Exercise II, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

These problems are taken from various sources at EPFL and on the Internet, too numerous to cite individually.

1 (Exercise 2.3-4 in the book) We can express insertion sort as a recursive procedure as follows. In order to sort $A[1 \dots n]$, we recursively sort $A[1, \dots n-1]$ and then insert $A[n]$ into the sorted array $A[1 \dots n-1]$.

1a Write a recurrence for the worst-case running time of this recursive version of insertion sort.

1b Solve the recurrence to give a tight asymptotic bound (in terms of the Θ-notation).

Solution:

1a. Since our initial problem size decreases by 1 in each recursive call, we can see that we will have n recursive calls. Because we are looking for the worst case, we can approximate the cost to insert the new element at each recursive step at $\Theta(n)$ (the time needed to check all element of the array before inserting the new one). So we found the following recursion: $T(n) = T(n-1) + \Theta(n)$.

1b. We can easily see that we are going to make n recursive calls and at each recursive call we have a cost equal to cn for some constant $c > 0$. So a good guess for a tight asymptotic bound for the worst case running time is $\Theta(n^2)$. The following calculation makes us strongly believe that our guess is good:

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-2) + cn + c(n-1) \\ &= ... \\ &= cn + c(n-1) + c(n-2) + ... + 3c + 2c + c \\ &= c\sum_{i=1}^{n} i \\ &= c \cdot \frac{n(n+1)}{2} = \Theta(n^2) \end{aligned}$$

Now we will prove that our guess is correct using the substitution method. We first need to prove that n^2 is indeed an asymptotic upper bound, i.e., $T(n) = O(n^2)$. We prove by weak induction on $n \geq 1$ that there exists a constant $a > 0$ such that $T(n) \leq an^2$ for all $n \geq 1$.

The **base case** holds as long as $a \geq c$, as $T(1) = c$. We now provide the **inductive step**. We have $T(n) \leq T(n-1) + cn$, and hence by the inductive hypothesis

$$T(n) \leq a(n-1)^2 + cn = an^2 + a - 2an + cn = an^2 + a + (c - 2a)n.$$

Choosing $a \geq c$, we ensure that $a + (c - 2a)n \leq a - an = a(1 - n) \leq 0$, and thus $T(n) \leq an^2$, as required. This completes the proof of the upper bound.

We now prove the lower bound of $\Omega(n^2)$. Similarly to the above, we prove by weak induction on n that there exists a constant $b > 0$ such that $T(n) \geq bn^2$ for all $n \geq 1$. The **base case** holds as long as $0 < b \leq c$, as $T(1) = c$. We now provide the **inductive step**. We have $T(n) = T(n-1) + cn$, and hence by the inductive hypothesis

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &\geq b(n-1)^2 + cn \\ &= bn^2 + b - 2bn + cn \\ &= bn^2 + b + n(c - 2b) \\ &\geq bn^2 \end{aligned}$$

as long as $0 < b \leq c/2$. This completes the proof of the lower bound.

Putting the two bounds together, we conclude that $T(n) = \Theta(n^2)$, as required.

2 Use the substitution method (mathematical induction) to solve the following recurrences

2a Show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$

is $T(n) = n \log_2 n$.

2b Consider the recurrence $T(n) = 8T(n/2) + \Theta(n^2)$.

- Draw the recursion tree to make a qualified guess of the value of $T(n)$ in terms of the Θ-notation.
- (half a $*$) Then verify this guess using the substitution method (i.e., mathematical induction). Ask the TA's for a hint if you get stuck for too long.

Solution:

2a. We will argue by induction on n that $T(n) = n \log_2 n$ when $n \geq 2^k \; \forall \; k \geq 1$. In the inductive hypothesis, we assume that:

