El Magnate de la Bondiola

Problema

El Magnate de la Bondiola piensa reventar la Costanera instalando puestos en la franja que está al sur de Ciudad Universitaria. Tiene una lista de n posibles lugares donde ubicar sus puestos. Para cada posible lugar conoce la distancia d_i en metros desde la puerta del Pabellón 1 de Ciudad Universitaria, y una estimación b_i del beneficio que le daría un puesto instalado ahí ($i = 1, 2, \ldots, n$). Esta información está ordenada de forma creciente por distancia. El empresario tiene dinero suficiente para instalar los n puestos. Sin embargo, sabe que si dos puestos están muy cerca, el beneficio de cada uno se reduce, de modo que no quiere que ningún par de puestos instalados quede a menos de k metros de distancia uno de otro. Esto significa que si dos puestos $i_1 < i_2$ son instalados, entonces debe cumplirse que $d_{i_2} - d_{i_1} \geq k$.

Diseñar un algoritmo eficiente basado en programación dinámica que le indique a El Magnate de la Bondiola qué puestos debe instalar a fin de maximizar su beneficio estimado. Mostrar la correctitud y determinar la complejidad del algoritmo propuesto (temporal y espacial). Justificar.

El mejor algoritmo que conocemos tiene complejidad temporal y espacial $O(n)$. Sólo se consideraran eficientes los algoritmos que tengan complejidad temporal $O(n^2)$.

Resoluciones

Vamos a pensar soluciones progresivamente mejores. Es importante programar todas, así tenemos práctica de programar. Es importante también entender las ventajas y desventajas de cada tipo de solución. Para algunas situaciones, una solución "lenta y obvia" puede ser la mejor. Para otras, dedicar más tiempo a pensar en una solución eficiente puede ser un mejor uso de nuestro tiempo.

Solución trivial

Empecemos con una solución totalmente trivial: Probamos todas las maneras de poner puestos, sacamos las inválidas, y nos quedamos con la que más beneficio nos otorgue.

```
#include <iostream>
#include <vector>
#include <algorithm>

typedef std::vector<int> vint;
typedef std::vector<bool> vbool;

// Devuelve un vector v, tal que v tiene longitud n,
// y v[i] es true si y sólo si el i-ésimo bit de k es 1.
vbool int_to_bool_vector(int k, int n) {
    vbool res(n);
    for (int i = 0; i < n; ++i) {
        res[i] = (k >> i) & 1; // i-ésimo bit de k
    }
    return res;
}
```

```cpp
// Dado un vector v de longitud n, que indica qué puestos
// queremos usar, un vector de posiciones pos, y una mínima
// distancia k, devuelve true si ningún par de puestos está
// a distancia menos que k.
bool es_valido(const vbool& v, const vint& pos, int k) {
    int n = v.size();
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == j) continue;
            if (v[i] && v[j] && abs(pos[i] - pos[j]) < k) return false;
        }
    }
    return true;
}

// Devuelve cuánta plata ganamos si ponemos los puestos i, donde
// que v[i] = true.
int valor_puestos(const vbool& v, const vint& plata) {
    int res = 0, n = v.size();
    for (int i = 0; i < n; ++i) {
        res += v[i] * plata[i];
    }
    return res;
}

// Devuelve la mejor forma de poner puestos de bondiola, donde el i-ésimo
// puesto está en la posición pos[i], ponerlo nos da plata[i] pesos, y
// no podemos poner a dos puestos a distancia menor a k.
int solucion_trivial(const vint& pos, const vint& plata, int n, int k) {
    /* Pruebo todos los vectores
    booleanos de n elementos. */
    int m = 1 << n; // 2^n, cuidado que puede dar overflow.
    int best = 0;
    for (int i = 0; i < m; ++i) {
        vbool candidato = int_to_bool_vector(i, n);
        if (!es_valido(candidato, pos, k)) continue;
        best = max(best, valor_puestos(candidato, plata));
    }
    return best;
}
```

Esto obviamente funciona, porque estamos probando *todas* las maneras de poner puestos, usando fuerza bruta. Lo único interesante sobre esta solución es el uso de enteros como "bit masks", para indicar subconjuntos. Dado un conjunto S de n elementos S, podemos hacer una biyección entre los `int` s de n bits, y los subconjuntos de S. En particular, si $S = \{s_1, \ldots, s_n\}$, y tenemos un subconjunto $T \subset S$, entonces la representación de T es un entero cuyo i-ésimo bit es 1 si y sólo si $s_i \in T$.

