Review

This set of notes provides a quick review about what should have been learned in the pre-requisite courses. The review is helpful to those who have come from a different background; or to those who have forgotten a lot of these topics. Even for those who already know the stuff, we will try to use new examples whenever possible.

We will give examples for each of the following commonly used algorithm design techniques:

- Recursion
- Divide and conquer
- Invent (or augment) a data structure
- Greedy algorithm
- Dynamic programming

We will not review undecidability and intractability. They will be dealt with later in this course.

Recursion

Recursion means an algorithm (or a function) is defined based on itself on a smaller-sized input. For example the Fibonacci number series: $F(n) = F(n-1) + F(n-2)$. An important component of a recursion is the base case: $F(1) = F(2) = 1$.

Note that a careless recursion may be disastrous. For example, if you calculate Fibonacci number with recursion you'll get an exponential time complexity of $O(1.618^n)$. But a careful recursion can do much good for us.

Top-2

Problem: Given an array A with n elements. Find the largest two elements.

Algorithm Trivial: Scan the array twice for the largest and second largest.

Complexity in terms of number of comparisons: $2n$.

Note that some comparison can be expensive (such as comparing strings). So it is meaningful to measure the complexity with number of comparisons. Can we do better?

Algorithm TOP2:

Input: Array A.

Output: A pair of indices (i_1, i_2) of the largest two elements in A, respectively.

1. If $|A| \leq 4$ use a trivial algorithm and return the solution.
2. Create a new half size array $B[i] = \max(A[2i-1], A[2i])$ for $i = 1, \dots, \frac{n}{2}$.
3. $(j_1, j_2) \leftarrow \text{TOP2}(B)$.
4. Return the top two of $A[2j_1 - 1]$, $A[2j_1]$, $A[2j_2 - 1]$, and $A[2j_2]$.

Suppose the number of comparisons of this algorithm is $T(n)$ for this algorithm on a size-n array. Step 2 requires $\frac{n}{2}$ comparisons. Step 3 requires $T\left(\frac{n}{2}\right)$. Step 4 requires constants (actually only one comparison is absolutely necessary). Thus,

$$T(n) = \frac{n}{2} + T\left(\frac{n}{2}\right) + 1 = \frac{n}{2} + \frac{n}{4} + \cdots + 1 + \log_2 n = n + \log_2 n$$

This is better than a trivial solution.

Many Liars

Now imagine you are facing N people who are either honest or liars; and you want to figure out who are honest. What you know is that an honest person always tells the truth and a liar lies as he wishes (he may choose to tell the truth or the opposite of the truth at different times). Luckily more than half of the people are honest. Everybody knows each other's attribute. But you don't. What you are allowed to do is to ask the following question to any pair of people "Is that guy a liar?" and they will always answer.

How many pairs of people should you ask in order to find out at least one honest person?

Three sets of answers may come up when you ask a pair of persons:

1. {liar, liar}
2. {liar, honest}
3. {honest, honest}

For both 1 and 2, we know there must be at least one liar in the pair. So we can eliminate both from the list to get a smaller problem. (Recursion!!!) Remember that we have more than half honest people.

But we may end up with 3, where both are liars or both are honest. We actually end up with a list of honest pairs and liar pairs. And we'll have more honest pairs than liar pairs. So, we simply take one person from each pair and get a half-sized problem. (Recursion again!!!)

$$T(n) \leq \frac{n}{2} + T\left(\frac{n}{2}\right).$$

So we need to ask at most N pairs to find out one honest person.

Pseudo code is omitted here.

Detail: what if N is odd (or becomes odd during recursion)?

Recursion is good by itself. But more importantly, recursion serves as the basis of many other algorithm design techniques. One such example is divide and conquer.

Divide and Conquer

The most straightforward example for divide and conquer is the merge-sort algorithm. Given a list of elements that you want to sort, how many comparisons are needed to sort them?

Merge-sort(A): A is a list of n elements.

1. If $n \leq 3$ sort A with a trivial algorithm and return.
2. Merge-sort(A[1..n/2])
3. Merge-sort(A[n/2+1..n])
4. A ← Merge(A[1..n/2], A[n/2+1..n]).

Here I omit the pseudo code for merging two sorted lists. The merging costs only n comparisons. So, the number of comparisons needed for merge-sort is

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n = 4 \cdot T\left(\frac{n}{4}\right) + 2n = 8 \cdot T\left(\frac{n}{8}\right) + 3n = \cdots = O(n \cdot \log_2 n)$$

Note that we've used a recurrence relation to calculate $T(n)$ a few times. There is a Master's theorem for getting a "closed-form" formula from such recursive relation. We give it without proof.

