Chapter 6: Decrease-and-Conquer

Introduction

Decrease by a constant

The size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one.

Example: Computing a^n.

$$f(n) = \begin{cases} f(n-1) \times a & n > 1 \\ a & n = 1 \end{cases}$$

Example: Insertion sort

Decrease by a constant factor

The size of an instance is reduced by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

Example: Binary search

Example: Computing a^n.

$$a^n = \begin{cases} (a^{n/2})^2 & n \equiv_2 0 \\ \left(a^{\lfloor n/2 \rfloor}\right)^2 \times a & n \equiv_2 1 \wedge n \geq 1 \\ 1 & n = 0 \end{cases}$$

Example: Russian peasant method of multiplication

$$n \times m = \begin{cases} \frac{n}{2} \times 2m & n \equiv_2 0 \\ \left\lfloor \frac{n}{2} \right\rfloor \times 2m + m & n \equiv_2 1 \end{cases}$$

Variable size decrease

The size-reduction pattern varies from one iteration of an algorithm to another.

Example: Euclid's algorithm for computing the greatest common divisor

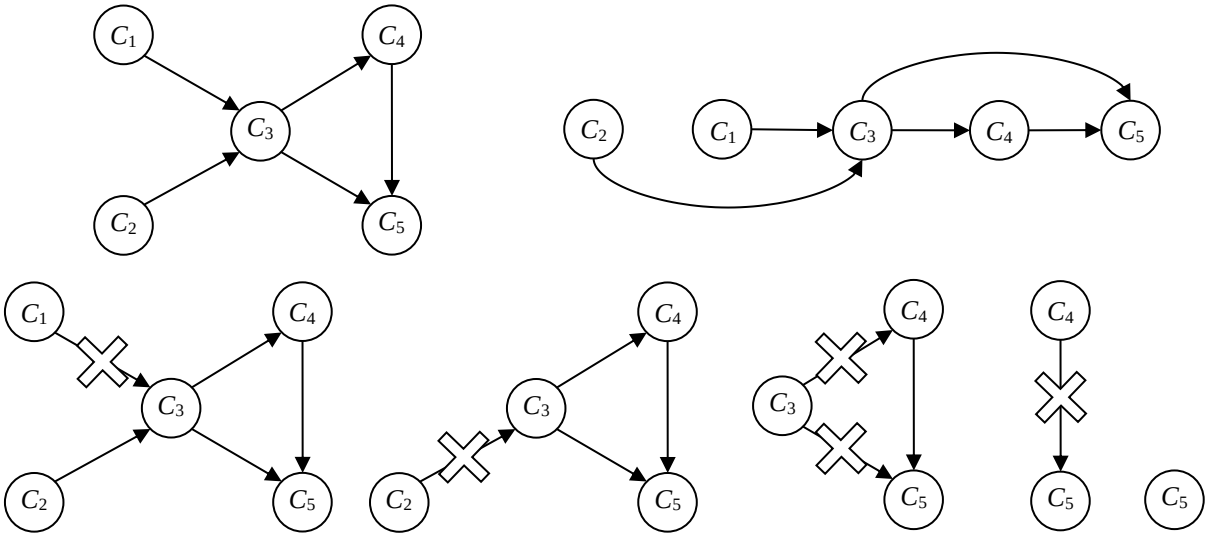
$$USCLN(a, b) = USCLN(b, a \,\%\, b)$$

Example: Finding an element in a binary search tree

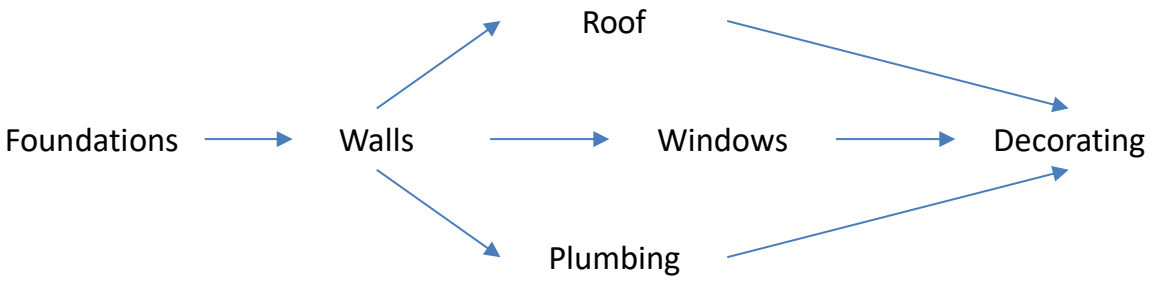
Decrease by a constant

Topological Sorting

Example: Consider a set of five required courses $\{C_1, C_2, C_3, C_4, C_5\}$. The courses can be taken in any order as long as the following course prerequisites are met: C_1 and C_2 have no prerequisites, C3 requires C_1 and C_2, C_4 requires C_3, and C_5 requires C_3 and C_4. A student can take only one course per term. In which order should the student take the courses?



