Algoritmi e Strutture Dati

Appello del 21/2/2025 - Prova di programmazione (12 punti)

NOME: …… MATR: ……

1. (2 punti)

È dato un ADT `editori_adt`, in grado di rappresentare un elenco di editori (un vettore allocato dinamicamente). Ogni editore, oltre al nome (una stringa allocata dinamicamente), contiene un vettore di puntatori a libri (tipo struct libro): i libri sono da considerarsi come esterni alla struttura dati: l'ADT è quindi di tipo "aggregato", per quanto concerne i libri, che non vanno né allocati né deallocati. Sono date le definizioni dei tipi. Si scrivano due funzioni: una che duplica e una che libera l'ADT.

```
typedef struct {
  char *nome;
  struct libro **libri;
  int n_libri;
} editore_t;
```

```
struct editori {
  editore_t *editori;
  int n_editori;
};

typedef struct editori *editori_adt;
```

EditoriDup

```
// si omettono i controlli di errore
editori_adt EditoriDup(editori_adt e) {
  editori_adt e2 = malloc(sizeof(*e2));
  e2->n_editori = e->n_editori;
  e2->editori = malloc(e2->n_editori*sizeof(editore_t));
  for (int i=0; i<e2->n_editori; i++) {
    int nl = e->editori[i].n_libri;
    e2->editori[i].n_libri = nl;
    e2->editori[i].nome = strdup(e->editori[i].nome);
    e2->editori[i].libri = malloc(nl*sizeof(struct libro *));
    for (int l=0; l<nl; l++) {
      e2->editori[i].libri[l] = e->editori[i].libri[l];
    }
  }
  return e2;
}
```

EditoriFree

```
void EditoriFree(editori_adt e) {
  for (int i=0; i<e->n_editori; i++) {
    free(e->editori[i].nome);
    free(e->editori[i].libri);
  }
  free(e->editori);
  free(e);
}
```

2. (4 punti)

È dato un BST (o albero binario, il problema è equivalente). Si scriva una funzione che verifichi se l'albero è bilanciato: si utilizzi come criterio di bilanciamento quello basato sulle dimensioni dei sotto-alberi. La funzione abbia prototipo:

```
int BSTisBalanced(BST b);
```

Si richiede, oltre alla funzione, la definizione del tipo BST e del tipo usato per il nodo. Si assuma che il nodo dell'albero NON sia stato esteso in modo da contenere la dimensione del relativo sotto-albero.

```
typedef struct  binarysearchtree *BST;
…
typedef struct BSTnode* link;
struct BSTnode { Item item; link p; link l; link  r; int N; } ;
struct binarysearchtree { link root;  link z; };

// due risultati: ritorna dimensione di sotto-albero
// e  "by pointer" il bilanciamento.
static int countAndCheckBalR(link h, link z, int *balP) {
 if (h == z)
   return 0;
 int cnt_l = countAndCheckBalR(h->l, z, balP);
 int cnt_r = countAndCheckBalR(h->r, z, balP);
 if (abs(cnt_l-cnt_r)>1)
   *balP = 0;
 return cnt_r + cnt_l + 1;
}

int BSTisBalanced(BST b) {
  int balanced=1;
  int cnt = countAndCheckBalR(b->root,b->z, &balanced);
  return balanced;
}
```

3. (**6 punti**)

È dato un vettore V contenente N interi non negativi: il vettore non è ordinato e sono possibili dati ripetuti. Si vogliono generare, come risultato, due vettori (allocati dinamicamente) V1 e V2, in cui suddividere gli interi, in modo tale che, date la somma S1 degli interi in V1 e la somma S2 degli interi in S2, si minimizzi |S2-S1|. La funzione deve poter essere chiamata come segue:

```
int N1, N2, *V1, *V2;
N1 = bestSplit(V, N, &V1, &V2);
N2 = N-N1;
```