Master's Theorem: Let $a \geq 1$, $b \geq 1$, $c \geq 1$ be constants. Let $T(n)$ be defined on nonnegative integers by recurrence:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + n^c$$

Then:

$$T(n) = \begin{cases} \Theta(n^c), \text{ if } a < b^c \\ \Theta(n^c \cdot \log n), \text{ if } a = b^c \\ \Theta\left(n^{\log_b a}\right), \text{ if } a > b^c \end{cases}$$

Note that the three cases correspond to the "fix-up" step dominates; balance; and small problems dominate, respectively. Roughly speaking, you want to keep a and c small but b large.

Large Number Multiplication

The problem: Let x and y be two n-digit numbers. Compute $x \cdot y$.

A straightforward algorithm (like what a human does) would require $O(n^2)$ time to multiply each digit of x with each digit of y. Let's try to do divide-and-conquer.

Let $m = \frac{n}{2}$. Let $x = 2^m \cdot x_1 + x_2$ and $y = 2^m \cdot y_1 + y_2$. Then

$$x \cdot y = 2^n \cdot x_1 y_1 + 2^m \cdot (x_1 y_2 + x_2 y_1) + x_2 y_2$$

So, we have $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + O(n)$. By Master's theorem, $a = 4$, $b = 2$, and $c = 1$. So, $T(n) = n^2$. Disappointing…

The trick is to reduce a.

$$x \cdot y = x_1 y_1 (2^m + 1)2^m - (x_1 - x_2)(y_1 - y_2)2^m + x_2 y_2 (2^m + 1)$$

Note that the multiplication with $2^m + 1$ takes $O(n)$ time. So,

$$T(n) = 3 \cdot T\left(\frac{n}{2}\right) + O(n)$$

Thus, by Master's theorem, $a = 3$, $b = 2$, and $c = 1$. So, $T(n) = n^{\log_2 3} < n^{1.59}$.

A similar trick works for matrix multiplication. Let A and B be two $n \times n$ matrices. The product of A, B takes $O(n^3)$ number multiplications by its definition. By divide and conquer with some tricks, one can reduce this to $O\left(n^{\log_2 7}\right) < O(n^{2.81})$. Read online to find out how.

Median Selection

The problem: Given a list of numbers, find the median.

Median is important to the robust statistics. For example, as long as less than half of the data is contaminated, median will not give an arbitrarily bad value.

A human would normally sort the list to find the median. This takes $O(n \log n)$ time. One can do much better with divide and conquer. Let us generalize the problem to find the k-th largest element. For simplicity let us assume all numbers are distinct.

The idea is similar to quick sort:

Algorithm Select(L, k): L is the list and k is the rank.

1. If L is short then use sorting algorithm and return.
2. Find a "pivot" number x. Put all elements $\geq x$ in list A, and all elements $< x$ in list B.
3. If $|A| > k$ then return Select(A, k).
4. Else return Select(B, k-|A|).

So, suppose we know how to choose the pivot so that $\max(|A|, |B|) \leq \frac{n}{1+\epsilon}$ for a constant number $0 < \epsilon < 1$. Then,

$$T(n) \leq T\left(\frac{n}{1+\epsilon}\right) + O(n).$$

By Master's theorem, this would be a linear time algorithm, i.e., $T(n) = O(n)$. So, the key is to find that pivot.

Algorithm FindPivot

Input: A list L of length n.

Output: A pivot from L.

1. Divide L into $\frac{n}{5}$ groups of 5 elements each.
2. Find the median of each group.
3. Return the median of the $\frac{n}{5}$ medians by recursion.

Note that for simplicity we assume n is divisible by 5. Otherwise, we just need to do some simple treatment to the last group that has fewer than 5 elements.

Now we only need to show that the pivot found by Algorithm FindPivot divides the list with a better than constant ratio. Let x be the pivot found.

At least half of the $\frac{n}{5}$ medians are $\geq x$. That is, at least $\frac{n}{10}$ groups have their medians $\geq x$. So, these groups contain at least $\frac{3}{10} \cdot n$ elements that are $\geq x$. Similarly, we can show that at least $\frac{3}{10} \cdot n$ elements are $\leq p$. So x divides the elements with a better than constant ratio.

The complexity is consists of $T\left(\frac{n}{5}\right)$ for the recursion in FindPivot; at most $T\left(\frac{7n}{10}\right)$ for the recursion in Select(L,K); and $O(n)$ overhead. Thus,

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

This recurrence relation can't be solved by Master's theorem. But we can prove $T(n) \leq c \cdot n$ for some constant c by induction. This is better than the $O(n \log n)$ for sorting.