Cuánto tiempo toma esta solución? En `solucion_trivial` hacemos un ciclo desde el 0 hasta $m = 2^n - 1$, inclusive. Esto son 2^n iteraciones. Por cada iteración, llamamos a `int_to_bool_vector(i, n)`, que toma $\Theta(n)$ tiempo. `es_valido` toma $\Theta(n^2)$ tiempo (podés implementarlo en $\Theta(n)$?). Finalmente, `valor_puestos` toma $\Theta(n)$ tiempo, y tomar `max` es una sola operación. Luego, `solucion_trival` toma $\Theta(2^n n^2)$ operaciones.

Cuánto espacio toma esta solución? En cada iteración del ciclo de `solucion_trivial`, estamos creando un vector de `n` bools, y lo destruímos al terminar cada iteración. Luego, el máximo espacio que necesitamos en cualquier momento de nuestro programa usa $\Theta(n)$ bits de espacio.

Divide and conquer

Pensando un poco más, podemos pensar que si ponemos el i-ésimo puesto de bondiolas en d_i, entonces no podemos poner un puesto en ninguna posición en $[d_i - k, d_i + k]$. Como hicimos muchas veces en programación dinámica, vamos a ir construyendo la solución agregando elementos en alguna posición, en este caso al final. Luego, si i es el último puesto que pusimos, sabemos que no pusimos a ningún puesto en $[d_i - k, d_i]$. OK, perfecto. Consideremos una solución óptima que termina poniendo un puesto en la posición i, como queremos hacer. Consideremos los puestos que tiene nuestra solución, antes del i-ésimo. Seguro que todos esos puestos están *en o antes* de $d_i - k$. Asimismo, a *cualquier* forma válida de poner puestos que terminen en o antes de $d_i - k$, le podemos agregar un puesto en la posición d_i, y sigue siendo válida. Luego, para buscar la mejor forma de poner puestos que terminen *en o antes* de el i-ésimo, podemos tomar el máximo beneficio entre:

- Poner un puesto de bondiola en la posición i, y luego buscar la mejor forma de poner puestos que vengan antes de $d_i - k$.
- No poner un puesto de bondiola en la posición i, y buscar la mejor solución hasta el puesto $i - 1$.

Esto es divide-and-conquer. Encontramos que para saber la solución a un problema grande, necesitamos resolver un número de subproblemas de tamaño más chico. La esperanza es que esto nos deje resolver el problema en menor tiempo total.

Creamos entonces una función recursiva, le damos semántica, y escribimos una ecuación correcta para la misma. La semántica es:

$$f(i) = \text{La mejor ganancia que podemos obtener poniendo puestos de bondiola, usando sólo los primeros } i+1 \text{ puesto}$$

El "+1" es para poder definir fácilmetne que $f(0)$ es `plata[0]`. Podríamos sacar ese " + 1", y tendríamos que definir $f(0) = -\infty$. Para hacer fácil la implementación, evitamos eso.

