CMSC351 Notes

A struggling 351 student

25th June, 2023

Contents

Lecture Notes

1 Theta Notation and Order Notation

Example: Something like $\frac{n(n+1)(n+2)}{6}$ would be written as $\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3}$ so that we can see the highest-order term more clearly. When n is very big, $\frac{n^3}{6}+\frac{n^2}{2}+\frac{n}{3} \approx \frac{n^3}{6} \approx n^3$, so the running time is $\Theta(n^3)$.

Order notation is really about sets, e.g. $\Theta(n^3)$ is the set of all functions that run in cubic time.

Definition: $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 \in \mathbb{N}^{\geq 1}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$
($\Theta(g(n))$ is the set of all functions $f(n)$ such that after n goes beyond some very large point (represented by n_0), $f(n)$ is basically the same as $g(n)$ (you can sandwich $f(n)$ between two constant multiples of $g(n)$))

Basically, Θ is is both an upper and lower bound, while O is just for upper bounds and Ω is just for lower bounds.

Analogy	Order
$=$	Θ
$\leq$	O
$\geq$	Ω
$<$	o
$>$	ω

$f = O(g)$ means that functions that grow at the rate of f grow slower than or at the same rate as functions that grow at the rate of g. ($\in$ would make more sense than $=$ since they're sets but mathematicians are weird). Some examples:

$$\begin{aligned}&6n^3 = O(2n^3)\\&6n^3 = O(n^4)\ \ (n^3 \text{ is less than } n^4)\\&6n^3 \neq \Omega(n^4)\end{aligned}$$

2 Summations

Memorize these:

- $\sum_{i=0}^{\infty} r^i = \frac{1}{1-r}$ $(r < 1)$
- $\sum_{i=0}^{n-1} r^i = \frac{1-r^n}{1-r}$ $(r \neq 1)$
  If $r = 1$, then the sum is n
- $\sum_{i=0}^{\infty} ir^{i-1} = \frac{1}{(1-r)^2}$ $(-1 < r < 1)$
- $\sum_{i=0}^{???} r^i = 1$
- $\sum_{i=0}^{n-1} ir^{i-1} = ???$

3 Triangulated Polygons

A triangulated polygon is either

- A polygon with three sides
- A polygon with more than three sides with a straight line between two corners not next to each other forming two polygons sharing that new line such that each new polygon is triangulated

Structural induction proofs are inherently strong induction proofs.

Proof by structural induction

Theorem: A triangular polygon with n sides can be 3-colored.

Base case: Triangle

Inductive Step: Consider a triangulated polygon with n sides
Since it's triangulated, it can be split into two triangulated polygons with fewer sides.
By the inductive hypothesis, both sides are 3-colorable, so color them.

4 Recurrence relations

Catalan numbers:

$c_0 = 1$
$c_n = \sum_{k=1}^{n-1} c_k c_{n-k-1}$

5 Full Binary Trees

Definition: In a full binary tree, every node has 0 or 2 children.

Definition: An **internal node** is a non-leaf node (a node that has one or more children).

Theorem: The number of internal nodes in a full binary tree is 1 more than the number of leaves.

Definition: External path length is the sum of the distances to the leaves.

Definition: Internal path length is the sum of the distances to the internal nodes.

$E = I + 2n$, where n is the number of internal nodes, E is external path length, and I is internal path length.

$E = N + I - 1$

6 Heapsort

Definition: A **heap** is a rooted tree where every node has a value, and the value of a node is greater than or equal to the values of its children (in min-heaps, the value of each node is less than or equal to its children, but it seems Kruskal's going to ignore min-heaps).

Heapsort is a kind of **selection sort** algorithm because it involves finding the next maximum at each step.

For heapsort, all heaps are almost-complete binary trees.

Definition: A **complete** tree has all its leaves on the same level (number of nodes is a power of 2). An **almost-complete** tree is a complete tree missing some leaves on the right side in the last level.

Heaps are stored as arrays. If a node is at index i in the array, then its left child is at index $2i$, its right child is at index $2i + 1$, and its parent is at $\lfloor \frac{i}{2} \rfloor$. This is very convenient for computers. Can sort in-place.

Two phases:

1. **Create heap:**
   Build up sub-heaps from the bottom up. Start at the second-last level, because the last level is already all mini heaps.
   Going from right to left (skipping the nodes in the last level), **sift down** each node to make a sub-heap (sifting down explained below)
2. **Finish:**
   (a) Swap the root node and the current last node (array is built up from right to left in-place)
   (b) **Sift down** that new root node
      (a) If the current node (starts out as the new root node) is greater than or equal to its children, we're done
      (b) Otherwise, swap it with the larger of its children (2 comparisons if done right)
      (c) Repeat from that new, lower position
   (c) Repeat until no more nodes left

Analysis

While building heap:
Number of comparisons on ith level from *the bottom* (bottom level is $i = 1$): $2(i-1)\frac{n}{2^i}$
Comes out to approximately $2n + O(1)$

While finishing:

- Number of sift down operations: $n - 1 \sim n$
- Number of levels per sift: $\sim \lg(n)$
- Number of comparisons per level during sift: 2
- So total number of comparisons: $\sim 2n \lg n$
  More accurate: $\sum_{i=0}^{n-1} 2 \lg n$ (because size of tree shrinks with each step)
  Can be written as $\sum_{i=1}^{n} 2 \lg i = 2 \lg(n!)$
  Using **Stirling's formula** to approximate $n!$, that can be approximated as $2n \lg n + O(n)$

So total number of comparisons is $2n \lg n + O(n)$

Heap creation time recurrence relation:

$$H(n) = 2H\left(\frac{n-1}{2}\right) + 2 \lg n, H(1) = 0$$

Binary search is $\lg n$ going down but $\lg(\lg n)$ going up.

