بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

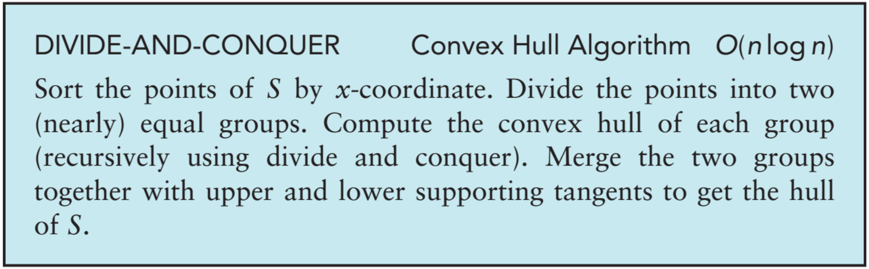
حسین فلسفین

یک الگوریتم دیگر مبتنی بر راهبرد تقسیم و و غلبه که از مرتبهٔ $O(n\log_2(n))$ است:

MergeHull

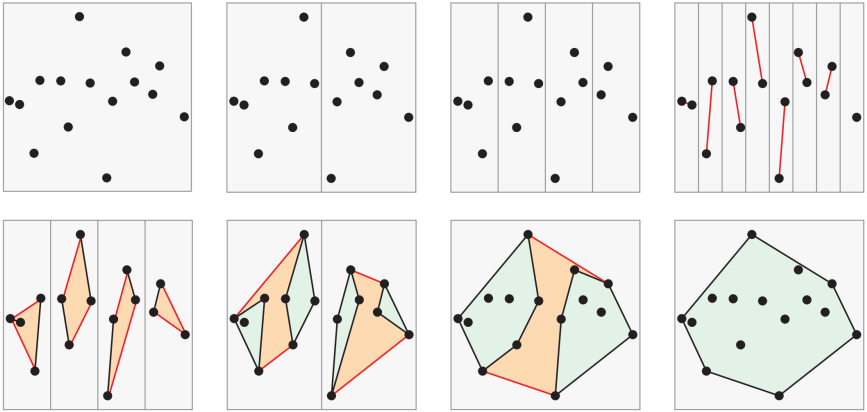
Let S be a point set in general position, with no three points collinear and no two points on the same vertical line. The divide-and-conquer algorithm begins by sorting the points according to x-coordinate. Divide the points into two (nearly) equal groups, A and B, where A contains the left $\lceil \frac{n}{2} \rceil$ points and B the right $\lfloor \frac{n}{2} \rfloor$ points. We then compute the convex hull of A and B recursively (by using the divide-and-conquer algorithm). Finally, we merge $\mathrm{conv}(A)$ and $\mathrm{conv}(B)$ to obtain $\mathrm{conv}(S)$.

Divide and conquer recursively calls itself, with smaller and smaller point sets, until three or fewer points are in each subset. The hull is then immediate. So the recursive step is quite straightforward; the cleverness and geometry come into play in the merge step. Our problem then is to find two tangent lines between the polygons $\mathrm{conv}(A)$ *and* $\mathrm{conv}(B)$*, one supporting the two convex hulls from below and the other from above.*

