Temas Avanzados de Programación Dinámica

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp Argentina 2019
UNC - FAMAF

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

“Pay heed to the tales of old wives. It may well be that they alone keep in memory what it was once needful for the wise to know.”

J. R. R. Tolkien, The Lord of the Rings

“Memory is the thing you forget with.”

Alexander Chase, Perspectives, 1966.

“I cannot but remember such things were,
That were most precious to me.”

William Shakespeare, Macbeth
Act IV, scene 3, line 222

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Visión tradicional

- Programación Dinámica y Divide and Conquer son dos técnicas basadas en recursión
- En ambas queremos calcular algún(os) valor(es) de f, una función recursiva
- En divide and conquer, los subproblemas son completamente independientes entre sí, de modo que podemos implementar la recursión directa y resolver
- Se suele explicar programación dinámica, como “Divide and Conquer, con memoria para reutilizar subproblemas repetidos”

Visión constructiva

- Mi forma favorita de pensar programación dinámica
- Aplica principalmente para calcular “algo” sobre caminos en un DAG: **Son la gran mayoría de los casos**.
- Dado el problema que queremos encarar, imaginamos un “proceso de construir una solución”
- El proceso parte de un **estado** inicial (eventualmente varios)
- Existen **transiciones** posibles, a través de las cuáles podemos pasar de un estado a otro
- El grafo de estados y transiciones define un DAG (si hay ciclos, no se puede aplicar DP directamente)
- El estado **captura toda la información** que necesitamos **para seguir construyendo la solución** desde ese punto

Visión constructiva (cont)

- El proceso de construcción de la solución consiste en recorrer un camino por los estados, hasta llegar a algún “estado solución” (casos base)
- Nuestro plan es calcular alguna “cosa” sobre las soluciones que se pueden construir desde cada estado
- Cosas típicas:
  - Cantidad de soluciones (cantidad de caminos)
  - Solución óptima (“mejor” camino)
  - Solución lexicográficamente más chica
  - Conjunto de “valores” que puede tomar una solución (por ejemplo, conjunto de “estados solución” alcanzables desde cada estado)

Ejemplos

- Dominós en un tablero de $2 \times n$
- Cruzar la matriz
- Problema de la mochila
- Problema del viajante de comercio
- Precomputar $\sum_{S' \subseteq S} f(S')$ dada f para todos los $S \subseteq T$
- "Camino en DAG" (mínimo, cantidad, etc). Casi todos los problemas de DP son caso particular de este (como lo son todos los anteriores)

Cómo se implementa con DP

- Vamos a calcular con DP:
  $f(e) =$ valor de la cosa que queremos calcular en el estado e
- Principio de optimalidad: Se podrá aplicar DP al problema, cuando para los estados elegidos se pueda calcular $f(e)$, a partir de los $f(e_s)$ para todos los sucesores e_s de e.
- A lo anterior nos referíamos cuando decíamos que el estado captura la información **necesaria**:
  - Siempre podemos elegir un estado que haga válido el principio del óptimo. El trivial es tomar **todo el camino** recorrido como estado
  - La gran ganancia de programación dinámica surge de **olvidar** casi todos los detalles sobre el camino exacto utilizado, quedándose solamente con **lo importante** para poder completar la solución
  - Por ejemplo para mochila, no podemos sacar del estado la capacidad restante, o no sabremos si un objeto entra o no

Visión constructiva: Ventajas

- Al reconstruir el camino, la reconstrucción lo recorre **en el orden del proceso de construcción de la solución** (“al derecho”).
- Por la propiedad anterior, es relativamente simple modificarlo para encontrar:
  - La solución lexicográficamente más chica
  - La solución lexicográficamente más grande
  - Más en general, la *i*-ésima solución en orden lexicográfico (¡Notar que depende del orden! La visión constructiva ayuda a razonar en el orden en que queremos, **de entrada**)
- En mi experiencia personal subjetiva, esta forma de razonar me ayuda **mucho** a encontrar algoritmos de DP para problemas más difíciles.
  - Evidencia anecdótica: Cuando expliqué esta forma en una charla de DP en el TC UBA 2014, un equipo cordobés me dijo que le encantó y que “al fin entendimos lo que nuestro coach nos decía todo el tiempo: ‘¡Busquen el estado! ¡Busquen el estado!’ ”