7 Sorting networks

You can sort in parallel. Need non-adaptive or oblivious algorithms. Mergesort can be done in parallel

Unlike adaptive algorithms, oblivious sorting algorithms always compare the same elements regardless of the data they get. Since mergesort and quicksort change depending on the data, not oblivious.

More here: https://cs.stackexchange.com/a/58032

8 n-coins problem

Lemma: Assuming we can solve the n-coins problem in k weighings, then we can solve the $3n$-coins problem in $k+1$ weighings.

Lemma: Assuming that we can solve the nonadaptive (or oblivious) n-coins problem in k weighings, we can solve the nonadaptive/oblivious $3n$-coins problem in $k+1$ weighings.

Lemma: Assuming that we can solve the nonadaptive/oblivious n-coins problem in k weighings, where no coin is on one scale the entire time, then we can solve the nonadaptive/oblivious $(3n+3)$-coins problem in $k+1$ weighings, where no coin is one scale the entire time.

9 Selection

Problem: In a list of n values, find the kth smallest.

The specific algorithm we use to solve this is quickselect (Wikipedia article). Like quicksort, it involves finding a pivot and putting smaller elements on its left and larger elements on its right. Unlike quicksort, we only have to look on either the left or right side of the pivot, we don't have to deal with both sides.

Algorithm

1. Put the elements into a $5 \times \frac{n}{5}$ grid (no comparisons)
2. Find the median of each column ($\frac{n}{5} \cdot 10 = 2n$ comparisons)
3. Within each column, move the small elements in the top, large elements in the bottom, and median to the middle (no comparisons)
4. Find the median of medians ($T(\frac{n}{5})$ comparisons)
5. Move the columns with small medians to the left, large medians to right, and median of medians to middle (no comparisons)
6. Partition using median of medians as pivot ($n-1$ comparisons)
7. Recursively call algorithm on proper side ($T(\frac{7n}{10})$ comparisons)

Why 5?: It's the smallest odd number that gives you linear time performance.

Wikipedia article on median of medians algorithm is good.

Pseudocode

Pseudocode for recursive in-place algorithm (tail-recursive, so can use non-recursive loop instead):

```
def selection(A, p, r, k):
  s = approximate_median(A, p, r) # Like quicksort pivot
  q = partition(A, p, r, s) # Like quicksort partition step
  if k < q:
    return selection(A, p, q - 1, k)
  else if k > q:
    return selection(A, q + 1, r, k)
  else:
    return q
```

Analysis

Best case (assume picked pivot is always median): $T(1) = 0, T(n) = T\left(\frac{n}{2}\right) + n - 1$, so $\underline{T(n) \approx 2n}$
Average case is also linear: $30n$ (see analysis below)
Worst case is $O(n^2)$

Recurrence

(see Algorithm for where the parts of the relation come from)

$$T(n) \leq 2n + T\left(\frac{n}{5}\right) + (n-1) + T\left(\frac{7n}{10}\right)$$

Using constructive induction (guess $T(n) = an$), we get that $\boxed{T(n) \leq 30n}$

10 Breadth-First and Depth-First Search

10.1 Breadth-First Search

Search all the nodes adjacent to current node, then search all nodes adjacent to all of those, and so on (with a tree, search level by level).

Uses a queue.

BFS takes $\Theta(m+n)$, where n is number of vertices and m number of edges.

10.2 Depth-First Search

For each node adjacent to current node, search it and then search the nodes adjacent to it (drill down).

Uses a stack.

Also takes $\Theta(m+n)$.

11 Upper and Lower Bounds

For maximum contiguous sum, a particular algorithm could be $\Theta(n^3)$ or $\Theta(n)$ or something else, but the problem itself would be written as $O(n^3)$ or $O(n)$ just in case there's a faster algorithm (the optimal algorithm is $\Theta(n)$).

Comparison model: Can only ask questions about how one element compares to another.

- Finding smallest/largest element: $n-1$ comparisons
- Finding two smallest/largest elements: $n + \lg n + O(1)$ comparisons
- Finding smallest and largest elements: TODO look it up
- Merging two sorted lists each of size n: $2n-1$ comparisons
- Searching a sorted list: $\lg n + O(1)$ comparisons
- Sorting: $n \lg n + O(n)$ comparisons

Number of leaves in a decision tree for sorting N elements: $N!$
Height of such a decision tree is $\geq \lg(N!) \geq N \lg N - N \lg e + O(1)$

Lower bound for sorting algorithm for N elements: $N \lg N - N \lg e + O(1)$ comparisons
Upper bound for sorting (mergesort where N is power for 2): $N \lg N - N + 1$ comparisons

Video Notes

0 Mathematical Induction

Constructive Induction

Mathematical induction is very powerful in proving results but very weak in deriving results.

With constructive induction, if you know or guess the form of an answer, you can sometimes derive the actual answer while doing mathematical induction to prove it.

Before base case, make a guess with some coefficients that haven't been fixed yet. The base case will help constrain them a bit. Then in the inductive step, you solve for them.