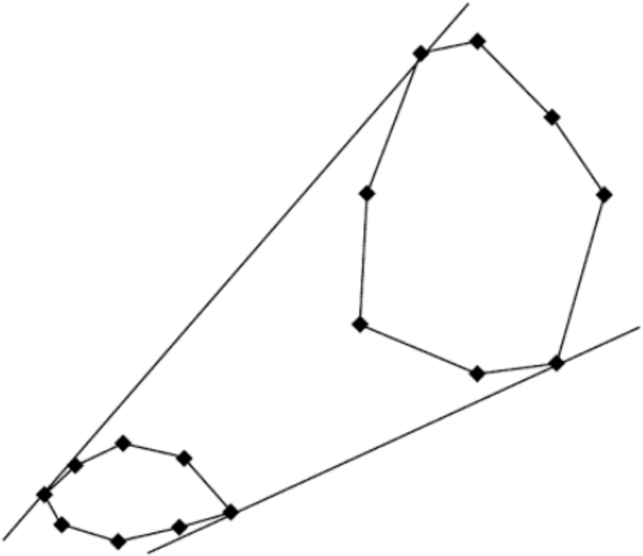


DIVIDE-AND-CONQUER Convex Hull Algorithm $O(n \log n)$

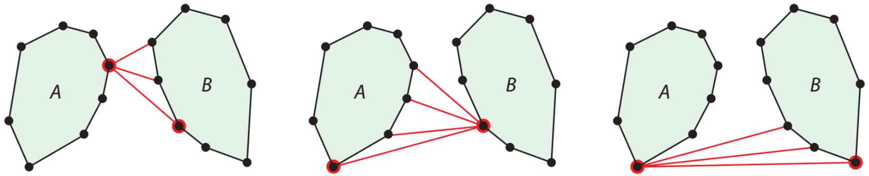
Sort the points of S by x-coordinate. Divide the points into two (nearly) equal groups. Compute the convex hull of each group (recursively using divide and conquer). Merge the two groups together with upper and lower supporting tangents to get the hull of S.



Merging two convex-hulls



With cleverness and geometric intuition, Preparata and Hong were able to find the tangent lines in linear time. Let's describe how to find the lower tangent line, the one supporting the two polygons from below. The sorting step along with our general position assumption (no two points lie on the same vertical line) guarantees that A is to the left of B, separated by a vertical line. Let α be the rightmost point of A and β the leftmost point of B. Assuming that α is a fixed point, proceed by walking counterclockwise from β along the vertices of B. Continue this until a lower tangent line at a vertex of B is found passing through α. Let β be this new vertex of B. Fixing β now, walk clockwise from α around A until a new α is found, which will be a lower tangent to A passing through β. As we repeat this process of walking along A and B, we will eventually reach a lower tangent line supporting both A and B.

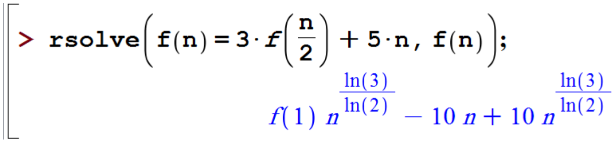

A
B
A
B
A
B

The algorithm for finding the upper tangent line is analogous. The cost of finding the tangent lines is linear, walking around A and B at the same time. Let's now consider the time complexity of the entire divide-and-conquer algorithm. Let $T(n)$ be the time complexity of the divide-and-conquer hull algorithm for n points. Then $T(n) = 2T(n/2) + O(n)$, where $2T(n/2)$ are the recursion halves and $O(n)$ is the merge step: $T(n) \in O(n \log_2(n))$.

ابزارهایی برای به دست آوردن صورت صریح روابط بازگشتی

Maple

تابع *rsolve* در *Maple*:

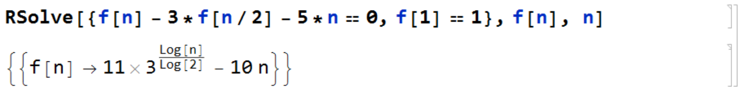


```
> rsolve(f(n) = 3·f(n/2) + 5·n, f(n));
```

$$f(1)\,n^{\frac{\ln(3)}{\ln(2)}} - 10\,n + 10\,n^{\frac{\ln(3)}{\ln(2)}}$$

WOLFRAM MATHEMATICA 11

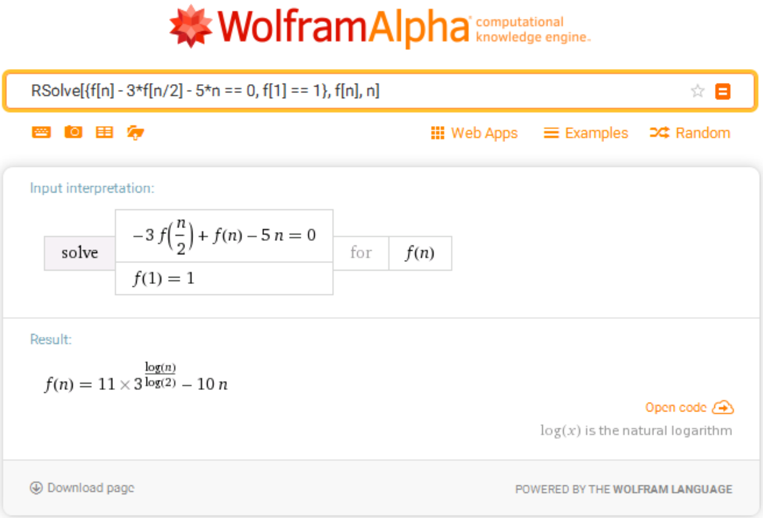
تابع *RSolve* در *Mathematica*:



```
RSolve[{f[n] - 3 * f[n / 2] - 5 * n == 0, f[1] == 1}, f[n], n]
```

