Section 2 Supplement

CPSC 365: Algorithms

Amy Zhao

February 3, 2024

Outline

Divide and Conquer

Divide and Conquer: Main Idea

1. Divide the input into smaller subproblems.
2. Conquer the subproblems recursively.
3. Combine the solutions for the subproblems into a solution for the original problem.

All divide and conquer algorithms follow a similar format: start with the entire input data and make a comparison of two values. Depending on this comparison, determine which portion of the input data to recurse on.

Divide and Conquer: Quicksort

Main Idea: Divide the array on a pivot element which is positioned in a way such that elements less than the pivot are kept on the left side, and elements greater than the pivot are kept on the right side.

Recurse on each side until the subarrays become single elements before combining.

Time Complexity: $O(n^2)$ (worst case); $O(n \log n)$ (average case)

```
1  Function Quicksort(A, leftIndex, rightIndex):
2      if leftIndex < rightIndex then
3          pivot ← Partition(A, leftIndex, rightIndex)
4          Quicksort(A, leftIndex, pivot − 1)
5          Quicksort(A, pivot, rightIndex)

6  Function Partition (A, leftIndex, rightIndex):
7      set rightIndex as pivot
8      storeIndex ← leftIndex − 1

9      for i ← leftIndex + 1 to rightIndex do
10         if A[i] < A[pivot] then
11             swap A[i] and A[storeIndex]
12             storeIndex ← storeIndex + 1
13         swap A[pivot] and A[storeIndex + 1]
14     end
15     return storeIndex + 1
```

Divide and Conquer: Quickselect

Main Idea: Quickselect uses the same overall approach as quicksort to find the kth smallest element in an unordered list.

It chooses a pivot element and partitions the data on the pivot. It only recurses on the side where the target element is.

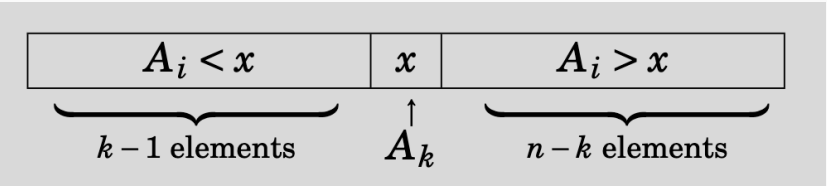
Time Complexity: $O(n)$

1. Divide the n elements of A into $\lfloor \frac{n}{5} \rfloor$ groups of five elements each. Add the additional elements to their own group of size n mod 5.

2. Find the median of each group.

3. Use the algorithm recursively to find the median (denote it with x) of the medians found in the previous step.

4. Partition A around x (reorder the elements of A in place so that all elements prior to x are less than x). Let $k = \text{rank}(x)$.



5. Recursively call the algorithm on the appropriate part of the array.

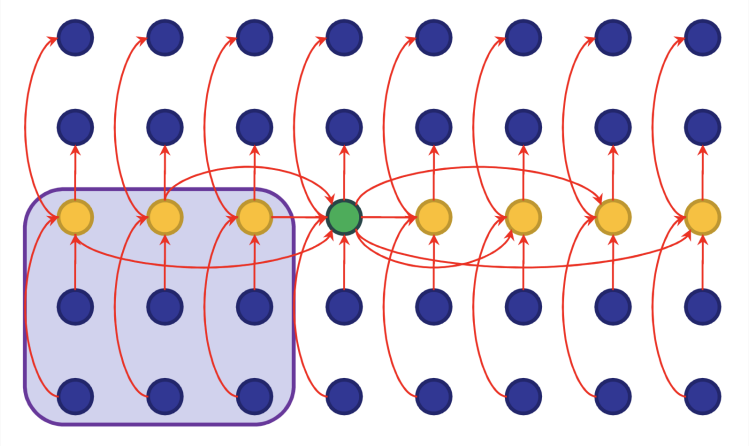
   - If $i = k$, return x.
   - If $i < k$, recursively call the algorithm on $A[1, \ldots k-1]$ with target order i.
   - If $i > k$, recursively call the algorithm on $A[k+1, \ldots i]$ with new target order $i - k$.

