بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ــ دانشکدهٔ مهندسی برق و کامپیوتر

(نیم‌سال تحصیلی ۴۰۰۱)

طراحی الگوریتم‌ها

حسین فلسفین

Chapter 2: Divide-and-Conquer

The divide-and-conquer approach divides an instance of a problem into two or more smaller instances. The smaller instances are usually instances of the original problem. If solutions to the smaller instances can be obtained readily, the solution to the original instance can be obtained by combining these solutions. If the smaller instances are still too large to be solved readily, they can be divided into still smaller instances. This process of dividing the instances continues until they are so small that a solution is readily obtainable.

The divide-and-conquer approach is a top-down approach. That is, the solution to a top-level instance of a problem is obtained by going down and obtaining solutions to smaller instances.

The Fibonacci Sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...)

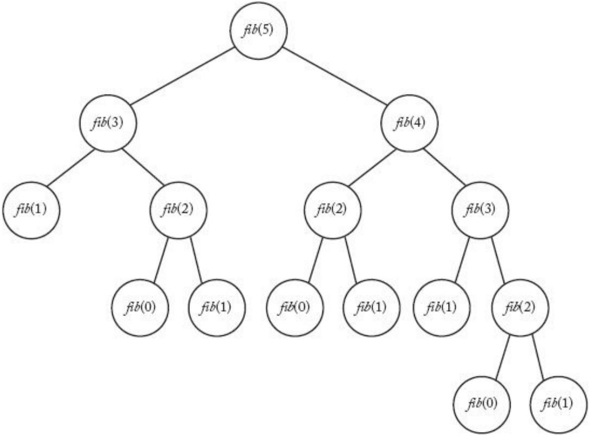
$$f_0 = 0,\ f_1 = 1,\ f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2.$$

There are various applications of the Fibonacci sequence in computer science and mathematics.

Problem: *Determine the nth term in the Fibonacci sequence.*
Inputs: *a nonnegative integer n.*
Outputs: *The nth term of the Fibonacci sequence.*

```
int fib (int n)
{
    if (n <= 1)
        return n;
    else
        return fib(n-1) + fib(n-2);
}
```

Although the algorithm was easy to create and is understandable, it is extremely inefficient. The divide-and-conquer approach leads to very efficient algorithms for some problems, but very inefficient algorithms for other problems.

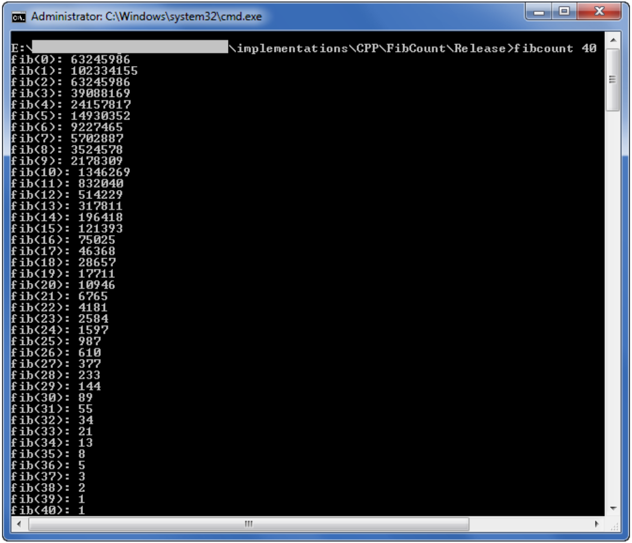


```cpp
#include <vector>
#include <string>
#include <iostream>
using namespace std;

vector<unsigned> cnt;

unsigned fib(unsigned i)
{
    cnt[i]++;
    if (i < 2)
        return i;
    return fib(i - 1) + fib(i - 2);
}

void main(int argc, char** argv)
{
    unsigned n = stoul(argv[1]);
    cnt.resize(n + 1);
    fib(n);
    for (unsigned i = 0; i <= n; i++)
        cout << "fib(" << i << "): " << cnt[i] << "\n";
}
```


Administrator: C:\Windows\system32\cmd.exe
E:\[redacted]\implementations\CPP\FibCount\Release>fibcount 40
fib(0): 63245986
fib(1): 102334155
fib(2): 63245986
fib(3): 39088169
fib(4): 24157817
fib(5): 14930352
fib(6): 9227465
fib(7): 5702887
fib(8): 3524578
fib(9): 2178309
fib(10): 1346269
fib(11): 832040
fib(12): 514229
fib(13): 317811
fib(14): 196418
fib(15): 121393
fib(16): 75025
fib(17): 46368
fib(18): 28657
fib(19): 17711
fib(20): 10946
fib(21): 6765
fib(22): 4181
fib(23): 2584
fib(24): 1597
fib(25): 987
fib(26): 610
fib(27): 377
fib(28): 233
fib(29): 144
fib(30): 89
fib(31): 55
fib(32): 34
fib(33): 21
fib(34): 13
fib(35): 8
fib(36): 5
fib(37): 3
fib(38): 2
fib(39): 1
fib(40): 1

