

CS 341 — Algorithms

Kevin James

Winter 2015

Contents

1	Solving Recurrences	2
2	Algorithm Design Techniques	4
2.1	Divide and Conquer	4
2.1.1	Multiplying Large Numbers	5
3	Selection	6
3.0.2	Pivot Selection	6
3.1	Greedy Algorithms	8
3.1.1	Interval Coloring	8
3.2	Pairing Algorithms	8
3.2.1	Stable Marriage	8
3.2.2	Gale and Shapley’s Algorithm	8
4	Dynamic Programming	9
5	Graph Problems	9
5.1	Shortest Paths	9
5.1.1	Correctness Proof	9
5.1.2	All-Pairs Shortest Paths	10
5.2	P vs NP	10
5.2.1	P-Problems	10
5.2.2	NP-Problems	11
5.2.3	NPC-Problems	11

1 Solving Recurrences

Example 1.1. *Determine the runtime of mergesort.*

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

which we can solve to find

$$T(n) \in \Theta(n \log n)$$

Definition 1.1. *Stoogesort:*

```
def stoogesort(A[1..n]):
    if n <= 1:
        return
    if A[1] > A[2]:
        swap(A[1], A[2])
    stoogesort(A[1..⌊2n/3⌋])
    stoogesort(A[⌊n/3⌋+1..n])
    stoogesort(A[1..⌊2n/3⌋])
```

Example 1.2. *Let $T(n)$ be the runtime of stoogesort on n elements. Then*

$$T(n) = \begin{cases} 3T\left(\frac{2n}{3}\right) + \Theta(1) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

There are three methods for solving this recurrence:

1. Recursion Tree method: Expand for k iterations to get a tree of terms, set k to reach the base case, sum across rows, then over all levels.
2. Master method: Look up the answer. The Master Theorem: Let $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ if $n > n_0$ and c if $n = n_0$. Set $d = \log_b a$ and pick a small constant $\varepsilon > 0$. Case 1: $f(n) = O(n^{d-\varepsilon}) \implies T(n) = \Theta(n^d)$. Case 2: $f(n) = \Theta(n^d) \implies T(n) = \Theta(n^d \log n)$. Case 3: $\lim_{n \rightarrow \infty} f(n)/n^{d+\varepsilon} = \infty \implies T(n) = \Theta(f(n))$.
3. Substitution method: Guess the form of the solution $T(n) \leq x$, then verify your guess by induction and fill in any constants.

Example 1.3.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n^2, 7 \\ T(n) &\leq cn^2 \\ n = 1 : T(n) &= 7 \leq cn^2, c \geq 7 \\ T\left(\frac{n}{2}\right) &\leq c\left(\frac{n}{2}\right)^2 \end{aligned}$$

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n^2 \\
&\leq 2c\left(\frac{n}{2}\right) + n^2 \\
&= \frac{2cn^2}{4} + n^2 \\
&= \left(\frac{c}{2} + 1\right)n^2 \\
&\leq cn^2, \frac{c}{2} + 1 \leq c, c \geq 2
\end{aligned}$$

We can then pick $c = 7$ and solve as

$$T(n) \leq 7n^2 \implies T(n) \in O(n^2)$$

We can also note that $T(n) \geq n^2$, so $T(n) \in \Theta(n^2)$.

Example 1.4.

$$\begin{aligned}
T(n) &= 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 1, 1 \\
T(n) &\leq cn^2 \\
n = 1 : T(n) = 1 &\leq cn^2, c \geq 1 \\
T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) &\leq c\left\lfloor \frac{n}{2} \right\rfloor^2 \\
T(n) &= 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 1 \\
&\leq 3c\left\lfloor \frac{n}{2} \right\rfloor^2 + 4c\left\lfloor \frac{n}{4} \right\rfloor^2 + 1 \\
&\leq 3c\left(\frac{n}{2}\right)^2 + 4c\left(\frac{n}{4}\right)^2 + 1 \\
&= \left(\frac{3}{4} + \frac{4}{16}\right)cn^2 + 1 \\
&= cn^2 + 1
\end{aligned}$$