Por el argumento que dimos arriba, una ecuación que define a f es:

$$\begin{gathered} f(i) = \begin{cases} \text{plata}[0] & \text{si } i = 0 \\ \max(f(i-1), \text{plata}[\text{i}] + \text{resto}) & \text{si } i > 0 \end{cases} \\ \textbf{donde resto} = \begin{cases} 0 & \text{si puestos } = \emptyset \\ f(\max(\text{puestos})) & \text{si puestos } \neq \emptyset \end{cases} \\ \text{puestos} = \{j \mid 0 \leq j < i, |d_i - d_j| \geq k\} \end{gathered} \tag{3}$$

La solución al problema está, entonces, llamando a $f(n)$. Notar que en `resto`, estamos llamando a f sólo en el último puesto (en distancia a Ciudad Universitaria) que es válido. Por como definimos $f(i)$, puede perfectamente pasar que la mejor solución que usa los primeros i puestos, **no use el i-ésimo puesto**. Por eso no necesitamos tomar algo como $\max_{p \in \text{puestos}}\{f(p)\}$, sino sólamente $f(\max(puestos))$.

```
/***
Gobernada por la ecuación recursiva, en pseudo-Haskell:
```

```
f(0) = plata[0]
f(i) = max {f(i-1), plata[i] + resto}
    where
        resto = if null puestos_atras
                then 0
                else f(max puestos_atras)
        puestos_atras = [j | j < i, |pos[i] - pos[j]| >= k]
***/

int solucion_dq_f(const vint& pos, const vint& plata, int i, int k) {
    if (i == 0) return plata[0];
    int x = solucion_dq_f(pos, plata, i - 1, k);
    int resto = 0;
    for (int j = i - 1; j >= 0; --j) {
        if (abs(pos[i] - pos[j]) >= k) {
            resto = solucion_dq_f(pos, plata, j, k);
            break;
        }
    }
    return max(x, plata[i] + resto);
}

int solucion_dq(const vint& pos, const vint& plata, int n, int k) {
    return solucion_dq_f(pos, plata, n - 1, k);
}
```

Cuánto tiempo toma nuestra solución? Planteemos una función $T(i)$, que nos de una cota superior de cuántas operaciones hacemos cuando nos dan una entrada de tamaño i. Para este problema, el tamaño de la llamada `solucion_dq_f(const vint& pos, const vint& plata, int i, int k)` es `i`.

Podemos ver que llamar a $T(i)$ llama a $T(i-1)$, y a lo sumo a algún $T(j)$ para $j < i$. Podemos acotar $T(j)$ con $T(i-1)$ también, porque todo $f(x)$ eventualmente llama a $f(y)$ para todo $0 \leq y < x$. Como lo único que nos queda es un ciclo de $O(i)$ iteraciones, tenemos que $T(i) \leq 2T(i-1) + O(i)$. Esto nos dice que $T(i) \in O(2^i)$, lo cual es una mejora cuadrática con respecto a la solución trivial. Les dejo como ejercicio probar que efectivamente $T(i) \in O(2^i)$.

Podemos ser aún más vivos, y darnos cuenta que si $j = i - 1$, entonces ya sabemos el valor de $f(i-1)$, no necesitamos llamarlo dos veces. Luego, `resto` sólo va a tener que computar, a lo sumo, $T(i-2)$, no $T(i-1)$. Esto nos daría un algoritmo distinto, que cumple que $T(i) \leq T(i-1) + T(i-2) + i - 1$. Esto nos da una mejor cota, en particular nos dice que $T(i) \in O(F_{i+1})$, con F la sucesión de Fibonacci. Esta crece como φ^n, con $\varphi = \frac{1+\sqrt{5}}{2} \simeq 1.62 < 2$.

Cuánto espacio necesitamos? En cada ejecución de $f(i)$ vamos a usar un stack de a lo sumo altura i, y espacio constante por stack frame (las tres variables locales de `solucion_dq_f` tienen todas tamaño constante). Luego necesitamos $O(n)$ espacio.

