Algorithms Example Sheet 1: Problems Solution Notes

Exercise 1.P.1 [Asymptotics] For each of the following "=" lines, identify the constants k, k_1, k_2, N as appropriate. For each of the "$\neq$" lines, show they can't possibly exist.

(a) $|\sin(n)| = \mathcal{O}(1)$,

(b) $200 + \sin(n) = \Theta(1)$,

(c) $123456n + 654321 = \Theta(n)$,

(d) $2n - 7 = \mathcal{O}\left(17n^2\right)$,

(e) $\log(n) = \mathcal{O}(n)$,

(f) $\log(n) \neq \Theta(n)$,

(g) $n^{100} = \mathcal{O}(2^n)$,

(h) $1 + 100/n = \Theta(1)$,

(i) (+++) $|\sin(n)| \neq \Theta(1)$.

(a) We know that $|\sin(n)| \leq 1 < 2$ for all n, so By choosing $N = 0$ and $k = 1$, we get $|\sin(n)| = \mathcal{O}(1)$.

(b) Again using the fact that $|\sin(n)| < 2 \Leftrightarrow -2 < \sin(n) < 2$ for all n, we have that $198 < 200 + \sin(n) < 202$ for all n so $\sin(n) \in \mathcal{O}(1)$.

(c) Let $f(n) = 123456n + 654321$. We will start by showing that $f(n) = \Omega(n)$. This holds for $k = 1$ and $N = 0$, since for every $n > N$, $123456n + 654321 > n \Leftrightarrow 123455n + 654321 > 0$ which is true. To show that $f(n) = \mathcal{O}(n)$, we have to find kn such that it dominates $f(n)$ By choosing $k = 123457$, we have $123457n > 123456n + 654321 \Leftrightarrow n > 654321$ which is true for all $n > 654321$. So $f(n) = \mathcal{O}(n)$. Hence, $f(n) = \Theta(n)$.

Note: Here, any $k > 123456$ would work.

(d) Let's see when $2n - 7 \leq 17n^2$ holds. By re-arranging, we get $0 \leq 17n^2 - 2n + 7$, where the RHS is a quadratic with discriminant $\Delta = 4 - 4 \cdot 17 \cdot 7 < 0$. Hence, it has no roots and this means that it is always positive. So, the desired inequality holds for all n and for $k = 1$.

(e) Consider the function $d(n) = n - \log(n)$, then $d'(n) = 1 - \frac{1}{n}$ and $d''(n) = \frac{1}{n^2} > 0$. So it attains a minimum at $n = 1$, which means that $d(n) \geq d(1) = 1 \Leftrightarrow n \geq \log(n) + 1$. So, by choosing $k = 1$ and $N = 1$, we get that $n > \log n$ for $n > N$.

(f) To show that $\log(n) \neq \Theta(n)$, we need to show that $\log(n) \neq \mathcal{O}(n)$ or $\log(n) \neq \Omega(n)$. In the previous step, we showed that $\log(n) \neq \mathcal{O}(n)$, so we will prove that $\log(n) \neq \Omega(n)$.

Recall the definition for big-Ω,

$$f(n) \in \Omega(g(n)) \Leftrightarrow \exists k, N > 0. \forall n > N. 0 \leq k \cdot g(n) \leq f(n).$$

To show that $\log(n) \neq \Omega(n)$, we need to show the negation of this, so

$$f(n) \notin \Omega(g(n)) \Leftrightarrow \forall k, N > 0. \exists n > N. \neg(0 \leq k \cdot g(n) \leq f(n)).$$

Let $k > 0$ and $N > 0$ be arbitrary. We define $d(n) = kn - \log n$, so $d'(n) = k - \frac{1}{n}$ and $d''(n) = \frac{1}{n^2} > 0$. Hence, it has a minimum at $n = \frac{1}{k}$, i.e.

$$d(n) \geq d(1/k) \Leftrightarrow kn - \log n \geq 1 - \log(n/k) \Leftrightarrow kn \geq 1 - \log(n/k) + \log(n) = kn \geq 1 - \log(k).$$

Since the LHS depends on n and the RHS does not, for any $n \geq (1 - \log(k))/k$, the inequality holds. So, $k \cdot g(n) \leq f(n)$ cannot hold. Hence, $\log(n) \neq \Omega(n)$ and in particular $\log(n) = o(n)$.

(g) Note that $n^{100} = 2^{100 \log_2 n}$, but we have shown that $100 \log_2 n = on$, hence for any $k > 0$ (choose $k = 1$ here), and for some $N > 0$, for all $n > N$, $100 \log_2 n < n$. Since exponentiation is increasing, $2^{100 \log_2 n} < 2^n$.

(h) $f(n) = 1 + 100/n \geq 1$ for any n, so $f(n) \in \Omega(1)$ (for $k = 1$, $N = 1$). For the upper bound, for $n > 100$, $100/n < 1$, so $f(n) = 1 + 100/n < 1 + 1 = 2$. Hence, for $k = 1$ and $N = 100$, we get $f(n) = \mathcal{O}(1)$.

(i) This is a harder question that the lecturer probability intended to be. The reason why the equality does not hold is the lower bound. For no k can it hold that $k \leq \sin(n)$ for all $n > N$. The reason is that as $n \to \infty$, $\sin(n)$ goes arbitrarily close to 0, so it will get at $|\sin(n)| < k$. This can be proven rigorously using the properties of rational approximations (see for example this discussion).

Exercise 1.P.2 [More Asymptotics]

(a) Show that for $a > b > 0$, $n^b \in \mathcal{O}(n^a)$.

(b) Show that for $a > b > 0$, $b^n \in \mathcal{O}(a^n)$.

(c) Compare $n \cdot 2^n$ and 3^n.

(d) Compare n and $(\log n)^{10}$.

(e) Compare n and $\exp((\log n)^{1/2})$.

(f) Compare n^3 and $\exp((\log n)^{9/2})$.

