Algorithms

Freely using the textbook by Cormen, Leiserson, Rivest, Stein

Péter Gács

Computer Science Department
Boston University

Fall 2010

The class structure

See the course homepage.
In the notes, section numbers and titles generally refer to the book:
CLSR: Algorithms, third edition.

Computational problem example: sorting.

Input, output, instance.

Algorithm example: insertion sort.

Algorithm 2.1: INSERTION-SORT(A)

```
1  for j ← 2 to A.size do
2      key ← A[j]
       // Insert A[j] into the sorted sequence A[1 .. j − 1]
3      i ← j − 1
4      while i > 0 and A[i] > key do
5          A[i + 1] ← A[i]
6          i−−
7      A[i + 1] ← key
```

Pseudocode conventions

Assignment

Indentation

Objects Arrays are also objects.
Meaning of $A[3\,..\,10]$.

Parameters Ordinary parameters are passed by value.
Objects are always treated like a pointer to the body of data, which themselves are not copied.

By size of a problem, we will mean the size of its input (measured in bits).

- Cost as *function of input size*. In a problem of fixed size, the cost will be influenced by too many accidental factors: this makes it hard to draw lessons applicable to future cases.
- Our model for computation cost: Random Access Machine Until further refinement, one array access is assumed to take constant amount of time (independent of input size or array size).
- Resources: running time, memory, communication (in case of several processors or actors).

- When a program line involves only a simple instruction we assume it has constant cost. The main question is the number of times it will be executed. It is better to think in terms of larger program parts than in terms of program lines.
- If a part is a loop then the time is the overhead plus the sum of the execution times for each repetition.
- If a part is a conditonal then the time depends on which branch is taken.
- And so on.

In the example, assume:

- c_1 is the cost of comparison.
- c_2 is the cost of copying.
- Ignore everything else.
- The cost of iteration j is t_j: depends on how many times is the condition $A[i] > key$ of the while loop satisfied.

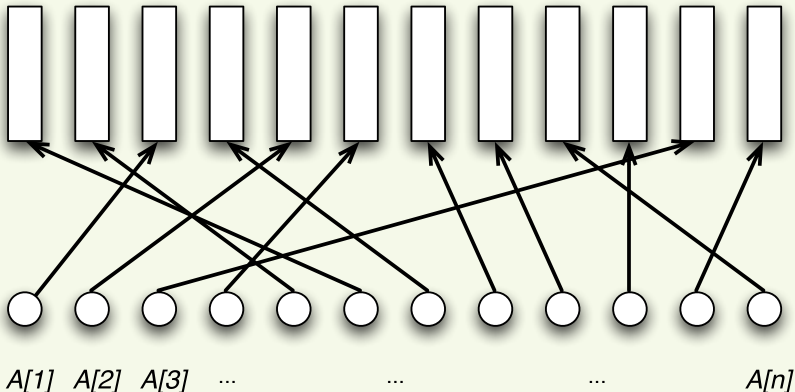
Leading term: $\sum_j t_j = \Theta(n^2)$.

In the insertion sort, every time $A[i] > key$ is found, two assignments are made. So we perform 2 comparisons (cost c_1) and 2 assignments (cost c_2). But the bound c_2 papers over some differences. The assignment

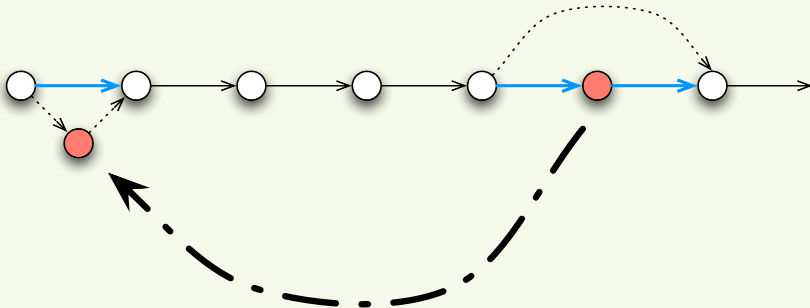
$$A[i+1] \leftarrow A[i]$$

could be the costly (who know how much data is in $A[i]$). Even without changing the algorithm, by choosing the way of storing the data can influence this cost significantly. Improvements:

- $A[i]$ should not contain the actual data if it is large, only the address of the place where it be found (a link).
- Instead of an array, use a linked list. Then insertion does not involve pushing back everything above.



The array contains links to the actual data, so the copying during sorting has a fixed low cost.



When moving an element to a different place in a linked list, most intermediate elements are not touched.

Worst case, average case

Worst case We estimated the largest cost of an algorithm for a given input size. This is what we will normally do.

Average case What does this mean?

- The notion of a random, or typical input is problematic.
- But we can introduce random choices in our algorithm, by a process called randomization. We will see examples when this can give an average performance significantly better than the worst-case performance, on *all inputs*.

Strong and weak domination

For the insertion sort, in first approximation, we just want to know that its worst-case cost “grows as” n^2, where n is the file size. In your introductory courses CS112 and CS131 you already have seen the basic notions needed to talk about this “grows as”. Here, I collect some this information: more is found here than what I have time for initially in class.

$f(n) \ll g(n)$ means $\lim_{n\to\infty} f(n)/g(n) = 0$: in words, $g(n)$ grows faster than $f(n)$. Other notation:

$$f(n) = o(g(n)) \Leftrightarrow f(n) \ll g(n) \Leftrightarrow g(n) = \omega(f(n)).$$

Example: $n - 4 \gg 116\sqrt{n} + 80$. We may also write

$$116\sqrt{n} + 80 = o(n).$$

Generally, when we write $f(n) = o(g(n))$ then $g(n)$ has a simpler form than $f(n)$ (this is the point of the notation).

$f(n) \overset{*}{<} g(n)$ means $\sup_n f(n)/g(n) < \infty$, that is $f(n) \leqslant c \cdot g(n)$ for some constant c. Other (the common) notation:

$$f(n) = O(g(n)) \Leftrightarrow f(n) \overset{*}{<} g(n) \Leftrightarrow g(n) = \Omega(f(n)).$$

(The notation $\overset{*}{<}$ is mine, you will not find it in your books.)
This is a preorder. If $f \overset{*}{<} g$ and $g \overset{*}{<} f$ then we write $f \overset{*}{=} g$, $f = \Theta(g)$, and say that f and g have the same rate of growth.
Example: $n^2 - 5n$ and $100n(n+2)$ have the same rate of growth.
We can also write

$$100n(n+2) = O(n^2), \quad 100n(n+2) = \Theta(n^2).$$

On the other hand, $n + \sqrt{n} = O(n^2)$ but not $\Theta(n^2)$.

Important special cases

- $O(1)$ denotes any function that is bounded by a constant, for example $(1+1/n)^n = O(1)$.
- $o(1)$ denotes any function that is converging to 0 as $n \to \infty$. For example, another way of writing Stirling's formula is

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}(1 + o(n)).$$

Not all pairs of functions are comparable

Here are two functions that are not comparable. Let $f(n) = n^2$, and for $k = 0, 1, 2, \ldots$, we define $g(n)$ recursively as follows. Let n_k be the sequence defined by $n_0 = 1$, $n_{k+1} = 2^{n_k}$. So, $n_1 = 2$, $n_2 = 4$, $n_3 = 16$, and so on. For $k = 1, 2, \ldots$ let

$$g(n) = n_{k+1} \text{ if } n_k < n \leqslant n_{k+1}.$$

So,
$g(n_k + 1) = n_{k+1} = 2^{n_k} = g(n_k + 1) = g(n_k + 2) = \cdots = g(n_{k+1})$.
This gives $g(n) = 2^{n-1}$ for $n = n_k + 1$ and $g(n) = n$ for $n = n_{k+1}$.
Function $g(n)$ is sometimes much bigger than $f(n) = n^2$ and sometimes much smaller: these functions are incomparable for $\ll, \overset{*}{<}$.

Important classes of increasing functions of n:

- Linear functions: (bounded by) $c \cdot n$ for arbitrary constant c.
- Polynomial functions: (bounded by) n^c for some constant $c > 0$, for $n \geqslant 2$.
- Exponential functions: those (bounded by) c^n for some constant $c > 1$.
- Logarithmic functions: (bounded by) $c \cdot \log n$ for arbitrary constant c. Note: If a function is logarithmic with $\log_2$ then it is also logarithmic with $\log_b$ for any b, since

$$\log_b x = \frac{\log_2 x}{\log_2 b} = (\log_2 x)(\log_b 2).$$

These are all equivalence classes under $\stackrel{*}{=}$.

Some simplification rules

- Addition: take the maximum, that is if $f = O(g)$ then $f + g = O(g)$. Do this always to simplify expressions. Warning: do it only if the number of terms is constant! This is wrong: $n + n + \cdots (n \text{ times}) \cdots + n \neq O(n)$.
- $f(n)^{g(n)}$ is generally worth rewriting as $2^{g(n) \log f(n)}$. For example, $n^{\log n} = 2^{(\log n) \cdot (\log n)} = 2^{\log^2 n}$.
- But sometimes we make the reverse transformation:

$$3^{\log n} = 2^{(\log n) \cdot (\log 3)} = (2^{\log n})^{\log 3} = n^{\log 3}.$$

The last form is the most meaningful, showing that this is a polynomial function.

$$n/\log\log n + \log^2 n \overset{*}{=} n/\log\log n.$$

Indeed, $\log\log n \ll \log n \ll n^{1/2}$, hence
$n/\log\log n \gg n^{1/2} \gg \log^2 n$.

Order the following functions by growth rate:

$$
\begin{aligned}
&n^2 - 3\log\log n && \overset{*}{=} n^2, \\
&\log n/n, \\
&\log\log n, \\
&n\log^2 n, \\
&3 + 1/n && \overset{*}{=} 1, \\
&\sqrt{5n}/2^n, \\
&(1.2)^{n-1} + \sqrt{n} + \log n && \overset{*}{=} (1.2)^n.
\end{aligned}
$$

Solution:

$$
\begin{aligned}
\sqrt{5n}/2^n \ll \log n/n \ll 1 \ll \log\log n \\
\ll n/\log\log n \ll n\log^2 n \ll n^2 \ll (1.2)^n.
\end{aligned}
$$

You must know the following three sums:

Arithmetic series $1+2+3+\cdots+n = \frac{n(n+1)}{2}$.

Geometric series $1+q+q^2+\cdots+q^{n-1} = \frac{1-q^n}{1-q}$.

Infinite geometric series If $|q|<1$ then $1+q+q^2+\cdots = \frac{1}{1-q}$.

Simplification of sums

For rates of growth, the following is more important:

Geometric series grows as fast as its largest element:

$$6 + 18 + \cdots + 2 \cdot 3^n \stackrel{*}{=} 3^n$$

Even more true of series growing faster, say,

$$1! + 2! + \cdots + n! \stackrel{*}{=} n!.$$

Sum of n^c (for example arithmetic series) For rate of growth, replace each term with the maximal one:

$$2^2 + 5^2 + 8^2 + \cdots + (2 + 3n)^2 \stackrel{*}{=} (n + 1)(2 + 3n)^2 \stackrel{*}{=} n^3.$$

Even more true of a series growing slower:

$$\log n! = \log 2 + \log 3 + \cdots + \log n \stackrel{*}{=} n \log n.$$

Let us derive formally, say $1^2 + 2^2 + \cdots + n^2 \stackrel{*}{=} n^3$. The upper bound is easy.
Lower bound, with $k = \lceil n/2 \rceil$:

$$\begin{aligned} 1^2 + \cdots + n^2 &\geqslant k^2 + (k+1)^2 + \cdots + n^2 \\ &\geqslant (n/2 - 1)(n/2)^2 \stackrel{*}{=} n^3. \end{aligned}$$

We will prove the following, via rough estimates:

$$1/3 + 2/3^2 + 3/3^3 + 4/3^4 + \cdots < \infty.$$

Since any exponentially growing function grows faster than the linear function, we know $n \overset{*}{<} 3^{n/2}$. Therefore $n \cdot 3^{-n} \overset{*}{<} 3^{n/2} \cdot 3^{-n} = 3^{-n/2}$, and the whole sum is

$$\overset{*}{<} 1 + q + q^2 + \cdots = \frac{1}{1-q}$$

where $q = 3^{-1/2}$.

Another example:

$$1 + 1/2 + 1/3 + \cdots + 1/n = \Theta(\log n).$$

Indeed, for $n = 2^{k-1}$, upper bound:

$$\begin{aligned} &1 + 1/2 + 1/2 + 1/4 + 1/4 + 1/4 + 1/4 + 1/8 + \cdots \\ &\qquad = 1 + 1 + \cdots + 1 \text{ (k times).} \end{aligned}$$

Lower bound:

$$\begin{aligned} &1/2 + 1/4 + 1/4 + 1/8 + 1/8 + 1/8 + 1/8 + 1/16 + \cdots \\ &\qquad = 1/2 + 1/2 + \cdots + 1/2 \text{ (k times).} \end{aligned}$$

An efficient algorithm can frequently obtained using the following idea:

1. Divide into subproblems of equal size.
2. Solve subproblems.
3. Combine results.

In order to handle subproblems, a more general procedure is often needed.

1. Subproblems: sorting $A[1\,..\,n/2]$ and $A[n/2+1\,..\,n]$
2. Sort these.
3. Merge the two sorted arrays of size $n/2$.

The more general procedures now are the ones that sort an merge arbitrary parts of an array.

Algorithm 3.1: $\textsc{Merge}(A, p, q, r)$
Merges $A[p..q]$ and $A[q+1..r]$.

