=n downto 2 do
    $k \leftarrow 1$;
    for j=2 to i do
        if $A[j] > A[k]$ then
            $k \leftarrow j$;
        end
    end
    swap (A[k], A[i]);
end
```

Algorithm 7: Selection Sort

Best Case Comparisons

The comparison is always run, so the comparison value should be the same for each case.

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

Exchanges

Exchanges don't occur inside the conditional, so exchanges number is simply n-1.
How many times do we exchange a number with itself? How many times does i=k? $\sum 1/i = H_n$

7 Merge Sort

Divide and Conquer. Recursively split array down to 1, then merge sorted sublists. First call of Mergesort is (A,1,n)

```
MERGESORT (A,p,r);
//p and r are indecies in array defining sub array.;
if p<r then
    $q \leftarrow \lfloor \frac{p+r}{2} \rfloor$;
    MERGESORT (A,p,q);
    MERGESORT (A,q+1,r);
    MERGE (A, (p,q), (q+1,r));
end
```

Algorithm 8: Merge Sort

Merge Comparison analysis

Equal length arrays

Merge algorithm is linear over 2 sorted lists of same size. Total number of comparisons in the worst case during merge is $2n-1$, best case comparisons is n, where 1 array is strictly greater than the other.
Can't do better than $2n-1$ in worst case, because worst case situation is 2 lists with interleaved values. In this case the granularity of difference is on the scale of 1 value. So you must check every value to ensure against that.
This gives us very tight bounds and lets us know everything there is to know about merging.

Different length arrays

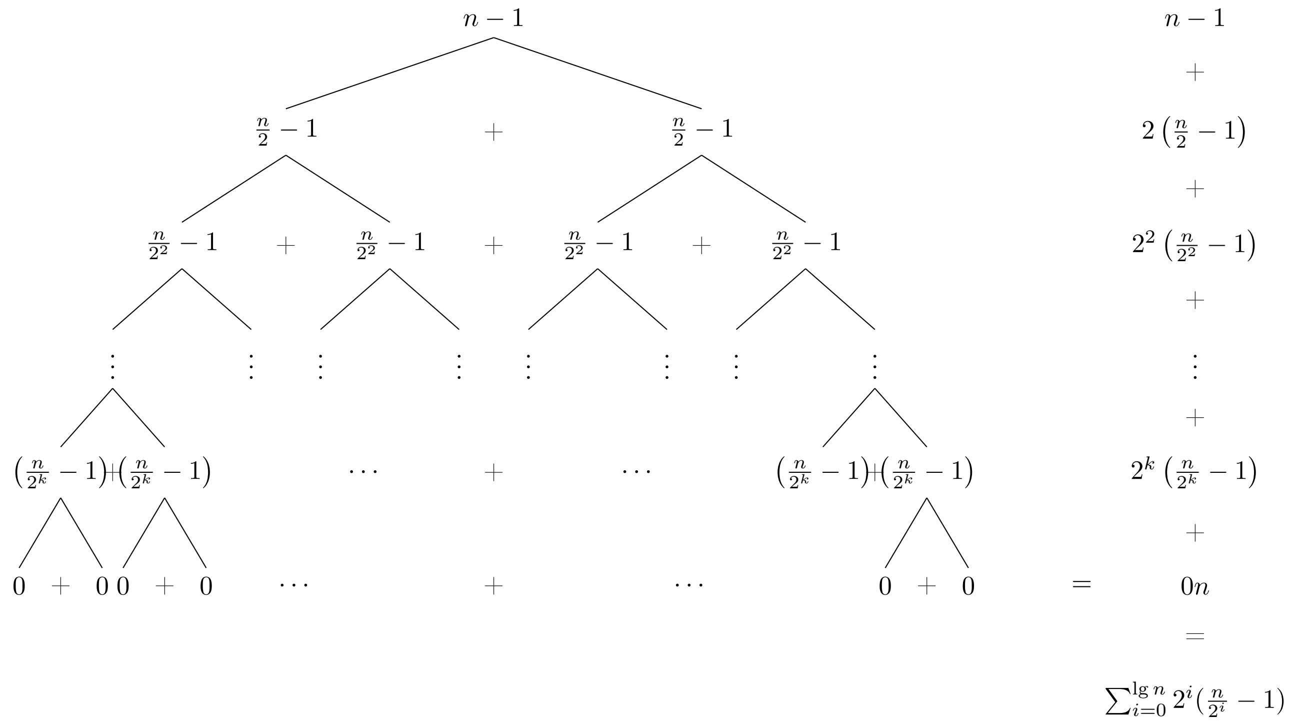
Arrays of size m and n where $m \leq n$ Worst case comparisons $= m + n - 1$
When $m << n$ you can get close to $\log n$ running time using binary search. But when m is close to n you are much closer to the $m + n - 1$ worst case.

Mergesort Comparison Analysis

Exact Number of Comparisons During a merge

$$(q - p + 1) + (r - (q + 1) + 1) - 1 = (r - p) \text{ comparisons}$$

starting at index p and ending at index q. So for the full list, where p=1 and q=n a MERGESORT will start with (n-1) comparisons and have the following behavior.



$n-1$
$\frac{n}{2}-1$ + $\frac{n}{2}-1$
$\frac{n}{2^2}-1$ + $\frac{n}{2^2}-1$ + $\frac{n}{2^2}-1$ + $\frac{n}{2^2}-1$
$\left(\frac{n}{2^k}-1\right)+\left(\frac{n}{2^k}-1\right)$ ⋯ + ⋯ $\left(\frac{n}{2^k}-1\right)+\left(\frac{n}{2^k}-1\right)$
0 + 0 0 + 0 ⋯ + ⋯ 0 + 0
$n-1$
+
$2\left(\frac{n}{2}-1\right)$
+
$2^2\left(\frac{n}{2^2}-1\right)$
+
+
$2^k\left(\frac{n}{2^k}-1\right)$
+
=
$0n$
=
$\sum_{i=0}^{\lg n} 2^i(\frac{n}{2^i}-1)$

Because this gives $\lg n$ terms, we can make it a little easier by dropping the bottom row, which sums to 0 anyway, because it's $0n$.

$$\begin{aligned}
\sum_{i=0}^{\lg n - 1} 2^i\left(\frac{n}{2^i} - 1\right) &= \sum_{i=0}^{\lg n - 1} (n - 2^i) \\
&= \sum_{i=0}^{\lg n - 1} n - \sum_{i=0}^{\lg n - 1} 2^i \\
&= (n \lg n) - (2^{\lg n - 1 + 1} - 1) && \text{geometric series} \\
&= (n \lg n) - (n - 1) \\
&= n \lg n - n + 1
\end{aligned}$$

Merge Sort Recurrence

When expressed as a recurrence relation, the mergesort algorithm given previously behaves like

$$\begin{aligned}
S(n) &= S\left(\frac{n}{2}\right) + S\left(\frac{n}{2}\right) + n - 1 \\
&= 2 \cdot S\left(\frac{n}{2}\right) + n - 1
\end{aligned}$$

Which gives the following recurrence relation

$$\begin{aligned} S(0) &= S(1) = 0 \\ S(n) &= S\left(\lfloor \frac{n}{2} \rfloor\right) + S\left(\lceil \frac{n}{2} \rceil\right) \end{aligned}$$

8 Integer arithmetic

Addition

Just like in elementary addition, the time to add 2 n digit numbers is proportional to the number of digits. $\Theta(n)$. Since each digit must be used to compute the sum this is the right order for best case time.

$$\begin{array}{rr} & {\scriptstyle 1\;1\;\;} \\ & 1\,2\,3\,4 \\ + & 5\,6\,7\,8 \\ \hline & 6\,9\,1\,2 \end{array}$$

We assume that each digital additions take constant time (with carry also, so ignore that little trouble spot) and label this constant α

Problem size

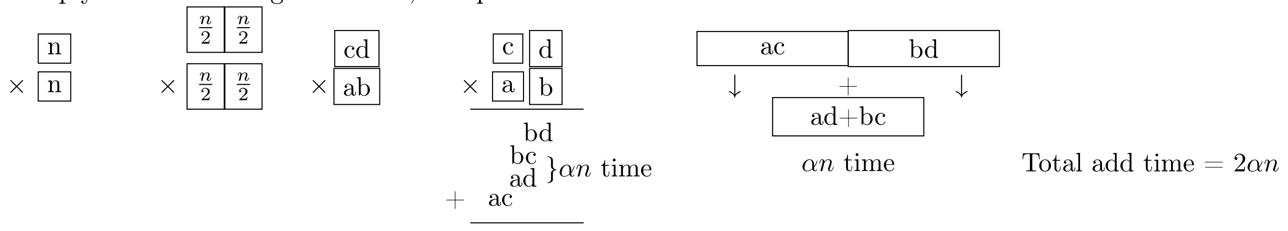
If we were attempting to determine if a number was prime we would divide it by consecutive primes up to the square root of that number. In Computer Science we want to approach all problems in terms of the problem size. So we should look at both in terms of the NUMBER OF DIGITS, and not the size of the number. So while the prime identification problem is $\Theta\sqrt{N}$ where N is the number, expressed in binary as 2^n, n being the number of binary digits. $\Theta(2^{\frac{n}{2}})$. Giving an exponential time algorithm.

Multiplication

Elementary school approach to multiplication runs in n^2 time because every digit on top is multiplied by every digit on the bottom.

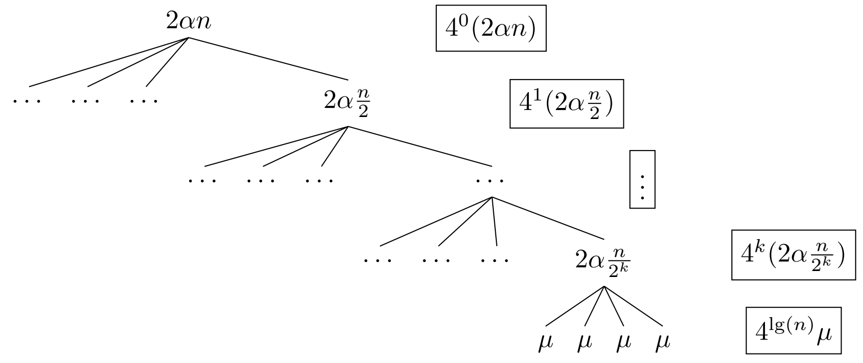
$$\begin{array}{rr} & 1\,2\,3\,4 \\ \times & 5\,6\,7\,8 \\ \hline & 9\,8\,7\,2 \\ & 8\,6\,3\,8\;\; \\ & 7\,4\,0\,4\;\;\;\; \\ & 6\,1\,7\,0\;\;\;\;\;\; \\ \hline & 7\,0\,0\,6\,6\,5\,2 \end{array}$$

By memorizing the numbers in large bases, say base 100, 2 digit decimal numbers can be treated as 1 digit numbers in base 100. That makes the multiplication easier, because there are fewer steps.