(g) Compare $f(n) = \sum_{i=1}^{n} i^5$ and 2^n. (*Hint:* Use the Discrete Maths exercise for the sum of k-th powers)

(h) Sort the functions in increasing order of asymptotic complexity: $f_1(n) = n^{0.999} \log n$, $f_2(n) = 10^9 \cdot n$, $f_3(n) = 1.00001^n$ and $f_4(n) = n^{1.01}$.

(i) Sort the functions in increasing order of asymptotic complexity: $f_1(n) = 2^{2^{5000}}$, $f_2(n) = 2^{1000n}$, $f_3(n) = \binom{n}{2}$ and $f_4(n) = n\sqrt{n}$.

(j) Sort the functions in increasing order of asymptotic complexity: $f_1(n) = n^{\sqrt{n}}$, $f_2(n) = 2^n$, $f_3(n) = n^{10} \cdot 2^{n/2}$ and $f_4(n) = \sum_{i=1}^{n} i$.

(k) (+) Attempt **[2016P1Q8 (a)]**.

(l) (optional ++) Problems 3.2, 3.3, 3.4, 3.6 from CLRS.

(a) We need to find N such that for every N, such that $n^a > n^b \Leftrightarrow n^{b-a} > 1$. This holds for every $n > 1$ (since $b > a$). So, $k = 1$, gives the answer.

(b) Similarly, $a^n > b^n \Leftrightarrow \left(\frac{a}{b}\right)^n > 1$. Since $a > b$, this holds for $n > 0$ (we can prove it by induction since $\left(\frac{a}{b}\right)^{n+1} = \left(\frac{a}{b}\right)^n \cdot \frac{a}{b} > 1$, because $\frac{a}{b} > 1$ and $\left(\frac{a}{b}\right)^n$).

(c) One way of approach this to express both in the base base $n \cdot 2^n = 2^{\log_2 n + n}$ and $3^n = 2^{(\log_2 3)n}$. From this we have that $(\log_2 3)n = (1.58..)n > 1.1n > \log_2 n + n$. Hence, $3^n = 2^{1.58..n} > 2^{1.1n} > n \cdot 2^n$, so by the previous question it is that $n \cdot 2^n \in o(3^n)$.

(d) $n = e^{\log n}$ and $(\log n)^{10} = e^{10 \log\log n}$. By comparing exponents $e^{\log n} \in o(e^{10 \log\log n})$.

(e) $n = e^{\log n}$ and $e^{(\log n)^{1/2}}$. By comparing exponents $e^{(\log n)^{1/2}} \in o(e^{\log n})$.

(f) $n^3 = e^{3 \log n}$ and $\exp((\log n)^{9/2})$. By comparing exponents $\exp((\log n)^{9/2}) \in o(n^3)$.

(g) **(Solution 1)** In Discrete mathematics you showed that $\sum_{i=1}^{n} i^k$ can be computed by a polynomial of order $k + 1$ for $k \in \mathbb{N}$. Hence, $\sum_{i=1}^{n} i^5 = \Theta(n^6)$.

(Solution 2) A different approach is to lower bound the sum by breaking it in half, $\sum_{i=1}^{n} i^5 \geq \sum_{i=n/2}^{n} i^5 \geq \sum_{i=n/2}^{n} (n/2)^5 = n/2 \cdot (n/2)^5 = n^6/2^6 = \Omega(n^6)$.

An upper bound can be obtained by $\sum_{i=1}^{n} i^5 \leq \sum_{i=1}^{n} n^5 = n \cdot n^5 = n^6 = \mathcal{O}\left(n^6\right)$. Hence, $\sum_{i=1}^{n} i^5 = \Theta(n^6)$.

(h) $f_1(n) = n^{0.999} \log n$, $f_2(n) = 10^9 \cdot n$, $f_4(n) = n^{1.01}$, $f_3(n) = 1.00001^n$.

(i) $f_1(n) = 2^{2^{5000}}$, $f_4(n) = n\sqrt{n} = \Theta(n^{1/5})$, $f_3(n) = \binom{n}{2}$ $(= n(n-1)/2 = \Theta(n^2))$, $f_2(n) = 2^{1000n}$.

(j) $f_4(n) = \sum_{i=1}^{n} i$, $f_1(n) = n^{\sqrt{n}}$ $(= e^{(\log n)^{1/5}})$, $f_3(n) = n^{10} \cdot 2^{n/2}$, $f_2(n) = 2^n$.

Exercise 1.P.3 [Matrix exponentiation (++)] (Only attempt this if you know about matrices). Consider two 2×2 matrices A and B.

(a) Implement an OCaml function that takes the elements of A and B and returns the matrix product of these two.

$$A \cdot B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

(b) Modify the `power` function (you learnt in FoCS) to compute the power of a matrix.

(c) What is the time complexity of your algorithm?

(d) Implement an OCaml function that takes a 2×2 matrix A and a 2-element vector v and computes $A \cdot v$.

(e) The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$ (for $n > 1$) with $F_0 = 0$ and $F_1 = 1$. Show that for $n > 0$

$$\begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix}$$

(f) Using the functions you developed above, show how to compute the n-th Fibonacci number in $\mathcal{O}(\log n)$ time.

