Design and Analysis of Algorithms
Part II
Dynamic Programming, Recursive and Greedy Algorithms

José Dorronsoro
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Contents

1 Dynamic Programming

1.1 Minimum Distance Problems in Graphs

Back to Minimum Distances

- If (G, c) is a weighted directed graph, we can consider in principle three minimum distance problems:
  - For u, v fixed, find **only** the minimum distance between u and v
  - For u fixed, find the minimum distance between u and **all other** $v \in V$
  - For **all** $u, v \in V$, find the minimum distance between u and v
- While the first problem seems easier, no algorithm for general graphs is better than those available for the second one
  - Notice that a minimal path from u to v is also minimal for all vertices in between
- We can solve the third problem iterating an algorithm for the second over all $u \in V$
- For instance, iterating Dijkstra over all $u \in V$ has a cost $|V| \times O(|E| \log |V|) = O(|V||E| \log |V|)$
- If G is dense, the cost is then $O(|V|^3 \log |V|)$

Improving on Dijkstra

- Assume $V = \{1, \dots, N\}$ and a nonnegative cost c, and let d_{ij} be the minimum distance between i and j
- Set d_{ij}^k be the minimum distance between i, j but where the only intermediate nodes are taken from $\{1, \dots, k\}$
- An optimal path between i and j with $\{1, \dots, k\}$ as intermediate nodes may or may not contain k
- If it doesn't, we have $d_{ij}^k = d_{ij}^{k-1}$
- If it does, we have $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$, for

  - A path having another k between i and k or between k and j cannot be optimal, as we could remove the subpath from k to k
  - A path from i to j is optimal iff the partial subpaths between i and k and k and j are optimal, i.e., $d_{ij}^k = d_{ik}^k + d_{kj}^k$
  - But obviously $d_{ik}^k = d_{ik}^{k-1}$, $d_{kj}^k = d_{kj}^{k-1}$

- We can conclude $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$

Floyd–Warshall Algorithm

- It is clear that $d_{ij}^0 = c(i,j)$ and $d_{ij}^N = d_{ij}$
- Working with adjacency matrices, this suggest the following pseudocode

```
def floyd_warshall(m_g):
    num_nodes = m_g.shape[0]
    d = np.zeros( (num_nodes, num_nodes) )
    d[ : , : ] = m_g
    for k in range( num_nodes ):
        for i in range( num_nodes ):
            for j in range( num_nodes ):
                if i != j:
                    d[i, j] = min(d[i, j], d[i, k] + d[k, j])
    return d
```

Floyd–Warshall Cost

- The time cost is $O(N^3)$, better than iterated Dijkstra for dense graphs
- The space cost is $O(N^2)$ as we use N matrices $N \times N$;
  - We have $d_{ik}^k = d_{ik}^{k-1}$, $d_{kj}^k = d_{kj}^{k-1}$
  - If $c = d_{ik}^{k-1} + d_{kj}^{k-1}$, we have $d_{ij}^k = \min\{d_{ij}^{k-1}, c\}$
- Q: how can we recover the optimal paths?
- Observation: F–W is our first example of a problem solvable by a **DP** algorithm
  - An optimization problem with an **optimal substructure** (obvious: any optimization problem has it) that we are able to make **explicit**
  - The explicit substructure formula also shows F–W to be correct

Minimum Distances for Negative Costs

- Notice that if c is negative, the minimum distance problem does not make sense if G has a negative cycle
  - Simply going around the cycle the cost would go to $-\infty$
- Even if there are no have negative cycles, Dijkstra doesn't work nor do simple ideas such as adding a constant to all costs to make them positive
- **Examples:**

$$\begin{tikzcd}
 & 1 \arrow[dr, "3"] & \\
0 \arrow[ur, "2"] \arrow[dr, "1"] & & 2 \arrow[dl, "-5"] \\
 & 3 &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & 1 \arrow[dr, "2"] & \\
0 \arrow[ur, "2"] \arrow[d, "3"] & & 2 \\
4 \arrow[rr, "-3"] & & 3 \arrow[u, "3"]
\end{tikzcd}$$

Floyd–Warshall with Negative Costs

- Floyd–Warshall also works for directed graphs without negative costs cycles.
  - The explicit optimal substructure also guarantees it, since optimal paths cannot have cycles
- Moreover, it detects whether there are negative cycles
  - Since $c(i, i) = 0$, a path from i to k and from k to i improves on this if its cost is negative
  - If at the end we have $d_{ii}^N < 0$ for some i, there is a negative cycle
- However, F–W means extra work if we only want the minimum paths from a single u to all the other v

Bellman–Ford Algorithm

- To find all minimal paths starting from a fixed u in a directed graph without negative costs cycles, the **Bellman–Ford** algorithm is in general more efficient

- Assume that G is a weighted directed graph without negative cycles; then every optimal path must be cycle–free:
  - A path with a positive cycle cannot be optimal
- Therefore, an optimal path has $\leq |V| - 1$ **nodes other than** u
- Let d_i^k be the minimum distance from u to the node i with at most k nodes other than u, $1 \leq k \leq |V| - 1$
- We then have $d_i^1 = c(u, i)$ and for $2 \leq k \leq |V|$

$$d_i^k = \min\{d_i^{k-1}, d_j^{k-1} + c(j, i) : j = 1, \dots, |V|\}$$

- This leads naturally to a DP algorithm

A First Try at the Bellman–Ford Algorithm

- A first B–F pseudocode is

```
def bellman_ford(u, m_g):
    num_nodes = m_g.shape[0]
    for i = 1, ..., num_nodes:
        d[i, 1] = m_g[u, i]

    for k = 2, ..., num_nodes:
        for i = 1, ..., num_nodes, i != u:
            d[i, k] = d[i, k-1]
            for (j, i) in E:
                d[i,k] = min( d[i, k], d[j, k-1] + m_g[j, i] )

    return d[ : , num_nodes]
```

- The cost is clearly $O(|V|) + (|V| - 1)O(|E|) = O(|V| \times |E|)$
- As in Floyd–Warshall it is clear that we can save in memory by working with just a vector d_i instead of a matrix d_i^k
- We can thus simplify slightly the previous algorithm

The Bellman–Ford Algorithm

- The new pseudocode is

```
def bellman_ford(u, m_g):
    num_nodes = m_g.shape[0]

    for i = 1, ..., num_nodes:
        d[i] = m_g[u, i]

    for k = 2, ..., num_nodes:
        for i = 1, ..., num_nodes, i != u:
            for (j, i) in E:
                d[i] = min (d[i], d[j] + m_g(j,i))
    return d
```

- Exercise: modify the BF algorithm to identify the shortest paths
- Q: Can we detect negative cycles? Good exercise!

A Bellman–Ford Example

- **Example:**

$$\begin{tikzcd}
 & & 1 \arrow[drr, "2"] & & \\
0 \arrow[urr, "2"] \arrow[dr, "3"] & & & & 2 \\
 & 4 \arrow[rr, "-3"] & & 3 \arrow[ur, "3"] &
\end{tikzcd}$$

Explicit Optimal Substructure in Optimization

- In general, any optimization problem has an **Optimal Substructure** property: **the components of a global optimal solution are also optimal for their corresponding subproblems**
  - If we improve any of them, we obviously improve the full solution
- An obvious example is the Minimum Distance Problem:
  - If the path $\{(u_1, u_2), \dots (u_i, u_{i+1}), \dots (u_{K-1}, u_K)\}$ is optimal,
  - Then the same must be true for all the subpaths from u_i to u_j, $1 \leq i < j \leq K$
- In some cases, we cannot exploit on this to derive efficient algorihms: we couldn‘t do it for Dijkstra, Prim or Kruskal
- However, in some cases we can make this structure **explicit**, leading to a possible algorithm
- **Dynamic Programming** algorithms take advantage of an **Explicit Optimal Substructure** (EOS) property

The EOS Property in Floyd–Warshall

- Recall that in Floyd–Warshall we have

$$d_{ij}^{k} = \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}$$

- The EOS property usually suggests naturally iterative (and often expensive) bottom–up algorithms
- Sometimes, while expensive, the algorithms are also efficient
- For instance, the cost of Floyd–Warshall is $O(N^3)$, high but notice that we have to find $O(N^2)$ paths and each may have up to N nodes
- In other words, the problem that DP tries to solve may be quite expensive, even NP–complete in some examples
- But in other cases, the algorithms may be unnecessarily too expensive and other options are better

Bottom–Up Algorithms

- At first sight it would seem that an alternative way to exploit the EOS property could be to write a recursive, top–down algorithm
- However, recursive Divide and Conquer algorithms are efficient only when the subproblem's sizes are
  1. noticeably smaller and
  2. somehow "disjoint": recall for example MergeSort
- However, a recursive algorithm may be far too expensive if the subproblems have an overlapping structure
- Example (though not on optimization): a recursive computation of the Fibonacci number recomputes each number many times over (in fact, a Fibonacci number of times)
- The efficient way to compute Fibonacci numbers is bottom–up, storing computed values in a table

Why Dynamic Programming?

- A third property, often obvious but always needed, is that the subproblems to be solved are **independent**: the solution of a given subproblem does not depend on that of others

  - In F–W, when computing $d^k_{i\,j}$, the optimal path from i to k is independent from the path from k to j
  - Bottom–up solutions (as in Floyd–Warshall) to overlapping, independent problems using matrices to store intermediate results is the second ingredient of DP algorithms

- Q: Why the name Dynamic Programming?
  - Answer in **Richard Ernest Bellman** by S. Dreyfus in the International Transactions in Operational Research, Vol 10 (2003)

1.2 Knapsacks and Sums

0–1 Knapsack

- Assume we have N elements with integer weights w_i and values v_i and a knapsack that stands a maximum weight W
- We want a choice of elements $i_1, \dots, i_K$ such that $\sum_j w_{i_j} \leq W$ and $\sum_j v_{i_j}$ is maximum
- Mathematically, we want to solve a **Constrained Optimization Problem**:

$$\max \sum_1^N v_i x_i \text{ subject to } \sum_1^N w_i x_i \leq W \text{ and } x_i \in \{0,1\}$$

- The "0–1" name derives from the constraint $x_i \in \{0,1\}$

A Greedy Solution?

- There is a natural greedy solution: order the i by decreasing relative values $\pi_i = \frac{v_i}{w_i}$ and select the first K ones such that

$$\sum_1^K w_{i_j} \leq W < \sum_1^{K+1} w_{i_j}$$

- The cost is $O(N)$ (plus that of sorting) but **the solution may not be optimal**
- Consider 3 elements with values $\{15, 11, 10\}$, weights $\{5, 4, 4\}$ and $W = 8$:

  - The greedy knapsack takes element 1 with value 15 but the $\{2, 3\}$ knapsack has a value of 21

- However we'll see later that it works for **Fractionary** Knapsacks, where we can take any fraction of the weight w_i of an element
  - That is, if we go from an integer–valued problem to a real–valued one

Dynamic Programming Solution

- For $0 \le i \le N$, $0 \le w \le W$, let $v(i, w)$ be the optimum value of a knapsack with the first i elements and bound w
- If i is in the optimal selection for $v(i, w)$ we must have
$$v(i, w) = v_i + v(i - 1, w - w_i)$$
- And if i is not in the optimal selection for $v(i, w)$, we must have
$$v(i, w) = v(i - 1, w)$$
- Thus, for $0 \le i \le N$ and $0 \le w \le W$ we arrive at
$$v(i, w) = \max\{v_i + v(i - 1, w - w_i), v(i - 1, w)\}$$

The Iterative Algorithm

- Notice that to compute the i–th row $v(i, w)$ we just need to have already computed the previous row $v(i - 1, w)$
- Clearly $v(0, w) = v(i, 0) = 0$ for all w and i, and $v(i, w) = v(i - 1, w)$ for all $w < w_i$
- **Example:** 3 elements with values $v = \{15, 11, 10\}$, weights $w = \{5, 4, 4\}$ and $W = 8$:

But . . .

