Soluzioni Esercitazione II ASD 2022-23

Alessandra Raffaetà   Lorenzo Cazzaro

Esercizio 1. Sia `arr` un array di lunghezza $\texttt{n} - \texttt{k}$ con $\texttt{k} \geq 2$ e $\texttt{k} \leq \texttt{n}$, privo di ripetizioni e contenente interi nell'intervallo `[n*n + 1, n*n + n]`. Si consideri il problema di determinare i `k` numeri interi appartenenti all'intervallo `[n*n + 1, n*n + n]` che non compaiono in `arr`. Si scriva una funzione **EFFICIENTE** che, dati `arr`, `n` e `k`, risolva il problema proposto inserendo gli interi che non compaiono in `arr` in un vettore ordinato.

Il prototipo della funzione è:

```
vector<int> determinaK(const vector<int>& arr, int n, int k)
```

Calcolare e giustificare opportunamente, formalmente e in modo dettagliato la complessità dell'algoritmo proposto.

Soluzione:

```
/* Restituisce il vettore contenente gli interi dell'intervallo che non
    compaiono in arr.*/
vector<int> determinaK(const vector<int>& arr, int n, int k){
    vector<bool> occ(n, false); //vettore delle occorrenze di dimensione n e
    con tutte le celle a false.
    vector<int> ris;
    size_t i{0};

    //viene sfruttata l'ipotesi riguardo l'intervallo di numeri naturali. Il
    minimo valore e' n*n+1
    int min = n * n + 1;
    for (size_t j=0; j< arr.size(); j++)
        occ[arr[j] - min] = true; //viene utilizzato true per indicare i valori
     presenti in arr

    while (k > 0){
        if (!occ[i]){
            ris.push_back(i + min); //vengono inseriti in ris solo gli elementi
     non contenuti in arr
            k--;
        }
        i++;
    }
    return ris;
}
```

Per il calcolo della complessità temporale asintotica nel caso peggiore, consideriamo i due cicli all'interno della funzione `determinaK`. Notiamo che tramite il ciclo `for` viene scansionato l'intero array `arr` di dimensione $n - k$ dunque tale ciclo ha costo $\mathcal{O}(n)$. Il corpo del ciclo `while` viene eseguito al più `n` volte per scansionare il vettore delle occorrenze `occ`, dunque il costo è $\mathcal{O}(n)$. Sommando i due costi si ottiene che la complessità nel caso peggiore è $\mathcal{O}(n)$.

Esercizio 2. Sia `arr` un array di `n` numeri naturali. Si consideri il problema di restituire in ordine crescente tutti i numeri che compaiono in `arr` almeno $\lceil$`n/k`$\rceil$ volte, dove `k` $>$ `0` è una costante.

Si scriva una funzione **EFFICIENTE** che, dati `A` e `k`, risolva il problema proposto.

Valutare e giustificare opportunamente, formalmente e in modo dettagliato la complessità dell'algoritmo proposto.

Il prototipo della funzione è:

```
vector<int> cercaValori(vector<int>& arr, int k)
```

Soluzione:

```
void mymerge(vector<int>& arr, int p, int med, int r){

  vector<int> aux;
  int i = p, j = med +1;

  while (i <= med && j <= r)
    if (arr[i] <= arr[j]){
      aux.push_back(arr[i]);
      ++i;
    }
    else {
      aux.push_back(arr[j]);
      ++j;
    }

  if (i <= med){
    for (j = med; j>= i; --j){
      arr[r] = arr[j];
      --r;
    }
  }
  for (i = 0; i < aux.size(); ++i)
    arr[p + i] = aux[i];
}

/*N.B.: la funzione di ordinamento doveva essere implementata*/
void mymergesort(vector<int>& arr, int p, int r){
  if (p < r){
    int med = (p + r)/2;
```

```
    mymergesort(arr, p, med);
    mymergesort(arr, med + 1, r);
    mymerge(arr, p, med, r);
  }
}

/* Ritorna il vettore ris che contiene in ordine crescente gli elementi di arr
    che compaiono almeno ceil(arr.size()/(double)k) volte.*/
vector<int> cercaValori(vector<int>& arr, int k){

    int soglia, i;
    vector<int> ris;
    mymergesort(arr, 0, arr.size() - 1); //viene utilizzato l'ordinamento per
    trovare in modo efficiente gli elementi che soddisfano la condizione

    soglia = ceil(arr.size()/(double)k); //soglia ipotesi del problema
    i = 0;
    while (i + soglia <= arr.size())
        //dato che arr e' ordinato, se arr[i] == arr[i + soglia - 1] allora arr
     contiene almeno ceil(arr.size()/(double)k) volte il numero contenuto
    nelle celle con indice tra i e i + soglia - 1.
        if (arr[i] == arr[i + soglia - 1]){
            ris.push_back(arr[i]);
            i += soglia;
            //arr potrebbe contenere piu' di ceil(arr.size()/(double)k) del
    numero
            while (i < arr.size() && arr[i - 1] == arr[i])
                i++;
        }
        else
            i++;

    return ris;
}
```

