CS 4120: Homework 1

Adam Camilli (aocamilli@wpi.edu)

March 23, 2018

Problem 1

Find the least integer k such that $f(n)$ is $O(n^k)$ for each of the following functions. Include values for c and n_0 as described in section 3.1, page 47 of the textbook.

- $f(n) = 2n^2 + n^3 \log n$

Using formal definition of $O(n)$, we obtain

$$\exists c, n_0 \mid 0 \leq 2n^2 + n^3 \log n \leq c(n^k) \text{ for all } n \geq n_0$$

Asymptotically, we can say

$$2n^2 + n^3 \log n \leq cn^4, n \geq n_0 \ (n_0 \geq \text{the largest root of } \pm cn^4 \mp f(n) = 0)$$

but we cannot say

$$2n^2 + n^3 \log n \leq cn^3$$

for any c or n_0, since asymptotically $\log n > c$. Therefore, the least integer is $k = 4$.

- $f(n) = 3n^5 + (\log n)^4$

Using formal definition of $O(n)$, we obtain

$$\exists c, n_0 \mid 0 \leq 3n^5 + (\log n)^4 \leq c(n^k) \text{ for all } n \geq n_0$$

Asymptotically, we can say

$$3n^5 + (\log n)^4 \leq cn^5, c > 3, n \geq n_0 \ (n_0 \geq \text{the largest root of } \pm cn^5 \mp f(n) = 0)$$

but we cannot say

$$3n^5 + (\log n)^4 \leq cn^4$$

for any c or n_0, since asymptotically $n^5 > n^4$. Therefore, the least integer is $k = 5$.

- $f(n) = \frac{n^4+n^2+1}{n^4+1}$

Using formal definition of $O(n)$, we obtain

$$\exists c, n_0 \mid 0 \leq \frac{n^4+n^2+1}{n^4+1} \leq c(n^k) \text{ for all } n \geq n_0$$

Asymptotically, we can say

$$\frac{n^4+n^2+1}{n^4+1} \leq cn^0, c > 1, n \neq 0$$

but we cannot say

$$\frac{n^4+n^2+1}{n^4+1} \leq cn^{-1}, n \neq 0$$

for any c or n_0, since asymptotically $\frac{n^4+n^2+1}{n^4+1} > n^{-1}$. Specifically,

$$\lim_{n\to\infty} \frac{n^4+n^2+1}{n^4+1} = 1 > \lim_{n\to\infty} n^{-1} = 0$$

Therefore, the least integer is $k = 0$.

- $f(n) = \frac{n^3+5\log n}{n^4+1}$

Using formal definition of $O(n)$, we obtain

$$\exists c, n_0 \mid 0 \leq \frac{n^3+5\log n}{n^4+1} \leq c(n^k) \text{ for all } n \geq n_0$$

Asymptotically, we can say

$$\frac{n^3+5\log n}{n^4+1} \leq cn^{-1}, n \neq 0 \geq n_0, \ (n_0 \geq \text{the root of } \pm cn^{-1} \mp f(n) = 0)$$

but we cannot say

$$\frac{n^3+5\log n}{n^4+1} \leq cn^{-2}, n \neq 0$$

for any c or n_0, since asymptotically $\frac{n^3+5\log n}{n^4+1} > n^{-2}$.
Therefore, the least integer is $k = -1$.

Problem 2

You have n quarters and a balance. You know that $n-1$ quarters have the same weight, and one weighs less than the others. Give an algorithm (in pseudocode) to identify the light quarter which uses the balance only $\log_3 n$ times in the worst case.

Assumptions:

1. We can use the balance to measure any amount of coins (gathering etc. costs nothing)
2. We are given normal weight W_n.
3. The worst case allowed is $\lceil \log_3 n \rceil$ uses (impossible otherwise).
4. Removing items from the balance **does not count as a use**