but since we could not get rid of the constant, we try

$$\begin{aligned}
T(n) &\leq cn^2 - c_0 \\
n = 1 : T(n) = 1 &\leq cn^2 - c_0, c \geq 1 + c_0 \\
T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) &\leq c\left\lfloor \frac{n}{2} \right\rfloor^2 - c_0 \\
T(n) &= 3T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 4T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + 1 \\
&\leq 3\left(c\left\lfloor \frac{n}{2} \right\rfloor^2 - c_0\right) + 4\left(c\left\lfloor \frac{n}{4} \right\rfloor^2 - c_0\right) + 1
\end{aligned}$$

$$\begin{aligned}
&\leq 3c\left(\frac{n}{2}\right)^2 - 3c_0 + 4c\left(\frac{n}{2}\right)^2 - 4c_0 + 1 \\
&= \left(\frac{3}{4} + \frac{4}{16}\right)cn^2 - 7c_0 + 1 \\
&= cn^2 - c_0, -7c_0 + 1 \leq -c_0, c_0 \geq \frac{1}{6}
\end{aligned}$$

Pick $c_0 = \frac{1}{6}, c = \frac{7}{6}$ and we see that

$$T(n) \leq \frac{7}{6}n^2 - \frac{1}{6} \implies T(n) \in O(n^2)$$

2 Algorithm Design Techniques

2.1 Divide and Conquer

Divide your problem into subproblems of the same type, then use recursion to solve each problem and combine the results.

Example 2.1. (Maxima): Given a set P of n points in 2D, we say point p dominates point q if and only if p has both a greater x and y value than q . We say point q is maximal if and only if $q \in P$ and no point in P dominates q . Find all maximal points.

Solutions:

- *Brute Force:* for each $q \in P$, check if no points dominate q . Total time: $\Theta(n^2)$
- *Divide and Conquer:* divide into two subarrays of size $\frac{n}{2}$.
- *Divide by Medians:* instead of dividing by size, divide by a median vertical line.

```

maxima(sorted[p_1..p_n]):
1. if n == 1: return p_1
2. [q_1..q_l] = maxima([p_1..p_{n/2}])
3. [s_1..s_m] = maxima([p_{n/2}..p_n])
4. i = 1
5. while q_i.y > s_1.y
6.   i += 1
7. return [q_1..q_l, s_1..s_m]

```

which is an $O(n \log n)$ algorithm.

Example 2.2. (Closest Pair): Given set P of n points in 2D, find a pair $p, q \in P$ such that these points have a smaller distance between them than any other pair of points in P , ie. $d(p, q) = \sqrt{(p.x - q.x)^2 + (p.y - q.y)^2}$.

Solutions:

- *Brute Force Algorithm:* $\Theta(n^2)$

- *Shamos' Algorithm: $\Theta(n \log^2 n)$. Note that if we refine the Shamos algorithm by pre-sorting P also by y -coordinate at the beginning, we find that our complexity becomes $O(n \log n)$.*

```
def bruteForce(P):
    distance = infinity
    for each p in P:
        for each q in P (q neq P):
            distance = min(distance, d(p,q))
    return distance

def shamos(P):
    if n <= 10:
        return bruteForce(P)
    x_m = median x_coordinate
    P_L = { p in P : p.x < x_m }
    P_R = { p in P : p.x > x_m }
    d_L = shamos(P_L)
    d_R = shamos(P_R)
    d = min(d_L, d_R)
    <p_1..p_m> = sorted_by(points in { p in P : x_m - d <= p.x <= x_m + d }, y)
    for i = 1 to m do
        j = i + 1
        while p_j.y <= p_i.y + d:
            d = min(d, d(p_i, p_j))
            j++
    return d
```

2.1.1 Multiplying Large Numbers

Example 2.3. *Given two n -bit numbers $A = a_{n-1}, a_{n-2}, \dots, a_0$, $b = b_{n-1}, b_{n-2}, \dots, b_0$ in binary, compute $AB = c_{n-1}, c_{n-2}, \dots, c_0$.*

Solution: *The “Elementary School” algorithm for this involves doing*

```

      1011
x   1101
-----
      1011
     1011
    1011
   -----
10001111
```

which is $O(n)$ shifts and $O(n)$ additions, thus making it an $O(n^2)$ algorithm.