To generalize this for n digit numbers, treat them as base $10^{\frac{n}{2}}$ and cut each in half. Each has $\frac{n}{2}$ digits, and we know the base. Then, recurse through this method until we reach a base we really have memorized the table for. Then multiply them as 2 2 digit numbers, and pull out the answer.



ac is an n digit number, as are *ad*, *bc*, *bd*. When adding, *bd* is not shifted, but *ad*, and *bc* are shifted by $\frac{n}{2}$ digits, and *ac* is shifted by n digits. Since there's no overlap for *ac* and *bd*, you can add them simply by concatenation. $ad + bc$ takes αn time. That value plus the center (overlapping) segment of $ac + bd$ is also αn.

So without writing out the algorithm: $M(n) = 4M\left(\frac{n}{2}\right) + 2\alpha n$ which gives us a nice recursion to work on. Let's assume the time to multiply in the base case is μ. The base case is when both of our numbers are 1 digit long $M(1) = \mu$.



Which is fun, I'm having fun.

$$
\begin{aligned}
\text{atomic actions} &= \sum_{i=0}^{\lg(n)-1} (4^i(2\alpha\frac{n}{2^i})) + 4^{\lg(n)}\mu \\
&= 2\alpha n \sum_{i=0}^{\lg n - 1} \frac{4^i}{2^i} \cdots \\
&= 2\alpha n \sum_{i=0}^{\lg n - 1} 2^i \cdots \\
&= 2\alpha n \left(2^{\lg n - 1 + 1} - 1\right) && \text{geometric series} \\
&= 2\alpha n\,(n-1)\cdots \\
&\cdots && n^{\lg 4}\mu \\
&\cdots && n^2\mu \\
&= 2\alpha\left(n^2 - n\right) + \mu n^2
\end{aligned}
$$

We're doing n^2 multiplications and n^2 additions, so our application of recursion didn't improve the running time of our algorithm. There's a better way!

Better Multiplication

Because of the distributive law $(a+b)(c+d) = ac + ad + bc + bd$. So we can use this property to improve the fast multiplication algorithm. This will let us replace 1 recursive multiplications with 1 new addition and 1 subtraction, as

$$
\begin{aligned}
&\boxed{w = (a+b)(c+d)} && 2\cdot\tfrac{n}{2} \text{ additions, 1n multiplication} && \boxed{u}\boxed{v} \\
&= ac + ad + bc + bd && && + \boxed{z} \\
&u = ac \text{ 1n mult.} \\
&v = bd \text{ 1n mult.} \\
&\boxed{z = w - (u+v)} && \text{1n addition, and 1n subtraction.}
\end{aligned}
$$

we shall see.

By the same logic as the earlier multiplication algorithm, ac and bd are shifted by n bits, and can be concatenated. Let $w = (a+b)(c+d)$, and $z = w - (ac+bd) = (ad+bc)$ so to calculate z takes 3 Multiplications, 3 additions and 1 subtraction. Let's treat the subtraction as an addition, 4 additions. As such, the recursion for this algorithm is $T(n) = 3T\left(\frac{n}{2}\right) + 4\alpha n, T(1) = \mu$.

$$
\begin{aligned}
\sum_{i=0}^{\lg n - 1} 3^i\left(4\alpha\frac{n}{2^i}\right) + 3^{\lg n}\mu &= 4\alpha n \sum \frac{3^i}{2^i} && + n^{\lg 3}\mu \\
&= 4\alpha n \cdot \frac{\left(\frac{3}{2}\right)^{\lg n - 1 + 1} - 1}{\frac{3}{2} - 1} && \vdots
\end{aligned}
$$

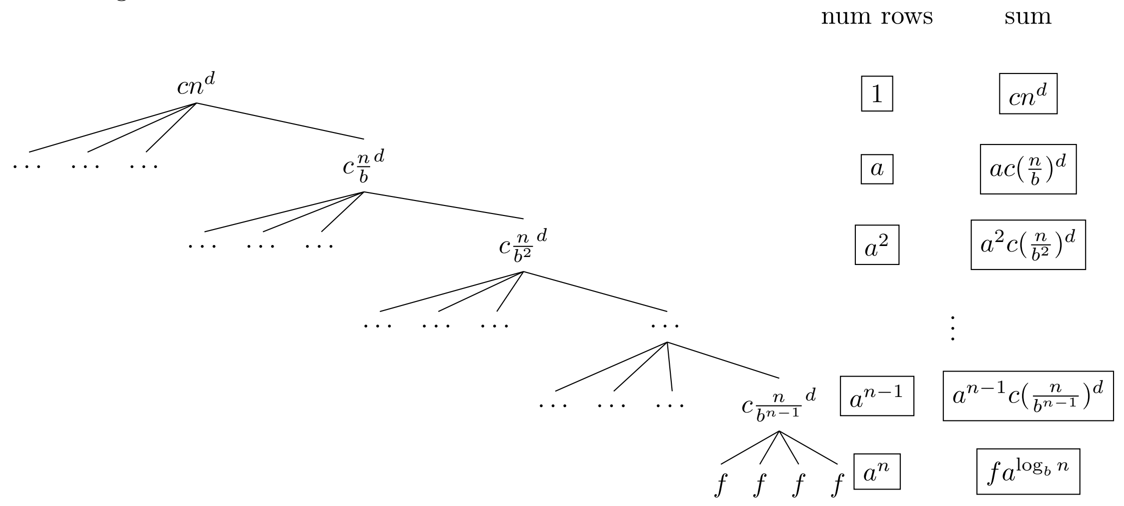
$$
\begin{aligned}
&= 4\alpha n \cdot \frac{\left(\frac{3}{2}\right)^{\lg n} - 1}{\frac{1}{2}} \\
&= 8\alpha n \left[n^{\lg\left(\frac{3}{2}\right)} - 1 \right] \\
&= 8\alpha n \left[n^{\lg 3 - \lg 2} - 1 \right] \\
&= 8\alpha n \left[\frac{n^{\lg 3}}{n^{\lg 2}} - 1 \right] \\
&= 8\alpha n \left[\frac{n^{\lg 3}}{n} - 1 \right] \\
&= 8\alpha \left[n^{\lg 3} - n \right] + n^{\lg 3}\mu \\
&\approx n^{\lg 3} \left[8\alpha + \mu \right] \\
&\approx n^{1.57} \left[8\alpha + \mu \right]
\end{aligned}
$$

This is $O(n^{1.57})$, a better result than the previous algorithm.

9 Recursion Master Formula

$$T(n) = aT(\frac{n}{b}) + cn^d,\ T(1) = f$$

Kruskal claims this is a better formula than the master theorem in the book, easier to use and more applicable. Let's solve using tree method.


num rows
sum
cn^d
$c\frac{n}{b}^d$
$c\frac{n}{b^2}^d$
$c\frac{n}{b^{n-1}}^d$
f f f f
$\boxed{1}$ $\boxed{cn^d}$
$\boxed{a}$ $\boxed{ac(\frac{n}{b})^d}$
$\boxed{a^2}$ $\boxed{a^2c(\frac{n}{b^2})^d}$
$\boxed{a^{n-1}}$ $\boxed{a^{n-1}c(\frac{n}{b^{n-1}})^d}$
$\boxed{a^n}$ $\boxed{fa^{\log_b n}}$

You will get $\log_b n$ levels, the branching factor is a.

$$
\begin{aligned}
\sum_{i=0}^{\log_b n - 1} a^i c\left(\frac{n}{b^i}\right)^d &= cn^d \sum_{i=0}^{\log_b n - 1} \frac{a^i}{{b^i}^d} && + fn^{\log_b a} \\
&= cn^d \sum_{i=0}^{\log_b n - 1} \frac{a^i}{b^{d^i}} && + \vdots \\
&= cn^d \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^d}\right)^i \\
&= cn^d \left(\frac{\left(\frac{a}{b^d}\right)^{\log_b n} - 1}{\frac{a}{b^d} - 1} \right)
\end{aligned}
$$

$$
\begin{aligned}
&= cn^d \left(\frac{n^{\log_b(\frac{a}{b^d})} - 1}{\frac{a}{b^d} - 1} \right) \\
&= cn^d \left(\frac{n^{\log_b a - \log_b b^d} - 1}{\frac{a}{b^d} - 1} \right) \\
&= cn^d \left(\frac{n^{\log_b a - d} - 1}{\frac{a}{b^d} - 1} \right) \\
&= \frac{cn^{\log_b a} - cn^d}{\frac{a}{b^d} - 1} + \vdots \\
&= \frac{cn^{\log_b a}}{\frac{a}{b^d} - 1} - \frac{cn^d}{\frac{a}{b^d} - 1} + fn^{\log_b a} \\
&= \left(\frac{c}{\frac{a}{b^d} - 1} + f \right) \cdot n^{\log_b a} - \frac{cn^d}{\frac{a}{b^d} - 1}
\end{aligned}
$$

This formula can't apply to the situation when the value of a geometric sum would be 1. The geometric sum isn't applicable in that case. SO this equation isn't applicable when $a = b^d$, we need a special case. When the r term = 1, a geometric series becomes a simple summation of 1, thus in this case if $a = b^d$ the formula becomes $cn^d \log_b n + fn^{\log_b a}$. In the special case of merge sort, we drop the constant -1 from the non recursion part of the term, and apply the special case, then we repeat the process with the n dropped, and add the two results.