From these assumptions, we develop the following algorithm, using a mathematical generalization for this class of problem (source) that states the minimum number of weightings needed X can be found by

$$\frac{3^X - 3}{2} = \text{\# of coins}$$

(Pseudocode on next page)

```
/* Given values */
Wn // Normal weight
balance // the state of the balance

/* Returns whether weight is normal */
normalWeight():
  if (balance.weight == Wn*balance.coinsWeighed())
    return true
  else
    return false

/* Remove these coin(s) from scale
   DOES NOT COUNT AS A USE OF SCALE */
remove(coins):
  //elided

/* A reduced recursive algorithm for an amount of coins that is a power of 3
   Returns light coin, using exactly log3(coins.size) weighings */
3-algo(coins):
  if (coins.size <= 3):
    weigh(coins) // ONLY USE OF THE SCALE IN THIS FUNCTION (THIS IF-ELSE BLOCK MUST RETURN)
    if (normalWeight()) // This group of 3 is all normal coins
      return NULL
    else
      remove(coins[0]) // Arbitrary, could be any of the 3
      if normalWeight() // Must return by this point if size==1
         return coins[0]
      else
         remove(coins[1]) // Arbitrary, could be either of the 2
         if normalWeight()
           return coins[1] // Must return by this point if size == 2
         else // If size was two or one, will have returned
           return coins[2]
  /* Use this exact same structure with exactly 3 groups of coins. Since coins.size is a
     power of 3, it is guaranteed to eventually reduce to the if-else block above.
     Since the scale is used only once per call of 3-algo(), this means it will be used exactly
     log3(coins.size) times! */
  groups = split(coins,coins/3)
  weigh(groups) // ONLY USE OF THE SCALE IN THIS FUNCTION
    if (normalWeight()) // All coins are normal
      return NULL
    else
      remove(groups[0]) // Arbitrary, could be any of the 3
      if normalWeight()
         return 3-algo(groups[0])
      else
         remove(groups[1]) // Arbitrary, could be either of the 2
         if normalWeight()
           return 3-algo(groups[1])
         else
           return 3-algo(groups[2])

/* Main algorithm for any number of coins. Essentially, it calculates floor(X) where X is the
   generalization explained above, and therefore can split N coins into groups of 3^X, 3^(X-1), ...
   This can result in three possible groupings:

   3^X 3^(X-1) ... 3^1       (if coins.size mod 3 == 0)
   --------> just call 3-algo()
   3^X 3^(X-1) ... 3^0       (if coins.size mod 3 == 1)
   --------> call 3-algo(), if NULL then last coin (3^0) is light
   3^X 3^(X-1) ... 3^0 3^0   (if coins.size mod 3 == 2)
   --------> call 3-algo(), just call 3-algo() with last two combined as an "extra" group of two

   In the first two cases, there are exactly floor(log3(coins.size)) uses of the scale. In case 3, however,
   there will be floor(log3(coins.size)) + 1 uses. THIS IS OK, due to our assumption that
   MAX_USES = ceil(log3(coins.size)) */
lightestCoin(coins):
   // Solve generalization ((3^X) - 3)/2 = coins.size for X
   X = floor(log3(2(coins.size) + 3))
   switch(X):
     case (X mod 3 == 0):
       return 3-algo(coins)
     case (X mod 3 == 1):
       if ((result = 3-algo(coins.subArray(coins.size - 1))) != NULL)
         return result
       else
         return coins[coins.size - 1] // Will be last coin, since arrays counted from 0
     default: // X mod 3 must be 2
       if ((result = 3-algo(coins.subArray(coins.size - 2))) != NULL)
         return result
       else
         return 3-algo(coins.subArray(coins.size - 2, coins.size))
```

Problem 3

Use the Master Theorem to find the asymptotic solutions for the following recurrences 3

- $T(n) = 7T(\frac{n}{2}) + n^2$

Adapting this relation to the general formula

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

we obtain:

$$a = 7, b = 2$$
$$f(n) = n^2 \rightarrow O(n^c), c = 1$$
$$c_{\text{crit}} = \log_2 7 \approx 2.81$$
$$c < c_{\text{crit}} \rightarrow \text{ CASE 1}$$

This gives us the solution:

$$T(n) = \Theta(n^{\log_2 7})$$

- $T(n) = T(\frac{n}{2}) + 1$

Adapting this relation to the general formula

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

we obtain:

$$a = 1, b = 2$$
$$f(n) = 1 \rightarrow \Theta(n^c log^k n), c = 0, k = 0$$
$$c_{\text{crit}} = \log_2 1 = 0$$
$$c = c_{\text{crit}} \rightarrow \text{ CASE 2}$$

This gives us the solution (choose any $k \geq 0$):

$$T(n) = \Theta(n^{\log_2 1} log^{k+1} n) = \Theta(n^0 \log^1 n = \Theta(n \log n$$