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Planteo

- En todos los ejemplos que siguen, primero queremos plantear el proceso de crear una solución, como vimos
- Queremos plantearlo de forma que “cada camino hasta un estado final” lleve a una posible solución al problema

Conteo de soluciones

- Contar cuántas soluciones hay, es contar caminos
- Usaremos la recursión $f(e) = \sum_{e_s} f(e_s)$, con $f(e) = 1$ para los estados finales

Solución lexicográficamente más chica

- Suponemos que algunas soluciones “funcionan”, y otras no. Queremos la lexicográficamente más chica que funciona
- Calculamos
  $f(e) =$ desde e se puede llegar a una solución que funciona
- Al reconstruir el camino, elegimos siempre el sucesor más chico que funciona
- Es muy común combinar esto con camino mínimo (solución óptima lexicográficamente más chica)

Solución i-ésima en orden lexicográfico

- Suponemos que i indexa desde 0 (la solución 0 es la que vimos antes, la lexicográficamente menor)
- En lugar de calcular solo si hay solución que funciona como en la anterior, las contamos sumando como ya vimos
- $f(e) =$ Cantidad de soluciones que funcionan desde e
- Al reconstruir el camino, si buscamos la i-ésima y la primera opción tiene $T > i$ soluciones, nos movemos a esa opción
- Sino, le restamos T a i, y verificamos lo mismo para la segunda opción.
- Seguimos así hasta mandarnos por una opción. Si ninguna funcionó, i es demasiado grande y por lo tanto no hay i-ésima
- Cuando llegamos a un estado final con $i = 0$, el camino que recorrimos describe la i-ésima solución

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Idea

- El estado contiene un **subconjunto** $S \subseteq T$ de algún conjunto T relevante al problema
- Implementación: número de 0 a $2^n - 1$ (máscara de bits: `0` a `(1<<n)-1`)
  - $\cup$ es el `|`
  - $\cap$ es el `&`
  - $\{i\}$ es `1<<i`
  - T es `(1<<n)-1`
  - $\emptyset$ es `0`
  - S^c es `T&(~S)` o `T-S`
  - $A - B = A \cap B^c$ es `A&(~B)` o `A&(T-S)`
- Notar que el estado podría tener más cosas, o más de un subconjunto

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Ejemplos

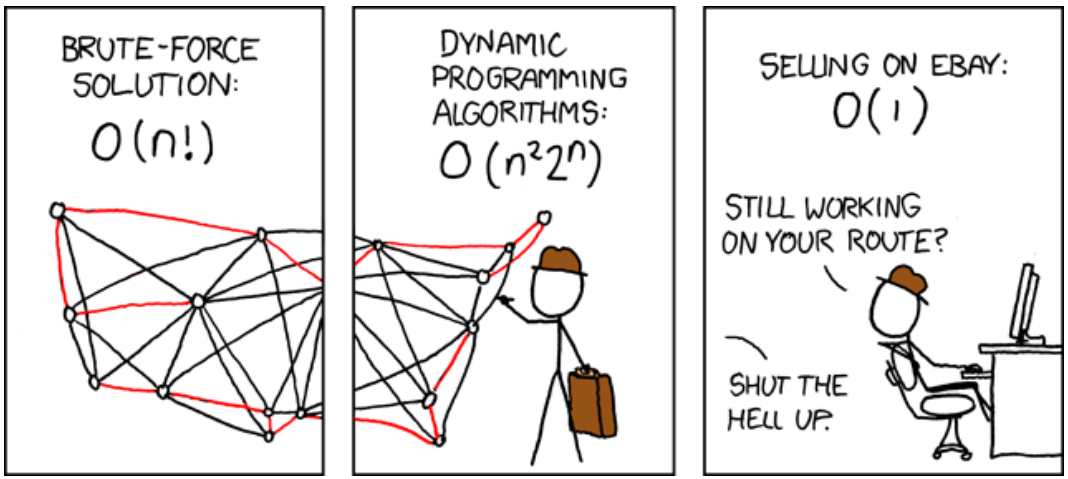
- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$