1. $n_1 \leftarrow q - p + 1$; $n_2 \leftarrow r - q$
2. create array $L[1..n_1+1]$ and $R[1..n_2+1]$
3. **for** $i \leftarrow 1$ **to** n_1 **do** $L[i] \leftarrow A[p+i-1]$
4. **for** $j \leftarrow 1$ **to** n_2 **do** $R[j] \leftarrow A[q+j]$
5. $L[n_1+1] \leftarrow \infty$; $R[n_2+1] \leftarrow \infty$
6. $i \leftarrow 1$, $j \leftarrow 1$
7. **for** $k \leftarrow p$ **to** r **do**
8.   **if** $L[i] \leqslant R[j]$ **then** $A[k] \leftarrow L[j]$; $i{+}{+}$
9.   **else** $A[k] \leftarrow R[j]$; $j{+}{+}$

Why the ∞ business? These sentinel values allow to avoid an extra part for the case that L or R are exhausted. This is also why we used new arrays for the input, rather than the output.

Algorithm 3.2: MERGE-SORT(A, p, r)
Sorts $A[p\,..\,r]$.

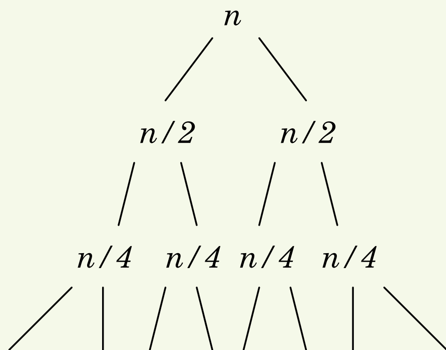
```
1  if p < r then
2      q ← ⌊(p + 1)/2⌋
3      MERGE-SORT(A, p, q); MERGE-SORT(A, q + 1, r)
4      MERGE(A, p, q, r)
```

Analysis, for the worst-case running time $T(n)$:

$$T(n) \leqslant \begin{cases} c_1 & \text{if } n = 1 \\ 2T(n/2) + c_2 n & \text{otherwise.} \end{cases}$$

Draw a recursion tree based on this inequality.

Recursion tree



Work on top level

Total work on level 1

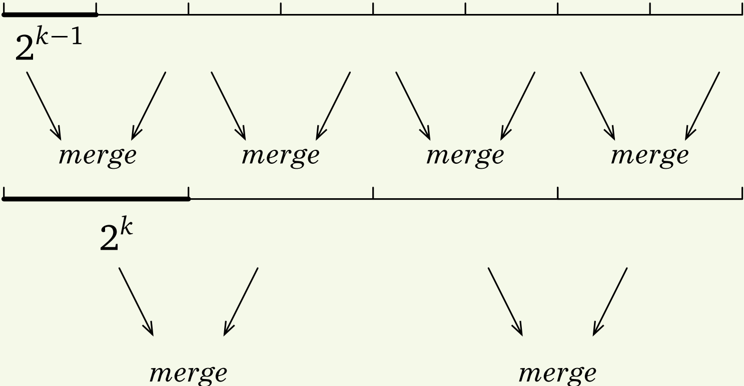
Total work on level 2

...

Assume $n = 2^k$:

$$
\begin{aligned}
&c_2 n(1 + 2 \cdot 1/2 + 4 \cdot 1/4 + \cdots + 2^k \cdot 2^{-k}) + c_1 n, \\
&= c_2 nk + c_1 n = n(c_2 \log n + c_1) = O(n \log n)
\end{aligned}
$$

Perform first the jobs at the bottom level, then those on the next level, and so on. In passes k and $k+1$:



- What if we have to sort such a large list A, that it does not fit into random access memory?
- If we are at a position p we can inspect or change $A[p]$, but in one step, we can only move to position $p+1$ or $p-1$.
- Note that the Algorithm $\text{Merge}(A, p, q, r)$ passed through arrays A, L, R in one direction only, not jumping around. But both the recursive and nonrecursive $\text{Merge-Sort}(A)$ uses random access.
- We will modify the nonrecursive $\text{Merge-Sort}(A)$ to work without random access.

Runs

For position p in array A, let $p' \geqslant p$ be as large as possible with

$$A[p] \leqslant A[p+1] \leqslant \cdots \leqslant A[p'].$$

The sub-array $A[p \mathrel{.\,.} p']$ will be called a run.
Our algorithm will consist of a number of passes. Each pass goes through array A and merges consecutive runs: namely

- performs $\textsc{Merge}(A, p, q, r)$ where $p = 1$, $q = p'$, $r = (q+1)'$,
- does the same starting with $p \leftarrow r + 1$
- and so on.

Algorithm 3.3: MERGE-RUNS(A)

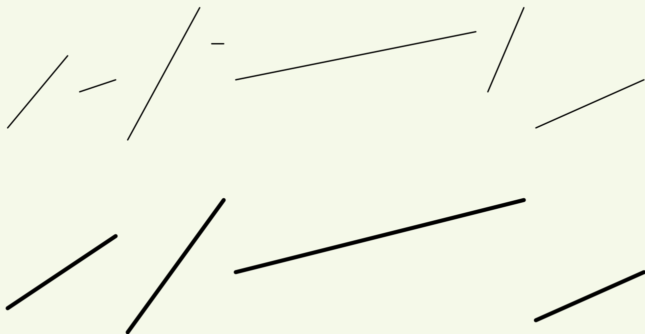
Merge consecutive runs in A, return the new number of runs.

```
1  n ← A.size; r ← 0; k ← 1
2  while r < n do
3      p ← r + 1; q ← p
4      while q < n and A[q + 1] ⩾ A[q] do q++          // q ← p′
5      if q = n then r ← n else
6          r ← q + 1; k++
7          while r < n and A[r + 1] ⩾ A[r] do r++
8          MERGE(A, p, q, r)
9  return k
```

Algorithm 3.4: DISK-SORT(A)

Sort without random access.

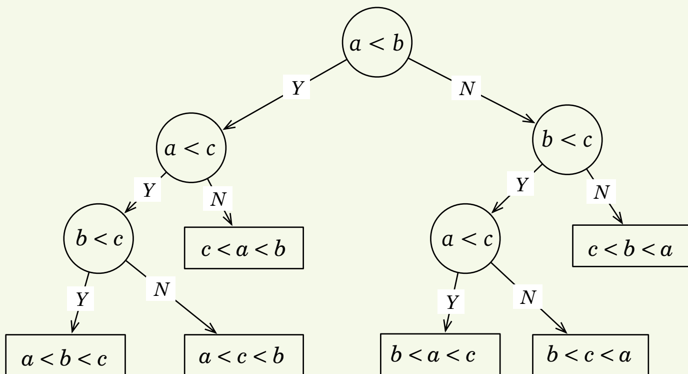
```
1  repeat k ← MERGE-RUNS(A) until k = 1
```



- Each MERGE-RUNS(A) roughly halves the number of runs.
- If you view the arrays as being on tapes, the algorithm uses still too many rewind operations to return from the end of the tape to the beginning. In an exercise, I will ask you to modify DISK-SORT(A) (and MERGE-RUNS(A)) to so as to eliminate almost all rewinds.

- Lower bounds are difficult, in general: hard to prove that every algorithm needs at least time $T(n)$ in the worst case of our problem, on inputs of length n.
- We do not have such a result even on sorting.
- Frequently, we show the lower bound only on algorithms belonging to a specific class.
- In case of sorting, there is a natural class: algorithms that get information about the array elements only in one way: comparing them to each other.
  They do not not look at details of an element: no arithmetical operations on elements, or looking at individual bits.
  We will only count the number of comparisons used.

From the work of any such algorithm, we can derive a **decision tree**.



Each execution of the algorithm gives a downward path in the tree. Worst-case time is lower-bounded by longest downward path (height) in this binary tree.

Theorem

A tree of height h has at most 2^h leaves.

Indeed, when we increase the height by 1, each leaf can give rise to at most two children, so the number of leaves can at most double.

Corollary

If a tree has L leaves, then its height is at least $\log L$.

The number of leaves is at least $n!$. Hence the height is at least $\log n!$: in fact, it is $\lceil \log n! \rceil$.

$$\log n! = \log 2 + \log 3 + \cdots + \log n \leqslant n \log n.$$

We want lower bound. Let us show that for n large enough, this is $\geqslant 0.8 n \log n$.

$$\begin{aligned} &\log 2 + \log 3 + \cdots + \log n \\ &\quad \geqslant \log(n/10) + \log(n/10 + 1) + \cdots + \log n \\ &\quad \geqslant 0.9 n \log(n/10) = 0.9 n (\log n - \log 10) \geqslant 0.8 n \log n. \end{aligned}$$

Of course, we could have written 0.99 in place of 0.8 in this result.

Theorem

Sorting n items needs at least

$$\log n! = n \log n(1 + o(1))$$

comparisons in the worst case.

Average case? We need a lower bound on the average pathlength in a binary tree with a given number L of leaves. This seems hard to compute, but we only need a lower bound! Call a binary tree optimal, if its average pathlength is as small as possible for its number of leaves.

Claim

In an optimal binary tree with height h, every path has length $h-1$ or h: hence, the average path length is $\geqslant \lfloor \log L \rfloor$.

Indeed, if there is any shorter path to some leaf l, take a leaf l' with longest path, and move it under l. This decreases the average pathlength.

Corollary

The average number of comparisons on a random list with n elements is $\geqslant \lfloor \log n! \rfloor$.

Here is an intuitive retelling of the above argument.
If all we know about an object x is that it belongs to some known set E of size m, we say that we have an uncertainty of size $\log m$ (m bits). Each comparison gives us at most 1 bit of information (decrease of uncertainty) about the permutation in question. Since there are $n!$ permutations, our uncertainty is $\log n!$, so we need that many bits of information to remove it.

Here is another decision-tree lower bound, that is not information-theoretic.

Problem Given a list of n numbers, find the largest element.

Solution (You know it: one pass.)

Other solution Tournament.

Cost $n-1$ comparisons in both cases.

Question Is this optimal?

The information-theoretical lower bound is $\log n$.

Theorem

We need at least $n-1$ comparisons in order to find the maximum.

Indeed, every non-maximum element must participate in a comparison that shows it is smaller than the one it is compared to.

Here is another algorithm that is guaranteed to take a long time: list all permutations of an array.

Why is it taking long? For a trivial reason: the output is long ($n! \cdot n$ items).

How to implement? It is not so easy to organize going through all possibilities. We will use recursion: similar techniques will help with other problems involving exhaustive search.

In what order to list? Lexicographical order: If $a_1 \dots a_k \dots a_n$ and $b_1 \dots b_k \dots b_n$ are permutations, $a_i = b_i$ for $i < k$ and $a_k < b_k$ then $a_1 \dots a_k \dots a_n$ comes first.

Algorithm 4.1: INSERT(A, i, j)

```
1  x ← A[i]
2  if i < j then  for k ← i + 1 to j do A[k − 1] ← A[k]
3  else if j < i then  for k ← i − 1 to j do A[k + 1] ← A[k]
4  A[j] ← x
```

Algorithm 4.2: LIST-PERMS-LEX(A, p)

List, in lexicographic order, all permutations of A that do not change $A[1\,..\,p-1]$. Leave A in the original order.

```
1  if p = n then print A else
2      LIST-PERMS-LEX(A, p + 1)
3      for i = p + 1 to n do
4          INSERT(A, i, p); LIST-PERMS-LEX(A, p + 1); INSERT(A, p, i)
```

The solution below uses only swaps, but lists in some non-lexicographic order.

Algorithm 4.3: SWAP(A, i, j)

```
1  x ← A[i]; A[i] ← A[j]; A[j] ← x
```

Algorithm 4.4: LIST-PERMS-SWAP(A, p)
List, all permutations of A that do not change $A[1 \mathinner{..} p-1]$.
Leave A in original order.

```
1  if p = n then print A else
2      LIST-PERMS-SWAP(A, p + 1)
3      for i = p + 1 to n do
4          SWAP(A, i, p); LIST-PERMS-SWAP(A, p + 1); SWAP(A, p, i)
```

- If we sort an array, then it becomes very easy to find the largest element, the second largest, and so on. Sometimes we need a data structure that just does this.
- For this, we define the abstract data type, or object class called priority queue.

The priority queue object maintains a set S of elements, each associated with a value (say, an integer) called a key.

Operations INSERT(S, x), MAXIMUM(S), EXTRACT-MAX(S).

Later INCREASE-KEY(x, k)

Applications Job scheduling in a multiuser operating system.
Event-driven simulation.
Later in this course: helping various algorithms to use data.

Unordered array All operations but INSERT take $\Omega(n)$ steps.

Sorted array All but MAXIMUM take $\Omega(n)$ steps.

Heap (has nothing to do with the heaps of memory management)
  Maximum takes 1 step, others $\log n$.

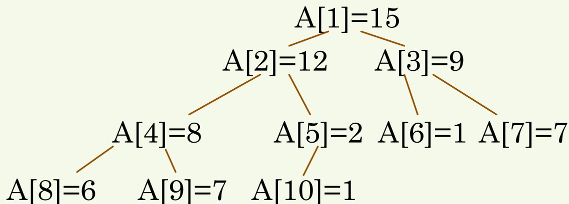
Fibonacci heap (later) All operations cost $O(\log n)$, and have
  amortized cost $O(1)$. (See definition of amortized cost later).

1. (Binary) tree with keys increasing towards root:

$$A[\textsc{Parent}(i)] \geqslant A[i].$$

2. A special array implementation of this tree:

$$\textsc{Parent}(i) = \lfloor i/2 \rfloor,$$
$$\textsc{Left}(i) = 2i, \quad \textsc{Right}(i) = 2i + 1.$$



HEAPIFY(A, i) will be called by several of our methods. Let

$$\mathrm{argmax}_A(i, j, k) = i \text{ if } A[i] \geqslant A[j], A[k], \text{ else } j \text{ if } A[j] \geqslant A[k], \text{ else } k.$$

Algorithm 5.1: HEAPIFY(A, i)

Assuming the trees under LEFT(i), RIGHT(i) are heaps, make the tree under i also a heap.