$$T(n/2) = (n/2) \log_2 (n/2)$$

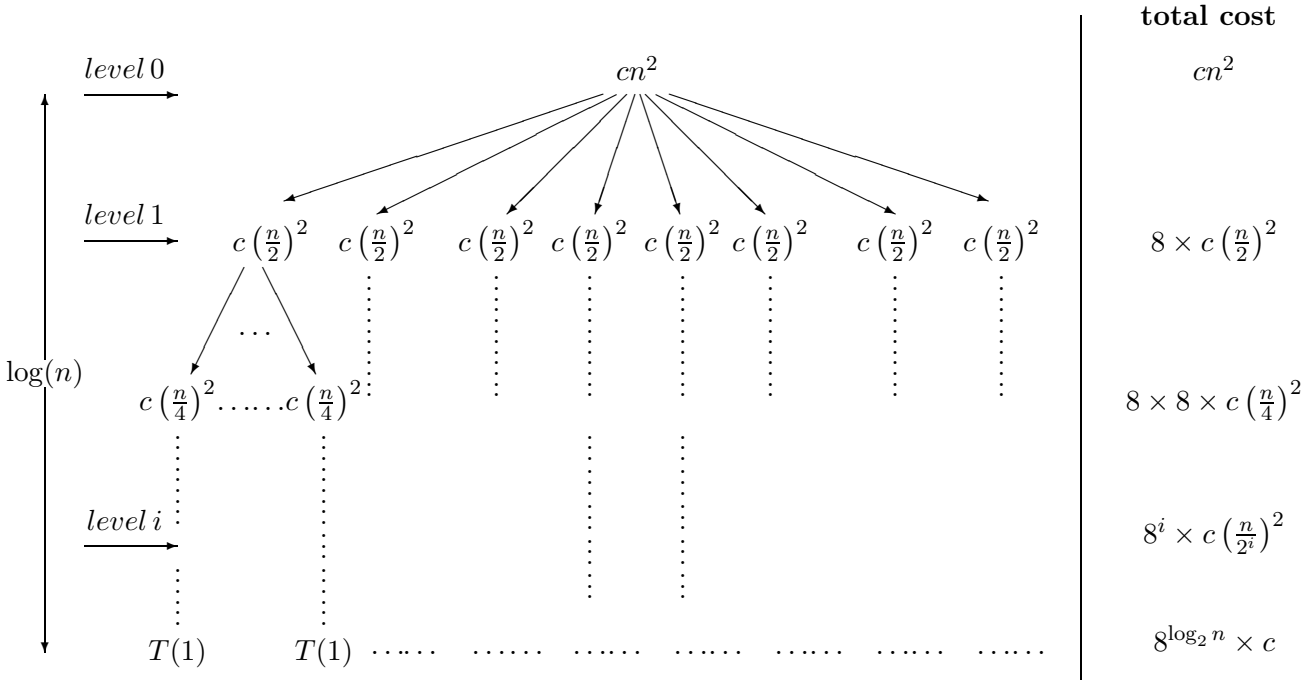
Here we need $n/2 \geq 2$, i.e. $n \geq 4$. Using the inductive hypothesis, we get:

$$\begin{aligned} T(n) &= 2T(n/2) + n && (1) \\ &= 2(n/2) \log_2 (n/2) + n && (2) \\ &= n(\log_2 n - \log_2 2) + n \\ &= n \log_2 n - n + n \\ &= n \log_2 n \end{aligned}$$

where (1) follows from the recurrence equation and (2) follows from the inductive hypothesis. Since the induction step works for $n \geq 4$, it is sufficient to show that $T(n) = n \log_2 n$ for $n = 2$ (the base case):

$$T(2) = 2 = 2 \log_2 2$$

2b. Using the recursion tree method, we will make an educated guess that $T(n) = \Theta(n^3)$.



The total running time is as follows:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_2 n - 1} c \times 8^i \times \left(\frac{n}{2^i}\right)^2 \\
&= \sum_{i=0}^{\log_2 n - 1} c \times 8^i \times \frac{n^2}{4^i} \\
&= c \times n^2 \sum_{i=0}^{\log_2 n - 1} 2^i \\
&= c \times n^2 (2^{\log_2 n} - 1) \\
&= c(n^3 - n^2) \\
&\leq cn^3 = \Theta(n^3)
\end{aligned}$$

To verify our guess, we can use substitution method. In order to show that $T(n) = \Theta(n^3)$, we will have to to show that $T(n) = \Omega(n^3)$ and $T(n) = O(n^3)$.
For $O(n^3)$-part: The recurrence implies that $T(n) \leq 8T(n/2) + cn$ for a constant $c > 0$ if $n > 1$, and $T(1) \leq c$. We will argue by induction on n that $T(n) = O(n^3)$. Specifically, we will show by strong induction on n that $T(n) \leq dn^3 - d'n^2$ for some constants $d, d' > 0$. Using the the inductive hypothesis, we get:

$$\begin{aligned}
T(n) &\leq 8T(n/2) + cn^2 \\
&\leq 8d(n/2)^3 - 8d'(n/2)^2 + cn^2 \\
&= dn^3 + n^2(-2d' + c) \\
&\leq dn^3 - d'n^2 \quad \text{if } d' \geq c
\end{aligned}$$

The base case is provided by $n = 1$, where $T(n) = T(1) \leq c$ and is satisfied by choosing d, d' so that $d - d' \geq c$. Thus, setting $d = 2c$ and $d' = c$ satisfies both constraints and completes the proof.

