Algoritmi e Strategie Risolutive

corso regionale di preparazione per le OII

Giorgio Audrito

Università degli Studi di Torino

30 Settembre 2010

Perchè devo usare il C/C++?

- Maggiore rapidità di scrittura del codice.
- Maggiore rapidità di esecuzione del codice.
- Il pascal non lo usa più nessuno.
- Possibilità di comunicazione con gli altri programmatori.
- Alcune funzioni importanti già presenti nella libreria di base `stdlib` (*qsort*, ricerca binaria).
- Possibilità di utilizzare le STL e accedere quindi a strutture dati complesse già implementate.

Perchè devo usare Linux/MacOS?

- Nel caso di Linux, è gratis e *free*.
- Ha gli strumenti di compilazione/editing del software integrati con il sistema operativo.
- È l'ambiente che viene usato per la correzione dei compiti nelle gare, quindi non rischiate sorprese.
- Windows è già stato eliminato dalle IOI e probabilmente presto lo sarà anche dalle OII.
- Perchè sì.

Riferimenti web

http://www.algoritmica.org/

http://www.sgi.com/tech/stl/

http://correttore.olimpiadi-informatica.it/

http://orientamento.educ.di.unito.it/

http://www.ubuntu-it.org/

Un problema di riferimento: *Ladro*

Situazione

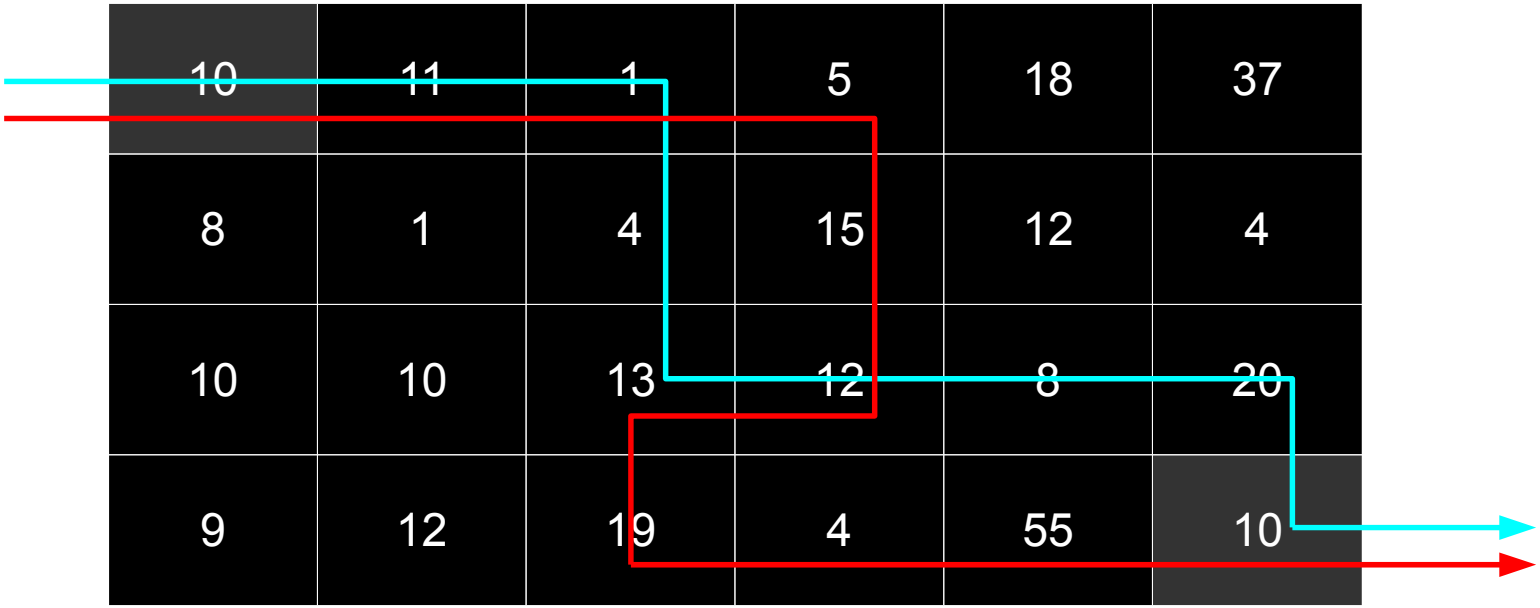
- Un ladro deve rapinare un castello di $m \times n$ stanze.
- L'ingresso è nella stanza $(1; 1)$ e l'uscita è nella stanza $(m; n)$.
- In ogni stanza ci sono V_{ij} euro derubabili.
- In ogni stanza ci si può muovere a est o a sud.

Obbiettivo

Massimizzare il valore rubato.

Un problema di riferimento: *Ladro*

Esempio



Nota

Il percorso blu è valido, quello rosso invece no.

Tecnica Greedy

Cos'è?

Ogni volta che bisogna prendere una scelta, si sceglie la cosa al momento più promettente (tecnica *golosa*)

Caratteristiche

- Spesso è semplice da intuire.
- Quasi sempre è molto veloce in esecuzione.
- Purtroppo, di solito è sbagliata.

Tecnica Greedy

Esempi in cui funziona

- Segmento di somma massima in un array.
- Massimo numero di intervalli disgiunti (problema *Nimbus*).
- Scheduling (problema *Missioni*).
- Problema *Numeri Antipatici*.