- $T(n) = 4T(\frac{n}{2}) + n^3$

Adapting this relation to the general formula

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

we obtain:

$$a = 4, b = 2$$
$$f(n) = n^3 \rightarrow \Omega(n^c), c = 3$$
$$c_{\text{crit}} = \log_2 4 = 2$$
$$c > c_{\text{crit}} \rightarrow \text{ CASE 3}$$

Since this is case 3, we must additionally satisfy the regularity condition

$$af\left(\frac{n}{b}\right) \leq kf(n(), k < 1$$

Using $k =$, we can do so:

$$4\left(\frac{n^3}{2^3}\right) \leq kn^3, k = \frac{1}{2}$$

Therefore $T(n)$ is dominated by splitting term n^3, and we obtain asymptotic solution

$$T(n) = \Theta(n^3)$$

Problem 4

$A[1...n]$ is a **sorted** array of **distinct** integers. We want to decide whether there is an index i where $A[i] = i$.

- Describe a divide-and-conquer algorithm that solves this problem.
- Use the Master Theorem to estimate the running time of the algorithm. Your algorithm should run in $O(\log n)$ time

Insights:

1. After passing index i, we no longer need to check for $A[i] = i$ since A is sorted and distinct. Therefore by simply iterating through array, checking at each index, we achieve linear runtime.
2. Since A is sorted, if $A[i]$ is found that is greater than i (i.e. $A[2] = 4$) we can skip all following indices that are less than $A[i]$.

Consider an array B where

$$B[i] = A[i] - i$$

Since $A[i] < A[i + 1]$ because A is sorted and distinct,

$$B[i] = A[i] - i \leq A[i + 1] - 1 - i = B[i + 1]$$

An index where $A[i] = i$ corresponds to one where $B[i] = 0$, therefore we can simply binary search for 0 in B, an $O(\log n)$ operation.

Problem 5

Suppose you are tossing m balls into n bins. Each ball is equally likely to land in each bin, and the ball tosses are independent. What is the expected number of bins that contain exactly k balls? Use indicator random variables to find the solution.

Let indicator variable X_i be defined as

$$X_i = \begin{cases} 1, & \text{if bin}_i \text{ contains } k \text{ balls} \\ 0, & \text{otherwise} \end{cases}$$

Let indicator variable Y equal the number of bins containing k balls:

$$Y = X_1 + X_2 + \ldots + X_n$$

By linearity of expectation,

$$E[Y] = E[X_1] + E[X_2] + \ldots + E[X_n]$$

We now define the probability of a bin i containing k balls. This is a binomial distribution, where $p = \frac{1}{n}$ and $q = \frac{n-1}{n}$. Therefore we need to choose which k of the m balls go into bin i, and then which of the other $n-1$ bins will hold the other $m-k$ balls. Divide this by the total number of ways to distribute m balls among n bins:

$$P(X_i) = \binom{m}{k} \frac{(n-1)^{m-k}}{n^m}$$

The expected value of Y can then be derived:

$$E[Y] = \sum_{i=1}^{n} E[X_i] = \binom{m}{k} \frac{(n-1)^{m-k}}{n^{m-1}}$$

Problem 6

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(\frac{n}{2}) + n^2$. Use the substitution method to verify your answer.

The rate of increase in the number of subproblems in each recursion = 1
The rate of decrease in subproblem size = 2

Therefore in each level of the tree, there is only one node with cost $c\left(\frac{n}{2^i}\right)^2$ at depth $i = 0, 1, \ldots \log n$.
Total cost:

$$T(n) = \sum_{i=0}^{\log n} c\left(\frac{n}{2^i}\right)^2 \leq cn^2 \cdot \sum_{i=0}^{\infty} c\left(\frac{1}{4}\right)^i = O(n^2) \text{ (since sum is independent)}$$

Verify using substitution method:

$$T(n) = T\left(\frac{n}{2}\right) + n^2 \leq d\left(\frac{n}{2}\right)^2 + n^2 = \frac{dn^2}{4} + n^2 \leq dn^2, \qquad \left(d \geq \frac{4}{3}\right)$$

Problem 7

Using Figure 1 as a model (also can be find in the textbook page 161), illustrate the operation of HEAPSORT on the array $A = [5, 13, 12, 25, 71, 37, 27, 9, 22]$.