Note that 5 is the smallest group size for the purpose. Why? How about having larger groups, such as 7?

Remark: Divide-and-conquer is a simple but powerful algorithm design technique. While merge sort is one of its simplest application, other applications involve problem-specific techniques in order to be able to divide-and-conquer.

Greedy

Greedy algorithm is often used in optimization problems that involve many steps of choices. At each step, the greedy algorithm tries to maximize the gain for the moment. Hopefully the local optimum choices can final yield a solution that has a provable performance (ideally optimal). This does not always work. But it works fairly often.

One example is to give changes in Canadian coin system: 200, 100, 25, 10, 5, 1 cent. A cashier tries to use the minimum number of coins to add up to m cents for $0 < m < 1000$. A cashier can use the following strategy:

Repeat

1. Adding the maximum valued coin that is smaller than m. Suppose the coin has value k.
2. $m \leftarrow m - k$.

Until $m = 0$.

This is a greedy algorithm. You actually can prove that at least in Canadian coin system, this algorithm will give the optimal solution.

Other popular examples of greedy algorithms include Huffman code and Minimum Spanning Tree. In these examples greedy also provide the optimal solution. We examine Minimum Spanning Tree here.

Minimum Spanning Tree

A spanning tree of a graph G is a connected, acyclic subgraph of G. If G is weighted, a minimum spanning tree (MST) is the one with the minimum total edge weight.

Algorithm MST

Input: Graph $G = \langle V, E \rangle$

Output: An edge set $T \subseteq E$ that form the MST

1. $T \leftarrow \emptyset$
2. While T is not a spanning tree yet

2.1 Find the minimum weighted edge $e \in E \setminus T$ such that $T \cup \{e\}$ is acyclic.

2.2 $T \leftarrow T \cup \{e\}$.

This is clearly a greedy algorithm. The algorithm clearly outputs a spanning tree. The question is whether this is the minimum weighted one.

Theorem. Algorithm MST finds the minimum spanning tree.

Proof: Suppose another spanning tree T' is optimal. Examine the first edge $e \in T \setminus T'$ that was added to T in algorithm MST. Adding e to T' will form exactly one cycle. On the cycle, there must be at least one edge $e' \notin T$. We claim that $w(e') \geq w(e)$. Otherwise, MST would have chosen e' instead of e at this step. Now we modify T' by replacing e' with e. This will not increase the weight (if not reducing) but modify T' closer to T.

Keep doing this will either disprove the optimality of T' or modify T' to be equal to T. In the latter case, T is also optimal.

QED.

Dynamic Programming

Dynamic programming is a very powerful algorithm design strategy and used very frequently. In general, a problem can be tried with dynamic programming if the solution of a larger instance can be constructed by the solution of a smaller instance. This statement reminds us about recursion. But there are some subtle differences between the two. The first difference is that

recursion starts with the large instance but dynamic programming starts with the smaller instances.

An improper example is the Fibonacci number: $F(n) = F(n-1) + F(n-2)$. If recursion is used straightforwardly, then it takes exponential time. But the following simple trick makes it run in linear time:

1. F[1] ← 1, F[2] ← 1.
2. For i from 3 to n
2.1 F[n] ← F[n-1] + F[n-2]

Of course, dynamic programming has other characteristics that make it hard to argue that this simple algorithm is dynamic programming. Let's check a more nontrivial example:

Word Wrap

A text editor or a web browser would wrap a long text into multiple lines to fit the width of the window/page/screen. To make things simple, let's assume the wrap can only happen between the words (no hyphened break of a word). To make things even simpler, punctuations are regarded as a part of their preceding word. There are two different goals for a line wrap:

1. Minimize the number of lines needed.
2. Make it visually appealing.

It turns out that the two goals are different. For the first goal, a simple greedy algorithm will do the work: Keeping adding new words to the end of current line until it overflows; then start a new line.

Theorem: The greedy algorithm produces a line wrap solution that has the minimum number of lines.

Proof: Clearly removing the first word from a text will not increase the number of lines needed. So, it makes no point to start a new line when the next word can fit in the current line.

QED.

This method is used by many modern word processors, such as OpenOffice.org Writer and Microsoft Word (according to Wikipedia). This algorithm always uses the minimum possible number of lines but may lead to lines of widely varying lengths.