$$
\begin{aligned}
&a \neq b^d && \left(\frac{c}{\frac{a}{b^d} - 1} + f \right) \cdot n^{\log_b a} - \frac{cn^d}{\frac{a}{b^d} - 1} \\
&a = b^d && cn^d \log_b n + fn^{\log_b a}
\end{aligned}
$$

The relative sizes of these constant factors will overwhelm in differing circumstances.

$$
\begin{aligned}
&a > b^d \rightarrow T(n) \approx \left(\frac{c}{\frac{a}{b^d} - 1} + f \right) \cdot n^{\log_b a} && = \Theta(n^{\log_b a}) \\
&a < b^d \rightarrow T(n) \approx \frac{cn^d}{1 - \frac{a}{b^d}} && = \Theta(n^d) \\
&a = b^d \rightarrow T(n) \approx cn^d \log_b n \quad \text{because } d = \log_b a && = \Theta(n^d(\log_b n))
\end{aligned}
$$

Applying to known algorithms

Multiplication

$$
\begin{aligned}
T(n) = 4T(\frac{n}{2}) &+ 2\alpha n, T(1) = \mu \\
&a = 4, b = 2, c = 2\alpha, d = 1, f = \mu \\
&\left(\frac{2\alpha}{2-1} + \mu \right) n^{\lg 4} - \frac{2\alpha n}{2-1} \\
&= (2\alpha + \mu) n^2 - 2\alpha n \approx \Theta(n^2) \\
&= 2\alpha \left(n^2 - n \right) + \mu n^2 \\
&\approx \Theta(n^2)
\end{aligned}
$$

Better Multiplication

$$
\begin{aligned}
T(n) = 3T(\frac{n}{2}) + 4\alpha n, T(1) = \mu \\
a = 3, b = 2, c = 4\alpha, d = 1, f = \mu \\
\left(\frac{4\alpha}{\frac{3}{2}-1} + \mu\right) n^{\lg 3} - \frac{4\alpha n}{\frac{3}{2}-1} \\
= (8\alpha + \mu)\, n^{1.57} - 8\alpha n \approx \Theta(n^{1.57})
\end{aligned}
$$

Mergesort

When we get

$$\begin{tikzcd}
 & T(n) = 2T(\frac{n}{2}) + n - 1, T(1) = 0 \arrow[dl, dash] \arrow[dr, dash] & \\
\text{n term: } a = 2, b = 2, c = 1, d = 1 \arrow[d, dash] & & \text{-1 term: } a = 2, b = 2, c = -1, d = 0 \arrow[d, dash] \\
a = b^d \arrow[d, dash] & & a \neq b^d \arrow[d, dash] \\
cn^d \log_b n \arrow[d, dash] & & \left(\frac{c}{\frac{a}{b^d}-1} + f\right) \cdot n^{\log_b a} - \frac{cn^d}{\frac{a}{b^d}-1} \arrow[d, dash] \\
1 \cdot n^1 \log_2 n = n \lg n & & \frac{-1}{\frac{2}{2^0-1}-1} n^{\lg 2} - \frac{-1 \cdot n^0}{\frac{2}{2^0}-1} \arrow[d, dash] \\
 & & = -n + 1
\end{tikzcd}$$

Implementation details

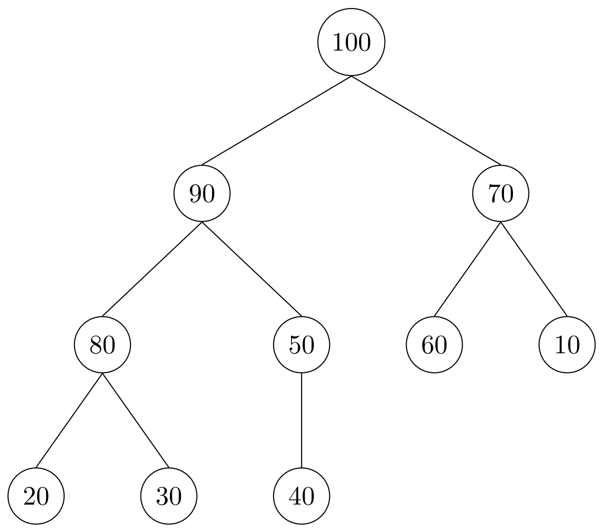
This gives us the incredible ability to use the constants of the recurrence into a running time and determine if it will be greater than another algorithm in constant time. There's also the Schonhage Strassen algorithm, which is $\Theta(n \log n \log \log n)$.
In real life, the size of T (1) is the word size of your machine, because the atomic multiplication is implemented at that level. So our atomic base is 2^{64} for most machines.

10 Heapsort

Created by J.W.J. Williams Improved by Robert Floyd

- Create Heap
- Finish

Definition 10.1 (Heap)**.** A binary tree where every value is larger than its children. Equivalently its descendants. In this class we require that all binary trees are full binary trees.



Create Heap

Turn a tree into a heap.
The traditional way to create a heap is to insert at the end of the array and sift up. Robert Floyd created a better way to create the heap. Treat the tree as a recursive heap: each parent is the parent to 2 heaps, and sift it from the bottom up. Create heap in left, create heap on right, then sift root down, and move up a level.

Heap Creation Analysis

In a binary tree, most nodes are near bottom, so doing bottom up technique most work is done near bottom, and because you're near the bottom you don't have far to go down. So we take advantage of that fact: Most elements are near the bottom and don't have far to go.
There are $\frac{n}{2}$ leaves. Each doing 0 comparisons.
There are $\frac{n}{4}$ parents of leaves. Each doing 2 comparisons.
There are $\frac{n}{8}$ grandparents of leaves. Each doing 4 comparisons (2 per level).
There are $\frac{n}{16}$ greatgrandparents of leaves. Each doing 6 comparisons (2 per level).

$$\begin{aligned}
\frac{n}{2}\cdot 0+\frac{n}{4}\cdot 2+\frac{n}{8}\cdot 4+\frac{n}{16}\cdot 6+\frac{n}{32}\cdot 8+\frac{n}{64}\cdot 10+\cdots \\
&= n\cdot\left[\frac{1}{2}+\frac{2}{4}+\frac{3}{6}+\frac{4}{16}+\frac{5}{32}+\cdots\right] \quad \text{This sum is:} \\
&= \frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\cdots = 1 \\
&\quad +\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\cdots = \frac{1}{2} \\
&\quad +\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\cdots = \frac{1}{4} \\
&\quad +\frac{1}{16}+\frac{1}{32}+\cdots = \frac{1}{8} \\
&\quad +\vdots \\
&= 2n
\end{aligned}$$

So Heap creation does $\Theta(2n)$ comparisons

Finish

- Put root node at the bottom of the array, as it must be the largest element, so it will go at the end of the sorted array.

- Then take the bottom right hand leaf and move to a tmp space.
- Then sift, reordering the tree and put the tmp leaf in its proper spot.
- Repeat.

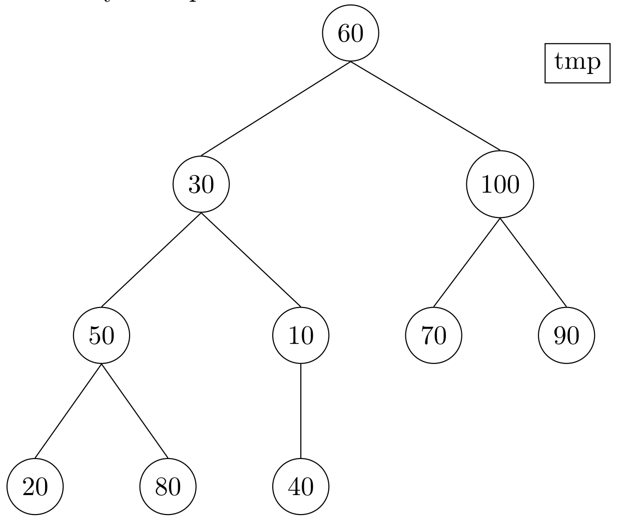
Sift comparisons: each level has 2 comparisons, child and tmp. There are $\approx \lg n$ levels. Total for a sift $\approx 2\lg n$ This must be done for each element in the tree, so our heap has an upper bound of $\approx 2n\lg n$.
But the heap shrinks, each iteration removes an element. So let's sum 0 to n-1 (because we remove an element first before sifting).

$$\begin{aligned}
\sum_{i=0}^{n-1} 2\lg(i+1) &\approx 2\sum_{i=1}^{n} \lg i \\
&= 2\left[\lg 1 + \lg 2 + \lg 3 + \cdots + \lg n\right] \\
&= 2\lg(1\cdot 2\cdot 3\cdot \cdots \cdot n) \\
&= 2\lg(n!) && \text{Stirling's Formula} \quad n! \approx \left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n} \\
&= 2\lg\left[\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}\right] \\
&\approx 2\left[n\lg(\frac{n}{e}) + \lg((2\pi n)^{\frac{1}{2}})\right] \\
&= 2\left[n\left[\lg n - \lg e\right] + \frac{1}{2}\lg(2\pi n)\right] \\
&= 2n\lg n - 2n\lg e + \lg n + \lg(2\pi) \\
&= 2n\lg n + O(n)
\end{aligned}$$

This isn't much better, it makes sense that it's not much better than our conservative guess, because in a full binary tree, half the elements are leaves. So while the tree shrinks, it doesn't shrink very fast, asymptotically you're still doing $\Theta(2n\lg n)$ comparisons, plus some linear term.

Implementation Details

Use array to implement tree structure.



1	60
2	30
3	100
4	50
5	10
6	70
7	90
8	20
9	80
10	40

Node has index i

Left child: $2i$

Right child: $2i+1$

Parent: $\lfloor \frac{i}{2} \rfloor$

CREATE The First parent is at index $\lfloor \frac{n}{2} \rfloor$. Start there and sift down during heap creation. Siblings can be reached by adding or subtracting 1. Result is a created heap.
FINISH Pop bottom into tmp first, then move heap root into bottom spot.

Optimization

Why is this result still worse than merge sort? $\Theta(2n \lg n)$ vs $\Theta(n \lg n)$. We're comparing tmp against both children and this doubles our number of comparisons. Instead let's sift the hole left by the root down to the bottom in $\lg n$ comparisons (1 per level) then put tmp in the hole an sift it back up into position.
