Laboratorio di Algoritmi e Strutture Dati
Docenti: M. Goldwurm, V. Lonati
Progetto “Die Hard”

Premessa: questo progetto dovrebbe risolvere il problema della gestione di n contenitori, che attraverso operazioni di riempimento, svuotamento e travaso, dovrebbero misurare una quantità esatta di acqua per disinnescare un ordigno esplosivo.
Ho quindi fatto delle assunzioni, a priori:

- il numero n dei contenitori che gestisce l'utente non è eccessivamente alto;
- le singole capacità inserite dall'utente non sono eccessivamente grandi.

Si presuppone infatti che la bilancia che misura la quantità d'acqua necessaria per il disinnesco della bomba sia una bilancia precisa, con un'alta sensibilità: quindi i contenitori non avranno un'altissima capacità (sono contenitori, non cisterne!); e si presuppone che “l'artificiere” gestisca un numero concreto e accettabile di contenitori.
Quindi, l'input medio previsto è fino a 5-6 contenitori da 10-15 lt. Naturalmente, se si abbassa il numero di contenitori, allora possono alzarsi le capacità dei singoli col giusto rapporto, e viceversa. Oltre questo limite il programma può diventare molto lento, sino ad un punto in cui può esaurire la memoria: si pensi solo che su input N 55 66 77 115 si possono generare fino a 3,537,455 possibili configurazioni dei contenitori. E' altamente improbabile, se non impossibile, che un artificiere debba disinnescare una bomba con contenitori di queste cifre.

Detto questo, passiamo alla fase di progettazione e implementazione del progetto.

Denotiamo con un vettore $a = (a_1, a_2, ..., a_n)$ di interi la configurazione dei livelli d’acqua nei contenitori, dove $0 \le a_i \le c_i$ per ogni $i = 1, \ldots, n$.
Ho inteso sia la sequenza di capacità, che le varie configurazioni, come array di interi.

L'idea implementativa del mio progetto si basa su una “funzione crescente”, che associa ad ogni nuova configurazione un nuovo id. Utilizzo poi tre strutture dati:

- un albero binario di ricerca, dove inserisco le nuove configurazioni: è l'ideale per la ricerca e l'inserimento, in quanto ha un costo logaritmico. Data una configurazione, io cerco il suo id nell'albero. Assumendo m come numero delle a configurazioni presenti nell'albero, ed n interi presenti nelle configurazioni:
  - costo della ricerca: O(n*log m);
  - costo dell'inserimento di una nuova configurazione: O(n*log m);
  - spazio occupato dall'albero: O(n*m);

- una tabella hashtable, implementata come un array di puntatori a configurazioni presenti nell'albero. Dato un id, io accedo direttamente alla configurazione, grazie alla mia hashtable. Essendo un array, conoscendo la posizione desiderata, ho accesso diretto:
  - costo della ricerca: O(1);
  - costo dell'inserimento di una nuova configurazione: O(1);
  - spazio occupato dalla tabella: O(m puntatori a configurazioni);

- un grafo orientato di interi, che sarebbero i singoli id che associo ad ogni nuova configurazione, implementato attraverso un array di liste di adiacenza.
  L'array di liste di adiacenze in posizione 0 è la lista di adiacenza del nodo 0, e così via.

  - costo della ricerca nella singola lista di adiacenza: O(m);
  - costo della ricerca della lista di adiacenza di un nodo, conoscendolo: O(1);
  - costo dell'inserimento di un nuovo lato: O(1);
  - spazio occupato dal grafo: O(n+m), dove n sono i nodi ed m i lati.

- una struttura contenitore, che comprende l'array delle singole capacità, e una configurazione, quella “attuale”, che l'utente modifica man mano con le operazioni elementari che svolge.

Analisi approfondita dell'albero.

Ogni struttura ha degli item: per l'albero ho creato degli item “esclusivi”, a suo solo uso, definiti nei files bist_itemkey. Questi sono composti appunto da un array di interi, e dal loro id. Gli array di interi sono le configurazioni. Gli id partono da 0 sino ad m, e provengono dalla “funzione crescente”.
In bist_itemkey.h ho anche dichiarato la mia funzione ID, come una variabile globale. E' una variabile globale che terrà conto delle configurazioni inserite nell'albero: ad ogni nuovo inserimento nell'albero, questa viene incrementata in bist_insert. Rappresenta la funzione che associa un id a una configurazione. Essendo un valore comune e condiviso in tutto il progetto, ed essendo anche una semplice variabile intera, ho deciso di dichiarare ID come variabile globale.
E' una quantità che cambia durante il corso del programma ad ogni nuovo inserimento nell'albero, e l'albero è la struttura “portante” del mio progetto, quindi ho deciso di dichiarare ID nell'interfaccia degli oggetti dell'albero. Siccome ne ho bisogno praticamente in tutti i file .c, ho deciso di dichiararla come variabile globale nell'header, inizializzarla a 0 al momento della creazione dell'albero, e incrementarla man mano con ogni inserimento.