Per brevità, si rimanda alla dimostrazione vista a lezione della complessità temporale asintotica dell'algoritmo di ordinamento *mergesort*, che risulta essere $\Theta(n * logn)$, dove `n` è la dimensione del vettore in input. Si ricorda che la dimostrazione doveva comunque essere inserita come commento al codice della soluzione fornita.

La funzione `cercaValori` richiama la funzione `mymergesort` la cui complessità temporale asintotica è $\Theta(n * logn)$, dove `n` è la dimensione del vettore `arr`. Il ciclo `while` all'interno della funzione esegue una scansione del vettore `arr` e ciascun elemento viene visitato al più una volta. Di conseguenza, la complessità temporale asintotica nel caso peggiore è $\Theta(n * logn) + \mathcal{O}(n) = \Theta(n * logn)$.

Esercizio 3. Dato un albero binario `T` contenente `n` (>0) chiavi intere distinte, si consideri il problema di verificare se esiste un albero binario di ricerca `T'` avente la stessa visita in pre-ordine di `T`.

1. Dato `T` esiste sempre `T'` che soddisfa le condizioni sopra descritte? In caso affermativo fornire una dimostrazione. In caso negativo fornire un controesempio

e descrivere una condizione sulla visita in pre-ordine di `T` che garantisca l'esistenza di `T'`.

2. Scrivere una funzione efficiente `preOrder` che dato un vettore contenente la visita in pre-ordine di `T`, restituisce un albero binario di ricerca `T'` avente la stessa visita in pre-ordine di `T`, `nullptr` se non esiste.

3. Calcolare e giustificare opportunamente, formalmente e in modo dettagliato la complessità della funzione `preOrder`.

Il prototipo della funzione è:

```
PTree preOrder(const vector<int>& v)
```

Il tipo PNode è così definito:

```
struct Node {
       int key;
       Node* p;
       Node* left;
       Node* right;
       Node(int val, Node* parent = nullptr, Node* sx = nullptr, Node* dx = nullptr)
            :  key{val}, p{parent}, left{sx}, right{dx} {}
 };

typedef Node* PNode;

struct Tree {
       PNode root;
       Tree(PNode r = nullptr)
            :  root{r} {}
 };

typedef Tree* PTree;
```

Soluzione:

1. No, `T'` non esiste sempre. Ecco un controesempio. Sia $[10, 5, 20, 9]$ la visita in pre-ordine di `T`. Assumiamo per assurdo che `T'` esista. Il nodo con chiave 10 deve essere la radice dell'albero `T'` e il nodo con chiave 20 deve necessariamente appartenere al sottoalbero destro della radice in quanto `T'` è un albero binario di ricerca. Visto come opera la visita in pre-ordine, il nodo con chiave 9 deve appartenere al sottoalbero sinistro o destro del nodo con chiave 20, dunque è un nodo del sottoalbero destro della radice di `T'`. Questo è assurdo in quanto essendo `T'` un albero binario di ricerca ogni chiave nel sottoalbero destro della radice deve avere chiave maggiore o uguale a 10. Di conseguenza non può esistere alcun albero binario di ricerca `T'` avente la stessa visita in pre-ordine di `T`.