How far will tmp need to go to sift back up and re-form the heap? Not far, since $1/2$ the nodes are in the bottom layer, and $1/4$ of the nodes are in the 2nd, etc. It takes 1 comparison to confirm tmp belongs in the bottom layer, 2 to check the 2nd layer up...

$$\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \frac{1}{16} \cdot 4 + \cdots = 2$$

Giving 2n comparisons on average. We can further optimize by binary searching up. This gives Heapsort $n \lg n + n \lg \lg n = \Theta(n \lg n)$ performance.

Pseudocode

```
Function Heap_Sort(A,n) is
    // Create Heap
    for $r = \lfloor \frac{n}{2} \rfloor$ to 1 do
        Sift(r,n,A[r])
    end
    // Finish Sort
    for $m = n$ To 2 do
        $s \leftarrow A[m]$
        $A[m] \leftarrow A[1]$
        Sift(1,m-1,s)
    end
end
Function Sift(r,n,s) is
    // r:root index, n:size index, s:sift value, p:parent index, c:child index
    $p \leftarrow r$
    while $2p \leq n$ do
        if $2p < n$ then
            if $A[2p] \geq A[2p+1]$ then
                $c \leftarrow 2p$
            else
                $c \leftarrow 2p+1$
            end
        else
            $c \leftarrow 2p$
        end
        if $A[c] > s$ then
            $A[p] \leftarrow A[c]$
            $p \leftarrow c$
        else
            Break
        end
    end
    $A[p] \leftarrow s$
end
```

Algorithm 9: Heap Sort

11 Finding Bounds to summation functions

Mathematical Preliminaries

Geometric Series

For $|r| \leq 1$ recall:

$$\sum_{i=0}^{\infty} r^i = \frac{1}{1-r} \qquad \text{Infinite Geometric Series}$$

$$\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r} = \frac{r^n-1}{r-1} \qquad \text{Finite Geometric Series}$$

How do we solve this? Or at least get a reasonable estimate?

$$\begin{aligned}
\sum_{i=0}^{\infty} i \cdot r^i &= \sum_{i=1}^{\infty} i \cdot r^i \\
&= r \cdot \sum_{i=1}^{\infty} i \cdot r^{i-1} \\
&= r \cdot \sum_{i=1}^{\infty} \left(\int i r^{i-1} \right)' && \text{sum of derivatives is the derivative of the sum} \\
&= r \cdot \sum_{i=1}^{\infty} \left(r^i \right)' \\
&= r \left(\sum_{i=1}^{\infty} r^i \right)' \\
&= r \left(\frac{1}{1-r} - 1 \right)' \\
&= r \left(\frac{1}{(1-r)^2} \right) \\
&= \frac{r}{(1-r)^2}
\end{aligned}$$

How can we check to confirm? We've already calculated the $r = 1/2$ case, where the infinite sum is 2.

$$\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$$

Now for the finite series.

$$\begin{aligned}
\sum_{i=0}^{n-1} i \cdot r^i &= \sum_{i=1}^{n-1} i \cdot r^i \\
&= r \cdot \sum_{i=1}^{n-1} i \cdot r^{i-1} \\
&= r \cdot \sum_{i=1}^{n-1} \left(\int i r^{i-1} \right)' && \text{sum of derivatives is the derivative of the sum} \\
&= r \cdot \sum_{i=1}^{n-1} \left(r^i \right)'
\end{aligned}$$

$$
\begin{aligned}
&= r\left(\sum_{i=1}^{n-1} r^i\right)' \\
&= r\left[\left(\frac{r^n-1}{r-1}-1\right)'\right] \\
&= r\left[\frac{nr^{n-1}(r-1)-(r^n-1)}{(r-1)^2}\right] \\
&= r\left[\frac{nr^n-nr^{n-1}-r^n+1}{(r-1)^2}\right] \\
&= r\left[\frac{(n-1)r^n-nr^{n-1}+1}{(r-1)^2}\right] \\
&= \frac{(n-1)r^{n+1}-nr^n+r}{(r-1)^2}
\end{aligned}
$$

Gauss's Sum

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Without knowing the answer already, we can see an easy upper bound. If every term is bound by it's largest element it can't be larger than n^2. We can also get a lower bound by flooring everything to the lowest value of 1, since there are n of them we know that the result must be larger than n. $n \le sum \le n^2$.
Next we can try to split the sum in half and floor both to the lowest term in that series.

$$
\begin{aligned}
SUM &= 1+2+3+4+\cdots+\frac{n}{2}+\frac{n}{2}+1+\cdots+n \\
&\ge 1+1+1+1+\cdots+1 \quad + \quad \frac{n}{2}+1+\cdots+\frac{n}{2}+1 \\
&\ge \frac{n}{2}+\frac{n}{2}\left(\frac{n}{2}+1\right) \\
&\ge \frac{n}{2}\left[1+\frac{n}{2}+1\right] \\
&\ge \frac{n}{2}\left[\frac{n+4}{2}\right] \\
&\ge \frac{n^2}{4}+n
\end{aligned}
$$

Harmonic Sum

$$H_n = \sum_{i=1}^{n} \frac{1}{i} = 1+1/2+1/3+1/4+1/5+\cdots+1/n$$

$\approx \ln n$

We can approximate it using the same technique we applied to the Gaussian Sum

$\le 1+[1/2+1/2]+[1/4+1/4+1/4+1/4]+\cdots$

$= 1+1+1+\cdots+1$ num ones = num groups

$$= \sum_{i=0}^{k-1} 2^i = n$$

k is the number of groups. The number of

$$\begin{aligned}
&\text{items per group, summed, which must sum to n}\\
&= 2^k - 1 = n\\
&= 2^k = n+1\\
&= k = \lg(n+1)\\
H_n &\leq \lg(n+1)
\end{aligned}$$

Using continuous math to solve discrete math

Assume a function that's bounded from m to n. The Riemann sum of areas in discrete terms is bounded by the integral of some function.

$$\int_{m-1}^{n} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m}^{n+1} f(x)dx$$

Provided $f(x)$ is increasing.

$$\int_{m}^{n+1} f(x)dx \leq \sum_{i=m}^{n} f(i) \leq \int_{m-1}^{n} f(x)dx$$

Provided $f(x)$ is decreasing.

Gauss's Sum

$$\begin{aligned}
\int_{m-1}^{n} xdx \leq \sum_{i=m}^{n} i &\leq \int_{m}^{n+1} xdx\\
\left.\frac{x^2}{2}\right|_0^n \leq \quad &\leq \left.\frac{n^2}{2}\right|_i^{n+1}\\
\frac{n^2}{2} - \frac{0^2}{2} \leq \quad &\leq \frac{(n+1)^2}{2} - \frac{1^2}{2}\\
\frac{n^2}{2} - 0 \leq \quad &\leq \frac{n^2+2n+1}{2} - \frac{1}{2}\\
\frac{n^2}{2} \leq \quad &\leq \frac{n^2+2n}{2}\\
\frac{n^2}{2} \leq \quad &\leq \frac{n(n+2)}{2}
\end{aligned}$$

Harmonic Sum

$$\begin{aligned}
\int_{m}^{n+1} \frac{1}{x}dx \leq \sum_{i=m}^{n} \frac{1}{i} &\leq \int_{m-1}^{n} \frac{1}{x}dx\\
\ln x\Big|_1^{n+1} \leq \quad &\leq \ln(x)\Big|_0^n\\
\ln(n+1) - \ln(1) \leq \quad &\leq \ln(n) - \ln 0\\
\ln(n+1) - 0 \leq \quad &\leq \ln(n) - (-\infty)\\
\ln(n+1) \leq \quad &\leq \ln n + \infty\\
&\leq +\infty
\end{aligned}$$

Less than infinity isn't helpful. Let's avoid it by removing the 1 term from the sum, so we don't integrate 0 to 1 for ln x

$$\leq 1 + \int_{1}^{n} f(x)dx$$

$$
\begin{aligned}
&\le 1 + \ln(x)\Big|_1^n \\
&\le 1 + \ln n - \ln 1 \\
&\le 1 + \ln n - 0 \\
&\le \ln(n) + 1
\end{aligned}
$$

12 Order Notation

Did you know that some algorithms can be faster than others? It's true!
https://en.wikipedia.org/wiki/Time_complexity

Name	Running time $T(n)$	Examples of running times
Constant	$O(1)$	10
Iterated logarithmic	$O(\log^* n)$	
Log-logarithmic	$O(\log\log n)$	
Logarithmic	$O(\log n)$	$\log n, \log(n^2)$
Polylogarithmic	$poly(\log n)$	$(\log n)^2$
Fractional power	$O(n^c)$ where $0 < c < 1$	$n^{1/2}, n^{2/3}$
Linear	$O(n)$	n
N log star n	$O(n \log^* n)$	
Linearithmic	$O(n \log n)$	$n \log n, \log n!$
Quadratic	$O(n^2)$	n^2
Cubic	$O(n^3)$	n^3
Polynomial	$2^{O(\log n)} = poly(n)$	$n, n\log n, n^{10}$
Quasi-polynomial	$2^{poly(\log n)}$	$n^{\log\log n}, n^{\log n}$
Exponential (with Linear exponent)	$2^{O(n)}$	$1.1^n, 10^n$
Exponential	$2^{poly(n)}$	$2^n, 2^{n^2}$
Factorial	$O(n!)$	$n!$
Double exponential	$2^{2^{poly(n)}}$	2^{2^n}

$$
\begin{aligned}
\text{Example:}\qquad & 17n^3 + 24n^2 - 6n + 8 \\
=& 17n^3 + O(n^2) \\
=& 17n^3 + o(n^3) \\
\approx& 17n^3 \\
=& \Theta n^3
\end{aligned}
$$

Order notation can hide simple truths like a large constant in a lower order term that can seriously affect the running time for small n. Little o means the function is always less than what's in the bounds. $o(n^3)$ could be $O(n^{2.9998})$ which could throw off all of our calculations.

