COMP3121: Algorithms & Programming Techniques
Summary notes – Week 1

Gerald Huang

Updated: June 6, 2020

Contents

1 Lecture 1 – Introduction

Content covered here can be found in the Lecture 1A and Lecture 1B recordings.

Date: June 2, 2020

- **Algorithm**: Collection of precisely defined steps that are executable using certain specified mechanical methods.
  - **Mechanical**: methods that do not involve any creativity, intuition or intelligence.

- **COMP3121**: Only deal with **sequential deterministic** algorithms.
  - *Sequential*: Given as sequences of steps, assume that only one step can be executed.
  - *Deterministic*: Action of each step gives the same result whenever this step is executed for the same input.

1.1 Examples of algorithms

Problem 1: Two thieves

Two thieves have robbed a warehouse and have to split a pile of items without price tags on them. Design an algorithm for doing this in a way that ensure that each thief believes that he has got at least one half of the loot.

Solution:
Let the thieves be T_1, T_2.

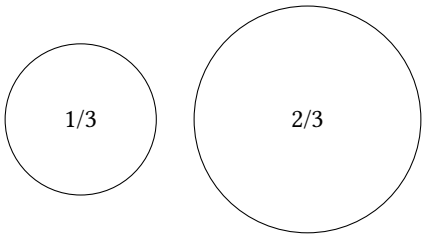
T_1 splits the pile into two parts, so that he believes that both parts are of equal value. The other thief then chooses the part that he believes is no worse than the other.

Problem 2: Three thieves

Three thieves have robbed a warehouse and have to split a pile of items without price tags on them. Design an algorithm for doing this in a way that ensure that each thief believes that he has got at least one third of the loot.

Solution:
Let T_1, T_2, T_3 be the thieves.

T_1 splits the loot into two piles, one containing a third and the other being two-thirds.



T_1 asks T_2 if he believes that Pile 1≤ 1/3. If T_2 agrees, then T_1 asks T_3. If T_3 agrees, then T_1 takes Pile 1. If T_3 disagrees, then T_3 takes (note that T_1 and T_2 both agree that Pile 1 ≤ 1/3). If T_2 disagrees, then T_2 splits Pile 1 into two parts and distributes some to Pile 2. T_2 asks if Pile 1 is now ≤ 1/3 to T_3. If T_3 agrees, then T_2 takes. Else, T_3 takes. The remaining two thieves split the loot using the technique from Problem 1.

Algorithm

Listing 1: *Three thieves algorithm*

```
1  T1 makes a pile P1 which he believes is 1/3 of the whole loot;
2  T1 process to ask T2 if T2 agrees that P1 ≤ 1/3;
3  If T2 agrees, then T1 asks of T3 agrees that P1 ≤ 1/3;
4     If T3 agrees then T1 takes P1;
5     T2 and T3 split the rest as in Problem 1.
6     Else if T3 says no, then T3 takes P1;
7          T1 and T2 split the rest as in Problem 1.
8  Else if T2 says no, then T2 reduces the size of P1 to P2 < P1 such that T2 thinks P2 = 1/3;
9     T2 then proceeds to ask T3 if he agrees that P2 ≤ 1/3;
10    If T3 agrees then T2 takes P2;
11         T1 and T3 split the rest as in Problem 1.
12    Else if T3 says no, then T3 takes P2;
13         T1 and T2 split the rest as in Problem 1.
```

1.2 Role of proofs

- Use of proofs: when it is not clear whether an algorithm terminates and returns a designated result by inspecting the algorithm using common sense!

Use proofs when:

- it is **NOT** clear from a description of an algorithm that such an algorithm will not enter an infinite loop.
- it is **NOT** clear that an algorithm will not run in exponentially many steps (in the size of the input), which is usually almost as bad as never terminating.

1.2.1 The Stable-Matching Problem (Gale-Shapley Algorithm)

Problem: Stable-Matching Problem

Assume that you are running a dating agency and have n men and n women as customers.

- *They all attend a dinner party; after the party:*
  - *every man gives you a list with his ranking of all women present,*
  - *every woman gives you a list with her ranking of all men present.*
- *Design an algorithm which produces a* ***stable matching****.*

Solution:
Employ **Gale-Shapley algorithm** (runs in $O(n^2)$ time). Define a man who has not been paired with a woman to be **free**.

