CS2223 Algorithms. B Term 2013
Homework 2 Solutions
By Artem Gritsenko, Ahmedul Kabir, Yun Lu, and Prof. Ruiz

We start by including the Master Theorem here for your reference.

Master Theorem: Suppose that T(n) is a function on the nonnegative integers that satisfies the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Let $k = log_b a$. Then:

Case 1. If $f(n) = O(n^{k-\varepsilon}) for\ some\ constant\ \varepsilon > 0, then\ T(n) = \Theta(n^k)$

Case 2. If $f(n) = \Theta(n^k log^p n) then\ T(n) = \Theta(n^k log^{p+1} n)$

Case3. If $f(n) = O(n^{k+\varepsilon}) for\ some\ constant\ \varepsilon > 0, and\ if\ af\left(\frac{n}{b}\right) \le cf(n) for\ some\ constant\ c < 1,$ $and\ all\ sufficiently\ lare\ n, then\ T(n) = \Theta(f(n))$.

Problem1: (Solutions by Yun Lu)

Algorithm I: finds a solution to its input problem by dividing it into 5 subproblems, each of half the size of its input problem, recursively solves each of these 5 subproblems, and then combines their solutions to form the solution of the input problem in linear time (= n).

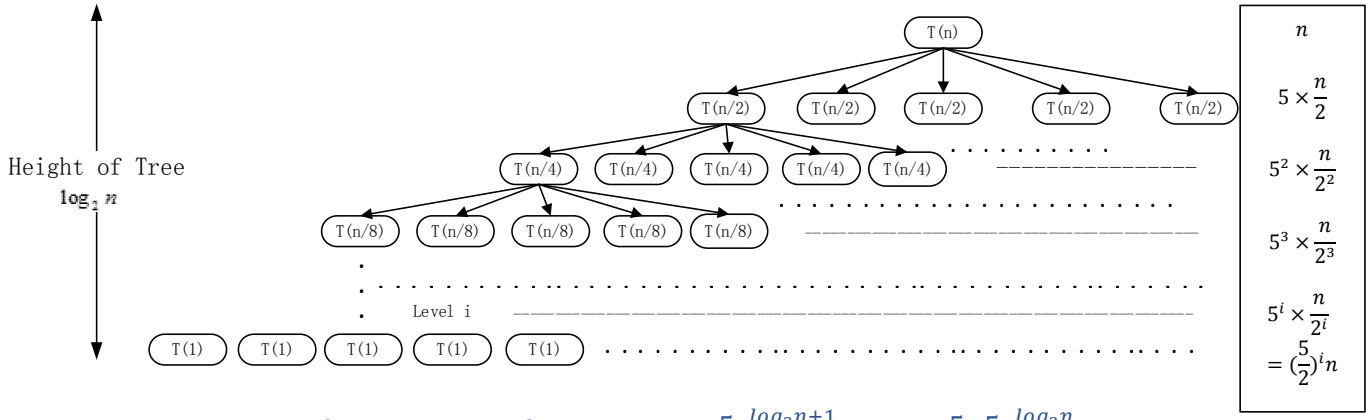
Recurrence for algorithm I: $\begin{cases} T(n) = 5T\left(\frac{n}{2}\right) + n \\ T(1) = 1 \end{cases}$

1) Solution to this recurrence using the Master Theorem:
   $a = 5,\ b = 2,\quad f(n) = n,\quad k = log_2 5,\ n^{log_2 5} = n^{2.322}$

   Case 1 of the master theorem applies here since f(n)= $O(n^{2.322-1})$, where $\varepsilon = 1 > 0$.

   Therefore, T(n) =$\Theta(n^{log_2 5})$

2) Solution to this recurrence using the Recursion Tree Method:



$$T(n) = \sum_{i=0}^{log_2 n} \left(\frac{5}{2}\right)^i n = n \sum_{i=0}^{log_2 n} \left(\frac{5}{2}\right)^i = n \frac{\left(\frac{5}{2}\right)^{log_2 n+1} - 1}{\left(\frac{5}{2}\right) - 1} = n \frac{\frac{5}{2}\left(\frac{5}{2}\right)^{log_2 n} - 1}{\left(\frac{3}{2}\right)}$$

$$= \frac{2n}{3}\left[\frac{5}{2}\left(\frac{5^{log_2 n}}{2^{log_2 n}}\right) - 1\right] = \frac{2n}{3}\left[\frac{5}{2}\left(\frac{n^{log_2 5}}{n}\right) - 1\right] = \frac{5}{3} n^{log_2 5} - \frac{2}{3} n$$

Since, $log_2 5 = 2.322 > 1$

Hence, $T(n) = \Theta\left(\frac{5}{3} n^{2.322} - \frac{2}{3} n\right) = \Theta\left(n^{log_2 5}\right)$.