(a)
```
let mat_mult [[a11;a12];[a21;a22]] [[b11;b12];[b21;b22]] =
   [[a11 * b11 + a12 * b21; a11 * b12 + a12 * b22];
    [a21 * b11 + a22 * b21; a21 * b12 + a22 * b22]];;
```

(b) The following code works because matrix multiplication is associative:

```
let rec fast_mat_pow m n =
  if n = 1 then m
  else if n mod 2 = 1 then mat_mult m (fast_mat_pow (mat_mult m m) (n / 2))
  else fast_mat_pow (mat_mult m m) (n / 2);;
```

(c) The time complexity of the algorithm is $\mathcal{O}(\log n)$ since the 2×2 matrix multiplications take constant time.

```
let mat_vec_mult ((a11,a12),(a21,a22)) (v1, v2) =
   (a11 * v1 + a12 * v2, a21 * v1 + a22 * v2);;
```

(d)

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} F_n + F_{n-1} \\ F_n + 0 \end{bmatrix} = \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

(e)
```
let fib n =
  let _, ans = mat_vec_mult (fast_mat_pow ((1,1), (1, 0)) n) (1, 0) in
    ans;;
```

We will prove that this works by induction. For the base case $n = 1$, we have

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

For the induction step, assume that it is true for $n = k$, then for $n = k + 1$,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{k+1} \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \left(\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k \begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} \right) = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} F_k \\ F_{k-1} \end{bmatrix} = \begin{bmatrix} F_{k+1} \\ F_k \end{bmatrix}$$

Exercise 1.P.4 [Max/Min]

(a) Design an algorithm for finding the maximum in an array of n elements.

(b) What is the time complexity of your algorithm? Prove a corresponding lower bound.

(c) Repeat the first two parts for the minimum.

(d) Given an array of n elements, find the pair of elements that has the largest difference. You can submit your solution on **[SPOJ EIUMADIS]**.

(e) (+) Think about how you would solve **[SPOJ DIFERENC]** in $\mathcal{O}\left(N^2\right)$ time (no need to implement it).

(a) The following code finds the maximum in an array:

```python
def find_max(a):
    # Maybe throw an error if the array has no elements.
    cur_max = a[0]
    for i in range(1, len(a)):
        if a[i] > cur_max:
            cur_max = a[i]
    return cur_max


print(find_max([1, 10, 23, 8, 17]) == 23)
print(find_max([1, 10, 3, 8, 17]) == 17)
print(find_max([5]) == 5)
```

(b) This implementation requires $n-1$ comparisons, where n is the number of items in the array.

(c) Consider the graph formed by the pairs of comparison. Then if there are fewer than $n-1$ comparison pairs, it means that the graph is not connected (since the graph is the minimal connected graph (see Supervision 5)). So, there are at least two components. If the algorithm A outputs a maximum from one component, then we can argue that the other component had the maximum (since they are in different components there will be no contradicting comparisons). So, the algorithm will output the wrong answer.

(d) The minimum is exactly the same for $>$ instead of $<$.

(e) We can just the difference between the maximum and the minimum element, for any i, j, the difference $a_j - a_i \leq (\max a_i) - a_i \leq (\max a_i) - (\min a_j)$.

Exercise 1.P.5 [Second max]

(a) Describe an algorithm for finding the second largest element in an array of n elements.

(b) What is the time complexity of your algorithm? Prove an asymptotic lower bound (i.e. we do not care about multiplicative constants).

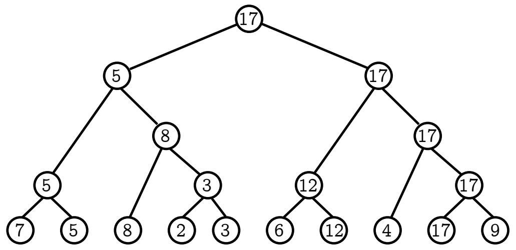
(c) (optional) See **[2007P10Q10 (d)]**.

(a) Here is an algorithm that requires at most $n-1+n-2 = 2n-3$ comparisons:

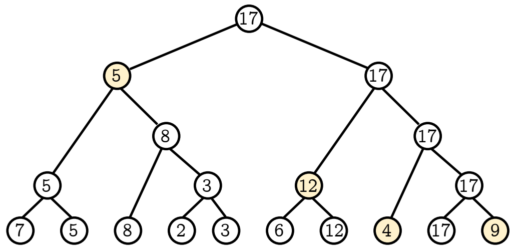
```python
def find_second_max(a):
    # Maybe throw an error if the array has no elements.
    cur_max = max(a[0], a[1])
    second_max = min(a[0], a[1])
    for i in range(2, len(a)):
        if a[i] > second_max:
            second_max = a[i]
            if second_max > cur_max:
                tmp = cur_max
                cur_max = second_max
                second_max = tmp
    return cur_max, second_max


print(find_second_max([1, 10, 23, 8, 17]) == (23, 17))
print(find_second_max([1, 10, 3, 8, 17]) == (17, 10))
print(find_second_max([1, 17, 3, 8, 17]) == (17, 17))
print(find_second_max([5, 7]) == (7, 5))
print(find_second_max([5, 7]) == (7, 5))
```

We can design a more involved algorithm to improve the worst-case number of comparisons. Note that the worst-case is when the new value we are comparing against is always larger than our current maximum. The problem is that we could compare the candidate maximum to all $n-1$ elements, so we need to find the maximum among these.

One idea is to reduce the number of items to which we compare the maximum. Consider an elimination tournament (similar to the ones found in sports). In the each round, we pair up the remaining elements, compare them and eliminate the ones that were smaller. By repeating this procedure we create the tournament tree of height $\lceil \log_2 n \rceil$. For example, if the input sequence was `[7; 5; 8; 2; 3; 6; 12; 4; 17; 9]` one possible tournament tree is the following:



Each comparison eliminates exactly one element, and exactly $n-1$ elements are eliminated, so Notice that all values that where compared to the maximum are candidates for being the second maximum. All values that were not compared are not candidates since they are smaller than one value that is smaller (or equal) than the maximum. So it suffices to compare the remaining $\lceil \log_2 n \rceil$ of those that can be done using $\lceil \log_2 n \rceil - 1$ comparisons. In the example above, this corresponds to finding the maximum among the highlighted $\lfloor \log_2 n \rfloor = 4$ values:



The code below shows one possible implementation for this:

```
type 'a btree = Br of 'a * 'a btree * 'a btree | Lf;;
type 'a option = Some of 'a | None;;

(* Common auxiliary functions BEGIN *)
let rec take i = function
| [] -> []
| x::xs ->
   if i > 0 then x :: take (i - 1) xs
   else [];;

let rec drop i = function
| [] -> []
| x::xs ->
   if i > 0 then drop (i-1) xs
   else x::xs;;

let is_geq = function
(Br(a, _, _), Br(b, _, _)) -> a >= b
| (Lf, Br(b, _, _)) -> false
| (Br(a, _, _), Lf) -> true;;

let get_val (Br(a, _, _)) = a;;
let max_option = function
  (None, Some b) -> Some b
| (Some a, None) -> Some a
| (Some a, Some b) -> Some (if a >= b then a else b)
| _ -> None;;
(* Common auxiliary functions END *)

(* Constructs a tournament tree for a given list of integers.
   It also places the smaller value on the right, so we can find
   it efficiently when searching for the second max value. *)
let rec find_tournament_tree = function
[] -> Lf
| [x] -> Br(x, Lf, Lf)
| xs -> let left = find_tournament_tree(take((List.length xs) / 2) xs) in
        let right = find_tournament_tree(drop((List.length xs) / 2) xs)
        in
           if is_geq(left, right) then Br(get_val(left), left, right)
           else Br(get_val(right), right, left);;

(* Find the maximum values among the right branches in the path of the maximum.
    *)
let rec find_second_best = function
  (Br(_, _, Lf)) -> None
| (Br(_, ell, Br(x, _, _))) -> max_option (find_second_best ell, Some x);;


let tournament_tree = find_tournament_tree [7; 5; 8; 2; 3; 6; 12; 4; 17; 9];;
find_second_best tournament_tree;;
```

Further reading: It turns out there is a matching lower bound for this (see here or here).

(b) Complexities were given above.

Binary search

Exercise 1.P.6 [Binary Search on the answer] Read the statement of **[SPOJ AGGRCOW]**

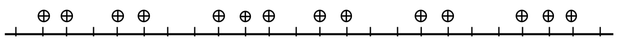
(a) Try to solve this problem first: Given that all stalls are separated by a distance of at least d, determine if it is possible to place the cows.

(b) Let $f(d)$ (with $f : \mathbb{N} \to \{0, 1\}$) be the answer to the above problem. Argue that this function is monotonic.

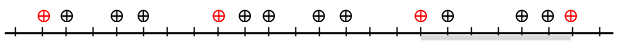
(c) Use binary search to solve the original problem.

(d) (optional) Implement the solution.

(a) *What is the problem asking?* We are given N possible positions to place cows so as to maximise the minimum distance. For example, if the following cow positions were allowed and there were 4 cows:



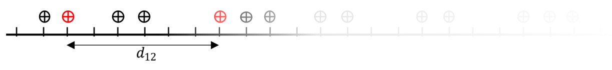
then the best possible placement is shown below (red colour corresponds to a cow) and the minimum distance between two cows is shown in gray:



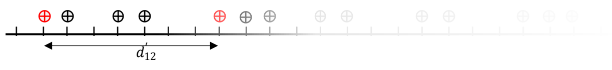
Let's sort the candidate positions. When searching for a way to place the cows so that they are all at distance at least d from one another, we can make the the following observation:

Observation 1: If there is a way of placing the cows, then there will be a way where we use the first (and last) candidate position.

It is simple to prove this observation. Consider a configuration where every cow is placed at a distance at least d from each other; and assume the first cow is not used:



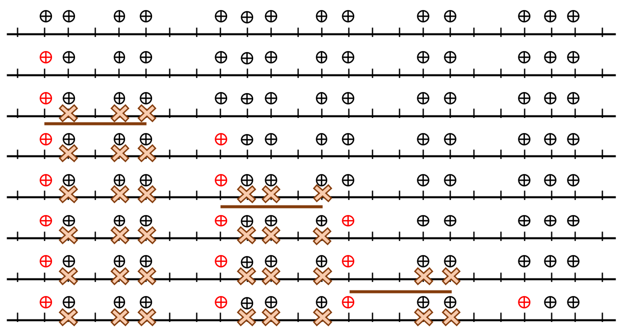
Then, consider the leftmost cow. By moving the cow to the first stall, we obtain a valid configuration since the distance of the first to the second cow (the only distance that is affected) increases:



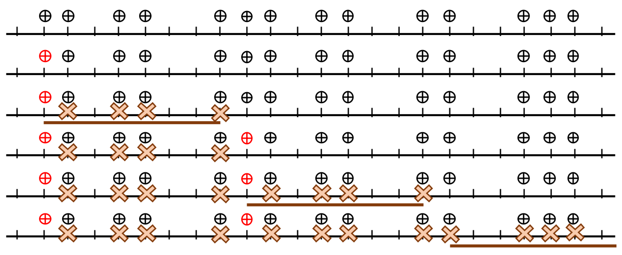
This observation leads to the following algorithm:

1. Sort the candidate stall positions.
2. Assign the first cow to the leftmost stall.
3. Find the leftmost stall that is at least d units away from the last placed cow. Place the next cow there.
4. Repeat step **??** until all cows have been placed or no stalls available.
5. If there are no cows left, return `TRUE`. Otherwise, return `FALSE`.

Below is a visualisation of the steps for checking placement with $d = 5$, where it is possible to place the 4 cows:



Below is a visualisation of the steps for checking placement with $d = 8$, where it is not possible to place the cows:



(b) This function is monotonic because assuming that $f(d) = 1$, which means that it is possible to place N cows with a maximum distance of d, then the same configuration works for any distance $d' \geq d$. This means that we can binary search to find the smallest possible d such that $f(d) = 1$.