Questi problemi sono già stati trattati nel precedente corso e sono disponibili come slide sul sito del corso.

Tecnica Greedy

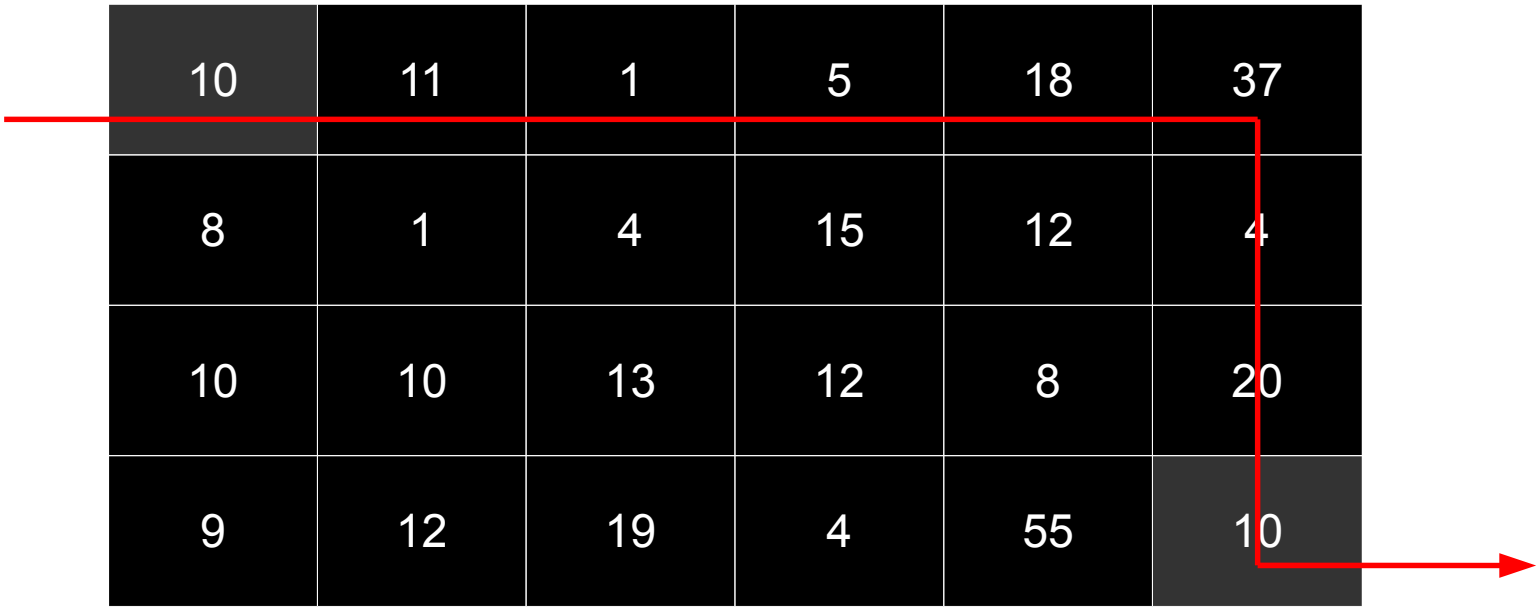
E per il problema *Ladro*?

10	11	1	5	18	37
8	1	4	15	12	4
10	10	13	12	8	20
9	12	19	4	55	10

Che soluzione troverà
l'algoritmo greedy?

Tecnica Greedy

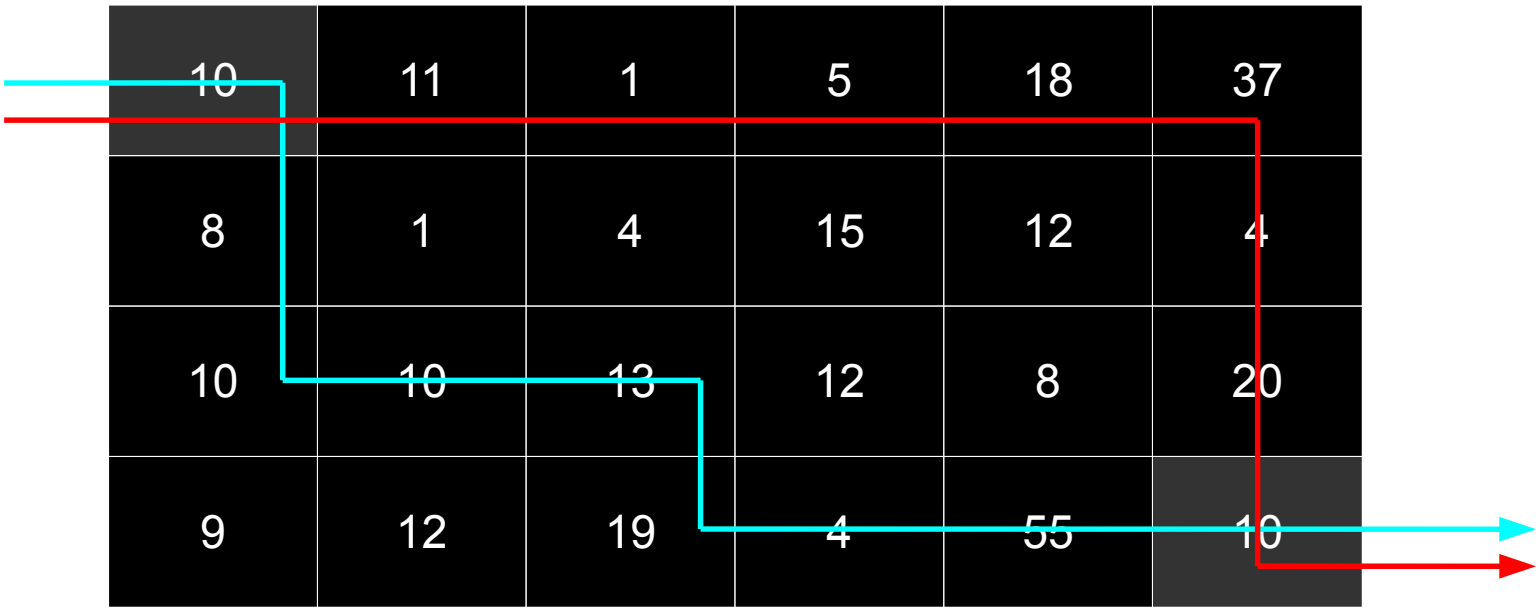
E per il problema *Ladro*?