Ejemplos

- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$
- Minimum set cover: $O(M2^N)$

Ejemplos

- Matching perfecto de costo mínimo en grafo completo: $O(N2^N)$
- Minimum set cover: $O(M2^N)$
- TSP: $O(N^2 2^N)$



- Todas son **muchísimo** más eficientes que el backtracking directo

Contenidos

1 Repaso de Programación Dinámica
- Dos visiones
- Cosas calculables con visión constructiva

2 Dinámicas con subconjuntos
- Idea
- Ejemplos

3 Tabla aditiva multidimensional
- Idea
- Ejemplos

4 Dinámicas con frente
- Idea
- Ejemplos

5 Técnicas de optimización de DP
- Optimización de Knuth
- Optimización de Divide and Conquer

Sumas parciales (1D)

- *dp* se inicializa igual que el vector de entrada
- Se hace `dp(i) += dp(i-1)` para $i > 0$, en orden

Con este cómputo $O(N)$ está la suma de todos los **anteriores**

Sumas parciales (2D)

- *dp* se inicializa igual que la matriz de entrada
- Se hace `dp(i,j) += dp(i-1,j)` para $i > 0$, para todos los j
- **Luego** se hace `dp(i,j) += dp(i,j-1)` para $j > 0$, y todo i

Con este cómputo $O(NM)$ está la suma de todos los **anteriores** en la matriz: $dp(i,j) = \sum_{a=0}^{i} \sum_{b=0}^{j} v(a,b)$

Es exactamente la idea del teorema de Fubini (con sumatorias).

Código multidimensional 5D

Ejemplo conceptual para 5 dimensiones. Misma idea, acumular en cada una por separado sucesivamente:

- Poner en *dp* la entrada
- Hacer:

```
for dim = 0 to 4
 for a = 0 to na-1
  for b = 0 to nb-1
   for c = 0 to nc-1
    for d = 0 to nd-1
     for e = 0 to ne-1
      pa = a-(dim==0); pb = b-(dim==1); pc = c-(dim==2);
      pd = d-(dim==3); pe = e-(dim==4);
      if (pa >= 0 && pb >= 0 && pc >= 0 &&
          pd >= 0 && pe >= 0)
       dp(a,b,c,d,e) += dp(pa,pb,pc,pd,pe)
```

Para una matriz de T celdas en total y con d dimensiones, este cómputo $O(Td)$ / $O(Td^2)$ calcula la suma de todos los **anteriores**

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Sumas para los subconjuntos

- Un subconjunto de T, que tiene n elementos, es un elemento de una matriz de $2 \times 2 \times \cdots \times 2$ (n veces) [hipercubo]
- Esto pues la máscada de bits nos da las n coordenadas, 0 o 1
- Los subconjuntos de un elemento, son a su vez los anteriores en dicha matriz
- Con un for de 1 a n, y luego pasando por los 2^n conjuntos haciendo una suma, se computan todas las sumas sobre subconjuntos de cada subconjunto:

```
for i = 0 to n
 for mask = 0 to (1«n) - 1
  if (mask & (1«i))
   dp(mask) += dp(mask - (1«i))
```

Sumas para los divisores

- Sea S es un conjunto cerrado por divisores, y P la lista de primos que aparecen en los números de S
- Un número tiene $|P|$ coordenadas, los exponentes de los primos.
- Los divisores son justamente los “menores”, vistos con las coordenadas
- Se puede sumar los valores sobre los divisores usando la idea de tabla aditiva $|P|$ dimensional:

```
for p ∈ P
 for x ∈ S (DE MENOR A MAYOR)
  if (p divide a x)
   dp(x) += dp(x / p)
```