For $\Omega(n^3)$-part: The recurrence implies that $T(n) \geq 8T(n/2) + cn$ for a constant $c > 0$, and $T(1) \geq c$. We will argue by induction on n that $T(n) = \Omega(n^3)$. Specifically, we will show by strong induction on n that $T(n) \geq dn^3$ for a constant $d > 0$. Using the recurrence and the inductive hypothesis, we get:

$$\begin{aligned} T(n) &\geq 8T(n/2) + cn^2 \\ &\geq 8d(n/2)^3 + cn^2 \\ &= dn^3 + cn^2 \\ &\geq dn^3 \quad \text{for any } c, d > 0 \end{aligned}$$

The base case is provided by $n = 1$, where $T(1) \geq c$, and is satisfied by choosing $d \leq c$.

3 (Exercise 4.5-1 in the book) Use the master method to give tight asymptotic bounds for the following recurrences.

3a $T(n) = 2T(n/4) + 1$.

3b $T(n) = 2T(n/4) + \sqrt{n}$.

3c $T(n) = 2T(n/4) + n$.

3d $T(n) = 2T(n/4) + n^2$.

Solution:

3a. The recurrence is of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a = 2$, $b = 4$ and $f(n) = 1, \forall n$ (i.e., constant). Let $c = 0$, we have $f(n) = \Theta(1) = \Theta(n^c)$ and $c = 0 < \frac{1}{2} = \log_4(2) = \log_b(a)$. It follows from the first case of the master theorem that $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^{\log_4(2)}) = \Theta(n^{\frac{1}{2}}) = \Theta(\sqrt{n})$.

3b. We have $a = 2$, $b = 4$ and $f(n) = n^{\frac{1}{2}} = \Theta(n^c)$ for $c = \frac{1}{2} = \log_4(2) = \log_b(a)$. It follows from the second case of the master theorem that $T(n) = \Theta(n^{\log_b(a)} \log(n)) = \Theta(\sqrt{n} \log(n))$.

3c. We have $a = 2$, $b = 4$ and $f(n) = n = \Theta(n^c)$ for $c = 1 > \frac{1}{2} = \log_4(2) = \log_b(a)$. It follows from the third case of the master theorem that $T(n) = \Theta(f(n)) = \Theta(n)$.

3d. We have $a = 2$, $b = 4$ and $f(n) = n^2 = \Theta(n^c)$ for $c = 2 > \frac{1}{2} = \log_4(2) = \log_b(a)$. It follows from the third case of the master theorem that $T(n) = \Theta(f(n)) = \Theta(n^2)$.

4 Let $f(n)$ and $g(n)$ be the functions defined for positive integers as follows:

```
Function f(n):
1: ans ← 0
2: for i = 1, 2, ..., n − 1 do
3:     for j = 1, 2, ..., n − i do
4:         ans ← ans + 1
5:     end for
6: end for
7: return ans
```

```
Function g(n):
1: if n = 1 then
2:     return 1
3: else
4:     return g(⌊n/2⌋) + g(⌊n/2⌋)
5: end if
```

4a Find closed-form formulas of $f(n)$ and $g(n)$.

4b What is, in Θ-notation, the running time of these algorithms?

4c What is, in Θ-notation, the *running time* of algorithm $g(n)$ if we at line 4 replace $g(\lfloor n/2 \rfloor) + g(\lfloor n/2 \rfloor)$ by $2g(\lfloor n/2 \rfloor)$?

Solution:

4a. For $f(n)$, in the i^{th} iteration, we add 1 to ans for $n-i$ times. We do it for $i = 1, 2, \cdots, n-1$. Thus, $f(n) = \sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$.

For $g(n)$, from the description of the algorithm,

$$g(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2g(\lfloor n/2 \rfloor) & \text{if } n \geq 2 \end{cases}.$$

List $g(n)$ for small n, and we can make a guess $g(n) = 2^{k-1}$ for $2^{k-1} \leq n \leq 2^k - 1$ and positive integer k. It's not difficult to prove the correctness of this guess by induction on k.

We observe that $k = \lfloor \log_2 n \rfloor + 1$ when $2^{k-1} \leq n \leq 2^k - 1$. Therefore, $g(n) = 2^{\lfloor \log_2 n \rfloor}$.

4b. For $f(n)$, the running time we need for the i^{th} iteration is $c \cdot (n-i)$ for some constant $c > 0$, and thus the total running time is $O(1) + \sum_{i=1}^{n-1} c \cdot (n-i) = O(1) + c\frac{n(n-1)}{2} = \Theta(n^2)$.

For $g(n)$, we denote the running time as $T(n)$. We can get the following recurrence for $T(n)$.

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n/2) + T(n/2) + O(1) & \text{if } n \geq 2 \end{cases}.$$

Because we call g on $\lfloor n/2 \rfloor$ twice.

By the master method, we can get the total running time $T(n) = \Theta(n)$.

