Eksamensnoter - Divide and Conquer

André Oskar Andersen (wpr684)

March 17, 2020

2.3 Designing algorithms

2.3.1 The divide-and-conquer approach

- The divide-and-conquer paradigm involves three steps at each level of the recursion:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subroblems in a straightforward manner
  - **Combine** the solutions to the subproblems into the solution for the original problem
- The *merge sort* algorithm closely follows the divide-and-conquer paradigm:
  - **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
  - **Conquer:** Sort the two subsequences recursively using merge sort
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer

2.3.2 Analyzing divide-and-conquer algorithms

- When an algorithm contains a recursive call to itself, we can often describe its running time by a *recurrence equation* or *recurrence*, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs. When can then use mathematical tools to solve the recurrence and provide bounds on the performance of the algorithm
- We let $T(n)$ be the running time on a problem of size n. If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes $\Theta(1)$. Suppose that our division of the problem yields a subproblems, each of which is $1/b$ tbe size of the original. It takes time $T(n/b)$ to solve one subproblem of size n/b, and so it takes time $aT(n/b)$ to solve a of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence

$$T(n) = \begin{cases} \Theta(1) & in \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of merge sort

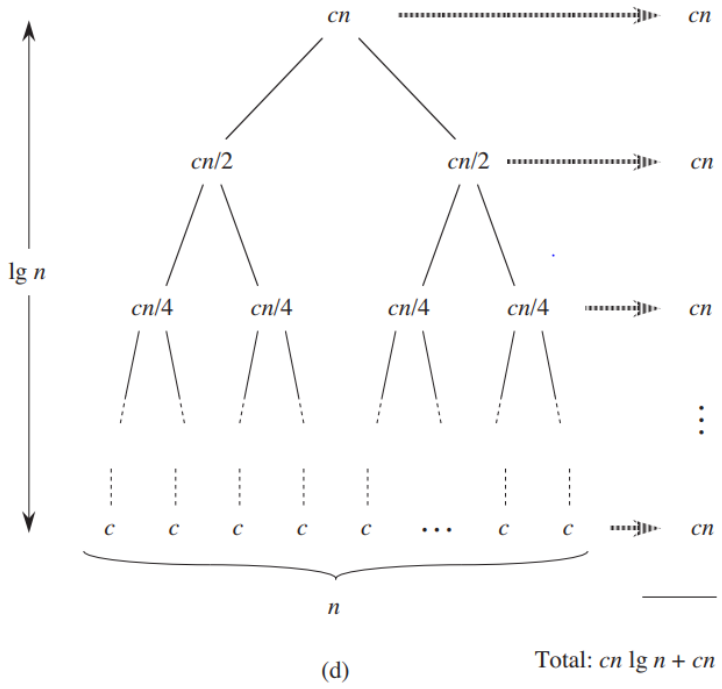
- We reason as follows to set up the recurrence for $T(n)$, the worst-case running time of merge sort on n numbers. Merge sort on just one element takes constant time. When we have $n > 1$ elements ,we break down the running time as follows:
  - **Divide:** The divide step just computes the middle of the subarray, which takes constant time. Thus, $D(n) = \Theta(1)$
  - **Conquer:** We recursively solve two subproblems, each of size $n/2$, which contributes $2T(n/2)$ to the running time
  - **Combine:** Merging n-elements takes $\Theta(n)$, and so $C(n) = \Theta(n)$
- Letting $D(n) + C(n) = \Theta(1) + \Theta(n) = \Theta(n)$ and adding this to the $2T(n/2)$ term from the "conquer" step gives the recurrence for the worst-case running time $T(n)$ of merge sort:

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

- We can use an *recursion tree* to understand why $T(n) = \Theta(n \lg n)$. This is done, by first rewritting $T(n)$:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

  where c represents the time requiered to solve problems of size 1 as well as the time per array element of the divide and combine steps (it is unlikely that the same constant represents both. We can get around this by letting c be the larger/lesser of these times and understanding that our recurrence gives an upper/lower bound on the running time).



