Announcements

- Homework 2 Due on Friday

Last Time

- Shortest Paths in Graph with Negative Edge Weights
  - Negative weight cycles
    - might mean no shortest paths
  - `Bellman-Ford`
    - Computes shortest path length with ≤ k edges for various k
    - If no negative weight cycles, take k = |V|-1 to get correct distances.

Algorithm

```
Bellman-Ford(G,s,ℓ)
  dist_0(v) ← ∞ for all v
    //cant reach
  dist_0(s) ← 0
  For k = 1 to n
    For w ∈ V
      dist_k(w)←min(dist_k-1(v)+ℓ(v,w))
    dist_k(s) ← min(dist_k(s),0)
      // s has the trivial path
```

Result

Proposition: If $n \geq |V|-1$ and if G has no negative weight cycles, then for all v,

$$dist(v) = dist_n(v).$$

`Bellman-Ford` computes shortest paths if no negative weight cycles in time $O(|V||E|)$.

Today

- Detecting Negative Weight Cycles.
- Shortest Paths in DAGs.
- Divide & Conquer

Detecting Negative Cycles

If there are no negative weight cycles, Bellman-Ford computes shortest paths (and they might not exist otherwise).

How do we know whether or not there are any?

Cycle Detection

Proposition: For any $n \geq |V| - 1$, there are no negative weight cycles reachable from s if and only if for every $v \in V$

$$dist_n(v) = dist_{n+1}(v)$$

- Detect by running one more round of Bellman-Ford.
- Need to see if *any* v's distance changes.

Proof of “Only If”

- Suppose no negative weight cycles.
- For any $n \geq |V| - 1$, $dist_n(v) = dist(v)$.
- So

$$dist_n(v) = dist(v) = dist_{n+1}(v)$$

Proof of "If"

Suppose $\text{dist}_n(v) = \text{dist}_{n+1}(v)$ for all v.

$$\begin{aligned}\text{dist}_{n+2}(w) &= \min_{(v,w)\in E}(\text{dist}_{n+1}(v) + \ell(v,w)) \\ &= \min_{(v,w)\in E}(\text{dist}_{n}(v) + \ell(v,w)) \\ &= \text{dist}_{n+1}(w).\end{aligned}$$

So

$$\text{dist}_n(v) = \text{dist}_{n+1}(v) = \text{dist}_{n+2}(v) = \text{dist}_{n+3}(v) = \ldots$$

But if there were a negative weight cycle, distances would decrease eventually.

Alternative Proof

- Assume $dist_n(v) = dist_{n+1}(v) = d(v)$ for all v.
- $d(w) = \min(d(v)+\ell(v,w))$.
  - $d(w) \leq d(v)+\ell(v,w)$ for all $(v,w) \in E$
  - $\ell(v,w) \geq d(w) - d(v)$
- Given cycle $v_1,v_2,v_3,\ldots,v_m$ total length of cycle is

$$\begin{aligned} & \ell(v_1, v_2) + \ell(v_2, v_3) + \ldots + \ell(v_m, v_1) \\ \geq\ & -d(v_1) + d(v_2) - d(v_2) + d(v_3) - \ldots - d(v_m) + d(v_1) = 0. \end{aligned}$$

Potential Function

- Let $\ell'(v,w) = \ell(v,w) - d(v) + d(w) \geq 0$
  - Imagine somebody lending you d(w) when you arrive at w, but having to pay it back when you leave.
- For any s-t path P, $s, v_1, v_2, \ldots, t$

$$\begin{aligned}\ell'(P) &= \ell'(s,v_1) + \ell'(v_1,v_2) + \ldots + \ell'(v_m,t) \\ &= \ell(s,v_1) + \ell(v_1,v_2) + \ldots + \ell(v_m,t) \\ &\quad - d(s) + d(v_1) - d(v_1) + d(v_2) + \ldots - d(v_m) + d(t) \\ &= \ell(P) - d(s) + d(t).\end{aligned}$$

- Shortest paths same. Non-negative edges.

Shortest Paths in DAGs

We saw that shortest paths is harder when we needed to deal with negative weight cycles. For general graphs, we needed to use Bellman-Ford, which is much slower than our other algorithms.

One way to avoid this was to make edge weights non-negative. In this case, we could use Dijkstra.

Another way to get rid of negative weight cycles, is to get rid of cycles. If G is a DAG, there are better algorithms.