1 Maximum Contiguous Sum

Definition: The maximum possible sum gotten by summing a subarray of some array.
e.g. in [3, 2, -4, -5, 6, 1, -3, 7, -8, 2], the maximum contiguous sum is 11 (gotten by summing the subarray [6, 1, -3, 7])

The empty list counts as a subarray, so if all the elements are negative, the maximum contiguous sum would be 0 (sum of [])

Brute force solution (cubic)

The following brute solution has 3 nested loops that iterate roughly n items, so its time complexity is $\Theta(n^3)$:

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

We can count more accurately how many times the innermost loop gets executed with summations:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\sum_{k=i}^{j}1 = \frac{n(n+1)(n+2)}{6}$$

This confirms that it is cubic.

Smarter solution (quadratic)

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

We can again count how many times the inner most loop executes using a summation:

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

This confirms that it is quadratic.

Dynamic programming solution (linear)

```
M = 0
# The maximum sum that ended on the previous element
S = 0
for i in 1 to n do
  # Max with 0 in case it goes negative (empty subarray always allowed)
  S = max(S + A[i], 0)
  M = max(M, S)
```

Loop runs $\sum_{i=1}^{n} 1 = n$ times, so $\Theta(n)$

2 Intro to NP

Easy and hard problems:

- Easy problem - Solvable in polynomial time $(O(n), O(n \log n), O(n^2), ...)$
  Most "natural problems" that are in polynomial time have "low degree" polynomial times
- Hard problem - Exponential time $(O(2^n), O(n^n), ...)$
  Most "natural problems" that are in exponential time have "high degree" exponential times

A natural problem is one that comes up in real life or "feels" natural. An unnatural problem doesn't come up in real life and "feels" unnatural.

Traveling Salesman Problem (TSP)

Salesman needs to find a way to go to all the cities, all connected with roads. There are $(n-1)!$ possible routes, where n is the number of cities.

There are algorithms for this that run in exponential time, but none that run in polynomial time.

The NP class

Definition: A **decision problem** is a Yes/No question, e.g. "Is a given number prime?".

Definition: The class **P** is the set of decision problems that can be solved in polynomial time.

- **EXP** is the class of problems that can be solved in exponential time
- **NP** is a class inside EXP
- **P** is inside NP
- **NP-complete** is another class inside NP consisting of provably hard problems inside NP (Traveling Salesman Problem is NP-complete)

Open question in Computer Science: Does P = NP?
If any NP-complete problem can be proved to be in P, then all problems in NP are in P.

Most "natural" problems inside NP are either in P or NP-complete.

3 Bubble Sort

Type of ”selection sort” algorithm (not to be confused with *the* Selection Sort algorithm)

Pseudocode:

```
for i = n downto 2:
  for j = 1 to i - 1:
    if A[j] > A[j + 1]:    # Comparison
      swap(A[j], A[j + 1]) # Exchange
```

Number of comparisons: $\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = \frac{n(n-1)}{2} = \binom{n}{2}$

Number of exchanges:

- Best case: If already sorted, then no exchanges will happen
- Worst case: If already sorted in reverse, then exchange will always happen, so same as number of comparisons: $\frac{n(n-1)}{2}$
- Average case: Each permutation is equally likely.
  Need to count number of **inversions** (aka **transpositions**), which is a pair of elements out of order
  Number of inversions is same as number of exchanges
  Number of inversions is just half the number of pairs, which is $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4}$

Modified Bubble Sort

Keep track of where last swap was made in last iteration, and you won’t have to go beyond that next iteration.

```
i = n - 1
while i > 0:
  t = 1
  for j = 1 to i:
    if A[j] > A[j + 1]:
      swap(A[j], A[j + 1])
      t = j
  i = t - 1
```

Cocktail Shaker Sort

Instead of going left to right and starting back at the left again, it goes left to right, then right to left, then left to right, like it’s being shaken.

4 Insertion Sort

4.1 Insertion Sort with Sentinel

Build up an array on the left. For each element `A[i]`, move elements in the sorted part right until you get to its proper location, then insert `A[i]` there.

```
A[0] = -infinity
for i in 2 to n
  t = A[i]
  j = i - 1
  while t < A[j]
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = t
```

Number of comparisons:

- Best case (already sorted): $n-1$
- Worst case (sorted in reverse): $\frac{n(n-1)}{2} - 1 = \frac{(n+2)(n-1)}{2}$
- Average case:
  Probability that you end up in some position j: $\frac{1}{i}$
  Average case: $\displaystyle\sum_{i=2}^{n}\sum_{j=1}^{i}\frac{1}{i}(i-j+1) = \frac{(n-1)(n+4)}{4}$

Number of moves:
There are always n more moves than comparisons

- Best case (already sorted): $2(n-1)+1 = 2n-1$ (take the element out and put it right back) (+1 for the sentinel at the top)
- Worst case (sorted in reverse): $\dfrac{(n+2)(n-1)}{2} + n$
- Average case: $\dfrac{(n-1)(n+4)}{4} + n$

4.2 Brick Stacking Problem

Can stack arbitrarily many bricks by making top brick stick out by 1/2, brick below that stick out by 1/4, brick below that stick out by 1/6, then 1/8, 1/10, ...