Equivalence Function	Order Set
$=$	Θ
$\leq$	O
$\geq$	Ω
$<$	o
$>$	ω

$$\begin{aligned} &\text{Informally} \quad f(n) = \Theta(g(n)) \\ &\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0 \\ &\lim_{n\to\infty} \frac{7n^2+3n-8}{4n^2+20n+4} = \frac{7}{4} \end{aligned}$$

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &= c \geq 0 \longrightarrow f(n) \in O(g(n)) \\ \lim_{n\to\infty} \frac{g(n)}{f(n)} &= c \geq 0 \longrightarrow f(n) \in \Omega(g(n)) \\ \lim_{n\to\infty} \frac{f(n)}{g(n)} &= 0 \longrightarrow f(n) \in o(g(n)) \\ \lim_{n\to\infty} \frac{g(n)}{f(n)} &= 0 \longrightarrow f(n) \in \omega(g(n)) \end{aligned}$$

What about non-polynomial functions? Like $f(n) = n^2 \cdot (10 + \sin(n))$ The book has a non-limiting approach which gives us a nice way of addressing oscillating functions.
To a first approximation you can do plain algebra using Theta notation. $2^{\Theta n^2} \cdot 2^{\Theta n^3} = 2^{\Theta n^2 + \Theta n^3}$.
What's faster? $n^{\lg n}$or$(\lg n)^n$
Exponentials grow much faster than polynomials. How do we state that formally?

$$n^a = o(b^n) \text{ for } a \geq 0, b > 1$$

$$(\log n)^a = o(n^b) \;\; b > 0$$

$$(\log \log n)^a = o((\log n)^b)$$

$$\begin{aligned} &n^{\lg n} \;\; ? \;\; (\lg n)^n \\ &\lim_{n\to\infty} \frac{n^{\lg n}}{(\lg n)^n} \\ &= \lim_{n\to\infty} \frac{2^{\lg n \cdot \lg n}}{2^{\lg (\lg n)^n}} \\ &= \lim_{n\to\infty} \frac{2^{\lg^2 n}}{2^{n \lg(\lg n)}} \\ &= 0 \\ \therefore\; &n^{\lg n} = o((\lg n)^n) \end{aligned}$$

Polylogs grow slower than polynomials, and Quasi-polynomials grow slower than exponentials with variable bases.

13 Quicksort

An efficient sorting algorithm developed by Tony Hoare in 1959 while trying to translate Russian. Recursively partition the array, put small numbers to the left and large numbers to the right.

```
Function QUICKSORT(A,p,r) is
    if $p < r$ then
        $q \leftarrow$ PARTITION(A,p,r)
        QUICKSORT(A,p,q-1)
        QUICKSORT(A,q+1,r)
    end
end
Function PARTITION(A,p,r) is
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i \leftarrow i + 1$
            Exchange $A[i], A[j]$
        end
    end
    $i \leftarrow i + 1$
    Exchange $A[i], A[r]$
    return i
end
```

Comparison Analysis

First note that PARTITION is always linear in comparisons on $n = r - p + 1$ terms. The FOR loop runs $n - 1$ comparisons regardless of the result of the IF condition.

Quicksort Worst case comparisons

The worst case output for PARTITION must occur when the pivot doesn't move, and the recursive call to QUICKSORT is only 1 smaller than the previous call. Thus, partitioning on n elements gets n-1 comparisons. Worst Case Comparisons Occurs when pivot is at far end of array:

$$\begin{aligned} T(n) &= T(n-1) + n - 1 \quad , \quad T(1) = 0 \\ &= (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 \qquad\qquad \text{pivot} = 2 \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1) \cdot n}{2} \end{aligned}$$

Quicksort Best case comparisons

The best case occurs when PARTITION moves the pivot to the middle of the array, so that the recursive calls to QUICKSORT are $\frac{n-1}{2}$ smaller than the last call. Best Case Comparisons Occurs when pivot is right in the middle of array:

$$\begin{aligned} T(n) &= 2 \cdot T(\frac{n-1}{2}) + n - 1 \quad , \quad T(1) = 0 \\ &\leq 2 \cdot T(\frac{n}{2}) + n - 1 \\ &= n \cdot \lg(n) - n + 1 \end{aligned}$$

$$= O(n \cdot \lg n)$$

Proof by Constructive Induction

Mathematical Induction is great at proving an answer is true if you already know it. But if you don't know that what the answer is for sure, you can make an educated guess, and use induction to derive the right answer. We know the answer to the worst case comparisons of quicksort is quadratic, but we don't know the coefficients.
For example, let's prove $1 + 2 + 3 + \cdots + n$ is quadratic.

$$\sum_{i=1}^{n} i \text{ Guess} \sum_{i=1}^{n} i = an^2 + bn + c$$

Base case $i = 1$. $\sum_{i=1}^{1} i = a(1)^2 + b(1) + c \quad = a + b + c \quad = 1$
Inductive Hypothesis: $\sum_{i=1}^{n-1} i = a(n-1)^2 + b(n-1) + c$

$$\begin{aligned}\sum_{i=1}^{n} i &= n + \sum_{i=1}^{n-1} i \\ &= n + a(n-1)^2 + b(n-1) + c \\ &= n + a(n^2 - 2n + 1) + b(n-1) + c \\ &= an^2 + (-2a + b + 1)n + a - b + c\end{aligned}$$

If our assumption was true then we should get a system of simultaneous equations matching the quadratic. Then we could solve the system for the coefficients of the result.

$$\begin{aligned}a &= a \\ b &= -2a + b + 1 \\ c &= a - b + c \\ a &= \frac{1}{2}, a = b, c = 0\end{aligned}$$

$$\sum_{i=1}^{n} i = \frac{1}{2}n^2 + \frac{1}{2}n = \frac{n(n+1)}{2}$$

Quicksort Worst Case from Recurrence

$T(n) = T(n-1) + n - 1,\ \ T(1) = 0$. Let's guess that the result is quadratic: $an^2 + bn + c$
Base: $n = 1, a(1)^2 + b(1) + c = 0$
Inductive Hypothesis: Assume $T(n-1) = a(n-1)^2 + b(n-1) + c$

$$\begin{aligned}T(n) &= T(n-1) + n - 1 \\ &= a(n-1)^2 + b(n-1) + c + n - 1 \\ &= a(n^2 - 2n + 1) + b(n-1) + c + n - 1 \\ &= an^2(-2a + b + 1)n + c - b - 1 + a \\ &\text{Find coefficients} \\ a + b + c &= 0 \\ a &= a \\ -2a + b + 1 &= b \\ c - 1 + a - b &= c \\ a &= \frac{1}{2} \\ b &= -\frac{1}{2} \\ c &= 0\end{aligned}$$

$$\begin{aligned} T(n) &= \frac{1}{2}n^2 - \frac{1}{2}n \\ &= \frac{n(n-1)}{2} \end{aligned}$$

Quicksort Best Case from Recurrence

$T(n) = 2T(n/2) + n - 1,\ \ T(1) = 0$. Guess upper bound: $T(n) = an \lg n$
Base: $n = 1, a(1)\lg(1) = 0$
Inductive Hypothesis: By strong induction, assume $T(k) = ak \lg k$ for all $k < n$

$$\begin{aligned} T(n) &= 2T(n/2) + n - 1 \\ &\le 2\left[a \cdot \frac{n}{2} \cdot \lg \frac{n}{2}\right] + n - 1 \\ &= an[\lg n - \lg 2] + n - 1 \\ &= an[\lg n - 1] + n - 1 \\ &= an \lg n - an + n - 1 \\ &= an \lg n + (-a+1)n - 1 \\ \text{To be true, must be } &\le an \lg n \\ &\text{drop -1 because } < \\ &(-a+1)\text{must be } \le 0, \rightarrow a \ge 1 \end{aligned}$$

So a constant a greater than or equal to 1 solves the recurrence, but the best case value occurs when a is 1

$$\begin{aligned} T(n) &\le 1 \cdot n \lg n \\ &= n \lg n \end{aligned}$$

Average case analysis of quicksort from approximate recurrence.

Best case occurs when pivot is in the middle, while worst case occurs when pivot is at the end. Let's approximate the average case happening when the pivot is at the one and three quarter marks.

$$T(n) = T(\frac{3n}{4}) + T(\frac{n}{4}) + n - 1 \quad ,T(1) = 0$$

Let's guess optimistically that our upper bound is $T(n) \le an \lg n$
Base case n=1: $a(1)\lg(1) = 0 \le 0$✓
Inductive Hypothesis: By strong induction, assume true for $k < n\ \ T(k) \le ak \lg k$

$$\begin{aligned} T(n) &= T(\frac{3n}{4}) + T(\frac{n}{4}) + n - 1 \\ &\le a\frac{n}{4} \lg\left(\frac{n}{4}\right) + a\frac{3n}{4} \lg\left(\frac{3n}{4}\right) + n - 1 \\ &= a\frac{n}{4}(\lg n - \lg 4) + \frac{3an}{4}(\lg n - \lg 4 + \lg 3) + n - 1 \\ &= a\frac{n}{4} \lg n - \frac{an}{2} + \frac{3an}{4} \lg n - \frac{3an}{2} + \lg 3 \cdot \frac{3an}{4} + n - 1 \\ &= an \lg n + \left(-\frac{a}{2} - \frac{3a}{2} + \frac{a \lg 3}{4} + 1\right) n - 1 \\ &= an \lg n + \left(-2a + \frac{a}{4} \lg 3 + 1\right) n - 1 \\ \text{To be true, must be } &\le an \lg n \\ &-2a + \frac{a}{4} \lg 3 + 1 \le 0 \end{aligned}$$

$$
\begin{aligned}
(\frac{\lg 3}{4} - 2)a &\geq 1 \\
a &\geq \frac{1}{2 - \frac{3\lg 3}{4}} \\
T(n) &\geq \left(\frac{1}{2 - \frac{3\lg 3}{4}}\right) n\lg n \\
T(n) &\approx 1.23 \cdot n\lg n
\end{aligned}
$$

Average case analysis of quicksort from exact recurrence.

When we choose a pivot element, the pivot will end up at some index q. We don't know where q is yet. Since we will be iterating over the whole array, the probability of choosing a particular index is 1/n. Then you have to do the group on the left and the group on the right. And do it for all n indices. Solving this will require everything we know to do.