Example: Building a house



A possible sequence:

Foundations → Walls → Roof → Windows → Plumbing → Decorating

A rough algorithm:

Repeatedly, identify in a remaining digraph a *source*, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it. The order in which the vertices are deleted yields a solution to the topological sorting problem.

Note: If there are several sources, break the tie arbitrarily. If there are none, stop because the problem cannot be solved.

Giải thuật:

```
TopologicalSort(Graph G) {
  indegree[1 .. |V|] = {0};
  priorQueue  Q = ∅;

  for (each vertex u ∈ V)
    for (each vertex v that is adjacent from u)
      indegree[v] ++;
  for (each vertex v ∈ V)
    if (indegree[v] == 0)
      enQueue(Q, v);

  while (!isEmpty(Q)) {
    Vertex u = deQueue(Q);
    Output(u);
    for (each vertex v that is adjacent from u)
      if( -- indegree[v] == 0 )
        enQueue(Q, v);
  }
}
```

Analysis of the algorithm

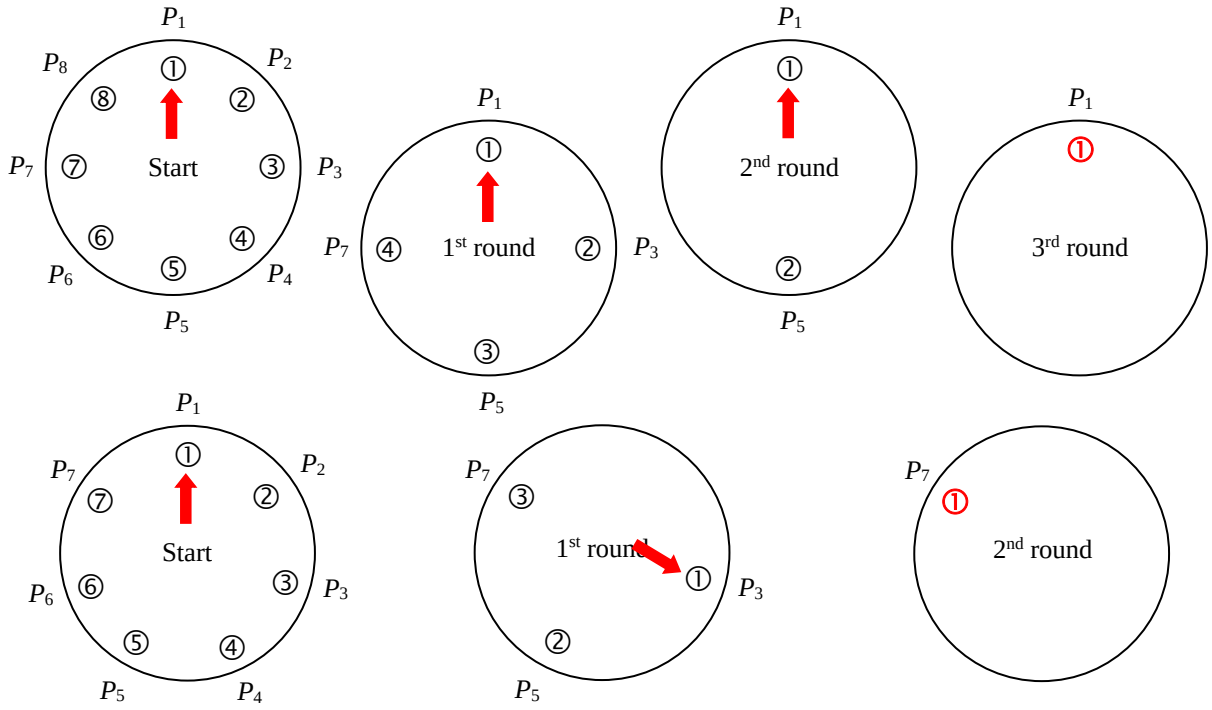
If the graph representation is an adjacency list, the running time of the algorithm is $\Theta(|V| + |E|)$.

Decrease by a constant factor

Josephus Problem

Let n people numbered 1 to n stand on a circle. Starting the count with person number 1, we eliminate every second person until only one survivor is left. The problem is to determine the survivor's number $J(n)$.