Che soluzione troverà
l'algoritmo greedy?

Tecnica Greedy

E per il problema *Ladro*?



La soluzione trovata con la tecnica greedy (rossa) non è ottimale: si ha una somma di 116 anzichè 139 del percorso ottimo (blu).

Tecnica Greedy

E poteva andare anche peggio!!

1	1	1000	1000	1000	1000
2	1	1	1	1	1
1	1	1	1	1	1
1	1	1	1	1	1

In questo caso con l'algoritmo greedy trova una somma di 10 anzichè 4005 del percorso ottimo.

Tecnica Greedy

Proviamo a cambiare strategia.

Tecnica Esaustiva (*Backtracking*)

Cos'è?

Ogni volta che bisogna prendere una scelta, si provano tutte le possibilità una alla volta, finché non si trova una soluzione.

Caratteristiche

- Spesso è semplice da intuire.
- È sempre corretto, salvo errori di implementazione.
- Purtroppo, di solito è estremamente lento in esecuzione, fino a diventare praticamente intrattabile.

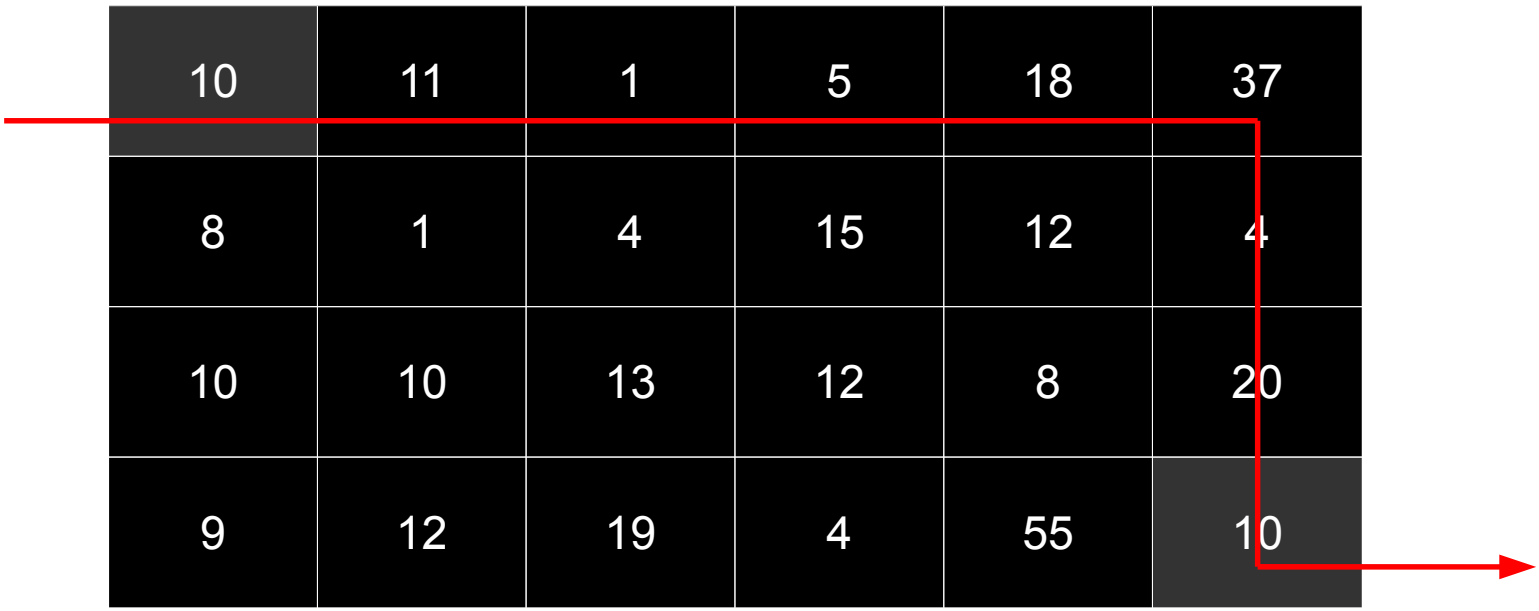
Tecnica Esaustiva (*Backtracking*)

Esempi già visti nel corso

- Il problema delle otto regine.
- Nient'altro, comunque qualunque problema con istanze abbastanza piccole è un valido esempio.

Tecnica Esaustiva (*Backtracking*)

E per il problema *Ladro*?