3) Solution to this recurrence using the Substitution Method:

Need to find: g(n) such that $T(n) = O(g(n))$

1. Guess $g(n) = cn^{log_2 5} + dn$
2. Use mathematical induction to prove that $T(n) \leq cn^{log_2 5} + dn$ for all n > 0:
   - **Base case:** $n = 1, T(1) = 1 \leq 1^{log_2 5}\, c + d$ so we must have $1 \leq c + d$
   - **Induction hypothesis:** Let n > 1. Assume that for all $k < n,\ T(k) \leq ck^{log_2 5} + dk$.
     Prove that this property holds for n also:

$$T(n) = 5T\left(\frac{n}{2}\right) + n$$

$$\leq 5\left[c\left(\frac{n}{2}\right)^{log_2 5} + d\left(\frac{n}{2}\right)\right] + n$$

by the induction hypothesis since $\frac{n}{2} < n$,

$$= 5c\frac{n^{log_2 5}}{2^{log_2 5}} + 5d\left(\frac{n}{2}\right) + n$$

$$= 5c\frac{n^{log_2 5}}{5} + \frac{5dn}{2} + n$$

$$= cn^{log_2 5} + n\left(\frac{5d}{2} + 1\right)$$

and we want the above expression to be $\leq cn^{log_2 5} + dn$

Thus, constants c and d must satisfy the following two constraints:

$$\frac{5d}{2} + 1 \leq d, which\ means\ that\ d \leq -\frac{2}{3}$$

And, $1 \leq c + d$ (from the base case)

Let $\begin{cases} d = -\frac{2}{3} \\ c = \frac{5}{3} \end{cases}$, $\quad then\ we\ have: g(n) = \frac{5}{3}n^{log_2 5} - \frac{2}{3}n$

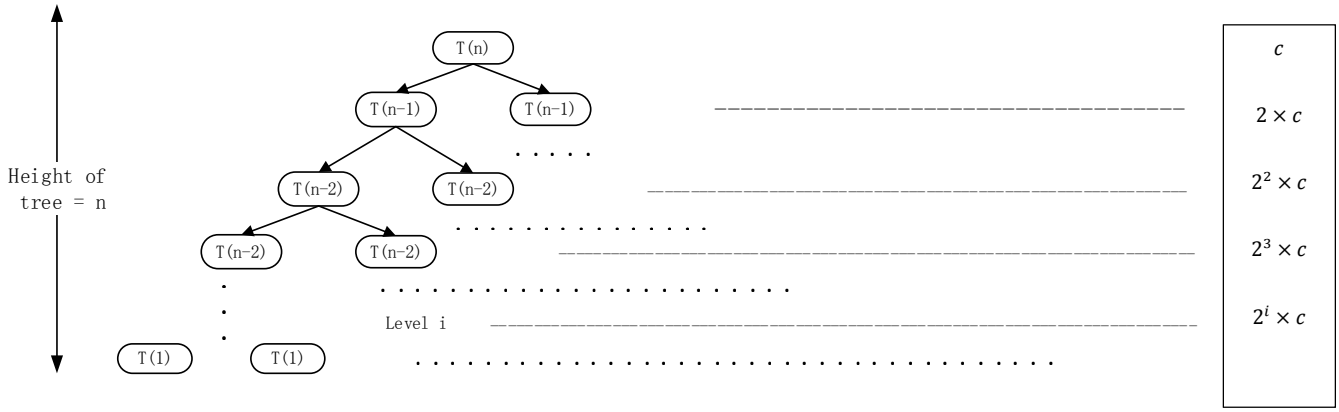
$T(n) = \Theta\left(2n^{log_2 5} + (-1)n\right) = \Theta(n^{log_2 5})$

Algorithm II: finds a solution to its input problem of size n by recursively solving 2 subproblems of size n-1, and then combines their solutions to form the solution of the input problem in constant time.

Recurrence for algorithm II: $\begin{cases} T(n) = 2T(n-1) + c \\ T(1) = 1 \end{cases}$

1) Solution to this recurrence using the Master Theorem:
   The master theory is not applicable here since the size of the subproblems is not a fraction of the size of the input problem. That is, there is no constant *b* for which the size of the subproblems is *n/b*.