Implementation: float down $A[i]$.

```
1  l ← Left(i); if l > A.heap-size then l ← i
2  r ← Right(i); if r > A.heap-size then r ← i
3  largest ← argmax_A(i, l, r)
4  if largest ≠ i then
5      Swap(A, i, largest)
6      Heapify(A, largest)
```

Implementing EXTRACT-MAX(A):

- Put the last element into the root: $A[1] \leftarrow A[A.heap\text{-}size]$.
- HEAPIFY($A, 1$).

Implementing INSERT(A, k):

- $A.heap\text{-}size{+}{+}$; $A[A.heap\text{-}size] \leftarrow k$
- Float up $A[A.heap\text{-}size]$.

The above implementation would be sufficient for sorting since we can build a heap by repeated insertion, and then extract the maximum repeatedly. But the following is better.

Algorithm 5.2: BUILD-HEAP(A)
Build a heap, assuming the elements are in the array A.

1 **for** $i \leftarrow A.\mathit{heap\text{-}size}$ **downto** 1 **do** HEAPIFY(A, i)

The naive estimate for the cost is $O(n \log n)$, if $n = A.\mathit{heap\text{-}size}$. But more analysis shows:

Theorem

The cost of algorithm BUILD-HEAP(A) *on a heap of size n is $O(n)$.*

The largest depth is $D = \lfloor \log n \rfloor$, starting with the root at depth 0. For the 2^d elements i is at depth d, the cost of Heapify(A, i) is $h - d$. Total cost, converting the depth d to the height $h = D - d$:

$$\sum_{d=D}^{0} 2^d (D - d) = \sum_{h=0}^{D} h 2^{D-h} = 2^D \sum_{h=0}^{D} h 2^{-h}$$
$$\leqslant n \sum_{h=0}^{\infty} h 2^{-h}.$$

Though there is an exact formula for the last sum, a rough estimate is more instructive for us. Since $h \ll 2^{h/2}$, write, with some constant c:

$$\sum_{h \geqslant 0} h 2^{-h} \leqslant c \sum_{h \geqslant 0} 2^{-h/2} < \infty,$$

as a geometric series with quotient < 1.

Abstract data types (objects)

- Some pieces of data, together with some operations that can be performed on them.
- Externally, only the operations are given, and only these should be accessible to other parts of the program.
- The internal representation is private. It consists typically of a number of records, where each record has key, satellite data. We will ignore the satellite data unless they are needed for the data manipulation itself.

This operation (giving more priority to some element) is important in some applications where the priority queue helps some algorithm (say Dijkstra's shortest path algorithm).
Let the changed item float up:

Algorithm 5.3: INCREASE-KEY(A, i, key)
Increase $A[i]$ to $key > A[i]$.

1 $A[i] \leftarrow key$
2 **while** $i > 1$ **and** $A[\text{PARENT}(i)] < A[i]$ **do**
3   SWAP$(A, i, \text{PARENT}(i))$
4     $i \leftarrow \text{PARENT}(i)$

The INSERT(A, key) operation can also be seen as a special case of INCREASE-KEY(A, i, key).

Another way to apply "divide and conquer" to sorting. Both here and in mergesort, we sort the two parts separately and combine the results.

- With Mergesort, we sort first two predetermined halves, and combine the results later.
- With Quicksort, we determine what elements fall into two "halves" (approximately) of the sorted result, using Partition. Calling this recursively on both "halves" completes the sort.

Algorithm 6.1: PARTITION(A, p, r)

Partition $A[p \mathinner{..} r]$ into elements $\leqslant x = A[r]$ placed in $p, \dots, i$, into $i + 1$ containing x, and the rest containing $> x$.
Return $i + 1$.

```
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1 do
4      if A[j] ⩽ x then
5          i++; Swap(A, i, j)
6  Swap(A, i + 1, r); return i + 1
```

Example $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

Analysis Invariant: at beginning of loop,

$$\underbrace{p, \dots, i}_{\leqslant x}, \underbrace{i + 1, \dots, j - 1}_{> x}$$

Algorithm 6.2: QUICKSORT(A, p, r)

```
1  if p < r then
2      q ← PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1); QUICKSORT(A, q + 1, r)
```

Let $T(n)$ be the worst-case number of comparisons. If $q = \textsc{Partition}(A, 1, n)$ then

$$T(n) = T(q-1) + T(n-q) + n - 1.$$

This is not immediately helpful, since we do not know q. But $q = n$ is possible (say, the array was already sorted), giving $T(n) = T(n-1) + n - 1$. Resolving the recursion shows that all comparisons are performed!

Balanced partitioning

- Hypothetical partitioning algorithm that always gives $0.1n < q < 0.9n$.
- Analysis with recursion tree gives $n \log n / \log(10/9)$.
- Approximate balanced partitioning by partitioning around a random array element.
- Random number generator $\textsc{Random}(a, b)$ returns a random, equally distributed integer between a and b ($b - a + 1$ possibilities).
- In practice, we only have a pseudo-random number generator. This is a separate topic, for the moment let us hope it gives numbers that behave just like “true” random ones.

Algorithm 6.3: Randomized-Partition(A, p, r)

```
1  Swap(A, r, Random(p, r))
2  return Partition(A, p, r)
```

Algorithm 6.4: Randomized-Quicksort(A, p, r)

```
1  q ← Randomized-Partition(A, p, r)
2  Randomized-Quicksort(A, p, q − 1)
3  Randomized-Quicksort(A, q, r)
```

In order to analyze randomized quicksort, we learn some probability concepts. I assume that you learned about random variables already in an introductory course (Probability in Computing, or its equivalent).

For a random variable X with possible values $a_1, \dots, a_n$, its expected value $\mathbf{E}X$ is defined as

$$a_1 \mathbf{P}\left[X = a_1 \right] + \cdots + a_n \mathbf{P}\left[X = a_n \right].$$

Example: if Z is a random variable whose values are the possible outcomes of a toss of a 6-sided die, then

$$\mathbf{E}Z = (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5.$$

Example: If Y is the random variable that is 1 if $Z \geqslant 5$, and 0 otherwise, then

$$\mathbf{E}Y = 1 \cdot \mathbf{P}[Z \geqslant 5] + 0 \cdot \mathbf{P}[Z < 5] = \mathbf{P}[Z \geqslant 5].$$

Theorem

For random variables X, Y (on the same sample space):

$$\mathbf{E}(X+Y) = \mathbf{E}X + \mathbf{E}Y.$$

Example

For the number X of spots on top after a toss of a die, let A be the event $2|X$, and B the event $X > 1$. Dad gives me a dime if A occurs and Mom gives one if B occurs. What is my expected win?
Let I_A be the random variable that is 1 if A occurs and 0 otherwise.

$$\mathbf{E}(I_A + I_B) = \mathbf{E}I_A + \mathbf{E}I_B = \mathbf{P}(A) + \mathbf{P}(B) = 1/2 + 5/6 \text{ dimes}.$$

Let the sorted order be $z_1 < z_2 < \cdots < z_n$. If $i < j$ then let

$$Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}.$$

Let $C_{ij} = 1$ if z_i and z_j will be compared sometime during the sort, and 0 otherwise.
The only comparisons happen during a partition, with the pivot element. Let π_{ij} be the first (random) pivot element entering Z_{ij}. A little thinking shows:

Lemma

We have $C_{ij} = 1$ if and only if $\pi_{ij} \in \{z_i, z_j\}$. Also, for every $x \in Z_{ij}$, we have

$$\mathbf{P}\left[\, \pi_{ij} = x \,\right] = \frac{1}{j - i + 1}.$$

It follows that $\mathbf{P}\left[\, C_{ij} = 1 \,\right] = \mathbf{E} C_{ij} = \frac{2}{j-i+1}$. The expected number of comparisons is, with $k = j - i + 1$:

$$\sum_{1 \leqslant i < j \leqslant n} \mathbf{E} C_{ij} = \sum_{1 \leqslant i < j \leqslant n} \frac{2}{j - i + 1} = 2 \sum_{k=2}^{n} \sum_{i=1}^{n-k+1} \frac{1}{k}$$

$$= 2 \sum_{k=2}^{n} \frac{n-k+1}{k} = 2 \left(\frac{n-1}{2} + \frac{n-2}{3} + \cdots + \frac{1}{n} \right)$$

$$= 2(n+1) \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \right) - 4n.$$

From analysis we know $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = \ln n + O(1)$. Hence the average complexity is $O(n \log n)$.

Warning: We want to know also the probability of the cost exceeding, say, $10 n \log n$. Analysis of the variance of the random cost gives this, too.

Median and order statistics

- The median of a list is the middle element (or the first one, if there are two) in the sorted order. We will try to find it faster than by full sorting.
- More generally, we may want to find the ith element in the sorted order. (This generalizes minimum, maximum and median.) This is called selection.
- We will see that all selection tasks can be performed in linear time.

Algorithm 7.1: RANDOMIZED-SELECT(A, p, r, i)
Return the ith element in the sorted order of $A[p \,..\, r]$.

```
1  if p = r then return p
2  q ← RANDOMIZED-PARTITION(A, p, r)
3  k ← q − p + 1
4  if i = k then return A[q]
5  else if i < k then return RANDOMIZED-SELECT(A, p, q − 1, i)
6  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- Worst-case running time is $O(n^2)$, even for finding the minimum!
- We will find an upper bound $T(n)$ on the expected running time of RANDOMIZED-SELECT(A, p, r, i), independently of i, when $r - p + 1 = n$.
- Assume that cn is the cost of partition and the decision. Denoting by τ_0 the running time of the whole process, and τ_1 the running time of the recursive call, we have

$$\tau_0 \leqslant an + \tau_1,$$
$$\mathbf{E}\tau_0 \leqslant an + \mathbf{E}\tau_1,$$

where an is the cost of partition.

Let $Q = \text{RANDOMIZED-PARTITION}(A, p, r)$. Let us bound $\mathbf{E}\tau_1$ under the condition $Q - p + 1 = k$:

- If $i = k$, then $\mathbf{E}\tau_1 = 0$.
- Else if $i < k$ then $\mathbf{E}\tau_1 \leqslant T(k-1)$,
- Else $\mathbf{E}\tau_1 \leqslant T(n-k)$.

$$\mathbf{E}(\tau_1 | Q - p + 1 = k) \leqslant T(\max(k-1, n-k)).$$

Since $\mathbf{P}\left[\, Q - p + 1 = k \,\right] = \frac{1}{n}$, we can now sum, using Bayes's Theorem:

$$\mathbf{E}\tau_1 \leqslant \sum_{k=1}^{n} \frac{1}{n} T(\max(k-1, n-k)) \leqslant \frac{2}{n} \sum_{k=n/2}^{n-1} T(k),$$

where we assumed n even, for simplicity.

We got the recursion $T(n) \leqslant an + \frac{2}{n}\sum_{k=n/2}^{n-1} T(k)$. We will prove by induction $T(n) \leqslant cn$ for some $c > 0$. Assume it is true for $< n$, we prove it for n (choosing c appropriately).

$$\frac{2}{n}\sum_{k=n/2}^{n-1} ck = \frac{2c}{n} \cdot \frac{n}{2} \cdot \frac{n/2 + n - 1}{2} \leqslant \frac{3}{4}cn,$$

where we used the sum formula of an arithmetic series. Now $an + \frac{3}{4}cn = cn$ if we make $a + 0.75c = c$, that is $c = 4a$. For the induction we also need the initial condition $T(1) \leqslant c \cdot 1$, but this is true with $c = 4a$.

Size of input Just as before, we count the number of input bits. If the input is a positive integer x of 1000 bits, the input size is $1000 = \lceil \log x \rceil$, and not x. So, counting from x down to 1 is an exponential algorithm, as a function of the input size.

Cost of operations Most of the time, for "cost", we still mean the execution time of the computer, which has a fixed word size. So, multiplying two very large numbers x, y is not one operation, since the representation of the numbers x, y must be spread over many words. We say that we measure the number of bit operations.

Sometimes, though, we will work with a model in which we just measure the number of arithmetic operations. This is similar to the sorting model in which we only counted comparisons.

- Divisibility and divisors. Notation $x|y$.
- Prime and composite numbers.

Theorem

If a, b are integers, $b > 0$, then there are unique numbers q, r with

$$a = qb + r, \quad 0 \leqslant r < b.$$

Of course, $q = \lfloor a/b \rfloor$. We write $r = a \bmod q$.
Cost of computing the remainder and quotient: $O(\mathrm{len}(q)\mathrm{len}(b))$.

Indeed, the long division algorithm has $\leqslant \mathrm{len}(q)$ iterations, with numbers of length $\leqslant \mathrm{len}(b)$.

If $u \bmod b = v \bmod b$ then we write

$$u \equiv v \pmod{b}$$

and say "u is congruent to v modulo b". For example, numbers congruent to 1 modulo 2 are the odd numbers.

It is not easy to find divisors of a number: most cryptographic algorithms are based on this difficulty of factorization.
But, as we will see, common divisors of two (or more) numbers are easy to find.

Theorem

For numbers a, b there are integers x, y such that

$$d = xa + yb > 0$$

is a common divisor of a and b.

It is easy to see that every common divisor of a, b divides d, so it is the greatest common divisor, denoted by $\gcd(a, b)$.
Relatively prime integers a, b: those without common divisor, that is $\gcd(a, b) = 1$.

Lemma (Fundamental)

If p is prime, then $p|ab$ if and only if $p|a$ or $p|b$.

Proof. If p does not divide a, then $\gcd(a,p) = 1 = xa + yp$, so $b = xab + ypb$. Since $p|ab$, this shows $p|b$. □

This implies

Theorem (Fundamental theorem of arithmetic)

Every positive integer n has a unique prime decomposition $n = p_1^{e_1} \cdots p_k^{e_k}$.

The existence of some decomposition can be seen easily: just keep dividing as long as you can, and collect the indivisible parts (the primes). The lemma is needed to prove uniqueness.

Based on the observation that if $a = qb + r$ then the common divisors of a, b are the same as the common divisors of b, r.

Algorithm 8.1: EUCLID(a, b)
Computes $\gcd(a, b)$

1. **if** $b = 0$ **then return** a
2. **else return** EUCLID$(b, a \bmod b)$

Example: $\gcd(30, 21)$.

$$
\begin{aligned}
30 &= 1 \cdot 21 + 9 \\
21 &= 2 \cdot 9 + 3 \\
9 &= 3 \cdot 3 \\
\gcd &= 3
\end{aligned}
$$

Assume $b < a$:

$$\begin{aligned} a = bq + r \geqslant b + r > 2r, \\ a/2 > r, \\ ab/2 > br. \end{aligned}$$

In each iteration, the product decreases by a factor of 2.
$\lceil \log(ab) \rceil$ iterations, using division.

Proof of the theorem that some common divisor of a, b (namely the greatest one) can be written as $xa + yb$.
It is true if $b = 0$. Assume it is true for numbers for which the Euclidean algorithm takes $< n$ steps, and prove by induction.

$$a = bq + r.$$

By induction,

$$d = x'b + y'r = x'b + y'a - y'qb = y'a + (x' - qy')b.$$

Algorithm 8.2: EXTENDED-EUCLID(a, b)

Returns (d, x, y) where $d = \gcd(a, b) = xa + yb$.