   Affinché la sequenza s ottenuta dalla visita in pre-ordine di `T` corrisponda a quella di un albero binario di ricerca `T'` è necessario che sia soddisfatta la seguente condizione. Detta r la prima chiave della sequenza s deve essere possibile partizionare la parte rimanente di s in due sottosequenze consecutive s_1 e s_2 tali che ogni elemento di s_1 è minore di r e ogni elemento di s_2 è maggiore di r. La condizione deve valere ricorsivamente su ciascuna delle due sottosequenze s_1 e s_2 a meno che non siano vuote.

2.
```
/*Restituisce l'albero costruito a partire dalla visita in pre-ordine di
    un albero generico contenuta in v. min e max indicano il limite
    inferiore e superiore delle chiavi del sottoalbero che si va ad
    allocare.*/
PNode checkaux(const vector<int>& v, int& i, int min, int max, PNode
    padre){
    int key;
    PNode r;

    //sono stati visitati tutti gli elementi del vettore
    if (i == v.size())
        return nullptr;

    key = v[i];
    //la chiave non rispetta la proprieta' di ricerca, quindi non puo'
    essere inserita in questo sottoalbero
    if (key < min || key > max)
      return nullptr;

    r = new Node(key, padre);

    i+= 1;
    //nel sottoalbero sinistro posso solo aver valori con limite
    superiore key, chiave di r
    r->left = checkaux(v, i, min, key, r);
    //nel sottoalbero destro posso solo aver valori con limite inferiore
    key, chiave di r
    r->right = checkaux(v, i, key, max, r);
    return r;
}

//Cancella l'albero allocato
void deleteTree(PNode r){
    if (r != nullptr){
        deleteTree(r->left);
        deleteTree(r->right);
        delete r;
    }
}

/*Restituisce l'albero binario di ricerca costruito dalla visita in pre-
    ordine contenuta in v se esiste, altrimenti viene restituito nullptr
    */
PTree preOrder(const vector<int>& v){
    PTree t;
    int i = 0;
    PNode r;
```

```
    r = checkaux(v, i, INT_MIN, INT_MAX, nullptr);
    //se tutti gli elementi del vettore v sono stati allocati, allora  l'
    albero binario di ricerca esiste
    if (i == v.size()){
        t = new Tree(r);
        return t;
    }
    //se l'albero non e' valido, libera la memoria allocata
    deleteTree(r);

    return nullptr;
}
```

3. Indichiamo con `n` la dimensione del vettore `v`. La funzione `preOrder` chiama le funzioni ausiliarie `checkaux` e `deleteTree`. Per stabilire la complessità temporale asintotica di `preOrder` è sufficiente stabilire quella di `checkaux` e `deleteTree`.

   La funzione `checkaux` scansiona l'intero vettore `v` nel caso peggiore, cioè quando dalla visita in pre-ordine in `v` è possibile costruire un albero binario di ricerca con numero di nodi pari a `n`. Tuttavia, la scansione del vettore `v` potrebbe essere interrotta prima di essere completata, nel caso in cui la visita in pre-ordine contenuta in `v` non soddisfi la condizione di esistenza dell'albero binario di ricerca corrispondente. Quindi, la complessità temporale asintotica della funzione `checkaux` è $\mathcal{O}(n)$.

   La funzione `deleteTree` viene chiamata nel caso in cui la visita in pre-ordine contenuta in `v` non sia valida. In questo caso, l'albero radicato in `r` conterrà al massimo `n` -1 nodi, quindi la complessità temporale asintotica della funzione `deleteTree` è $\mathcal{O}(n)$.

   La complessità temporale asintotica della funzione `preOrder` è quindi $\mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$.

Esercizio 4. Progettare una realizzazione del tipo di dato coda di max-priorità che utilizza un max-heap. Discutere la complessità in tempo di ciascuna operazione.