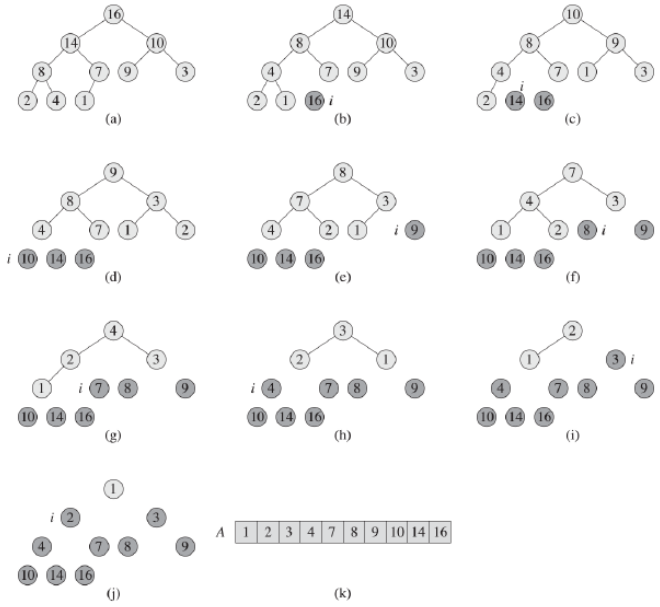
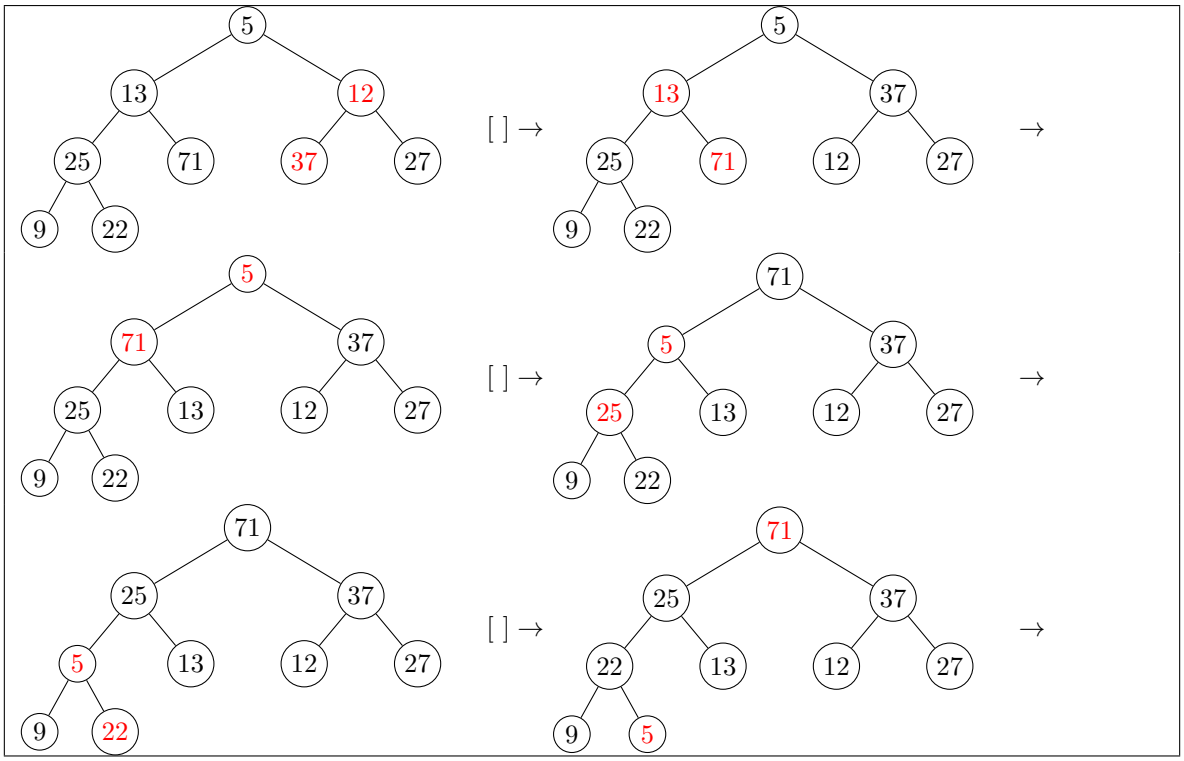