```
1  if b = 0 then return (a, 1, 0)
2  else
3      (d′, x′, y′) ← EXTENDED-EUCLID(b, a mod b)
4      return (d′, y′, x′ − ⌊a/b⌋y′)
```

The following version is better for analysis and hand calculation:

Algorithm 8.3: EXTENDED-EUCLID(a, b)
nonrecursive version.

1 $r \leftarrow a$; $r' \leftarrow b$
2 $x \leftarrow 1$; $y \leftarrow 0$
3 $x' \leftarrow 0$; $y' \leftarrow 1$
4 **while** $r' \neq 0$ **do**
5 $\quad q \leftarrow \lfloor r/r' \rfloor$
6 $\quad (r, x, y, r', x', y') \leftarrow (r', x', y', r - qr', x - qx', y - qy')$
7 **return** (r, x, y)

Proof of validity: show by induction that always $r = xa + yb$.

Solving the equation

$$ax + by = c$$

in integers. Let $d = \gcd(a, b)$. There is a solution only if $d|c$. By the Fundamental Lemma, there are x', y' with $ax' + by' = d$. Hence we get a solution

$$x = x'(c/d), \quad y = y'(c/d).$$

If $\gcd(a, m) = 1$ then there is an a' with

$$aa' \bmod m = 1.$$

Indeed, we just need to solve the equation $ax + my = 1$. The number a' is called the multiplicative inverse of a modulo m.
Application: If p is prime then every $a < p$ has a multiplicative inverse.

Congruences behave somewhat like equations:

Theorem

If $a_1 \equiv b_1 \pmod{m}$ *and* $a_2 \equiv b_2 \pmod{m}$ *then*

$$a_1 \pm a_2 \equiv b_1 \pm b_2 \pmod{m},$$
$$a_1 a_2 \equiv b_1 b_2 \pmod{m}.$$

Verify this in the lab. Division is also possible in the cases when there is a multiplicative inverse:

Theorem

If $\gcd(a, m) = 1$ *then there is an* a' *with* $a'a \equiv 1 \pmod{m}$.

In modular arithmetic, we can exponentiate in a polynomial time:

Theorem

Given numbers a, n, m, there is an algorithm that computes $a^n \bmod m$ in time polynomial in the size of input (which is $\log a + \log m + \log m$).

This is not true for ordinary exponentiation: 2^n has $\Omega(n)$ digits, so it cannot be computed in time $O(\log n)$.
The trick that helps in the modular case is repeated squaring:
Compute $a^2 \bmod m$, $a^4 \bmod m$, $a^8 \bmod m$, ..., using

$$a^{2k} \bmod m = (a^k \bmod m)^2 \bmod m.$$

Then multiply the needed powers. For example if $n = 10 = 2^3 + 2^{2^1}$, then $a^{10} = a^{2^3} a^{2^1}$.

These are abstract data types, just like heaps. Various subtypes exist, depending on which operations we want to support:

- SEARCH(S, k)
- INSERT(S, k)
- DELETE(S, k)

A data type supporting these operations is called a dictionary.

Other interesting operations:

- MINIMUM(S), MAXIMUM(S)
- SUCCESSOR(S, x) (given a pointer to x), PREDECESSOR(S, x)
- UNION(S_1, S_2)

Elementary data structures

You have studied these in the Data Structures course.

Stack operations:

- STACK-EMPTY(S)
- PUSH(S, x)
- POP(S)

Implement by array or linked list.

Queue operations:

- ENQUEUE(Q, x)
- DEQUEUE(S, x)

Implement by circular array or linked list (with link to end).

Singly linked, doubly linked, circular.

- LIST-SEARCH(L, k)
- LIST-INSERT(L, x) (x is a record)
- LIST-DELETE(L, x)

In the implementation, sentinel element $nil[L]$ between the head and the tail to simplify boundary tests in a code.

Dynamic storage allocation

This underlies to a lot of data structure implementation.
You must either explicitly implement freeing unused objects of various sizes and returning the memory space used by them to free lists of objects of various sizes, or use some kind of garbage collection algorithm.
The present course just assumes that you have seen (or will see) some implementations in a Data Structures course (or elsewhere).

Binary tree data type, at least the following functions:

$$\text{PARENT}[x], \text{LEFT}[x], \text{RIGHT}[x].$$

We can represent all three functions explicitly by pointers.

Rooted trees with unbounded branching: One possible representation via a binary tree is the left-child, right-sibling representation.

Other representations : Heaps, only parent pointers, and so on.

Problem A very large universe U of possible items, each with a different key. The number n of actual items that may eventually occur is much smaller.

Solution idea Hash function $h(k)$, hash table $T[0..m-1]$. Key k hashes to hash value (bucket) $h(k)$.

New problem Collisions.

Resolution Chaining, open hashing, and so on.

Uniform hashing assumption Assumes that

- Items arrive "randomly".
- Search takes $\Theta(1 + n/m)$, on average, since the average list length is n/m.

What do we need? The hash function should spread the (hopefully randomly incoming) elements of the universe as uniformly as possible over the table, to minimize the chance of collisions.

Keys into natural numbers It is easier to work with numbers than with words, so we translate words into numbers.
For example, as radix 128 integers, possibly adding up these integers for different segments of the word.

To guarantee the uniform hashing assumption, instead of assuming that items arrive “randomly”, we we choose a random hash function, $h(\cdot, r)$, where r is a parameter chosen randomly from some set H.

Definition

The family $h(\cdot, \cdot)$ is universal if for all $x \neq y \in U$ we have

$$\mathbf{P}\left[\, h(x, r) = h(y, r) \,\right] \leqslant \frac{1}{m}.$$

If the values $h(x, r)$ and $h(y, r)$ are pairwise independent, then the probability is exactly $\frac{1}{m}$ (the converse is not always true). Thus, from the point of view of collisions, universality is at least as good as pairwise independence.

An example universal hash function

We assume that our table size m is a prime number. (Not too restrictive: it is known that for every m there is a prime between m and $2m$.) Let $d > 0$ be an integer dimension. We break up our key x into a sequence

$$x = \langle x_1, x_2, \dots, x_d \rangle, \quad x_i < m.$$

Fix the random coefficients $r_i < m$, $i = 1, \dots, d + 1$, therefore $|H| = m^{d+1}$.

$$h(x, r) = r_1 x_1 + \dots + r_d x_d + r_{d+1} \bmod m.$$

Let us show that all values $h(x, r)$ are pairwise independent, therefore our random hash function is universal. We will prove that $h(x, r)$ takes each of its m possible values with the same probability, independently of how we fix $h(y, r)$.

- Every value $A = h(x, r)$ appears exactly m^d times. Indeed, no matter how we choose $r_1, \dots, r_d$, one can choose r_{d+1} uniquely to make $h(x, r) = A$.
- Now take some $y \neq x$, for example $x_1 \neq y_1$. Fix $h(x, r) = A$, $h(y, r) = B$. Chooose $r_2, \dots, r_d$ arbitrarily, and consider

$$\delta = x_1 - y_1, \quad W = r_2(x_2 - y_2) + \cdots + r_d(x_d - y_d),$$
$$A - B \equiv r_1\delta + W \pmod{m}.$$

The last equation has exactly one solution $r_1 = \delta^{-1}(A - B - W)$, where δ^{-1} is the multiplicative inverse of δ modulo the prime m. Given $r_1, \dots, r_d$ there is a unique r_{d+1} with $h(y, r) = B$. So every pair of values A, B appears exactly m^{d-1} times. This proves the independence.

Optional: perfect hashing

If we know the set S of n keys in advance, it is also possible to hash without any collisions, into a table of size $6n$, using just 2 probes. (Method of Fredman, Komlós and Szemerédi).

Observations:

1. If $m > n^2$ then there is a vector $c = (c_1, \ldots, c_{d+1})$ such that the hash function $h(\cdot, c)$ has no collisions. Indeed, for each pair $x \neq y$, the probability of a collision is $1/m$. There are $n(n-1)/2$ such pairs, so the probability of having a collision on even one pair is at most $n(n-1)/(2m) < 1/2$. Therefore there is a vector c such that $h(\cdot, c)$ has no collisions for S.
2. Now assume that $m > 3n$. For each vector r and position z, let $C_z(r)$ be the number of keys colliding on position z. We will show later that there is a vector b with

$$\sum_z C_z(b)^2 < 3n.$$

Using the above observations, here is a 2-step hashing scheme.

- Using vector b, create a table U_z of size $C_z(b)^2$ at each position z of our table T of size $3n$ where there are collisions. For each table $U_z(b)$, use a hash function $h(\cdot, c(z))$ to hash x to a unique position in U_z. This is possible due to argument 1 above.
- Now, first use $h(x, b)$ to hash key x to a position z in table T. If there were collisions then T at position z will hold the key $c(z)$ of the second hash function $h(x, c(z))$ that hashes x to a unique position in table U_z.
- Total size: $3n$ for table T, and $< 3n$ for the union of all nonempty tables U_z.

It remains to show that for some value b of r, we have $\sum_z C_z(b)^2 < 3n$.
Fix position z. For item i, let $X_i = 1$ if $h(i, r) = z$ and 0 otherwise. Then $C_z = \sum_{i=1}^{n} X_i$. The variables X_i are pairwise independent, with $\mathbf{P}\left[X_i = 1 \right] = 1/m$. Because of the independence, $\mathbf{E} X_i X_j = \mathbf{E} X_i \mathbf{E} X_j$ for $i \neq j$, allowing to write

$$
\begin{aligned}
C_z^2 &= \sum_i X_i^2 + \sum_{i \neq j} X_i X_j, \\
\mathbf{E} C_z^2 &= \sum_i \mathbf{E} X_i^2 + \sum_{i \neq j} \mathbf{E} X_i \mathbf{E} X_j = \frac{n}{m} + \frac{n(n-1)}{m^2}, \\
\mathbf{E} \sum_z C_z^2 &= n\left(1 + \frac{n-1}{m}\right) < n(1 + 1/3).
\end{aligned}
$$

Since this is the average over all vectors r there is obviously some vector b with $\sum_z C_z(b)^2 < 3n$.

Open addressing

- Hashing must sometimes be very fast, for example in caches. In this case, even the pointer operations of the chaining collision resolution are considered slow.
- Open addressing puts the collided elements into the same table, at a new position computed very fast. Looking for the new position is called probing. We need a whole probe sequence $h(k,0), h(k,1,), \dots$.
- In open addressing schemes, deletion is problematic, though there are several marking methods.
- It is not easy to analyze open addressing, since the uniform hashing assumption is too strong.

Linear probing $h(k,i) = (h'(k) + i) \bmod m$. There is a problem of primary clustering: long runs of collided elements build up.

Quadratic probing tries to correct this: $h(k,i) = h'(k) + c_1 i + c_2 i^2$. There is, however, a problem of secondary clustering, since probing sequences are independent of k.

Double hashing solves these issues:

$$h(k,i) = h_1(k) + i h_2(k),$$

where $h_2(k)$ is relatively prime to m.
Example: $h_1(k) = k \bmod m$ with m a prime,
$h_2(k) = 1 + k \bmod (m-1)$.

- Ordered left-to-right, items placed into the tree nodes (not only in leaves).
- Inorder tree walk.
- Search, min, max, successor, predecessor, insert, delete. All in $O(h)$ steps.

- Iterative or recursive TREE-SEARCH(x, k), where x is (a pointer to) the root.
- Minimum and maximum are easy.

TREE-SUCCESSOR(x)

- if right subtree of x is nonempty then its minimum.
- else go up: return the first node to whom you had to step right-up (if it exists).

Insertion always into a leaf.

Deletion TREE-DELETE(T, z) is more complex. We use the procedure

$$\text{TRANSPLANT}(T, u, v),$$

which replaces the subtree starting at u in T with the tree rooted in v. Warning:

- Before, the tree of v need not be part of T.
- After, the pointer u still points to the record u (along with any subtree hanging from it).

Algorithm 10.1: TREE-DELETE(T, z)

Delete node z from tree T.

```
1  if z.left = nil then TRANSPLANT(T, z, z.right)
2  else if z.right = nil then TRANSPLANT(T, z, z.left)
3  else
4      y ← TREE-MINIMUM(z.right)      // y has no left child.
5      if y.parent ≠ z then
           // Splice it out from between parent and right child.
6          TRANSPLANT(T, y, y.right)  // Does not lose original y.
7          y.right ← z.right; y.right.parent ← y
8      TRANSPLANT(T, z, y)
9      y.left ← z.left; y.left.parent ← y
```

Algorithm 10.2: BS-TREE-SPLIT(T, k)

Split T around key k into two trees.