Harmonic sum: $H_n = 1 + \dfrac{1}{2} + \dfrac{1}{3} + \dfrac{1}{4} + ... + \dfrac{1}{n} \approx \ln n$

4.3 Insertion Sort Without Sentinel

Same as before, but instead of having sentinel, check that `j` hasn't gone past the start of the list. Now, need to check that `j > 0` at each iteration of the while loop, so the previous version is actually more efficient.

```
for i in 2 to n
  t = A[i]
  j = i - 1
  while j > 0 and t < A[j]
    A[j + 1] = A[j]
    j = j - 1
  A[j + 1] = t
```

Number of comparisons:

- Best case (already sorted): same as before: $n-1$
- Worst case (sorted in reverse): OLD $-(n-1)$
- Average case:
  $\text{OLD} - \displaystyle\sum_{i=2}^{n}\frac{1}{i} = \frac{(n-1)(n+4)}{4} - (H_n - 1)$

Number of moves is same as before, except you don't set the sentinel, so it's one less move.

5 Selection Sort

Start building up a sorted part on the right side of the array. At each iteration, get the biggest of the unsorted elements on the left and put it at the start of the sorted part on the right.

```
for i in n downto 2
  k = 1
  for j in 2 to i
    if A[j] > A[k]
      k = j
  swap(A[k], A[i])
```

Number of comparisons (best/worst/average are all same): $\dfrac{n(n-1)}{2}$

Number of moves: $n - 1$

6 Merge sort

Mergesort is a **divide-and-conquer** algorithm (splits up problem into subproblems)

Merging

Merge sort uses merging. Possible to do in-place but not practical, so we won't do it.

Number of comparisons when merging arrays A and B:

- Best case: n (when $A_n < B_1$ or $B_n < A_1$)
- Worst case: $2n - 1$ (when A_n and B_n are the two largest elements)
- Average case: $2n - 2 + \dfrac{2}{n+1}$

In general, number of comparisons if $m \le n$ (m is size of A, n is size of B) (same algorithm as before):

- Best case: m (when $A_m < B_1$)
- Worst case: $m + n - 1$ (A_m and B_n are the two largest elements)
- Better algorithms exist when m is much smaller than n. For example, when $m = 1$, you can basically use binary search ($\lg n$)

Pseudocode

```
def mergesort(A, p, r):
  if p < r:
    q = floor((p + r) / 2)       # Find middle of list
    mergesort(A, p, q)           # Sort left half
    mergesort(A, q + 1, r)       # Sort right half
    merge(A, (p, q), (q + 1, r)) # Merge the two halves
```

Analysis

We'll use tree method to analyze mergesort.

Number of comparisons at depth i is $n - 2^i$ (root node at depth 0), except for the lowest level, where it's 0 because those sublists have either 0 or 1 elements.

Number of levels in total is $\lg n$ but the lowest level has 0 comparisons

So total number of comparisons is $\displaystyle\sum_{i=0}^{\lg n - 1} n - 2^i = n \lg n - n + 1$

7 Heap Sort

See lecture notes

8 Recurrences (TODO)

9 Arithmetic

9.1 Elementary Arithmetic

Addition

Pseudocode:

```
def add(x, y, z):
  carry = 0
  for i in range(0, n):
    carry, z[i] = add_digits(x[i], y[i], carry)
  z[n] = carry
```

Adding two digits and a carry-in is an **atomic add** (`add_digits` above).
The time for an atomic add is α, so the time for the algorithm above is αn, where n is the number of digits.
You can't do better than linear time.

Multiplication

Multiplying two digits is an **atomic multiply**.
The time for an atomic multiply is μ, so the time for the standard multiplication algorithm is $n^2\mu + 2n(n-1)\alpha$ (ignoring carries)
Can do better than quadratic time with Karatsuba's algorithm

9.2 Recursive multiplication

Two-digit multiplication

Multiplying two 2-digit numbers: $\underline{\text{ab}} \times \underline{\text{cd}} = (10a + b)(10c + d) = 10^2ac + 10(ad + bc) + bd$
We can "add" 10^2ac and bd by simply concatenating the result of ac and the result of bd together.
Then we add $10(ad + bc)$ to that in the normal way.

Every multiplication is a two-digit multiplication with a large enough base.

Recurrence for the time to multiply:

$$M(1) = \mu, M(n) = 4M\left(\frac{n}{2}\right) + A(n) + A(n) = 4M\left(\frac{n}{2}\right) + 2\alpha n$$

Un-recurrence-ified: $M(n) = n^2\mu + 2\alpha n(n-1)$ (same as standard algorithm)

9.3 Karatsuba's algorithm

Observe that we don't need ad and bc separately, we only need $ad + bc$. $ad + bc = (a + b)(c + d) - (ac + bd)$
Now product is $10^2ac + 10((a + b)(c + d) - (ac + bd)) + bd$
So we only multiply 3 times (ac, bd, and $(a + b)(c + d)$)

Recurrence for time to multiply using Karatsuba's:

$$M(1) = \mu, M(n) = 3M\left(\frac{n}{2}\right) + 2A\left(\frac{n}{2}\right) + 2A(n) + A(n) = 3M\left(\frac{n}{2}\right) + 4\alpha n$$

Un-recurrence-ified: $M(n) = (\mu + 8\alpha)n^{\lg 3} - 8\alpha n$

This has a larger constant factor but smaller exponent than standard algorithm, so better for large n (big numbers).

In our examples, an atomic value was base 10, but in real life, it's 2^{64}.

Better algorithms:

- Toom-Cook: Generalizes Karatsuba's so you can get $\Theta(n^c)$ for c arbitrarily close to 1. Trade-offs not completely understood.
- Harvey-van der Hoeven: $\Theta(n \log n)$. Only useful for ludicrously large numbers.

