CPSC 320: Divide and Conquer Algorithms Solutions*

Invented by Tony Hoare in the late 1950's, the Quicksort algorithm was a breakthrough in sorting methods. Variants of Hoare's original algorithm are still a sorting method of choice today. In this worksheet, you will gain experience with the divide and conquer algorithmic design approach, as well as the analysis of recurrence relations, that led Hoare to this breakthrough. You will apply this algorithm in the next worksheet to the problem of finding the median of a list of values.

1 The Master Theorem

For each of the following recurrence relations, determine whether or not the Master Theorem can be used. If it can, give the solution to the recurrence. If it can not, explain why not. In all cases, assume that $T(n) \in \Theta(1)$ when n is sufficiently small.

1. $T(n) = 5T(\sqrt{n}) + n$

   SOLUTION: The Master Theorem can not be used, because b is not a constant (it's $\sqrt{n}$).

2. $T(n) = T(n/2) + 1$

   SOLUTION: Here $a = 1$ and $b = 2$, which means $\log_b a = \log_2 1 = 0$, and $f(n) = 1 = n^0$. We are therefore in case 2 of the Master theorem and $T(n) \in \Theta(\log n)$.

3. $T(n) = T(n/4) + n$

   SOLUTION: Here $a = 1$ and $b = 4$, which means $\log_b a = \log_4 1 = 0$. Also $n = n^1 \in \Omega(n^{0+1})$, which means the only possible case might be case 3. We still need to check the regularity condition: $af(n/b) = f(n/4) = n/4$, and $n/4 < \delta n$ for any δ in the range $0.4 < \delta < 1$. So case 3 applies, and $T(n) \in \Theta(n)$.

4. $T(n) = 3T(n/9) + \sqrt{n}\log_2 n$

   SOLUTION: Here $a = 3$ and $b = 9$, which means $\log_b a = \log_9 3 = 0.5$, and $f(n) = \sqrt{n}\log_2 n = n^{0.5}\log_2 n$. We are therefore in case 2 of the Master theorem and $T(n) \in \Theta(\sqrt{n}\log^2 n)$.

5. $T(n) = \sqrt{n}T(n/3) + n^2$

   SOLUTION: The Master Theorem can not be used because a is not a constant (it is $\sqrt{n}$).

6. $T(n) = 9T(n/3) + n\log n$

   SOLUTION: Here $a = 9$ and $b = 3$, and so $\log_b a = \log_3 9 = 2$. Moreover, $f(n) = n\log n \in O(n^{2-0.5})$ so we are in case 1. This implies $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^2)$.

7. $T(n) = 2T(n/2) + n/\log n$

   SOLUTION: The Master Theorem can not be used: $a = 2$ and $b = 2$, and so $\log_b a = \log_2 2 = 1$. However there is no $\varepsilon > 0$ such that $n/\log n \in O(n^{1-\varepsilon})$.

2 Quicksort Runtime Analysis

Here is a basic version of Quicksort. We will assume that the array $A[1..n]$ to be sorted has n distinct elements.

```
function QUICKSORT(A[1..n])      ▷ returns the sorted array A of n distinct numbers
  if n > 1 then                                                          ▷ Θ(1)
    Choose pivot element p = A[1]                                        ▷ Θ(1)
    Let Lesser be an array of all elements from A less than p            ▷ Θ(n)
    Let Greater be an array of all elements from A greater than p        ▷ Θ(n)
    LesserSorted ← QuickSort(Lesser)                                     ▷ T_Q(⌈n/4⌉ − 1)
    GreaterSorted ← QuickSort(Greater)                                   ▷ T_Q(⌊3n/4⌋)
    return the concatenation of LesserSorted, [p], and GreaterSorted     ▷ O(1)
  else
    return A                                                             ▷ Θ(1)
```

1. Suppose that QuickSort happens to always select the $\lceil \frac{n}{4} \rceil$-th smallest element as its pivot. Give a recurrence relation for the running time of QuickSort.

   SOLUTION: Let $T_Q(n)$ be the runtime (number of steps) of QuickSort on an array of length n. For these solutions, we've annotated the code above with the runtime of each step. This leads us to the following recurrence. It's convenient in recurrences to replace big-O or Θ terms with some constant, and it's ok to use the same constant everywhere.