(c) The following code passes the testcases on SPOJ:

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>

using namespace std;

const size_t MAXN = 100000;
```

```
int N, C;
int A[MAXN];

int F(int d){
   // prev :: location of previously placed cow (initially -INF)
   // rem..:: how many cows we should still place.
   int prev = -A[N-1], remaining = C;
   for (int i = 0; i!=N; i++){
      // If the next candidate stall is further than d units
      // from the previously placed cow, then we assign a cow.
      if (A[i] - prev >= d) {
         prev = A[i];
         remaining--;
      }
      // If there are no cows remaining, we are done.
      if (remaining == 0) return true;
   }
   // If there are still cows, then it is not possible.
   return false;
}

int main(){
   int T;
   scanf("%d", &T);
   while(T--) {
   scanf("%d%d", &N, &C);
   for (int i = 0; i<N; i++) scanf("%d", &A[i]);
   // Sort the stalls.
   sort(A, A+N);
   // Binary search.
   int st = 0, en = A[N-1] - A[0], mn;
   while(st < en){
      mn = (st+en+1)/2;
      if (F(mn) == true) st = mn;
      else en = mn - 1;
   }
   printf ("%d\n", st);
   }
   return 0;
}
```

Note: We can either run the binary search on the interval $[s_0, s_{|s|-1}]$ (i.e. the distance between the first and the last stall) or only at the $N-1$ distances between adjacent stalls (*why?*). The time complexity of the first implementation is $\mathcal{O}(S \log D)$ while the second one has $\mathcal{O}(S \log S)$. If D is very large the second one is preferred.

Exercise 1.P.7 [Binary Search on the answer] Try to solve the problem **[SPOJ BOOKS1]** using the same technique as in **[SPOJ AGGRCOW]**.

Again, given the maximum sum we can check if it is possible to split the books into consecutive subarrays where the maximum sum of a subarray is $\leq d$. We do this by aggregating as many books as long as the sum is $\leq d$.
Again, this binary function f is monotonic. So we can use binary search to find the minimum maximum sum. Hence, we can solve this problem in $\mathcal{O}(N \log U)$ time, where U is the total sum of the pages in the books.

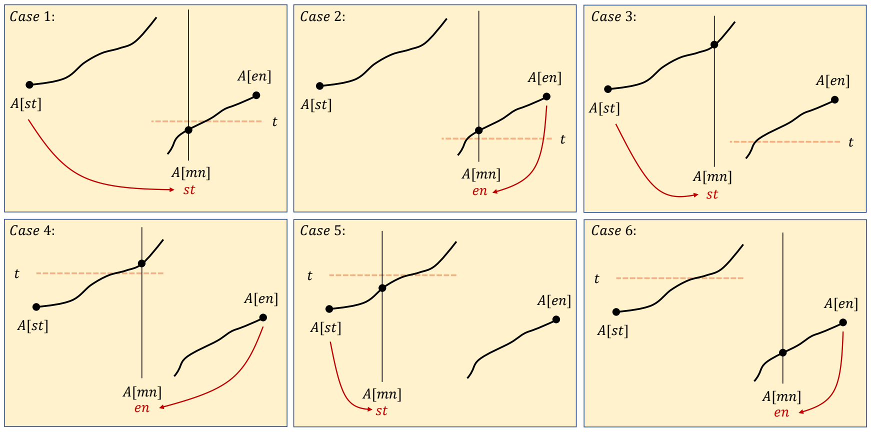
Exercise 1.P.8 [Binary Search on rotated array] Try the following common interview question: **[LeetCode 81]**.

There are at least two ways to solve it. One is to first use binary search to find the turning point k of the array. Then use normal binary search on the array $A'[i] = A[(i+k) \mod |A|]$.

The second is to use the following modified binary search:

```
class Solution {
    public int search(int[] nums, int target) {
        int st = 0, en = nums.length - 1;
        while (st < en) {
            int mn = (st + en)/2;
            if (nums[st] <= nums[en]) {
                // We are doing normal binary search.
                if (nums[mn] == target) return mn;
                else if (nums[mn] < target) st = mn + 1;
                else en = mn - 1;
            } else {
                // We are still searching for the valid endpoint.
                if (nums[mn] == target) return mn;
                else if (nums[en] >= target) {
                    if (nums[mn] <= nums[en] && nums[mn] > target) en = mn - 1;
                    else st = mn + 1;
                } else {
                    if (nums[mn] >= nums[st] && nums[mn] < target) st = mn + 1;
                    else en = mn - 1;
                }
            }
        }
        return nums[st] == target ? st : -1;
    }
}
```

It handles the following 6 cases:



Question: *What happens if we allow for duplicates?* The problem is harder and binary search cannot be applied. Consider the case of $[1; 1; 1; 1; 1; 1; 1; 1; 0; 1; 1; 1; 1; 1]$. This is a rotated sorted array, but finding 0 will take linear time. (Can you formally prove the lower bound here?)

Exercise 1.P.9 [Removing duplicates] Design an algorithm to remove all duplicate elements from an array. For example, given $[1; 2; 6; 2; 1; 3; 2]$, it should return $[1; 6; 3; 2]$ (in any order). What is the worst-case time complexity of your algorithm?

(Brute force) One way of doing this is for each element go through all other elements of the array and remove the ones that are equal to it. For the i-th element we need to check $n - i$ elements, so this gives a total of $\Theta(n^2)$.
(Efficient) A different way of doing this is to first sort the array. Then equal elements appear in neighbouring positions. We can keep only the first ones from each group. This can be done in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(1)$ extra space.
The following code implements the algorithm for **[LeetCode 26]** which provides a sorted input array:

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;
        int prev = nums[0];
        int j = 1;
        for (int i = 1; i < nums.length; ++i) {
            if (nums[i] == prev) continue;
            prev = nums[i];
            nums[j] = nums[i];
            ++j;
        }
        return j;
    }
}
```

Question: *How would you only retain the elements that appear at least twice?* The code below implements this for **[LeetCode 80]**

```
class Solution {
   public int removeDuplicates(int[] nums) {
     if (nums.length == 0) return 0;
     int prev = nums[0];
     int cnt = 1;
     int j = 1;
      for (int i = 1; i < nums.length; ++i) {
         if (nums[i] == prev) {
            if (cnt == 2) continue;
            nums[j] = nums[i];
            ++j;
            ++cnt;
         } else {
            prev = nums[i];
            nums[j] = nums[i];
            ++j;
            cnt = 1;
         }
      }
      return j;
   }
}
```

Exercise 1.P.10 [Intersection of two arrays] Describe an algorithm to compute the intersection of two arrays. For example, given $[1;2;6;2;1;3;2]$ and $[6;1;7;2;2;4;]$, it should return $[1;6;2;2;]$ (in any order). What is the worst-case time complexity of your algorithm?
You can test your implementation on **[LeetCode 350]**.