10 Approximating Summations

To get the degree of the high-order term, can get lower and upper bound. If the degrees of the lower and upper bounds match, then you have it. Techniques for getting lower and upper bounds:

- Gauss's sum (sum of i) can be split into half to refine the lower bound.
- For sum of $\frac{1}{i}$, the upper bound can be taken as $1 + \frac{1}{2} + \frac{1}{2} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \ldots$ (groups of powers of 2 so that each group adds up to 1). For that summation, upper bound is $\lg(n+1)$ since that's how many groups there are.
- Lower and upper bounds for a monotonically increasing continuous function f:

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

11 Quicksort

The algorithm:

1. Pick out some item, call it the **pivot**.
   Note: Kruskal wants to use the last element as the pivot.
2. The **partition** step.
3. Repeat on the left side and the right side.

The partition step

Put all smaller items left, all larger items right, and put the pivot in between.

- Process list from left to right. Group small values on the left, then large values, and unprocessed values will be at the end.
- i keeps track of where the small values end, j keeps track of where the large values end. p is the start of the subarray, r is the end of the subarray.
- At the start, $i = p - 1$, j really starts at p
- When processing A_{j+1}:
  - If it's larger than the pivot, then it's already in the right place, so just increment j.

    – If it's smaller than the pivot, exchange A_{i+1} and A_{j+1} so that it's now at the end of the small values group.
      Then increment both i and j.

Pseudocode:

```
function partition(A, p, r)
  X = A[r]
  i = p - 1
  for j in p to r - 1:
    if A[j] <= X:
      i += 1
      swap(A[i], A[j])
  swap(A[i + 1], A[r])
  return i + 1
```

Best and Worst Case Comparison

$n-1$ comparisons in one partition step (compare every element to pivot)

Total # of comparisons in worst case (pivot on smallest/largest element): $\dfrac{n(n-1)}{2}$

Total # of comparisons in best case (pivot on median): $\sim n \lg n$

Average Case Comparisons Approximation

Assume average case happens when pivot is the $n/4$th value (halfway between smallest value and median, or halfway between median and largest value)

Recurrence for number of comparisons:

$$S(0) = \begin{cases} 0 & \text{if } n = 0, 1 \\ S\left(\frac{n}{4}\right) + S\left(\frac{3n}{4}\right) + (n-1) & \text{otherwise} \end{cases}$$

Total # of comparisons: $\sim n \lg n$

Exact Average Case Comparisons

Let $S(n)$ be the average number of comparisons
Assume the first partition pivots on the qth smallest value
Then the average number of comparisons to sort will be $(n-1) + S(q-1) + S(n-q)$

Recurrence for average number of comparisons:

$$S(0) = \begin{cases} 0 & \text{if } n = 0, 1 \\ (n-1) + \displaystyle\sum_{q=1}^{n} \frac{1}{n} \left(S(q-1) + S(n-q)\right) & \text{otherwise} \end{cases}$$

Using constructive induction, we get $\boxed{\sim 2n \ln n}$.

Choosing Pivots

Risky to pivot on last element. Better choices are

- Middle element
- Median of first, middle, and last elements

- Random element
- Median of 3/5/7/whatever random elements

Comments on Quicksort Being In-Place

Quicksort needs $O(\log n)$ space for the call stack, so not constant memory. Best case is if pivot is smallest element, when call stack is empty.

Definition: An algorithm is in-place if it uses $O(1)$ extra variables and (on average) $O(\log n)$ extra index variables.

Definition: Since index variables use $O(\log n)$ bits, we can also say an algorithm is in-place if it uses $O(1)$ extra variables and (on average) $O((\log n)^2)$ extra bits.

Not a fundamental definition. Just one Kruskal uses?

12 Comparison of Sorting Algorithms

Temporal and Spatial Locality

Temporal locality: When you access something several times within a relatively short time.

Spatial locality: When you have access things that are close together (e.g. processing an array requires accessing a bunch of data that's close together).

Why quicksort is preferred

Heap sort does not have good spatial locality, unlike bubblesort, insertion sort, selection sort, mergesort, and quicksort.

Mergesort isn't really in-place (can be done, but very hard to implement).

13 Counting Sort

Naive version

Input: Array `A` with n values, where each value is an integer between 0 and $k-1$, inclusive.
Output: Array `A` sorted into array `B`
Overview:

1. Count how many times each value in `A` occurs
2. Put the counts into an array `C` so that `C[i]` is the number of times value `i` occurs.
3. Iterate through `C` putting value `i` into the next `C[i]` locations of `B`.

Total time: $\Theta(n+k)$

But this will only repeat the same element multiple times. It won't work if we want to be able to sort data by a certain property.

Better version

1. Count how many times each value in A occurs
2. Put the counts into an array `C` so that `C[i]` is the number of times value `i` occurs.
3. Form the partial sums of the values in `C`

4. Iterate backwards through `A` putting value `A[i]` into the proper location of `B` using `C` to determine the index.
   When you place a value `i` from `A` into `B` at index `C[i]`, decrement `C[i]`.

Pseudocode:

```
# Initialize C
for i in range(0, k) do C[i] = 0
# Put counts into C
for i in range(0, n) do C[A[i]] += 1
# Form partial sums
for i in range(1, k) do C[i] += C[i - 1]
# Final step
for j in range(n, 1) do
  B[C[A[j]]] = A[j]
  C[A[j]] -= 1
```

Total time is again $\Theta(n + k)$

14 Radix Sort

Use a stable sort (e.g. counting sort) to sort by last digit, then sort by second-to-last digit, then by third-to-last digit, and so on (move right to left).