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

? ? ? ?
? ? ? ?
? ? ? ?
? ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 ? ? ?
? ? ? ?
? ? ? ?
? ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 ? ? ?
1 ? ? ?
? ? ? ?
? ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 ? ? ?
1 ? ? ?
0 ? ? ?
? ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 ? ? ?
1 ? ? ?
0 ? ? ?
1 ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 1 ? ?
1 ? ? ?
0 ? ? ?
1 ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 1 ? ?
1 0 ? ?
0 ? ? ?
1 ? ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0	1	?	?
1	0	?	?
0	0	?	?
1	?	?	?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 1 ? ?
1 0 ? ?
0 0 ? ?
1 0 ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 1 0 ?
1 0 ? ?
0 0 ? ?
1 0 ? ?

Idea

- Tenemos que considerar formas de “llenar” un tablero.
- Como siempre en programación dinámica, queremos “olvidar lo más posible”
- Al ir llenando el tablero, suele alcanzar con saber únicamente la situación en el **frente** por dónde vamos llenando.
- Por ejemplo si queremos llenar un tablero con 0 y 1 pero sin que haya dos 1 pegados:

0 1 0 ?
1 0 0 ?
0 0 ? ?
1 0 ? ?

Estado

- El estado tendrá la posición (x, y) actual en el tablero.
- Además, para un tablero de $N \times M$, guardará N valores (o a veces $N + 1$) con el frente.
- Si los valores son binarios como en el ejemplo hay $O(NM2^N)$ estados, pero en cambio hay 2^{NM} tableros.

Contenidos

1 Repaso de Programación Dinámica
- Dos visiones
- Cosas calculables con visión constructiva

2 Dinámicas con subconjuntos
- Idea
- Ejemplos

3 Tabla aditiva multidimensional
- Idea
- Ejemplos

4 Dinámicas con frente
- Idea
- Ejemplos

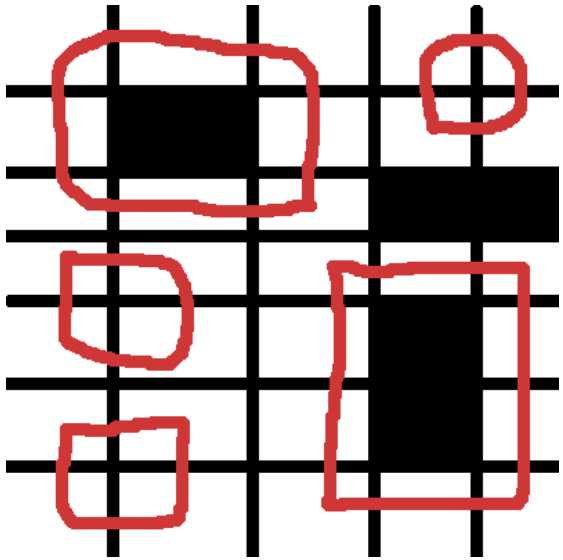
5 Técnicas de optimización de DP
- Optimización de Knuth
- Optimización de Divide and Conquer

Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas

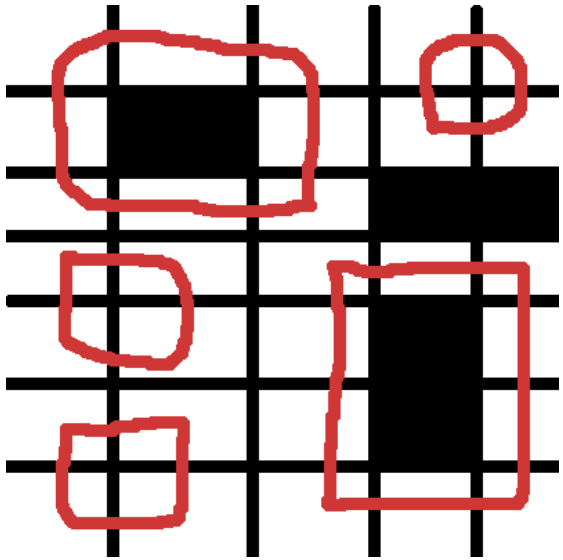
Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