Sort both of the arrays and then iterate using two pointers (in a way similar to the `merge` part of mergesort). Whenever $A[i_1] = B[i_2]$, we add $A[i_1]$ to the output and increment i_1 and i_2. If $A[i_1] > B[i_2]$, then increment i_2, else increment i_1.
The following code implements this:

```
class Solution {
   public int[] intersect(int[] nums1, int[] nums2) {
      Arrays.sort(nums1);
      Arrays.sort(nums2);

      int i1 = 0, i2 = 0, j = 0;
      int[] out = new int[nums1.length + nums2.length];
      while (i1 < nums1.length && i2 <nums2.length) {
         if (nums1[i1] == nums2[i2]) {
            out[j] = nums2[i2];
            ++j;
            ++i1;
```

```
                ++i2;
            } else if (nums1[i1] < nums2[i2]) {
                ++i1;
            } else {
                ++i2;
            }
        }
        return Arrays.copyOfRange(out, 0, j);
    }
}
```

Exercise 1.P.11 [Union of two arrays] Design an algorithm to compute the union of two arrays. For example, given [1; 2; 6; 2; 1; 3; 2] and [6; 1; 7; 2; 2; 4;], it should return [1; 6; 7; 2; 4; 3] (in any order). What is the worst-case time complexity of your algorithm?
You can test your implementation on **[GeeksForGeeks Union of Arrays]**.

```
class Solution{
    public static int doUnion(int a[], int n, int b[], int m)
    {
        Arrays.sort(a, 0, n);
        Arrays.sort(b, 0, m);

        int i1 = 0, i2 = 0;
        ArrayList<Integer> union = new ArrayList<>();
        while (i1 < n && i2 < m) {
            if (a[i1] == b[i2]) {
                int cur = a[i1];
                union.add(cur);
                // Remove all elements equal to the currently
                // added element from each array.
                while (i1 < n && cur == a[i1]) ++i1;
                while (i2 < m && cur == b[i2]) ++i2;
            } else if (a[i1] < b[i2]) {
                int cur = a[i1];
                union.add(cur);
                while (i1 < n && cur == a[i1]) ++i1;
            } else {
                int cur = b[i2];
                union.add(cur);
                while (i2 < m && cur == b[i2]) ++i2;
            }
        }
        while (i1 < n) {
            int cur = a[i1];
            union.add(cur);
            while (i1 < n && cur == a[i1]) ++i1;
        }
        while (i2 < m) {
            int cur = b[i2];
            union.add(cur);
            while (i2 < m && cur == b[i2]) ++i2;
        }
        return union.size();
    }
}
```

Exercise 1.P.12 (optional) Attempt **[2010P1Q5 (c)]**.

See official solution notes

Exercise 1.P.13 (optional) Attempt **[2018P1Q7 (a)]**.

See official solution notes

Exercise 1.P.14 [Most frequent elements] Describe an algorithm to find the most frequent elements in an array. For example, given $[1; 2; 1; 1; 3; 3; 2; 4; 3]$, it returns $[1; 3]$ since they both occur 3 times.

(Brute force): For each element in the array, iterate over the array and count the number of equal elements, keeping a list of those with maximum frequency.
(Efficient Solution): By sorting the elements, the equal elements appear next to each other, so it is easy to count the number of times each element occurs. The sorting takes $\mathcal{O}(n \log n)$ time.

```java
import java.util.ArrayList;
import java.util.Arrays;

public class MostFrequentElement {

   public static ArrayList<Integer> mostFrequentElements(int[] a) {
      ArrayList<Integer> out = new ArrayList<>();
      if (a.length == 0) return out;
      Arrays.sort(a);
      int prev = a[0];
      int cnt = 1, max_cnt = 1;
      out.add(prev);
      for (int i = 1; i < a.length; ++i) {
         if (a[i] == prev) {
            ++cnt;
         } else {
            prev = a[i];
            cnt = 1;
         }
         // Update the maximum count and update the
         // collection of items with maximum frequency.
         if (cnt == max_cnt) {
            out.add(a[i]);
         } else if (cnt > max_cnt) {
            out.clear();
            out.add(a[i]);
            max_cnt = cnt;
         }
      }
      return out;
   }

   public static void main(String[] args) {
      int[] a = {1, 2, 1, 1, 3, 3, 2, 4, 3};
      ArrayList<Integer> out = mostFrequentElements(a);
      for (int x : out) {
         System.out.println(x);
      }
   }

}
```

Exercise 1.P.15 (optional) Attempt **[2011P1Q5]**.

See official solution notes

Chapter 2.11/2.12

Exercise 1.P.16 Attempt **[2006P1Q4 (c)]**.

See official solution notes

Exercise 1.P.17 [Finding top k elements]

(a) You are given an array containing pairs of `(friendID, time spent last week)`. How you would find the IDs of your friends with whom you spent most time together during the last week? What is the time complexity of your algorithm?

(b) (optional) See **[2007P10Q10 (b)]**.

(a) We can find the k-th entry x_k in $\mathcal{O}(n)$ time using quickselect. Hence, using a single iteration over the array we keep the k elements that are $\geq x_k$.

Note: In case of a tie as the k-th element we might need to drop a few extra elements.

(b) This is different from the first part because we need the elements in sorted order.

We could sort the k elements returned from the first part using any comparison-based sorting method in $\mathcal{O}(k \log k)$ time (so $\mathcal{O}(n + k \log k)$ total).

We can use a max heap and do k extractions, giving a total $\mathcal{O}(n + k \log n)$ time, since heapify takes $\mathcal{O}(n)$ time. However, this is slightly slower.