2) Solution to this recurrence using the Recursion Tree Method:



$$T(n) = \sum_{i=0}^{n-1} (2)^i c = c\frac{2^n - 1}{2 - 1} = c(2^n - 1)$$

$T(n) = \Theta(2^n - 1) = \ominus(2^n - 1) = \Theta(2^n)$

3) Solution to this recurrence using the Substitution Method:
   Need to find: g(n) such that $T(n) = O(g(n))$
   1. Guess $g(n) = c2^n + d$
   2. Use mathematical induction to prove $T(n) \leq c2^n + d$:
      - **Base case:** $n = 1, T(1) = 1 \leq c2^1 + d$ so we need $1 \leq 2c + d$
      - **Induction hypothesis:** Let n > 1. Assume that for all $k < n,\ T(k) \leq c2^k + d$.

Prove that this property holds for n also:

$$T(n) = 2T(n-1) + 1$$

$$\leq 2[c2^{n-1} + d] + 1, \qquad by\ the\ induction\ hypothesis\ since\ n-1 < n$$

$$= 2c2^{n-1} + 2d + 1$$
$$= c2^n + 2d + 1$$
$$\leq c2^n + d$$

Thus, constants c and d must satisfy the following two constraints:

$2d + 1 \leq d \rightarrow d \leq -1$

And $1 \leq 2c + d$ (from the base case)

Let $\begin{cases} d = -1 \\ c = 1 \end{cases}$, $\qquad then\ we\ have: g(n) = 2^n + (-1)$

$T(n) = \Theta(2^n + (-1)) = \Theta(2^n)$

Algorithm III: finds a solution to its input problem of size n by recursively solving 9 subproblems of size n/3, and then combines their solutions to form the solution of the input problem in quadratic time (= n^2).

For algorithm III: $\begin{cases} T(n) = 9T\left(\frac{n}{3}\right) + n^2 \\ T(1) = 1 \end{cases}$

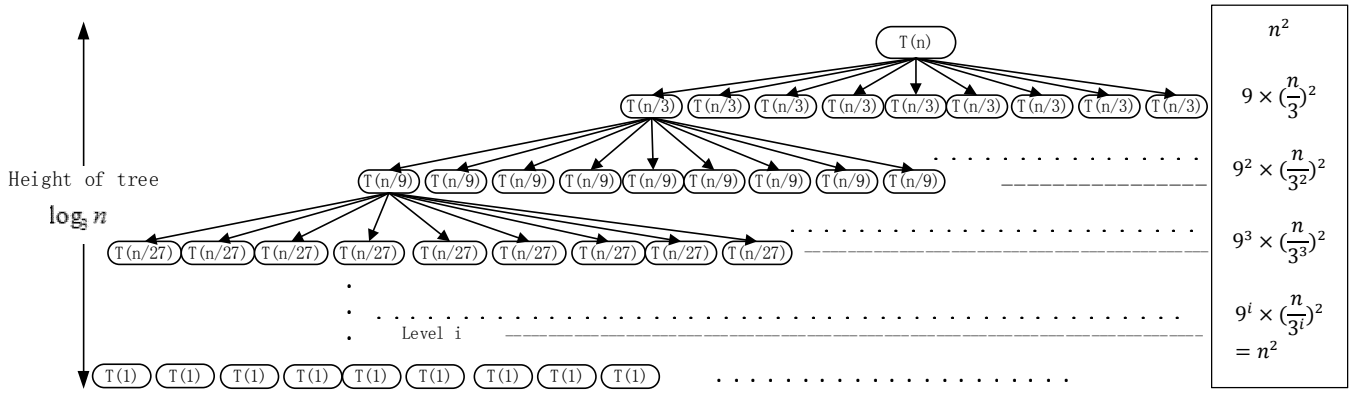
1) Solution to this recurrence using the Master Theorem:

   $a = 9, \quad b = 3, \quad f(n) = n^2, \quad k = log_3 9, \quad n^{log_3 9} = n^2$

   Case 2 of the master theorem applies here since f(n)= $\Theta(n^2)$

   Therefore, T(n) = $\Theta(n^2 \log n)$

2) Solution to this recurrence using the Recursion Tree Method:



$$T(n) = \sum_{i=0}^{log_3 n} n^2 = n^2 log_3 n = \Theta(n^2 \log n)$$

3) Solution to this recurrence using the Substitution Method:

Need to find: $\mathrm{g(n)}$ such that $T(n) = \mathrm{O(g(n))}$

1. Guess $\mathrm{g(n)} = cn^2 log_3 n + dn^2 + bn$
2. Use mathematical induction to prove $T(n) \leq cn^2 log_3 n + dn^2 + bn$:
   - **Base case:** $\mathrm{n} = 1, \mathrm{T}(1) = 1 \leq clog_3 1 + d + b, so\ we\ must\ have\ that: 1 \leq d + b$
   - **Induction hypothesis:** Let n > 1. Assume that for all $k < n, T(k) \leq ck^2 log_3 k + dk^2 + bk$.