Ejemplos

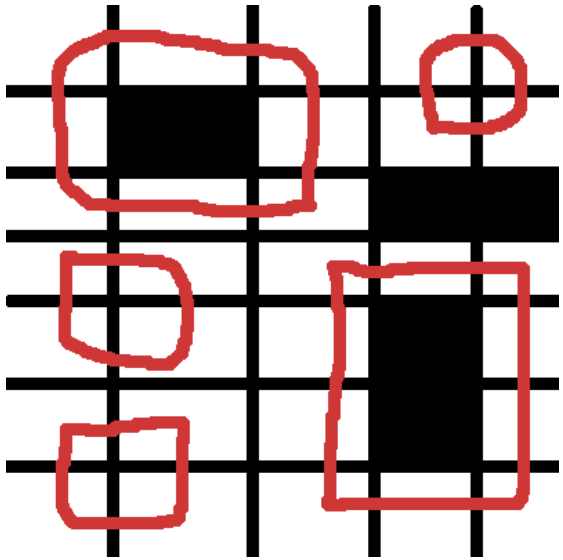
- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



- El frente anterior tiene $O(2^N)$ valores posibles. ¿Y si quisiéramos saber la mínima cantidad de ciclos necesarios?

Ejemplos

- Cantidad de maneras de cubrir con dominós, un tablero con agujeros en posiciones dadas
- Cantidad de maneras de cubrir con “tuberías” cerradas (ciclos), un tablero con agujeros en posiciones dadas



- El frente anterior tiene $O(2^N)$ valores posibles. ¿Y si quisiéramos saber la mínima cantidad de ciclos necesarios?
- Se puede con un frente con $O(3^N)$ valores posibles.

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Ejemplo motivador

- Dados n valores enteros distintos v_i, junto a sus frecuencias f_i, dar un árbol binario de búsqueda óptimo para los valores.

Ejemplo motivador

- Dados n valores enteros distintos v_i, junto a sus frecuencias f_i, dar un árbol binario de búsqueda óptimo para los valores.
- $dp(i,j) = \min_{k=i}^{j-1} dp(i,k) + dp(k+1,j) + sum_f(i,j)$
- Complejidad: $O(n^3)$
- ¿Se podrá mejorar?

Contexto

- ¡Sí! Con la optimización de Knuth
- Dado un algoritmo de dp de dos parámetros cualquiera, es decir $dp(i, j)$ con $0 \leq i, j \leq N$ (ejemplo $i \leq j$ si es **DP en rangos**)
- Si su recursión tiene la forma $dp(i, j) = \min_k g(i, j, k)$ para cierta g que solo usa valores $dp(a, b)$ con $a \geq i$ y con $b \leq j$
- Podemos definir $K(i, j)$ como el menor k en donde se alcanza el mínimo de la expresión para $dp(i, j)$
- Para escribir las complejidades, notaremos $|K|$ a la cantidad de valores posibles para k en la expresión. Típicamente $|K|$ es aproximadamente N
- En las típicas dp en rango, k se mueve entre i y j.

Condición de Knuth

- $K(i, j-1) \leq K(i, j) \leq K(i+1, j)$
- Equivalentemente: K es **monótona en ambos parámetros**.
- En criollo para DPs en rangos:
  - Si agregamos un elemento por **izquierda**, el K se mueve **a la izquierda**
  - Si agregamos un elemento por **derecha**, el K se mueve **a la derecha**
- Llamamos a la anterior la *Condición de Knuth*
- Suele ser mucho más difícil demostrar que se cumple, que convencerse o intuir que así será

