Oliver Song
6.046 Exam 1 crib sheet

Bachmann-Landau notations (Big-O shit)

f(n) ∈ ***Θ****(g(n))* - *f* is bounded both above and below asymptotically
or: $g(n) * k_1 \le f(n) \le g(n) * k_2$, for some positive k_1 k_2
f(n) ∈ ***O****(g(n))* - *f* is bounded above by *g* (up to a constant factor)
or: $|f(n)| \le g(n) * k$, for some *k*
f(n) ∈ ***Ω****(g(n))* - *f* is bounded below by g (up to a constant factor)
or: $f(n) \ge g(n) * k$, for some positive *k*
f(n) ∈ ***o****(g(n))* - *f* is dominated by *g* asymptotically
or: $|f(n)| \le |g(n)| * \epsilon$, for all ∈
f(n) ∈ ***ω****(g(n))* - *f* dominates g asymptotically
or: $|f(n)| \ge |g(n)| * \epsilon$, for all ∈

Recurrences

Master theorem: Applies to relations of this form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \ge 1,\ b > 1$$

n is the size of the problem. **a** is the number of subproblems in the recursion. **n/b** is the size of each subproblem. ***f (n)*** is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

Case 1*:* If $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for $\epsilon > 0$ then: $T(n) = \Theta\left(n^{\log_b a}\right)$
Case 2*:* If true for some k ≥ 0 that: $f(n) = \Theta\left(n^{\log_b a}\log^k n\right)$ *then:* $T(n) = \Theta\left(n^{\log_b a}\log^{k+1} n\right)$
Case 3*:* If true that $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ for $\epsilon > 0$ AND also true that: $af\left(\frac{n}{b}\right) \le cf(n)$ for *c* > 1, sufficiently large *n*: $T(n) = \Theta(f(n))$

Iteratively / with substitution

Substitute in step by step until you notice a pattern. Then use that pattern to extract the solution.
Problem: T(n) = T(n-1) + n
Solution: T(n) = T(n-1) + n = T(n-2) + n-1 + n = ... = 1+ 2 + ... + n = (n+1)n/2 = Θ(n^2)

Divide & Conquer

Split into subproblems and recursively solve each subproblem.
Subproblems must be much smaller than original problem.
Examples are: mergesort, quicksort

Fast Fourier Transform (FFT)

Used to reduce polynomial multiply time to Θ(nlogn)

1) Choose n roots of unity as points
2) [Evaluation] DFT to go from coefficient form to point-wise form **O(nlogn)**
3) Point wise multiplication in **O(n)** time
4) [Interpolation] Inverse DFT to go back from point-wise **O(nlogn)**

How to get the polynomial in the first place??? (pset 2 #1)

Problem: Let A be a finite set with n distinct integer elements in the range of [1, 10n]. Give an efficient algorithm that runs in time O(n log n) to find whether there is some triple of three distinct elements
(x, y, z) such that z = x + y, z = x + y + 1, or z = x + y - 1.
Solution: Create a polynomial P(x) from the set:
$P(x) = a_1x^1 + a_2x^2 + ... + a_nx^n$
where $a_i = 1$ if element i ∈ A. Square the polynomial with FFT to obtain a new polynomial

$$\begin{aligned} P(x)^2 &= Q(x) \\ &= q_0x^{b_0} + q_1x^{b_1} + ... + q_{2n}x^{b_n} \end{aligned}$$

The values b_k will be exactly the values of $a_i + a_j$! This resulting polynomial's coefficients q_k will be non-zero positive integers if there exists a pair of elements i, j ∈ A such that k = i + j. The resulting polynomial will have up to 20n elements, but we only need to check the first 10n.