The **radix** (R) is the range of the letters or digits.
D is number of digits.

For words, you would pad on right with blanks. For numbers, you would pad on left with 0s.

Time for counting sort inside radix sort: $\Theta(N + R)$
Time for radix sort: $\Theta(D(N + R))$ (see below for time in terms of S)

Optimizing time

$\Theta(D(N + R))$ has too many variables, so make new variable S (size/range of one number) that combines D and R.
Let S be R^D.

$\lg S$ is the number of bits in each value.

Tradeoff between radix and digits: If you increase the radix, you'll have fewer digits, so you'll have fewer passes, but each pass will take longer time.

Optimal $R \approx \frac{\alpha N}{\ln N}$ (radix depends only on N, not S)

Substituting that in, time is $\Theta\left(\frac{N \lg S}{\lg N}\right)$ (doesn't have to be log base 2, can be other bases too)

If you think of it in terms of words, time is $\Theta\left(\frac{wW}{\lg W}\right)$, where w is bits in each word and W is number of words.

15 Bucket Sort

Bucket sort is a distribution sort (as opposed to a comparison-based sort). It works by distributing numbers into buckets.

Works well when the numbers being sorted are uniformly distributed.

Average case linear time.

1. Put into buckets based on first digit (you don't have to have 10 buckets, you can divide them up in any number of equal intervals).
2. Sort each bucket using bubblesort (other algorithms work too - you can even recursively use bucketsort).
3. Concatenate all the buckets together and you're done.

Pseudocode:

```
k <- Number of buckets we want
M <- max(A) - min(A)
# Initialize the buckets
for i in 0 to k-1:
  bucket B[i] = Empty
# Put each element into its bucket
for i in 1 to n:
  put A[i] into bucket B[floor(k * A[i] / M)]
# Sort buckets
for i in 0 to k-1:
  bubble sort bucket B[i]
# Concatenate the buckets
return B[0] ++ B[1] ++ ... ++ B[k-1]
```

Analysis

Time to initialize, fill up, and concatenate buckets is $\Theta(n)$
Time to sort buckets depends on data:

- If all n elements might be in a single bucket instead of being distributed, then $\Theta(n^2)$
- If $\sqrt{n}$ mediocre buckets each of size $\sqrt{n}$, then $\Theta(n^{\frac{3}{2}})$

On average, the largest bucket has $\sim \dfrac{\ln n}{\ln \ln n}$ elements.

Why Bubblesort? Because almost all buckets are small. Might even want a special sorting algorithm for each value of n.

16 Selection

Recorded videos seem to be optional since already went over it in class.

17 Graphs

Notation: Graph $G = (V, E)$, where V is set of vertices, E is set of edges, and $|E|$ is number of edges. **Note:** Sometimes people use E to mean number of edges.

Adjacency matrices and lists

Adjacency matrix: A 2D matrix where $A_{i,j}$ is the number of edges going from i to j (or the number of edges going from j to i, depending on preference).
Space required is $\Theta(n^2)$.

Adjacency list: An array where the indices represent vertices and the value at index i is a list of all the vertices adjacent to vertex i (maps vertices to their neighbors).
Space required is $\Theta(m + n)$, where m is number of edges. m is bounded by n^2 (assuming you can only have one edge from one vertex to another)

If the matrix is sparse (not too many edges), adjacency lists are preferred.

Time complexity comparison:

	Adjacency matrices	Adjacency lists
Check if two vertices adjacent	$\Theta(1)$	$O(n)$
Find all edges	$\Theta(n^2)$	$\Theta(m+n)$
Find Eulerian cycle	$\Theta(n^2)$	$\Theta(m)$

17.1 Eulerian Cycles

Terminology

Walk: A sequence of vertices and edges on a graph that you traverse.

Trail: A walk in which all edges are distinct.

Path: A trail in which all vertices (and edges) are distinct. TODO figure out the right definition, because by this definition, an Eulerian "path" wouldn't be a real path since vertices repeat

Eulerian path (a.k.a. Eulerian trail): A trail that visits every edge exactly once (may revisit vertices). The start and end vertices maybe different.

Eulerian cycle (a.k.a. Eulerian circuit, Eulerian tour): An Eulerian path which starts and ends at the same vertex, i.e. a cycle that uses each edge exactly once.

Degree: The degree of a vertex is how many edges are incident to it (equivalently, how many vertices are adjacent to it). A loop contributes 2 to the degree, one for each endpoint.

Strongly connected (like connectedness but for directed graphs): Every vertex is reachable from every other vertex.

Condition for having Eulerian cycle

An undirected, connected graph $G = (V, E)$ has an Eulerian cycle if and only if every vertex has even degree.

Similarly, for directed graphs, the graph must be *strongly* connected and the in degree of each vertex has to equal its out degree (need one edge to go to each vertex and another to go out from that vertex).

Condition for having Eulerian path

An undirected, connected graph $G = (V, E)$ has an Eulerian path if and only if exactly zero or two vertices have odd degree (not possible for only one vertex to have odd degree).

Similarly, for directed graphs, the graph must be *strongly* connected and either every the in degree equals the out degree for every vertex OR there is one vertex whose in degree is 1 greater than its out degree, there is one vertex whose in degree is 1 less than its out degree, and all the other vertices have equal in degrees and out degrees.

18 Spanning Trees

Tree: A connected, acyclic graph.

Spanning tree: A subset of the edges of the graph that includes all the vertices but has no cycles. We'll be talking only about undirected graphs.