Optimización de Knuth

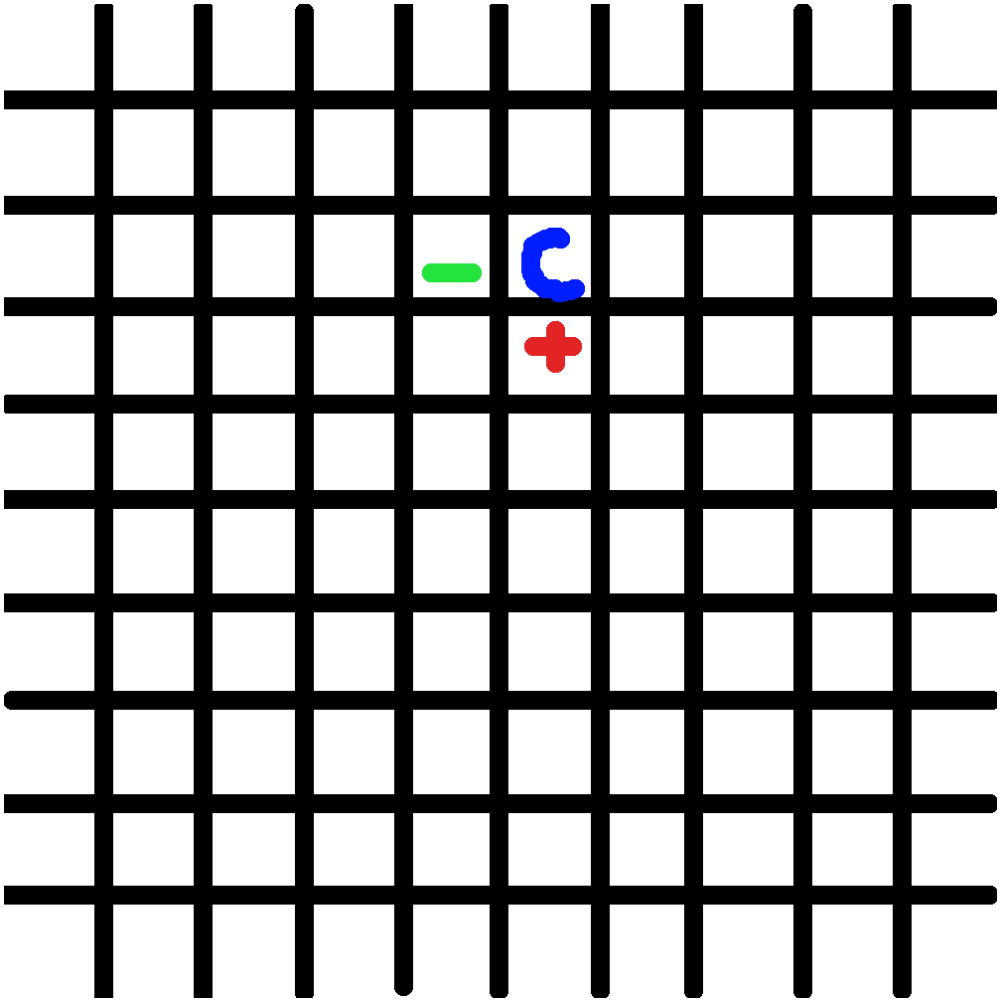
- Como vale la condición de Knuth, es muy simple cambiar en el código la recursión usando las cotas para iterar menos:
- $dp(i,j) = \min_{k=K(i,j-1)}^{K(i+1,j)} g(i,j,k)$
- Los K los podemos ir calculando en el mismo algoritmo junto a los valores dp.
- Las cotas sirven para iterar menos... ¿Pero estamos mejorando la complejidad asintótica?

Optimización de Knuth

- Como vale la condición de Knuth, es muy simple cambiar en el código la recursión usando las cotas para iterar menos:
- $dp(i,j) = \min_{k=K(i,j-1)}^{K(i+1,j)} g(i,j,k)$
- Los K los podemos ir calculando en el mismo algoritmo junto a los valores dp.
- Las cotas sirven para iterar menos... ¿Pero estamos mejorando la complejidad asintótica?
- Teorema: Con este sencillísimo cambio al código básico, el algoritmo es $O(N(N+|K|))$ (la cota anterior era $O(N^2|K|)$
- Demostración: La sumatoria de los costos es telescópica en 2D (dibujos)
- Si evaluar g no es $O(1)$, el costo son $O(N(N+|K|))$ llamadas a g