Soluzione:

```
struct MaxHeap::Impl{
  vector<int> elements;
  size_t heapsize;
};

//Post: restituisce il padre di i se i non e' la radice, altrimenti -1.
int parent(size_t i){
  if (i == 0)
    return -1;
  return (i-1)/2;
```

```
}

//Post: restituisce il figlio sinistro del nodo i
size_t left(size_t i){
  return i*2 + 1;
}

//Post: restituisce il figlio destro del nodo i
size_t right(size_t i){
  return i*2 + 2;
}

//Post: il sottoalbero radicato nel nodo i e' un max-heap
void MaxHeap::maxHeapify(size_t i){

  size_t massimo, l, r;
  int temp;

  l = left(i);
  r = right(i);

  if (l < pimpl->heapsize && pimpl->elements[l] > pimpl->elements[i])
    massimo = l;
  else
    massimo = i;

  if (r < pimpl->heapsize && pimpl->elements[r] > pimpl->elements[massimo])
    massimo = r;

  if (i != massimo){
    temp = pimpl->elements[i];
    pimpl->elements[i] = pimpl->elements[massimo];
    pimpl->elements[massimo] = temp;
    maxHeapify(massimo);
  }
}

//Post: restituisce un Max Heap vuoto (heapsize = 0)
MaxHeap::MaxHeap(){

  pimpl = new Impl;
  pimpl->heapsize = 0;
}

//Post: trasforma il vettore arr in un Max Heap
MaxHeap::MaxHeap(vector<int>& arr){

  pimpl = new Impl;
  pimpl->elements = arr;
  pimpl->heapsize = arr.size();

  for (int i = floor(pimpl->heapsize / double(2)) - 1; i >= 0; --i)
    this->maxHeapify(i);
```

```
}

//Post: rimuove il Max Heap ternario
MaxHeap::~MaxHeap(){
  delete pimpl;
}

//Post: restituisce true se lo heap e' vuoto, false altrimenti
bool MaxHeap::heapEmpty() const{

  return pimpl->heapsize == 0;
}

//Pre: lo heap e' non vuoto
//Post: restituisce la chiave piu' grande nello Heap
int MaxHeap::heapMaximum() const{

  return pimpl->elements[0];
}

//Pre: lo heap e' non vuoto
//Post: restituisce la chiave piu' piccola nello Heap
int MaxHeap::heapMinimum() const{

    size_t min = floor(pimpl->heapsize/double(2));
    for (size_t i = min + 1; i < pimpl->heapsize; ++i)
        if (pimpl->elements[i] < pimpl->elements[min])
            min = i;

    return pimpl->elements[min];
}

//Pre: lo heap e' non vuoto
//Post: elimina e restituisce la chiave piu' grande nello heap.
int MaxHeap::heapExtractMax(){

  int max;

  max = pimpl->elements[0];
  pimpl->elements[0] = pimpl->elements[pimpl->heapsize - 1];
  pimpl->heapsize -= 1;
  maxHeapify(0);

  pimpl->elements.pop_back();
  return max;
}

//Post: inserisce l'elemento con chiave k nello heap
void MaxHeap::heapInsert(int k){

  int i = pimpl->heapsize;
  int temp;
```

```
  pimpl->elements.push_back(k);
  pimpl->heapsize += 1;

  while (i > 0 && pimpl->elements[parent(i)] < pimpl->elements[i]) {
    temp = pimpl->elements[parent(i)];
    pimpl->elements[parent(i)] = pimpl->elements[i];
    pimpl->elements[i] = temp;
    i = parent(i);
  }
}

//Post: cancella l'elemento in posizione i dello Heap
void MaxHeap::heapDelete(size_t i){

  int delvalue, temp;

  if (pimpl->heapsize == 1)
    pimpl->heapsize -= 1;
  else {
    delvalue = pimpl->elements[i];
    pimpl->heapsize -= 1;
    pimpl->elements[i] = pimpl->elements[pimpl->heapsize];
    if (delvalue > pimpl->elements[i])
      maxHeapify(i);
    else
      while (i > 0 && pimpl->elements[parent(i)] < pimpl->elements[i]) {
          temp = pimpl->elements[parent(i)];
          pimpl->elements[parent(i)] = pimpl->elements[i];
          pimpl->elements[i] = temp;
          i = parent(i);
      }
  }
  pimpl->elements.pop_back();
}

//Post: fornisce l'indice di un nodo del max-heap che contiene il valore k se k
     e' presente nel max-heap, altrimenti restituisce -1.
int MaxHeap::auxSearch(int k, size_t node) const{

  int ris;

  if (pimpl->heapsize <= node || pimpl->elements[node] < k)
    return -1;

  if (pimpl->elements[node] == k)
    return node;

  ris = auxSearch(k, left(node));
  if (ris != -1)
    return ris;

  return auxSearch(k, right(node));
}
```

```
//post: restituisce la posizione dello heap che contiene k se k e' nello heap
    altrimenti -1
int MaxHeap::heapSearch(int k) const{

  return auxSearch(k, 0);

}
```