Solución de programación dinámica, top-down

Como siempre que usamos divide-and-conquer, podemos usar un cache para los valores de retorno de las funciones llamadas. **Esto es un proceso totalmente mecánico, no hay absolutamente nada que pensar**. Cuando el espacio de parámetros es pequeño, en relación al número de llamadas totales que hay en el árbol de recursión, vamos a tener muchos subproblemas compartidos, y por tanto podemos memoizar las respuestas.

```
#define VACIO (-1)

int solucion_dp_f(const vint& pos,
                  const vint& plata,
                  int i,
                  int k,
                  vint& cache) {
    if (cache[i] != VACIO) return cache[i];
    int x = solucion_dp_f(pos, plata, i - 1, k, cache);
    int resto = 0;
    for (int j = i - 1; j >= 0; --j) {
        if (abs(pos[i] - pos[j]) >= k) {
            resto = solucion_dp_f(pos, plata, j, k, cache);
            break;
        }
    }
    cache[i] = max(x, plata[i] + resto);
    return cache[i];
}

int solucion_dp(const vint& pos, const vint& plata, int n, int k) {
    vint cache(n, VACIO);
    cache[0] = plata[0];
    return solucion_dp_f(pos, plata, n - 1, k, cache);
}
```

Cuánto tiempo toma esta solución? Podemos pensar el árbol de llamadas de `solucion_dp_f`, y cómo su segunda línea va a significar que primero llenamos `cache[1]`, luego `cache[1]`, luego `cache[2]`, etc. Todas las llamadas de la forma `solucion_dq_f(pos, plata, t, k)`, van a responderse de forma inmediata, porque para cuando volvimos de la llamada `solucion_dp_f(pos, plata, i - 1, k, cache)`, ya llenamos todos los `cache[t]` para todo $t \leq i - 1$. Luego, la ejecución se va a ver como un procedimiento que llena cada `cache[i]`, en órden. Esto va a tomar $O(i)$ tiempo cada vez, por el ciclo de j que tenemos, que usa $O(i)$ operaciones. Luego, todo el algoritmo toma $O(n^2)$ operaciones.

Cuánto espacio usamos? `cache`, y espacio de stack para las recursiones. Ambos usan $\Theta(n)$ bytes de espacio.

Reflexión

Pensemos en lo que hicimos hasta ahora. Pasamos de algo que tomaba $O(2^n n^2)$ a algo que toma $O(2^n)$. Uno podría, hasta ese momento, decir "uhh qué genial, de galácticamente intratable pasamos a solo estúpidamente intratable, gracias Fede!". Pareciera que por magia, *con un procedimiento enteramente mecánico*, obtenemos una solución polinomial, en particular que toma $O(n^2)$ tiempo. Esta es la promesa de programación dinámica. Lo que pasó es que la mitad del trabajo lo hicimos cuando redujimos el espacio de parámetros del problema, de algo que considera "todos los subconjuntos" (un número exponencial de subproblemas) a algo que considera "la mejor forma usando los primeros i puestos" (un número lineal de subproblemas). El que luego de esta transformación seguíamos haciendo un número exponencial de llamadas es, realmente, *incidental*. No es fundamental. El número total de subproblemas es lineal, y el trabajo que hacemos en cada uno es lineal también. Esto ya nos dice que poner un cache, que no recompute la solución a subproblemas, nos va a dar algo que tome tiempo cuadrático.

Esto asume que leer y escribir el cache es una sóla operación, pero eso es verdad acá. En cualquier caso, usualmente vamos a poder pagar a lo sumo un costo logarítmico (por ejemplo, guardando soluciones en un árbol binario de búsqueda balanceado) para usar nuestro cache.

Solución de programación dinámica, bottom-up

Al analizar la complejidad temporal de nuestro algoritmo anterior, consideramos que la ejecución va a ir llenando los valores de `cache[i]` en órden creciente de `i`. Además, teníamos un montón de llamadas recursivas que van a terminar todas inmediatamente, porque, como razonamos, sus valores ya han sido guardados en el cache. Si esto es así, por qué siquiera hacemos esas llamadas? Mejor, llenemos el cache directamente, y nos ahorramos un poco de espacio de stack, y operaciones innecesarias.

```
int iterativa(const vint& pos,
              const vint& plata,
              int i,
              int k,
              const vint& cache) {
    int x1 = cache[i - 1];
    int x2 = 0;
    for (int j = i - 1; j >= 0; --j) {
        if (abs(pos[i] - pos[j]) >= k) {
            x2 = cache[j];
            break;
        }
    }
    return max(x1, x2 + plata[i]);
}

int solucion_dp_iterativa(const vint& pos,
                          const vint& plata,
                          int n,
                          int k) {
    vint cache(n, VACIO);
    cache[0] = plata[0];
    for (int i = 1; i < n; ++i) {
        cache[i] = iterativa(pos, plata, i, k, cache);
    }
    return cache[n - 1];
}
```

Ahora es bastante más obvio que esto toma $O(n^2)$ operaciones, y usa $\Theta(n)$ espacio.