- The iterative DP algorithm has to fill a $N \times W$ matrix
- Since computing $v(i, w)$ has a $O(1)$ cost, the overall cost is

$$O(N \times W) = O(N \times 2^{\lg W})$$

- Since the problem size is $O(N + \lg W)$, we see that the cost of the DP solution is exponential in problem size
- In fact, the decision version of the 0–1 Knapsack is NP–complete
- If we change slightly the problem assuming that there is any number of copies of an element of type i, it can be shown that the greedy algorithm is approximate with bound 2, i.e. $V_G \leq 2V^*$ with V_G, V^* the greedy and optimal values
- But in general greedy selection may be far from optimal
- However, polynomial algorithms can be devised with approximation bound as close to 1 as we want

Approximation for the Knapsack Problem

- 0–1 Knapsack is an example of **Integer Programming** problems

$$\max \sum_1^N v_i x_i \text{ subject to } \sum_1^N w_i x_i \leq W,\ x_i \in \{0, 1\}$$

- If we relax the constraint $x_i \in \{0, 1\}$ to $0 \leq x_i \leq 1$, we arrive at a **Linear Programming** (LP) problem
- An LP problem seeks to optimize a linear function subject to linear constraints
- The simplest algorithm for them is the **simplex** method of G. Dantzig, that "walks" through the vertices of the feasible region
- The simplex method is usally very efficient but for some problems it may require a number of steps exponential in problem size

Polynomial Time Approximation Schemes

- In any case, LP problems can be solved in polynomial time
- Intuitively, one can go from real to integer values by "rounding"
- These kind of ideas lead to the concept of **Polynomial Time Approximation Schemes** (PTAS)
- For 0–1 Knapsack we have:
  for any $\epsilon > 0$ there is a polynomial time algorithm that returns a knapsack with value V_ϵ such that

$$V_\epsilon \geq (1 - \epsilon)V^*$$

- To learn more: The Design of Approximation Algorithms. D.P. Williamson, D.B. Shmoys. Cambridge University Press

The Sum Problem

- Given a sequence of positive integers $\Lambda = \{\lambda_1, \ldots, \lambda_N\}$ and another S, the Sum (or Subset Sum) problem is to determine whether S can be decomposed as a sum of integers in Λ
  - I.e., whether there is a bit sequence $b_1, \ldots, b_N$ such that $S = \sum_1^N b_i \lambda_i$
- For $0 \leq i \leq N$, $0 \leq s \leq S$, define $d(i, s)$ to be 0 or 1 depending on whether s can be decomposed as s sum of elements in $\Lambda_i = \{1, \ldots, i\}$
  - If $d(i, s) = 1$ and i is part of the decomposition, then $d(i - 1, s - \lambda_i) = 1$
  - If $d(i, s) = 1$ but i is not part of the decomposition, then $d(i - 1, s) = 1$
  - And if $d(i, s) = 0$, then $d(i - 1, s) =$ and $d(i - 1, s - \lambda_i) = 0$

The Sum Problem

- For $0 \leq i \leq N$, $0 \leq s \leq S$, we thus arrive to

$$d(i, s) = \max\{d(i - 1, s), d(i - 1, s - \lambda_i)\} \tag{1}$$

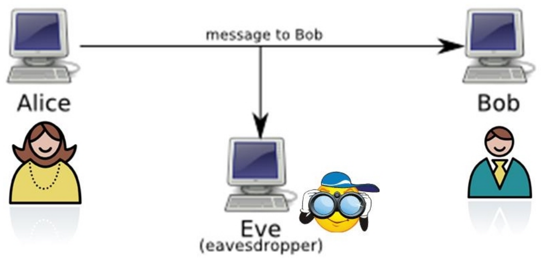
- Obviously $d(i, 0) = 1$ and $d(0, s) = 0$; moreover, if $s < \lambda_i$ $d(i, s) = d(i - 1, s)$
- Thus we can compute $d(N, S)$ by an iterative algorithm but with an exponential cost $O(NS) = O(N2^{\lg S})$

- Again, the general Sum problem is NP–complete
  - The 3–CNF satisfiability problem reduces to the Sum problem (hard), which is thus is NP–complete
  - Moreover the Sum problem reduces to the Subset Partition problem of finding subsets Λ_1, Λ_2 of Λ with equal sums (hard), which is therefore also NP–complete
  - And the Subset Partition problem also reduces to the Knapsack (easy), which is thus NP–complete too
- This property made it the core of the first proposal for Public Key Cryptography

1.3 Public Key Cryptography

Cryptographic Model

- We consider the following general encrypted communication model



- **Base** (plain) text: B (P): text to be encrypted

  Encrypted text C: encrypted version of B

Definitions

- Encryption/decryption keys K, K'; they may coincide
- Encryption algorithm E: we have $C = E(B; K)$

  Decryption algorithm D: we (must) have $B = D(C; K')$
- **Cryptosystem property**: $B = D(E(B; K); K')$
- We need both strong and public encryption/decryption algorithms and long (and secret!) keys

  - As a minimum, the longer the key, the larger the number of brute force tries neede to break it
  - A well known public algorithm is likely to attract many attacks (i.e., free security consultancy services)
  - If it stands them, its security should be good and if it doesn't, better to know as soon as possible
  - On the other hand, there are many examples of failures in "security by obscurity"

Secret Key Cryptography

- The K, K' are only known by A and B (often $K = K'$)
- Standard secret key algoritms
  - DES (Digital Encryption System), designed by IBM and the NAS, the standard from 1975 until about 2005
  - AES (Advanced Encryption Standard): designed by Vincent Rijmen and Joan Daemen, the US NIST standard since 2002
- Main problem: a **physical key interchange is needed** before A and B can interchange encrypted messages
- Clearly undesirable in an Internet world
- Alternative: a **public key** cryptosystem, an idea first proposed by Whitfield Diffie and Martin Hellman around 1975
- Requires different encryption and decryption keys

Public Key Cryptography

- Model: we have an encryption algorithm $E(P; K_E)$ and a decryption one $D(C; K_D)$ over different keys K_E, K_D such that
  - We have a cryptosystem: $D(E(B; K_E); K_D) = B$
  - Obviously it should be very hard to get K_D from E, K_E and D
  - The system should stand **selected text** attacks: since K_E is public we can get as many samples $B, C = E(B; K_E)$ as we want

- If A wants to receive encrypted messages, he makes available software for the E and D algorithms, selects the keys K_E (and K_D), makes public K_E and saves carefully K_D
- If B wants to send a message P to A, he simply uses the public E and K_E to compose $C = E(P; K_E)$
- A gets back P as $P = D(C; K_D)$
- Consequence: anyone can comunicate securely with A ... if a such system indeed can be set up!
- Q: how to do it?

Sum Encryption

- A makes public a (long) list Λ of positive integers
- Assume the list is

```
l_lambda = [575, 436, 1586, 1030, 1921, 569, 721, 1183, 1570]
```

- Since it has 9 elements, P has to be split in 9–bit blocks before getting C
- To encrypt, say, a block `[1 0 1 1 0 0 1 1 1]` , B computes $C = \sum_1^9 b_i \lambda_i$, i.e.,

```
C = 575 * 1  + 436 * 0 + 1586 * 1 + 1030 * 1 + 1921 * 0 +  \
    569 * 0 +  721 * 1 + 1183 * 1 + 1570 * 1
  = 575 + 1586 + 1030 + 721 + 1183 + 1570
  = 6665
```

- A decrypts C by solving the sum problem over C and Λ for each 9–bit block

However ...

- Security base: the sum problem is NP–complete
- Decryption is not computationally feasible if Λ is long enough
- Consequence: nobody can decrypt sum–encrypted messages, not even A!!
- But observe that there is not yet any secret key
- Trick: A will work with a **super–increasing** list $\Lambda' = \{\lambda_1, \ldots, \lambda_N\}$ such that for all $i = 2, \ldots, N$,

$$\lambda_i > \sum_1^{i-1} \lambda_j$$

- The super–increasing sum problem can be solved in linear time

Super–increasing Sum Algorithm

- Assume `s` can be decomposed in terms of the super–increasing sequence `l_lambda` and consider the following pseudocode:

```
def sum_super_crec(l_lambda, s):
    num_lambda= len(l_lambda)
    l_bits = num_lambda * [0]

    for i in range(num_lambda-1, -1, -1):
        if s >= l_lambda[i]:
            l_bits[i] = 1
            s -= l_lambda[i]

    if s == 0:
        return l_bits
```

- The algorithm runs in linear time and is correct:
  - Clearly, if `s > l_lambda[i]` we need `l_lambda[i]` to decompose `s` as the sum of the remaining `l_lambda[j], j < i` is not enough to get `s`
  - And if `s < l_lambda[i]` , it is not part of the decomposition

A Super–increasing Sum Example

- Assume `l_lambda = [2, 5, 9, 21, 45, 103, 215, 450, 946]` and `s= 1643` ; we then have

```
1643 ≥ 946 ⇒ X9=1, and 1643-946=697;
697 ≥ 450 ⇒ X8 = 1, and 697-450 = 247;
247 ≥ 215 ⇒ X7 = 1, and 247-215 = 32;
32 < 103 ⇒ X6 = 0;
32 < 45 ⇒ X5 = 0;
32 ≥ 21 ⇒ X4 = 1, and 32-21 = 11;
11 ≥ ⇒ X3 = 1, and 11-9 = 2;
2 < 5 ⇒ X2 = 0;
2 ≥ 2 ⇒ X1 = 1;
```

- This would yield the plain block `[1 0 1 1 0 0 1 1 1]`

Sum Encryption Trapdoor

- Ralph Merkle's trick: to transform a super increasing sequence Λ_{sc} into another one Λ that is not in an invertible way
- This is an example of a **trap door** that allows easy decryption if you know about it

- We transform $\Lambda_{sc} = \{\lambda_1^s, \ldots, \lambda_N^s\}$ into Λ using
  - A **module** $M > \lambda_1^s + \ldots + \ldots \lambda_N^s$
  - A **multiplier** P prime relative with M, i.e., $lcf(M, P) = 1$, with lcf the largest common factor
- This implies there is a Q such that $PQ\ (M) = 1$, i.e., $PQ = 1 + kM$ for some k
- Then, if $\lambda_i = P * \lambda_i^s\ \%M$, the public key is $\Lambda = \{\lambda_1, \ldots, \lambda_N\}$ and M
- The private key is P that allows to compute Q and Λ_{sc}

Decrypting the Sum Encryption

- We decrypt each encryption C of a block $B = [b_1 \ldots b_N]$ in two steps
  - Compute $C^s = CQ\ \%M$
  - Solve the super–increasing sum problem over C^s and Λ_{sc}
- C^s is just the encryption of B by Λ_{sc}:
  - If $C = \sum_1^N b_j \lambda_j$, then

$$\begin{aligned} C^S &= CQ\ \%M = \left(\sum_1^N b_j \lambda_j Q\right)\ \%M = \left(\sum_1^N b_j \lambda_j^s\right) PQ\ \%M \\ &= \sum_1^N b_j \lambda_j^s \end{aligned}$$

- We obtain the bit sequence b_j solving the super increasing sum over C^S
- Practical implementation with lists of about 250 integers with about 200 bits each, and a 200 bit multiplier P

But . . . It Is Not Safe Enough!!