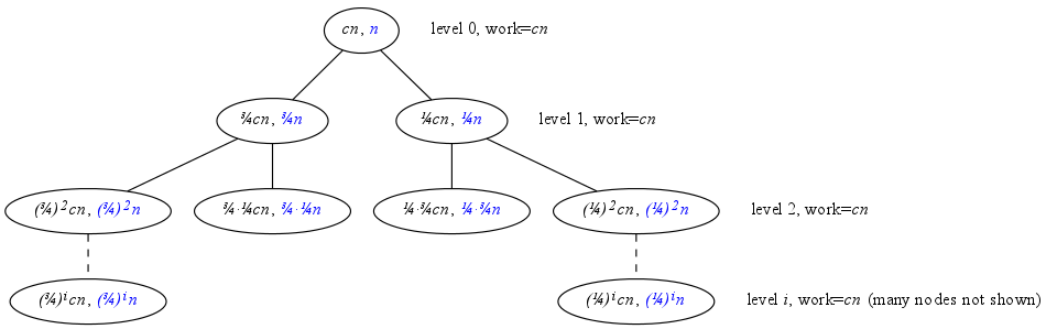
$$T_Q(n) = \begin{cases} c, & \text{if } n = 0 \text{ or } n = 1 \\ T_Q(\lceil \frac{n}{4} \rceil - 1) + T_Q(\lfloor \frac{3n}{4} \rfloor) + cn, & \text{otherwise.} \end{cases}$$

   Ignoring floors, ceilings, and constants we get a slightly simpler recurrence:

$$T_Q(n) = \begin{cases} c, & \text{if } n = 0 \text{ or } n = 1 \\ T_Q(\frac{n}{4}) + T_Q(\frac{3n}{4}) + cn, & \text{otherwise.} \end{cases}$$

2. Using the recurrence, draw a recursion tree for QuickSort. Label each node by the number of elements in the array at that node's call (the root is labeled n) and the amount of time taken by that node but not its children. Also, label the total work (time) for each "level" of calls. Show the root at level 0 and the next two levels below the root, and also the node at the leftmost and rightmost branches of level i.

   SOLUTION: We show the work done at each node (and not its children) in black, and the array size at that node in blue.



3. Find the following two quantities.

   (a) The number of levels in the tree down to the shallowest leaf. *Hint:* Is the shallowest leaf on the leftmost side of the tree, the rightmost side, or somewhere else? If you've already described the problem size of the leftmost and rightmost nodes at level i as a function of i, then set that equal to the problem size you expect at the leaves and solve for i.

   SOLUTION: The $\frac{n}{4^i}$ branch will reach the base case fastest. If we set that equal to 1, we get $\frac{n}{4^i} = 1$, or $n = 4^i$. Taking logs on both sides, $\log_4 n = i$. Also $\log_4 n = 0.5 \log_2 n$ (recall that more generally, $\log_b n = \log_c n / \log_c b$). So the number of levels to the shallowest leaf is $\lg n/2$.

   (b) The number of levels in the tree down to the deepest leaf.

   SOLUTION: Similarly, the $(\frac{3}{4})^i n$ branch reaches the base case slowest, and by a similar analysis, we find $i = \log_{\frac{4}{3}} n$. That looks nasty but is only a constant factor away from $\log_4 n$. (Since $\frac{4}{3} > 1$, this really is a nice, normal log.)

4. Use the work from the previous parts to find asymptotic upper and lower bounds for the solution of your recurrence.

   SOLUTION: For the lower bound, we perform cn work at each level up to the level of the first leaf, which has depth $\Omega(\log n)$, so the total work done is $\Omega(n \log n)$. For the upper bound, we perform at most cn work at each level, including levels after that of the first leaf. There are $O(\log n)$ levels in total, so the total work done is $O(n \log n)$. Putting the upper and lower bounds together we see that the total work done is $\Theta(n \log n)$.

5. Now, we will relax our assumption that Quicksort always selects the $\lceil \frac{n}{4} \rceil$-th smallest element as its pivot. Instead, consider a weaker assumption that the rank of the pivot is always in the range between $\lceil \frac{n}{4} \rceil$ and $\lfloor \frac{3n}{4} \rfloor$ (the *rank* of an element is k if the element is the kth smallest in the array). What can you say about the running time of Quicksort in this case?

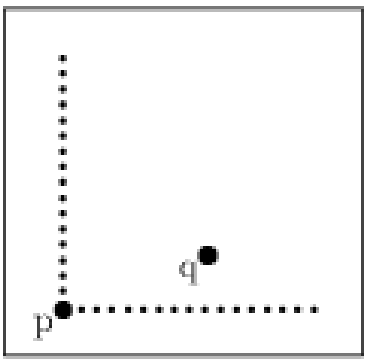
   SOLUTION: If the pivot always lies in the range between $\lceil \frac{n}{4} \rceil$ and $\lfloor \frac{3n}{4} \rfloor$, the shallowest leaf has depth at least $\lg n$, and the deepest leaf has depth at most $\log_{4/3} n$. Also, the total work per level is cn, up to the level of the shallowest leaf, and at most cn for the remaining levels. So our asymptotic upper and lower bounds still hold, and the running time is still $\Theta(n \log n)$.

