CMPT 307 : Divide-and-Conqer (Study Guide)
Should be read in conjunction with the text
September 23, 2019

1 Introduction

The divide-and-conquer strategy is a general paradigm for algorithm design. This strategy has three processes.

1. **Divide** the problem into smaller subproblems.
2. **Conquer** by solving these problems.
3. **Combine** these results together.

This technique is used to solve many problems such as binary search, integer multiplication, matrix multiplication, merge sort, quick sort, Fast Fourier Transform (FFT) etc.

2 Binary Search

This problem is well known. The algorithm can be informally described as follows:

```
-----------------------------------------------------------------
Binary-Search(A[1..n],a,b,x)

Input: A sorted array of elements; A[a..b] is being searched for x.
Output: Returns the index r where A[r]=x;
        If there is no such index, -1 is returned instead;

if a > b then return -1;
r = floor((a+b)/2); /* middle index of the array*/
if x = A[r] then
    return r
else
    if x < A[r] then
        Binary-Search(A, a, r, x)
```

```
    else
        Binary-Search(A, r+1, b, x)
    endif
endif
--------------------------------------------------------------------
```

- The initial call is Binary-Search $(A, 1, n, x)$.
- Let $T(n)$ denote the worst-case time to binary search in an array of length n.
- **Basis:** $T(1) = O(1)$.
- **Recurrence:** $T(n) = T(\lfloor \frac{n}{2} \rfloor) + O(1)$.
- The solution is $T(n) \in O(\log n)$.

3 Multiplying Numbers

Using the school method the cost of multiplying two n-bit integers is $O(n^2)$. When n is large, typically 256 to 1024 bits long, $O(n^2)$ multiplication is too slow. We will see that, by applying a divide-and-conquer strategy, we can multiply using only $O(n^{\log_2 3})$ steps.

As a first step toward multiplying x and y, split each of them into their left and right halves, which are about $\frac{n}{2}$ bits long. I am assuming here that n is a perfect power of 2, i.e. $n = 2^k$, for some k. This assumption will guarantee an equal size division every time.

$$x = \boxed{x_L}\ \boxed{x_R} = 2^{n/2}x_L + x_R$$
$$y = \boxed{y_L}\ \boxed{y_R} = 2^{n/2}y_L + y_R.$$

The product of x and y can then be written as

$$xy = (2^{n/2}x_L + x_R)(2^{n/2}y_L + Y_R) = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R.$$

The straight forward divide-and-conquer algorithm to compute xy is:

```
--------------------------------------------------------------------------------
MultiplyNumbers(x, y)
Input: Two integers with the same number of bits
Output: The product xy

if the number of bits used to represent x and y is small (say c) then return xy;
Determine x(L) and x(R) from x;
Determine y(L) and y(R) from y;
a = Multiply(x(L), y(L));
b = Multiply(x(L), y(R));
c = Multiply(x(R), y(R));
d = Multiply(x(R), y(L));
return 2^n a + 2^(n/2)(b+c) + d; (why?)
--------------------------------------------------------------------------------
```

We can write the recurrence as

$$T(n) = 4T(n/2) + O(n),\ n \geq 2;\ T(1) = O(1).$$

The above recurrence solves to $T(n) \in O(n^2)$, no improvement!

3.1 An Example

Instead of binary bits, we use decimal digits. The modifications are straightforward.

- Let $X = 4729$ and $Y = 1326$. We split X and Y as $X_L = 47$; $X_R = 29$; $Y_L = 13$; $Y_R = 26$.
- Now $X_L * Y_L = 47 * 13 = 611$; $X_L * Y_R = 47 * 26 = 1222$; $X_R * Y_L = 29 * 13 = 377$; $X_R * Y_R = 29 * 26 = 754$.
- $XY = 10^4 * 611 + 10^2 * (1222 + 377) + 754 = 6270654$.

3.2 An improved algorithm

The previous divide-and-conquer algorithm is a radically different algorithm which can be improved by observing that

$$(x_L + x_R) \times (y_L + y_R) = x_L y_L + x_R y_R + x_L y_R + x_R y_L.$$

Therefore,

$$xy = 2^n x_L y_L + 2^{n/2}(x_L y_R + x_R y_L) + x_R y_R = 2^n x_L y_L + x_R y_R + 2^{n/2}((x_L + x_R) \times (y_L + y_R) - x_L y_L - x_R y_R).$$

The modified divide-and-conquer algorithm to compute xy is:

```
--------------------------------------------------------------------------------
MultiplyNumbers-Modified(x, y)
Input: Two integers with the same number of bits
Output: The product xy

if the number of bits used to represent x and y is small (say t) then return xy;
Determine x(L) and x(R) from x;
Determine y(L) and y(R) from y;
a = Multiply(x(L), y(L));
b = Multiply((x(L) + y(L)) * (x(R) + y(R)));
c = Multiply(x(R), y(R));
return 2^n a + 2^(n/2)(b - a - c) +c;
--------------------------------------------------------------------------------
```

Now we can write the recurrence as

$$T(n) = 3T(n/2) + O(n),\ n \geq 2;\ T(1) = O(1).$$

The above recurrence solves to $T(n) \in O(n^{\log_2 3})$, a great improvement.