Example: $J(8) = 1, J(7) = 7, J(6) = 5$.



In order to determine $J(n)$, we consider the cases of even and odd n's separately:

- $n = 2h$:

$$J(2h) = 2J(h) - 1$$

- $n = 2h + 1$:

$$J(2h + 1) = 2J(h) + 1$$

A closed-form solution to the two-case recurrence is as follows:

$$J(2^k + i) = 2i + 1, i \in [0, 2^k - 1]$$

Variable size decrease

Euclid's algorithm for computing the $gcd(a, b)$*, where* $a, b \in \mathbb{Z}^+$

Algorithm

```
gcd(a, b) {
   while (b) {
      t = b;
      b = a % b;
      a = t;
   }
   return a;
}
```

Analysis of the algorithm

Lame's theorem: Let a and b be positive integers with $a \geq b \geq 2$. Then the number of divisions used by the Euclidean algorithm to find $gcd(a, b)$ is less than or equal to five times the number of decimal digits in b.

Finding an element in a binary search tree

Analysis of the algorithm

- The best case: $\Theta(\log n)$
- The worst case: $\Theta(n)$
- The average case: $\Theta(\log n)$

Selection problem

Finding the k^{th} smallest number in an array $S = \{s_1, s_2, \dots, s_n\}$

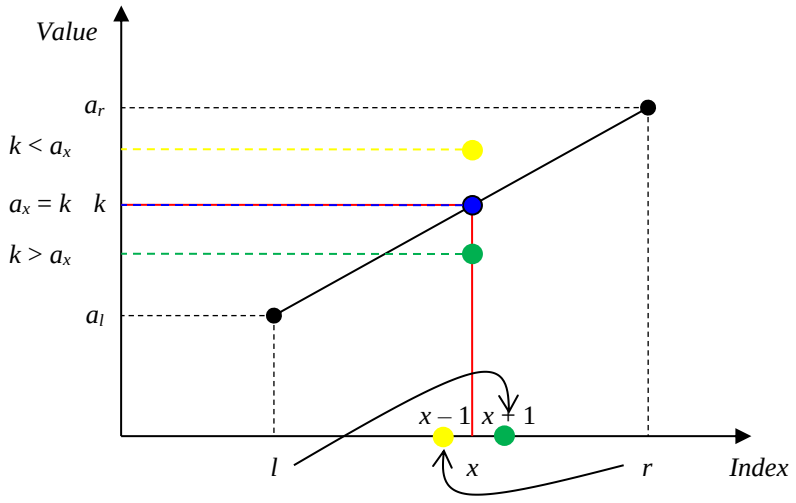
Giải thuật

```
Selection(S[], lower, upper, k) {
  h = random(lower, upper);

  pos = Partition(S, lower, upper, h);

  if (pos == k)
    return S[pos];
  if (pos > k)
    Selection(S, lower, pos - 1, k);
  if (pos < k)
    Selection(S, pos + 1, upper, k);
}
Partition(S, lower, upper, pos) {
  if (lower == upper)
    return lower;

  pivot = S[pos];
  S[lower] ⇆ S[pos];
  pos = lower;
  for (i = lower + 1; i ≤ upper; i++)
    if (pivot > S[i]) {
      pos++;
      S[i] ⇆ S[pos];
    }
  S[lower] ⇆ S[pos];

  return pos;
}
```

Analysis of the algorithm

- The best case: $\Theta(n)$
- The worst case: $O(n^2)$
- The average case: $\Theta(n)$

Interpolation Search



Let k be the search key. The algorithm assumes that the array values are in ascending order. The search key k will be compared with the element whose index (called x) is computed as (the round-off of) the x-coordinate of the point on the straight line through the points (l, a_l) and (r, a_r) whose y-coordinate is equal to k (Figure).

The following formula shows the way to compute x:

$$\frac{x-l}{r-l} = \frac{k-a_l}{a_r-a_l} \Longrightarrow x = l + \left\lfloor \frac{(k-a_l)(r-l)}{a_r-a_l} \right\rfloor$$

After comparing k with a_x, there are three possibilities

- $a_x = k$: The algorithm stops.
- $a_x > k$: The algorithm proceeds by searching in the same manner among the elements indexed between l and $(r =)x - 1$.
- $a_x < k$: The algorithm proceeds by searching in the same manner among the elements indexed between $(l =)x + 1$ and r.

Analysis of the algorithm

The analysis of the algorithm's efficiency shows that interpolation search uses fewer than $\log_2 \log_2 n + 1$ key comparisons on the average when searching in an array of n random keys.