Chapter 2.14

Exercise 1.P.18 [All items] You are given n boxes each one coloured with one of the M available colours. Describe an algorithm that checks if there is at least one box from each possible color.

Create an array C with M positions. Iterate over the n boxes and set $C[A[i]] = 1$. Then we can just check if all M positions are set to 1. This takes $\Theta(n + U)$ time.

Exercise 1.P.19 [Union/Intersection using counting sort] Modify your solution for finding the union and intersection of two arrays to use counting sort. What is the time complexity of this approach?

Given two arrays A and B, we can create the counting arrays C_A and C_B.
The intersection of A and B has counting array $C_I[x] = \min(C_A[x], C_B[x])$ (e.g. if A has 3 occurrences of x and B has 7 occurrences of x, then their intersection has 3).
The intersection of A and B has counting array $C_U[x] = \mathbf{1}_{C_A[x] \geq 1 \vee C_B[x] \geq 1}$ (a formal way of writing that $C_U[x]$ is 1 if either is 1).
Given the counting array, in either case we can construct the output array by iterating through the values. Hence, the total complexity is $\Theta(|A| + |B| + U_A + U_B)$ where U_X denotes the maximum element in table X.

Exercise 1.P.20 [Does pair exist with given sum] Given an array A of integers and an integer K, determine if there are two elements $A[i]$ and $A[j]$ such that $A[i] + A[j] = k$. What is the time complexity of your algorithm?

Insert all the elements in A in a counting table C (sometimes also called a hash table). Then for each element $A[i]$ check if $C[k - A[i]] > 0$. If it is true, then it means that there is some $A[j] = k - A[i]$. So, $A[i] + A[j] = k$. Note that if $A[i] = k/2$, then we need to check that $C[k - A[i]] > 1$, so that we don't pair it with itself. This takes $\Theta(n + U)$ time.
Note: Instead of a hash table we could have used a balanced binary search tree and the time complexity would have been $\Theta(n \log n)$. An alternative solution is to sort the array and keep two pointers one left ℓ and one right r. If $A[\ell] + A[r] > k$, then move r one place to the left, otherwise move ℓ one place to the right. So (requires further justification), at some point these pointers should either cross each other or find a pair with sum k.

Exercise 1.P.21 [Median using counting sort] How can you use counting sort to find the median of an array?

We insert the elements of A into the counting array and then go from 0 to U keeping track of the number z elements we have seen so far. When $z = n/2$, we have found the median.
The same approach works for finding the k-th element in time $\Theta(n + U)$.

Exercise 1.P.22 [Bucket Sort] (optional) Attempt **[2018P1Q7 (b)]**.

See official solution notes

Problems using sorting techniques

Exercise 1.P.23 [Union of intervals] You are given n intervals $[a_i, b_i]$ (with $a_i \neq b_i \in \mathbb{N}$). The area covered by the intervals is that of covered by the union of intervals. For example, the intervals

$$[[1, 5]; [2, 6]; [4, 9]; [14, 16]]$$

cover a total area of 12.

(a) Describe an algorithm for finding the area covered by the intervals. What is the time complexity of your algorithm?

(b) (optional) How would you modify your solution if $a_i, b_i \in \mathbb{Z}$?

(c) (optional) What if $a_i, b_i \in \mathbb{R}$?

If you want you can submit an implementation to **[LeetCode 56]**

(Solution 1) Let U be the largest coordinate. We can create a boolean array of length $U + 1$, where $C[i]$ represents whether the interval $[i, i + 1)$ is covered. We can fill-in this table by iterating over the intervals and for each interval iterating over the length of the interval and setting the entries to 1.

```java
class Solution {
    static class Interval {
      int x, y;
      public Interval(int x, int y) {
         this.x = x;
         this.y = y;
      }
    }
    // Note: This code does not pass the tests because it does not
    // handle the case of [x, x].
    public int[][] merge(int[][] intervals) {
        int[] is_covered = new int[10001];
        for (int i = 0; i < intervals.length; ++i) {
            for (int j = intervals[i][0]; j < intervals[i][1]; ++j) {
                is_covered[j] = 1;
            }
        }
        ArrayList<Interval> ans = new ArrayList<>();
        int start = 0;
        for (int i = 1; i < is_covered.length; ++i) {
            if (is_covered[i] == 1 && is_covered[i-1] == 0) {
                start = i;
            } else if (is_covered[i] == 0 && is_covered[i-1] == 1) {
                ans.add(new Interval(start, i));
            }
        }
        // Fragment for transforming to requested format.
        int[][] ansArr = new int[ans.size()][2];
        for (int i = 0; i < ans.size(); ++i) {
            ansArr[i][0] = ans.get(i).x;
            ansArr[i][1] = ans.get(i).y;
        }
        return ansArr;
    }
}
```

(Solution 2) For each interval $[x, y]$ create a "begin at x event" and a "end at y event". Sort these events by their coordinate and in case of tie place the starting event before an ending event. Then, we iterate through the sorted events and we keep a counter `countCover`, the number of itervals that cover the current coordinate. When this becomes 0 it means that no interval is covering it. When it changes from 0 to 1, it means that a new interval starts.

```
class Solution {
    static class Event implements Comparable<Event> {
        int x;
        boolean is_start;
        public Event(int x, boolean is_start) {
            this.x = x;
            this.is_start = is_start;
        }

        @Override
        public int compareTo(Event other) {
         if (x == other.x) return Boolean.compare(other.is_start, is_start);
         return Integer.compare(x, other.x);
        }
    }
    static class Interval {
      int x, y;
      public Interval(int x, int y) {
         this.x = x;
         this.y = y;
      }
    }
    public int[][] merge(int[][] intervals) {
        ArrayList<Event> arr = new ArrayList<>();
        for (int i = 0; i < intervals.length; ++i) {
            arr.add(new Event(intervals[i][0], true));
            arr.add(new Event(intervals[i][1], false));
        }
        // It is important when we sort to place the starts before
        // the ends.
        Collections.sort(arr);
        ArrayList<Interval> ans = new ArrayList<>();
        int countCover = 0;
        int start = 0;
        for (Event ev : arr) {
         if (ev.is_start) {
            if (countCover == 0) {
               start = ev.x;
            }
            ++countCover;
         } else {
            --countCover;
            if (countCover == 0) {
               ans.add(new Interval(start, ev.x));
               // If we wanted to find the area, we would change this to
               //    area += ev.x - start
            }
         }
        }

        // Fragment for transforming to requested format.
        int[][] ansArr = new int[ans.size()][2];
        for (int i = 0; i < ans.size(); ++i) {
         ansArr[i][0] = ans.get(i).x;
         ansArr[i][1] = ans.get(i).y;
        }
        return ansArr;
    }
}
```