4 Merge sort

Merge sort is a divide-and-conquer sorting algorithm. We are given an array of n unsorted elements. The algorithm can be be written as follows (from the text)

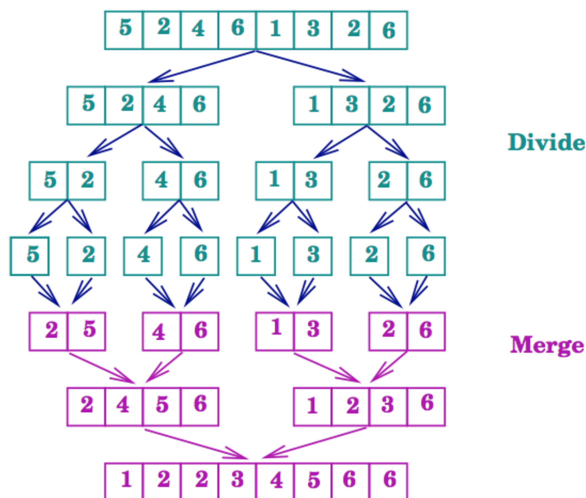
```
function mergesort(a[1...n])
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋ + 1...n]))
else:
  return a
```

The following **merge** function merges two sorted arrays $x[1...k]$ and $y[1...l]$. The process is very efficient. The very first element of the merged list is either $x[1]$ or $y[1]$ whichever is smaller. The rest of the merged list can be constructed recursively.

```
function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
  return x[1] ∘ merge(x[2...k], y[1...l])
else:
  return y[1] ∘ merge(x[1...k], y[2...l])
```

The following example illustrates the algorithm (taken from Suri's notes).



The routine **merge** takes linear time in total. Every time the function merge is called, the total size of the two lists is reduced by one. The body of the function takes constant time to execute.

4.1 Analysis

We can write the recurrence as

$$T(n) = 2T(n/2) + O(n),\ n \geq 2;\ T(1) = O(1).$$

The above recurrence solves to $T(n) \in O(n \log n)$.

5 Solving Recurrence Relations

Consider a trivial example:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + 1,\ \ n \geq 2. \end{aligned}$$

The above recurrence describes the sequence $1, 2, 3 \ldots$. The nth term, i.e. $T(n)$ is n. $T(n) = n$ is the closed form of the nth term.

We use recurrences to analyze the performance of recursive algorithms. We will first describe a method called guess-and-verify. Given a recurrence relation, we will first guess the closed form, and then prove that our guess is correct. At the end we will develop rules of thumb which will allow us to solve many recurrences without any calculation.

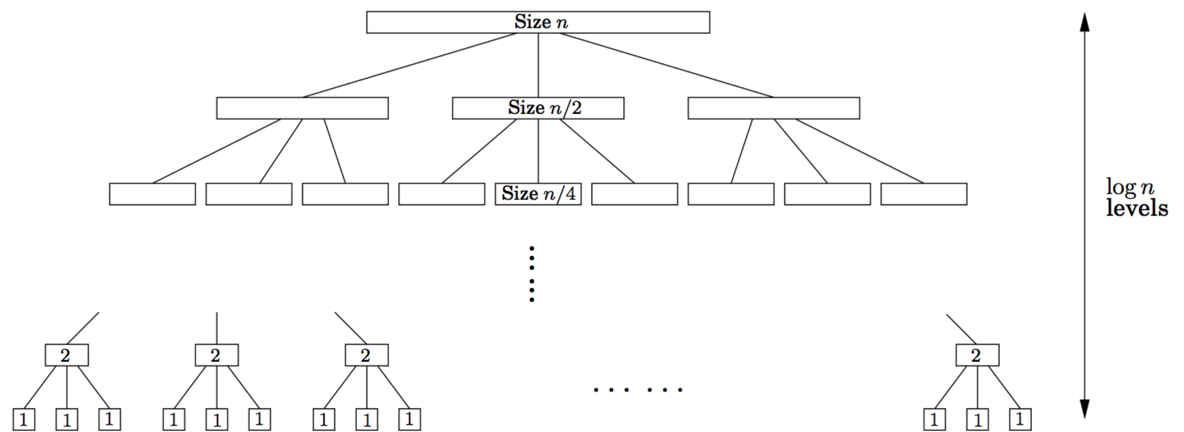
Let us describe our approach to solve the recurrence relation obtained from our example of integer multiplication. The recurrence relation is

$$\begin{aligned} T(t) &= O(t) \\ T(n) &= 3T(n/2) + O(n),\ \ n > t. \end{aligned}$$