Dato che ogni percorso è formato da 6 passi a est e 4 passi a sud, l'algoritmo esaustivo impiegherà $\frac{10!}{6!4!} = 210$ passi (pari al numero di percorsi).

Tecnica Esaustiva (*Backtracking*)

In questa specifica istanza del problema l'algoritmo esaustivo è in grado di trovare la soluzione esatta in tempi ragionevoli.

Cosa succede se si ingrandisce l'input?

Se prendiamo un castello di dimensione per esempio 40×40, l'algoritmo esaustivo impiega ben $2^{40} \approx 10^{12}$, mille miliardi di passi!

Tecnica Esaustiva (*Backtracking*)

Proviamo di nuovo a cambiare strategia.

Divide et Impera

Cos'è?

Questa volta non procediamo per scelte successive.
Invece, a ogni passo suddividiamo il problema in due problemi più piccoli, risolviamo separatamente i due problemi, e poi ricombiniamo le soluzioni.

Caratteristiche

- Quasi sempre è veloce in esecuzione.
- Purtroppo, di solito è difficile da intuire.
- Purtroppo, di solito non è nemmeno applicabile.

Divide et Impera

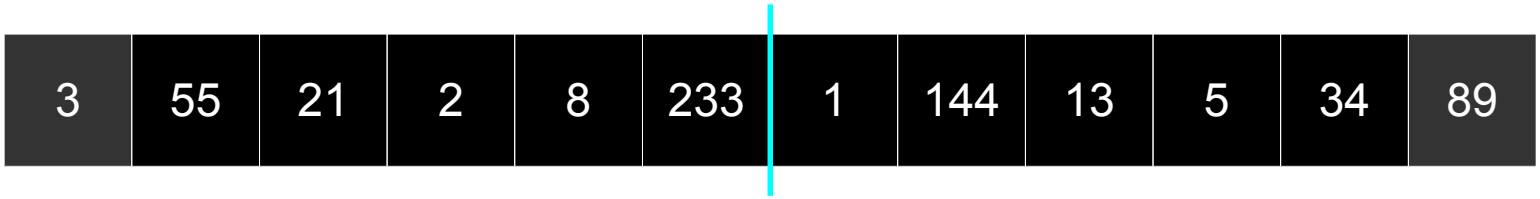
Esempi in cui funziona

- Ordinamento (algoritmo *Merge Sort*).
- Ricerca binaria.
- In generale, in tutti i problemi che si possono suddividere in sotto-problemi indipendenti.

Vediamo qualche esempio.

Divide et Impera

Merge Sort



Obbiettivo

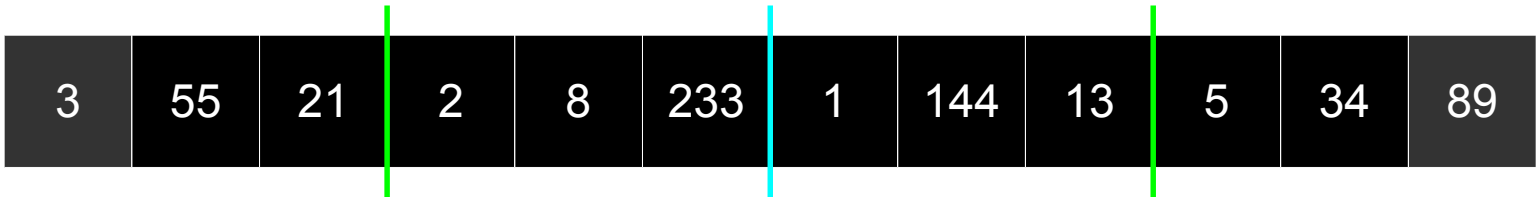
Dato un array $V[1..N]$, ordinare i suoi elementi in ordine crescente.

Intuizione

Se dividiamo l'array a metà e ordiniamo le due parti, queste si possono poi facilmente mettere insieme e ottenere l'ordine desiderato (*merge*).

Divide et Impera

Merge Sort

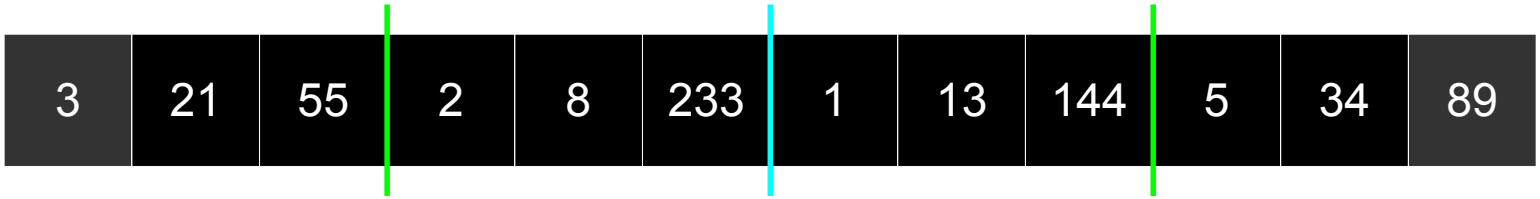


Ricorsione

Per ciascuna delle parti che dobbiamo ordinare, utilizziamo lo stesso algoritmo *merge sort*, che le suddivide di nuovo.