Randomization

Basic idea is that randomization removes bias from inputs, and increases average algorithm efficiency (avoid worst case scenarios)

How to do probabilistic analysis (based on hiring problem)

Use indicator random variables:
Instead of assigning one variable to count number of occurrences, assign many variables for each occurrence. X_i is the event where candidate i is hired (1 if hired, 0 if not). Target X = sum of all Xi's. So you need the probability per candidate.
E[X_i] = Pr {candidate i is hired}
Happens when candidate i is better than 1 through i-1. Candidates arrive randomly, so all are equally liked to be best. **Candidate i has a 1/i chance of being better than 1 through i-1.**

$$\begin{aligned} &= \sum_{i=1}^{n} \mathrm{E}[X_i] \\ &= \sum_{i=1}^{n} 1/i \\ &= \ln n + O(1) \end{aligned}$$

Dynamic Programming

Requires: *Optimal substructure* and overlapping *subproblems*
Substructure*:* can subproblem solutions be combined?
Overlapping Subproblems*:* want many redundant subproblems
Recursion: Create the recursive equation and write the algorithm
Memoization: keep tabs on subproblems already solved

Multivariate Polynomials (what)

The problem: given two polynomials, find out if P≡Q, or P-Q=0
Definition 1: the degree of multivariable poly is the **highest sum of exponents**: eg degree of $x^2y^3z+x^6yz$ is 8
What is different? number of roots can be infinite, expanding into monomials can result in exponential terms growth

The general idea is to subtract (O(n) time) then test if it is 0. How? Randomly sample a shit ton of points and see if they all evaluate to 0. If so, then it is very, very likely that it is 0. How likely? That is bounded by **d/|S|** (d = degree, S = set of numbers sampled). This is Shwartz-Zippel Lemma.

Man on the moon problem (AKA string equality)

The idea is so treat strings like polynomials and then do polynomial equality.
Let a = $a_0a_1a_2$ ... a_n and b = $b_0b_1b_2$ ... b_n (everything is a 0 or 1).
The polynomial is: $a(x) = \sum_{i=0}^{n} a_i x^i$ and the same for b.
Size limitations: we can't have that much stuff transmitted so we need to have some sort of compression. So we take all the bits and sample them mod p, some prime number.

Honestly I don't know why they don't just compare md5 hashes.

Key Proofs and Relations

1+2+...+n-1+n == **(n+1)*n/2**
$2^{n-1}+2^{n-2}+2^{n-3}+...+2^2+2^1$ == **2^n - 1**
$a^n+a^{n-1}+a^{n-2}+...+a^2+a^1+1$ == **$(a^{n+1}-1)/(a-1)$**

$$\begin{aligned} &= \sum_{i=1}^{n} 1/i \quad \text{(by equation (5.4))} \\ &= \ln n + O(1) \quad \text{(by equation (A.7))} \end{aligned}$$

Loop invariants

A statement that holds no matter what, every iteration
Initialization- true prior to the first iteration
Maintenance- true before an iteration of the loop, remains true before next iteration
Termination- When loop terminates, invariant gives us a useful property that helps show that the algorithm is correct

Algorithms

Convex hull: Find points with highest y intercept of line between shapes and points with lowest y intercept of line and connect them.

Greedy recursive for selection: choose earliest to finish

s=start times f=finish times k=current index n=length of schedule

```
RECURSIVE-ACTIVITY-SELECTOR(s,f,k,n):
m=k+1
while m<=n and s[m] < f[k]  //find first
finishing activity
     m=m+1
if m<=n
      return {a_m} U RECURSIVE-ACTIVITY-
SELECTOR(s,f,m,n)
else return 0
```

Running time **O(n)**, if activities already sorted. Else **O(n log n)**

Greedy iterative for selection: choose earliest to finish

s=start times f=finish times k=current index n=length of schedule

```
GREEDY-ACTIVITY-SELECTOR(s,f):
n = s.length
A = {a_1}
k=1
for m = 2 to n:
     if s[m] >= f[k]
           A = A U {a_m}
           k = m
return A
```

Running time O(n)

SELECT median finding: recursive partitioning

A[p..r] = array we are analyzing i=ith smallest element to return

```
RANDOMIZED-SELECT(A,p,r,i):
if p == r:
     return A[p]
q = RANDOMIZED-PARTITION(A,p,r) //random pivot
k = q - p + 1
if i == k:  //the pivot is the answer
     return A[q]
elif i < k:
     return RANDOMIZED-SELECT(A,p,q-1,i)
else:
     return RANDOMIZED-SELECT(A,q+1,r,i-k)
```

