CS 577: Introduction to Algorithms Fall 2015

Divide and Conquer

Instructor: Dieter van Melkebeek Scribe: Andrew Morgan

When presented with a large, complicated problem in life, it is good advice to try to break the problem down into smaller pieces and solve each piece separately. Divide and conquer is an algorithmic technique that realizes this wisdom in the algorithms world. More precisely, divide and conquer refers to the technique of dividing a problem into simpler parts, solving ('conquering') each part separately, and then combining the solutions of each part into a solution for the original problem. When this idea is combined with the notion of recursion, oftentimes a seemingly complicated and difficult problem can be solved with surprising simplicity and efficiency.

In these notes, we present a few examples of this algorithmic paradigm. The main focus will be on the *efficiency* of these algorithms, that is, *how long* they take to run. In fact, we will only cover the main ideas for the correctness arguments and leave the full details as exercises. To understand the efficiency of our algorithms, we will revisit the notion of recursion trees, as these will also provide a useful tool for counting the work performed by recursive algorithms. Along the way, we will review asymptotic notation and discuss some additional aspects of sorting.

1 Powering, Revisited

Our first example of a divide and conquer algorithm will be an algorithm to solve the powering problem (presented in the "Program Correctness" notes). The specific algorithm we provide will be both correct and tremendously more efficient, and require no more than a few extra characters of code.

1.1 Previous Powering Approach

Recall the Powering Problem:

Input: (a, b) with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$

Output: a^b

and the recursive solution presented previously:

Algorithm 1

Input: $a \in \mathbb{Z}, b \in \mathbb{Z}^+$
Output: a^b

procedure Rec-Power(a, b)
  if $b = 1$ **then**
    return a
  else
    return Rec-Power$(a, b-1) \cdot a$

We can interpret Algorithm 1 as a divide and conquer algorithm as follows: The original problem is to compute a^b. This approach divides this into the two problems of computing a^{b-1} and a^1. It then solves the problem of computing a^{b-1} recursively and considers the problem of computing a^1 already solved. Finally, we combine these solutions by multiplying a^{b-1} with a to get a^b.

We have already proved correctness of this algorithm, and will not repeat its proof here. We will, however, understand its efficiency, that is, how much work is done over the course of the entire computation. To do this, recall the notion of a recursion tree; before, we used the recursion tree of Algorithm 1 to establish its correctness. We can also use the recursion tree to understand its efficiency.

Recursion Trees Can Measure Efficiency The general strategy begins by associating to each node of the recursion tree the amount of work done *locally* at the call represented by that node, and then adding all of these contributions together. Here, "locally" means that we only account for operations done exactly within the associated call; we will not account for operations performed within recursive subcalls. (The operations performed in subcalls are counted during the subcall's node in the recursion tree.) The effect is that, when we sum all the costs associated to each node of the recursion tree, then we will have accounted for every operation performed over the entire course of the algorithm's execution exactly once. Thus recursion trees allow us to reduce the problem of "understanding the running time of a recursive algorithm" to the problems of "understanding the local running time of a recursive call" and "understanding the size and shape of the recursion tree". These two problems are almost always easier to understand.

Let us now figure out the running time of Algorithm 1 with this technique. One measure of this is the number of multiplications performed in the running of the algorithm. Many other measures exist, but this one will be easiest to reason about here, while also having good correlation with the efficiency. Thus, we have the following understanding of the efficiency of Algorithm 1:

Claim 1. *Algorithm 1 uses $b-1$ multiplications to compute a^b.*

Proof. Figure 1 serves as a pictorial representation of the recursion tree for Algorithm 1 on input (a, b), complete with a visual representation of the proof presented here.

We first characterize the cost locally within each node:

($b \geq 2$) On inputs of the form (a, b) with $b \geq 2$, Algorithm 1 makes one multiplication locally, on Line 5. We thus charge nodes of this form a cost of 1. Note that we do not consider the cost of the recursive call.

($b = 1$) When $b = 1$, Algorithm 1 performs no multiplications. We thus charge nodes of this form a cost of 0.

Next we sum each of these costs over the whole recursion tree. It is easy to see that there are exactly $b-1$ nodes with $b \geq 2$, and there is exactly 1 node with $b = 1$. Combining this with our characterization, we see that the total number of multiplications is given by

$$(b-1) \cdot 1 + (1) \cdot 0 = b - 1$$

as desired. □

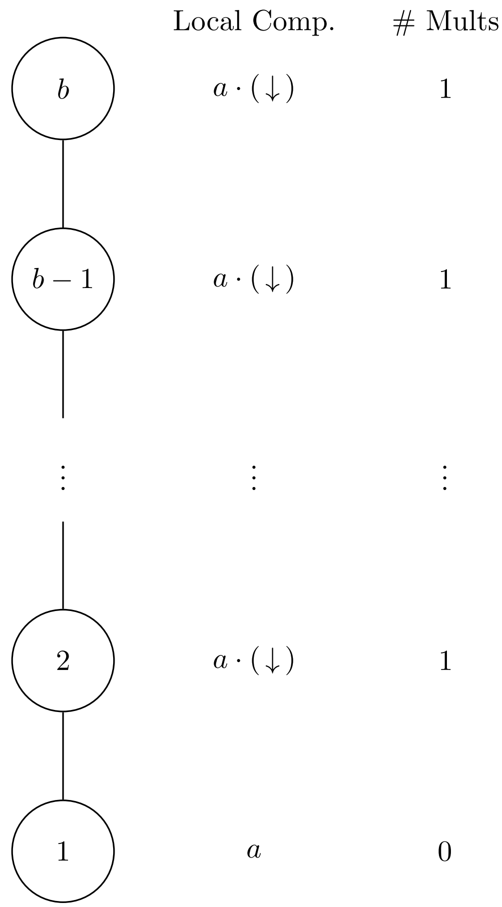


Figure 1: Recursion tree of Algorithm 1. The $\downarrow$'s represent the result of recursion.

1.2 Fast Powering

As was stated earlier, there is a much faster way to compute a^b than Algorithm 1. The basic intuition is that we can be smarter about how we divide a problem into subproblems. As an example, suppose we wished to compute a^{16}. Algorithm 1 would compute it by computing a^2, then a^3, then a^4, and so on, up to a^{16}. A smarter way is to compute a^2, then use this to compute a^4 by multiplying a^2 with itself (rather than with just a), then square again to get a^8, and one last time to get a^{16}. This requires in only 4 multiplications, instead of 15.

We can extend this intuition into a full algorithm, given in Algorithm 2.

Correctness of Algorithm 2 can be shown in a similar way as for Algorithm 1; we leave this as an exercise.

The efficiency of Algorithm 2 can also be understood similarly to the efficiency of Algorithm 1. The main difference is that the recursion tree in Algorithm 2 is considerably smaller—only of size at most $\log(b)$, rather than size b. Specifically, we have the following claim:

Claim 2. *Algorithm 2 uses at most $2 \cdot \log(b)$ multiplications to compute a^b.*

Proof. We proceed by induction on b, starting from $b = 1$:

Algorithm 2

Input: $a \in \mathbb{Z}, b \in \mathbb{Z}^+$
Output: a^b

```
procedure FAST-POWER(a, b)
    if b = 1 then
        return a
    else
        c ← FAST-POWER(a, ⌊b/2⌋)
        c ← c · c
        if b is odd then
            c ← c · a
        return c
```

Base case: When $b = 1$, the above procedure performs 0 multiplications, and $0 \leq 2\log(1)$.

Inductive case: When $b > 1$, the algorithm recursively calls itself with argument $\lfloor b/2 \rfloor$. By induction, this computation uses at most $2\log(\lfloor b/2 \rfloor)$ multiplications. The rest of the call uses at most two additional multiplications. Adding these together we have the following upper bound

$$2\log(\lfloor b/2 \rfloor) + 2 \leq 2\log(b/2) + 2 = 2\log(b)$$

which completes the proof.

□

Per usual in algorithms, we are only interested in the asymptotic behavior of algorithms. Small contributions, such as the "-1" in Claim 1, are not considered significant, and it is useful to reflect this in our notation. Toward this end, we use "Big-Oh" notation, which we review presently.

Asymptotic Notation (Big-Oh Notation) In algorithms analysis, we largely only concern ourselves with the asymptotic performance of algorithms. The difference between programs which run in n steps versus programs that run in $n + 7$ steps is considered negligible. We even consider the difference between programs that run in n and $2n$ steps to be a negligible difference—and it is when compared against programs which run in say n^2 time, or even $n \log(n)$ time.

One way to express that a function $f(n)$ is "no worse" than a function $g(n)$ is to say that "eventually, the values taken by $f(n)$ are no more than a uniform constant factor times the corresponding values of $g(n)$". Formally, we say the following: "$f(n)$ is big-oh $g(n)$" (symbolically: $f(n) = O(g(n))$) when the following logical expression is true

$$\left(\exists c \in \mathbb{Z}^+\right) (\exists N \in \mathbb{N}) \, (\forall n \geq N) \, f(n) \leq c \cdot g(n) \tag{1}$$

We furthermore say that "$f(n)$ is big-theta $g(n)$" (symbolically: $f = \Theta(g(n))$) when $f(n) = O(g(n))$ and $g(n) = O(f(n))$, and say that "$f(n)$ is big-omega $g(n)$" (symbolically: $f = \Omega(g(n))$) when $g(n) = O(f(n))$. We present some examples before returning to the remainder of this section (proofs are left as exercises):

- $n^3 = \Omega(n^2)$, but $n^3 \neq O(n^2)$

- $\log(n) = O(n)$, but $\log(n) \neq \Omega(n)$
- $\frac{(n)(n-1)}{2} = \Theta(n^2)$

Coming back to our two algorithms for powering, Claim 1 can be restated to say that Algorithm 1 makes $\Theta(b)$ multiplications, and Claim 2 can be restated to say that Algorithm 2 makes $O(\log(b))$ multiplications. Since we know that $\log(b)$ is asymptotically much smaller than b, we can clearly see that Algorithm 2 drastically out-performs Algorithm 1.

One subtle issue which shows up in the Powering problem is the size of the integers involved, and the effect this has on the actual work required to compute products of integers. When the parameter b is small, this is not an issue, but when b is large and $a \geq 2$, the bit length of the output can be shown to be $\Theta(b \log(a))$. Thus simply asking for $2^{1,000,000}$, while requiring no more than 40 multiplications, has multiplications involving $500,000$-bit numbers.

With grade-school multiplication, which takes work roughly $\Theta(n^2)$ on numbers of bit-length n, this can take a very long time. We will see later in these notes a faster multiplication algorithm, taking $\Theta(n^{\log_2(3)})$ time, which is an improvement, since $\log_2(3) \approx 1.585 < 2$.