Divide et Impera

Merge Sort

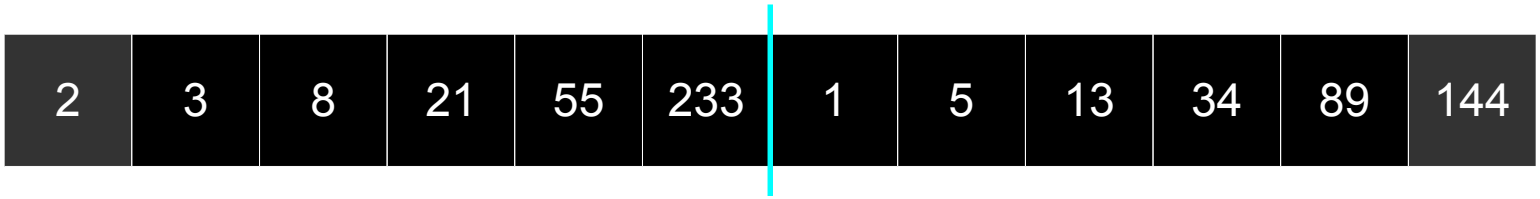


Ricorsione

La suddivisione prosegue finchè possibile, poi si iniziano a ordinare le parti più piccole con le prime applicazioni di *merge*.

Divide et Impera

Merge Sort



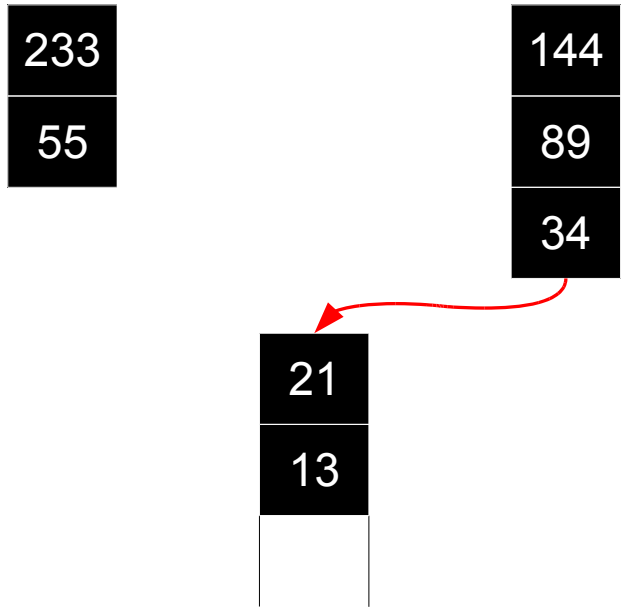
Ricorsione

Tramite il processo di *merge*, diventano ordinate anche le parti progressivamente più grandi.

Divide et Impera

Merge Sort

Seguiamo l'ultima applicazione del *merge*.



Le due metà vengono ricombinate togliendo ogni volta l'elemento più piccolo tra le due e aggiungendolo al vettore del risultato.

Divide et Impera

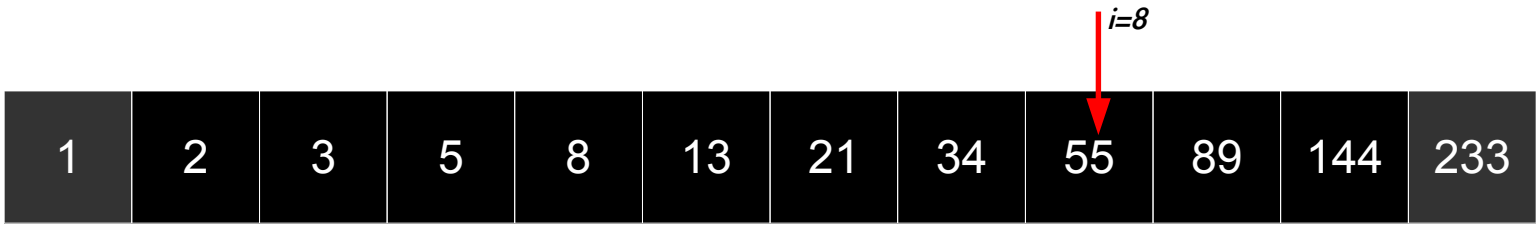
Merge Sort

1	2	3	5	8	13	21	34	55	89	144	233

A questo punto il vettore sarà ordinato, e per esempio possiamo utilizzarlo per effettuare delle ricerche con l'algoritmo di *ricerca binaria*.

Divide et Impera

Ricerca binaria



Obbiettivo

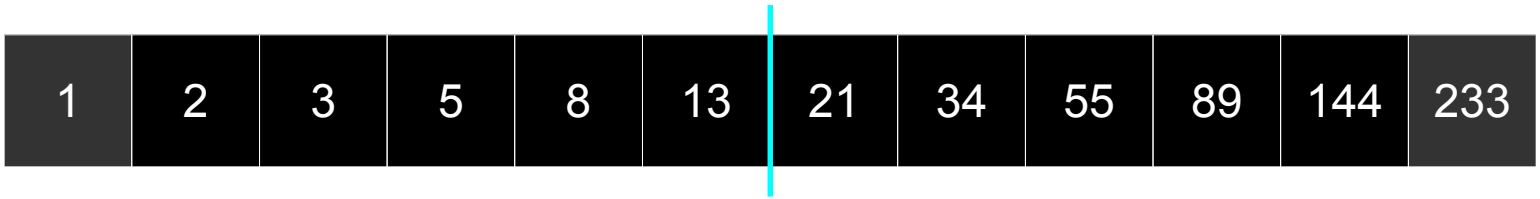
Dato un array ordinato $V[1..N]$ e un valore K, trovare la posizione nell'array in cui si trova K (se è presente).
In questo caso, $K = 55$ e il risultato è $i = 8$.