4c. If we at line 4 replace $g(\lfloor n/2 \rfloor) + g(\lfloor n/2 \rfloor)$ by $2g(\lfloor n/2 \rfloor)$, the new recurrence for the worst-case running time $T(n)$ would be

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(n/2) + O(1) & \text{if } n \geq 2 \end{cases}.$$

Because we only call g on $\lfloor n/2 \rfloor$ once.

By the master method, we can get the total running time $T(n) = \Theta(\log n)$.

5 (Problem 2-1 in the book) Although merge sort runs in $\Theta(n \log n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort make it faster for small n. Thus, it makes sense to use insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

5a Show that the n/k sublists, each of length k, can be sorted by insertion sort in $\Theta(nk)$ worst-case time.

5b Show that the sublists can be merged in $\Theta(n \log(n/k))$ worst-case time.

5c Given that the modified algorithm runs in $\Theta(nk + n \log(n/k))$ worst-case time, what is the largest asymptotic value of k, as a function of n, for which the modified algorithm has the same asymptotic running time as standard merge sort?

5d How should k be chosen in practice?

Solution:

5a. We know that insertion sort has $\Theta(n^2)$ worst-case time. Since the length of each array is k, then the worst-case time to sort each such array is $\Theta(k^2)$. Since we have n/k different lists to sort, the worst-case time is equal to $\Theta\left(\frac{n}{k} \cdot k^2\right) = \Theta(nk)$

5b. Just extending the 2-list merge to merge all the lists at once would take $\Theta(n \cdot (n/k)) = \Theta(n^2/k)$ time (n from copying each element once into the result list, n/k from examining n/k lists at each step to select next item for result list). To achieve $\Theta(n \log(n/k))$ time merging, we merge the lists pairwise, then merge the resulting lists pairwise, and so on, until there is just one list. The pairwise merging requires $\Theta(n)$ work at each level, since we are still working on n elements, even if they are partitioned among sublists. The number of levels, starting with n/k lists (with k elements each) and finishing with 1 list (with n elements), is $\log(n/k)$ Therefore, the total running time for the merging is $\Theta(n \log(n/k))$.

5c. The modified algorithm runs in $\Theta(nk + n \log(n/k))$ worst-case time and the standard merge sort runs in $\Theta(n \log n)$ worst-case time. So if we want that both have the same asymptotic running time we need the following equality to be true :

$$\Theta(nk + n \log(n/k)) = \Theta(nk + n \log n - n \log k) = \Theta(n \log n).$$

This equation will be true either if $nk = \Theta(n \log n)$, or if $n \log(n/k) = \Theta(n \log n)$. Since we are looking for the biggest k, we will choose $k = \log n$.

5d. In practice, k should be the largest list length on which insertion sort is faster than merge sort.

6 (*, Problem 2-4 in the book) Let $A[1 : n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an inversion.

6a List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

6b What array with elements from the set $\{1, 2, \ldots n\}$ have the most inversions? How many does it have?

6c What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

6d Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \log n)$ worst-case time. (Hint: Modify merge sort.)

Solution:

6a. The five inversions of the array are (1,5), (2,5), (3,4), (3,5) and (4,5).

6b. The number of inversions in an array is upper-bounded by the possible number of the unordered pairs (i, j), namely $\sum_{i=1}^{n} i - 1 = \frac{n(n-1)}{2}$. This upper bound is achieved with equality when the array is in reversed order.

6c. The running time time of insertion sort on an array $A[1, 2 \dots n]$ is $\Theta(n + I)$, where I is the number of inversions of $A[1, 2 \dots n]$.
Consider the pseudo-code of the insertion sort in CLRS [Chap2]. For every element k in the array, the inner loop runs for $1 + |I_k|$, where $I_k = \{j : (j, k) \text{ is an inversion}\}$. Hence the overall running time of the algorithm is $\Theta\left(\sum_{k=2}^{n} (1 + I_k)\right) = \Theta(n + I)$.

6d. let A be an array of size n, and let A_l and A_r denote the left and right sub-arrays of A respectively. Define $I(A)$ to be the number of inversions in an array A, Hence :

$$I(A) = \underbrace{I(A_l) + I(A_r)}_{(1)} + \underbrace{|\{(i, j) : A_l(i) > A_r(j)\}|}_{(2)}$$

In order to count the total number of inversions of an array A, we proceed as in merge sort (CLRS [Chap2]) by dividing it into left and right halves and proceeding recursively on each half to get (1). Note that the number of inversions in an array of size 1 (base-case) is 0. In order to get (2), we mimic the merge-sort algorithm by modifying the merge subroutine. When merging the left and right sorted sub-arrays, specifically while inserting $A_l(i)$ in the resulting merged array, we increment the total number of inversions by the number of elements in A_r that are strictly less than $A_l(i)$ (namely, the number of elements from A_r that are already added to the merged array).
The running time of this algorithm is thus the same as merge sort, i.e. $\Theta(n \log n)$.