Fundamental Shortest Paths Formula

$$\text{For } w \neq s,$$
$$\text{dist}(w) = \min_{(v,w) \in E} \text{dist}(v) + \ell(v, w).$$

Hard to apply in general because there's no order to solve equations in.

DAG gives topological order!

Algorithm

Runtime
O(|V|+|E|)

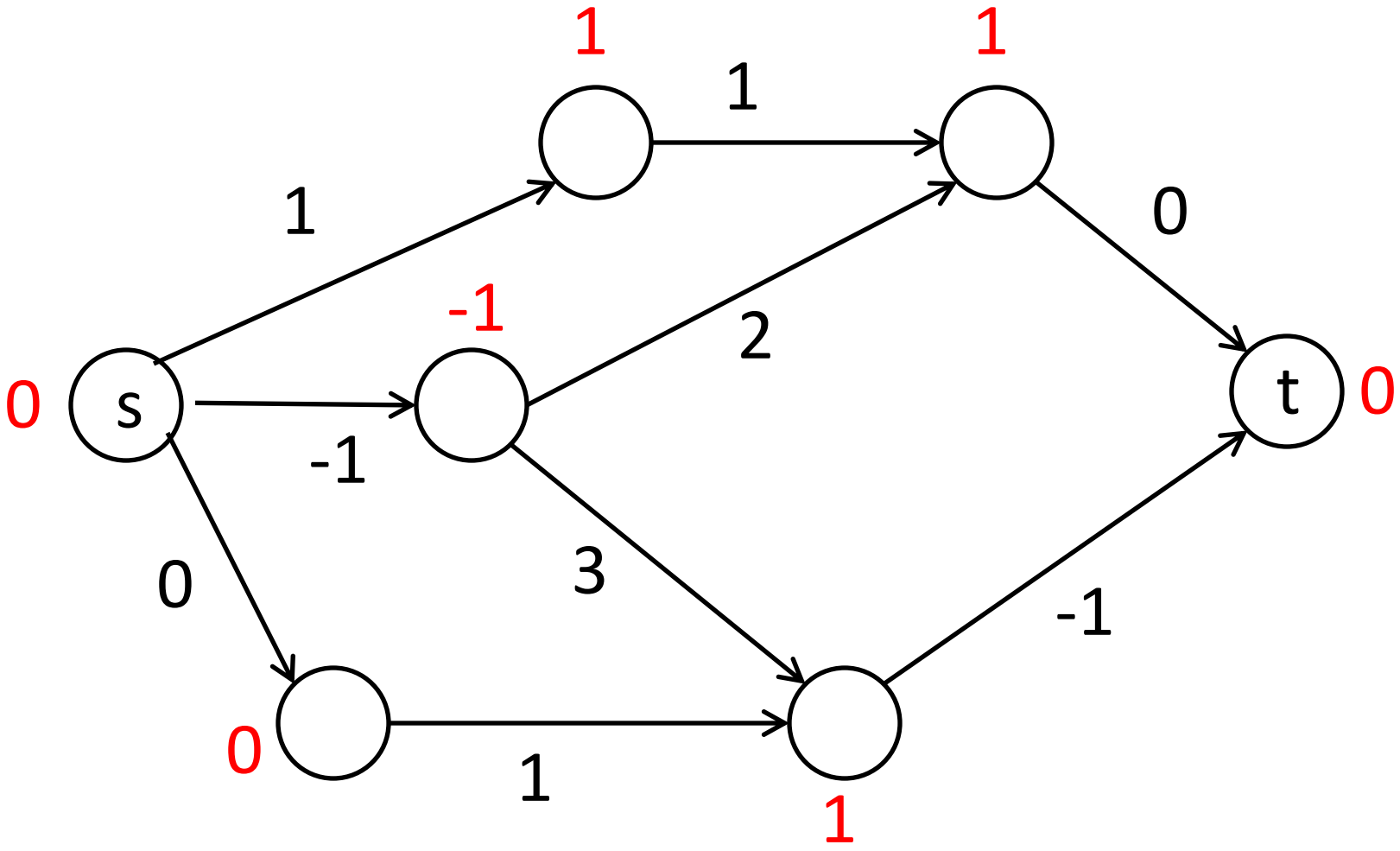
```
ShortestPathsInDAGs(G,s,ℓ)
  TopologicalSort(G)
  For w ∈ V in topological order
    If w = s, dist(w) ← 0
    Else
     dist(w)←min(dist(v)+ℓ(v,w))
\\ dist(v) for all upstream v
  already computed
```