$$\left\{\left\{f[n] \to 11 \times 3^{\frac{\mathrm{Log}[n]}{\mathrm{Log}[2]}} - 10\,n\right\}\right\}$$

https://www.wolframalpha.com/


WolframAlpha computational knowledge engine.
Web Apps
Examples
Random
Open code
Download page
POWERED BY THE WOLFRAM LANGUAGE

```
RSolve[{f[n] - 3*f[n/2] - 5*n == 0, f[1] == 1}, f[n], n]
```

Input interpretation:

solve $-3\,f\left(\frac{n}{2}\right) + f(n) - 5\,n = 0$, $f(1) = 1$ for $f(n)$

Result:

$$f(n) = 11 \times 3^{\frac{\log(n)}{\log(2)}} - 10\,n$$

$\log(x)$ is the natural logarithm

دیگر الگوریتم‌های مطرحی که مبتنی بر راهبرد تقسیم و غلبه هستند:

Strassen’s Matrix Multiplication Algorithm:

$\Theta(n^3) \to \Theta(n^{\log_2(7)})$, *where* $\log_2(7) \approx 2.807354922$

A straightforward method of finding the median is to sort all elements and pick the middle one. This takes $\Omega(n\log(n))$ time, as any comparison based sort process must spend at least this much time in the worst case. It turns out that the median, or in general the kth smallest element, in a set of n elements can be found in optimal linear time. This problem is also known as the selection problem.

Binary Search

Binary Search locates a key x in a sorted (nondecreasing order) array by first comparing x with the middle item of the array. If they are equal, the algorithm is done. If not, the array is divided into two subarrays, one containing all the items to the left of the middle item and the other containing all the items to the right. If x is smaller than the middle item, this procedure is then applied to the left subarray. Otherwise, it is applied to the right subarray. That is, x is compared with the middle item of the appropriate subarray. If they are equal, the algorithm is done. If not, the subarray is divided in two. This procedure is repeated until x is found or it is determined that x is not in the array.

The steps of Binary Search can be summarized as follows. If x equals the middle item, quit. Otherwise:

⁎ *Divide the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger than the middle item, choose the right subarray.*

⁎ *Conquer (solve) the subarray by determining whether x is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.*

⁎ *Obtain the solution to the array from the solution to the subarray.*

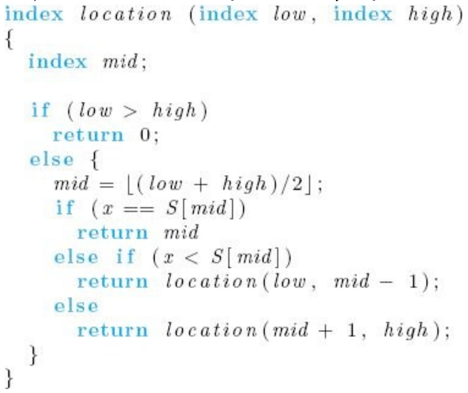
Binary Search is the simplest kind of divide-and-conquer algorithm because the instance is broken down into only one smaller instance, so there is no combination of outputs. The solution to the original instance is simply the solution to the smaller instance.