However, the approach used in our fast-exponentiation procedure extends to more than just multiplying integers; it extends to any sense of multiplication, which here just means an associative binary operation. For example, it applies to matrix multiplication, or to multiplication of integers modulo a fixed modulus. In the case of matrix multiplication, there are still issues of bit-length blow-up, but the savings are still huge, as each matrix multiplication induces a large number of integer multiplications. For computing powers modulo the modulus, we can always reduce intermediate results modulo the modulus, which keeps the bit length small. In this case, the bound of $O(\log(b))$ multiplications is actually a more complete reflection of the actual performance of the fast-exponentiation algorithm.

2 Sorting

One of the most fundamental algorithmic operations that one can do to an array is to sort it, and needs little explicit motivation to justify its study. Instead, let us briefly specify the sorting problem before jumping into some algorithms for sorting.

Input: An array $A[0, \ldots, n-1]$ of size n.

Output: A sorted copy of A

Typically we measure the efficiency of a sorting algorithm by the number of comparisons it makes. (This is not always the case; see Section 2.4.)

2.1 Simple Sorting Algorithms

We first review some basic sorting algorithms.

Selection Sort Selection sort works by constructing the output array one element at a time. In the first iteration, the smallest element of $A[0 \dots (n-1)]$ is found, and swapped with the first element of the output array. In the second iteration, the smallest element of $A[1 \dots (n-1)]$ is found, and then swapped with the second element of the output array. This procedure is continued until we have swapped the correct element into $A[n-2]$, where we can stop (since $A[n-1]$ must already be correct). Formally, we have Algorithm 3.

Algorithm 3

Input: $A[0 \cdots n-1]$, an array of length n
Output: A, sorted

1: **procedure** SELECTION-SORT(A)
2:     **for** $k = 0 \dots (n-1)$ **do**
3:         $i_{\text{best}} \leftarrow k$
4:         **for** $i = k \dots (n-1)$ **do**
5:             **if** $A[i] < A[i_{\text{best}}]$ **then**
6:                 $i_{\text{best}} \leftarrow i$
7:         SWAP($A[k]$, $A[i_{\text{best}}]$)

The correctness of selection sort is intuitively clear; concocting a formal proof using loop invariants is left as an exercise. Here we are more interested in efficiency. Toward this end, note that Line (5) is executed $n-k$ times for each value of k from 0 to $(n-1)$. Thus the total number of comparisons made by selection sort is $0 + 1 + \cdots (n-1) = \frac{(n)(n-1)}{2} = \Theta(n^2)$.

Insertion Sort Insertion sort works by maintaining a sorted prefix of A, and *inserting* the next element of A into this prefix. This insertion is done by simply swapping adjacent elements from the right until this would result in an out-of-order prefix. Formally, we have Algorithm 4.

Algorithm 4

Input: $A[0 \cdots n-1]$, an array of length n
Output: A, sorted

1: **procedure** INSERTION-SORT(A)
2:     **for** $k = 0 \dots n-1$ **do**
3:         $i \leftarrow k$
4:         **while** $i > 0$ **and** $A[i-1] > A[i]$ **do**
5:             SWAP($A[i-1], A[i]$)
6:             $i \leftarrow i - 1$
7:     **return** A

As with insertion sort, we leave the proof of correctness as an exercise. For measuring efficiency, consider what happens when A is sorted in reverse order. In this case, we will always move $A[k]$ down to $A[0]$; *i.e.*, the while loop in Line 4 is only exited when $i = 0$. This involves k comparisons on the kth iteration of the for loop of Line 2, and thus a total of $0+1+\ldots+n-1 = \frac{(n)(n-1)}{2} = O(n^2)$ comparisons. This is the worst possible case for insertion sort—any other array can only break out of the while loop earlier. Thus insertion sort will make a maximum of $O(n^2)$ comparisons on any input.

This is not to say that insertion sort is 'just as good/bad as selection sort'. Selection sort *always* uses $\Theta(n^2)$ comparisons. But on an array which is already sorted, insertion sort will actually only make $n - 1 = \Theta(n)$ comparisons. Thus while they have identical worst-case performances, their non-worst-case performances can differ significantly.

Bubble Sort Bubble sort is a sorting algorithm which works by repeatedly sweeping across the input array from left to right, swapping adjacent elements of the array if they are out of order. This process is repeated until no swaps are made over the course of a sweep. Formally, we have Algorithm 5.

Algorithm 5

Input: $A[0 \cdots n-1]$, an array of length n
Output: A, sorted

1: **procedure** BUBBLE-SORT(A)
2:     halt $\leftarrow$ **false**
3:     **while** $\neg$ halt **do**
4:         halt $\leftarrow$ **true**
5:         **for** $k = 0 \ldots n-2$ **do**
6:             **if** $A[k] > A[k+1]$ **then**
7:                 SWAP($A[k]$, $A[k+1]$)
8:                 halt $\leftarrow$ **false**
9:     **return** A

The correctness of bubble sort informally follows from the observation that an array is sorted if and only if every pair of adjacent elements is in their correct order. We leave the formal proof as an exercise.

The efficiency of bubble sort informally follows from the observation that, after the kth iteration of the while loop on Line (3), the kth largest element of A is in the correct position. This means this loop is executed at most n times. In each iteration of the loop, $n - 1$ comparisons are made, leading to an upper bound of $(n)(n-1) = O(n^2)$ comparisons for bubble sort. This upper bound is tight, as witnessed by bubble sort's performance when given a reverse-sorted array of distinct elements.

2.2 Divide and Conquer Sorting Algorithms

Merge Sort We now move on to the algorithm known as merge sort, which uses a divide-and-conquer approach to sort its input array. The specific strategy is to divide the array into a 'left' and 'right' half, recursively sort each of these, and then merge the two sorted arrays. As we will see, when the merging step can be done sufficiently efficiently, this yields an algorithm whose performance, on *any* input, is only $O(n \log(n))$.

As the merging step has a somewhat difficult to explain implementation, but a simple specification, we will defer its implementation until after we have analyzed the rest of merge sort, and instead only provide its specification for now:

Input: Two *sorted* arrays, L and R, of lengths n_L and n_R respectively.

Output: An array M, which is a sorted copy of the concatenation of L with R.

Efficiency: Merging must be done using at most $O(n_L + n_R)$ comparisons among elements of L and R

Let us now move on to the description and analysis of merge sort. Its implementation is given in Algorithm 6.

Algorithm 6

Input: $A[0 \cdots n-1]$, an array of length n
Output: B, a sorted copy of A

procedure Merge-Sort(A)
  if $n = 1$ **then**
    return A
  else
    $m \leftarrow \lfloor n/2 \rfloor$
    $L \leftarrow$ Merge-Sort($A[0 \cdots (m-1)]$)
    $R \leftarrow$ Merge-Sort($A[m \cdots (n-1)]$)
    return Merge(L, R)

The correctness of Algorithm 6 (assuming a correct implementation of the merging procedure) can be shown using the techniques presented in the lecture on program correctness; we leave the details as an exercise. We instead focus on its performance.

The performance of Algorithm 6 is given formally in Claim 3. As in the earlier sorting algorithms, we are largely interested in its performance as given by the number of comparisons that are made.

Claim 3. *Algorithm 6 makes at most $O(n \log(n))$ comparisons among elements of A.*

Proof. We again use recursion trees to organize our argument. However, unlike with Algorithms 1 and 2, the recursion tree for merge sort is not a straight line. Instead, since the recursive case of the implementation makes two recursive calls, the recursion tree will be a binary tree, in which every node is a leaf (corresponding to the base case) or has two children (the recursive case). Pictorially, Figure 2 gives a representation of the recursion tree.

As before, we will count the total number of comparisons made over the course of Algorithm 6's execution by first counting the comparisons made locally within a node of the recursion tree, and then summing these up over all the nodes in the tree. The following proposition gives the essential ingredient needed to understand both of these.

Proposition 1. *A recursive call at depth d (counting from $d = 0$) in the recursion tree has as input an array of size at most $n/2^d$.*

This proposition can be proven with an inductive argument, which is left as an exercise.

We can now compute the local work for nodes in the recursion tree. Sppose we are at depth d in the recursion tree at a node whose input array has size larger than one. Using the performance guarantees of the merge procedure together with Proposition 1, we know that there are at most $O(n/2^d)$ comparisons made locally at this node. (Note that here we are charging for the comparisons made within the merge procedure even though it is used as a subroutine. However, we do *not* charge for comparisons made within the recursive calls to merge-sort.) This will suffice for our purposes of understanding the local performance of Algorithm 6.

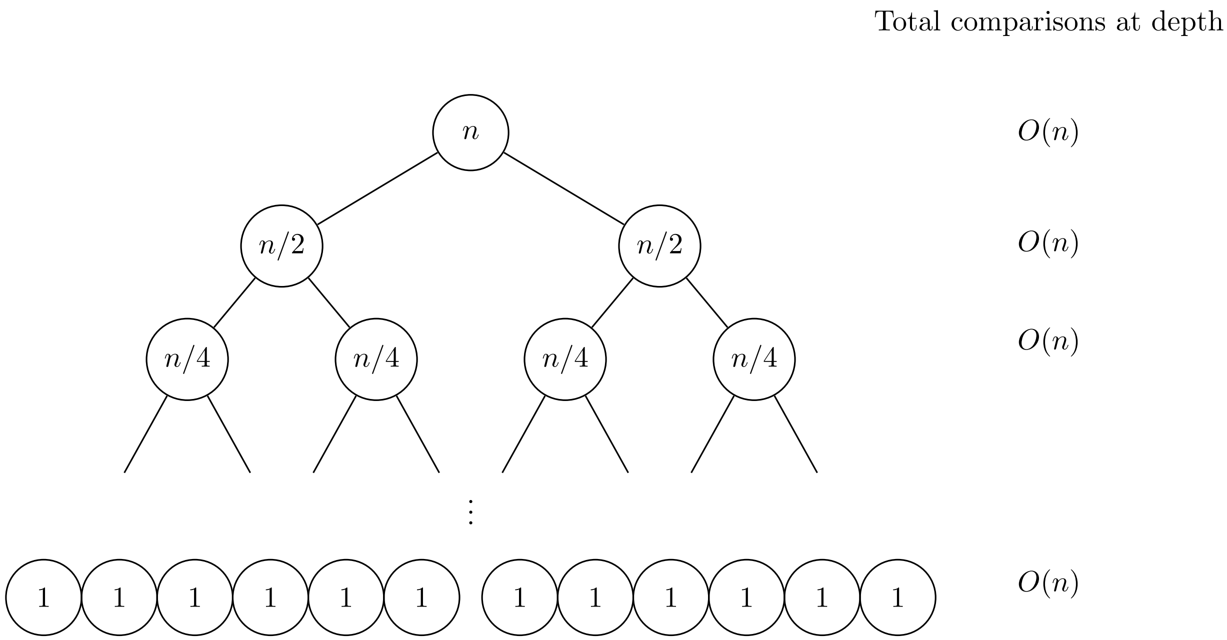


Figure 2: Recursion tree of Algorithm 6

To aggregate this local understanding into a global understanding of Algorithm 6, we need to have a more quantitative understanding of the structure of the recursion tree. This comes in two parts:

- For a fixed level d of the recursion tree, the sum of the contribution of all nodes at level d is at most $O(n)$. This follows from Proposition 1 and the fact that, in a binary tree, there are at most 2^d nodes at depth d.
- The recursion tree has depth at most $\lceil \log(n) \rceil$. This follows from Proposition 1, since plugging in the value $d = \lceil \log(n) \rceil$ yields an upper bound of $n/2^{\lceil \log(n) \rceil} \leq 1$ on the size of the list given as input to recursive calls of depth d. Since Algorithm 6 does not make any further recursive calls under this condition, the recursion tree cannot possibly be any deeper.