- It is said that Merkle offered \$100 to any one who broke the first version of Merkle–Hellman cryptosystem

  Adi Shamir got them

- Then Merkle strenghtned the algorithm and offered $1,000 to any one who broke this version

  Ron Rivest got them

- There was no further offer (perhaps $10,000?) so Leon Adleman didn't get it
- Moral of the story: the knapsack cryptosystem of Merkle and Hellman was not used, but it paved the way to practical implementations of public key cryptosystems
- Many such systems have beem proposed, but the de facto standard has been the RSA (Rivest, Shamir, Adleman) system

The RSA Cryptosystem

- Proposed in 1978 by Rivest, Shamir, Adleman
- Based on number theory
- Setting up the cryptosystem:
  - Choose two prime numbers P, Q with about 250 bits
  - Compute $N = PQ$ and $Z = (P-1)(Q-1)$
  - Choose D such that $lcf(D, Z) = 1$ and find E such that $ED = 1\ (Z)$
- Encryption and decryption work as follows
  - To encrypt a plain text split it in 500 bit blocks B and encrypt B as $C = B^E \,\%\, N$
  - To decrypt C compute $B = C^D \,\%\, N$

The RSA Cryptosystem II

- Encryption and decryption are inverses of each other through **Fermat's Little Theorem**: if M and N are relatively primer, then

$$M^{\phi(N)} = 1\ (N)$$

  with $\phi(N)$ Euler's ϕ function, i.e., the number of integers $< N$ and prime relative to it

- If $N = PQ$ and P, Q are relatively primes, $\phi(N) = (P-1)(Q-1) = Z$ and, therefore,

$$\begin{aligned} C^D \,\%\, N &= (B^E)^D \,\%\, N = B^{ED} \,\%\, N = B^{1+kZ} \,\%\, N \\ &= B(B^Z \,\%\, N)^k = B(B^{\phi(N)} \,\%\, N)^k = B \end{aligned}$$

- Public key: E, N
- Private key: D (and N)

RSA's Strenght

- Strenght derived from the difficulty of factoring large numbers into primes
- Everybody knows N but not P or Q: they are the **trapdoor**
- If they were known, we immediately get Z and then apply Euclid algorithm to Z and E to derive D
- RSA takes advantage of about 300 years of free consulting on factorization by some of the leading mathematicians of all times
- But ... if it is very difficult to decompose a number into prime factors, how do we get the large primes P and Q needed in RSA?

How Frequent Are Prime Numbers?

- **Prime Number Theorem:** *The number $\pi(N)$ of primes $\leq N$ is $\simeq \frac{N}{\log N}$*
- Thus, the probability that a random of integer between 1 and N be prime is then approximately

$$\frac{N/\log N}{N/2} = \frac{2}{\log N}$$

- Thus, if we randomly select

$$\frac{\log 2^{1024}}{2} \simeq 710$$

odd integers, there should be a 1024–bit prime among them

Pseudoprimes

- But how we decide which one of the previous 710 integers is prime?
  - Recall that trying to factor them is not an option
- Solution: work with **pseudoprimes**, numbers with a very small probability of not being primes
- It is computationally feasible to check whether P is a pseudoprime to a certain probability level
- The key is again Fermat's Little Theorem: if P is a prime, we have

  $$\phi(P) = P - 1$$

  and, therefore,

  $$a^{P-1} = 1(P)$$

  for any a relatively prime with P

Other Things on RSA

- As of today, RSA is free from brute force attacks; if computer power grows we simply take more bits (2048, 4096) for N
- The RSA attacks are usually based on possible implementation errors
- RSA is quite slow, as we have to work with large integers via software
- Frequent use: encrypt a message using a secret key algorithm and use RSA to encrypt just the key
- Very important use: **digital signatures** and **digital certificates**

RSA Digital Signatures and Certificates I

- Usually RSA is not used to sign an entire message: It would be too time consuming and slow for massive use
- It is used to sign a **hash**, a summary of it

- Basic idea in hashing: to reduce a long item to a much shorter way in a probabilistic one–to–one way
- Formally: there is a function h that transforms objects O in (small) hashes H such that $P(h(O) = h(O')) \simeq 0$ even if O and O' are slightly different
- Secure hash methods: hash methods that ensure that that probability is essentially 0
  - It is unfeasible to get a message with a given hash (i.e., to get a collision)
- Examples: MD5 (Rivest; not secure), SHA–2 (required by US government)

RSA Digital Signatures and Certificates II

- RSA digital signature: A computes first a secure hash value for the message and encrypts it with the secret RSA key
  - The hash algorithm is public and once B gets the message (possibly open), he applies to it the hash algorithm and then "decrypts" the signature using A public key
  - If they coincide, A did indeed signed the message
- **Digital certificate:** a public key signed by a **certification authority** (e.g., VeriSign, FNMT in Spain)
- Signatures and certificates are the basis of Public Key Infrastructures (PKIs): algorithms + protocols + legal context
- RSA is at the core of many/most PKI implementations

1.4 More Examples of DP Algorithms

The Change Problem

- Assume we have work as supermarket cashiers and our clients want change in as few coins as possible
- How can we proceed?
- Simplest idea: give at each step the largest coin smaller than the amount that remains to change

- Example: how to give change of 3,44 euros?
  - Easy: one 2 euro coin, one 1 euro coin, two 20 cent coins, two 2 cent coins
- Have to write down the algorithm but the general idea is **greedy**:
  - We try to minimize **globally** the total number of coins
  - Using **locally** at each step the largest coin possible to minimize the amount still to change

The Greedy Change Algorithm

- We will work with an ordered list of coin values, say

  ```
  1, 2, 5, 10, 20, 50, 100, 200
  ```

  and save the number of coins of each type used in a dict

  ```
  def change(c):
      """
      """
      assert c >= 0, "change for positive amounts only"

      l_coin_values = [1, 2, 5, 10, 20, 50, 100, 200]
      d_change = {}

      for coin in sorted(l_coin_values)[ : : -1]:
          d_change[ coin ] = c // coin
          c = c % coin

      return d_change
  ```

Does It Work?

- At first sight yes, but ...
- Try it to give change of 7 maravedis with coin values

  ```
  1, 3, 4, 5
  ```

- What is the answer of the algorithm?
  - Correct answer: just 2 coins, one of 4 maravedis and one of 3
- Recall that this often happens with greedy algorithms
  - They are very natural but may give wrong results!!
- Let's try a dynamic programming approach

Back To Giving Change

- We can apply a reasoning similar to that in the knapsack problem to find the minimum number of change coins
- Assume we want to change an amount C with $v_1, \ldots, v_N$ coin denominations
- Let $n(i, c)$ be the minimum number of coins to change an amount c using only the first i coins
- Depending on whether coin i enters the change we have the following equations

$$\begin{aligned} n(i, c) &= n(i-1, c) \text{ if coin } i \text{ doesn't enter the change,} \\ &= 1 + n(i, c - v_i) \text{ if coin } i \text{ enters the change} \end{aligned}$$

The DP Change Algorithm

- Therefore , we arrive at

$$n(i, c) = \min\{n(i-1, c), 1 + n(i, c - v_i)\}$$

- Again, we simply have to fill the DP matrix with a cost

$$O(N \times C) = O(N \times 2^{\lg C})$$

- And, again, the change problem is NP–complete
- But there is a linear complexity algorithm for super–increasing coin systems

Superincreasing Coin Systems

- Sensible coin systems are super–increasing:
  - For the euro, the list (in cents) would be
    `[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, ..., 100000]`
- The previous greedy algorithm can be used to give change in linear time
  - We assume coin values in `l_coin_values` and use a dict `d_change` to store the change of `c` to be given
  - Recall the code:

```
d_change = {}

for coin in sorted(l_coin_values)[ : : -1]:
    d_change[ coin ] = c // coin
    c = c % coin

return d_change
```

Efficient Matrix Multiplication

- Given N matrices $A_1, \ldots, A_N$ with sizes $c_{i-1} \times c_i$, $0 \leq i \leq N$, the order in which we compute $A_1 \times \ldots \times A_N$ doesn't matter mathematically for the final solution
- But it affects greatly the computation cost
- Recall that the cost of computing $A_i A_{i+1}$ is $c_{i-1} \times c_i \times c_{i+1}$ number multiplications
- Consider 4 matrices A_1, A_2, A_3, A_4 with respective sizes 50×10, 10×40, 40×30 and 30×5
  - The cost of the order $A_1(A_2(A_3 A_4))$ is then 10,500 products
  - But the cost of the order $((A_1 A_2) A_3) A_4$ is then 87,500 products
- We want to find how to order matrix multiplication so that the overall cost is minimum

EOS for Matrix Multiplication Ordering

- Let $m_{L,R}$ be minimum number of number products needed to compute $A_L \ldots A_R$
  - We have $m_{i,i+1} = c_{i-1} \times c_i \times c_{i+1}$
  - We set $m_{i,i} = m_{i,i-1} = 0$ and we want to compute $m_{1,N}$
- To arrive at the EOS property, assume that the optimal ordering is $(A_L \ldots A_j)(A_{j+1} \times A_R)$; then

$$m_{L,R} = c_{L-1} \times c_j \times c_R + m_{L,j} + m_{j+1,R}$$

- We don't know j but we can simply compute all the above options and take the minimum, i.e.

$$m_{LR} = \min\{c_{L-1} \times c_j \times c_R + m_{Lj} + m_{j+1R} : L \leq j \leq R-1\} \qquad (2)$$

DP Algorithm

- The EOS property (2) suggests an iterative bottom up algorithm

  - Compute first the $m_{i,i+1}$ for $1 \leq i \leq N-1$ values
  - Compute then the $m_{i,i+2}$ for $1 \leq i \leq N-2$ values and keep on going
  - Until we arrive at $m_{1,N}$

- Since we have to compute $O(N^2)$ values m_{ij} for $2 \leq j \leq N$ and $1 \leq i < j$ at a cost of $O(j-i) = O(N)$ for each term, the overall cost will be $O(N^3)$
- Exercise: derive the precise value
- We can store the $m_{L,R}$ values in a triangular matrix $a_{i,j} = m_{j,N-i}$, i.e., the L values define the columns and the R values define the rows in inverse order

An Example

- Consider again the matrices A_1, A_2, A_3, A_4 with respective sizes 50×10, 10×40, 40×30 and 30×5

Optimal Binary Search Trees

- Assume we have N ordered keys $k_1 \leq k_2 \ldots \leq k_N$ with probabilities p_i and let T be a binary search tree (BST) built from the keys
- Since the number of comparisons needed to find k_j in T is $1 + prof_T(k_j)$, with $prof_T(k)$ the depth of k in T, the average cost of searching T is

$$c(T) = \sum_j p_j \left(1 + prof_T(k_j)\right) = 1 + \sum_j p_j \, prof_T(k_j)$$

- Notice that the actual key values are irrelevant, as we could take them as $1, 2, \ldots, N$; the important information are the probabilities p_i

- There are two obvious "greedy" algorithms to find an optimal T^*
  - Build a tree as shallow as possible placing the keys in a quasicomplete binary tree
  - Try to put the most frequent keys as high as possible by inserting them in descending probability order

A First Example

- Consider the keys a to g with probabilities $[0.22, 0.18, 0.20, 0.05, 0.25, 0.02, 0.08]$
- The first greedy algorithm gives a tree T_1 with the keys inserted as $\{d, b, f, a, c, e, g\}$ with $c(T_1) = 2.7$
- The second greedy algorithm gives a tree T_2 with the keys inserted as $\{e, a, c, b, g, d, f\}$ with $c(T_2) = 2.3$
- But the tree T_3 with the keys inserted as $\{c, a, e, b, d, g, f\}$ has a lower cost of $c(T_3) = 2.15$
- How can we found in general such an optimal T?
- We know by now that to develop a DP algorithm we must introduce somehow a matrix that captures partial optimum costs
- Let T_{LR} be an optimal BST with keys $\{k_L, \dots, k_R\}$ and probabilities $\{p_L, \dots, p_R\}$, and let $c_{LR} = c(T_{LR})$