Come chiavi degli item dell'albero, ho scelto di utilizzare le configurazioni, per bilanciare al massimo il mio albero, e ottenere un tempo il più logaritmico possibile per le mie operazioni.
La comparazione tra chiavi avviene scorrendo i singoli elementi dei due array: mi fermo appena trovo un numero in posizione i nel primo array che sia diverso dall' i-esimo nel secondo array. Precisamente restituirò una qtà negativa non appena trovo un i-esimo numero del primo array minore del secondo, o viceversa una qtà positiva. Se scorrendo tutti e due gli array (che hanno per forza stessa lunghezza, in quanto pari alla qtà di contenitori gestita dall'utente), non ho trovato un numero i diverso dall'altro, i due array sono uguali: quindi restituisco 0.
Questo è un tentativo di comparazione che vuole cercare di bilanciare al meglio un albero di ricerca binaria di configurazioni, per ottenere una ricerca che si avvicini il più possibile ad un costo O(log n) dove n = nodi inseriti. Infatti, se avessi scelto come chiave degli item dei nodi il loro id (cosa che potrebbe risultare più intuitiva forse) l'albero sarebbe comunque venuto uno schifo, completamente sbilanciato a destra, e la ricerca sarebbe stata esattamente O(n) (tanto quanto una lista!), in quanto gli id vanno in crescendo, da 0 a n-1. Non sarebbero quindi state sfruttate le proprietà di un albero di ricerca binaria!
Avrei potuto utilizzare una funzione random per assegnare l'id, anziché utilizzare una funzione crescente: per esempio, includere l'header time.h, inizializzare il seme srand e poi generare numeri. Ma sarebbero stati solo numeri pseudo-casuali, in quanto non posso avere la certezza al 100% che dopo n inserimenti, venga generato un id n+1-esimo diverso da tutti gli n precedenti, nonostante io abbia una configurazione n+1-esima diversa da tutte le n precedenti. Non posso quindi fare affidamento su srand.
Escludendo quindi la funzione random, mi rimane una "funzione crescente" che sono sicuro al 100% generi numeri da 0 a n-1 tutti diversi. Dovendo adottare questa funzione per generare gli id, ho scelto di adottare come chiavi le configurazioni. In questo modo l'albero è ugualmente

sbilanciato a destra (già partendo dalla radice, perchè come primo inserimento io inserisco un nodo con un item che ha una chiave $0[c_1]...0[c_n]$: tutti gli inserimenti successivi avverranno nel sottoalbero di destra). Ma comunque è molto meno sbilanciato che con l'altra ipotesi d'implementazione (id come chiavi), quindi è un ottimo compromesso che fa costare la mia ricerca quasi O(log n)!

Analisi approfondita della tabella hashtable.

Gli oggetti della hashTable sono sostanzialmente puntatori alle configurazioni. Dato un id, la mia table mi permette di accedere ad una configurazione. Questo per allocare UNA SOLA VOLTA in memoria lo spazio per una possibile configurazione (al momento dell'inserimento nell'albero). Ho scelto di implementare la tabella come un vettore, per le caratteristiche proprie dell'array: l'accesso diretto, che costa O(1). Essendo però l'array una struttura statica, ho scelto di effettuare una creazione di una tabella di dimensioni "massime", ed eseguire poi una realloc, solo al momento della conoscenza effettiva del numero di configurazioni create (che avviene alla fine di set_all), per liberare lo spazio inutilizzato.
Questo perchè l'input è comunque, in media, limitato.
Piuttosto che creare un'array di dimensioni minime, e riallocarlo ogni volta che raggiungevo la fine, ho scelto di allocare un array di dimensioni “massimali”, per ottenere un notevole vantaggio in termini di prestazioni, di velocità d'esecuzione, evitando tot realloc.
Avrei potuto scegliere di utilizzare una lista, dove non avrei dovuto utilizzare realloc: ma in una lista avrei dovuto costruire degli item appositi, che avessero anche loro id, oltre che puntatore a configurazione; inoltre, e soprattutto, avrei dovuto scorrere tutta lista alla ricerca dell'id cercato, ed avrei avuto quindi un costo di O(n) ogni volta si fosse presentata la necessità di accedere ad una configurazione, dato un id, sia per effettuare una ricerca che un inserimento.
Ma a me la tabella serve proprio per accedere alle configurazioni, è la sua funzione principale.
Invece, sfruttando il fatto che ho definito ID come variabile globale, e quindi ho sempre pieno accesso a questa, posso sfruttarla accedendo direttamente alla posizione desiderata, eseguendo un'operazione tipo hashTable->configurazioni[id]. Questo, ripeto, a scapito di un costo x*O(realloc), perchè per creare la mia hashTable calcolo il numero massimo di configurazioni possibili, eseguendo il prodotto di tutte le mie capacità (limite massimo certamente irraggiungibile: sto creando più spazio di quello che me ne serve!). Una volta che avrò generato tutte le mie possibili configurazioni, in base alle capacità inserite dall'utente, con una funzione sistema apposta per la mia table, successivamente, deallocherò lo spazio inutilizzato.