```
1   (L, R) ← new trees
2   if T is empty then L ← empty; R ← empty
3   else if k < T.root.key then
4       R ← T
5       (L, R′) ← BS-TREE-SPLIT(T.left)
6       R.left ← R′
7   else
8       L ← T
9       (L′, R) ← BS-TREE-SPLIT(T.right)
10      L.right ← L′
11  return (L, R)
```

Randomized binary search trees

- It can be shown (see book) that if n keys are inserted into a binary search tree in random order, then the expected height of the tree is only $O(\log n)$.
- You cannot expect that keys will be inserted in random order. But it is possible (Martínez-Roura) to randomize the insertion and deletion operations in a way that will still guarantee the same result.
- Here is the idea of randomized insertion into a binary tree of size n.
  With probability $\frac{1}{n+1}$, insert the key as root, and split the tree around it.
  With probability $1 - \frac{1}{n+1}$, randomized-insert recursively into the left or right subtree, as needed.
- There are corresponding deletion and join operations.

Algorithm 10.3: RAND-BS-TREE-INSERT(T, k)
Randomized insert into a binary search tree.

```
1  n ← T.size
2  r ← RANDOM(0, n)
3  if r = n then                                  // Insert at the root.
4      (L, R) ← BS-TREE-SPLIT(T, k)
5      T ← new tree; T.key ← k
6      (T.left, T.right) ← (L, R);
7  else if k < T.root.key then RAND-BS-TREE-INSERT(T.left, k)
8  else RAND-BS-TREE-INSERT(T.right, k)
```

Balanced trees

- The cost of most common operations on search trees (search, insert, delete) is $O(h)$, where h is the height.
- We can claim $h = O(\log n)$ for the size n, if the tree is balanced. But keeping a tree completely balanced (say a binary tree as complete as possible) is as costly as maintaining a sorted array.
- Idea: keep the tree only nearly balanced! The idea has several implementations: AVL trees, B-trees, red-black trees. We will only see B-trees.

Search trees, with the following properties, with a parameter $t \geqslant 2$.

- The records are kept in the tree nodes, sorted, “between” the edges going to the subtree.
- Every path from root to leaf has the same length.
- Every internal node has at least $t-1$ and at most $2t-1$ records (t to $2t$ children).

Total number of nodes $n \geqslant 1 + t + \cdots + t^h = \frac{t^{h+1}-1}{t-1}$, showing $h = O(\log_t n)$.

Insertion Insert into the appropriate node. Split, if necessary. If this requires split in the parent, too, split the parent.
There is an implementation with only one, downward, pass. It makes sure that each subtree in which we insert, the root is not full: has fewer than $2t - 1$ keys. To achieve it, we split proactively.

Deletion More complex, but also doable with one, downward, pass. Make sure that when we delete from a subtree, the root has more than $t - 1$ (the minimum) keys. To achieve this, merge proactively.

Algorithm 10.4: B-TREE-SPLIT-CHILD(x, i)

Split the ith child c_i of non-full node x: before split, the child has $2t - 1$ keys. No surprises.

1 $z \leftarrow$ new node; $y \leftarrow x.c_i$
2 $z.n \leftarrow t - 1$; $z.leaf \leftarrow y.leaf$ // If y is a leaf, so be z.
3 **for** $j = 1$ **to** $t - 1$ **do** $z.key_j \leftarrow y.key_{j+t}$
4 **if not** $y.leaf$ **then** **for** $j = 1$ **to** t **do** $z.c_j \leftarrow y.c_{j+t}$
5 $y.n \leftarrow t - 1$ // Now z contains the second half of y.
6 **for** $j = x.n + 1$ **downto** $i + 1$ **do** $x.c_{j+1} \leftarrow x.c_j$
7 $x.c_{i+1} \leftarrow z$ // Now z is a child of x.
8 **for** $j = x.n$ **downto** i **do** $x.key_{j+1} \leftarrow x.key_j$
9 $x.key_i \leftarrow y.key_t$ // The middle key of y is inserted into x.
10 $x.n$++

Algorithm 10.5: B-TREE-INSERT-NONFULL(x, k)

Insert into a subtree at a non-full node.

```
1   i ← x.n
2   if x.leaf then                         // Just insert the key into the leaf.
3       while i ⩾ 1 and k < x.key_i do  x.key_{i+1} ← x.key_i; i−−
4       x.key_{i+1} ← k; x.n++
5   else
6       while i ⩾ 1 and k < x.key_i do i−−
7       i++                   // Found the child to insert into.
8       if x.c_i.n = 2t − 1 then
9           B-TREE-SPLIT-CHILD(x, i)
10          if k > x.key_i then i++
11      B-TREE-INSERT-NONFULL(x.c_i, k)
```

Algorithm 10.6: B-TREE-INSERT(T, k)

```
1  r ← T.root
2  if r.n < 2t − 1 then B-TREE-INSERT-NONFULL(r, k)
3  else                    // Add new root above r before splitting it.
4      s ← new node; s.leaf ← false; s.n ← 0
5      s.c_1 ← r; T.root ← s
6      B-TREE-SPLIT-CHILD(s, 1)
7      B-TREE-INSERT-NONFULL(s, k)
```

Deletion

The deletion algorithm is more complex. See the book for an outline, and try to implement it in pseudocode as an exercise.

- Cache in each node the size of the subtree.
- Retrieving an element with a given rank (selection).
- Determining the rank of a given element.
- Maintaining subtree sizes.

When there is a recursive algorithm based on subproblems but the total number of subproblems is not too great, it is possible to cache (memoize) all solutions in a table.

- Fibonacci numbers.
- Binomial coefficients.

(In both cases, the algorithm is by far not as good as computing the known formulas.)

- The diff program in Unix: what does it mean to say that we find the places where two files differ (including insertions and deletions)? Or, what does it mean to keep the "common" parts?
- Let it mean the longest subsequence present in both:

$$
\begin{array}{llllllllll}
X = & a & b & & c & & b & & d\ a\ b \\
Y = & & b & d & c & a & b & a & b\ a \\
 & & b & & c & & b & & \quad a
\end{array}
$$

- Running through all subsequences would take exponential time. There is a faster solution, recognizing that we only want to find some longest subsequence.
- Let $c[i, j]$ be the length of the longest common subsequence of the prefix of $X[1..i]$ and $Y[1..j]$. Recursion:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise.} \end{cases}$$

  We are not computing a function $C(X, Y, i, j)$ by naive recursion, but collect the values $c[i, j]$ as they are computed, in array c: $C(X, Y, i, j, c)$ checks whether $c[i, j]$ is defined. If yes, it just returns $c[i, j]$; else it uses the above recursion, and assigns $c[i, j]$ before returning the value.

Non-recursive implementation

- Compute the table $c[i,j]$ “bottom-up”.
- Also, store the value $b[i,j] = \{\leftarrow\}$, $\{\uparrow\}$ or $\{\nwarrow\}$ depending on whether the optimum is $c[i,j-1]$, $c[i-1,j]$ or $c[i-1,j-1]+1$. (We also use the value $\{\leftarrow,\uparrow\}$ in case $c[i,j] = c[i-1,j] = c[i,j-1]$, though this is not important.)
- Find the longest common subsequence walking backwards on the arrows.

	j	1	2	3	4	5	6
i		b	d	c	a	b	a
1	a	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	b	↖ 1	← 1	← 1	↑ ← 1	↖ 2	← 2
3	c	↑ 1	↑ ← 1	↖ 2	← 2	↑ ← 2	↑ ← 2
4	b	↖ 1	↑ ← 1	↑ 2	↑ ← 2	↖ 3	← 3
5	d	↑ 1	↖ 2	↑ ← 2	↑ ← 2	↑ 3	↑ ← 3
6	a	↑ 1	↑ 2	↑ ← 2	↖ 3	↑ ← 3	↖ 4
7	b	↖ 1	↑ 2	↑ ← 2	↑ 3	↖ 4	↑ ← 4

Algorithm 11.1: LCS-LENGTH(X, Y)

```
1   m ← X.length; n ← Y.length
2   b[1..m, 1..n], c[0..m, 0..n] ← new tables
3   for i = 1 to m do c[i, 0] ← 0
4   for j = 1 to n do c[0, j] ← 0
5   for i = 1 to m do
6       for j = 1 to n do
7           if x_i = y_j then
8               c[i, j] ← c[i − 1, j − 1] + 1
9               b[i, j] ← {↖}
10          else
11              c[i, j] ← max(c[i, j − 1], c[i − 1, j]); b[i, j] ← ∅
12              if c[i − 1, j] = c[i, j] then b[i, j] ← {↑}
13              if c[i, j − 1] = c[i, j] then b[i, j] ← b[i, j] ∪ {←}
14  return c, b
```

Algorithm 11.2: LCS-PRINT(b, X, i, j)

Print the longest common subsequence of $X[1..i]$ and $Y[1..j]$ using the table b computed above.

```
1  if i = 0 or j = 0 then return
2  if ↖ ∈ b[i, j] then LCS-PRINT(b, X, i − 1, j − 1); print x_i
3  else if ↑ ∈ b[i, j] then LCS-PRINT(b, X, i − 1, j)
4  else LCS-PRINT(b, X, i, j − 1)
```

Given: volumes $b \geqslant a_1, \dots, a_n > 0$, and integer values $w_1 \geqslant \cdots \geqslant w_n > 0$.

$$
\begin{array}{ll}
\text{maximize} & w_1 x_1 + \cdots + w_n x_n \\
\text{subject to} & a_1 x_1 + \cdots + a_n x_n \leqslant b, \\
& x_i = 0, 1, \ i = 1, \dots, n.
\end{array}
$$

In other words, find a subset $i_1 < \cdots < i_k$ of the set of items $1, \dots, n$ (by choosing which $x_i = 1$) such that

- the sum of their volumes $a_{i_1} + \cdots + a_{i_k}$ is less than the volume b of our knapsack,
- the sum of their values $w_{i_1} + \cdots + w_{i_k}$ is maximal.

Subset sum problem find $i_1, \dots, i_k$ with $a_{i_1} + \dots + a_{i_k} = b$.
Obtained by setting $w_i = a_i$. Now if there is a solution with value b, we are done.

Partition problem Given numbers $a_1, \dots, a_n$, find $i_1, \dots, i_k$ such that $a_{i_1} + \dots + a_{i_k}$ is as close as possible to $(a_1 + \dots + a_n)/2$.

Solve by dynamic programming. For $1 \leqslant k \leqslant n$,

$$m_k(p) = \min\{\, a_1 x_1 + \cdots + a_k x_k : w_1 x_1 + \cdots + w_k x_k = p \,\}.$$

If the set is empty the minimum is ∞. Let $w = w_1 + \cdots + w_n$. The array $\langle m_{k+1}(0), \ldots, m_{k+1}(w) \rangle$ can be computed from $\langle m_k(0), \ldots, m_k(w) \rangle$:

- If $w_{k+1} > p$ then $m_{k+1}(p) = m_k(p)$ (item $k+1$ cannot be used).
- Otherwise, decide whether to use item $k+1$:

$$m_{k+1}(p) = \min\{\, m_k(p), a_{k+1} + m_k(p - w_{k+1}) \,\}.$$

The optimum is $\max\{\, p : m_n(p) \leqslant b \,\}$.
Complexity: $O(nw)$ steps, (counting additions as single steps).
Why is this not a polynomial algorithm?

What if we want the exact equation $a_{i_1} + \cdots + a_{i_k} = b$? Assume that a_i, b are also integers.
For $1 \leqslant k \leqslant n$,

$$A_k(p) = \{\, a_1 x_1 + \cdots + a_k x_k : w_1 x_1 + \cdots + w_k x_k = p \,\} \cap \{1, \ldots, b\}.$$

The array $\langle A_{k+1}(0), \ldots, A_{k+1}(w) \rangle$ can be computed from $\langle A_k(0), \ldots, A_k(w) \rangle$:

- If $w_{k+1} > p$ then $A_{k+1}(p) = A_k(p)$.
- Otherwise,

$$A_{k+1}(p) = A_k(p) \cup a_{k+1} + A_k(p - w_{k+1}) \cap \{1, \ldots, b\},$$

  where $x + \{u_1, \ldots, u_q\} = \{x + u_1, \ldots, x + u_q\}$.

Now the optimum is $\max\{\, p : b \in A_n(p) \,\}$.
Complexity: $O(nwb)$.

Example: Let $\{a_1, \ldots, a_5\} = \{2, 3, 5, 7, 11\}$, $w_i = 1$, $b = 13$. Otherwise,

	1	2	3	4
1	$\{2\}$	$\{\}$	$\{\}$	$\{\}$
2	$\{2, 3\}$	$\{5\}$	$\{\}$	$\{\}$
3	$\{2, 3, 5\}$	$\{5, 7, 8\}$	$\{10\}$	$\{\}$
4	$\{2, 3, 5, 7\}$	$\{5, 7, 8, 9, 10, 12\}$	$\{10, 12\}$	$\{\}$
5	$\{2, 3, 5, 7, 11\}$	$\{5, 7, 8, 9, 10, 12, 13\}$	$\{10, 12\}$	$\{\}$

$\max\{ p : 13 \in A_5(p) \} = 2$.

Application: money changer problem. Produce the sum b using smallest number of coins of denominations $a_1, \ldots, a_n$ (at most one of each).

There is no general recipe. In general, greedy algorithms assume that

- There is an objective function $f(x_1, \dots, x_n)$ to optimize that depends on some choices $x_1, \dots, x_n$. (Say, we need to maximize $f()$.)
- There is a way to estimate, roughly, the contribution of each choice x_i to the final value, but without taking into account how our choice will constrain the later choices.
- The algorithm is greedy if it still makes the choice with the best contribution.

The greedy algorithm is frequently not the best, but sometimes it is.

Example: activity selection. Given: activities

$$[s_i, f_i)$$

with starting time s_i and finishing time f_i. Goal: to perform the largest number of activites.

Greedy algorithm: sort by f_i. Repeatedly choose the activity with the smallest f_i compatible with the ones already chosen. Rationale: smallest f_i restricts our later choices least.

Example

$$(1,4),(3,5),(0,6),(5,7),(3,8),(5,9),(6,10),$$
$$(8,11),(8,12),(2,13),(12,14).$$

Chosen: $(1,4),(5,7),(8,11),(12,14)$.

Is this correct? Yes, by design, since we always choose an activity compatible with the previous ones.

Is this best? By induction, it is sufficient to see that in the first step, the greedy choice is best possible. And it is, since if an activity is left available after the some choice, it is also left available after the greedy choice.

- The knapsack problem, considered above under dynamic programming, also has a natural greedy algorithm. It is easy to see that this is not optimal. (Exercise.)
- But there is a version of this problem, called the fractional knapsack problem, in which we are allowed to take any fraction of an item. The greedy algorithm is optimal in this case. (Exercise.)

There are cases of algorithmic problems when nothing much better is known than trying out all possibilities.
Example: Maximum independent set in a graph. Let $G = (V, E)$, be an undirected graph on the set of nodes V, with set of edges E.

- A set S of vertices is independent if no two vertices $u, v \in S$ are connected by an edge: $\{u, v\} \notin E$.
- A maximal independent set is one that cannot be extended to a larger independent set.
- A maximum independent set is an independent set of maximal size.

The black point forms by itself a maximal independent set, the white ones a maximum independent set.

It is easy to find a maximal independent set, but it is very hard to find a maximum independent set. This is what we are interested in. Let $N_G(x)$ denote the set of neighbors of point x in graph G, including x itself.

Let us fix a vertex x. There are two kinds of independent vertex set S: those that contain x, and and those that do not.

- If $x \notin S$ then S is an independent subset of the graph $G \setminus \{x\}$ (of course, the edges adjacent to x are also removed from G).
- If $x \in S$ then S does not intersect $N_G(x)$, so $S \setminus \{x\}$ is an independent subset of $G \setminus N_G(x)$.

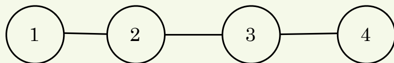
This gives rise to a simple recursive search algorithm.

Algorithm 13.1: MAX-INDEP(G)

Returns a maximum independent set of the graph $G = (V, E)$.
We write $V = V(G)$.