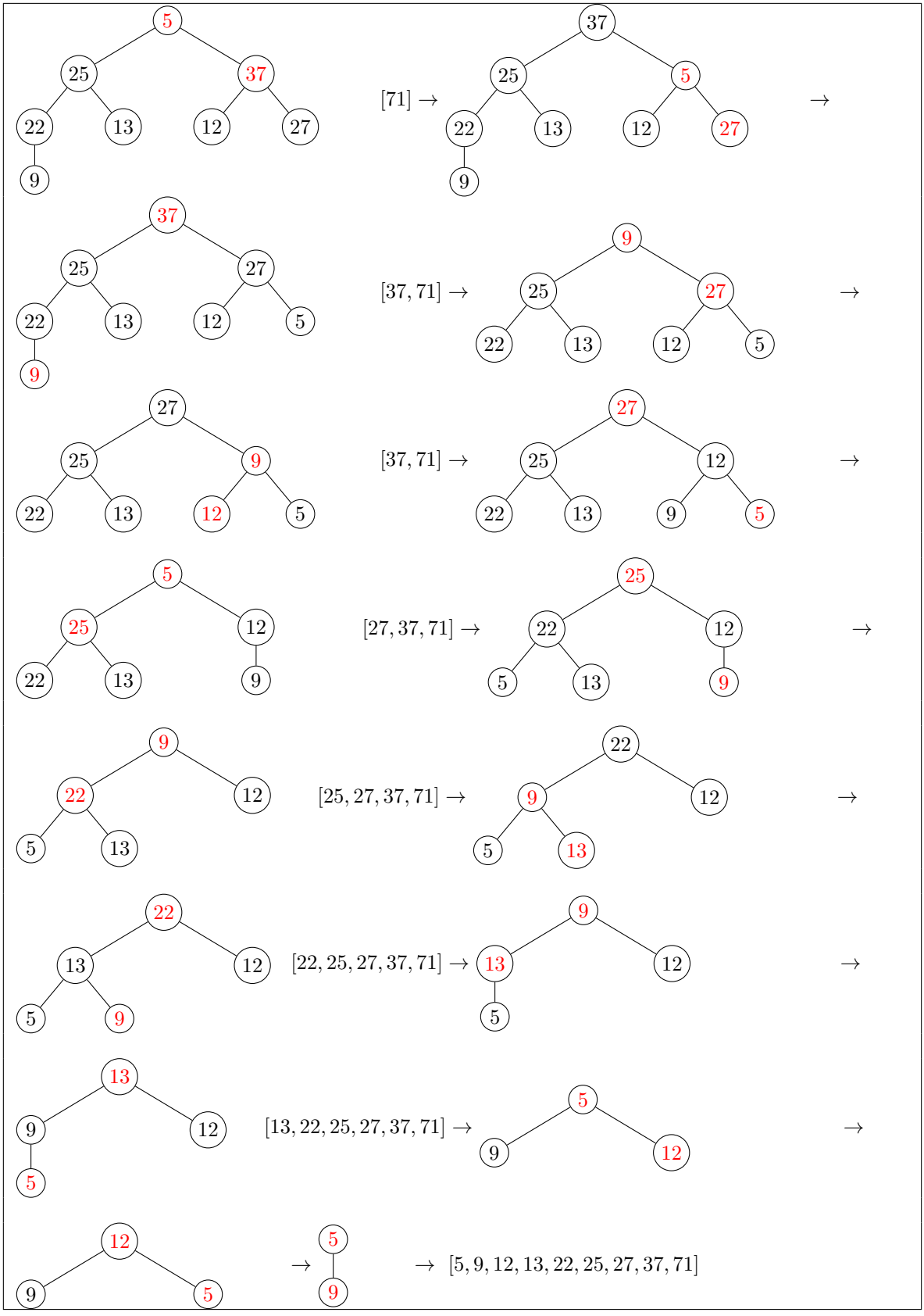