     Prove that this property holds for n also:

$$T(n) = 9T\left(\frac{n}{3}\right) + n^2$$
$$\leq 9\left[c\left(\frac{n}{3}\right)^2 log_3\left(\frac{n}{3}\right) + d\left(\frac{n}{3}\right)^2 + b\left(\frac{n}{3}\right)\right] + n^2$$
$$by\ the\ induction\ hypothesis\ since\ \frac{n}{3} < n,$$
$$= 9c\left(\frac{n}{3}\right)^2 log_3\left(\frac{n}{3}\right) + 9d\left(\frac{n}{3}\right)^2 + 9b\left(\frac{n}{3}\right) + n^2$$
$$= cn^2 log_3\left(\frac{n}{3}\right) + dn^2 + 3bn + n^2$$
$$= cn^2(log_3 n - log_3 3) + (d+1)n^2 + 3bn$$
$$= cn^2 log_3 n - cn^2 + (d+1)n^2 + 3bn$$
$$= cn^2 log_3 n + (d + 1 - c)n^2 + 3bn$$

and we want the above expression to be $\leq cn^2 log_3 n + dn^2 + bn$

Thus, constants b, c and d must satisfy the following two constraints:

$$\begin{cases} (d + 1 - c) \leq d \\ 3b \leq b \end{cases}$$

$And\ 1 \leq d + b$ (from the base case)

$$let \begin{cases} d = 1 \\ c = 1, \\ b = 0 \end{cases}$$ then we have: $g(n) = n^2 log_3 n + n^2$

$T(n) = \Theta(n^2 log_3 n + n^2) = \Theta(n^2 log_3 n)$

Everything else being equal, which one of the 3 algorithms would you choose to solve the problem? Explain your answer.

For algorithm I: $T(n) = 5T\left(\frac{n}{2}\right) + n = \Theta(n^{log_2 5})$

For algorithm II: $T(n) = 2T(n-1) + 1 = \Theta(2^n)$

For algorithm III $T(n) = 9T\left(\frac{n}{3}\right) + n^2 = \Theta(n^2 \log n)$

Since algorithm II runs in exponential time, it will be the slowest. So we need to compare algorithm I and algorithm III.

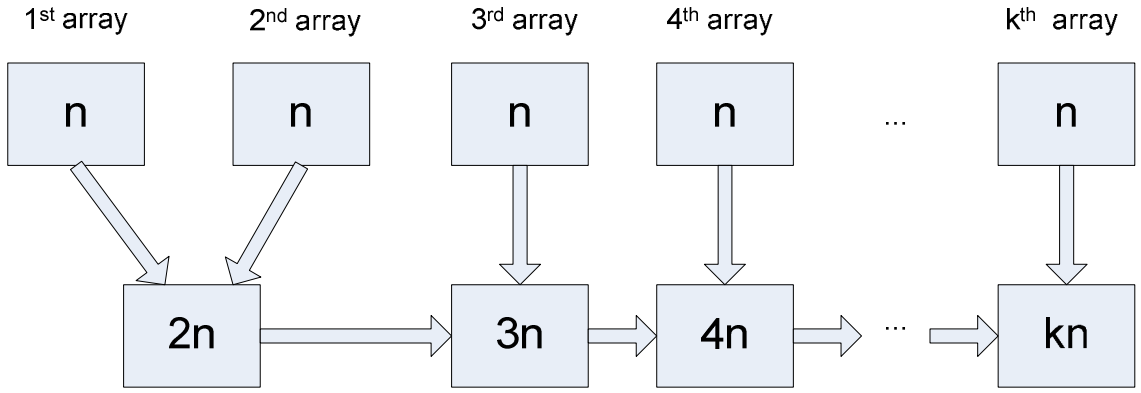
$$\lim_{n\to\infty} \frac{n^{log_2 5}}{n^2 log_3 n} = \lim_{n\to\infty} \frac{n^{log_2 5 - 2}}{log_3 n} = \lim_{n\to\infty} \frac{n^{2.32-2}}{log_3 n} = \lim_{n\to\infty} \frac{n^{0.32}}{log_3 n}$$
$$= \lim_{n\to\infty} \frac{0.32n^{-0.68}}{\frac{1}{nln3}}\ (using\ L'Hôpital's\ rule) = \lim_{n\to\infty} (0.32n^{-0.68}) n\ ln3 = \infty$$

$Thus, n^2 log_3 n = O(n^{log_2 5})$.

In conclusion, based on the asymptotic behavior of the runtime of these algorithms, I will chose algorithm III that has runtime $\Theta(n^2 \log n)$ to solve the problem.