```
1  if G has no edges then return V(G)
2  else
3      x ← a vertex from G
4      S_1 ← Max-Indep(G \ {x})
5      S_2 ← {x} ∪ Max-Indep(G \ N_G(x))
6      return the larger of S_1, S_2
```

We can improve on the simple search by noticing that some searches cannot lead to improvement. For example in the graph



if we found the set $\{1, 3\}$ then since the possibility that $1 \in S$, $3 \notin S$ cannot give us an independent set of size > 2 anymore, we need not explore it. There is also no need to explore any possibilities after $2 \in S$. We formalize this as follows:

$$\textsc{Max-Indep-LB}(G, b)$$

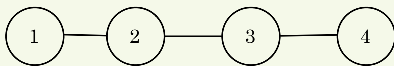
is an algorithm that returns an independent subset of G if it can return one of size $> b$, otherwise it returns $\emptyset$.

Algorithm 13.2: Max-Indep-LB(G, b)
Returns a maximum independent set of the graph G if there is one of size $> b$, otherwise returns $\emptyset$.

```
1  if |V(G)| ⩽ b then return ∅
2  else if G has no edges then return V(G)
3  else
4      x ← a vertex from V
5      S_1 ← Max-Indep-LB(G \ {x}, b)
6      b' ← max(b, |S_1|)
7      S_2 ← {x} ∪ Max-Indep-LB(G \ N_G(x), b' − 1)
8      S ← the larger of S_1, S_2
9      if |S| > b then return S else return ∅
```

We will call the algorithm as Max-Indep-LB$(G, 0)$.
This kind of algorithm is called a branch-and-bound algorithm: maintaining a bound sometimes helps pruning the branches.

Let us trace this algorithm on the graph



$(\cancel{1}) : (G \setminus \{1\}, 0)$
$(\cancel{1}, \cancel{2}) : (G \setminus \{1,2\}, 0)$
$(\cancel{1}, \cancel{2}, \cancel{3}) : (G \setminus \{1,2,3\}, 0)$ returns $\{4\}$, $b' = 1$
$(\cancel{1}, \cancel{2}, 3) : (G \setminus \{1,2,3,4\}, 1-1=0)$ returns $\emptyset$,
so $(\cancel{1}, \cancel{2}) : (G \setminus \{1,2\}, 0)$ returns $\{4\}$, $b' = 1$.
$(\cancel{1}, 2) : (G \setminus \{1,2,3\}, 1-1=0)$ returns $\{4\}$,
so $(\cancel{1}) : (G \setminus \{1\}, 0)$ returns $\{2,4\}$, $b' = 2$.
$(1) : (G \setminus \{1,2\}, 2-1=1)$
$(1, \cancel{2}, \cancel{3}) : (G \setminus \{1,2,3\}, 1)$ returns $\emptyset$ by bound, $b' = 2$.
$(1, \cancel{2}, 3) : (G \setminus \{1,2,3,4\}, 1-1=0)$ returns $\emptyset$,
so $(1) : (G \setminus \{1,2\}, 2-1=1)$ returns $\emptyset$ by bound,
giving the final return $\{2,4\}$.

Graphs $G(V, E)$: set V of vertices or nodes, or points, and set E of edges or lines.
Directed or undirected. Loop edges. Parallel edges.
Representations:

Adjacency list good for sparse graphs, works for all kinds of graph.

Adjacency matrix (vertex-vertex), good for dense graphs. If there are parallel edges, they can be represented by multiplicities.

Incidence matrix (vertex-edge).

Some other graph notions: path, (no repeated edges or nodes), walk (repetitions allowed), cycle.
We write $u \rightsquigarrow v$ if there is a directed path from u to v.

Directed graph $G = (V, E)$. Call some vertex s the source. Finds shortest path (by number of edges) from the source s to every vertex reachable from s.

Rough description of breadth-first search:

Distance $x.d$ from s gets computed.

Black nodes have been visited.

Gray nodes frontier.

White nodes unvisited.

Algorithm 14.1: Generic graph search

```
1  paint s grey
2  while there are grey nodes do
3      take a grey node u, paint it black
4      for each white neighbor v of u do
5          v.d ← u.d + 1
6          make v a grey child of u
```

To turn this into breadth-first search, make the set of grey nodes a queue Q. Now Q is represented in two ways, as a queue, and as a subset of V (the grey nodes). The two representations must remain synchronized.

Analysis: $O(|V| + |E|)$ steps, since every link is handled at most once.

Algorithm 14.2: BFS(G,s)
Input: graph $G=(V,Adj)$.
Returns: the tree parents $u.\pi$, and the distances $u.d$.
Queue Q makes this search breadth-first.

```
1  for u ∈ V \ {s} do
2      u.color ← white; u.d ← ∞; u.π ← nil
3  s.color ← gray; s.d ← 0; Q ← {s}
4  while Q ≠ ∅ do
5      u ← Q.head; DEQUEUE(Q)
6      for v ∈ u.Adj do
7          if v.color = white then
8              v.color ← gray; v.d ← u.d + 1; v.π ← u
9              ENQUEUE(Q, v)
10     u.color ← black
```

In the breadth-first tree defined by π, each path from node to root is a shortest path.

Sometimes, it takes some thinking to see that a problem is a shortest path problem.
Example: how to break up some composite words?

> *Personaleinkommensteuerschätzungskommissionsmitglieds-reisekostenrechnungsergänzungsrevisionsfund* (Mark Twain)

With a German dictionary, break into relatively few components.

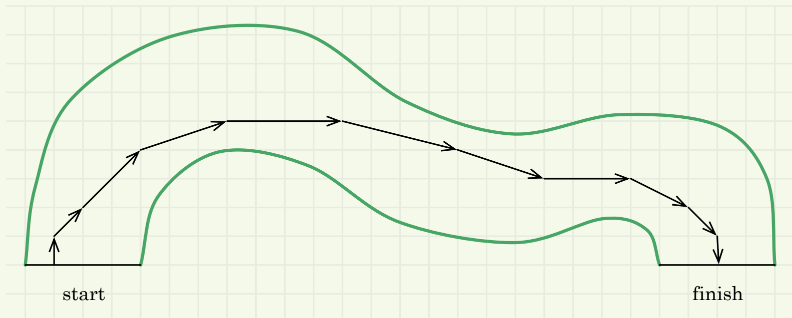
Graph points all division points of the word, including start and end.

Edges if the word between the points is in the dictionary (maybe without the "s" at the end).

Path between the start and end corresponds to a legal breakup.

(Note that this graph is acyclic.)

- The word breakup problem is an example where the graph is given only implicitly: To find out whether there is an edge between two points, you must make a dictionary lookup.
- We may want to minimize those lookups, but minimizing two different objective functions simultaneously (the number of division points and the number of lookups) is generally not possible.
  (For minimizing lookups, depth-first search seems better.)



In each step, the speed vector can change only by 1 in each direction. We have to start and arrive with speed 1, vertical direction. There is a graph in which this is a shortest path problem.

Vertices (point, speed vector) pairs (p, v).

Edges between (p_1, v_1) and (p_2, v_2): if $p_2 - p_1 = v_1$, $|v_2 - v_1|_\infty \leqslant 1$.
Here $|(x, y)|_\infty = \max(|x|, |y|)$ is the so-called maximum norm.

- Similar to breadth-first search, but in our generic graph search, put grey nodes on a stack, rather than a queue.
- In applications, it provides a useful structure for exploring the whole graph, not just the nodes reachable from a given source. So it is restarted as long as there are any unvisited nodes.
- Predecessor $v.\pi$ as before. Now the predecessor subgraph may be a depth-first forest, not just a depth-first tree, when there are several sources.
- Global variable *time* used to compute timestamps: discovery time $v.d$, finishing time $v.f$ (when all descendants have been explored).

Recursive algorithm DFS-VISIT(G, u). Cost $O(V + E)$.

Algorithm 15.1: DFS-Visit(G, u)

Assumes that unvisited nodes of G are white.

```
1  time++; u.d ← time; u.color ← gray
2  for each v ∈ u.Adj do
3      if v.color = white then  v.π ← u; DFS-Visit(G, v)
4  u.color ← black; time++; u.f ← time
```

Algorithm 15.2: DFS(G)

```
1  for each vertex u ∈ G.V do
2      u.color ← white; u.π ← nil
3  for each vertex u ∈ G.V do
4      if u.color = white then DFS-Visit(G, u)
```

Properties of depth-first search

Theorem (Parenthesis)

$v.d, v.f$ behave as parentheses.

Theorem (White path)

Vertex v is a descendant of vertex u iff at time $u.d$, it can be reached from u on a white path.

Classification of edges:

Tree edges

Back edges

Forward edges (nonexistent in an undirected graph)

Cross edges (nonexistent in an undirected graph)

- Assume that in a directed graph $G = (V, E)$ there are no cycles. Then it is called a directed acyclic graph (dag).
- Example: course scheduling with prerequisites.
- Write $u \preceq v$ if there is a path (possibly of length 0) from u to v, and $u \prec v$ when $u \preceq v$ and $u \neq v$.
- The relation $\preceq$ is transitive: $u \preceq v$ and $v \preceq w$ implies $u \preceq w$.
- The relation $\preceq$ is also antisymmetric: $u \preceq v$ and $v \preceq u$ implies $u = v$. Indeed, a walk $u \to v \to u$ would contain a cycle.
- A relation $\preceq$ that is transitive and antisymmetric is called a partial order. So a directed acyclic graph always gives rise to a partial order.
- A partial order $\preceq$ is a total order if for all x, y we have $x \preceq y$ or $y \preceq x$.

- Examples of partial order:
  - $a|b$ were a, b are positive integers.
  - $A \subset B$ where A, B are sets.
  - $a < b$ where a, b are real numbers. This is also a total order.
- Every partial order $\prec$ gives rise to an acyclic graph: introduce edges $u \to v$ for every u, v with $u \prec v$.
- Call an edge (u, v) of a directed acyclic graph necessary if it is the only directed path from u to v.
- How to find all necessary edges?

Theorem

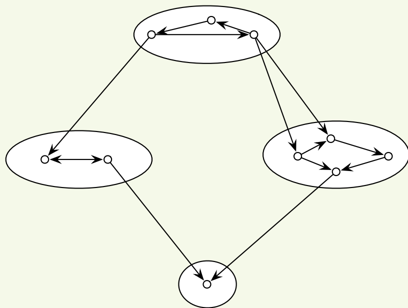
Let E' be the set of all necessary edges of a directed acyclic graph $G = (V, E)$. Then E' defines the same relation $\prec$ on V as E.

- Can we turn every partial order $\prec$ into a total order by adding some more $x \prec y$ relations?
- This is equivalent to the following: Given an acyclic graph $G = (V, E)$, is there a listing $v_1, \ldots, v_n$ of its vertices in such a way that if $(v_i, v_j) \in E$ then $i < j$?
- Yes, and DFS(G) finds the listing very efficiently. Sorting by $v.f$ (backwards) is the desired order.

Theorem

The graph G is acyclic iff DFS(G) *yields no back edges.*

- If the graph G is not acyclic, then write $u \asymp v$ if $u \preceq v$ and $v \preceq u$.
- $u \asymp v$ is an equivalence relation (reflexive, symmetric, transitive). It breaks up V into equivalence classes, called strongly connected components.
- Taking these components as new vertices, we get an acyclic graph.



Let G^T be the graph in which each edge of G is reversed.

Algorithm 15.3: Find strongly connected components

1. DFS(G) to compute finishing times $u.f$
2. DFS(G^T), taking points in order of decreasing $u.f$
3. **return** each tree of the last step as a separate strongly connected component.

To show correctness, look at the last element u chosen to perform DFS(G, u). This is the first one to do DFS(G^T, u).
Claim: The elements visited in DFS(G^T, u) are in the strong component of u.

Indeed, if $u \overset{G^T}{\rightsquigarrow} v$ then $v \overset{G}{\rightsquigarrow} u$. Then DFS($G$) has not visited v before choosing u, otherwise would have found u in the search that visited v. Then $u \rightsquigarrow v$, since DFS(G, u) left nothing unvisited.
Now delete the strong component of u, and repeat the reasoning...

Shortest paths with edge weights (lengths)

- Weight of edge $e = (u, v)$: $w(e) = w(u, v)$.
- Weight of path: the sum of the weights of its edges.
- Shortest path: lightest path.
- Distance $\delta(u, v)$ is the length of lightest path from u to v.

Variants of the problem:

- Single-pair (from source s to destination t).
- Single-source s: to all reachable points. Returns a tree of lightest paths, represented by the parent function $v \mapsto v.\pi$.
- All-pairs.

Negative weights? These are also interesting, but first we assume that all weights are nonnegative.

- Dijkstra's algorithm for single-source shortest paths is a refinement of breadth-first search.
- During the algorithm, we maintain the candidate distance $v.d$ for each vertex v. By the end, $v.d$ will equal the distance $\delta(s, v)$ from s to v.

The updating of $v.d$ using new information is called relaxation.

Algorithm 16.1: RELAX(u, v, w)

Update the candidate distance from s to v using the edge length $w(u, v)$. If the distance through u is shorter, make u the parent of v.

1 **if** $v.d > u.d + w(u, v)$ **then**
2 $\quad d.v \leftarrow u.d + w(u, v)$
3 $\quad v.\pi \leftarrow u;$

For Dijkstra's algorithm, the points v are on a min-heap Q keyed by the field $v.key = v.d$. It supports the operation

$$\textsc{Decrease-Key}(Q, v, k),$$

which decreases the key value of v to k (floating v up the heap Q as needed):

Algorithm 16.2: $\textsc{Relax}(u, v, w, Q)$