Figure 1: Example for heap sort. The original input array is [16, 14, 10, 8, 7, 9, 3, 2, 4, 1].




[71] →
[37, 71] →
[37, 71] →
[27, 37, 71] →
[25, 27, 37, 71] →
[22, 25, 27, 37, 71] →
[13, 22, 25, 27, 37, 71] →
→
→ [5, 9, 12, 13, 22, 25, 27, 37, 71]

Problem 8

Using QUICKSORT to sort the array $A = [5, 13, 12, 25, 71, 37]$. You just need to show the result after each round. Here is a example for $A = [2, 8, 7, 1, 3, 5, 6, 4]$, suppose you pick the last element in a region as its pivot:

round 1: region=A, result=$[2, 1, 3, 4, 7, 5, 6, 8]$

round 2: region$_1$=$[2, 1, 3]$, region$_2$=$[7, 5, 6, 8]$, result=$[2, 1, 3, 4, 7, 5, 6, 8]$.

round 3: region$_1$=$[2, 1]$, region$_2$=$[7, 5, 6]$, result=$[1, 2, 3, 4, 5, 6, 7, 8]$.

The fig 2 shows the the detail operations during the round 1.

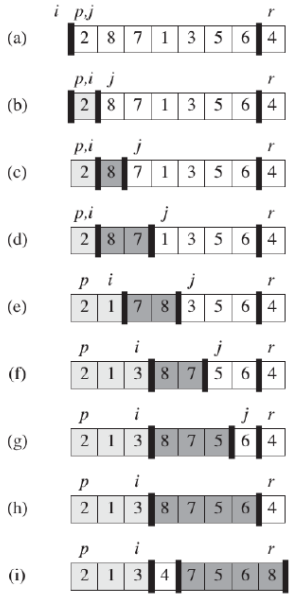


Figure 2: Example of operations in one round of quick sort. $A[r]$ is the pivot element. Elements from $A[p]$ to $A[i]$ are smaller than $A[r]$, those from $A[i + 1]$ to $A[j1]$ are larger than $A[r]$. In your answer you just need to show the step (i) for each round.

Region	**Result**						**Round**
$[5, 13, 12, 25, 71, 37]$	5	13	12	25	71	37	1
$[5, 13, 12], [71, 37]$	5	13	12	**25**	71	37	2
$[5], [13], [71]$	5	**12**	13	**25**	**37**	71	3

Problem 9

The input is two sets $S1$ and $S2$ containing n real numbers in total, and a real number x. You can either show pseudo code or describe solutions in English.

a. Find a $O(n \log n)$ time algorithm that determines whether there exists an element from $S1$ and an element from $S2$ whose sum is exactly x.

Sort $S1$ using heap sort (worst case $O(n \log n4)$) and then for every element $n \in S1$, simply perform binary search for $x - n$ in $S2$ (worst case $O(\log n)$).

b. Suppose now that the two sets are given in sorted order. Find a $O(n)$-time algorithm solving this problem.

Let y be the sum of the smallest element of $S1$ and the largest element of $S2$. If $y = x$, done. If $y > x$, largest element of $S2$ is not part of solution. Recalculate y with second largest, and so on (worst case $O(n)$). If $y < x$, the same argument applies to smallest element of $S1$.

Problem 10

Show that $2n - 1$ comparisons are necessary in the worst case to merge two sorted lists containing n elements each.

Let lists A and B be sorted in ascending order such that $A_1 \leq \ldots \leq A_n$ and $B_1 \leq \ldots \leq B_n$.

Let list C be the correctly merged list A and B, such that

$$C = a_1 \leq b_1 \leq a_2 \leq b_2 \leq \ldots \leq a_n \leq b_n$$

Intuitively, we can see that to create C, there are only two comparisons must be performed for each a_i, since both lists are sorted and of the same length:

1. $a_i \leq a_i$
2. $a_i \leq b_{i+1}$

In the worst case, these must be done for all n elements *except for the last element* a_n, on which only the first one can be done. Therefore, if every element is compared, $2n - 1$ total comparisons must be made. In fact, we also show this to be the lower bound:

1. Suppose there exists an algorithm that runs in $(2n - 2)$ comparisons or less that correctly merges lists X and Y of size n.
2. Let X be a list where $x_i = 2i - 1$ for $i = 1 \ldots n$. Let Y be a list where $y_i = 2i$ for $i = 1 \ldots n$. Let all elements be unique.
3. Run the algorithm on X and Y. Since it takes $(2n - 2)$ comparisons, there must be at least one element x_i in the merged list Z which has not been compared to y_i and y_{i+1}. Therefore if the uncompared y_i or the uncompared y_{i+1} was greater than x_i, they will not have been swapped, and the result will be incorrect.
4. Therefore there is no algorithm that is fully correct that runs in less than $2n - 1$ comparisons.

Problem 11 $\star$

Show an example that COUNTING SORT can be slower than any comparison sorts you have learned. (This problem is for practice, but will not be counted towards the grade)