Intuizione

Se dividiamo l'array a metà e capiamo in quale delle due parti si può trovare il numero K cercato, è poi sufficente cercarlo all'interno di quella parte.

Divide et Impera

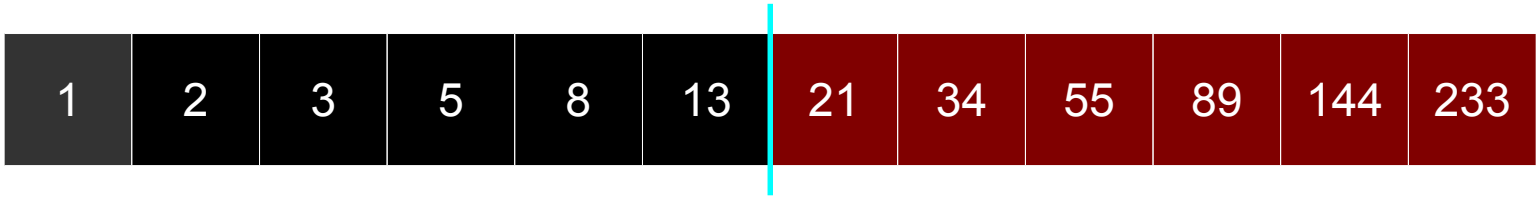
Ricerca binaria



Dividiamo quindi in due l'array, e per capire dove cercare confrontiamo K con il valore al centro del vettore.

Divide et Impera

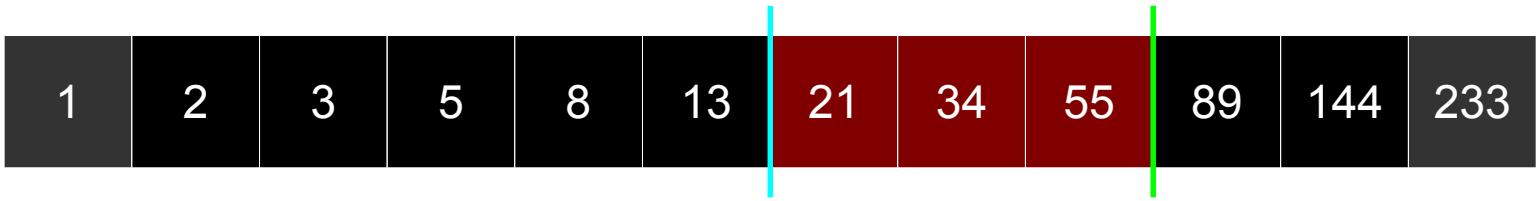
Ricerca binaria



Individuamo in questo modo la metà dell'array in cui il valore cercato K potrebbe trovarsi.

Divide et Impera

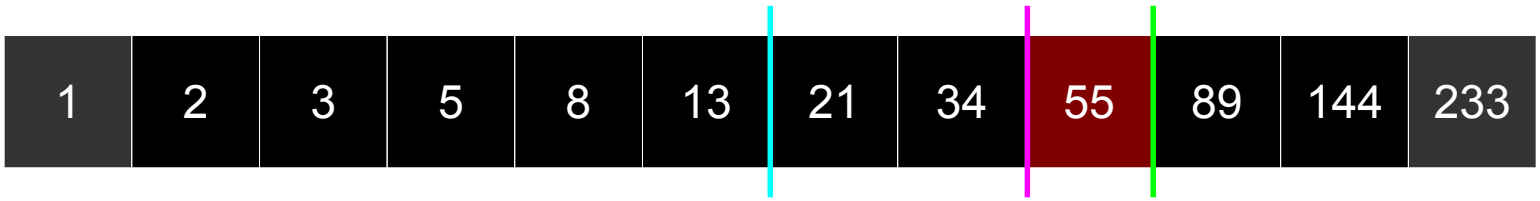
Ricerca binaria



Per cercare il valore K nella metà scelta utilizziamo lo stesso algoritmo, che la suddivide di nuovo.

Divide et Impera

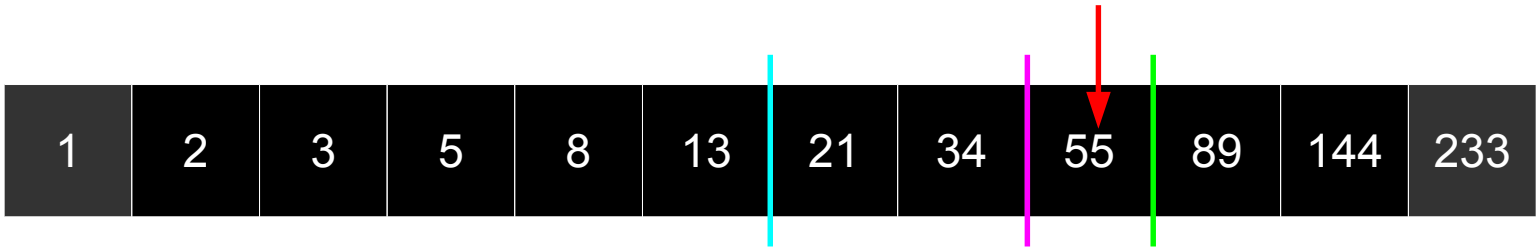
Ricerca binaria



Si continua in questo modo a suddividere, finchè non ci si riduce a una parte contenente un unico numero.