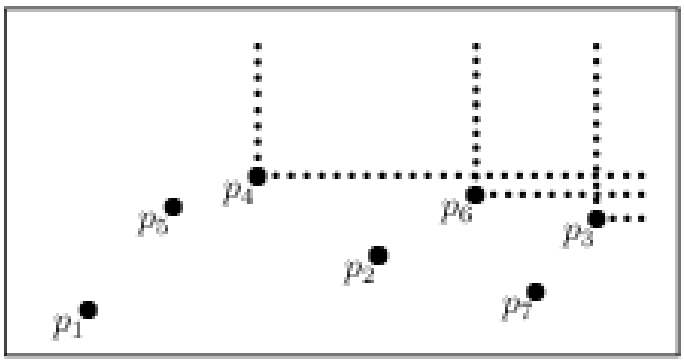
3 The Stock Market, dividends and risks

A firm of financial analysts has hired you to select (efficiently) the stocks that they should recommend to their clients. For each company, they have a pair of values (x, y) where

- x is the expected annual dividends for that company's stock.
- y is a real number between -10 and 10 quantifying the risk of investing in the company (with -10 being high risk, and +10 being low risk).

So each company is represented by the point (x, y) in the plane. We will assume without loss of generality that no two companies are mapped to the same point. Given two such points, we will say that a point $q = (q.x, q.y)$ *dominates* the point $p = (p.x, p.y)$ if $q.x \geq p.x$ and $q.y \geq p.y$. That is, q lies to the right of and above p, as illustrated in picture on the left.





Clearly, a point p that is dominated by a point q is of no interest, since q is both worth more **and** less risky than p. Hence, to help the firm decide which stocks to recommend, you want to only return companies that correspond to *maximal* points: those that no other point dominates. These are the points p_3, p_4 and p_6 in the figure on the right.

1. Describe an algorithm that takes as input a set P of points, and a point q, and returns all points of P that q does not dominate. Hint: this is simple, so don't think too hard about it.

   SOLUTION: All that we need to do is check each point of P one at a time.

   ```
   Algorithm UndominatedPoints(P, q)
     S ← ∅
     for each point p of P do
       if q.x < p.x or q.y < p.y then
         add p to S
     return S
   ```

2. In order to solve this problem using a divide-and-conquer algorithm, we need to divide it into two or more subproblems. How many subproblems should we use, and is how is the input divided between these subproblems?

   SOLUTION: We will use two subproblems, and divide the input by sorting the points by increasing x-coordinate. If two or more points have the same x-coordinate, then they are sorted by increasing y-coordinate. We then divide the sorted list in halves. Let us call these the *left* and *right* subproblems.

3. If a point is maximal in the last one of the subproblems, is it automatically maximal in P? Why or why not?

   SOLUTION: Yes, it is: no point in the left subproblem can dominate a point in the right subproblem. So points that are maximal in the right subproblem must be maximal in P.

4. If a point is maximal in the first of the subproblems, is it automatically maximal in P? Why or why not?

   SOLUTION: No, it is not. It might be dominated by one or more points in the right subproblem.

5. Suppose that a point p in the first subproblem is dominated by one or more points in the other subproblem(s) (assuming this is possible). Name *one* point in the other subproblem(s) that is guaranteed to dominate x.

   SOLUTION: If a point in the right suproblem dominates p, then the point q in the right subproblem with the largest y-coordinate is guaranteed to dominate p: $q.x \geq p.x$ because of how we constructed the subproblems.

6. Using the results of the previous steps, write pseudo-code for an efficient algorithm `MaximalPoints` that takes a input a set P of points, and return a set of all maximal points of P.

   SOLUTION:

```
Algorithm MaximalPoints(P)
  sort P by increasing x-coordinate
  return MaximalPointsHelper(P, 0, length[P]-1)

Algorithm MaximalPointsHelper(P, first, last)
  if first = last then
    return { P[first] {

  mid ← ⌊(first + last)/2⌋
  Sl ← MaximalPointsHelper(P, first, mid)
  Sr ← MaximalPointsHelper(P, mid+1, last)

  q ← point of Sr with largest y-coordinate
  return Sr union UndominatedPoints(Sl, q)
```

7. Analyze the running time of your algorithm.

   SOLUTION: Let $H(n)$ denote the running time of `MaximalPointsHelper` when it is called with a portion of P that contains n elements. Finding the point of `Sr` with largest y-coordinate and the call to `UndominatedPoints` takes $O(n)$ time, and so $H(n)$ satisfies the following recurrence relation:

$$H(n) = \begin{cases} H(\lceil n/2 \rceil) + H(\lfloor n/2 \rfloor) + \Theta(n) & \text{if } n \geq 2 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

   By case 2 of the Master theorem, $H(n) \in \Theta(n \log n)$. The sorting step of algorithm `MaximalPoints` can also be done in $\Theta(n \log n)$ time, using `MergeSort`, and so algorithms `MaximalPoints` runs in $\Theta(n \log n)$ time.