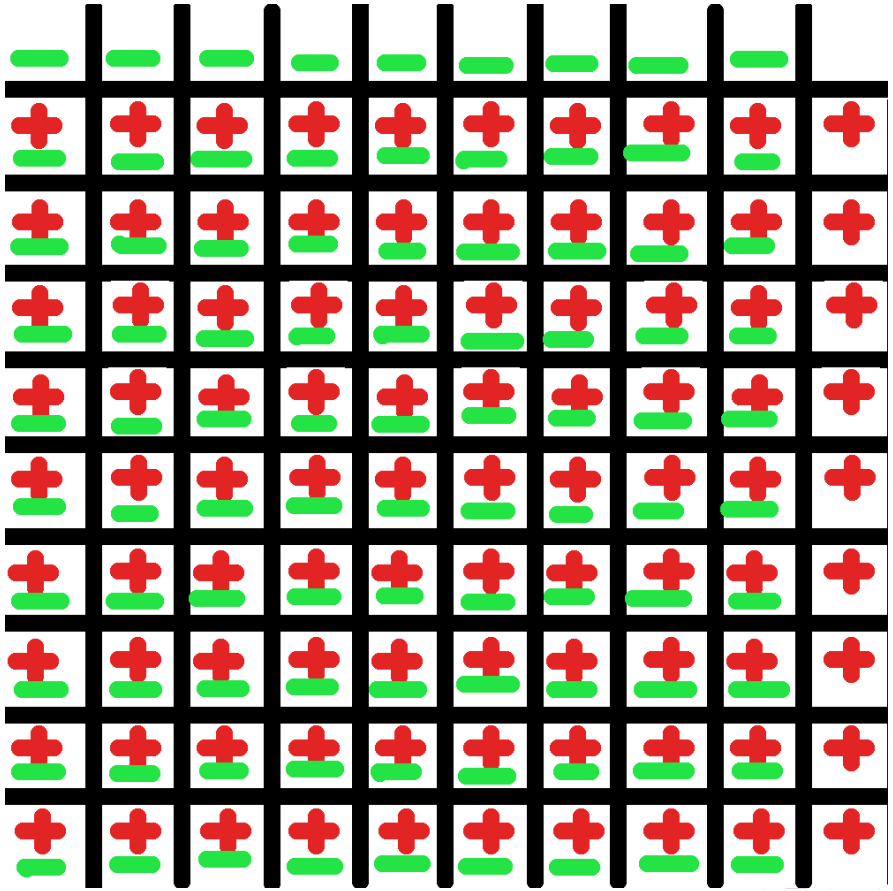
Optimización de Knuth (dibujos)

- Costo para calcular $dp(i, j)$: $K(i+1, j) - K(i, j-1) + 1$



Optimización de Knuth (dibujos)

- A lo largo de casi toda la matriz, los términos K se cancelan.
- Las N^2 casillas pagan el término 1: Total $O(N^2)$
- Las $O(N)$ casillas de borde pagan $O(|K|)$ Total $O(N|K|)$



Comentarios

- El dibujo muestra una matriz “completa” de $N \times N$
- La demostración aplica si “borde” $O(N)$ y tamaño $O(N^2)$
- Ejemplo: la parte triangular superior (dp en rangos)

Ejercicio

- Ejercicio: Verificar que la condición de Knuth aplica en la recursión que vimos antes
- Intuitivamente tiene muchísimo sentido, ¡pero demostrarlo es mucho más difícil que programarlo!

Contenidos

1. Repaso de Programación Dinámica
   - Dos visiones
   - Cosas calculables con visión constructiva
2. Dinámicas con subconjuntos
   - Idea
   - Ejemplos
3. Tabla aditiva multidimensional
   - Idea
   - Ejemplos
4. Dinámicas con frente
   - Idea
   - Ejemplos
5. Técnicas de optimización de DP
   - Optimización de Knuth
   - Optimización de Divide and Conquer

Ejemplo motivador

- Dados n valores x_i enteros positivos, el costo de un intervalo $[i, j)$ es $\sum_{i \leq a < b < j} x_a x_b$. O sea, sumar los productos de a pares.
- Particionar el arreglo $[0, n)$ en k intervalos, minimizando la suma de los k costos.