Solución dynamic programming, bottom-up, lineal

Algo que estamos haciendo bastante redundante es el ciclo de j. Pongamos nombre a las cosas para poder usar matemática y lógica. Llamemos $p(i) =$ el último anterior puesto que podemos usar, si elegimos poner el i-ésimo. Es decir, el máximo j, tal que $j < i$, y $|d_i - d_j| \geq k$. Podemos ver que para todo i, tenemos $p(i+1) \geq p(i)$. Esto es porque si podemos usar el puesto $p(i)$ justo anterior al puesto i, sabemos que $d_i - d_{p(i)} \geq k$ y que $p(i) < i$. Como $d_{i+1} > d_i$, sabemos que $d_{i+1} - d_{p(i)} \geq k$ y que $p(i) < i + 1$. Luego, el máximo j tal que $j < i + 1$ y $d_{i+1} - d_j \geq k$ (es decir, lo que llamamos $p(i+1)$), es al menos tan grande como $p(i)$. Luego, $p(i+1) \geq p(i)$.

Como estamos llenando `cache` de menor a mayor, podemos guardar un "cursor" que va valiendo $p(1)$, luego $p(2)$, luego $p(3)$, etc. Cada vez lo incrementamos un poco, pero no volvemos a empezar desde cero cada vez. Lo vamos a aumentar a lo sumo $n - 1$ veces a lo largo de todo el algoritmo, puesto que en todo momento tenemos que $p(i) < i$. Luego, amortizamos el costo de computar los $p(i)$ durante toda nuestra ejecución.

```
int solucion_lineal(const vint& pos, const vint& plata, int n, int k) {
    int cursor = 0, prev_cursor, extra;
    vint cache(n);
    cache[0] = plata[0];
    for (int i = 1; i < n; ++i) {
        prev_cursor = cursor;
        while (abs(pos[cursor] - pos[i]) >= k) ++cursor;
        // El 0 solo pasa cuando todavía no hay ningún puesto anterior
        // que pueda poner.
        extra = (prev_cursor == cursor) ? 0 : cache[--cursor];
        cache[i] = max(plata[i] + extra, cache[i - 1]);
    }
    return cache[n - 1];
}
```

Esto ahora toma $\Theta(n)$ operaciones, y $\Theta(n)$ espacio. Como al menos tenemos que leer toda la entrada, todo algoritmo que resuelve el problema tiene que gastar $\Omega(n)$ operaciones, luego nuestro algoritmo es asintóticamente óptimo en tiempo. Podría ser que exista un algoritmo que use menos espacio que nosotros, pero yo no lo conozco.

Notemos que esto sólo lo pudimos hacer porque controlamos el órden en el que estamos llenando `cache`. Es decir, porque estamos haciendo programación dinámica bottom-up. Si estuviéramos haciendo top-down, no controlaríamos necesariamente el órden en que se resuelven los subproblemas, entonces no podríamos usar (al menos no tan fácilmente) la lógica que usamos acá para calcular `cursor`.