Next, we add the costs across each level of the tree. In general, the level i below the top has 2^i nodes, each contributing a cost of $c(n/2^i)$, so that the ith level below the top has total cost $2^i c(n/2^i) = cn$.
The toal number of levels of the recursion tree is $\lg n + 1$, where n is the number of leaves, corresponding to the input size. An informal inductive argument justifies this claim:

- The base case occurs when $n = 1$, in which case the tree has only one level. Since $\lg 1 = 0$, we have that $\lg n + 1$ gives the correct number of levels.
- Assume, that the number of levels of a recursion tree with 2^i leaves is $\lg 2^i + 1 = i + 1$. Because we are assuming that the input size is a power of 2, the next input size to consider is 2^{i+1}. A tree with $n = 2^{i+1}$ leaves has one more level than a tree with 2^i leaves, and so the total number of levels is $(i + 1) + 1 = \lg 2^{i+1} + 1$.

To compute the total cost represented by the recurrence $T(n)$, we add up the costs of all the levels; the tree has $\lg n + 1$ levels, each costing cn, for a total of $cn(\lg n + 1) = cn \lg n + cn = \Theta(n \lg n)$.

4 Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the *recursive case*. Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms, out" and that we have gotten down to the *base case*.

Recurrences

- Recurrences go hand in hand with the divide-and-conquer paradigm, because they give us a natural way to characterize the running time of divide-and-conquer algorithms
- There are three methods for solving recurrences - that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:
  - In the *substitution method*, we guess a bound and then use mathematical induction to prove our guess correct
  - The *recursion-tree method* converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion. We use techniques for bounding summations to solve the recurrence.
  - The *master method* provides bounds for recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1, b > 1$, and $f(n)$ is a given function. A recurrence of this form characterizes a divide-and-conquer algorithm that creates a subproblems, each of which is $1/b$ the size of the original problem, and in which the divide and combine steps together take $f(n)$ time.

Technicalities in recurrences

- In practice, we neglect certain technical details when we state and solve recurrences.
- Boundary conditions represent a class of details that we typically ignore. Since the ruuning time of an algorithm on a constant-sized input is a aconstant, the recurrences that arise from the running times of algorithms generally have $T(n) = \Theta(1)$ for sufficiently small n. For convenience, we shall generally omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n. For example, we normally state the recurrence

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

as

$$T(n) = 2T(n/2) + \Theta(n)$$

without explicitly giving values for small n.l The reason is that alothough changing the values of $T(1)$ changes the exact solution to the recurrence, the solution typically doesn't change by more than a constant factor, and so the order of growth is unchanged.

- Floors and ceilings are another class of details that we typically ignore. For example, if we call merge-sort on n elements when n is odd, we end up with subproblems of size $\lfloor n/2 \rfloor$ *and* $\lceil n/2 \rceil$. Neither size is actually $n/2$, because $n/2$ is not an integer when n is odd.

4.3 The substitution method for solving recurrences

- The *substitution method* for solving recurrences comprises two steps:

  1. Guess the form of the solution
  2. Use mathematical induction to find the constants and show that the solution works

- We substitute the guessed solution for the function when applying the inductive hypothesis to smaller values; hence thename "substitution method".

- We can use the substitution method to establish either upper or lower bounds on a recurrence. As an example, let us determine an upper bound on the recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

  We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that his bound holds for all positive $m < n$, in particular $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Subtituting into the recurrence yields

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \end{aligned}$$

  where the last step holds as long as $c \geq 1$.

- Mathematical induction now requires us to show that our solution holds for the boundary conditions. Typically, we do so by showing that the boundary conditions are suitable as base cases for the inductive proof. For this example, we must show that we an choose the constant c large enough so that the bound $T(n) \leq cn \lg n$ works for the boundary conditions as well. This requirement can sometimes lead to problems:

  1. Let us assume, that $T(1) = 1$. Then for $n = 1$, the bound $T(n) \leq cn \lg n$ yields $T(1) \leq c1 \lg 1 = 0$, which is at odds with $T(1) = 1$

We can overcome this obstacle:

1. By observing that for $n > 3$, the recurrence does not depend directly on $T(1)$, we can replace $T(1)$ by $T(2)$ and $T(3)$ as the base cases in the inductive prrof, letting $n_0 = 2$.
2. We derive from the recurrence that $T(2) = 4$ and $T(3) = 5$.
3. Now we can complete the inductive proof that $T(n) \leq cn \lg n$ for some constant $c \geq 1$ by choosing c large enough so that $T(2) \leq c2 \lg 2$ and $T(3) \leq c3 \lg 3$. As it turns out, any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold.