Exercise 1.P.24 [Interval with all types of elements] There are n cows standing on a line at known positions x_i for the i-th cow. The type of i-th cow is t_i. There are B breads of cows. You want to take a photograph of the cows. The photograph covers M unit steps of the line. Describe an algorithm to check if there is an interval of length M (where M is given) that contains all B types of cows.

(Solution 1) We could just go over the possible intervals of length M which have the form $[i, i + M)$. Then we can iterate through the cows of the interval and count the different number of breads. If this is equal to the total number of breads then we are done. To count the number of different breads we can either use sorting (which requires $\mathcal{O}(M \log M)$ time) or use a counting array (which requires $\mathcal{O}(M + B)$). In total this gives an $\mathcal{O}(n \log n + M \cdot n)$ time.
(Solution 2) One observation is that cows should be at the boundary of the intervals. Hence, we just have to find the interval of length M that terminates at each cow i. If we know this interval for cow i (say $[\ell, i]$), then we can find the interval for $i+1$ by discarding the cows from $\ell, \ell+1, \ldots$ that are more than M units away from cow $i+1$). It is easy to remove a cow from the counting array and update the number of different breads that we have seen in the interval. The following code has the implementation details.

```
/* Determines if it is possible to find an interval of length
   at least len which contains all cows. */
bool F(int len){
   // lef: leftmost cow
   int lef = 0, breads_seen = 0;
   bool found = false;
   int i;
   for (i = 0; i<n; i++){
      // We will find the number of different breads
      // if the cow interval ends at i.
      // So, remove all cows that further that len units
      // from the i-th cow. The interval becomes: [left, i]
      while(K[i].x - K[lef].x > len) {
         A[K[lef].r]--; // Decrease bread count.
         // If this was the last cow of the bread, decrease
         // the number of different cows.
         if (A[K[lef].r] == 0) breads_seen--;
         // remove the cow.
         lef++;
      }
      //  Add the i-th cow and update breads seen.
      if (A[K[i].r] == 0) breads_seen++;
      A[K[i].r]++;
      // If we have seen all breads, we are done.
      if (breads_seen == num_breads) {
         found = true; ++i;
         break;
      }
   }
   // Cleanup the counting array.
   while(i > lef) {
      --A[K[lef].r];
      ++lef;
   }
   return found;
}
```

Exercise 1.P.25 [Smallest interval with all types of elements] Extending the setting of Exercise 24, but this time we want to search for the smallest interval length M such that you can capture all B types. **[Usaco Cow Lineup]**

Let $f(len)$ be the binary function that checks if there is an interval of length *len* that contains all possible types of cows. Then f should be false until some threshold point where it becomes true. Hence, it is monotonic and the transition point will give the length of the smallest interval.

```
#include <cstdio>
#include <cstdlib>
#include <algorithm>

using namespace std;

const size_t MAXN = 1'000'000;

struct cow{
   int x, r;
   bool operator<(const cow &C)const{
      return x < C.x;
   }
} K[MAXN];

int n, num_breads;
int A[MAXN];

/* Determines if it is possible to find an interval of length
   at least len which contains all cows. */
bool F(int len){
   // lef: leftmost cow
   int lef = 0, breads_seen = 0;
   bool found = false;
   int i;
   for (i = 0; i<n; i++){
      // We will find the number of different breads
      // if the cow interval ends at i.
      // So, remove all cows that further that len units
      // from the i-th cow. The interval becomes: [left, i]
      while(K[i].x - K[lef].x > len) {
         A[K[lef].r]--; // Decrease bread count.
         // If this was the last cow of the bread, decrease
         // the number of different cows.
         if (A[K[lef].r] == 0) breads_seen--;
         // remove the cow.
         lef++;
      }
      //  Add the i-th cow and update breads seen.
      if (A[K[i].r] == 0) breads_seen++;
      A[K[i].r]++;
      // If we have seen all breads, we are done.
      if (breads_seen == num_breads) {
         found = true; ++i;
         break;
      }
   }
   // Cleanup the counting array.
   while(i > lef) {
      --A[K[lef].r];
      ++lef;
   }
   return found;
}
```

```
int main(){
   freopen("I.","r",stdin);
   freopen("O.","w",stdout);
   scanf("%d", &n);
   for (int i = 0; i<n; i++){
      scanf("%d%d", &K[i].x, &K[i].r);
   }
   // Sort by bread and translate breads in [0, n).
   sort(K, K+n, [](const cow& c1, const cow& c2) { return c1.r < c2.r; });
   int prev = -1, cur_id = -1;
   for (int i = 0; i < n; ++i) {
      int old = K[i].r;
      if (K[i].r == prev) K[i].r = cur_id;
      else {
         prev = K[i].r;
         K[i].r = ++cur_id;
      }
   }
   num_breads = cur_id + 1;
   // Sort breads by position.
   sort(K, K+n);
   int st = 0, en = K[n-1].x - K[0].x, mn;
   while (st < en) {
      mn = (st+en)/2;
      if (F(mn) == true) en = mn;
      else st = mn + 1;
   }
   printf("%d\n", st);
   return 0;
}
```