To count the total number of comparisons made by the algorithm, we can then multiply our upper bound on the per-level number of comparisons by our upper bound on the total number of levels in the tree, and obtain an upper bound on the total number of comparisons across the whole tree. This results in an upper bound of $O(n \log(n))$, as desired. □

The Merge Procedure Let us now return to the 'merge' subroutine used in Algorithm 6. A naïve implementation is to simply compare each element of L with each element of R, and then deduce the output array M from this information. This, however, uses $\Theta(n_L \cdot n_R)$ comparisons, which is too many to meet the specification.

A problem with this naïve approach is that it makes no use of the precondition that L and R are sorted. In particular, it makes all $\Theta(n_L \cdot n_R)$ comparisons before even deciding which element goes first in M! On the other hand, since we know that L and R are sorted, we know that the first element in M should be either the first element of L or the first element of R; every other element of L or R comes after one of these two. We can determine which will go first in M by making a single comparison, namely comparing these two elements.

Let's suppose we have made this comparison, and appropriately copied the correct element to the first position in M. What's left? We have the rest of L and R, which are still sorted arrays, and the rest of M to fill. In fact, we have essentially the same problem as in the specification of the merge procedure! Hence we could apply recursion. The net effect of this is that we make one comparison for each new element of M that we discover, and so make at most $n_L + n_R$ comparisons overall—enough to meet the specification.

However, a recursive implementation may suffer from some overhead issues: a poor implementation could end up copying "what's left" of L and R to new arrays, or otherwise run into a lot of overhead in maintaining stack frames for the recursion. In practice, an iterative implementation is used, and this is what is given in Algorithm 7, and is that on which we will base our upcoming arguments. For a visual demonstration of the algorithm, see the slides that accompany these lecture notes.

Algorithm 7

Input: $L[0 \cdots (n_L - 1)], R[0 \cdots (n_R - 1)]$, arrays of size n_L and n_R, respectively
Output: M, the concatenation of L and R, sorted

```
procedure MERGE(L, R)
    M ← array indexed by 0 ··· (n_L + n_R − 1)
    ℓ ← 0
    r ← 0
    m ← 0
    while ℓ < n_L and r < n_R do
        if L[ℓ] ≤ R[r] then
            M[m] ← L[ℓ]
            ℓ ← ℓ + 1
        else
            M[m] ← R[r]
            r ← r + 1
        m ← m + 1
    while ℓ < n_L do
        M[m] ← L[ℓ]
        ℓ ← ℓ + 1
        m ← m + 1
    while r < n_R do
        M[m] ← R[r]
        r ← r + 1
        m ← m + 1
    return M
```

We have already informally argued the correctness of Algorithm 7; a formal proof can be constructed using loop invariants and is left as an exercise.

To verify the efficiency specification for Algorithm 7, we note that, for every comparison among elements of L and R, there is a corresponding increase in the variable ℓ or the variable r. Since comparisons are made only when $\ell < n_L$ and $r < n_R$, the quantity $n_L + n_R$ is an upper bound on the number of comparisons made by Algorithm 7.

This concludes the discussion of the linear-time merge procedure. This was the one missing ingredient from our previous discussion on the efficiency of merge-sort; with it completed, we now have a complete specification of an $O(n \log(n))$-time sorting algorithm.

Quick Sort A different divide and conquer approach to sorting is given by the quick sort algorithm. In quick sort, the problem is subdivided through the choice of a *pivot*, which is just some element of the input array. Once the pivot is chosen, the input array is partitioned into three arrays, consisting of elements which are respectively less than, equal to, and greater than the pivot element. The 'less-than' and 'greater-than' arrays are sorted recursively. Quick sort then returns the concatenation of the sorted 'less-than' array, the 'equal-to' array, and the 'greater-than' array, concatenated in that order. Pseudocode for this is provided in Algorithm 8, with an undefined procedure 'Get-Pivot' which determines a pivot element.

Algorithm 8

Input: $A[0 \cdots n-1]$, an array of length n
Output: B, a sorted copy of A

```
procedure Quick-Sort(A)
    if n = 1 then
        return A
    else
        p ← Get-Pivot(A)
        L, n_L ← empty array, 0
        M, n_M ← empty array, 0
        R, n_R ← empty array, 0
        for i = 0 ... (n − 1) do
            if A[i] < p then
                L[n_L] ← A[i]
                n_L ← n_L + 1
            else if A[i] = p then
                M[n_M] ← A[i]
                n_M ← n_M + 1
            else
                R[n_R] ← A[i]
                n_R ← n_R + 1
        L ← Quick-Sort(L)
        R ← Quick-Sort(R)
        return L + M + R
```

One can show that quick sort correctly sorts its input array, no matter how the pivot element is chosen. What is arguably more interesting is how quick sort's performance varies with the choice of pivot. In particular, if the pivot is always "well-chosen" (the median of the input array is always a "good" choice in this sense), then one can show that quick sort uses at most $O(n \log(n))$

comparisons on every input. On the other hand, if the pivot is poorly chosen—say the 'less-than' array is always empty—then one can show that quick sort ends up performing $\Theta(n^2)$ comparisons.

Quick sort satisfies some additional performance guarantees, in addition to upper bounds on the number of comparisons for arbitrary inputs. In particular, if one computes the *average* number of comparisons, taken over all possible re-orderings of a fixed input array, then quick-sort only makes $\Theta(n \log(n))$ comparisons. Also, if the pivot is chosen randomly among the elements of the input array, then, on average, it will makes $\Theta(n \log(n))$ comparisons. These results mean that, on most real-world instances of sorting, quick-sort still does about as well as merge-sort.

2.3 Sorting Lower Bounds

(Forthcoming)

2.4 Noncomparison Sorting

The lower bound in the previous section assumes that sorting algorithms only ever make *comparison queries* about their input arrays. In other words, the algorithms are not allowed to look at their inputs except insofar as to determine how two elements compare. In this section, we will provide a sorting algorithm which is *not* comparison-based, and indeed is able to beat the $\Omega(n \log(n))$ barrier with a running time on the order of $\Theta(n)$.

This algorithm is known as "bucket sort". The name comes from its basic idea: suppose the input array contains integers in the range $1, 2, \ldots, m$. Then we will make m buckets, $B_1, B_2, \ldots, B_m$, and we will put each occurrence of the number i of the input array into bucket B_i. We can then recover a sorted copy of the original array by pulling out the elements of bucket B_1, then the elements of bucket B_2, and so on, in that order. More formally, we have Algorithm 9.

Algorithm 9

Input: $A[0 \cdots n-1]$, an array of length n of integers in the range $1 \ldots m$
Output: B, a sorted copy of A

1: **procedure** BUCKET-SORT(A)
2: $\quad C[1 \ldots m] \leftarrow$ array of length m, initialized to 0
3: $\quad$ **for** $i = 0 \ldots n-1$ **do**
4: $\quad\quad C[A[i]] \leftarrow C[A[i]] + 1$
5: $\quad B[0 \ldots (n-1)] \leftarrow$ array of length n
6: $\quad k \leftarrow 0$
7: $\quad$ **for** $i = 1 \ldots m$ **do**
8: $\quad\quad$ **for** $j = 1 \ldots C[i]$ **do**
9: $\quad\quad\quad B[k] \leftarrow i$
10: $\quad\quad\quad k \leftarrow k + 1$
11: $\quad$ **return** B

It is easy to see that bucket sort is correct. That it runs in time $\Theta(n + m)$ is also not hard to prove[1], so we leave these as exercises. We simply add that, if $m = n$, then the input array can take on the form of any array, at least as far as the results of making comparisons is concerned.

[1] Hint: What is the value $\sum_{i=1}^{m} C[i]$ after the loop on Line (3)? How many times is Line (9) executed?

(*e.g.*, the arrays $1, 4, 2, 3$ and $17, 400, 25, 202$ appear "the same" from the perspective of making comparisons. 1 and 17 are the smallest elements, 4 and 400 are the largest, 2 and 25 the second smallest, and 3 and 202 the second largest.) This means that the sorting lower bound applies to this class of arrays. Meanwhile bucket sort is able to run in time $\Theta(n)$; it is able to do this because it uses more information than just the results of comparisons. This has a drawback: if m is very large—say on the order of n^{100}, 2^{2^n}, or even just $n\log(n)^2$—then bucket sort does poorly compared to comparison-based algorithms, which still achieve their $O(n\log(n))$ worst-case performance.

3 Counting Inversions

Another example we will see for the divide and conquer paradigm is a solution to the problem of *counting inversions*. Informally, an *inversion* in an array $A[0\ldots(n-1)]$ of n integers is a pair of positions of A whose corresponding elements are out of order. Formally, an inversion is a pair (i, j) with $0 \le i < j \le n-1$ and $A[i] > A[j]$.

Counting inversions has applications in voting theory, collaborative filtering, and analysis of search engine rankings. This is essentially because the number of inversions in an array is a heuristic by which to measure the 'sorted-ness' of the array. Consider the following properties:

- An array A is sorted if and only if it has no inversions.
- No matter how the elements of A are ordered, A has at most $\frac{(n)(n-1)}{2}$ inversions.
- An array A with all elements distinct is reverse-sorted if and only if it has $\frac{(n)(n-1)}{2}$ inversions (the maximum possible number).

We leave formal proofs of most of these facts as exercises, but we illustrate the main idea by proving that reverse-sorted arrays A have $\frac{(n)(n-1)}{2}$ inversions:

One way the see this fact is to count the number of inversions of the form $(0, j)$, the number of inversions of the form $(1, j)$, and so on, up to those of the form $(n-1, j)$, and then add all these quantities up. Since A is reverse-sorted and has every element distinct, every pair (i, j) with $i < j$ has $A[i] > A[j]$, so we simply need to count the number of pairs (i, j) with $i < j$. This yields the sum

$$(n-1) + (n-2) + \cdots + 1 + 0 = \frac{(n)(n-1)}{2}$$

Another way to see it (and which motivates the introduction of new notation) is that we are counting the number of subsets of $\{0, 1, \ldots, n-1\}$ which have size two. This is because each pair (i, j) with $i < j$ can be mapped to the set $\{i, j\}$, which has size two since $i \ne j$; conversely, the set $\{i, j\}$ can be mapped to the pair (i, j), which we can assume satisfies $i < j$ since $i \ne j$ and $\{i, j\} = \{j, i\}$. Since these mappings are inverses of each other, this means the number of inversions is equal to the number of subsets of size two. The number of subsets of $\{0, 1, \ldots, n-1\}$ of size k is a common quantity appearing in combinatorics, and is denoted $\binom{n}{k}$. So another way to express the number of inversions in a reverse-sorted array is to say it has $\binom{n}{2}$ inversions. (It should come as no surprise then that $\binom{n}{2} = \frac{(n)(n-1)}{2}$.)