In order to keep our focus to the major issues, we will make the following assumptions. Later we will see that the asymptotic order remains unchanged when these assumptions are not valid. The assumptions (for recurrences of the above type) we make are

1. In the divide-and-conquer setting, n is always assumed to be 2^k for some integer k.
2. We will assume the basis to be $T(1) = 1$.
3. The general recurrence is assumed to be $T(n) = 3T(n/2) + n$.

For a given recursive algorithm we consider the recursion tree. The recursion tree of the modified multiplication algorithm can be viewed as follows (taken from the text)



We note the following from the recursion tree.

1. The root node represents the multiplication of two n-bit integers. The leaf nodes represent multiplication of two 1-bit integers.

2. The size of each node is one half of the size of its parent.

3. The root node is a zero-level node. All leaf nodes are at level $k = \log n$, where the recursion ends. There are $k+1$ levels in total.

4. The branching factor is 3.

5. There are $3^k = 3^{\log n} = n^{\log_2 3}$ leaf nodes in the tree.

6. The number of nodes (subproblems) at level i is 3^i. Each node (subproblem) at level i represents a multiplication of two $\frac{n}{2^i}$-bit integers.

7. The solution to a subproblem at level i is obtained by merging the solutions to three subproblems at level $i+1$. The cost of the merging of a level i subproblem is $\frac{n}{2^i}$ steps (from the recurrence relation). The total merging cost at level i is, therefore, 3^i times $\frac{n}{2^i}$.

8. The total merging cost of all the subproblems is

$$\sum_{i=0}^{i=k} 3^i \frac{n}{2^i} = n \sum_{i=0}^{i=k} (\frac{3}{2})^i = n \times \frac{(\frac{3}{2})^{k+1} - 1}{\frac{3}{2} - 1}.$$

9. Total number of nodes in the recursion tree is $\sum_{i=0}^{k} 3^i$ which is $\frac{3^{k+1}-1}{3-1}$, i.e. $O(3^{\log_2 n})$. This cost is sometimes called **splitting cost**, i.e. time to split up the problems. Note here that $O(3^{\log_2 n})$ is also a bound on the number of leaf nodes. This is a property associated with geometric series (see problem 0.2 of the text)

The solution to the recurrence relation

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/2)+n, \;\; n>1. \end{aligned}$$

is $O(n(\frac{3}{2})^k)$ which is $O(n^{\log_2 3})$. Note that $k = \log_2 n$.

Relaxing the constraints

We have imposed three restrictions on the recurrence relation. We will relax these one by one.

A. Replace T(n) = 3T(n/2) + n by T(n) = 3T(n/2) + cn, c a positive constant

In this case, the cost of the merging of a level i subproblem is $c\frac{n}{2^i}$ steps (from the recurrence relation). The total merging cost at level i is, therefore, $3^i \times c\frac{n}{2^i}$. Thus the total merging cost is

$$\sum_{i=0}^{i=k} c \times 3^i \frac{n}{2^i} = cn \sum_{i=0}^{i=k} (\frac{3}{2})^i = cn \times \frac{(\frac{3}{2})^{k+1}-1}{\frac{3}{2}-1}.$$

The total cost is still $O(n(\frac{3}{2})^k)$.

Therefore, the asymptotic complexity of $T(1) = 1$, and $T(n) = 3T(n/2) + cn$, $n > 1$ is the same as that of $T(1) = 1$, and $T(n) = 3T(n/2) + n$, $n > 1$.

We are interested in finding the upper bound cost of $T(n)$. Therefore, the recurrence relation of the form $T(1) = 1$, and $T(n) = 3T(n/2) + O(n)$, $n > 1$ and $T(1) = 1$, and $T(n) \leq 3T(n/2) + cn$ have the same asymptotic complexity.

B. Replace $T(1) = 1$ by $T(n) = O(t)$ for any $n \leq s$ where s, t are constants

Suppose $s = 2^g$ for some constant g. In this case the last g levels of the recursion tree may have increased costs which is at most $O(t)$, a constant, times the previous cost. Therefore, the asymptotic upper bound of the original simplified recurrence relation remains unchanged by the change in the basis equation. The asymptotic solution to a divide-and-conquer recurrence is independent of boundary conditions.