Binary Search (Recursive)

Problem: ***Determine whether x is in the sorted array S of size n.***

Inputs: ***positive integer n, sorted (nondecreasing order) array of keys S indexed from 1 to n, a key x.***

Outputs: ***location, the location of x in S (0 if x is not in S).***

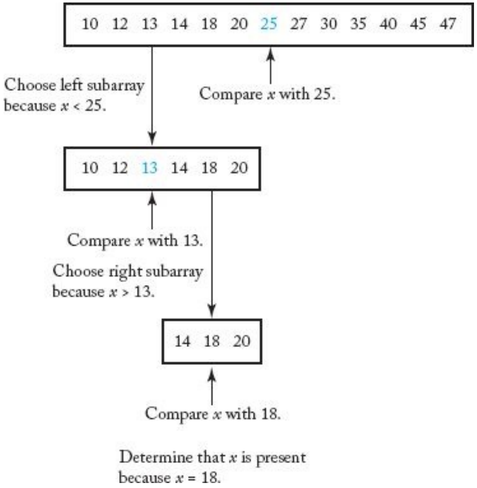


```
index location (index low, index high)
{
  index mid;

  if (low > high)
    return 0;
  else {
    mid = ⌊(low + high)/2⌋;
    if (x == S[mid])
      return mid
    else if (x < S[mid])
      return location(low, mid − 1);
    else
      return location(mid + 1, high);
  }
}
```

Notice that n, S, and x are not parameters to function location.

فرض کنید که داریم $x = 18$:



Worst-Case Time Complexity (Binary Search, Recursive)

Basic operation: the comparison of x with $S[mid]$.

Input size: n, the number of items in the array.

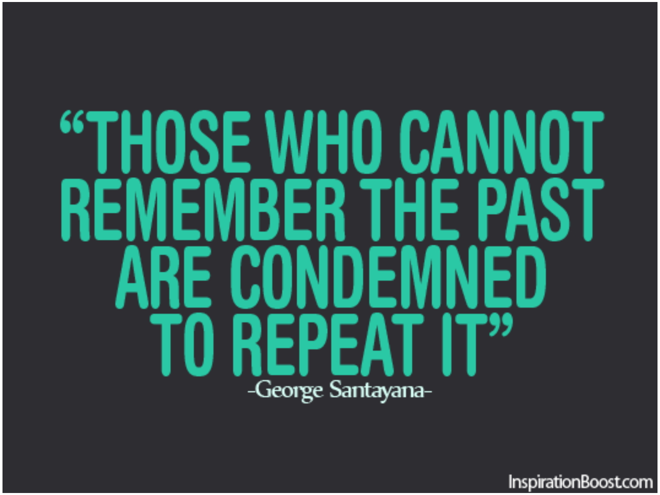
One way the worst case can occur is when x is larger than all array items. If n is a power of 2 and x is larger than all the array items, each recursive call reduces the instance to one exactly half as big.

$$W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{Comparisons in recursive call}} + \underbrace{1}_{\text{Comparison at top level}}$$

مثلاً با *Master Theorem* می‌توان دید که:

$$W(n) \in \Theta(\log(n))$$

Dynamic Programming


"THOSE WHO CANNOT REMEMBER THE PAST ARE CONDEMNED TO REPEAT IT"
-George Santayana-
InspirationBoost.com

Recall that the number of terms computed by the divide-and-conquer algorithm for determining the nth Fibonacci term is exponential in n. The reason is that the divide-and-conquer approach solves an instance of a problem by dividing it into smaller instances and then blindly solving these smaller instances. This is a top-down approach. It works in problems such as Mergesort, where the smaller instances are unrelated. They are unrelated because each consists of an array of keys that must be sorted independently. However, in problems such as the nth Fibonacci term, the smaller instances are related.

For example, to compute the fifth Fibonacci term we need to compute the fourth and third Fibonacci terms. However, the determinations of the fourth and third Fibonacci terms are related in that they both require the second Fibonacci term. Because the divide-and-conquer algorithm makes these two determinations independently, it ends up computing the second Fibonacci term more than once. In problems where the smaller instances are related, a divide-and-conquer algorithm often ends up repeatedly solving common instances, and the result is a very inefficient algorithm.