Listing 2: *Gale-Shapley algorithm*

```
Begin with all men free;
While there exists a free man who has not proposed to all women:
   pick a free man m and have him propose to the highest ranking woman w on his list he has not proposed to yet;

   If no one has proposed to w yet:
       she always accepts and a pair p=(m,w) is formed;
   Else she is already in a pair p'=(m',w);
       If m is higher on her pref. list than m', then
           the pair p'=(m',w) is deleted;
           m' becomes a free man;
           a new pair p=(m,w) is formed;

       Else m is lower on her pref. list than m';
           the proposal is rejected and m remains free;
```

Claim 1: Algorithm terminates after $\leq n^2$ rounds of the "while" loop.
Proof

- In every round of the while loop, one man proposes to one woman.
- Every man can propose to a woman at most once.
- Thus every man can make at most n proposals.
- There are n men, so in total there can be $\leq n^2$ proposals.

Thus, there while loop can be executed **no more** than n^2 many times. □

Claim 2: Algorithm produces a matching.

Proof

- Assume that the while loop has terminated but m is still free.
- This means that m has already proposed to every woman.
- Thus every woman is paired with a man because a woman is not paired with anyone only if no one has made a proposal to her.
- But this would mean that n women are paired with all of n men so m cannot be free (contradiction!) □

Claim 3: The matching produced by the algorithm is **stable**.

Proof

Note that during the while loop:

- a woman is paired with men of increasing ranks on her list.
- a man is paired with women of decreasing ranks on his list.

Assume now that the matching is **NOT** stable. Then there exist two pairs

$$p = (m, w), \quad p' = (m', w')$$

such that

$$m \text{ prefers } w' \text{ over } w,$$
$$w' \text{ prefers } m \text{ over } m'.$$

- Since m prefers w' over w, he must have proposed to w' before proposing to w.
- Since m is paired with w, woman w' must have either
  - rejected him because she was already with someone whom she prefers, or
  - dropped him later after a proposal from someone whom she prefers.

  In both cases, she would now be with m' whom she prefers over m (contradiction!).

Extra reading:

- *Wikipedia article*
- *Original paper*
- *Geeks for Geeks - Stable Marriage Problem*
- *Stable Marriage Problem - Numberphile*

2 Lecture 2 – Divide and conquer

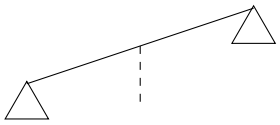
Content covered here can be found in the Lecture 2A and Lecture 2B recordings.

Date: June 4, 2020

- **Counterfeit coin puzzle**:
  - 27 coins, same denomination.
  - The counterfeit coin is lighter than the regular coins.
- **Goal**: Find the counterfeit coin by weighing them on a balance only three times.

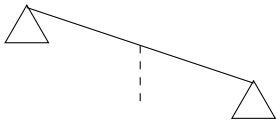
Solution.
Split the coins into three groups of nine. Place two groups on the balance. There are three outcomes.

1. Balance tilted on the left.



   $\implies$ RHS is lighter $\implies$ Counterfeit coin is in the RHS group.

2. Balance tilted on the right.



   $\implies$ LHS is lighter $\implies$ Counterfeit coin is in the LHS group.

3. Balance is evenly distributed.

   $\implies$ Both groups have even weight $\implies$ Counterfeit coin is in the remaining unchosen group.

Repeat the same process on the chosen group (split the 9 coins into three groups of 3). Repeat the same process on the 3 coins.

- This is known as **divide and conquer**.

2.1 Inversions of an array

- **Definition**: An *inversion* is a pair (i, j) such that $i < j$ but $a[i] > a[j]$.

- **Example**: Consider the array $A[1, 11, 9, 12, 7, 10, 3, 4, 6, 8, 2, 5]$. The tuple $i = 1$ and $j = 2$ (starting at 0) gives us $A[1] = 11$ and $A[3] = 9$. So $i < j$ but $A[1] > A[2]$. Hence the pair $(1, 2)$ is an inversion.
- Tells you how far the array is from being sorted. An inversion of 0 implies that the array is sorted.