C. Replace $n = 2^k$ by any n, $2^{k-1} < n \leq 2^k$

Such k always exists for any n (problem 2.2 of the text).

In this case $k = \lceil \log_2 n \rceil$, and $T(n) \leq T(2^k)$ since $T(n)$ is an increasing function of n. We know that $T(2^k) \in O(n(\frac{3}{2})^k)$. Therefore $T(n) \in O(n(\frac{3}{2})^{\lceil \log_2 n \rceil})$.

Now note that $\lceil \log_2 n \rceil < 1 + \log_2 n$. Therefore, $T(n) \in O(n(\frac{3}{2})^{1+\log_2 n})$ which is equivalent to $T(n) \in O(3^{\log_2 n})$.

In our approach to solving a recurrence relation, we will first clean the recurrence making sure that the asymptotic order is not disturbed. Once it is clean, we look at the resulting recursion tree, and estimate the running time of the algorithm.

6 Master Theorem

Master Theorem handles some of the recurrences that commonly arise in computer science. We consider solving the following generic recurrence relation.

$$\begin{aligned} T(n) &= O(1), \;\; n < b \\ T(n) &= aT(n/b) + O(n^d), \;\; n \geq b. \end{aligned}$$

Here $a(\geq 1, integer), b(\geq 1, integer), d(\geq 0, real)$. Observe that

- $a = 3, b = 2, d = 1$ settings realize the improved integer multiplication recurrence relation.
- $a = 4, b = 2, d = 1$ settings realize the straightforward integer multiplication recurrence relation.
- $a = 2, b = 2, d = 1$ settings realize the merge sort recurrence relation.
- $a = 1, b = 2, d = 0$ settings realize the binary search recurrence relation.

Assumptions: First, we make the following assumptions to simplify our things.

- n is a perfect power of b, i.e. $n = b^k$ where $k = \log_b n$.
- Our new basis is $T(n) = 1, \;\; n < b$.
- Our new recurrence relation is $T(n) = aT(n/b) + n^d, \;\; n \geq b$.

Analysis:
Let us consider the recursion tree of the above recurrence relation. It is shown in Fig. 2.3 of the text (page 50). From the tree we can conclude that (please make sure that you understand this)

1. The height of the tree is $\log_b n$, say k.
2. The branching factor is a.
3. The number of nodes (subproblems) at level i is a^i.
4. The total number of subproblems created is $1 + a + a^2 + \ldots + a^{\log_b n}$, which is $O(a^{\log_b n})$. This is nothing but the splitting cost. Since $a^{\log_b n} = n^{\log_b a}$ (you need to convince yourself), the splitting cost is $O(n^{\log_b a})$.
5. The size of each subproblem at level i is $\frac{n}{b^i}$.
6. The cost of merging for a subproblem at level i is $(\frac{n}{b^i})^d$.
7. The total of the merging costs of all the subproblems at level i is $a^i \times (\frac{n}{b^i})^d$ which is $n^d(\frac{a}{b^d})^i$.
8. The total merging cost is $\sum_{i=0}^{\log_b n} n^d(\frac{a}{b^d})^i$. This is a geometric series:
$$n^d(1 + (\frac{a}{b^d}) + (\frac{a}{b^d})^2 + \ldots + (\frac{a}{b^d})^{\log_b n}).$$
The total merging cost has the following asymptotic bound depending on the value of $\frac{a}{b^d}$. The properties of geometric series playing a role here (exercise 0.2 of the text).
   - When $\frac{a}{b^d} < 1$, then the merging cost is $O(n^d)$.
   - When $\frac{a}{b^d} > 1$, then the merging costs is $O((\frac{a}{b^d})^{\log_b n})$ which is $O(n^{\log_b n})$. This is nothing but the splitting cost.

- When $\frac{a}{b^d} = 1$, the merging cost is $O(n^d \log_b a)$

The above bounds are also true when we consider the general recurrence relation ($a(\geq 1,\ integer), b(\geq 1,\ integer), d(\geq 0, real)$).

$$\begin{array}{rcl} T(n) & = & O(1),\ \ n < b \\ T(n) & = & aT(n/b) + O(n^d),\ \ n \geq b. \end{array}$$

Therefore, the closed form (this is what we are looking for) of Master Theorem is:

$$T(n) = \begin{cases} O(n^d) & \text{if } \frac{a}{b^d} < 1 \ \ i.e.\ \ d > \log_b a \\ O(n^d \log_b n) & \text{if } \frac{a}{b^d} = 1 \ \ i.e.\ \ d = \log_b a \\ O(n^{\log_b a}) & \text{if } \frac{a}{b^d} > 1 \ \ i.e.\ \ d > \log_b a \end{cases}$$

Master Theorem will allow us to tackle almost all the commonly used recurrence relations. If you have any problem fitting your recurrence relation to take advantage of Master Theorem, you can always use the recursion tree to determine the closed form of the recurrence relation.