Quickselect: Time Complexity

The non-recursive steps all take $O(n)$ time.

Notice that the recursive call to find the median of medians takes input data of size $\lceil \frac{n}{5} \rceil$ elements. How do we analyze the runtime for the recursive call in step 5?

WLOG, assume that we need to recurse on the elements larger than the median of medians x.



In the diagram, the median of each group is shown in yellow, and x is shown in green. The smaller elements point to the larger elements.

In general, there are $\lceil \frac{n}{5} \rceil$ groups, which implies there are $\lfloor \frac{1}{2} \lceil \frac{n}{5} \rceil \rfloor$ groups with median at most x, and $\lfloor \frac{1}{2} \lceil \frac{n}{5} \rceil \rfloor - 1$ groups with median less than x.

Each group contributes three elements less than x, so we have

$$3\left(\left\lfloor \frac{1}{2}\left\lceil \frac{n}{5}\right\rceil \right\rfloor -1\right)\geq 3\left(\left\lfloor \frac{n}{10}\right\rfloor -1\right)$$
$$=3\left(\frac{n}{10}-2\right)=\frac{3n}{10}-6$$

minimum total number of elements less than x, and $\frac{7n}{10}+6$ elements greater than x. Therefore, our recurrence is:

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

We can solve this recurrence through substitution. Assume $T(n) < cn$:

$$T(n) = an + T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right)$$

$$cn \geq c\left(\frac{n}{5}\right) + c\left(\frac{7n}{10}\right) + an$$

$$c \geq \frac{9c}{10} + a \Longrightarrow \frac{c}{10} \geq a \Longrightarrow c \geq 10a$$

Thus, $T(n) = O(n)$.

Divide and Conquer: Mergesort

Main Idea: Divide the array into smaller subarrays, sort each subarray, and merge the sorted subarrays back together.

At each iteration, the algorithm contains two recursive calls because both subarrays need to be sorted.

Time Complexity: $O(n \log n)$

```
1  Function Mergesort(A, n):
2      if n > 1 then
3          r ← n/2
4          L ← A[: r], M ← A[r :]
5          Mergesort(L, r)
6          Mergesort(M, r)
7          i ← 0, j ← 0, k ← 0
8          while i < r and j < r do
9              if L[i] < M[j] then
10                 A[k] ← L[i], i ← i + 1
11             else
12                 A[k] ← M[j], j ← j + 1
13             k ← k + 1
14         end
15         while i < r do
16             A[k] ← L[i], i ← i + 1, k ← k + 1
17         end
18         while j < r do
19             A[k] ← M[j], j ← j + 1, k ← k + 1
20         end
```

Divide and Conquer: Counting Inversions

Main Idea: Divide the array into two subarrays, and recursively count the inversions in each half. In the step to combine, count inversions where a_i and a_j are in different halves, and return the sum of the three quantities.

Time Complexity: $O(n \log n)$

Suppose we divide array A into L and R subarrays. There are three cases for an inversion (i, j):

- $i \in L$ and $j \in R$
- $i \in L$ and $j \in L$
- $i \in R$ and $j \in R$

We need to handle case one differently.

```
1  Function CaseOne(L, R):
2      answer ← 0, j ← 0
3      for i in 0 to len(L) − 1 do
4          while j < len(R) and R[j] < L[i] do
5              j ← j + 1
6          end
7          answer ← answer + j
8      end
9      return answer
10 Function CountInversions(A, n):
11     if n ≤ 1 then
12         return 0
13     L ← A[: n/2 − 1], R ← A[n/2 :]
14     answer ← CountInversions(L, len(L))
15     answer ← answer + CountInversions(R, len(R))
16     answer ← answer + CaseOne(L, R)
17     return answer
```

Divide and Conquer: Closest Pair

Main Idea: Compute a vertical line that roughly divides the points in half. Find the closest pair in each side recursively, and find the closest pair with one point in each side. Return the best of three solutions.