Let us now formally specify the problem of counting inversions before moving on to a divide and conquer solution:

Input: A sorted array $A[0\ldots(n-1)]$ of size n

Output: The number of inversions of A

Divide and Conquer Algorithm We will now describe a divide and conquer algorithm for computing the number of inversions in a given array $A[0 \ldots (n-1)]$. Before jumping straight there, let us first observe that there is a straightforward algorithm which uses $\Theta(n^2)$ comparisons: simply check every pair of indices i, j for being an inversion, and count the ones that are. Thus our goal with the divide and conquer algorithm will be to beat this. Indeed we will give an algorithm which uses only $\Theta(n \log(n))$ comparisons.

Note that this is potentially less than the total number of inversions—our algorithm will need to count many inversions per comparison on average. How can we accomplish this? Suppose we have three indices, i, i', j with $i < j$ and $i' < j$ and we know in advance that $A[i] \leq A[i']$. Then, if we compare $A[i]$ and $A[j]$ and see that $A[i] > A[j]$, then we can deduce that *both* (i, j) *and* (i', j) are inversions. In other words, a single comparison would tell us the existence of two inversions.

Such a situation may seem difficult to get into without using too many additional comparisons. However, note that we have accomplished a very similar goal in the course of giving a $\Theta(n \log(n))$ algorithm for sorting: information on $\Theta(n^2)$ comparisons was deduced using only $\Theta(n \log(n))$ comparisons.

Since the two problems seem so related, let's consider dividing the input array A into its left and right halves, L and R respectively, just like we did in merge sort. We can easily count the number of inversions with i, j both indexing L or both indexing R—this is just recursion. But not every inversion in A has this form. The remaining inversions, however, are exactly of the form (i, j) where i indexes L and j indexes R. Since every index i into L is less than every index j into R, this is already reminiscient of the situation we described above—we just need to find a choice for i'. Better yet, we just need to know *how many* choices of i' there are, *i.e.*, how many elements of L are at least $A[i]$.

This is actually quite easy to do if L is sorted—there are exactly $|L| - i$ many! (assuming L has 0-based indexing) So a natural thing to try is to just sort L and R. The problem with this is that it leads to an algorithm whose local performance is $\Theta(n \log(n))$ instead of $\Theta(n)$ without changing the size or shape of the recursion tree. If we plug this worse local performance into our analysis, the end result is a $\Theta(n \log(n)^2)$ algorithm, which is worse than the $\Theta(n \log(n))$ that we are shooting for.

However, we can actually sort L and R *as part of the recursion.* One might imagine that we modified the specification of counting inversions to return not just the number of inversions, but also a sorted copy of the input array. Then when we recursively count the inversions in L and R, we also can assume that L and R become sorted—fantastic! Of course, we now have a more stringent requirement to meet: we now have to return a sorted copy of the input array A, in addition to the number of inversions in A. But we know how to merge L and R (after they are sorted) in linear time, and this is results in a sorted copy of A, so this is not a problem. The end result is still a locally linear-time algorithm whose recursion tree is identical in size and shape to the recursion tree for merge sort.

We are not quite done yet. We have an idea for how to count many inversions under a single comparison, but not a complete algorithm yet. In particular, what we still need is a linear-time algorithm for computing the number of inversions which cross the L–R split. This exists and uses the ideas above, but we will save its discussion until after we wrap up the divide-and-conquer aspect of counting inversions. For now, we just provide the specification for this cross-inversion-counting subroutine:

Input: Two arrays, $L[0 \ldots (n_L - 1)]$ and $R[0 \ldots (n_R - 1)]$, of lengths n_L and n_R respectively. L and R are sorted.

Output: The number of inversions in the concatenation of L then R.

Efficiency: Use at most $O(n_L + n_R)$ comparisons.

With the above specification in place, we are now ready to give an implementation of a $\Theta(n \log(n))$ algorithm for counting inversions. At a high level, we split our input array A into the two halves L and R, recursively count-and-sort L and R, apply the above-specified algorithm to count the inversions across L and R, use the merge procedure from merge-sort to get a sorted copy of A, and then finally return the total number of inversions in A and the sorted copy of A. More precise pseudocode is given in Algorithm 10, which assumes the existence of a procedure "Count-Cross" which meets the above specification.

Algorithm 10

Input: $A[0 \cdots n-1]$, an array of length n
Output: (c, B), where c is the number of inversions in A and B is a sorted copy of A

procedure Count-And-Sort(A)
  if $n = 1$ **then**
    return $(0, A)$
  else
    $m \leftarrow \lfloor n/2 \rfloor$
    $(c_L, L) \leftarrow$ Count-And-Sort($A[0 \cdots (m-1)]$)
    $(c_R, R) \leftarrow$ Count-And-Sort($A[m \cdots (n-1)]$)
    $c_{\text{cross}} \leftarrow$ Count-Cross(L, R)
    $c \leftarrow c_L + c_R + c_{\text{cross}}$
    $B \leftarrow$ Merge(L, R)
    return (c, B)

Taking Count-Cross as a blackblox and using the above discussion for the main ideas, it is reasonably straightforward to prove correctness of the above algorithm. We leave filling in these details as an exercise.

Regarding efficiency, the recursion tree for Algorithm 10 is identical to the recursion tree for Algorithm 6. Using the fact that the efficiency of the 'Count-Cross' routine is assumed to be linear, the remainder of the analysis for the efficiency of Algorithm 6 also goes through for Algorithm 10. Then end result is a running time of $\Theta(n \log(n))$.

Count-Cross We now detail the algorithm 'Count-Cross', which takes in two arrays L and R and outputs the number of inversions in the concatenation of L then R. We let $L + R$ denote the concatenation.

To start, let us go back to our original motivation for looking at merge sort: we know that if L and R are sorted, then we can hope to come up with a more efficient means of counting the inversions of $L + R$. Indeed, suppose we have a pair (i, j) with i a position of L and j a position of R, and further suppose that i is the smallest index of L for which $(i, n_L + j)$ is an inversion of $L + R$. We know that $(i', n_L + j)$ is not an inversion for every $i' < i$, by how we chose i. But we also know that $(i', n_L + j)$ is an inversion for every i' indexing L with $i \leq i'$: we have $L[i] < L[i']$ (from sortedness of L) and we have $R[j] < L[i]$ (since $(i, n_L + j)$ is an inversion of $L + R$), and thus we can conclude that $R[j] < L[i']$. This means we can deduce that there are exactly $n_L - i$ indices i' indexing L for which $(i', n_L + j)$ is an inversion of $L + R$.

We now almost have a complete algorithm. The remaining piece is to provide a means of finding the smallest i so that $(i, n_L + j)$ is an inversion for each j indexing R.

Binary search would be a reasonable choice here: for each j, we can find the smallest i for which $(i, n_L + j)$ is an inversion in $\Theta(\log(n))$ comparisons. However, this will lead to an inversion-counting algorithm which makes $\Theta(n \log^2(n))$ comparisons overall.

Instead, we can do better with the following observation: suppose we have the smallest index i so that $(i, n_L + j)$ is an inversion, and we wish to find the smallest index i' so that $(i', n_L + j + 1)$ is an inversion. Since $R[j] < R[j + 1]$ and L is sorted, we can infer that $i' \geq i$. So let's suppose we do a linear scan *starting at* i (ie try $i' = i$, then $i' = i + 1$, etc.) to find i'. How many comparisons does this take? For a single index j, it's possible we will scan over the entire array L. However, when we consider the comparisons across *all* j, there are at most $n_L + n_R$ comparisons used. The reasoning for this is similar to the reasoning we used to show that the merge procedure (Algorithm 7) uses at most a linear number of comparisons.

With all these ideas in place, we can now present a formal algorithm to compute the number of inversions in an array. This is given by Algorithm 11.

Algorithm 11

Input: $L[0 \cdots (n_L - 1)], R[0 \cdots (n_R - 1)]$, sorted arrays of size n_L and n_R, respectively
Output: c, the number of pairs (i, j) with $L[i] > R[j]$

1: **procedure** COUNT-CROSS(L, R)
2:     $\ell \leftarrow 0$
3:     $r \leftarrow 0$
4:     $c \leftarrow 0$
5:     **while** $\ell < n_L$ and $r < n_R$ **do**
6:         **if** $L[\ell] \leq R[r]$ **then**
7:             $\ell \leftarrow \ell + 1$
8:         **else**
9:             $c \leftarrow c + n_L - \ell$
10:             $r \leftarrow r + 1$
11:     **return** c

An example simulation of this algorithm is provided in the slides accompanying these lecture notes.

The correctness of Algorithm 11 essentially follows from our previous reasoning. It can be formally argued using loop invariants. We leave the full argument as an exercise, but point out that whenever Line 9 is executed, ℓ is the least index so that $(\ell, n_L + r + 1)$ is an inversion. The efficiency of Algorithm 11 can be proven similar to the efficiency of Algorithm 7.

We do wish to point out the similarity with Algorithm 7. Indeed, the two procedures can be merged into a single-pass 'merge-and-count' procedure, which can be used to simplify the implementation of Algorithm 10.

4 Closest Pair of Points in the Plane

We now move on to a new problem of a more geometric flavor. This problem is to find the closest pair of points from among some set of points in the plane. The formal specification is the following:

Input: A finite set of points X in the plane.

Output: The smallest distance between a pair of points of X.

Efficiency: Use at most $O(n \log(n))$ steps.

A naïve approach would be to compute distances for every pair of points in X, and outputting the smallest of these distances. This approach requires computing $\binom{n}{2}$ distances, and thus must use at least $\Omega(n^2)$ many steps.

The naïve algorithm however does not take advantage of all of the structure in this problem. Put another way, this algorithm does not distinguish much between the problem of finding the closest pair of points in the plane, and finding the minimum number in an arbitrary array of $\Theta(n^2)$ elements. On the other hand, we know that if two points have many points 'between' them (speaking informally), then we do not have to consider these two points for being the closest pair of points. Exploiting this kind of phenomenon allows us to develop an asymptotically faster algorithm for finding the closest pair of points.