Note that Master Theorem cannot be directly applied to the following recurrence relation:

$$\begin{array}{rcl} T(n) & = & O(1),\ \ n < 2 \\ T(n) & = & 2T(n/2) + O(n^2 \log n),\ \ n \geq 2. \end{array}$$

Here $a = b = 2$. What can we say about the growth rate of $n^2 \log n$? We can say that it is $\Omega(n^2)$ and $O(n^{2+\varepsilon})$, for any $\varepsilon > 0$ (you need to convince yourself). In order to deal with the recurrence relation of this type, we first find α such that $\frac{a}{b^\alpha} = 1$. Note that for $a = b = 2$, $\alpha = 1$. Now α is less than 2 (and hence $2 + \varepsilon$, for any ε), the merging cost dominates the splitting cost. Therefore, $T(n) \in O(n^2 \log n)$ is the solution to the recurrence relation:

$$\begin{array}{rcl} T(n) & = & O(1),\ \ n < 2 \\ T(n) & = & 2T(n/2) + O(n^2 \log n),\ \ n \geq 2. \end{array}$$

Observe the difference when the recurrence relation is

$$\begin{array}{rcl} T(n) & = & O(1),\ \ n < 2 \\ T(n) & = & 2T(n/2) + O(n^{0.5} \log n),\ \ n \geq 2. \end{array}$$

In this case we notice that $\alpha = 1$ is greater than $0.5+\varepsilon$, the splitting cost dominates the running time. Therefore, $T(n) \in O(n)$.

What happens when the recurrence relation is

$$\begin{aligned} T(n) &= O(1),\ \ n < 2 \\ T(n) &= 2T(n/2) + O(n\log n),\ \ n \geq 2? \end{aligned}$$

In this case $\alpha = 1$ is not less than 1 and more than $1+\varepsilon$. In this case the merging cost at every level of the recursion tree have the same asymptotic rate. We therefore claim that $T(n) \in O(n\log^2 n)$ (show this using the recursion tree).

6.1 Cook-book approach to apply Master Theorem

The cook-book approach to solve the recurrence relation of the form

$$\begin{aligned} T(n) &= O(1),\ \ n < b \\ T(n) &= aT(n/b) + f(n),\ \ n \geq b. \end{aligned}$$

is as follows:

Step 1: Find the largest d such that $f(n) \in \Omega(n^d)$ and $f(n) \in O(n^{d+\varepsilon})$, for any $\varepsilon > 0$.

Step 2: Compute α such that $\frac{a}{b^\alpha} = 1$.

Step 3: Apply the following rule:

- If $\alpha < d$, return $T(n) \in O(f(n))$. (Merging cost dominates)
- If $\alpha > d+\varepsilon$, return $T(n) \in O(n^{\log_b a})$. (Splitting cost dominates)
- If $\alpha = d$, return $T(n) \in O(f(n)\log n)$. (The base of log is missing. Why?) (Both the costs are the same)

6.2 Examples

Exercise 2.1 Let $x = 10011011$ and $y == 10111010$. We first divide x and y into two equal parts each: $x_L = 1001$, $x_R = 1011$ and $y_L = 1011$, $y_L = 1010$. Recursively, compute $x_L y_L, x_R y_R, x_L y_R, x_R y_L$.

Exercise 2.5 of the text: Note that Master Theorem results are also valid when Big-theta (Θ) replaces Big-oh (O) (why?).

Problems (a) through (e) fit Master Theorem formulation exactly. For (f), $a = 49, b = 25$. We note that $\frac{49}{25^\alpha} = 1$ implies $\alpha = 1.2096....$ We also note that $n^2 \log n \in \Omega(n^{\alpha+\varepsilon})$ for some $\varepsilon > 0$. This implies that the merging cost lies to the right of n^α on a scale representing running time, and therefore, the merging cost dominates the splitting cost. Hence, $T(n) \in O(n^{3/2} \log n)$.

Problems (g) through (j) don't fit Master Theorem. You should be able to find the solution to the recurrences using the recurrence tree.

For problem (k) assume $n = 2^{2^k}$.

Exercise 2.12 If $T(n)$ represents the number of times the line is printed, $T(n) = 2T(n/2) + 1$. Master Theorem can now be applied.

Exercise 2.13 Note that $B_1 = 0; B_3 = 1; B_5 = 2$. Now $B_7 = B_1 * B_5 + B_3 * B_3 + B_5 * B_1 = 5$. Can you now generalize?