Problem 2. (By Artem Gritsenko)

Strategy I: Naïve-Multi-Merge.



1) We call the merge procedure for first two input arrays, and then we merge the resulting array with the next input array one by one. The merge procedure for the first two arrays would take time proportional to that of *2n* comparisons, the merge procedure for the resulting array and the 3rd input array would require *3n* comparisons in the worst case and so on. Let's say that each comparison is performed in time *c*. This would result in the following runtime function:

$$T(n,k) = 2nc + 3nc + 4nc + ... + knc = cn(2 + 3 + 4 + ... + k) = cn(\frac{(k+1)k}{2} - 1)$$

$$= cn(\frac{k^2 + k - 2}{2}) = \frac{1}{2}ck^2 n + \frac{1}{2}ckn - cn\,.$$

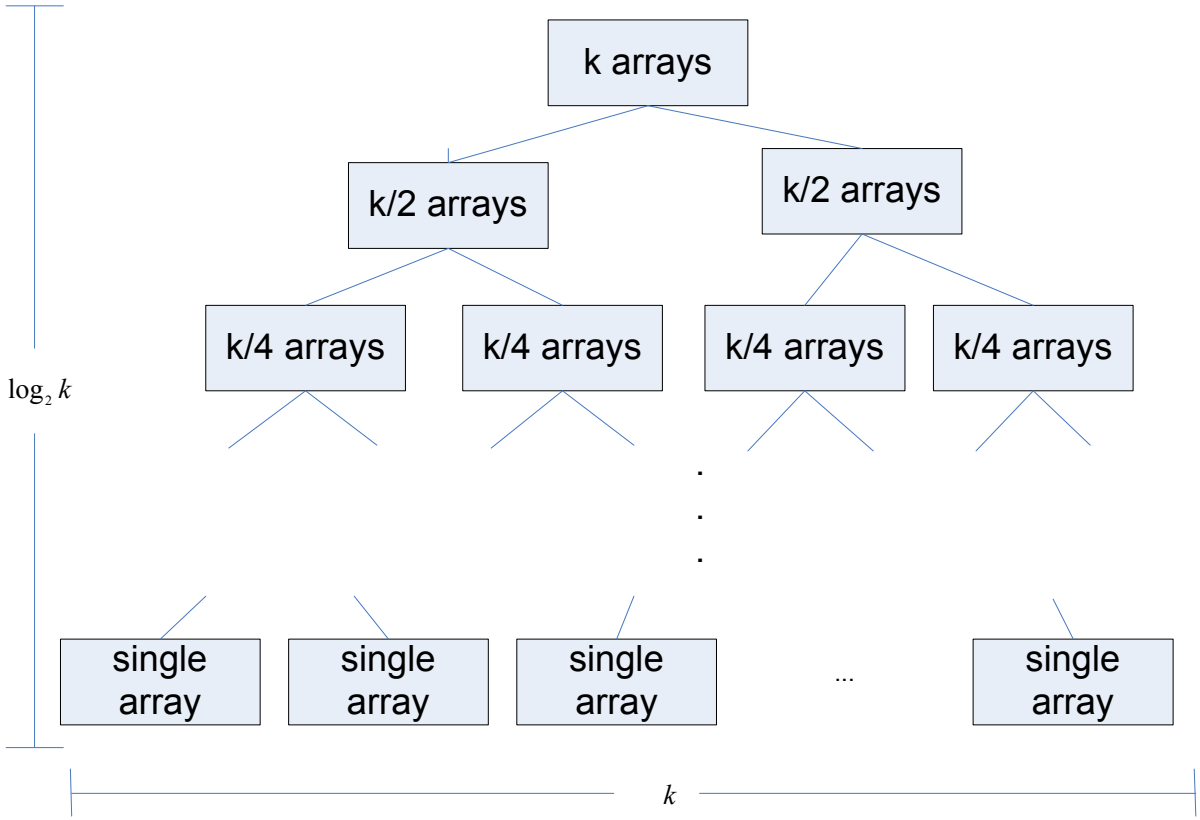
2) Claim: $T(n) = O(k^2 n)$

Proof: To prove this we need to find $n_0 > 0$ and $c_0 > 0$ such that for all $n \geq n_0$ the following holds: $T(n) \leq c_0 g(n)$. We can say that $\frac{cnk^2}{2} < ckn^2$, $\frac{cnk}{2} < ckn^2$, $-cn < ckn^2$ for n > 1. That means we can represent $T(n) = cnk^2 + cnk^2 + cnk^2 = c_0 nk^2$, where $c_0 = 3c$.

Thus, $T(n) = O(k^2 n)$.