Subtleties

- Consider the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + \lceil n/2 \rceil + 1$$

We guess that the solution is $T(n) = O(n)$, and we try to show that $T(n) \leq cn$ for an appropriate choice of the constant c. Substituting our guess in the recurrence, we obtain

$$T(n) \leq c \lfloor n/2 \rfloor + c \lceil n/2 \rceil + 1 = cn + 1$$

which does not impoy $T(n) \leq cn$ for any choice of c; we are off only by the constant 1. We overcome our difficulty by subtracting a lower-order term from our previous guess. Our new guess it $T(n) \leq cn - d$, where $d \geq 0$ is a constant. We now have

$$T(n) \leq (c \lfloor n/2 \rfloor - d) + (c \lceil n/2 \rceil) + 1 = cn - 2d + 1 \leq cn - d$$

as long as $d \geq 1$

4.4 The recursion-tree method for solving recurrences

- Drawing out a recursion tree serves as a traghtforward way to devise a good guess, which then can be verified using the *substitution method*. If you are very careful when drawing out a recursion tree and summing the costs, however. you can use a recursion tree as adirect proof of a solution to a recurrence.
- When using a recursion tree to generate a good guess, you can often tolerate a small amount of "sloppiness", since you will be verifying your guess later.
- In a *recursion tree*, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion
- An example on how to come up with a good guess using an recursion tree, can be seen in section 2.3.2 where merge sort is analysed.

4.5 The master method for solving recurrences

- The master method provides a "cookbook" method for solving recurrences of the form
$$T(n) = aT(n/b) + f(n)$$
where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

- The recurrence describes the running time of an algorithm that divides a problem of size n into a subproblems, each of size n/b, where a and b are positive constants. The a subproblems are solvd recursively, each in time $T(n/b)$. The function $f(n)$ encompasses the cost of dividing the problem and combining the results of the subproblems.

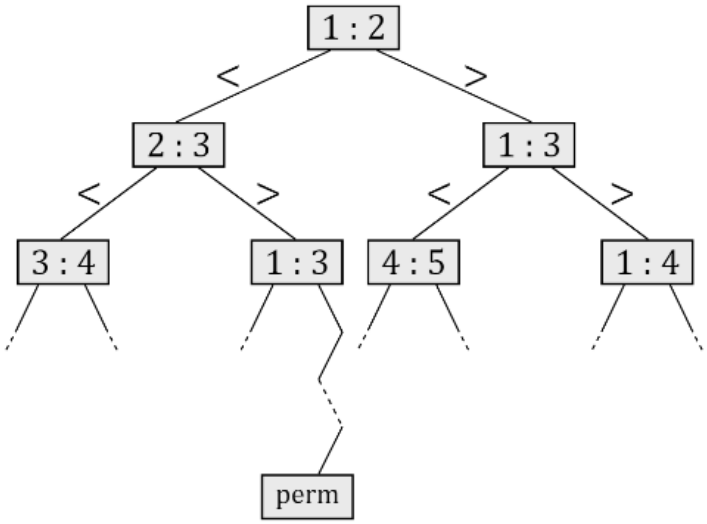
The master theorem

- The master method depends on the following theorem:
  - Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be an asymptotically positive function, and let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$ where we interpret n/b to mean either $\lfloor n/2 \rfloor$ or $\lceil n/2 \rceil$. Then $T(n)$ has the following asymptotic bounds:
    1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constants $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
    2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
    3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Using the master method

- Consider
$$T(n) = 9T(n/3) + n$$
For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 0 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 and conclude that the solution is $T(n) = \Theta(n^2)$

8 Sorting in Linear Time



8.1 Lower bounds for sorting

- In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\{a_1, a_2, ..., a_n\}$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j, a_i \leq a_j, a_i = a_j, a_i > a_j, a_i \geq a_j$ to determine their relative order. We maynot inspect the values of the elements or gain order information about them in any other way