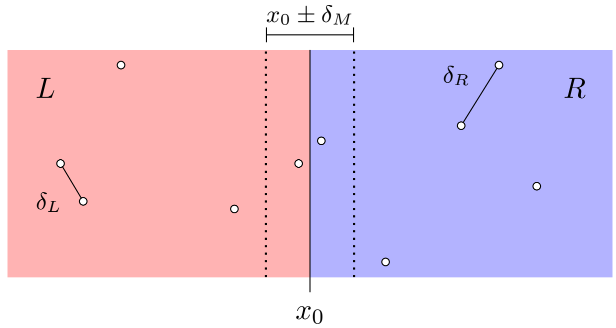
Divide and Conquer Algorithm As hinted by the theme of this lecture, we will use a divide and conquer strategy.

It will help convey the intuition behind the algorithm if we assume a certain kind of access to sets of points in the plane. Specifically, for any given set S of points, we assume that we can enumerate the points in order of increasing x-coordinates (with ties broken in favor of increasing y-coordinates), and that we can enumerate the points in order of increasing y-coordinates (with ties broken in favor of increasing x-coordinates). If the order of enumeration is contextually important, we will denote these two enumerations by $S_{\mathtt{x}}$ and $S_{\mathtt{y}}$ respectively. Having the ability to access a set of points in this way may seem overly-powerful; however, we will address this at the end of this section. For now, let us just assume that we have it.

A natural divide-and-conquer approach for computing the closest pair of points in X begins by separating the input set of points X into 'left' and 'right' halves L and R according to $X_{\mathtt{x}}$. The approach then recursively solves these subproblems to find the smallest distance between a pair of points in L and the smallest distance between a pair of points in R. All that remains is to consider distances between pairs of points with one point in L and the other point in R. The smallest of these three quantities is then the smallest distance between a pair of points in X. Recursion automatically takes care of the first two quantities, so if we can figure the third one out, we can give a complete algorithm.

To work out the smallest distance between points in L and points in R, let's first see what information we have available to us. We have an x-value x_0 so that about half the points (those in L) of X have x-value at most x_0, and the remaining half (those in R) have x-value at least x_0. After recursing on L and R, we obtained values δ_L and δ_R for the minimum distance among points in L and among points in R respectively. A pictorial representation of this setup is provided in Figure 3.

Figure 3: Divide and conquer approach to the closest pair of points problem



The vertical line at $x = x_0$ separates the points into the red region (L) and blue region (R). The shortest distance between points in L and are R are computed recursively, yielding δ_L and δ_R respectively. The combination step can then just consider the points in L and R which are less than $\delta_M \doteq \min(\delta_L, \delta_R)$ from the vertical line $x = x_0$.

The picture suggests that, in order to compute the smallest distance between a point in L and a point in R, we should focus mostly on points which have x-coordinate close to x_0. But how should we define close?

Let δ be the smallest distance between a pair of points in X, *i.e.*, the answer we are looking for. We know that δ_L and δ_R are upper bounds for δ. For simplicity, let $\delta_M \doteq \min(\delta_L, \delta_R)$, which is also an upper bound for δ. Let δ_{cross} be the smallest distance between a point in L and a point in R. We are interested in computing δ_{cross}, but we know $\delta = \min(\delta_{\text{cross}}, \delta_M)$, so, really, we are only interested in computing δ_{cross} if it is less than δ_M. Since points in L which have x-coordinate at most $x_0 - \delta_M$ have distance at least δ_M from any point in R, we can rule out every one of those points. Similarly, we can rule out any points in R with x-coordinate at least $x_0 + \delta_M$.

Already this sounds like it might be a pretty substantial speed up: if we imagine the points as having been placed haphazardly across the plane, then we should expect there to be very few that are within δ_M of the line $x = x_0$. However, as Figure 4 demonstrates, this will not always help us reduce the total number of points we consider.

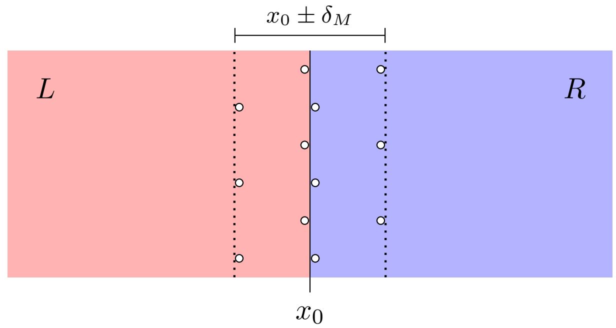
That being said, note that the configuration of points in Figure 4 is quite careful—if we try moving points around much, we will likely shrink the value of δ_M and narrow the strip around x_0. This suggests that restricting our attention only to points near the line $x = x_0$ has somehow imposed extra structure on our problem, and we might be able to exploit the added structure to get a fast combination step.

As it turns out, this is indeed the case. However, the details are a bit much to cover right now, so we will simply provide the specification for a subroutine that does what we need, and come back to the implementation later. Do note, though, that, once we restrict our attention to points in L and R which are within δ_M of the line $x = x_0$, these points will satisfy the preconditions below:

Input: L, R, two sets of points; δ_M some number. These inputs are such that there exists a number x_0 satisfying the following:

- No two points in L are within δ_M of each other.

Figure 4: Some configurations of points are all very close to the center line



Each pair of points in L (or pair of points in R) has distance at least δ_M from one another, but every point has x-coordinate less than δ_M from x_0.

- No two points in R are within δ_M of each other.
- Every point of L has x-coordinate strictly greater than $x_0 - \delta_M$.
- Every point of L has x-coordinate at most x_0.
- Every point of R has x-coordinate at least x_0.
- Every point of R has x-coordinate strictly less than $x_0 + \delta_M$.

Output: δ_{cross}, the smallest distance between a point in L and a point in R.

Efficiency: Use at most $O(|L| + |R|)$ operations.

Let Closest-Across-Split be an implementation of the above specification. Then our final divide and conquer algorithm for finding the closest pair of points of X works as follows: split X into L and R, recursively solve L and R and compute δ_M, then run Closest-Across-Split on (L, R, δ_M) to get δ_{cross}, and finally return $\min(\delta_{\text{cross}}, \delta_M)$. A pseudocode implementation of this is given in Algorithm 12.

The correctness of Algorithm 12 follows straightforwardly from the preceding discussion and the correctness of the Closest-Across-Split subroutine.

The efficiency of Algorithm 12 will follow the same general outline as for Merge-Sort in Algorithm 6: We start by associating to each recursive call in the recursion tree the amount of work performed by that call, and then sum the total work across every node in the recursion tree.

For the local performance, recall that we are assuming (for now) that accessing $X_{\mathbf{x}}$ has the same cost as a normal array lookup. We are also assuming that Closest-Across-Split runs in time linear in $|L| + |R| = n$. It is then easy to see that Algorithm 12 locally runs in linear time.[2] Since the recursion tree for Closest-Pair has the same size and shape as the one for Merge-Sort, the same analysis as there shows that Closest-Pair overall runs in time $\Theta(n \log(n))$.

[2] In most real-world languages, the 'Remove' operation runs in time linear in the size of the array, whereas here we are considering it to be an atomic (unit-cost) operation. Implemented as-is, this algorithm would then locally use quadratic time, which is not sufficient for our purposes. However, since all of our removing happens in a single pass, we can work around this in real-world languages. The workaround is to instead build up auxiliary arrays L' and R' of elements which are not removed from L and R respectively, and then use L' and R' in the call to Closest-Across-Split. We did not do this here in order to preserve understandability of the pseudocode.

Algorithm 12

Input: $X[0, \ldots, n-1]$, an array of size n of points in the plane
Output: δ, the smallest distance between two points of X

```
procedure Closest-Pair(X)
    if n = 1 then
        return ∞
    else
        m ← ⌊n/2⌋
        L[0 ... m − 1] ← X_x[0 ... m − 1]
        R[0 ... n − m − 1] ← X_x[m ... n − 1]
        δ_L ← Closest-Pair(L)
        δ_R ← Closest-Pair(R)
        x_0 ← x-coordinate of X_x[m]
        δ_M ← min(δ_L, δ_R)
        for i = 0 ... m − 1 do
            if L[i].x ≤ x_0 − δ_M then
                Remove L[i] from L
        for i = 0 ... n − m − 1 do
            if R[i].x ≥ x_0 + δ_M then
                Remove R[i] from R
        δ_cross ← Closest-Across-Split(L, R, δ_M)
        δ ← min(δ_M, δ_cross)
        return δ
```

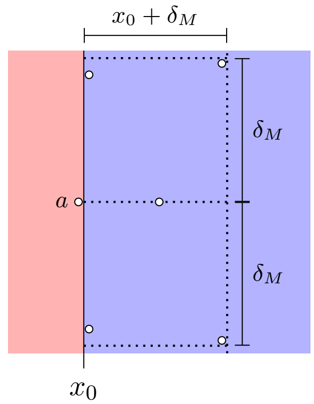
Note that if we were to sort the points X in order of $X_{\mathtt{x}}$ in every recursive call, then this recursive algorithm would locally require $O(n \log(n))$ time. If we plug this into the logic for Merge-Sort's efficiency, this leads to a running time of $O(n \log(n)^2)$ overall, which is asymptotically worse than $O(n \log(n))$. An easy fix for this would just be to sort the points of X in order of $X_{\mathtt{x}}$ once in a wrapping function, and add a precondition that X be sorted in the specification of Closest-Pair. However, Closest-Across-Split will actually need to be able to access the points in L and R in order of $L_{\mathtt{y}}$ and $R_{\mathtt{y}}$, which we cannot compute faster than sorting. That being said, we will still be able to address these problems, but we save this until after giving the implementation for Closest-Across-Split.

Closest-Across-Split We now provide the implementation of Closest-Across-Split, and show that it meets the specifications outlined in the previous section.

Recall the list of preconditions we placed on the input. A high level interpretation of these preconditions is that points on the same side of x_0 cannot be too close to each other, while we are only interested in comparing points in L with points in R when they are close together. The key insight behind the development Closest-Across-Split is that, if the point a in L is close to all of the points $b_1, b_2, \ldots, b_k$ in R, and if all the b_i's are far apart from each other, then there can only be a few of them (*i.e.*, k must be small).

As it turns out, one can show that for a fixed point a in L, there can be at most *four* points b_1, b_2, b_3, b_4 in R so that the distance between a and b_i is at most δ_M for each i, while the distance

Figure 5: Visual aide for Closest-Across-Split



The five points in the blue region are all distance at least δ_M from each other. (This is perhaps easiest to visualize by imagining non-overlapping circles of radius $\delta_M/2$ based at each point.) They also have y-coordinates in the range $(a.y - \delta_M, a.y + \delta_M)$, and x-coordinates between x_0 and $x_0 + \delta_M$.

between b_i and b_j for $i \neq j$ is at least δ_M.

This (or rather a slight variant thereof) is what enables us to get a linear-time implementation of Closest-Across-Split. In particular, since we are only interested in distances between points a in L and b in R when the distance between them is less than δ_M, and since there are at most four such choices of b for any particular a, we can hope to only ever consider $4|L| = O(|L|)$ such pairs a and b. Of course, we should also expect to consider every point in R too, so a more refined hope is to make $O(|L| + |R|)$ comparisons.