O(|V+|E|)

O(|V|) total

O(|E|) total

Example



Shortest Path Algorithms Summary

Unit Weights: `Breadth First Search`
O(|V|+|E|)

Non-negative Weights: `Dijkstra`
O(|V|log|V|+|E|)

Arbitrary Weights: `Bellman-Ford` O(|V||E|)

Arbitrary Weights, graph is a DAG:
`Shortest-Paths-In-DAGs` O(|V|+|E|)

Divide & Conquer (Ch 2)

- General Technique
- Master Theorem
- Karatsuba Multiplication
- Strassen's Algorithm
- Merge Sort
- Order Statistics
- Binary Search
- Closest Pair of Points

Divide and Conquer

This is the first of our three major algorithmic techniques.

1. Break problem into pieces
2. Solve pieces recursively
3. Recombine pieces to get answer

Example: Integer Multiplication

Problem: Given two n-bit numbers find their product.

Naïve Algorithm: Schoolboy multiplication. The binary version of the technique that you probably learned in elementary school.

Schoolboy Multiplication

$$
\begin{array}{rl}
 & a_1 \quad a_2 \quad a_3 \quad \ldots a_{n-1} \quad a_n \\
\times & b_1 \quad b_2 \quad b_3 \quad \ldots b_{n-1} \quad b_n \\
\hline
 & a_1b_1 \; a_2b_1 \; a_3b_1 \ldots a_{n-1}b_1 \; a_nb_1 \\
 & a_1b_2 \; a_2b_2 \; a_3b_2 \; a_4b_2 \ldots a_nb_2 \; 0 \\
 & \ldots \quad \ldots \quad \ldots \quad \ldots \quad \ldots \; \ldots \quad \ldots \\
+ & a_1b_n \; a_2b_n \; a_3b_n \ldots \; a_nb_n \; 0 \quad 0 \quad \ldots 0 \quad 0 \\
\hline
\end{array}
$$

ANSWER

Question: Runtime

What is the asymptotic runtime of the schoolboy algorithm?

A) $O(n)$

B) $O(n \log(n))$

C) $O(n^2)$

D) $O(n^3)$

E) $O(2^n)$

Need to write down $O(n^2)$ bits of numbers to add. Addition can be done in linear time.

Two Digit Multiplication

```
              a         b
            x c         d
              ----------
              ad       bd
+ ac          bc        0
  ----------------------
  ac   ad+bc       bd
```

Two-Digit Multiplication

(ab) ·(cd) = [ac][bc+ad][bd]

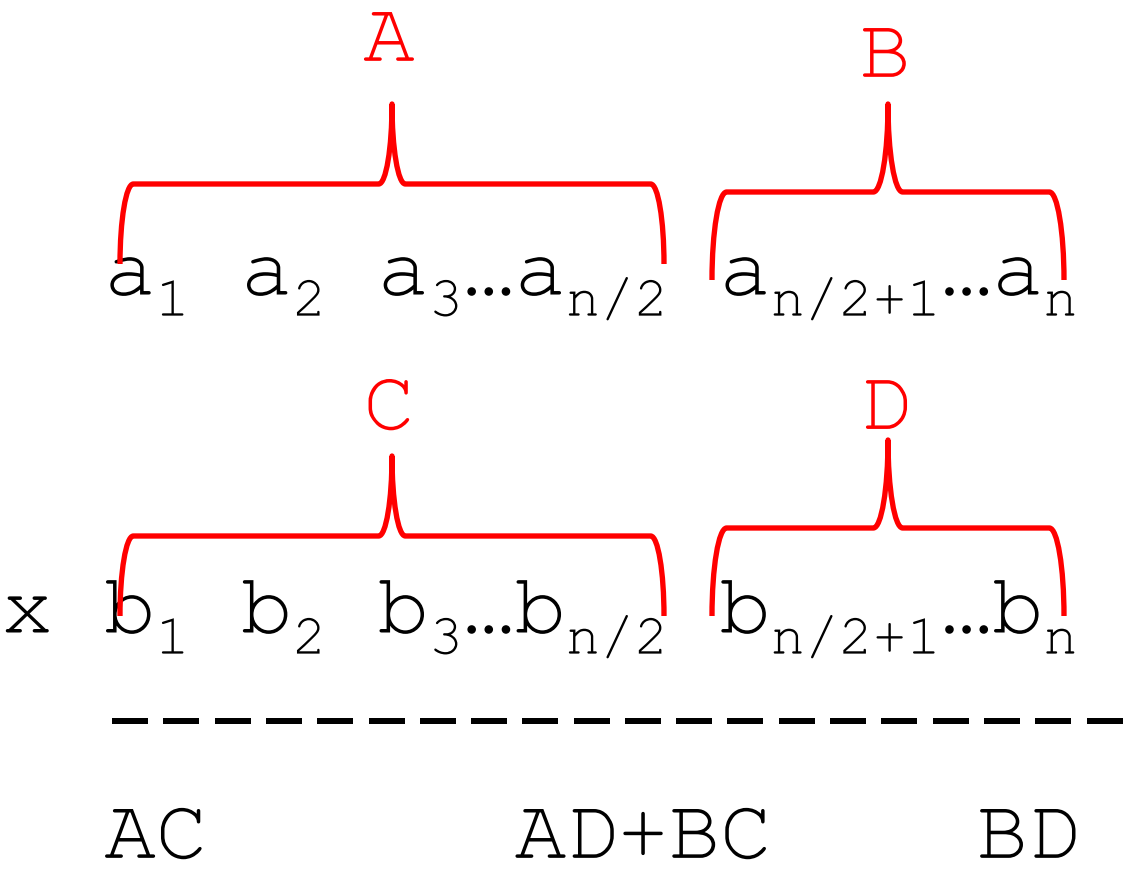
Requires 4 one-digit multiplications and one addition.