Minimum spanning tree: A spanning tree where the sum of the edge weights is minimized (if you don't have edge weights, this means you need to have as few edges as possible).

Kruskal's algorithm and Prim's algorithm can be used to find an MST. Both are **greedy** (at each step, do whatever is locally optimal).

18.1 Kruskal's algorithm

Greedy algorithm that builds up the MST edge by edge.

```
edges <- sort edges by edge weight
T <- empty tree # our result
for edge in edges:
  if adding edge to T doesn't result in cycle:
    add edge to T
return T
```

This is $\Theta(E \log E)$ where E is number of edges.

18.2 Prim's algorithm

Greedy algorithm that grows the MST starting at an arbitrary vertex and keeps track of distance from starting vertex. Similar to Djikstra's algorithm, except we add the vertex closest to the current tree rather than the vertex closest to the source vertex.

1. Start with any vertex v.
2. Initialize array of distances for each vertex (initially ∞ for all of them).
3. Repeat the following until all vertices are in your tree:
   (a) Of the edges that connect v to another vertex not in the MST, pick the one that's closest to the tree.
   (b) Add that edge to the MST.
   (c) Update v to a new vertex (in the MST) whose distance from the starting vertex is minimized.
   (d) Update the distance and predecessor for the new v (the predecessor is the old v).

With a normal list that has 1s to represent unvisited vertices and 0s to represent visited vertices and an adjacency matrix to represent the graph itself, this is $\Theta(V^2)$ (better for dense graphs).

Alternatively, can use min heap acting as priority queue to store unvisited vertices, with runtime of $\Theta(E \log V)$ (better for sparse graphs).

See video for pseudocode of the min heap version.

19 Shortest paths: Dijkstra's Algorithm

Exactly the same as Prim's algorithm, except we want the vertex that's closest to the "root", not the vertex closest to the current tree (Dijkstra's adds previous distance to edge weight when checking if new edge smaller or not).

20 NP-Completeness

Classes:

- **P**: $O(n^k)$ with constant k (decision problems solvable in polynomial time)
- **E**: $O(b^n)$ with constant b (exponential time)

- **EXP/EXPTIME**: $O(e^{n^k})$ with constant k (*also* exponential time)
- **NP**: Problems where YES answers are verifiable in polynomial time (can use a "certificate")
- **NP-hard**: Problems that are at least as hard as the hardest problems in NP

P is a subset of NP (no one knows if the reverse is true too)

NP is further partitioned into:

- **P**: The easy problems
- **NP-complete**: The hard problems (NP-complete is the intersection of NP and NP-hard)
- **NPI** (NP Intermediate): The intermediate problems

Recording 20.3 has an example of reduction (reducing 2-component coloring to 1-component coloring).

Recording 20.4 shows that Formula SAT and Circuit SAT are equivalent.

Proofs of NP-Completeness (Rec 20.5)

In order to show that some problem A is NP-Complete:

- Show that A is in NP
- For some NP-complete problem B, show that $B \leq_{\mathrm{P}} A$

The Structure of NP (Rec 20.6)

If a problem A is in P, then its complement $\bar{A}$ is also in P.

Graph isomorphism is in NP but not known if it's in Co-NP or not (probably in Co-NP though)

P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME $\subseteq$ EXPSPACE

Undecidable problems: Can't even solve (e.g. halting problem)

Recording 20.7 has equivalence of decision and optimization for coloring problem

21 Parallel Computing

21.1 Parallel Computing Models

Two types of parallel computers: direct connection machines and shared memory machines.

Direct connection machines

Processors connected to neighbors.

Shared memory machines

Different models:

1. Exclusive Read, Exclusive Write (EREW): No two processors can read from or write to the same location at the same time.
2. Concurrent Read, Exclusive Write (CREW)
3. Concurrent Read, Concurrent Write (CRCW)
   Need some kind of rule for concurrent writes, one of the following:
   - All processors must write same value to location

- May write different values but only one is actually written. Which value to write could be chosen in one of the following ways:
  - Random
  - Arbitrary (machine can choose anything, not the same as random)
  - Priority
- May write many values and all are written in some order. Again, need rule to determine order (random, arbitrary, or priority)

21.2 Parallel Algorithms

Define $T_P(N)$ to be the time for P processors to solve problem of size N

Speedup ($S_P(N)$): Sequential time divided by parallel time $\left(\dfrac{T_1(N)}{T_P(N)}\right)$

Efficiency ($E_P(N)$): Speedup divided by number of processors $\left(\dfrac{S_P(N)}{P} = \dfrac{T_1(N)}{PT_P(N)}\right)$

An algorithm is **efficient** if $E_P(N) = \Theta(1)$

Lemma: $T_P(N) \leq T_1(N) \leq PT_P(N)$

Corollary: $1 \leq S_P(N) \leq P$

Corollary: $\dfrac{1}{P} \leq E_P(N) \leq 1$

General phenomenon: When $N = P$ the parallel algorithm is fast but inefficient, but when $N >> P$ the parallel algorithm is fast and efficient.

Summing

Video has a way to make the parallel algorithm efficient about 3 minutes before video ends.

22 Parallel Merging

Merging 2 lists of sizes M and N on a sequential computer takes $M + N - 1$ comparisons ($2N - 1$ if same size)

Merging with 2 processors

One processor starts comparing elements at the beginning, the other starts comparing elements at the end, meet in the middle.

Works on machine with Exclusive Read Exclusive Write (EREW).