Exercise 2.19 Let $T(i)$ denote the time to merge first i lists. Therefore, $T(i) = T(i-1) + i.n$. Show by induction that $T(k) \in O(nk^2)$. This another way to solve a recurrence relation if we can guess correctly. By the Master Theorem $T(n) \in O(nk \log k)$.

The divide and conquer algorithm will realize the recurrence relation $T(k) = 2T(k/2) + n.k$.

Exercise 2.25 (a) Note that $10^n = (10^{n/e})^2$. $z = pwr2bin(n/2)$. The recurrence relation is $T(n) = 2T(n/2) + O(n^a)$.

(b) **return fastmultiply(dec2bin(x_L),pwr2bin(n/2)) + dec2bin(x_R)** (why?) The recurrence relation is again $T(n) = 2T(n/2) + O(n^a)$.

Exercise 2.29 We have seen these two parts before. The first part involves O(n) multiplications. The second part needs O(logn) multiplications.

7 Comparison-based lower bounds for sorting

So far our objective is: given some problem P of size n, can we construct an algorithm that runs in $O(f(n))$ time? This often called an **upper bound** problem because we are determining a complexity within which the problem is solve. In

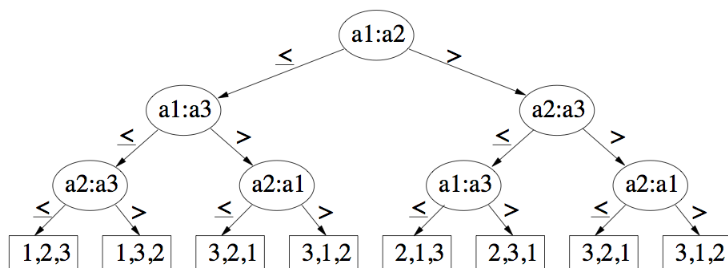


Figure 1: Decision tree of selection sort

this setting, our goal is to make $f(n)$ as small as possible. Now our objective here is to establish a bound $g(n)$ for P, called **lower bound**, below which P cannot be solved. In this setting, our goal is to make $g(n)$ as large as possible. We will study the sorting problem for the lower bound analysis.

We only consider the sorting algorithms which are comparison-based. Merge sort, quick sort, heap sort are comparison-based sorting algorithms, where as bucket sort, radix sort are not comparison-based sorting algorithms. A formal definition of comparison-based sorting algorithm is as follows.

Definition: A comparison-based sorting algorithm takes as input an array $[a_1, a_2, \ldots, a_n]$ of n elements, and can only gain information about the items by comparing pairs of them. Each comparison (" is $a_i < a_j$") returns YES or NO and takes one time-step. In the end, the algorithm outputs a permutation of the input in which all the items are in sorted order.

A comparison-based sorting algorithm can be depicted as trees.

Theorem: Any comparison-based sorting algorithm must perform $\Omega(nlog_2 n)$ comparisons in the worst case to sort n elements.

Proof: Consider any comparison-based sorting algorithm. The algorithm can be described by a decision tree. Each of its leaves is labeled by a permutation of $\{1, 2, \ldots, n\}$, and every permutation must appear as the label of a leaf. Thus the decision tree of any algorithm must have at least $n!$ leaves.

The decision tree of any sorting-based algorithm is a binary tree with at least $n!$ leaves. We know that a binary tree with depth d has at most 2^d leaves. Thus a binary tree with $n!$ leaves has depth at least $\log_2 n!$. We know that $c_1 n \log n \leq \log(n!) \leq c_2 n \log n$ for some positive constant c_1 and c_2, and for all $n \geq n_0$. Therefore, any comparison-based sorting algorithm must make $\Omega(n \log n)$ comparisons

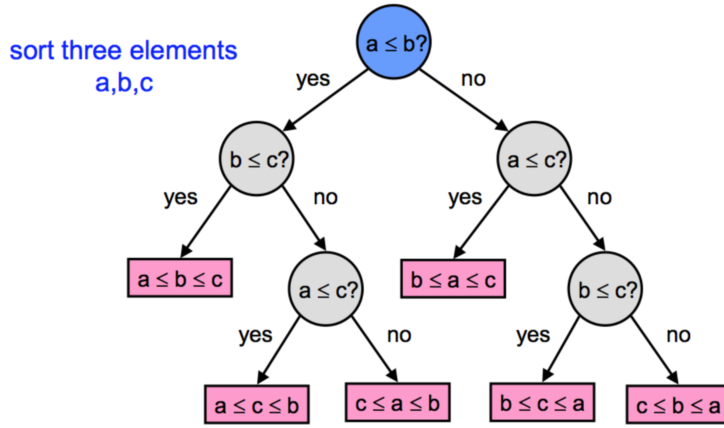


Figure 2: Decision tree of insertion sort

in the worst case.