Another program, TeX, written by Don Knuth, tried to avoid those ugly lines. For the second goal, one wants to avoid long gaps at the end of each line (except for the last line). We'd rather have several small gaps than having a long gap in one line but no gap in other lines. To formalize this idea, the "raggedness" is defined as $\sum_{\ell=1}^{k} g_\ell^2$, where g_ℓ is the number of whitespaces at end of each line. (Don't worry about the last line yet. We'll deal with it later.) We want to minimize the raggedness.

The following example shows that greedy algorithm will fail on the raggedness measure:

Input text: `aaa bb cc ddddd`

With a line width of 6, the greedy algorithm gives

```
------    Line width: 6
aaa bb    Remaining space: 0
cc        Remaining space: 4
ddddd     Remaining space: 1
```

with a total raggedness of $0^2 + 4^2 + 1^2 = 17$. But a better solution is

```
------    Line width: 6
aaa       Remaining space: 3
bb cc     Remaining space: 1
ddddd     Remaining space: 1
```

with a total raggedness of $3^2 + 1^2 + 1^2 = 11$.

Under the raggedness measure, the optimal word wrap can be found by dynamic programming.

Let $w_1, \ldots, w_n$ be a list of integers that represent the lengths of words (in terms of letters or dots). Let w_0 be the length of a whitespace between two words. Let W be the maximum width of a line. Define $D[i]$ be the minimum raggedness of wrapping the first i words, *with the last line contributing to the raggedness*.

Note that the optimal solution for $D[i]$ consists of a solution for the first j words, plus the last line consisting of $w_{j+1}, \ldots, w_i$. The cost of the last line is $f(j+1, i) = W - \sum_{k=j+1}^{i} w_k - (i-j)w_0$. The cost of the other lines should be the optimal cost for the first j words. Otherwise, one can replace the wrapping with the optimal one to reduce $D[i]$, a contradiction. Therefore,

$$D[i] = D[j] + f(j+1, i)$$

The problem is that we do not know j. But we can try every j such that $\alpha_{j+1,i} \geq 0$, and take the one that minimizes the above value. In another word

$$D[i] = \min_{j<i \text{ s.t. } f(j+1,i)\geq 0} \big(D[j] + f(j+1, i)\big)$$

Now the algorithm to compute the minimum raggedness (including the last line):

Algorithm WordWrapDP:

1. Let k be the largest number such that $f(1, k) \geq 0$.
2. For i from 1 to k
2.1 $D[i] \leftarrow f(1, i)$.

3. For i from $k+1$ to n
3.1 $D[i] \leftarrow \min_{j<i \text{ s.t. } f(j+1,i)\geq 0}\big(D[j]+f(j+1,i)\big)$

A few questions arise immediately:

1. Time complexity?
2. How to compute the actual solution?
3. How to exclude the last line?

Time complexity:

Straightforwardly, $f(j+1,i)$ takes $O(n)$, the min function repeats $O(n)$ times, 3.1 repeats $O(n)$ times. So, it is $O(n^3)$.

But more carefully, $\sum_{k=j+1}^{i} w_k$ can be computed in $O(1)$ time with a pre-computed prefix sum:

$$\sum_{k=j+1}^{i} w_k = \sum_{k=1}^{i} w_k - \sum_{k=1}^{j} w_k$$

So, we can pre-calculate $\sum_{k=1}^{i} w_k$ for each i and store them in an array. Then each calculation of $f(j+1,i)$ takes constant time.

Also, the min function in step 3.1 only repeats $O(m)$ times, where m is the maximum number of words a line can hold. Thus the total running time is $O(mn)$.

Compute the actual solution:

The algorithm only computes the optimal cost, but not the actual wrapping solution. The solution is computed by a standard backtracking procedure.

Backtracking:
1. $i \leftarrow n$
2. While $i \geq 0$
2.1 Let $j' = \operatorname{argmin}_{j<i \text{ s.t. } f(j+1,i)\geq 0}\big(D[j]+f(j+1,i)\big)$
2.2 Print j'.
2.3 $i \leftarrow j'$.

The backtracking algorithm will print the indices of the last word of each line, beginning from the second last line to the first line.

Time complexity of the backtracking is $O(mk)$ where k is the number of lines in the optimal solution. This is strictly smaller than computing the dynamic programming table $D[\cdot]$. This is usually the case for a dynamic programming algorithm. Thus, most dynamic programming algorithm will compute the optimal cost table first, and do the backtracking in a separate step. Constructing the optimal solution while calculating the table is okay but usually cost more time.

Exclude the last line:

The last line really shouldn't be included in the cost (raggedness). It's actually very simple, the optimal raggedness not including the last line is

$$\min_{j<i \text{ s.t. } f(j+1,i)\geq 0} D[j]$$

Backtracking just start with the j that minimizes the above formula.