2. **Smart-Multi-Merge:**

1) The idea of the algorithm is to utilize recursion to make the multi-merge more efficient. To do this we split the input list of arrays $[A_1,...A_k]$ into two parts of the same size $[A_1,...A_{k/2}]$ and $[A_{k/2},...A_k]$. Each of these parts we split in two subparts again. We continue the process until we reach a list with a single array $[A_i]$. From this stage we can start merging the subparts to acquire the sorted arrays. We make this recursively until we merge all the arrays.



2) **SmartMultiMerge Pseudo-code.**

The input parameters of the function is *ListOfArrays* - the list of arrays to divide.

SmartMultiMerge(*ListOfArrays*)

```
k = size(ListOfArrays)
if k=1 then
        return ListOfArrays
else
        mid = k / 2
        LeftList = [ListOfArrays[1],..., ListOfArrays[mid]]
        left = SmartMultiMerge (LeftList)
        RightList = [ListOfArrays[mid+1],..., ListOfArrays[k]]
        right = SmartMultiMerge (RightList)
        return merge(left, right)
```

The procedure is the following: we pass the initial list of arrays. If the length of the list is 1 that means there is only one array in the list and we are done. Otherwise, we split the list into two parts and execute the function recursively for each of the parts. The recursive calls stop when we achieve lists with single arrays (already sorted) in both left and right parts. After we acquire the sorted left and right parts we merge them and repeat the procedure recursively.

3) **Correctness of SmartMultiMerge**.

We will show that the algorithm works correctly, using a proof by (strong) induction on *k*.

For the base case, consider a list of one element-array (which is the base case of the algorithm). Such an array is already sorted, so the base case is correct.

For the induction step, suppose that algorithm will correctly merge any list of less than *k* arrays. Now suppose we call SmartMultiMerge on a list of *k* arrays. It will recursively call SmartMultiMerge on two lists of arrays of size *k/2*. By the strong induction hypothesis, these calls will merge these list of arrays correctly. Hence, after the recursive calls, we would have two sorted arrays. We can apply the merge procedure to these arrays, since this procedure is correct (see section 2.3.1 of the textbook).

This concludes our proof.

4) Recurrence for the runtime T(n,k) of SmartMultiMerge

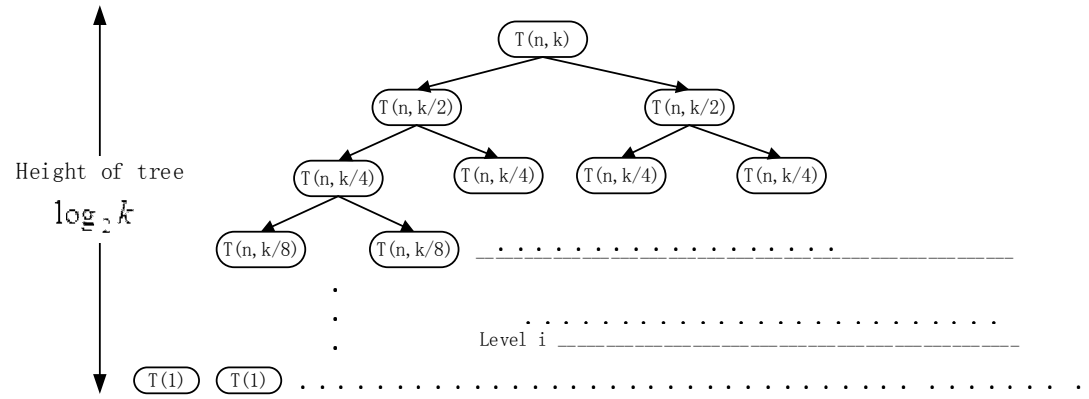
$$T(n,k) = \begin{cases} c, \; if \; k = 1 \\ 2T(n,\frac{k}{2}) + nk, if \; k > 1 \end{cases}.$$

5) **Big-O analysis of the runtime T(n,k) of SmartMultiMerge**

We use the "unrolling" recurrence tree method. Because we split the k elements (arrays) on two parts, the total number of splitting levels (tree height) is equal to $\log_2 k$.