8 Randomized divide-and-conquer

We have seen how divide-and-conquer (D&C) technique facilitates efficient algorithmic design. Randomized algorithms are algorithms that use random numbers (a library function that returns the result of a random coin flip) to make decisions during the executions of the algorithm. The running time becomes a random variable. D&C often works well in conjunction with randomization. We will illustrate this fact by looking at the problem of finding the median of a list of elements.

8.1 Finding the median

We are given a set of numbers $S = \{a_1, a_2, \dots, a_n\}$. We are required to find the kth smallest element in S. This problem is called For example:

- $k = 1$: find the smallest element
- $k = n$: find the largest element
- $k = \frac{n}{2}$: find the median element
- Popular statistics are quantiles: items of rank $\frac{n}{4}$, $\frac{n}{2}$, $\frac{3n}{4}$, etc.

- SAT scores of 95 percentile? Item of rank $0.95n$.

This problem is called the selection problem.

Consider the following divide-and-conquer algorithm for the selection problem.

```
Selection(S,k)

If |S| = 1 then return S[1]

Choose a splitter element x of S
For each element y of S
    Put y in S- if y < x
    Put y in S+ if y > x
Endfor

If |S-| = k - 1 then
    The splitter x was in fact the desired answer
Else if |S-| >= k then
    The kth smallest element lies in S-
    Recursively call Selection(S-,k)
Else suppose |S-| = m < k - 1
    The kth smallest element lies in S+
    Recursively call Selection(S+, k - 1 - m)
Endif
```

Note that the algorithm is always called recursively on a strictly smaller set, and therefore, it must terminate.

Choosing a good splitter

Suppose we select a splitter in linear time. The running time of Selection is affected by the splitter. If we choose the splitter to be the median element of S, the recurrence relation $T(n) = T(n/2) + O(n)$ realizes $T(n) \in O(n)$, the best one can do. If we choose always the minimum element as the splitter, we end up with a recurrence relation $T(n) = T(n-1) + O(n)$ whose solution is $T(n) \in O(n^2)$ (why?).

If we had a way to choose splitters x such that there were at least εn elements both larger and smaller than x, for any fixed constant $\varepsilon > 0$, the size of the set $(S - /S+)$ would shrink by a factor of at least $(1 - \varepsilon)$ each time. In this case,

the recurrence relation is $T(n) = T((1-\varepsilon)n) + O(n)$. Here we can apply Master Theorem where $a = 1, b = \frac{1}{1-\varepsilon}, d = 1$. The result is $T(n) \in O(n)$. Thus any splitter which is not very ”off-center” (unlucky) splitter should be good. The good splitters are recognized as “well-centered splitters” (lucky splitters). Median splitter is clearly well-centered splitter.

We will use **random splitters**. Our choice is

Choose a splitter $x \in S$ uniformly at random

The intuition is very natural: since a fairly large fraction of elements (within the 25th and 75th percentile) are reasonably well-centered, we will be likely to end up with a good splitter simply by choosing an element at random. We note that a randomly chosen x has a 50% chance of being good. Thus

Lemma: On average a fair coin needs to be tossed two times before a “heads” is seen. (from the text)

Example. How many heads would you expect if you flipped a coin twice?

Let X= number of heads = $\{0,1,2\}$

$Pr(0) = 1/4, Pr(1) = 1/2, Pr(2) = 1/4$

Weighted average $= 0*1/4+1*1/2+2*1/4 = 1$

Therefore, after two split operations on average, the array will shrink to at most three-fourths of its size. Letting $T(n)$ be the expected running time on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n).$$

The second term can be read as the time to reduce the size of the array to $\leq 3n/4$.

8.2 Quicksort

I have used the notes available at
https://www.cs.duke.edu/ reif/courses/alglectures/skiena.lectures/lecture5.pdf
Please refer the book by CLRS (Corman, Lieserson, Rivest and Styn) for more information.

Although Megesort algorithm is $O(n \log n)$, in practice, Quicksort algorithm is the fastest. Quicksort algorithm uses partitioning as its main idea.

Example Pivot about 10 (as in the selection problem):

17	12	6	19	23	8	5	10	—	before
6	8	5	10	23	19	12	12	—	after

This partitioning process can be done in linear time (you must convince yourself). Note that after the partitioning, the left and the right lists can be sorted independently. The psueodocode of the algorithm is as follows.

