# ANLY-550 Homework 3

Yi Li, Mar. 23

1. Prove that there is a unique minimum spanning tree on a connected undirected graph when the edge weights are unique.

Suppose there are two minimum spanning trees $T_1$ and $T_2$ for a connected undirected graph $G(V, E)$. As $T_1$ and $T_2$ are distinct, the edge with the minimum weight $e_1$ must in either $T_1$ or $T_2$. Without loss of generality, we suppose this edge $e_1$ appears in $T_1$. Then $T_2 \cup \{e_1\}$ must contain a cycle, and there is one edge $e_2$ in this cycle that $e_2 \in T_2$, while $e_2 \notin T_1$. As the edge weights are unique, we know that $weight(e_1) < weight(e_2)$. Note that $T = T_2 \cup \{e_1\} \backslash \{e_2\}$ is a spanning tree, but the total weight of $T$ is smaller than the total weight of $T_2$. This is a contradiction.

2. Show the greedy algorithm yields a completion time within a factor of 3/2 of the best possible placement of jobs. Give an example showing that the bound 3/2 is tight for this algorithm.

Let $R_i$ be the load of Machine $i$ (here $i$=1 or 2). Let $R^*$ be the completion time of the best placement. For the best placement, we know that the completion time is at least as big as the biggest job, that is, $R^* \geq \max_{j=1}^{n} r_j$. Also, the completion time is at least as big as half the sum of jobs, that is, $R^* \geq 1/2 \sum_{j=1}^{n} r_j$.

Let $R = \max\{R_1, R_2\}$ be the result of the greedy algorithm ($R \geq R^*$). Without loss of generality, we suppose Machine 1 be machine with the maximum load, so $R = R_1$. Let $j_j$ be last job put on Machine 1. By greedy property, $R_1 \leq R_2 + r_j$ (Otherwise, if $R_2 < R_1 - r_j$, $j_j$ would be put on Machine 2). Also, $r_j \leq \max_{k=1}^{n} r_k$.

Now we can get that $2R_1 \leq R_1 + R_2 + r_j \leq \sum_{k=1}^{n} r_k + \max_{k=1}^{n} r_k \leq 2R^* + R^* = 3R^*$, that is, $R = R_1 \leq 3/2R^*$. This proves that this strategy yields a completion time within a factor of 3/2 of the best possible placement of jobs.

Example: Suppose we have three jobs, and their running time are 1, 1, 2. Then $R_1 = 1 + 2 = 3$, $R_2 = 1$, and $R^* = 1 + 1 = 2$. We see that $R = R_1 = 3/2R^*$.

3. (a)

For two $n$-digit numbers $x, y$ ($n$ can be divisible by 3), we can split $x$ and $y$ into $x = 10^{2n/3} \cdot a + 10^{n/3} \cdot b + c, y = 10^{2n/3} \cdot d + 10^{n/3} \cdot e + f$.
Thus, $x \cdot y = 10^{4n/3} \cdot ad + 10^n \cdot [(a+b)(d+e) - ad - be] + 10^{2n/3} \cdot [(a+c)(d+f) - ad - cf + be] + 10^{n/3} \cdot [(b+c)(e+f) - be - cf] + cf$. It uses only six multipications on the smaller parts to multiply $x$ and $y$.

3. (b)

The asymptotic running time: $T(n) = 6T(n/3) + O(n)$, so $T(n) = O(n^{\log_3 6}) \approx O(n^{1.631})$. As the asymptotic running time of Karatsuba's algorithm is $O(n^{\log_2 3}) \approx O(n^{1.585})$, which is smaller than $O(n^{1.631})$, I would rather split it into two parts as in Karatsuba's algorithm.

3. (c)

If I could use only five multiplications instead of six, then the asymptotic running time is $T(n) = O(n^{\log_3 5}) \approx O(n^{1.465})$, which is smaller than $O(n^{1.585})$. So, I would rather split it into three parts.

4. Suppose we have an array $A$ containing $n$ numbers, some of which may be negative. We wish to find indices $i$ and $j$ so that $\sum_{k=i}^{j} A[k]$ is maximized. Find an algorithm that runs in time $O(n)$.

Algorithm:

Initialization: Set the start index $i = 0$, the end index $j = 0$, the current maximum sum $curr\_max = A[0]$, and the final maximum sum $max = A[0]$.

```
For $k$ in $range(1, n)$:
  If $A[k] > curr\_max + A[k]$, then $i = k, curr\_max = A[k]$.
  Otherwise, $curr\_max = curr\_max + A[k]$.
  If $max < curr\_max$, then update $j = k, max = curr\_max$.
Return $i, j, max$.
```

Correctness and Time Analysis: If $A[k] > curr\_max + A[k]$, We can start from the index $k$ to find the maximum subarray sum ($curr\_max$ may be negative). As the $curr\_max$ stores the current maximum sum, it may be smaller than the final maximum sum. We update the final maximum sum only when $max < curr\_max$, and update the end index $j$ to index $k$. The loop visits each index for one time, so the running time is $O(n)$.

5(a). We first change the minimal imbalance problem into finding the minimum of the largest partition's weight. If we minimize the largest partition's weight, we also get the results for the minimal imbalance problem.

Let $M[n,k]$ be the value of the largest partition's weight of an array $A$ containing $n$ numbers with $k$ indices that partition the array into $k+1$ subarrays. Consider the subproblem $M[i,j]$ that $M[i,j]$ is the value of the largest partition's weight of subarray $A$ containing the first $i$ numbers with $j$ indices that partition the array into $j+1$ subsubarrays.