The “Karatsuba and Ofman” algorithm involves a different process:

```
A' = [a_{n-1}..a_{n/2}]
A" = [a_{n/2 - 1}..a_0]
```

$A = [A' \dots A'']$

$B' = [b_{\{n-1\}} \dots b_{\{n/2\}}]$

$B'' = [b_{\{n/2 - 1\}} \dots b_0]$

$B = [B' \dots B'']$

$AB = A'B'2^n + (A'B'' + A''B')2^{(n/2)} + A''B''$

which gives us a complexity of $O(n^2)$.

3 Selection

Example 3.1. Given n numbers $a_1 \dots a_n$ and k , find the k^{th} smallest number.

Solutions:

- Method 0: sort and look up index k . $O(n \log n)$ time.
- Method 1 (selection sort variation): find minimum, remove, and repeat k times. $O(nk)$ time.
- Method 2 (heapsort variation): find minimum, remove, build heap, and repeat k times. $O(n + k \log n)$ time.
- Method 3 (quicksort variation [“quickselect”])

```
def quickselect([a_1..a_n], k):
    if n = 1:
        return a_1
    pick pivot x
    L = { a_i : a_i <= x }, l = sizeof(L)
    R = { a_i : a_i > x }
    if k <= l:
        return quickselect(L, k)
    else:
        return quickselect(R, k-1)
```

Analysis of quickselect: let $T(n)$ be the runtime on n elements. Then

$$T(n) \leq T(\max\{l, n - l\}) + O(n)$$

In the ideal case, $l = \frac{n}{2}$. In this case, we have (using the master theorem) $T(n) \in O(n)$. In the worst case, $l = 1$ or $l = n - 1$, which gives us $T(n) \in O(n^2)$. The challenge of quickselect, then, is determining a pivot selection strategy which brings us closer to the ideal case.

3.0.2 Pivot Selection

Selecting a pivot at random gives us an expected case near $O(n)$.

Randomized Analysis: l is equally likely to be $1, 2, \dots, n$. Thus $\text{prob}[\frac{n}{4} < l \leq \frac{3n}{4}] = \frac{1}{2}$. Expected number of iterations to get this case is 2. So then

$$T(n) \leq T\left(\frac{3n}{4}\right) + 2 * O(n)$$

which is an (expected) $O(n)$ runtime.

Unfortunately, this algorithm gives us only an expected runtime. We do not have a worst case upper bound specifying $O(n)$.

The Blum, Floyd, Prattt, Rivest, and Tarjan algorithm tries to deterministically find a good pivot. They find an approximate median by recursion: this is the “median of medians of five” algorithm.

```
def quickselect([a_1..a_n], k):
    if n = 1:
        return a_1

    // pivot selection in n time
    split a_1..a_n into groups G_1..G_{n/5} of five elements each
    for i = 1 to n/5:
        x_i = median of G_i
    x = quickselect([x_i..x_{n/5}], n/10)
    // end pivot selection

    L = { a_i : a_i <= x }, l = sizeof(L)
    R = { a_i : a_i > x }
    if k <= l:
        return quickselect(L, k)
    else:
        return quickselect(R, k-1)
```

Lemma 3.1.

$$\frac{3n}{10} \lesssim l \lesssim \frac{7n}{10}$$

Proof. How many groups G_i with $x_i \leq x$? $\frac{n}{10}$. For each such group, how many elements are less than or equal to x_i ? 3. Thus we have the number of elements less than or equal to x is greater than or equal to $\frac{3n}{10}$. Similarly, the number of elements greater than x is greater than or equivalent to $\frac{3n}{10}$. \square

By this lemma, we see that the final recursion lines in this algorithm are in $\leq T(\frac{7n}{10})$. We have the overall complexity of this algorithm as

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

which solves to $T(n) \in O(n)$.

3.1 Greedy Algorithms

Greedy Algorithms are usually used for optimization problems, where we find the solution by minimizing or maximizing some objective subject to some restraints. These algorithms incrementally build the solution by choosing what seems best at each step.