Si rimanda al libro di testo per le complessità asintotiche della maggior parte dei metodi da implementare in questo esercizio. Si ricorda che era necessario indicare sotto forma di commento le complessità asintotiche e le motivazioni (eventualmente lo sketch delle dimostrazioni). Di seguito viene indicata e motivata la complessità temporale asintotica dei metodi non descritti nel libro di testo (`n` è la dimensione di `elements`):

- `heapMinimum`: vengono esaminate tutte le foglie dello heap, quindi viene eseguita una scansione del vettore `elements` a partire dall'indice $\lfloor \frac{\mathtt{n}}{2} \rfloor$. La complessità temporale asintotica è quindi $\Theta(n)$.
- `heapDelete`: la complessità temporale asintotica è determinata dalla complessità di `maxHeapify` e dalla complessità del ciclo `while`. In entrambi i casi la complessità è $\mathcal{O}(logn)$, quindi la complessità asintotica di `heapDelete` è $\mathcal{O}(logn)$.
- `heapSearch`: nel caso peggiore tutti i nodi dello heap sono esaminati una volta. Quindi la complessità temporale asintotica è $\mathcal{O}(n)$.

Esercizio 5. Utilizzando le funzioni della classe `MaxHeap` dell'esercizio 4, si realizzi in modo efficiente la procedura `Differenza(H1,H2)` che dati due `Max-Heap H1` e `H2` contenenti rispettivamente `n1` e `n2` interi (anche ripetuti), ritorna in output un nuovo `Max-Heap` contenente la differenza di `H1` e `H2`. Se `x` compare `k1` volte in `H1` e `k2` volte in `H2`, nel Max-Heap differenza `x` dovrà comparire `max(0, k1-k2)`. Si determini e giustifichi la complessità in funzione di `n1` e `n2`.

Il prototipo della procedura è

```
void differenza(MaxHeap& h1, MaxHeap& h2, MaxHeap& hris)
```

Soluzione:

```
/* Restituisce il max-heap come risultato della differenza dei due heap in
    input.*/
void differenza(MaxHeap& h1, MaxHeap& h2, MaxHeap& hris){
    int max1, max2;

    while (!h1.heapEmpty() && !h2.heapEmpty()){
        max1 = h1.heapMaximum();
        max2 = h2.heapMaximum();
        if (max1 == max2){
```

```
            h1.heapExtractMax();
            h2.heapExtractMax();
        }
        else
            if (max1 > max2){
                hris.heapInsert(max1);
                h1.heapExtractMax();
            }
            else
                h2.heapExtractMax();
    }

    while (!h1.heapEmpty())
        hris.heapInsert(h1.heapExtractMax());
}
```

La complessità asintotica di `heapInsert` è $\mathcal{O}(logn)$, con `n` dimensione dell'heap. In questo caso la sua complessità è $\Theta(1)$ poiché l'elemento inserito nell'heap `hris` sarà sempre non maggiore di tutti gli altri elementi già inseriti. Per costruzione `hris` è un vettore ordinato in modo non crescente. Quindi la complessità asintotica della funzione `differenza` è determinata dal costo dell'estrazione del massimo dagli heap e dal numero di iterazioni eseguite. Dato che, nel caso peggiore, è necessario estrarre tutti gli elementi contenuti nei due max-heap, la complessità temporale asintotica della procedura `differenza` è $\mathcal{O}(n1 * log(n1) + n2 * log(n2))$.

Esercizio 6. L'urbanista Francesco Altissoni ha a disposizione `n` possibili ubicazioni in cui collocare delle case lungo un viale. Tali posizioni sono contenute nel vettore `possibili_posizioni` e sono espresse in metri di distanza rispetto al punto iniziale del viale. L'obiettivo di Francesco è quello di collocare `k` (`k`$\leq$`n`) case in modo tale da massimizzare la distanza minima fra le case, al fine di garantire la massima riservatezza agli abitanti di ciascuna abitazione. Aiuta Francesco nel suo scopo: realizza un algoritmo efficiente che gli permetta di individuare le `k` posizioni delle case che devono essere inserite all'interno del vettore `res` (che risulta già allocato e di dimensione `k`).