The main difficulty in achieving this goal is finding, for each point a in L, the set of points in R that are within δ_M of a. Graphically (refer *e.g.*, to Figure 5), it is easy to observe that all of these points have similar y-values as a. More precisely we know that we are only interested in choices of b in R with y-coordinate between $a.y - \delta_M$ and $a.y + \delta_M$. These points are easy to locate if we suppose that the points in R are sorted by their y-coordinates. But it's possible that this range includes some—ostensibly many—false-positives. This can happen, for instance, with points b which are further than δ_M from a, but still have y-coordinate within δ_M of $a.y$. For instance, it could be that the x-coordinate of b is far from $a.x$.

However, every point b in R has to be within δ_M of x_0. This means that the set of points in R with y-values within δ_M of $a.y$ are very highly constrained: they all have to lie within a fixed rectangle of width δ_M and height $2 \cdot \delta_M$, *and* they have to be at least δ_M apart from each other. As you might guess, only a small number of points can do this; in fact five is the maximum.[3]

Thus we know that the following algorithm is correct: we consider R in order of $R_\mathtt{y}$. For each point a in L, find the point $b(a)$ in $R_\mathtt{y}$ which is the first point b in $R_\mathtt{y}$ with $b.y > a.y - \delta_M$. Suppose $b(a)$ is at position $i(a)$. Then consider the distance between a and each of the points

[3] The following is a proof of an upper bound of seven: Packing k points into a rectangle of height $2\delta_M$ and width δ_M so that no two points are within δ_M of each other is equivalent to packing k circles of radius 1 into a rectangle of height 6 and width 4 so that no two circles overlap except on their boundaries. (The points correspond with the centers of the circles.) The area inside the k circles sums to $k\pi$, while the area of the rectangle is 24. Since we must have $k\pi \leq 24$ in order to get all the circles inside the rectangle without overlapping, we have $k \leq 24/\pi$. Since k is integral, we have $k \leq \lfloor 24/\pi \rfloor = 7$.

$R_{\mathtt{y}}[i(a)], R_{\mathtt{y}}[i(a)+1], \ldots, R_{\mathtt{y}}[i(a)+4]$ whenever defined. The smallest of these distances, taken over all a in L, is the value δ_{cross}.

The efficiency of this algorithm is suspect though. For each choice of a in L, we can find $b(a)$ via binary search, but this yields a running time of $O(|L| \log(|R|))$, which is not good enough.

However, consider the points from L in order of $L_{\mathtt{y}}$, and let a' come after a in this order. Then we know that $a'.y \geq a.y$, which means $b(a').y \geq b(a).y$, and thus $i(a') \geq i(a)$; *i.e.*, we know that $b(a')$ comes after $b(a)$ in $R_{\mathtt{y}}$. This means we can implement a similar "walking-pointer" technique as in the Merge algorithm in merge sort. Specifically, we find $i(a')$ by linearly scanning forward from $i(a)$.

Plugging this approach into our existing algorithm, we get the implementation given in Algorithm 13.

Algorithm 13

Input: $L[0 \ldots n_L - 1]$, $R[0 \ldots n_R - 1]$ and δ_M, satisfying the necessary preconditions
Output: δ_{cross}, the smallest distance between a point of L and a point of R

1: **procedure** CLOSEST-ACROSS-SPLIT(L, R, δ_M)
2: $\quad$ $\delta_{\text{cross}} = \infty$
3: $\quad$ $r \leftarrow 0$
4: $\quad$ **for** $\ell = 0 \ldots n_L - 1$ **do**
5: $\quad\quad$ **while** $r < n_R$ and $R_{\mathtt{y}}[r].y < L_{\mathtt{y}}[\ell].y - \delta_M$ **do**
6: $\quad\quad\quad$ $r \leftarrow r + 1$
7: $\quad\quad$ **for** $d = 0 \ldots 4$ **do**
8: $\quad\quad\quad$ **if** $r + d < n_R$ **then**
9: $\quad\quad\quad\quad$ dist $\leftarrow$ distance from $L_{\mathtt{y}}[\ell]$ to $R_{\mathtt{y}}[r+d]$
10: $\quad\quad\quad\quad$ $\delta_{\text{cross}} \leftarrow \min(\delta_{\text{cross}}, \text{dist})$
11: $\quad$ **return** δ_{cross}

Its correctness follows from our previous discussion. The efficiency requirements follow from the following three facts, which can be easily verified by inspection:

- Lines (2) and (3) are executed once, and each costs a constant amount of work.
- Lines (5) and (6) are executed at most $|R|$ many times.
- The remaining lines are executed at most $6 \cdot |L| + 1$ times.

When adding these contributions together, we get a total running time bound of $O(|L| + |R|)$, as desired.

Addressing the sorting assumption In Algorithms 12 and 13, we assumed access to sets S of points in the orders $S_{\mathtt{x}}$ and $S_{\mathtt{y}}$ with the promise that it would be addressed later. Let's now address it.

As hinted at above, we can add a wrapping procedure around Algorithm 12 which sorts its input points X in the order $X_{\mathtt{x}}$, and then add this as a precondition to Algorithm 12. The problem with this was that Algorithm 13 needed to be able to access its points L and R in the orders $L_{\mathtt{y}}$ and $R_{\mathtt{y}}$, which cannot be computed from $L_{\mathtt{x}}$ and $R_{\mathtt{y}}$ without sorting.

However, we can adapt the wrapping procedure to sort X in *both* orders $X_{\mathtt{x}}$ and $X_{\mathtt{y}}$, and pass these into Algorithm 12. Most of the details for adapting Algorithms 12 and 13 are clear, but it might not be obvious how to compute $L_{\mathtt{y}}$ and $R_{\mathtt{y}}$.

This turns out not to be too difficult since we know the point $X_{\mathtt{x}}[m]$. The observation is that a point is in L if and only if it has a smaller x-coordinate than $X_{\mathtt{x}}[m]$, or else the same x-coordinate and a smaller y-coordinate. Thus we can enumerate the points of X in the order $X_{\mathtt{y}}$, and include them in $L_{\mathtt{y}}$ if they satisfy that condition, or else put them in $R_{\mathtt{y}}$.

5 Integer Multiplication

When we covered the powering problem, we mentioned that assuming multiplications have unit-cost is not always a reasonable assumption to make. The reason for this is that when computing large powers of numbers, the result has a large bit-length—much larger than what fits into a typical word of RAM. For instance, even 2^{65} would not fit into a typical integer type on modern 64-bit machines, and only asking for 2^{500} outpaces even what the IEEE 754 64-bit double type can store. On the other hand, modern RSA encryption involves working with numbers on the order of 2^{1024}, 2^{2048}, or even 2^{4096}. Testing primality of the largest known prime number, the Mersenne prime $2^{57,885,161} - 1$, required multiplying numbers with tens of millions of bits. In these situations, the cost of multiplying numbers is considerable. Hence a fast algorithm for multiplication of integers is desirable.

To start, let's specify the problem:

Input: Two numbers a and b, in binary. We let n be the number of bits in a and b.[4]

Output: The number $a \cdot b$, in binary. (This number will have at most $2n$ bits.)

We all know how to multiply numbers in base 10, as it's an essential requirement for passing early grade school. We can give a similar algorithm for multiplying numbers in binary, which we illustrate by computing the product $(10010)_2 \cdot (10111)_2$:

$$
\begin{array}{ccccccccccc}
 & & & & & & 1 & 0 & 0 & 1 & 0 \\
\times & & & & & & 1 & 0 & 1 & 0 & 1 \\
\hline
 & & & & & & 1 & 0 & 0 & 1 & 0 \\
 & & & & 1 & 0 & 0 & 1 & 0 & & \\
 & & & 1 & 0 & 0 & 1 & 0 & & & \\
+ & & 1 & 0 & 0 & 1 & 0 & & & & \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{array}
$$

It is straightforward to see that this algorithm runs in time $O(n^2)$, and furthermore that this is tight, as witnessed by its performance on the multiplication of $(111 \cdots 1)_2$ with itself. As we will soon see, we can improve this to $O(n^{\log_2(3)})$ with divide and conquer.

Divide and Conquer, Round 1 We begin with a divide and conquer algorithm which will serve as a basis for our final algorithm. This first algorithm will run in time $\Theta(n^2)$, which is no improvement over the grade-school algorithm. However, we will use an algebraic trick to shrink

[4] We can assume that a and b both have the same number of bits by simply padding the shorter one with zeros.

the recursion tree without asymptotically increasing the local amount of work at any node, which will net us the improved running time. Before jumping to that however, it is instructive to analyze this first, more intuitive algorithm.

We can think of the input numbers as arrays of bits. Then, like with most divide and conquer algorithms, we divide our input arrays into halves, apply recursion, and reconstruct the solution to the original instance. For simplicity, let's assume that n is even. (This is without loss of generality, since we can always add an extra 0 to a and b to increase n by 1.) Splitting our 'array' a means writing it as $a = a_L 2^{n/2} + a_R$ for the unique $n/2$-bit integers a_L and a_R for which $0 \leq a_R < 2^{n/2}$. Similarly define b_L and b_R from b so that $b = b_R 2^{n/2} + b_L$. Then we can express the product $a \cdot b$ as

$$\begin{aligned} a \cdot b &= (a_L 2^{n/2} + a_R) \cdot (b_L 2^{n/2} + b_R) \\ &= a_L b_L \cdot 2^n + a_L b_R \cdot 2^{n/2} + a_R b_L \cdot 2^{n/2} + a_R b_R \end{aligned}$$

Note that multiplying by 2^k for any value k is very easy for us—we just have to add zeros to the end, similar to how multiplying by powers of ten means putting zeroes at the front of a number in base ten. The remaining numbers only have $n/2$ bits in their representations. In other words, we can recursively compute the products $a_L b_L, a_L b_R, a_R b_L, a_R b_R$, then we can simply compute the appropriate shifts and add all of the results. This is our complete divide and conquer algorithm, and it is easily seen to be correct. For completeness, pseudocode is given as Algorithm 14.