- Advantages: simple, fast
- Disadvantages: local maxima over global maxima

3.1.1 Interval Coloring

Example 3.2. *Given n intervals $[a_1, b_1], \dots, [a_n, b_n]$, assign each interval a color such that no two intervals of the same color intersect while minimizing the number of colors used.*

Greedy Algorithm Idea: scan intervals from left to right and use a new color only when needed.

Algorithm:

```
k = 0
for each interval [a_i, b_i] in increasing order of a:
    pick any color c in {1, ..., k} not within any intervals intersecting [a_i, b_i]
    if c exists:
        assign [a_i, b_i] color c
    else:
        assign [a_i, b_i] color k + 1
        k++
```

3.2 Pairing Algorithms

3.2.1 Stable Marriage

The **stable marriage** algorithm is stable if for all matched pairs $(C_1, E_1), (C_2, E_2)$ we don't have both C_1 prefers E_2 over E_1 and C_2 prefers C_1 over C_2 .

3.2.2 Gale and Shapley's Algorithm

```
while exist unmatched employer E
    pick C = next best candidate in E's preference list
    // E makes offer to C
    if C is unmatched or C prefers E over C's current employer E_0
        unmatch(C, E_0)
        match(C, E)
```

We can see that this algorithm is bounded by $O(n^2)$.

4 Dynamic Programming

Dynamic Programming can be used for optimization problems. We simply define a class of subproblems, derive a recursive formula to explore all choices at each step, and then evaluate that formula bottom-up using a table. This is successful when the number of total subproblems is not overly large.

5 Graph Problems

5.1 Shortest Paths

Given a directed graph with weights along each path, find the cheapest route between two points.

Special Cases:

- Unweighted (or, all are equal to 1): simply find the path which uses the fewest amount of edges (with BFS). Analysis: $O(n + m)$.
- DAG (directional acyclic graph): use dynamic programming. Subproblem: $\forall v \in V, \delta[v] =$ the weight of the shortest path from s to v . Our answer, then, is $\delta[t]$. Base Case: $\delta[s] = 0$. Recursive Formula: $\delta[v] = \min(\delta[u] + w(u, v), \{u : (u, v) \in E\})$. Note the evaluation order for the recursive formula is in the same order as a topological sort over the edges. Analysis: $O(n + \sum_{v \in V} \text{in-deg}(v)) = O(n + m)$.

General Case: we could consider taking the smallest weighted edge at every step, but this would not necessarily give us the correct answer. Instead, we use **Dijkstra's Algorithm**:

```
// assume positive weights
// solve single-source (all destination) problem
// compute d[v]: shortest path weight from s to v for all v in V.
S = {s}, d[s] = 0
while S != V:
    pick edge (u, v) with u in S, v in V - S minimizing d[u] + w(u, v)
    insert v into S
    d[v] = d[u] + w(u, v)
```

The detailed implementation is:

```
// TODO: Prim's, with changes to lines 7-8
if y in emptyset and key[v] + w(u, v) < key[y]:
    key[y] = key[v] + w(u, v), pi[y] = v
```

Analysis: no data structure: $O(n^2)$, heap: $O((n + m) \log n)$, Fibonacci heap: $O(m + n \log n)$.

5.1.1 Correctness Proof

Claim: if (u, v) is the edge from S to $V \setminus S$ minimizing $\delta[u] + w(u, v)$, then $\delta[v] = \delta[u] + w(u, v)$.

Proof: There is a path from s to v with weight $\delta[u] + w(u, v)$. Thus $\delta[v] \leq \delta[u] + w(u, v)$. Now, take any arbitrary path p from s to v . p must contain an edge (u', v') with $u' \in S$ and $v' \in V \setminus S$. $w(p) \geq \delta[u'] + w(u', v') + 0 \implies \delta[v] \geq \delta[u] + w(u, v)$. Thus, we have shown $\delta[v] = \delta[u] + w(u, v)$.

Note that this is only correct if our graph contains no negative weights.

5.1.2 All-Pairs Shortest Paths

Given a weighted directed or undirected graph $G = (V, E)$, find the shortest path between all pairs of vertices.