2.1.1 Counting number of inversions

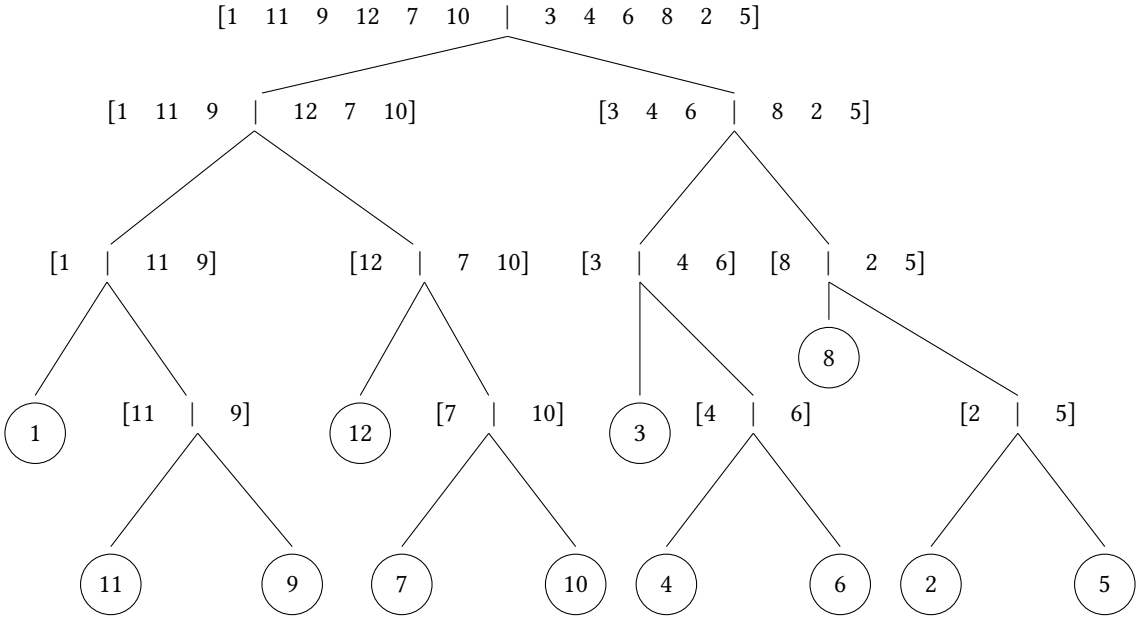
- The main idea is to tweak the **merge sort** algorithm by extending it to recursively both sort the array and determine the number of inversions in A.
- Split the array into two (approximately) equal parts.

$$A_{\text{top}} = A[1, \dots, \lfloor n/2 \rfloor], \quad A_{\text{bottom}} = A[\lfloor n/2 \rfloor + 1, \dots, n].$$

- Recursively sort the arrays A_{top} and A_{bottom} and obtain the number of inversions $I\left(A_{\text{top}}\right)$ and $I\left(A_{\text{bottom}}\right)$.
- Merge the two sorted arrays A_{top} and A_{bottom} while counting the number of inversions $I\left(A_{\text{top}}, A_{\text{bottom}}\right)$ which are across the two subarrays.
  - In each merge, place two pointers i, j to the smallest elements in each subarray. Note that $i < j$.
  - If $a[i] > a[j]$, then $a[i + k] > a[j]$ for all $k \geq 0$ in the first group. Increment the number of inversions by $(k + 1)$. Increment the j pointer.
  - If $a[i] < a[j]$, then increment i.
  - Merge the two sorted arrays and repeat the same for the remaining groups.

Consider the array $A = [1, 11, 9, 12, 7, 10, 3, 4, 6, 8, 2, 5]$. Find the number of inversions in A.

Begin by applying the merge sort on the array.