Divide et Impera

Ricerca binaria



A questo punto o si è trovato l'obbiettivo K, oppure si è certi che questo non è presente nell'array.

Confronto Backtracking/Divide et Impera

Somiglianze

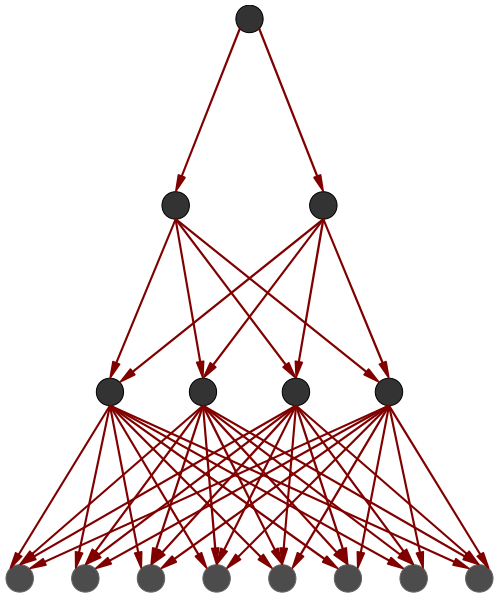
- Da un unico problema ci si riconduce a un certo numero di problemi più piccoli.
- Si risolvono i problemi più piccoli con chiamate *ricorsive* dell'algoritmo.
- Entrambi gli algoritmi effettuano tutte le sequenze di scomposizioni possibili.

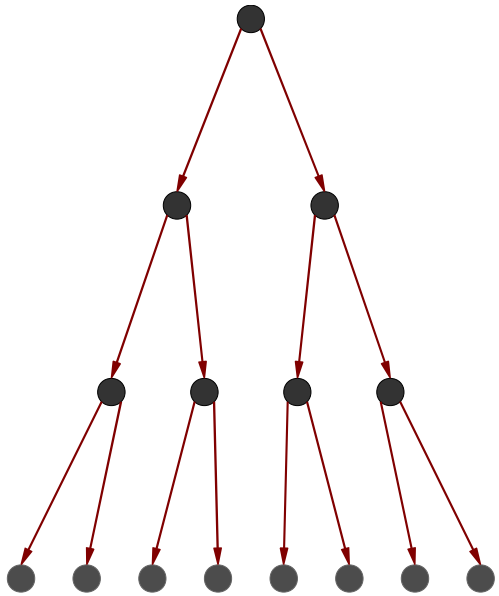
Differenze

- I problemi a cui ci si riconduce sono separati e indipendenti nel Divide et Impera, mentre sono interconnessi e dipendenti nel Backtracking.
- Per questo motivo, nel Backtracking le possibili sequenze di scomposizioni sono moltissime mentre nel Divide et Impera sono poche, da cui l'enorme differenza nei tempi di esecuzione.

Confronto Backtracking/Divide et Impera

Cerchiamo di visualizzare la differenza con un grafico (*grafico delle chiamate ricorsive*).





Ogni punto rappresenta un diverso sottoproblema, mentre le linee indicano a quali sottoproblemi più piccoli si riconduce un sottoproblema più grande.
A sinistra un tipico Backtracking, a destra un tipico Divide et Impera.

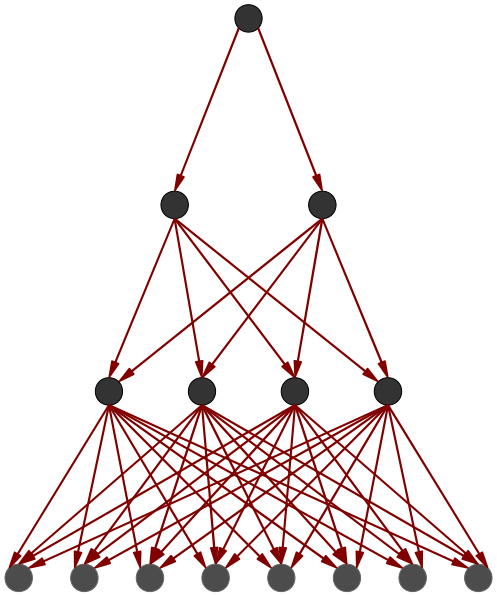
Confronto Backtracking/Divide et Impera

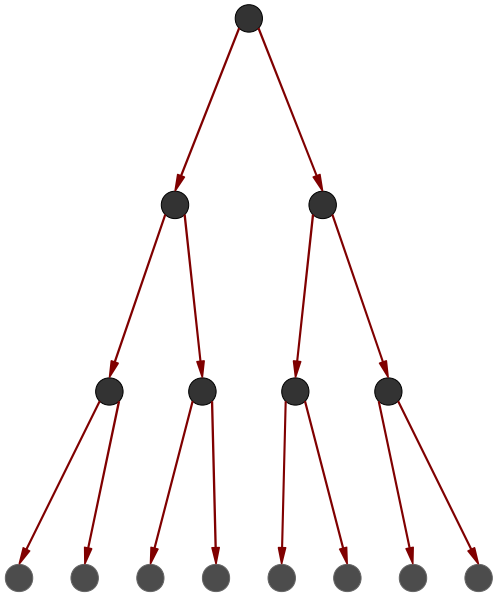
C'è un modo per mettere insieme i vantaggi del Divide et Impera con la generalità del Backtracking?

Fortunatamente, sì.