$$
\begin{aligned}
T(n) &= \sum_{q=1}^{n} \frac{1}{n}(T(q-1) + T(n-q)) + n - 1 \\
&= \frac{1}{n}\sum_{q=1}^{n}(T(q-1) + T(n-q)) + n - 1 \\
&= \frac{1}{n}\sum_{q=1}^{n}(T(q-1)) + (\frac{1}{n}\sum_{q=1}^{n} T(n-q)) + n - 1
\end{aligned}
$$

first is sum of Ts from 0 to n-1 and the other is the downward sum from n-1 to 0. So we get two sums of T

$$
= \frac{2}{n}\sum_{q=0}^{n-1} T(q) + n - 1
$$

now apply strong constructive induction, guess $an\lg n$

$$
\begin{aligned}
&\text{Base n} = 1 a(1)\lg(1) = 0 \checkmark \\
&\text{I.H. For } 1 \leq k < n \ \ T(k) \leq ak\lg k \\
T(n) &= \frac{2}{n}\sum_{q=1}^{n-1} T(q) + n - 1 \\
&\leq \frac{2}{n}\sum_{q=1}^{n-1} aq\lg q + n - 1 \\
&= \frac{2a}{n}\sum_{q=1}^{n-1} q\lg q + n - 1 \\
&\approx \frac{2a}{n}\int_{1}^{n} x\lg x dx + n - 1 \\
&= \frac{2a}{n}\left[\frac{x^2\lg x}{2} - \frac{x^2\lg e}{4} + c\right]\Big|_1^n + n - 1 \\
&= \frac{2a}{n}\left[\frac{n^2\lg n}{2} - \frac{n^2\lg e}{4} + c - (\frac{1^2\lg 1}{2} - \frac{1^2\lg e}{4} + c)\right] + n - 1 \\
&= \frac{2a}{n}\left[\frac{n^2\lg n}{2} - \frac{n^2\lg e}{4} + \frac{\lg e}{4}\right] + n - 1 \\
&= an\lg n - \frac{an\lg e}{2} + \frac{a\lg e}{2n} + n - 1
\end{aligned}
$$

$$
\begin{aligned}
&= an\lg n + (\frac{a\lg e}{2}+1)n - 1 + \frac{a\lg e}{2n} \\
\text{need } - &\frac{a\lg e}{2} + 1 \le 0 \\
a &\ge \frac{2}{\lg e} = 2\ln 2 \approx 1.39
\end{aligned}
$$

choose a = 1.39 and the last term becomes a number always less than or equal to 1, which is dominated by minus 1

$$
\begin{aligned}
&\text{and the second highest term becomes 1 minus 1=0} \\
T(n) &\le 1.39 n\lg n \\
&= \frac{2}{\lg e} n \lg n \\
&= 2n\ln n
\end{aligned}
$$

New Book method

Given What is the probability that the smallest and largest elements are compared? $(x_1, x_n)?\frac{2}{n}$ If you pick the largest or smallest as the pivot, then it will compare against every other element including the other extreme. But if you don't pick an extreme the two will be partitioned away and never be compared. So on average the number of comparisons you get from comparing the smallest and largest is also $\frac{2}{n}$.
What is the probability that $x_i, x_j : (i < j)$ will be compared? If you choose a pivot less than i, the two will end up on the same pivot together on the large side. If you choose a pivot greater than j, the two will end up on the small side together. The only way not to compare the two is to choose a pivot in between i and j. That probability is $\frac{2}{j-i+1}$
If you have 2 elements next to one another in the sorted array, they will always be compared. Otherwise how will you know? $\frac{2}{(i+1)-i+1} = 1$

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \\
&= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= 2\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{j-i+1} \\
&= 2\sum_{i=1}^{n}(H_{n-i+1} - 1) \\
&= 2\sum_{i=1}^{n}(H_i - 1) \\
&= 2\sum_{i=1}^{n} H_i - 2\sum_{i=1}^{n} 1
\end{aligned}
$$

Big important insight, that you can calculate the probability that any 2 arbitrary elements can be compared.

Is Quicksort In place?

In place algorithms use a constant amount of extra space, it isn't a function of the number of elements n. Because Quick sort is recursive it can add more variables to the stack (multiple versions of q, and r). So to be in place you use extra memory $\Theta(1)$ for algorithm variables + $\Theta(\log n)$ stack average. There isn't really a formal definition of In place. Informally it's that an algorithm doesn't use a lot of extra space.

One way to avoid using much extra space is to check which side of a partition is smaller and do that side of the partition first.

Randomized pivot selection

You can use a random pivot element as seen in the book.

3 Median Pivot

You can make sure that you've got a good pivot by picking 3 elements randomly and using their median as the pivot. This gives you a much better chance of pivoting near the middle.

Small Size Optimization

In practice when qucksort gets down to 10 to 20 elements many implementations use a more efficient algorithm like insertion sort to sort the small groups of elements.

14 Algorithm Strengths and Weaknesses

Algorithm	Comparisons	Moves	Spatial Locality	In Place	Good Worst Case
Merge Sort	$n \lg n$	unknown	✓	X	✓
Heap Sort	$n \lg n$	$n \lg n$	X	✓	✓
Quick Sort	$1.39 n \lg n$	$\frac{1}{2} n \lg n$	✓	✓	X

Memory Hierarchy

Memory Hierarchy allows further, machine specific optimizations. Doing a lot of work in fast memory is preferable to waiting to grab data from slower memories Merge sort and quick sort both take advantage of this spacial locality of fast memory. It sub sorts the array in pieces that fit into caches. Heap sort doesn't as it's sorting mechanism jumps all over the tree. It lacks spatial locality. Merge Sort also has the problem that it isn't In-place, so it uses a lot of extra memory.

15 Linear Sorting Algorithms

Re-evaluating Sort

All of our algorithms have thus far been strongly dependent upon comparisons. Let's take a closer look at that.

Find biggest element in array

Run one instance of selection sort, find biggest element. $n - 1$ comparisons.

Find second largest element in array

First idea: form max heap and pull two off the top. Create Heap + sift: $\approx n + \lg n$

Find k largest elements in array

Sift again: $n + (k - 1) \lg n$

Natural approach: Hold a Tournament

Comparisons are like playing games in a sportsball match. You need to eliminate $n-1$ teams to find the best, so you must at least run $n-1$ comparisons. This gives us a lower bound. In a double elimination tournament when one of the playing groups loses it must lose again to be eliminated. The superior team will never lose. All other teams must lose at most twice. If you lose to someone and they lose to someone else, you're no longer second best. So you don't need to incur an extra comparison. If you keep track of who lost to whom you'll conserve comparisons and be bound by the height of the tournament bracket, $\lg n - 1$ When you run the tournament you incur $n-1$ to find the best player. Then another $\lg n - 1$ to order everyone else, giving $n + \lg n$ comparisons.

Finding Best and Worst

Run tournament to find best, then take all teams who lost both matches ($n/2$), and run again in reverse. There are $\frac{n}{2} - 1$ following comparisons to find the worst team. $n - 1 + \frac{n}{2} - 1 = n + \frac{n}{2} - 2 \approx \frac{3}{2}n$.

16 Counting Sort

Say you have a list of duplicate integers in a range and you want to sort them. n integers in the range $0, \ldots, k-1$. Sort A into B. So simply count the number of integers of each size, and dole them out in order.

```
for $i = 0$ to $k - 1$ do
    $C[i] \leftarrow 0$;
end
for $j = 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[i]] + 1$;
end
$t \leftarrow 0$;
for $i = 0$ to $k - 1$ do
    for $j = 0$ to $C[i]$ do
        $t \leftarrow t + 1$;
        $B[t] \leftarrow i$;
    end
end
```

Algorithm 10: Counting Sort

The running time is the number of numbers plus the range of the numbers. $\Theta(n + k)$. If you're sorting a lot of numbers in a small range, it's the numbers that will dominate. If it's a large range with sparse numbers, the range will dominate.
An alternative method is to form the partial sums of C. Where each value in C is the number of elements less than or equal to the index. Then iterate backwards through the original array. When you encounter an element in A, lookup that index in C, then assign an element into B, then decrement the value in C. This has the added value of treating each element as a unique entity, instead of as simply an integer. So you could do this on comparable objects.

```
for $i = 0$ to $k - 1$ do
    $C[i] \leftarrow 0$;
end
for $j = 1$ to $n$ do
    $C[A[j]] \leftarrow C[A[i]] + 1$;
end
for $i = 1$ to $k - 1$ do
    $C[i] \leftarrow C[A[i]] + 1$;
end
for $j = n$ down to $1$ do
    $B[C[A[j]]] \leftarrow A[j]$;
    $C[A[j]] \leftarrow C[A[j]] - 1$;
end
```

Algorithm 11: Partial Sum Counting Sort

Running time $\Theta(n + k)$.
Now the C array represents the number of numbers less than the value of the index.
Why do we go through A backwards? To maintain stability.

17 Radix Sort

Sort on the least significant digit, then the 2nd least significant digit, etcetera until the largest digit arrives. This only works if a stable sort is used for each intermittent pass over a digit.
Proved by example in lecture. Proved by induction on the intermittent sorts.

Attribute	Value	Variable
Size	10	n
Radix	9	r
Digits	3	d

Running Time: $\Theta(d(n + r))$

Analysis of running time

First let's decide which radix is best for generic input. Let's introduce a new parameter: Size of Values: s. This is the total range of numbers. s, d, r are all related by $s = r^d$. If given 6 digit numbers in base 10, $s = 10^6$. Taking logs we get $\log_r s = d$ This gives us the new Running time of $\Theta((\log_r s)(n + r))$. Taking out the r is trickier. Let's think about optimizing this function. We can optimize by trying to find a minimum with respect to r. We can do that by taking the derivative and solving for 0.

$$\begin{aligned}
\left[\frac{\ln s}{\ln r}(n+r)\right]' &= (\ln s)\frac{\ln r - \frac{1}{r}(n+r)}{(\ln r)^2} \\
&= (\ln s)\frac{\ln r - \frac{n}{r}}{(\ln r)^2} = 0 \\
0 &= \ln r - \frac{n}{r} \\
r &= \frac{n}{\ln r} \\
r &= \frac{n}{\ln(\frac{n}{\ln n})} \\
&= \frac{n}{\ln n - \ln\ln n} \\
&\approx \frac{n}{\ln n}
\end{aligned}$$

So surprisingly the running time of radix sort can depend entirely on the input size, and not on the range of values.