Time Complexity: $O(n \log^2(n))$, but can be reduced to $O(n \log n)$

```
1  Function ClosestPair(P):
2      P_x ← P sorted by x-coordinate
3      P_y ← P sorted by y-coordinate
4      (p_0^*, p_1^*) ← ClosestPairRec(P_x, P_y)

5  Function ClosestPairRec(P_x, P_y):
6      if |P| ≤ 3 then
7          return closest pair through computation

8      Q ← first ⌈n/2⌉ points in P_x, Q_x ← Q sorted by x-coordinate, Q_y ← Q
         sorted by y-coordinate
9      R ← remaining ⌊n/2⌋ points in P_x, R_x ← R sorted by x-coordinate,
         R_y ← R sorted by y-coordinate

10     (q_0^*, q_1^*) ← ClosestPairRec(Q_x, Q_y)
11     (r_0^*, r_1^*) ← ClosestPairRec (R_x, R_y)

12     δ ← min{dist(q_0^*, q_1^*), dist(r_0^*, r_1^*)}
13     x^* ← maximum x-coordinate ∈ Q

14     L ← {(x, y) : x = x^*}
15     S ← points ∈ P within distance δ of L, S_y ← S sorted by y-coordinate

16     for s ∈ S_y do
17         compute distance from s to next 15 points in S_y
18         (s, s') ← pair which achieves minimum distance
19     end

20     if dist(s, s') < δ then
21         return (s, s')
22     else if dist(q_0^*, q_1^*) < dist(r_0^*, r_1^*) then
23         return (q_0^*, q_1^*)
24     else
25         return (r_0^*, r_1^*)
```

Divide and Conquer: Strassen's Algorithm

Main Idea: Divide the two matrices into smaller sub-matrices, and calculate the values recursively.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} S_1 + S_2 - S_4 + S_6 & S_4 - S_5 \\ S_6 + S_7 & S_2 - S_3 + S_5 - S_7 \end{pmatrix}$$

$$S_1 = (B - D)(G + H) \quad S_2 = (A + D)(E + H) \quad S_3 = (A - C)(E + F)$$
$$S_4 = (A + B)H \quad S_5 = A(F - H) \quad S_6 = D(G - E)$$
$$S_7 = (C + D)E$$

Time Complexity: $O(n^{\log_2 7})$

```
1  Function Strassen(M, N):
2      if M 1 × 1 then
3          return M_11 N_11
4      M ← ( A  B ), N ← ( E  F )
           ( C  D )        ( G  H )
5      S_1 ← Strassen(B − D, G + H)
6      S_2 ← Strassen(A + D, E + H)
7      S_3 ← Strassen(A − C, E + F)
8      S_4 ← Strassen(A + B, H)
9      S_5 ← Strassen(A, F − H)
10     S_6 ← Strassen(D, G − E)
11     S_7 ← Strassen(C + D, E)
12     return ( S_1 + S_2 − S_4 + S_6    S_4 − S_5                )
              ( S_6 + S_7               S_2 − S_3 + S_5 − S_7    )
```

Divide and Conquer: Karatsuba's Algorithm

Main Idea: Given two integers (in base-2), perform three recursive multiplications. It relies on the fact that the two integers can be written as lower order and higher order bits:

$$x = x_1 \cdot 2^{\frac{n}{2}} + x_0$$
$$y = y_1 \cdot 2^{\frac{n}{2}} + y_0$$

The product becomes the following:

$$\begin{aligned} xy &= (x_1 \cdot 2^{\frac{n}{2}} + x_0)(y_1 \cdot 2^{\frac{n}{2}} + y_0) \\ &= x_1 y_1 \cdot 2^n + (x_1 y_0 + x_0 y_1) \cdot 2^{\frac{n}{2}} + x_0 y_0 \end{aligned}$$

Time Complexity: $O(n^{\log_2 3})$

```
1 Function FasterMultiplication(x, y):
2     write x as x_1 · 2^(n/2) + x_0
3     write y as y_1 · 2^(n/2) + y_0
4     x_10 ← x_1 + x_0
5     y_10 ← y_1 + y_0
6     p ← FasterMultiplication(x_10, y_10)
7     x_1y_1 ← FasterMultiplication(x_1, y_1)
8     x_0y_0 ← FasterMultiplication(x_0, y_0)
9     return x_1y_1 · 2^n + (p − x_1y_1 − x_0y_0) · n/2 + x_0y_0
```