Algorithm 14

Input: a, b, two n-bit integers
Output: $a \cdot b$

procedure INTEGER-MULTIPLICATION-SIMPLE(a, b)
  if $n = 1$ **then**
    return $a \cdot b$
  else
    if n is odd **then**
      Pad `0` to the front of a and b
      $n \leftarrow n + 1$
    $a_L \leftarrow a$ **mod** $2^{n/2}$
    $a_R \leftarrow (a - a_L)/2^{n/2}$
    $b_L \leftarrow b$ **mod** $2^{n/2}$
    $b_R \leftarrow (b - b_L)/2^{n/2}$
    $c_{LL} \leftarrow$ INTEGER-MULTIPLICATION-SIMPLE(a_L,b_L)
    $c_{LR} \leftarrow$ INTEGER-MULTIPLICATION-SIMPLE(a_L,b_R)
    $c_{RL} \leftarrow$ INTEGER-MULTIPLICATION-SIMPLE(a_R,b_L)
    $c_{RR} \leftarrow$ INTEGER-MULTIPLICATION-SIMPLE(a_R,b_R)
    $A \leftarrow c_{LL}$ shifted by n positions
    $B \leftarrow (c_{LR} + c_{RL})$ shifted by $n/2$ positions
    $C \leftarrow c_{RR}$
    return $(A + B + C)$

As for the running time, note that addition and shifting take linear time, which means our local work will be linear. However, we are now making four recursive calls instead of two, despite only shrinking the problem size by a factor of two. Let's see how this affects our running time:

Since we know the local work in our recursion, we just need to understand the size and shape of the recursion tree. Each non-leaf node on the tree has exactly four children, and each child has problem size reduced by half. This means the tree has height at most $\log(n) + 1$, and the total work at level d is $\Theta(4^d \cdot n/2^d) = \Theta(2^d \cdot n)$. Summing this over all levels $d = 0 \ldots \log(n) + 1$ yields the formula

$$\sum_{d=0}^{\log_2(n)+1} c2^d n$$

for the total work, where c is the constant from the $\Theta(\,\cdot\,)$ notation. With some additional algebraic manipulation, we have

$$\begin{aligned}
\sum_{d=0}^{\log_2(n)+1} c2^d n &= cn \cdot \sum_{d=0}^{\log_2(n)+1} 2^d && (2)\\
&= cn \cdot \left(\frac{2^{\log_2(n)+2} - 1}{2-1}\right) && (3)\\
&= cn \cdot \left(\frac{4n-1}{2-1}\right) && (4)\\
&= \Theta(n^2) && (5)
\end{aligned}$$

where Line (3) follows from Line (2) by the following identity for geometric sums

$$\sum_{i=0}^{k} \alpha^i = \begin{cases} \frac{\alpha^{k+1}-1}{\alpha-1} & : \alpha \neq 1 \\ k+1 & : \alpha = 1 \end{cases}$$

With this analysis in mind, let's move on to the improved divide and conquer algorithm.

Divide and Conquer, Round 2 Having just seen the previous divide and conquer algorithm, let's see how to improve it. The main idea is that there are really only three quantities we are interested in computing before applying shifts and adding. These three quantities are the coefficients of the powers of 2 below:

$$a \cdot b = (a_L b_L) \cdot 2^n + (a_L b_R + a_R b_L) \cdot 2^{n/2} + (a_R b_R)$$

What we did in our original divide and conquer strategy was compute each of these coefficients in the straightforward, 'most obvious' way: we computed $a_L b_L, a_L b_R, a_R b_L$, and $a_R b_R$ recursively, and then added $a_L b_R$ and $a_R b_L$ to get $(a_L b_R + a_R b_L)$. However, we can hope to compute the value $(a_L b_R + a_R b_L)$ with fewer multiplications. In fact we can; consider the following algebraic identity:

$$a_L b_R + a_R b_L = (a_L + a_R)(b_L + b_R) - a_L b_L - a_R b_R \tag{6}$$

While this may appear to use more multiplications, the fact is that we are *already* computing the values $a_L b_L$ and $a_R b_R$. So we can re-use them here in order to derive the value $a_L b_R + a_R b_L$ after only computing the product $(a_L + a_R)(b_L + b_R)$ and some additions. In other words, we will compute $a_L b_L, a_R b_R$, and $(a_L + a_R)(b_L + b_R)$ recursively, and then compute $(a_L b_R + a_R b_L)$ using the equation (6). This uses only three recursive calls. All together, this gives us Algorithm 15.

Algorithm 15

Input: a, b, two n-bit integers
Output: $a \cdot b$

procedure INTEGER-MULTIPLICATION(a, b)
    if $n = 1$ **then**
        return $a \cdot b$
    else
        if n is odd **then**
            Pad `0` to the front of a and b
            $n \leftarrow n + 1$
        $a_L \leftarrow a \textbf{ mod } 2^{n/2}$ ▷ first $n/2$ bits
        $a_R \leftarrow (a - a_L)/2^{n/2}$ ▷ second $n/2$ bits
        $b_L \leftarrow b \textbf{ mod } 2^{n/2}$
        $b_R \leftarrow (b - b_L)/2^{n/2}$
        $c_{LL} \leftarrow$ INTEGER-MULTIPLICATION(a_L,b_L)
        $c_{RR} \leftarrow$ INTEGER-MULTIPLICATION(a_R,b_R)
        $c_M \leftarrow$ INTEGER-MULTIPLICATION($a_L + a_R$, $b_L + b_R$) $-c_{LL} - c_{RR}$
        $A \leftarrow c_{LL}$ shifted by n positions
        $B \leftarrow c_M$ shifted by $n/2$ positions
        $C \leftarrow c_{RR}$
        return $(A + B + C)$

The correctness of Algorithm 15 follows from the above discussion.

Its efficiency can be analyzed in the same way as we analyzed Algorithm 14, except that now every non-leaf node in the recursion tree has only three children. This leads to the following formula for the total amount of work performed by Algorithm 15:

$$\sum_{d=0}^{\log_2(n)+1} c\left(\frac{3}{2}\right)^d n$$

The same algebraic manipulation as before then gives a total running time of

$$\begin{aligned}
\sum_{d=0}^{\log_2(n)+1} c\left(\frac{3}{2}\right)^d n &= cn \cdot \sum_{d=0}^{\log_2(n)+1} \left(\frac{3}{2}\right)^d && (7)\\
&= cn \cdot \left(\frac{\left(\frac{3}{2}\right)^{\log_2(n)+2} - 1}{\left(\frac{3}{2}\right) - 1}\right) && (8)\\
&= \Theta(n^{\log_2(3)}) && (9)
\end{aligned}$$

6 Selection

Up until now, most of the recursion trees we have seen have been fairly simple. They were either lines or balanced trees, and the work done in a particular node of the tree depended only on the depth of the node. We were able to easily compute the total work done by the algorithm by simply adding all the contributions within each layer of the tree, and then adding all the layers' contributions together.

In this section we develop a linear-time algorithm for the selection problem. What will be interesting here is the shape of the recursion tree—it will no longer be balanced, and the work done in a particular node will vary even among nodes of the same depth. This complicates the analysis framework we have employed previously. However, we will still be able to apply the general strategy from before, where we associate to each node in the recursion tree the amount of work done locally at that node, and then add up the contributions of every node in the tree.

Our algorithm will also feature some parameters, which we won't fix until the end of the analysis—think of them as hard coded constants. With poor choices of parameters, we will get a slow algorithm, but under the right choice of parameters, we will actually be able to have an algorithm that runs in *linear* time, as opposed to the "$n \log(n)$" or similar running times we have seen before.

Let's begin by specifying the selection problem itself:

Input: An array A of n numbers, and an integer k with $1 \leq k \leq n$.

Output: The k-th element (using 1-based indexing) of A when sorted in non-decreasing order.

Motivation Interesting special cases include $k = 1$ (minimum), $k = n$ (maximum), and $k = \lceil n/2 \rceil$ (median). As we will soon see, finding the median element turns out to be the 'most difficult', in the sense that a good algorithm for finding the median can be turned into a good algorithm for selection in general. Here are a couple reasons why finding medians is particularly interesting in its own right:

- Sometimes the median is the final solution to the problem. An example is the *driveway problem*. We are given the coordinates of n houses in the plane, and want to figure out where to build a street parallel to the x-axis so as to minimize the total length of all driveways. Driveways will be constructed between each house and the new street, and run parallel to the y-axis.

  To see how the median provides "the answer" to this problem, let's assume for simplicity that the y-coordinates of all n houses are distinct. If we build the street at y-coordinate y^*, shifting it upwards over a small distance δ shortens the driveway by δ for those houses with y-coordinate larger than y^*, and lengthens the driveway by δ for the other houses. Thus, the ideal y^* is such that there are as many houses above it as below, since otherwise we could move the new street up or down and have a shorter total length of driveways. For odd n, this is exactly the median of the y-coordinates. For even n, any location between the $\frac{n}{2}$-th y-coordinate and the next one is optimal.

- Oftentimes finding the median is an intermediate step in the solution of a larger problem. In particular, if we try to solve a problem on an array A of numbers using divide-and-conquer, we may want to split A into a left array L and a right array R such that both are about the same size and all elements of L come before the elements of R. For example, this is what one aims to do in quick-sort.

  One way to realize it is by first figuring out the median m of A, and then dividing the elements of A into L and R according to whether they are less than or greater than m (and equally dividing the copies of m).

6.1 Algorithms

Here are some simple approaches to the selection problem:

- One approach consists of sorting A, and then picking the kth element of the sorted array. Due to the sorting, the best running time we can achieve this way is $\Theta(n \log(n))$.
- Another attempt is to split A into two halves of equal size, and recurse on both of them, just like we did in merge-sort and in our algorithm for finding a closest pair of points in the plane. It's not immediately clear how precisely to fill in the details here, but we can infer one thing: creating the halves takes at least linear time, and so the resulting running time is $\Theta(n \log n)$ at best.

This second idea hints at an essential barrier to applying divide and conquer to median finding—our algorithm *cannot* divide its array in half, recurse on each half, and ultimately be a linear-time algorithm. One way to get around this is, instead of reducing to two halves, we can attempt to reduce to a *single* half.

Suppose for a second that we can find the median m of A in linear time, and consider the subarrays L and R, which contain, respectively, the elements less than m and greater than m from A. Suppose we also distribute the copies of m equally among L and R, so that $|L| + |R| = |A|$. If $k \leq |L|$, then the k-th smallest element of A is also the k-th smallest element of L, so we can recurse on L. If $k > |L|$, then the k-th smallest element of A is also the $(k - |L|)$-th smallest element of R, so we can recurse on R. (If the indexing is confusing, see the diagram below.)

$$
\begin{array}{lcccccccc}
L\colon & 1 & 2 & \cdots & |L| & & & \\
 & a_1 & a_1 & \cdots & a_{|L|} & a_{|L|+1} & \cdots & a_n \\
R\colon & & & & & 1 & \cdots & |R|
\end{array}
$$

The resulting recursion tree is just a line, as shown in Figure 6. Because we found the exact median of A, each node has input size half the input size of its parent. Since the amount of work associated locally with a node is linear in its input size, say at most c times its input size, the total amount of work across the entire tree is

$$cn + c(n/2) + c(n/4) + \cdots + c(n/2^{\log(n)}) \leq \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right) \cdot c \cdot n \leq 2c \cdot n = O(n)$$

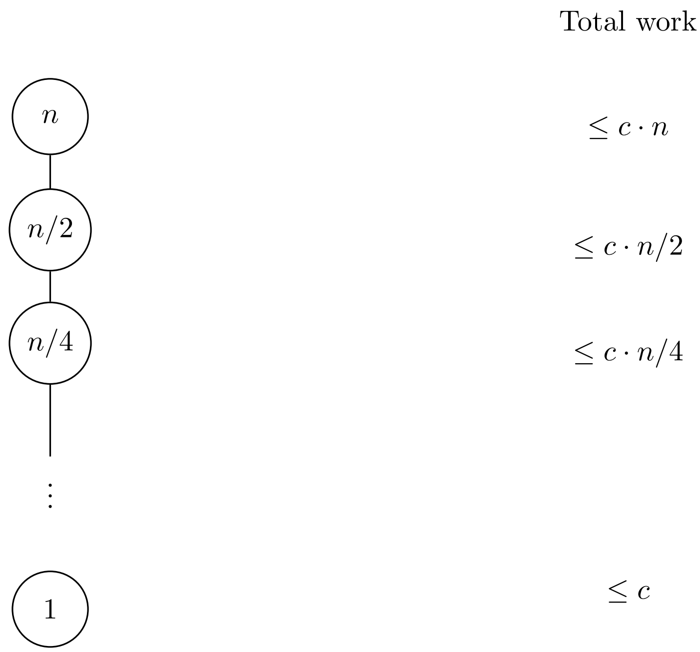
where we used the fact that the geometric series with ratio $\frac{1}{2}$ sums to 2.