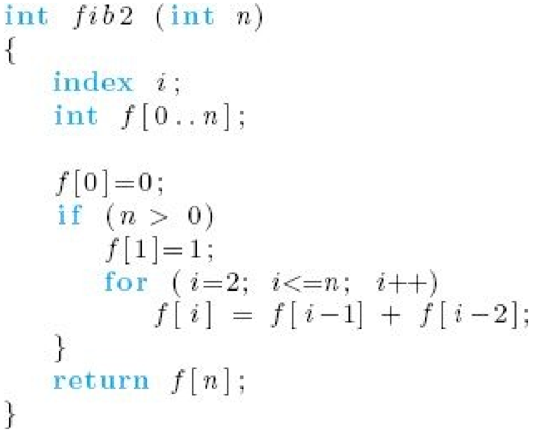
Dynamic programming takes the opposite approach. Dynamic programming is similar to divide-and-conquer in that an instance of a problem is divided into smaller instances. However, in this approach we solve small instances first, store the results, and later, whenever we need a result, look it up instead of recomputing it. In a dynamic programming algorithm, we construct a solution from the bottom up in an array (or sequence of arrays). Dynamic programming is therefore a bottom-up approach. The steps in the development of a dynamic programming algorithm are as follows: 1. Establish a recursive property that gives the solution to an instance of the problem. 2. Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.

nth Fibonacci Term

***Problem:* Determine the nth term in the Fibonacci sequence.**

***Inputs:* a nonnegative integer n.**

***Outputs:* $fib2$, the nth term in the Fibonacci sequence.**



```
int fib2 (int n)
{
    index i;
    int f[0..n];

    f[0]=0;
    if (n > 0)
        f[1]=1;
        for (i=2; i<=n; i++)
            f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

The Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 0 \leq k \leq n,$$

For values of n and k that are not small, we cannot compute the binomial coefficient directly from this definition because $n!$ is very large even for moderate values of n.

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n, \\ 1, & k = 0 \text{ or } k = n. \end{cases}$$

We can eliminate the need to compute $n!$ or $k!$ by using this recursive property.

Binomial Coefficient Using Divide-and-Conquer

***Problem:* Compute the binomial coefficient.**

***Inputs:* nonnegative integers n and k, where $k \leq n$.**

***Outputs:* bin, the binomial coefficient $\binom{n}{k}$.**

```
int bin (int n, int k)
{
   if (k == 0 || n == k)
      return 1;
   else
      return bin(n-1, k - 1)+bin(n - 1, k);
}
```

. . . .

This algorithm is very inefficient.

We showed that the divide-and-conquer algorithm for the above algorithm computes $2\binom{n}{k} - 1$ terms to determine $\binom{n}{k}$.

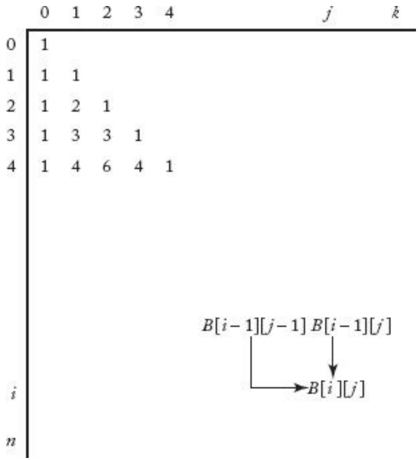
The problem is that the same instances are solved in each recursive call. For example, $bin(n-1,k-1)$ and $bin(n-1,k)$ both need the result of $bin(n-2,k-1)$, and this instance is solved separately in each recursive call. The divide-andconquer approach is always inefficient when an instance is divided into two smaller instances that are almost as large as the original instance.

ما از آرایه‌ای دوبعدی با نام B که مقدار درایهٔ موضع (i,j) آن برابر با $\binom{i}{j}$ است بهره می‌گیریم. حال با توجه به رابطهٔ

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n, \\ 1, & k = 0 \textbf{ or } k = n. \end{cases}$$

داریم:

$$B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j], & 0 < j < i, \\ 1, \quad j = 0 \text{ or } j = i. \end{cases}$$

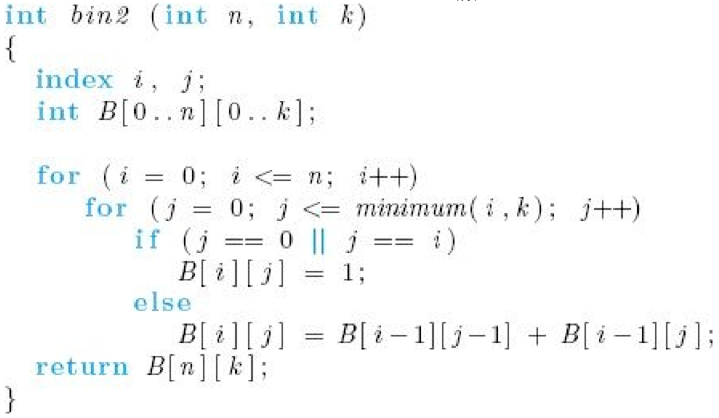

0 1 2 3 4 j k
0 1 2 3 4 i n
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
B[i − 1][j − 1]
B[i − 1][j]
B[i][j]

Binomial Coefficient Using Dynamic Programming

Problem: ***Compute the binomial coefficient.***

Inputs: ***nonnegative integers n and k, where $k \leq n$.***

Outputs: ***bin2, the binomial coefficient $\binom{n}{k}$.***