Ejemplo motivador

- Dados n valores x_i enteros positivos, el costo de un intervalo $[i, j)$ es $\sum_{i \leq a < b < j} x_a x_b$. O sea, sumar los productos de a pares.
- Particionar el arreglo $[0, n)$ en k intervalos, minimizando la suma de los k costos.
- $dp(n, k) = \min_{i=0}^{n-1} dp(i, k-1) + val(i, n)$
- Complejidad: $O(N^2 K)$
- ¿Se podrá mejorar?

Contexto

- ¡Sí! Con la optimización de Divide and Conquer
- Dado un algoritmo de dp con dos parámetros, es decir $dp(n, k)$ con $0 \leq n \leq N$ y $0 \leq k \leq K$
- Si su recursión tiene la forma $dp(n, k) = \min_i g(n, k, i)$ para cierta g que solo usa los $dp(j, k')$ con $k' < k$
- Podemos definir $I(n, k)$ como el menor i en donde se alcanza el mínimo de la expresión para $dp(n, k)$
- Para escribir las complejidades, notaremos $|I|$ a la cantidad de valores posibles para i en la expresión. Típicamente $|I|$ es aproximadamente N

Condición de Divide and Conquer

- $I(n,k) \leq I(n+1,k)$
- Equivalentemente: I es **monótona en n**.
- En criollo para DPs de particionar: si para k fijo agrando el rango, el último punto de corte también (mejor dicho: no retrocede)
- Llamamos a la anterior la *Condición de Divide and Conquer*
- Igual que antes, suele ser mucho más difícil demostrar que se cumple, que convencerse o intuir que así será

Optimización de Divide and Conquer

- Esta optimización no es tan simple de implementar como la de Knuth, pero la idea también es sencilla.
- Supongamos que para calcular todos los $dp(n, k)$ para k fijo, calculamos primero el $dp(n', k)$:
  - Para los $n > n'$, alcanza con probar el i **desde** $I(n', k)$
    Es decir, no más de $L_1 = |I| - I(n', k) + 1$ valores
  - Para los $n < n'$, alcanza con probar el i **hasta** $I(n', k)$
    Es decir, no más de $L_2 = I(n', k) + 1$ valores
- Esto nos parte el rango $[0, N)$ que debíamos calcular en dos restantes: $[0, n')$ y $[n' + 1, N)$.
- En la primera parte hay que probar hasta L_1 valores, y en la segunda hasta L_2 valores.

Si profundizamos...

- Podemos seguir partiendo estos rangos en dos recursivamente
- En el paso k tendremos 2^k rangos, cada uno con hasta L_i opciones factibles para el i.
- Observación: En cada paso, los L_i suman $O(N + |I|)$
- Por lo tanto, procesar cada paso es $O(N + |I|)$
- Partiendo siempre a la mitad, serán $O(\lg N)$ pasos y la complejidad es $O((N + |I|) \lg N)$
- El algoritmo final cuesta $O(K(N + |I|) \lg N)$ evaluaciones de g

Ejercicio

- Ejercicio: Verificar que la condición de Divide and Conquer aplica en la recursión que vimos antes
- Pasa lo mismo que antes: Es más fácil intuirlo que probarlo

Resumen

Optimización de Knuth:

- Prototípico para DPs en rangos
- Punto óptimo debe ser monótono en ambas variables
- Requiere cuadrados o "similares": $O(N)$ borde, $O(N^2)$ estados
- Le saca un $|I|$ (ejemplo: $N^3 \to N^2$)

Optimización de Divide and Conquer:

- Prototípico para DPs "de particionar"
- Punto óptimo debe ser monótono en una variable
- Cualquier forma (por ejemplo sirve aún si $K << N$)
- Cambia un $N|I|$ por $(N + |I|) \lg N$ (ejemplo: $KN^2 \to KN \lg N$)