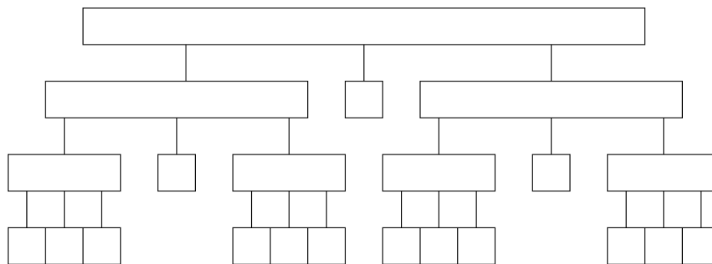
Pseudocode

```
Sort(A)
      Quicksort(A,1,n)


Quicksort(A, low, high)
      if (low < high)
            pivot-location = Partition(A,low,high)
            Quicksort(A,low, pivot-location - 1)
            Quicksort(A, pivot-location+1, high)


Partition(A,low,high)
      pivot = A[low]
      leftwall = low
      for i = low+1 to high
            if (A[i] < pivot) then
                  leftwall = leftwall+1
                  swap(A[i],A[leftwall])
      swap(A[low],A[leftwall])
```

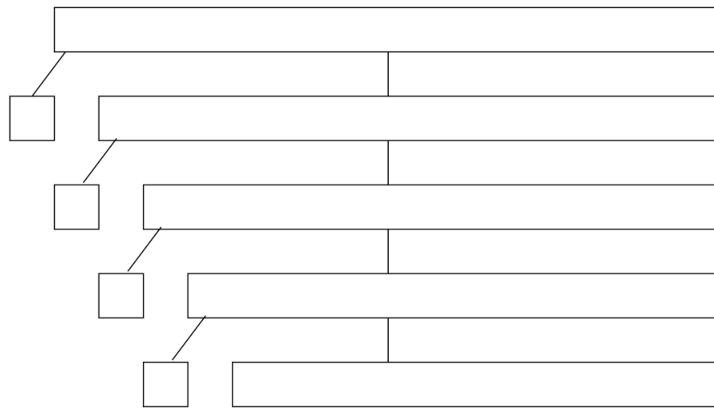
The best case for divide-and-conquer comes when we split the input as evenly as possible. The recursion tree for Quicksort for the best case looks like this:



For this recursion tree, the recurrence relation is $T(n) = 2T(n/2) + O(n)$

which results in an $O(n \log n)$ running time. Note here that $O(n)$ here is the cost of the partition process (in the case of merge sort, O(n) represents the cost of the merging process.)

The worst case scenario is when the splitting in the recursion tree is completely uneven which looks like:



For this recurrence tree, the recurrence relation is $T(n) = T(c) + T(n-c) + O(n)$ where c is a small constant. The running time in this case $O(n^2)$.

We thus need a good pivot to split the list.

As discussed for the selection problem, we choose a pivot from the array uniformly at random. This will result in partitioning the list, after two pivot operations on an average, into two lists where the size of the smallest list is at least $n/4$. The recurrence relation for this case is then $T(n) \leq T(n/4) + T(3n/4) + O(n)$ where $T(n)$ is the expected running time. The solution to this recurrence relation is $O(n \log n)$. Master Theorem does not directly apply. We can modify Master Theorem to accommodate recurrence relation of the above type. Here is how it is done for the relation $T(n) \leq T(n/4) + T(3n/4) + O(n)$.

1. Solve $(\frac{1}{4})^\alpha + (\frac{3}{4})^\alpha = 1$ for α. We note that $\alpha = 1$ satisfies the equation. We can show , just like before, that when the merging time is n^α, the splitting cost and the merging cost are the same.

2. Since the merging (partitioning in this case) cost is the same as n^α, $T(n) \in \Theta(n \log_2 n)$.

8.3 Effects of Randomization

Randomization is a very important and useful idea. By picking a random event or scrambling the input array before sorting it we can say:

"With high probability, the randomized Quicksort runs in $\Theta(n \log n)$ time."

instead of saying:

"If you give me random input data, Quicksort runs in expected $\Theta(n \log n)$ time."

Randomization is a general tool to improve algorithms with bad worst-case but good average-case. The worst case is still there, but we almost certainly won't see it.

9 Matrix Multiplication

The straightforward way of multiplying two $n \times n$ matrices X and Y looks like

```
Multiply(X,Y,n)

Define an n x n matrix C.
if n = 1 then return XY
for i = 1 to n
    for j = 1 to n
        c(i,j)=0
        for k = 1 to n
            c(i,j) = c(i,j) + X(i,k) * Y(k,j)

Return C
```

This multiplication costs $O(n^3)$ (why?).

We can apply the divide-and-conquer approach and after some fancy arguments, the text book has shown that the matrix multiplication can be done in $O(n^{\log_2 7})$ time. The treatment of this in the text is simple.