```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for (i = 0; i <= n; i++)
     for (j = 0; j <= minimum(i,k); j++)
        if (j == 0 || j == i)
           B[i][j] = 1;
        else
           B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```

The following table shows the number of passes for each value of i:

i	0	1	2	3	$\cdots$	k	$k+1$	$\cdots$	n
Number of passes	1	2	3	4	$\cdots$	$k+1$	$k+1$	$\cdots$	$k+1$

The total number of passes is therefore given by

$$1+2+3+4+\cdots+k+\underbrace{(k+1)+(k+1)\cdots+(k+1)}_{n-k+1 \text{ times}}.$$

This expression equals

$$\frac{k(k+1)}{2}+(n-k+1)(k+1)=\frac{(2n-k+2)(k+1)}{2}\in\Theta(nk).$$

By using dynamic programming instead of divide-and-conquer, we have developed a much more efficient algorithm.

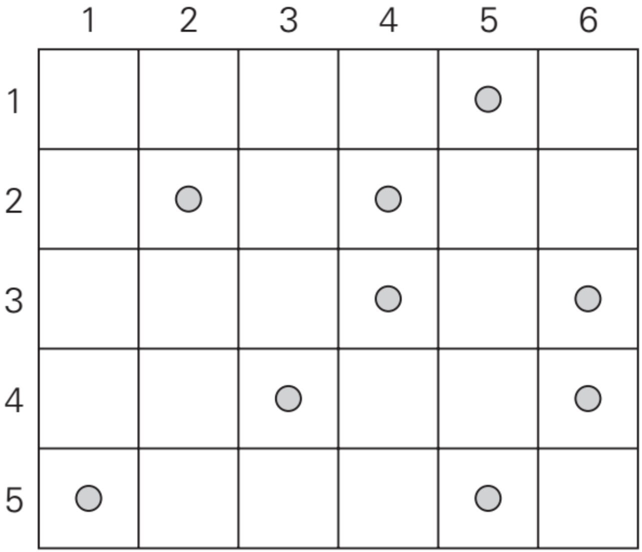
Dynamic programming is similar to divide-and-conquer in that we find a recursive property that divides an instance into smaller instances. The difference is that in dynamic programming we use the recursive property to iteratively solve the instances in sequence, starting with the smallest instance, instead of blindly using recursion. In this way we solve each smaller instance just once. Dynamic programming is a good technique to try when divide-and conquer leads to an inefficient algorithm.

Once a row is computed, we no longer need the values in the row that precedes it. Therefore, the algorithm can be written using only a one-dimensional array indexed from 0 to k. Another improvement to the algorithm would be to take advantage of the fact that $\binom{n}{k} = \binom{n}{n-k}$.

Exercise: ***Modify the algorithm (Binomial Coefficient Using Dynamic Programming) so that it uses only a one-dimensional array indexed from 0 to k.***

Coin-collecting problem

Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.


1
2
3
4
5
6
1
2
3
4
5

The brute-force algorithm ($n \times m$ *table*)

در یک جدول $n \times m$ ما $n - 1$ حرکت رو به پایین و $m - 1$ حرکت رو به راست داریم: در مجموع $n + m - 2$ حرکت. پس تعداد کل مسیرها برابر است با

$$\binom{n+m-2}{m-1} = \binom{n+m-2}{n-1}.$$

Let $F(i,j)$ be the largest number of coins the robot can collect and bring to the cell (i,j) in the ith row and jth column of the board. It can reach this cell either from the adjacent cell $(i-1,j)$ above it or from the adjacent cell $(i,j-1)$ to the left of it. The largest numbers of coins that can be brought to these cells are $F(i-1,j)$ and $F(i,j-1)$, respectively. Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of the cells in the first column. For those cells, we assume that $F(i-1,j)$ and $F(i,j-1)$ are equal to 0 for their nonexistent neighbours. Therefore, the largest number of coins the robot can bring to cell (i,j) is the maximum of these two numbers plus one possible coin at cell (i,j) itself.