The number of operations that we need to execute on each level of the tree is equal to *kn*. We can figure it out by going from the top level of the tree down to the bottom. On the first level we have one problem and the time needed to execute is *nk*, according to the recurrence (4). On the second level we have 2 subproblems each one of size *n(k/2)*. To solve them we need *2n(k/2)* time, which is equal to *nk*. On the third level we have 4 subproblems each of the size of *n(k/4)* that results in *4n(k/4)=nk* time. Continuing in the same manner we can see that each level of the tree is takes exactly *kn* time (except the very last level of the tree, which is executed in *k* time), multiply by the number of tree levels gives us:

$$T(n,k) = nk(\log_2 k - 1) + k = nk\log_2 k - nk + k = O(nk\log k)$$



nk

$2n\frac{k}{2} = nk$

$2^2 n\left(\frac{k}{2^2}\right) = nk$

$2^3 n\left(\frac{k}{2^3}\right) = nk$

$2^{i-1} n\left(\frac{k}{2^{i-1}}\right) = nk$

$T(n,k) = nk\log_2 k$

Problem 2 Part 3 (Ahmedul Kabir)

Python Implementation:

```
import time

# Regular merge function. Takes two sorted lists left and right, and returns their merged list
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0

    while left_idx < len(left) and right_idx < len(right):
        # Find the minimum of the two and append it to the result array
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    # Fill the result of the result array with the elements from the one that is still left
    if left:
        while left_idx < len(left):
            result.append(left[left_idx])
            left_idx += 1
    if right:
        while right_idx < len(right):
            result.append(right[right_idx])
            right_idx += 1
    return result

# Naive implementation of multi-merge. Takes a list one at a time and merges it with newArr
def naiveMultiMerge(arrays):
    newArr = arrays[0]                  # Start with first list
    for i in range(1, len(arrays)):
        newArr = merge(newArr,arrays[i])     # Keep merging one at a time
    return newArr

# Smart implementation of multi-merge. Uses a recursive procedure to merge lists two at a time
# Merges the lists starting from 'first' to 'last' position
def smartMultiMerge(arrays, first, last):
    if first == last:
        return arrays[first]           # Just one list, divide part complete
    if first < last:
        mid = (first + last) / 2
        left = smartMultiMerge(arrays, first, mid)      # Merge the first half from first to mid
        right = smartMultiMerge(arrays, mid + 1, last)  # Merge the second half from mid+1 to last
        return merge(left, right)        # Conquer: Merge the two merged halves

# Runtime evaluation
n = 100
k = 16
arrays = [[i for i in range(n)] for j in range(k)]

start_time = time.clock()
naiveMultiMerge(arrays)
end_time = time.clock()
print ("Naive multi-merge took " + str(end_time - start_time) + " seconds")

start_time = time.clock()
smartMultiMerge(arrays,0,len(arrays)-1)
end_time = time.clock()
print ("Smart multi-merge took " + str(end_time - start_time) + " seconds")
```

Runtimes

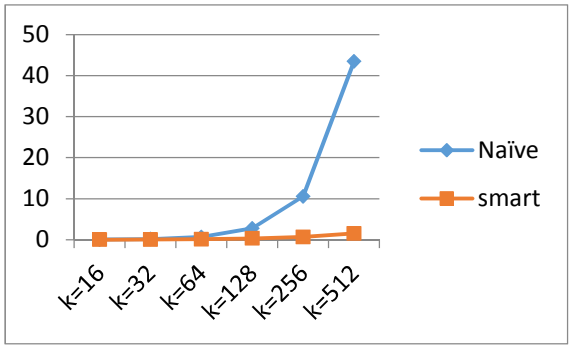
Naïve merge

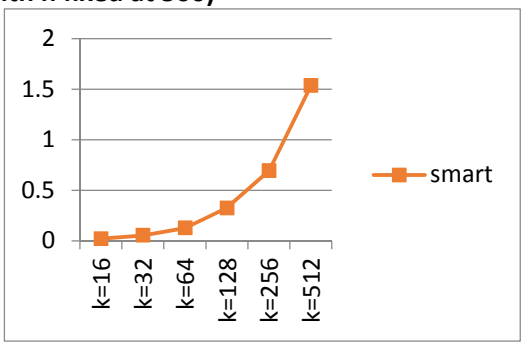
n\k	16	32	64	128	256	512
100	0.004	0.035	0.131	0.523	2.067	8.178
200	0.018	0.067	0.27	1.041	4.131	16.977
300	0.026	0.102	0.397	1.55	6.381	25.618
400	0.036	0.135	0.541	2.155	8.37	34.205
500	0.043	0.171	0.701	2.787	10.602	43.456

Smart merge

n\k	16	32	64	128	256	512
100	0.009	0.01	0.024	0.058	0.135	0.301
200	0.008	0.02	0.049	0.115	0.262	0.628
300	0.013	0.031	0.076	0.174	0.412	0.946
400	0.017	0.043	0.1	0.255	0.544	1.244
500	0.02	0.053	0.127	0.325	0.692	1.535