The DP Approach

- We obviously want to find T_{1N} and its optimum cost c_{1N}
- Assume that k_j is the key at the root of T_{LR} and let T_l and T_r be its left and right subtrees; we have

$$\begin{aligned}
c_{LR} &= c(T_{LR}) = \sum_{L}^{R} p_i\{1 + \ prof_{T_{LR}}(k_i)\} \\
&= \sum_{L}^{R} p_i + \sum_{L}^{j-1} p_i\, prof_{T_{LR}}(k_i) + \sum_{j+1}^{R} p_i\, prof_{T_{LR}}(k_i) \\
&= \sum_{L}^{R} p_i + \sum_{L}^{j-1} p_i(1 + prof_{T_l}(k_i)) + \sum_{j+1}^{R} p_i(1 + prof_{T_r}(k_i)) \\
&= \sum_{L}^{R} p_i + c(T_l) + c(T_r)
\end{aligned}$$

The DP Approach II

- But for T_{LR} to be optimal, T_l and T_r must also be optimal, i.e., we have $c_{LR} = c_{Lj-1} + c_{j+1R} + \sum_L^R p_i$
- Again, we do not know the optimal k_j but we can simply write

$$c_{LR} = \sum_{L}^{R} p_i + \min\{c_{Lj-1} + c_{j+1R} : L \leq j \leq R\}$$

- Defining $c_{ii} = p_i$, this naturally suggests a bottom–up DP algorithm
- As in matrix ordering, we can store the c_{LR} values in a triangular matrix $a_{ij} = c_{j,N-i}$ i.e., the L values define the columns and the R values define the rows in inverse order

An Example

- Consider the keys a, b, c, d with probabilities `0.22, 0.18, 0.25, 0.35`

Editing Strings

- Assume we have two strings and want to edit, i.e., transform one string `s_1` into another `s_2` according to the following allowable operations
  - Change one character in either string
  - Insert a character in either string
  - Remove a character from either string
- Notice that removing a remove a character from string `s_1` is equivalent to adding a character to string `s_2`
- Also we can keep one of the strings fixed and do all the change/add/remove operations on the other

The Edit Distance

- The **Edit Distance** `d(s_1, s_2)` between two strings `s_1` into another `s_2` is the minimum number of edit operations that we have to make to turn, say, `s_2` into `s_1`
- For instance, the edit distance between `unnecessarily` and `unescessaraly` is 3:

```
unne cessarily
  a r      c
un escessaraly
```

  as we just add to the second string an `'n'` , remove an `'s'` and change `'a'` into `'i'`
- The edit distance can also be used for **approximate string searches**: given a string `s` find another `t` in a string list so that their edit distance is minimal

A Dynamic Programming Solution

- Given the full strings S and T with M and N characters respectively, consider the substrings
$$S_i = [s_1, \ldots, s_i], \quad T_j = [t_1, \ldots, t_j]$$
- If $d_{i,j}$ is the edit distance between S_i and T_j, we want to find $d_{M,N} = \text{dist}(S, T)$
- Observe that if $s_i = t_j$, then $d_{i,j} = d_{i-1,j-1}$

- And if $s_i \neq t_j$ we have three options
  - Change t_j into s_i; then $d_{i,j} = 1 + d_{i-1,j-1}$
  - Remove t_j from T_j; then $d_{i,j} = 1 + d_{i,j-1}$
  - Remove s_i from S_i; then $d_{i,j} = 1 + d_{i-1,j}$

Filling The DP Matrix

- We thus arrive to the following equations for the Edit Distance problem

$$\begin{aligned} d_{i,j} &= d_{i-1,j-1} \text{ if } s_i = t_j; \\ &= 1 + \min\{d_{i-1,j-1}, d_{i,j-1}, d_{i-1,j}\} \text{ if } s_i \neq t_j \end{aligned}$$

  which result in an easy to apply algorithm

- Example: find the edit distance between `biscuit` and `suitcase`
- The cost is clearly $O(M \times N)$, no longer NP–complete but, still, costly
- And an initial memory cost is also $O(M \times N)$, just awful, as problem size is $O(M + N)$
  - But, in fact, quite easy to alleviate if we only care about $d_{M,N}$

Alignment with general penalties

- Sequence alignment with general scoring penalties is very important in DNA sequence comparison
- We consider different penalties for switching s_i or t_j into the other or for inserting one or the other
  - For switching we assume a penalty of α_{s_i,t_j}
  - For inserting (or removing) we assume a character independent penalty of δ
- We know talk about **alignment costs** $c_{m,n}$ and the new DP penalty equations are

$$c_{i,j} = \min\left\{\alpha_{s_i,t_j} + c_{i-1,j-1},\ \ \delta + c_{i,j-1},\ \ \delta + c_{i-1,j}\right\}$$

  where we assume $\alpha_{c,c} = 0$

- This (plus quite a bit of Biochemistry) ends up in the **Needleman–Wunsch** sequence similarity measure

The Longest Common Substring

- Given again strings S, T, we want to find the longest (non necessarily consecutive) common substring (LCS) to both
- As before, let first $\ell_{i,j}$ be the length of the LCS between S_i and T_j. We have now:

$$\begin{aligned} \ell_{i,j} &= 1 + \ell_{i-1,j-1} \ \text{ if } \ s_i = t_j; \\ &= \max\{\ell_{i,j-1}, \ell_{i-1,j}\} \ \text{ if } \ s_i \neq t_j \end{aligned}$$

  - Again this results in an easy to apply algorithm with cost $O(M \times N)$
  - Example: find the LCS between `biscuit` and `suitcase`
- Good (and easy) exercise: write down a Python function to find the length of the LCS
- Very good exercise: modify the previous Python function so that it gives one of the possible LCS

Other Problems With DP Solutions

- There is a large number of problems, essentially different and often useful in practice, that can be solved via DP algorithms
- We list some of them:
  - Cut a rod of a lenght L in pieces that get as much money as possible when sold
  - Finding the longest palindromic substring of a given string
  - Fit Least Squares segments to data
  - Speech recognition through the Viterbi algorithm
- And many more!

2 Recursive Algorithms

2.1 Matrix Multiplication

General Approach

- In general, recursive algorithm design to solve a problem P works in three steps:
  - Decompose P in several subproblems with balanced sizes
  - Solve each subproblem recursively
  - Combine these solutions in a solution for P
- Classical examples are recursive sorting algorithms such as **MergeSort** and **QuickSort**, searching algorithms such as **Binary Search** or many tree (and graph) algorithms
- In general Recursion is a powerful design tool
  - There is no other way to devise algorithms for problems such as the Hanoi Towers puzzle
- But recursive algorithms can be very inefficient
  - Straight recursive solutions to dynamic programming problems
- However, they can also yield efficient algorithmic approaches, provided recursion's hidden costs are avoided

Recursive Matrix Multiplication I

- If A, B are $N \times N$ matrices, the standard algorithm to multiply $C = A \times B$ has a cost N^3
- If we assume N even, breaking A, B and C in $N/2 \times N/2$ blocks A_{ij}, B_{ij}, C_{ij} and computing

  $$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$$

  suggests that one $N \times N$ multiplication can be reduced to eight $N/2 \times N/2$ multiplications
- In turn, this suggests a simple recursive matrix multiplication algorithm, RMM

- Assuming N even, its cost $T_{RMM}(N)$ verifies

$$T_{RMM}(N) = 8T_{RMM}\left(\frac{N}{2}\right)$$

Recursive Matrix Multiplication II

- Unwinding $T_{RMM}(N)$ for $N = 2^K$, we get

$$T_{RMM}(N) = 8T_{RMM}\left(\frac{N}{2}\right) = 8^2T_{RMM}\left(\frac{N}{2^2}\right) = \ldots = 8^KT_{RMM}\left(\frac{N}{2^K}\right)$$

  and since $T_{RMM}(1) = 1$, we get

$$T_{RMM}(N) = 8^K = (2^K)^3 = N^3$$

- Thus, there is no gain over the standard algorithm

Strassen's Algorithm

- But if we can decompose one $N \times N$ multiplication into M multiplications of $N/2 \times N/2$ matrices, the recurrence becomes

$$T_{RMM}(N) = M \times T_{RMM}\left(\frac{N}{2}\right)$$

  that unwinds as

$$T_{RMM}(N) = M^K = (2^{\lg M})^K = (2^K)^{\lg M} = N^{\lg M}$$

- Thus, we improve on the classical algorithm if $M < 8$
- Strassen's algorithm achieves this with $M = 7$
- It uses seven matrices $M_1, \ldots, M_7$ and formulae such as $C_{11} = M_1 + M_4 - M_5 + M_7$, $M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$ and so on
- See en.wikipedia.org/wiki/Strassen_algorithm

Recursive Number Multiplication

- The same ideas can be used for number multiplication: if A has an even number N of bits we can write it as

$$\begin{aligned} A &= a_{N-1}2^{N-1} + \ldots + a_{N/2}2^{N/2} + a_{N/2-1}2^{N/2-1} + \ldots + a_0 \\ &= \left(a_{N-1}2^{N/2-1} + \ldots + a_{N/2}\right)2^{N/2} + a_{N/2-1}2^{N/2-1} + \ldots + a_0 \\ &= A_1 2^{N/2} + A_0 \end{aligned}$$

- If AB also has N bits and we decompose A and B as above, we have

$$AB = A_1B_1 2^{N-1} + (A_1B_0 + A_0B_1)2^{N/2-1} + A_0B_0,$$

which suggests a recursive number multiplication algorithm RNM

Analyzing Recursive Multiplication

- Its cost function in terms of binary multiplications verifies

$$T_{RNM}(N) = 4 \times T_{RNM}\left(\frac{N}{2}\right) \text{ and } T_{RNM}(1) = 1$$

- If $N = 2^K$, unwinding $T_{RNM}(N)$ gives

$$T_{RNM}(N) = 4^K \times T_{RNM}\left(\frac{N}{2^K}\right) = 4^K = (2^K)^2 = N^2$$

- We thus get the same cost of the standard number multiplication algorithm
- But we can easily improve on this

Karatsuba's Algorithm

- Notice that we can also write

$$AB = A_1B_1 2^{N-1} + \left((A_1 + A_0)(B_1 + B_0) - A_1B_1 - A_0B_0\right)2^{N/2} + A_0B_0,$$

requiring only 3 multiplications of $N/2$–bit numbers (but 4 sums)

- If we build a recursive algorithm RNM_K on this, its cost function verifies $T_{RNM_K}(N) = 3 \times T_{RNM_K}\left(\frac{N}{2}\right)$ and $T_{RNM_K}(1) = 1$

- For $N = 2^K$, it unwinds as

$$T_{RNM_K}(N) = 3^K = (2^K)^{\lg 3} = N^{\lg 3}$$

- This plus a clever implementation constitute Karatsuba's method for (large) number multiplication, used for instance in Public Key Cryptography

2.2 The Fast Fourier Transform

Polynomial Multiplication

- The standard algorithm to multiply to polynomials of degree $N-1$ (i.e., with N coefficients) requires N^2 single digit multiplications

- Moreover, if we have if $P(X) = 3X + 5$ and $Q(X) = 4X + 2$, then

$$P(X)Q(X) = 12X^2 + (20 + 6)X + 10,$$

  that requires 4 products

- This coincides with the cost of multiplying two numbers, as both algorithms essentially coincide

- First, notice that we have

$$245 = 2 \times 10^2 + 4 \times 10 + 2 = P(10),$$

  where we have

$$P(X) = 2X^2 + 4X + 2$$

Polynomial and Number Multiplication

- Thus, if, say, $35 = 3 \times 10 + 5$ and $42 = 4 \times 10 + 2$, then

$$35 = P(10), 42 = Q(10)$$

  and as a consequence

$$35 \times 42 = 1470 = 12 \times 100 + 26 \times 10 + 10 = PQ(10)$$

- Thus better algorithms for polynomial multiplication will result in better algorithms for number multiplication
- We next explore this