Reconstrucción de la solución óptima

En los programas que vimos arriba, estamos calculando el máximo beneficio posiblem, pero no estamos calculando explícitamente cuál es la forma de poner puestos de bondiolas que nos da ese máximo. En el código que viene, hacemos eso. Les dejo versiones constructivas de tres de las formas que vimos. Ninguna cambia la complejidad espacial o temporal, comparado con la solución a la que corresponden.

```
vbool solucion_trivial_constructiva(const vint& pos,
                                    const vint& plata,
                                    int n,
                                    int k) {
    /* Pruebo todos los vectores
    booleanos de n elementos. */
    int m = 1 << n; // 2^n, cuidado que puede dar overflow.
    int best = 0, valor;
    vbool res;
    for (int i = 0; i < m; ++i) {
        vbool candidato = int_to_bool_vector(i, n);
        if (!es_valido(candidato, pos, k)) continue;
        valor = valor_puestos(candidato, plata);
        if (valor > best) {
```

```
            best = valor;
            res = candidato;
        }
    }
    return res;
}

int iterativa_constructiva(const vint& pos,
                           const vint& plata,
                           int i,
                           int k,
                           vint& cache,
                           vint& anterior) {
    int x1 = cache[i - 1];
    int x2 = 0, antes = VACIO;
    for (int j = i - 1; j >= 0; --j) {
        if (abs(pos[i] - pos[j]) >= k) {
            x2 = cache[j];
            antes = j;
            break;
        }
    }
    if (x1 > x2 + plata[i]) {
        anterior[i] = i - 1;
        return x1;
    }
    anterior[i] = antes;
    return x2 + plata[i];
}

vbool solucion_dp_iterativa_constructiva(const vint& pos,
                                         const vint& plata,
                                         int n,
                                         int k) {
    // anterior[i] guarda el índice del subproblema que elegimos usar,
    // al construir la mejor solución que usa hasta el puesto i.
    // Puede ser i - 1, o puede ser algún j con j < i, y |pos[i] - pos[j]| >= k.
    // Es decir, nos dice cuál de las dos ramas del "max" de f elegimos.
    vint cache(n, VACIO), anterior(n, VACIO);
    cache[0] = plata[0];
    for (int i = 1; i < n; ++i) {
        cache[i] = iterativa_constructiva(pos, plata, i, k, cache, anterior);
    }

    /* Reconstruyo la solución.*/
    /* Empezando por el último puesto que puse, me pregunto a quién
       tenía antes como subproblema. Pero ojo, que el subproblema
       anterior puede haber decidido que la mejor solución era no ponerse
       a si mismo. Esto pasa si y solo si cache[x] == cache[anterior[x]].
       Luego, mientras esto se cumpla, y todavía no encontré el primer
```

```
        puesto que puse (o sea, el x tal que anterior[x] == VACIO, y
        elegí ponerlo en la solución), sigo yendo hacia atrás, buscando
        subproblemas anteriores, que hayan afectado al cache.

        Cuando encuentro un x que puse, lo marco en res, y sigo la
        cadena de subproblemas hacia atrás.

        Puedo empezar por x = n-1, porque de última retrocedo
        al último puesto que en realidad usé.
    */

    int x = n - 1;
    vbool res(n);
    while (x != VACIO) {
        while (anterior[x] != VACIO && cache[x] == cache[anterior[x]]) {
            x = anterior[x];
        }
        res[x] = true;
        x = anterior[x];
    }
    return res;
}

vbool solucion_lineal_constructiva(const vint& pos,
                                   const vint& plata,
                                   int n,
                                   int k) {
    vint cache(n), anterior(n, VACIO);
    int cursor = 0, prev_cursor, extra;
    cache[0] = plata[0];
    for (int i = 1; i < n; ++i) {
        prev_cursor = cursor;
        while (abs(pos[cursor] - pos[i]) >= k) ++cursor;
        extra = (prev_cursor == cursor) ? 0 : cache[--cursor];
        if (plata[i] + extra > cache[i - 1]) {
            cache[i] = plata[i] + extra;
            anterior[i] = cursor;
        } else {
            cache[i] = cache[i - 1];
            anterior[i] = i - 1;
        }
    }

    int x = n - 1;
    vbool res(n);
    while (x != VACIO) {
        while (anterior[x] != VACIO && cache[x] == cache[anterior[x]]) {
            x = anterior[x];
        }
        res[x] = true;
```

```
            x = anterior[x];
        }
        return res;
    }
```