The recursion is:

$$M[i,j] = \max\{M_{1 \le x < i}[x, j-1], \sum_{l=x+1}^{i} A[l]\}.$$

Algorithm:
Initialization: let list $p$ store the sum of the subarrays which start from the first value in the array, that is, $p[0] = 0, p[i] = p[i-1] + A[i]$. Set $M[i,0] = p[i], M[0,j] = 0$, for others, set $M[i,j] = \infty$. Set $index[i,j] = []$ to record the position of the partition.

```
For $i$ in $range(n)$,
    for $j$ in $range(k)$,
        for $x$ in $range(i)$,
            if $M[i,j] > \max\{M_{1 \le x < i}[x, j-1], p[i] - p[x]\}$,
                $M[i,j] = \max\{M_{1 \le x < i}[x, j-1], p[i] - p[x]\}$, and set $index[i,j] = x$
```

Use $index[n,k]$ to reconstruct the optimal partition and calculate the weight of each subarray $w(i)$.
Calculate the imbalance value: $\max_i |w(i) - (\sum_{l=1}^{n} A[l])/(k+1)|$.

Correctness: Once we place the last divider $k$, the largest partition's weight will be either (1) the weight of the last partition $\sum_{l=j_{k-1}+1}^{n} A[l]$, or (2) the largest partition's weight before the last divider $M_{1 \le i < n}[i, k-1]$. So $M[n,k] = \max\{M_{1 \le i < n}[i, k-1], \sum_{l=j_{k-1}+1}^{n} A[l]\}$. The goal is to minimize $M[n,k]$, so once $M[i,j] > \max\{M_{1 \le x < i}[x, j-1], p[i] - p[x]\}$, we need to update $M[i,j]$.

The imbalance value is either the biggest weight minus the average weight, $w(i) - (\sum_{l=1}^{n} A[l])/(k+1)$, or the average weight minus the smallest weight, $|w(i) - (\sum_{l=1}^{n} A[l])/(k+$

1)|. As $M[n,k] \geq (\sum_{l=1}^{n} A[l])/(k+1)$ is always true, when we minimize $M[n,k]$, we minimize the biggest weight and also maximize the smallest weight (as the sum is a constant), so we minimize the imbalance value.

Time analysis: According to the loop, the time is $O(n*k*i), i \leq n$, so the time is $O(kn^2)$.

5(b). We only need to change the last step of the algorithm: Calculate the imbalance value: $\sum_i |w(i) - (\sum_{l=1}^{n} A[l])/(k+1)|$.

Actually the new calculation formula of the imbalance is more relax than the previous one, so the new algorithm will give the same partition way as the previous one, and only the imbalance value may be different.

6. Find a dynamic programming algorithm to determine the neatest way to print a paragraph.

Let $c[n]$ be the total penalty for the neatest way to print words $l_1$ through $l_n$.

Consider the subproblem that $c[j]$ is the optimized total penalty for arranging words from $l_1$ to $l_j$.

Let $extras[i,j]$ indicate the extra space on a line containing words $l_i$ through $l_j$. Let $lc[i,j]$ indicate the penalty on a line containing words $l_i$ through $l_j$, which is the cube of $extras[i,j]$ except the last of the cube of the extra space at the end of the line.

The recursion is:

$$c[j] = \min\{c[i] + lc[i+1,j], c[j]\}(1 \leq i < j).$$

Algorithm:

```
For $j$ from 1 to $n$,
  for $i$ from 1 to $j$,
    $c[j] = \min\{c[i] + lc[i+1,j], c[j]\}$.
```

Correctness: The sum penalty for words $l_1$ through $l_j$ ($c[j]$) can either be (1) $c[j]$ itself (if adding the word $l_j$ still keep all words on one line) or (2) the sum penalty for all lines except the last line, which is the words $l_1$ through some $l_i$ ($1 \leq i < j$), plus the penalty for words on the last line $l_{i+1}$ through $l_j$.

Analysis: According to the loop, the time is $O(n^2)$.

7. Find the size of the maximum-sized independent set and the set itself in linear time.

Let $I(j)$ be the size of the maximum-sized independent set in the subtree rooted at vertex $j$. Let $G(j)$ be the set of all the grandchildren of vertex $j$, and let $C(j)$ be the set of all the children of vertex $j$. The recursive equation is:

$$I(j) = \max\{1 + \sum_{u \in G(j)} I(u), \sum_{v \in C(j)} I(v)\}.$$

Algorithm:

```
Initialization: Set the initial size of the maximum independent set $I = 0$, and set the initial set of the maximum independent set $s = []$.
For each vertex $j$ of $G$ in the postorder of the DFS of $G$,
    if $j$ is a leaf, then $s.append[j]$ and $I = I + 1$,
    else if $1 + \sum_{u \in G[j]} I(u) > \sum_{v \in C[j])} I(v)$, then $s.append[j]$ and $I = I + 1$.
Return the size $I$, and the set $s$.
```

Correctness: Suppose that we know the size of the maximum-sized independent set of all subtrees below a vertex $j$. There are only two cases of the maximum independent set in the subtree hanging from $j$: either $j$ is in the maximum independent set, or it is not. If it is not, then the maximum independent set is simply the union of the maximum independent sets of the subtrees of the children of $j$. If $j$ is in the maximum independent set, then the maximum independent set consists of $j$, plus the union of the maximum independent sets of the subtrees of the grandchildren of $j$.

As we visit each vertex in postorder, we always find a leaf before the leaf's parent. The number of leaves hanging from a vertex is always no smaller than 1, so always adding leaves to the independent set can make sure the independent set is maximized and does not generate conflict.

Time analysis: As we visit each vertex only once in postorder, the time is $O(n)$.