$$F(i,j) = \max\{F(i-1,j), F(i,j-1)\} + c_{ij} \text{ for } 1 \le i \le n, 1 \le j \le m,$$

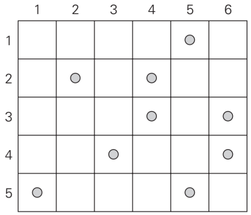
$$F(0,j) = 0 \text{ for } 1 \le j \le m, \text{ and } F(i,0) = 0 \text{ for } 1 \le i \le n,$$

where $c_{ij} = 1$ if there is a coin in cell (i,j), and $c_{ij} = 0$ otherwise.

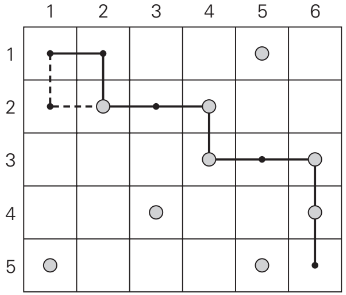
Using these formulas, we can fill in the $n \times m$ table of $F(i,j)$ values either row by row or column by column, as is typical for dynamic programming algorithms involving two-dimensional tables.

Since computing the value of $F(i,j)$ for each cell of the table takes constant time, the time efficiency of the algorithm is $\Theta(nm)$. Its space efficiency is, obviously, also $\Theta(nm)$.

Tracing the computations backward makes it possible to get an optimal path: if $F(i-1,j) > F(i,j-1)$, an optimal path to cell (i,j) must come down from the adjacent cell above it; if $F(i-1,j) < F(i,j-1)$, an optimal path to cell (i,j) must come from the adjacent cell on the left; and if $F(i-1,j) = F(i,j-1)$, it can reach cell (i,j) from either direction. If ties are ignored, one optimal path can be obtained in $\Theta(n+m)$ time.



	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**



The maximum-subarray problem

ما یک الگوریتم مبتنی بر تقسیم و غلبه که برای آن داشتیم $T(n) \in \Theta(n \log(n))$ برای این مسئله ارائه کردیم. حالا خواهیم دید که با بهره‌گیری از راهبرد برنامه‌ریزی پویا می‌شود یک الگوریتم با $T(n) \in \Theta(n)$ برای این مسئله طراحی کرد.

آرایهٔ ورودی را A بنامید. ما برای حل مسئله از یک آرایهٔ کمکی بهره می‌گیریم. مقدار درایهٔ iاُم از آرایهٔ کمکی، اگر صفر نباشد، برابر با مجموع درایه‌های بهترین زیرآرایهٔ متصلی است که با درایهٔ $A[i]$ خاتمه می‌یابد:

$$\text{auxiliary}[i] = \max\{0, \text{auxiliary}[i-1] + A[i]\}.$$

(در مثال زیر سطر اول همان آرایهٔ ورودی A است و سطر دوم آرایهٔ کمکی است.)

13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7
13	10	0	20	17	1	0	18	38	31	43	38	16	31	27	34

```java
static int maximumSubarray(int[] A) {
    int maxSoFar = 0;
    int[] auxiliary = new int[A.length];
    auxiliary[0] = Math.max(0, A[0]);
    for (int i = 1; i < auxiliary.length; i++) {
        auxiliary[i] = Math.max(0, A[i] + auxiliary[i - 1]);
        maxSoFar = Math.max(maxSoFar, auxiliary[i]);
    }
    return maxSoFar;
}
```

به‌جای یک آرایه می‌توان صرفاً از یک متغیر استفاده کرد (حافظهٔ کمتر):

```java
static int betterMaximumSubarray(int[] A) {
    int maxSoFar = 0, maxEndingHere = 0;
    for (int i = 0; i < A.length; i++) {
        maxEndingHere = Math.max(0, A[i] + maxEndingHere);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }
    return maxSoFar;
}
```