Theorem: *If $T(n)$ is the number of terms in the recursion tree corresponding to our algorithm, then, for $n \geq 2$, $T(n) \geq 2^{n/2}$.*

Proof: *The proof is by induction on n.*

Induction base: We need two base cases because the induction step assumes the results of two previous cases. For $n = 2$ and $n = 3$, the above figure shows that $T(2) = 3 > 2^1 = 2^{2/2}$ and $T(3) = 5 > 2.8323 \approx 2^{3/2}$.

Induction hypothesis: One way to make the induction hypothesis is to assume that the statement is true for all $m < n$. Then, in the induction step, show that this implies that the statement must be true for n. This technique is used in this proof. Suppose for all m such that $2 \leq m < n$

$$T(m) > 2^{m/2}.$$

Induction step: ***We must show that*** $T(n) > 2^{n/2}$***. The value of*** $T(n)$ ***is the sum of*** $T(n-1)$ ***and*** $T(n-2)$ ***plus the one node at the root. Therefore,***

$$T(n) = T(n-1) + T(n-2) + 1 > 2^{(n-1)/2} + 2^{(n-2)/2} + 1 > 2^{(n-2)/2} + 2^{(n-2)/2} = 2 \times 2^{\frac{n}{2}-1} = 2^{n/2}.$$

The Binomial Coefficient

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

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.

Proposition: *For all positive integers k and n, $k \leq n$,*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

Proof.

$$\begin{aligned}
\binom{n-1}{k-1} + \binom{n-1}{k} &= \frac{(n-1)!}{(k-1)! \cdot (n-k)!} + \frac{(n-1)!}{k! \cdot (n-k-1)!} \\
&= \frac{k \cdot (n-1)! + (n-k) \cdot (n-1)!}{k! \cdot (n-k)!} \\
&= \frac{n \cdot (n-1)!}{k! \cdot (n-k)!} = \frac{n!}{k! \cdot (n-k)!} = \binom{n}{k}.
\end{aligned}$$

$\square$

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.

Exercise: *Use induction on n to show that the divide-and-conquer algorithm for the above algorithm computes $2\binom{n}{k} - 1$ terms to determine $\binom{n}{k}$.*

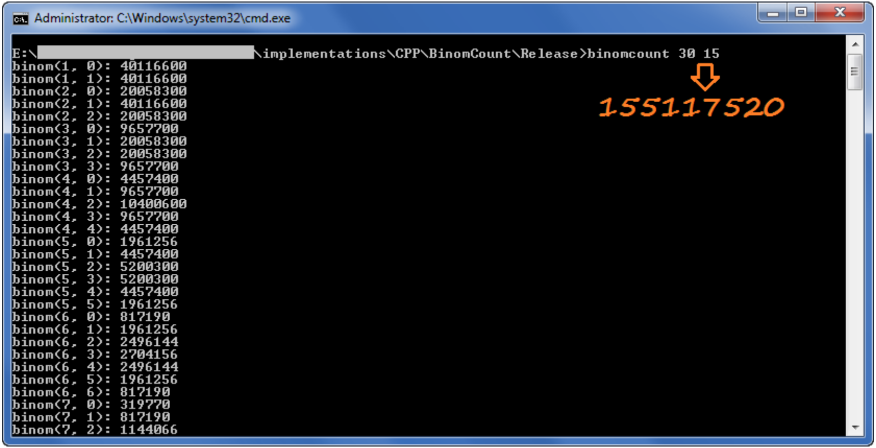
The problem is that the same instances are solved in each recursive call. For example, $bin(n-1,k-1)$ and $bin(n-1,k)$ both need the result of $bin(n-2,k-1)$, and this instance is solved separately in each recursive call. The divide-andconquer approach is always inefficient when an instance is divided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

```cpp
#include <map>
#include <string>
#include <iostream>
using namespace std;

map<pair<unsigned, unsigned>, unsigned> cnt;

unsigned binom(unsigned n, unsigned k)
{
    cnt[make_pair(n, k)]++;
    if (k == 0 || k == n)
        return 1;
    return binom(n - 1, k - 1) + binom(n - 1, k);
}

void main(int argc, char** argv)
{
    unsigned n = stoul(argv[1]), k = stoul(argv[2]);
    binom(n, k);
    for (auto it = cnt.begin(); it != cnt.end(); it++)
        cout << "binom(" << (*it).first.first << ", " << (*it).first.second << "): " << (*it).second << "\n";
}
```