$\Theta(\frac{\ln s}{\ln(\frac{n}{\ln n})}(n + \frac{n}{\ln n}))$
$\Theta(\frac{n \ln s}{\ln n})$
But that's not a great number, so in real life pick the closest power of 2. It's nicest for the computer, and a digit looks like a set of bits. Take every group of r bits and compare based on those bits and return them. So we want to use Radix sort when it's running time is less than Quicksort.

$$\Theta\left(\frac{n \log s}{\log n}\right) < \Theta(n \log n)\,\Theta(\log s) \qquad < \Theta\left(\log^2 n\right)\log s < \Theta(\log n)\log n \qquad = \log n^{\Theta(\log n)} s < n^{\Theta(\log n)}$$

So Radix sort is theoretically better when the range is relatively small. But for 1 million numbers in binary, that gives us a very large range: $1,000,000^{20}$
It's space-wise inefficient, but has spatial locality.

18 Bucket Sort

Assume you have a uniformly distributed array of real numbers between 0 and 1. Create n buckets.

```
Clear B;
for $i = 1$to$n$ do
    Put A[i] into bucket $B[\lfloor n \cdot A[i] \rfloor]$;
end
Sort each bucket;
Concatenate each bucket;
```

On average there will be 1 element per bucket. If you can rely on the buckets in the worst case not being very large you can use a quadratic sort within the bucket and still get a linear running time.
You can even apply this to non uniform distributions. Assume a Normal distribution. Simply change the size of the buckets relative to the mean, with buckets closer to the mean having a smaller range and ones farther from the range on either side being larger.
Also space wise inefficient, but has spacial locality.

19 Selection

How do we select the kth smallest element from a group of numbers? How do we select the median from a group of numbers? Median $= n + \frac{n}{2} \lg n$

```
Function SELETCT(A,k,p,r)is
    Find Approximate median for 'qth smallest';
    Partition based on 'qth smallest';
    if $k < q - p$ then
        SELETCT(A,k,p,q-1);
    end
    else if k>q-p+1 then
        SELETCT(A,k-q-p+1,q+1,r);
    end
    else
        k=q-p+1
    end
    returnq,A[q];
end
```

Take the list and find some approximate median, then look on the left or right side depending on how k compares to our approximate median.

Analyzing the recurrence

It sounds like selection should be a very important problem, but in real life it doesn't come up very much.

Lower Bound

Let's assume we know the true median, or that our approximate median is good enough.

$$\begin{aligned}
T(n) &= n - 1 + T\left(\lceil \frac{n-1}{2} \rceil\right) \\
&\approx T(\frac{n}{2}) + n - 1 \\
&\approx 2n - 1 - \lg n \\
&\approx 2n
\end{aligned}$$

So if we got really lucky and got the true median we could find the kth element in linear time. As such we have a reasonable assumption for our lower bound. You'll never do better than 2n comparisons in a real case when you don't have the perfect median.

Constrained case analysis

Let's assume that we get a q at the $\frac{1}{4}$ mark, and assume that our k is always on the larger side.

$$\begin{aligned} T(n) &= n - 1 + T(\frac{3n}{4}) \\ &= T(\frac{3n}{4}) + n - 1 \\ &\approx (n-1) + (\frac{3}{4}n - 1) + (\frac{3}{4}^2 n - 1) + \dots \\ &= \frac{1}{1 - 3/4}n - c\log n \\ &\approx 4n \end{aligned}$$

For the general fractional partition.

$$\begin{aligned} T(n) &= n - 1 + T((1-r)n) \\ &= T((1-r)n) + n - 1 \\ &\approx (n-1) + ((1-r)n - 1) + ((1-r)^2 n - 1) + \dots \\ &= \frac{1}{1 - 1 + r}n - c\log n \\ &\approx \frac{1}{r}n \end{aligned}$$

Absolute worst case q.

$$\begin{aligned} T(n) &= n - 1 + T(n-1) \\ &= (n-1) + (n-2) + (n-3) + \dots \\ &= \frac{(n-1)n}{2} \\ &\approx \frac{n^2}{2} \end{aligned}$$

Average case Analysis

Sum over all possible pivots and let's assume you always end up on the bigger side, for the pessimistic view. So we get the probabilistic analysis.

$$\begin{aligned} T(n) &= n - 1 + \sum_{q=1}^{n} \frac{1}{n} \cdot T(\max(q-1, n-q)) \\ &= \frac{2}{n} \sum_{\frac{n}{2}}^{n-1} T(q) \end{aligned}$$

Constructive Induction, guess the algorithm is linear

guess $T(n) \le an$

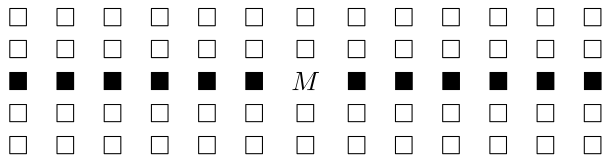
$$
\begin{aligned}
&= n - 1 + \frac{2}{n}\sum_{\frac{n}{2}}^{n-1} aq \\
&= n - 1 + \frac{2a}{n}\sum_{\frac{n}{2}}^{n-1} q \\
&= \frac{2a}{n}\left[\sum_{q=1}^{n-1} - \sum_{q=1}^{\frac{n}{2}-1}\right] + n - 1 \\
&= \frac{2a}{n}\left[\frac{n(n-1)}{2} - \frac{\frac{n}{2}(\frac{n}{2}-1)}{2}\right] + n - 1 \\
&= \frac{2a}{n}\left[\frac{n^2}{2} - \frac{n}{2} - \frac{n^2}{8}\frac{n}{4}\right] + n - 1 \\
&= an - a - \frac{an}{4} + \frac{a}{2} + n - 1 \\
&= \frac{3}{4}an - \frac{a}{2} + n - 1 \\
&= (\frac{3}{4}a + 1)n - \frac{a}{2} - 1 \quad \text{for induction to work} \leq an \therefore a \geq 4
\end{aligned}
$$

$$T(n) \approx 4n$$

20 Finding Median explicitly during selection

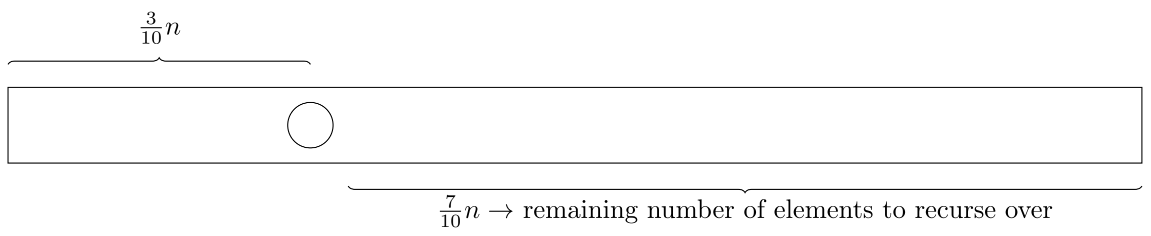
Take the numbers and put them into a 2 dimensional grid with 5 rows and n/5 columns.
Computer Memory is a 1 dimensional array, meaning that representing a 2 dimensional array requires a clever trick to access. A[i,j] Represented differently in row major order vs column major order. Column major order A[5(i-1) + j]
Without using very specific pseudocode, let's look theoretically at how something like this algorithm might work at a very high level.

```
1 Put items into 5x$\frac{n}{5}$ grid
2 Bubble Sort
3 // Find median of each column. 10*n/5 = 2n comparisons.
4 Find the median of these medians recursively. // smaller subset of elements it should take $T(n/5)$
     comparisons.
5 Move columns with small medians left and columns with large medians right. // This puts the median of
     medians in the middle element.
6 Partition using the median of medians. // Partitioning takes $n-1$ comparisons.
7 Recursively select on correct side, knowing what we know about median of medians. // Takes $T(\frac{7n}{10})$
     comparisons.
```



Worst Case analysis

About 25% of elements are guaranteed to be smaller, and about 25% are guaranteed to be bigger than the median of medians. The exact value is $\frac{3}{10}n$ This gives us an upper bound on the size of a recursive call to partition at $\frac{7}{10}n$, which is the remaining number of elements to recurse over.


$\frac{3}{10}n$
$\frac{7}{10}n \rightarrow$ remaining number of elements to recurse over

$$\begin{aligned}
T(n) &\leq 2n + T(\frac{n}{5}) + T(\frac{7}{10}n) + n - 1 \\
&= T(\frac{n}{5}) + T(\frac{7}{10}n) + 3n - 1 \text{ Constructive induction, guess } T(n) \leq an \\
&\leq a\frac{n}{5} + an\frac{7}{10} + 3n - 1 \\
&= \left(\frac{9}{10} + 3\right) n - 1 \\
&\text{need } an \implies \quad \textit{need } \frac{9}{10}a + 3 \leq a \therefore a \geq 30 \\
\therefore T(n) &\leq 30n
\end{aligned}$$

Median is doable in linear time, but 30 is a very large coefficient. Compare it to quick sort at $n \lg n$,n must be $\sim 2^{30}$ for our linear median selection to be better.

Theory

Say the upper bound, as some researchers claim to have found, that the upper bound on median selection is 3n.

$$\begin{array}{ccc}
2n < T(n) & & < n3n \\
(2+\varepsilon)n & & (3-\delta)n \\
& T(n) \approx \frac{5}{2}n &
\end{array}$$

21 Graph Algorithms

Graph description.
Undirected graph List representation is a bit awkward because it double counts shared edges. Altering an edge on one vertex requires altering the edge on its partner too. Requiring that you traverse both lists. One awkward way to alleviate this is to keep extra pointers between the list elements in shared edges.

Memory Usage

Matrix, where n = number of vertices, Memory usage = $\Theta(n^2)$
List, where m = number of edges, Memory usage = $\Theta(m + n)$
How big can m be? M can be as large as n^2, but will probably be smaller.