Now, merge the first two nodes. Note that $1 < 11$ so there's no inversion. Repeating that process gives the following subarrays

$$[1, 11] \quad [9, 12] \quad [7, 10] \quad [3, 4] \quad [6, 8] \quad [2, 5].$$

We then merge the subarrays and keep track of any inversion. Our current inversion count is 0. Let i point to the smallest element in the first subarray and j point to the smallest element in the second subarray.

$$\begin{array}{cc} [1, 11] & [9, 12] \\ \uparrow & \uparrow \end{array}$$

Since the first pointer is smaller than the second pointer, then there is no inversion. We just increment our first pointer.

$$\begin{array}{cc} [1, 11] & [9, 12] \\ \uparrow & \uparrow \end{array}$$

Now since the new pointer is bigger than the second pointer, any number after this pointer will **also** be bigger than the second pointer j. In this case, it will just be 1 inversion. So we increment our inversion counter. When we get to the end of the second subarray, we then just merge the two subarrays and return the number of inversions.

Repeat this process until we have a sorted array. In this example, we should end up with $1 + 4 + 4 + 11 + 21 = 41$ inversions.

Extra reading:

- *Geeks for Geeks – Counting inversions in an array*
- *Super helpful YouTube video that explains counting the # of inversions*

2.2 Arithmetic and Multiplication

Algorithm for adding two numbers

- Adding 3 bits (carry and two numbers) can be done in constant time.
- Addition runs in linear time ($O(n)$ may steps).
  - Cannot be made faster since the algorithm needs to read every bit of input.

Algorithm for multiplying two numbers

- Assume that two X's can be multiplied in $O(1)$ time.
- Algorithm runs in $O(n^2)$ time.

Divide and conquer approach:

- Take two input numbers, A and B. Split them into two halves:

$$A = A_1 \cdot 2^{n/2} + A_0 \qquad \underbrace{XX\cdots X}_{n/2}\underbrace{XX\cdots X}_{n/2}$$

$$B = B_1 \cdot 2^{n/2} + B_0$$

- A_1 and B_1 are the **most significant bits** (ie the first $n/2$ digits) while A_0 and B_0 are the **least significant bits** (ie the last $n/2$ digits).
- AB can be calculated as

$$\boxed{AB = A_1B_1 \cdot 2^n + (A_1B_0 + B_1A_0) \cdot 2^{n/2} + A_0B_0.}$$

Algorithm

Listing 3: *Recursive multiplication algorithm*

```
function MULT(A, B)
    if |A| = |B| = 1 then return AB;
    else
        A_1 ← MoreSignificantPart(A),
        B_1 ← MoreSignificantPart(B),
        A_0 ← LessSignificantPart(A),
        B_0 ← LessSignificantPart(B),
        X ← MULT(A_0, B_0),
        Y ← MULT(A_0, B_1),
        Z ← MULT(A_1, B_0),
        W ← MULT(A_1, B_1),

        return W · 2^n + (Y + Z) · 2^{n/2} + X;
    end if
end function
```

- Each multiplication of two n digit numbers is replaced by four multiplications of $n/2$ digit numbers, plus we have a linear overhead to shift and add:

$$T(n) = \underbrace{4T(n/2) + cn}_{\text{4×[run time of half] plus some overhead}}.$$

Claim: If $T(n) = 4T(n/2) + cn$, then $T(n) = n^2(c+1) - cn$.

Proof. Assume that this statement is true for $n/2$. Then

$$T\left(\frac{n}{2}\right) = \left(\frac{n}{2}\right)^2 (c+1) - c\left(\frac{n}{2}\right).$$

Then

$$\begin{aligned} T(n) = 4T(n/2) + cn &= 4\left[\left(\frac{n}{2}\right)^2 (c+1) - c\left(\frac{n}{2}\right)\right] + cn \\ &= 4\left[\frac{n^2}{4}(c+1) - \frac{cn}{2}\right] + cn \\ &= n^2(c+1) - 2cn + cn \\ &= n^2(c+1) - cn. \end{aligned}$$

- Note that $T(n) = O(n^2)$ – runs in *Quadratic* time.

2.3 Karatsuba trick

- Take two input numbers, A and B and split them into two halves.
- AB can be calculated as

$$\begin{aligned} AB &= A_1B_1 \cdot 2^n + (A_1B_0 + A_0B_1) \cdot 2^{n/2} + A_0B_0 \\ &= A_1B_1 \cdot 2^n + [(A_1 + A_0)(B_1 + B_0) - A_1B_0 - A_0B_0] \cdot 2^{n/2} + A_0B_0. \end{aligned}$$

- Saves one multiplication at each recursion.