Analisi approfondita del grafo.

Il grafo orientato (così come la lista bidirezionale con cui è implementato e la coda per le visite) utilizza come item interi: questi interi sarebbero gli id, a cui corrispondono le configurazioni generate dalle singole capacità inserite dall'utente: queste capacità sono una quantità esattamente ID (poichè io incremento ID ad ogni inserimento nell'albero binario, e l'albero controlla che non vengano inseriti due oggetti con stessa chiave, ovvero con stessa configurazione). Quindi io, non conoscendo il numero dei nodi del grafo al momento della sua creazione (che avviene in case N nel main, assieme alle altre strutture), eseguo la stessa procedura che per hashTable: alloco per il grafo uno spazio massimo pari a hash->size (perchè in hash->size, per creare la Table, faccio già il calcolo della quantità massima delle mie permutazioni).
Eseguo questa procedura perchè, nel limite del possibile, mi aspetto un input “limitato”.
So di certo che il numero di configurazioni generabili dalle capacità che inserisce l'utente, è certamente minore del prodotto di tutte le capacità(+1 per contare anche lo zero come stadio possibile di un contenitore in una configurazione). So già che non userò tutto questo spazio, così per il grafo come per la mia tabella: quindi ho creato due procedure sistema (una per la table, l'altra per

il grafo), che riallochino esattamente uno spazio pari a ID interi per il grafo (e puntatori per la table), così da liberare lo spazio allocato alla creazione della struttura, ma non utilizzato perchè era in più. Questo è più vantaggioso che effettuare ID realloc, ognuna al momento di un inserimento di un id nel grafo (e nella table un puntatore a configurazione) per aver inserito una nuova configurazione nell'albero.

Il vantaggio nel popolare il grafo con id sta anche, e soprattutto, nella facilità del gestire un grafo di interi.
Ricordo che il grafo è implementato attraverso liste di adiacenza "bidirezionali" (con un puntatore ala testa e alla coda), che effettuano un inserimento sia in testa che in coda in O(1).

Programma.

Quando l'utente inizializza una sequenza di contenitori, io in case N creo tutte le mie strutture, in ordine di "importanza": l'albero, la tabella hash e il grafo.
Poi chiamo una procedura, set_all, che prende come parametri le strutture, e le riempie con tutte le possibili configurazioni che si possono generare dalle capacità che ha inserito l'utente, utilizzando a cascata le funzioni di riempimento, svuotamento e travaso sulle configurazioni che man mano genero, a partire da quella iniziale 0...0.
Questa è la soluzione più ottimale che sono riuscito a trovare: partendo dalla configurazione iniziale 0...0, la inserisco nell'albero (in un tempo logaritmico), allocando effettivamente spazio in memoria e inserisco nella tabella un puntatore ad essa, nella posizione 0 (posizione data dal suo id = 0), in un tempo costante.
Ad ogni nuovo inserimento nell'albero, la funzione id è incrementata di uno, cosicché è sempre pronto il prossimo id disponibile.
Successivamente prelevo una ad una le configurazioni dalla tabella (all'inizio ci sarà solo la prima 0...0), e per tutte le operazioni elementari possibili che riesco ad effettuare sugli n contenitori, se genero una nuova configurazione (perché controllo che non sia presente nell'albero, in un tempo logaritmico), allora creo un nuovo item apposito, lo inserisco nell'albero, in tabella, e nel grafo inserisco un lato dalla configurazione su cui sto facendo tutte le operazioni a quella che genero per ultima.
Da notare che io manipolo di volta in volta solo la configurazione presente nella struttura contenitore, e solo se serve alloco spazio in memoria.

