Tema 4:Branch&Bound:

Algorisme que avaluà solucions parcials de forma paral·lela, consisteix a buscar mitjançant camins parcials + heurística, sempre respectant les restriccions.

Sempre dona una solució optima, velocitat dependent de l'heurística.

Backtracking VS Branch&Bound:

Backtracking intenta completar una única solució. Si no pot, elimina l'últim element que ha afegit a la solució i prova a afegir un altre element en canvi B&B treballa amb vàries solucions parcials a les que afegeix elements per completar-les.

Tipus de nodes:

-Node viu: no s'han generat tots el fills

-Node mort: node que ja no generarà mes fills per que no hi ha, viola restriccions o solucio pitjor a actual millor.

-Node en expansió: s'estan generant fills.

Es generen tots els fills del node en curs abans que qualsevol altre node viu passi a ser el nou node en curs, es selecciona el següent mitjançant FIFO(cua, per amplada) o LIFO(pila, de dreta a esquerra) o amb cua amb prioritat(Amb Heurística).

```
repetir
    expandir el node viu més prometedor;
    generar tots els seus fills;
    un cop generats, es mata el pare;
    per a cada fill fer
        si te un cost esperat pitjor que el de la millor solució en curs
            llavors es mata
        sino_si té un cost esperat millor que el de la millor
                solució en curs i no és solució
            llavors es passa a la llista de nodes vius
        sino //té un cost esperat millor que el de la millor solució en
                curs i és solució (el cost no és estimat sinó real)
            passa a ser la millor solució en curs i es revisa tota la llista
            de nodes vius, eliminant els que prometen alguna cosa
            pitjor de l'aconseguit
        fisi
    fiper
mentre la llista no estigui buida
```

OBSERVACIONS

-Per poder escriure tot el camí des de l'arrel fins a la solució, quan afegim un node a la cua, hem de desar a una taula auxiliar de pares de nodes aquest node amb el seu pare.

-La finalització de l'algorisme només està garantida si l'espai d'estats és finit.

-Si l'espai d'estats és infinit i existeix al menys una solució es pot garantir la finalització amb una tria adient de la funció d'estimació.

-Si l'espai d'estats és infinit i no hi ha solució, l'algorisme no acaba…

-S'acostuma a restringir la cerca a nodes amb cost estimat menor que C.

Tema 5 Dinamica I_II

Programació Dinamica vs Divide and Concoquer

PD i D&C s'assemblen en que els dos divideixen el problema a resoldre en subproblemes més simples amb la mateixa estructura, i la solució del problema és la composició de les solucions els subproblemes. PD i D&C es diferencien en que mentre D&C necessita que els subproblemes siguin disjunts PD necessita que siguin dependents, i per no fer recàlculs innecessaris reaprofita els càlculs comuns, mitjançant una taula o vector.

Això ho veiem a fibbonnacci, a on si dividim el problema en subproblemes més simples: per exemple fib(n)=fib(n-1)+fib(n-2), tenen molts càlculs compartits. Per això veiem que, tot i que es pot solucionar el problema utilitzant les dues tècniques, PD és la tècnica adequada mentre D&C té una complexitat innecessàriament elevada pel fet de no dividir el problema en subproblemes disjunts i per tant haver de fer aquest recàlcul.

Programació dinàmica

- S'utilitza usualment per resoldre problemes d'optimització
- Permet resoldre problemes mitjançant una seqüència de decisions
- A diferència de l'esquema *greedy*, es produeixen vàries seqüències de decisions i només al final es sap quina és la millor
- Està basada en el **principi d'optimalitat de Bellman**:
  "En una seqüència de decisions òptima tota subseqüència ha de ser també òptima"

-Subproblemes Simples: Hem de poder dividir el problema original en subproblemes més petits amb la mateixa estructura
-Subestructura òptima dels problemes (Bellman): La solució al problema ha de ser la composició de les solucions dels subproblemes
-Subproblemes Dependents:L'espai de solucions del diferents subproblemes no és disjunt

Estratègies:

Top-down: Només resol els subproblemes estrictament necessaris per trobar la solució. Es recursiu.