Administrator: C:\Windows\system32\cmd.exe
E:\ \implementations\CPP\BinomCount\Release>binomcount 30 15
binom(1, 0): 40116600
binom(1, 1): 40116600
binom(2, 0): 20058300
binom(2, 1): 40116600
binom(2, 2): 20058300
binom(3, 0): 9657700
binom(3, 1): 20058300
binom(3, 2): 20058300
binom(3, 3): 9657700
binom(4, 0): 4457400
binom(4, 1): 9657700
binom(4, 2): 10400600
binom(4, 3): 9657700
binom(4, 4): 4457400
binom(5, 0): 1961256
binom(5, 1): 4457400
binom(5, 2): 5200300
binom(5, 3): 5200300
binom(5, 4): 4457400
binom(5, 5): 1961256
binom(6, 0): 817190
binom(6, 1): 1961256
binom(6, 2): 2496144
binom(6, 3): 2704156
binom(6, 4): 2496144
binom(6, 5): 1961256
binom(6, 6): 817190
binom(7, 0): 319770
binom(7, 1): 817190
binom(7, 2): 1144066
155117520

Finding the peak entry of a unimodal sequence

Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], ..., A[n]$ is unimodal: For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.) You'd like to find the "peak entry" p without having to read the entire array–in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log(n))$ entries of A.

We should probe the midpoint of the array and try to determine whether the "peak entry" p lies before or after this midpoint. So suppose we look at the value $A\left[\frac{n}{2}\right]$. From this value alone, we can't tell whether p lies before or after $\frac{n}{2}$, since we need to know whether entry $\frac{n}{2}$ is sitting on an "up-slope" or on a "down-slope." So we also look at the values $A\left[\frac{n}{2}-1\right]$ and $A\left[\frac{n}{2}+1\right]$. There are now three possibilities.

* If $A\left[\frac{n}{2}-1\right] < A\left[\frac{n}{2}\right] < A\left[\frac{n}{2}+1\right]$, then entry $\frac{n}{2}$ must come strictly before p, and so we can continue recursively on entries $\frac{n}{2}+1$ through n.

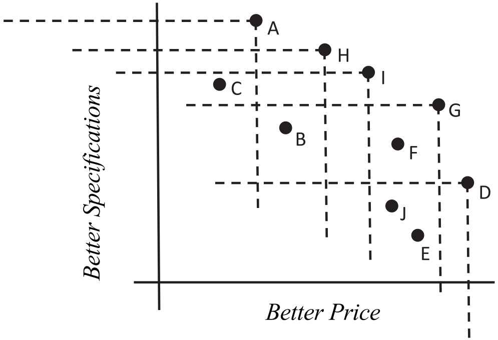
* If $A\left[\frac{n}{2}-1\right] > A\left[\frac{n}{2}\right] > A\left[\frac{n}{2}+1\right]$, then entry $\frac{n}{2}$ must come strictly after p, and so we can continue recursively on entries 1 through $\frac{n}{2}-1$.

* Finally, if $A\left[\frac{n}{2}\right]$ is larger than both $A\left[\frac{n}{2}-1\right]$ and $A\left[\frac{n}{2}+1\right]$, we are done: the peak entry is in fact equal to $\frac{n}{2}$ in this case.

If one needs to compute something using only $O(\log(n))$ operations, a useful strategy is to perform a constant amount of work, throw away half the input, and continue recursively on what's left.

The Maxima-Set Problem

فرض کنید که قصد داریم یک گوشی موبایل بخریم و آنچه مدنظر ما است اولاً قیمت و ثانیاً مشخصات سخت‌افزاری و قدرتمند بودن سخت‌افزار است.



بدیهی است که گزینه‌های B, C, E, F, J را دیگر نباید مدنظر قرار دهیم. چرا؟ چون گزینه یا گزینه‌هایی را در اختیار داریم که هم از حیث قیمت و هم از حیث قدرت سخت‌افزار ارجحیت بیشتری دارند نسبت به گزینه‌های فوق.