V1 e V2 sono (puntatori a) vettori di interi che vanno allocati dinamicamente nella bestSplit, N1 è il numero di interi in V1, mentre N2 è il numero di interi in V2. Si riporta una soluzione parziale che va completata

```
void bestSolR(int *val, int *sol,
              int *bestSol, int sum,
              int *bestSumP,
              int goal, int n, int start) {
   int i;
   if (checkTerminal(n,start,sum,goal)) {
     checkBestSol(n,sol,bestSol,sum,
                  bestSumP,goal);
     return;
   }
   for (i = <1>; i < n; i++) {
      sol[i] = 1;
      bestSolR(<2>);
      sol[i] = 0;
   }
}
```

```
int bestSplit(<3>) {
  int *sol = calloc(N,sizeof(int));
  int *bestSol = calloc(N,sizeof(int));
  int bestSum=0;
  int goal = 0;
  int n1;
  for (int i=0; i<N; i++) goal += V[i];
  goal /= 2;
  // la funzione ricorsiva trova i dati per
  // uno dei due vettori; i dati non selezionati
  // andranno nel secondo vettore.
  bestSolR(V, sol, bestSol, 0,
           &bestSum, goal, N, 0);
  n1 = buildSol(V,N,bestSol,V1p,V2p);
  free(sol);
  free(bestSol);
  return n1;
}
```

DOMANDA	RISPOSTA
<1> va sostituito con	start
<2> va sostituito con	val, sol, bestSol, sum+val[i], bestSumP, goal, n, i+1
<3> va sostituito con	int *V, int N, int **V1p, int **V2p
Cosa enumera la bestSolR, in relazione allo spazio delle scelte/soluzioni? (motivare il modello del calcolo combinatorio indicato)	Enumera il powerset dell'insieme di numeri, con una variante del powerset "divide et impera", in cui si usa come soluzione un vettore di flag (anziché vettore di valori) perché questo rende piú semplice il passo finale di divisione del vettore in due parti. Il caso terminale viene identificato dalla somma de valori scelti, invece che dal conteggio: data la somma S di tutti i dati, appena si raggiunge o supera S/2, si è raggiunto l'obiettivo. Infatti, l'ottimo è dato da due sottoinsiemi con somma esattamente S/2 oppure tali da minimizzare la differenza rispetto a S/2.

Scrivere la funzione checkTerminal	int checkTerminal(int n, int start, int sum, int goal) {
	  return sum>=goal;
	}

Scrivere la funzione checkBestSol	void checkBestSol(int n, int *sol, int *bestSol,
	  int sum, int *bestSumP, int goal) {
	  if (abs(sum-goal) < abs(*bestSumP-goal)) {
	  *bestSumP = sum;
	  for (int i=0; i<n; i++) {
	  bestSol[i] = sol[i];
	  }
	  }
	}

Scrivere la funzione buildSol	int buildSol(int *V, int N, int *bestSol, int **V1p, int**V2p) {
	  int n1=0, i1=0, i2=0;
	  for (int i=0; i<N; i++)
	  if (bestSol[i]) n1++;
	  *V1p = malloc(n1*sizeof(int));
	  *V2p = malloc((N-n1)*sizeof(int));
	  for (int i=0; i<N; i++)
	  if (bestSol[i])
	  (*V1p)[i1++] = V[i];
	  else
	  (*V2p)[i2++] = V[i];
	  return n1;
	}

Algoritmi e Strutture Dati

Appello del 21/2/2025 - Prova di programmazione (18 punti)

Descrizione del problema

Si vuole risolvere un problema di pianificazione di attività. È dato un elenco di attività. Ogni attività è caratterizzata da nome (stringa di non più di 20 caratteri), tempo di inizio, tempo di fine e costo. È poi dato un elenco di dipendenze, ognuna data da due attività: la dipendenza (A_j,A_k) indica che l'attività A_j, non può iniziare prima che sia terminata A_k.
Esempio
Elenco attività: `(A1A,1,4,12), (BB75,12,14,270),(AB12B,0,6,201), (ADD,5,7,45), (A4,3,5,36), (CNT3,5,9,74), (T3B,8,11,130), (YA,6,10,103)`
Elenco precedenze: `(ADD,A1A), (T3B,ADD), (T3B,AB12B), (CNT3,A4)`

Ogni attività deve essere eseguita da una persona, più attività possono essere eseguite in (parziale) sovrapposizione nel tempo, purché siano assegnate a un numero sufficiente di persone: l'inizio di un'attività nel tempo in cui ne termina un'altra non va considerato come sovrapposizione. Si vuole verificare che le attività possano essere eseguite e si vuole determinare un'assegnazione ottima di attività a persone.