Lagrange Representation

- **Theorem (Lagrange).** *The coefficients of a polynomial P of degree $N-1$ are uniquely determined by the values $p_k = P(x_k)$ at N different points $x_0, \ldots, x_{N-1}$*
- **Proof sketch:** Assume $P(X) = \sum_0^{N-1} c_k X^k$;
  - We may look at the coefficients c_k as the unknowns of a linear system whose k–th equation is

    $$p_k = c_0 + c_1 x_k + c_2 (x_k)^2 + \ldots + c_{N-1} (x_k)^{N-1}$$

  - We have a system with N equations and N unknowns, that has a unique solution if the coefficients' determinant is not zero
  - But the coefficient matrix is a Vandermonde matrix, whose determinant is given by

    $$\prod_{0 \leq i < j \leq N-1} (x_i - x_j),$$

    which will be nonzero if we have $x_i \neq x_j$

Alternative Polynomial Multiplication

- Lagrange's representation can be used to define an alternative polynomial multiplication algorithm
- Assume that the polynomial PQ has degree $N-1$ and we are given N different points $x_0, \ldots, x_{N-1}$
- We can then obtain the coefficients of PQ from those of P and Q as follows:
  - Compute the values $p_k = P(x_k)$ and $q_k = Q(x_k)$, $0 \leq k \leq N-1$
  - Compute the values of PQ at the x_k as $PQ(x_k) = P(x_k)Q(x_k)$
  - Obtain back the coefficients of PQ from the $PQ(x_k)$ values using Lagrange's representation or other method

The Cost of Alternative Polynomial Multiplication

- We estimate the cost of this algorithm for general $x_0, \ldots, x_{N-1}$
- The first step has a cost of $N \times O(N) = O(N^2)$ if we use an efficient algorithm such as **Hörner's rule** to evaluate the polynomials P, Q at the N points x_k
  - Hörner's rule has cost $O(N)$
- The cost of the second step is clearly $O(N)$
- The third step has a cost $O(N^3)$ as we have to solve a $N \times N$ linear system
- The $O(N^3)$ cost for general x_k is much worse than that of the standard algorithm
- But this will change a lot for the values $x_k = e^{\frac{2\pi ik}{N}} = \left(e^{\frac{2\pi i}{N}}\right)^k$

Euler's Formula

- We have $e^{i\theta} = \cos\theta + i\sin\theta$, with $i = \sqrt{-1}$
- The proof applies Taylor's formulae for the exponential, sine and cosine functions, and that $i^{2k} = (i^2)^k = (-1)^k$ and $i^{2k+1} = i(-1)^k$:

$$\begin{aligned} e^{i\theta} &= \sum_n \frac{(i\theta)^n}{n!} = \sum_n \frac{(i\theta)^{2n}}{2n!} + \sum_n \frac{(i\theta)^{2n+1}}{(2n+1)!} \\ &= \sum_n \frac{(-1)^n\theta^{2n}}{2n!} + i\sum_n \frac{(-1)^n\theta^{2n+1}}{(2n+1)!} \\ &= \cos\theta + i\sin\theta \end{aligned}$$

- The standard properties of the real exponential also hold for arguments Ξ, Υ that can be added and multiplied such as complex numbers and matrices:

$$e^{\Xi+\Upsilon} = e^{\Xi}e^{\Upsilon}; \;\; \left(e^{\Xi}\right)^m = e^{m\Xi} \;\text{ for integers } m \geq 0$$

The Discrete Fourier Transform

- Given a table $[c_0, c_1, \ldots, c_{N-1}]$, its **Discrete Fourier Transform** (DFT) is the table $[p_0, p_1, \ldots, p_{N-1}]$, with

$$p_j = \sum_{k=0}^{N-1} c_k \, e^{2\pi i \frac{kj}{N}} = \sum_{k=0}^{N-1} c_k \, \left(e^{2\pi i \frac{j}{N}}\right)^k$$

- In other words, if we consider $P(X) = \sum_{k=0}^{N-1} c_k X^k$, then

$$p_j = P\left(e^{2\pi i \frac{j}{N}}\right)$$

- For general points, the cost of the DFT is $O(N^2)$
- However, when $N = 2^K$, there is a fast recursive algorithm to compute the DFT over the points $e^{2\pi i \frac{j}{2^K}}$, $0 \leq j \leq 2^K - 1$

The Fast Fourier Transform (FFT)

- If P has coefficients $c_0, \ldots, c_{2^K-1}$ (in what follows we will say that P has **order** K), we have for $p_j = P\left(e^{2\pi i \frac{j}{2^K}}\right)$:

$$\begin{aligned}
p_j &= \sum_{k=0}^{2^K-1} c_k \left(e^{2\pi i \frac{j}{2^K}}\right)^k = \sum_{k=0}^{2^K-1} c_k e^{2\pi i \frac{kj}{2^K}} \\
&= \sum_{k=0}^{2^{K-1}-1} c_{2k} e^{2\pi i \frac{2kj}{2^K}} + \sum_{k=0}^{2^{K-1}-1} c_{2k+1} e^{2\pi i \frac{(2k+1)j}{2^K}} \\
&= \sum_{k=0}^{2^{K-1}-1} c_{2k} e^{2\pi i \frac{kj}{2^{K-1}}} + \sum_{k=0}^{2^{K-1}-1} c_{2k+1} e^{2\pi i \frac{kj}{2^{K-1}}} e^{2\pi i \frac{j}{2^K}} \\
&= \sum_{k=0}^{2^{K-1}-1} c_{2k} \left(e^{2\pi i \frac{j}{2^{K-1}}}\right)^k + e^{2\pi i \frac{j}{2^K}} \sum_{k=0}^{2^{K-1}-1} c_{2k+1} \left(e^{2\pi i \frac{j}{2^{K-1}}}\right)^k
\end{aligned}$$

The Fast Fourier Transform II

- Denoting by P_e and P_o the polynomials of order $K-1$ with coefficients c_{2k} and c_{2k+1}, with $0 \leq k \leq 2^{K-1} - 1$, we have shown

$$p_j = P\left(e^{2\pi i \frac{j}{2^K}}\right) = P_e\left(e^{2\pi i \frac{j}{2^{K-1}}}\right) + e^{2\pi i \frac{j}{2^K}} P_o\left(e^{2\pi i \frac{j}{2^{K-1}}}\right) \tag{3}$$

that is, $p_j = p_j^e + e^{2\pi i \frac{j}{2^K}} \, p_j^o$

- So it would seem that we have reduced the DFT of a polynomial P of order K to that of the two polynomials P_e, P_o, of order $K-1$
- However we use 2^K points in the DFT of P but only have 2^{K-1} in these of P_e, P_o
- We have to do something about p^e_j, p^o_j for $2^{K-1} \le j \le 2^K - 1$

The Fast Fourier Transform III

- Notice that for these j we have

$$j = 2^{K-1} + \ell, \;\; 0 \le \ell \le 2^{K-1}$$

and for $0 \le k \le 2^{K-1}$ we have

$$e^{2\pi i \frac{kj}{2^{K-1}}} = e^{2\pi i \frac{k(2^{K-1}+\ell)}{2^{K-1}}} = e^{2\pi i \left(k + \frac{k\ell}{2^{K-1}}\right)} = e^{2\pi i k} e^{2\pi i \frac{k\ell}{2^{K-1}}} = e^{2\pi i \frac{k\ell}{2^{K-1}}}$$

for $e^{2\pi i k} = \cos 2\pi k + i \sin 2\pi k = 1$

- As a consequence, for $j = 2^{K-1} + \ell$, $0 \le \ell \le 2^{K-1}$, we have

$$P_e\left(e^{2\pi i \frac{j}{2^{K-1}}}\right) = P_e\left(e^{2\pi i \frac{\ell}{2^{K-1}}}\right);\; P_o\left(e^{2\pi i \frac{j}{2^{K-1}}}\right) = P_o\left(e^{2\pi i \frac{\ell}{2^{K-1}}}\right)$$

that is, $p^e_j = p^e_{j\%2^{K-1}}$ and $p^o_j = p^o_{j\%2^{K-1}}$

The Fast Fourier Transform IV

- Therefore, since $\ell = j\%2^{K-1}$, we can rewrite (3) as

$$\begin{aligned} DFT(P)_j = p_j &= P_e\left(e^{2\pi i \frac{j\%2^{K-1}}{2^{K-1}}}\right) + e^{2\pi i \frac{j}{2^K}} P_o\left(e^{2\pi i \frac{j\%2^{K-1}}{2^{K-1}}}\right) \\ &= p^e_{j\%2^{K-1}} + e^{2\pi i \frac{j}{2^K}} p^o_{j\%2^{K-1}} \\ &= DFT(P_e)_{j\%2^{K-1}} + e^{2\pi i \frac{j}{2^K}} DFT(P_o)_{j\%2^{K-1}}, \end{aligned} \quad (4)$$

which holds for all $j = 0, \ldots, 2^K - 1$

- This allows us to derive the **Fast Fourier Transform**, an efficient recursive algorithm to compute the DFT

The FFT Algorithm

- Formula (4) leads to the following pseudocode:

```
def FFT(c, K):
  """c: table with 2**K coefficients"""
  if K==0: return c
  else:
    c_e = even(c); c_o = odd(c) #coeffs in even and odd positions
    f_e = FFT(c_e, K-1); f_o = FFT(c_o, K-1)
    for j = 0, ..., 2**K-1:
      f[j] = f_e[j % 2**(K-1)] +
             e**(2*pi*i * j/2**K} * f_o[j % 2**(K-1)]
  return f
```

- If $n_{FFT}(N)$ is the number of complex multiplications in FFT, we have for $N = 2^K$

$$\begin{aligned} n_{FFT}(N) &= N + 2n_{FFT}(N/2) = 2N + 2^2 n_{FFT}(N/2^2) = \dots \\ &= K\ N + 2^K n_{FFT}(N/2^K) = K\ N = N \lg N \end{aligned}$$

Back to Multiplying Polynomials

- The cost of the FFT is much better than the previous quadratic cost
- Getting back to our previous three step algorithm to multiply polynomials, computing the values p_k, q_k, $0 \leq j \leq 2^K - 1$, has now a $O(N \lg N)$ cost
- The DFT can also be used in the third step, as it essentially inverts itself: if $[p_j] = DFT([c_j])$, we have

$$[c_0, \dots, c_{2^K-1}] = \frac{1}{2^K} \overline{DFT[\overline{p}_0, \dots, \overline{p}_{2^K-1}]}$$

- It follows that we can multiply two polynomials of degree N with cost $O(N \lg N)$

Multiplying Polynomials Using the FFT

- **Example:** $p(X) = 1 + 2X + X^2, Q(X) = 1 + X$

Back to Multiplying Numbers

- We saw how the standard algorithm for number multiplication can be reduced to that of polynomial multiplication, to which we have to add handling the carries
- It is thus natural to take advantage of the FFT to do this
- This is at the heart of the Schönhage–Strassen algorithm for the multiplication of large numbers, that has a cost $O(N \lg N \lg \lg N)$
- However, in practice it has a larger cost than other multiplication methods for moderately large numbers
- The GNU Multiple Precision Arithmetic Library (GNU MP) uses it for very large numbers (above about 50,000 decimal digits), but other methods such as Karatsuba or Toom–Cook (Toom 3) for smaller ones

2.3 Signal Processing and the DFT

Signal Processing and the DFT

- Euler's formula gives basic cosine/sine signal $e^{2\pi i\omega} = \cos(2\pi\omega) + i\sin(2\pi\omega)$ with frequency 1 and amplitude 1
- We can also use it to get a "pure" signal s with frequency k and amplitude a_k

$$s_k(\omega) = a_k\cos(2\pi k\omega) + ia_k\sin(2\pi k\omega) = a_k e^{2\pi ik\omega} = a_k(e^{2\pi i\omega})^k$$