- We assume without loss of generality, that all the input elements are distinct. Thus, $a_i = a_j$ are useles, so we can assume that no comparisons of this form are made. We also note that the other comparisons are all equivalent in that they yield identical information. We therefore assume that all comparisons have the form $a_i \leq a_j$

The decision-tree model

- We can view comparison sorts abstractly in terms of decision trees, by annotating each internal node by $i : j$ for some i and j in the range $1 \leq i; j \leq n$, where n is the number of elements in the input sequence. We also annotate each leaf by a permutation $\{\pi(1), \pi(2), ..., \pi(n)\}$. The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf. Each internal node indicates a comparison $a_i \leq a_j$. The left subtree then dictates subsequent comparisons once we know that $a_i \leq a_j$, and the right subtree dictates subsequent comparisons knowing that $a_i > a_j$.

A lower bound for the worst case

- The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree. A lower bound on the heights of all decision trees i which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

Theorem 8.1

- Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case
  proof:
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \leq l$. Since a binary tree of height h has no more than 2^h leaves, we have

    $$n! \leq l \leq 2^h$$

    which, by taking logarithms, implies

    $$h \geq \lg(n!) = \Omega(n \lg n)$$

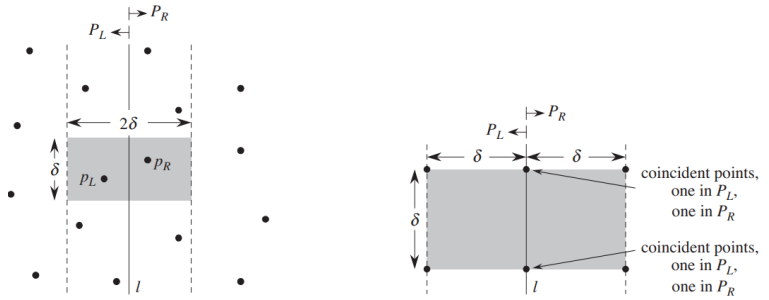
- Since the $O(n \lg n)$ upper bounds on the running times for heapsort and merge sort match the $\Omega(n \lg n)$ worst-case lower bound, heapsort and merge sort are asymptotically optimal comparison sorts.

33 Computational Geometry

33.4 Finding the closest pair of points

- Consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points.
- The divide-and-conquer algorithm for this problem, has a running time described by $T(n) = 2T(n/2) + O(n)$

The divide-and-conquer algorithm



- Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrrays X and Y, each of which contains all the points of the inpput subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate.

- A given recursive invocation with inputs P, X, and Y first checks whether $|P| \leq 3$. If so, the invocation simply performs the brute-force method: try all $\binom{|P|}{2}$ pairs of points and return the closest pair. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows:

  - **Divide**: Find a line l that bisects the point set P into two equally sized sets P_L and P_R. Divide the array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.
  - **Conquer**: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of poitns in P_L and the other to find the closest pair of points in P_R. The input to the first call are the subset P_L and arrays X_L and Y_L; the second call receives the inputs P_R, X_R and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$
  - **Combine**: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R. If a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, they both must reside in the 2δ-wide vertical strip centered at line l. To find such a pair, we do the following:

    1. Create an array Y', which is the array Y with all points not in the 2δ-wide vertical strip removed. The array Y' is sorted by y-coordinate, just as Y is.
    2. For each point p in the array Y', try to find points in Y' that are within δ units of p. Only the 7 points in Y' that follow p need to

be considered. Compute the distance from p to each of these 7 points, and keep track of the closest-pair distance δ' found over all pairs of points in Y'

3. If $\delta' < \delta$, then the verical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

Correctness

- By bottoming out the recursion when $|P| \leq 3$, we ensure that we never try to solve a subproblem consisting of only one point.
- We need only to check the 7 points following each point p in array Y'; suppose that at some level of the recursion, the closest pair of points is $p_L \in P_L$ and $p_R \in P_R$. Thus, the distance δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ units of each other vertically. Thus, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at l.
- We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square. Similarly, at most 4 points in P_R can reside within the δ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within $\delta \times 2\delta$ rectangle.
- Now, we can see why we need to check only the 7 points following each point in the array Y'. Still assuming that the closest pair is p_L and p_R, let us assume that p_L precedes p_R in array Y'. Then, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.