Confronto Backtracking/Divide et Impera

Osserviamo meglio il grafico delle chiamate ricorsive.





Il Backtracking perde molto tempo ricalcolando le soluzioni agli stessi problemi. Cerchiamo di evitare di farlo così da ottenere una velocità simile a quella del Divide et Impera.

Ricorsione con Memorizzazione

Cos'è?

Ogni volta che bisogna prendere una scelta, si provano tutte le possibilità una alla volta, finché non si trova una soluzione.
Nel frattempo però si memorizzano i risultati dei sottoproblemi in una tabella, utilizzandoli poi per evitare di ricalcolare gli stessi sottoproblemi più volte.

Caratteristiche

- In linea di principio è sempre applicabile.
- Spesso è veloce in esecuzione.
- Purtroppo, di solito è difficile da intuire.

Ricorsione con Memorizzazione

Individuamo una scaletta per cercare di ottenere un algoritmo ricorsivo con memorizzazione.

Primo punto della Programmazione Dinamica

1. Identificare i sottoproblemi che è necessario considerare.

Nel problema *Ladro*

Trovare i percorsi di massimo guadagno da una qualunque casella alla fine.

Ricorsione con Memorizzazione

Individuamo una scaletta per cercare di ottenere un algoritmo ricorsivo con memorizzazione.

Secondo punto della Programmazione Dinamica

2. Trovare dei parametri che siano in grado di identificare univocamente i sottoproblemi.

Nel problema *Ladro*

Le due coordinate x, y della casella di partenza.

Ricorsione con Memorizzazione

Implementazione

- Allocare una tabella $M[..]$ indicizzata con tutti i parametri necessari (nel nostro caso x e y).
- Inizializzarla con un valore "impossibile" che consenta di identificare quali caselle non sono ancora state calcolate (solitamente -1).
- Scrivere una funzione ricorsiva analogamente al Backtracking (nel nostro caso `migliorPercorso(x,y)`).
- Inserire all'inizio della funzione un test che verifichi se il risultato è già stato calcolato:
  `if (M[x][y] != -1) return M[x][y];`
- Sostituire la riga di ritorno `return r` aggiungendoci la memorizzazione `return (M[x][y] = r)`.

Dinamica

La strategia appena vista viene chiamata *Ricorsione con Memorizzazione*.
La vera e propria Programmazione Dinamica richiede ancora un terzo punto:

Terzo punto della Programmazione Dinamica

3. Trovare un ordine in cui riempire la tabella $M[..]$, di modo che per il calcolo di un qualsiasi elemento siano sufficenti le caselle della tabella riempite prima di esso.

Nel problema *Ladro*

Il calcolo di $M[x][y]$ si riconduce a $M[x+1, y]$ e $M[x, y+1]$.
L'ordine cercato è quindi $x = X_{max}..0$, $y = Y_{max}..0$.

Dinamica

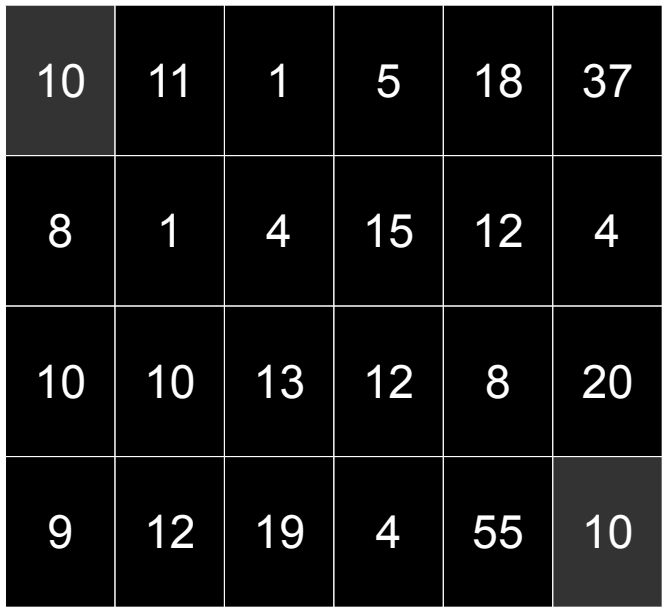
Una volta risolto questo terzo punto, si può evitare totalmente la ricorsione riempiendo la tabella nell'ordine trovato con dei cicli `for`.

Vantaggi

- Evitando la ricorsione, si risparmia un piccolo ammontare di tempo di esecuzione (meno chiamate a funzione).
- Evitando la ricorsione, l'algoritmo risulta più compatto e più veloce da scrivere.
- Riempiendo la tabella in modo sequenziale si possono sfruttare delle tecniche (che vedremo più avanti) per ridurre notevolmente l'utilizzo della memoria.

Dinamica

Esempio



La tabella che l'algoritmo di programmazione dinamica riempie per il problema *Ladro*

Conclusioni

Tre approcci alla risoluzione di un problema:

- Greedy: veloce ma spesso sbagliato
- Backtracking: sempre giusto ma troppo lento
- Divide et Impera: veloce ma raramente applicabile

Conclusioni

Scaletta per la soluzione dinamica di un problema:

1. Identificare i sottoproblemi che è necessario considerare.
2. Trovare dei parametri che determinino univocamente i sottoproblemi.
3. Trovare un ordine in cui riempire la tabella $M[..]$, di modo che per il calcolo di un qualsiasi elemento siano sufficenti le caselle della tabella riempite prima di esso.