- This is just the evaluation of the polynomial $P_k(X) = a_k X^k$ at the point $X = e^{2\pi i\omega}$
- If we sample $s_k(\omega)$ at the 2^K points $j/2^K$, $0 \leq j \leq 2^K - 1$, the table

$$s[\,] = [a_k e^{2\pi i k \frac{0}{2^K}}, \ldots, a_k e^{2\pi i k \frac{j}{2^K}}, \ldots, a_k e^{2\pi i k \frac{2^K-1}{2^K}}],$$

contains the DFT of the coefficient table $a[\,]$ of $P_k(X)$ with zeros at all places but the k–th one, where we have a_k

DFT Filtering of Single Frequencies I

- This suggests to process a signal s exploiting its DFT $s[\,]$
- Assume we have $s[\,] = DFT(a[\,])$;
applying the inversion formula for the DT, we can now get back $s[\,]$ as

$$s[\,] = DFT\,(a[\,]) = DFT\left(\frac{1}{2^K}\overline{DFT\left(\overline{s[\,]}\right)}\right) = \frac{1}{2^K}DFT\left(\overline{DFT\left(\overline{s[\,]}\right)}\right)$$

- If we want to filter, i.e., to remove the k frequency signal, we can simply replace by 0 the a_k coefficient in $a[\,] = DFT\left(\overline{s[\,]}\right)$

  We then get the table $a'[\,] = [0, \ldots, 0]$ whose DFT $s'[\,] = DFT(a'[\,])$ is then 0

DFT Filtering of Single Frequencies II

- In other words, we can remove the k–th frequency component of a signal s by
  1. Computing the DFT $\alpha[\,]$ of its conjugate signal $\overline{s[\,]}$
  2. Setting the k–th component of $\alpha[\,]$ to 0, and
  3. Computing the DFT of the conjugate of the new $\alpha[\,]$ and dividing it by 2^K
- Since the DFT is a linear transformation, the preceding makes possible to simultaneously removing a frequency subset
- But we have to take care of the aliasing and frequence folding effects

Folding

- If we all frequencies to 0 except frequency k that we set to 1, take the real part of its DFT (i.e., a cosinusoid) and compute back its DFT, we get a "new" frequency at $2^K - k$ with the same coefficient
  - This effect is often called frequency folding
- The reason is that for all j

$$\cos\left(2\pi\frac{jk}{2^K}\right) = \cos\left(2\pi\frac{j(k-2^K)}{2^K}\right) = \cos\left(-2\pi\frac{j(k-2^K)}{2^K}\right) = \cos\left(2\pi\frac{j(2^K-k)}{2^K}\right)$$

- The same happens with sinusoids but now with the new coefficient at $2^K - k$ being the negative of that at k, for now

$$\sin\left(2\pi\frac{jk}{2^K}\right) = -\sin\left(2\pi\frac{j(2^K-k)}{2^K}\right)$$

- Thus, if we want to remove a component at frequency k we must also do so with the frequency $2^K - k$

DFT Filtering of (Discrete) Signals I

- Assume we have a sum of basic signals of frequencies k and amplitudes a_k, i.e.,

$$\phi(\omega) = \sum_{j=0}^{2^K-1} a_k e^{2\pi i k\omega}$$

- Sampling ϕ at the points $\omega_j = j/2^K$, $0 \leq j \leq 2^K - 1$, the table $\phi[\,]$ given by $\phi_j = \phi(j/2^K)$ is just the DFT of the amplitudes' table

$$a[\,] = [a_0, \dots, a_j, \dots, a_{2^K-1}]$$

- Thus, if we want to remove from ϕ some frequency subset $k_1, \dots, k_M$, we simply have to
  1. Compute $a[\,] = DFT\left(\overline{\phi[\,]}\right)$ and set to 0 the coefficients k_m of $a[\,]$ and
  2. Compute then $\frac{1}{2^K}DFT\left(\overline{a'[\,]}\right)$ of the resulting table $a'[\,]$

DFT Filtering of (Discrete) Signals II

- Notice though that if $a[\ ]$ is real and $\phi = DFT(a)$, $\phi_j = \overline{\phi_{2^K - j}}$, $1 \leq j \leq 2^{K-1} - 1$
- Thus there are essentially 2^{K-1} independent frequencies
- When the frequencies removed are in the range $\{1, \ldots, K_1\}$, we talk about a **low pass** filter; when we remove the range $\{K_2, \ldots, 2^{K-1}\}$ we talk about a **high pass** filter and removing the range $\{K_1, \ldots, K_2\}$ yields a **band pass** filter
- The above can be extended from the previous particular functions ϕ to general, "reasonable" periodic functions f through the theory of Fourier series
- In fact it is then possible to expand such an $f(\omega)$ as the trigonometric series

$$f(\omega) = \sum_{0}^{\infty} a_k \cos(2\pi k\omega) = \sum_{-\infty}^{\infty} a_k e^{2\pi i k\omega}$$

2.4 The Selection Problem

The Selection Problem

- Given a table T of size N and an index k, return the element that occupies the k–th position in a sorting of T
- Simplest solution: sort T and return $T[k]$
- Its cost is $O(N \log N)$, too large if we have to find the maximum $k = N$ or the minimum $k = 1$, for which standard algorithms have cost $N - 1$
- An **adversary analysis** can be used to show that this gives a lower bound: any algorithm must have a cost $\Omega(N)$
- An important case is the computation of the **median** with $k = \lfloor N/2 \rfloor$
- Q: can we solve the general selection problem in linear time?

Tweaking QuickSort

- One idea is to somehow mix QuickSort and Binary Search; assuming $P = 1$, $U = N$,

  - Compute the approximate middle index `m` returned by the `split` routine of QuickSort and return it if `k == m`
  - If `k<m` look for the k–th element in the left subtable
  - If `k>m` look for the $(k - M)$–th element in the right subtable

- The known analysis of QuickSort hints at some results:

  - The algorithm will have cost $\Omega(N)$, since the cost of `split` is $N - 1$
  - If we use the first element as the pivot in `split` and want to find the last element, the cost will be $\Theta(N^2)$
  - But there is only one recursive call so we should get a linear–cost average case

Quick Select

- The pseudocode for positions k between 1 and N (i.e., if `k==1` return `p`) would be the following:

```
def qselect(t, p, u, k):
    if k > u-p+1 or k < 1:
        return

    m = split(t, p, u)

    if k == u-p+1:
        return m
    elif k < m-p+1:
        return qselect(t, p, m-1, k)
    else: #k > m-p+1:
        return qselect(t, m+1, u, k-(m-p+1))
```

Average Case of Quick Select I

- **Proposition.** *If $A_{QSel}(N, k)$ denotes the average number of key comparisons (KCs) that $QSelect$ does, we have $A_{QSel}(N, k) \leq 4N$ for all k*

- **Proof sketch:** We retrace and adapt the argument for the average case of Quick-Sort Assuming $P = 1, U = N$, we have

$$n_{QSel}(T, 1, N, k) = N - 1 + \nu,$$

  with $N - 1 = n_{split}(T, 1, N)$ and ν the number of recursive KCs

- If `split` returns m and $k = m$ after, then $\nu = 0$

- For the $k < m$ and $k > m$ cases we have respectively
  - $\nu = n_{QSel}(T, 1, m-1, k) \simeq A_{QSel}(m-1, k)$;
  - $\nu = n_{QSel}(T, m+1, N, k-m) \simeq A_{QSel}(N-m, k-m)$

Average Case of Quick Select II

- Assuming all cases equiprobable, we have

$$\begin{aligned} A_{QSel}(N,k) &\simeq N - 1 + \\ &\frac{1}{N}\sum_{m=1}^{k-1} A_{QSel}(N-m, k-m) + \\ &\frac{1}{N}\sum_{m=k+1}^{N} A_{QSel}(m-1, k) \end{aligned}$$

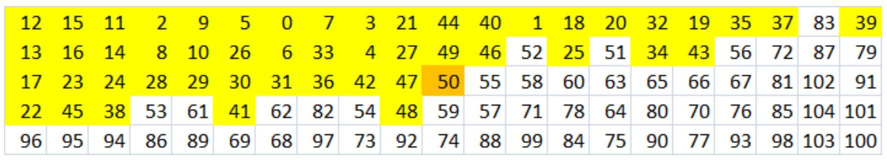
- And applying induction on this we arrive at $A_{QSel}(N,k) \leq 4N$ (details at the end)

Improving the Worst Case of Quick Select

- The worst case of QuickSelect (and of QuickSort) comes from the bad behavior over ordered tables of the pivot used in `split`
- To improve in this we need to improve pivot selection, ensuring that both subtables are large enough
- Assuming $N = 10L+5 = 5L+5+5L$, we can use QuickSelect itself proceeding as follows:
  - Split T into $2L+1$ consecutive subtables T_i of 5 elements
  - For each subtable T_i obtain its median m_i
  - Obtain the median m^* of $T' = [m_1, \ldots, m_{2L+1}]$ using QuickSelect with this kind of pivoting and return it as the pivot
- Now, the question is: does this pivot ensure that both subtables have a size $\Omega(N)$?

An Example of Median of 5 Pivoting

- Consider for instance the table below with $L = 10$, i.e., $10 \times 10 + 5 = 105$ elements
- We compute the medians of the 21 subtables with 5 elements by applying Merge-Sort, with a maximum cost of 8 key comparisons per table
- The median of the T' table is 50
- Looking at a reordering of T according to the medians in T', we mark as yellow the elements smaller than the 50 pivot and as white the larger elements



12	15	11	2	9	5	0	7	3	21	44	40	1	18	20	32	19	35	37	83	39
13	16	14	8	10	26	6	33	4	27	49	46	52	25	51	34	43	56	72	87	79
17	23	24	28	29	30	31	36	42	47	50	55	58	60	63	65	66	67	81	102	91
22	45	38	53	61	41	62	82	54	48	59	57	71	78	64	80	70	76	85	104	101
96	95	94	86	89	69	68	97	73	92	74	88	99	84	75	90	77	93	98	103	100

- Set $Y = \{i : T[i] < m^*\} = \mathcal{L}$, $W = \{i : T[i] > m^*\} = \mathcal{R}$

Analyzing the Median of 5 Pivot

- Clearly $|Y|$ and $|W|$ are $\geq 3L + 2$
- For the right–of–the–median set $|\mathcal{R}|$ we clearly have

$$
\begin{aligned}
|\mathcal{R}| &= N - |\{i : T[i] \leq m^*\}| = N - 1 - |Y| \\
&\leq (10L + 5) - 1 - (3L + 2) = 7L + 2 \\
&\leq 7\frac{N-5}{10} + 2 \leq \frac{7}{10}N
\end{aligned}
$$

- Similarly, we have $|\mathcal{L}| = N - 1 - |W| \leq 7N/10$
- It follows that after `split5` , the left and right subtables have at most $7N/10$ elements

The Cost of QuickSelect5

- Set $W_{QS5} = \max_{\sigma,k} n_{QS5}(\sigma, k)$
- The median of 5 pivoting computes first $N/5$ medians using MergeSort with a total cost of $8N/5$ kcs

- It then recursively applies QuickSelect5 over a table with $N/5$ elements, with a cost at most $W_{QS5}(\frac{N}{5})$ kcs
- The overall cost of QuickSelect5 can be bounded as

$$\begin{aligned} n_{QS5}(T,1,N,k) &= n_{Pivot5}(T,1,N) + n_{split}(T,1,N) + \\ &\quad \max\{n_{QS5}(T,1,M-1,k), n_{QS5}(T,M+1,N,k-M)\} \\ &\le \frac{8N}{5} + W_{QS5}\left(\frac{N}{5}\right) + N - 1 + \\ &\quad \max\{W_{QS5}(M-1), W_{QS5}(N-M)\} \end{aligned} \tag{5}$$

Worst Case of QuickSelect5

- **Proposition:** *We have* $W_{QS5}(N) \le 26N$
- **Proof sketch:** We will apply induction using (5)
  Assuming $W_{QS5}(N') \le 26N'$ for $N' < N$, we have:
  - $n_{Pivot5}(T,1,N) \le \frac{8}{5}N + W_{QS5}\left(\frac{N}{5}\right) \le \left(\frac{8}{5} + \frac{26}{5}\right)N \le \frac{34}{5}N$;
  - The cost of `split` is N-1
  - Since $M-1 \le 7N/10$ and $N-M \le 7N/10$, then

$$\begin{aligned} W_{QS5}(M-1) &\le W_{QS5}\left(\frac{7N}{10}\right) \le 26\frac{7}{10}N; \\ W_{QS5}(N-M) &\le W_{QS5}\left(\frac{7N}{10}\right) \le 26\frac{7}{10}N \end{aligned}$$

- Putting everything together in (5), we have

$$n_{QS5}(T,1,N,k) \le \left(\frac{34}{5} + 1 + 26 \times \frac{7}{10}\right)N - 1 \le 26N$$

What About QuickSort?