Richieste del problema

A seguire una sintesi delle richieste del problema. Per ogni richiesta si troverà una domanda dedicata nelle sezioni a seguire con una descrizione più dettagliata per le richieste.

Strutture dati e letture

Si definisca, come ADT di prima classe, una struttura dati ACT, in grado di rappresentare le attività e le dipendenze e di risolvere (nel modo più efficiente possibile), i problemi che seguono. Attenzione: non è richiesta alcuna lettura di file o funzione di inizializzazione/allocazione, sono ***solo richieste le definizioni dei tipi di dato***.

```
// un'attivita'
typedef struct {
  char nome[MAXC];
  int inizio;
  int fine;
  int costo;
} attivita_t;

// una dipendenza
typedef struct {
  int act;
  int dep;
} dep_t;
```

```
/* ADT collezione con wrapper che contiene
      -  tabella di simboli (internamente è meglio usare indici)
      -  vettore di attivita' e vettore di dipendenze
      -  nMin e soluzione migliore */
typedef struct act *ACT;
struct act {
  ST tab;
  int na;
  int nd;
  attivita_t *elenco_att;
  dep_t *elenco_dip;
  int nMin;
  float bestSum;
  int *bestPlan;
};
```

Problema di verifica

Si scriva una funzione in grado di verificare se un elenco di attività e delle relative dipendenze sia fattibile, cioè che nessuna attività inizi prima del termine delle attività da cui dipende: per l'esempio proposto, l'eventuale precedenza `(ADD,AB12B)` farebbe fallire la verifica. Si realizzi una seconda funzione che, se la precedente verifica ha avuto successo, determini se il grafo delle dipendenze sia un DAG (grafo diretto aciclico) oppure un albero radicato (o foresta) di alberi radicati: a tale scopo si noti che ***una dipendenza (X,Y), in cui l'attività X dipende dalla Y, va intesa come arco orientato X->Y***. La funzione stampi, oltre al risultato della verifica, nel caso di un DAG le sorgenti (source) e i pozzi (sink), nel caso di albero (o foresta) la(e) radice(i) e le foglie: quindi ***una foglia di albero (o sink/pozzo di DAG) è un'attività priva di dipendenze (ragionamento duale per radici/sorgenti***).
Le funzioni richieste abbiano prototipi

```
int ACTcheckDep(ACT a);
void ACTprintSrcSnk(ACT a);
```

```
// Verifica compatibilita' tra i tempi
int ACTcheckDep(ACT a) {
  for (int i=0; i<a->nd; i++) {
    int att = a->elenco_dip[i].act;
    int dip = a->elenco_dip[i].dep;
    if (a->elenco_att[dip].fine >
        a->elenco_att[att].inizio)
      return 0;
  }
  return 1;
}
```

```
// I source hanno in_degree==0, i sink out_degree==0
void ACTprintSrcSnk(ACT a) {
  int *in_degree=calloc(a->na,sizeof(int));
  int *out_degree=calloc(a->na,sizeof(int));
  int isTree=1;
  for (int i=0; i<a->nd; i++) {
    if ((in_degree[a->elenco_dip[i].dep]++)>0)
      isTree = 0;
    out_degree[a->elenco_dip[i].act]++;
  }
  printf("le dipendenze sono un %s\n",isTree?"albero":"DAG");
```

```
  printf("root(s)/source(s):");
  for (int i=0; i<a->na; i++)
    if (in_degree[i]==0) printf(" %s", a->elenco_att[i].nome);
  printf("\n");
  printf("leave(s)/sink(s):");
  for (int i=0; i<a->na; i++)
    if (out_degree[i]==0) printf(" %s", a->elenco_att[i].nome);
  printf("\n");
  free(in_degree);
  free(out_degree);
}
```

Problema di ricerca e ottimizzazione

Si scriva una funzione `ACTminPers` che, dato un elenco di attività, determini il minimo numero di persone N_{min} in grado di svolgerle tutte. Si scriva infine una funzione `ACTbestPlan` che calcoli un'assegnazione ottima di attività alle N_{min} persone: la persona P_j riceverà come compenso CP_j, la somma dei costi delle attività svolte/assegnate a P_j. Il criterio di ottimizzazione è dato dal distribuire i compensi nel miglior modo possibile, cioè in modo che sia minimo $SUM_j(|CP_j - CP_{avg}|)$, indicando con CP_{avg} la media aritmetica dei CP_j. La ripartizione delle attività va memorizzato nella struttura dati (l'ADT): non è quindi necessario che sia stampata.