This approach seems—and is—a vicious circle, especially if our goal is to find the median of A: In order to find it in linear time, we are assuming that we can find it in linear time, which means we do not seem to have gotten anywhere at all.

However, notice that for the correctness of the above procedure, the fact that m is the median of A does not matter—any choice of m works, similar to how every choice of pivot for quick-sort leads to a correct sorting. The fact that m is the median only comes into play in the analysis of the running time. In this regard, notice that for the above recursive procedure to run in linear time, it actually suffices to guarantee that both subarrays L and R have size at most $\rho \cdot n$ for *some* constant $\rho < 1$; we do not actually need $\rho = \frac{1}{2}$. Indeed, the expression for the running time then becomes

$$c \cdot \left(1 + \rho + \rho^2 + \cdots\right) \cdot n \leq c \cdot \frac{1}{1 - \rho} \cdot n,$$

Figure 6: Recursion tree for our first attempt at a selection algorithm



where we used the general expression for the sum of a geometric series with ratio $\rho < 1$. Since ρ is a constant, this is still $O(n)$, although the constant hidden in the Big-Oh notation is larger.

Thus, it would suffice to find an *approximate* median m' of A in linear time, and use m' instead of m to split the array A into L and R. By an approximate median we mean an element m' such that at most $\rho \cdot n$ elements of A are smaller than m' (the set L) and at most $\rho \cdot n$ are larger (the set R), for some constant $\rho < 1$.

6.2 Finding an approximate median

Here is the key insight: Consider breaking up the array A into $\lceil n/w \rceil$ consecutive segments of length at most w, for a constant w to be determined later; think of w as something small-ish, like 7 or 41. Determine for each of these segments their median, and let A' be the array consisting of these $\lceil n/w \rceil$ medians.

Claim 4. *The median of A' is an approximate median of A with $\rho = 3/4$.*

Proof. Let m' denote the median of A'. At least half of the elements x' of A' satisfy $x' \leq m'$. As each of these elements x' are the median of their respective segments, at least half of the elements x in the segment of x' satisfy $x \leq x'$. Since the segments are disjoint, this means that at least a fraction $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$ of the elements x of A satisfy $x \leq m'$. Thus, the set R of elements that exceed m' satisfies $|R| \leq (1 - \frac{1}{4}) \cdot n = \rho \cdot n$ for $\rho = \frac{3}{4}$. A similar argument shows that the set L of elements that are smaller than m' satisfies $|L| \leq \rho \cdot n$.

Figure 7 gives a visual depiction of this proof.

□

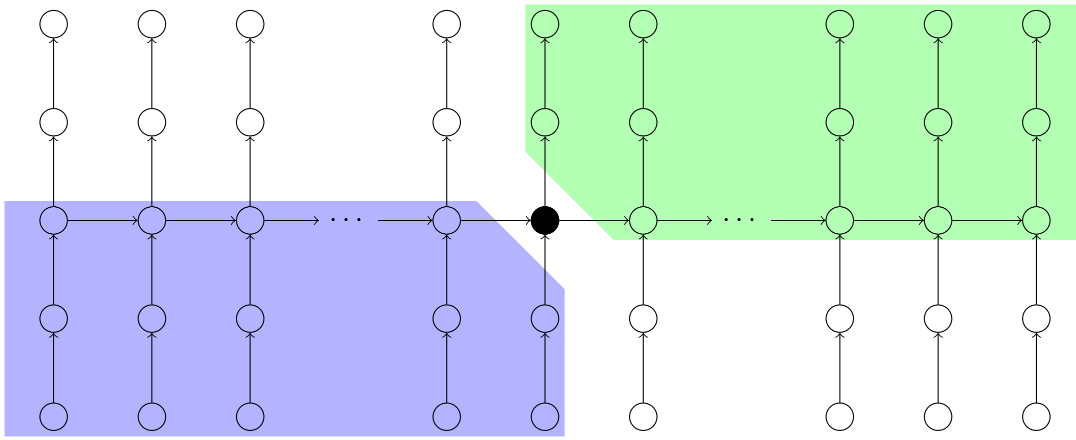


Figure 7: Median of medians is an approximate median

We organize the elements of A so that the columns are the size-w groupings, the nodes within each column are ordered from least to greatest vertically, and the columns themselves are ordered from least median to greatest median. The black dot represents the "median-of-medians", m'. The dots in the blue region are all less than m', and thus are not in R; the dots in the green region are similarly not in L. Each region contains at least 1/4-th of all points, so their complements (corresponding to L and R) contain at most 3/4-ths of all points.

How do we find the median of A'? First, we need to construct A', *i.e.*, , we need to find the median of each segment. As the length of the segments is bounded by the constant w, we can find each individual median in constant time (*e.g.*, by sorting), resulting in $O(n)$ time overall to construct A'. Note that A' is shorter than A, so once we have constructed A', we can find its median by making *another recursive call* to our selection procedure.

Note that this isn't exactly what we set out to do, which was to design a (separate) linear-time procedure to find an approximate median of A. Instead, what we are doing is computing an approximate median of A by computing the exact median of an array A' of smaller size, and do so by making a recursive call to the procedure we're designing. This also means we need to redo the analysis of the overall procedure, which becomes more complicated.

6.3 Final algorithm and analysis

Our final divide-and-conquer algorithm makes *two* recursive calls to the selection procedure, and works as follows on a non-trivial input A:

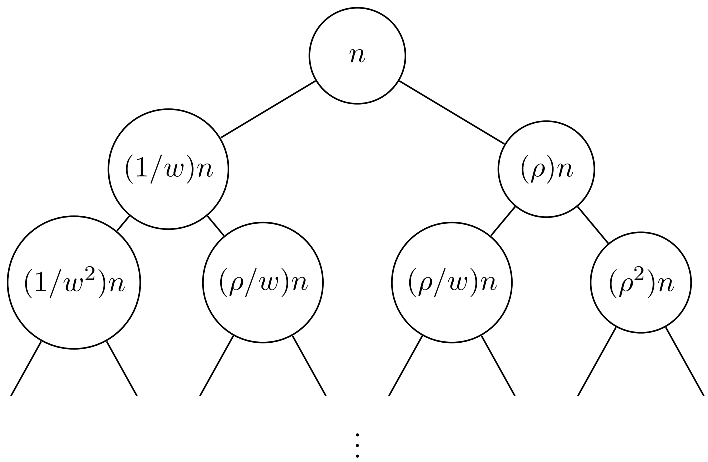
1. First, we construct the array A' and make a recursive call to find the median m' of A'. The size of this recursive call is $|A'| = \lceil n/w \rceil$.

2. Next, we use m' to construct the sets L and R as before, recursively call the selection procedure on either L or R, and return that answer. The size of this recursive call is at most $\rho \cdot n$.

Note how this is different than splitting A into two halves of size $n/2$ and recursing on each of those: Instead, we are recursing on an array of size $\rho \cdot n$ and an array of size $\frac{1}{w} \cdot n$. When ρ and $\frac{1}{w}$ are small enough, then this will lead to a better overall running time than the $\Theta(n \log(n))$ we would get if we recursed on two halves of size $n/2$.

Let's see this in more detail. Figure 8 depicts the recursion tree for our algorithm for the selection problem. Note that here the tree is potentially highly unbalanced, since $1/w$ and ρ may be quite different. However, this won't affect the general structure of our analysis very much.

Figure 8: Recursion tree for our final algorithm for selection



We will start by observing that the local work performed at each node is linear in the size of the node. So now all we have to do is count the total contribution of all the nodes in the recursion tree.

To do this, note that the total size of all the nodes in one level is at most $(1/w + \rho)$ times the total size of all nodes in the level above. This is because, for every node of size s in one level, its children contribute s/w and ρs to the level below it, or else zero if there are no children.

We can use this fact to bound the total contribution of nodes of a fixed depth, as we did before: Let $\alpha \doteq (1/w + \rho)$. Then what we have just argued is that the total size of all nodes at depth $d+1$ is at most α times the total size of all nodes at depth d. Thus the total size of nodes at level d is at most $\alpha^d \cdot n$.

Since the work done at each node in linear, there is a constant c so that the work done at a node of size s is bounded by $c \cdot s$. Thus at the each level d of the recursion tree, at most $c \cdot \alpha^d \cdot n$ work is done.

Thus when we sum up the contribution from every level of the recursion tree, we get the following expression as an upper bound on the total work done:

$$c \cdot \left(1 + \alpha + \alpha^2 + \cdots\right) \cdot n, \tag{10}$$

where the number of terms is $\max(\log_w(n), \log_{1/\rho}(n))$. This is $O(\log(n))$ when we regard w and ρ as fixed constants.

- For $\alpha < 1$, the amount of work reduces by a constant factor at every level, and we can estimate the above sum by the geometric series

  $$c \cdot n \cdot \sum_{k=0}^{\infty} \alpha^k$$

  When $\alpha < 1$, this geometric series converges, and the resulting running time is bounded by $c \cdot \frac{1}{1-\alpha} \cdot n = O(n)$.

- For $\alpha = 1$, our bound on the amount of work per level of recursion is $c \cdot n$ at every level, resulting in a bound of $O(n \log n)$ on the running time, since the recursion tree has depth $O(\log(n))$.

- For $\alpha > 1$, the amount of work per level grows by a constant factor (up until the point where some branches die out, which happens after a logarithmic number of levels), and the total amount of work is dominated by the lower levels. The resulting upper bound on the running time is worse than $O(n \log n)$; the exact bound depends on w and ρ.

Note that with our intended choice of $\rho = 3/4$, $\alpha < 1$ if and only if $w > 4$. Thus, if we pick $w = 5$, the resulting selection algorithm runs in linear time!

Larger values of w result in smaller values of α, but larger values of c in Equation (10); the latter is due to the extra work needed to construct A'. The optimal trade-off depends on the implementation details and system parameters.