- Taking as the pivot the first element of a table, the average case cost of QuickSort was $O(N \lg N)$ but the worst case cost was $O(N^2)$
- The reason was that a first element pivot will be in the average near the middle of the table, but it may also be at the beginning or the end

- This was also the case with the first version of QuickSelect but was fixed in QuickSelect5
- If we use a median of 5 pivot, an $O(N \lg N)$ worst case for QuickSort can also be achieved

Back to $A_{QSel}(N, k)$

- We end the estimate of $A_{QSel}(N, k)$; assuming all cases equiprobable, we have

$$\begin{aligned} A_{QSel}(N,k) &\simeq N - 1 + \\ & \frac{1}{N}\sum_{M=1}^{k-1} A_{QSel}(N-M, k-M) + \\ & \frac{1}{N}\sum_{M=k+1}^{N} A_{QSel}(M-1, k) \end{aligned}$$

- Applying induction on N we have

$$A_{QSel}(N-M, k-M) \leq 4(N-M), \;\; A_{QSel}(M-1, k) \leq 4(M-1)$$

Back to $A_{QSel}(N, k)$ II

- Since for any m between 1 and N we have $N - m < N$ for $1 \leq m \leq k-1$ and $m - 1 < N$ for $k+1 \leq m \leq N$, it follows that

$$\begin{aligned} A_{QSel}(N,k) &\simeq N - 1 + \frac{1}{N}\sum_{m=1}^{k-1} A_{QSel}(N-M, k-m) + \\ & \frac{1}{N}\sum_{m=k+1}^{N} A_{QSel}(m-1, k) \\ &\leq N - 1 + \frac{4}{N}\left(\sum_{m=1}^{k-1}(N-m) + \sum_{m=k+1}^{N}(m-1)\right) \\ &= N - 1 + \frac{4}{N}\left(N(k-1) - \sum_{m=1}^{k-1} m + \sum_{m=k}^{N-1} m\right) \\ &= N - 1 + \frac{4}{N}\left(N(k-1) + \sum_{m=1}^{N-1} m - 2\sum_{m=1}^{k-1} m\right) \end{aligned}$$

Back to $A_{QSel}(N,k)$ III

- We thus have

$$\begin{aligned}
A_{QSel}(N,k) &= N-1+\frac{4}{N}\left(\frac{(N-1)N}{2}+N(k-1)-(k-1)k\right)\\
&= 3(N-1)+\frac{4}{N}\left(N(k-1)-k(k-1)\right)\\
&= 3(N-1)+\frac{4}{N}(N-k)(k-1)\\
&\leq 3(N-1)+\frac{4}{N}\frac{(N+1)(N-3)}{4}\\
&= 3(N-1)+N-2-\frac{3}{N}\leq 4N
\end{aligned}$$

2.5 The Cost of Recursion

Recursion's General Set–up

- General structure of recursive algorithms: for a size N problem
  - Split it in subproblems with size N/K
  - Solve M of these subproblems
  - Combine the subsolutions
- This leads to a general cost equation

$$T(N) = MT\left(\frac{N}{K}\right)+f(N),$$

  where $f(N)$ collects the cost of splitting and combining, and a base case $T(1) = 0$
- The concrete f are problem–dependent; in what follows we assume $f(N) = N^P$

Examples

- The following table collects some examples of algorithms that follow the previous set–up:

	K	P	M
MergeSort	2	1	2
Binary Search	2	0	1
FFT	2	1	2
Strassen	2	0	7
QSelect	2	1	1

Solving the Cost Recursion

- We assume $N = K^L$ for some natural L; unwinding the recursion we have

$$\begin{aligned} T(N) &= N^P + M\left(\left(\frac{N}{K}\right)^P + MT\left(\frac{N}{K^2}\right)\right) \\ &= N^P + N^p\frac{M}{K^P} + M^2T\left(\frac{N}{K^2}\right) = \ldots \\ &= N^P\sum_0^{L-1}\left(\frac{M}{K^P}\right)^j + M^LT\left(\frac{N}{K^L}\right) \\ &= N^P\sum_0^{L-1}\left(\frac{M}{K^P}\right)^j \end{aligned}$$

- The behaviour of $T(N)$ depends thus on $\frac{M}{K^P}$

Solving the Cost Recursion II

- If $M < K^P$, then $\frac{M}{K^P} < 1$ and

$$T(N) = N^P \sum_0^{L-1}\left(\frac{M}{K^P}\right)^j = O(N^P)$$

- If $M = K^P$, then $\frac{M}{K^P} = 1$ and since $L = \log_K N$,

$$T(N) = N^P \times L = O(N^P \log N)$$

	K	P	K^P	M	cost
MergeSort	2	1	2	2	$N^1 \log N = N \log N$
Binary Search	2	0	1	1	$N^0 \log N = \log N$
FFT	2	1	2	2	$N^1 \lg N = N \log N$
Strassen	2	0	1	7	$N^{\lg 7}$
QSelect	2	1	2	1	$N^1 = N$

- If $M > K^P$, then $\frac{M}{K^P} > 1$ and

$$\begin{aligned} T(N) &= N^P \frac{1}{\frac{M}{K^P} - 1} \left(\left(\frac{M}{K^P} \right)^L - 1 \right) = O\left(N^P \frac{M^L}{K^{LP}} \right) \\ &= O(M^L) = O\left(\left(K^{\log_K M} \right)^L \right) = O\left(N^{\log_K M} \right) \end{aligned}$$

Solving the Cost Recursion III

- Therefore, the actual cost of recursion depends on M, K and P
- If $M < K^P$, recursion is "cost–free" with respect to the cost N^P of splitting and combination
- If $M = K^P$, recursion adds a $\log N$ factor to the base N^P cost
- If $M > K^P$, the recursion cost adds a term $O\left(N^{\log_K M - P}\right)$ to the joint cost $O(N^P)$ of splitting and combination

Back to the Examples

- We can expand the previous algorithm table using these insights:
- For MergeSort, binary search and the FFT, recursion adds an extra $\log N$ cost term
- Recursion dominates in Strassen's algorithm
- Recursion is "free" for QuickSelect

3 Greedy Algorithms

3.1 The Fractionary Knapsack Problem

The Greedy Approach to Optimization

- Greedy algorithms try to solve a **global** optimization problem making at each step a **locally** optimal choice.
- This is a natural approach in optimization and usually yields efficient algorithms.
  - Sometimes it works: Dijkstra, Prim, Kruskal.
  - Often it doesn't: Nearest Neighbor in TSP, greedy knapsack.
- In some sense the greedy and DP approaches are opposite:
  - DP algorithms are hard to design but once derived, they are more or less automatically correct.
  - Greedy algorithms are easy to design but it is hard to prove them correct (and sometimes impossible!).

The Fractionary Knapsack Problem

- Again we have N elements with integer weights w_i and values v_i and a knapsack that stands a maximum weight W.
- We want a choice of weights w'_i, $0 \leq w'_i \leq w_i$, of the elements $i_1, \ldots, i_K$ such that $\sum_j w'_{i_j} \leq W$ and $\sum_j v_{i_j}$ is maximum.
- In other words, we may take **any weight fraction** of each element.
- If $\pi_i = v_i/w_i$, the constrained optimization problem we have to solve now is

$$\max \sum_1^N \pi_i w'_i \text{ subject to } \sum_1^N w'_i x_i \leq W \text{ and } 0 \leq w'_i \leq w_i.$$

- Now we have **real** constraints (and not integer ones).

The Greedy Knapsack

- We recall the greedy selection: order the i by decreasing relative values $\pi_i = \frac{v_i}{w_i}$ and select the first K ones such that

$$\sum_1^K w_{i_j} \leq W < \sum_1^{K+1} w_{i_j}.$$

- In more detail, assume the π_i values are sorted decreasingly and that $\sum_1^K w_i \leq W < \sum_1^{K+1} w_i$.

- The greedy knapsack is then $w_i^g = w_i$ when $i \leq K$, $w_{K+1}^g = W - \sum_1^K w_i$ and $w_i^g = 0$ for $K+2 \leq i \leq N$.

- Its cost is $O(N \lg N)$ for sorting and $O(N)$ to select the knapsack.

- The greedy knapsack didn't work in the 0–1 case but now it does.

Optimality of the Greedy Knapsack

- **Proposition:** *For any other selection w'_i we have*

$$V' = \sum_1^N w'_i \pi_i \leq \sum_1^N w_i^g \pi_i = V^g.$$

- To prove it notice that $w_i^g \geq w'_i$ for $i = 1, \dots K$ and thus,

$$\begin{aligned}
V^g - V' &= \sum_1^K \pi_i (w_i^g - w'_i) + \pi_{K+1}(w_{K+1}^g - w'_{K+1}) - \sum_{K+2}^N \pi_i w'_i \\
&\geq \pi_{K+1} \left(\sum_1^{K+1} w_i^g - \sum_1^N w'_i \right) \\
&\geq \pi_{K+1} \left(W - \sum_1^N w'_i \right) \geq 0.
\end{aligned}$$

- Therefore, $V^g \geq V'$ and the greedy knapsack is optimum.

3.2 An Introduction to Data Compression

Coding Alphabets

- Assume an alphabet $\mathcal{A} = \{\alpha_1, \dots, \alpha_M\}$; assume also a symbol set $\Sigma = \{\sigma_1, \dots, \sigma_S\}$.
- For instance, we could take as $\mathcal{A}$ the ascii characters and $\Sigma = \{0, 1\}$.
- A codification of $\mathcal{A}$ in terms of Σ is a variable lenght representation of each α_j as $\beta_j = (\sigma_{j_1} \dots \sigma_{j_{\ell_j}})$.
- We call $\mathcal{C} = \{\beta_1, \dots, \beta_M\}$ the code
- In a **prefix** (or instantaneous) code, no $\beta_i \in \mathcal{C}$ is a prefix of any other β_j.
- We can interpret the character codes of a prefix code as the **leaves of a tree** where each node has at most $S = |\Sigma|$ sons.

Binary Prefix Codes

- If $\Sigma = \{0, 1\}$, the symbols of a binary prefix code (BPC) $\mathcal{C}$ are in the leafs of a binary tree $T_\mathcal{C}$.
- In fact, this leaf property is a characterization of a prefix code.
- In other words, **a binary code $\mathcal{C}$ is prefix if and only if its symbols are in the leafs of a binary tree**.
- Moreover the lenght ℓ_j of a symbol β_j in a BPC (i.e., the number of bits) equals the depth of its leaf in $T_\mathcal{C}$.
- In a non–prefix binary code, some symbols will be in inner nodes.
- Notice that any prefix code can be decoded in a streaming fashion.