Trick: Compute ac, bd, (a+b)(c+d).
Note that bc+ad = (a+b)(c+d) – ac – bd.

Requires 3 one-digit multiplications and 4 addition/subtractions.

How can we apply this to larger problems?

Larger Base


A
B
a_1 a_2 $a_3 \ldots a_{n/2}$ $a_{n/2+1} \ldots a_n$
C
D
x b_1 b_2 $b_3 \ldots b_{n/2}$ $b_{n/2+1} \ldots b_n$
AC
AD+BC
BD

Formally

Want to multiply N and M:

1. Let $X \approx \sqrt{(N+M)}$ be a power of 2.
2. Write $N = AX+B$, $M = CX+D$
   - This can be done by just taking the high and low bits.
3. $N \cdot M = AC \cdot X^2+(AD+BC)X+BD$
   $= AC \cdot X^2+[(A+B)(C+D)-AC-BD]X + BD$
   - The multiplications by X are just bit shifts.

Improved Multiplication

```
ImprovedMult(N,M)
   Let X be a power of 2^⌊log(N+M)/2⌋
   Write N = AX + B, M = CX + D
   P1 ← Product(A,C)
   P2 ← Product(B,D)
   P3 ← Product(A+B,C+D)
   Return P1X^2 + [P3-P1-P2]X + P3
```

O(n)

O(n²)

O(n)

Runtime: $O(n^2)$. No asymptotic improvement!

More Detailed Analysis

This algorithm shows no *asymptotic* improvement, but it is better.

To analyze this, lets suppose that computing the product of two n-bit numbers using the schoolboy algorithm takes n^2 time.

Improved Multiplication

```
ImprovedMult(N,M)
   Let X be a power of 2^⌊log(N+M)/2⌋
   Write N = AX + B, M = CX + D
   P1 ← Product(A,C)
   P2 ← Product(B,D)
   P3 ← Product(A+B,C+D)
   Return P1X^2 + [P3-P1-P2]X + P3
```

O(n)

3(n/2)²

O(n)

Runtime: $(3/4)n^2+O(n)$.

Better than n^2!

Further Improvements

So this trick does help. Saving a multiplication at the cost of a few extra additions is a big deal, when multiplications are $O(n^2)$ and additions are $O(n)$.

Can we do better?

Yes. Our algorithm is still using schoolboy multiplication to do the smaller multiplications. We can instead use our faster algorithm.

Further Improvement

```
KaratsubaMult(N,M)
   If N+M<99, Return Product(N,M)
   Let X be a power of 2^⌊log(N+M)/2⌋
   Write N = AX + B, M = CX + D

   P1 ← Product(A,C)
   P1 ← KaratsubaMult(A,C)
   P2 ← Product(B,D)
   P2 ← KaratsubaMult(B,D)
   P3 ← Product(A+B,C+D)
   P3 ← KaratsubaMult(A+B,C+D)
   Return P1X^2 + [P3-P1-P2]X + P3
```

O(n)

???

O(n)