CS 341 — Algorithms

Kevin James

Winter 2015

Contents

1 Solving Recurrences

Example 1.1. *Determine the runtime of mergesort.*

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

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..\frac{2n}{3}])
  stoogesort(A[\frac{n}{3}+1..n])
  stoogesort(A[1..\frac{2n}{3}])
```

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

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

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 $T(n) = aT(\frac{n}{b}) + f(n)$ if $n > n + 0$ and c if $n = n_0$. Set $d = \log_b a$i 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: $\lim_{n\to\infty} f(n)/n^{d+\varepsilon} = \infty \implies T(n) = \Theta(f(n))$.
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.

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

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

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.

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

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

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

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

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 dominat 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: $\Theta(n)$*

- *Shamos' Algortithm:* $\Theta(n \log^2 n)$. *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* $A = a_{n-1}, a_{n-2}, \ldots a_0$, $b = b_{n-1}, b_{n-2}, \ldots b_0$ *in binary, compute* $AB = c_{n-1}, c_{n-2}, \ldots c_0$.

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'..A"]

B' = [b_{n-1}..b_{n/2}]
B" = [b_{n/2 - 1}..b_0]
B = [B'..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* $a_1..a_n$ *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-l)
```

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

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

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 $1, 2, \ldots n$. Thus prob$\left[\frac{n}{4} < l \leq \frac{3n}{4}\right] = \frac{1}{2}$. 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-l)
```

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 $\frac{3n}{10}$. □

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

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

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 incrememntally 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* $[a_1, b_1], \ldots, [a_n, b_n]$*, 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_1, b_1] in increasing order of a:
    pick any color c in {1, ..., k} not within any intervals intersecting [a_i, b_1]
    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 $O(n^2)$.

3.3 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.