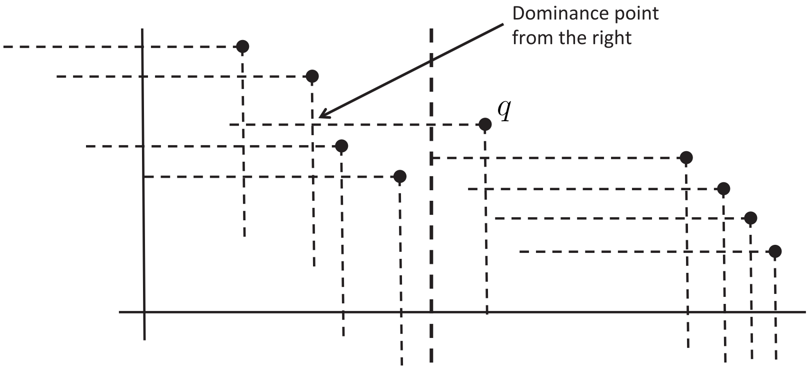
The Maxima-Set Problem

This problem is motivated from multi-objective optimization, where we are interested in optimizing choices that depend on multiple variables. A point is a maximum point in S if there is no other point, (x', y'), in S such that $x \leq x'$ and $y \leq y'$. Points that are not members of the maxima set can be eliminated from consideration, since they are dominated by another point in S. Thus, finding the maxima set of points can act as a kind of filter that selects out only those points that should be candidates for optimal choices.

The Maxima-Set Problem

Given a set, S, of n points in the plane, there is a simple divide-and-conquer algorithm for constructing the maxima set of points in S. If $n \leq 1$, the maxima set is just S itself. Otherwise, let p be the median point in S according to a lexicographic ordering of the points in S, that is, where we order based primarily on x-coordinates and then by y-coordinates if there are ties. If the points have distinct x-coordinates, then we can imagine that we are dividing S using a vertical line through p. Next, we recursively solve the maxima-set problem for the set of points on the left of this line and also for the points on the right.

The Maxima-Set Problem



The Maxima-Set Problem

Given these solutions, the maxima set of points on the right are also maxima points for S. But some of the maxima points for the left set might be dominated by a point from the right, namely the point, q, that is leftmost. So then we do a scan of the left set of maxima, removing any points that are dominated by q, until reaching the point where q's dominance extends. The union of remaining set of maxima from the left and the maxima set from the right is the set of maxima for S.

The Maxima-Set Problem

Algorithm MaximaSet(S):

Input: A set, S, of n points in the plane
Output: The set, M, of maxima points in S

if $n \leq 1$ **then**
    return S
Let p be the median point in S, by lexicographic (x, y)-coordinates
Let L be the set of points lexicographically less than p in S
Let G be the set of points lexicographically greater than or equal to p in S
$M_1 \leftarrow$ MaximaSet(L)
$M_2 \leftarrow$ MaximaSet(G)
Let q be the lexicographically smallest point in M_2
for each point, r, in M_1 **do**
    if $x(r) \leq x(q)$ **and** $y(r) \leq y(q)$ **then**
        Remove r from M_1
return $M_1 \cup M_2$

The Divide-and-Conquer Approach

You should now better understand the following general description of this approach. The divide-and-conquer design strategy involves the following steps:

1. *Divide an instance of a problem into two or more smaller instances.*
2. *Conquer (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.*
3. *If necessary, combine the solutions to the smaller instances to obtain the solution to the original instance.*

The reason we say "if necessary" in Step 3 is that in algorithms such as our algorithm for finding the peak entry of a unimodal sequence, the instance is reduced to just one smaller instance, so there is no need to combine solutions.

Master Theorem

In the most typical case of divide-and-conquer a problem's instance of size n is divided into two instances of size $\frac{n}{2}$. More generally, an instance of size n can be divided into b instances of size $\frac{n}{b}$, with a of them needing to be solved. (Here, a and b are constants; $a \geq 1$ and $b > 1$.) Assuming that size n is a power of b to simplify our analysis, we get the following recurrence for the running time $T(n)$:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

where $f(n)$ is a function that accounts for the time spent on dividing an instance of size n into instances of size $\frac{n}{b}$ and combining their solutions. The recurrence $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ is called the general divide-and-conquer recurrence. Obviously, the order of growth of its solution $T(n)$ depends on the values of the constants a and b and the order of growth of the function $f(n)$.

The efficiency analysis of many divide-and-conquer algorithms is greatly simplified by the following theorem:

Master Theorem: ***If*** $f(n) \in \Theta(n^k)$ ***where*** $k \geq 0$ ***in the recurrence*** $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, ***then***

$$T(n) \in \begin{cases} \Theta(n^k), & \text{if } a < b^k, \\ \Theta(n^k \log(n)), & \text{if } a = b^k, \\ \Theta(n^{\log_b(a)}), & \text{if } a > b^k. \end{cases}$$

Of course, this approach can only establish a solution's order of growth to within an unknown multiplicative constant, whereas solving a recurrence equation with a specific initial condition yields an exact answer (at least for n*'s that are powers of* b*).*