Takes $\frac{M+N}{2}$ comparisons (N if both lists are same size).

Speedup is $\dfrac{2N-1}{N} \approx 2 = P$
Efficiency is ~ 1

Unfortunately, this algorithm not easily generalizable to more processors.

Merging with N processors

This algorithm seems to require both lists being the same size.

Assign one processor to every element in one of the arrays, then have each processor do binary search to figure out where to place its assigned element.

Requires Concurrent Reads (CREW) because every processor will do binary search at the same time.

Takes $\sim \lg N$ comparisons.

Speedup: $\sim \dfrac{2P}{\lg P}$, Efficiency: $\sim \dfrac{2}{\lg P}$

Merging when $P < N$

Watch Rec 22.1 around 13:00

Requires Concurrent Read Exclusive Write (CREW) machine.

Algorithm:

1. In both arrays, mark elements $\left\lfloor \dfrac{2N}{P} \right\rfloor, 2\left\lfloor \dfrac{2N}{P} \right\rfloor, 3\left\lfloor \dfrac{2N}{P} \right\rfloor, \ldots$ (these elements are sort of separators creating $\frac{P}{2}$ groups of elements in each array)
2. In parallel, binary search each marked element into the other list, creating about P small merging problems. This takes $\sim \lg N$ comparisons.
3. In parallel, do each small merging problem, assigning two processors to each problem. This takes $\sim \frac{2N}{P}$ comparisons.

Total comparisons: $\sim \dfrac{2N}{P} + \lg N$

To be efficient, need the number of comparisons to be $O(\dfrac{2N}{P})$, which requires $N = \Omega(P \lg P)$.

22.1 Parallel Mergesort

At level i, do $2^i - 1$ comparisons.

Need $\frac{N}{2}$ processors.

Total comparisons: $\sim 2N$

Speedup: $\sim \dfrac{\lg P}{2}$, Efficiency: $\sim \dfrac{\lg P}{2P}$

Mergesort for Large N (first try)

Assume $N \geq P$

1. Assign $\frac{N}{P}$ numbers per processor. Each processor sorts its numbers sequentially ($\dfrac{N}{P} \lg \left(\dfrac{N}{P}\right)$ for each processor).
2. In parallel, pairwise merge lists until there is one sorted list ($\frac{2N}{P}$ for first level from bottom, $\frac{4N}{P}$ for second level from bottom, $\frac{8N}{P}$ for next level, and so on).

Total comparisons for merges (not counting sorting at the bottom): $\sim 2N$

Total comparisons (including sorting at bottom): $\sim \dfrac{N \lg N}{P} + 2N$

To be efficient, we need $N = 2^{\Omega(P)}$

N needs to be exponential in the number of processors, so this is only efficient for very very large N. Can improve by using parallel merges:

Mergesort using Parallel Merge

Assume $P = \frac{N}{2}$

The first level from the bottom requires 2 processors and does $\lg 2$ comparisons, second level requires 4 processors and does $\lg 4$ comparisons, third level requires 8 and does $\lg 8$ comparisons, and so on.

Total comparisons: $\sim \dfrac{(\lg N)^2}{2}$

Speedup: $\sim \dfrac{4P}{\lg P}$, Efficiency: $\sim \dfrac{4}{\lg P}$

Mergesort for Large N (second try)

Assume $N \geq P$

Same as before, but this time, merge in parallel (requires CREW)

1. Assign $\frac{N}{P}$ numbers per processor. Each processor sorts its numbers sequentially ($\dfrac{N}{P} \lg \left(\dfrac{N}{P}\right)$ for each processor).
2. In parallel, pairwise merge lists until there is one sorted list ($\frac{N}{P} + \lg 2$ for first level from bottom, $\frac{N}{P} + \lg 4$ for second level from bottom, $\frac{N}{P} + \lg 8$ for next level, and so on).

Total comparisons for merges (not counting sorting at the bottom): $\sim \dfrac{N \lg P}{P} + \dfrac{(\lg P)^2}{2}$

Total comparisons (including sorting at bottom): $\sim \dfrac{N \lg N}{P} + \dfrac{(\lg P)^2}{2}$

To be efficient, need $N = \Omega(P \lg P)$ (doesn't need to be huge, unlike before)

Sorting with Optimal Merging Algorithm

Can merge two lists each of size N on CREW PRAM with $P \leq N$ processors in $\sim 2\frac{N}{P} + \lg \lg P$ comparison steps.

So can sort on CREW PRAM with $\sim \dfrac{N \lg N}{P} + (\lg P) \lg \lg P$ comparison steps.

This algorithm is efficient for $N = \Omega(P \lg \lg P)$

Pipelining

One processor can start merging as soon as processors on bottom levels are done with elements at the start, even if they're not fully finished.

With pipelining, mergesort is $\Theta\left(\dfrac{N \lg N}{P}\right)$ if $N \geq P$

Efficient for $N = \Omega(P)$

Extra Stuff

1 Algorithm Complexities

Algorithm	Worst case	Average case	Best case	Comments
Quicksort	$O(n^2)$	$O(n \lg n)$	$O(n \lg n)$	
Counting sort		$\Theta(n+k)$		
Radix sort	$\Theta(d(n+r))$	$\Theta(d(n+r))$	$\Theta(d(n+r))$	Optimal: $\Theta\left(\frac{n \lg S}{\lg n}\right)$
Bucket sort	$O(n^2)$	$O(n+k)$	$O(n+k)$	