Bottom-up: L'enfocament es millor que top-down amb el problema que potser es tenen en compte més sub-problemes dels necessaris. L'enfocament és lleugerament millor quant a cost temporal o espacial. Es iteratiu

```
funcio fib(n:enter)
   var f:vector[n]
   f[0]=0; f[1]=1;
   per i=2 fins n fer
     f[i]=f[i-2]+f[i-1]
   fiper
   retorna f[n]
ffuncio
```

distànciade Hamming: només substitució(noméss'aplicaa les cadenesde caràcters de la mateixa longitud).
1011101 y **1001001** es 2.

La **distància de Levenshtein** entre dues cadenes xi y, *D(x,y),*és el mínim nombre d'operacions d'edició que cal fer per transformar $x$en y. **d(X,Y) = min{c(*S*), *S* seqüència d'edició entre X i Y}**

Operacions d'edició:

- **substitució** d'un símbol $a \in \Sigma$ per un de diferent $b \in \Sigma$
- **inserció** d'un símbol $a \in \Sigma$
- **esborrat** d'un símbol $b \in \Sigma$

Si λ representa la cadena buida:

- **substitució:** $a \rightarrow b$
- **inserció:** $\lambda \rightarrow a$
- **esborrat:** $b \rightarrow \lambda$

D(paper,carrer) = 3

paper->caper
caper->carer
carer->carrer

Tema 6 Probabilístics

L'algorisme correcte: preguntar a tots els electors què votaran. No és possible fer això per falta de recursos i col·laboració dels electors.
Algorisme probabilístic: Preguntar el vot a una mostra aleatòria de votants.. Quant més grossa sigui la mostra més fiable serà el resultat. La predicció serà correcte amb certa probabilitat. Es pot definir una certa forquilla on casi segur estaran els resultats de la votació.
Comportament: a més recursos→millor resultat(precisió Up), això comporta més temps i memòria. No determinista (pot comportar-se diferent aplicat a les dades)

Determinista vs Probabilistics: DET→mai se li permet que no acabi: fer una divisió per 0. Si hi ha més d'una solució, sempre troba la mateixa. No pot calcula solucions incorrectes PROB→pot permetre que passi probabilitat molt petita d'un grup de dades(Si passa, repeteix). Troba solucions diferents, executant-se varies vegades. Pot equivocar-se, però a mida de repetir, arriba a una solució correcta. Mai troben solucions 100% exactes

Classificació probabilístics. *Dos primers, no garanteixen correcció. Els altres garanteixen*

-Numèrics: Solució aprox, interval de confiança, més temps aprox. Per arribar a més precisió, necessita 100 vegades més treball (arrel inversament proporcional)
-**Monte Carlo:** dona resposta exacta amb alta prob, més execució→menor error. Es p-correcte si la solució te major o igual a **p**(depèn de la mida d'entrada) per a qualsevol dada de entrada. **Ampl. estocàstica→Esbiaxat** a mida de repetir, podem saber si correcte. **No esbiaixat**, a mida de repetir ens dona les mateixes prob, llavors poden ser les correctes. Ex: Algorisme vector Majoritari.
L'algorisme de MC 0,5-correcte **esbiaixat** ens assegura que és cert en alguna de les seves respostes, mentre que tirar una moneda no. L'algorisme de MC-05-correcte **esbiaixat** si l'executem un nombre k de vegades millora la seva probabilitat d'encert a 0,5^k.

```
funció majoritariN(T[1..n],e)
        k← log0.5(e)
        per i ←1 fins k fer
                si majoritari(T)
                llavors retorna cert
retorna (fals)
```

Problema de la ruleta. Aleatori
-Las vegas: decisions a l'atzar, diuen si la solu no es correcte, repeteix fins solu correcte. Si es p-correcte, el criteri es el mateix que al monte-carlo. Ex: Problema de les n-reines

- Pot fer-se millor combinant els dos algorismes:
  - Posar les primeres reines a l'atzar i deixar-les fixes i amb la resta utilitzar un algorisme de backtracking.
- Com més reines posem a l'atzar:
  - Menys temps necessitem per a trobar una solució o fallar.
  - Més gran és la probabilitat de fallar.

Ex: Greedy amb aleatorietat, minimitza la busqueda

-Sherwood: temps d'execució equiprobable si els inputs son de mateixa mida, no millora cost en mitjana. Són algorismes pels quals existeix una solució determinista que és molt més ràpida en mitjana que en el pitjor dels casos. Els algorismes de Sherwood, afegint aleatorietat redueixen temps entre entrades ("treuen eficiència de les entrades riques per donar-la a les pobres". Un exemple és el quicksort amb la tria del pivot aleatòria.