File Compression

- Assume a file F is made of characters from the $\mathcal{A} = \{\alpha_1, \dots, \alpha_M\}$ and each character α_j appears in F with (absolute) frequency f_j.
- If $\mathcal{C} = \{\beta_1, \dots, \beta_M\}$ is a BPC for $\mathcal{A}$ and $\ell_j =$ lenght (β_j), the size in bits of F compressed by $\mathcal{C}$ will be

$$\tau(F) = \sum_1^M f_j \ell_j .$$

- Natural question: how to choose $\mathcal{C}$ so that $\tau(F)$ is minimum?
- Natural answer: try something greedy.
  - Greedy Idea 1: place the symbols as the leaves of a binary tree with depth $\lceil \lg M \rceil$.
  - Greedy Idea 2: give one bit to the character most frequent, two to the second and so on.
  - But neither one is any good

The Huffman Approach

- Huffman's is an iterative greedy algorithm.
- At each step the two less frequent (meta)–characters of $\mathcal{A}$ are merged as a new meta–character with frequency the sum of the frequencies of the merged ones.
- Each meta–character can be seen as a partial BPC; we thus work with a forest of partial BPCs.
- After $M - 1$ such mergers we arrive at a single meta–character ρ that we put at the root of a binary tree.
- We obtain the BPC tree $T_{\mathcal{C}}$ de–merging the various meta–characters in the order in which they were created.

A Huffman Code

- **Example:** Assume F is made of characters a to g with respective frequencies 10, 15, 12, 3, 4, 13, 1.

Some Facts on Huffman Coding

- There may be several different Huffman BPCs for the same file F.
- Huffman coding requires three steps:
  - A pass through F to compute character frequencies, with cost $O(N)$ if F has N characters.
  - The actual computation of the BPC, with a cost $O(M \lg M)$ if we keep the metacharacters in a min–heap.
  - A second pass through F to compress it, with cost $O(N)$.
- The total cost is $O(N)$ but with two passes through F.
- There are one–pass alternatives such as the Lempel–Ziv family of compression algorithms.

Optimality of Huffman Codes

- **Theorem.** *Huffman codes $\mathcal{C}_H$ are optimal BPCs in the sense that if $\mathcal{C}'$ is another BPC for F, $\tau_{\mathcal{C}_H}(F) \leq \tau_{\mathcal{C}'}(F)$.*
  The proof relies on two lemmas.
- **Lemma 1.** *There is an optimal BPC for which the two less frequent characters are at the deepest leaves, have the same lenght and differ only in the last bit.*
- **Lemma 2.** *Let Σ be an alphabet with frequencies f_j and Σ' another in which we merge the two less frequent characters c_i, c_j in another c' with frequency $f_i + f_j$ and leave the other characters and frequencies unchanged. Assume that T' is a BPC for Σ' and T is obtained by demerging T'. Then T'* ***is an optimal BPC if and only if*** *T* ***is an optimal BPC.***

Proving the Lemmas

- To prove the first lemma, one assumes that c_1, c_2 are the less frequent characters and that c_1', c_2' are the two “deepest” characters in an optimum T, with $f_1' \leq f_2'$.
- Since $f_1 \leq f_1'$ and $f_2 \leq f_2'$ but $\ell_1 \leq \ell_1'$ and $\ell_2 \leq \ell_2'$, is easy to see that interchanging them diminishes $\tau(F)$.
- The second lemma captures the optimal substructure of any optimal code.

- To prove it one writes down $\tau_T(F)$ in terms of $\tau_{T'}(F)$ and then checks comparing the expression that $\tau_T(F)$ is optimal iff $\tau_{T'}(F)$ is optimal.

End of the Argument

- The optimality of a Huffman code follows from these lemmas.
- Lemma 1 is applied in each of the merging steps.
- Lemma 2 ensures that at each demerging step of the algorithm if the starting code is optimal, the code after demerging is also optimal.
  - It is a kind of loop invariant.
- The iterative demerging phase starts from a obviously optimal single leaf BPC and uses Lemma 2 at each step.
- We must thus arrive at an optimal Huffman BPC at the end of the demerging iterations.

3.3 Information and Entropy

Information and Disorder

- Natural question: Given a file F, how much can it be compressed?
- Intutively, the amount of information in F should establish a bound on how much it can be compresssed.
- New question: How do we measure information?
- Intuitively, the more "disordered" the content of a file is, the larger the information needed to describe it.
- In other words, we can assimilate information to disorder which leads to a new question:

  How do we measure disorder?

Entropy

- Disorder in the physical world is well studied and understood through Thermodynamics using a key physical magnitude, **entropy**.

- Claude Shannon introduced entropy in information theory.
- The **entropy** of a discrete distribution $P = (p_1, \dots, p_M)$ is given by

$$H = H(P) = \sum_1^M p_i \lg \frac{1}{p_i} = -\sum_1^M p_i \lg p_i.$$

- Intuitively, we may think that the information of the i–th character needs $\lg \frac{1}{p_i}$ bits to be stored.
- Thus, the entropy H is the average information in bits of the characters $\alpha_1, \dots, \alpha_M$.

Entropy Properties

- Obviously $H(P) \geq 0$ for any discrete distribution P.
- Obviously $H(0, \dots, 1, \dots, 0) = 0$.
  - In other words, since only one of the M characters will actually happen, there is no information in the character set.
- The entropy of the uniform distribution is $H(\frac{1}{M}, \dots, \frac{1}{M}) = \lg M$.
- **Proposition.** *The uniform distribution has the largest entropy; i.e., we have* $H(P) \leq \lg M$ *for any* P.
  - To prove it, consider for any $P = (p_1, \dots, p_M)$ the Lagrangian

$$L(P) = -\sum_1^M p_i \lg p_i + \lambda(1 - \sum p_i),$$

solve $\frac{\partial}{\partial p_i} L(P) = 0$ and plug it back into the constrain $\sum p_i = 1$.

Entropy and Optimal Compression

- Assume F has N characters with absolute frequencies f_i.
- The relative frequencies $p_i = f_i/N$ form a discrete probability distributon P.
- If we compress F with a BPC code $\mathcal{C}$ of M symbols with ℓ_i bits, its average character lenght in bits $L_{\mathcal{C}}$ is

$$L_{\mathcal{C}} = \sum_1^M \ell_i \frac{f_i}{N} = \sum_1^M \ell_i p_i.$$

- **Theorem.** *For any BPC code $\mathcal{C}$, we have*

$$L_{\mathcal{C}} \geq H(p_1, \ldots, p_M). \tag{6}$$

- In other words, the entropy H of the relative character frequencies in a file gives a bound for the **minimum average number of bits per character**.

Kraft's Inequality

- **Proposition.** *If the symbols in a BPC $\mathcal{C}$ have lenghts ℓ_i, then*

$$\sum_1^M 2^{-\ell_i} \leq 1. \tag{7}$$

- To prove it, recall that ℓ_i is the depth of the i–th symbol in $T_{\mathcal{C}}$ and choose $K \geq \ell_i$ for all i.
- Let T the binary tree of depth K built by growing $T_{\mathcal{C}}$ adding to each of its leaves descendants up to depth K.
- The maximum number of leaves in T is 2^K but each leaf of $T_{\mathcal{C}}$ adds $2^{K-\ell_i}$ leaves to T.
- We must thus have $\sum_1^N 2^{K-\ell_i} \leq 2^K$, and, therefore, (8) follows.

Proving the Entropy Bound I

- To prove (6), consider the following constrained minimization problem

$$\min L(\ell_1, \ldots, \ell_M) = \sum_1^M p_i \ell_i \;\; s.t. \;\; \sum p_i = 1, \sum_1^M 2^{-\ell_i} \leq 1, \tag{8}$$

  and where the ℓ_i are positive integers.
- An approach to simplify the argument is to relax the ℓ_i to be positive real numbers such that $\sum_1^M 2^{-\ell_i} = 1$.
- We can then work in the standard Lagrangian multiplier framework to show that the optimal ℓ_i^* verify $\ell_i^* = \lg 1/p_i$.
- We give next a more direct argument.

Proving the Entropy Bound II

- Denoting the entropy by H and $L(\ell_1, \dots, \ell_M)$ by L, we have:

$$\begin{aligned} H - L &= \sum_1^M p_i \left(\lg \frac{1}{p_i} - \ell_i \right) = \sum_1^M p_i \lg \frac{2^{-\ell_i}}{p_i} \\ &= \frac{1}{\log 2} \sum_1^M p_i \log \frac{2^{-\ell_i}}{p_i} \leq \frac{1}{\log 2} \sum_1^M p_i \left(\frac{2^{-\ell_i}}{p_i} - 1 \right) \\ &= \frac{1}{\log 2} \sum_1^M p_i \left(\frac{2^{-\ell_i} - p_i}{p_i} \right) = \frac{1}{\log 2} \sum_1^M \left(2^{-\ell_i} - p_i \right) \\ &= \frac{1}{\log 2} \left(\sum_1^M 2^{-\ell_i} - 1 \right) \leq 0. \end{aligned}$$

  where we have used that $\log x \leq x - 1$ and Kraft's inequality.

Shannon Coding

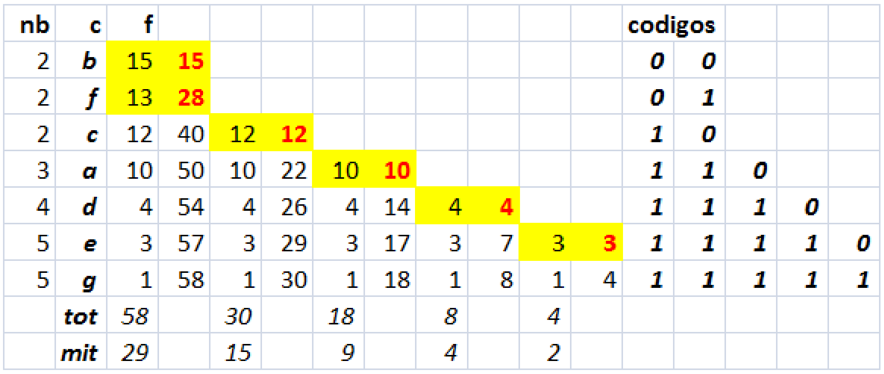
- In general $\lg 1/p_i$ will not be an integer but $\lceil \lg 1/p_i \rceil$ is.
- Shannon (or Shannon–Fano) codes build a BPC with at most $\lceil \lg 1/p_i \rceil$ bits per character.
- If $\mathcal{S}$ is a Shannon code, we have

$$L_{\mathcal{S}} = \sum_1^M p_i \left\lceil \lg \frac{1}{p_i} \right\rceil \leq 1 + \sum_1^M p_i \lg \frac{1}{p_i}.$$

- Thus the average number of bits per character in a Shannon code is at most 1 above the entropy optimum.
- To build a Shannon code, we sort the alphabet symbols by decreasing frequencies, and iteratively split them in left and right parts so that their probability sum is about the same.
- We assign then a bit 0 to the left part and a bit 1 to the right one.

Shannon Coding II

- It can be shown that this ensures lenghts such that $\ell_i \leq \lceil \lg 1/p_i \rceil$.
- **Example:** Assume again F is made of characters a to g with respective frequencies 10, 15, 12, 3, 4, 13, 1.
- The Shannon coding process is:



nb	c	f										codigos				
2	*b*	15	**15**									***0***	***0***			
2	*f*	13	**28**									***0***	***1***			
2	*c*	12	40	12	**12**							***1***	***0***			
3	*a*	10	50	10	22	10	**10**					***1***	***1***	***0***		
4	*d*	4	54	4	26	4	14	4	**4**			***1***	***1***	***1***	***0***	
5	*e*	3	57	3	29	3	17	3	7	3	**3**	***1***	***1***	***1***	***1***	***0***
5	*g*	1	58	1	30	1	18	1	8	1	4	***1***	***1***	***1***	***1***	***1***
	tot	*58*		*30*		*18*		*8*		*4*						
	mit	*29*		*15*		*9*		*4*		*2*						

- In this case it coincides with the Huffman optimum code.

A Huffman and Shannon Example

- For characters a to e with frequencies 15, 7, 6, 6, 5, the Huffman code gives a better compression.