```
1  if v.d > u.d + w(u, v) then
2      Decrease-Key(Q, v, u.d + w(u, v))
3      v.π ← u;
```

Algorithm 16.3: DIJKSTRA(G, w, s)

On termination, $u.d = \delta(s, u)$, the distance, and $u.\pi$ represents a tree.

```
1  Initialize-Single-Source(G, s)
2  Q ← V, a min-heap
3  while Q ≠ ∅ do
4      u ← Extract-Min(Q)
5      for each v ∈ u.Adj do Relax(u, v, w, Q)
```

Correctness: Order the nodes by distance $\delta(s, u)$ in a sequence $s = u_1, \ldots, u_n$. We prove by induction on k that at the end, $u_k.d = \delta(s, u)$. True for $k = 1$. Let u_j be the previous element on some shortest path to u_k. By inductive assumption, the final $u_j.d = \delta(s, u_j)$. The final $u_k.d > \delta(s, u_j)$ is computed after the final $u_j.d$. Relaxation with $u_j.d$ gives $u_k.d = u_j.d + w(u_j, u_k)$.

Cost of Dijkstra's algorithm

Let $n = |V|$, $m = |E|$. Potentially a priority queue update for each edge: $O(m \log n)$. This depends much on how dense is G.

All pairs, negative edges

- Assume that there are no negative edge weights: To compute distances for all pairs, repeat Dijkstra's algorithm from each vertex as source. Complexity $O(mn \log n)$, not bad. But can be improved, if G is dense, say $m = \Theta(n^2)$.
- If there are negative edges, new question: how about a negative cycle C? Then the minimum length of walks is $-\infty$ (that is does not exist) between any pair of nodes u, v such that $u \rightsquigarrow C$ and $C \rightsquigarrow v$. (A walk can go around a cycle any number of times.)
- More modest goal: find out whether there is a negative cycle: if there is not, find all distances (shortest paths are then easy).

Dynamic programming. Recursion idea: Call a k-path one that uses only $1, \dots, k$ for intermediate vertices. $d_{ij}^{(k)}$ = distance over k-paths.
A k-path from i to j that is not a $k-1$-path consists of a $k-1$-path from i to k and a $k-1$-path from k to j.

Algorithm 16.4: FLOYD-WARSHALL(W)
If there is any i in a negative cycle then $d_{ii}^{(n)} < 0$.
Otherwise $d_{ij}^{(n)}$ = length of a lightest path $i \to j$.

1 **for** $i, j = 1$ **to** n **do** $d_{ij}^{(0)} \leftarrow w(i, j)$ // Adjacency matrix
2 **for** $k = 1$ **to** n **do**
3   **for** $i = 1$ **to** n **do**
4   **for** $j = 1$ **to** n **do** $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

Cost: $O(n^3)$.

- Transitive closure of a directed graph.
- Arbitrage. Exchange rate x_{ij}: this is how much it costs to buy a unit of currency j in currency i. Let $w(i,j) = -\log x_{ij}$. We can become infinitely rich if and only if there is negative cycle in this graph.

Longest path in an acyclic graph

The Floyd-Warshall algorithm with negative weights would do it, but the following is more efficient.

```
1  for each vertex u in backwards topologically sorted order do
2      for for each vertex v in u.Adj do Relax(u, v, w)
```

(Of course, RELAX(u, v, w) is taking the maximum here, not minimum.)

Application: finding the longest path is the basis of all project planning algorithms. (PERT method). You are given a number of tasks (for example courses with prerequisites). Figure out the least time needed to finish it all. This assumes unlimited resources, for example ability to take any number of courses in a semester.

With respect to connectivity, another important algorithmic problem is to find the smallest number of edges that still leaves an undirected graph connected. More generally, edges have weights, and we want the lightest tree. (Or heaviest, since negative weights are also allowed.)

Generic algorithm: add a safe edge each time: an edge that does not form a cycle with earlier selected edges.

A cut of a graph G is any partition $V = S \cup T$, $S \cap T = \emptyset$. It respects edge set A if no edge of A crosses the cut. A light edge of a cut: a lightest edge across it.

Theorem

If A is a subset of some lightest spanning tree, S a cut respecting A then after adding any light edge of S to A, the resulting A' still belongs to some lightest spanning tree.

Keep adding a light edge adjacent to the already constructed tree.

Algorithm 17.1: MST-PRIM(G, w, r)
Lightest spanning tree of (G, w), root r, returned in $v.\pi$.

1 **for** each vertex u **do** $u.key \leftarrow \infty$; $u.\pi \leftarrow nil$
2 $r.key \leftarrow 0$; $Q \leftarrow G.V$, a min-heap keyed by $v.key$
3 **while** $Q \neq \emptyset$ **do**
4     $u \leftarrow$ EXTRACT-MIN(Q)
5   **for** $v \in u.Adj$ **do**
6   **if** $v \in Q$ **and** $w(u, v) < v.key$ **then** $v.\pi \leftarrow u$
7   DECREASE-KEY$(Q, v, w(u, v))$

The only difference to Dijkstra's algorithm is in how the key gets decreased: it is set to the candidate lightest edge length from v to the tree, not the candidate lightest path length from v to the root.

Example (Workers and jobs)

Suppose that we have n workers and n jobs. Each worker is capable of performing some of the jobs. Is it possible to assign each worker to a different job, so that workers get jobs they can perform?

It depends. If each worker is familiar only with the same one job (say, digging), then no.

Example

At a dance party, with 300 students, every boy knows 50 girls and every girl knows 50 boys. Can they all dance simultaneously so that only pairs who know each other dance with each other?

- Bipartite graph: left set A (of girls), right set B (of boys).
- Matching, perfect matching.

Theorem

If every node of a bipartite graph has the same degree $d \geqslant 1$ then it contains a perfect matching.

Examples showing the (local) necessity of the conditions:

- Bipartiteness is necessary, even if all degrees are the same.
- Bipartiteness and positive degrees is insufficient.

Example

6 tribes partition an island into hunting territories of 100 square miles each. 6 species of tortoise, with disjoint habitats of 100 square miles each.
Can each tribe pick a tortoise living on its territory, with different tribes choosing different totems?

For $S \subseteq A$ let

$$\mathbf{N}(S) \subseteq B$$

be the set of all neighbors of the nodes of A.
Special property: *For every $S \subseteq A$ we have $|\mathbf{N}(S)| \geqslant |S|$.*
Indeed, the combined hunting area of any k tribes intersects with at least k tortoise habitats.

The above property happens to be the criterion also in the general case:

Theorem (The Marriage Theorem)

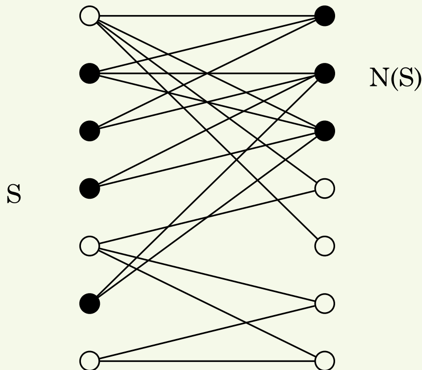
A bipartite graph has a perfect matching if and only if $|A| = |B|$ and for every $S \subseteq A$ we have $|\mathbf{N}(S)| \geqslant |S|$.

The condition is necessary.

Proposition

The condition implies the same condition for all $S \subseteq B$.

Prove this as an exercise.



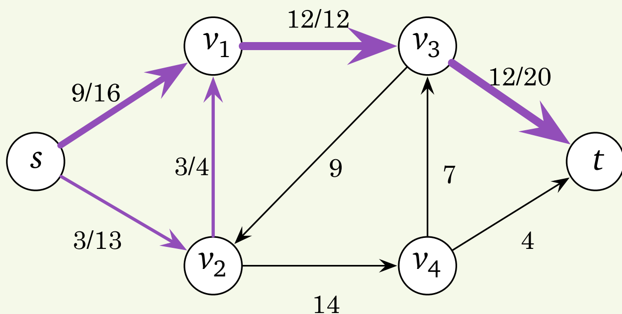
- Directed graph. Source s, sink t. Every vertex is on some path from s to t.
- Flow: function $f(u,v)$ on all edges (u,v) showing the amount of material going from u to v. We are only interested in the net flow $\hat{f}(u,v) = f(u,v) - f(v,u)$: then $\hat{f}(v,u) = -\hat{f}(u,v)$. So we simply require

$$f(v,u) = -f(u,v).$$

- The total flow entering a non-end node equals the total flow leaving it:

$$\text{for all } u \in V \setminus \{s,t\} \sum_{v} f(u,v) = 0.$$

- Each edge (u,v) imposes a capacity $c(u,v) \geqslant 0$ on the flow: $f(u,v) \leqslant c(u,v)$. (We may have $c(u,v) \neq c(v,u)$.)

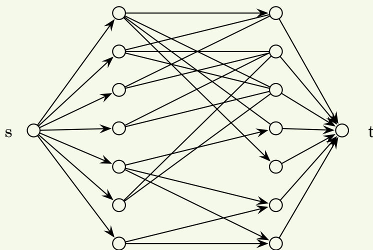


The notation f/c means flow f along an edge with capacity c.

- Our goal is to maximize the value of the flow f, that is

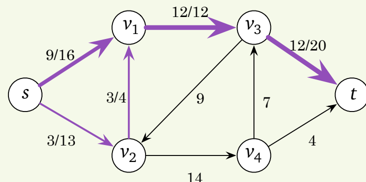
$$|f| = \sum_v f(s,v) = \sum_v f(v,t).$$

Application to matching

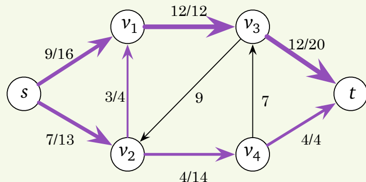


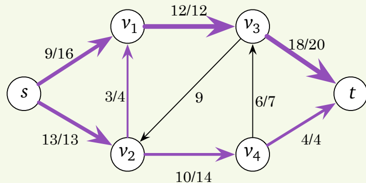
- n points on left, n on right. Edges directed to right, with unit capacity.
- Perfect matching → flow of value n.
- Flow of value n → perfect matching? Not always, but fortunately (as will be seen), there is always an integer maximum flow.

Idea for increasing the flow: do it along some path.



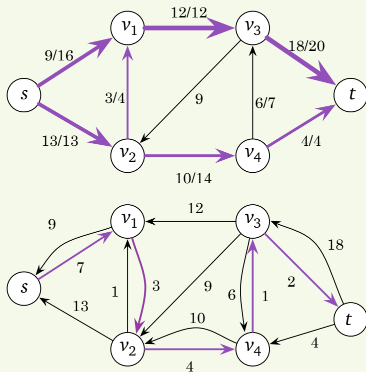
Increment it along the path s-v_2-v_4-t by 4.
Then increment along the path s-v_2-v_4-v_3-t by 6. Now we are stuck: no more direct way to augment.





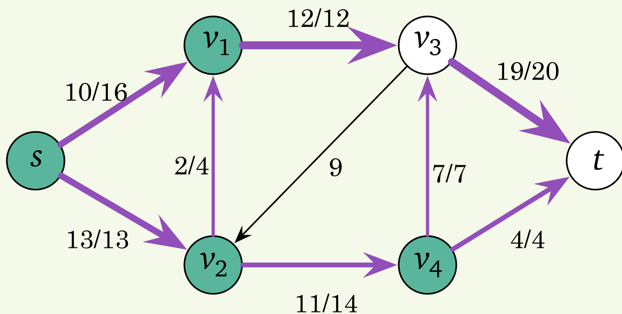
New idea: Increase (by 1) on (s, v_1), decrease on (v_1, v_2), increase on (v_2, v_4, v_3, t).

Generalization: Given a flow f, residual capacity $c_f(u,v) = c(u,v) - f(u,v)$. Makes sense even with negative $f(u,v)$. The residual network may have edges (with positive capacity) that were not in the original network. An augmenting path is an s-t path in the residual network (with some flow along it). (How does it change the original flow?)



Residual capacity of the path: the minimum of the residual capacities of its edges (in the example, 1).

We obtained:



This cannot be improved: look at the cut (S, T) with $T = \{v_3, t\}$.

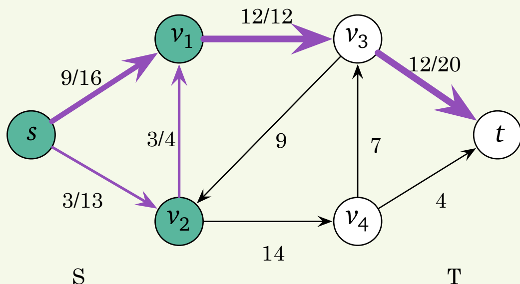
Keep increasing the flow along augmenting paths.

Questions

1. If it terminates, did we reach maximum flow?
2. Can we make it terminate?
3. How many augmentations may be needed? Is this a polynomial time algorithm?

Neither of these questions is trivial. The technique of cuts takes closer to the solution.

Cut (S, T) is a partition of V with $s \in S$, $t \in T$.
Net flow $f(S,T) = \sum_{u \in S, v \in T} f(u,v)$.
Capacity $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$. Obviously, $f(S,T) \leqslant c(S,T)$.



In this example, $c(S,T) = 26$, $f(S,T) = 12$.

Lemma

$f(S,T) = |f|$, *the value of the flow.*

Corollary

The value of any flow is bounded by the capacity of any cut.

Theorem (Max-flow, min-cut)

The following properties of a flow f are equivalent.

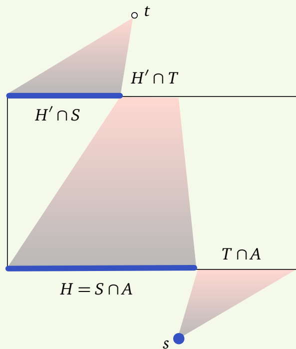
1. $|f| = c(S,T)$ *for some cut* (S,T).
2. f *is a maximum flow.*
3. *There are no augmenting paths to* f.

The equivalence of the first two statements says that the size of the maximum flow is equal to the size of the minimum cut.

Proof: 1⇒2 and 2⇒3 are obvious. The crucial step is 3⇒1.

Given f with no augmenting paths, we construct (S,T): let S be the nodes reachable from s in the residual network G_f.

Using Max-Flow Min-Cut. Assume there is no perfect matching in the bipartite graph $G = (A \cup B, E)$, with $|A| = |B| = n$. We find a bottleneck $H \subseteq A$ with $|\mathbf{N}(H)| < |H|$.
Flow network over $A \cup B \cup \{s, t\}$ as before. Since there is no perfect matching, the maximum flow has size $< n$. So there is a cut (S, T), $s \in S$, $t \in T$, with $c(S, T) < n$.



Let $H = S \cap A$, $H' = \mathbf{N}(H)$.

$$
\begin{aligned}
n &> c(S, T) \\
&= c(\{s\}, T) + c(S \cap A, T \cap B) + c(S, \{t\}) \\
&\geqslant (n - |H|) + |H' \cap T| + |H' \cap S| \\
&= n - |H| + |H'|,
\end{aligned}
$$

$|H| > |H'|$.

- Does the Ford-Fulkerson algorithm terminate? Not necessarily (if capacities are not integers), unless we choose the augmenting paths carefully.
- Integer capacities: always terminates, but may take exponentially long.
  Network derived from the bipartite matching problem: each capacity is 1, so we terminate in polynomial time.
- Edmonds-Karp: use breadth-first search for the augmenting paths. Why should this terminate?

Lemma

In the Edmonds-Karp algorithm, the shortest-path distance $\delta_f(s,v)$ increases monotonically with each augmentation.

Proof: Let $\delta_f(s,u)$ be the distance of u from s in G_f, and let f' be the augmented flow. Assume, by contradiction $\delta_{f'}(s,v) < \delta_f(s,v)$ for some v: let v be the one among these with smallest $\delta_{f'}(s,v)$. Let $u \to v$ be be a shortest path edge in $G_{f'}$, and

$$d := \delta_f(s,u)(= \delta_{f'}(s,u)), \text{ then } \delta_{f'}(s,v) = d+1.$$

Edge (u,v) is new in $G_{f'}$; so (v,u) was a shortest path edge in G_f, giving $\delta_f(s,v) = d-1$. But $\delta_{f'}(s,v) = d+1$ contradicts $\delta_{f'}(s,v) < \delta_f(s,v)$.

An edge is said to be critical, when it has just been filled to capacity.

Lemma

Between every two times that an edge (u, v) is critical, $\delta_f(s, u)$ increases by at least 2.

Proof: When it is critical, $\delta_f(s, v) = \delta_f(s, u) + 1$. Then it disappears until some flow f'. When it reappears, then (v, u) is critical, so

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geqslant \delta_f(s, v) + 1 = \delta_f(s, u) + 2.$$

Corollary

We have a polynomial algorithm.

Proof: Just bound the number of possible augmentations, noticing that each augmentation makes some edge critical.

Let $n = |V|$, $m = |E|$. Each edge becomes critical at most $n/2$ times. Therefore there are at most $m \cdot n/2$ augmentations. Each augmentation may take $O(m)$ steps: total bound is

$$O(m^2 n).$$

There are better algorithms: Goldberg's push-relabel algorithm, also given in your book, achieves $O(n^3)$.

In some cases we cannot hope to find a complete solution to some algorithmic problem, in reasonable time: we have to be satisfied with an approximate solution. This makes sense in case of optimization problems.

Suppose we have to maximize a positive function. For object function $f(x, y)$ for $x, y \in \{0, 1\}^n$, for input x, the optimum is

$$M(x) = \max_y f(x, y)$$

where y runs over the possible values which we will call witnesses.

For $0 < \lambda$, an algorithm $A(x)$ is a λ-approximation if

$$f(x, A(x)) > M(x)/\lambda.$$

For minimization problems, with minimum $m(x)$, we require $f(x, A(x)) < M(x)\lambda$.

Try local improvements as long as you can.

Example (Maximum cut)

Graph $G = (V, E)$, cut $S \subseteq V$, $\overline{S} = V \setminus S$. Find cut S that maximizes the number of edges in the cut:

$$|\{\, \{u, v\} \in E : u \in S, v \in \overline{S} \,\}|.$$

Greedy algorithm:

> *Repeat: find a point on one side of the cut whose moving to the other side increases the cutsize.*

Theorem

If you cannot improve anymore with this algorithm then you are within a factor 2 of the optimum.

Proof. The unimprovable cut contains at least half of all edges. □

Generalize maximum cut for the case where edges e have weights w_e, that is maximize

$$\sum_{u \in S, v \in \overline{S}} w_{uv}.$$

- Question The greedy algorithm brings within factor 2 of the optimum also in the weighted case. But does it take a polynomial number of steps?
- New idea: decide each "$v \in S$?" question by tossing a coin. The expected weight of the cut is $\frac{1}{2}\sum_e w_e$, since each edge is in the cut with probability $1/2$.

What does the greedy algorithm for vertex cover say?

The following, less greedy algorithm has better performance guarantee.

`Approx_Vertex_Cover`(G):