Ma passiamo in concreto all'analisi delle funzioni richieste dal progetto.

– Contenitore crea_contenitori(int n, int *C).

In crea_contenitori io creo in memoria lo spazio per una variabile di tipo contenitore. Successivamente creo spazio per due array di n interi, le capacità c e la configurazione. Pongo gli a_i a 0, tramite la calloc, e i c_i vengono copiati dall'array passato come parametro.
Tramite una funzione poi, cerco la massima tra le capacità, e la assegno a max, terzo campo della variabile contenitore. Eseguo quindi 3 volte n assegnamenti, per un costo di O(n).

– int riempi(Contenitore cont, int i).

Eseguo un controllo, e se questo ha esito positivo faccio un assegnamento e ritorno un valore positivo, altrimenti un valore nullo. Questo ha un costo di O(1).

- int svuota(Contenitore cont, int i).

Stessa cosa di riempi(i).

– int travasa(Contenitore cont, int i, int j).

E' la più costosa delle tre operazioni elementari. Eseguo in sostanza un ciclo, dove decremento il contenitore i e incremento il contenitore j, sino a che i non è vuoto o j non è pieno. Il costo dell'operazione è dato quindi da O(Min{c_i , c_j}), dove c_i, c_j sono le capacità del contenitore i-esimo e j-esimo.

– void visualizza(Capacita c, Config a, int n).

Esegue una stampa. Il costo è dato principalmente dal ciclo eseguito n volte, dove si stampano a_i e c_i per ogni i = 1. . n. Il costo finale è quindi O(n).

– void esiste(unsigned short int *esist, int k).

La funzione esiste merita una breve spiegazione della sua implementazione.
Per l'implementazione di esiste(k) sfrutto il fatto che non può esistere un livello k di un qualsiasi contenitore maggiore della massima capacità introdotta dall'utente. Quindi creo un vettore esistenze, lungo quanto la massima capacità inserita dall'utente +1. Questo perchè esistenze dev'essere un array che ha come indici da 0 alla capacità massima (es.: N 3 5 -->> esistenze[0]...esistenze[5]), e contiene 1 o 0:

- 1 se l'indice dell'array è un livello che è stato raggiunto da almeno un contenitore;
- 0 altrimenti.

Il vettore esistenze poi rimane allocato nel main (insieme a tutte le mie altre strutture "di supporto"), sino a che l'utente non inserisce una nuova sequenza.
E' un'ottima soluzione, in quanto il costo di un inserimento in un array è praticamente nullo, perchè avviene tramite accesso diretto (e gli inserimenti in esistenze[] avvengono in set_all(), quando popolo tutte le mie strutture).
Successivamente, verificare lo stato di una posizione dell'array conoscendone l'indice, è ancora nullo! Questo è quello che fa esiste: in pratica, il costo di esiste(k) è O(1); lo spazio che richiede esistenze sarà O(contenitore->max+1).
Questa è un'ottima implementazione di esiste, se pensata per essere eseguita più di una volta! Piuttosto che scorrere la mia tabella, e verificare se esista o meno uno stadio k tra le mie m configurazioni di lunghezza n! Meglio O(1) che O(m*n), anche se devo allocare un array di interi, proporzionale comunque alla massima capacità: comunque lo alloco UNA SOLA VOLTA nel main, e mi rimane come struttura d'appoggio, assieme a tutte le mie altre strutture.

– void raggiungibile(Bist_Tree tree, Config a).

Io riempio un albero binario di ricerca con tutte le configurazioni possibili che sono riuscito a generare. Quindi per eseguire raggiungibile(a[]) semplicemente ricerco nel mio albero, già implementato in case N: se a è una chiave già presente nell'albero, stamperò SI', oppure NO.
La ricerca costerà O(log m), dove m sono tutte le possibili configurazioni.

– void configurazioni(Contenitore cont, Bist_Tree tree, HashTable hash, Graph graph, int d).