- Method 0: Run Dijkstra's algorithm on each vertex, ie. run it n times. This method assumes there are no negative weights. Complexity: $O(n(m + n \log n))$.
- Method 1:

5.2 P vs NP

5.2.1 P-Problems

We define P as the set of all decision problems solveable in “polytime” (worst-case polynomial time).

For example:

Example 5.1. We define TSP with inputs: graph $G = (V, E)$, $w : E \implies \mathbb{N}$ and output: cycle that visits every vertex exactly once minimizing total weight.

We define TSP-DECIS with inputs: graph $G = (V, E)$, $w : E \implies \mathbb{N}$, number w and output: yes if and only if there exists a cycle that visits every vertex once and has total weight $\leq w$.

If TSP is solveable in polytime, then TSP-DECIS $\in P$.

Proof. Obviously. □

If TSP-DECIS $\in P$, then we can find the optimal weight in polytime.

Proof. By binary search. We need $O(\log \sum_{e \in E} w(e))$ calls to the decision algorithm, which is polynomial in the number of input bits. □

If TSP-DECIS $\in P$, then we can find the optimal cycle in polytime.

Proof. We first find the optimal weight w^* in polytime. For each $e \in E$, we find the optimal weight w' for $G'(V, E - \{e\})$ in polytime and if $w' = w^*$ we delete e from E . We do this recursively and return all edges remaining in E . We need $O(m)$ calls to this algorithm. □

5.2.2 NP-Problems

Our theory concentrates on one common type of “search” problem: “does there exist...?”. Thus we define NP as the set of all decision problems that can be expressed in the form: input: Object X , output: yes if and only if there exists object y such that property $R(X, y)$ holds where object y has polysize (“certificate”) and property R can be checked in polytime (“verification algorithm”).

For example:

Example 5.2. We define 3-COLORING with input: graph $G = (V, E)$ and output: yes if and only if there exists a coloring of vertices such that no two adjacent vertices have the same color.

We define 3-COLORING $\in NP$ with certificate coloring: $O(n)$ size and verification no two adjacent vertices have the same color: $O(m)$ time.

Example 5.3. TSP-DECIS has inputs: graph $G = (V, E)$, $w : E \Rightarrow \mathbb{N}$, number w and output: yes if and only if there exists a cycle which visits every vertex once and has total weight less than or equal to w .

We define TSP-DECIS $\in NP$ with certificate cycle C : $O(n)$ size and verification C visits every vertex exactly once and has total weight $\leq w$: $O(1 + n)$ time.

Note that NP does not stand for “non-polynomial” but for “non-deterministic polynomial”. Many real-life problems are NP (answers are easy to verify but hard to find).

Theorem 5.1. These problems are subsets of each other: $P \subseteq NP \subseteq EXPTIME$.

Proof. $P \subseteq NP$: ignore certificate. $NP \subseteq EXPTIME$: brute force, try all certificates of polysize $p(n)$: $2^{p(n)}$. \square

Though we have not yet proved it, we assume $P \neq NP$.

5.2.3 NPC-Problems

NPC problems are NP -complete.

We must define the “hardest” problem in NP ... but what does “ L_1 is easier than L_2 mean”? One way to define this is as L_1 “reduces” to L_2 (selection reduces to sorting).

Definition 5.1. Given decision problems L_1, L_2 , a reduction from L_1 to L_2 is an algorithm f such that the output of L_1 on x is “yes” if and only if the output of L_2 on $f(x)$ is “yes”. If such a reduction exists, $L_1 \leq_p L_2$.

Then, to solve L_1 : $x \rightarrow f(x) \rightarrow L_2[f(x)]$.

Lemma 5.2. If $L_1 \leq_p L_2$ and $L_2 \in P$, then $L_1 \in P$.

Lemma 5.3. *If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.*

Definition 5.2. *L is NP-complete if and only if $L \in NP$ and $\forall L' \in NP, L' \leq_p L$.*

Lemma 5.4. *Let L be an NP-complete problem. Then $L \notin P \iff P \neq NP$.*

Proof. Suppose $L \notin P$. Then $L \in NP$ so $P \neq NP$. Suppose $L \in P$. Then $\forall L' \in NP \implies L' \leq_p L$ sp $L' \in P$ and thus $NP = P$. \square