Il prototipo della procedura è

```
void posizioniMigliori(vector<int>& possibili_posizioni, int k,
                       vector<int>& res)
```

Soluzione:

```
void mymerge(vector<int>& arr, int p, int med, int r){
  vector<int> aux;
  int i = p, j = med +1;

  while (i <= med && j <= r)
    if (arr[i] <= arr[j]){
      aux.push_back(arr[i]);
      ++i;
```

```
    }
    else {
      aux.push_back(arr[j]);
      ++j;
    }

  if (i <= med){
    for (j = med; j>= i; --j){
      arr[r] = arr[j];
      --r;
    }
  }
  for (i = 0; i < aux.size(); ++i)
    arr[p + i] = aux[i];
}

/*N.B.: la funzione di ordinamento doveva essere implementata*/
void mymergesort(vector<int>& arr, int p, int r){
  if (p < r){
    int med = (p + r)/2;
    mymergesort(arr, p, med);
    mymergesort(arr, med + 1, r);
    mymerge(arr, p, med, r);
  }
}

//Restituisce true se e' possibile trovare k posizione con distanza minima mid
bool feasible(int mid, vector<int>& possibili_posizioni, int k){
    int pos = possibili_posizioni[0];
    int i = 1;
    k--;

    while(k > 0 && i < possibili_posizioni.size()){
        if (possibili_posizioni[i] - pos >= mid) {
            pos = possibili_posizioni[i];
            k--;
        }
        i++;
    }

    return !k;
}

//Riempie res con k posizioni che distano al minimo un valore pari a distanza
void assegnaPosizioni(int distanza, vector<int>& possibili_posizioni, int k,
    vector<int>& res) {
    int pos = possibili_posizioni[0], i = 1, j = 0;
    res[j] = possibili_posizioni[0];
    j++;

    while(j < k && i < possibili_posizioni.size()) {
        if (possibili_posizioni[i] - pos >= distanza) {
            res[j] = possibili_posizioni[i];
```

```
            pos = possibili_posizioni[i];
            j++;
        }
        i++;
    }
}

//Riempie res con le posizioni a massima distanza minima.
void posizioniMigliori(vector<int>& possibili_posizioni, int k, vector<int>&
    res){
    //L'array viene ordinato per rendere piu' efficiente la ricerca
    mymergesort(possibili_posizioni, 0, possibili_posizioni.size() - 1);

    //Consideriamo la minima e la massima distanza possibile a cui possiamo
    collocare le case
    int left = 1, right = possibili_posizioni[possibili_posizioni.size() - 1];

    //Ricerca binaria per la distanza minima
    int best = 1;
    while (left <= right) {
        int mid = (left + right) / 2;

        if (feasible(mid, possibili_posizioni, k)) {
            left = mid + 1;
            best = mid;
         } else
            right = mid - 1;
    }
    assegnaPosizioni(best, possibili_posizioni, k, res);
}
```

Al fine di stabilire la complessità asintotica della procedura `posizioniMigliori` è necessario determinare la complessità asintotica delle funzioni `feasibile` e `assegnaPosizioni`.

La complessità asintotica della funzione `feasible` è $\mathcal{O}(n)$, in quanto ciascun elemento del vettore `possibili_posizioni` viene esaminato al più una volta. La complessità è la stessa anche per la procedura `assegnaPosizioni`.

L'intervallo di valori all'interno del quale cerchiamo il massimo valore di minima distanza contiene inizialmente *limit* distanze da 1 a *limit*, in quanto consideriamo solo distanze come numeri interi, dove $limit = max(possibili\_posizioni)$, cioè la distanza dell'ultima casa della via dall'inizio via. L'intervallo viene dimezzato progressivamente tramite un approccio divide-et-impera.

L'equazione di ricorrenza che descrive la complessità temporale asintotica della funzione `posizioniMigliori`, che richiama `feasible` e `assegnaPosizioni`, è:

$$T(limit, n) = \begin{cases} \Theta(n) & limit \leq 1 \\ T(\frac{limit}{2}, n) + \Theta(n) & limit > 1 \end{cases}$$

Tramite il metodo iterativo o il metodo di sostituzione, si ottiene la complessità $\mathcal{O}(log_2(limit) * n)$