Per configurazioni utilizzo la tabella HashTable e il grafo di id. Se l'utente vuole vedere che configurazioni può raggiungere dalla configurazione attuale, prima recupero l'id della configurazione a cui l'utente è arrivato dall'albero. Poi chiamo una procedura ricorsiva che si basa su una visita in profondità del grafo, e che stampa appunto le configurazioni (recuperandole dai puntatori nella hashtable) grazie agli id visitati nel grafo.
N.B.: se l'utente digita 0, non eseguo nemmeno la funzione ricorsiva: l'unica posizione raggiungibile con 0 mosse è quella dove sono!
Il costo di configurazioni è dato da: la ricerca dell'albero + l'allocazione di un array di n_V interi, a 0 (che utilizzo come array delle “marche” dei nodi), dove n_V sono i nodi del grafo + il costo di

graph_config (la procedura ricorsiva). Quindi O(log n_V) + O(n_V) + O(m_E) = O(n_V+m_E);
N.B.: il costo di recupero di una configurazione, dato un id è O(1), perchè la tabella hash è un vettore.
Lo spazio che richiederà configurazioni, quindi, sarà nel caso peggiore quello richiesto dallo stack per la ricorsione, più ovviamente l'array delle marche: O(n) + O(n) = O(n).

Particolare attenzione però dedica il case N: esso è la parte del programma che esegue le operazioni più costose. Ho scelto appunto questa struttura del progetto, poiché generalmente un utente esegue una/due volte massimo un case N, mentre hanno una frequenza maggiore tutte le altre operazioni. Quindi, se il resto del programma viene eseguito in un tempo molto veloce, quasi istantaneo, il case N richiede un costo operazionale maggiore.
Infatti io nel case N:
- Memorizzo i numeri letti da input dinamicamente;
- Se l'utente ha già inserito una sequenza di contenitori e vuole inserirne una nuova, cancello e dealloco tutte le strutture precedentemente create;
- Creo una nuova sequenza di contenitori;
- Creo il mio albero di configurazioni;
- Creo la mia tabella (di massima capacità) di puntatori a configurazioni presenti nell'albero;
- Creo un grafo (di massima capacità) di id delle configurazioni presenti nell'albero;
- Creo un vettore lungo quanto la massima capacità inserita dall'utente;
- Riempio le mie strutture con set_all(cont, tree, hash, graph, esistenze).

E' chiaro che il costo preso da un case N dipende fortemente dal riempimento di tutte le mie strutture.
Analizzando più in dettaglio set_all():

- inserisco la configurazione iniziale 0...0 nell'albero, e un puntatore ad essa nella tabella;
- per tutte le configurazioni m che riesco a generare:
  - estraggo una configurazione dalla tabella hash, in O(1).
  - per tutti gli n contenitori:
    - riprisitino il contenitore alla configurazione i-esima perchè viene continuamente modificato ad ogni iterazione dei cicli più interni (perchè alloco memoria solo se effettivamente mi serve, cioè solo se una configurazione non è presente nell'albero. Il problema sta nel fatto che già dopo riempi(1), la mia configurazione iniziale da 0...0 diventerà 1...0; poi però io devo ritornare alla configurazione "originaria" (i-esima) per eseguire le restanti operazioni: riempi(2), riempi(3)...riempi(n)...svuota(n)...travasa(n,k). Quindi eseguo ripristina, in O(n).
    - pongo a 1 la posizione dello stadio j-esimo, nell'array delle esistenze, in O(1).
    - Eseguo riempi, in O(1) e se ha esito positivo
      - cerco la configurazione appena creata, per vedere se c'è già, nell'albero, in O(n*log m).
      - se non presente, la inserisco nell'albero in O(n*log m), e inserisco un puntatore alla configurazione nella tabella in O(1).
      - altrimenti, se la ricerca non ha esito negativo, vuol dire che nell'albero era già presente; in ogni caso, che sia appena stata creata o era già presente, recupero il suo id, e inserisco un arco tra l'id della configurazione attuale e questo, in O(1).
    - Eseguo poi le stesse operazioni per svuota per travasa.

Essendo travasa l'operazione elementare più costosa, l'ordine di grandezza di set_all sarà dato

dall'ordine di grandezza delle operazioni eseguite nel ciclo competente a travasa.
Io nel caso peggiore travaserò tutti i contenitori in tutti i contenitori, quindi:
$O(m) * \{ O(n*n) * [O(n) + \text{Min}\{c_i , c_j\} + 2*O(n*\log m)] \} = O(m*\log n*n^3)$.
m sono tutte le possibili configurazioni, n sono tutti i contenitori.

Compilare con:
gcc Main.c contenitore.c bist_Tree.c bist_itemkey.c hashTable.c graph.c itemkey_int.c queue.c bid_list.c my_alloc_IO.c arrays.c

/* MICHELE CORRIAS MAT. 741863 */