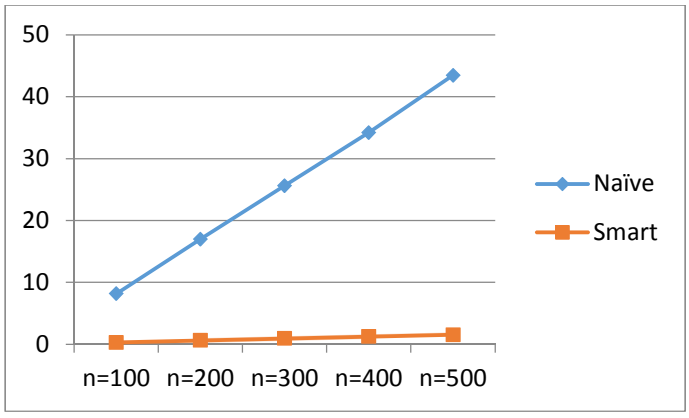
Growth with respect to k (with n fixed at 500)

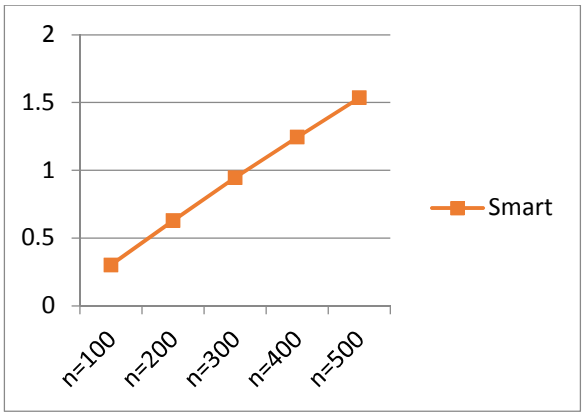




As ***k*** increases, Naïve-merge grows in a quadratic manner, i.e. in the order of k^2, whereas Smart-merge grows in the order of ***klogk***.

Growth with respect to n (with n fixed at 512)





As **n** increases, both Naïve-merge and Smart-merge grow linearly.

Final conclusion:

For Naïve-merge, the order is $O(nk^2)$

For Smart-merge, the order is O(nklogk)

Solution to Problem 3 (Ahmedul Kabir)

Problem: State whether the base of a logarithmic expression in each of the cases below can be ignored or not. That is, whether the particular base of the logarithm ($\log_2$, $\log_{10}$, ln, ...) makes a difference or not. If it can be ignored, prove it rigorously. If it cannot be ignored, show an example in which the base of the logarithm makes a difference.

1. **O($\log_a n$), where a is a constant greater than 1.**

   Base of the logarithm does NOT make a difference in this case. Note that $\log_a n = \frac{\log_c n}{\log_b a}$ where b is a constant. Here $\log_b a$ is also a constant. So changing the base from *a* to *b* changes the value of the logarithm only by a constant factor, which doesn't make a difference in terms of order of the function. In other words, O($\log_a n$) = O($\log_b n$) for all a, b > 1.

2. **O($n^{\log_a b}$), where a and b are constants greater than 1.**

   Here the base of the logarithm DOES matter. Suppose a = 2 and b = 4. Then $n^{\log_a b} = n^{\log_2 4} = n^2$ However, if we change the value of a to 4, then the function becomes $n^{\log_a b} = n^{\log_4 4} = n$. Clearly n and n^2 are of different orders.

3. **O($\log_{f(n)} n$), where f(n) is a function of n (e.g., f(n) = √ n).**

   The base DOES matter in this case as well. Note that $\log_{f(n)} n = \frac{\log_c n}{\log_b f(n)}$ (where c is a constant). But this time, the denominator is no longer a constant, but a function of *n*. So changing the base from *f(n)* to *c* changes the value of the logarithm by a factor which is a function of *n*, which can make a difference in the order of the function. For a concrete example, let f(n) = $\sqrt{n}$ which leads to $\log_{f(n)} n = \log_{\sqrt{n}} n = 2$ which is a constant. Now if we change the base to 2, the function becomes $\log_{f(n)} n = \log_2 n$ which grows faster than a constant.

4. **O($\log_a n$), where a is a constant smaller than 1.**

   In this case, the base DOES matter, since $\log_a n$ when a < 1 is a *strictly decreasing* function, i.e. its value decreases as *n* increases. If we change *a* to any constant *c* > 1, then the resulting function will be strictly increasing, and so it will not be of the same order as the previous one. To illustrate that $\log_a n$ with a < 1 is monotonically decreasing, let us use a = $\frac{1}{2}$ which gives us $\log_{1/2} n = -\log_2 n$ since $\log_{1/2} n$ = x iff $(\frac{1}{2})^x$ = n iff 2^{-x} = n iff x = $-\log_2 n$