Tema 7 Test

Algorisme es fiable si:**Correcte:** capacitat de compotar-se d'acord amb la seva especificació. **Robust:** capacitat de reaccionar a casos extrems, overflow, manca memòria.

Especificació Formal

Especificació: Què fa l'algorisme sota unes certes condicions inicials. Diu què fa, però no com ho fa. **Algorisme:**Fa el que diu l'especificació. **Implementació**: Fa el que diu l'algorisme considerant les restriccions que pugui suposar l'ordinador (memòria, temps d'execució, precisió dels càlculs, etc.)

Per Contracte:

Asserció:expressió lògica que fa servir entitats de l'algorisme i estableix una propietat (condició) que aquestes han de satisfer en certs moments de la seva execució. Tipus: *precondicions, postcondicions, invariants de bucle(*condicions a complir a cada iteració), *invariants de classe*(condicions a complir qualsevol obj de la classe). Aquestes s'implementen com condicions, si no es compleixen, stop execució. Les assercions poden alentir el programa.

Precondicions: Condicions que han de complir les dades d'entrada de l'algorisme.

Postcondicions: Condicions que s'han de complir després d'executar l'algorisme.
L'especificació per contracte es pot aplicar a:
Algorismes, Implementacions, Parts d'algorismes o implementacions (aplicació recursiva), Moduls, funcions, objectes, llibreries, etc.
·L'especificació pot variar al fer la implementació degut a limitacions del hardware.
Invariants de bucle: l'assert es posa al final del bucle, i sempre va relacionat amb la condició del bucle i alguna variable interna que depengui d'aquest. while(i<n) { i=i+1; r=r*i; } // invariant: r==i!.

Invariants de classe: Defineix l'estat de qualsevol instància de la classe. S'expressa com a un predicat que relaciona els valors dels atributs d'una classe. Les instàncies han de mantenir aquest estat mentre realitzen les seves operacions Cada operació de la classe pot assumir que la invariant serà certa a l'entrada i que ha de deixar-la certa a la sortida. Són les propietats semàntiques (significat) i les restriccions d'integritat de la classe. Exemple: classe llista ordenada. Hauria de mantenir l'ordre abans i després de realitzar una operació.

On s'han de complir les invariants de classe
- Al final dels constructors.
- A l'inici dels destructor.
- A l'inici i al final dels mètodes.

On **no** es compliran les invariants de classe
- Al inici o al mig dels constructors.
- Al final o al mig dels destructor.
- Al mig dels mètodes.

Exemple:
- A la classe List m_ppEnd sempre apunta a l'apuntador que conté NULL del final de la llista
- A la classe Node m_pElement!=NULL

```
// PopFrontElement ---------------------
Element* List::PopFrontElement() {
   assert(m_pStart!=NULL);
   Node *pN=m_pStart;
   Element *pE=pN->m_pElement;
   m_pStart=m_pStart->m_pNext;
   delete pN;
   return pE;
}
```

```
List::~List() {
   while (m_pStart) {
         Node *pN=m_pStart;
         m_pStart=m_pStart->m_pNext;
         delete pN->m_pElement;
         delete pN;
   }
   assert(m_pStart==NULL);
}
```

```
// ~List -------------------------------------------------------
List::~List() {
   assert(m_ppEnd!=NULL); // nomes al principi
   assert(*m_ppEnd==NULL); // nomes al final
   while (m_pStart) {
         Node *pN=m_pStart;
         m_pStart=m_pStart->m_pNext;
         delete pN->m_pElement;
         delete pN;
   }
```

```
// List --------------------------------------------------------
List::List() {
   m_pStart=NULL;
   m_ppEnd=&m_pStart;
   assert(m_ppEnd!=NULL); // nomes al final
   assert(*m_ppEnd==NULL); // nomes al final
}
```

```
Element* List::PopFrontElement() {
   assert(m_ppEnd!=NULL);
   assert(*m_ppEnd==NULL);
   assert(m_pStart!=NULL);
   Node *pN=m_pStart;
   Element *pE=pN->m_pElement;
   m_pStart=m_pStart->m_pNext;
   delete pN;
   assert(pE!=NULL);
   assert(m_ppEnd!=NULL);
   assert(*m_ppEnd==NULL);
   return pE;
}
```