Le funzioni abbiano prototipo

```
int ACTminPers(ACT a);
void ACTbestPlan(ACT a);
```

Il numero minimo di persone è determinato dall'intervallo di tempo con il massimo numero di sovrapposizioni. Per fare questo occorre percorrere i tempi (di inizio e fine) in ordine crescente, incrementando un contatore ogni volta che inizia un'attività e decrementandolo ogni volta che termina (si veda l'esempio delle intersezioni tra rettangoli presentato al termine degli IBST).

Una volta determinato il numero di lavoratori, l'algoritmo di ottimizzazione consiste nel determinare il partizionamento migliore, essendo noto il numero di partizioni: la soluzione proposta utilizza l'algoritmo delle disposizioni con ripetizione, ma come alternativa si potrebbe usare Er, nella variante con k noto. La soluzione proposta, oltre alla soluzione, usa un vettore di costi (con un costo per ogni partizione/persona): si usa pruning, evitando attività incompatibili con altre già assegnate alla persona scelta. La checkBest gestisce il confronto con la soluzione ottima.

```
int ACTminPers(ACT a) {
  int i, f, minPers=0, cntPers=0;
  int *t_inizio = malloc(a->na*sizeof(int));
  int *t_fine = malloc(a->na*sizeof(int));
  for (i=0; i<a->na-1; i++) {
    t_inizio[i] = a->elenco_att[i].inizio;
    t_fine[i] = a->elenco_att[i].fine;
  }
  sortInt(t_inizio,a->na);
  sortInt(t_fine,a->na);
  for (i=f=0; i<a->na;) {
    if (t_fine[f]<=t_inizio[i]) {
      cntPers--; f++;
    }
    else {
      cntPers++; i++;
      if (cntPers>minPers) minPers=cntPers;
    }
  }
  a->nMin = minPers;
  free(t_inizio);
  free(t_fine);
  return minPers;
}

void checkBest(ACT a, int *sol, int *costi) {
  float s=0.0, avg = 0.0;
  for (int i=0; i<a->nMin; i++)
    avg += costi[i];
  avg /= a->nMin;
  for (int i=0; i<a->nMin; i++)
    s += abs(costi[i]-avg);
  if (a->bestSum<0.0 || s<a->bestSum) {
    a->bestSum = s;
    for (int i=0; i<a->na; i++)
      a->bestPlan[i] = sol[i];
  }
}
```

```
int pruningCheck(ACT a, int pos, int *sol, int p) {
  for (int i=0; i<pos; i++) {
    if (sol[i]==p) {
      if (a->elenco_att[pos].inizio<a->elenco_att[i].fine &&
          a->elenco_att[i].inizio<a->elenco_att[pos].fine)
        return 0;
    }
  }
  return 1;
}

void disp_ripet(ACT a, int pos, int *sol, int *costi) {
  if (pos >= a->na) {
    checkBest(a, sol, costi);
  }
  for (int p = 0; p < a->nMin; p++) {
    if (pruningCheck(a,pos,sol,p)) {
      sol[pos] = p;
      costi[p] += a->elenco_att[pos].costo;
      disp_ripet(a, pos+1, sol, costi);
      costi[p] -= a->elenco_att[pos].costo;
    }
  }
}

void ACTbestPlan(ACT a) {
  int *sol = calloc(a->na,sizeof(int));
  int *costi = calloc(a->nMin,sizeof(int));
  a->bestSum = -1.0;
  a->bestPlan = calloc(a->na,sizeof(int));
  disp_ripet(a,0,sol,costi);
  printf("La pianificazione e'\n");
  for (int i=0; i<a->na; i++)
    printf("%-20s: %d\n", a->elenco_att[i].nome, a->bestPlan[i]);
  free(sol);
  free(costi);
}
```