```
1  C ← ∅
2  E′ ← E[G]
3  while E′ ≠ ∅ do
4      let (u, v) be an arbitrary edge in E′
5      C ← C ∪ {u, v}
6      remove from E′ every edge incident on either u or v
7  return C
```

Theorem

`Approx_Vertex_Cover` *has a ratio bound of 2.*

Proof. The points of C are endpoints of a matching. Any optimum vertex cover must contain half of them. $\square$

How bad is greedy vertex cover?

Greedy algorithm: Repeat the following, as long as you can:

> *Find a vertex with the largest possible degree. Delete it from the graph, along with all the edges incident to it.*

We will show a graph where this algorithm does not even have a constant approximation ratio.

- $A = \{a_1, \dots, a_n\}$.
- $B_2 = \{b_{2,1}, \dots, b_{2,\lfloor n/2 \rfloor}\}$. Here, $b_{2,1}$ is connected to a_1, a_2, further $b_{2,2}$ to a_3, a_4, and so on.
- $B_3 = \{b_{3,1}, \dots, b_{3,\lfloor n/3 \rfloor}\}$. Here $b_{3,1}$ is connected to a_1, a_2, a_3, further $b_{3,2}$ to a_4, a_5, a_6, and so on.
- and so on.
- B_n consists of a single point connected to all elements of A.

The n elements of A will cover all edges. Greedy algorithm:

- Each element of A has degree $\leqslant (n-1)$, so it chooses B_n first.
- Then B_{n-1}, then B_{n-2}, and so on, down to B_2.

The total number of points chosen is

$$\lfloor n/2 \rfloor + \lfloor n/3 \rfloor + \cdots + \lfloor n/n \rfloor \geqslant n(1/2 + \cdots + 1/n) - n \sim n \ln n.$$

Can it be worse than this? No, even for a more general problem: set cover.

Given $(X, \mathscr{F})$: a set X and a family $\mathscr{F}$ of subsets of X, find a min-size subset of $\mathscr{F}$ covering X.

Example: Smallest committee with people covering all skills.

Generalization: Set S has weight $w(S) > 0$. We want a minimum-weight set cover.

The algorithm `Greedy_Set_Cover`$(X, \mathscr{F})$:

```
1  U ← X
2  𝒞 ← ∅
3  while U ≠ ∅ do
4      select an S ∈ ℱ that maximizes |S ∩ U|/w(S)
5      U ← U \ S
6      𝒞 ← 𝒞 ∪ {S}
7  return 𝒞
```

Let $H(n) = 1 + 1/2 + \cdots + 1/n (\approx \ln n)$.

Theorem

`Greedy_Set_Cover` *has a ratio bound* $\max_{S \in \mathcal{F}} H(|S|)$.

See the proof in a graduate course (or in the book).

- Application to vertex cover.

Recall the knapsack problem:
Given: integers $b \geqslant a_1, \dots, a_n$, and integer weights $w_1 \geqslant \cdots \geqslant w_n$.

$$\begin{array}{ll} \text{maximize} & \sum_j w_j x_j \\ \text{subject to} & \sum_j a_j x_j \leqslant \quad b, \\ & \qquad x_i = 0, 1, \ i = 1, \dots, n. \end{array}$$

Idea for approximation: break each w_i into a smaller number of big chunks, and use dynamic programming. Let $r > 0$, $w_i' = \lfloor w_i/r \rfloor$.

$$
\begin{array}{ll}
\text{maximize} & \sum_j w_j' x_j \\
\text{subject to} & \sum_j a_j x_j \leqslant \quad b, \\
 & \qquad x_i = 0, 1,\ i = 1, \ldots, n.
\end{array}
$$

When r is large then our numbers are small, so the dynamic programming solution is not so expensive. But we still get a good approximation to the optimum. (See the detailed estimate in a graduate course.)

Examples

- shortest vs. longest simple paths
- compositeness
- subset sum
- perfect matching
- graph isomorphism
- 3-coloring
- Ultrasound test of sex of fetus.

Decision problems vs. optimization problems vs. search problems.

Example

Given a graph G.

Decision Given k, does G have an independent subset of size $\geqslant k$?

Optimization What is the size of the largest independent set?

Search Given k, give an independent set of size k (if there is one).

Optimization+search Give a maximum size independent set.

Abstract problems Instance. Solution.

Encoding We can pretend that all our problems are about strings.

Example

Hamiltonian cycles.

An NP problem is defined with the help of a polynomial-time function

$$V(x, w)$$

with yes/no values that verifies, for a given input x and witness (certificate) w whether w is indeed witness for x.

Decision problem: Is there a witness (say, a Hamilton cycle)?

Search problem: Find a witness!

The same decision problem may belong to very different verification functions (search problems):

Example

Input: positive integer x that is of the form z^k for any $k > 0$.
Decision: Is x composite?

Witness kind 1: Positive $y \neq x, 1$ dividing x.

Witness kind 2: : Different u, v, w with $u^2 \equiv v^2 \equiv w^2 \pmod{x}$.

The following theorem will be probably proved for you in cryptography or an algebraic algorithms course:

Theorem

If x is not of the form z^k for $k > 0$ then it is composite if and only if there is a b such that the equation $t^2 \equiv b \pmod{x}$ has more than two solutions in t.

Reduction of problem A_1 to problem A_2 (given by verification functions V_1, V_2) with a reduction (translation) function τ:

$$\exists w V_1(x, w) \Longleftrightarrow \exists u V_2(\tau(x), u).$$

We will say $A_1 \leqslant_P A_2$, if there is a polynomially computable τ.

Examples

- Reducing matching to maximum flow.
- Vertex cover to maximum independent set: C is a vertex cover iff $V \setminus C$ is an independent set.

NP-hardness.
NP-completeness.

- Boolean formulas.

$$F(x,y,z) = (x \wedge y \wedge \neg z) \vee (x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z)$$

  is 1 if and only if exactly two of the variables x, y, z are 1. So it is satisfiable, say the assignment $x = 1, y = 1, z = 0$ satisfies it.

$$G(x,y,z,t) = (\neg x \vee y) \wedge (\neg y \vee z) \wedge (\neg z \vee t) \wedge x \wedge \neg t$$

  is not satisfiable. Indeed, $\neg x \vee y$ means the same as $x \Rightarrow y$, that is $x \leqslant y$. So G requires $x \leqslant y \leqslant z \leqslant t$, but also$x = 1,\ t = 0$.
- Example expression of some combinatorial problem via Boolean formulas: 2-coloring.

Theorem

Satisfiability is NP-complete.

Theorem

INDEPENDENT SET is NP-complete.

Reducing SAT to it.
Most NP-complete problems are proved such by a chain of reductions from INDEPENDENT SET, or directly from SAT.

Example

Set cover $\geqslant$ vertex cover $\sim$ independent set.

The End