Running time O(n)

Quick sort

1) pick pivot randomly (best is randomized)
2) Put all points higher than pivot on the right, all lower on the left
3) Recursively sort these two sides

Running time O(n log n)

Randomized Hiring problem

1) choose k random candidates to use as trial period
2) For the next n-k candidates, if any > max (1 to k) then auto choose that one to hire

1/e probability of hiring best candidate

Industry assembly line DP problem

i=station # j=line # $a_{i,j}$=assembly time $t_{i,j}$=transfer time e_i=entry time x_i=exit time n=number of stations f= q= f= q*=*

```
FASTEST-WAY(a,t,e,x,n):
f_1[1] = e_1 + a_1,1
f_2[1] = e_2 + a_2,1
for j=2 to n:
     if f_1[j-1] + a_1,j ≤ f_2[j-1] + t_2,j-1 + a_1,j:
           f_1[j] = f_1[j-1] + a_1,j
           q_1[j] = 1
     else:
           f_1[j] = f_2[j-1] + t_2,j-1 + a_1,j
           q_1[j] = 2
     if f_2[j-1]+a_2,j ≤ f_1[j-1] + t_1,j-1 + a_2,j:
           f_2[j] = f_2[j-1]+a_2,j
           q_2[j] = 2
     else:
           f_2[j] = f_1[j-1]+t_1,j-1 + a_2,j
           q_2[j] = 1
if f_1[n] + x_1 ≤ f_2[n] + x_2:
     f* = f_1[n] + x_1
     q* = 1
else:
     f*, q* = f_2[n] + x_2 , 2
```

Running time O(n)

Bellman ford pathfinding

Works on negative edges too O(V*E)

```
1. Assign all nodes to infinity
2. for i from 1 to size(vertices)-1:
    for each edge uv in edges:
        // uv is the edge from u to v
        u := uv.source
        v := uv.destination
        if u.distance + uv.weight < v.distance:
            v.distance := u.distance + uv.weight
            v.predecessor := u
```

3. Check for negative cycles

Djikstra pathfinding

No negative edges $O(V^2)$

1. Assign all nodes infinity
2. Mark all nodes unvisited. Set the initial node as current.
3. Consider all unvisited neighbors and calculate their tentative distances, current distance + estimate
4. After considering all neighbors, mark current node visited
5. Stop when destination is visited
6. Go to next closest neighbor, loop to step 3

Floyd-warshall find all shortest paths

Can do negative edges, no neg cycles $O(V^3)$

shortestPath(i,j,0) = w(i,j)

shortestPath(i,j,k) = min(shortestPath(i,j,k-1), shortestPath(i,k,k-1) + shortestPath(k,j,k-1))

Johnson's find all shortest paths

$O(V^2 \log V + VE)$

Can do negative edges, no neg cycles, better to spare large graphs

1. add node q to graph. connect by **zero-weight edges** to all nodes
2. Bellman-ford to find distance from q to every vertex shortest path
3. Reweigh edges with bellman-ford results- an edge from u to v, having length w(u,v), is given the new length w(u,v) + h(u) – h(v)
4. Remove q, use Dijkstra to find paths from all nodes to each other

MST grow Kruskall

O(E log E), greedy algorithm

1. create a forest F (a set of trees), where each vertex in the graph is a separate tree
2. create a set S containing all the edges in the graph
3. while S is nonempty and F is not yet spanning
    a. remove an edge with minimum weight from S
    b. if that edge connects two different trees, then add it to the forest, combining two trees into a single tree otherwise discard that edge.
4. At the termination of the algorithm, the forest has only one component and forms a minimum spanning tree of the graph.

MST grow Prim

$O(V^2)$, but O(E+VlogV) with Fibonacci heap, greedy, undirected

1. create a tree containing one vertex chosen randomly from graph
2. create a set containing all the edges in the graph
3. loop until every edge in the set connects two vertices in the tree
    a. remove from the set an edge with minimum weight that connects a vertex in the tree with a vertex not in the tree
    b. add that edge to the tree