Listing 4: *Karatsuba algorithm*

```
function MULT(A, B)
    if |A| = |B| = 1 then return AB;
    else
        A_1 ← MoreSignificantPart(A),
        B_1 ← MoreSignificantPart(B),
        A_0 ← LessSignificantPart(A),
        B_0 ← LessSignificantPart(B),
        U ← A_0 + A_1,
        V ← B_0 + B_1,
        X ← MULT(A_0, B_0),
        W ← MULT(A_1, B_1),
        Y ← MULT(U, V),

        return W · 2^n + (Y − X − W) · 2^{n/2} + X;
    end if
end function
```

- Run time satisfies $T(n) = 3T(n/2) + cn$.
  - Replace n with $n/2$:
    $$T(n/2) = 3T(n/2^2) + cn/2.$$
  - Replace n with $n/2^2$:
    $$T(n/2^2) = 3T(n/2^3) + cn/2^2.$$
- So

$$\begin{aligned}
T(n) &= 3T\left(\frac{n}{2}\right) + \frac{cn}{2} = 3\left[3T\left(\frac{n}{2^2}\right) + \frac{cn}{2}\right] + cn \\
&= 3^2 T\left(\frac{n}{2^2}\right) + nc\left[1 + \frac{3}{2}\right] \\
&\vdots \\
&= 3^{\lfloor \log_2 n \rfloor} T\left(\frac{n}{\lfloor \log_2 n \rfloor}\right) + cn\left[1 + \frac{3}{2} + \cdots + \left(\frac{3}{2}\right)^{\lfloor \log_2 n \rfloor - 1}\right] \\
&= 3^{\log_2 n} T(1) + cn\left[\frac{(3/2)^{\log_2 n} - 1}{3/2 - 1}\right] \approx 3^{\log_2 n} T(1) + 2cn\left[\left(\frac{3}{2}\right)^{\log_2 n} - 1\right].
\end{aligned}$$

Use the fact that $a^{\log_b n} = n^{\log_b a}$ to get

$$\begin{aligned}
T(n) &= n^{\log_2 3} T(1) + 2cn\left[n^{\log_2(3/2)} - 1\right] \\
&= n^{\log_2 3} T(1) + 2cn\left[n^{\log_2 3 - 1} - 1\right] \\
&= n^{\log_2 3} T(1) + 2c \cdot n^{\log_2 3} - 2cn \\
&= O(n^{\log_2 3}) = O(n^{1.58\cdots}) < O(n^2).
\end{aligned}$$

2.4 Strassen's algorithm

- By brute force, matrix multiplication is $\Theta(n^3)$.
- We can do it faster with **divide and conquer**!

- Split each matrix into four blocks of approximate size $\frac{n}{2} \times \frac{n}{2}$.
  - $a, b, c, d, e, \ldots$ are matrices!

$$P = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad Q = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad R = \begin{pmatrix} r & s \\ t & u \end{pmatrix}.$$

  - Evaluate

$$A = a(f-h) \qquad B = (a+b)h \qquad C = (c+d)e \qquad D = d(g-e)$$
$$E = (a+d)(e+h) \qquad F = (b-d)(g+h) \qquad H = (a-c)(e+f)$$

$$\begin{aligned} E + D - B + F &= (a+b)(e+h) + d(g-e) - (a+b)h + (b-d)(g+h) \\ &= ae + bg = r. \\ A + B &= (f-h)a + (a+b)h = af + bh = s. \\ C + D &= (c+d)e + d(g-e) = ce + dg = t. \\ E + A - C - H &= (a+d)(e+h) + a(f-h) - e(c+d) - (a-c)(g+h) \\ &= cf + dh = u. \end{aligned}$$

- **Remark**: We have obtained all four components using only 7 matrix multiplications and 18 matrix additions/subtractions!
- Thus, the run time is

$$T(n) = 7T(n/2) + O(n^2) = \Theta(n^{\log_2 7}) = O(n^{2.808\ldots}) \qquad \text{(by the Master Theorem)}$$