You could also use a bit matrix of size $\frac{n^2}{\text{word size}}$.

Edge Lookup

List: $\Theta(n)$ Matrix: $\Theta(1)$

List all Edges

List, Go down each index, then traverse each list. Amortized analysis is to visit n elements. $\Theta(m+n)$.
Matrix, Go across each row, looking at all the 0s, even when you don't care about them. $\Theta(n^2)$.

Take aways

Typically we want to use the Adjacency list representation, because it's linear for in the average case.

Changing the number of nodes

Adding Nodes

Add dummy values that you can fill later to the next power of 2. If you have 11 nodes build a matrix with 16 columns and just read the first 11, when a new node is added just use the dummy space, when you reach 17 you need to copy the old matrix into the new one of size 32.

Removing Nodes

Shrink the array in a way similar to adding a node.

Depth First Search

```
Mark v at visited;
for all v such that (u,v) is an edge and v has not been visited ;
DFSv;
```

Adjacency Matrix: $O(n^2)$
Adjacency List: $O(n+m)$

Applications

Identify Connected Components.

Breadth First Search

While depth first search uses a recursive call, or a stack (which are naturally similar in modern machines) BFS uses a queue, which while similar in order analysis for most things is not naturally represented in a stack based machine. Does have some nice applications, like shortest distance algorithms.

Identifying Connected Components

A graph is connected if there is a path from every vertex to every other vertex in the graph.

A connected component is a connected subgraph, or a connection of such components.

```
Function Component(G) is
    for all $x \in V[G]$ do
        $visited[x] \leftarrow False$;
        $t \leftarrow 0$;
    end
    for all $x \in V[G]$ do
        if not visited[x] then
            $t \leftarrow t + 1$;
            Compvisit [x];
        end
    end
end
Function Compvisit(x) is
    $visited[x] \leftarrow True$;
    $CompNum[x] \leftarrow t$;
    for all $y \in adj[x]$ do
        if not visited[y] then
            Compvisit[y];
        end
    end
end
```

Algorithm 12: Connected Component

Adjacency Matrix: $O(n^2)$
Adjacency List: $O(n + m)$

Bridges of Köningsberg

Theorem 21.1 (Eulerian Cycle 1736)**.** An undirected, (connected) graph $G = (v, e)$ has an Eulerian Cycle if and only if every vertex is connected and every vertex has even degree.

22 Shortest Path Algorithms

Types

- Single source, Single sink.
- All Pairs.
- Single source.

It's very hard to solve the Single source, single sink problem without also solving the single source problem for every node along the way. Same is true for All pairs.

Dijkstra's Algorithm

```
for all $x \in V[G]$ do
    // $\Theta(n)$
    $D[x] \leftarrow \infty$;
    $\Pi[x] \leftarrow NIL$;
end
$Q \leftarrow V[G]$ // Q will hold the vertices which haven't been checked off yet
$D[a] \leftarrow 0$;
while $Q \neq \emptyset$ do
    // $\Theta(n)$
    $x \leftarrow$Pop-minimum[Q];
    for all y adjacent to x in Q do
        // Depends on implementation. Can be adjacency matrix, checkable in $\Theta(n)$
        if $D[x] + W[x,y] < D[y]$ then
            // W[x,y] is the weight of edge x,y
            $D[y] \leftarrow D[x] + W[x,y]$;
            $\Pi[y] \leftarrow x$;
        end
    end
end
```

This bears a similarity to Breadth First Search. From the source, you reach the vertices separated by 1 edge first, then those separated by 2 edges, then 3, etc. This forms an expanding shell of visited nodes with known shortest distance. We have connections to edges outside the shell, with known edge weights. Since we are picking the smallest such edge, the BFS-like process guarantees that you determine the shortest path to that vertex.
This only works when you have non-negative weight edges.

Correctness Proof

Base case: Nothing is in the circle, except the source vertex with distance 0.
Inductive Step: Remove smallest edge weight from consideration and update connecting vertices if smaller.
Since you know the shortest path to everything in the circle, the available path to the next shortest edge is guaranteed to be the shortest path to that vertex.

Implementation details

Create Q as array. Let Q hold 1's for vertices that haven't been eliminated yet, and 0's otherwise. Naive approach is to travel Q looking for 1, lookup weight in D and track smallest value. If implemented with Adjacency Matrix, the whole thing done in $\Theta(n^2)$.
The alternative is to use a priority queue / heap for Q, removing the smallest value. This necessitates some extra heap functions to grab an arbitrary node in the heap and change its weight. That let's you pop from the heap in logarithmic time. There's an additional requirement to visit $|adj(x)|$ additional vertices, and update them with our heap manipulation function to update weight, which is an additional $\Theta(\lg n)$ Since Dijkstra does this for every element the running time is $\Theta(m \cdot \lg n)$.
So if your graph is dense it's better to use the Adjacency Matrix algorithm, which is quadratic in the number of vertices. If your graph is sparse it's better to use the $m \log n$ algorithm with a priority queue.
Of course in reality the quadratic, Adj. Matrix algorithm is usually better at the low level because of spatial locality.

23 NP-Completeness

Goal is to separate problems easy to solve from those hard to solve. Loosely speaking easy means polynomial time and hard means exponential time.
These are not perfect definitions, in spite of that fact it's still okay.

- Nature is kind to us. Natural problems that are polynomial time tend to have low degree. Not a theorem, just an empirical statement.

- We don't really know what a computer is. Real world computers have memory hierarchy and variable running times, there's concurrency as well. All models are equivalent up to polynomial time. One exception is quantum computers, but not really proven in reality.
- Polynomials are closed under addition, multiplication, and composition.

Decision Problem Theory

Decision problems are yes/no questions.
Does a graph have an Eulerian cycle? But not What is the Eulerian Cycle.

Definition 23.1 (P)**.** Decision problems solvable in polynomial time.

Definition 23.2 (NP)**.** Decision problems verifiable in polynomial time with a certificate.

Problems in the class NP have the dichotomy that if the answer is yes you can demonstrate it in polynomial time, but many of them are not able to be disproved in polynomial time. When the answer is yes it's easy to prove, otherwise it's very hard.
Certificate is usually the solution to the problem. Example would be the coloring problem from the homework.

Definition 23.3 (SAT)**.** Satisfiability Problem Given a Boolean formula, is there a way of setting the variables to true and false such that the formula is true?

If it is then it can be evaluated in polynomial time, the certificate is the assignment. If it isn't satisfiable the only way to show that is to try every possibility of assignments, which takes exponential time.

The Class of NP problems

Is it the case that if something can be solved in polynomial time it can be verified in polynomial time? Does $P = NP$? Or more directly, are P and NP-Complete classes distinct? No one knows.
NP-Complete: The class of problems that are all equivalent to one another under reduction. Formula-SAT, circuit-SAT and coloring are all the same problem, really. They can be reduced to one another.
Most things in real life are in the NP class. Empirical Statement: For problems occurring in nature, Once in the class NP, it's either P or NP-Complete. There's a theorem that says that if $P \neq NP - Complete$ then there are an infinite class of problem between them.

Reduction

Having proved that a problem is NP complete, how do we prove other problems are also NP complete?

Theorem 23.1. Circuit SAT is NP-complete.

Proof.
- Show in NP
- Show hard for NP
- Show that Circuit SAT is at least as hard as some other problem which we know is NP-complete.

□

Formula SAT $\leq_p$ Circuit SAT
Circuit SAT $\leq_p$ Formula SAT

Converting from circuit to formula is harder than the other way. But the two are still verifiable in the same class of running time.

Traveling Salesman Problem

Eulerian Cycles require that you hit every path once, while Hamiltonian Cycles require that you hit every vertex once. This is the crux of the traveling salesman problem. Hamiltonian Cycles are NP-Complete.
The Traveling Salesman problem is this problem of finding the shortest path that hits all the nodes in a graph. Because the traveling salesman problem isn't a yes no question the technically correct way of formulating it is to ask if there is a path with length less than some target length.
This is a generalization of the Hamiltonian Cycle problem, which is NP complete, therefore the traveling salesman problem is NP complete.

theory

There were generalizable characteristics of the Eulerian cycle problem that allowed Mathematicians to determine proofs around them, but the Hamiltonian Cycle problem doesn't have any generalizable characteristics like that and it wasn't until the advent of the NP completeness theorem that this was understood.

Problems between P and NP-Complete

- Factoring
- Discrete Logarithm
- Graph Isomorphism

Their exact location isn't known.

Exam Materials

Must prove a problem is in class NP by stating what the cert is and showing how to use the cert to verify.
No Reductions.
Show that you can use the fact of the class of the decision problem form of the question to derive the class of the optimization problem form. If you can determine the yes/no condition of whether a formula is satisfiable, you can determine the certificate for that factor in Polynomial time.

Formula SAT

Assume Decision version of Formula SAT is in P. We want to find the assignment.
Given Formula A variables $x_1, x_2, \ldots, x_n$

```
if not SAT-decidability(A) then
    return False
end
for $i = 1 \textbf{to} n$ do
    $B \leftarrow A$;
    Assign True to $x_i$in A // Every time you see xi in the formula substitute with true
    Simplify to remove True;
    if not SAT-decidability(A) then
        $A \leftarrow B$;
        Assign False to xi and simplify;
    end
end
return X;
```

Graph Coloring Problem

Given an integer k, is a graph k-colorable?
That's the decision problem. You can use that to solve the real question of what's the smallest number k such that a

graph is k-colorable.

```
Algorithm: Graph Coloring Problem
$c \leftarrow 1$;
while not Colorable(G,c) do
    $c \leftarrow c + 1$;
end
for i=1 to n do
    for d =1 to c do
        Set vertex i to color d;
        if Colorable(G',d) then
            exit;
        end
    end
end
```

Assign arbitrary colors to the graph nodes one at a time and then ask the routine if the graph is still c-colorable. But assignment of colors isn't allowed in our routine. So we need to use a trick.
Make a complete graph of c vertexes. Then connect our goal vertex in the original graph G to every vertex except the color we want. This forces the color determiner to require a color at that vertex.
Size of $G' = |G| + c^2 + (c-1)n$